MyBatis_Day1

1. 什么是 MyBatis?

MyBatis 是一款优秀的持久层框架，它支持定制化 SQL、存储过程以及高级映射。
MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。
MyBatis 可以使用简单的 XML 或注解来配置和映射原生信息，**将接口和 Java 的 POJOs(Plain Old Java Objects,普通的 Java 对象)映射成数据库中的记录。**

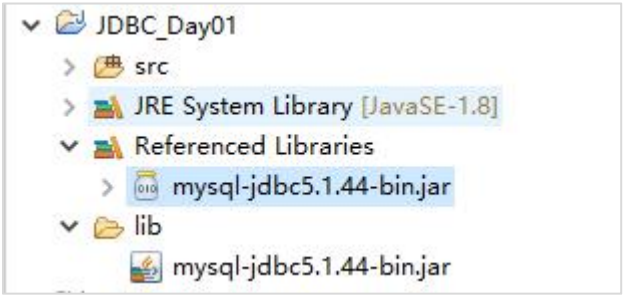
2. JDBC 回顾

2.1. 创建数据库

创建数据库 mybatis，执行 SQL 文件

2.2. 创建项目

创建 Java 项目，并导入相应的 jar 包



2.3. JDBC 开发步骤

1. 注册驱动
2. 获取连接对象 Connection
3. 通过 Connection 对象获取 Statement 对象

4. 使用 Statement 执行 SQL 语句
5. 遍历返回的结果集
6. 关闭资源

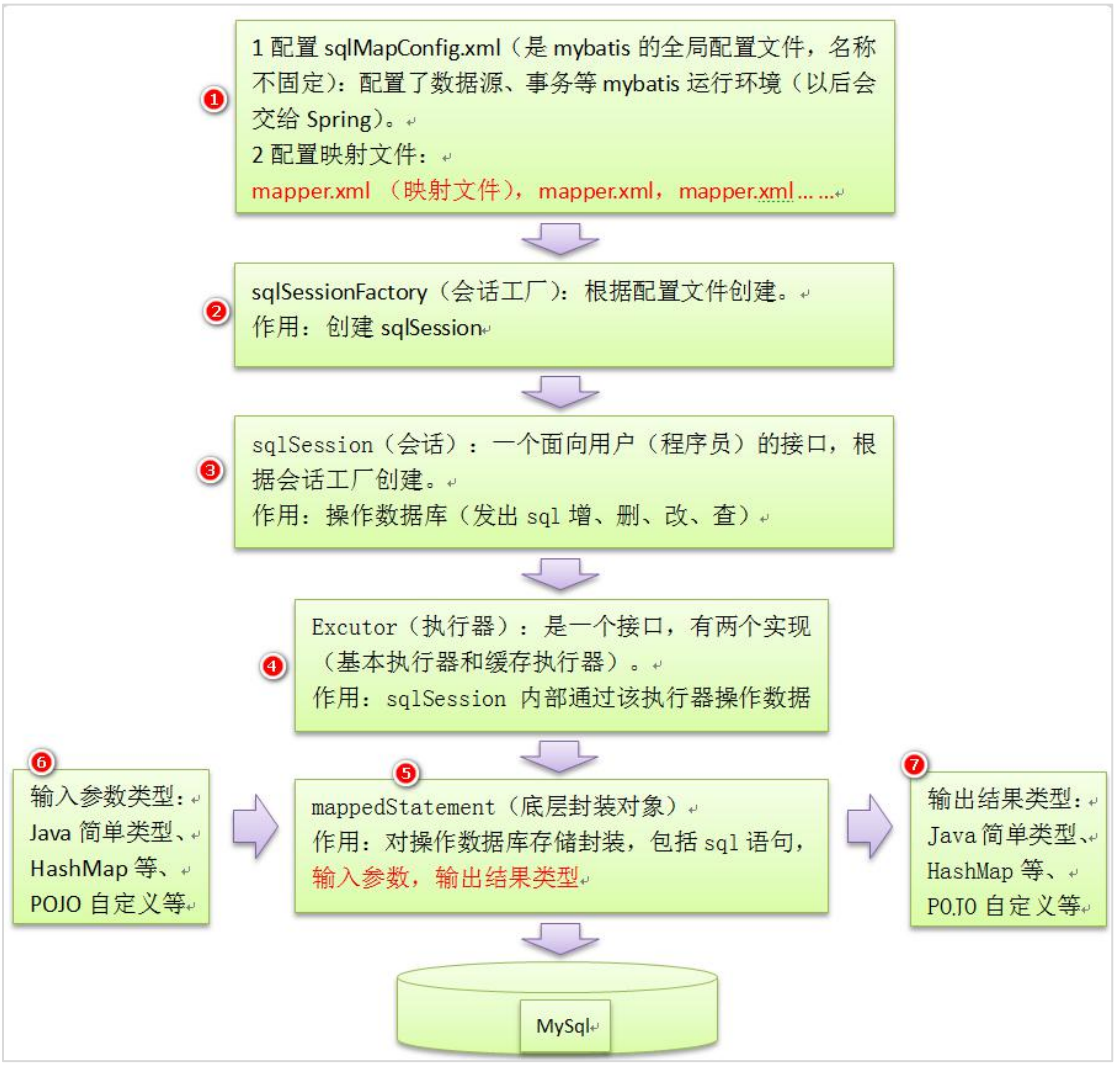
```
public static void main(String[] args) {
    Connection connection = null;
    PreparedStatement preparedStatement = null;
    ResultSet resultSet = null;

    try {
        // 加载数据库驱动
        Class.forName("com.mysql.jdbc.Driver");

        // 通过驱动管理类获取数据库链接
        connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mybatis?charac
terEncoding=utf-8", "root", "123456");
        // 定义 sql 语句 ?表示占位符
        String sql = "select * from grade where grade = ?";
        // 获取预处理 statement
        preparedStatement = connection.prepareStatement(sql);
        // 设置参数，第一个参数为 sql 语句中参数的序号（从 1 开始），第二个参数为
设置的参数值
        preparedStatement.setString(1, "2018");
        // 向数据库发出 sql 执行查询，查询出结果集
        resultSet = preparedStatement.executeQuery();
        // 遍历查询结果集
        while (resultSet.next()) {
            System.out.println(resultSet.getString("id") + "  " +
resultSet.getString("grade"));
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        // 释放资源
        if (resultSet != null) {
            try {
                resultSet.close();
            } catch (SQLException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        if (preparedStatement != null) {
            try {
                preparedStatement.close();
            } catch (SQLException e) {
```

```
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        if (connection != null) {
            try {
                connection.close();
            } catch (SQLException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
}
```

2.4. 使用 JDBC 的问题

1. 数据库连接创建、释放频繁造成系统资源浪费，从而影响系统性能。如果使用数据库连接池可解决此问题。
2. Sql 语句在代码中硬编码，造成代码不易维护，实际应用中 sql 变化的可能较大，sql 变动需要改变 java 代码。
3. 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。
4. 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 架构



1. MyBatis 配置

   SqlMapConfig.xml

   此文件作为 mybatis 的全局配置文件，配置了 MyBatis 的运行环境等信息。

   Mapper.xml

   文件即 sql 映射文件，文件中配置了操作数据库的 SQL 语句。此文件需要在 SqlMapConfig.xml 中加载。

2. 通过 MyBatis 环境等配置信息构造 SqlSessionFactory 即会话工厂
3. 由会话工厂创建 SQLSession 即会话，操作数据库需要通过 SQLSession 进行。
4. MyBatis 底层自定义了 Executor 执行器接口操作数据库，Executor 接口有两个实现，一个是基本执行器、一个是缓存执行器。
5. MappedStatement 也是 MyBatis 一个底层封装对象，它包装了 MyBatis 配置信息及 SQL 映射信息等。Mapper.xml 文件中一个 SQL 对应一个 MappedStatement 对象，SQL 的 id 即是 MappedStatement 的 id。
6. 输入参数映射，在执行方法时，MappedStatement 对 SQL 执行输入参数进行定义，包括 HashMap、基本类型、pojo，Executor 通过 MappedStatement 在执行 SQL 前将输入的 Java

对象映射至 SQL 中，输入参数映射就是 JDBC 编程中对 preparedStatement 设置参数。

7. 输出结果映射，在数据库中执行完 SQL 语句后，MappedStatement 对 sql 执行输出结果进行定义，包括 HashMap、基本类型、pojo，Executor 通过 MappedStatement 在执行 sql 后将输出结果映射至 Java 对象中，输出结果映射过程相当于 JDBC 编程中对结果的解析处理过程。

4. MyBatis 快速入门

4.1. 案例演示

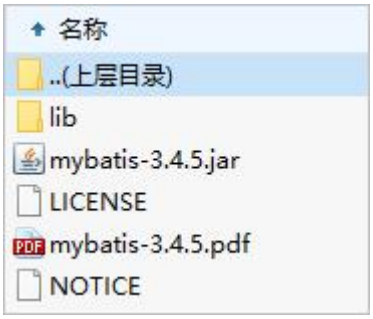
- 使用 MyBatis 实现以下功能：
  1. 根据用户 id 查询一个年级信息
  2. 根据年级名称模糊查询年级列表
  3. 添加年级
  4. 编辑年级
  5. 删除年级

4.2. 环境配置

4.2.1. 下载 jar

4.2.1.1. 源码地址

下载地址：https://github.com/mybatis/mybatis-3/releases



- 文件夹介绍

  mybatis-3.2.7.jar：mybatis 的核心包

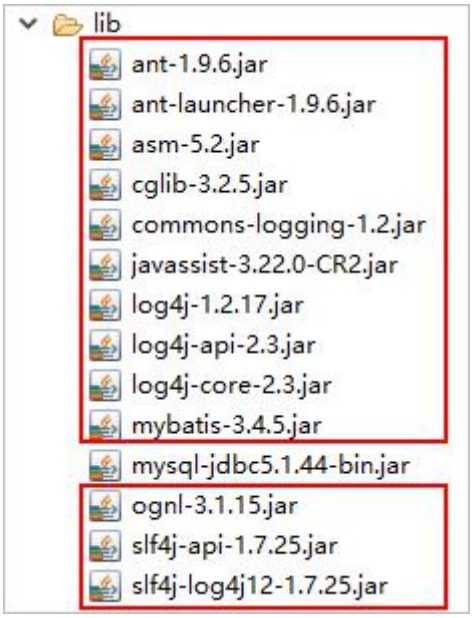
  lib 文件夹：mybatis 的依赖包所在

  mybatis-3.2.7.pdf：mybatis 使用手册

4.2.1.2. Maven 地址

```
<!-- https://mvnrepository.com/artifact/org.mybatis/mybatis -->
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.4.6</version>
</dependency>
```

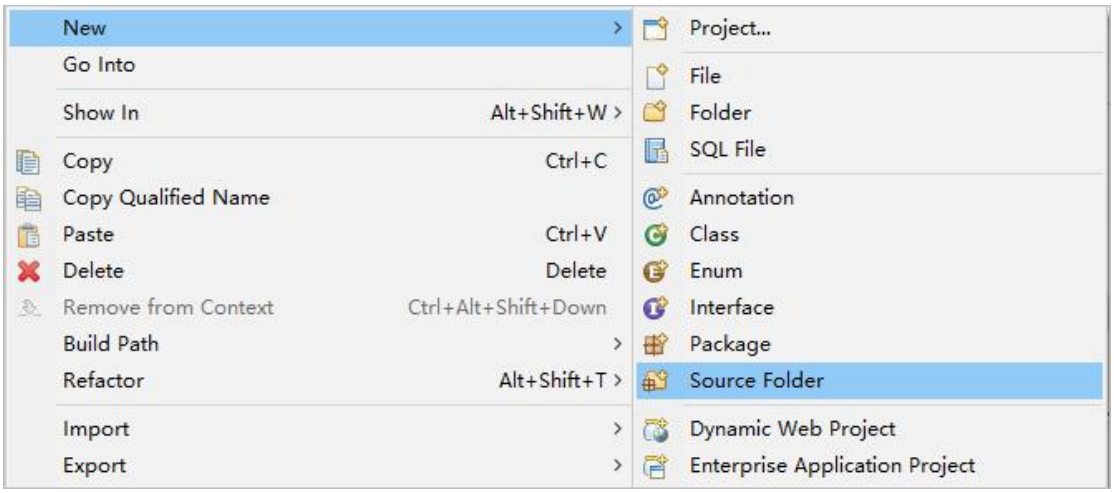
4.2.2. 创建项目，导入 jar



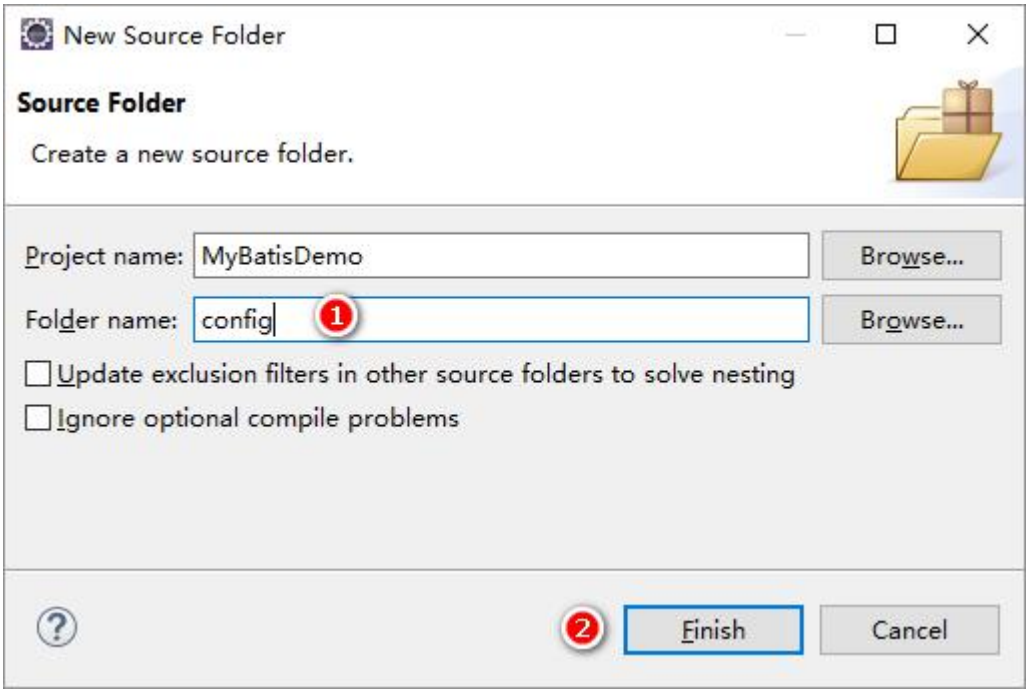
4.2.3. 创建资源文件夹

创建资源文件夹 **config（idea 需要标记为资源文件）**，加入 **log4j.properties** 和 **SqlMapConfig.xml** 配置文件

选中项目更目录，右键 new/Siource Folder



填入文件夹名字



4.2.3.1. 创建 log4j.properties

因为 mybatis 默认使用 log4j 作为输出日志信息，需要在 config 下创建 log4j.properties 如下

```
# Global logging configuration
log4j.rootLogger=DEBUG, stdout
# Console output...
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
```

```
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```

4.2.3.2. SqlMapConfig.xml

SqlMapConfig.xml 是 mybatis 核心配置文件，配置文件中包含了对 MyBatis 系统的核心设置，包含获取数据库连接实例的数据源（DataSource）和决定事务作用域和控制方式的事务管理器（TransactionManager）。

在 config 下创建 SqlMapConfig.xml，如下

```
<?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>
    <!-- 和 spring 整合后 environments 配置将废除 -->
    <environments default="development">
        <environment id="development">
            <!-- 使用 jdbc 事务管理 -->
            <transactionManager type="JDBC" />
            <!-- 数据库连接池 -->
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver" />
                <property name="url"
value="jdbc:mysql://localhost:3306/examsystem?characterEncoding=utf-8"
/>
                <property name="username" value="root" />
                <property name="password" value="123456" />
            </dataSource>
        </environment>
    </environments>
</configuration>
```

4.2.4. 创建 pojo

pojo 类作为 mybatis 进行 sql 映射使用，po 类通常与数据库表对应

```
public class Grade{
    private int id;
    private int grade;// 用户姓名
    //get/set……
}
```

4.2.5. 配置 sql 映射文件

在 config 下创建 sql 映射文件 Grade.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">
<!-- namespace：命名空间【重要】-->
<mapper namespace="grade">
</mapper>
```

4.2.6. 加载配置文件

mybatis 框架需要加载 Mapper.xml 映射文件

将 Grade.xml 添加在 SqlMapConfig.xml，如下：

```
            <dataSource type="POOLED">
                <property name="driver" value="
                <property name="url"
                          value="jdbc:mysql://l
                <property name="username" value
                <property name="password" value
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <mapper resource="Grade.xml"/>
    </mappers>
</configuration>
```

4.3. 实现

4.3.1. 根据 id 查询年级信息

查询的 SQL

```
SELECT * FROM grade WHERE id = 1
```

4.3.1.1. 配置映射文件

在 Grade.xml 中添加 select 标签，编写 SQL

```
<mapper namespace="grade">
    <!-- id:表示命名空间的唯一标识，与命名空间配合使用，组合后必须唯一-->
    <!-- parameterType：传入的 SQL 语句参数类的全类名或者别名（参数类型） -->
    <!-- resultType：执行 SQL 语句中返回的结果类型的全类名或者别名，如果是集合类型，那么返回的是集合可以包含的类型，而不是集合本身 -->
    <!-- #{}：输入参数的占位符，相当于 jdbc 的[?] -->
    <select id="findGradeById" parameterType="int"
        resultType="com.neuesoft.sz.pojo.Grade">
        SELECT * FROM Grade WHERE id = #{id}
    </select>
</mapper>
```

4.3.1.2. 测试

测试程序步骤：

1. 创建 SqlSessionFactoryBuilder 对象，并加载 SqlMapConfig.xml 配置文件
2. 调用 SqlSessionFactoryBuilder 的 build 方法，创建 SqlSessionFactory
3. 调用 SqlSessionFactory.openSession()创建 SqlSession 对象
4. 执行 SqlSession.selectOne()，进行查询，并获取结果 Grade
5. 处理结果、释放资源

```
@Test
public void testQueryGradeById() throws IOException {
    // 1. 创建 SqlSessionFactoryBuilder 对象，加载 SqlMapConfig.xml 配置文件
    SqlSessionFactoryBuilder sessionFactoryBuilder = new
SqlSessionFactoryBuilder();
    InputStream is = Resources.getResourceAsStream("SqlMapConfig.xml");
    // 2. 创建 SqlSessionFactory 对象
    SqlSessionFactory sessionFactory = sessionFactoryBuilder.build(is);
    // 3. 创建 SqlSession 对象
    SqlSession sqlSession = sessionFactory.openSession();
    // 4. 执行 SqlSession 对象执行查询，获取结果 Grade
    Grade grade = sqlSession.selectOne("findGradeById", 1);
    // 5. 处理结果、释放资源
    System.out.println(grade);
    sqlSession.close();
}
```

- 执行结果

```
DEBUG [main] - Created connection 949057310.
DEBUG [main] - Setting autocommit to false on JDBC Connection [
DEBUG [main] - ==>  Preparing: SELECT * FROM grade WHERE id = ?
DEBUG [main] - ==> Parameters: 1(Integer)
DEBUG [main] - <==      Total: 1
2018
1
```

4.3.2. 根据年级模糊查询年级列表

模糊查询的 sql：

```
SELECT * FROM Grade WHERE grade LIKE '%18%'
```

4.3.2.1. 方式一：#{}

5.5.1.1. 映射文件

在 Grade.xml 配置文件中添加如下内容：

```
<!-- 返回结果如果是集合类型，那么返回的是集合可以包含的类型，而不是集合本身 -->
<!-- resultType 的配置和返回一个结果的配置一样 -->
<select id="findGradeById" parameterType="java.lang.String"
resultType="com.neuesoft.sz.Grade">
    SELECT * FROM grade WHERE grade= #{grade}
</select>
```

4.3.2.2. 方式二：${}

```
<!-- ${} 使用拼接符：字符串原样拼接，如果传入的参数是基本数据类型，那么${}中的
变量名称必须为：value-->
<!--注意：拼接符有 sql 注入的风险-->
<select id="findGradeByNameLike2" parameterType="string"
resultType="com.neuesoft.sz.Grade" >
    SELECT * FROM grade where grade like '%${value}%'
</select>
```

为什么是 value？推荐链接

mybatis 中的 ${value} 替代符

4.3.2.3. 测试

方式一

```
@Test
public void testFindGradeByNameLike1() throws IOException {
    SqlSessionFactory sf = new
SqlSessionFactoryBuilder().build(Resources.getResourceAsStream("SqlMapC
```

```
onfig.xml"));
    SqlSession ss = sf.openSession();
    List<Grade> grades = ss.selectList("grade.findGradeByNameLike1",
"%2%");
    System.out.println(grades);
}
```

方式二

```
@Test
public void testFindGradeByNameLike2() throws IOException {
    SqlSessionFactory sf = new
SqlSessionFactoryBuilder().build(Resources.getResourceAsStream("SqlMapC
onfig.xml"));
    SqlSession ss = sf.openSession();
    List<Grade> grades = ss.selectList("grade.findGradeByNameLike2", "2");
    System.out.println(grades);
}
```

4.3.3. 小结

4.3.3.1. #{}、${}区别

- **#{}**

  表示一个占位符号，通过#{}可以实现 preparedStatement 向占位符中设置值

  默认将其当成字符串

  #{}可以有效防止 sql 注入

- **${}**

  ${}表示拼接 sql 串，通过${}可以将 parameterType 传入的内容拼接在 sql 中

  不做任何处理

  ${}可以接收简单类型值或 pojo 属性值，如果 parameterType 传输单个简单类型值，${}括号中只能是 value。

推荐链接：

mybatis 中的#和$的区别

4.3.3.2. parameterType 和 resultType

parameterType

指定输入参数类型，mybatis 通过 ognl 从输入对象中获取参数值拼接在 sql 中。

resultType

指定输出结果类型，mybatis 将 sql 查询结果的一行记录数据映射为 resultType 指定类型的对象。如果有多条数据，则分别进行映射，并把对象放到容器 List 中

4.3.3.3. selectOne 和 selectList

selectOn： 查询一条记录，如果使用 selectOne 查询多条记录则抛出异常：

```
    org.apache.ibatis.exceptions.TooManyResultsException: Expected one result (or null) to be returned by selectOne(), but found: 3
    at org.apache.ibatis.session.defaults.DefaultSqlSession.selectOne(DefaultSqlSession.java:70)
```

selectList： 可以查询一条或多条记录。

4.3.4. 添加年级

sql 语句：INSERT INTO Grade (id,grade) VALUES (1,2018)

4.3.4.1. 配置映射文件

```
<!--保存年级信息-->
<insert id="saveGrade" parameterType="com.neuesoft.sz.Grade">
    INSERT INTO Grade (id,grade) VALUES (#{id},#{grade})
</insert>
```

4.3.4.2. 测试

```
@Test
public void testSaveGrade() throws IOException {
    SqlSessionFactory sf = new
SqlSessionFactoryBuilder().build(Resources.getResourceAsStream("SqlMapC
onfig.xml"));
    SqlSession ss = sf.openSession();
    Grade grade = new Grade();
    grade.setId(4);
    grade.setGrade(2015);
    ss.insert("grade.saveGrade",grade);
    ss.commit();
    ss.close();
}
```

4.3.4.3. 设置主键自动增长

方式一：

查询 id 的 sql：SELECT LAST_INSERT_ID()

LAST_INSERT_ID():是 mysql 的函数，返回 auto_increment 自增列新记录 id 值

通过使用 keyProperty 属性指定 PO 类的某个属性接受主键的返回值（通常会设置到 id 上）

通过修改 Grade.xml 映射文件，可以将 mysql 自增主键返回：

添加 selectKey 标签

```
<!-- 保存年级信息-主键自动增长 -->
<insert id="saveGradeAK2" parameterType="com.neuesoft.sz.pojo.Grade">
    <!-- selectKey 标签实现主键返回 -->
    <!-- keyColumn:主键对应的表中的哪一列 -->
    <!-- keyProperty：主键对应的pojo 中的哪一个属性 -->
    <!-- order：设置在执行 insert 语句前执行查询 id 的 sql-->
    <!-- resultType：设置返回的 id 的类型 -->
    <selectKey keyColumn="id" keyProperty="id" order="AFTER"
               resultType="int">
        SELECT LAST_INSERT_ID()
    </selectKey>
    INSERT INTO Grade (id,grade) VALUES (#{id},#{grade})
</insert>
```

方式二：

针对于 Mysql 这种自己维护主键的数据库，可以直接使用以下配置在插入后获取插入主键，如果你的数据库支持自动生成主键的字段（比如 MySQL 和 SQL Server），那么你可以设置 useGeneratedKeys=”true”，然后再把 keyProperty 设置到目标属性上就 OK 了。
例如，如果上面的 Grade 表已经对 id 使用了自动生成的列类型，那么语句可以修改为:

```
<!-- 保存年级信息-主键自动增长 -->
<insert id="saveGradeAK2" useGeneratedKeys="true" keyProperty="id"
parameterType="com.neuesoft.sz.pojo.Grade">
    INSERT INTO Grade (grade) VALUES (#{grade})
</insert>
```

测试

```
@Test
public void testSaveGradeAK() throws IOException {
    SqlSessionFactoryBuilder sfb = new SqlSessionFactoryBuilder();
    SqlSessionFactory sf =
sfb.build(Resources.getResourceAsStream("SqlMapConfig.xml"));
    SqlSession ss = sf.openSession();
    Grade grade = new Grade();
    grade.setGrade(2010);
    int i = ss.insert("grade.saveGradeAK2", grade);
    ss.commit();
    ss.close();
    System.out.println(i);
}
```

selectKey 介绍

在上面的示例中，selectKey 元素将会首先运行，Grade 的 id 会被设置，然后插入语句会被调用。设置了处理自动生成主键类似的行为

selectKey 元素描述如下：

```
<selectKey
  keyProperty="id"
  resultType="int"
  order="BEFORE"
  statementType="PREPARED">
```

selectKey Attributes	
属性	**描述**
keyProperty	仅对 insert 和 update 有用，此属性的作用是将插入或更新操作时的返回值赋值给 PO 类的某个属性，通常会设置为主键对应的属性，如果需要设置联合主键，可以在多个值之间用逗号隔开。
keyColumn	匹配属性的返回结果集中的列名称。如果希望得到多个生成的列，也可以是逗号分隔的属性名称列表。
resultType	结果的类型。MyBatis 通常可以推算出来，但是为了更加确定写上也不会有什么问题。MyBatis 允许任何简单类型用作主键的类型，包括字符串。如果希望作用于多个生成的列，则可以使用一个包含期望属性的 Object 或一个 Map。
order	这可以被设置为 BEFORE 或 AFTER。如果设置为 BEFORE，那么它会首先选择主键，设置 keyProperty 然后执行插入语句。如果设置为 AFTER，那么先执行插入语句，然后是 selectKey 元素 - 这和像 Oracle 的数据库相似，在插入语句内部可能有嵌入索引调用。
statementType	与前面相同，MyBatis 支持 STATEMENT，PREPARED 和 CALLABLE 语句的映射类型，分别代表 PreparedStatement 和 CallableStatement 类型。

4.3.4.4. Mysql 使用 uuid 实现主键

需要增加通过 select uuid()得到 uuid 值

注意这里使用的 order 是“BEFORE”

配置文件

```
<insert id="insertTeacher" parameterType="com.neuesoft.sz.pojo.Teacher">
    <selectKey resultType="java.lang.String" order="BEFORE"
               keyProperty="id">
        SELECT uuid()
```

```
    </selectKey>
    INSERT INTO Student (id,name,password,modified)
    VALUES (#{id},#{name},#{password},#{modified})
</insert>
```

测试代码和之前保存代码一样

4.3.5. 删除年级

配置文件

```
<!--删除年级-->
   <delete id="deleteGradeById" parameterType="int">
   DELETE FROM Grade WHERE id=#{id}
</delete>
```

测试代码

```
@Test
public void testDeleteGrade() throws IOException {
    SqlSessionFactory sf = new
SqlSessionFactoryBuilder().build(Resources.getResourceAsStream("SqlMapC
onfig.xml"));
    SqlSession ss = sf.openSession();
    ss.delete("grade.deleteGradeById", 5);
    ss.commit();
    ss.close();
}
```

4.3.6. 更新年级

配置文件

```
<!--更新年级-->
<update id="updateGradeById" parameterType="com.neuesoft.sz.pojo.Grade">
    UPDATE Grade SET grade = #{grade} WHERE id = #{id}
</update>
```

测试代码

```
@Test
public void testUpdateGradeById() throws IOException {
    SqlSessionFactory sf = new
SqlSessionFactoryBuilder().build(Resources.getResourceAsStream("SqlMapC
onfig.xml"));
    SqlSession ss = sf.openSession();
    Grade grade = new Grade();
```

```
    grade.setId(1);
    grade.setGrade(2020);
    ss.update("grade.updateGradeById", grade);
    ss.commit();
    ss.close();
}
```

5. 核心对象

5.1. SqlSessionFactory

SqlSessionFactory 是一个接口，其主要作用是创建 SqlSession（openSession 方法）
SqlSessionFactory 对象的实例可以通过 SqlSessionFactoryBuilder 对象来构建

SqlSessionFactoryBuilder

SqlSessionFactoryBuilder 用于创建 SqlSessionFacoty

5.1.1. 从 XML 中构建 SqlSessionFactory

每个基于 MyBatis 的应用都是以一个 SqlSessionFactory 的实例为中心的。SqlSessionFactory 的实例可以通过 SqlSessionFactoryBuilder 获得。而 SqlSessionFactoryBuilder 则可以从 XML 配置文件或一个预先定制的 Configuration 的实例构建出 SqlSessionFactory 的实例。

从 XML 文件中构建 SqlSessionFactory 的实例非常简单，建议使用类路径下的资源文件进行配置。但是也可以使用任意的输入流(InputStream)实例，包括字符串形式的文件路径或者 file:// 的 URL 形式的文件路径来配置。
MyBatis 包含一个名叫 Resources 的工具类，它包含一些实用方法，可使从 classpath 或其他位置加载资源文件更加容易。

```
String resource = "org/mybatis/example/mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
```

5.1.2. SqlSessionFactory 注意事项

SqlSessionFactory 对象是线程安全的，它一旦被创建，整个应用程序执行期间就会存在
如果我们多次地创建数据库的 SqlSessionFactory，那么次数据库的资源很容易被耗尽，为了

解决此问题，通常每一个数据库都会只对应一个 SqlSessionFactory，所以在构建 SqlSessionFactory 实例时，**建议使用单例**

5.2. SqlSession

SqlSession 是应用程序与持久层之间执行交互操作的一个单线程对象，其主要作用是执行持久化操作
SqlSession 中封装了对数据库的操作，如：查询、插入、更新、删除等。

每一个线程都应该有一个自己的 SqlSession 实例，并且该实例是不能被共享的，同时，SqlSession 实例也是线程不安全的，因此使用范围最好在一次请求或一个方法中，绝不能将其放在一个类的静态字段、实例字段或任何类的管理范围（如 Servlet 的 HttpSession）中使用，使用完 SqlSession 对象后，要及时地关闭它，通常将其放在 finally 块中关闭，代码如下

```
SqlSession session = sqlSessionFactory.openSession();
try {
        // do work
} finally {
      session.close();
}
```

6. MyBatis 配合开发 Dao

使用 Mybatis 开发 Dao，通常有两个方法，即普通 Dao 开发方法和 Mapper 接口开发方法。

6.1. 手动编写 Dao 实现类

手动编写 Dao 实现类方法需要程序员编写 Dao 接口和 Dao 实现类。
映射文件（和之前一样）

```
<select id="findGradeById" parameterType="int"
resultType="com.neuesoft.sz.pojo.Grade">
    SELECT * FROM grade WHERE id = #{id}
</select>

<select id="findGradeByNameLike2" parameterType="string"
resultType="com.neuesoft.sz.pojo.Grade">
    SELECT * FROM grade where grade like '%${value}%'
</select>
```

Dao 接口

```
public interface GradeDao {
```

```
    public Grade findGradeById(Integer id);
    public List<Grade> findGradeByGradeName(String grade);
}
```

接口实现类

```
public class GradeDaoImpl implements GradeDao {
    private SqlSessionFactory sqlSessionFactory;

    //通过构造方法注入
    public GradeDaoImpl(SqlSessionFactory sqlSessionFactory) {
        this.sqlSessionFactory = sqlSessionFactory;
    }

    @Override
    public Grade findGradeById(Integer id) {
        //sqlSesion 是线程不安全的,所以它的最佳使用范围在方法体内
        SqlSession openSession = sqlSessionFactory.openSession();
        Grade grade = openSession.selectOne("grade.findGradeById", id);
        return grade;
    }

    @Override
    public List<Grade> findGradeByGradeName(String grade) {
        SqlSession openSession = sqlSessionFactory.openSession();
        List<Grade> list =
openSession.selectList("grade.findGradeByNameLike2", grade);
        return list;
    }
}
```

测试

```
public class GradeDaoTest {
    private SqlSessionFactory factory;
    //作用:在测试方法前执行这个方法
    @Before
    public void setUp() throws Exception {
        String resource = "SqlMapConfig.xml";
        //通过流将核心配置文件读取进来
        InputStream inputStream = Resources.getResourceAsStream(resource);
        //通过核心配置文件输入流来创建会话工厂
        factory = new SqlSessionFactoryBuilder().build(inputStream);
    }
    @Test
    public void testFindGradeById() {
```

```
        //将初始化好的工厂注入到实现类中
        GradeDao gradeDao = new GradeDaoImpl(factory);
        Grade grade = gradeDao.findGradeById(1);
        System.out.println(grade);
    }

    @Test
    public void testFindGradeByGradeName() {
        GradeDao gradeDao = new GradeDaoImpl(factory);
        List<Grade> list = gradeDao.findGradeByGradeName("20");
        System.out.println(list);
    }
}
```

6.2. Mapper 动态代理方式

6.2.1.1. 开发规范

Mapper 接口开发方法只需要程序员编写 Mapper 接口（相当于 Dao 接口），由 Mybatis 框架根据接口定义创建接口的动态代理对象，代理对象的方法体同上边 Dao 接口实现类方法。

Mapper 接口开发需要遵循以下规范：

1. Mapper 接口名称和对应的 Mapper.xml 文件必须相同
2. Mapper.xml 文件中的 namespace 与 Mapper 接口的类路径相同。
3. Mapper 接口方法名和 Mapper.xml 中定义的每个 statement 的 id 相同
4. Mapper 接口方法的输入参数类型和 Mapper.xml 中定义的每个 sql 的 parameterType 的类型相同
5. Mapper 接口方法的输出参数类型和 Mapper.xml 中定义的每个 sql 的 resultType 的类型相同

只要遵循了以上规范，MyBatis 就可以自动生成 Mapper 接口实现类的代理对象，从而简化我们的开发

6.2.1.2. Mapper.xml(映射文件)

定义 mapper 映射文件 GradeMapper.xml，需要修改 namespace 的值为 GradeMapper 接口路径。将 GradeMapper.xml 放在 classpath 下 mapper 目录下。

```
<?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">
<!-- namespace：命名空间【重要】-->
```

```
<mapper namespace="com.neuesoft.sz.mapper.GradeMapper">
    <select id="findGradeById" parameterType="int"
resultType="com.neuesoft.sz.pojo.Grade">
        SELECT * FROM grade WHERE id = #{id}
    </select>

    <select id="findGradeByGradeName" parameterType="string"
resultType="com.neuesoft.sz.pojo.Grade">
        SELECT * FROM grade where grade like '%${value}%'
    </select>

    <!-- 保存年级信息-主键自动增长 -->
    <insert id="saveGrade" parameterType="com.neuesoft.sz.pojo.Grade">
        <selectKey keyColumn="id" keyProperty="id" order="AFTER"
                   resultType="int">
            SELECT LAST_INSERT_ID()
        </selectKey>
        INSERT INTO Grade (id,grade) VALUES (#{id},#{grade})
    </insert>
</mapper>
```

6.2.1.3. Mapper.java(接口文件)

```
public interface GradeMapper {
    public Grade findGradeById(Integer id);
    public List<Grade> findGradeByGradeName(String grade);
    public void saveGrade(Grade grade);
}
```

接口定义有如下特点：

1、 Mapper 接口方法名和 Mapper.xml 中定义的 statement 的 id 相同

2、 Mapper 接口方法的输入参数类型和 mapper.xml 中定义的 statement 的 parameterType 的类型相同

3、 Mapper 接口方法的输出参数类型和 mapper.xml 中定义的 statement 的 resultType 的类型相同

6.2.1.4. 加载 GradeMapper.xml 文件

```
<!-- 加载映射文件 -->
<mappers>
   <package name="com.neuesoft.sz.mapper"/>
</mappers>
```

6.2.1.5. 测试

```
public class GradeMapperTest {
    private SqlSessionFactory factory;
    //作用:在测试方法前执行这个方法
    @Before
    public void setUp() throws Exception {
        String resource = "SqlMapConfig.xml";
        //通过流将核心配置文件读取进来
        InputStream inputStream = Resources.getResourceAsStream(resource);
        //通过核心配置文件输入流来创建会话工厂
        factory = new SqlSessionFactoryBuilder().build(inputStream);
    }
    @Test
    public void testFindGradeById() {
        SqlSession openSession = factory.openSession();
        //通过 getMapper 方法来实例化接口
        GradeMapper mapper = openSession.getMapper(GradeMapper.class);
        Grade grade = mapper.findGradeById(1);
        System.out.println(grade);
    }
    @Test
    public void testFindGradeByGradeName() {
        SqlSession openSession = factory.openSession();
        //通过 getMapper 方法来实例化接口
        GradeMapper mapper = openSession.getMapper(GradeMapper.class);
        List<Grade> grades = mapper.findGradeByGradeName("20");
        System.out.println(grades);
    }
    @Test
    public void testSaveGrade() throws Exception{
        SqlSession openSession = factory.openSession();
        //通过 getMapper 方法来实例化接口
        GradeMapper mapper = openSession.getMapper(GradeMapper.class);
        Grade grade = new Grade();
        grade.setGrade(2099);
        mapper.saveGrade(grade);
        openSession.commit();
    }
}
```

6.2.1.6. 小结

selectOne 和 selectList

动态代理对象调用 sqlSession.selectOne()和 sqlSession.selectList()是根据 mapper 接口方法的返回值决定，如果返回 list 则调用 selectList 方法，如果返回单个对象则调用 selectOne 方法。

namespace

mybatis 官方推荐使用 mapper 代理方法开发 mapper 接口，程序员不用编写 mapper 接口实现类，使用 mapper 代理方法时，输入参数可以使用 pojo 包装对象或 map 对象，保证 dao 的通用性。

7. Mapper XML

MyBatis 的真正强大在于它的映射语句，也是它的魔力所在。由于它的异常强大，映射器的 XML 文件就显得相对简单。如果拿它跟具有相同功能的 JDBC 代码进行对比，你会立即发现省掉了将近 95% 的代码。MyBatis 就是针对 SQL 构建的，并且比普通的方法做的更好。

SQL 映射文件有很少的几个顶级元素（按照它们应该被定义的顺序）：

cache – 给定命名空间的缓存配置。

cache-ref – 其他命名空间缓存配置的引用。

resultMap – 是最复杂也是最强大的元素，用来描述如何从数据库结果集中来加载对象。

parameterMap – 已废弃！老式风格的参数映射。内联参数是首选,这个元素可能在将来被移除，这里不会记录。

sql – 可被其他语句引用的可重用语句块。

insert – 映射插入语句

update – 映射更新语句

delete – 映射删除语句

select – 映射查询语句

select

查询语句是 MyBatis 中最常用的元素之一

简单查询的 select 元素是非常简单的。比如：

```
<select id="selectPerson" parameterType="int" resultType="hashmap">
  SELECT * FROM PERSON WHERE ID = #{id}
</select>
```

这个语句被称作 selectPerson，接受一个 int（或 Integer）类型的参数，并返回一个 HashMap 类型的对象，其中的键是列名，值便是结果行中的对应值。

注意参数符号：

```
#{id}
```

这就告诉 MyBatis 通过 JDBC 创建一个预处理语句参数，这样的一个参数在 SQL 中会由一个“?”来标识，并被传递到一个新的预处理语句中，就像这样：

```
// Similar JDBC code, NOT MyBatis…
String selectPerson = "SELECT * FROM PERSON WHERE ID=?";PreparedStatement
ps = conn.prepareStatement(selectPerson);
ps.setInt(1,id);
```

当然，这需要很多单独的 JDBC 的代码来提取结果并将它们映射到对象实例中，这就是 MyBatis 节省你时间的地方。我们需要深入了解参数和结果映射，细节部分我们下面来了解。select 元素有很多属性允许你配置，来决定每条语句的作用细节。

```
<select
  id="selectPerson"
  parameterType="int"
  parameterMap="deprecated"
  resultType="hashmap"
  resultMap="personResultMap"
  flushCache="false"
  useCache="true"
  timeout="10000"
  fetchSize="256"
  statementType="PREPARED"
  resultSetType="FORWARD_ONLY">
```

Select 属性	
属性	描述
id	在命名空间中唯一的标识符，可以被用来引用这条语句。
parameterType	将会传入这条语句的参数类的完全限定名或别名。这个属性是可选的，因为 MyBatis 可以通过 TypeHandler 推断出具体传入语句的参数，默认值为 unset。
parameterMap	这是引用外部 parameterMap 的已经被废弃的方法。使用内联参数映射和 parameterType 属性。
resultType	从这条语句中返回的期望类型的类的完全限定名或别名。注意如果是集合情形，那应该是集合可以包含的类型，而不能是集合本身。使用 resultType 或 resultMap，但不能同时使用。
resultMap	外部 resultMap 的命名引用。结果集的映射是 MyBatis 最强大的特性，对其有一个很好的理解的话，许多复杂映射的情形都能迎刃而解。使用 resultMap 或 resultType，但不能同时使用。
flushCache	将其设置为 true，任何时候只要语句被调用，都会导致本地缓存和二级

Select 属性	
属性	描述
	缓存都会被清空，默认值：false。
useCache	将其设置为 true，将会导致本条语句的结果被二级缓存，默认值：对 select 元素为 true。
timeout	这个设置是在抛出异常之前，驱动程序等待数据库返回请求结果的秒数。默认值为 unset（依赖驱动）。
fetchSize	这是尝试影响驱动程序每次批量返回的结果行数和这个设置值相等。默认值为 unset（依赖驱动）。
statementType	STATEMENT，PREPARED 或 CALLABLE 的一个。这会让 MyBatis 分别使用 Statement，PreparedStatement 或 CallableStatement，默认值：PREPARED。
resultSetType	FORWARD_ONLY，SCROLL_SENSITIVE 或 SCROLL_INSENSITIVE 中的一个，默认值为 unset （依赖驱动）。
databaseId	如果配置了 databaseIdProvider，MyBatis 会加载所有的不带 databaseId 或匹配当前 databaseId 的语句；如果带或者不带的语句都有，则不带的会被忽略。
resultOrdered	这个设置仅针对嵌套结果 select 语句适用：如果为 true，就是假设包含了嵌套结果集或是分组了，这样的话当返回一个主结果行的时候，就不会发生有对前面结果集的引用的情况。这就使得在获取嵌套的结果集的时候不至于导致内存不够用。默认值：false。
resultSets	这个设置仅对多结果集的情况适用，它将列出语句执行后返回的结果集并每个结果集给一个名称，名称是逗号分隔的。

推荐链接：
http://www.mybatis.org/mybatis-3/zh/sqlmap-xml.html

8. SqlMapConfig.xml 配置文件

8.1. 主要元素

SqlMapConfig.xml 中配置的内容和顺序如下：

properties（属性）
settings（全局配置参数）
typeAliases（类型别名）
typeHandlers（类型处理器）
objectFactory（对象工厂）
plugins（插件）
environments（环境集合属性对象）
　　environment（环境子属性对象）
　　　　transactionManager（事务管理）
　　　　dataSource（数据源）
mappers（映射器）

8.2. properties（属性）

将内部的配置文件外在化，在整个配置文件中被用来替换需要动态配置的属性值
在 classpath 下定义 db.properties 文件

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/examsystem?characterEncoding=utf-8
jdbc.username=root
jdbc.password=123456
```

SqlMapConfig.xml 引用如下：

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

注意： MyBatis 将按照下面的顺序来加载属性：

- 在 properties 元素体内定义的属性首先被读取。
- 然后会读取 properties 元素中 resource 或 url 加载的属性，它会覆盖已读取的同名属性。

8.3. typeAliases（类型别名）

<typeAliases> 为 Java 类型设置一个短的名字。它只和 XML 配置有关，存在的意义仅在于用来减少类完全限定名的冗余。

8.3.1. 自定义别名

在 SqlMapConfig.xml 中配置：

```
<!-- 自定义别名 -->
<typeAliases>
    <!-- 单个别名定义 -->
    <!-- <typeAlias type="com.neuesoft.sz.pojo.Grade" alias="grade"/> -->

    <!-- 批量别名定义（推荐） -->
    <!-- package：指定包名称来为该包下的po 类声明别名，默认的别名就是类名（首字母大小写都可） -->
    <package name="com.neuesoft.sz.pojo" />
</typeAliases>
```

type 属性用于指定需要被定义别名的类的全类名
alias 属性的属性值就是定义的别名

当 POJO 类过多时，还可以通过自动扫描包的形式自定义别名，具体如下

```
<!-- 自定义别名 -->
<typeAliases>
    <package name="com.neuesoft.sz.pojo" />
</typeAliases>
```

上述代码，MyBatis 会将所有 com.neuesoft.sz.pojo 包中的 POJO 类以首字母小写的非限定类名来作为它的别名，如 com.neuesoft.sz.pojo.Grade 类的别名为 grade

8.3.2. MyBatis 支持别名

mybatis 框架还默认为许多常见的 Java 类型提供了相应的类型别名，如下所示

别名	映射的类型
_byte	byte
_long	long
_short	short
_int	int
_integer	int
_double	double
_float	float
_boolean	boolean
string	String
byte	Byte
long	Long
short	Short
int	Integer

integer	Integer
double	Double
float	Float
boolean	Boolean
date	Date
decimal	BigDecimal
bigdecimal	BigDecimal
map	Map

注意：由于别名不区分大小写，所以在使用是需要注意重复定义的覆盖问题。

8.4. mappers（映射器）

Mapper 配置的几种方法

8.4.1. <mapper resource=" " />

使用相对于类路径的资源如：

```
<mapper resource="sqlmap/Grade.xml" />
```

8.4.2. <mapper class=" " />

使用 mapper 接口类路径如：

```
    <mapper class="com.neuesoft.sz.mapper.GradeMapper"/>
```

注意：此种方法要求 mapper 接口名称和 mapper 映射文件名称相同，且放在同一个目录中。

8.4.3. <package name=""/>

注册指定包下的所有 mapper 接口如：

```
    <package name="com.neuesoft.sz.mapper"/>
```

注意：此种方法要求 mapper 接口名称和 mapper 映射文件名称相同，且放在同一个目录中。