# Dichroite

一个基础的数据访问层实现，依托于Fluorite自动配置所运行而与Fluorite协同使用，内部原理参考Mybatis，使用方法与Mybatis大体相同，但仅支持注解式。

## 使用

|  |
| --- |
| @RunnerAs(debug = false , debugFormAop = false, debugFromTransaction = false)  @EnableTransactionManagement  @MapperScanner("com.zy.mapper")  public class App {  public static void main(String[] args) {  ConfigurableApplicationContext run = FluoriteApplication.run(App.class, args);  run.close();  }  } |

## 原理概述

### 一、Dichroite启动与初始化

1. 由标注的@ MapperScanner导入Mapper接口扫描器和自动配置类启动。
2. 在未开启事务环境是在非事务环境下运行，在开启事务后则在事物环境下运行，依据事务环境中注册的相关事务对象来保证程序的同步性

### 二.SQL构建

方法有动态sql注解时才会启用动态SQL式的入参解析和SQL构建

1) 静态SQL

(1) 支持的参数捕获：#{propretyName}使用于单个对象属性参数、#{parameterIndex}使用于一或多个基本数据类型参数或单个List集合或数组参数、#{mapKey}使用于单个Map类型参数。

(2) 对于一个方法参数中有基本数据类型又有数组、集合类型之中任意一个或多个参数的情况建议封装为类，通过#{propretyName}引用，即若集合或数组类型作为多个入参当中的一员时，无法解析其中的数据，只能将这个集合或数组其封装对象注入SQL

(3) Method传入的参数对象其原始属性名会被舍弃，需在解析方法时保存其有序的属性集合。所以对于可将参数值直接注入SQL的参数使用#{argsIndex}捕获，argsIndex自0开始

2) 动态SQL

入参限制：有且只有一个参数且此参数封装了所需可能应用于SQL中的属性

本质：为MyBatis动态SQL的另一种实现方式，只是封装了构建SQL的操作

1. 入参必须标注@BeanMapping注解，而其属性可通过@Column注解描述其对应的表字段名和属性优先级。在进行SQL构建中排除空值空集的属性，而其他的属性则为可应用于构建SQL的有效属性,通过。
2. 解析动态update语句时，根据@Column中的level值来确定非默认属性值之中谁作为查询字段。梯度按升序排列，相对靠前的属性为查询字段，相对靠后的属性的为被更新字段。但查询字段只限定为一个字段，其余字段则视为需要更新的字段
3. @QueryWhere注解的mode属性可控制查询语句的拼接逻辑。

|  |
| --- |
| 串行等同于MyBatis的 |
| <where>  <choose>  <when></when>  <otherwise></otherwise>  </choose>  </where> |
| 并行等同于MyBatis的 |
| <where>  <if></if>  <if></if>  <where> |

### 三.返回值封装

针对反射后泛型擦除的原因导致Mapper方法定义的返回值泛型失效，所以在返回值创建时严格按照返回值泛型构造返回值对象。使用对于的TypeHandler实现设值

|  |  |
| --- | --- |
| 支持的对象类型： | |
| Map<String,T> | 封装单行数据，key为数据库字段名，value为字段值 |
| Pojo对象 | 封装单行数据，属性名为字段名，可通过@Column进行字段名映射。属性值为字段值，若结果集中为找到与字段名匹配的属性名则此属性为默认值 |
| List<T> | T：类型可为Map或pojo对象类型 |
| 封装多行数据 |
| Map<T1,T2> | 与Map<String,T>不同，可封装多行数据，使用@MapKey指定结果集中哪些个属性作为Map的key，而单行结果集作为其value值 |
| 可行的返回值：Map<String, Map<String,?>>，Map<String, JavaBean> |

### 四、待定实现

1. 命名空间缓存
2. 插件功能