Mybatis从入门到放弃_chencc

第一章 Mybatis入门

1.1 mybatis简介

来看官网的介绍,传送门

MyBatis 是一款优秀的持久层框架，它支持自定义 SQL、存储过程以及高级映射。MyBatis 免除了几乎所有的 JDBC 代码以及设置参数和获取结果集的工作。MyBatis 可以通过简单的 XML 或注解来配置和映射原始类型、接口和 Java POJO（Plain Old Java Objects，普通老式 Java 对象）为数据库中的记录。

首先提取出来几个关键词

- 持久层框架
- 支持自定义SQL,存储过程,以及高级映射
- 免除了几乎所有的JDBC代码以及设置参数和获取结果集的工作
- 可以通过简单的XML配置文件或者注解来配置和映射原始类型,接口和Java Pojo为数据库中的记录

  这句话怎么理解呢?

  1.可以通过简单的XML配置文件或者注解来配置,原始类型,接口(Interface),Java pojo(实际就是JavaBeans)

  2.可以通过简单的XML配置文件或者注解来映射,原始类型,接口(interface),Java pojo(JavaBeans)为数据库中的记录

上面这些配置就做了一件事情,配置接口或者JavaBean的相关属性,并映射为数据库中的记录,数据都被映射在了数据库中,自然是已经持久化了!

有什么特点

- 通过mapper.xml配置文件方式将SQL与接口进行映射,SQL与代码分离,耦合程度低
- 简单易学：本身就很小且简单。没有任何第三方依赖，最简单安装只要两个jar文件+配置几个sql映射文件易于学习，易于使用，通过文档和源代码，可以比较完全的掌握它的设计思路和实现。
- 灵活：mybatis不会对应用程序或者数据库的现有设计强加任何影响。 sql写在xml里，便于统一管理和优化。通过sql语句可以满足操作数据库的所有需求。
- 解除sql与程序代码的耦合：通过提供DAO层，将业务逻辑和数据访问逻辑分离，使系统的设计更清晰，更易维护，更易单元测试。sql和代码的分离，提高了可维护性。
- 提供映射标签，支持对象与数据库的orm字段关系映射
- 提供对象关系映射标签，支持对象关系组建维护
- 提供xml标签，支持编写动态sql。

1.2 安装Mybatis导入相关依赖

要想在项目中安装mybatis非常简单,只需要导入mybatis相关的jar包就行,如果该项目是一个普通的项目,

要使用 MyBatis,只需将 mybatis-x.x.x.jar 文件置于类路径（classpath）中即可。

如果该项目使用Maven构建工具构建,则在pom.xml文件中引入依赖即可

```
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>x.x.x</version>
</dependency>
```

现在的项目基本都会使用Maven来构建吧!我这里也创建一个普通的Maven项目,

创建完成后,

删除掉src目录 ,至于为什么要删除 这里我们这个新建的Maven项目当作一个父项目来使用,后面要建立关于mybatis的项目都可以依赖这个父项目,减少不必要的麻烦

根据官网所说,要在maven项目中使用mybatis,只需要依赖相关的jar包即可

当然我们在项目中导入一个mybatis的依赖是远远不够的!

mybatis的出现旨在封装我们以前需要在Java代码中利用JDBC代码去操作数据库的那一部分内容的,

这也是我们学习技术的一个思路:

- **平白无故的为什么会产生这个技术?**
- **这个技术是用来解决哪类问题的?**
- **这个技术代替了我们以前的那个技术所需要作的工作?**

因为我们之前在Java代码中使用JDBC去操作数据库,不仅仅会造成代码冗余,而且会出现SQL与代码之间的高耦合,不利用开发并且不利用项目维护,所以出现了Mybatis,还有一个叫作(Hibernate)的持久层框架,因为mybatis更加好用所以使用了mybatis,现在我们知道了Mybatis这项技术的出现使用来解决Java访问数据库持久化操作的!代替了我们以前利用Java代码操作数据库的工作

上面说了这么多,就一句话,项目中要使用mybatis,单纯的依赖一个mybatis的jar是不够的,mybatis是用来做数据持久化工作的,我们项目中就需要关于mysql的驱动,要有数据源,一般都会再添加一个Junit依赖,方便我们开发的时候进行测试


```
<dependencies>
    <!--导入Junit依赖-->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>

    <!--导入mybatis3依赖-->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.6</version>
    </dependency>

    <!--导入mysql驱动-->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.40</version>
    </dependency>

    <!--导入数据库连接池-->
    <dependency>
```

```
        <groupId>com.alibaba</groupId>
        <artifactId>druid</artifactId>
        <version>1.1.10</version>
    </dependency>
</dependencies>
```

1.3 构建SqlSessionFactory

1.3.1 使用XML配置文件构建

这里官方文档有一句很重要的话,

每一个基于Mybatis的应用都是以SqlSessionFactory实例为核心的,

SqlSessionFactory的实例可以通过SqlSessionFactoryBuilder获得

其实在Mybatis中实际帮助我们执行SQL语句的一个SqlSession对象,这个对象是由SqlSessionFactory实例创建出来的,

设计模式在这里很明显的就体现了出来

- SqlSessionFactory 使用的是工厂模式
- SqlSessionFactoryBuilder 使用的是建造者模式

由此,我们知道想要构建SqlSessionFactory对象,就必须有SqlSessionFactoryBuilder,

即创建SqlSessionFactory为什么可以有两种方式?是因为创建SqlSessionFactoryBuilder有两种方式

来看官方的操作

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。 但也可以使用任意的输入流（InputStream）实例，比如用文件路径字符串或 file:// URL 构造的输入流。MyBatis 包含一个名叫 Resources 的工具类，它包含一些实用方法，使得从类路径或其它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);
```

1.3.2 Mybatis核心配置文件

XML 配置文件中包含了对 MyBatis 系统的核心设置，包括获取数据库连接实例的数据源（DataSource）以及决定事务作用域和控制方式的事务管理器（TransactionManager）。后面会再探讨 XML 配置文件的详细内容，这里先给出一个简单的示例：

```
<?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>
  <environments default="development">
    <environment id="development">
       <!--事务管理器-->
      <transactionManager type="JDBC"/>

      <!--dataSource数据源-->
```

```
      <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>

  <!--mapper映射文件-->
  <mappers>
    <mapper resource="org/mybatis/example/BlogMapper.xml"/>
  </mappers>

</configuration>
```

当然，还有很多可以在 XML 文件中配置的选项，上面的示例仅罗列了最关键的部分。 注意 XML 头部的声明，它用来验证 XML 文档的正确性。environment 元素体中包含了事务管理和连接池的配置。mappers 元素则包含了一组映射器（mapper），这些映射器的 XML 映射文件包含了 SQL 代码和映射定义信息。

1.3.3 利用Java代码构建

如果你更愿意直接从 Java 代码而不是 XML 文件中创建配置，或者想要创建你自己的配置建造器，MyBatis 也提供了完整的配置类，提供了所有与 XML 文件等价的配置项。


```
DataSource dataSource = BlogDataSourceFactory.getBlogDataSource();
TransactionFactory transactionFactory = new JdbcTransactionFactory();
Environment environment = new Environment("development", transactionFactory,
dataSource);
Configuration configuration = new Configuration(environment);
configuration.addMapper(BlogMapper.class);
SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(configuration);
```

注意该例中，configuration 添加了一个映射器类（mapper class）。映射器类是 Java 类，它们包含 SQL 映射注解从而避免依赖 XML 文件。不过，由于 Java 注解的一些限制以及某些 MyBatis 映射的复杂性，要使用大多数高级映射（比如：嵌套联合映射），仍然需要使用 XML 配置。有鉴于此，如果存在一个同名 XML 配置文件，MyBatis 会自动查找并加载它（在这个例子中，基于类路径和 BlogMapper.class 的类名，会加载 BlogMapper.xml）。具体细节稍后讨论。

1.4 获取SqlSession对象

我们可以通过SqlSessionFactory实例获取SqlSession对象，

SqlSession对象就是真正执行Sql语句的实例

既然有了 SqlSessionFactory,顾名思义,我们可以从中获得 SqlSession 的实例。

SqlSession 提供了在数据库执行 SQL 命令所需的所有方法。

你可以通过 SqlSession 实例来直接执行已映射的 SQL 语句。

例如：

```
try (SqlSession session = sqlSessionFactory.openSession()) {
  Blog blog = (Blog)
session.selectOne("org.mybatis.example.BlogMapper.selectBlog", 101);
}
```

诚然,这种方式能够正常工作,对使用旧版本 MyBatis 的用户来说也比较熟悉。

但现在有了一种更简洁的方式——使用和指定语句的参数和返回值相匹配的接口（比如 BlogMapper.class），

现在你的代码不仅更清晰，更加类型安全，还不用担心可能出错的字符串字面值以及强制类型转换。

例如：

```
try (SqlSession session = sqlSessionFactory.openSession()) {
  BlogMapper mapper = session.getMapper(BlogMapper.class);
  Blog blog = mapper.selectBlog(101);
}
```

这里的SqlSession对象都执行了已经映射的Sql语句,不同之处在于

第一种方法直接把映射的sql语句写死了,如果要执行另一个Sql语句就需要在写一个这样的方法,非常容易出错

第二种方法通过SqlSession对象获取到了Sql的映射文件,这样一来的好处就是,这个Sql映文件中的全部Sql语句SqlSession对象都能拿得到,并返回一个对象,以后想执行哪一个Sql就利用返回的这个对象来调用就好,比的一种方法更智能,耦合性也更低!

1.5 Sql映射文件mapper.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="org.mybatis.example.BlogMapper">
  <select id="selectBlog" resultType="Blog">
    select * from Blog where id = #{id}
  </select>
</mapper>
```

1.4章节中提到的SqlSession对象执行已经映射到sql中的第二种方式的好处就在这里体现的淋漓尽致,如果这个mapper.xml映射文件中有好多,需要映射的sql语句,难道把每一个映射好的sql都利用代码写一行吗?显然不太现实

为了这个简单的例子，我们似乎写了不少配置，但其实并不多。在一个 XML 映射文件中，可以定义无数个映射语句，这样一来，XML 头部和文档类型声明部分就显得微不足道了。文档的其它部分很直白，容易理解。 它在命名空间 “org.mybatis.example.BlogMapper” 中定义了一个名为 “selectBlog” 的映射语句，这样你就可以用全限定名 “org.mybatis.example.BlogMapper.selectBlog” 来调用映射语句了，就像上面例子中那样：

```
Blog blog = (Blog) session.selectOne("org.mybatis.example.BlogMapper.selectBlog",
101);
```

你可能会注意到，这种方式和用全限定名调用 Java 对象的方法类似。这样，该命名就可以直接映射到在命名空间中同名的映射器类，并将已映射的 select 语句匹配到对应名称、参数和返回类型的方法。因此你就可以像上面那样，不费吹灰之力地在对应的映射器接口调用方法，就像下面这样：

```
BlogMapper mapper = session.getMapper(BlogMapper.class);
Blog blog = mapper.selectBlog(101);
```

第二种方法有很多优势，首先它不依赖于字符串字面值，会更安全一点；其次，如果你的 IDE 有代码补全功能，那么代码补全可以帮你快速选择到映射好的 SQL 语句。

1.6 Namespaces(命名空间)

Namespaces(命名空间的作用)

- **第一个作用**

  使用全限定类名实现了接口的绑定 并且将映射的不同的SQL语句隔离开来

- **第二个作用 命名解析**

  1. 全限定名（比如 “com.mypackage.MyMapper.selectAllThings）将被直接用于查找及使用。
  2. 短名称（比如 “selectAllThings”）如果全局唯一也可以作为一个单独的引用。 如果不唯一，有两个或两个以上的相同名称（比如 “com.foo.selectAllThings” 和 “com.bar.selectAllThings”），那么使用时就会产生“短名称不唯一”的错误，这种情况下就必须使用全限定名。

1.7 注解实现Sql映射

对于像 BlogMapper 这样的映射器类来说，还有另一种方法来完成语句映射。 它们映射的语句可以不用 XML 来配置，而可以使用 Java 注解来配置。比如，上面的 XML 示例可以被替换成如下的配置：


```
package org.mybatis.example;
public interface BlogMapper {
  @Select("SELECT * FROM blog WHERE id = #{id}")
  Blog selectBlog(int id);
}
```

使用注解来映射简单语句会使代码显得更加简洁，但对于稍微复杂一点的语句，Java 注解不仅力不从心，还会让你本就复杂的 SQL 语句更加混乱不堪。 因此，如果你需要做一些很复杂的操作，最好用 XML 来映射语句。

选择何种方式来配置映射，以及认为是否应该要统一映射语句定义的形式，完全取决于你和你的团队。换句话说，永远不要拘泥于一种方式，你可以很轻松的在基于注解和 XML 的语句映射方式间自由移植和切换。

一句话总结;注解也可以实现sql的映射,但是稍微复杂一点sql语句,注解就不太支持,有力不从心,还是得使用mapper.xml映射文件方式

1.8 作用域(Scope)和生命周期

SqlSessionFactoryBuilder

这个类可以被实例化、使用和丢弃，一旦创建了 SqlSessionFactory，就不再需要它了。

因此 SqlSessionFactoryBuilder 实例的最佳作用域是方法作用域（也就是局部方法变量);

你可以重用 SqlSessionFactoryBuilder 来创建多个 SqlSessionFactory 实例，但最好还是不要一直保留着它，以保证所有的 XML 解析资 源可以被释放给更重要的事情。

SqlSessionFactory

SqlSessionFactory 一旦被创建就应该在应用的运行期间一直存在，没有任何理由丢弃它或重新创建另一个实例。

使用 SqlSessionFactory 的最佳实践是在应用运行期间不要重复创建多次,多次重建 SqlSessionFactory 被视为一种代码“坏习惯”。

因此 SqlSessionFactory 的最佳作用域是应用作用域

有很多方法可以做到,最简单的就是使用单例模式或者静态单例模式。

SqlSession

每个线程都应该有它自己的 SqlSession 实例,SqlSession 的实例不是线程安全的，因此是不能被共享的，所以它的最佳的作用域是请求或方法作用域.

绝对不能将 SqlSession 实例的引用放在一个类的静态域甚至一个类的实例变量也不行.也绝不能将 SqlSession 实例的引用放在任何类型的托管作用域中，比如 Servlet 框架中的 HttpSession。

如果你现在正在使用一种 Web 框架，考虑将 SqlSession 放在一个和 HTTP 请求相似的作用域中。

换句话说，每次收到 HTTP 请求，就可以打开一个 SqlSession，返回一个响应后，就关闭它。 这个关闭操作很重要，为了确保每次都能执行关闭操作，你应该把这个关闭操作放到 finally 块中。 下面的示例就是一个确保 SqlSession 关闭的标准模式：


```
try (SqlSession session = sqlSessionFactory.openSession()) {
  // 你的应用逻辑代码
}
```

在所有代码中都遵循这种使用模式，可以保证所有数据库资源都能被正确地关闭。

总结:

- SqlSessionFactoryBuilder实例创建出了SqlSessionFactory实例后就应该被丢弃,不需要再继续被创建或者怎么样
- SqlSessionFcatory实例是用来创建SqlSession实例的,应用运行的整个声明周期都应该存在,
- SqlSession实例每个线程应该对应一个SqlSession实例,该实例不是安全的,不能共享,SqlSession是用来执行映射器映射的Sql语句的,相当于Java中的JDBC,执行完对应的sql语句后,就应该被摧毁,因此,SqlSession实例的作用域应该只存在于请求或方法作用域,用完及摧毁,避免浪费系统资源

1.9 映射器实例mapper

映射器实例

映射器是一些绑定映射语句的接口。映射器接口的实例是从 SqlSession 中获得的。虽然从技术层面上来讲，任何映射器实例的最大作用域与请求它们的 SqlSession 相同。但方法作用域才是映射器实例的最合适的作用域。 也就是说，映射器实例应该在调用它们的方法中被获取，使用完毕之后即可丢弃。 映射器实例并不需要被显式地关闭。尽管在整个请求作用域保留映射器实例不会有什么问题，但是你很快会发现，在这个作用域上管理太多像 SqlSession 的资源会让你忙不过来。 因此，最好将映射器放在方法作用域内。就像下面的例子一样：

```
try (SqlSession session = sqlSessionFactory.openSession()) {
  BlogMapper mapper = session.getMapper(BlogMapper.class);
  // 你的应用逻辑代码
}
```

第二章 XML配置文件

2.1 properties属性

mybatis-config.xml 核心配置文件

```
<?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>

    <!--导入外部的数据库配置信息,方便动态引用-->
    <properties resource="jdbc.properties"></properties>

    <!--配置mybatis的环境-->
    <environments default="development">
        <environment id="development">

            <!--事务管理器-->
            <transactionManager type="JDBC"/>

            <!--数据源类型-->
            <dataSource type="POOLED">
                <!--引入jdbc数据库信息-->
                    <property name="driver" value="${driver}"/>
                    <property name="url" value="${url}"/>
                    <property name="username" value="${username}"/>
                    <property name="password" value="${password}"/>
            </dataSource>
        </environment>
    </environments>

    <!--引入映射文件-->
    <mappers>
        <mapper resource="mybatis/mappers/UserMapper.xml"/>
    </mappers>
</configuration>
```


jdbc.properties配置文件

```
driver=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/devdata?useUnicode=true&characterEncoding=utf-8
username=root
password=mysql
```

在核心配置文件中有一个标签

这个标签是用来引入JDBC数据库连接信息的,这样写在配置文件中是写死的,

mybatis中可以使用标签,来引入外部的配置配置进行动态获取,不仅可以动态引入也可以在这个标签内写属性

类似于这样:

```
<properties resource="jdbc.properties">
    <property name="driver0" value="xxx"/>
</properties>
```

我们不仅可以通过xml配置文件的方式进行传值,也可以在创建SqlSessionFactory实例时传入参数

也可以在 SqlSessionFactoryBuilder.build() 方法中传入属性值。例如:

```
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader, props);

// ... 或者 ...

SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader,
environment, props);
```


以上三种方式可以设定关于数据库的参数的

那么如果同时设置呢?到底哪一个生效?

官方给出的优先级顺序如下:

方法参数里的参数有限级大于Properties引用的大于property中参数,同名参数覆盖,不同名参数互补

2.2 property参数占位符

这里有一个很有趣的东西

如果我们关于数据库的某项参数我们没有配置,可以设置一个默认值,当我们没有配置时,这个默认值会生效

这个特性默认是关闭的。要启用这个特性，需要添加一个特定的属性来开启这个特性.

例如:

```
<properties resource="org/mybatis/example/config.properties">
  <!-- ... -->

   <!-- 启用默认值特性 -->
  <property name="org.apache.ibatis.parsing.PropertyParser.enable-default-value"
value="true"/>
</properties>


<!--开启后就可以在dataSource中使用默认值-->
<dataSource type="POOLED">
  <!-- ... -->
  <property name="username" value="${username:ut_user}"/> <!-- 如果属性 'username'
没有被配置，'username' 属性的值将为 'ut_user' -->
</dataSource>
```

2.3 settings设置

在mybatis核心配置文件中,可以通过标签来设置mybatis，

通过这些设置可以改变mybatis的运行时行为,这是mybatis中极为重要的调整

2.3.1 完成的settings配置文件

通常在mybatis核心配置文件中包含以下设置，一个完成的settings设置配置如下：


```
<settings>

  <!--开启二级缓存，一级缓存是默认开启-->
  <setting name="cacheEnabled" value="true"/>

  <!--开启懒加载-->
  <setting name="lazyLoadingEnabled" value="true"/>

  <!--   是否允许单个语句返回多结果集（需要数据库驱动支持）。-->
  <setting name="multipleResultSetsEnabled" value="true"/>

  <!--使用列标签代替列名-->
  <setting name="useColumnLabel" value="true"/>

  <!--   允许 JDBC 支持自动生成主键，需要数据库驱动支持。如果设置为 true，将强制使用自动生成
主键-->
  <setting name="useGeneratedKeys" value="false"/>

  <!--指定 MyBatis 应如何自动映射列到字段或属性。-->
  <setting name="autoMappingBehavior" value="PARTIAL"/>

  <!--指定发现自动映射目标未知列（或未知属性类型）的行为。-->
  <setting name="autoMappingUnknownColumnBehavior" value="WARNING"/>

  <!--配置默认的执行器。SIMPLE 就是普通的执行器；-->
  <setting name="defaultExecutorType" value="SIMPLE"/>

  <!--设置超时时间，它决定数据库驱动等待数据库响应的秒数。-->
  <setting name="defaultStatementTimeout" value="25"/>
```

```
  <!--    为驱动的结果集获取数量（fetchSize）设置一个建议值。此参数只可以在查询设置中被覆盖-->
  <setting name="defaultFetchSize" value="100"/>

  <!--是否允许在嵌套语句中使用分页（RowBounds）。如果允许使用则设置为 false。-->
  <setting name="safeRowBoundsEnabled" value="false"/>

  <!--是否开启驼峰命名自动映射，即从经典数据库列名 A_COLUMN 映射到经典 Java 属性名
aColumn。-->
  <setting name="mapUnderscoreToCamelCase" value="false"/>

  <!--mybatis本机缓存中的作用域,默认为session-->
  <setting name="localCacheScope" value="SESSION"/>

  <!--当没有为参数指定特定的 JDBC 类型时，空值的默认 JDBC 类型。-->
  <setting name="jdbcTypeForNull" value="OTHER"/>

  <!--    指定对象的哪些方法触发一次延迟加载-->
  <setting name="lazyLoadTriggerMethods"
value="equals,clone,hashCode,toString"/>
</settings>
```

2.3.2 官方表格

以上为官方提供的给关于mybatis中关于settings中的所有配置,其实除过这些配置之外还有其余的配置,以下为官方提供的表格，请参考表格

设置名	描述	有效值	默认值
cacheEnabled	全局性地开启或关闭所有映射器配置文件中已配置的任何缓存。	true | false	true
lazyLoadingEnabled	延迟加载的全局开关。当开启时，所有关联对象都会延迟加载。 特定关联关系中可通过设置 `fetchType` 属性来覆盖该项的开关状态。	true | false	false
aggressiveLazyLoading	开启时，任一方法的调用都会加载该对象的所有延迟加载属性。 否则，每个延迟加载属性会按需加载（参考 `lazyLoadTriggerMethods`）。	true | false	false （在 3.4.1 及之前的版本中默认为 true）
multipleResultSetsEnabled	是否允许单个语句返回多结果集（需要数据库驱动支持）。	true | false	true
useColumnLabel	使用列标签代替列名。实际表现依赖于数据库驱动，具体可参考数据库驱动的相关文档，或通过对比测试来观察。	true | false	true
useGeneratedKeys	允许 JDBC 支持自动生成主键，需要数据库驱动支持。如果设置为 true，将强制使用自动生成主键。尽管一些数据库驱动不支持此特性，但仍可正常工作（如 Derby）。	true | false	False
autoMappingBehavior	指定 MyBatis 应如何自动映射列到字段或属性。 NONE 表示关闭自动映射；PARTIAL 只会自动映射没有定义嵌套结果映射的字段。 FULL 会自动映射任何复杂的结果集（无论是否嵌套）。	NONE, PARTIAL, FULL	PARTIAL
autoMappingUnknownColumnBehavior	指定发现自动映射目标未知列（或未知属性类型）的行为。`NONE`：不做任何反应`WARNING`：输出警告日志（`'org.apache.ibatis.session.AutoMappingUnknownColumnBehavior'` 的日志等级必须设置为 `WARN`）`FAILING`：映射失败 (抛出 `SqlSessionException`)	NONE, WARNING, FAILING	NONE
defaultExecutorType	配置默认的执行器。SIMPLE 就是普通的执行器；REUSE 执行器会重用预处理语句（PreparedStatement）； BATCH 执行器不仅重用语句还会执行批量更新。	SIMPLE REUSE BATCH	SIMPLE
defaultStatementTimeout	设置超时时间，它决定数据库驱动等待数据库响应的秒数。	任意正整数	未设置 (null)
defaultFetchSize	为驱动的结果集获取数量（fetchSize）设置一个建议值。此参数只可以在查询设置中被覆盖。	任意正整数	未设置 (null)
defaultResultSetType	指定语句默认的滚动策略。（新增于 3.5.2）	FORWARD_ONLY | SCROLL_SENSITIVE | SCROLL_INSENSITIVE | DEFAULT（等同于未设置）	未设置 (null)
safeRowBoundsEnabled	是否允许在嵌套语句中使用分页（RowBounds）。如果允许使用则设置为 false。	true | false	False
safeResultHandlerEnabled	是否允许在嵌套语句中使用结果处理器（ResultHandler）。如果允许使用则设置为 false。	true | false	True
mapUnderscoreToCamelCase	是否开启驼峰命名自动映射，即从经典数据库列名 A_COLUMN 映射到经典 Java 属性名 aColumn。	true | false	False
localCacheScope	MyBatis 利用本地缓存机制（Local Cache）防止循环引用和加速重复的嵌套查询。 默认值为 SESSION，会缓存一个会话中执行的所有查询。 若设置值为 STATEMENT，本地缓存将仅用于执行语句，对相同 SqlSession 的不同查询将不会进行缓存。	SESSION | STATEMENT	SESSION
jdbcTypeForNull	当没有为参数指定特定的 JDBC 类型时，空值的默认 JDBC 类型。 某些数据库驱动需要指定列的 JDBC 类型，多数情况直接用一般类型即可，比如 NULL、VARCHAR 或 OTHER。	JdbcType 常量，常用值：NULL、VARCHAR 或 OTHER。	OTHER
lazyLoadTriggerMethods	指定对象的哪些方法触发一次延迟加载。	用逗号分隔的方法列表。	equals,clone,hashCode,toString
defaultScriptingLanguage	指定动态 SQL 生成使用的默认脚本语言。	一个类型别名或全限定类名。	org.apache.ibatis.scripting.xmltags.XMLLanguageDriver
defaultEnumTypeHandler	指定 Enum 使用的默认 `TypeHandler` 。（新增于 3.4.5）	一个类型别名或全限定类名。	org.apache.ibatis.type.EnumTypeHandler
callSettersOnNulls	指定当结果集中值为 null 的时候是否调用映射对象的 setter（map 对象时为 put）方法，这在依赖于 Map.keySet() 或 null 值进行初始化时比较有用。注意基本类型（int、boolean 等）是不能设置成 null 的。	true | false	false
returnInstanceForEmptyRow	当返回行的所有列都是空时，MyBatis默认返回 `null` 。 当开启这个设置时，MyBatis会返回一个空实例。 请注意，它也适用于嵌套的结果集（如集合或关联）。（新增于 3.4.2）	true | false	false
logPrefix	指定 MyBatis 增加到日志名称的前缀。	任何字符串	未设置
logImpl	指定 MyBatis 所用日志的具体实现，未指定时将自动查找。	SLF4J | LOG4J | LOG4J2 | JDK_LOGGING | COMMONS_LOGGING | STDOUT_LOGGING | NO_LOGGING	未设置
proxyFactory	指定 Mybatis 创建可延迟加载对象所用到的代理工具。	CGLIB | JAVASSIST	JAVASSIST （MyBatis 3.3 以上）
vfsImpl	指定 VFS 的实现	自定义 VFS 的实现的类全限定名，以逗号分隔。	未设置
useActualParamName	允许使用方法签名中的名称作为语句参数名称。 为了使用该特性，你的项目必须采用 Java 8 编译，并且加上 `-parameters` 选项。（新增于 3.4.1）	true | false	true
configurationFactory	指定一个提供 `Configuration` 实例的类。 这个被返回的 Configuration 实例用来加载被反序列化对象的延迟加载属性值。 这个类必须包含一个签名为`static Configuration getConfiguration()` 的方法。（新增于 3.2.3）	一个类型别名或完全限定类名。	未设置
shrinkWhitespacesInSql	从SQL中删除多余的空格字符。请注意，这也会影响SQL中的文字字符串。 (新增于 3.5.5)	true | false	false
defaultSqlProviderType	Specifies an sql provider class that holds provider method (Since 3.5.6). This class apply to the `type`(or `value`) attribute on sql provider annotation(e.g. `@SelectProvider`), when these attribute was omitted.	A type alias or fully qualified class name	Not set

2.4 typeAliases(类型别名)

先说一下,类型别名有什么用?可以为Java类型设置一个简单的别名,用于减少代码的冗余

2.4.1 普通方式设置别名

```
<typeAliases>
  <typeAlias alias="Author" type="domain.blog.Author"/>
  <typeAlias alias="Blog" type="domain.blog.Blog"/>
  <typeAlias alias="Comment" type="domain.blog.Comment"/>
  <typeAlias alias="Post" type="domain.blog.Post"/>
  <typeAlias alias="Section" type="domain.blog.Section"/>
  <typeAlias alias="Tag" type="domain.blog.Tag"/>
</typeAliases>
```

2.4.2 包扫描设置别名

上述的方法,是用配置文件一一对于Java的类型,也可以通过包扫描的方法配置别名

```
<typeAliases>
  <package name="domain.blog"/>
</typeAliases>
```

2.4.3 注解设置别名

除过通过一一对应与包扫描方式设置Java类型的别名外，还可以通过注解方式设置Java的类型别名

每一个在包 `domain.blog` 中的 Java Bean，在没有注解的情况下，会使用 Bean 的首字母小写的非限定类名来作为它的别名。 比如 `domain.blog.Author` 的别名为 `author`；若有注解，则别名为其注解值。见下面的例子：

```
@Alias("author")
public class Author {
    ...
}
```

.....中间省略不需要学习,具体参考官方文档.....

2.5 环境配置environments

在mybatis核心配置文件中可以配置多个环境用于不同的开发场景,但是注意我们尽管可以配置多个环境,但是每一个SqlSessionFactory实例只能选择一个选择一个环境

所以，如果你想连接两个数据库，就需要创建两个 SqlSessionFactory 实例，每个数据库对应一个。而如果是三个数据库，就需要三个实例，依此类推，记起来很简单

- **每个数据库对应一个 SqlSessionFactory 实例**

为了指定创建哪种环境，只要将它作为可选的参数传递给 SqlSessionFactoryBuilder 即可。可以接受环境配置的两个方法签名是：

```
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader,
environment);
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader,
environment, properties);
```

如果忽略了环境参数，那么将会加载默认环境，如下所示：

```
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader);
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader,
properties);
```

environments 元素定义了如何配置环境。

```
<environments default="development">
  <environment id="development">
    <transactionManager type="JDBC">
      <property name="..." value="..."/>
    </transactionManager>
    <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>
```

注意一些关键点:

- 默认使用的环境 ID（比如：default="development"）。
- 每个 environment 元素定义的环境 ID（比如：id="development"）。
- 事务管理器的配置（比如：type="JDBC"）。
- 数据源的配置（比如：type="POOLED"）。

默认环境和环境 ID 顾名思义。 环境可以随意命名，但务必保证默认的环境 ID 要匹配其中一个环境 ID。

2.6 事务管理器

在 MyBatis 中有两种类型的事务管理器（也就是 type="[JDBC|MANAGED]"）：

JDBC – 这个配置直接使用了 JDBC 的提交和回滚设施，它依赖从数据源获得的连接来管理事务作用域。

```
  <!--事务管理器-->
            <transactionManager type="JDBC"/>
```

MANAGED – 这个配置几乎没做什么。它从不提交或回滚一个连接，而是让容器来管理事务的整个生命周期（比如 JEE 应用服务器的上下文）。 默认情况下它会关闭连接。然而一些容器并不希望连接被关闭，因此需要将 closeConnection 属性设置为 false 来阻止默认的关闭行为。例如:

```
<transactionManager type="MANAGED">
  <property name="closeConnection" value="false"/>
</transactionManager>
```

提示 如果你正在使用 Spring + MyBatis，则没有必要配置事务管理器，因为 Spring 模块会使用自带的管理器来覆盖前面的配置。

2.7 数据源 dataSource

数据源（dataSource）

dataSource 元素使用标准的 JDBC 数据源接口来配置 JDBC 连接对象的资源。

- 大多数 MyBatis 应用程序会按示例中的例子来配置数据源。虽然数据源配置是可选的，但如果要启用延迟加载特性，就必须配置数据源。

有三种内建的数据源类型(也就是 type="[UNPOOLED|POOLED|JNDI]")

第一种 UNPOOLED 表示无数据源,每次请求时打开和关闭连接

第二种 POOLED 代表有数据源,有一个"池"的概念,每次请求过来不需要再创建对象,从"池"中拿即可

第三种 JNDI 这个数据源实现是为了能在如 EJB 或应用服务器这类容器中使用，容器可以集中或在外部配置数据源，然后放置一个 JNDI 上下文的数据源引用

```
<!--数据源类型-->
<dataSource type="POOLED">
    <!--引入jdbc数据库信息-->
        <property name="driver" value="${driver}"/>
        <property name="url" value="${url}"/>
        <property name="username" value="${username}"/>
        <property name="password" value="${password:mysql}"/>
</dataSource>
```

关于数据源的其余配置请参考官方文档

中间省略不需要学习,具体参考官方文档

2.8 映射器 mappers

指定我们映射的Sql语句的mapper.xml文件位置

既然 MyBatis 的行为已经由上述元素配置完了，我们现在就要来定义 SQL 映射语句了。 但首先，我们需要告诉 MyBatis 到哪里去找到这些语句。 在自动查找资源方面，Java 并没有提供一个很好的解决方案，所以最好的办法是直接告诉 MyBatis 到哪里去找映射文件。 你可以使用相对于类路径的资源引用，或完全限定资源定位符（包括 `file:///` 形式的 URL），或类名和包名等。例如：

```
<!-- 使用相对于类路径的资源引用 -->
<mappers>
  <mapper resource="org/mybatis/builder/AuthorMapper.xml"/>
  <mapper resource="org/mybatis/builder/BlogMapper.xml"/>
  <mapper resource="org/mybatis/builder/PostMapper.xml"/>
</mappers>

-------------------------------------------------------------------------------
---------------------------
<!-- 使用完全限定资源定位符（URL） -->
<mappers>
  <mapper url="file:///var/mappers/AuthorMapper.xml"/>
  <mapper url="file:///var/mappers/BlogMapper.xml"/>
  <mapper url="file:///var/mappers/PostMapper.xml"/>
</mappers>
```

```
-----------------------------------------------------------------------------------------
---------------------------
<!-- 使用映射器接口实现类的完全限定类名 -->
<mappers>
  <mapper class="org.mybatis.builder.AuthorMapper"/>
  <mapper class="org.mybatis.builder.BlogMapper"/>
  <mapper class="org.mybatis.builder.PostMapper"/>
</mappers>

-----------------------------------------------------------------------------------------
---------------------------
<!-- 将包内的映射器接口实现全部注册为映射器 -->
<mappers>
  <package name="org.mybatis.builder"/>
</mappers>
```

第三章 XML映射文件

3.1 映射器

MyBatis 的真正强大在于它的语句映射，这是它的魔力所在。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 致力于减少使用成本，让用户能更专注于 SQL 代码。

SQL 映射文件只有很少的几个顶级元素（按照应被定义的顺序列出）：

- `cache` – 该命名空间的缓存配置。
- `cache-ref` – 引用其它命名空间的缓存配置。
- `resultMap` – 描述如何从数据库结果集中加载对象，是最复杂也是最强大的元素。
- `parameterMap` – 老式风格的参数映射。此元素已被废弃，并可能在将来被移除！请使用行内参数映射。文档中不会介绍此元素。
- `sql` – 可被其它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。

具体内容参考官方文档,这里不做介绍,

3.2 sql

这个元素可以用来定义可重用的 SQL 代码片段，以便在其它语句中使用。 参数可以静态地（在加载的时候）确定下来，并且可以在不同的 include 元素中定义不同的参数值。比如：

```
<sql id="userColumns"> ${alias}.id,${alias}.username,${alias}.password </sql>
```

这个 SQL 片段可以在其它语句中使用，例如：

```
<select id="selectUsers" resultType="map">
  select
    <include refid="userColumns"><property name="alias" value="t1"/></include>,
    <include refid="userColumns"><property name="alias" value="t2"/></include>
  from some_table t1
    cross join some_table t2
</select>
```

也可以在 include 元素的 refid 属性或内部语句中使用属性值，例如：

```
<sql id="sometable">
  ${prefix}Table
</sql>

<sql id="someinclude">
  from
    <include refid="${include_target}"/>
</sql>

<select id="select" resultType="map">
  select
    field1, field2, field3
  <include refid="someinclude">
    <property name="prefix" value="Some"/>
    <property name="include_target" value="sometable"/>
  </include>
</select>
```


中间部分省略,具体参考官方文档

3.2 结果映射

resultMap是mybatis中最重要最强大的元素,默认是隐性的

resultMap的设计思想是对简单的Sql语句做到零配置,对复杂的语句只需要描述它们之间的关系即可

之前你已经见过简单映射语句的示例，它们没有显式指定 `resultMap`。比如：

```
<select id="selectUsers" resultType="map">
  select id, username, hashedPassword
  from some_table
  where id = #{id}
</select>
```

上述语句只是简单地将所有的列映射到 `HashMap` 的键上，这由 `resultType` 属性指定。虽然在大部分情况下都够用，但是 HashMap 并不是一个很好的领域模型。你的程序更可能会使用 JavaBean 或 POJO（Plain Old Java Objects，普通老式 Java 对象）作为领域模型。MyBatis 对两者都提供了支持。看看下面这个 JavaBean：

```
package com.someapp.model;
public class User {
  private int id;
  private String username;
```

```
  private String hashedPassword;

  public int getId() {
    return id;
  }
  public void setId(int id) {
    this.id = id;
  }
  public String getUsername() {
    return username;
  }
  public void setUsername(String username) {
    this.username = username;
  }
  public String getHashedPassword() {
    return hashedPassword;
  }
  public void setHashedPassword(String hashedPassword) {
    this.hashedPassword = hashedPassword;
  }
}
```

基于 JavaBean 的规范，上面这个类有 3 个属性：id，username 和 hashedPassword。这些属性会对应到 select 语句中的列名。

这样的一个 JavaBean 可以被映射到 `ResultSet`，就像映射到 `HashMap` 一样简单。

```
<select id="selectUsers" resultType="com.someapp.model.User">
  select id, username, hashedPassword
  from some_table
  where id = #{id}
</select>
```


类型别名是你的好帮手。使用它们，你就可以不用输入类的全限定名了。比如：

```
<!-- mybatis-config.xml 中 -->
<typeAlias type="com.someapp.model.User" alias="User"/>

<!-- SQL 映射 XML 中 -->
<select id="selectUsers" resultType="User">
  select id, username, hashedPassword
  from some_table
  where id = #{id}
</select>
```

在这些情况下，MyBatis 会在幕后自动创建一个 `ResultMap`，再根据属性名来映射列到 JavaBean 的属性上。如果列名和属性名不能匹配上，可以在 SELECT 语句中设置列别名（这是一个基本的 SQL 特性）来完成匹配。比如：

```xml
<select id="selectUsers" resultType="User">
  select
    user _id             as "id",
    user_name           as "userName",
    hashed_password     as "hashedPassword"
  from some_table
  where id = #{id}
</select>
```

在学习了上面的知识后，你会发现上面的例子没有一个需要显式配置 `ResultMap`，这就是 `ResultMap` 的优秀之处——你完全可以不用显式地配置它们。 虽然上面的例子不用显式配置 `ResultMap`。 但为了讲解，我们来看看如果在刚刚的示例中，显式使用外部的 `resultMap` 会怎样，这也是解决列名不匹配的另外一种方式。

```xml
<resultMap id="userResultMap" type="User">
  <id property="id" column="user_id" />
  <result property="username" column="user_name"/>
  <result property="password" column="hashed_password"/>
</resultMap>
```

然后在引用它的语句中设置 `resultMap` 属性就行了（注意我们去掉了 `resultType` 属性）。比如:

```xml
<select id="selectUsers" resultMap="userResultMap">
  select user_id, user_name, hashed_password
  from some_table
  where id = #{id}
</select>
```


如果这个世界总是这么简单就好了。

3.2.1 字段名与属性名不一致

这时候如果出现属性名与字段名不一致的情况,

两种方式可以解决

第一种: 在相关的sql语句中起别名;

```xml
<select id="query"  resultType="User">
    select id,  name as username, password from user
</select>
```

第二种方式:利用mybatis中的resultMap标签解决

```xml
<select id="query"  resultMap="resultmap">
    select id,  name password from user
</select>

<resultMap id="resultmap" type="com.staticzz.entity.User">
    <id property="id" column="id"/>
    <result property="username" column="name"/>
    <result property="password" column="password"/>
</resultMap>
```

我们可以通过resultMap结果集映射实现

3.2.2 自动映射

如果我们数据库中的字段名只有部分与Java对象中的属性名不相同,那么我们只需要在结果映射中配置不相同的地方即可,其余相同的字段,属性,mybatis会帮助我们进行自动映射

```
<select id="query"  resultMap="resultmap">
   select id,  name , password from user
</select>

<resultMap id="resultmap" type="User">
   <!--  &lt;!&ndash;会帮助我们自动映射&ndash;&gt;
     <id property="id" column="id"/>
     <result property="password" column="password"/>-->
   <result property="username" column="name"/
 </resultMap>
```

有三种自动映射等级：

- `NONE` - 禁用自动映射。仅对手动映射的属性进行映射。
- `PARTIAL` - 对除在内部定义了嵌套结果映射（也就是连接的属性）以外的属性进行映射
- `FULL` - 自动映射所有属性。

默认使用PARTAL等级

无论设置的自动映射等级是哪种，你都可以通过在结果映射上设置 `autoMapping` 属性来为指定的结果映射设置启用/禁用自动映射。

```
<resultMap id="userResultMap" type="User" autoMapping="false">
   <result property="password" column="hashed_password"/>
</resultMap>
```


3.2.3 多对一处理方式

给一个场景来理解这个关系

一位老师可能教学好多学生,但对于这些学生来说,只有一位老师,这就是典型的一对多,多对一

在mybatis中

多对一用关联来表示,

一对多用集合来表示,

从学生角度看,多对一,多个学生关联一个老师

从老师角度看,一对多,一个老师有多个学生,就是一个集合

学生(Student) 有以下字段

id,name,gender,tid

老师(Teacher)有以下字段

id,name

现在我们需要把学生与老师关联起来,就需要往里面传入一个老师的对象

建表语句如下:

```
学生建表语句
CREATE TABLE `devdata`.`Untitled`  (
   `id` int(0) NOT NULL AUTO_INCREMENT,
   `name` varchar(255) NULL,
   `gender` varchar(255) NULL,
   `tid` int(0) NULL,
   PRIMARY KEY (`id`)
);
----------------------------------------
老师建表语句
CREATE TABLE `devdata`.`Untitled`  (
   `id` int(0) NOT NULL,
   `name` varchar(255) NULL,
   PRIMARY KEY (`id`)
);
```

现在有一个需求,我们想要知道所有学生对应的老师信息如何来做

```
##sql语句如下

select * from student s, teacher t where s.tid=t.id;
```

实体类如下:

```
public class Student {
   private Integer id;
   private String name;
   private String gender;

   //多个学生关联一个老师
   private Teacher teacher;

}
```



```
public class Teacher {

   private Integer id;
   private String name;
}
```

Dao层接口:

```
public interface StudentDao {

   public List<Student> students();

   public List<Student> students2();

   public Teacher teacher();

}
```

在mapper映射文件中我们可以通过以下方式查询到

方式一:按照查询进行嵌套处理<类似于mysql中的子查询>

```
<!--查询所有学生,要求携带老师信息,所有结果对象不能为单一的-->
<select id="students" resultMap="s_t">
   select * from student s,teacher t where s.tid=t.id;
</select>

<!--通过resultMap惊醒结果映射-->
<resultMap id="s_t" type="Student" >

   <!--通过association,关联老师这个复杂类型,把学生表中的tid作为参数,select携带出去,再下一次查询时使用-->
   <association property="teacher"  column="tid"  javaType="Teacher"
 select="teacher"/>
</resultMap>

<!--通过select携带的参数查询老师-->
<select id="teacher" resultType="Teacher">
   select * from teacher where id=#{tid}
</select>
```

方式二:按照结果进行嵌套处理<类似于mysql中的联表查询>

```
<!--按照结果进行嵌套处理-->
<select id="students2" resultMap="s_t_1">
   select s.id sid,s.name sname,s.gender sgender,t.id tid,t.name tname from
student s,teacher t where        s.tid=t.id;
</select>

<resultMap id="s_t_1" type="Student">
   <result property="id" column="sid"/>
   <result property="name" column="sname"/>
   <result property="gender" column="sgender"/>
   <association property="teacher" javaType="Teacher">
      <result property="id" column="tid"/>
      <result property="name" column="tname"/>
   </association>
</resultMap>
```


测试类如下

```
/**
 * 根据查询嵌套测试
 * @throws IOException
 */
@org.junit.Test
public void students() throws IOException {
    MybatisUtils mybatisUtils = new MybatisUtils();

    SqlSession sqlSession = mybatisUtils.sqlSession();


    StudentDao mapper = sqlSession.getMapper(StudentDao.class);
```

```
    List<Student> students = mapper.students();

    System.out.println(students);

    sqlSession.close();
}


/**
 * 根据结果嵌套测试
 * @throws IOException
 */
@org.junit.Test
public void students1() throws IOException {
    MybatisUtils mybatisUtils = new MybatisUtils();

    SqlSession sqlSession = mybatisUtils.sqlSession();


    StudentDao mapper = sqlSession.getMapper(StudentDao.class);

    List<Student> students = mapper.students2();

    System.out.println(students);

    sqlSession.close();
}
```


3.2.4 一对多处理方式

给一个场景来理解这个关系

一位老师可能教学好多学生,但对于这些学生来说,只有一位老师,这就是典型的一对多,多对一

在mybatis中

多对一用关联来表示,

一对多用集合来表示,

从学生角度看,多对一,多个学生关联一个老师

从老师角度看,一对多,一个老师有多个学生,就是一个集合

建表语句如下:

```
学生建表语句
CREATE TABLE `devdata`.`Untitled`  (
   `id` int(0) NOT NULL AUTO_INCREMENT,
   `name` varchar(255) NULL,
   `gender` varchar(255) NULL,
   `tid` int(0) NULL,
   PRIMARY KEY (`id`)
);
----------------------------------------
老师建表语句
CREATE TABLE `devdata`.`Untitled`  (
   `id` int(0) NOT NULL,
```

```
    `name` varchar(255) NULL,
    PRIMARY KEY (`id`)
);
```

现在有一个需求,我们想要获取指定老师对应的所有学生信息如何来做

```
##sql语句如下

select t.name tname,t.id tid,s.id sid,s.name sname

from teacher t,student s where t.id=s.tid;
```

实体类如下:

```
public class Student {
    private Integer id;
    private String name;
    private String gender;
    private Integer tid;
}
```

```
public class Teacher {
    private Integer id;
    private String name;

    //一个老师有多个学生
    private List<Student> students;
}
```


Dao层接口:

```
public interface TeacherDao {

    public List<Teacher> teachers(@Param("id") int id);

}
```

在mapper映射文件中我们可以通过以下方式查询到

按结果嵌套查询

```
<!---按照结果嵌套查询-->
<select id="teachers" resultMap="t_s">
  select t.name tname,t.id tid,s.id sid,s.name sname,
  s.tid stid,s.gender sgender

  from teacher t,student s where t.id=s.tid and t.id=#{id};
</select>
<resultMap id="t_s" type="Teacher">
    <result property="id" column="tid"/>
    <result property="name" column="tname"/>

    <!--老师有多个学生,用集合处理
    JavaType为指定属性的类型
    集合中的泛型信息使用ofType获取
```

```
    -->
    <collection property="students" ofType="Student" >
        <result property="id" column="sid"/>
        <result property="name" column="sname"/>
        <result property="gender" column="sgender"/>
        <result property="tid" column="stid"/>
    </collection>
</resultMap>
```

按子查询的方式

```
<!--按子查询的方式-->
<select id="teachers2" resultMap="t_s_1">
    select  * from teacher where id=#{id}
</select>

<resultMap id="t_s_1" type="Teacher">
    <id property="id" column="id"/>
    <collection property="students" ofType="Student" column="id"
select="student">
    </collection>
</resultMap>

<select id="student" resultType="Student">
    select * from student where tid=#{id}
</select>
```

测试类如下:


```
public class TeacherTest {

    /**
     * 测试根据结果嵌套查询
     * @throws IOException
     */
    @Test
    public  void  test() throws IOException {
        MybatisUtils mybatisUtils = new MybatisUtils();

        SqlSession sqlSession = mybatisUtils.sqlSession();

        TeacherDao mapper = sqlSession.getMapper(TeacherDao.class);

        for (Teacher teacher : mapper.teachers(2)) {
            System.out.println(teacher);
        }

        sqlSession.close();

    }

    /**
     * 测试根据查询嵌套查询
     * @throws IOException
     */
    @Test
```

```
    public  void  test1() throws IOException {
        MybatisUtils mybatisUtils = new MybatisUtils();

        SqlSession sqlSession = mybatisUtils.sqlSession();

        TeacherDao mapper = sqlSession.getMapper(TeacherDao.class);

        for (Teacher teacher : mapper.teachers2(2)) {
            System.out.println(teacher);
        }

        sqlSession.close();

    }
```

总结:

- 关联---多对一　association
- 集合 一对多　　collection
- javaType用来指定Java实体类中属性的类型
- ofType用来指定映射到List或者集合中的类型,泛型中的约束类型!

注意点:

- 保证代码的规范与可读性
- 多对一与一对多尽量保证代码的属性与字段名简洁具有可读性

第四章 动态SQL

4.1 简介

动态 SQL 是 MyBatis 的强大特性之一。如果你使用过 JDBC 或其它类似的框架，你应该能理解根据不同条件拼接 SQL 语句有多痛苦，例如拼接时要确保不能忘记添加必要的空格，还要注意去掉列表最后一个列名的逗号。利用动态 SQL，可以彻底摆脱这种痛苦。

简单一句话,就是可以根据不同的条件自动拼接不同的sql语句,再也不用我们在代码中手动判断进行拼接

4.2 常用标签

- if
- choose (when, otherwise)
- trim (where, set)
- foreach

if标签讲解

使用动态 SQL 最常见情景是根据条件包含 where 子句的一部分。比如：

```
<select id="findActiveBlogWithTitleLike"
     resultType="Blog">
  SELECT * FROM BLOG
  WHERE state = ‘ACTIVE’
  <if test="title != null">
    AND title like #{title}
  </if>
</select>
```

这条语句提供了可选的查找文本功能。如果不传入 “title”，那么所有处于 “ACTIVE” 状态的 BLOG 都会返回；如果传入了 “title” 参数，那么就会对 “title” 一列进行模糊查找并返回对应的 BLOG 结果（细心的读者可能会发现，“title” 的参数值需要包含查找掩码或通配符字符）。

如果希望通过 “title” 和 “author” 两个参数进行可选搜索该怎么办呢？首先，我想先将语句名称修改成更名副其实的名称；接下来，只需要加入另一个条件即可。

```
<select id="findActiveBlogLike"
     resultType="Blog">
  SELECT * FROM BLOG WHERE state = ‘ACTIVE’
  <if test="title != null">
    AND title like #{title}
  </if>
  <if test="author != null and author.name != null">
    AND author_name like #{author.name}
  </if>
</select>
```


choose、when、otherwise

有时候，我们不想使用所有的条件，而只是想从多个条件中选择一个使用。针对这种情况，MyBatis 提供了 choose 元素，它有点像 Java 中的 switch 语句。

还是上面的例子，但是策略变为：传入了 “title” 就按 “title” 查找，传入了 “author” 就按 “author” 查找的情形。若两者都没有传入，就返回标记为 featured 的 BLOG（这可能是管理员认为，与其返回大量的无意义随机 Blog，还不如返回一些由管理员精选的 Blog）。

```
<select id="findActiveBlogLike"
     resultType="Blog">
  SELECT * FROM BLOG WHERE state = ‘ACTIVE’
  <choose>
    <when test="title != null">
      AND title like #{title}
    </when>
    <when test="author != null and author.name != null">
      AND author_name like #{author.name}
    </when>
    <otherwise>
      AND featured = 1
    </otherwise>
  </choose>
</select>
```

trim、where、set

前面几个例子已经方便地解决了一个臭名昭著的动态 SQL 问题。现在回到之前的 “if” 示例，这次我们将 “state = ‘ACTIVE’” 设置成动态条件，看看会发生什么。

```
<select id="findActiveBlogLike"
     resultType="Blog">
  SELECT * FROM BLOG
  WHERE
  <if test="state != null">
    state = #{state}
  </if>
  <if test="title != null">
    AND title like #{title}
  </if>
  <if test="author != null and author.name != null">
    AND author_name like #{author.name}
  </if>
</select>
```

如果没有匹配的条件会怎么样？最终这条 SQL 会变成这样：

```
SELECT * FROM BLOG
WHERE
```

这会导致查询失败。如果匹配的只是第二个条件又会怎样？这条 SQL 会是这样:

```
SELECT * FROM BLOG
WHERE
AND title like ‘someTitle’
```

这个查询也会失败。这个问题不能简单地用条件元素来解决。这个问题是如此的难以解决，以至于解决过的人不会再想碰到这种问题。

MyBatis 有一个简单且适合大多数场景的解决办法。而在其他场景中，可以对其进行自定义以符合需求。而这，只需要一处简单的改动：

```
<select id="findActiveBlogLike"
     resultType="Blog">
  SELECT * FROM BLOG
  <where>
    <if test="state != null">
         state = #{state}
    </if>
    <if test="title != null">
        AND title like #{title}
    </if>
    <if test="author != null and author.name != null">
        AND author_name like #{author.name}
    </if>
  </where>
</select>
```

where 元素只会在子元素返回任何内容的情况下才插入 “WHERE” 子句。而且，若子句的开头为 “AND” 或 “OR”，*where* 元素也会将它们去除。

如果 *where* 元素与你期望的不太一样，你也可以通过自定义 trim 元素来定制 *where* 元素的功能。比如，和 *where* 元素等价的自定义 trim 元素为：

```
<trim prefix="WHERE" prefixOverrides="AND |OR ">
  ...
</trim>
```

prefixOverrides 属性会忽略通过管道符分隔的文本序列（注意此例中的空格是必要的）。上述例子会移除所有 *prefixOverrides* 属性中指定的内容，并且插入 *prefix* 属性中指定的内容。

用于动态更新语句的类似解决方案叫做 *set*。*set* 元素可以用于动态包含需要更新的列，忽略其它不更新的列。比如：

```
<update id="updateAuthorIfNecessary">
  update Author
    <set>
      <if test="username != null">username=#{username},</if>
      <if test="password != null">password=#{password},</if>
      <if test="email != null">email=#{email},</if>
      <if test="bio != null">bio=#{bio}</if>
    </set>
  where id=#{id}
</update>
```

这个例子中，*set* 元素会动态地在行首插入 SET 关键字，并会删掉额外的逗号（这些逗号是在使用条件语句给列赋值时引入的）。

来看看与 *set* 元素等价的自定义 *trim* 元素吧：

```
<trim prefix="SET" suffixOverrides=",">
  ...
</trim>
```


注意，我们覆盖了后缀值设置，并且自定义了前缀值。

foreach标签讲解

动态 SQL 的另一个常见使用场景是对集合进行遍历（尤其是在构建 IN 条件语句的时候）。比如：

```
<select id="selectPostIn" resultType="domain.blog.Post">
  SELECT *
  FROM POST P
  WHERE ID in
  <foreach item="item" index="index" collection="list"
      open="(" separator="," close=")">
        #{item}
  </foreach>
</select>
```

foreach 元素的功能非常强大，它允许你指定一个集合，声明可以在元素体内使用的集合项（item）和索引（index）变量。它也允许你指定开头与结尾的字符串以及集合项迭代之间的分隔符。这个元素也不会错误地添加多余的分隔符，看它多智能！

提示 你可以将任何可迭代对象（如 List、Set 等）、Map 对象或者数组对象作为集合参数传递给 *foreach*。当使用可迭代对象或者数组时，index 是当前迭代的序号，item 的值是本次迭代获取到的元素。当使用 Map 对象（或者 Map.Entry 对象的集合）时，index 是键，item 是值。

忽略部分查看官方网站..

第五章 日志功能

5.1 日志实现

Mybatis 通过使用内置的日志工厂提供日志功能。内置日志工厂将会把日志工作委托给下面的实现之一：

- SLF4J
- Apache Commons Logging
- Log4j 2
- Log4j
- JDK logging

MyBatis 内置日志工厂会基于运行时检测信息选择日志委托实现。它会（按上面罗列的顺序）使用第一个查找到的实现。当没有找到这些实现时，将会禁用日志功能。

不少应用服务器（如 Tomcat 和 WebShpere）的类路径中已经包含 Commons Logging。注意，在这种配置环境下，MyBatis 会把 Commons Logging 作为日志工具。这就意味着在诸如 WebSphere 的环境中，由于提供了 Commons Logging 的私有实现，你的 Log4j 配置将被忽略。这个时候你就会感觉很郁闷：看起来 MyBatis 将你的 Log4j 配置忽略掉了（其实是因为在这种配置环境下，MyBatis 使用了 Commons Logging 作为日志实现）。如果你的应用部署在一个类路径已经包含 Commons Logging 的环境中，而你又想使用其它日志实现，你可以通过在 MyBatis 配置文件 mybatis-config.xml 里面添加一项 setting 来选择其它日志实现。

```
<configuration>
  <settings>
    ...
    <setting name="logImpl" value="LOG4J"/>
    ...
  </settings>
</configuration>
```


5.2 日志配置

日志配置

你可以通过在包、映射类的全限定名、命名空间或全限定语句名上开启日志功能，来查看 MyBatis 的日志语句。

再次提醒，具体配置步骤取决于日志实现。接下来我们会以 Log4j 作为示范。配置日志功能非常简单：添加一个或多个配置文件（如 log4j.properties），有时还需要添加 jar 包（如 log4j.jar）。下面的例子将使用 Log4j 来配置完整的日志服务。一共两个步骤：

步骤 1：添加 Log4j 的 jar 包

由于我们使用的是 Log4j，我们要确保它的 jar 包可以被应用使用。为此，需要将 jar 包添加到应用的类路径中。Log4j 的 jar 包可以在上面的链接中下载。

对于 web 应用或企业级应用，你可以将 `log4j.jar` 添加到 `WEB-INF/lib` 目录下；对于独立应用，可以将它添加到 JVM 的 `-classpath` 启动参数中。

步骤 2：配置 Log4j

配置 Log4j 比较简单。假设你需要记录这个映射器的日志：

```
package org.mybatis.example;
public interface BlogMapper {
  @Select("SELECT * FROM blog WHERE id = #{id}")
  Blog selectBlog(int id);
}
```

在应用的类路径中创建一个名为 `log4j.properties` 的文件，文件的具体内容如下：

```
# 全局日志配置
log4j.rootLogger=ERROR, stdout
# MyBatis 日志配置
log4j.logger.org.mybatis.example.BlogMapper=TRACE
# 控制台输出
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```

5.3 日志输出

通过在log4j.properties配置文件中配置相关配置即可

打印映射器日志

为了实现更细粒度的日志输出，你也可以只打印特定语句的日志。以下配置将只打印语句 `selectBlog` 的日志：

```
log4j.logger.org.mybatis.example.BlogMapper.selectBlog=TRACE
```


或者，你也可以打印一组映射器的日志，只需要打开映射器所在的包的日志功能即可：

```
log4j.logger.org.mybatis.example=TRACE
```

打印SQL语句,并忽略返回的结果集

某些查询可能会返回庞大的结果集。这时，你可能只想查看 SQL 语句，而忽略返回的结果集。为此，SQL 语句将会在 DEBUG 日志级别下记录（JDK 日志则为 FINE）。返回的结果集则会在 TRACE 日志级别下记录（JDK 日志则为 FINER)。因此，只要将日志级别调整为 DEBUG 即可：

```
log4j.logger.org.mybatis.example=DEBUG
```

通过命名空间打印日志

但如果你要为下面的映射器 XML 文件打印日志，又该怎么办呢?

```
<?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="org.mybatis.example.BlogMapper">
  <select id="selectBlog" resultType="Blog">
    select * from Blog where id = #{id}
  </select>
</mapper>
```

这时，你可以通过打开命名空间的日志功能来对整个 XML 记录日志：

```
log4j.logger.org.mybatis.example.BlogMapper=TRACE
```

而要记录具体语句的日志，可以这样做：

```
log4j.logger.org.mybatis.example.BlogMapper.selectBlog=TRACE
```

你应该会发现，为映射器和 XML 文件打开日志功能的语句毫无差别。

第六章 Mybatis缓存

6.1 缓存简介

Mybatis缓存分为一级缓存与二级缓存,一级缓存是默认开启的,级别为SqlSession级别

- **一级缓存 也称为本地缓存:**
  - 与数据库同一次会话期间查询到的数据会放在本地缓存中
  - 如果需要数据直接从缓存中拿即可,没有必要再去数据库中查询,但是这个会话被关闭了,则缓存数据丢失
- **二级缓存 也成为全局缓存:**
  - 一级缓存的作用域太低了,所以产生了二级缓存,二级缓存是基于namespaces(命名空间的)
  - 不同的mapper(映射器)查出的缓存会放在自己的缓存中

6.2 一级缓存

MyBatis 内置了一个强大的事务性查询缓存机制，它可以非常方便地配置和定制。 为了使它更加强大而且易于配置，我们对 MyBatis 3 中的缓存实现进行了许多改进。

默认情况下，只启用了本地的会话缓存，它仅仅对一个会话中的数据进行缓存。

基本上就是这样。这个简单语句的效果如下:

- 映射语句文件中的所有 select 语句的结果将会被缓存。
- 映射语句文件中的所有 insert、update 和 delete 语句会刷新缓存。
- 缓存会使用最近最少使用算法（LRU, Least Recently Used）算法来清除不需要的缓存。
- 缓存不会定时进行刷新（也就是说，没有刷新间隔）。
- 缓存会保存列表或对象（无论查询方法返回哪种）的 1024 个引用。
- 缓存会被视为读/写缓存，这意味着获取到的对象并不是共享的，可以安全地被调用者修改，而不干扰其他调用者或线程所做的潜在修改。

6.3 二级缓存

如果想要使用二级缓存,我们只需要在映射文件中加一个标签即可

缓存只作用于 cache 标签所在的映射文件中的语句

但是只是在映射文件中加入这个标签是不会起到作用的!

我们还需要在mybatis核心配置文件中开启缓存

```
<settings>
    <!--配置日志实现-->
    <setting name="logImpl" value="LOG4J"/>
    <!--开启全局缓存-->
    <setting name="cacheEnabled" value="true"/>
</settings>
```

提示 缓存只作用于 cache 标签所在的映射文件中的语句。如果你混合使用 Java API 和 XML 映射文件，在共用接口中的语句将不会被默认缓存。你需要使用 @CacheNamespaceRef 注解指定缓存作用域。

这些属性可以通过 cache 元素的属性来修改。比如：

```
<cache
  eviction="FIFO"
  flushInterval="60000"
  size="512"
  readOnly="true"/>
```

这个更高级的配置创建了一个 FIFO 缓存，每隔 60 秒刷新，最多可以存储结果对象或列表的 512 个引用，而且返回的对象被认为是只读的，因此对它们进行修改可能会在不同线程中的调用者产生冲突。

可用的清除策略有：

- `LRU` – 最近最少使用：移除最长时间不被使用的对象。
- `FIFO` – 先进先出：按对象进入缓存的顺序来移除它们。
- `SOFT` – 软引用：基于垃圾回收器状态和软引用规则移除对象。
- `WEAK` – 弱引用：更积极地基于垃圾收集器状态和弱引用规则移除对象。

默认的清除策略是 LRU。

flushInterval（刷新间隔）属性可以被设置为任意的正整数，设置的值应该是一个以毫秒为单位的合理时间量。 默认情况是不设置，也就是没有刷新间隔，缓存仅仅会在调用语句时刷新。

size（引用数目）属性可以被设置为任意正整数，要注意欲缓存对象的大小和运行环境中可用的内存资源。默认值是 1024。

readOnly（只读）属性可以被设置为 true 或 false。只读的缓存会给所有调用者返回缓存对象的相同实例。 因此这些对象不能被修改。这就提供了可观的性能提升。而可读写的缓存会（通过序列化）返回缓存对象的拷贝。 速度上会慢一些，但是更安全，因此默认值是 false。

提示 二级缓存是事务性的。这意味着，当 SqlSession 完成并提交时，或是完成并回滚，但没有执行 flushCache=true 的 insert/delete/update 语句时，缓存会获得更新。

总结

- 只要开启了二级缓存,在同一个Mapper下就有效
- 所有的数据都会先放在一级缓存中
- 只有当会话提交,或者关闭的时候,才会提交到二级缓存中

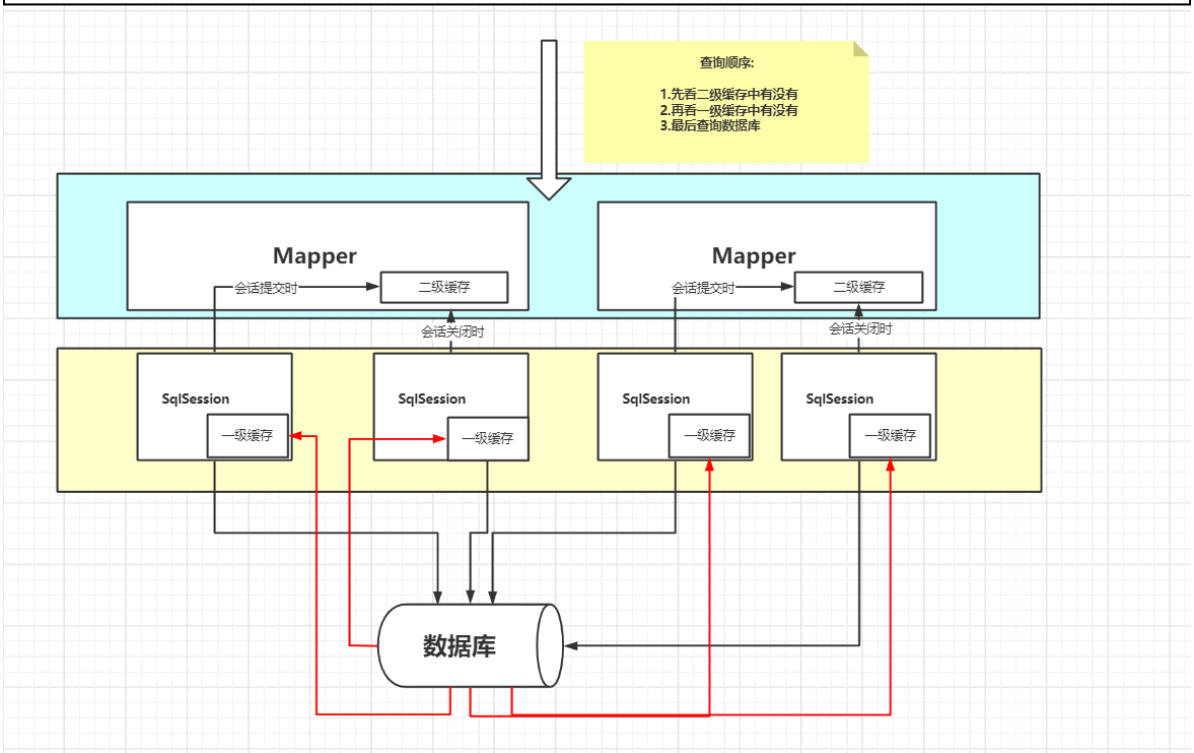
6.4 缓存原理

当第一次查询数据时,先去数据库中查找,查到之后把数据放在SqlSession会话中

如果会话被提交或者被关闭时,前提是二级缓存为开启状态,所有数据都会存放到二级缓存中

socra ， 2020/11/2 18:34:40

6.4 mybatis缓存原理



当用户发起请求时,先去二级缓存中查询,若没有再去一级缓存中查询,最后才是数据库中查询