# Mybatis

## 1. 介绍

Mybatis是一个ORM框架。

> 1. ORM：Object Relationship Mapping。对象关系映射。其实就是可以把Java对象映射为关系型数据库表中的记录，也可以把关系型数据库表中的记录映射为Java对象。
>
>    本质上来说，其实就是去操作数据库。所以，**Mybatis本质上来说就是操作数据库的一个框架。**
>
> 2. 什么框架呢?
>
>    框架是一个半成品软件。框架开发了一些基础的功能，我们开发者一般可以在框架的基础之上进行二次开发，开发自己的功能，形成一个 独立的、完整的软件。
>
>    框架就好比简历模板，这个简历模板不是一个完整的简历，需要我们在这个模板的基础之上进行二次修改，填充我们自己的信息才能成为一个完整的简历。
>
>    框架的目的是为了提高开发的效率。常见的框架有：
>
>    SSM （SpringMVC | Spring | Mybatis）、Springboot、Dubbo...

Mybatis的出现提升了我们操作数据库的效率。

## 2. 入门

Mybatis官网

源码地址

### 导包

```xml
<!-- Mybatis -->
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.5.10</version>
</dependency>
```

### 配置

1. 配置Mybatis的核心配置文件

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!-- xml的文件头 -->

<!-- 约束文件 约束了configuration 这个标签中出现的标签的名字，顺序 -->
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">

<!-- 根标签-->
<configuration>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://localhost:3306/44th?
useSSL=false&amp;characterEncoding=utf8"/>
                <property name="username" value="root"/>
                <property name="password" value="123456"/>
            </dataSource>
        </environment>
    </environments>

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


</configuration>
```

2. 配置mapper映射文件

   mapper映射文件其实就是用来写 SQL语句的（管理SQL语句的）

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


<!-- namespace：命名空间，每一个mapper.xml的namespace是不一样的，要全局唯一-->
<mapper namespace="cskaoyan">



</mapper>
```

3. 把Mapper映射文件配置到主配置文件中来


4. 在mapper.xml中写SQL语句

```xml
<!-- id: 在这个文件中唯一 -->
<!-- namespace + id : 全局唯一，这个叫做SQL语句的坐标 -->
<delete id="deleteUserById">

    delete from user where id = #{id}

</delete>
```

## 使用

使用其实就是去执行我们刚刚写的SQL语句，那么如何执行呢?

```java
 public static void main(String[] args) throws IOException {

//        读取核心配置文件
//        String resource = "org/mybatis/example/mybatis-config.xml";
//        InputStream inputStream = Resources.getResourceAsStream(resource);
//
//        获取SqlSessionFactory的实例对象
//        SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

//        ClassLoader classLoader = Main.class.getClassLoader();
////        InputStream inputStream = classLoader.getResourceAsStream("mybatis-config.xml");


        // 1. 获取核心配置文件的文件流
        InputStream inputStream = Resources.getResourceAsStream("mybatis-config.xml");


        // 2. 获取SqlSessionFactory
        SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
        SqlSessionFactory sqlSessionFactory = builder.build(inputStream);

        // 3. 通过SqlSessionFactory 获取SqlSession
        // sqlSession其实就是Mybatis和 MySQL数据库之间的连接，就是之前的Connection对象的封装
        // 获取的SqlSession是不会自动提交事务的，如果需要自动提交事务，openSession(true);
        SqlSession sqlSession = sqlSessionFactory.openSession(true);

        // 4. 执行SQL语句
        int affectedRows = sqlSession.delete("cskaoyan.deleteUserById", 1004);

        System.out.println("affectedRows:" + affectedRows);


    }
```

```xml
<!-- namespace：命名空间，每一个mapper.xml的namespace是不一样的，要全局唯一-->
<mapper namespace="cskaoyan">


    <!-- id: 在这个文件中唯一 -->
```

```
    <!-- namespace + id : 全局唯一，这个叫做SQL语句的坐标  cskaoyan.deleteUserById-->
    <delete id="deleteUserById">

        delete from user where id = #{id}

    </delete>


    <!-- 增加 -->
    <insert id="insertUser">
        insert into user values (#{id},#{username},#{password},#{nickname})
    </insert>


    <update id="updateUserById">
        update user set username=#{username} where id = #{id}
    </update>



    <select id="selectUserById" resultType="com.cskaoyan.bean.User">
        select * from user where id = #{id}
    </select>


    <select id="selectAllUser" resultType="com.cskaoyan.bean.User">
        select * from user
    </select>



</mapper>
```

> 目前这个Mybatis入门案例的使用方式不是我们以后最常用的方式，还是有一些缺陷：
>
> - 我们之前说DBUtils的SQL语句和代码耦合在一起，不方便管理；目前使用了Mybatis之后，虽然SQL语句没有和代码耦合在一起了，但是现在代码和SQL语句的坐标耦合在一起了，不符合我们调用的逻辑
>
> 后续我们会学习Mybatis的动态代理，动态代理是自动去获取SQL语句的坐标。

## 3. 动态代理

Mybatis的动态代理是把接口中的方法和SQL语句绑定起来，后续执行语句，我们就不用写SQL语句的坐标了，可以直接通过执行接口中的方法来执行SQL。

### 导包

不需要导入额外的包

```
<!-- Mybatis -->
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.5.10</version>
</dependency>
```

### 配置

- 配置主配置文件
- 写一个接口
- 配置mapper.xml配置文件
- 在接口中声明方法
- 写方法对应的标签以及标签中的SQL语句

> 动态代理的使用有几个要求：
>
> 1. 接口的名字和它对应的mapper.xml配置文件的名字要一致
> 2. 接口和mapper.xml配置文件在编译之后在同一级路径
>
>    说明：以上的两个配置在某些使用场景中可以不遵守，是行业的规范，建议大家遵守
>
> 3. mapper.xml中的namespace的值必须和接口的全限定名称保持一致
> 4. mapper.xml配置文件中的id值和接口中的方法名要保持一致
> 5. 接口中的方法的参数和返回值要和标签中（resultType）的保持一致

### 使用

```
static SqlSession sqlSession;

static UserMapper userMapper;

static {
    sqlSession = MybatisUtils.getSqlSession();

    // 获得接口的实例对象 (代理对象, MapperProxy)
    userMapper = sqlSession.getMapper(UserMapper.class);
}


@Test
public void testSelectUserById(){

    // 通过代理对象执行对应的SQL语句
    User user = userMapper.selectUserById(2001);

    System.out.println(user);

}
```

## 4. 配置

主要是给大家介绍 `mybatis-config.xml` 这个核心配置文件。

### properties

properties主要是支持Mybatis在内部或者是外部去读取properties配置文件中的内容。

```
<!-- 1. 在内部声明配置-->
<!--<properties>-->
    <!--<property name="username" value="root"/>-->
    <!--<property name="password" value="123456"/>-->
<!--</properties>-->

<!-- 2. 在外部配置文件中声明配置-->
<properties resource="jdbc.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>
```

取值，${key} 来取值

可以再外部或者是内部配置

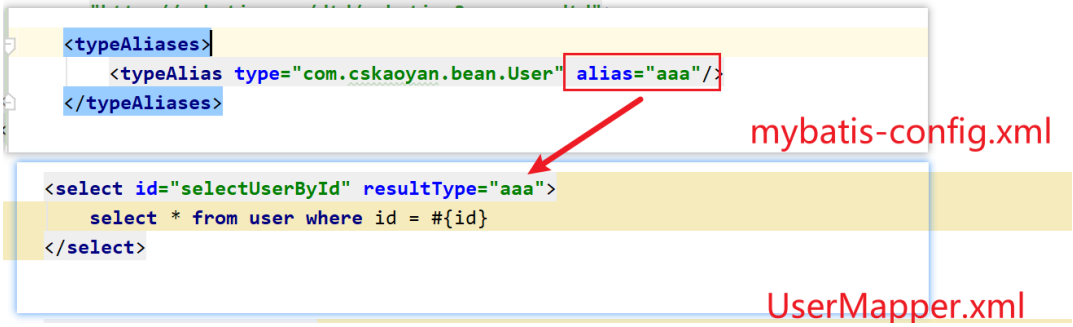
### settings

settings配置是Mybatis极为重要的调整设置，会改变Mybatis运行时的行为。

```
<settings>
    <!-- 日志的配置-->
    <setting name="logImpl" value="STDOUT_LOGGING"/>

    <!-- 其他的配置暂时不用配-->
    ...

</settings>
```

### TypeAlies

这个别名的配置。我们可以对一些Java类起别名，来避免在mapper.xml配置文件反复的用全限定名称，简化开发。







其实，Mybatis也对一些常见的基本类型、包装类型、集合类型、String等 有内置的别名。

| 别名 | 映射的类型 |
| --- | --- |
| _byte | byte |
| _char (since 3.5.10) | char |
| _character (since 3.5.10) | char |
| _long | long |
| _short | short |
| _int | int |
| _integer | int |
| _double | double |
| _float | float |
| _boolean | boolean |
| string | String |
| byte | Byte |
| char (since 3.5.10) | Character |
| character (since 3.5.10) | Character |
| long | Long |
| short | Short |
| int | Integer |
| integer | Integer |
| double | Double |
| float | Float |
| boolean | Boolean |
| date | Date |
| decimal | BigDecimal |
| bigdecimal | BigDecimal |
| biginteger | BigInteger |
| object | Object |
| date[] | Date[] |
| decimal[] | BigDecimal[] |
| bigdecimal[] | BigDecimal[] |
| biginteger[] | BigInteger[] |
| object[] | Object[] |
| map | Map |
| hashmap | HashMap |
| list | List |
| arraylist | ArrayList |
| collection | Collection |
| iterator | Iterator |

## TypeHandler

类型处理器。可以帮助Mybatis把Java类型转化为数据库类型，也可以把数据库类型转化为Java类型。

也可以自己定义类型处理器。

一般情况下使用默认的类型处理器就可以了。

## ObjectFactory

对象工厂。负责实例化对象

```
// 1. 找到对应的SQL语句

// 2. 预编译SQL语句

// 3. 传入参数

// 4. 执行SQL（产生结果集 ResultSet）

// 5. 解析结果集，把结果集的结果封装到对象中去（这里的对象由对象工厂来创建）

// 6. 返回对象
```

对象工厂在第五步开始工作，创建出适合的对象来接收结果。

## environments

```
<!-- 默认使用哪个环境 -->
<environments default="test">

    <environment id="test">
        <!-- 事务管理器
        MANAGED: 把事务交给外部的容器来管理
        JDBC: 把事务交给JDBC连接对象来管理，之前学过的传统的事务管理方式
        -->
        <transactionManager type="JDBC"/>

        <!-- datasource: 数据库连接池 -->
        <!--
            POOLED: 使用内部自带的数据库连接池
            UNPOOLED: 不使用数据库连接池
            JNDI: 使用第三方的数据库连接池
        -->
        <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>


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


    <environment id="prod">
        <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

```
<!-- 这个Mapper标签其实就是告诉Mybatis去哪里找SQL语句的映射文件 -->
<mappers>
    <!--  方式一-->
    <mapper resource="com/cskaoyan/mapper/UserMapper.xml"/>


    <!-- 方式二 -->
    <package name="com.cskaoyan.mapper"/>
</mappers>
```

# 5. *输入映射

输入映射其实就是指在Mybatis中，如何通过 方法中的参数给SQL语句传值。

## 5.1 一个简单参数

简单参数：基本类型 + 包装类型 + String

mapper

```
// 1. 一个简单参数
User selectUserById(Integer id);
```

mapper.xml

```
<select id="selectUserById" resultType="com.cskaoyan.bean.User">
    select * from user where id = #{sefwsg}
</select>
```

通过**#{任意值}** 来取值

## 5.2 多个简单参数

传递多个简单参数的时候，需要给参数添加@Param注解

mapper

```
// 2. 多个简单参数
int insertUser(@Param("uid") Integer id,
               @Param("username") String username,
               @Param("password") String password,
               @Param("nickname") String nickname);
```

mapper.xml

```
<insert id="insertUser">
    insert into user values (#{uid},#{username},#{password},#{nickname})
</insert>
```

通过 **#{注解值}** 来取值。

## 5.3 使用对象传值

mapper

```
// 3. 使用对象传值
int insertUserUseObject(@Param("user") User user);
```

mapper.xml

```
<insert id="insertUserUseObject">
    insert into user values (#{user.id},#{user.username},#{user.password},#{user.nickname})
</insert>
```

> 总结：
>
> 1. 当对象**没有注解**的时候，直接使用 **#{成员变量名}** 来取对应的值
> 2. 当对象**有注解**的时候，使用**#{注解值 . 成员变量名}** 来取值

## 5.4 使用map来传值

不推荐使用Map来传值。

> 在以后的开发过程中，最好不要使用map来当做方法的参数和返回值

mapper

```
// 4. 传入map
int insertUserUseMap(Map<String,Object> map);
```

mapper.xml

```
<insert id="insertUserUseMap">
    insert into user values (#{id},#{username},#{password},#{nickname})
</insert>
```

> 总结：
>
> 传递Map和传递对象是类似的。
>
> 1. 当map**没有注解**的时候，使用**#{key}** 来取值

> 2. 当map**有注解**的时候，使用**#{注解值 . key}** 来取值

## 5.5 按位置传值

不推荐使用按照位置来传值。

mapper

```
// 5. 按照位置来传值
int insertUserUseIndex(Integer id,String username,String password,String nickname);
```

mapper.xml

```
<insert id="insertUserUseIndex">
<!--         insert into user values (#{arg0},#{arg1},#{arg2},#{arg3}) -->
  insert into user values (#{param1},#{param2},#{param3},#{param4})
</insert>
```

可以通过 #{arg0}... #{arg1}... #{arg2} ... 来按照参数的位置取值

也可以通过 #{param1} ... #{param2} ...#{param3} ... 这种方式来取值

## 5.6 #{}和${}取值的区别


> 很明显，#{}底层取值的时候使用的是PreparedStatement，${}取值的时候是直接把参数拼接到SQL语句中。所以在以后的工作，我们一般使用 #{}来取值，因为它比较安全
>
> 那么 ${}有没有使用的场景呢？当传递表名或者是传递列名的时候需要使用字符串拼接，也就是使用 ${} 来取值。
>
> - 传递表名
>
>   什么时候需要传递表名？
>
>   通常我们认为MySQL中的InnoDB存储引擎是把数据存储在B+树上的，随着数据量的增加，B+树高度会变高，查询效率变慢；所以我们通常认为MySQL的单表是有性能极限的。通常认为MySQL单表的性能极限是 500w~800w
>
>   条数据，假如超过了这个数据，MySQL的索引性能会急剧下降。在公司中，假如一个表中的数据超过了MySQL单表性能极限的数据（500w~800w条），一般需要分表。
>
>   分表是指：把同样类型的数据存储到若干名字不同，但是结构相同的表中。
>
>   假如业务上出现了分表的话， 那么去做查询的时候，我们就需要传递表名了。
>
>   那么如何进行分表呢?
>
>   - 横向拆分
>
>     例如把用户数据拆分成用户表和用户详情表，把数据存储到不同的表中
>   - 纵向拆分
>
>     一般是按照业务来拆分（id，时间）
>
>   分表之后会带来一些问题：例如排序、分组等变麻烦了，那么此时可以利用一些中间件（框架）来处理这些由于分表带来的问题。 MyCat、 Sharing-JDBC
>
> - 传递列名
>
>   什么时候需要传递列名?
>
>   在我们需要对不同的字段进行排序，进行的分组的时候。
>
>   order by columnName | group by columnName ...

# 6. *输出映射

输出映射其实就是Mybatis在执行完SQL语句之后，得到一个ResultSet结果集对象，然后把这个结果集对象中的值解析到不同的Java类型中，有哪些方式呢?

## 简单类型

- 单个简单类型

  mapper

```
// 单个简单类型
String selectUserNameById(@Param("id") Integer id);
```

  mapper.xml

```
<!-- resultType 写方法的返回值类型 即可-->
<select id="selectUserNameById" resultType="java.lang.String">
    select username from user where id = #{id}
</select>
```

- 多个简单类型

  mapper

```
// 多个简单类型
// 当我们需要集合的时候就定义为集合，需要数组的时候就定义为数组
List<String> selectAllUserName();


String[] selectAllUserNameArray();
```

  mapper.xml

```
<!-- resultType这个地方写单个值的类型 -->

<select id="selectAllUserName"  resultType="string">
    select username from user
</select>


<select id="selectAllUserNameArray"  resultType="string">
    select username from user
</select>
```

## JavaBean

- 单个对象

> 当查询结果的临时表中的列名和成员变量名不一致的时候，对应的成员就赋值不进去。这个时候可以考虑通过对列名起别名来解决问题

  mapper

```
Account selectAccountById(@Param("id") Integer id);
```

  mapper.xml

```
<!--
1. resultType写返回值的全限定名称
2. 列名和成员变量名不一致的时候，起别名来让其保持一致
-->
<select id="selectAccountById" resultType="com.cskaoyan.bean.Account">
    select id,name,age,create_time as createTime from account where id = #{id}
</select>
```

- 多个对象

  mapper

```
// 多个对象
User[] selectAllUserList();
List<User> selectAllUserList();
```

  mapper.xml

```
<select id="selectAllUserList" resultType="com.cskaoyan.bean.User">
    select * from user
</select>
```

> 不管在方法中定义的是集合还是数组，mapper.xml中的resultType永远是写单个值的类型。

### ResultMap

ResultMap是一个映射器，是Mybatis中最强大的元素之一。可以帮助我们把列名和成员变量一一映射。

mapper

```
Account selectAccountByIdUseResultMap(@Param("id") Integer id);
```

mapper.xml

```
<!-- 查询的入口-->
<select id="selectAccountByIdUseResultMap" resultMap="accountMap">
    select id,name,age,create_time from account where id = #{id}
</select>


<!--id: 映射主键-->
<!--result: 映射普通的列-->
<!--column: 列名-->
<!--property: 成员变量名-->
<resultMap id="accountMap" type="com.cskaoyan.bean.Account">
    <id column="id" property="id"/>
    <result column="name" property="name"/>
    <result column="age" property="age"/>
    <result column="create_time" property="createTime"/>
</resultMap>
```

## 7. 插件

### Lombok

lombok是一个开源的组件，这个组件可以帮助我们对JavaBean自动生成getter、setter、toString 等等。

如何使用呢?

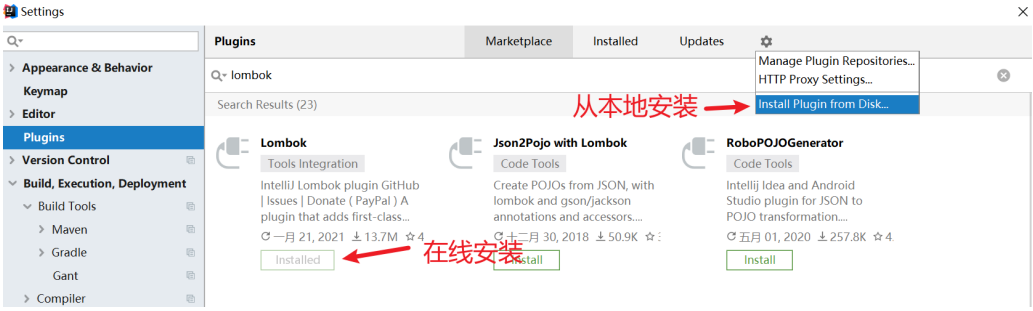
- 导包

  ```
  <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <version>1.18.12</version>
  </dependency>
  ```

- 配置

  lombok不需要任何配置文件。

  但是要在idea中下载一个插件




- 使用

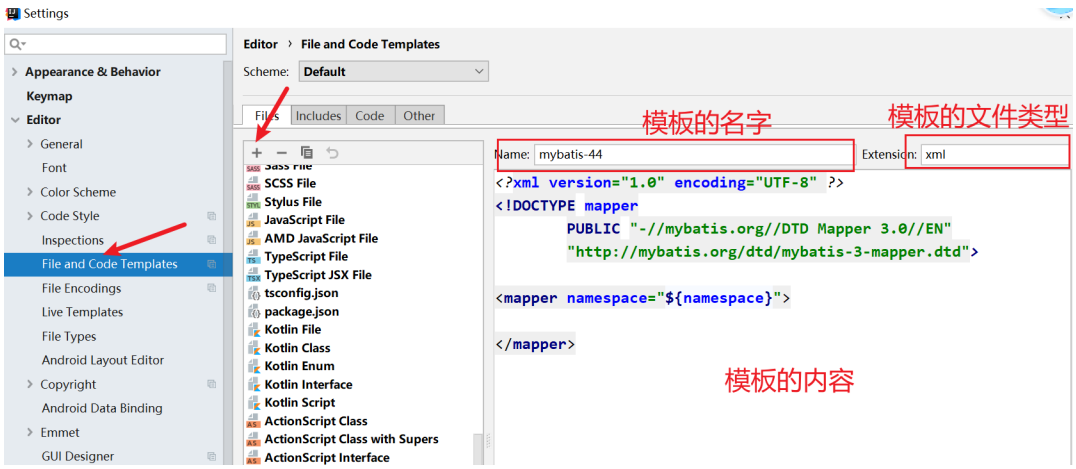
  lombok可以在编译的时候自动生成getter、setter、ToString等方法。

  在对应的Java类上面添加 @Data注解即可。

  ```
  @Data
  public class Student {
  }
  ```

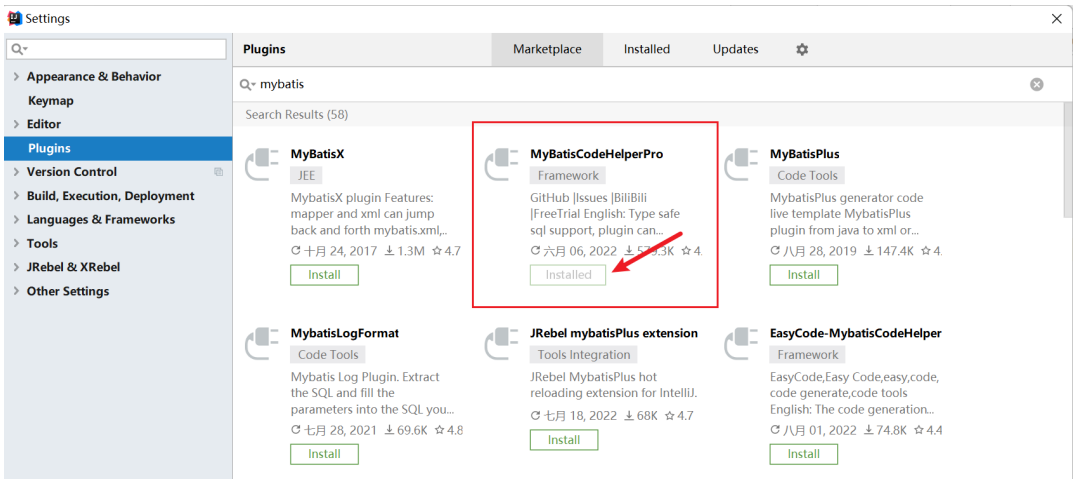
### Idea的模板功能

主要是为了去解决 每次都要复制mapper.xml中的标签头、约束文件等等内容。




## MybatisCodeHelperPro

这个是一个Mybatis的插件。其实Mybatis的插件有很多，这个只是其中的一个。




这个插件有什么功能呢?

1. 接口与XML互相跳转，更换图标
2. 一键添加Param注解
3. 参数以及变量名自动提示的功能
4. 一键生成

```
<select id="selectUserById" resultType="com.cskaoyan.bean.User">
    select * from user
    <where>
        id = #{id}
    </where>
</select>
```

mapper

```
// 根据可变的条件来查询用户
// user中有些成员变量有值，有些成员变量没有值，不确定哪些有值，哪些没有值
// 当传入User之后，我们要根据传入的成员变量来查询
// 例如：
//  1. 当传入的User中只有id的时候， select * from user where id = ?
// 2. 当传入的User中有id,username,password的时候，select * from user where id = ? and username = ? and password = ?
// 3.当User中没有任何参数的时候，应该查询所有的User，select * from user
List<User> selectUserBySelective(@Param("user") User user);
```

mapper.xml

```
<select id="selectUserBySelective" resultType="com.cskaoyan.bean.User">
    select * from user
    <where>
```

```
        <if test="user.id != null">
            and id = #{user.id}
        </if>
        <if test="user.username != null">
            and username = #{user.username}
        </if>
        <if test="user.password != null">
            and password = #{user.password}
        </if>
        <if test="user.nickname != null">
            and nickname = #{user.nickname}
        </if>
    </where>
</select>
```

1. 帮助我们自动生成where关键字
2. 当where标签中有if标签的时候，可以帮助我们自动去除和where标签相邻的 and 或者是 or关键字
3. 当where标签中没有条件满足的时候，不会拼接where关键字

## *8.2 if

if标签实际上就类似于Java中的if关键字，可以帮助我们去判断，根据传入的参数的值去判断，然后动态的拼接不同的SQL片段。

mapper

```
// if标签
// 根据id来查询UserList
// 假如传入的id值大于3000，那么查询所有id比3000大的用户
// 假如传入的id小于或者等于3000，那么查询id小于等于3000的用户
List<User> selectUserListById(@Param("id") Integer id);
```

mapper.xml

```
<select id="selectUserListById" resultType="com.cskaoyan.bean.User">

    select * from user where
    <!-- test = "", 引号中，也要写参数的名字,和#{}中取值的方式是一样的 -->
    <if test="id &gt; 3000">
        id &gt; 3000
    </if>
    <if test="id &lt;= 3000">
        id &lt;= 3000
    </if>
</select>
```

## 8.3 choose when otherwise

相当于 if else ...

mapper

```
// choose when otherwise
// 根据id来查询UserList
// 假如传入的id值大于3000，那么查询所有id比3000大的用户
// 否则，查询id小于等于3000的用户
List<User> selectUserListByIdUseChoose(@Param("id") Integer id);
```

mapper.xml

```
<select id="selectUserListByIdUseChoose" resultType="com.cskaoyan.bean.User">
    select * from user where
    <choose>
        <!-- 相当于if -->
        <when test="id &gt; 3000">
            id &gt; 3000
        </when>
        <!-- 相当于else -->
        <otherwise>
            id &lt;= 3000
        </otherwise>
    </choose>
</select>
```

## 8.4 trim

修剪的意思。

mapper

```
// trim
// 根据user中的id来修改用户
// 1. id必须要传
// 2. user中至少有一个除了id之外的其他的字段传入
// 3. 假如username传入了，那么就要修改username
// 4. 假如username和password都传了，那么username和password都要修改
int updateUserByIdSelective(@Param("user") User user);
```

mapper.xml

```
<update id="updateUserByIdSelective">
    <!--update user set username = ?,password = ?,nickname = ? where id = ?-->
    update user  set

    <!--
    prefix: 增加指定的前缀
    prefixOverrides: 去除指定的前缀
    suffix: 增加指定的后缀
    suffixOverrides:去除指定的后缀
    -->

    <trim prefix="" suffixOverrides="," suffix="" prefixOverrides="">
        <if test="user.username != null">
            username = #{user.username},
        </if>
        <if test="user.password != null">
            password = #{user.password},
        </if>
        <if test="user.nickname != null">
            nickname = #{user.nickname},
        </if>
    </trim>

    where id = #{user.id}
</update>
```

## *8.5 set

set标签是专门用来去应对修改的这种业务场景的。

相当于 trim标签的这个设置：

mapper

```
int updateUserByIdSelectiveUseSet(@Param("user") User user);
```

mapper.xml

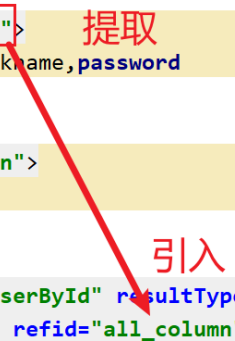
```
<update id="updateUserByIdSelectiveUseSet">
    update user
    <set>
        <if test="user.username != null">
            username = #{user.username},
        </if>
        <if test="user.password != null">
            password = #{user.password},
        </if>
        <if test="user.nickname != null">
            nickname = #{user.nickname},
        </if>
    </set>
    where id = #{user.id}
</update>
```

## *8.6 sql-include

sql-include是两个标签配合起来一起使用。

- sql： 可以帮助我们提取公共的SQL片段
- include：引入SQL片段

```xml
<sql id="all_column">
    id,username,nickname,password
</sql>

<sql id="base_column">
    id,username
</sql>

<select id="selectUserById" resultType="com.cskaoyan.bean.User">
    select <include refid="all_column"/> from user
    <where>
        id = #{id}
    </where>
</select>
```




> sql-include标签会比较严重的破坏SQL语句的结构，影响SQL的可读性，一般只用来提取一些重复使用的列名。

## *8.7 foreach

foreach这个标签可以帮助我们去循环获取传入的参数。

> - 假如参数有注解，（不管是数组还是集合）那么collection取注解中的值
> - 假如参数没有注解，如果传入的是一个集合，那么就使用collection | list 来获取参数；假如传入的是一个数组，那么就使用array来获取值
>
> ```xml
> <select id="selectUserListByIdList" resultType="com.cskaoyan.bean.User">
>     <!-- select * from user where id in (?,?,?,?,?)-->
>     select <include refid="all_column"/>
>     from user
>     where id in
>
>     <foreach collection="list" item="uid" separator="," open="(" close=")" index="">
>         #{uid}
>     </foreach>
>
> </select>
> ```

- in查询

  mapper

  ```java
  // in 查询
  // 传入list
  List<User> selectUserListByIdList(List<Integer> idList);

  //传入数组
  List<User> selectUserListByIdArray(Integer[] idArray);
  ```

  mapper.xml

  ```xml
  <!--
      collection: 参数（集合）的名字
      item: 集合中的元素
      separator: 集中中的元素在循环的时候，以什么符号间隔开来
      open: 在循环开始增加指定的字符
      close:  在循环结束增加指定的字符
      index: 循环中元素的下标
  -->
  <select id="selectUserListByIdList" resultType="com.cskaoyan.bean.User">
      <!-- select * from user where id in (?,?,?,?,?)-->
      select <include refid="all_column"/>
      from user
      where id in

      <foreach collection="list" item="uid" separator="," open="(" close=")" index="">
          #{uid}
      </foreach>

  </select>

  <select id="selectUserListByIdArray" resultType="com.cskaoyan.bean.User">
      select <include refid="all_column"/>
      from user
      where id in
  ```

```
    <foreach collection="array" item="uid" separator="," open="(" close=")" index="">
        #{uid}
    </foreach>
</select>
```

- 批量插入

  mapper

```
// 批量插入
int insertUserList(@Param("userList") List<User> userList);
```

  mapper.xml

```
<!-- 批量插入-->
<insert id="insertUserList">
    <!--insert into user values (?,?,?,?),(?,?,?,?),(?,?,?,?),(?,?,?,?)-->
    insert into user values
    <foreach collection="userList" item="user" separator="," open="" close="" index="">
        (#{user.id},#{user.username},#{user.password},#{user.nickname})
    </foreach>
</insert>
```

## 8.8 selectkey

这个标签可以帮助我们在执行目标SQL语句之前或者是之后执行一条额外的SQL语句。

mapper

```
// 插入一个User，并且同时返回这个User自增的主键
int insertUserAndReturnId(@Param("user") User user);
```

mapper.xml

```
<insert id="insertUserAndReturnId">
    insert into user values (null,#{user.username},#{user.password},#{user.nickname})
    <!--
    order: BEFORE |AFTER 是指在目标SQL语句执行之前或者是之后执行
    resultType: sql语句的返回值类型
    keyProperty: 把对应的结果封装到哪个参数中去

    -->
    <selectKey order="AFTER" resultType="int" keyProperty="user.id">
        select LAST_INSERT_ID()
    </selectKey>
</insert>
```

## 8.9 useGeneratedKeys

这个是一个属性，可以帮助我们在插入的时候，获取自增的id

mapper

```
// UseGeneratedKeys
// 插入一个User，并且同时返回这个User自增的主键
int insertUserAndReturnIdUseGeneratedKeys(@Param("user") User user);
```

mapper.xml

```
<insert id="insertUserAndReturnIdUseGeneratedKeys" useGeneratedKeys="true" keyProperty="user.id">
    insert into user values (null,#{user.username},#{user.password},#{user.nickname})
</insert>
```

# 9. *多表查询

其实就是复杂映射。

## 一对一

- 表设计（首先得有相关的业务场景）

| id | username | nickname | gender | age |
|---|---|---|---|---|
| 1 | 孙悟空 | 齐天大圣 | male | 600 |
| 2 | 猪八戒 | 天蓬元帅 | male | 300 |
| 3 | 嫦娥 | 广寒仙子 | female | 800 |
| 4 | 土地公 | 老头 | male | 1000 |

| id | user_id | height | weight | pic |
|---|---|---|---|---|
| 1 | 2 | 170 | 200 | 猪八戒.jpg |
| 3 | 3 | 170 | 100 | 嫦娥.jpg |
| 4 | 4 | 150 | 180 | 土地公.gif |

- 建立JavaBean

```java
@Data
public class User {

    Integer id;
    String username;
    String nickname;
    String gender;
    Integer age;

    UserDetail userDetail;
}
```

引用对象

```java
@Data
public class UserDetail {

    Integer id;
    Integer userId;
    Integer height;
    Integer weight;
    String pic;

}
```

### 分次查询

```
select * from user where id = #{id}
select * from user_detail where user_id = #{id}
```

mapper

```
// 查询用户信息，并且同时查询出这个用户的 详细信息
//分次查询
User selectUserByIdWithUserDetail(@Param("id") Integer id);
```

mapper.xml

```
<!-- 1. 查询的入口-->
<select id="selectUserByIdWithUserDetail" resultMap="userMap">
    select id,username,nickname,gender,age from user
    where id = #{id}
</select>

<!-- 2. 使用resultMap去映射 -->
<resultMap id="userMap" type="com.cskaoyan.bean.User">

    <id column="id" property="id"/>
    <result column="username" property="username"/>
    <result column="nickname" property="nickname"/>
    <result column="gender" property="gender"/>
    <result column="age" property="age"/>

    <!--
    property：成员变量的名字
    javaType: 成员变量的类型
    select: 第二次查询的SQL语句的坐标，namespace.id
    column: 把哪一列的值传入到第二次查询中去
    -->
    <association property="userDetail"
                 javaType="com.cskaoyan.bean.UserDetail"
                 select="com.cskaoyan.mapper.UserMapper.selectUserDetailByUserId"
                 column="id"/>


</resultMap>

<!-- 3. 第二次查询 -->
<select id="selectUserDetailByUserId" resultType="com.cskaoyan.bean.UserDetail">
    select id,user_id,height,weight,pic from user_detail where user_id = #{id}
</select>
```

### 连接查询

```
-- 整体思路：
-- 1. 通过连接查询的SQL语句把所有你需要的信息都查出来
-- 2. 映射

select * from user left join user_detail on user.id = user_detail.user_id
where user.id = #{}
```

mapper

```
// 连接查询
User selectUserByIdWithUserDetailUseCrossQuery(@Param("id") Integer id);
```

mapper.xml

```
<!-- 连接查询  -->
<!--1. 查询 -->
<select id="selectUserByIdWithUserDetailUseCrossQuery" resultMap="userCrossMap">
    SELECT
            u.id AS id,
            u.username AS username,
            u.nickname AS nickname,
            u.gender AS gender,
            u.age AS age,
            d.id AS did,
            d.user_id AS userId,
            d.height AS height,
            d.weight AS weight,
            d.pic AS pic
    FROM
            user AS u
            LEFT JOIN user_detail AS d ON u.id = d.user_id
    WHERE
            u.id = #{id}
</select>


<!--2. 映射 -->
<resultMap id="userCrossMap" type="com.cskaoyan.bean.User">

    <id column="id" property="id"/>
    <result column="username" property="username"/>
    <result column="nickname" property="nickname"/>
    <result column="gender" property="gender"/>
    <result column="age" property="age"/>

    <association property="userDetail" javaType="com.cskaoyan.bean.UserDetail">
        <id column="did" property="id"/>
        <result column="userId" property="userId"/>
        <result column="height" property="height"/>
        <result column="weight" property="weight"/>
        <result column="pic" property="pic"/>
    </association>

</resultMap>
```

## 一对多

- 建立表模型

| id | name |
|---|---|
| 1 | 一班 |
| 2 | 二班 |
| 3 | 三班 |
| 4 | 四班 |
| 5 | 五班 |
| 6 | 六班 |

| id | name | age | gender | clazz_id |
|---|---|---|---|---|
| 1 | 张三 | 18 | male | 1 |
| 2 | 罗老师 | 20 | male | 1 |
| 3 | 珠珠 | 18 | female | 1 |
| 4 | 孔融 | 18 | male | 2 |
| 5 | 诸葛亮 | 20 | male | 2 |
| 6 | 黄月英 | 18 | female | 2 |
| 7 | 吕布 | 18 | male | 3 |
| 8 | 董卓 | 42 | male | 3 |
| 9 | 丁原 | 40 | male | 3 |
| 10 | 貂蝉 | 18 | female | 3 |
| 11 | 马超 | 30 | male | 4 |
| 12 | 刘星 | 20 | male | 10 |

- 建立JavaBean

```java
@Data
public class Clazz {


    Integer id;
    String name;

    List<Student> studentList;
}
```

```java
@Data
public class Student {

    Integer id;
    String name;
    Integer age;
    String gender;
    Integer clazzId;
}
```

**分次查询**

```
# 根据班级Id查询出班级信息以及这个班级对应的学生信息

# 1. 查询班级信息
select * from clazz where id = #{};

# 2. 查询学生信息
select * from student where clazz_id = #{}
```

mapper

```
// 1. 分次查询
Clazz selectClazzByIdWithStudentList(Integer id);
```

mapper.xml

```
<!-- 1. 查询的入口-->
<select id="selectClazzByIdWithStudentList" resultMap="clazzMap">
    select id,name from clazz where id = #{id}
</select>

<!--2. 映射-->
<resultMap id="clazzMap" type="com.cskaoyan.bean.Clazz">
    <id column="id" property="id"/>
    <result column="name" property="name"/>

    <!-- 关联查询StudentList-->
    <collection property="studentList"
                ofType="com.cskaoyan.bean.Student"
                select="com.cskaoyan.mapper.ClazzMapper.selectStudentListByClazzId"
                column="id"/>


</resultMap>
```

```
<!-- 3. 第二次查询-->
<select id="selectStudentListByClazzId" resultType="com.cskaoyan.bean.Student">
    select id,name,age,gender,clazz_id as clazzId from student
    where clazz_id = #{id}
</select>
```

### 连接查询

mapper

```
// 2. 连接查询
Clazz selectClazzByIdWithStudentListUseCrossQuery(Integer id);
```

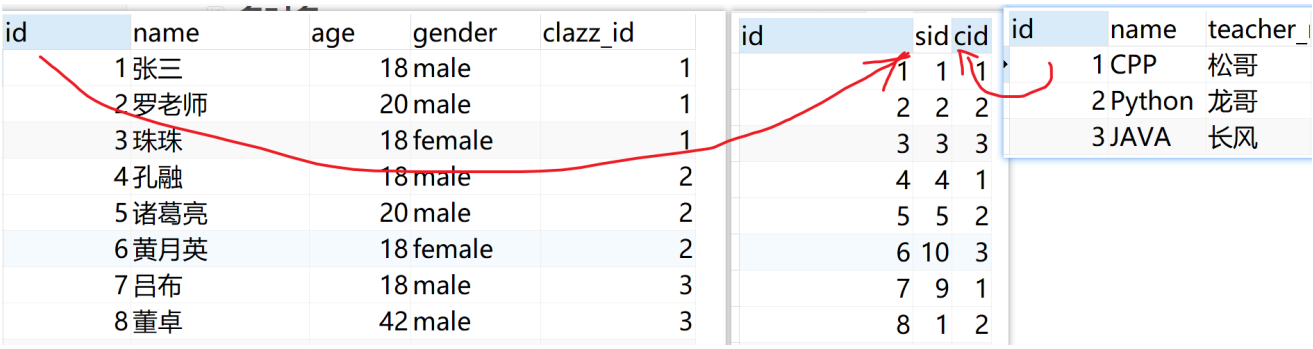
mapper.xml

```
<!-- 连接查询 1. 查询 -->
<select id="selectClazzByIdWithStudentListUseCrossQuery" resultMap="clazzCrossMap">
    SELECT
            c.id AS cid,
            c.name AS cname,
            s.id AS sid,
            s.name AS sname,
            s.age AS age,
            s.gender AS gender,
            s.clazz_id AS clazzId
    FROM
            clazz AS c
                    LEFT JOIN student AS s ON c.id = s.clazz_id
    WHERE
            c.id = #{id}
</select>


<!-- 2. 映射 -->
<resultMap id="clazzCrossMap" type="com.cskaoyan.bean.Clazz">
    <id column="cid" property="id"/>
    <result column="cname" property="name"/>

    <collection property="studentList" ofType="com.cskaoyan.bean.Student">
        <id column="sid" property="id"/>
        <result column="sname" property="name"/>
        <result column="age" property="age"/>
        <result column="gender" property="gender"/>
        <result column="clazzId" property="clazzId"/>
    </collection>


</resultMap>
```

## 多对多

多对多的本质是互为一对多。

- 建表



| id | name | age | gender | clazz_id |
|---|---|---|---|---|
| 1 | 张三 | 18 | male | 1 |
| 2 | 罗老师 | 20 | male | 1 |
| 3 | 珠珠 | 18 | female | 1 |
| 4 | 孔融 | 18 | male | 2 |
| 5 | 诸葛亮 | 20 | male | 2 |
| 6 | 黄月英 | 18 | female | 2 |
| 7 | 吕布 | 18 | male | 3 |
| 8 | 董卓 | 42 | male | 3 |

| id | sid | cid |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 2 | 2 |
| 3 | 3 | 3 |
| 4 | 4 | 1 |
| 5 | 5 | 2 |
| 6 | 10 | 3 |
| 7 | 9 | 1 |
| 8 | 1 | 2 |

| id | name | teacher_ |
|---|---|---|
| 1 | CPP | 松哥 |
| 2 | Python | 龙哥 |
| 3 | JAVA | 长风 |

- JavaBean

```java
@Data
public class Student {

    Integer id;
    String name;
    Integer age;
    String gender;
    Integer clazzId;

    List<Course> courseList;
}
```

```java
@Data
public class Course {

    Integer id;
    String name;
    String teacherName;
    Integer score;

    List<Student> studentList;

}
```

> 需要说明的是：两边都可以去维护他们之间的关系，但是查询的时候只用查询一边即可。

### 分次查询

```
<!--
    1. 查询学生信息
    select * from student where id = #{}


    2. 根据学生的id查询这个学生的选课信息
    select * from course where id in (select cid from s_c where sid = #{});

-->
```

mapper

```
// 1. 分次查询
Student selectStudentByIdWithCourseList(@Param("id") Integer id);
```

mapper.xml

```
<!-- 1. 查询的入口 -->
<select id="selectStudentByIdWithCourseList" resultMap="studentMap">
    select id,name,age,gender,clazz_id from student where id = #{id}
</select>

<!-- 2. 映射，关联-->
<resultMap id="studentMap" type="com.cskaoyan.bean.Student">
    <id column="id" property="id"/>
    <result column="name" property="name"/>
    <result column="age" property="age"/>
    <result column="gender" property="gender"/>
    <result column="clazz_id" property="clazzId"/>

    <collection property="courseList"
                ofType="com.cskaoyan.bean.Course"
                select="selectCourseListByStudentId"
                column="id"/>
</resultMap>

<!-- 3. 查询这个学生的课程信息-->
<select id="selectCourseListByStudentId" resultType="com.cskaoyan.bean.Course">
    select id,name,teacher_name as teacherName,score from course where id in (select cid from s_c where sid = #{id})
</select>
```

### 连接查询

```
select * from student as s
    left join s_c as sc on s.id = sc.sid
    left join course as c on c.id = sc.cid;
```

mapper

```
// 2. 查询所有的学生对应的学生信息以及他们各自对应的课程信息
List<Student> selectStudentListWithCourseList();
```

mapper.xml

```
<!-- 连接查 -->
```

```xml
<select id="selectStudentListWithCourseList" resultMap="studentCrossMap">
    SELECT
            s.id AS sid,
            s.name AS sname,
            s.age AS age,
            s.gender AS gender,
            s.clazz_id AS clazzId,
            c.id AS cid,
            c.name AS cname,
            c.teacher_name AS teacherName,
            c.score AS score
    FROM
            student AS s
                    LEFT JOIN s_c AS sc ON s.id = sc.sid
                    LEFT JOIN course AS c ON c.id = sc.cid
</select>


<resultMap id="studentCrossMap" type="com.cskaoyan.bean.Student">
    <id column="sid" property="id"/>
    <result column="sname" property="name"/>
    <result column="age" property="age"/>
    <result column="gender" property="gender"/>
    <result column="clazzId" property="clazzId"/>

    <collection property="courseList" ofType="com.cskaoyan.bean.Course">
        <id column="cid" property="id"/>
        <result column="cname" property="name"/>
        <result column="teacherName" property="teacherName"/>
        <result column="score" property="score"/>
    </collection>

</resultMap>
```

## 10. 懒加载

懒加载指的是MyBatis在进行分次查询的时候，如果不需要第二条SQL语句的执行结果，那么就不执行关联的第二条SQL语句；当需要用到第二条SQL语句的执行的结果的时候，再去执行第二条SQL语句。

如何开启懒加载呢?

- 总开关

```xml
<!-- 懒加载总开关 -->
<setting name="lazyLoadingEnabled" value="false"/>
```

- 局部开关

```xml
<association property="userDetail"
             javaType="com.cskaoyan.bean.UserDetail"
             select="com.cskaoyan.mapper.UserMapper.selectUserDetailByUserId"
             column="id"
             fetchType="lazy"
/>
```

懒加载的局部开关 lazy | eager

> 懒加载默认是关闭的。
>
> 当总开关和局部开关冲突的时候，以局部开关为准。

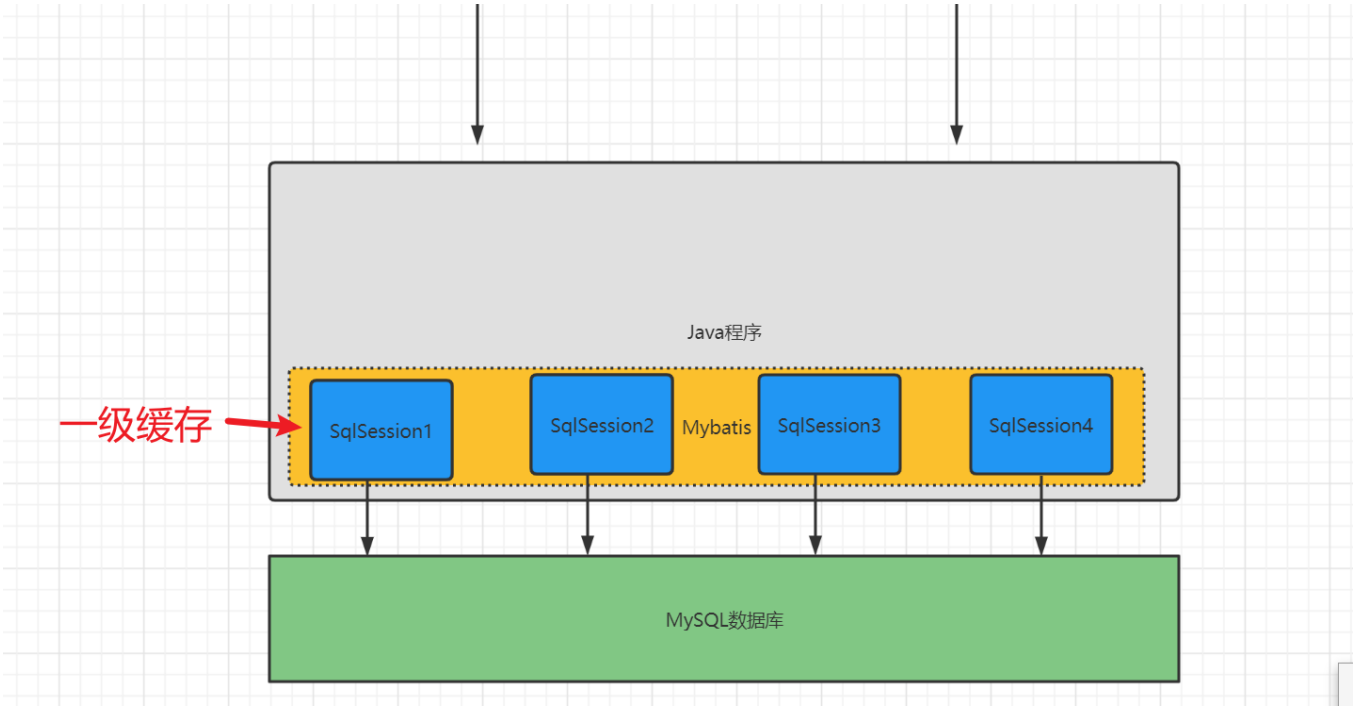
## 11. 缓存

### 一级缓存

Mybatis的一级缓存是一个sqlSession级别的缓存。

本质上就是内存中有一个map对象。key就是SQL语句以及它对应的参数，value就是这个SQL语句执行的结果映射的Java对象。

Mybatis的一级缓存是默认开启的，并且没有提供给用户来关闭。



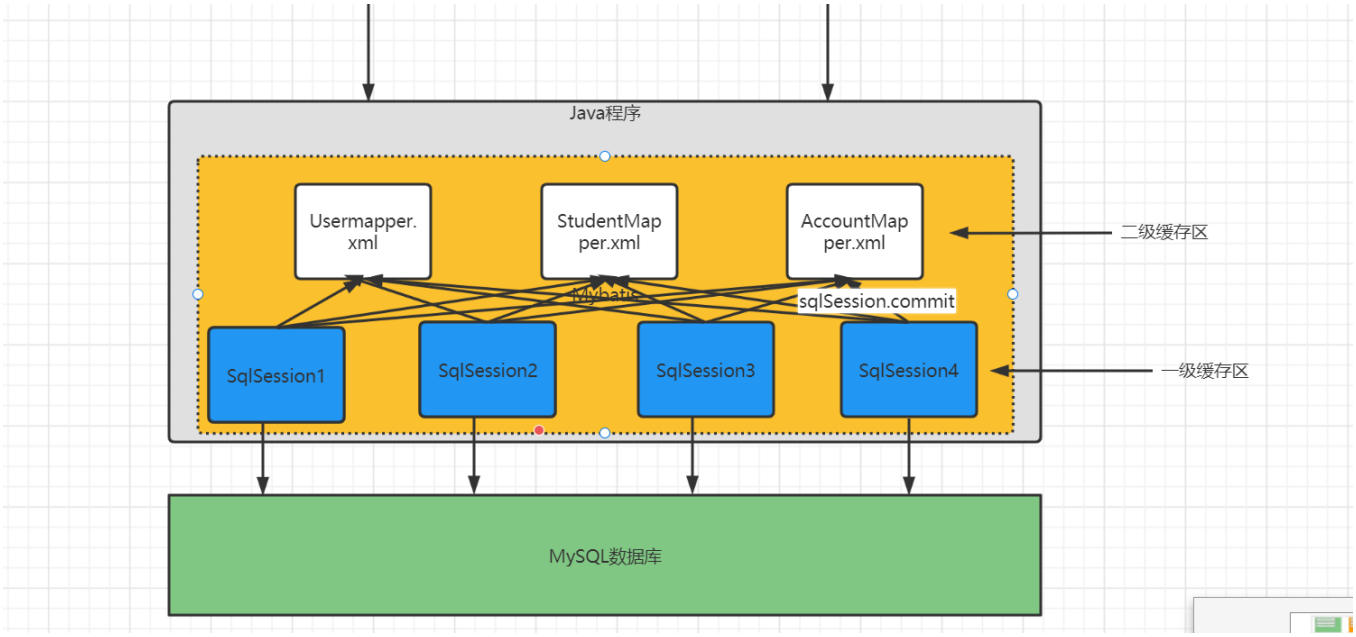

一级缓存什么时候失效呢?

- sqlSession关闭的时候
- sqlSession提交的时候，sqlSession.commit();

> 总结：一级缓存使用的局限性比较强（因为你需要使用同一个SqlSession来连续执行同一个Sql语句两次或以上，这中间不能发生增删改，这样才能使用上一级缓存），大多数的时候没有太大的作用。

## 二级缓存

Mybatis的二级缓存是一个Namespace级别的缓存。每一个Mapper.xml文件都有一个自己的命名空间，也有一个自己对应的二级缓存空间。




Mybatis的二级缓存默认是关闭的，如何使用Mybatis的二级缓存呢?

- 打开总开关

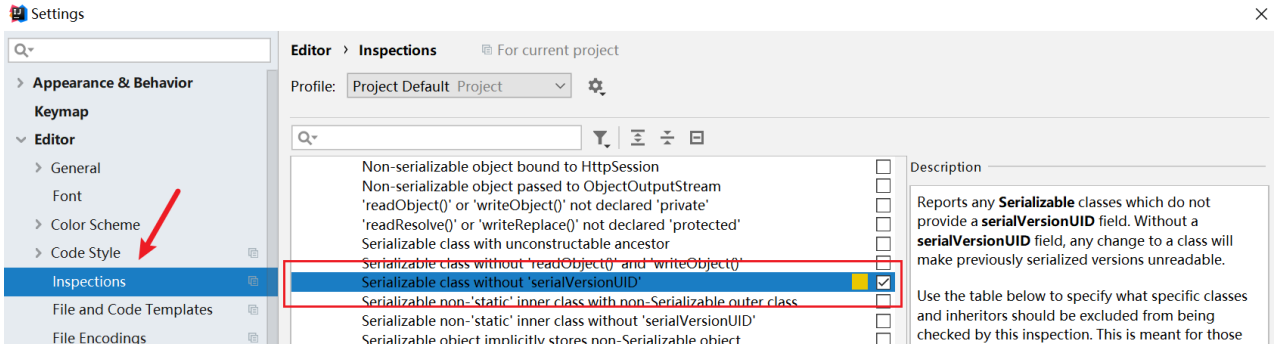
  其实可以不用做，因为总开关默认是开启的

```
<!-- 二级缓存总开关 总开关是默认开启的-->
<!-- 全局性地开启或关闭所有映射器配置文件中已配置的任何缓存 -->
<setting name="cacheEnabled" value="true"/>
```

- 打开局部开关

```xml
<mapper namespace="com.cskaoyan.mapper.UserMapper">

    <!-- 表示开启UserMapper.xml 的二级缓存 -->
    <cache/>
```

- 配置序列化
  - 配置idea




  - 实现序列化接口，生成序列化id

```java
@Data
public class UserDetail implements Serializable {

    private static final long serialVersionUID = 4543432565644000970L;

    Integer id;
    Integer userId;
    Integer height;
    Integer weight;
    String pic;

}
```

总结：Mybatis的二级缓存适用性比一级缓存要强很多，因为没有sqlSession的限制了，但是在实际的工作中，我们依然不会使用Mybatis的二级缓存，因为二级缓存对于我们用户来说是透明的，是不可直接操作的，用起来比较不方便。

那么在实际的工作中，对于某些数据，我们需要让应用程序读取速度很快，这个时候该怎么办呢？

一般在实际的工作，如果需要用到缓存，那么会考虑使用NoSQL数据库，例如Redis。