MyBatis映射器：实现动态 SQL 语句

> 本文为稀土掘金技术社区首发签约文章，30天内禁止转载，30天后未获授权禁止转载，侵权必究！

大家好，我是[王有志](http://cxyroad.com/ ”https://www.yuque.com/wangyouzhi-u3woi/cr7d5y/uw8c5iyvpgnqpzmg”)，一个分享硬核 Java 技术的金融摸鱼侠，欢迎大家加入 Java 人自己的交流群“[共同富裕的 Java 人](http://cxyroad.com/ ”https://www.yuque.com/wangyouzhi-u3woi/cr7d5y/nwry2mdlktok50bt”)”。

上一篇文章中，我们已经学习了如何在 MyBatis 的映射器中通过简单的 SQL 语句实现增删改查，今天我们来一起学习如何在 MyBatis 的映射器中构建动态 SQL 语句来实现较为复杂的查询。

今天的内容主要围绕着 MyBatis 提供的 7 个用于构建动态 SQL 语句的元素展开：

* **if 元素，MyBatis 映射器中用于实现条件判断的元素；**
* **where 元素，MyBatis 映射器中用于构建条件语句的元素；**
* **set 元素，MyBatis 映射器中用于构建 update 语句中赋值语句的元素；**
* **trim 元素，MyBatis 映射器中用于动态构建子句，处理子句前后字符串的元素；**
* **foreach 元素，MyBatis 映射器中用于遍历集合，字典的元素；**
* **choose 元素，MyBatis 映射器中用于构建条件选择语句的元素，与 Java 中的**`switch...case...default`**语句相同；**
* **bind 元素，MyBatis 映射器中，用于创建自定义局部变量的元素。**

提前叠个甲，本文的重点是展示如何在 MyBatis 映射器中构建动态 SQL 语句，因此很多案例并不符合实际的业务场景。通常，如果有一个比较复杂的查询场景，我们会先通过 Java 代码进行前期的处理（如参数处理），然后再去构建较为简单的 SQL 语句，而不会直接构建复杂的 SQL 语句进行查询，这也是符合当下大家较为认可的开发规范（尽可能使用简单的 SQL 语句）。

if 元素

if 元素是我们在 MyBatis 映射器中最常使用的动态 SQL 语句元素，没有之一。**MyBatis 中的 if 元素与 Java 中的 if 关键字功能是一样的，用于实现条件判断，它只有一个属性 test，用于编写条件判断语句**。

假设我们有如下需求，实现根据输入的用户信息来查询所有符合条件的用户，如果没有输入任何用户信息，则查询全部用户，那么我们可以这样来写 MyBatis 映射器中的 SQL 语句：

```
<select id="selectByRecord" parameterType="com.wyz.entity.UserDO"
resultType="com.wyz.entity.UserDO">
select * from user
where 1 = 1
<if test="user.name != null">
and name = #{user.name, jdbcType=VARCHAR}
</if>
<if test="user.age != null">
and age = #{user.age, jdbcType=INTEGER}
</if>
<if test="user.gender != null">
and gender = #{user.gender, jdbcType=VARCHAR}
</if>
<if test="user.idType != null">
and id_type = #{user.idType, jdbcType=INTEGER}
</if>
<if test="user.idNumber != null">
and id_number = #{user.idNumber, jdbcType=VARCHAR}
</if>
</select>
```

Mapper 接口中的方法：

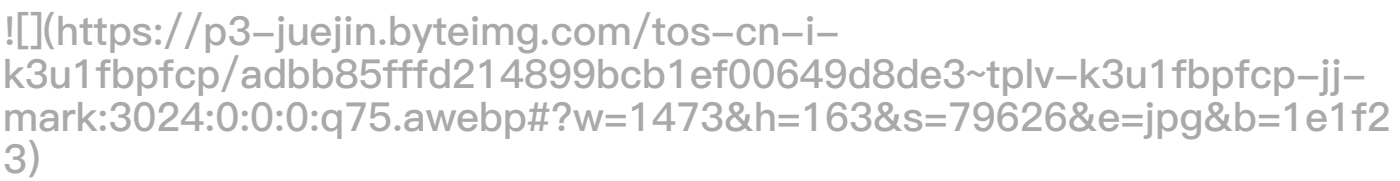
```
List<UserDO> selectByRecord(@Param("user") UserDO userDO);
```

最后我们来写单元测试代码：

```
public void testSelectByRecord() throws IOException {
SqlSession sqlSession = sqlSessionFactory.openSession();
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

UserDO userDO = new UserDO();
userDO.setName("李四");
userDO.setAge(20);
userDO.setGender("F");
userDO.setIdType(1);
userDO.setIdNumber("110101199907087 85X");

List<UserDO> users = userMapper.selectByRecord(userDO);
log.info("查询全部用户：{}", JSON.toJSONString(users));

sqlSession.close();
}
```

查看控制台输出：

我们删除测试案例中的部分参数赋值，只留下为 name 字段赋值的部分，再来执行测试案例，并观察控制台输出的 SQL 语句：



可以看到，此时输出的查询语句中，条件语句中只剩下了根据 name 字段查询的部分了。

if 元素支持的运算符

上面的案例中，我们在 if 元素中使用了运算符不等于（!=），除此之外我们还可以在 if 元素中使用其它的常见运算符：

运算符	转义符	说明
==	eq	等于
!=	neq	不等于
>		大于
>=	>=	大于等于
<	<	小于
<=	<=	小于等于
&&	and	逻辑与
||	or	逻辑或

注意，如果使用的运算符中含有“<”或者“&”，你需要使用它的转义符，否则在 XML 解析的过程中无法处理。例如，在上面的表格中，小于（<），小于等于（<=）和逻辑与（&&）需要使用它们的转义符。

实际上，MyBatis 支持的运算符远远不止这些，MyBatis 甚至连移位运算符（如右移运算符>>）和异或运算符（^或 xor）都可以支持。不过在日常工作中，上面表格中展示的运算符已经能够满足我们绝大部分的场景了。

Tips：实际上，MyBatis 中支持的这种表达式就是 OGNL 表达式，**除了 if 元素的 test 属性支持 OGNL 表达式外，when 元素的 test 属性，bind 元素的 value 属性以及 foreach 元素的 collection 属性都支持 OGNL 表达式**。

where 元素

上面的案例中，我们使用了“丑陋”的条件语句：1 = 1。

想必很多小伙伴都这么写过 MyBatis 的 SQL 语句吧？这也是无奈之举，因为 SQL 的语法规则中，条件语句的第一个条件不能以 and 开头，因此我们加上“1 = 1”来保证动态组装的条件不会作为条件语句的第一个条件。

虽然，MySQL 5.7 之后对“1 = 1”进行了优化，来保证不会产生性能问题，但是这么做会让代码变得非常丑陋，其次也会显得我们功力浅薄。

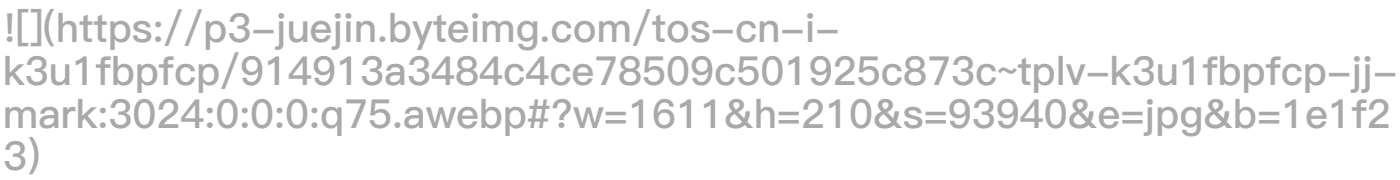
MyBatis 贴心的为我们准备了解决办法：**where 元素，它会先根据 if 元素的处理结果生成子句，然后将子句前缀的“and”和“or”移除，最后在子句的开始处插入“where”，如果不存在子句，则不会插入“where”。**

我们将上面的案例改写为使用 where 元素的方式，如下：

```
<select id=”selectByRecord” parameterType=”com.wyz.entity.UserDO” resultType=”com.wyz.entity.UserDO”>
select * from user
<where>
<if test=”user.name != null”>
and name = #{user.name, jdbcType=VARCHAR}
</if>
<if test=”user.age != null”>
and age = #{user.age, jdbcType=INTEGER}
</if>
<if test=”user.gender != null”>
and gender = #{user.gender, jdbcType=VARCHAR}
</if>
<if test=”user.idType != null”>
and id_type = #{user.idType, jdbcType=INTEGER}
</if>
<if test=”user.idNumber != null”>
and id_number = #{user.idNumber, jdbcType=VARCHAR}
</if>
</where>
</select>
```

这时我们再执行测试案例，可以看到控制台输出的 SQL 语句的条件语句中，首个条件前的 and 已经不见了：

现在我们删除所有参数赋值，再来执行测试案例：



可以看到，此时控制台输出的 SQL 语句中，已经完全没有了 where 的身影。

set 元素

set 元素与 where 元素类似，不过 set 元素是用在 update 语句中的，**set 元素会先根据 if 元素的处理结果生成子句，然后将子句结尾处的“,”移除，最后在子句的开始处插入“set”，如果不存在子句则不会插入“set”**。与 where 元素不同的是，在 SQL 规范中，update 语句是必须包含 set 子句的，否则无法处理 SQL 语句。

我们来写一个使用 set 元素更新用户信息的案例，首先来写 MyBatis 映射器中的 SQL 语句：

```
<update id=”updateByRecord”
parameterType=”com.wyz.entity.UserDO”>
  update user
  <set>
    <if test=”user.name != null”>
      name = #{user.name, jdbcType=VARCHAR},
    </if>
    <if test=”user.age != null”>
      age = #{user.age, jdbcType=INTEGER},
    </if>
    <if test=”user.gender != null”>
      gender = #{user.gender, jdbcType=VARCHAR},
    </if>
    <if test=”user.idType != null”>
      id_type = #{user.idType, jdbcType=INTEGER},
    </if>
    <if test=”user.idNumber != null”>
      id_number = #{user.idNumber, jdbcType=VARCHAR},
    </if>
  </set>
  where user_id = #{user.userId, jdbcType=INTEGER}
```

</update>

```

可以看到，在 update 语句中，最后一个赋值语句里我们同样在`id_number = #{user.idNumber, jdbcType=VARCHAR}`后添加了“,”。

接着我们来定义 Mapper 接口中的方法：

```
int updateByRecord(@Param(”user”) UserDO userDO);
```

最后我们来写测试代码：

```
public void testUpdateByRecord() throws IOException {
SqlSession sqlSession = sqlSessionFactory.openSession();
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

UserDO userDO = new UserDO();
userDO.setUserId(1);
userDO.setName(”王二麻子”);
userDO.setAge(19);
userDO.setGender(”M”);
userDO.setIdType(1);
userDO.setIdNumber(”12345678901234567”);

int count = userMapper.updateByRecord(userDO);

sqlSession.commit();
sqlSession.close();
}
```

执行测试案例，可以看到控制台输出的 SQL 语句中，set 元素成功的为 update 语句插入了 set 语句，并且为 set 语句中的最后一项赋值操作自动删除了“,”，如下：

![](https://p3-juejin.byteimg.com/tos-cn-i-
```

k3u1fbpfcp/462926fd722a4cb7a1975e99586609b2~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1243&h=85&s=41888&e=jpg&b=1e1f23)

trim 元素

上面我们展示了在 select 语句中和 update 语句中动态组装 where 语句和 set 语句，那么如果想要在 insert 语句中动态组装插入字段呢？是不是有类似 where 元素和 set 元的元素呢?

实在抱歉，这个真没有。不过 MyBatis 提供了另一个更加灵活（意味着你需要做更多的处理来实现你的需求） 的元素：**tirm 元素**。

trim 元素中定义了 4 个属性，可以将它们分为两组：

* 前缀相关的属性：
+ **prefix 属性：子句中要在开始处插入的字符串；**
+ **prefixOverrides 属性：子句中要被移除的前缀字符串；**
* 后缀相关的属性：
+ **suffix 属性：子句中要在结尾处插入的字符串；**
+ **suffixOverrides 属性：子句中要被移除的后缀字符串。**

trim 元素中，同样先是根据 if 元素的处理结果生成子句，然后根据 prefixOverrides 属性和 suffixOverrides 属性的配置移除子句前后缀的字符串，最后根据 prefix 属性和 suffix 属性的配置，在子句的开始处和结尾处插入字符串。

Tips：你可能会被属性名称中的 Overrides 所迷惑，认为是 prefix（suffix）中指定的字符串替换了 prefixOverrides（suffixOverrides）中指定的字符串，实际上在 MyBatis 的源码中，MyBatis 是通过移除后拼接来实现的。

我们使用 trim 元素实现具有动态插入功能的 insert 语句，如下：

```
<insert id=”insertByRecord” parameterType=”com.wyz.entity.UserDO”>
insert into user
```

```
<trim prefix="(" suffix=")" suffixOverrides=",">
<if test="user.userId != null">
user_id,
</if>
<if test="user.name != null">
name,
</if>
<if test="user.age != null">
age,
</if>
<if test="user.gender != null">
gender,
</if>
<if test="user.idType != null">
id_type,
</if>
<if test="user.idNumber != null">
id_number,
</if>
</trim>
<trim prefix="values (" suffix=")" suffixOverrides=",">
<if test="user.userId != null">
#{user.userId, jdbcType=INTEGER},
</if>
<if test="user.name != null">
#{user.name, jdbcType=VARCHAR},
</if>
<if test="user.age != null">
#{user.age, jdbcType=INTEGER},
</if>
<if test="user.gender != null">
#{user.gender, jdbcType=VARCHAR},
</if>
<if test="user.idType != null">
#{user.idType, jdbcType=INTEGER},
</if>
<if test="user.idNumber != null">
#{user.idNumber, jdbcType=VARCHAR},
</if>
</trim>
</insert>
```

同样的我们来定义接口方法：

```
int insertByRecord(@Param(”user”) UserDO userDO);
```

最后我们来写一个测试方法：

```
public void testInsertByRecord() throws IOException {
  Reader mysqlReader = Resources.getResourceAsReader(”mybatis-config.xml”);
  SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(mysqlReader);
  SqlSession sqlSession = sqlSessionFactory.openSession();
  UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

  UserDO userDO = new UserDO();
  userDO.setUserId(5);
  userDO.setName(”赵六”);
  userDO.setAge(1);
  userDO.setGender(”F”);
  userDO.setIdType(1);
  userDO.setIdNumber(”12345678901234567 8”);

  int count = userMapper.insertByRecord(userDO);

  sqlSession.commit();
  sqlSession.close();
}
```

我们执行测试代码，来观察控制台的输出：

3)实际上，我们前面看到 where 元素和 set 元素都应该算是 trim 元素的子元素，毕竟在 MyBatis 的实现中，**where 元素的实现类 WhereSqlNode 和 set 元素的实现类 SetSqlNode 都继承自 trim 元素的实现类 TrimSqlNode**，继承关系如下：

mark:3024:0:0:0:q75.awebp#?w=1024&h=748&s=24084&e=png&b=1e1f22)

foreach 元素
---

foreach 元素提供了在映射器中遍历集合（对应 Java 中的 List，Set）和字典的能力（对应 Java 中的 Map）的能力，通常我们会在构建 in 条件语句中使用。foreach 元素提供了 6 个属性：

属性	说明
collection	传入的集合或字典的名称（即入参名称）
item	集合中迭代元素的别名
index	当前元素在集合中的下标（从 0 开始）
open	指定开始处的字符串
close	指定结尾处的字符串
separator	指定元素间分割的字符串

假设我们有一个查询页面，其中的一个查询条件为证件类型，并且证件类型允许多选，因此我们在处理查询语句时需要考虑到传入多个证件类型的场景，此时最容易想到的方式是使用 in 来构建条件语句。

我们先来写 Mapper 接口中的方法：

```
List<UserDO> selectByIdTypes(@Param(”idTypes”) List<Integer> idTypes);
```

接着来写映射器中的 SQL 语句：

```
<select id=”selectByIdTypes” resultType=”com.wyz.entity.UserDO”>
select * from user
where id_type in
```

```
<foreach collection="idTypes" item="idType" separator="," open="(" close=")">
#{idType, jdbcType=INTEGER}
</foreach>
</select>
```

最后我们来写单元测试：

```
public void testSelectByIdTypes() throws IOException {
  Reader mysqlReader = Resources.getResourceAsReader("mybatis-config.xml");
  SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(mysqlReader);
  SqlSession sqlSession = sqlSessionFactory.openSession();
  UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

  List<Integer> idTypes = List.of(1, 2);
  List<UserDO> users = userMapper.selectByIdTypes(idTypes);
}
```

执行单元测试，来看控制台输出的 SQL 语句：

最后我们来看一下在上面的案例中没有出现的 index 属性，由于使用场景极少，大家了解即可。index 属性在集合中为元素在集合中下标的别名，在字典中为 key 的别名，这个我们就不写例子了（实在想不到比较合适的使用场景），大家可以自行测试下，比如在 foreach 元素中引入 index 属性，并在查询条件中使用，然后观察控制台输出的 Parameters，虽然没有任何实际意义，但是能完整的展示 index 属性。

使用 foreach 元素实现批量插入

foreach 元素的功能非常强大，我们不仅仅可以用 foreach 元素来构建 in 条件

语句，也可以使用 foreach 元素来实现批量插入。

首先来类 Mapper 接口中的方法：

```
int batchInsert(@Param("users") List<UserDO> users);
```

接着我们来写 MyBatis 映射器中的 SQL 语句：

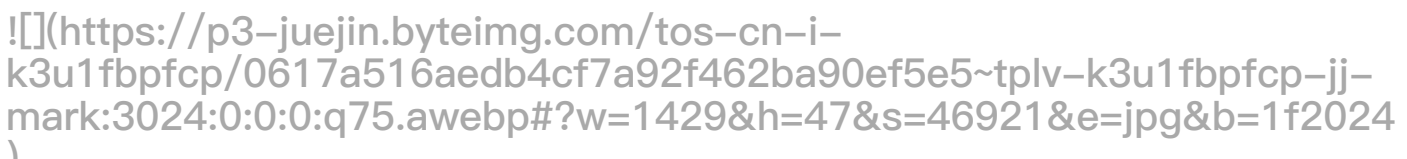
```
<insert id="batchInsert">
insert into user(user_id, name, gender, age, id_type, id_number)
values
<foreach collection="users" item="user" open="(" separator="), ("
close=")">
#{user.userId, jdbcType=INTEGER},
#{user.name, jdbcType=VARCHAR},
#{user.gender, jdbcType=VARCHAR},
#{user.age, jdbcType=INTEGER},
#{user.idType, jdbcType=INTEGER},
#{user.idNumber, jdbcType=VARCHAR}
</foreach>
</insert>
```

最后我们来写单元测试：

```
public void testBatchInsert() throws IOException {
  Reader mysqlReader = Resources.getResourceAsReader("mybatis-
config.xml");
  SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(mysqlReader);
  SqlSession sqlSession = sqlSessionFactory.openSession();
  UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

  List<UserDO> users = new ArrayList<>();
  for (int i = 0; i < 2; i++) {
```

```
    UserDO user = new UserDO();
    user.setUserId(3 + i);
    user.setName("批量" + (i + 1));
    user.setAge(18 + i);
    user.setGender("F");
    user.setIdType(1);
    user.setIdNumber("11010319981112325" + i);
    users.add(user);
  }
  userMapper.batchInsert(users);
  sqlSession.commit();
  sqlSession.close();
}
```

我们来观察控制台输出的 SQL 语句：



除了以上两种简单的用法外，foreach 元素还可以帮助我们实现更多复杂的 SQL 语句，比如之前我利用 foreach 元素实现过联表 union all 之后的条件查询语句，更多的用法大家可以自行探索。

choose 元素
---

choose 元素的功能与 Java 中的 switch 相似，choose 元素包含两个子元素：when 元素和 otherwise 元素。choose 元素连同它的子元素在用法上与 Java 中的`switch...case...default`语句的用法相同。

比如说我们有个需求，根据传入的条件查询用户信息，允许传入姓名和性别，传入了姓名就按姓名查询（忽略性别），如果没有传入姓名就按照性别查询，如果两者都没有传入，就查询证件类型为 1 的用户，这里我们使用 UserDO 作为入参，那么 MyBatis 映射器中的 SQL 语句我们可以这么写：

```
<select id="selectByUserNameOrGender"
```

```
parameterType="com.wyz.entity.UserDO"
resultType="com.wyz.entity.UserDO">
  select * from user
  <where>
    <choose>
      <when test="user.name != null">
        and name = #{user.name, jdbcType=VARCHAR}
      </when>
      <when test="user.gender != null">
        and gender = #{user.gender, jdbcType=VARCHAR}
      </when>
      <otherwise>
        and id_type = 1
      </otherwise>
    </choose>
  </where>
</select>
```

接着我们来写单元测试的代码：

```
public void testSelectByUserNameOrGender() throws IOException {
  Reader mysqlReader = Resources.getResourceAsReader("mybatis-
config.xml");
  SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(mysqlReader);
  SqlSession sqlSession = sqlSessionFactory.openSession();
  UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

  UserDO user = new UserDO();
  user.setName("张三");
  user.setGender("F");
  List<UserDO> users = userMapper.selectByUserNameOrGender(user);
}
```

单元测试中，我们构建查询参数时同时传入了姓名和性别，最后我们来执行单元测试，观察控制台的输出：

可以看到，在控制台输出的 SQL 语句的查询条件中只有 name 这一个条件，这也符合我们对 choose 元素的认知。

Tips：你可以在构建查询参数时删除为 name 字段赋值的 Java 代码来观察控制台输出的 SQL 语句，以及同时删除为 name 和 gender 赋值的 Java 代码，再来观察控制台输出的 SQL 语句。

bind 元素
---

bind 元素允许你在 MyBatis 映射器中创建一个自定义局部变量，最常见的场景就是我们在模糊查询中使用 bind 元素。

通常我们在 MyBatis 中构建模糊查询语句，可能会使用如下几种方式：

* 直接在 Java 代码中为参数添加“%”后传递到 SQL 语句中；
* 直接在 MyBatis 映射器文件中拼接“%”，例如：select * from user where name like '${surname}%'；
* 使用 MySQL 提供的 concat 函数（Oracle 中使用 “||”）拼接“%”，例如：select * from user where name like concat(#{surname, jdbcType=VARCHAR}, '%')。

除了以上几种方法外，我们还可以使用 bind 元素来构建 MyBatis 映射器中的局部变量，来实现模糊查询的功能。例如，我们需要查询某个姓氏的用户有多少，可以这样构建 MyBatis 映射器中的 SQL 语句：

```
<select id="selectBySurname" resultType="com.wyz.entity.UserDO">
  <bind name="bindValue" value="surname + '%'" />
  select * from user where name like #{bindValue}
</select>
```

我们来写单元测试的代码：

```
public void testSelectBySurname() throws IOException {
```

```
  Reader mysqlReader = Resources.getResourceAsReader("mybatis-config.xml");
  SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(mysqlReader);
  SqlSession sqlSession = sqlSessionFactory.openSession();
  UserMapper userMapper = sqlSession.getMapper(UserMapper.class);


  List<UserDO> users = userMapper.selectBySurname("张");
  sqlSession.commit();
  sqlSession.close();
}
```

最后我们来看控制台输出的 SQL 语句：

MyBatis 映射器中允许使用 bind 元素来创建多个自定一局部变量，如：

```
<select id="selectBySurname" resultType="com.wyz.entity.UserDO">
  <bind name="bindValue" value="surname + '%'" />
  <bind name="bindValue2" value="'%' + surname + '%'" />
  select * from user where name like #{bindValue}
</select>
```

好了，今天的内容就到这里了，如果本文对你有帮助的话，希望多多点赞支持，如果文章中出现任何错误，还请批评指正。**最后欢迎大家分享硬核 Java 技术的金融摸鱼侠**[王有志](http://cxyroad.com/"https://www.yuque.com/wangyouzhi-u3woi/wvkm9u/uw8c5iyvpgnqpzmg")，我们下次再见！

原文链接: https://juejin.cn/post/7365074668208013362