每个基于 MyBatis 的应用都是以一个 SqlSessionFactory 的实例为核心的。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配置的 Configuration 实例来构建出 SqlSessionFactory 实例。
如以下例子用xml来创建

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

Mybatis的一般使用步骤

添加 mybatis依赖和数据库驱动

```
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.5.9</version>
</dependency>
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.25</version>
</dependency>
```

创建SqlSessionFactory类，可通过读取xml配置文件或者通过configuration类。
若使用xml创建，则需要编写xml配置文件，以及映射器文件
若用configuration类创建，不需要写配置文件，但是要写映射器类，例如下面的BlogMapper（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））

```
使用xml配置文件创建
String mybatisConfig = new String("mybatis-config.xml");
```

```
InputStream configStream = Resources.getResourceAsStream(mybatisConfig);
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(configStream);

//使用configuration类创建
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);

/、对于已经用XML配置文件创建了的SqlSessionFactory，可以如下添加映射器类
factory.getConfiguration().addMapper(StudentMapper.class);
```

编写XML配置文件以及映射器配置文件（XML配置文件方式创建SqlsessionFactory）
或编写映射器类

mybatis-cong.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>
        <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://127.0.0.1:3306/xxxx"/>
        <property name="username" value="xxx"/>
        <property name="password" value="xxx"/>
    </properties>
    <environments default="development">
        <environment id="development">
            <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>
    <mappers>
        <mapper resource="./mappers/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="hth.studentMapper">
    <select id="selectStudent" resultType="com.hth.entity.Student">
        select * from student where sage = #{age}
    </select>
</mapper>
```

映射器类 StudentMapper

```
public interface StudentMapper {
    @Select("select * from student where sage = #{age}")
    public List<Student> selectByAge(int age);
}
```

获取Sqlsession实例与数据库交互

注意Sqlsession的作用域，SqlsessionFactory应该是全局范围的，伴随着整个应用的生命周期，而Sqlsession最好的作用域应该与一个请求的作用域类似，在返回响应或结果后关闭。

配置

mybatis配置文件结构如下所示

- configuration（配置）
  - properties（属性）
  - settings（设置）
  - typeAliases（类型别名）
  - typeHandlers（类型处理器）
  - objectFactory（对象工厂）
  - plugins（插件）
  - environments（环境配置）
    - environment（环境变量）
      - transactionManager（事务管理器）
      - dataSource（数据源）
  - databaseIdProvider（数据库厂商标识）
  - mappers（映射器）

属性 properties

可以在典型的配置文件xxx.properties中配置属性，也可在<properties>标签下配置属性

```
//resource指定外部配置文件路径
<properties resource="org/mybatis/example/config.properties">
  <property name="username" value="dev_user"/>
  <property name="password" value="F2Fa3!33TYyg"/>
</properties>
```

也可在SqlsessionFactorybuilder中添加属性配置

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

以上三种方式的优先级如下：

1.SqlsessionFactorybuilder

2.resource指定外部配置文件路径

3.<properties>标签

设置

略

类型别名 （TypeAliases）

为Java类型设置一个简短的缩写名字，降低全限定类名的复杂

```
<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>
```

指定一个包名，mybatis会在该包下面搜索需要的类，若没有用@Alias 注解来指定类名，那么被搜索到的类的名字为默认开头字母小写

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

类型处理器

略

对象工厂

Mybatis与数据库交互返回结果集时，会默认使用一个对象工厂来创建结果集的实例，其仅仅是实例化结果集目标类，通过无参构造方法或者有参构造方法

可通过使用自定义的对象工厂覆盖原本的默认工厂

```
// ExampleObjectFactory.java
public class ExampleObjectFactory extends DefaultObjectFactory {
  public Object create(Class type) {
    return super.create(type);
```

```
  }
  public Object create(Class type, List<Class> constructorArgTypes, List<Object> constructorArgs) {
    return super.create(type, constructorArgTypes, constructorArgs);
  }
  public void setProperties(Properties properties) {
    super.setProperties(properties);
  }
  public <T> boolean isCollection(Class<T> type) {
    return Collection.class.isAssignableFrom(type);
  }}
```

在配置文件中注册对象工厂

```
<!-- mybatis-config.xml -->
<objectFactory type="org.mybatis.example.ExampleObjectFactory">
  <property name="someProperty" value="100"/>
</objectFactory>
```

插件 Plugins

在执行映射语句的某一个过程中，可以对其进行拦截（类似AOP），插件就是用来拦截的，其能够拦截的为以下四个：

- Executor (update, query, flushStatements, commit, rollback, getTransaction, close, isClosed)
- ParameterHandler (getParameterObject, setParameters)
- ResultSetHandler (handleResultSets, handleOutputParameters)
- StatementHandler (prepare, parameterize, batch, update, query)

使用插件需要实现Interceptor接口，并指定想要拦截的方法的签名

```
@Intercepts({@Signature(
        type= Executor.class,
        method = "update",
        args = {MappedStatement.class,Object.class})})
public class UpdatePlugin implements Interceptor {
    @Override
    public Object intercept(Invocation invocation) throws Throwable {
        System.out.println("开始执行update方法咯");
        Object res = invocation.proceed();
```

```
        System.out.println("结束了");
        return res;
    }

    @Override
    public Object plugin(Object target) {
        return Interceptor.super.plugin(target);
    }

    @Override
    public void setProperties(Properties properties) {
        Interceptor.super.setProperties(properties);
    }
}
```

xml写入配置

```
<plugins>
    <plugin interceptor="com.hth.UpdatePlugin">
    </plugin>
</plugins>
```

环境配置 environments

利用环境配置可以将sql映射应用于多种数据库，但是每个SqlsessionFactory对应一个数据库连接，若要连接多个数据库，则要创建多个SqlsessionFactory

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>
```

如下两种创建SqlsessionFactory的方式是没有环境参数的重载方法，则会加载默认环境配置

```
//未指定环境参数。加载默认环境配置
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader);
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader, properties);

//指定了环境配置
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader, environment);
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader, environment, properties);
```

事务管理器

JDBC 和 MANAGED

TIPS: 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。

JDBC: 使用JDBC的事务提交和回滚机制，依赖从数据源获得的连接来管理事务

MANAGED： 让其所在的应用上下文管理事务的整个生命周期，这个越等于什么都没做

数据源 datasource

共有三种数据源类型 POOLED UNPOOLED JNDIの

UNPOOLED： 每次请求都会打开连接关闭连接，顾名思义，没有数据库连接池

POOLED： 即有连接池

JNDI： 略

可通过实现接口 DatasourceFactory 自定义数据源

映射器 mappers

用基于类路径的配置文件引用

```
<!-- 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>
```

使用映射器类的全限定类名引用

```
<!-- 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>
```

将包内的所有映射器接口实现类加载为映射器

```
<!-- 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>
```

XML映射器

xml映射器文件有如下的元素，严格按照顺序进行定义

- 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。
- 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。

select

select语句很常用，一个最常用且简单的select语句如下所示

```
<select id="selectPerson" parameterType="int" resultType="hashmap">
```

```
  SELECT * FROM PERSON WHERE ID = #{id}
</select>

//该语句返回一个HashMap，键是列名，值为对应列的值
```

select元素的属性如下图所示

Select 元素的属性

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的类全限定名或别名。这个属性是可选的，因为 MyBatis 可以通过类型处理器（TypeHandler）推断出具体传入语句的参数，默认值为未设置（unset）。
~~parameterMap~~	~~用于引用外部 parameterMap 的属性，目前已被废弃。请使用行内参数映射和 parameterType 属性。~~
resultType	期望从这条语句中返回结果的类全限定名或别名。 注意，如果返回的是集合，那应该设置为集合包含的类型，而不是集合本身的类型。 resultType 和 resultMap 之间只能同时使用一个。
resultMap	对外部 resultMap 的命名引用。结果映射是 MyBatis 最强大的特性，如果你对其理解透彻，许多复杂的映射问题都能迎刃而解。 resultType 和 resultMap 之间只能同时使用一个。
flushCache	将其设置为 true 后，只要语句被调用，都会导致本地缓存和二级缓存被清空，默认值：false。
useCache	将其设置为 true 后，将会导致本条语句的结果被二级缓存缓存起来，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。默认值为未设置（unset）（依赖数据库驱动）。
fetchSize	这是一个给驱动的建议值，尝试让驱动程序每次批量返回的结果行数等于这个设置值。 默认值为未设置（unset）（依赖驱动）。
statementType	可选 STATEMENT，PREPARED 或 CALLABLE。这会让 MyBatis 分别使用 Statement，PreparedStatement 或 CallableStatement，默认值：PREPARED。
resultSetType	FORWARD_ONLY，SCROLL_SENSITIVE, SCROLL_INSENSITIVE 或 DEFAULT（等价于 unset） 中的一个，默认值为 unset （依赖数据库驱动）。
databaseId	如果配置了数据库厂商标识（databaseIdProvider），MyBatis 会加载所有不带 databaseId 或匹配当前 databaseId 的语句；如果带和不带的语句都有，则不带的会被忽略。
resultOrdered	这个设置仅针对嵌套结果 select 语句：如果为 true，将会假设包含了嵌套结果集或是分组，当返回一个主结果行时，就不会产生对前面结果集的引用。 这就使得在获取嵌套结果集的时候不至于内存不够用。默认值：false。
resultSets	这个设置仅适用于多结果集的情况。它将列出语句执行后返回的结果集并赋予每个结果集一个名称，多个名称之间以逗号分隔。

到statementTypes之前的都比较有用

insert，delete，update

这三条数据库更新语句都比较类似

```
<insert
  id="insertAuthor"
  parameterType="domain.blog.Author"
  flushCache="true"
  statementType="PREPARED"
  keyProperty=""
  keyColumn=""
  useGeneratedKeys=""
  timeout="20">

<update
  id="updateAuthor"
  parameterType="domain.blog.Author"
  flushCache="true"
  statementType="PREPARED"
```

```
  timeout="20">

<delete
  id="deleteAuthor"
  parameterType="domain.blog.Author"
  flushCache="true"
  statementType="PREPARED"
  timeout="20">
```

相比select，额外的子属性

useGeneratedKeys	（仅适用于 insert 和 update）这会令 MyBatis 使用 JDBC 的 getGeneratedKeys 方法来取出由数据库内部生成的主键（比如：像 MySQL 和 SQL Server 这样的关系型数据库管理系统的自动递增字段），默认值：false。
keyProperty	（仅适用于 insert 和 update）指定能够唯一识别对象的属性，MyBatis 会使用 getGeneratedKeys 的返回值或 insert 语句的 selectKey 子元素设置它的值，默认值：未设置（unset）。如果生成列不止一个，可以用逗号分隔多个属性名称。
keyColumn	（仅适用于 insert 和 update）设置生成键值在表中的列名，在某些数据库（像 PostgreSQL）中，当主键列不是表中的第一列的时候，是必须设置的。如果生成列不止一个，可以用逗号分隔多个属性名称。

对于insert语句

如果连接的数据库支持自动生成主键，那么可以用你可以设置 useGeneratedKeys=”true”，然后再把 keyProperty 设置为目标属性就 OK 了

```
<insert id="insertAuthor" useGeneratedKeys="true"
    keyProperty="id">
  insert into Author (username,password,email,bio)
  values (#{username},#{password},#{email},#{bio})
</insert>
```

如果支持多行插入，可以传入一个数组或者集合

```
insert id="insertAuthor" useGeneratedKeys="true"
    keyProperty="id">
  insert into Author (username, password, email, bio) values
  <foreach item="item" collection="list" separator=",">
    (#{item.username}, #{item.password}, #{item.email}, #{item.bio})
  </foreach>
</insert>
```

sql

此元素用来定义可被重用的sql片段,也可在sql属性内使用

比如

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


<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>
//其中alias参数被分别指定为t1 , t2
```

参数

原始类型或简单数据类型可以直接用他们的值作为参数

如果是对象，则会查找对象之中相应的字段赋值

```
<insert id="insertUser" parameterType="User">
  insert into users (id, username, password)
  values (#{id}, #{username}, #{password})
</insert>
```

也可通过如下方式指定特殊的参数

```
#{property,javaType=int,jdbcType=NUMERIC}
```

要进一步地自定义参数类型，可以使用自定义类型处理器

```
#{property,javaType=int,jdbcType=NUMERIC}
```

对于数值类型，还可以设置 numericScale 指定小数点后保留的位数

```
#{height,javaType=double,jdbcType=NUMERIC,numericScale=2}
```

字符串替换

默认情况下，当使用 #{} 语法时，会默认使用preparedStatement参数占位符，如果你想插入不转义的参数占位符，可以如下

${}语法

```
ORDER BY ${columnName}
```

```
@Select("select * from user where id = #{id}")
User findById(@Param("id") long id);

@Select("select * from user where name = #{name}")
User findByName(@Param("name") String name);

@Select("select * from user where email = #{email}")
User findByEmail(@Param("email") String email);


//下面这样写一个方法就能够对某一列进行查询
@Select("select * from user where ${column} = #{value}")
User findByColumn(@Param("column") String column, @Param("value") String value);
```

结果映射

结果映射主要指的是resultMap 属性

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

如上mybatis会默认使用resultmap，将获取的列名与User类的字段对应

```
<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>
```

若是像上面一样查询的字段与类字段不匹配，可以用sql语句本身来修改

如下是resultmap的使用

首先定义resultmap

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

随后在select中指定

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

ResultMap有两种常用的属性 id type

resultmap的子元素如下

- id
- result
- association
- constructor
- discriminator
- discriminator

常用的id和result

Id 和 Result 的属性

属性	描述
property	映射到列结果的字段或属性。如果 JavaBean 有这个名字的属性（property），会先使用该属性。否则 MyBatis 将会寻找给定名称的字段（field）。无论是哪一种情形，你都可以使用常见的点式分隔形式进行复杂属性导航。比如，你可以这样映射一些简单的东西：“username”，或者映射到一些复杂的东西上：“address.street.number”。
column	数据库中的列名，或者是列的别名。一般情况下，这和传递给 resultSet.getString(columnName) 方法的参数一样。
javaType	一个 Java 类的全限定名，或一个类型别名（关于内置的类型别名，可以参考上面的表格）。如果你映射到一个 JavaBean，MyBatis 通常可以推断类型。然而，如果你映射到的是 HashMap，那么你应该明确地指定 javaType 来保证行为与期望的相一致。
jdbcType	JDBC 类型，所支持的 JDBC 类型参见这个表格之后的“支持的 JDBC 类型”。只需要在可能执行插入、更新和删除的且允许空值的列上指定 JDBC 类型。这是 JDBC 的要求而非 MyBatis 的要求。如果你直接面向 JDBC 编程，你需要对可以为空值的列指定这个类型。
typeHandler	我们在前面讨论过默认的类型处理器。使用这个属性，你可以覆盖默认的类型处理器。这个属性值是一个类型处理器实现类的全限定名，或者是类型别名。

<constructor>元素可以指定通过构造方法来创造实例，而不是通过属性的setters，如下

User类有以下的构造方法

```
public class User {
   //...
   public User(Integer id, String username, int age) {
     //...
  }
//...
}

<constructor>
   <idArg column="id" javaType="int"/>
   <arg column="username" javaType="String"/>
   <arg column="age" javaType="_int"/>
</constructor>
//可通过name属性指定形参的名字
<constructor>
   <idArg column="id" javaType="int" name="id" />
   <arg column="age" javaType="_int" name="age" />
   <arg column="username" javaType="String" name="username" />
</constructor>
```

关联

关联（association）元素处理“有一个”类型的关系。 比如，在我们的示例中，一个博客有一个用户。

关联有两种方式处理：嵌套select查询 和 嵌套的结果映射

嵌套select查询

```
<select id="selectAuthor" resultType="Author">
  SELECT * FROM AUTHOR WHERE ID = #{id}
```

```
</select>

<resultMap id="blogResult" type="Blog">
  <association property="author" column="author_id" javaType="Author" select="selectAuthor"/>
</resultMap>
```

嵌套结果映射

示例如下

```
<resultMap id="blogResult" type="Blog">
  <id property="id" column="blog_id" />
  <result property="title" column="blog_title"/>
  <association property="author" column="blog_author_id" javaType="Author" resultMap="authorResult"/>
</resultMap>
//这种写法，authoreResult这个结果映射可以被重用，当然，也可以直接写在<association>里面
/*<association property="author" javaType="Author">
    <id property="id" column="author_id"/>
    <result property="username" column="author_username"/>
    <result property="password" column="author_password"/>
    <result property="email" column="author_email"/>
    <result property="bio" column="author_bio"/>
  </association>
****/

<resultMap id="authorResult" type="Author">
  <id property="id" column="author_id"/>
  <result property="username" column="author_username"/>
  <result property="password" column="author_password"/>
  <result property="email" column="author_email"/>
  <result property="bio" column="author_bio"/>
</resultMap>

<select id="selectBlog" resultMap="blogResult">
  select
    B.id            as blog_id,
    B.title         as blog_title,
```

```
    B.author_id     as blog_author_id,
    A.id            as author_id,
    A.username      as author_username,
    A.password      as author_password,
    A.email         as author_email,
    A.bio           as author_bio
  from Blog B left outer join Author A on B.author_id = A.id
  where B.id = #{id}
</select>
```

集合

与联合基本一样，只是要注意Oftype属性，他是为了区分存储javabean的字段名和查询出的结果集的类

```
private ArrayList<User> users;    //users是存储javabean的字段名    而oftype指定查询出的结果为User
```

鉴定器discriminator

略

自动映射

自动映射之前的自动匹配java类的字段并赋值，

自动映射可以和结果映射一起工作

缓存

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。 要启用全局的二级缓存，只需要在你的 SQL 映射文件中添加一行

```
<cache/>
```

这句代码的效果如下

- 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。

这些属性可以通过 cache 元素的属性来修改，例如

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

动态SQL

根据不同的条件生成不同的SQL

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

if

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

```
若传入的title不为null，则会添加title的模糊查询
```

choose

多个条件选一个使用，跟编程语言switch效果一样

```
<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>
```

where，set

按上面If的例子，若是state=active 这个条件也是可选的，那么如果state 和 title都没传入参数满足条件，则语句可能会变成这样

```
 SELECT * FROM BLOG WHERE
```

为了解决这个问题，可以如下

```
<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>
```

上面的where仅会在它的子元素任何一个有值的情况下才会生成，若都没有，语句会变成

```
SELECT * FROM BLOG
```

用于update语句的动态更新功能为set，set可以生成包含需要更新的列，忽略不需要更新的列

```
<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>
```

foreach

没啥好说的，用于遍历，比如用到in的情况下

```
<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>
```

JAVA API

没看懂罗里吧嗦讲了什么，认为有用如下

- select 方法的三个高级版本，它们允许你限制返回行数的范围，或是提供自定义结果处理逻辑，通常在数据集非常庞大的情形下使用。

```
<E> List<E> selectList (String statement, Object parameter, RowBounds rowBounds)
<T> Cursor<T> selectCursor(String statement, Object parameter, RowBounds rowBounds)
<K,V> Map<K,V> selectMap(String statement, Object parameter, String mapKey, RowBounds rowbounds)
void select (String statement, Object parameter, ResultHandler<T> handler)
void select (String statement, Object parameter, RowBounds rowBounds, ResultHandler<T> handler)


//RowBounds 参数会告诉 MyBatis 略过指定数量的记录，并限制返回结果的数量
int offset = 100;
int limit = 25;
RowBounds rowBounds = new RowBounds(offset, limit);
```

- Mybatis 使用到了两种缓存：本地缓存（local cache）和二级缓存（second level cache）。

每有一个新的Sqlsession创建，就会有一个与之对应的本地缓存创建，任何通过此session的查询结果，都会被保存在被你缓存中。
默认情况下，缓存的生命周期与session相同。
对于某个对象，MyBatis 将返回在本地缓存中唯一对象的引用，对返回的对象进行修改的话，可能会影响本地缓存的对象，所以不要对返回的对象进行修改。

```
try(SqlSession s = factory.openSession()){
    var ss = s.selectList("hth.studentMapper.selectStudent",18);
    for(var item:ss){
        var student = (Student)item;
        student.sname="jiejie";
    }
    //相同的查询
```

```
    ss = s.selectList("hth.studentMapper.selectStudent",18);
    for(var item:ss){
        System.out.println(item);
    }
}

输出如下：
Student(sno=200215123, sname=jiejie, ssex=女, sage=18, sdept=MA)
Student(sno=200215125, sname=jiejie, ssex=女, sage=18, sdept=CS)
```

- 映射器注解

注解提供了一种简单且低成本的方式来实现简单的映射语句，太多了记不住
简单的增删改查用注解，复杂的直接写xml。