

Tips: mybatis在多线程环境下存在问题。

Mybatis是面向接口编程的，但为什么mapper接口没有定义实现类就可以调用：

简单来说有两步：

1. 配置文件解读：解析配置文件并将属性值加载到Configuration对象中。
2. 动态代理增强Mapper接口：通过MapperMethod类将接口方法与sql组合，底层还是通过ibatis执行sql的方法（即通过sqlSession）去操作数据库。

详细流程看binding模块的分析（这两篇博客很重要）：

1. <https://www.jianshu.com/p/83785a294f8e>
2. <https://blog.csdn.net/newbie0107/article/details/102769100>

1、ORM (Object Relational Mapping) 对象关系映射；

1、JNDI数据源是配置在tomcat中的数据源，部署在tomcat中的所有应用都可以共享此数据源。而在应用中配置的连接池只有此应用单独使用。

2、jdbc步骤

2、PreparedStatement和Statement的用法区别
(<https://www.jianshu.com/p/d73e83bb5d7d>)

3、mybatis父标签中配置那些字标签可以点进去看。

4、TypeHandlerRegistry类型转换器。

5、**typeHandlers**当需要转换枚举和一些特殊类型的时候需要用到。其他的mybatis已经做好了。

5、**SqlSessionFactoryBuilder**、**SqlSessionFactory**、**SqlSession**的线程安全以及作用域最佳实践 (<https://www.cnblogs.com/yulinfeng/p/6002379.html>)

5、**SqlSession**默认自动提交为false（构造方法点进去看），所以需要我们手动提交。

6、查询结果集映射建议使用自定义的**resultMap**，这样使数据库和bean解耦，便于维护。**id**和**result**都是映射单列值到一个属性或字段的简单数据类型。在自定义的**resultMap**中第一列通常是主键**id**，唯一不同是。**id**是作为唯一标识的，当和其他对象实例对比的时候，这个**id**很有用，尤其是应用到缓存和内嵌的结果映射。

result可以有多个，分别对应数据库中的各个字段和实体类中的属性。

```
<resultMap id="BaseResultMap" type="TUser">
    <id column="id" property="id" jdbcType="INTEGER" />
    <result column="user_name" property="userName" jdbcType="VARCHAR" />
    <result column="real_name" property="realName" jdbcType="VARCHAR" />
    <result column="sex" property="sex" jdbcType="TINYINT" />
    <result column="mobile" property="mobile" jdbcType="VARCHAR" />
```

6、注意**resultType**和**resultMap**不能同时使用，当我们需要查询总条数等时可以通过**resultType**将返回结果指定为int，当多表联查的时候我们可以定义一个POJO，并通过**resultMap**做映射。这俩各有用处，别忽略了。

7、三种传参方式为map、**@Param**、自定义javaBean。一般五个参数以下使用**@Param**，五个参数以上使用**javaBean**。在**mapper.xml**中配置**parameterType**时只能指定一种类型，但是业务需求一般都要传多个参数，**map**的可读性差，所以一般使用第二、第三种方式：

- map (不建议使用，可读性差)

```
List<TUser> selectByEmailAndSex1(Map<String, Object> param);
```

- **@Param** (xml中不需要指定 **parameterType**)

```
List<TUser> selectByEmailAndSex2(@Param("email")String email,@Param("sex")Byte sex);
```

- 自定义javaBean

```
List<TUser> selectByEmailAndSex3(EmailSexBean esb);
```

7、Mybatis提供了三种执行器SIMPLE、REUSE、BATCH，默认的执行器为SIMPLE，为什么不需要我们优化一下将它改成REUSE执行器（REUSE执行器会重用预处理语句【prepared statements】，执行效率比SIMPLE高）？因为虽然全局执行器默认为SIMPLE，但是<resultMap/>标签中的statementType="PREPARED"，所以实际执行sql的时候还是使用的PreparedStatement：

```
<!-- 配置默认的执行器。SIMPLE执行器没有什么特别之处。REUSE执行器重用预处理语句。BATCH执行器重用语句和批量更新 -->
<setting name="defaultExecutorType" value="SIMPLE" />
```

设置参数	描述	有效值	默认值
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（prepared statements）；BATCH 执行器将重用语句并执行批量更新。	SIMPLE、REUSE、BATCH	SIMPLE

```
<select id="selectUserPosition1" resultMap="userAndPosition1" statementType="PREPARED">
    select
        ...
</select>
public static class Builder {
    private MappedStatement mappedStatement = new MappedStatement();
    ...
    public Builder(Configuration configuration, String id, SqlSource sqlSource, SqlCommandType sqlCommandType) {
        mappedStatement.configuration = configuration;
        mappedStatement.id = id;
        mappedStatement.sqlSource = sqlSource;
        mappedStatement.statementType = StatementType.PREPARED; // 如果不在<resultMap/>标签中显示配置，则构造 MappedStatement时会默认设置为PREPARED
        mappedStatement.parameterMap = new ParameterMap.Builder(configuration, id: "defaultParameterMap", type: null,
        ...
    }
}
```

7、为什么我们使用SIMPL而不使用REUSE： SIMPLE和RESUE的区别在于SIMPLE每次都会开启新的statement，而RESUE会将statement缓存起来。

(<https://www.jianshu.com/p/96ddaec4aea7>)

8、<resultMap/>的子标签<constructor/>的作用。但pojo没有无参构造时就需要通过<constructor/>作为结果入口（讲道理pojo只有无参构造但不提供set方法时说明这个类废了，无法对属性赋值没啥用了），无论有参构造中有几个参数还是有没有提供set方法，只要配置了映射关系，mybatis都会通过反射去赋值，不调用set方法：

```
25
26  public TUser(Integer id, String userName) {
27      super();
28      this.id = id;
29      this.userName = userName;
30  }
```

```

<mapper namespace="com.enjoyLearning.mybatis.mapper.TUserMapper">
    <resultMap id="BaseResultMap" type="TUser">
        <constructor>
            <idArg column="id" javaType="int"/>
            <arg column="user_name" javaType="String"/>
        </constructor>
        -->
        <id column="id" property="id" jdbcType="INTEGER" />
        <result column="user_name" property="userName" jdbcType="VARCHAR" />
        <result column="real_name" property="realName" jdbcType="VARCHAR" />
    </resultMap>

```

8、resultMap可以继承，这样可以减少字段重复配置。当多表关联查询时如果两个表中有相同的字段可以通过在resultMap中配置别名的方式解决。

```

<resultMap id="userAndPosition1" extends="BaseResultMap" type="TUser">
    <association property="position" javaType="IPosition" columnPrefix="post_">
        <id column="id" property="id"/>
        <result column="name" property="postName"/>
        <result column="note" property="note"/>
    </association>
</resultMap>

```

9、useGeneratedKeys和keyProperty需要搭配使用，前者指是否自动自增，后者指定那个字段自增。会默认从当前插入过的最大值开始递增，加入插入id=100，当把id=100这条记录删除以后，下次插入值它会默认从101开始（LAST_INSERT_ID()函数），这个最大值不是记录在数据库中的，而是记录在数据库日志中的：

```

</delete>

<insert id="insert1" parameterType="TUser" useGeneratedKeys="true" keyProperty="id">
    insert into t_user (id, user_name, real_name,
    sex, mobile,
    email,
    note, position_id)
    values (#{id,jdbcType=INTEGER},
    #{userName,jdbcType=VARCHAR},
    #{realName,jdbcType=VARCHAR},
    #{sex,jdbcType=TINYINT}, #{mobile,jdbcType=VARCHAR},
    #{email,jdbcType=VARCHAR},

```

9、在insert、update等标签中配置useGeneratedKeys和keyProperty这两个属性则我们在代码中就会拿到自动生成的主键值：

```

<insert id="insert1" parameterType="TUser" useGeneratedKeys="true" keyProperty="id">
    insert into t_user (id, user_name, real_name, sex, mobile, email, note, position_id)
    values (#{id,jdbcType=INTEGER},#{userName,jdbcType=VARCHAR},#{realName,jdbcType=VARCHAR},#{sex,jdbcType=TINYINT},
    #{mobile,jdbcType=VARCHAR},#{email,jdbcType=VARCHAR},#{note,jdbcType=VARCHAR},#{position.id,jdbcType=INTEGER})
</insert>                                         生成自增主键

// 3. 获取对应mapper
TUserMapper mapper = sqlSession.getMapper(TUserMapper.class);
// 4. 执行查询语句并返回结果
TUser user1 = new TUser(); user1.setUserName("test1"); user1.setRealName("realname1");
user1.setEmail("myemail1");
mapper.insert1(user1);                           执行插入语句并提交事务以后就可以拿到id了
sqlSession.commit();                           会自动映射复制到实体类的id字段上边
System.out.println(user1.getId());
```

```

10、useGeneratedKeys是比较简易的配置方法，实际本质是通过LAST\_INSERT\_ID()函数获取的：

```
<insert id="insert2" parameterType="TUser">
 <selectKey keyProperty="id" order="AFTER" resultType="int">
 select LAST_INSERT_ID()
 </selectKey>
 insert into t_user (id, user_name, real_name,
 sex, mobile,
 email,
 . . .
```

11、order="before"和order="after":

4.selectKey用法的坑

SelectKey需要注意order属性，像MySQL一类支持自动增长类型的数据库中，order需要设置为after才会取到正确的值，像Oracle这样取序列的情况，需要设置为before。

11、<typeAliases/>标签用来定义别名，减少包名冗余

12、sql元素：用来定义可重用的SQL代码段，可以包含在其他语句中：

```
<sql id="Base_Column_List">
 id, user_name, real_name, sex, mobile, email, note,
 position_id
</sql>

<select id="selectByPrimaryKey" resultMap="BaseResultMap" parameterType="java.lang.Integer">
 select
 <include refid="Base_Column_List" />
 from t_user
 where id = #{id,jdbcType=INTEGER}
</select>
```

13、#{ }和\${ }的区别（会不会自动加单引号，各有用处）：

- \${ }会预编译，所谓预编译就是在sql中设置占位符（即?），并将传进来的参数值加上单引号。可以有效防止sql注入（所谓sql注入就是将sql代码添加到输入参数中，传递到sql服务器解析并执行的一种攻击手法），因为非法sql通过参数传进来以后会被加上单引号，则执行效果如下：

```
<select id="selectBySymbol" resultMap="BaseResultMap">
 select
 #{inCol} ← 使用#{ }时执行的效果如下图
 from ${tableName} a
 where a.sex = #{sex}
 order by ${orderStr}
</select>
```

此处被加上单引号，相当于一个列名而已

- \${}不会预编译，传入的值会直接拼接到sql中，也不会加单引号，所以有时候使用\${}取传入参数的时候需要我们需要手动加单引号，但是当我们sql中的列名、表名、排序的条件时动态变化的时候我们就需要用\${}取值：

```
<select id="selectBySymbol" resultMap="BaseResultMap">
 select
 ${inCol}
 from ${tableName} a
 where a.sex = '${sex}'
 order by ${orderStr}
</select>
```

14、注解方式配置sql，可读性差，可维护性差，不推荐使用。导入方式不同：

```
<!-- 映射文件，mapper的配置文件 -->
<mappers>
 <!--直接映射到相应的mapper文件 -->
 < mapper resource="sqlmapper/TUserMapper.xml" />
 <!--
 < mapper resource="sqlmapper/TJobHistoryMapper.xml" />
 -->
 < mapper class="com.enjoyLearning.mybatis.mapper.TJobHistoryAnnoMapper" />
</mappers>
```

## 15、动态sql使用if时where解决方案：

- 加where1=1, and或者or要写在语句前面

```

select
<include refid="Base_Column_List" />
from t_user a
where 1=1
<if test="email != null and email != ''">
 and a.email like CONCAT('%', #{email}, '%')
</if>
<if test="sex != null ">
 and a.sex = #{sex}
</if>

```

- 使用`<where>``</where>`标签，如果条件都不成立不会拼接where子句并不会去掉and或or

```

<include refid="Base_Column_List" />
from t_user a
<where>
 <if test="email != null and email != ''">
 and a.email like CONCAT('%', #{email}, '%')
 </if>
 <if test="sex != null ">
 and a.sex = #{sex}
 </if>
</where>

```

这个and会被去掉

16、动态sql使用if时`<set>``</set>`标签会去掉最后逗号防止sql报错。

16、mybatis中choose (when, otherwise)标签，有一个匹配成功就会跳出choose不会继续往下走，和sql中的case、when、end不是一回事

(<https://www.bbsmax.com/A/MAzAKgQM59/>)

16、mysql的case、when用法有两种（写法有所不同）：

- 简单Case函数（case后边为条件）

CASE [col\_name] WHEN [value1] THEN [result1]...ELSE [default] END col\_name为列名, value1为值

- Case搜索函数（case后边什么都没有，when后边是表达式）

CASE WHEN [expr] THEN [result1]...ELSE [default] END expr为表达式，例如col>2

## 17、trim标签([https://blog.csdn.net/wt\\_better/article/details/80992014](https://blog.csdn.net/wt_better/article/details/80992014))

- **prefix**: 在条件语句前需要加入的内容。
- **suffix**: 在条件语句后需要加入的内容。
- **prefixOverrides**: 覆盖/去掉前一个前缀。
- **suffixOverrides**: 覆盖/去掉后一个前缀。

## 18、foreach标签的属性:

- **item** 表示集合中每一个元素进行迭代时的别名。
- **index** 指定一个名字，用于表示在迭代过程中每次迭代到的位置。
- **open** 表示该语句以什么开始。
- **separator** 表示在每次进行迭代之间以什么符号作为分隔符。
- **close** 表示以什么结束。
- **collection**的值为传入的数组或者list的变量名。

```
<select id="nonAccrualToResponseMetel" parameterType="map" resultType="com.csii.bank.core.model.generate.Rtxninfo">
 select bankorgnbr,branchnbr,postdate,currencycd,mjaccttypcd,miaccttypcd,
 ifnull(SUM(case when
 <foreach collection="subList" item="item" separator="or" open="(" close=")"
 (rtxntypcd = #{item.rtxntypcd} and balcatcd = #{item.balcatcd} and baltypcd = #{item.baltypcd})
 </foreach>
 THEN -tranamt else tranamt end),0) tranamt
 from cb_gl_rtxninfo where
 <foreach collection="paramList" item="item" separator="or" open="(" close=")"
 (rtxntypcd = #{item.rtxntypcd} and balcatcd = #{item.balcatcd} and baltypcd = #{item.baltypcd})
 </foreach>
 and postdate = #{postDate}
 <![CDATA[
 and taxflag <> 'FLFT'
 GROUP BY bankorgnbr,branchnbr,postdate,currencycd,mjaccttypcd,miaccttypcd
 HAVING tranamt<>0
]]>
```

```
map2.put("baltypcd",Dict.BALTYP_OVAT);
paramList.add(map1);
paramList.add(map2);
subList.add(map1);
subList.add(map2);
paramMap.put("subList",subList);
paramMap.put("paramList",paramList);
paramMap.put("rtxntypList",rtxntypList);
paramMap.put("fundtypcd",Dict.FUNDTYP_CD_JE);
paramMap.put("postDate",postDate);
rtxList = rtxninfoExtendMapper.nonAccruedLoanRepayment(paramMap);
if(rtxList.size()>0){
```

此处通过map传参的方式不优雅，可以通过封装成javabean  
(需多级多级封装)，这样取参的方式和传map是一样的

第二种是将sublist、paramList、rtxntypList、fundtypcd、  
postdate通过@Param的方式传进去，别直接装到一个map  
中

## 19、数据库批量操作两种方式:

- 通过foreach动态拼装SQL语句（即values (),(); ）。

- 使用mybatis的BATCH类型的executor，相应的jdbc中的addbatch()。

```
// 2. 获取sqlSession
SqlSession sqlSession = sqlSessionFactory.openSession(ExecutorType.BATCH, autoCommit: true);
// 3. 获取对应mapper
```

20、mybatis会将我们传入的非map参数转换成存到一个map中。因此我们在取参的时候都可以通过key取value。

- 当传单个参数时方法中的变量名就是key

```
> selectByEmailAndSex2(@Param("email") String email,@Param("sex") Byte sex);
 where a.email like CONCAT('%', #{email}, '%') and
 a.sex = #{sex}
```

- 当传入javabean时属性名就是key

```
int updateIfOper(TUser record);
if (record.getUsername() == null) {
 user_name = #{userName, jdbcType=VARCHAR},
} else {
```

21、mybatis需要批量操作多个sql并一起提交时需要用到批量插件并将自动提交关闭。但是SpringBatch的事务范围是一个tasklet，即要么全成功，要么全回滚。

22、mybatis与spring集成后批量操作也需要通过批量插件。

23、使用mybatis生成的example时不能进行sql调优，所以一般有些sql需要手动加xml写。

24、批量执行sql（即batch）原理：会将多条sql添加到一个队列中（即存在内存中），然后统一去执行。因此假如我们要执行一万次insert，最好将它分批次去执行，不然队列中维护的sql过多可能会导致内存溢出。具体没有条数限制，但我们需考虑全面。

25、对于批量更新操作缓存SQL以提高性能，但是无法获取到update、delete返回的行数。

25、mybatis的resultMap不是针对POJO的，而是针对一次查询的，根据不同的查询结果定制不同的resultMap，重复的部分可以通过extends继承。

26、对于【一对一、一对多、多对多】的关系可以使用关联查询，关联查询的两种关联方式为【嵌套结果：使用嵌套结果映射来处理重复的联合结果的子集。嵌套查询：通过执行另外一个SQL映射语句来返回预期的复杂类型】：

### 1. 一对一【exp：汽车与发动机唯一对应】

#### a. 嵌套结果：用一个sql从数据库多张表查询出数据，通过resultMap映射

```
<select id="selectUserPosition1" resultMap="userAndPosition1">
 select
 a.id, user_name, real_name, sex, mobile, email, a.note,
 b.id post_id, b.post_name, b.note post_note
 from
 t_user a, t_position b
 where
 a.position_id = b.id
</select>
```

当多表中有相同字段的时候用别名  
用一个sql从user和position表查询数据

```
<resultMap id="userAndPosition1" extends="BaseResultMap" type="TUser">
 <association property="position" javaType="TPosition" columnPrefix="post_">
 <id column="id" property="id"/>
 <result column="name" property="postName"/>
 <result column="note" property="note"/>
 </association>
</resultMap>
```

意为给association中的三个字段统一加上post\_，即id变为post\_id  
baseResultMap中配置的是user实体与表字段的映射信息，此处直接继承是为了减少重复配置  
一对关系用<association>标签，将这些结果映射到TUser实体的position属性，position是一个实体类，类型为TPosition

#### b. 嵌套查询：将查询user和position的sql分开写（通过column传参）

```
<select id="selectUserPosition2" resultMap="userAndPosition2">
 select
 a.id, a.user_name, a.real_name, a.sex, a.mobile, a.position_id
 from
 t_user a
 </select>
```

从主表即user表查询数据，resultMap为userAndPosition

```
<resultMap id="userAndPosition2" extends="BaseResultMap" type="TUser">
 <association property="position" fetchType="lazy" column="position_id" select="com.enjoylearning.mybatis.mapper.TPositionMapper.selectByPrimaryKey" />
</resultMap>
```

填写完整的路径名指向另外一个mapper (position表对应的mapper) 的selectByPrimaryKey查询  
设置为lazy时表示延迟加载，当我们需要获取user对应该的position信息时才会查询，例如我们调用属性的get方法等。  
设置为eager时第一次查询就会查出来  
对User实体的属性  
查询position表时所需的参数。这个参数根据user表的查询结果获得（即user表中的position\_id字段的值）。如果此处要传多个参数可以通过( position\_id, name, age )这种方式

一对多和一对一用法一样。只不过POJO中会通过list存值。标签换成collections

多对多关系需要中间表做映射的理由。避免数据冗余

映射器查询

嵌套查询的时候在xml中定义了方法，但是在接口中没定义，但是查询没报错，为什么？

27、mybatis的一级缓存生命周期是在同一个sqlSession（生命周期短）。sqlSession关闭之后缓存也就没了。如果中间sqlSession去执行commit操作（执行插入、更新、删除），会清空SqlSession中的一级缓存，这样在同一个SQLSession中避免了脏读。但是在多线程情况下当另一个线程中的sqlSession修改了数据以后也会出现脏读情况。解决办法是后续select之前，执行sqlSession.clearCache()方法来清除缓存就可以了。一级缓存也可以配合数据库事务也可以解决这个问题（将数据库事务级别设置为读已提交）。

27、在多对一(N+1)的时候就可以用一级缓存：例如多人属于同一个部门，使用嵌套查询语法的时候第一个人获取到这个部门信息以后，查其他人的部门信息时就可以从缓存中取数据了。

27、二级缓存存在于 SqlSessionFactory 的生命周期中，可以理解为跨sqlSession。但是以namespace为单位的，不同的namespace之间是相互隔离的。我们可以通过<cache-ref/>标签共享二级缓存。如果调用相同namespace下的mapper映射文件中的增删改SQL，并执行了commit操作。此时会清空该namespace下的二级缓存。

```
<mapper namespace="com.enjoylearning.mybatis.mapper.TUserMapper">
 <cache-ref namespace="com.enjoylearning.mybatis.mapper.TUserRoleMapper"></cache-ref>
```

28、如果开启了二级缓存和一级缓存，则首先会查二级、在查一级、最后查DB。

28、flushCache属性解读（flushCache="true"、flushCache="false"两种），一级二级通用：

- 如果查询语句设置成true，那么每次查询都是去数据库查询，即意味着该查询的二级缓存失效。相当于关闭了缓存。（如下图PPT中的表述有点不恰当）：

◦ 一级缓存默认会启用，想要关闭一级缓存可以在select标签上配置flushCache= "true"；

- 如果增删改语句设置成false，即使用二级缓存，那么如果在数据库中修改了数据，而缓存数据还是原来的，这个时候就会出现脏读。

29、二级缓存会出现脏读，项目中不要使用。产生脏读的原因：假如namespaceA和namespaceB缓存了同样的数据，如果namespaceA的sqlSession执行了增、删、改操作并commit了，此时namespaceA的二级缓存会被清除（意味着下次查询时直接读DB，避免了脏读），而namespaceB中缓存的数据还是修改前的，并没有人通知它数据被修改了，所以查询的时候它还是读的缓存中的数据而不去读DB，此时就会产生脏读。

30、mybatis集成spring之后由面向sqlSession编程转为面向接口编程。引入mybatis-spring依赖，mybatis-config.xml中只需要配置setting中的东西，其他的配置都转到了spring的xml中，看hivegl的配置方式就懂了。

31、mybatis整体架构按逻辑可分为三层（接口层、核心处理层、基础支撑层），上层依赖于下层，越往下层业务属性越低。基础支撑层的东西我们可以单独拿出来用在别的地方。

32、mybatis-spring扫描mapeer接口（不是xml）的三种方式：

（<https://blog.csdn.net/u010013573/article/details/87860078>）

### 33、@Mapper注解和@MapperScan注解

(<https://www.jianshu.com/p/6482c1278470>)

## 源码解析

### 一、日志模块

- 使用的设计模式：
  - 适配器模式：因为日志厂商，没有专门针对Mybatis的专门日志模块。所以Mybatis要引入外部的日志模块，并将日志级别统一适配到mybatis定义的四个级别。
  - 动态代理模式：通过ConnectionLogger、PreparedStatementLog、ResultSetLogger、StatementLogger增强接口实现类，在数据库操作的时打出相应的日志：

```
* @param conn - the original connection
* @return - the connection with Logging
*/
public static Connection newInstance(Connection conn, Log statementLog, int queryStack) {
 InvocationHandler handler = new ConnectionLogger(conn, statementLog, queryStack);
 ClassLoader cl = Connection.class.getClassLoader();
 return (Connection) Proxy.newProxyInstance(cl, new Class[]{Connection.class}, handler);
}
//一般不被static修饰的时候我们直接通过this将当前对象传递进来，但是此处被static修饰后我们只能新建一个
//ConnectionLogger对象，因为在static方法中不能使用this，在static方法中调用普通方法需要对象实例去调用
//对连接的增强
return the wrapped connection
//对连接的增强
public Object invoke(Object proxy, Method method, Object[] params)
 throws Throwable {
 try {
 //如果是从Object继承的方法直接忽略
 if (Object.class.equals(method.getDeclaringClass())) {
 return method.invoke(this, params);
 }
 //如果是调用prepareStatement、prepareCall、createStatement的方法，打印要执行的sql语句
 //如果是PreparedStatement的代理对象，如果prepareStatement也具备日志能力，打印参数
 if ("prepareStatement".equals(method.getName())) {
 if (isDebugEnabled()) {
 debug(text: " Preparing: " + removeBreakingWhitespace((String) params[0]), input: true); //打印SQL语句
 }
 PreparedStatement stmt = (PreparedStatement) method.invoke(connection, params);
 stmt = PreparedStatementLogger.newInstance(stmt, statementLog, queryStack); //创建代理对象
 return stmt;
 } else if ("prepareCall".equals(method.getName())) {
 if (isDebugEnabled()) {
 debug(text: " Preparing: " + removeBreakingWhitespace((String) params[0]), input: true); //打印SQL语句
 }
 PreparedStatement stmt = (PreparedStatement) method.invoke(connection, params); //创建代理对象
 stmt = PreparedStatementLogger.newInstance(stmt, statementLog, queryStack); //创建代理对象
 return stmt;
 } else if ("createStatement".equals(method.getName())) {
 Statement stmt = (Statement) method.invoke(connection, params);
 stmt = StatementLogger.newInstance(stmt, statementLog, queryStack); //创建代理对象
 return stmt;
 } else {
 return method.invoke(connection, params);
 }
 }
}
```

ConnectionLogger实现了InvocationHandler接口，用来对Connection进行增强，而且获取代理对象的方法是被static修饰的

一般不被static修饰的时候我们直接通过this将当前对象传递进来，但是此处被static修饰后我们只能新建一个ConnectionLogger对象，因为在static方法中不能使用this，在static方法中调用普通方法需要对象实例去调用

ConnectionLogger对Connection的增强，对于不同的方法调用打印不同的日志

- 日志源码解读：
  - mybatis的日志接口定义了trace、debug、warn、error四种日志级别。
  - 为不同的日志服务 (slf4J、commonsLogging、Log4J2、Log4J、JdkLog) 提供了不同的适配器
  - 通过LogFactory的static代码块进行日志服务的加载，优先级：slf4J → commonsLogging → Log4J2 → Log4J → JdkLog。应用里边配置了引入了那种日志框架，mybatis就会根据优先级去加载。

## 二、数据源模块

- 使用的设计模式：
  - 工厂模式：因为常见的数据源组件都实现了 javax.sql.DataSource 接口，一般情况下数据源的初始化过程参数较多，比较复杂，所以提供工厂类给客户端。
- 数据源源码解读：
  - mybatis自己定义的数据源实现有两个 (UnpooledDataSource【不使用连接池】、 PooledDataSource【使用连接池】)。
  - 为什么 Class.forName("com.mysql.jdbc.Driver") 后，驱动就被注册到 DriverManager?——> 因为通过 Class.forName() 加载 Driver 类的时候会执行静态代码块，而 Driver 中的静态代码块如下图所示：

```

public class Driver extends NonRegisteringDriver implements java.sql.Driver {
 public Driver() throws SQLException {
 }

 static {
 try {
 DriverManager.registerDriver(new Driver());
 } catch (SQLException var1) {
 throw new RuntimeException("Can't register driver!");
 }
 }
}

```

将Driver注册到了DriverManager

- **DataSourceFactory**用来生产**DataSource**（数据源），  
**DataSource**用来生产数据库连接**Connection**。

- **PoolState**用来记录连接池状态。其中活跃连接数和空闲连接数通过两个集合保存。其他状态有累计超时时间、等待次数、无效连接次数等：

```

17 public class PoolState {
18
19 protected PooledDataSource dataSource;
20 //空闲的连接池资源集合
21 protected final List<PooledConnection> idleConnections = new ArrayList<>();
22 //活跃的连接池资源集合
23 protected final List<PooledConnection> activeConnections = new ArrayList<>();
24 //请求的次数
25 protected long requestCount = 0;
26 //累计的获得连接的时间
27 protected long accumulatedRequestTime = 0;
28 //累计的使用连接的时间。从连接取出到归还，算一次使用的时间
29 protected long accumulatedCheckoutTime = 0;
30 //使用连接超时的次数
31 protected long claimedOverdueConnectionCount = 0;
32 //累计超时时间
33 protected long accumulatedCheckoutTimeOfOverdueConnections = 0;
34 //累计等待时间
35 protected long accumulatedWaitTime = 0;
36 //等待次数
37 protected long hadToWaitCount = 0;
38 //无效的连接次数
39 protected long badConnectionCount = 0;
40
41 }

```

- **PooledConnection**使用动态代理对**connection**进行了封装、增强：

```

class PooledConnection implements InvocationHandler {
 private static final String CLOSE = "close";
 private static final Class<?>[] IFACES = new Class<?>[] { Connection.class };

 private final int hashCode;
 //记录当前连接所在的dataSource对象，本次连接是有这个dataSource创建的，关闭后也是回到这个dataSource
 private final PooledDataSource dataSource;
 //真正的连接对象
 private final Connection realConnection;
 //从dataSource取来连接对象
 private final Connection proxyConnection;
 private long checkoutTimestamp;
 //连接创建的时间戳
 private long createdTimestamp;
 //连接最后一次使用的时间戳
 private long lastUsedTimestamp;
 //根据dataSourceUrl、用户名、密码生成一个hash值，唯一标识一个连接池
 private int connectionTypeCode;
 //连接是否有效
 private boolean valid;
}

```

- 通过**PooledConnection**对**Connection**进行了代理，当调用**connection**的**close()**方法时会进行增强，调用

## PooledDataSource.pushConnection(PooledConnection conn)

```
 public PooledConnection(Connection connection, PooledDataSource dataSource) {
 this.hashCode = connection.hashCode();
 this.realConnection = connection;
 this.dataSource = dataSource;
 this.createdTimestamp = System.currentTimeMillis();
 this.lastUsedTimestamp = System.currentTimeMillis();
 this.valid = true;
 this.proxyConnection = (Connection) Proxy.newProxyInstance(Connection.class.getClassLoader(), IFACES, h: this);
 }

 @Override
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
 String methodName = method.getName();
 if (CLOSE.hashCode() == methodName.hashCode() && CLOSE.equals(methodName)) { // 如果是调用连接的close方法，不是真正的关闭
 dataSource.pushConnection(this); // 通过pooled数据源进行回收
 return null;
 } else {

```

- PooledDataSource可以配置超时时间、最大连接数等参数。同时还可以配置测试连接是否有效的sql语句，mysql的测试语句为 【select 1】。可以看hivegl中对德鲁伊的配置
- mybatis提供的连接池PooledDataSource设计思路和线程池差不多。
- PooledDataSource获取连接【popConnection()】流程：
  - 判断是否有空闲连接，有的话直接取一个连接出来：

```
while (conn == null) {
 synchronized (state) { // 获取连接必须是同步的
 if (!state.idleConnections.isEmpty()) { // 检测是否有空闲连接
 // Pool has available connection
 // 有空闲连接直接使用
 conn = state.idleConnections.remove(index: 0);
 if (log.isDebugEnabled()) {

```

- 如果没有空闲连接，判断活跃连接数是否大于最大连接数。否就创建：

```
} else { // 没有空闲连接
 if (state.activeConnections.size() < poolMaximumActiveConnections) { // 判断活跃连接池中的数量是否大于最大连接数
 // 未达到最大连接数，直接创建
 conn = new PooledConnection(dataSource.getConnection(), dataSource: this);
 if (log.isDebugEnabled()) {
 log.debug(s: "Created connection " + conn.getRealHashCode() + ".");
 }
}

```

- 如果活跃连接等于最大连接数，则获取活跃连接集合中的第一个连接判断其是否超时（因为是按创建顺序加到活跃连接集合，所以只需判断一个是否超时就ok，第一个没超时，则后边的都不可能超时），如果已超时则判断此connection是否设置了自动提交事务，如果否则我们需要回滚此connection的事务，并且通过此connection的realConnection创建一个新的连接：

```

} else { // 如果已超时且最大连接数，则不能创建连接
 // 判断活跃连接集合中的第一个连接是否超时
 PooledConnection oldestActiveConnection = state.activeConnections.get(0);
 long longestCheckoutTime = oldestActiveConnection.getCheckoutTime();
 if (longestCheckoutTime > poolMaximumCheckoutTime) { // 检测是否已经以及超过最长使用时间
 // 如果超时，对超时连接的信息进行统计
 state.claimedOverdueConnectionCount++; // 超时连接次数+1
 state.accumulatedCheckoutTimeOfOverdueConnections += longestCheckoutTime; // 累计超时时间增加
 state.accumulatedCheckoutTime += longestCheckoutTime; // 累计的使用连接的时间增加
 state.activeConnections.remove(oldestActiveConnection); // 从活跃连接中删除
 if (!oldestActiveConnection.getRealConnection().getAutoCommit()) { // 如果超时连接未提交，则手动回滚
 try {
 oldestActiveConnection.getRealConnection().rollback();
 } catch (SQLException e) { // 发生异常仅记录日志
 log.error("Error occurred while rolling back connection: " + e.getMessage());
 }
 }
 Just Log a message for debug and continue to execute the following
 statement Like nothing happened.
 Wrap the bad connection with a new PooledConnection, this will help
 to not interrupt current executing thread and give current thread a
 chance to join the next competition for another valid/good database
 connection. At the end of this Loop, bad {@link @conn} will be set as null.
 }
 log.debug("Bad connection. Could not roll back");
}

// 根据realConnection创建新的PooledConnection
conn = new PooledConnection(oldestActiveConnection.getRealConnection(), dataSource: this);
conn.setCreatedTimestamp(oldestActiveConnection.getCreatedTimestamp());
conn.setLastUsedTimestamp(oldestActiveConnection.getLastUsedTimestamp());
// 让老连接失效
oldestActiveConnection.invalidate();
if (log.isDebugEnabled()) {
 log.debug("Claimed overdue connection " + conn.getRealHashCode() + ".");
}

```

根据realConnection创建新的  
PooledConnection

- 如果没有连接超时且无法创建新的连接，那么将此线程阻塞固定时间，并有重试机制，达到设置的重试次数后还没有获得连接的话就报错：

```

} else {
 // 无空闲连接，最早创建的连接没有失效，无法创建新连接，只能阻塞
 try {
 if (!countedWait) {
 state.hadToWaitCount++; // 连接池累计等待次数加1
 countedWait = true;
 }
 if (log.isDebugEnabled()) {
 log.debug("Waiting as long as " + poolTimeToWait + " milliseconds for connection.");
 }
 long wt = System.currentTimeMillis();
 state.wait(poolTimeToWait); // 阻塞等待指定时间
 state.accumulatedWaitTime += System.currentTimeMillis() - wt; // 累计等待时间增加
 } catch (InterruptedException e) {
 break;
 }
}

```

- pushConnection释放连接流程

- 1、从活跃连接队列移除；
- 2、conn.isValid()方法中判断

**realConnection是否已关闭，判断为true则结束，判断为false，则通过这个可用的realConnection再次创建一个PooledConnection并将它放到空闲队列中并同时唤醒其他阻塞的等待数据库连接的线程：**

```

359 protected void pushConnection(PooledConnection conn) throws SQLException {
360
361 synchronized (state) { //回收连接必须是同步的
362 state.activeConnections.remove(conn); //从活跃连接池中删除此连接
363 if (conn.isValid()) {
364 //判断对端连接池是否到达连接上限
365 if (state.idleConnections.size() < poolMaximumIdleConnections && conn.getConnectionTypeCode() == expectedConnectionTypeCode)
366 //没有达到上限,进行操作
367 state.accumulatedCheckoutTime += conn.getCheckoutTime();
368 if (!conn.getRealConnection().getAutoCommit()) {
369 conn.getRealConnection().rollback(); //如果还有事务没有提交,进行回滚操作
370 }
371 //基于该连接, 创建一个新的连接资源, 并刷新连接状态
372 PooledConnection newConn = new PooledConnection(conn.getRealConnection(), dataSource: this);
373 state.idleConnections.add(newConn);
374 newConn.setCreatedTimestamp(conn.getCreatedTimestamp());
375 newConn.setLastUsedTimestamp(conn.getLastUsedTimestamp());
376 //老连接失效
377 conn.invalidate();
378 if (log.isDebugEnabled()) {
379 log.debug("Returned connection " + newConn.getRealHashCode() + " to pool.");
380 }
381 //唤醒其他的阻塞的等待获取连接的线程
382 state.notifyAll();
383 } else //如果对端连接池已经处理上限了, 将连接真实关闭
384 state.accumulatedCheckoutTime += conn.getCheckoutTime();
385 if (!conn.getRealConnection().getAutoCommit()) {
386 conn.getRealConnection().rollback();
387 }
388 //关闭真的数据库连接
389 conn.getRealConnection().close();
390 if (log.isDebugEnabled()) {
391 log.debug("Closed connection " + conn.getRealHashCode() + ".");
392 }
393 //将连接对象设置为无效

```

此处当空闲连接数还没达到上限的时候将通过realConnection再次创建PooledConnection达到重复利用realConnection的作用

### 三、缓存模块

- 缓存模块使用了装饰器模式。
- Cache: Cache接口是缓存模块的核心接口，定义了缓存的基本操作；
- CacheKey: Mybatis中涉及到动态SQL的原因，缓存项的key不能仅仅通过一个String来表示，所以通过CacheKey来封装缓存的Key值，CacheKey可以封装多个影响缓存项的因素；判断两个CacheKey是否相同关键是比較两个对象的hash值是否一致：

■ 构成CacheKey的对象

- ✓ mappedStatement的id
- ✓ 指定查询结果集的范围（分页信息）
- ✓ 查询所使用的SQL语句
- ✓ 用户传递给SQL语句的实际参数值

■ 重点解读方法

- ✓ update(Object obj)
- ✓ equals(Object obj)

```
/*
 * public class CacheKey implements Cloneable, Serializable {
 *
 * private static final long serialVersionUID = 1146682552656046210L;
 *
 * public static final CacheKey NULL_CACHE_KEY = new NullCacheKey();
 *
 * private static final int DEFAULT_MULTIPLIER = 37;
 * private static final int DEFAULT_HASHCODE = 17;
 *
 * private final int multiplier; // 参与hash计算的乘数
 * private int hashCode; // CacheKey的hash值，在update函数中实时运算出来的
 * private long checksum; // 校验和 hash 值的和
 * private int count; // updatelist 的中元素个数
 * // 8/21/2017 - Sonarlint flags this as needing to be marked transient. While true if content is not serializable, this is
 * // 该集合中的元素觉得两个CacheKey是否相等
 * private List<Object> updatelist;
 * }
```

集合中存储了查询方法、参数、分页等信息的hash值

- CacheKey中重写了equals()方法，存储了分页等值，因为对于分页查询，虽然sql一样，但是第一页和第二页的数据是不一样的，查第二页的时候不能去取第一页的缓存，即第一页和第二页数据对应的是两个CacheKey。CacheKey的equals()方法比较严谨，先比较各hash值相加、相乘后的值是否一样，如果是则还需根据updatelist中各单项的hash值进行比较。
- PerpetualCache：在缓存模块中扮演ConcreteComponent角色（即本体），使用map来存储，map中的key为mybatis自己定义的CacheKey，而不是我们通常用的String。

```
public class PerpetualCache implements Cache {
 private final String id;
 private Map<Object, Object> cache = new HashMap<>();
}
```

在cache真正缓存数据

- mybatis提供的对PerpetualCache进行增强的缓存装饰器有BlockingCache、FifoCache、LruCache等。
- BlockingCache详解

```

public class BlockingCache implements Cache {
 //阻塞的超时时间
 private long timeout;
 //被装饰的底层对象，一般是PerpetualCache
 private final Cache delegate;
 //锁与key是一一对应的
 private final ConcurrentHashMap<Object, ReentrantLock> locks;
 public BlockingCache(Cache delegate) {
 this.delegate = delegate;
 }
 @Override
 public Object getObject(Object key) {
 acquireLock(key); //根据key获得锁对象，获取锁成功加锁，获取锁失败阻塞一段时间重试
 Object value = delegate.getObject(key);
 if (value != null) { //获取数据成功的，要释放锁
 releaseLock(key);
 }
 return value;
 }
 //根据key获得锁对象，获取锁成功加锁，获取锁失败阻塞一段时间重试
 private void acquireLock(Object key) {
 //获得锁对象
 Lock lock = getLockForKey(key); ← 获取key对应的锁
 if (timeout > 0) { //使用带超时时间的锁
 try {
 boolean acquired = lock.tryLock(timeout, TimeUnit.MILLISECONDS);
 if (!acquired) { //如果超时抛出异常
 throw new CacheException("Couldn't get a lock in " + timeout + " for the key " + key + " at the cache " + delegate);
 }
 } catch (InterruptedException e) {
 throw new CacheException("Got interrupted while trying to acquire lock for key " + key, e);
 }
 } else { //使用不带超时时间的锁
 lock.lock();
 }
 }
 private ReentrantLock getLockForKey(Object key) {
 ReentrantLock lock = new ReentrantLock(); //创建锁
 ReentrantLock previous = locks.putIfAbsent(key, lock); //把新锁添加到locks集合中，如果添加成功使用新锁，如果添加失败则返回旧锁
 return previous == null ? lock : previous; ← 如果该key对应的锁已经被创建则返回旧锁，如果没有创建则将新创建的锁加到map中
 }
}

```

- FifoCache内部维护了一个Deque按顺序来存储缓存的key，默认缓存的数量为1024。每次有新数据进来后都会插入到Deque的尾部，当Deque满了之后它会将头部的数据淘汰：

```

private void cycleKeyList(Object key) {
 keyList.addLast(key); ← 在尾部添加key
 if (keyList.size() > size) {
 Object oldestKey = keyList.removeFirst(); ← 满了以后淘汰头部key
 delegate removeObject(oldestKey);
 }
}

```

## 四、反射模块

- ORM框架从数据库加载数据的过程：
  - 从数据库加载数据
  - 通过resultMap找到POJO映射规则

- mybatis提供了对象工厂类**DefaultObjectFactory**, 通过反射去构建POJO。
  - 通过反射给POJO对象属性赋值。**ReflectorFactory**用来创建**Reflector**, **ObjectWrapperFactory**用来创建**ObjectWrapper**。
- 为什么使用反射：因为POJO与数据库列的对应关系对于mybatis来说不是透明的（黑盒）。我们开发者知道数据库中的列和POJO中的那个属性相对应，但是mybatis不知道，同时mybatis也不知道有哪些POJO类，因此我们需要定义**resultMap**将POJO属性与数据库列的对应关系配置起来，这样mybatis就可以根据这个映射关系通过反射将数据库查询结果赋值给POJO相应的属性。因为需要反射所以我们在mybatis中需要配置POJO的包名全路径：

```
<mapper namespace="com.enjoylearning.mapper.TPositionMapper">
 <resultMap id="BaseResultMap" type="com.enjoylearning.mybatis.entity.TPosition">
 </resultMap>
```

- jdk反射api比较复杂，mybatis对dk反射进行了封装，api比较简单易用。
- **DefaultObjectFactory**用来构建POJO：

```
public class DefaultObjectFactory implements ObjectFactory, Serializable {
 private static final long serialVersionUID = -8855120656740914948L;

 @Override
 public <T> T create(Class<T> type) { 使用无参构造构建POJO
 return create(type, constructorArgTypes: null, constructorArgs: null);
 }

 @SuppressWarnings("unchecked")
 @Override
 public <T> T create(Class<T> type, List<Class<?>> constructorArgTypes, List<Object> constructorArgs) { 构造方法参数类型
 Class<?> classToCreate = resolveInterface(type);
 // we know types are assignable
 return (T) instantiateClass(classToCreate, constructorArgTypes, constructorArgs); 构造方法参数值
 }
}
```

- **DefaultReflectorFactory**用来创建**Reflector**, 并将已创建的**Reflector**缓存起来：

```

public class DefaultReflectorFactory implements ReflectorFactory {
 private boolean classCacheEnabled = true;
 private final ConcurrentMap<Class<?>, Reflector> reflectorMap = new ConcurrentHashMap<>();
}

public DefaultReflectorFactory() {
}

@Override
public boolean isClassCacheEnabled() {
 return classCacheEnabled;
}

@Override
public void setClassCacheEnabled(boolean classCacheEnabled) {
 this.classCacheEnabled = classCacheEnabled;
}

@Override
public Reflector findForClass(Class<?> type) {
 if (classCacheEnabled) {
 // synchronized(type) removed - see issue #461
 return reflectorMap.computeIfAbsent(type, Reflector::new);
 } else {
 return new Reflector(type);
 }
}

```

用来缓存已创建的Reflector

computeIfAbsent()方法意为如果key对应的value为空  
则会调用第二个参数传入的函数并以key为参数

此处翻译过来就是当reflectorMap中key=type对应的value为空时，会调用  
Reflector reflector = new Reflector(type), [将key=type,  
value=reflector的键值对存到reflectorMap中]，并将reflector返回

如果不使用缓存，则直接创建一个新的Reflector

- **Reflector与POJO一一对应，它缓存了类的元信息：**

```

public class Reflector {
 private final Class<?> type; // 对应的class
 private final String[] readablePropertyNames; // 可读属性的名称集合，存在get方法即可读
 private final String[] writeablePropertyNames; // 可写属性的名称集合，存在set方法即可写
 private final Map<String, Invoker> setMethods = new HashMap<>(); // 保存属性相关的set方法
 private final Map<String, Invoker> getMethods = new HashMap<>(); // 保存属性相关的get方法
 private final Map<String, Class<?>> setTypes = new HashMap<>(); // 保存属性相关的set方法入参类型
 private final Map<String, Class<?>> getTypes = new HashMap<>(); // 保存属性相关的get方法返回类型
 private Constructor<?> defaultConstructor; // class默认的构造函数
}

```

无论POJO有没有提供set、get方法，都会被封装成不同的Invoker，当调用的时候通过invoke方法去赋值、取值

- mybatis直接通过暴力反射的方式给属性赋值，无论POJO有没有给属性提供set方法。因为mybatis提供了统一接口Invoker，实现有三个MethodInvoker、SetFieldInvoker、GetFieldInvoker：

- **MethodInvoker:** 当POJO提供了get、set方法时会被封装成MethodInvoker存到Reflector中，在invoke方法中通过反射取值、赋值：

```

@Override
public Object invoke(Object target, Object[] args) throws IllegalAccessException, InvocationTargetException {
 return method.invoke(target, args);
}

```

此处的method为POJO中提供的set或者get方法

- **SetFieldInvoker:** 当POJO没有提供set方法时会封装成SetFieldInvoker存到Reflector中，在invoke方法中通过反射给属性赋值：

```

@Override
public Object invoke(Object target, Object[] args) throws IllegalAccessException, InvocationTargetException {
 field.set(target, args[0]);
 return null;
}

```

field为没有set方法的属性

target为POJO对象，通过反射给属性赋值的写法就是这样

- **GetFieldInvoker**: 当POJO 没有提供get方法时会封装成**GetFieldInvoker**存到Reflector中，在invoke方法中通过反射获取属性值：

```
@Override
public Object invoke(Object target, Object[] args) throws IllegalAccessException, InvocationTargetException {
 return field.get(target);
}
```

- **ObjectWrapperFactory**: **ObjectWrapper** 的工厂类，用于创建**ObjectWrapper**。
- **ObjectWrapper**: 对对象的包装，抽象了对象的属性信息，他定义了一系列查询对象属性信息的方法，以及更新属性的方法。具体实现看**BeanWrapper**。
- **MetaObject**包装了mybatis中五个核心的反射类，也是提供给外部使用的反射工具类，我们在自己项目通可以面向**MetaObject**进行反射编程，api比较简单易用。使用流程如下：

- 获取**MetaObject**对象：

```
//反射工具类初始化
ObjectFactory objectFactory = new DefaultObjectFactory();
TUser user = objectFactory.create(TUser.class);
ObjectWrapperFactory objectWrapperFactory = new DefaultObjectWrapperFactory();
ReflectorFactory reflectorFactory = new DefaultReflectorFactory();
MetaObject metaObject = MetaObject.forObject(user, objectFactory, objectWrapperFactory, reflectorFactory);
```

- 通过**PropertyTokenizer**给属性赋值：

```
PropertyTokenizer prop = new PropertyTokenizer(fullname: "userName");
wrapperForUser.set(prop, "lison");
System.out.println(userTemp);
```

- mybatis还提供了**SystemMetaObject**用来获取**MetaObject**:

```
/*
public final class SystemMetaObject {

 public static final ObjectFactory DEFAULT_OBJECT_FACTORY = new DefaultObjectFactory();
 public static final ObjectWrapperFactory DEFAULT_OBJECT_WRAPPER_FACTORY = new DefaultObjectWrapperFactory();
 public static final MetaObject NULL_META_OBJECT = MetaObject.forObject(NullObject.class, DEFAULT_OBJECT_FACTORY, DEFAULT_OBJECT_WRAPPER_FACTORY);

 private SystemMetaObject() {
 // Prevent Instantiation of Static Class
 }

 private static class NullObject {
 }

 public static MetaObject forObject(Object object) {
 return MetaObject.forObject(object, DEFAULT_OBJECT_FACTORY, DEFAULT_OBJECT_WRAPPER_FACTORY, new DefaultReflectorFactory());
 }
}
```

## 五、binding模块

- 作用：mybatis之所以能面向接口编程就是通过binding实现的
- MapperRegistry：mapper接口和对应的代理对象工厂的注册中心，在configuration中存储的是
  - 获取mapper，实际获取到的是mapper的代理对象，需要注意的是每次调用getMapper()方法都会生成新的mapper代理对象：

```


// 3. 获取对应mapper
TUserMapper mapper = sqlSession.getMapper(TUserMapper.class);
// 4. 执行查询语句并返回结果
/unchecked/ 解析生成Configuration对时会为每个mapper接口生成一个MapperProxyFactory，并缓存到map中
public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
 final MapperProxyFactory<T> mapperProxyFactory = (MapperProxyFactory<T>) knownMappers.get(type);
 if (mapperProxyFactory == null) {
 throw new BindingException("Type " + type + " is not known to the MapperRegistry.");
 }
 try {
 return mapperProxyFactory.newInstance(sqlSession); ← 获取mapper接口的代理对象
 } catch (Exception e) {
 throw new BindingException("Error getting mapper instance. Cause: " + e, e);
 }
}


```

- MapperProxyFactory：用于生成mapper接口动态代理的实例对象：

```


public class MapperProxyFactory<T> {
 //mapper接口的Class对象
 private final Class<T> mapperInterface;
 //key是mapper接口中的某个方法的method对象，value是对应的MapperMethod。MapperMethod对象不记录任何状态信息，所以它可以在多个代理对象之间共享
 private final Map<Method, MapperMethod> methodCache = new ConcurrentHashMap<>();

 public MapperProxyFactory(Class<T> mapperInterface) {
 this.mapperInterface = mapperInterface;
 }

 public Class<T> getMapperInterface() {
 return mapperInterface;
 }

 public Map<Method, MapperMethod> getMethodCache() {
 return methodCache;
 }

 /unchecked/
 protected T newInstance(MapperProxy<T> mapperProxy) {
 //创建实现了mapper接口的动态代理对象
 return (T) Proxy.newProxyInstance(mapperInterface.getClassLoader(), new Class[] { mapperInterface }, mapperProxy);
 }

 public T newInstance(SqlSession sqlSession) {
 //每次调用都会创建新的MapperProxy对象
 final MapperProxy<T> mapperProxy = new MapperProxy<>(sqlSession, mapperInterface, methodCache);
 return newInstance(mapperProxy);
 }
}


```

- MapperProxy：实现了InvocationHandler接口，它是增强mapper接口的实现：

```

public class MapperProxy<T> implements InvocationHandler, Serializable {
 private static final long serialVersionUID = -642454039855972983L;
 private final SqlSession sqlSession;//记录关联的sqlSession对象
 private final Class<T> mapperInterface;/mapper接口对应的class对象;
 //key为mapper接口中的某个方法的method对象, value是对应的MapperMethod. MapperMethod对象不记录任何状态信息, 所以它可以在多个代理对象之间共享
 private final Map<Method, MapperMethod> methodCache;

 public MapperProxy(SqlSession sqlSession, Class<T> mapperInterface, Map<Method, MapperMethod> methodCache) {
 this.sqlSession = sqlSession;
 this.mapperInterface = mapperInterface;
 this.methodCache = methodCache;
 }

 @Override
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
 try {
 if (Object.class.equals(method.getDeclaringClass())) {//如果是Object本身的方法不增强
 return method.invoke(this, args);
 } else if (isDefaultMethod(method)) {
 return invokeDefaultMethod(proxy, method, args);
 }
 } catch (Throwable t) {
 throw ExceptionUtil.unwrapThrowable(t);
 }
 //从缓存中获取MapperMethod对象, 如果缓存中没有, 则创建一个, 并添加到缓存中
 final MapperMethod mapperMethod = cachedMapperMethod(method);
 //调用execute方法执行sql
 return mapperMethod.execute(sqlSession, args);
 }
}

```

- **MapperMethod: 封装了Mapper接口中对应方法的信息, 以及对应的sql语句的信息; 它是mapper接口与映射配置文件中sql语句的桥梁:**

- **有两个属性为SqlCommand、MethodSignature:**

```

public class MapperMethod {
 //从configuration中获取方法的命名空间, 方法名以及SQL语句的类型
 private final SqlCommand command;
 //封装mapper接口方法的相关信息(入参, 返回类型),
 private final MethodSignature method;
}

```

- **excute()方法, MapperProxy对mapper的增强就是调用excute()方法去执行sql, 最终还是通过sqlSession去执行sql (对用户来说mybatis是面向接口编程的, 比较方便, 但底层还是使用sqlSession去执行sql) :**

```

public Object execute(SqlSession sqlSession, Object[] args) {
 Object result;
 //根据sql语句的类型以及接口返回的参数选择调用不同的
 switch(command.getType()) {
 case INSERT: { //根据不同的sql语句类型以及接口返回的参数选择调用不同的sqlSession中的方法
 Object param = method.convertArgsToSqlCommandParam(args);
 result = rowCountResult(sqlSession.insert(command.getName(), param));
 break;
 }
 case UPDATE: {
 Object param = method.convertArgsToSqlCommandParam(args);
 result = rowCountResult(sqlSession.update(command.getName(), param));
 break;
 }
 case DELETE: {
 Object param = method.convertArgsToSqlCommandParam(args);
 result = rowCountResult(sqlSession.delete(command.getName(), param));
 break;
 }
 case SELECT: {
 if (method.returnsVoid() && method.hasResultHandler()) {//返回值为void
 executeWithResultHandler(sqlSession, args);
 result = null;
 } else if (method.returnsMany()) {//返回值为集合或者数组
 result = executeForMany(sqlSession, args);
 } else if (method.returnsMap()) {//返回值为map
 result = executeForMap(sqlSession, args);
 } else if (method.returnsCursor()) {//返回值为游标
 result = executeForCursor(sqlSession, args);
 } else //处理返回为单一对象的情况
 //通过参数解析器解析参数
 Object param = method.convertArgsToSqlCommandParam(args);
 result = sqlSession.selectOne(command.getName(), param);
 }
 }
 }
}

```

- **总结: 当我们通过mapper接口调用sql的过程最终是由mapper的代理对象交给sqlSession去执行的 (底层封装的还是ibatis的编程模型) :**

```

 @Test
 // 快速入门
 public void quickStart() throws IOException {
 // 2. 获取sqlSession
 sqlSession = sqlSessionFactory.openSession();
 // 3. 获取对应mapper
 TUserMapper mapper = sqlSession.getMapper(TUserMapper.class);
 // 4. 执行查询语句并返回结果
 TUser user = mapper.selectByPrimaryKey(1);
 System.out.println(user.toString());
 }

 @Test
 // 快速入门 本质分析
 public void originalOperation() throws IOException {
 // 2. 获取sqlSession
 sqlSession = sqlSessionFactory.openSession();
 // 3. 执行查询语句并返回结果
 TUser user = sqlSession.selectOne("com.enjoylearning.mybatis.mapper."
 + "TUserMapper.selectByPrimaryKey", 1);
 System.out.println(user.toString());
 }

```

## 六、SqlSession详解

1. 通过SQLSessionFactory获取sqlSession的方式有两种(与数据库建立一次连接就会创建一个sqlSession)：
  - a. 基于数据源获取，一般我们项目中都是使用的这种方式。因为我们会使用druid这种数据源，它会维护一个数据库连接池，创建sqlSession的时候会从数据源的连接池中获取connection：

```

// 从数据源获取数据库连接
private SqlSession openSessionFromDataSource(ExecutorType execType, TransactionIsolationLevel level, boolean autoCommit) {
 Transaction tx = null;
 try {
 // 获取mybatis配置文件中的environment对象
 final Environment environment = configuration.getEnvironment();
 // 从environment获取transactionFactory对象
 final TransactionFactory transactionFactory = getTransactionFactoryFromEnvironment(environment);
 // 创建事务对象
 tx = transactionFactory.newTransaction(environment.getDataSource(), level, autoCommit);
 // 根据配置创建executor
 final Executor executor = configuration.newExecutor(tx, execType);
 // 创建DefaultSqlSession
 return new DefaultSqlSession(configuration, executor, autoCommit);
 } catch (Exception e) {
 closeTransaction(tx); // may have fetched a connection so lets call close()
 throw ExceptionFactory.wrapException("Error opening session. Cause: " + e, e);
 } finally {
 ErrorContext.instance().reset();
 }
}

```

- b. 基于connection，这种方式在不使用数据源的情况下使用。我们先获得connection，然后在基于connection获取SqlSession：

```

// 从数据库连接获取sqlSession
private SqlSession openSessionFromConnection(ExecutorType execType, Connection connection) {
 try {
 // 获取当前连接是否设置事务自动提交
 boolean autoCommit;
 try {
 autoCommit = connection.getAutoCommit();
 } catch (SQLException e) {
 // Failover to true, as most poor drivers
 // or databases won't support transactions
 autoCommit = true;
 }
 // 获取mybatis配置文件中的environment对象
 final Environment environment = configuration.getEnvironment();
 // 从environment获取transactionFactory对象
 final TransactionFactory transactionFactory = getTransactionFactoryFromEnvironment(environment);
 // 创建事务对象
 final Transaction tx = transactionFactory.newTransaction(connection);
 // 根据配置创建executor
 final Executor executor = configuration.newExecutor(tx, execType);
 // 创建DefaultSqlSession
 return new DefaultSqlSession(configuration, executor, autoCommit);
 } catch (Exception e) {
 throw ExceptionFactory.wrapException("Error opening session. Cause: " + e, e);
 } finally {
 ErrorContext.instance().reset();
 }
}

```

2. 通过`SqlSessionFactory`获取`SqlSession`时每获取一次都会创建新的`sqlSession`, 而通过`SqlSessionManager`获取`sqlSession`时同一个线程中获取到的是同一个`sqlSession`, 因为`SqlSessionManager`通过`ThreadLocal`将`sqlSession`缓存了起来:

```

public class SqlSessionManager implements SqlSessionFactory, SqlSession {
 //底层封装的
 private final SqlSessionFactory sqlSessionFactory;
 private final SqlSession sqlSessionProxy;

 private final ThreadLocal<SqlSession> localSqlSession = new ThreadLocal<>();

 private SqlSessionManager(SqlSessionFactory sqlSessionFactory) {
 this.sqlSessionFactory = sqlSessionFactory;
 this.sqlSessionProxy = (SqlSession) Proxy.newProxyInstance(
 SqlSessionFactory.class.getClassLoader(),
 new Class[]{SqlSession.class},
 new SqlSessionInterceptor());
 }

 @Override
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
 final SqlSession sqlSession = SqlSessionManager.this.localSqlSession.get();
 if (sqlSession != null) {
 try {
 return method.invoke(sqlSession, args);
 } catch (Throwable t) {
 throw ExceptionUtil.unwrapThrowable(t);
 }
 } else {
 try (SqlSession autoSqlSession = openSession()) {
 try {
 final Object result = method.invoke(autoSqlSession, args);
 autoSqlSession.commit();
 return result;
 } catch (Throwable t) {
 autoSqlSession.rollback();
 throw ExceptionUtil.unwrapThrowable(t);
 }
 }
 }
 }
}

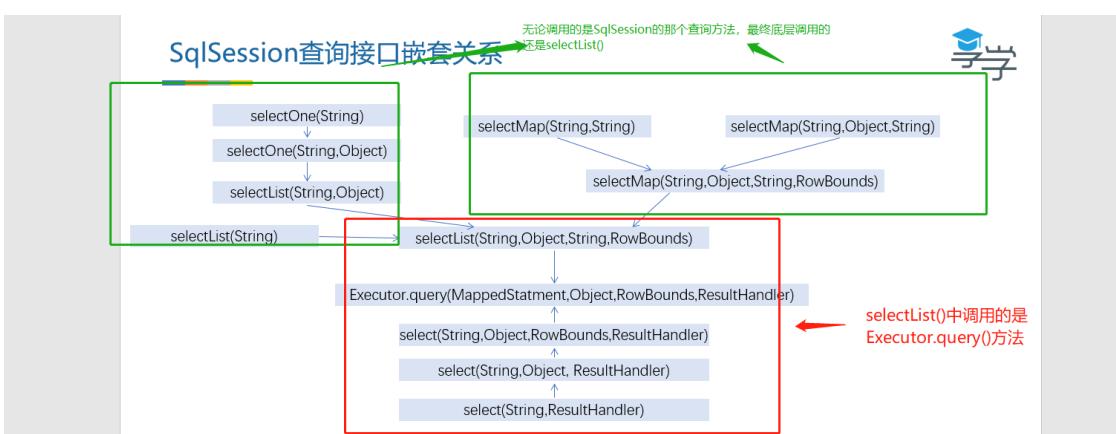
```

SqlSessionManager对SqlSessionFactory进行了封装, 同时还对SqlSessionFactory通过动态代理进行了增强, 增强的代理对象为sqlSessionProxy

对SqlSessionFactory进行增强, 优先从ThreadLocal中获取sqlSession

当ThreadLocal中获取不到时在调用openSession()方法去创建新的sqlSession

3. `SqlSession`的查询过程是一个万剑归宗的状态, 从用户的角度来看是通过`SqlSession`进行数据库操作的, 但实际`SqlSession`的功能都是基于`Executor`来实现的, `Executor`才是核心:

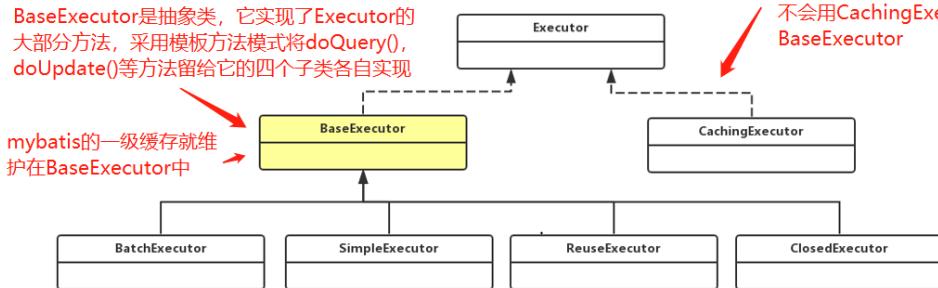


## 七、Executor模块

- Executor接口有两个实现类分别为BaseExecutor、CachingExecutor

■ Executor是MyBatis核心接口之一，定义了数据库操作最基本的方法，SqlSession的功能都是基于它来实现的；  
CachingExecutor用来维护二级缓存，采用装饰器模式，当不开启二级缓存时不会用CachingExecutor去装饰

BaseExecutor是抽象类，它实现了Executor的大部分方法，采用模板方法模式将doQuery()，doUpdate()等方法留给它的四个子类各自实现



- CachingExecutor详解：

- CachingExecutor的作用：

```
public class CachingExecutor implements Executor {
 private final Executor delegate;
 private final TransactionalCacheManager tcm = new TransactionalCacheManager();

 public CachingExecutor(Executor delegate) {
 this.delegate = delegate;
 delegate.setExecutorWrapper(this);
 }
}
```

本体Executor, SimpleExecutor、BatchExecutor、ReuseExecutor之一，由我们的配置文件决定  
当我们开启二级缓存功能时，CachingExecutor对delegate进行装饰，让他具有二级缓存的功能

- CachingExecutor的query()方法详解：

```
@Override
public <E> List<E> query(MappedStatement ms, Object parameterObject, RowBounds rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql)
throws SQLException {
 //从MappedStatement中获取二级缓存
 Cache cache = ms.getCache();
 if (cache != null) {
 flushCacheIfRequired(ms);
 if (ms.isUseCache() && resultHandler == null) {
 ensureNoOutParams(ms, boundSql);
 }
 /unchecked/
 List<E> list = (List<E>) tcm.getObject(cache, key); //从二级缓存中读取数据
 if (list == null) {
 //二级缓存为空，才会调用BaseExecutor.query
 list = delegate.<E> query(ms, parameterObject, rowBounds, resultHandler, key, boundSql);
 tcm.putObject(cache, key, list); // issue #578 and #116
 }
 return list;
 }
 return delegate.<E> query(ms, parameterObject, rowBounds, resultHandler, key, boundSql);
}
```

先从MappedStatement读取二级缓存，因为二级缓存实际是存储在MappedStatement中的，所以二级缓存不会随着CachingExecutor的消亡而消失，这也就是二级缓存会出现脏读的原因  
从缓存获取数据，开启二级缓存后BaseExecutor被CachingExecutor装饰，当调用BaseExecutor的query()方法时会先调用CachingExecutor的query()方法，这就是查二级缓存先于一级缓存的原因  
此处调用BaseExecutor子类的query()方法进行真正的数据库查询

- 当开启二级缓存时CachingExecutor是何时对BaseExecutor进行装饰的：

```

/*
public class DefaultSqlSessionFactory implements SqlSessionFactory {
 private final Configuration configuration;
 public DefaultSqlSessionFactory(Configuration configuration) {
 this.configuration = configuration;
 }
 @Override
 public SqlSession openSession() {
 return openSessionFromDataSource(configuration.getDefaultExecutorType(), level: null, autoCommit: false);
 }
}

//从数据源获取数据库连接
private SqlSession openSessionFromDataSource(ExecutorType execType, TransactionIsolationLevel level, boolean autoCommit) {
 Transaction tx = null;
 try {
 //从mybatis配置文件中的environment对象
 final Environment environment = configuration.getEnvironment();
 //从environment获取transactionFactory对象
 final TransactionFactory transactionFactory = getTransactionFactoryFromEnvironment(environment);
 //创建事务对象
 tx = transactionFactory.newTransaction(environment.getDataSource(), level, autoCommit);
 //根据配置创建executor
 final Executor executor = configuration.newExecutor(tx, execType); ← 2. 根据配置创建Executor, 继续跟进去看
 //创建DefaultSqlSession
 return new DefaultSqlSession(configuration, executor, autoCommit);
 } catch (Exception e) {
 closeTransaction(tx); // may have fetched a connection so lets call close()
 throw ExceptionFactory.wrapException("Error opening session. Cause: " + e, e);
 } finally {
 ErrorContext.instance().reset();
 }
}

public Executor newExecutor(Transaction transaction, ExecutorType executorType) {
 executorType = executorType == null ? defaultExecutorType : executorType;
 executorType = executorType == null ? ExecutorType.SIMPLE : executorType; ← 如果不显示传Executor的类型, 则获取配置文件中的, 如果配置文件中没配置, 则默认为SIMPLE, 即SimpleExecutor
 Executor executor;
 if (ExecutorType.BATCH == executorType) {
 executor = new BatchExecutor(configuration: this, transaction);
 } else if (ExecutorType.REUSE == executorType) {
 executor = new ReuseExecutor(configuration: this, transaction);
 } else {
 executor = new SimpleExecutor(configuration: this, transaction);
 }
 //如果启用了<cache>节点, 通过装饰器, 添加一级缓存的能力
 if (cacheEnabled) {
 executor = new CachingExecutor(executor); ← 如果开启了二级缓存功能, 则此处通过CachingExecutor对BaseEx进行装饰
 }
 //通过interceptorChain遍历所有的插件为executor增强, 添加插件的功能
 executor = (Executor) interceptorChain.pluginAll(executor);
 return executor;
}

```

- **BaseExecutor详解:**

- **BaseExecutor的作用:** BaseExecutor是mybatis的核心接口之一, sqlSession的所有功能都是基于它实现的, 并且它里边定义了一级缓存属性。它实现了Executor接口的大部分方法, 同时它采用模板方法模式对它的子类方法调用顺序进行了规范约束, 因为它是抽象类, 所以不能被实例化, 因此实际情况是mybatis根据配置实例化它的子类来进行数据库操作, 它有三个子类为**SimpleExecutor、ReuseExecutor、BatchExecutor**, 默认是**SimpleExecutor**:

```
public abstract class BaseExecutor implements Executor {

 private static final Log log = LogFactory.getLog(BaseExecutor.class);

 protected Transaction transaction;//事务对象
 protected Executor wrapper;//封装的Executor对象

 protected ConcurrentLinkedQueue<DeferredLoad> deferredLoads;//延迟加载的队列
 protected PerpetualCache localCache;//一级缓存的实现，PerpetualCache
 protected PerpetualCache localOutputParameterCache;//一级缓存用于要输出的结果
 protected Configuration configuration;//全局唯一configuration对象的引用

 protected int queryStack;//用于嵌套查询的层数
 private boolean closed;
```

因为BaseExecutor是抽象类，所以具体实例化的时候是它的子类，不过在它里边定义了一级缓存，且BaseExecutor随着SqlSession的创建而创建，当SqlSession被回收了他也就跟着被回收了，因此一级缓存的生命周期是SqlSession级别的

### ○ BaseExecutor的query()方法：

```
public void executeQuery(MappedStatement ms) throws SQLException {
 List<E> list = query(ms, null, null, null, null);
 if (list != null) {
 for (E e : list) {
 handleOutputParameters(ms, e);
 }
 }
}

@Override
public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql) throws SQLException {
 ErrorContext.instance().resource(ms.getResource()).activity("executing a query").object(ms.getId());
 if (closed) { // 检查当前executor是否关闭
 throw new ExecutorException("Executor was closed.");
 }

 if (queryStack == 0 && ms.isFlushCacheRequired()) { // 非嵌套查询，并且FlushCache属性为true时，每次查询前都会先清除一级缓存
 clearLocalCache();
 }

 List<E> list;
 try {
 queryStack++; // 查询层次加一
 list = resultHandler == null ? (List<E>) localCache.getObject(key) : null; // 查询以及缓存
 if (list == null) {
 // 针对调用在输出结果处理
 handleLocallyCachedOutputParameters(ms, key, parameter, boundSql);
 } else {
 // 缓存命中，从数据库加载数据
 list = queryFromDatabase(ms, parameter, rowBounds, resultHandler, key, boundSql);
 }
 } finally {
 queryStack--;
 }

 if (queryStack == 0) {
 for (DeferredLoad deferredLoad : deferredLoads) { // 延迟加载处理
 deferredLoad.load();
 }
 }
}

// 正真访问数据库获取结果的方法
private <E> List<E> queryFromDatabase(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql) throws SQLException {
 List<E> list;
 localCache.putObject(key, EXECUTION_PLACEHOLDER); // 在缓存中添加占位符
 try {
 // 调用抽象方法doQuery，方法查询数据库并返回结果，可选的实现包括：simple、reuse、batch
 list = doQuery(ms, parameter, rowBounds, resultHandler, boundSql);
 } finally {
 localCache removeObject(key); // 在缓存中删除占位符
 }
 localCache.putObject(key, list); // 将真正的结果对象添加到一级缓存
 if (ms.getStatementType() == StatementType.CALLABLE) { // 如果是调用存储过程
 localOutputParameterCache.putObject(key, parameter); // 缓存输出类型结果参数
 }
 return list;
}
```

`doQuery()`方法是一个抽象方法，由子类实现，在子类的`doQuery()`方法中可以看出还是通过JDBC那一套进行查询数据库的。

- 具体看SimpleExecutor和ReuseExecutor的doQuery()

方法的具体实现以及它们之间的区别：

```
private Statement prepareStatement(StatementHandler handler, Log statementLog) {
 Statement stmt;
 BoundSql boundSql = handler.getBoundSql();
 String sql = boundSql.getSql(); // 获取SQL语句
 if (isHasStatementFor(sql)) { // 如果SQL语句中包含缓存了对原的Statement
 stmt = getStatement(sql); // 获取缓存的Statement
 applyTransactionTimeout(stmt); // 设置超时时间
 } else { // 不是SQL语句
 Connection connection = getConnection(statementLog);
 stmt = handler.prepareStatement(connection, transaction.getTimeout());
 putStatement(sql, stmt); // 放入缓存中
 }
 // 根据parameterHandler的占位符
 handler.parameterize(stmt);
 ReuseExecutor会缓存Statement, 从缓存中取不到时在新建
 return stmt;
}

@Override
public <E> List<E> doQuery(MappedStatement ms, Object parameter, RowBounds rowBounds) {
 Configuration configuration = ms.getConfiguration(); // 获取configuration对象
 // StatementHandler对象
 StatementHandler handler = configuration.newStatementHandler(wrapper, ms, parameter);
 Statement stmt = prepareStatement(handler, ms.getStatementLog()); // StatementHandler对象从StatementHandler类直接调用stmt，并使用parameterHandler对占位符进行处理
 // StatementHandler对象从ReuseExecutor类直接调用resultHandler，将参数化语句作为参数对对象回
 return handler.<E>query(stmt, resultHandler);
}

@Override
protected <E> Cursor<E> doQueryCursor(MappedStatement ms, Object parameter, RowBounds rowBounds) {
 Configuration configuration = ms.getConfiguration();
 StatementHandler handler = configuration.newStatementHandler(wrapper, ms, parameter);
 Statement stmt = prepareStatement(handler, ms.getStatementLog()); // StatementHandler对象从StatementHandler类直接调用stmt，并使用parameterHandler对占位符进行处理
 return handler.<E>queryCursor(stmt, resultHandler);
}

// ReuseExecutor和SimpleExecutor的不同点在于获取Statement的不同，跟进prepareStatement()方法看
// 从这里可以看出，和JDBC的连接是一样的
// 创建Statement
private Statement prepareStatement(StatementHandler handler, Log statementLog) {
 Statement stmt;
 if (isHasStatementFor(statementLog)) { // 如果Statement对象中缓存了对原的Statement
 stmt = getStatement(statementLog); // 获取Statement对象
 } else { // 不通过StatementHandler，而是通过StatementHandler类直接调用prepareStatement()
 Connection connection = getConnection(statementLog);
 Connection connection = getConnect(statementLog);
 // 通过不同的StatementHandler类直接调用createStatement(prepare)方法
 Statement stmt = handler.prepareConnection(transaction.getTimeout());
 handler.setParameterHandler(parameterHandler);
 handler.parameterize(stmt);
 return stmt;
 }
}

// SimpleExecutor每次都是创建新的Statement
SimpleExecutor.java
```

- **BaseExecutor**是一个抽象类，它有三个子类分别为**SimpleExecutor**、**BatchExecutor**、**ReuseExecutor**。**BaseExecutor**使用了模板方法模式，它实现了**Executor**的大多数方法，其子类需要实现的抽象方法有**doUpdate()**,**doQuery()**，从子类实现的**doQuery()**方法可以看出来底层还是在玩**JDBC**的那一套。
- 如果在开启了二级缓存，则**DefaultSqlSessionFactory**构建**DefaultSqlSession**时会将**baseExecutor**（有三个子类，**SIMPLE**、**RESUE**、**BATCH**）的子类通过**CachingExecutor**进行装饰（装饰器模式），所以调**Executor.query()**方法时会先调**CachingExecutor**的**query()**方法，然后再去调**BaseExecutor**的**query()**方法。在**CachingExecutor**中会查询二级缓存，而一级缓存的查询是在**BaseExecutor**执行的（这就是mybatis先查二级缓存在查一级缓存的原因）

statementHandler模块

parameterHandler模块

ResultSetHandler模块

## 十、插件 (<https://github.com/pagehelper/Mybatis-PageHelper/blob/master/wikis/zh/Interceptor.md>)

1. 原理：实现**Interceptor**接口进行插件开发，如下图：



## 2. mybatis专门提供了生成被拦截类代理对象的工具类Plugin:

```
/*
 * public class Plugin implements InvocationHandler {
 * //封装的真正提供服务的对象
 * private final Object target; ← 被拦截的对象
 * //自定义的拦截器
 * private final Interceptor interceptor; ← 自定义的拦截器
 * //解析@Intercepts注解得到的Signature信息
 * private final Map<Class<?>, Set<Method>> signatureMap; ← 拦截器上边必须要加注解@Intercepts，注解里边配置了具体拦截那个类的那些方法
 *
 * private Plugin(Object target, Interceptor interceptor, Map<Class<?>, Set<Method>> signatureMap) {
 * this.target = target;
 * this.interceptor = interceptor;
 * this.signatureMap = signatureMap;
 * }
 *
 * //静态方法，用于帮助Interceptor生成动态代理
 * public static Object wrap(Object target, Interceptor interceptor) {
 * //解析Interceptors注解得到的signature信息
 * Map<Class<?>, Set<Method>> signatureMap = getSignatureMap(interceptor);
 * Class<?> type = target.getClass(); //获取目标对象的类型
 * Class<?>[] interfaces = getAllInterfaces(type, signatureMap); //获取目标对象实现的接口（拦截器可以拦截4大对象实现的接口）
 * if (interfaces.length > 0) {
 * //使用jdk的方式创建动态代理
 * return Proxy.newProxyInstance(
 * type.getClassLoader(),
 * interfaces,
 * new Plugin(target, interceptor, signatureMap));
 * }
 * return target;
 * }
 *
 * @Override
 * public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
 * try {
 * //获取当前接口可以被拦截的方法
 * Set<Method> methods = signatureMap.get(method.getDeclaringClass()); ← 获取被拦截的方法
 * if (methods != null && methods.contains(method)) { //如果当前方法在签名中需要被拦截，则调用interceptor.intercept方法进行拦截处理
 * return interceptor.intercept(new Invocation(target, method, args));
 * }
 * //如果当前方法不需要被拦截，则调用对象自身的方法
 * return method.invoke(target, args);
 * } catch (Exception e) {
 * throw ExceptionUtil.unwrapThrowable(e); ← 调用自定义拦截器的intercept方法，并将target（被拦截的类）、method（具体的方法）、args（参数）包装成Invocation对象传过去
 * }
 * }
 * }
```

## 3. Invocation类用来包装拦截目标信息:

```
/*
 * public class Invocation {
 * //被拦截的目标对象
 * private final Object target;
 * //具体拦截的方法
 * private final Method method;
 * //方法参数
 * private final Object[] args;
 *
 * public Invocation(Object target, Method method, Object[] args) {
 * this.target = target;
 * this.method = method;
 * this.args = args;
 * }
 *
 * public Object getTarget() { ← 返回被拦截的对象
 * return target;
 * }
 *
 * public Method getMethod() { ← 返回被拦截的具体方法
 * return method;
 * }
 *
 * public Object[] getArgs() { ← 返回参数
 * return args;
 * }
 *
 * public Object proceed() throws InvocationTargetException, IllegalAccessException {
 * return method.invoke(target, args); ← proceed()方法中实际是通过反射去调用目标对象的需要增强的具体方法
 * }
 * }
```

## 4. 自己实现一个拦截器，拦截StatementHandler的query()方法，判断sql查询时间是否超过配置的阈值，如果超过了则记录为慢查询：

```

<plugins> 在mybatis的配置文件中配置自定义插件

 <plugin interceptor="com.enjoylearning.mybatis.Interceptors.ThresholdInterceptor">
 <property name="threshold" value="10"/> ← 配置属性值
 </plugin>
 <!-- <plugin interceptor="com.github.pagehelper.PageInterceptor">
 <property name="pageSizeZero" value="true" />
 </plugin> -->
</plugins>

自定义插件必须如此注解 拦截的目标类型 拦截的具体方法 方法中参数的类型
@Intercepts({ @Signature(type=StatementHandler.class, method="query", args={Statement.class, ResultHandler.class})})
// @Signature(type=StatementHandler.class, method="query", args={MappedStatement.class, Object.class, RowBounds.class, ResultHandler.class})
})
public class ThresholdInterceptor implements Interceptor { ← 实现Interceptor接口

 private long threshold;

 @Override
 public Object intercept(Invocation invocation) throws Throwable {
 long begin = System.currentTimeMillis();
 Object ret = invocation.proceed(); ← 调用被拦截的真正方法，此处调用的是StatementHandler的query()方法
 long end=System.currentTimeMillis();
 long runtime = end - begin;
 if(runtime>threshold){
 Object[] args = invocation.getArgs(); ← 因为StatementHandle的query()方法的第一个参数是Statement，此处可以获取它
 Statement stat = (Statement) args[0];
 MetaObject metaObjectStat = SystemMetaObject.forObject(stat);
 PreparedStatementLogger statementLogger = (PreparedStatementLogger)metaObjectStat.getValue(name: "h");
 Statement statement = statementLogger.getPreparedStatement();
 System.out.println("sql语句: "+statement.toString()+"执行时间为: "+runtime+"毫秒，已经超过阈值！");
 }
 return ret;
 }
 @Override
 public Object plugin(Object target) {
 return Plugin.wrap(target, interceptor: this); ← 通过Plugin生成代理对象
 }
 @Override
 public void setProperties(Properties properties) {
 this.threshold = Long.valueOf(properties.getProperty("threshold")); ← 获取配置的参数
 }
}

```

## 5. mybatis加载插件过程：

- 配置拦截器插件：1、在Mybatis的xml中配置；2、在spring配置文件中配置。**
- mybatis解析xml的plugins节点并实例化插件对象按配置顺序添加到configuration对象的interceptorChain中，interceptorChain中维护了一个ArrayList来保存插件对象：**

```

private void parseConfiguration(XNode root) {
 try {
 //issue #117 read properties first
 //解析properties>节点
 propertiesElement(root.evalNode("properties"));
 //解析<settings>节点
 Properties settings = settingsAsProperties(root.evalNode("settings"));
 loadCustomVfs(settings);
 //解析<typeAliases>节点
 typeAliasesElement(root.evalNode("typeAliases"));
 //解析<plugins>节点
 pluginElement(root.evalNode("plugins")); ← 解析xml的plugins节点
 }
}

private void pluginElement(XNode parent) throws Exception {
 if (parent != null) {
 //遍历所有的插件配置
 for (XNode child : parent.getChildren()) {
 //获取插件的类名
 String interceptor = child.getStringAttribute(name: "interceptor");
 //获取插件的配置
 Properties properties = child.getChildrenAsProperties();
 //实例化插件对象
 Interceptor interceptorInstance = (Interceptor) resolveClass(interceptor).newInstance(); ← 通过反射获取插件实例
 //设置插件属性
 interceptorInstance.setProperties(properties); ← 设置插件属性
 //将插件添加到configuration对象，底层使用List保存所有的插件并记录顺序
 configuration.addInterceptor(interceptorInstance); ← 按插件配置顺序将插件对象保存到configuration的interceptorChain中，interceptorChain中维护了一个ArrayList
 }
 }
}

```

## 6. 插件能用插件拦截的接口和方法如下：

mybatis中能使用插件进行拦截的接口和方法如下： 只能拦截这四大组件

- Executor (update、query、flushStatement、commit、rollback、getTransaction、close、isClose)
- StatementHandler (prepare、paramterize、batch、update、query)
- ParameterHandler (getParameterObject、setParameters)
- ResultSetHandler (handleResultSets、handleCursorResultSets、handleOutputParameters)

## 7. 插件只能拦截Executor、StatementHandler、ParameterHandler、ResultHandler四个组件是因为这四个组件的对象都是通过Configuration类提供的newExecutor ()、newParameterHandler ()、newResultHandler ()、newStatementHandler () 方法创建的，而在这四个方法中又调用了interceptorChain.pluginAll () 方法对相应用对象进行了增强。

## 8. 插件代理对象生成过程 (Plugin类对目标对象进行增强)：

### a. 调用链：

Configuration → InterceptorChain.pluginAll() → Interceptor.plugin() → Plugin.wrap():

```
public StatementHandler newStatementHandler(Executor executor, MappedStatement mappedStatement, Object parameterObject) {
 // 创建RoutingStatementHandler对象，实际通过StatementType来指定真实的StatementHandler来实现
 StatementHandler statementHandler = new RoutingStatementHandler(executor, mappedStatement, parameterObject);
 statementHandler = (StatementHandler) interceptorChain.pluginAll(statementHandler);
 return statementHandler;
}

public Object pluginAll(Object target) {
 for (Interceptor interceptor : interceptors) {
 target = interceptor.plugin(target); ← 遍历插件列表，对目标对象进行增强
 }
 return target; ← 在拦截器的plugin()方法中调用Piugin.wrap()方法对目标对象进行增强
}

@Override
public Object plugin(Object target) {
 return Plugin.wrap(target, interceptor: this); ← 自定义拦截器中调用wrap()方法生成代理对象
}

// 静态方法，用于帮助Interceptor生成动态代理
public static Object wrap(Object target, Interceptor interceptor) {
 // 解析@Intercepts注解得到的signature信息
 Map<Class<?>, Set<Method>> signatureMap = getSignatureMap(interceptor);
 Class<?> type = target.getClass(); // 获取目标对象的类型
 Class<?>[] interfaces = getAllInterfaces(type, signatureMap); // 获取目标对象实现的接口（拦截器可以拦截4大对象实现的接口）
 if (interfaces.length > 0) {
 // 使用jdk的方式创建动态代理
 return Proxy.newProxyInstance(
 type.getClassLoader(),
 interfaces,
 new Plugin(target, interceptor, signatureMap));
 }
 return target;
}
```

## b. Plugin实现了InvocationHandler接口，对目标对象进行增强，invoke()方法如下：

```
@Override
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
 try {
 // 获取当前接口可以被拦截的方法
 Set<Method> methods = signatureMap.get(method.getDeclaringClass());
 if (methods != null && methods.contains(method)) { // 如果当前方法需要拦截，则调用interceptor的
 return interceptor.intercept(new Invocation(target, method, args)); // 判断该方法是否需要拦截，是则调用interceptor的
 } else { // 不需要拦截则通过反射直接调用目标对象自身方法
 return method.invoke(target, args);
 }
 } catch (Exception e) {
 throw ExceptionUtil.unwrapThrowable(e);
 }
}
```

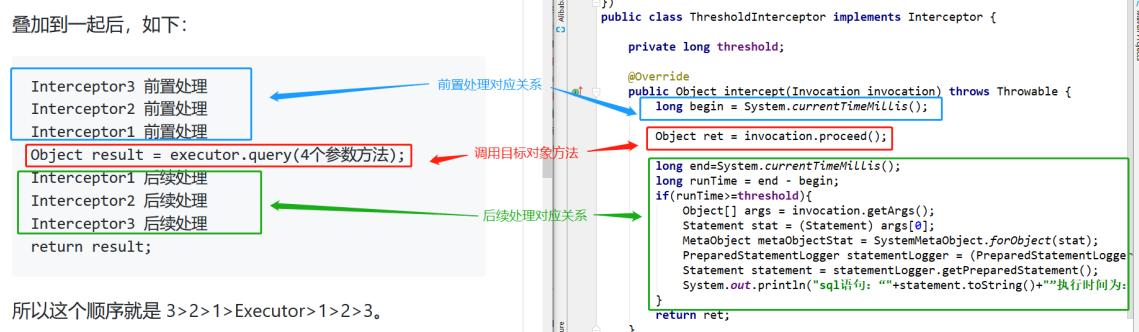
## 9. 当我们配置拦截器的顺序是1, 2, 3。在这里也会按照1, 2, 3的顺序被层层代理，代理后的结构如下：

```
Interceptor3: {
 Interceptor2: {
 Interceptor1: {
 target: Executor
 }
 }
}
```

此处Interceptor1为目标对象生成代理对象，而  
Interceptor2是为Interceptor1生成的代理对象在生成代理  
对象，Interceptor3是给Interceptor2生成的代理对象生成  
代理对象

## 10. 被层层代理的对象最终是按照3>2>1>Executor>1>2>3的顺序去执行的。以 拦截Executor的query()方法为例，Interceptor1对Executor对象代理之后生成的 代理对象中有query()方法，而通过看代理对象的class文件可以知道query()方法中 调用的是InvocationHandler的invoke()方法，在invoke()方法中我们才会通过通 过反射真正调用Executor目标对象的query()方法。而Interceptor2是对 Interceptor1生成的代理对象进行增强，即Interceptor2生成的代理对象中 query()方法实际调用的是Interceptor1生成的代理对象的query()方法，那么同理， Interceptor3生成的代理对象中的query()方法实际调用的是Interceptor2生成的 代理对象的query()方法。因此当我们手动调用Interceptor3生成的代理对象的 query()方法时，它会去调用Interceptor2生成的代理对象的query()方法，而它又 去调用Interceptor1生成的代理对象的query()方法，最终Interceptor1代理对象 去调用目标对象的query()方法。**study工程中的tireProxy包下代码还原了插件对目 标对象层层代理的过程，可以运行细看。**

## 11. 被多个插件层层代理的目标对象最终执行结果以及里边描述的前置处理、后续处 理与插件中代码的对应关系如下图：



## 十一、分页插件PageInterceptor:

- 如何使用：<https://github.com/pagehelper/Mybatis-PageHelper/blob/master/wikis/zh/HowToUse.md>
- 注意事项：<https://github.com/pagehelper/Mybatis-PageHelper/blob/master/wikis/zh/Important.md>
- 分页插件会解析查询语句并根据不同数据库生成不同的查询总条数的语句来得到totalnum。

## 核心流程三大阶段

### 一、初始化阶段

([https://blog.csdn.net/weixin\\_37139197/article/details/82717964](https://blog.csdn.net/weixin_37139197/article/details/82717964), 共三篇)

- 所谓初始化阶段其实是去创建Configuration对象，并将所有的xml配置加载到此对象中。Configuration是单例的，而且它的生命周期是应用级别的。
- Configuration中的属性都能与配置文件中的标签一一对应（即xml解析后将配置的值存储到Configuration属性中），并且很多属性在Configuration中设置了默认值，部分截图如下：

```

public class Configuration {
 protected Environment environment;
 /* 是否启用行内嵌套语句 */
 protected boolean safeRowBoundsEnabled;
 protected boolean safeResultHandlerEnabled = true;
 /* 是否启用数据驱动Column自动映射到Java类中的驼峰命名的属性 */
 protected boolean mapUnderscoreToCamelcase;
 /* 当对象使用延迟加载时 属性的加载取决于能被引用到的那些延迟属性,否则,按需加载 */
 protected boolean aggressiveLazyLoading;
 /* 是否允许单条SQL返回多个数据集 (取决于驱动的兼容性) default:true */
 protected boolean multipleResultSetsEnabled = true;
 /* 允许JDBC生成主键,需要驱动器支持.如果设为了true,这个设置将强制使用被protected boolean useGeneratedKeys;
 * 使用列标签代替列名.不同的驱动在这方面会有不同的表现,具体可参考相关帮助 */
 protected boolean useColumnLabel = true;
 /* 配置全局性的cache开关,默认为true */
 protected boolean cacheEnabled = true;
 protected boolean callSettersOnNulls;
 protected boolean useActualParamName = true;
 protected boolean returnInstanceForEmptyRow;
 /* 日志打印所有的前缀 */
 protected String logPrefix;
}

```

```

<!-- 参数设置 -->
<!-- 允许在嵌套语句中使用分页 -->
<setting name="safeRowBoundsEnabled" value="false" />
<!-- 是否开启自动驼峰命名规则 (camel case) 映射,即从经典数据库列名 A_CO
 的类似映射。-->
<setting name="mapUnderscoreToCamelcase" value="false" />
<!-- 全局启用或禁用延迟加载。当禁用时,所有关联对象都会即时加载 -->
<setting name="cacheEnabled" value="true" />
<!-- 允许对全局的映射器启用或禁用缓存 -->
<setting name="lazyLoadingEnabled" value="true" />
<!-- 当启用时,有延迟加载属性的对象在被调用时将会完全加载任意属性.否则,
 允许或不允许多种结果集从一个单独的语句中返回 (需要适合的驱动) -->
<setting name="aggressiveLazyLoading" value="true" />
<setting name="multipleResultSetsEnabled" value="true" />
<!-- 使用列标签代替列名.不同的驱动在这方面表现不同.参考驱动文档或充分测试
 使用 -->
<setting name="useColumnLabel" value="true" />
<!-- 允许JDBC支持生成的键.需要适合的驱动.如果设置为true则这个设置强制生效
 -->
<setting name="useGeneratedKeys" value="true" />
<!-- 指定MyBatis如何处理映射到字典属性. PARTIAL 只会自动映射简单,没有
 (默认值) WARNING: 警告日志形式的详细信息 FAILING: 映射失败,抛出异常 -->
<setting name="autoMappingUnknownColumnBehavior" value="WARNING" />
<!-- 配置默认的执行器. SIMPLE执行器没什么特别之处. REUSE执行器重用预处理
 语句 -->
<setting name="defaultExecutorType" value="SIMPLE" />
<!-- 设置超时时间.它决定驱动等待一个数据库响应的时间 -->
<setting name="defaultStatementTimeout" value="25000" />
<!-- 没设置查询返回值数量,可以被查詢函数覆盖 -->

```

- mybatis的核心配置文件由XMLConfigBuilder解析加载，流程如下：

- 配置文件输入流交给

## SqlSessionFactoryBuilder().build(inputStream):

```

String resource = "mybatis-config.xml"; ← 配置文件路径
InputStream inputStream = Resources.getResourceAsStream(resource);
// 1. 读取mybatis配置文件创建SqlSessionFactory
sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
inputStream.close();

```

- 实例化XMLConfigBuilder并调用parse()方法：

```

public SqlSessionFactory build(InputStream inputStream, String environment, Properties properties) {
 try {
 XMLConfigBuilder parser = new XMLConfigBuilder(inputStream, environment, properties); ← 实例化XMLConfigBuilder
 return build(parser.parse()); ← 调用parse()方法
 } catch (Exception e) {
 }
}

```

- 如果未被解析则调用parseConfiguration()方法进行解

析：

```

public Configuration parse() {
 if (parsed) { ← 判断是否已经解析过, 防止多次解析
 throw new BuilderException("Each XMLConfigBuilder can only be used once.");
 }
 parsed = true;
 parseConfiguration(parser.evalNode(expression: "/configuration")); ← 未被解析过则开始解析
 return configuration;
}

```

- parseConfiguration()方法详解：

```

private void parseConfiguration(XNode root) {
 try {
 //issue #117 read properties first
 //sf
 propertiesElement(root.evalNode("properties"));
 Properties settings = settingsAsProperties(root.evalNode("settings"));
 loadCustomVfs(settings);
 typeAliasesElement(root.evalNode("typeAliases"));
 pluginElement(root.evalNode("plugins"));
 objectFactoryElement(root.evalNode("objectFactory"));
 objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));
 reflectorFactoryElement(root.evalNode("reflectorFactory"));
 settingsElement(settings);
 // read it after objectFactory and objectWrapperFactory issue #631
 environmentsElement(root.evalNode("environments"));
 databaseIdProviderElement(root.evalNode("databaseIdProvider"));
 typeHandlerElement(root.evalNode("typeHandlers"));
 mapperElement(root.evalNode("mappers"));
 } catch (Exception e) {
 throw new BuilderException("Error parsing SQL Mapper Configuration. Cause: " + e, e);
 }
}

```

将不同的标签解析并赋值到 Configuration的不同属性中

此处就是使用的建造者模式给 Configuration的属性赋值

- **mapper.xml由XMLMapperBuilder解析加载，流程如下：**

- **从XMLConfigBuilder的parseConfiguration()方法到 mapperElement()方法进入XMLMapperBuilder解析：**

```

//解析<mappers>节点
mapperElement(root.evalNode("mappers")); 在parseConfiguration()中
} catch (Exception e) {
 ErrorContext.instance().resource(url);
 InputStream inputStream = Resources.getUrlAsStream(url); //加载mapper文件
 //实例化XMLMapperBuilder解析mapper映射文件
 XMLMapperBuilder mapperParser = new XMLMapperBuilder(inputStream, configuration, url, configuration.getSqlFragments());
 mapperParser.parse(); 进入解析
} else if (resource == null && url == null && mapperClass != null) { //如果class不为空
 Class<?> mapperInterface = Resources.classForName(mapperClass); //加载class对象
}

```

- **XMLMapperBuilder重点分析缓存、resultMap、sql的解析过程， XMLMapperBuilder用来解析读取配置文件中的属性值，对象建造过程一般都是交给MapperBuilderAssistant去做：**

```

private void configurationElement(XNode context) {
 try {
 //获取mapper节点的namespace属性
 String namespace = context.getStringAttribute("name", "namespace");
 if (namespace == null || namespace.equals("")) {
 throw new BuilderException("Mapper's namespace cannot be empty");
 }
 //设置builderAssistant的namespace属性
 builderAssistant.setCurrentNamespace(namespace);
 //解析cache-ref节点
 cacheElement(context.evalNode("cache-ref"));
 //重点分析：解析cache节点-----1-----
 cacheElement(context.evalNode("cache"));
 //解析parameterMap节点（参数映射）
 parameterMapElement(context.evalNodes("expression: /mapper/parameterMap"));
 //重点分析：解析resultMap节点（基于数据结果去理解）-----2-----
 resultMapElements(context.evalNodes("expression: /mapper/resultMap"));
 //解析sql节点
 sqlElement(context.evalNodes("expression: /mapper/sql"));
 //重点分析：解析select、insert、update、delete节点 -----3-----
 buildStatementFromContext(context.evalNodes("expression: select|insert|update|delete"));
 } catch (Exception e) {
 throw new BuilderException("Error parsing Mapper XML. The XML location is '" + resource + "'. Cause: " + e, e);
 }
}

```

重点分析缓存节点、 resultMap节点、 select、 insert、 update、 delete节点

- **缓存解析过程：**

```

 private void cacheElement(XNode context) throws Exception {
 if (context != null) {
 //根据cache节点的type属性，默認為PERPETUAL
 String type = context.getStringAttribute("name: "type", def: "PERPETUAL");
 //根据type对应的cache接口实现
 Class<? extends Cache> typeClass = typeAliasRegistry.resolveAlias(type);
 //根据eviction属性，直接用的淘汰策略，默認為LRU
 String eviction = context.getStringAttribute("name: "eviction", def: "LRU");
 //根据flushInterval属性，既缓存的刷新周期
 Long flushInterval = context.getLongAttribute("name: "flushInterval");
 //读取size属性，既缓存的容量大小
 Integer size = context.getIntAttribute("name: "size");
 //读取readOnly属性，既缓存的是只读
 boolean readOnly = !context.getBooleanAttribute("name: "readOnly", def: false);
 //读取blocking属性，既缓存的是否阻塞
 boolean blocking = context.getBooleanAttribute("name: "blocking", def: false);
 Properties props = context.getChildrenAsProperties();
 //通过builderAssistant创建缓存对象，并添加至configuration
 builderAssistant.useNewCache(typeClass, evictionClass, flushInterval, size, readOnly, blocking, props);
 }
 }

 //通过builderAssistant创建缓存对象，并添加至configuration
 public Cache useNewCache(Class<? extends Cache> typeClass,
 Class<? extends Cache> evictionClass,
 Long flushInterval,
 Integer size,
 boolean readOnly,
 boolean blocking,
 Properties props) {
 //经典的建造者模式 创建一个cache对象
 Cache cache = new CacheBuilder(currentNamespace)
 .implementation(valueOrDefault(typeClass, PerpetualCache.class))
 .addDecorator(valueOrDefault(evictionClass, LruCache.class))
 .clearInterval(flushInterval)
 .size(size)
 .readwrite(readOnly)
 .blocking(blocking)
 .properties(props)
 .build();
 //将缓存添加至configuration，注意二级缓存以命名空间为单位进行划分
 configuration.addCache(cache);
 currentCache = cache;
 return cache;
 }

 public Cache build() {
 //设置缓存的主要实现类为PerpetualCache
 setDefaultImplementations();
 //通过反射实例化PerpetualCache对象
 Cache cache = newBaseCacheInstance(implementation, id);
 setCacheProperties(); //根据cache节点下的<property>信息，初始化cache
 // issue #352, do not apply decorators to custom caches

 if (PerpetualCache.class.equals(cache.getClass())) { //如果cache是PerpetualCache的实现，则为其添加标准的装饰器
 for (Class<? extends Cache> decorator : decorators) { //为cache对象添加装饰器，这里主要处理缓存清空策略的装饰器
 cache = newCacheDecoratorInstance(decorator, cache);
 setCacheProperties(cache);
 }
 }

 //通过一些属性为cache对象添加装饰器
 cache = setStandardDecorators(cache);
 } else if (!LoggingCache.class.isAssignableFrom(cache.getClass())) {
 //如果cache不是PerpetualCache的实现，则为其添加日志的能力
 cache = new LoggingCache(cache);
 }
 return cache;
 }

 private Cache setStandardDecorators(Cache cache) {
 try {
 MetaObject metaCache = SystemMetaObject.forObject(cache);
 if (size != null && metaCache.hasSetter("name: "size")) {
 metaCache.setValue("name: "size", size);
 }
 if (clearInterval != null) {
 cache = new ScheduledCache(cache); //如果设置了定时清除，则使用ScheduledCache进行装饰
 ((ScheduledCache) cache).setClearInterval(clearInterval);
 }
 if (readWrite) {
 cache = new SerializedCache(cache);
 }
 cache = new LoggingCache(cache);
 cache = new SynchronizedCache(cache);
 if (blocking) {
 cache = new BlockingCache(cache); //如果设置了blocking属性，则使用BlockingCache进行装饰
 }
 return cache;
 } catch (Exception e) {
 throw new CacheException("Error building standard cache decorators. Cause: " + e, e);
 }
 }
}

```

## ○ <resultMap/>标签解析后通过ResultMap、

ResultMapping来存储。详细解析流程代码中有注解：

## ■ ResultMapping用来存储<resultMap/>中每一行的属性值:

```

/*
public class ResultMapping {
 private Configuration configuration; //引用的configuration对象
 private String property; //对应节点的property属性
 private String column; //对应节点的column属性
 private Class<?> javaType; //对应节点的javaType属性
 private JdbcType jdbcType; //对应节点的jdbcType属性
 private TypeHandler<?> typeHandler; //对应节点的typeHandler属性
 private String nestedResultId; //对应节点的resultId属性，嵌套结果时使用
 private String resultId; //对应节点的selectId，嵌套查询时使用
 private String sqlMap; //对应节点的sqlMap属性
 private String columnPrefix; //对应节点的columnPrefix属性
 private List<ResultFlag> flags; //标志，或者constructor
 private List<ResultMapping> composites;
 private String resultSet; //对应节点的结果集属性
 private String foreignColumn; //对应节点的foreignColumn属性
 private boolean lazy; //对应节点的fetchType属性，是否是延迟加载
}

```

## ■ ResultMap用来存储整个<resultMap/>解析后的值，即一个ResultMap中存储多个ResultMapping:

```

/*
public class ResultMap {
 private Configuration configuration; //configuration对象
 private String id; //resultMap的id属性
 private Class<?> type; //resultMap的type属性
 private List<ResultMapping> resultMappings; //除discriminator节点之外的映射
 private List<ResultMapping> idResultMappings; //记录或构造器的id
 private List<ResultMapping> constructorResultMappings; //记录<constructor>的
 private List<ResultMapping> propertyResultMappings; //记录<constructor>的
 private Set<String> mappedColumns; //记录所有有映射关系的columns字段
 private Map<String, Map<String, Properties>> mappedProperties; //记录所有有映射关系的property字段
 private Discriminator discriminator; //是否包含discriminator节点
 private boolean hasNestedResultMaps; //是否嵌套结果映射
 private boolean hasNestedQueries; //是否嵌套查询
 private Boolean autoMapping; //是否开启了自动映射
}

```

- select、insert、delete、update等涉及sql的节点交由XMLStatementBuilder去解析，由MappedStatement来存储解析后的结果，sql语句会被解析成SqlSource对象，经过解析SqlSource包含的语句最终仅仅包含？占位符，可以直接提交给数据库执行：

```

//处理所有的sql语句节点并注册至configuration对象
private void buildStatementFromContext(List<XNode> list, String requiredDatabaseId) {
 for (XNode context : list) {
 //创建XMLStatementBuilder,并注册到SqlSessionFactory
 final XMLStatementBuilder statementParser = new XMLStatementBuilder(configuration, builderAssistant, context, requiredDatabaseId);
 try {
 //解析sql语句节点
 statementParser.parseStatementNode();
 //sql语句由XMLStatementBuilder解析
 } catch (IncompleteElementException e) {
 configuration.addIncompleteStatement(statementParser);
 }
 }
}

//通过builderAssistant实例化MappedStatement，并注册至configuration对象
builderAssistant.addMappedStatement(id, sqlSource, statementType, sqlCommandType,
 fetchSize, timeout, parameterMap, parameterTypeClass, resultMap, resultTypeClass,
 resultSetTypeEnum, flushCache, useCache, resultOrdered,
 keyGenerator, keyProperty, keyColumn, databaseId, langDriver, resultSets);

```

XMLStatementBuilder将标签属性解析出来以后交由MapperBuilderAssistant去构建MappedStatement对象，存储解析出来的属性值

```

/*
public final class MappedStatement {
 private String resource; // 节点的完整的id属性，包括命名空间
 private Configuration configuration;
 private String id; // 节点的id属性
 private Integer fetchSize; // 节点的fetchSize属性，查询数据的条数
 private Integer timeout; // 节点的timeout属性，超时时间
 private StatementType statementType; // 节点的statementType属性，默认值：StatementType.PREPARED；疑问？
 private ResultsetType resultsetType; // 节点的resultsetType属性，jdbc知识
 private SqlSource sqlsource; // 节点中sql语句信息
 private Cache cache; // 对应的二级缓存
 private ParameterMap parameterMap; // 已废弃
 private List<ResultMap> resultMaps; // 节点的resultMaps 属性
 private boolean flushCacheRequired; // 节点的flushCache 属性是否刷新缓存
 private boolean useCache; // 节点的useCache 属性是否使用二级缓存
 private boolean resultOrdered;
 private SqlCommandType sqlCommandType; // sql语句的类型，包括：INSERT, UPDATE, DELETE, SELECT
 private KeyGenerator keyGenerator; // 节点keyGenerator属性
 private String[] keyProperties;
 private String[] keyColumns;
 private boolean hasNestedResultMaps; // 是否有嵌套resultMap
 private String databaseId;
 private Log statementLog;
 private LanguageDriver lang;
 private String[] resultSets; // 多结果集使用
}

```

```

* @author Clinton Begin
*/
public interface SqlSource {
 BoundSql getBoundSql(Object parameterObject);
}

```

解析出来的sql语句由BoundSql封装

- XMLConfigBuilder、XMLMapperBuilder、XMLStatementBuilder工作流程如出一辙，都是解析相应的标签，然后将属性值填充到相应的数据结构中并将数据结构赋值到Configuration的对应属性上边。看他们三个的工作流程前先需要了解对应的数据结构，即标签属性与数据结构是一一对应的，然后再看解析流程就比较顺畅了。需要注意的一点就是XMLMapperBuilder一般都是只解析标签，构建数据结构对象且赋值的工作是交给MapperBuilderAssistant去完成的（建造者模式）。

疑问：原生mybatis在Configuration的MapperRegistry属性中缓存的是MapperProxyFactory对象，我们每次getMapper()都会获取到新的mapper代理对象，但是为什么与spring整合后可以将mapper交给spring容器（单例的），它是如何保证线程安全的。

做分页的目的：

- 1、避免全表扫描。快速查询
- 2、查询的数据集比较小，降低占用网络带宽

### 3、降低内存占用

mybatis自带的**RowBounds**分页功能它会将所有数据查询出来缓存到内存中。然后维护一个偏移量给你返回指定的多少条。即从数据库查询的数据量很大，只不给客户端返回的是部分数据。即**RowBounds**是一个假分页

mybatis四大核心：Executor、StatementHandler、ParameterHandler、ResultHandler。其中StatementHandler和ResultHandler就是对JDBC中的Statement、ResultSet的封装

**SqlSessionFactoryBean**详解，**SqlSessionFactoryBean**通过实现**FactoryBean**接口将**SqlSessionFactory**注入到IOC容器中了。

MapperFactoryBean详解

实际是通过MapperFactoryBean注入的

本质注入的MapperFactoryBean

mapperFactoryBean实现了FactoryBean接口，在getObject()方法中还是调用的SqlSession的getMapper()方法，继而走的还是原生mybatis的那个流程，即获取新的Mapper接口的代理对象(每次调用都会新建)

通过FactoryBean接口注入到容器中的bean获取的时候实际调用的是getObject()方法。

```
public class MapperScannerConfigurer implements BeanDefinitionRegistryPostProcessor, InitializingBean, ApplicationContextAware, BeanFactoryAware, SqlSessionAware, SqlSessionFactoryBeanAware {
 private String basePackage;//用于指定要扫描的包
 private boolean addToConfig = true;
 private SqlSessionFactory sqlSessionFactory;
 private SqlSessionTemplate sqlSessionTemplate;
 private String sqlSessionFactoryBeanName;
 private String sqlSessionTemplateBeanName;

 private Class<? extends Annotation> annotationClass;//mapper接口上有指定的annotation才会被扫描
 private Class<?> markerInterface;//mapper接口继承与指定的接口才会被扫描

 private ApplicationContext applicationContext;//容器上下文
 private String beanName;
 private boolean processPropertyPlaceHolders;
 ...
}
```

当我们的项目有特殊需求时，比如同一个包下有很多mapper而只有其中一部分是提供给mybatis用的，那么此时我们就可以通过这两个配置来指定  
1、在需要扫描的mapper上边加上相同的注解(可以用自定义的注解)  
2、让mapper接口继承指定的接口

mybatis-spring是把mapper接口转换成MapperFactoryBean然后注入到容器中的。这个操作是通过MapperScannerConfigurer实现BeanDefinitionRegistryPostProcessor接口，在postProcessBeanDefinitionRegistry方法中做的。

MapperFactoryBean实现了FactoryBean接口，它的getObject()方法中

通过Sqlsession获得mapper代理对象之后，最终代理对象中调用的还是sqlsession的selectOne()、selectList()等方法。因为Ibatis的编程模式是直接通过sqlsession去查数据库，而Mybatis是通过mapper代理对象去调sqlsession中的查询方法。

**为什么每次都要创建新的mapper代理对象？因为mapper代理对象是线程不安全的**

**为什么mapper代理对象线程不安全？因为MapperProxy中维护了sqlsession。sqlsession是线程不安全的。**

**为什么sqlSession是线程不安全的？因为sqlsession中的executor维护了一级缓存，同时多个线程共用sqlsession时A线程可能会提交B线程未完成的事务。**

## Mybatis-spring中SqlSessionTemplate是如何使用DefaultSqlSession的：

(<https://blog.csdn.net/xlgen157387/article/details/79438676>)

hivegl中的<mybatis:scan/>标签也可以通过使用@MapperScan注解代替，因为@MapperScan注解中有属性可以指定使用哪个SqlSessionFactoryBean。

将mapper代理对象注册到IOC容器中有三种方式：

1. 通过@Mapper注解或者MapperFactoryBean注入，这两种方式的缺点是需要在每个mapper接口上边加@Mapper注解或者为每个mapper接口配置一个MapperFactoryBean，比较麻烦。同时@Mapper注解无法指定SqlSessionFactory对象，如下图：

```
@Documented
@Inherited
@Retention(RUNTIME)
@Target({ TYPE, METHOD, FIELD, PARAMETER })
public @interface Mapper {
 // Interface Mapper
}
```

@Mapper注解没有提供属性来指定SqlSessionFactory

```

@Mapper
public interface THealthReportFemaleMapper {
 /**
 <bean class="org.mybatis.spring.mapper.MapperFactoryBean">
 <property name="mapperInterface" value="com.enjoylearning.mybatis.mapper.TRoleMapper"/>
 <property name="sqlSessionFactory" ref="sqlSessionFactory"/>
 </bean> ← 指定SqlSessionFactory

```

2. @MapperScan注解方式（通过MapperScannerRegistrar注解连接器）：  
**MapperScannerRegistrar**最终还是通过**ClassPathMapperScanner**注入**mapper**接口代理对象，如下图：

```

指定mapper包路径 ←
@MapperScan(basePackages = {}, sqlSessionFactoryRef = "") ← 指定SqlSessionFactory
* @version Id
*/
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
@Import(MapperScannerRegistrar.class) ← @MapperScan注解最终通过
public @interface MapperScan { ← MapperScannerRegister类完成mapper接
 口代理对象注册功能

```

3. 通过配置连接 **MapperScannerConfigurer**注入，底层也是通过**ClassPathMapperScanner**实现的：

```


<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
 <property name="basePackage" value="com.enjoylearning.mybatis.mapper" /> ← 指定mapper包路径
 <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"/> ← 指定SqlSessionFactory
</bean>

```

当项目中有多个数据源时我们可以通过**@MapperScan**注解或者**MapperScannerConfigurer**这两种方式给不同的包下的**mapper**指定不同的**SqlSessionFactory**。