快速入门

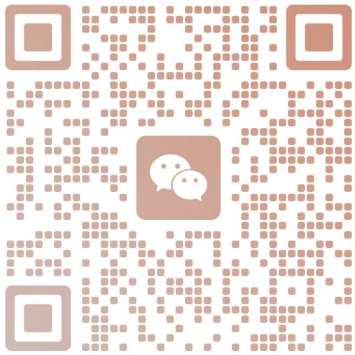
简介

DREAM（https://github.com/moxa-lzf/dream）是一个基于翻译的以技术为中心，辐射业务持久层框架，理论上支持用开发者习惯的SQL语法，完成对不同数据库的兼容

莫小哀

江苏 南京

联系微信：



扫一扫上面的二维码图案，加我为朋友。

特性

跨平台：支持采用mysql语法在非mysql环境下执行，并提供接口自定义翻译

轻量级：整个框架不依赖任何第三方框架

高性能：匠心独运的架构设计，性能基本达到极限

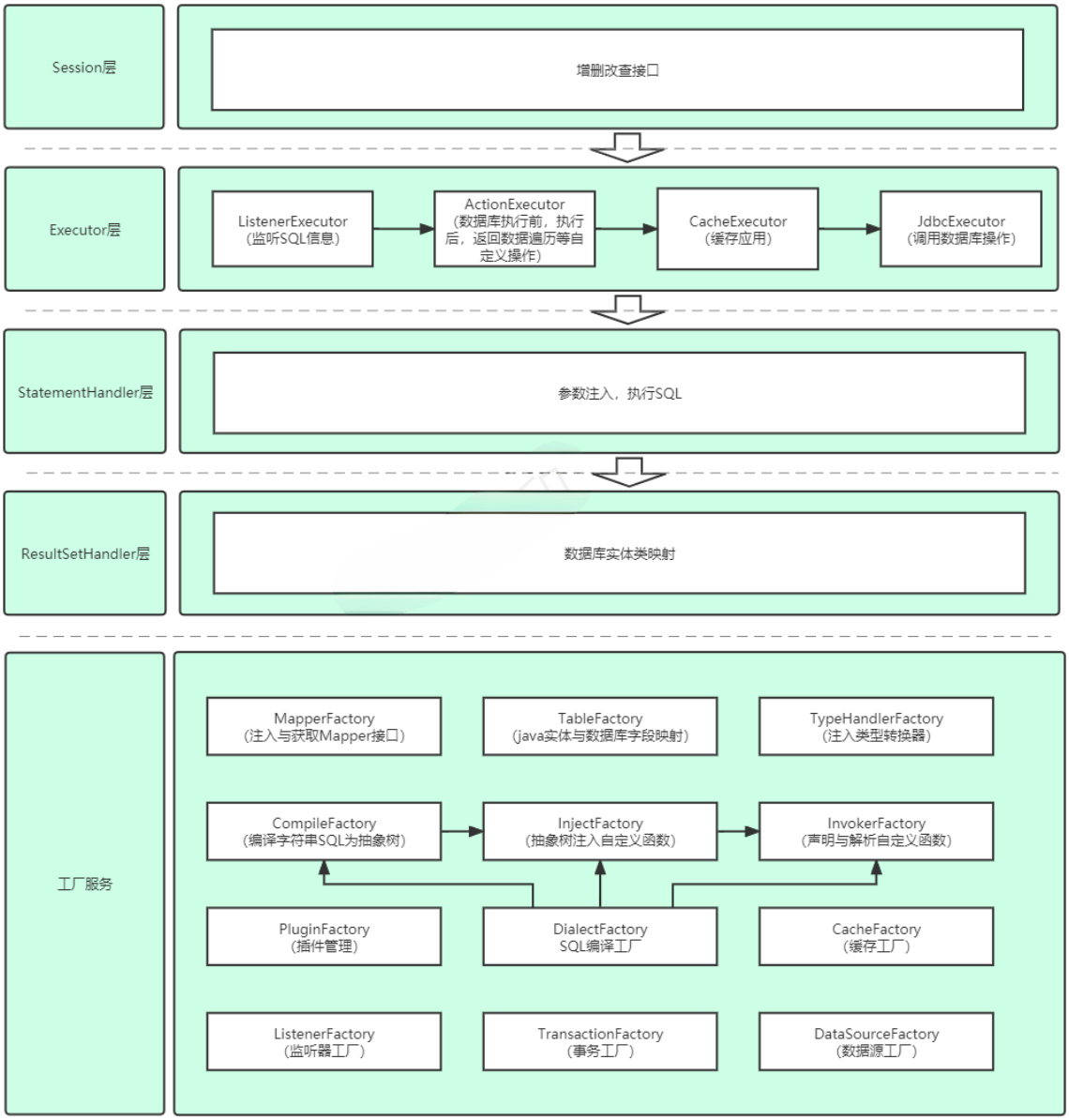
函数化：深度解析SQL特有，sql函数化，封装业务，简化sql写法（数据映射，缓存，数据权限，逻辑删除，关键字拦截，多租户等都基于此实现）

缓存机制：基于表的缓存，一切数据皆可缓存，若数据修改皆经过框架，保证读到的数据与数据库一致

校验器：基于注解的校验，可自定义化，列如：完成数据库插入唯一性校验等

开箱即用：数据权限，逻辑删除，多租户，多数据源，参数值注入（默认值，加密），主键序列、查询字段值提取（解密，字典等）

系统架构



优势

精简查询字段

```
select column1, column2, ..., columnN
from table_name
where 复杂条件
```

可改写成

```
select @all()
from table_name
where 复杂条件
```

采用自定义函数，根据实体属性判断查询的字段，若实体属性修改，同步修改SQL查询字段

屏蔽简单查询条件

对于mybatis语法

```
where 1=1
<if test="name!=null and name !=''">
    and user.name like('%',name,'%')
</if>
<if test="age!=null">
and age in
<foreach item="item" index="index" collection="age"
         open="(" separator="," close=")">
    #{item}
</foreach>
</if>
```

可改写成

```
public class UserCondition {
    @Conditional(table = "user", value = ContainsCondition.class)
    private String name;

    @Conditional(value = InCondition.class)
    private List<Integer> age;
}
```

采用注解生成条件有着更容易的阅读性，而且实体类只解析一次

java编写SQL

支持常见的任意字符串SQL，包括子查询，相同表做关联，皆可改写一下形式，几乎支持所有的MySQL函数，包括字符串，数学，日期，case语句等函数

```
select(user.dept_id)
                .from(user.as("u"))
                .leftJoin(blog)
                .on(user.id.eq(blog.user_id))

.where(user.id.eq(1).and(user.name.like("12")).and(user.tenant_id.notIn(1,2,3)))
                .groupBy(user.dept_id)
                .having(user.del_flag.eq(0))
                .orderBy(user.del_flag.desc())
                .limit(2,3)
```

无感屏蔽映射

使用mybatis需要用resultMap写Java属性与数据库字段的映射

1:实体类与数据库字段映射

```

@Table("user")
public class UserXXX {
```

```
    @Id
    @Column("id")
    private Integer id;
    @Column("name")
    private String name;
}

@Table("blog")
public class BlogXXX {
    @Id
    @Column("id")
    private Integer id;
    @Column("name")
    private String name;
}
```

2：引用数据库部分字段

```
@View(UserXXX.class)
public class User {
    private Integer id;
    private String name;
    private List<Blog> blogList;
}

@View(BlogXXX.class)
public class Blog {
    private Integer id;
    private String name;
}
```

额外增加View注解，是考虑到查询的数据大部分都是表的部分字段

无感屏蔽多租户

考虑同一个库，同一个schema情况，将现有项目改写成多租户，实现成本是多少，可能会说成本太大啦，所有SQL基本上都要翻新，而dream却给了你0成本方案，既然无感知，成本自然为0

查询用户表user和文章表blog的前一条数据

```
SELECT *
FROM (
         SELECT u.id,
                u.NAME,
                u.age,
                u.email,
                b.id   bId,
                b.NAME bName
         FROM USER u
                  LEFT JOIN blog b ON b.user_id = u.id
     ) t_tmp LIMIT 1
```

若用户表和文章表都存在租户字段，将其改造为多租户，dream可以让你不用修改当前SQL，在启动类添加开启多租户插件即可自动将其改造成多租户

```
SELECT *
FROM (
         SELECT u.id,
                u.NAME,
                u.age,
                u.email,
                b.id   bId,
                b.NAME bName
         FROM USER u
                  LEFT JOIN blog b ON (b.user_id = u.id)
             AND b.tenant_id = ?
         WHERE u.tenant_id = ?) t_tmp LIMIT 1
```

dream的识别是高强度的，不会因为SQL复杂，漏加任何租户条件，那性能如何？是等价于直接写租户条件的，无性能损耗

无感屏蔽数据权限

采用mybatis方案进行数据权限隔离，会在where条件注入 ${权限条件}，是否可以不写${权限条件}，一样完成数据权限注入，这样实现才是真正意义上的权限SQL与业务SQL解耦

同样SQL，需要注入数据权限，假如：查询自己所在部门

```
SELECT *
FROM (
         SELECT u.id,
                u.NAME,
                u.age,
                u.email,
                b.id   bId,
                b.NAME bName
         FROM USER u
                  LEFT JOIN blog b ON b.user_id = u.id
     ) t_tmp LIMIT 1
```

开启数据权限插件

```
SELECT *
FROM (
         SELECT u.id,
                u.NAME,
                u.age,
                u.email,
                b.id   bId,
                b.NAME bName
         FROM USER u
                  LEFT JOIN blog b ON b.user_id = u.id
         WHERE u.dept_id = 1
     ) t_tmp LIMIT 1
```

u.dept_id=1是开发者自己注入的数据权限，不要担心，dream会解析出别名告诉开发者，完成数据权限注入，此时，SQL非常清爽，性能等价于在SQL直接写注入权限条件

无感屏蔽逻辑删除

有些字段是需要进行逻辑删除的，有些字段不需要，区别在于表是否加了逻辑字段，假如：未来有个需求，这个表不需要逻辑删除，另一张表需要逻辑删除，代码修改必不可少，幸运的是有些框架提供了逻辑删除，自动将delete语句改成update语句，代码量基本上无改动，事实上，表与表之间关联条件以及where条件是否都加了逻辑条件，仍然需要一步一步改。

同样的SQL，假设用户表user和文章表都存在逻辑删除字段，改造为逻辑删除

```
SELECT *
FROM (
         SELECT u.id,
                u.NAME,
                u.age,
                u.email,
                b.id   bId,
                b.NAME bName
         FROM USER u
                  LEFT JOIN blog b ON b.user_id = u.id
     ) t_tmp LIMIT 1
```

开启逻辑删除插件

```
SELECT *
FROM (
         SELECT u.id,
                u.NAME,
                u.age,
                u.email,
                b.id   bId,
                b.NAME bName
         FROM USER u
                  LEFT JOIN blog b ON (b.user_id = u.id)
             AND b.del_flag = 0
         WHERE u.del_flag = 0
     ) t_tmp LIMIT 1
```

完成了SQL操作的逻辑字段追加，删除数据库里的逻辑字段就不采用逻辑删除，同样，希望某张表采用逻辑删除，加个逻辑字段即可，代码不需要做任何修改，性能等价于直接写逻辑删除条件，性能无损耗

数据库关键字处理

数据库关键字，不是关键字可以不加特殊符号，关键字必须要加，dream提供方案，SQL语句可以不加特殊符号对关键字处理，一样可以正常执行

SQL语句，若user和id为关键字，不做处理会执行报错，正确做法需要对user和id加特殊符号

```
SELECT *
FROM (
         SELECT u.id,
                u.NAME,
                u.age,
                u.email,
                b.id   bId,
                b.NAME bName
         FROM USER u
                  LEFT JOIN blog b ON b.user_id = u.id
     ) t_tmp LIMIT 1
```

开启关键字插件

```
SELECT *
FROM (
         SELECT u.`id`,
                u.NAME,
                u.age,
                u.email,
                b.`id` bId,
                b.NAME bName
         FROM `USER` u
                  LEFT JOIN blog b ON b.user_id = u.`id`
     ) t_tmp LIMIT 1
```

自动完成对user和id关键字处理，性能等价于直接写关键字处理

支持数据库

语法以MySQL为标准，将SQL翻译成抽象树，进而在不同数据库下做不同翻译，MySQL，PG已在生产环境下测试

MySQL，SqlServer， PostgreSQL，Oracle

用法教程

内置@函数

?

用法

与参数有关的函数，将参数改成?

举例

```
@Mapper
public interface UserMapper {
    @Sql("select id, name, age,email from user where name = @?(name)")
    User findByName(String name);
}
```

测试

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = BootApplication.class)
public class QueryTest {
    @Autowired
    private UserMapper userMapper;

    @Test
    public void test() {
        User user = userService.findByName("Jone");
    }
}
```

控制台输出

```
SQL:SELECT id,name,age,email FROM user WHERE name=?
PARAM:[Jone]
```

rep

用法

与参数有关的函数，将参数带入sql

举例

```
@Mapper
public interface UserMapper {
    @Sql("select id, name, age,email from user where name = @rep(name)")
    User findByName2(String name);
}
```

测试

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = BootApplication.class)
public class QueryTest {
    @Autowired
    private UserMapper userMapper;

    @Test
    public void test() {
        User user = userMapper.findByName2("'Jone'");
    }
}
```

控制台输出

```
SQL:SELECT id,name,age,email FROM user WHERE name='Jone'
PARAM:[]
```

foreach

用法

遍历集合或数组

举例：删除数组

```
@Mapper
public interface UserMapper {
    @Sql("delete from user where id in (@foreach(list))")
    int delete(List<Integer> idList);
}
```

测试

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = BootApplication.class)
public class DeleteTest {
    @Autowired
    private UserMapper userMapper;

    @Test
    public void deleteById2() {
        templateMapper.deleteByIds(User.class, Arrays.asList(1, 2, 3, 4, 5, 6));
    }
}
```

控制台输出

```
SQL:DELETE FROM user WHERE id IN (?,?,?,?,?,?)
PARAM:[1, 2, 3, 4, 5, 6]
```

non

用法

空条件剔除

举例

```
@Mapper
public interface UserMapper {
    @Sql("update user set @non(name=@?(user.name),age=@?(user.age),email=@?
(user.email)) where id=@?(user.id)")
    Integer updateNon(User user);
}
```

注：空字符串不为空

测试

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = BootApplication.class)
public class UpdateTest {
    @Autowired
    private UserMapper userMapper;

    @Test
    public void updateNonId2() {
        User user = new User();
        user.setId(1);
        user.setName("hli");
        user.setEmail("");
        userMapper.updateNon(user);
    }
}
```

控制台输出

```
SQL:UPDATE user SET name=?,email=? WHERE id=?
PARAM:[hli, , 1]
```

not

用法

空条件剔除

注：空字符串为空

all

用法

1：根据java属性识别查询字段

2：根据SQL查询前后文排除字段

举例

```
@Mapper
public interface UserMapper {
    @Sql("select @all(),'hello' name from user")
    List<User> findAll();
}
```

注：后文查询 'hello'

测试

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = BootApplication.class)
public class QueryTest {
    @Autowired
    private UserMapper userMapper;

    @Test
    public void test3() {
        List<User> userList = userMapper.findAll();
        userList.forEach(System.out::println);
    }
}
```

控制台输出

```
SQL:SELECT user.id,user.age,user.email,'hello' name FROM user
PARAM:[]
TIME:25ms

User{id=1, name='hello', age=18, email='test1@baomidou.com'}
User{id=2, name='hello', age=20, email='test2@baomidou.com'}
User{id=3, name='hello', age=28, email='test3@baomidou.com'}
User{id=4, name='hello', age=21, email='test4@baomidou.com'}
User{id=5, name='hello', age=24, email='test5@baomidou.com'}
```

table

用法

自动将表拼接成关联条件

举例

```
public interface UserMapper {
    @Sql("select @all() from @table(user,blog)")
    List<User> selectAll3();
}
```

测试

```
 @Test
public void test8(){
        List<User> userList=userMapper.selectAll3();
        }
```

控制台输出

```
执行SQL:SELECT
user.id,user.name,user.age,user.email,blog.id,blog.name,blog.user_id FROM  user
LEFT JOIN blog ON user.id=blog.user_id
执行参数:[]
执行用时：17ms
```

注解

Table

用法

绑定类对象与数据表

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Table {
    String value();

    boolean mapping() default true;
}
```

注解属性	描述
value	指定绑定的数据表
mapping	是否解析当前类与数据库表绑定

举例

```
@Table("user")
public class User {
}
```

Id

用法

声明表主键

注：应用于仅当表有且仅有一个主键

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface Id {

}
```

举例

```
@Table("user")
public class User {
    @Id
    @Column("id")
    private Integer id;
}
```

Column

用法

绑定类对象属性与数据表字段

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface Column {
    String value();

    int jdbcType() default Types.NULL;
}
```

注解属性	描述
value	绑定的数据表字段
jdbcType	数据表字段类型

举例

```
@Table("user")
public class User {
    @Id
    @Column("id")
    private Integer id;
    @Column(value = "name", jdbcType = Types.VARCHAR)
    private String name;
    @Column("age")
    private Integer age;
    @Column("email")
    private String email;
}
```

Join

用法

指明表于表关联关系，目的为消灭sql语句写表与表关联而生，@函数table基于此

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface Join {
    String column();

    String joinColumn();

    JoinType joinType() default JoinType.LEFT_JOIN;
}
```

注解属性	描述
column	该表字段名
joinColumn	关联表的字段名
joinType	关联类型

注：数据表名根据修饰的类属性判断

举例

```
@Table("user")
public class User {
    @Id
    @Column("id")
    private Integer id;
    @Column(value = "name", jdbcType = Types.VARCHAR)
    private String name;
    @Column("age")
    private Integer age;
    @Column("email")
```

```
    private String email;
    @Join(column = "id", joinColumn = "user_id", joinType =
Join.JoinType.LEFT_JOIN)
    private List<Blog> blogList;
}
```

注：类Blog必须有注解Table

View

用法

视图概念，截取数据表的部分数据操作

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface View {
    Class<?> value();
}
```

注解属性	描述
value	来源数据表映射类

注：View修饰的类属性，必须和Table修饰的属性一致，才能做到映射

举例

仅仅就想查询id以及name字段，email与age不查询

```
@View("user")
public class UserView2 {
    private Integer id;
    private String name;
}
```

```
  @Sql("select @all() from user")
    List<UserView2> selectAll2();
```

测试

```
  @Test
public void test7(){
        List<UserView2> userViews=userMapper.selectAll2();
        }
```

控制台输出

```
SQL:SELECT user.id,user.name FROM user
        PARAM:[]
        TIME:33ms
```

注：做到修改字段就可以间接修改SQL语句目的，存在情况，view字段与table字段一致，但不想查询，或者不想多表查询，可以使用Ignore忽略此字段

Ignore

用法

查询结果自动映射到对象，忽视此字段

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface Ignore {

}
```

举例

```
@View("user")
public class UserView {
    private Integer id;
    private String name;
    @Ignore
    private String email;
}
```

注：email存在Ignore，email数据为空

Mapper

用法

声明接口为可执行接口

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Mapper {
    Class<?> value() default NullObject.class;
}
```

属性名	描述

属性名	描述
value	根据java类生成的sql，value指定类的方法名称若为无参公共方法必须为mapper对应的接口方法名一致，且返回值类型必须是字符串或者ActionProvider类

举例

```
@Mapper
public interface UserMapper {
}
```

Sql

用法

方法绑定SQL

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Sql {
    String value();

    boolean cache() default true;

    boolean listener() default true;
}
```

属性名	描述
value	绑定的SQL语句
cache	是否进行数据缓存读取
listener	是否执行监听

举例

```
@Mapper
public interface UserMapper {
    @Sql("select id, name, age,email from user where name = @?(name)")
    User findByName(String name);
}
```

Param

用法

绑定参数名名称

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.PARAMETER)
public @interface Param {
    String value();
}
```

属性名	描述
value	参数名称

PageQuery

用法

分页

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface PageQuery {
    boolean offset() default false;

    String value() default "page";
}
```

属性名	描述
offset	是否使用offset分页，默认limit
value	Page对象地址

举例

```
@Mapper
public interface UserMapper {
    @Sql("select id, name, age,email from user order by id")
    @PageQuery("page")
    List<User> findByPage(@Param("page") Page page);
}
```

测试

```
@Test
public void testPage(){
        Page page=new Page(1,1);
        List<User> userList=userMapper.findByPage(page);
        page.setRows(userList);
        System.out.println("总数："+page.getTotal());
        }
```

控制台输出

```
SQL:SELECT id,name,age,email FROM user ORDER BY id LIMIT ?,?
PARAM:[0, 1]
SQL:SELECT COUNT(1) FROM user
PARAM:[]
TIME:24ms
TIME:36ms
总数：5
```

Extract

用法：对查询的值做处理，列如，解密，字段脱敏，反查字典等操作

```
public @interface Extract {
    Class<? extends Extractor> value();
}
```

属性名	描述
value	提取的具体操作类

```
public interface Extractor {
    void extract(MappedStatement mappedStatement, MappedColumn mappedColumn,
Object value, ObjectFactory objectFactory);
}
```

参数名	描述
mappedStatement	编译后的方法
mappedColumn	字段的所有信息
value	数据库查询的值
objectFactory	反射工厂，用来给字段填充值

JPA模板

注解

Wrap

用法：为模板查询服务，字段值修改，列如：填充默认值，字段加密等操作

```
public @interface Wrap {
    Class<? extends Wrapper> value();

    WrapType type() default WrapType.INSERT_UPDATE;
}
```

属性名	描述
value	处理的实现类
type	处理时机，更新，插入，更新或插入

```
public interface Wrapper {
    Object wrap(Object value);
}
```

参数名	描述
value	参数传入值，返回为处理后的值

Conditional

用法：为模板查询服务，指定生成的where条件

```
public @interface Conditional {
    String table() default "";

    boolean filterNull() default true;

    Class<? extends Condition> value();

}
```

属性名	描述
table	条件的表名
fillterNull	为空是否剔除
value	生成条件的实现类

```
public interface Condition {
    String getCondition(String table, String column, String field);
}
```

参数名	描述
table	表名称
column	数据库字段名
field	对象属性名称

已实现的Condition

Condition类	描述
ContainsCondition	like '%?%'
EndWithCondition	like '?%'
EqCondition	=?
GeqCondition	> =?
GtCondition	> ?
InCondition	in(?,?)
LeqCondition	<=?
LtCondition	<?
NeqCondition	<>?
NotNullCondition	is not null
NullCondition	is null
StartWithCondition	like '%?'

Sort

用法：为模板查询服务，排序

```
public @interface Sort {
    String table() default "";

    Order value() default Order.ASC;

    int order() default 0;
}
```

属性名	描述
table	表名称
value	排序方式
order	指定多个排序字段时，显示优先级，越小优先级越高

模板

方法名	描述
selectById	主键查询（支持多表关联查询）
selectByIds	主键批量查询(支持多表关联查询)
selectOne	根据注解生成条件，查询一条
selectList	根据注解生成条件，查询多条
selectPage	根据注解生成条件，分页查询多条
updateById	主键更新
updateNonById	主键非空更新
insert	插入
insertFetchKey	插入并获取主键值
deleteById	主键删除
deleteByIds	主键批量删除
existById	判断主键是否存在
exist	根据注解生成条件，判断是否存在
batchInsert	批量插入，可设置批次
batchUpdateById	批量主键更新，可设置批次

selectById

多表查询

```
@View("user")
public class UserView3 {
    private Integer id;
    private String name;
    private List<BlogView> blogList;
}

@View("blog")
public class BlogView {
    private Integer id;
    private String name;
}
```

测试

```
@Test
public void testSelectById2(){
        UserView3 userView3=templateMapper.selectById(UserView3.class,1);
        System.out.println(userView3);
        }
```

控制台输出

```
SQL:SELECT `user`.`id`,`user`.`name`,`blog`.`id`,`blog`.`name` FROM `user`  LEFT JOIN `blog` ON `user`.`id`=`blog`.`user_id` WHERE `user`.`id`=?
PARAM:[1]
TIME:26ms
com.moxa.dream.boot.view.UserView3@5477a1ca
```

监听器

用法

检查、阻断SQL，修改查询数据

```
public interface Listener {
    boolean before(MappedStatement mappedStatement);

    Object afterReturn(Object result, MappedStatement mappedStatement);

    void(Exception e, MappedStatement mappedStatement);
}
```

方法名	描述
before	返回false，SQL不执行，返回空
afterReturn	返回结果为查询结果
exception	出现异常调用此处

插件

用法

基于接口代理实现，可以修改参数

```
public interface Interceptor {
    Object interceptor(Invocation invocation) throws Throwable;

    Set<Method> methods();
}
```

方法名	描述

方法名	描述
interceptor	此处进行注入插件
methods	拦截感兴趣的方法

业务

以下功能默认不开启，开启后，不需要改现有代码，使用者无感知

关键字插件

开启插件

```
 @Bean
public Inject[]injects(){
        return new Inject[]{new BlockInject()};
        }
```

用法

开发者指定的字段默认为数据库关键字，字段特殊处理

测试

```
public interface UserMapper {
    @Sql("select id, name, age,email from user where name = @?(name)")
    User findByName(String name);
}
```

注：此时user已是关键字

```
    @Test
public void test(){
        User user=userMapper.findByName("Jone");
        }
```

输出

```
执行SQL:SELECT id,name,age,email FROM `user` WHERE name=?
执行参数:[Jone]
执行用时：15ms
```

多数据源

开启注解

```
public @interface EnableShare {
    Class<? extends DataSource> value();
}
```

属性	描述
value	DataSource实现类类型

数据源配置

```yaml
dream:
  datasource:
    master:
      driverClassName: com.mysql.jdbc.Driver
      jdbcUrl: jdbc:mysql://192.168.0.3/d-open
      username: root
      password: root
      keepaliveTime: 1000
      isReadOnly: false
    slave:
      driverClassName: com.mysql.jdbc.Driver
      jdbcUrl: jdbc:mysql://192.168.0.3/d-open-6c
      username: root
      password: root
```

注：dream.datasource固定，master和slave为数据连接池名称，其他为数据连接池字段属性

数据源选择

```java
public @interface Share {
    String value();
}
```

属性	描述
value	数据连接池名称，默认是master

举例

```java

@Share("master")
public interface UserMapper {
    @Sql("select id, name, age,email from user where name = @?(name)")
    List<User> findByName(String name);

    @Share("slave")
    @Sql("select id, name, age,email from user where name = @rep(name)")
    List<User> findByName2(String name);
}
```

多租户

单库单schema

开启多租户

```
  @Bean
public TenantHandler tenantHandler(){
        return()->1;
        }

@Bean
public Inject[]injects(TenantHandler tenantHandler){
        return new Inject[]{new TenantInject(tenantHandler)};
        }

@Bean
public Interceptor[]interceptors(TenantHandler tenantHandler){
        return new Interceptor[]{new TenantInterceptor(tenantHandler)};
        }
```

注：重写TenantHandler完成租户需求

```
public interface TenantHandler {
    default boolean isTenant(MethodInfo methodInfo, TableInfo tableInfo) {
        return tableInfo.getFieldName(getTenantColumn()) != null;
    }

    default String getTenantColumn() {
        return "tenant_id";
    }

    Object getTenantObject();
}
```

方法名	描述
isTenant	判断当前方法或当前表是否应用租户MethodInfo：记录了方法的一切信息 TableInfo：记录了表的一切信息
getTenantColumn	租户字段
getTenantObject	租户值

举例

注册租户插件

```
    @Bean
public Interceptor[]interceptors(){
        return new Interceptor[]{new TenantInterceptor(()->1)};
        }
```

```
@Mapper
public interface UserMapper {
    @Sql("select* from (select id, name, age,email from user where 1=1 or 1<>2)A
inner join user u on 1=2 where A.name=@?(name)")
    Map findByName(String name);
```

测试

```
    @Test
public void test(){
        Map map=userMapper.findByName("Jone");
        }
```

控制台输出

```
执行SQL:SELECT*FROM(SELECT id,name,age,email FROM user WHERE(1=1OR 1<>2)AND
user.tenant_id=?)A INNER JOIN user u ON(1=2)AND u.tenant_id=?WHERE A.name=?
        执行参数:[1,1,Jone]
        执行用时：19ms
```

注：一旦当前方法应用租户，插入对租户字段赋值，更新赋值将失效

数据权限

用于给主表字段增加where条件，起到权限控制作用

开启数据权限

```
    @Bean
public Inject[]injects(){
        return new Inject[]{new PermissionInject(new PermissionHandler(){
@Override
public boolean isPermissionInject(MethodInfo methodInfo,TableInfo tableInfo,int
life){
        return tableInfo.getFieldName("dept_id")!=null;
        }

@Override
public String getPermission(MethodInfo methodInfo,TableInfo tableInfo,String
alias){
        return alias+".dept_id=1";
        }
        })};
        }
```

```
public interface PermissionHandler {
    boolean isPermissionInject(MethodInfo methodInfo, TableInfo tableInfo, int
life);

    String getPermission(MethodInfo methodInfo, TableInfo tableInfo, String
alias);

}
```

方法名	描述
isPermissionInject	是否对当前查询语句注入where条件，methodInfo：记录了方法的一切信息tableInfo：记录了表的一切信息life：遇到查询语句的次数（嵌套查询，life+1）
getPermission	插入的where条件，不能为空，列如：1=1methodInfo：记录了方法的一切信息tableInfo：记录了表的一切信息alias：当前查询语句主表的别名

举例

注入数据权限插件

```
    @Bean
public Interceptor[]interceptors(){
        return new Interceptor[]{new PermissionInterceptor(new
PermissionHandler(){
@Override
public boolean isPermissionInject(MethodInfo methodInfo,TableInfo tableInfo,int
life){
        return tableInfo.getFieldName("dept_id")!=null;
        }

@Override
public String getPermission(MethodInfo methodInfo,TableInfo tableInfo,String
alias){
        return alias+".dept_id=1";
        }
        })};
        }
```

```
    @Sql("select*from(select id, name, age,email from user u where 1=1 or
2=2)A")
    List<Map> findByAll();
```

测试

```
    @Test
public void test4(){
        Object v=userMapper.findByAll();
        }
```

控制台输出

```
执行SQL:SELECT * FROM (SELECT id,name,age,email FROM user u WHERE (1=1 OR 2=2)
AND u.dept_id=1) A
执行参数:[]
执行用时：22ms
```

逻辑删除

用删除标志代替真正删除，查询时将删除标志作为条件，筛选数据

开启逻辑删除

```
  @Bean
public Invoker invoker(){
        return new LogicInvoker();
        }

@Bean
public Inject[]injects(){
        return new Inject[]{new LogicInject(()->"del_flag")};
        }
```

```
public interface LogicHandler {
    default boolean isLogic(MethodInfo methodInfo, TableInfo tableInfo) {
        return tableInfo.getFieldName(getLogicColumn()) != null;
    }

    default String getPositiveValue() {
        return "1";
    }

    default String getNegativeValue() {
        return "0";
    }

    String getLogicColumn();

}
```

方法名	描述
isLogic	是否使用逻辑删除methodInfo：记录了方法的一切信息tableInfo：记录了表的一切信息
getPositiveValue	逻辑删除后的值
getNegativeValue	未删除的值
getLogicColumn	逻辑删除字段

举例

注册逻辑删除

```
    @Sql("delete from user where id in (@foreach(list))")
    int delete(List<Integer> idList);

@Sql("select user.id, user.name, user.age,user.email from user left join user u
on user.id=u.id where user.name = @?(name)")
    User findByName(String name);
```

测试删除

```
    @Test
public void deleteById3(){
        userMapper.delete(Arrays.asList(1,2,3,4,5,6));
        }
```

控制台输出

```
执行SQL:UPDATE user SET del_flag=1  WHERE (id IN (?,?,?,?,?,?)) AND del_flag=0
执行参数:[1, 2, 3, 4, 5, 6]
执行用时：10ms
```

测试查询

```
    @Test
public void test(){
        User user=userMapper.findByName("Jone");
        }
```

控制台输出

```
执行SQL:SELECT user.id,user.name,user.age,user.email FROM user  LEFT JOIN user u
ON (user.id=u.id) AND u.del_flag=0 WHERE (user.name=?) AND user.del_flag=0
执行参数:[Jone]
执行用时：18ms
```

问题

1. 当SQL长度过长，达到几千行以上，翻译报java.lang.StackOverflowError，调大Xss参数