**前言**

低代码平台在其3.8版本中引入了数据权限功能。为了增强其可拓展性、可插拔性、还能支持逻辑的灵活性，平台引入了Codewave AOP机制。

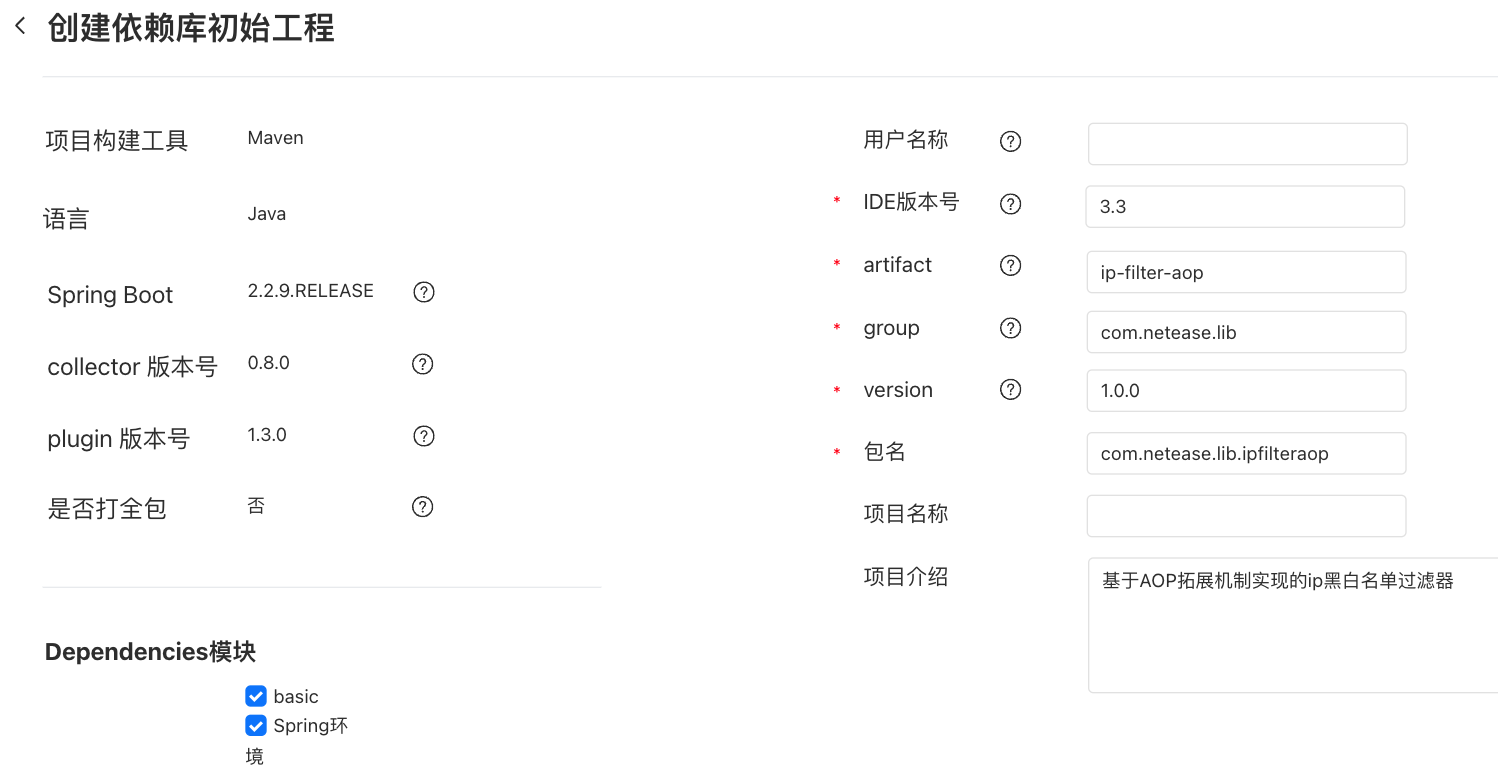
Codewave AOP机制落地形式是后端依赖库提供注解+切面，这种形式可以在不需要对平台本身进行迭代的情况下，为任意逻辑Logic和实体Entity拓展任意功能。

3.8版本平台的数据权限是AOP机制的第一个实现，也是最典型的实现。

这篇指南将通过具体案例，step-by-step地指导如何使用AOP机制为逻辑和实体拓展功能。

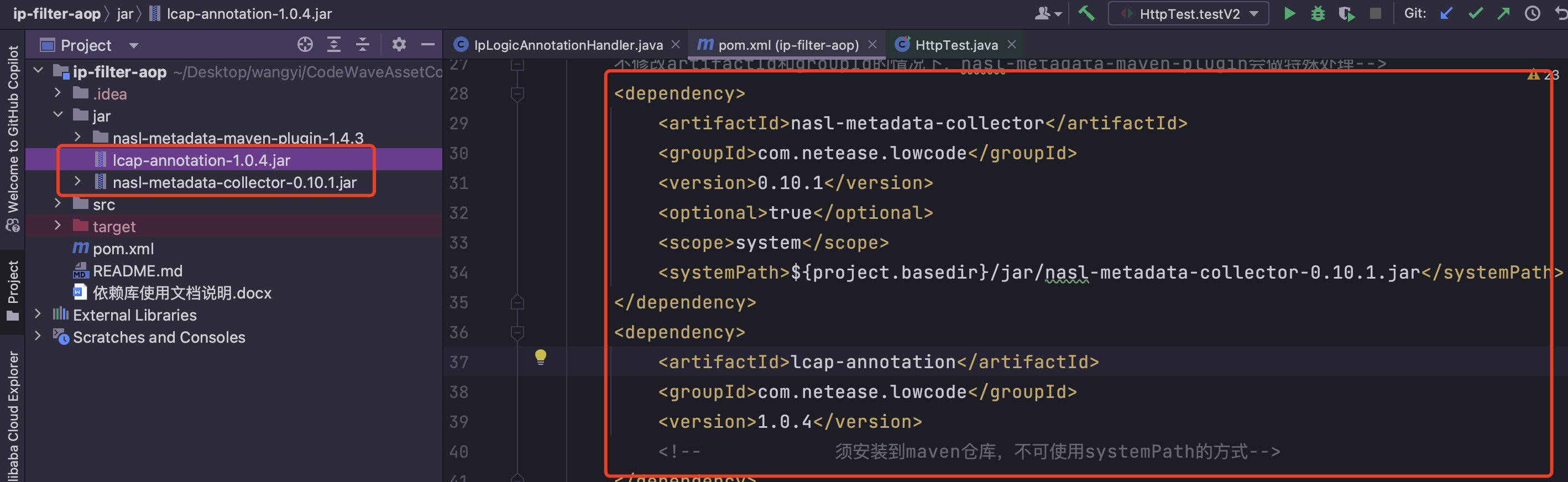
**快速开始**

1. 首先创建一个Maven Project，也可以使用<https://libraryinitializr-community1.app.codewave.163.com/init>脚手架一键生成包含Spring环境的工厂。IDE版本号即nasl.ide.version须为3.8。



2. 引入以下依赖

1. nasl-metadata-maven-plugin:1.4.3
   1. 打包工具
2. com.netease.lowcode:nasl-metadata-collector:0.10.1
   1. NASL收集器，任何后端依赖库都需要引入这个依赖。若步骤1的工程中已存在低版本的collector，需要升级至0.10.1。
3. com.netease.lowcode:lcap-annotation:1.0.4
   1. AOP的基础依赖，任何提供注解的后端依赖库都需要引入这个依赖
   2. 需要提前 git clone && mvn install。gitlab地址：<https://g.hz.netease.com/skiff-lowcode/extension/auth-library>



**逻辑切面拓展**

**案例1：为逻辑设置IP白名单**

通过这个案例讲解如何使用「逻辑注解」。可参考<https://github.com/netease-lcap/CodeWaveAssetCompetition2024>开源仓库中的ip-filter-aop依赖库。

**1、声明注解**

首先需要声明一个注解，它的落地形式是一个声明了@NaslAnnotation的public class，如下所示——

import com.netease.lowcode.annotation.LCAPLogicAnnotation;

import com.netease.lowcode.core.annotation.NaslAnnotation;

/\*\*

\* IP白名单注解 // 用于声明注解的标题，详见下图

\*/

@NaslAnnotation(

applyTo = {NaslAnnotation.Component.LOGIC} // 用于声明这是一个逻辑注解

)

public class IpLogicAnnotation extends LCAPLogicAnnotation { // 必须继承于LCAPLogicAnnotation

@NaslAnnotation.Property(

title = "是否开启IP白名单", // IDE将使用这个值作为默认开关的标题

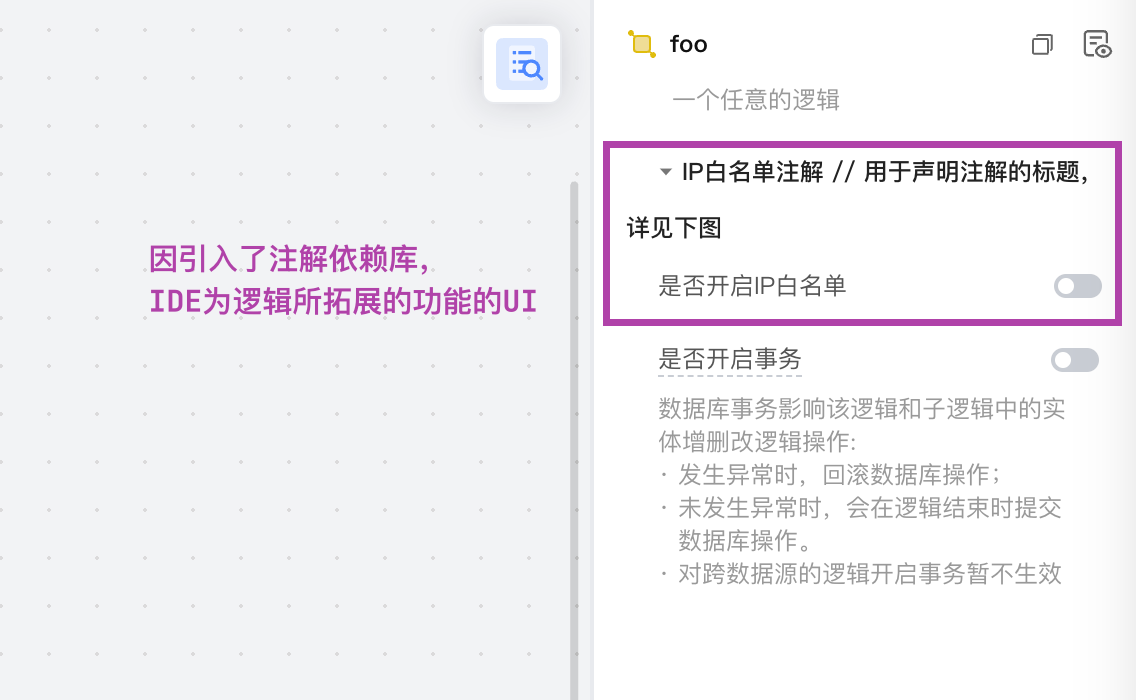
defaultValue = "false" // IDE将使用这个值作为开关的默认开启|关闭情况，useAnno的deaultValue必须为true或false

)

public Boolean useAnno; // 这个field代表的是注解的默认开关。因设计原因，public、Boolean、useAnno这三部分必须是固定的

}

至此，IDE开发者一旦在应用中引入了这个依赖库，导入时IDE会发现这个注解，并为制品的所有逻辑右侧区域增加了IP白名单注解及其默认开关的UI，如下图所示——



**自定义注解属性**

注解除了默认开关useAnno，还可以指定其他的自定义属性。示例如下：

@NaslAnnotation.Property(title = "输入框-字符串类型")

public String inputText;

@NaslAnnotation.Property(title = "开关框")

public Boolean enable;

@NaslAnnotation.Property(title = "输入框-Number类型")

public Long inputNumber;

@NaslAnnotation.Property(title = "多选框", multiple = true, options = {@NaslAnnotation.Option(value = "1", title = "1"), @NaslAnnotation.Option(value = "2", title = "2")})

public List<String> selectMultiple;

@NaslAnnotation.Property(title = "单选框", multiple = false, options = {@NaslAnnotation.Option(value = "1", title = "1"), @NaslAnnotation.Option(value = "2", title = "2")})

public List<String> selectSingle;

注意，为Boolean类型以外的@NaslAnnotation.Property指定defaultValue是无效的。

**2、实现切面处理器**

至此，注解声明方面的内容已说明完毕，有了注解还需要结合切面处理器才能发挥作用，下面开始介绍如何为开发切面处理器。

**2.1 声明逻辑切面处理器**

切面处理器分为逻辑切面处理器和实体切面处理器，在这个案例中首先介绍逻辑切面处理器。切面处理器的落地形式是一个实现了LCAPLogicAnnotationHandler的public class，如下所示——

public class IpLogicAnnotationHandler

implements LCAPLogicAnnotationHandler< // 因为这是一个逻辑切面处理器，所以必须实现这个接口

IpLogicAnnotation // 由于这是IpLogicAnnotation的处理器，所以必须在这个TypeParameter处指出它

> {

/\*\*

\* 声明了切点位置。

\* 框架层承诺：会在这个方法的返回值所代表的时机，对this.handle方法进行调用，以进行切面处理。

\* handle是一个抽象方法，开发者可以自由实现

\*

\* @return LCAPAnnotationHandlerAdvise 调用this.handle方法的时机，

\* BEFORE代表逻辑翻译出的Service被调用前，AFTER代表被调用后。

\*/

@Override

public LCAPAnnotationHandlerAdvise[] advises() {

return new LCAPAnnotationHandlerAdvise[]{LCAPAnnotationHandlerAdvise.BEFORE}; // 这是一个拦截类注解的案例，所以时机是BEFORE

}

/\*\*

\* 绑定的注解

\*/

@Override

public Class<? extends LCAPAnnotation> consume() {

return IpLogicAnnotation.class; // 由于这是IpLogicAnnotation的处理器，所以此处也必须返回它

}

/\*\*

\* 执行顺序，order越小，这个对象的handle在同一advise时机下，相比别的handler越先执行。

\* order相同时不保证执行顺序

\* 缺省为0

\*/

@Override

public Integer order() {

return LCAPLogicAnnotationHandler.super.order();

}

/\*\*

\* 处理方法 // 下文详细说明

\*/

@Override

public Object handle(Object[] args, Object result, LogicContext<IpLogicAnnotation> context) {

// TODO

}

/\*\*

\* 上报方法 // 下文详细说明

\*/

@Override

public Boolean report(List<LogicContext<IpLogicAnnotation>> logicContextList) {

return true; // 如果不需要使用"上报"机制，直接return即可

}

}

**2.2 开发切面处理器**

切面处理器handler的核心方法是handle。当低代码开发者打开了注解配置，在制品运行中，被调用的逻辑是能够与注解配置匹配，就会进入到此切面。代码示例如下：

/\*\*

\* 处理方法

\* 引入依赖库后，在右侧区域开启了注解的逻辑，在制品启动后，框架会在调用这些逻辑的this.advises()时机，调用这个方法。

\*

\* @param args 逻辑的实际参数列表

\* @param result 逻辑的实际返回值。若本方法在BEFORE那次时机被调用，那么result固定为null

\* @param context 注解上下文(含逻辑元信息)

\*/

@Override

public Object handle(

Object[] args,

Object result,

LogicContext<IpLogicAnnotation> context)

throws LCAPAnnotationLogicHandlerException {

// 这个方式可以获取到IDE中用户指定的"请求拦截提示"

// 这个案例中，假设本方法在制品运行时调用foo逻辑而进入的，这里的值会是"IP未添加到白名单，禁止访问"

// 至此，对自定义注解属性的指南便闭环了

log.info(context.getAnnotation().errMsg);

//拦截到的逻辑名称

log.info(context.getLogicName());

//逻辑描述

log.info(context.getLogicDescription());

//逻辑返回类型

log.info(context.getReturnShape());

//逻辑注解上下文

log.info(context.getProperties().toString());

List<IpRuleDto> the10IPs = WhatEverName.getIpRules(context.getLogicName())

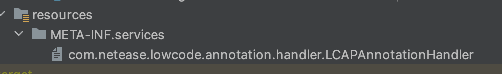
// TODO 基于context对象、逻辑参数、逻辑返回值，可以在此处自由实现各种功能。

}

**2.3 SPI声明**

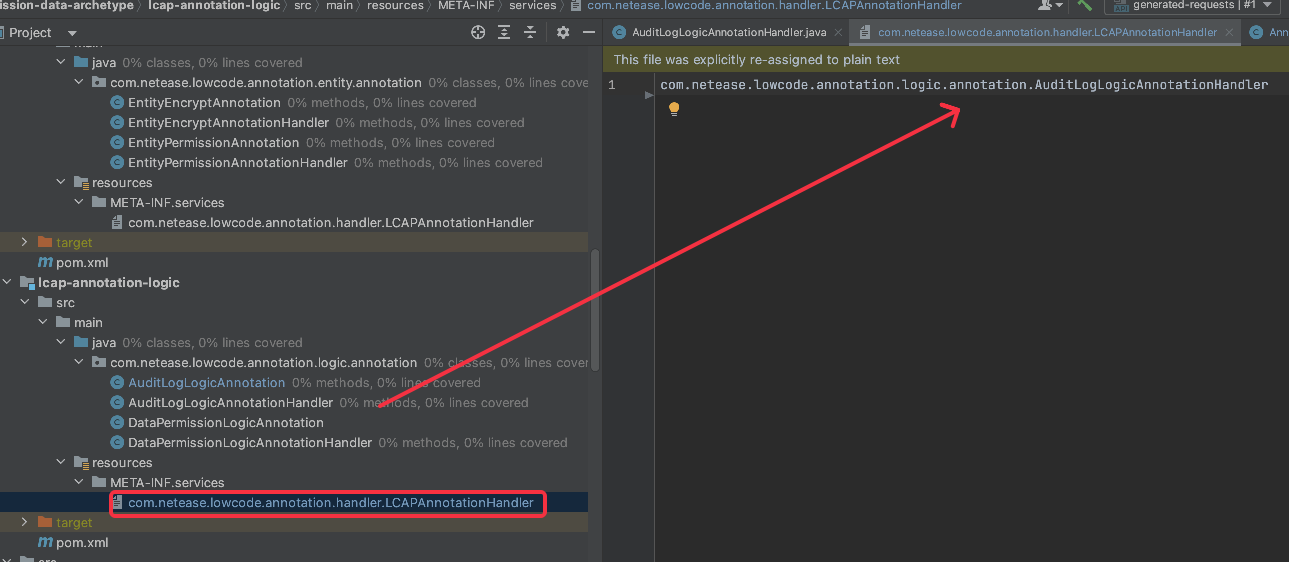
1. 新建META-INF/services文件夹

2.新建文件，文件名为：com.netease.lowcode.annotation.handler.LCAPAnnotationHandler，注意**文件名不可更改**



3. 打开文件，并填写业务处理器类名，比如：

com.netease.lowcode.annotation.logic.annotation.AuditLogLogicAnnotationHandler



注意：切面逻辑器通过SPI机制和制品斜街。无法直接使用spring上下文，可以使用下列方式获取：

private XXXClassName getBean() {

return (XXXClassName) OverriddenFrameworkHelper.getBean("xxxBeanName");

}

**2.4 上报、规则配置机制**

针对IP白名单这个案例，如果需要我们白名单是可配置的，可以声明一个同样的含义相同的NaslConfiguration，由低代码开发者在应用配置中填写。最终在handle内对当前HttpRequest的IP进行匹配，以实现过滤的功能，完成这个需求。

但是如果需要支持能够在运行时由用户去维护这个白名单而不是每次都得重新发布，那么需要通过实现Handler的report，来利用上报机制。

**复写逻辑**

上报机制需要用到复写逻辑，实现在依赖库中去调用IDE中的服务端逻辑。依赖库通过「自身声明NaslLogic，让低代码开发者去重写」的方式，来回调获取制品中的规则配置，是3.8版本首次出现的用法。

**定义复写逻辑**

依赖库里面，如果需要和IDE交互，则可通过声明一个可复写逻辑来实现，比如声明一个获取用户ID的逻辑。

其中 @NaslLogic(override = true)，代表此逻辑可被复写

/\*\*

\* 获取用户ID

\*

\* @return

\*/

@NaslLogic(override = true)

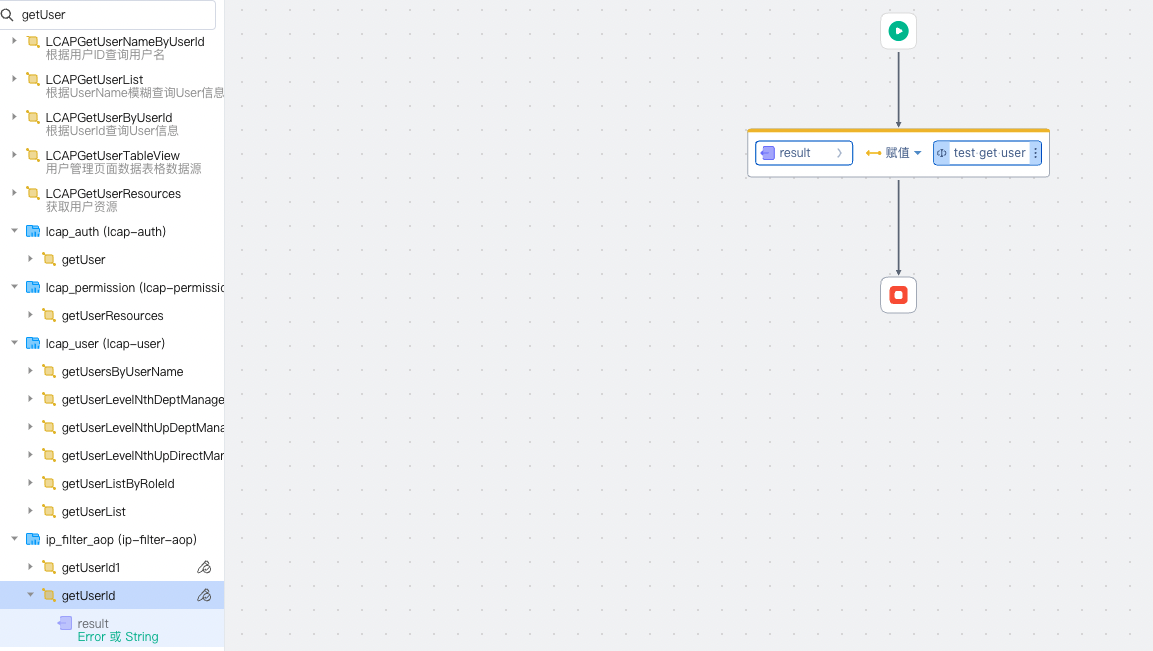
public static String getUserId() {

//TODO : IDE 可复写其逻辑

return null;

}

**IDE中重写逻辑**



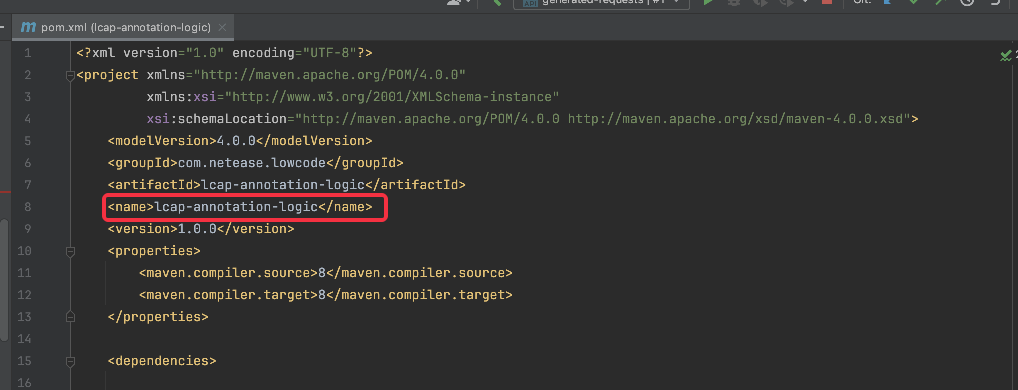
**使用复写逻辑**

依赖库如果想使用复写逻辑，则需要通过框架API来调用。

OverriddenFrameworkHelper

.invokeOverriddenMethod0(String methodName, String libraryName, Object... args);

1. methodName ：为方法名
2. libraryName：依赖库名称
   1. 查看方法：依赖库的 maven的pom.xml配置文件的name 标签



1. args：为方法入参，如果为空，则可不填

**上报机制说明**

2.1中声明的逻辑切面处理中IpLogicAnnotationHandler中有一个实现类report。在制品启动时，会将哪些逻辑开启了IpLogicAnnotation的逻辑权限，以及这些逻辑的其他信息，一起传入到report方法。

在依赖库的report方法中，可以选择：

1. 忽略数据，直接返回true；
2. 调用外部服务的open api，将数据存储到外部服务。依赖库的handle方法需要使用到这些数据时，通过调用外部open api获取。
3. 调用**复写逻辑**。在IDE的服务端逻辑中，实现复现逻辑，将数据持久化。如存储到实体中，供依赖库的handle方法切面过滤时使用。如3.8版本的数据权限功能，将逻辑数据权限信息保存在局部模板的实体数据库中。局部模板和依赖库的handle方法基于数据库的数据，编写上层业务。

/\*\*

\* 上报方法 // 下文详细说明

\*/

@Override

public Boolean report(List<LogicContext<IpLogicAnnotation>> logicContextList) {

return true; // 如果不需要使用"上报"机制，直接return即可

}

**抛出业务异常**

如果想要实现类似「当登录者余额不足时，无论是否在白名单都当作拦截，并且告知用户余额不足」，可以通过抛出LCAPAnnotationLogicHandlerException异常来实现。

@Override

public Object handle(

Object[] args,

Object result,

LogicContext<IpLogicAnnotation> context)

throws LCAPAnnotationLogicHandlerException {

if (!nsfCheck()) {

throw new LCAPAnnotationLogicHandlerException("余额不足"); // 这4个字会toast到页面

}

if (!nsfCheck()) {

// 不要抛出任何方法签名所声明以外的异常，这会被框架视为依赖库BUG

// 并toast 依赖库逻辑切面执行失败，请联系技术支持

throw new RuntimeException("余额不足");

}

// ...后续handle代码

}

**实体切面扩展**

**案例2：查询实体的字段时进行加密**

通过这个案例讲解如何使用「实体注解」。可参考<https://github.com/netease-lcap/CodeWaveAssetCompetition2024>开源仓库中的ip-filter-aop依赖库。

**1、声明注解**

首先需要声明一个注解，它的落地形式是一个声明了@NaslAnnotation的public class，如下所示——

import com.netease.lowcode.annotation.LCAPLogicAnnotation;

import com.netease.lowcode.core.annotation.NaslAnnotation;

/\*\*

\* IP白名单注解 // 用于声明注解的标题，详见下图

\*/

@NaslAnnotation(

applyTo = {NaslAnnotation.Component.ENTITY} // 用于声明这是一个逻辑注解

)

public class EncryptEntityAnnotation extends LCAPSQLAnnotation { // 必须继承于LCAPLogicAnnotation

@NaslAnnotation.Property(

title = "是否开启数据加密", // IDE将使用这个值作为默认开关的标题

defaultValue = "false" // IDE将使用这个值作为开关的默认开启|关闭情况，useAnno的deaultValue必须为true或false

)

public Boolean useAnno; // 这个field代表的是注解的默认开关。因设计原因，public、Boolean、useAnno这三部分必须是固定的

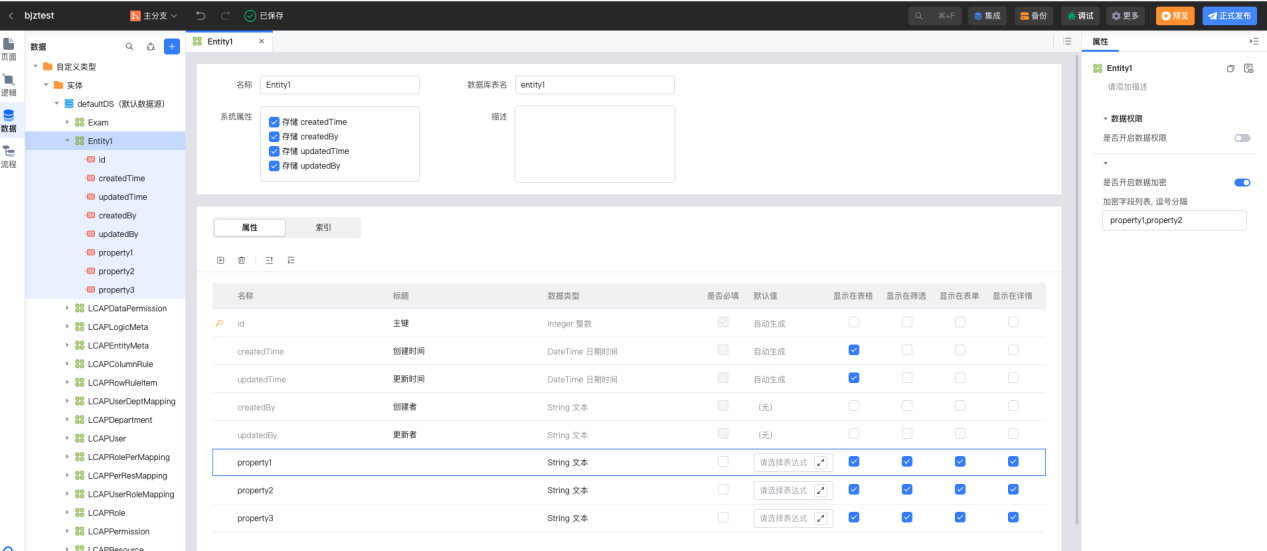
@NaslAnnotation.Property(

title = "加密字段列表, 逗号分隔")

public String encryptFieldList;

}

在数据菜单中，选中实体后，在右侧打开开关，并且填写注解信息。



**2、实现切面处理器**

至此，注解声明方面的内容已说明完毕，有了注解还需要结合切面处理器才能发挥作用，下面开始介绍如何为开发切面处理器。

**2.1 声明SQL切面处理器**

public class EncryptSQLAnnotationHandler implements LCAPSQLAnnotationHandler< // 不同于逻辑注解处理器，这里必须实现LCAPSQLAnnotationHandler

EncryptEntityAnnotation> {

/\*\*

\* 上报方法 // 详见案例1

\*/

@Override

public Boolean report(List<EntityContext<EncryptEntityAnnotation>> list) throws LCAPAnnotationSQLHandlerException {

return true;// 如果不需要使用"上报"机制，直接return即可

}

@Override

public Map<String, Object> listParamValues(String s, EntityContext<EncryptEntityAnnotation> entityContext) {

return null;

}

@Override

public String buildWhereCondition(BuildWhereConditionArgs buildWhereConditionArgs, EntityContext<EncryptEntityAnnotation> entityContext) throws LCAPAnnotationSQLHandlerException {

return null;

}

@Overridepublic String buildSelectExpr(BuildSelectExprArgs buildSelectExprArgs, EntityContext<EncryptEntityAnnotation> entityContext) throws LCAPAnnotationSQLHandlerException {

return null;

}

@Overridepublic boolean isFilterable(IsFilterableArgs isFilterableArgs, EntityContext<EncryptEntityAnnotation> entityContext) throws LCAPAnnotationSQLHandlerException {

return false;

}

@Overridepublic String buildAssignment(BuildAssignmentArgs buildAssignmentArgs, EntityContext<EncryptEntityAnnotation> entityContext) throws LCAPAnnotationSQLHandlerException {

return null;

}

/\*\*

\* 绑定的注解

\*/

@Override

public Class<? extends LCAPAnnotation> consume() {

return EncryptEntityAnnotation.class; // 由于这是IpLogicAnnotation的处理器，所以此处也必须返回它

}

/\*\*

\* 执行顺序，order越小，这个对象的handle在同一advise时机下，相比别的handler越先执行。

\* order相同时不保证执行顺序

\* 缺省为0

\*/

@Override

public Integer order() {

return LCAPSQLAnnotationHandler.super.order();

}

}

实体注解处理器与逻辑注解处理器的不同之处在于

1. 逻辑注解处理器在制品的切入点是Spring切面；实体注解处理器在制品的切入点是Mybatis ONGL表达式静态回调。
2. 一个逻辑注解处理器的执行时机由advise()方法决定，执行内容由handle方法决定；一个实体注解处理器的执行时机固定为listParamValues、buildWhereCondition、buildSelectExpr、isFilterable、buildAssignment这5个，开发者可以根据需要重写必要的方法。随着AOP机制的迭代，支持的时机会增加到更多。

**2.2 开发切面处理器**

handler的核心方法是以下5个——

1. buildWhereCondition，当一个实体开启了某实体注解，这个实体在制品的任意Mapping.xml的任意SQL（无论是通过实体内置逻辑、数据查询组件、SQL组件翻译而来的）中的From Clause出现时，会为其在Where Caluse添加“埋点”，制品运行期执行到该SQL时，AOP框架会回调开启的实体注解绑定的handler对象的buildWhereCondition方法，开发者可以通过重写buildWhereCondition方法，并返回where\_cond SQL片段，对实体的查询条件进行过滤条件增强。这个方法最典型的实践是数据权限的行权限功能。



import java.util.StringJoiner;

public class BuildWhereConditionArgs {

/\*\*

\* 代表为哪个实体构建where\_condition

\*/

private String entityName;

/\*\*

\* 代表实体对应的table在语句中的表名或者表别名

\*/

private String tableAlias;

/\*\*

\* 代表返回值将被AOP框架添加到 whereClause | havingClause | onClause 子句中

\*/

private String clause;

public String getEntityName() {

return entityName;

}

public void setEntityName(String entityName) {

this.entityName = entityName;

}

public String getTableAlias() {

return tableAlias;

}

public void setTableAlias(String tableAlias) {

this.tableAlias = tableAlias;

}

public String getClause() {

return clause;

}

public void setClause(String clause) {

this.clause = clause;

}

@Override

public String toString() {

return new StringJoiner(", ", BuildWhereConditionArgs.class.getSimpleName() + "[", "]").add(

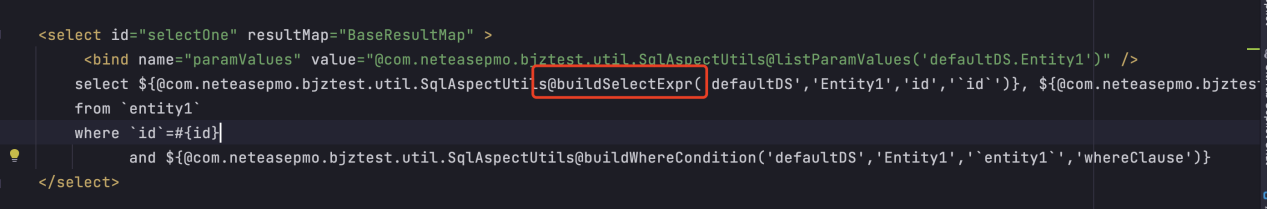
"entityName='" + entityName + "'").add("tableAlias='" + tableAlias + "'").add("clause='" + clause + "'")

.toString();

}

}

1. buildSelectExpr，原理同上，翻译器会将开启注解的实体的每个属性所出现的select\_expr替换为“埋点”，制品运行期执行到该SQL时，AOP框架会回调buildSelectExpr方法，开发者可以通过重写这个方法，并返回select\_expr SQL片段，对该实体的该字段的查询结果进行增强。这个方法最典型的实践是数据权限的列读权限功能、数据加密等。



import java.util.StringJoiner;

public class BuildSelectExprArgs {

/\*\*

\* 代表为哪个实体构建select\_expr

\*/

private String entityName;

/\*\*

\* 代表为实体的哪个属性构建select\_expr

\*/

private String propertyName;

/\*\*

\* 代表前一个LCAPSQLAspectHandler所构建的select\_expr片段；若本Handler是第一个，prevAssignment代表原SQL的原select\_expr

\* 如"prevSelectExpr":"`entity1`.`id` `T1\_C1`"，其中T1\_C1为别名

\*/

private String prevSelectExpr;

public String getEntityName() {

return entityName;

}

public void setEntityName(String entityName) {

this.entityName = entityName;

}

public String getPropertyName() {

return propertyName;

}

public void setPropertyName(String propertyName) {

this.propertyName = propertyName;

}

public String getPrevSelectExpr() {

return prevSelectExpr;

}

public void setPrevSelectExpr(String prevSelectExpr) {

this.prevSelectExpr = prevSelectExpr;

}

@Override

public String toString() {

return new StringJoiner(", ", BuildSelectExprArgs.class.getSimpleName() + "[", "]").add(

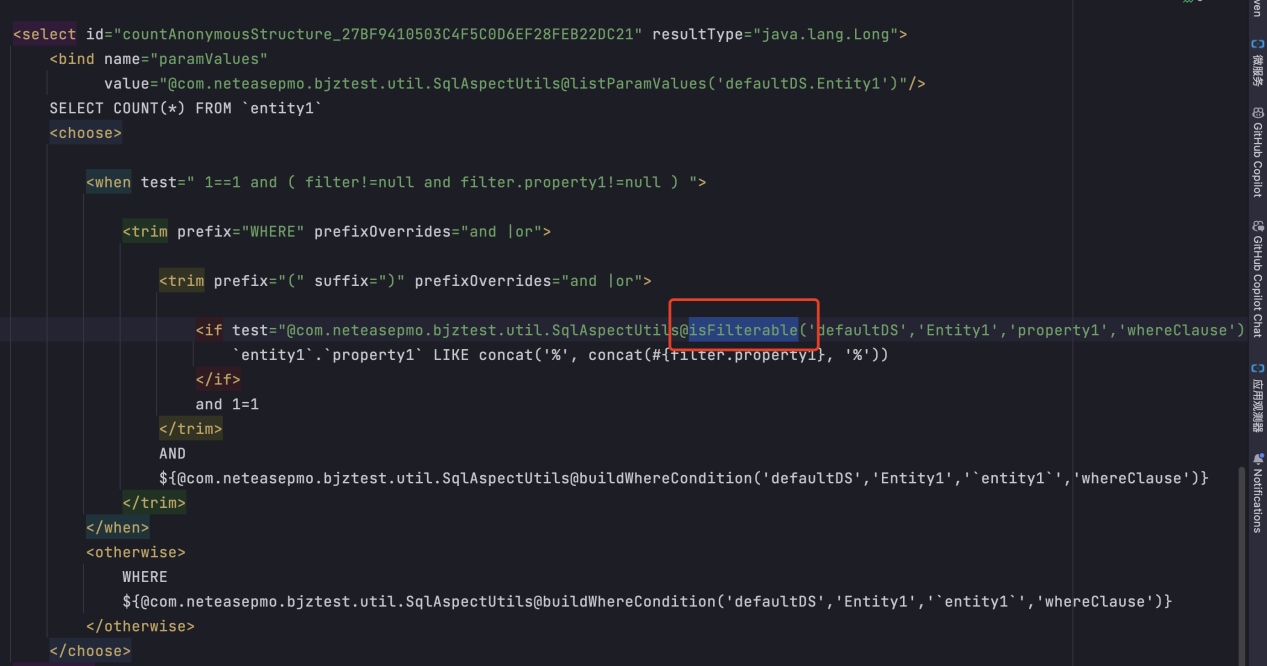
"entityName='" + entityName + "'").add("propertyName='" + propertyName + "'")

.add("prevSelectExpr='" + prevSelectExpr + "'").toString();

}

}

1. isFilterable，原理同上，开启了注解的属性，在作为where条件的时候，会去调用本方法。翻译器会将开启注解的实体的每个属性所出现在Where Clause中的binary\_expr替换为“埋点”，制品运行期执行到该SQL时，AOP框架会回调isFilterable方法，开发者可以通过重写这个方法，并返回true | false，对实体的该字段能否成为过滤条件进行控制。这个方法最典型的实践是数据权限的列读权限防撞库功能。



import java.util.StringJoiner;

public class IsFilterableArgs {

private String entityName;

private String propertyName;

private String clause;

/\*\*

\* 代表前一个LCAPSQLAspectHandler所判断的isFilterable结果，若本Handler是第一个，prevIsFilterable会是true

\*/

private Boolean prevIsFilterable;

public String getEntityName() {

return entityName;

}

public void setEntityName(String entityName) {

this.entityName = entityName;

}

public String getPropertyName() {

return propertyName;

}

public void setPropertyName(String propertyName) {

this.propertyName = propertyName;

}

public String getClause() {

return clause;

}

public void setClause(String clause) {

this.clause = clause;

}

public Boolean getPrevIsFilterable() {

return prevIsFilterable;

}

public void setPrevIsFilterable(Boolean prevIsFilterable) {

this.prevIsFilterable = prevIsFilterable;

}

@Override

public String toString() {

return new StringJoiner(", ", IsFilterableArgs.class.getSimpleName() + "[", "]").add(

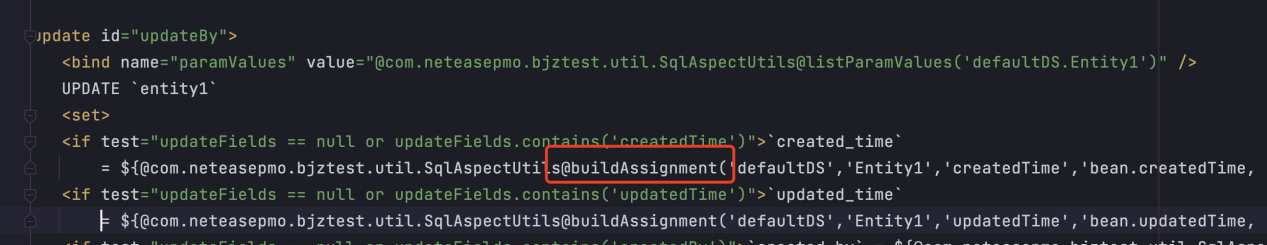
"entityName='" + entityName + "'").add("propertyName='" + propertyName + "'")

.add("clause='" + clause + "'").add("prevIsFilterable=" + prevIsFilterable).toString();

}

}

1. buildAssignment，原理同上翻译器会将开启注解的实体的每个属性所出现在Set Clause中的assignment替换为“埋点”，制品运行期执行到该SQL时，AOP框架会回调buildAssignment方法，开发者可以通过重写这个方法，并返回assignment SQL片段，对该实体的该字段的更新行为进行增强。这个方法最典型的实践是数据权限的列写权限功能



import java.util.StringJoiner;

public class BuildAssignmentArgs {

/\*\*

\* 代表为哪个实体构建assignment

\*/

private String entityName;

/\*\*

\* 代表为实体的哪个属性构建assignment

\*/

private String propertyName;

/\*\*

\* 代表前一个LCAPSQLAspectHandler所构建的assignment片段；若本Handler是第一个，prevAssignment代表原SQL的原assignment

\*/

private String prevAssignment;

public String getEntityName() {

return entityName;

}

public void setEntityName(String entityName) {

this.entityName = entityName;

}

public String getPropertyName() {

return propertyName;

}

public void setPropertyName(String propertyName) {

this.propertyName = propertyName;

}

public String getPrevAssignment() {

return prevAssignment;

}

public void setPrevAssignment(String prevAssignment) {

this.prevAssignment = prevAssignment;

}

@Override

public String toString() {

return new StringJoiner(", ", BuildAssignmentArgs.class.getSimpleName() + "[", "]").add(

"entityName='" + entityName + "'").add("propertyName='" + propertyName + "'")

.add("prevAssignment='" + prevAssignment + "'").toString();

}

}

1. listParamValues，上述能够产生SQL片段的buildXxx方法，如果组成返回值的片段中，存在着用户输入的文本，那么都面临一个问题，SQL注入风险，结合重写该方法，可以规避SQL注入风险。可以在buildXxx方法中，用#{}等占位符表示入参，制品会在执行sql时，调用listParamValues方法，查询当前需要作为param的所有字段信息。3.8版本数据权限功能就是通过buildWhereCondition和listParamValues联动来规避SQL注入风险的。

回到案例，可以通过重写buildSelectExpr方法，实现对管理员用户在运行时配置的字段进行查询结果加密。

@Overridepublic String buildSelectExpr(BuildSelectExprArgs buildSelectExprArgs, EntityContext<EncryptEntityAnnotation> entityContext) {

// 从entityContext中获取注解的值。entityContext是IDE中打开EncryptEntityAnnotation注解的实体的上下文对象，包括注解的值、实体的信息等

String encryptFieldList = entityContext.getAnnotation().encryptFieldList;

String[] encryptFields = encryptFieldList.split(",");

for (String encryptField : encryptFields) {

// buildSelectExprArgs example：

// 可以看到prevSelectExpr携带别名

// {"entityName":"defaultDS.Entity1","propertyName":"id","prevSelectExpr":"`entity1`.`id` `T1\_C1`"}

if (encryptField.equals(buildSelectExprArgs.getPropertyName())) {

//用空格分割 buildSelectExprArgs.getPrevSelectExpr()。兼容有别名的情况

if (!StringUtils.isEmpty(buildSelectExprArgs.getPrevSelectExpr())) {

String[] prevSelectExpr = buildSelectExprArgs.getPrevSelectExpr().split(" ");

String md5Expr = null;

if (prevSelectExpr.length == 1) {

md5Expr = "md5(" + buildSelectExprArgs.getPrevSelectExpr() + ")";

}

else if (prevSelectExpr.length == 2) {

md5Expr = " md5(" + prevSelectExpr[0] + ") " + prevSelectExpr[1];

}

return md5Expr;

}

}

}

return buildSelectExprArgs.getPrevSelectExpr();

}

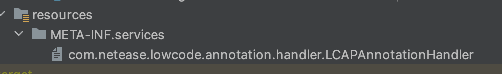
最终效果



**2.3 SPI声明**

1. 新建META-INF/services文件夹

2.新建文件，文件名为：com.netease.lowcode.annotation.handler.LCAPAnnotationHandler，注意**文件名不可更改**



3. 打开文件，并填写业务处理器类名，比如：

com.netease.lowcode.lib.handler.EncryptSQLAnnotationHandler

**2.4 其他上报、复写、异常等问题，详见案例1**