智能数据湖 AI DataLake

SDK 参考

文档版本 01

发布日期 2026-11-09



华为云计算技术有限公司

版权所有 © 华为云计算技术有限公司 2026。 保留一切权利。

非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为云计算技术有限公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为云计算技术有限公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

华为云计算技术有限公司

地址： 贵州省贵安新区黔中大道交兴功路华为云数据中心 邮编：550029

网址： https://www.huaweicloud.com/

目 录

1 AI DataLake SDK 简介 1
2 引擎专属 SDK 4
2.1 Python SDK 参考 4
2.1.1 Python SDK 概述 4
2.1.2 Python 环境配置 4
2.1.3 获取 AKSK 5
2.1.4 Catalog SDK 5
2.1.4.1 Catalog SDK 概述 5
2.1.4.2 Catalog SDK 获取与安装 6
2.1.4.3 初始化客户端 7
2.1.4.4 使用 AuraJobV2 运行示例 8
2.1.4.5 创建 UDF 10
2.1.4.6 创建 UDTF 12
2.1.4.7 创建 UDAF 14
2.1.4.8 查询函数详情 16
2.1.4.9 查询函数列表 17
2.1.4.10 删除函数 18
2.1.4.11 注册 Data-Juicer 算子 19
2.1.4.12 公共参数说明 21
2.1.4.12.1 signature 说明 21
2.1.4.12.2 runtime_env 说明 24
2.1.4.13 异常类说明 25
2.1.5 Aura DataFrame SDK（Python） 26
2.1.5.1 Aura DataFrame SDK 概述 26
2.1.5.2 Aura DataFrame SDK 获取与安装 28
2.1.5.3 Aura DataFrame SDK 新手入门 30
2.1.5.4 使用 Aura DataFrame SDK 提交作业 32
2.1.5.4.1 建立与 AuraJobV2 端点的连接 32
2.1.5.4.2 配置 UDF 部署工作区 34
2.1.5.4.3 创建表并写入数据 35
2.1.5.4.4 建立连接并查询指定表数据 37
2.1.5.4.5 数据加工与分析 38
2.1.5.4.6 使用 Scalar UDF 处理数据 40

 版权所有 © 华为云计算技术有限公司

2.1.5.4.7 使用 Class UDF 处理数据 41
2.1.5.4.8 使用聚合统计的 UDAF 44
2.1.5.4.9 直接使用已注册的 Scalar UDF 46
2.1.5.4.10 直接使用已注册的 UDAF 47
2.1.5.4.11 语音识别示例 49
2.1.5.4.12 视频帧描述与摘要生成示例 52
2.1.5.5 Aura DataFrame SDK 总览 59
2.1.5.6 Session APIs 62
2.1.5.7 Dataset APIs 63
2.1.5.8 Table APIs 82
2.1.5.9 I/O APIs 88
2.1.5.10 User-Defined Function APIs 92
2.1.5.11 AI APIs 108
2.1.5.12 用户自定义函数 UDF 110
2.1.5.12.1 UDF 概述 110
2.1.5.12.2 开发注意事项 113
2.1.5.12.3 数据类型映射 114
2.1.5.12.4 环境变量设置 115
2.1.5.12.5 Scalar UDF 118
2.1.5.12.6 Class UDF 120
2.1.5.12.7 Vectorized UDF 122
2.1.5.12.8 UDTF 125
2.1.5.12.9 UDAF 126
2.1.5.12.10 WITH ARGUMENTS 语法 129
2.1.5.12.11 自适应并行 131
2.1.5.12.12 UDF 错误处理与重试 131
2.1.5.12.13 UDF Actor 放置策略 138
2.1.5.12.14 Python UDF 性能调优 141
2.1.5.13 数据类型 143
2.1.5.13.1 图片 143
2.1.5.13.2 视频 144
2.1.5.13.3 音频 145
2.1.5.13.4 向量 146
2.1.6 Spark SDK（Python） 147
2.1.6.1 Spark SDK 获取与初始化 147
2.1.6.2 Spark SQL 作业提交 149
2.1.6.3 Spark Job 作业提交 151
2.1.6.4 Spark 连接数据库并查询 154
2.2 Java SDK 参考 156
2.2.1 Java 环境配置 156
2.2.2 Spark SDK（Java） 157
2.2.2.1 Spark SDK 的获取与初始化 157

 版权所有 © 华为云计算技术有限公司

2.2.2.2 Spark SQL 作业提交 159
2.2.2.3 Spark Job 作业提交 161
2.2.2.4 Spark 库表元数据查询 165

 版权所有 © 华为云计算技术有限公司

1 AI DataLake SDK 简介

AI DataLake SDK 简介

AI DataLake为开发者提供多种类型的SDK，用于简化与AI DataLake服务的集成开发。开发者无需关注底层REST API的调用细节，即可通过SDK完成作业提交、数据查询、自定义函数开发等操作。

AI DataLake的SDK按语言分为**Python SDK**与**Java SDK**两大类，覆盖 Aura、Ray、Spark 等计算引擎的各类作业场景。

本文档按语言分类SDK，您可以根据业务实际的技术栈直接定位对应SDK。

如果您需要了解SDK的隐私数据处理策略，请参见**华为云SDK隐私声明**。

SDK 语言分布

语言	SDK	适用引擎	适用场景	帮助文档
Python	UDF SDK	Aura（AuraJobV2）	自定义函数（UDF/UDAF/UDTF）注册与管理，多引擎复用	**2.1.4 Catalog SDK**
Python	Aura DataFrame SDK	Aura（AuraJobV2）	Python/Notebook交互式数据处理、多模态数据开发	**2.1.5 Aura DataFrame SDK（Python）**
Python	Spark SDK（Python）	Spark（Spark SQL、Spark Job）	Spark SQL查询、Spark批处理作业提交	**2.1.6 Spark SDK（Python）**
Java	Spark SDK（Java）	Spark（Spark SQL、Spark Job）	Spark SQL查询、Spark批处理作业提交、库表元数据查询	**2.2.2 Spark SDK（Java）**

版权所有 © 华为云计算技术有限公司

SDK 体系结构

AI DataLake 的 SDK 体系由**引擎专属SDK**和**跨引擎共享SDK**三层构成：

- **引擎专属SDK：**各引擎自研SDK（Aura DataFrame SDK、Ray SDK、Spark SDK）。
- **跨引擎共享SDK：**用于将UDF注册至LakeFormation，多引擎复用（UDF SDK）。

AI DataLake 适用 UDF 场景的 SDK

AI DataLake 提供的UDF SDK用于自定义函数的跨引擎复用。开发者通过UDF SDK编写一次函数，将函数注册至LakeFormation元数据服务，即可被多个引擎识别和调用，避免在各引擎中重复开发函数。

类型	说明
适用引擎	当前支持 Aura
适用场景	自定义函数（UDF/UDAF/UDTF）的一次开发、多引擎复用
实现机制	函数代码与元数据统一存储在 LakeFormation，各引擎通过元数据服务发现并调用
核心能力	函数注册（create_udf/create_udtf/create_udaf）、查询、删除、Data-Juicer算子适配

各引擎专属 SDK

Aura、Ray、Spark引擎的专属SDK，基于通用SDK或REST API进行二次封装，更贴近各引擎的使用方式，简化开发流程。

SDK	语言	适用端点类型	适用场景	核心特性	SDK参考
Aura DataFrame SDK	Python	AuraJobV2	Python/Notebook交互式数据处理、多模态数据开发	类Pandas的DataFrame API，结果直接返回Pandas DataFrame。 支持多模态数据类型（图片、视频、音频、向量）。 支持Python UDF全套能力（Scalar UDF、Class UDF、Vectorized UDF、UDTF、UDAF）。 支持算子级资源规格配置、亲和调度策略和性能观测。	2.1.5.1 Aura DataFrame SDK概述

版权所有 © 华为云计算技术有限公司

SDK	语言	适用端点类型	适用场景	核心特性	SDK参考
Spark SDK	Java、Python	Spark SQL、Spark Job	提交Spark SQL查询与Spark批处理作业	Java版提供Spark SQL、Spark Job、元数据查询三类客户端。 Python版提供统一客户端与Builder链式作业构建，支持Jar/SqlScripting等作业类型。	2.1.6 Spark SDK（Python）

更多连接方式

除SDK外，AI DataLake 还提供以下连接方式。SDK与连接方式的区别在于：SDK提供编程接口封装，适合应用集成开发；以下方式用于特定工具或协议场景：

连接方式	适用引擎	适用场景
Spark Connect	Spark	Notebook交互式开发、PySpark脚本对接
REST API	全部引擎	自动化作业提交、应用集成
DataArts Studio	Aura	可视化作业编排、生产环境调度

版权所有 © 华为云计算技术有限公司

2 引擎专属 SDK

2.1 Python SDK参考
2.2 Java SDK参考

2.1 Python SDK 参考

2.1.1 Python SDK 概述

AI DataLake为Python开发者提供多种类型的SDK，用于简化与AI DataLake服务的集成开发。开发者无需关注底层REST API的调用细节，即可通过Python SDK完成作业提交、数据查询、自定义函数开发等操作。

表 2-1 AI DataLake 的 Python SDK

SDK类型	适用引擎	适用场景
UDF SDK	Aura（AuraJobV2）	自定义函数（UDF/UDAF/UDTF）的一次开发、多引擎复用
Aura DataFrame SDK	Aura（AuraJobV2）	Python/Notebook交互式数据处理、多模态数据开发
Spark SDK-Python	Spark（Spark SQL、Spark Job）	提交Spark SQL查询与Spark批处理作业

2.1.2 Python 环境配置

操作场景

在安装和使用Python SDK前，确保您已经完成开发环境的基本配置。

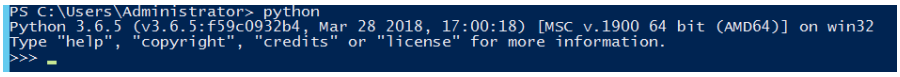
Python版本建议使用2.7.10和3.4.0以上版本，需要配置Visual C++编译环境Visual C++ build tools 或者 Visual Studio。

 版权所有 © 华为云计算技术有限公司

操作步骤

1. 从**Python官网**下载并安装Python版本。
2. 检验是否配置成功，运行cmd ，输入python。运行结果，请参见**图2-1**，显示版本信息，则说明安装和配置成功。

图 2-1 检验配置是否成功



说明

Python安装应用包时出现类似错误“error: Microsoft Visual C++ xx.x is required. Get it with Build Tools for Visual Studio ”，可能是由于缺少C++编译器导致的报错，建议您根据提示信息安装相应版本的Visual Studio编译器解决。部分操作系统Visual Studio安装后需重启才可以生效。

2.1.3 获取 AKSK

AI DataLake 的 Python SDK 均使用 AK/SK（Access Key ID/Secret Access Key）认证方式访问服务。使用 SDK 前，需先获取访问密钥。

约束与限制

- 认证用的AK/SK 硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中存放。
- 使用临时AK/SK访问时，需同时配置security_token（安全令牌），且临时凭证存在有效期，建议仅在测试场景使用。

操作步骤

1. 登录**AI DataLake管理控制台**。
2. 鼠标移动至页面右上方的用户名，在下拉列表中选择“我的凭证”。
3. 单击“访问密钥”页签
4. 单击“新增访问密钥”。
5. 输入访问密钥描述信息（非必填），单击“确定”。
6. 输入验证码或密码，完成身份验证。
7. 单击“确定”，生成并下载访问密钥文件（.csv）。
8. 打开 .csv 文件，获取用户的 AK（Access Key ID）与 SK（Secret Access Key）信息。

2.1.4 Catalog SDK

2.1.4.1 Catalog SDK 概述

本文为您介绍 AI DataLake服务Catalog模块提供的Python SDK，并列举SDK的获取方式以及SDK的使用说明。

版权所有 © 华为云计算技术有限公司

Catalog SDK 简介

SDK全称为 Software Development Kit，即软件开发工具包。它是一种为开发者提供的软件开发工具集合，包含了一系列的API、工具、示例代码、文档等资源，可以帮助开发者快速、高效地开发出各种应用程序。

AI DataLake Catalog Python SDK对LakeFormation与OBS（对象存储服务）的REST API进行封装，以简化UDF（用户自定义函数）的注册与管理开发工作，用于满足客户有UDF管理和使用分离的场景，该SDK注册的Python UDF可用于AI DataLake内部的Aura引擎使用。SDK以访问密钥（AccessKey）识别调用者身份，提供自动签名、类型推断、UDF打包上传、元数据注册等能力，方便您通过统一客户端提交和管理标量UDF（udf）、表函数（udtf）、聚合函数（udaf）以及Data-Juicer算子。

SDK核心入口为AIDatalake_Catalog_Client，封装以下能力：

- LakeFormation REST API调用（函数CRUD）。
- OBS上传/删除UDF包。
- 基于可调用对象注解的类型推断。
- 将UDF打包为zip文件并上传。

功能特性

表 2-2 SDK 能力列表

能力	对应方法	说明
注册标量UDF	create_udf	将标量函数注册到LakeFormation，逐行处理输入列。
注册表函数UDTF	create_udtf	将表函数注册到LakeFormation，可一次输入返回多行多列。
注册聚合函数UDAF	create_udaf	将聚合函数注册到LakeFormation，对多行数据进行聚合。
注册Data-Juicer算子	create_datajuicer_operator	将Data-Juicer的Filter/Mapper算子自动适配为标量UDF并注册。
查询函数	get_function	按名称从LakeFormation获取函数元数据。
查询函数列表	list_function	列出指定数据库下的全部函数。
删除函数	delete_function	删除LakeFormation函数记录并清理其OBS包。

2.1.4.2 Catalog SDK 获取与安装

SDK以Python包的形式发布在PyPI仓库，您可以通过pip命令一键安装。安装过程中会自动下载SDK依赖的第三方库。本节操作介绍如何获取并安装SDK的操作指导。

版权所有 © 华为云计算技术有限公司

前提条件

Python版本建议使用3.10或3.11 。具体安装，请参见2.1.2 Python环境配置。

操作步骤

步骤1 已联系技术支持获取SDK安装包。

步骤2 安装SDK。

1. 下载并解压SDK安装包。
2. 打开命令行窗口，进入解压目录。
3. 执行如下命令安装SDK，安装过程中会自动下载第三方依赖库。

```
pip install huawei_aidatalake_catalog-0.1.3-py3-none-any.whl
```

说明

SDK依赖包括esdk-obs-python（OBS）、LakeFormation REST客户端、cloudpickle、pyarrow、pandas等，安装过程中会自动获取。

----结束

2.1.4.3 初始化客户端

客户端AIDatalake_Catalog_Client是SDK的核心入口，初始化时需传入认证凭证（AK/SK）、LakeFormation与OBS的端点等信息。初始化完成后，您即可通过该客户端创建、查询、删除函数。

前提条件

- 已创建LakeFormation实例，并确认实例下存在数据目录（Catalog）与数据库。
- 已创建OBS并行文件系统。

AK/SK 认证方式样例代码

- 代码样例

```
from aidatalake_catalog.client import AIDatalake_Catalog_Client

client = AIDatalake_Catalog_Client(
    ak="your-ak",
    sk="your-sk",
    security_token=None,                    # 临时凭证时使用
    obs_endpoint="obs.cn-north-4.myhuaweicloud.com",
    lakeformation_endpoint="https://lakeformation.cn-north-4.myhuaweicloud.com",
    lakeformation_instanceid="your-instance-id",
    catalog_name="your-catalog-name",
    default_database="your_default_db",
    obs_path="your-bucket/udf/",            # bucket_name/dir/subdir
    project_id="your-project-id",
    ignore_ssl_verification=False,
)
```

- 参数说明

参数	是否必选	类型	说明
ak	是	str	访问密钥AK/SK。获取AK/SK的具体操作，请参见2.1.3 获取AKSK。

版权所有 © 华为云计算技术有限公司

参数	是否必选	类型	说明
sk	是	str	
security_token	否	str | None	临时凭证的安全令牌，使用AK/SK长期凭证时传None。
obs_endpoint	是	str	OBS终端节点，例如：obs.cn-north-4.myhuaweicloud.com。
lakeformation_endpoint	是	str	LakeFormation API终端节点URL。
lakeformation_instanceid	是	str	LakeFormation实例ID（用于IAM鉴权）。
catalog_name	是	str	LakeFormation目录名（三层结构：instance/catalog/database）。
default_database	是	str	函数操作的默认数据库名。
obs_path	是	str	OBS路径，格式为bucket_name/dir/subdir。
project_id	是	str	项目ID（用于LakeFormation API鉴权）。
ignore_ssl_verification	否	bool	是否禁用SSL证书校验，默认False。自签名证书场景可设为True。

说明

- 认证用的AK和SK硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。
- 本示例以AK和SK保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量xxx_SDK_AK和xxx_SDK_SK。

2.1.4.4 使用 AuraJobV2 运行示例

该示例实现了通过该SDK注册UDF到LakeFormation后，再使用AuraJobV2引擎读取指定数据库表中的数据列后使用UDF做计算的过程。

前提条件

- 已创建OBS并行文件系统，用于存放SDK上传的UDF包。具体操作，请参见**创建并行文件系统**。
- 已创建LakeFormation实例，并确认实例下存在数据目录（Catalog）与数据库。具体操作，请参见**创建LakeFormation实例并规划元数据**。
- 已在LakeFormation中注册数据表（本示例至少包含id、value两列）。
- 已创建AuraJobV2端点，并获取端点名称、工作空间ID等信息。具体操作，请参见**创建Aura引擎端点**。

版权所有 © 华为云计算技术有限公司

- 已获取AK和SK信息，具备OBS并行文件系统和LakeFormation访问权限。
- 已获取并安装Aura DataFrame SDK。具体操作，请参见**2.1.5.2 Aura DataFrame SDK获取与安装**。

示例

```
# 1. 导入依赖
import os
from aidatalake_catalog import Parameter, Signature
from aidatalake_catalog.client import AIDatalake_Catalog_Client
from aura_frame.multimodal import ai_lake

# 2. 定义带类型注解的UDF
class Multiplier:
    """Class 定义 UDF：乘以可配置 factor（默认 2）。"""
    def __init__(self, factor: int = 2):
        self._factor = factor
    def __call__(self, x: int) -> int:
        return x * self._factor

# 3. 定义函数签名（可选，未传入时由SDK根据类型注解自动推断）
sig = Signature(
    parameters=[Parameter(name="x", annotation=int)],
    return_annotation=int,
)
# 4. 初始化Catalog客户端（AK/SK建议从环境变量读取，避免硬编码）
client = AIDatalake_Catalog_Client(
    ak=os.getenv("XXX_SDK_AK"),
    sk=os.getenv("XXX_SDK_SK"),
    security_token=None,                    # 临时凭证时使用
    obs_endpoint="obs.cn-north-4.myhuaweicloud.com",
    lakeformation_endpoint="https://lakeformation.cn-north-4.myhuaweicloud.com",
    lakeformation_instanceid="your-instance-id",
    catalog_name="your-catalog-name",
    default_database="your_default_db",
    obs_path="your-bucket/udf/",            # bucket_name/dir/subdir
    project_id="your-project-id",
    ignore_ssl_verification=False,
)

# 5. 注册自定义函数
func = client.create_udf(
    Multiplier,
    name="class_multiplier_def",
    signature=sig,
    description="class 定义 UDF: 乘以 factor (默认 2)",
)

# 6.使用Aura引擎执行函数
"""
|  id  | value|
|------|------|
|  1   |  10  |
|  2   |  20  |
|  3   |  30  |
"""
con = ai_lake.connect(
        aura_endpoint=os.getenv("aura_endpoint"),
        aura_endpoint_name=os.getenv("aura_endpoint_name"),
        aura_workspace_id=os.getenv("aura_workspace_id"),
        lf_catalog_name=os.getenv("lf_catalog_name"),
        access_key=os.getenv("access_key"),
        secret_key=os.getenv("secret_key"),
        default_database=target_database,
        use_single_cn_mode=True
)

ds = con.load_dataset("table_1")  # table_1需要提前在LakeFormation注册
```


版权所有 © 华为云计算技术有限公司

```
ds = ds.map(fn=f'{database}.class_multiplier_def', on=['value'], as_col='mul_result')
df = ds.execute().sort_values('id').reset_index(drop=True)

print(f"  {'ID':<5} {'原始值':<10} {'mul_result':<15}")
print("-" * 35)
for _, r in df.iterrows():
    print(f"{int(r['id']):<5} {int(r['value']):<10} {int(r['mul_result']):<15}")
```

执行结果：

```
  ID    原始值        mul_result
-----------------------------------
  1     10          20
  2     20          40
  3     30          60
```

2.1.4.5 创建 UDF

将一个UDF（逐行处理输入列、返回单值）注册到LakeFormation。注册流程：类型推断->打包->上传OBS->写入LakeFormation元数据。若LakeFormation注册失败，会自动回滚已上传的OBS包。

前提条件

- 已完成SDK安装。具体操作，请参见**2.1.4.2 Catalog SDK获取与安装**。
- 已准备好UDF可调用对象（函数、函数类或类实例），并标注了类型注解。

函数签名

```
client.create_udf(
    udf, /, *,
    database=None,
    name,
    signature=None,
    imports=None,
    runtime_env=None,
    description=None,
    pretty_name=None,
    is_strict=True,
    volatility="VOLATILE",
    is_batch=False,
    batch_size=None,
    cpu_arch=None,
    cpus=None,
    memory=None,
    device_type=None,
    devices=None,
) -> Function
```

参数说明

参数	是否必选	类型	默认值	说明
udf	是	Any（位置参数）	-	UDF可调用对象（函数或可调用类）。
database	否	str | None	default_database	目标数据库名；为None时使用客户端默认数据库。

版权所有 © 华为云计算技术有限公司

参数	是否必选	类型	默认值	说明
name	是	str	-	函数名，须以字母开头，仅含字母、数字、下划线，长度为3~128。
signature	否	Signature | None	None	显式函数签名；为None时由UDF注解自动推断。 关于signature的更多信息，请参见**2.1.4.12.1 signature说明**。
imports	否	str | Iterable[str] | None	None	需要打入UDF包的文件/目录路径或URL列表。
runtime_env	否	str | Mapping | None	None	运行时环境配置（python_version、pypi依赖等），可传JSON字符串或字典。 关于runtime_env的更多信息，请参见**2.1.4.12.2 runtime_env说明**。
description	否	str | None	None	函数描述。
pretty_name	否	str | None	None	展示名，为None时使用name。
is_strict	否	bool	True	函数是否严格（输入为NULL时是否直接返回NULL）。
volatility	否	str	“VOLATILE”	函数易变性：IMMUTABLE/STABLE/VOLATILE。
is_batch	否	bool	False	是否启用批量处理模式。
batch_size	否	int | None	None	批量处理每批大小；is_batch=True时建议提供，须为正整数。
cpu_arch	否	str | None	None	CPU架构，取值arm或x86。
cpus	否	float | None	None	CPU核数，最小0.33。
memory	否	int | None	None	内存大小，单位MB，最小 1。
device_type	否	str | None	None	设备类型，例如npu、gpu。
devices	否	float | None	None	设备数量，需为正数。

版权所有 © 华为云计算技术有限公司

返回值

返回Function对象，其中包含LakeFormation分配的version_id、resource_uris（OBS包地址）及parameters（签名、运行时环境等元数据）。

示例代码

说明

- 若未提供signature，SDK会通过infer_signature从UDF的类型注解推断签名。类形式UDF取其__call__ 方法。
- 类定义形式UDF会抽取构造函数定义（ConstructorDef）并记录class_name，便于Aura引擎定位类。
- 类实例形式UDF通过cloudpickle序列化实例，class_name取自实例类型。

```
# 1. 定义带类型注解的UDF
def to_celsius(fahrenheit: float) -> float:
    return (fahrenheit - 32.0) * 5.0 / 9.0

# 2. 注册到LakeFormation
func = client.create_udf(
    udf=to_celsius,
    name="to_celsius",
    description="华氏度转摄氏度",
    runtime_env={"python_version": "3.11", "pypi": []},
)

print(f"Registered: {func.function_name}, version={func.version_id}")
```

2.1.4.6 创建 UDTF

将一个表函数（UDTF）注册到LakeFormation。表函数可接收输入并返回多行多列结果。

前提条件

- 已完成SDK安装。具体操作，请参见**2.1.4.2 Catalog SDK获取与安装**。
- 已准备好UDF可调用对象（函数、函数类或类实例），并标注了类型注解。
- 已开通OBS与LakeFormation服务，并获取对应endpoint、实例 ID、项目ID等。

函数签名

```
client.create_udtf(
    udtf, /, *,
    database=None,
    name,
    signature=None,
    imports=None,
    runtime_env=None,
    description=None,
    pretty_name=None,
    is_strict=True,
    volatility="VOLATILE",
    is_batch=False,
    batch_size=None,
    cpu_arch=None,
    cpus=None,
    memory=None,
    device_type=None,
    devices=None,
) -> Function
```


版权所有 © 华为云计算技术有限公司

参数说明

参数	是否必选	类型	默认值	说明
udtf	是	Any（位置参数）	-	UDTF 可调用对象（函数或可调用类）。
database	否	str | None	default_database	目标数据库名；为None时使用客户端默认数据库。
name	是	str	-	函数名，须以字母开头，仅含字母/数字/下划线，长度 3–128。
signature	否	Signature | None	None	显式函数签名；为None时由UDF注解自动推断。 关于signature的更多信息，请参见**2.1.4.12.1 signature说明**。
imports	否	str | Iterable[str] | None	None	需要打入UDF包的文件/目录路径或URL列表。
runtime_env	否	str | Mapping | None	None	运行时环境配置（python_version、pypi 依赖等），可传JSON字符串或字典。 关于runtime_env的更多信息，请参见**2.1.4.12.2 runtime_env说明**。
description	否	str | None	None	函数描述。
pretty_name	否	str | None	None	展示名；为None时使用name。
is_strict	否	bool	True	函数是否严格（输入为NULL时是否直接返回NULL）。
volatility	否	str	“VOLATILE”	函数易变性：IMMUTABLE/STABLE/VOLATILE。
is_batch	否	bool	False	是否启用批量处理模式。
batch_size	否	int | None	None	批量处理每批大小；is_batch=True时建议提供，须为正整数。
cpu_arch	否	str | None	None	CPU架构，取值arm或x86。
cpus	否	float | None	None	CPU核数，最小0.33。
memory	否	int | None	None	内存大小，单位MB，最小1。

版权所有 © 华为云计算技术有限公司

参数	是否必选	类型	默认值	说明
device_type	否	str | None	None	设备类型，例如npu、gpu。
devices	否	float | None	None	设备数量，需为正数。

返回值

返回Function对象。

示例代码

```
import pyarrow as pa

def split_range(start: int, end: int) -> pa.Table:
    return pa.table({"v": list(range(start, end))})

func = client.create_udtf(
    udf=split_range,
    name="split_range",
    description="生成 [start, end) 区间整数表",
)
print(f"UDTF registered: {func.function_name}")
```

2.1.4.7 创建 UDAF

将一个聚合函数（UDAF）注册到LakeFormation。聚合函数对多行数据进行聚合计算。除udf_type固定为udaf外，参数与create_udf完全一致。

前提条件

- 已完成SDK安装。具体操作，请参见**2.1.4.2 Catalog SDK获取与安装**。
- 已准备好UDF可调用对象（函数、函数类或类实例），并标注了类型注解。

函数签名

```
client.create_udaf(
    udaf, /, *,
    database=None,
    name,
    signature=None,
    imports=None,
    runtime_env=None,
    description=None,
    pretty_name=None,
    is_strict=True,
    volatility="VOLATILE",
    is_batch=False,
    batch_size=None,
    cpu_arch=None,
    cpus=None,
    memory=None,
    device_type=None,
    devices=None,
) -> Function
```


版权所有 © 华为云计算技术有限公司

参数说明

参数	是否必选	类型	默认值	说明
udaf	是	Any（位置参数）	-	udaf可调用对象（函数或可调用类）。
database	否	str | None	default_database	目标数据库名；为None时使用客户端默认数据库。
name	是	str	-	函数名，须以字母开头，仅含字母/数字/下划线，长度3–128。
signature	否	Signature | None	None	显式函数签名；为None时由UDF注解自动推断。 关于signature的更多信息，请参见**2.1.4.12.1 signature说明**。
imports	否	str | Iterable[str] | None	None	需要打入UDF包的文件/目录路径或URL列表。
runtime_env	否	str | Mapping | None	None	运行时环境配置（python_version、pypi 依赖等），可传 JSON 字符串或字典。 关于runtime_env的更多信息，请参见**2.1.4.12.2 runtime_env说明**。
description	否	str | None	None	函数描述。
pretty_name	否	str | None	None	展示名；为None时使用name。
is_strict	否	bool	True	函数是否严格（输入为NULL时是否直接返回NULL）。
volatility	否	str	“VOLATILE”	函数易变性：IMMUTABLE/STABLE/VOLATILE。
is_batch	否	bool	False	是否启用批量处理模式。
batch_size	否	int | None	None	批量处理每批大小；is_batch=True 时建议提供，须为正整数。
cpu_arch	否	str | None	None	CPU架构，取值arm或x86。
cpus	否	float | None	None	CPU核数，最小0.33。
memory	否	int | None	None	内存大小，单位MB，最小1。

版权所有 © 华为云计算技术有限公司

参数	是否必选	类型	默认值	说明
device_type	否	str | None	None	设备类型，例如npu、gpu。
devices	否	float | None	None	设备数量，需为正数。

返回值

返回Function对象。

示例代码

```
class WeightedSum:
    """UDAF：加权求和 sum(value * weight)。

    accumulate 逐行累积中间态（sum_vw），merge 合并分区，
    finish 返回最终聚合结果。
    """

    def __init__(self):
        self._agg_state = 0.0

    @property
    def aggregate_state(self) -> float:
        return self._agg_state
    def accumulate(self, value: int, weight: int) -> None:
        """逐行累积：value * weight 求和"""
        if value is not None and weight is not None:
            self._agg_state += value * weight
    def merge(self, other_state: float) -> None:
        """合并其他分区的中间态"""
        self._agg_state += other_state
    def finish(self) -> float:
        """返回最终聚合结果"""
        return self._agg_state

func = client.create_agg_function(
    WeightedSum,
    database=database,
    name="weighted_sum_udaf",
    signature=sig,
    comment="UDAF: 加权求和 sum(value * weight)",
)
```

2.1.4.8 查询函数详情

按名称从LakeFormation获取单个函数的元数据。

前提条件

- 已完成SDK安装。具体操作，请参见**2.1.4.2 Catalog SDK获取与安装**。
- 对应数据库存在目标函数。

函数签名

```
client.get_function(*, database=None, name) -> Function
```


版权所有 © 华为云计算技术有限公司

参数说明

参数	是否必选	类型	默认值	说明
database	否	str | None	default_database	数据库名；为None时使用默认数据库。
name	是	str	-	函数名。

返回值

返回Function对象，含 function_name、resource_uris、version_id、parameters、class_name等字段。

异常说明

异常	触发条件
FunctionNotFoundError	指定函数不存在。
LakeFormationAPIError	LakeFormation API 调用失败（网络、鉴权、5xx 等）。

示例代码

```
func = client.get_function(name="to_celsius")
print(func.function_name, func.version_id)
print(func.parameters.signature.to_dict())
```

2.1.4.9 查询函数列表

列出指定数据库下的全部函数。

前提条件

- 已完成SDK安装。具体操作，请参见**2.1.4.2 Catalog SDK获取与安装**。
- 已存在数据库。

函数签名

```
client.list_function(*, database=None) -> list[Function]
```

参数说明

参数	是否必选	类型	默认值	说明
database	否	str | None	default_database	数据库名；为None时使用默认数据库。

版权所有 © 华为云计算技术有限公司

返回值

返回list[Function]，数据库中的全部函数对象。

示例代码

```
for func in client.list_function():
    print(func.function_name, func.parameters.udf_type)
```

2.1.4.10 删除函数

删除LakeFormation中的函数记录，并清理其在OBS上的UDF包。删除流程如下：

1. 获取函数元数据（用于定位resource_uris）。
2. 删除对应的OBS包（函数已部分删除时按前缀清理以避免泄漏）。
3. 从LakeFormation删除函数记录。
4. 若OBS清理失败但LakeFormation删除成功，抛出异常以提示存在孤儿包。

前提条件

- 已完成SDK安装。具体操作，请参见**2.1.4.2 Catalog SDK获取与安装**。
- 目标数据库与函数名已知（若函数元数据不可用，SDK将按{database}/{name}/前缀清理OBS）。

函数签名

```
client.delete_function(*, database=None, name) -> None
```

参数说明

参数	是否必选	类型	默认值	说明
database	否	str | None	default_database	数据库名；为None时使用默认数据库。
name	是	str	-	函数名。

返回值

无返回值（None）。

异常说明

异常	触发条件
FunctionNotFoundError	函数不存在（此异常在删除流程中被安全忽略，SDK改用前缀清理OBS）。
LakeFormationAPIError	LakeFormation API调用失败（网络、鉴权、5xx 等），需向上传播以告知调用方清理未完成。

版权所有 © 华为云计算技术有限公司

异常	触发条件
OBSOperationError	OBS清理失败，但LakeFormation删除已成功时抛出，提示存在孤儿OBS包。

示例代码

```
client.delete_function(name="to_celsius")
print("Function deleted and OBS package cleaned")
```

说明

- 当函数元数据不可用时，SDK会按数据库/函数名/前缀清理OBS对象，避免遗留孤儿包。
- 若OBS删除失败但LakeFormation记录已删除，SDK会抛出OBSOperationError，调用方需据此排查孤儿包。

2.1.4.11 注册 Data-Juicer 算子

将一个Data-Juicer算子（Filter或Mapper子类实例）注册为标量UDF。内部通过dj_operator_to_aura_udf将算子适配为可调用UDF，再委托给create_udf。

由于cloudpickle对Data-Juicer算子类采用按引用序列化，SDK会自动将py-data-juicer加入runtime_env.pypi，确保Aura运行时已安装该包；用户自定义的runtime_env优先级高于自动注入项。

前提条件

已安装data_juicer包（本地用于构造算子实例）。

函数签名

```
client.create_datajuicer_operator(
    operator, /, *,
    name,
    input_columns,
    output_columns=None,
    database=None,
    signature=None,
    imports=None,
    runtime_env=None,
    description=None,
    pretty_name=None,
    is_strict=True,
    volatility="VOLATILE",
    is_batch=False,
    batch_size=None,
    cpu_arch=None,
    cpus=None,
    memory=None,
    device_type=None,
    devices=None,
) -> Function
```


版权所有 © 华为云计算技术有限公司

参数说明

参数	是否必选	类型	默认值	说明
operator	是	Any（位置参数）	-	已初始化的 Data-Juicer 算子实例（Filter或Mapper子类）。
name	是	str	-	注册到LakeFormation的函数名。
input_columns	是	list[str]	-	算子期望的输入列名，顺序与引擎传入的位置参数对齐。
output_columns	否	list[str] | None	None	算子产出的列名；Mapper操作必需，Filter操作忽略。
database	否	str | None	default_database	目标数据库名。
signature	否	Signature | None	None	显式函数签名。关于signature的更多信息，请参见**2.1.4.12.1 signature说明**。
imports	否	str | Iterable[str] | None	None	打入UDF包的额外文件/目录或URL。
runtime_env	否	str | Mapping | None	None	运行时环境配置；将与 {"pypi": ["py-data-juicer"]} 合并。 关于runtime_env的更多信息，请参见**2.1.4.12.2 runtime_env说明**。
description	否	str | None	None	函数描述。
pretty_name	否	str | None	None	展示名。
is_strict	否	bool	True	是否严格。
volatility	否	str	“VOLATILE”	函数易变性。
is_batch	否	bool	False	是否批量处理。
batch_size	否	int | None	None	批量大小。
cpu_arch	否	str | None	None	CPU 架构（arm/x86）。
cpus	否	float | None	None	CPU 核数（最小0.33）。
memory	否	int | None	None	内存（MB，最小1）。
device_type	否	str | None	None	设备类型。

版权所有 © 华为云计算技术有限公司

参数	是否必选	类型	默认值	说明
devices	否	float | None	None	设备数量。

返回值

返回Function对象（实际由create_udf返回）。

异常说明

异常	触发条件
ValueError	operator 为 Mapper 但未提供 output_columns。
ImportError	未安装 data_juicer 包。

示例代码

```
from data_juicer.ops.mapper import LowercaseMapper

op = LowercaseMapper(text_column="content")

func = client.create_datajuicer_operator(
    operator=op,
    name="dj_lowercase",
    input_columns=["content"],
    output_columns=["content"],          # Mapper 必填
    description="Data-Juicer 小写映射算子",
)
```

说明

- 支持Mapper和Filter算子类型。
- Filter算子无需output_columns，传入也会被忽略。
- runtime_env合并规则：用户配置优先，py-data-juicer作为补充依赖自动加入；重复包名按去重策略保留。

2.1.4.12 公共参数说明

2.1.4.12.1 signature 说明

signature用于描述UDF的函数签名，即函数的输入参数和返回值的类型信息。它由Parameter 列表 + return_annotation组成，注册函数时被写入LakeFormation元数据，供引擎调用时进行参数与类型的匹配。

signature 参数说明

signature对象由Parameter列表与return_annotation组成。

版权所有 © 华为云计算技术有限公司

表 2-3 属性说明

属性	类型	说明
name	str	参数名。
kind	Parameter Kind	参数类别：POSITIONAL_ONLY/ POSITIONAL_OR_KEYWORD/KEYWORD_ONLY/ VAR_POSITIONAL/VAR_KEYWORD。
default	Any	默认值；无默认值时为EMPTY。
annotation	Any	类型注解；无注解时为EMPTY。

什么情况下使用

场景	推荐做法
UDF有完整、准确的类型注解	传signature=None，交由SDK自动推断。
UDF注解与实际期望签名不一致，或需覆盖	显式传入Signature对象。
批量模式下需要使用元素类型作为签名	显式传入带元素类型的Signature，SDK据此映射Lakeformation列类型。

自动推断规则（signature=None 时）

当signature未显式传入（为 None）时，SDK依据UDF的代码形态与类型注解自动推断函数签名，无需手动构造Signature对象。推断失败时抛出SignatureInferenceError。

语法示例:

为UDF标注参数与返回值的类型注解，即可触发自动推断。

```
# 带类型注解的 UDF（推荐写法）
def to_celsius(fahrenheit: float) -> float:
    return (fahrenheit - 32) * 5 / 9

# signature=None 时，SDK 自动推断出等价签名：
#   signature = [Parameter(name="fahrenheit", kind=POSITIONAL_OR_KEYWORD, default=EMPTY, annotation=float)]
#   return_annotation = float
client.create_udf(to_celsius, name="to_celsius")
```

推断规则:

UDF 形式	推断行为
普通函数	使用inspect.signature + get_type_hints读取参数与返回注解。

版权所有 © 华为云计算技术有限公司

UDF 形式	推断行为
类定义形式UDF	检查类中的__call__方法，读取其注解；若类无自定义__call__，抛出SignatureInferenceError。
类实例形式UDF	检查实例所属类的__call__方法；实例本身已通过cloudpickle序列化，class_name取自 type(udf).__name__。
绑定方法	自动解包为普通函数推断。

说明

- 所有参数必须带类型注解；FunctionSignature.from_signature 会对无注解参数抛出ValueError。
- 裸容器类型（如list、dict而无元素类型）不被接受，须使用 list[T]、dict[K, V]等带元素类型的注解。
- 若signature为None且推断失败（例如函数无类型注解、类无 __call__），SDK会抛出SignatureInferenceError。

批量模式（is_batch=True）下的 signature 要求

批量模式（is_batch=True）下，UDF一次性处理一批数据而非单行数据，因此signature的参数注解必须使用批量类型（pyarrow或pandas的数组/表结构），且所有参数的批量类型必须一致，以匹配引擎的批量传参方式。

语法示例

使用批量类型注解定义批量UDF，并注册时开启 is_batch。

```
import pyarrow as pa

def double_batch(values: pa.Array) -> pa.Array:
    return pa.array([v * 2 for v in values])

client.create_udf(double_batch, name="double_batch", is_batch=True, batch_size=1024)
```

批量类型对照

批量类型	检测到的input_type
pyarrow.Array/ pyarrow.Table	pyarrow
pandas.Series/ pandas.DataFrame	pandas

说明

- 若signature由注解自动推断且参数为非批量元素类型，SDK抛ValueError。
- 若显式传入signature使用元素类型，SDK会改从UDF自身注解推断input_type。
- 建议使用 list[T] 携带元素类型信息，便于LakeFormation存储列原始数据类型；裸pyarrow.Array、pandas.Series不携带元素类型时会在from_signature阶段报错。

版权所有 © 华为云计算技术有限公司

2.1.4.12.2 runtime_env 说明

runtime_env用于声明UDF运行所需的运行时环境，包括 Python 版本、PyPI 依赖包、环境变量等。该配置会序列化为JSON字符串存入LakeFormation的parameters.runtime_env字段，UDF在Aura引擎运行时按此配置准备环境。

作用

- **保证可执行**：UDF依赖的第三方包、Python版本在引擎侧按声明安装，避免"本地能跑、远端报错"。
- **版本可控**：支持固定Python版本（3.10/3.11）与指定包版本，保证运行环境一致。
- **跨端同步**：环境配置随函数元数据持久化，与函数生命周期绑定。

什么情况下使用

表 2-4 场景说明

场景	用法
UDF 依赖第三方 Python 包	在pypi/pip中声明包名（如 ["numpy"]）。
需指定 Python 版本	设置 python_version（仅支持3.11或3.10）。
需注入运行环境变量	设置env_vars。
无特殊依赖	传None即可，SDK构造默认环境（python_version=3.11）。

构造方式

runtime_env接受以下三种形式，SDK内部统一构造为RuntimeEnv对象：

表 2-5 构造方式说明

传入形式	示例	说明
JSON字符串	`'{"python_version": "3.11", "pypi": ["numpy"]}'`	经json.loads解析。
Mapping/dict	`{"python_version": "3.11", "pypi": ["numpy"]}`	直接更新到内部数据结构。
None	-	构造默认RuntimeEnv，python_version设置为3.11。

版权所有 © 华为云计算技术有限公司

常见键

表 2-6 常见键说明

键	类型	说明
python_version	str	Python版本，仅支持3.11（默认）或3.10；传入其他版本会抛出ValueError。
pypi/pip	list[str]	PyPI依赖包列表；合并时按包名去重。
env_vars	dict	运行时环境变量。

默认值与校验行为

- 当runtime_env为None时，SDK构造空RuntimeEnv，并自动将python_version设为3.11。
- 若runtime_env提供了python_version，SDK校验其取值；非3.11/3.10时抛出ValueError。
- RuntimeEnv是类字典的可变映射（MutableMapping），支持to_dict()、to_json()、merge(other) 方法。merge采用深合并策略：dict递归合并、list去重合并（pypi/pip/dependencies 键按包名去重）、标量冲突时左值优先。

返回值

返回Function对象，其中包含LakeFormation分配的version_id、resource_uris（OBS包地址）及parameters（签名、运行时环境等元数据）。

2.1.4.13 异常类说明

SDK在调用LakeFormation/OBS等外部服务，或进行内部处理（类型推断、UDF打包）失败时，会抛出对应的异常类。您可以通过异常类别，快速判断是服务问题、存储问题、参数问题还是类型问题。SDK异常均继承自基类AIDatalakeCatalogError。

表 2-7 异常类说明

异常类	说明	关键属性
AIDatalakeCatalogError	所有SDK异常的基类。	-
LakeFormationAPIError	LakeFormation REST API调用失败。	status_code、function_name
OBSOperationError	OBS上传/删除操作失败。	obs_uri
TypeMappingError	Python/LakeFormation/Aura/Arrow类型无法映射。	-
FunctionNotFoundError	LakeFormation中未找到指定函数。	function_name、database

版权所有 © 华为云计算技术有限公司

异常类	说明	关键属性
FunctionAlreadyExistsError	LakeFormation中已存在同名函数。	function_name、database
SignatureInferenceError	从可调用对象推断签名失败。	callable_name
UDFPackagingError	UDF打包（生成zip）失败。	-

说明

参数校验类错误（如函数名非法、资源参数越界等）直接以ValueError抛出，不归属上述SDK异常层级。

2.1.5 Aura DataFrame SDK（Python）

2.1.5.1 Aura DataFrame SDK 概述

Aura DataFrame SDK提供了类Pandas的Python DataFrame开发能力，方便用户使用Python编写数据处理作业，并利用Aura内核高效的计算能力，为数据科学家、AI工程师等提供易用、高效的数据处理能力。

Aura DataFrame SDK基于Ibis Python DataFrame开源框架实现，将Ibis前端框架与Aura引擎对接。用户可基于熟悉的Ibis DataFrame API编写数据处理脚本，Ibis框架将Python API翻译为Aura引擎可执行的SQL语句并下发，实现计算逻辑在Aura引擎的高效处理。支持Ibis的API使用，具体Ibis API请参见**Ibis官方文档**。

Aura DataFrame SDK支持的环境：

- Python版本推荐使用3.11。
- UDF运行时环境的Python版本支持3.10和3.11系列，建议使用3.11.9版本。

表 2-8 不同业务场景支持的 Python 版本

业务场景	内置的Python包	支持的Python版本
Python UDF	• cloudpickle • msgpack • pyarrow • protobuf • pandas • numpy • redis • opentelemetry-api • opentelemetry-sdk • opentelemetry-exporter-otlp-proto-grpc	Python 3.10和Python 3.11

版权所有 © 华为云计算技术有限公司

业务场景	内置的Python包	支持的Python版本
多模态数据处理	• av (PyAV) • pillow • soundfile • decord • opencv-python-headless • numpy • scipy	Python 3.11
AI推理/模型服务（UDF）	• torch • torchvision • onnx • onnxruntime • torch_npu • onnxruntime-cann • vlm • yreserve • Ascend-cann-toolkit • kernels	Python 3.11

如果需要在 DataFrame开发中管理对象与权限（如创建带特殊表格式的表、配置对象授权），请参见《Aura语法参考》的"使用Aura SQL开发的流程"导读章节。

约束与限制

Aura DataFrame SDK仅支持AuraJobV2端点。

Aura DataFrame SDK 功能介绍

Aura DataFrame SDK 提供以下类型的接口能力。

表 2-9 Aura DataFrame SDK

接口	作用
2.1.5.6 Session APIs	负责会话建立与管理，是使用SDK的入口。包括建立连接（ai_lake.connect）、配置UDF部署工作区（set_function_staging_workspace）、加载数据集（load_dataset）、执行SQL（sql）等。所有数据操作前，需先通过该类接口建立会话。

版权所有 © 华为云计算技术有限公司

接口	作用
2.1.5.7 Dataset APIs	对数据集进行数据处理，是开发的核心。包括列操作（select_columns、add_column、drop_columns、rename_columns）、行筛选（filter、limit、take）、聚合统计（aggregate、groupby、max/min/mean）、数据变换（map、map_batches、flat_map、order_by、join）、触发执行（execute、show）、生成执行计划（explain_plan/explain_performance）等。
2.1.5.8 Table APIs	对数据库表进行管理。包括创建表（create_table）、删除表（drop_table/delete_table）、描述表（describe_table）、列出表（list_tables）、打开表（open_table），以及表的增删改（insert、update、delete）和列管理（add_columns、drop_columns、alter_column_type、rename_column）等。
2.1.5.9 I/O APIs	将数据写入湖仓存储格式，实现与数据湖的无缝集成。支持write_iceberg、write_orc、write_parquet等接口。
2.1.5.10 User-Defined Function APIs	用户自定义函数的管理与使用。包括创建函数（create_scalar_function、create_agg_function、create_table_function）、获取函数（get_function）、列出函数（list_functions）、删除函数（delete_function）等，支持Scalar UDF、Class UDF、UDTF、UDAF等函数类型。
2.1.5.11 AI APIs	调用大语言模型能力。包括文本生成（ai_generate）、模型注册（create_model）、模型列表（list_models）、模型详情（describe_model）、模型删除（drop_model）等，支持在DataFrame中直接进行大模型推理。

2.1.5.2 Aura DataFrame SDK 获取与安装

您可以通过pip命令一键安装Aura DataFrame SDK，安装过程中会自动下载SDK依赖的第三方库。本节操作介绍如何获取并安装SDK的操作指导。

前提条件

已安装Python 3.11环境。具体操作，请参见2.1.2 Python环境配置。

获取 SDK

下载SDK文件，并建议您同时下载.sha256文件，该文件可用于验证SDK包的完整性。验证方法请参考验证SDK包的完整性。

版权所有 © 华为云计算技术有限公司

表 2-10 获取 SDK 文件说明

SDK文件	说明	下载链接
huawei_aura_connectorapi-x.x.x-py3-none-any.whl	支持通过原生API或标准数据库接口（PEP249）执行SQL查询。	• **SDK安装包链接** • **校验文件（SHA256）链接**
huawei_aura_frame-x.x.x.x-py3-none-any.whl	基于Ibis框架，支持用类Dataframe/Dataset语法编写数据处理作业并自动转换为SQL在Aura引擎执行。	• **SDK安装包链接** • **校验文件（SHA256）链接**

验证 SDK 包的完整性

下载SDK安装包时，您需要同时下载.sha256文件，用于验证SDK包的完整性。本节介绍使用.sha256文件验证SDK包的完整性的操作步骤。

1. **查询SDK安装包的SHA - 256哈希值**
   - **在Windows系统中**
     i. 在开始菜单中搜索“cmd”打开命令窗口。
     ii. 使用cd命令切换到保存SDK安装包的文件夹目录。
     例如：SDK安装包保存在D盘的“SDK”文件夹中，输入**cd D:\SDK**。
     iii. 执行以下命令
     certutil -hashfile *SDK安装包文件名* SHA256
     将“SDK安装包文件名”替换为实际的SDK安装包文件名，如**certutil -hashfile huawei_aura_connectorapi-x.x.x-py3-none-any.whl SHA256**。
     iv. 命令执行后，命令提示符会输出SDK安装包的SHA - 256哈希值。
   - **在Linux系统中**
     i. 使用cd命令切换到保存SDK安装包的文件夹目录。
     例如，如果SDK安装包保存在用户主目录下的“SDK”文件夹中，输入**cd ~/SDK**。
     ii. 输入查询命令：
     sha256sum *SDK安装包文件名*
     将“SDK安装包文件名”替换为实际的SDK安装包文件名，如**sha256sum sdk - huawei_aura_connectorapi-x.x.x-py3-none-any.whl**。
     iii. 命令执行后，终端会输出SDK安装包的SHA - 256哈希值。
2. **对比哈希值**
   a. 使用文本编辑器打开下载的.sha256文件。
   b. 在.sha256文件中找到SDK安装包的SHA - 256哈希值。
   通常是一个长字符串，例如“a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6xxxxxxxxxxxxxxxxxxxxxxxx”。
   c. 将步骤**1查询SDK安装包的SHA - 256哈希值**得到的SDK安装包的SHA - 256哈希值与.sha256文件中的哈希值进行对比。

版权所有 © 华为云计算技术有限公司

- 如果两个哈希值完全一致，说明SDK安装包在下载过程中没有被篡改，文件是完整的。
- 如果哈希值不一致，这可能意味着SDK安装包在下载过程中出现了错误，或者文件被恶意修改。

  此时，建议重新下载SDK安装包和对应的.sha256文件，然后再次进行验证。

安装 SDK

步骤1 打开命令行窗口，进入SDK文件所在目录。

步骤2 执行如下命令安装SDK，安装过程中会自动下载第三方依赖库。

```
pip install huawei_aura_connectorapi-x.x.x-py3-none-any.whl
pip install huawei_aura_frame-x.x.x.x-py3-none-any.whl
```

x.x.x.x和x.x.x为具体版本号，以实际为准。

步骤3 执行以下命令，查看SDK版本。

```
python -c "import aura_frame; print(aura_frame.__version__)"
```

回显信息类似如下所示，表示安装SDK成功。

```
3.9.0.1
```

----结束

2.1.5.3 Aura DataFrame SDK 新手入门

本节介绍使用Aura DataFrame SDK开发数据的完整流程。

流程概览

步骤	操作	说明
步骤1：获取并安装SDK安装包	获取并安装SDK安装包	安装Aura DataFrame SDK包。
步骤2：建立连接	建立连接	通过ai_lake.connect()创建会话；使用UDF前配置部署工作区。
步骤3：准备数据	准备数据	加载已有数据集、创建数据表、写入数据。
步骤4：数据处理	数据处理	使用DataFrame API进行筛选、聚合、排序、连接等加工。
步骤5：更多数据开发操作	更多数据开发操作	根据业务需要使用UDF与多模态处理能力。
步骤6：查看作业运行结果	查看作业运行结果	验证作业是否按预期运行，关闭会话释放资源。

版权所有 © 华为云计算技术有限公司

步骤 1：获取并安装 SDK 安装包

获取SDK包后，打开命令行窗口，进入SDK文件所在目录，执行如下命令安装SDK，安装过程中会自动下载第三方依赖库。

```
pip install huawei_aura_connectorapi-x.x.x-py3-none-any.whl
pip install huawei_aura_frame-x.x.x.x-py3-none-any.whl
```

x.x.x.x和x.x.x为具体版本号，以实际为准。详细操作，请参见**2.1.5.2 Aura DataFrame SDK获取与安装**。

步骤 2：建立连接

通过ai_lake.connect() 创建与 Aura 端点的会话连接；使用UDF前还需配置部署工作区。

详细操作，请参见**2.1.5.4.1 建立与AuraJobV2端点的连接**。

创建会话，配置连接信息（Endpoint、工作空间ID、Catalog、AK/SK、数据库等）：

```
import os
import logging
from aura_frame.multimodal import ai_lake

con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),          # AuraJobV2端点所在的终端节点（Endpoint）
    aura_endpoint_id=os.getenv("aura_endpoint_id"),      # AuraJobV2端点ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),  # AuraJobV2端点名称
    aura_workspace_id=os.getenv("aura_workspace_id"),    # 工作空间ID
    lf_catalog_name=os.getenv("lf_catalog_name"),        # LakeFormation Catalog名称
    access_key=os.getenv("access_key"),                  # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),                  # 访问密钥（SK）
    default_database="postgres",                         # 默认数据库
    use_single_cn_mode=True,                             # 是否开启单CN模式
    logging_level=logging.INFO,                          # 设置日志级别
)
```

步骤 3：准备数据

创建数据表、写入数据、加载已有数据集。

1. 新创建表和数据。
   a. 在LakeFormation中创建数据库，例如：database1。具体操作，请参见**创建LakeFormation实例并规划元数据**。
   b. 创建数据表（库中无目标表时）。

      create_table支持通过store指定表格式（ORC/PARQUET/ICEBERG等）、通过partition_by指定分区列，详见**create_table**。

      ```
      import ibis
      table_name = "table_name"
      table_schema = ibis.schema({"col_a": "int32", "col_b": "string"})
      con.sql(f'DROP TABLE IF EXISTS database1."{table_name}";')
      con.create_table(
          table_name,
          schema=table_schema,
          database="database1",
          table_format="orc",
          if_not_exists=True,
      )
      ```

   c. 通过SQL，写入数据。

      ```
      con.sql("INSERT INTO database1.table_name VALUES(1, 's1'), (2, 's2')")
      ```

      详细操作请参考2.1.5.4.3 创建表并写入数据。

版权所有 © 华为云计算技术有限公司

2. 加载已有数据集。

```
ds = con.load_dataset("table_name", database="database1") # 从指定数据库加载表，建立表对象，table_name和database1需要替换为实际的表名和数据库名称
```

步骤 4：数据处理

使用DataFrame API对数据进行筛选、聚合、排序、连接等加工。

DataFrame 操作采用懒加载机制，以下操作只是“记录”计算意图，调用execute()才真正在Aura引擎执行并返回Pandas DataFrame。

详细操作请参考**2.1.5.4.5 数据加工与分析**。

```
# 选择列
ds=ds.select_columns([ds.col_a, ds.col_b])
# 筛选行
ds=ds.filter(ds.col_a > 0)
# 分组聚合（对数值列col_a做聚合）
ds=ds.groupby("col_a").aggregate(sum_b=ds.col_a.sum(), avg_b=ds.col_a.mean())
# 排序
ds=ds.order_by(ds.col_a.desc())
# 触发执行，返回Pandas DataFrame（关键一步）
df=ds.execute()
print(df)
```

步骤 5：更多数据开发操作

根据业务需要，进一步使用UDF与多模态处理能力。

- 注册和使用Scalar UDF/Class UDF/UDAF ，请参考**2.1.5.4.6 使用Scalar UDF处理数据**、**2.1.5.4.7 使用Class UDF处理数据**、**2.1.5.4.8 使用聚合统计的UDAF**
- UDF类型、注册方式、开发规范，请参考**2.1.5.12 用户自定义函数UDF**。

步骤 6：查看作业运行结果

作业运行成功后，运行日志会回显执行过程与结果（例如建表、执行SQL、返回查询结果等），可据此验证作业是否按预期运行。

更多示例参考**2.1.5.4 使用Aura DataFrame SDK提交作业**。

数据操作完成后关闭会话释放资源：

```
con.close() # 关闭会话，释放资源
```

2.1.5.4 使用 Aura DataFrame SDK 提交作业

2.1.5.4.1 建立与 AuraJobV2 端点的连接

通过ai_lake.connect()建立与AuraJobV2端点的会话连接，后续所有数据操作均基于该连接进行。

前提条件

- 已安装Aura DataFrame SDK。具体操作，请参见**2.1.5.2 Aura DataFrame SDK获取与安装**。
- 已创建AuraJobV2端点，并获取端点名称。具体操作，请参见**创建Aura引擎端点**。

版权所有 © 华为云计算技术有限公司

- 已创建LakeFormation实例，并在该实例中创建Catalog及数据库。具体操作，请参见**创建LakeFormation实例并规划元数据**。
- 已创建并行文件系统。具体操作，请参见**创建并行文件系统**。
- 代码中涉及os.getenv()中的参数已配置到环境变量。
- 运行时环境要求Python版本为3.11。

示例

```
import os
import logging
from aura_frame.multimodal import ai_lake

con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),         # AuraJobV2端点所在的终端节点（Endpoint）
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2端点ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2端点名称
    aura_workspace_id=os.getenv("aura_workspace_id"),  # 工作空间ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog名称
    access_key=os.getenv("access_key"),                # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),                # 访问密钥（SK）
    default_database="postgres",                       # LakeFormation数据库名称
    use_single_cn_mode=True,   # 是否开启单CN模式
    logging_level=logging.INFO # 设置日志级别
)
```

ai_lake.connect() 参数信息如下：

参数	含义	类型	默认值
aura_endpoint	指定连接的终端节点（Endpoint）信息，请参见**地区和终端节点**。	str	None
aura_endpoint_id	指定连接的AuraJobV2端点ID。	str	None
aura_endpoint_name	指定连接的AuraJobV2端点名称。	str	None
aura_workspace_id	指定连接的AuraJobV2端点所在的工作空间ID。	str	None
lf_catalog_name	指定连接的LakeFormation Catalog名称。	str	None
access_key	指定连接的访问密钥ID（AK）。	str	None
secret_key	指定连接的访问密钥（SK）。	str	None
default_database	指定连接的默认数据库名称。	str	None
use_single_cn_mode	表示是否开启单CN模式。	bool	False
logging_level	日志级别。	int	logging.INFO

以下示例展示建立连接并验证连通性的完整流程：

版权所有 © 华为云计算技术有限公司

```
import os
import logging
from aura_frame.multimodal import ai_lake

con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),          # AuraJobV2端点所在的终端节点（Endpoint），例如:
    aura_endpoint_id=os.getenv("aura_endpoint_id"),      # AuraJobV2端点ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"), # AuraJobV2端点名称
    aura_workspace_id=os.getenv("aura_workspace_id"),    # 工作空间ID
    lf_catalog_name=os.getenv("lf_catalog_name"),        # LakeFormation Catalog名称
    lf_instance_id=os.getenv("lf_instance_id"),          # LakeFormation实例ID
    project_id=os.getenv("project_id"),                  # 项目ID
    access_key=os.getenv("access_key"),                  # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),                  # 访问密钥（SK）
    default_database="postgres",                         # 默认数据库
    use_single_cn_mode=True,                             # 是否开启单CN模式
    logging_level=logging.INFO,                          # 设置日志级别
)
try:
    with con.sql("select 1;") as cur:                    # 执行SQL验证连通性
        result = cur.fetchall()
    print("连接成功, select 1 结果:", result)
finally:
    con.close()                                          # 关闭连接
```

相关文档

2.1.5.6 Session APIs

2.1.5.4.2 配置 UDF 部署工作区

使用UDF前，需配置UDF函数部署所需的临时工作区。

前提条件

- 已安装Aura DataFrame SDK。具体操作，请参见**2.1.5.2 Aura DataFrame SDK获取与安装**。
- 已创建AuraJobV2端点，并获取端点名称。具体操作，请参见**创建Aura引擎端点**。
- 已创建LakeFormation实例，并在该实例中创建Catalog及数据库。具体操作，请参见**创建LakeFormation实例并规划元数据**。
- 已创建并行文件系统。具体操作，请参见**创建并行文件系统**。
- 代码中涉及os.getenv()中的参数已配置到环境变量。
- 运行时环境要求Python版本为3.11。

示例

```
import os
from aura_frame.multimodal import ai_lake

target_database = "test"
#连接AuraJobV2端点，具体参数说明，请参见2.1.5.4.1 建立与AuraJobV2端点的连接。
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),        # AuraJobV2端点所在的终端节点（Endpoint）
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2端点ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2端点名称
    aura_workspace_id=os.getenv("aura_workspace_id"), # 工作空间ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog名称
    access_key=os.getenv("access_key"),                # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),                # 访问密钥（SK）
    default_database=target_database,                    # LakeFormation数据库名称
```


版权所有 © 华为云计算技术有限公司

```
    use_single_cn_mode=True,   # 是否开启单CN模式
    logging_level=logging.INFO # 设置日志级别
)
try:
    con.set_function_staging_workspace(
        obs_directory_base=os.getenv("obs_directory_base"), # OBS中UDF的存储路径
        obs_bucket_name=os.getenv("obs_bucket_name"),       # OBS的名称
        obs_server=os.getenv("obs_server"),                 # OBS终端节点（Endpoint）
        access_key=os.getenv("access_key"),                 # 访问密钥ID（AK）
        secret_key=os.getenv("secret_key"),                 # 访问密钥（SK）
    )
    print("UDF部署工作区配置完成")
finally:
    con.close()
```

set_function_staging_workspace()参数信息如下：

表 2-11 参数说明

参数	含义	类型	默认值
obs_directory_base	并行文件系统中UDF的存储路径。	str	None
obs_bucket_name	并行文件系统的名称。	str	None
obs_server	OBS访问地址，详情请参见终端节点（Endpoint）和访问域名。 指定连接的终端节点（Endpoint）信息，请参见**地区和终端节点**。	str	None
access_key	访问密钥（AK/SK）。具体获取，请参见**2.1.3 获取AKSK**。	str	None
secret_key		str	None
security_token	安全令牌（临时AK/SK场景下需要）。	str	None

相关文档

2.1.5.6 Session APIs

2.1.5.4.3 创建表并写入数据

创建数据表并写入数据，为后续查询和分析提供数据基础。

前提条件

- 已安装Aura DataFrame SDK。具体操作，请参见**2.1.5.2 Aura DataFrame SDK获取与安装**。
- 已创建AuraJobV2端点，并获取端点名称。具体操作，请参见**创建Aura引擎端点**。
- 已创建LakeFormation实例，并在该实例中创建Catalog及数据库。具体操作，请参见**创建LakeFormation实例并规划元数据**。

版权所有 © 华为云计算技术有限公司

- 已创建并行文件系统。具体操作，请参见**创建并行文件系统**。
- 代码中涉及os.getenv()中的参数已配置到环境变量。
- 运行时环境要求Python版本为3.11。

示例

本示例创建三个数据表（text_samples、sales_line_items、category_sales）并分别写入文本、销售明细和分类销售数据，为后续章节（**3.1.6.4.4**，**3.1.6.4.5**，**3.1.6.4.6**，**3.1.6.4.7**，**3.1.6.4.8**，**3.1.6.4.9**，**3.1.6.4.10**）的查询、分析和UDF使用提供数据基础。

1. 导入依赖并连接AuraJobV2端点。
2. 创建text_samples表（包含id和content列），并插入5条文本数据。
3. 创建sales_line_items表（包含quantity、unit_price、discount_rate、tax_rate、return_flag、line_status、ship_date共7列），并插入5条销售数据。
4. 创建category_sales表（包含category和amount列），并插入12条分类销售数据。
5. 关闭AuraJobV2端点连接。

```
import os
import logging
import ibis
from aura_frame.multimodal import ai_lake

target_database = "test"  #在LakeFormation中创建数据库，例如：test。具体操作，请参见创建LakeFormation实例并规划元数据
#连接AuraJobV2端点，具体参数说明，请参见2.1.5.4.1 建立与AuraJobV2端点的连接。
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),        # AuraJobV2端点所在的终端节点（Endpoint）
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2端点ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2端点名称
    aura_workspace_id=os.getenv("aura_workspace_id"),  # 工作空间ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog名称
    access_key=os.getenv("access_key"),               # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),               # 访问密钥（SK）
    default_database=target_database,                 # LakeFormation数据库名称
    use_single_cn_mode=True,   # 是否开启单CN模式
    logging_level=logging.INFO # 设置日志级别
)
try:
    table_name = "text_samples"
    table_schema = ibis.schema({"id": "int32", "content": "string"})
    con.sql(f'DROP TABLE IF EXISTS "{target_database}"."{table_name}";')
    con.create_table(
        table_name,
        schema=table_schema,
        database=target_database,
        table_format="orc",
        if_not_exists=True,
    )
    insert_sql = f'''
    INSERT INTO "{target_database}"."{table_name}" (id, content) VALUES
        (1, 'Hello, how are you today?'),
        (2, 'The quick brown fox jumps over the lazy cat.'),
        (3, 'Machine learning is transforming the way we process data.'),
        (4, 'Natural language processing enables computers to understand human language.'),
        (5, 'Tokenization is the first step in text processing pipelines.')
    '''
    con.sql(insert_sql)
    print(f"表 {table_name} 创建并写入数据完成")
    table_name = "sales_line_items"
    table_schema = ibis.schema({
```


版权所有 © 华为云计算技术有限公司

```
        "quantity": "int32",
        "unit_price": "float64",
        "discount_rate": "float64",
        "tax_rate": "float64",
        "return_flag": "string",
        "line_status": "string",
        "ship_date": "date",
    })
    con.sql(f'DROP TABLE IF EXISTS "{target_database}"."{table_name}";')
    con.create_table(
        table_name,
        schema=table_schema,
        database=target_database,
        table_format="orc",
        if_not_exists=True,
    )
    insert_sql = f'''
    INSERT INTO "{target_database}"."{table_name}" (quantity, unit_price, discount_rate, tax_rate,
return_flag, line_status, ship_date) VALUES
        (10, 1000.00, 0.05, 0.08, 'N', 'O', '1998-08-01'),
        (20, 2000.00, 0.10, 0.05, 'R', 'F', '1998-09-15'),
        (15, 1500.00, 0.08, 0.07, 'N', 'O', '1998-07-20'),
        (25, 2500.00, 0.05, 0.08, 'R', 'F', '1998-10-01'),
        (30, 3000.00, 0.02, 0.10, 'N', 'O', '1998-08-15')
    '''
    con.sql(insert_sql)
    print(f"表 {table_name} 创建并写入数据完成")
    table_name = "category_sales"
    table_schema = ibis.schema({"category": "string", "amount": "float64"})
    con.sql(f'DROP TABLE IF EXISTS "{target_database}"."{table_name}";')
    con.create_table(
        table_name,
        schema=table_schema,
        database=target_database,
        table_format="orc",
        if_not_exists=True,
    )
    insert_sql = f'''
    INSERT INTO "{target_database}"."{table_name}" (category, amount) VALUES
        ('Electronics', 1299.00),
        ('Electronics', 799.00),
        ('Electronics', 599.00),
        ('Electronics', 1099.00),
        ('Books', 29.99),
        ('Books', 49.99),
        ('Books', 19.99),
        ('Books', 39.99),
        ('Clothing', 89.00),
        ('Clothing', 159.00),
        ('Clothing', 49.00),
        ('Clothing', 199.00)
    '''
    con.sql(insert_sql)
    print(f"表 {table_name} 创建并写入数据完成")
finally:
    con.close()
```

相关文档

- 2.1.5.6 Session APIs
- 2.1.5.7 Dataset APIs
- 2.1.5.8 Table APIs

2.1.5.4.4 建立连接并查询指定表数据

建立与AuraJobV2端点的连接，查询已创建的数据表，返回查询结果。

版权所有 © 华为云计算技术有限公司

前提条件

- 已创建表并写入数据。具体操作，请参见**2.1.5.4.3 创建表并写入数据**。
- 运行时环境要求Python版本为3.11。

示例

本示例建立与Aura服务的连接，查询text_samples表中的文本数据并打印结果。

1. 导入依赖并连接AuraJobV2端点。
2. 加载text_samples数据集。
3. 选择id和content列。
4. 执行查询并打印结果。
5. 关闭AuraJobV2端点连接。

```
import os
import logging
import ibis
from aura_frame.multimodal import ai_lake

target_database = "test"  #在LakeFormation中创建数据库，例如：test。具体操作，请参见创建LakeFormation实例并规划元数据
#连接AuraJobV2端点，具体参数说明，请参见2.1.5.4.1 建立与AuraJobV2端点的连接。
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),        # AuraJobV2端点所在的终端节点（Endpoint）
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2端点ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2端点名称
    aura_workspace_id=os.getenv("aura_workspace_id"),  # 工作空间ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog名称
    access_key=os.getenv("access_key"),               # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),               # 访问密钥（SK）
    default_database=target_database,                 # LakeFormation数据库名称
    use_single_cn_mode=True,   # 是否开启单CN模式
    logging_level=logging.INFO # 设置日志级别
)
try:
    ds = con.load_dataset("text_samples", database=target_database)
    ds = ds.select_columns([ds.id, ds.content])
    df = ds.execute()
    print(df)
finally:
    con.close()
```

相关文档

- **2.1.5.6 Session APIs**
- **2.1.5.7 Dataset APIs**
- **2.1.5.8 Table APIs**

2.1.5.4.5 数据加工与分析

演示筛选、计算列、分组聚合、排序的完整Aura DataFrame处理流程。

场景描述

在销售数据分析场景中，需要统计截止某日期前的订单明细，按退货标志和订单状态分组计算汇总指标，包括总数量、基础价格总额、折扣后价格总额、含税价格总额，以及各指标的平均值和订单数。通过Aura DataFrame的链式操作，可以依次完成筛

版权所有 © 华为云计算技术有限公司

选、派生列计算、分组聚合和排序，实现与SQL等价的数据加工流程，为销售报表和经营决策提供数据支撑。

前提条件

- 已创建表并写入数据。具体操作，请参见**2.1.5.4.3 创建表并写入数据**。
- 运行时环境要求Python版本为3.11。

示例

本示例对sales_line_items表执行筛选、派生列计算、分组聚合和排序等Aura DataFrame操作，展示完整的数据加工与分析流程。

1. 导入依赖并连接AuraJobV2端点。
2. 加载sales_line_items数据集。
3. 筛选ship_date在截止日期之前的记录。
4. 计算折扣价和含税价格等派生列。
5. 按return_flag和line_status分组聚合（sum、avg、count等）。
6. 按return_flag和line_status排序。
7. 执行查询并打印结果。
8. 关闭AuraJobV2端点连接。

```
import os
from datetime import datetime, timedelta
from aura_frame.multimodal import ai_lake

target_database = "test"  #在LakeFormation中创建数据库，例如：test。具体操作，请参见创建LakeFormation实例并规划元数据
#连接AuraJobV2端点，具体参数说明，请参见2.1.5.4.1 建立与AuraJobV2端点的连接。
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),        # AuraJobV2端点所在的终端节点（Endpoint）
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2端点ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2端点名称
    aura_workspace_id=os.getenv("aura_workspace_id"),  # 工作空间ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog名称
    access_key=os.getenv("access_key"),                # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),                # 访问密钥（SK）
    default_database=target_database,                  # LakeFormation数据库名称
    use_single_cn_mode=True,   # 是否开启单CN模式
    logging_level=logging.INFO # 设置日志级别
)
try:
    # 加载数据集
    ds = con.load_dataset("sales_line_items", database=target_database)
    cutoff = datetime(1998, 12, 1) - timedelta(days=90)
    # 筛选：过滤不满足条件的行
    q = ds.filter(expr=ds.ship_date <= cutoff)
    # 计算列表达式
    discount_price = ds.unit_price * (1 - ds.discount_rate)
    charge = discount_price * (1 + ds.tax_rate)
    # 分组聚合
    q = q.groupby(["return_flag", "line_status"])
    q = q.aggregate(
        sum_qty=ds.quantity.sum(),
        sum_base_price=ds.unit_price.sum(),
        sum_disc_price=discount_price.sum(),
        sum_charge=charge.sum(),
        avg_qty=ds.quantity.mean(),
        avg_price=ds.unit_price.mean(),
        avg_disc=ds.discount_rate.mean(),
        count_order=ds.count(),
```


版权所有 © 华为云计算技术有限公司

```
    )
    # 排序
    q = q.order_by(["return_flag", "line_status"])
    # 触发执行
    df = q.execute()
    print(df)
finally:
    con.close()
```

相关文档

- 2.1.5.6 Session APIs
- 2.1.5.7 Dataset APIs
- 2.1.5.8 Table APIs

2.1.5.4.6 使用 Scalar UDF 处理数据

将自定义计算逻辑以Scalar UDF形式注册到数据库，并在Aura DataFrame中调用，实现行级数据处理。

场景描述

在销售数据分析场景中，需要根据商品原价、折扣率和税率计算含税最终价格。该计算逻辑涉及多个字段的组合运算，使用内置SQL函数表达不够直观，且难以复用。通过将计算逻辑封装为Scalar UDF并注册到数据库，可以在DataFrame中像调用内置函数一样使用，代码更清晰，逻辑更易维护。

约束与限制

- 仅支持Python语言编写的用户自定义函数。
- Python环境依赖管理由用户自行负责，需确保各依赖库版本兼容。
- 运行时环境要求Python版本为3.11。
- Scalar UDF的输入参数和返回值必须显式指定数据类型。
- 注册UDF前建议先调用delete_function清理同名函数，避免重复执行时报错。

前提条件

已创建表并写入数据。具体操作，请参见**2.1.5.4.3 创建表并写入数据**。

示例

本示例定义一个计算含税最终价格的Scalar UDF，将其注册到数据库后，在Aura DataFrame中通过map方法对sales_line_items表的每行数据执行行级计算。

1. 定义Scalar UDF函数calc_final_price，根据单价、折扣率和税率计算含税最终价格。
2. 连接AuraJobV2端点并设置UDF工作空间。
3. 删除已存在的同名UDF（支持重复执行）。
4. 注册Scalar UDF到目标数据库。
5. 加载sales_line_items数据集。
6. 通过map方法调用UDF对每行数据计算final_price列。

版权所有 © 华为云计算技术有限公司

7. 选择相关列并执行查询打印结果。
8. 关闭AuraJobV2端点连接。

```
import os
from aura_frame.multimodal import ai_lake

def calc_final_price(price: float, discount: float, tax: float) -> float:
    return price * (1 - discount) * (1 + tax)

target_database = "test"  #在LakeFormation中创建数据库，例如：test。具体操作，请参见创建LakeFormation实例并规划元数据
#连接AuraJobV2端点，具体参数说明，请参见2.1.5.4.1 建立与AuraJobV2端点的连接。
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),         # AuraJobV2端点所在的终端节点（Endpoint）
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2端点ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2端点名称
    aura_workspace_id=os.getenv("aura_workspace_id"),  # 工作空间ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog名称
    access_key=os.getenv("access_key"),                # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),                # 访问密钥（SK）
    default_database=target_database,                  # LakeFormation数据库名称
    use_single_cn_mode=True,   # 是否开启单CN模式
    logging_level=logging.INFO # 设置日志级别
)
#配置UDF部署工作区，具体参数说明，请参见2.1.5.4.2 配置UDF部署工作区。
con.set_function_staging_workspace(
    obs_directory_base=os.getenv("obs_directory_base"), # 并行文件系统中UDF的存储路径
    obs_bucket_name=os.getenv("obs_bucket_name"),       # 并行文件系统名称
    obs_server=os.getenv("obs_server"),                 # OBS终端节点（Endpoint）
    access_key=os.getenv("access_key"),                 # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),                 # 访问密钥（SK）
)
try:
    # 删除已存在的同名UDF（支持重复执行）
    con.delete_function("calc_final_price", database=target_database, if_it_exists=True)
    # 注册Scalar UDF
    udf = con.create_scalar_function(
        calc_final_price,
        database=target_database,
    )
    # 在DataFrame中使用UDF
    ds = con.load_dataset("sales_line_items", database=target_database)
    ds = ds.map(fn=udf, on=[ds.unit_price, ds.discount_rate, ds.tax_rate], as_col="final_price")
    ds = ds.select_columns(ds.unit_price, ds.discount_rate, ds.tax_rate, ds.final_price)
    df = ds.execute()
    print(df)
finally:
    con.close()
```

相关文档

- 2.1.5.6 Session APIs
- 2.1.5.7 Dataset APIs
- 2.1.5.8 Table APIs
- 2.1.5.10 User-Defined Function APIs

2.1.5.4.7 使用 Class UDF 处理数据

Class UDF基于面向对象编程范式实现，适合需要初始化参数、加载资源或有状态的处理逻辑。通过__init__方法初始化资源（如加载模型），__call__方法对每行数据执行计算，可按需通过__del__方法清理资源。

版权所有 © 华为云计算技术有限公司

场景描述

在文本处理场景中，需要使用SentencePiece分词模型对text_samples表中的文本列content进行子词切分（subword tokenization）。模型文件需要在UDF初始化时一次性加载，之后对每行文本重复使用。通过Class UDF，可以在__init__中加载spiece.model模型文件并配置EOS编码选项，在__call__中对每行文本执行分词并返回JSON格式的分词结果，避免逐行重复加载模型的开销。

约束与限制

- 类必须定义__init__方法，且其参数名称需与调用时的参数名称保持一致。
- 类必须定义 __call__ 成员方法，该方法为UDF的主函数入口。
- 类可按需定义 __del__ 成员方法，且该方法不支持传入参数。
- 调用Class UDF时，__init__ 方法的参数仅支持传入常量，不支持传入表的数据列或其他表达式。

前提条件

- 已创建表并写入数据。具体操作，请参见**2.1.5.4.3 创建表并写入数据**。
- 运行时环境要求Python版本为3.11。

准备工作

本示例使用SentencePiece分词模型，需要准备模型文件spiece.model。推荐从Hugging Face获取T5模型自带的SentencePiece分词器，获取方式如下：

1. 安装SentencePiece依赖：
   ```
   pip install sentencepiece==0.2.0
   ```
2. 安装huggingface命令行工具：
   ```
   pip install huggingface-hub
   ```
3. 下载模型文件到示例脚本同级目录：
   ```
   hf download google-t5/t5-small spiece.model --local-dir .
   ```

如果hf download命令无法使用，可通过浏览器手动下载：

- 在Hugging Face网站搜索google-t5/t5-small，进入模型主页。
- 单击“Files and versions”标签页。
- 找到spiece.model文件，单击文件名进入详情页。
- 单击文件内容右上方的下载按钮，将文件保存到本地。
- 将下载的spiece.model文件放置在与示例脚本同级目录下。

示例

本示例定义一个基于SentencePiece分词模型的Class UDF，在__init__中加载模型文件，在__call__中对每行文本执行子词切分，将其注册到数据库后通过map方法处理text_samples表的文本列。

1. 定义Class UDF类SPManager，在__init__中加载SentencePiece模型文件，在__call__中对文本执行分词并返回JSON格式结果。
2. 连接AuraJobV2端点并设置UDF工作空间。
3. 删除已存在的同名UDF（支持重复执行）。

版权所有 © 华为云计算技术有限公司

4. 注册Class UDF到目标数据库，指定依赖包和模型文件。
5. 获取已注册的UDF算子并加载text_samples数据集。
6. 通过map方法调用UDF对content列执行分词，生成pieces_column列。
7. 选择content和pieces_column列并执行查询打印结果。
8. 关闭AuraJobV2端点连接。

```
import os
from aura_frame.multimodal import ai_lake
from sentencepiece import SentencePieceProcessor

target_database = "test"  #在LakeFormation中创建数据库，例如：test。具体操作，请参见创建LakeFormation实例并规划元数据
#连接AuraJobV2端点，具体参数说明，请参见2.1.5.4.1 建立与AuraJobV2端点的连接。
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),        # AuraJobV2端点所在的终端节点（Endpoint）
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2端点ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2端点名称
    aura_workspace_id=os.getenv("aura_workspace_id"),  # 工作空间ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog名称
    access_key=os.getenv("access_key"),               # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),               # 访问密钥（SK）
    default_database=target_database,                 # LakeFormation数据库名称
    use_single_cn_mode=True,   # 是否开启单CN模式
    logging_level=logging.INFO # 设置日志级别
)
#配置UDF部署工作区，具体参数说明，请参见2.1.5.4.2 配置UDF部署工作区。
con.set_function_staging_workspace(
    obs_directory_base=os.getenv("obs_directory_base"),  # 并行文件系统中UDF的存储路径
    obs_bucket_name=os.getenv("obs_bucket_name"),      # 并行文件系统名称
    obs_server=os.getenv("obs_server"),                # OBS终端节点（Endpoint）
    access_key=os.getenv("access_key"),                # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),                # 访问密钥（SK）
)

class SPManager:
    def __init__(self, model_file: str, *, bos: bool = False, eos: bool = False, reverse: bool = False) -> None:
        sp = SentencePieceProcessor(model_file=model_file)
        opts = ":".join(opt for opt, flag in (("bos", bos), ("eos", eos), ("reverse", reverse)) if flag)
        sp.set_encode_extra_options(opts)
        self.spm = sp
    def __call__(self, input_text: str) -> str:
        import json
        return json.dumps(self.spm.encode(input_text, out_type=str))

try:
    # 删除已存在的同名UDF（支持重复执行）
    con.delete_function("SPManager", database=target_database, if_it_exists=True)
    # 注册Class UDF
    con.create_scalar_function(
        SPManager,
        database=target_database,
        packages=["sentencepiece==0.2.0"],
        imports=["spiece.model"],
    )
    # 获取UDF算子并在DataFrame中使用
    udf = con.get_function("SPManager", database=target_database)
    ds = con.load_dataset("text_samples", database=target_database)
    ds = ds.map(
        fn=udf, on=[ds.content], as_col="pieces_column",
        model_file="spiece.model", eos=True,
    )
    ds = ds.select_columns(ds.content, ds.pieces_column)
    df = ds.execute()
    print(df)
finally:
    con.close()
```


版权所有 © 华为云计算技术有限公司

相关文档

- 2.1.5.6 Session APIs
- 2.1.5.7 Dataset APIs
- 2.1.5.8 Table APIs
- 2.1.5.10 User-Defined Function APIs

2.1.5.4.8 使用聚合统计的 UDAF

UDAF（用户自定义聚合函数）接收多行数据返回一组聚合值，适用于内置聚合函数无法满足的自定义统计场景。

场景描述

在销售数据分析中，需要按商品类别（category）统计每组中金额（amount）最高的前K个值。内置的MAX函数只能返回最大值，无法返回Top-K列表。通过自定义UDAF PythonTopK，在 accumulate 中收集每行的金额值并保持降序排序，在 merge 中合并各分区的中间状态，在 finish 中输出JSON格式的Top-K列表，实现灵活的自定义聚合统计。

约束与限制

- UDAF必须实现aggregate_state、accumulate、merge、finish 4个方法。
- UDAF要求用户实现的accumulate、aggregate_state、merge方法是无副作用的（side-effect free），即这些方法的返回结果只取决于输入参数和聚合状态本身，而不应依赖于方法被调用的次数、顺序或外部环境。
- UDAF不支持作为窗口函数使用，即不支持在OVER从句中使用。
- UDAF不支持开启并发和向量化计算。
- 注册UDAF前建议先调用delete_function清理同名函数，避免重复执行时报错。

前提条件

- 已创建表并写入数据。具体操作，请参见2.1.5.4.3 创建表并写入数据。
- 运行时环境要求Python版本为3.11。

示例

本示例定义一个统计每组中金额最高的前K个值的UDAF，将其注册到数据库后，在category_sales表上按商品类别分组聚合，取每组金额前3的记录。

1. 定义UDAF类PythonTopK，实现aggregate_state、accumulate、merge、finish四个方法，用于统计每组中amount值最大的前K个记录。
2. 连接AuraJobV2端点并设置UDF工作空间。
3. 删除已存在的同名UDF（支持重复执行）。
4. 注册UDAF到目标数据库。
5. 加载category_sales数据集。
6. 通过AggregateFnBuilder构建聚合表达式，按category分组取amount前3。
7. 执行聚合查询并打印结果。
8. 关闭AuraJobV2端点连接。

版权所有 © 华为云计算技术有限公司

```
import os
import json
from aura_frame.multimodal import ai_lake
from aura_frame.multimodal.function import AggregateFnBuilder

target_database = "test"  #在LakeFormation中创建数据库，例如：test。具体操作，请参见创建LakeFormation实例并规划元数据
#连接AuraJobV2端点，具体参数说明，请参见2.1.5.4.1 建立与AuraJobV2端点的连接。
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),         # AuraJobV2端点所在的终端节点（Endpoint）
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2端点ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2端点名称
    aura_workspace_id=os.getenv("aura_workspace_id"),  # 工作空间ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog名称
    access_key=os.getenv("access_key"),                # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),                # 访问密钥（SK）
    default_database=target_database,                  # LakeFormation数据库名称
    use_single_cn_mode=True,  # 是否开启单CN模式
    logging_level=logging.INFO # 设置日志级别
)
#配置UDF部署工作区，具体参数说明，请参见2.1.5.4.2 配置UDF部署工作区。
con.set_function_staging_workspace(
    obs_directory_base=os.getenv("obs_directory_base"),  # 并行文件系统中UDF的存储路径
    obs_bucket_name=os.getenv("obs_bucket_name"),        # 并行文件系统名称
    obs_server=os.getenv("obs_server"),                  # OBS终端节点（Endpoint）
    access_key=os.getenv("access_key"),                  # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),                  # 访问密钥（SK）
)

# 定义UDAF：统计每组金额最高的前K个值
class PythonTopK:
    def __init__(self, k: int):
        self.k = k
        self._agg_state = []
    @property
    def aggregate_state(self):
        return self._agg_state
    def accumulate(self, value: float) -> None:
        if value is not None:
            self._agg_state.append(value)
            self._agg_state.sort(reverse=True)
            self._agg_state = self._agg_state[: self.k]
    def merge(self, other_state) -> None:
        merged = self._agg_state + other_state
        merged.sort(reverse=True)
        self._agg_state = merged[: self.k]
    def finish(self) -> str:
        return json.dumps(self._agg_state)
try:
    # 删除已存在的同名UDF（支持重复执行）
    con.delete_function("PythonTopK", database=target_database, if_it_exists=True)
    # 注册UDAF
    udaf = con.create_agg_function(PythonTopK, database=target_database)
    # 在DataFrame中使用UDAF：按category分组，取每组amount最高的前3个
    ds = con.load_dataset("category_sales", database=target_database)
    udaf_builder = AggregateFnBuilder(fn=udaf, on=[ds.amount], as_col="top3", constructor_kwargs={"k": 3})
    summary = ds.aggregate([udaf_builder], by=[ds.category])
    df = summary.execute()
    print(df)
finally:
    con.close()
```

相关文档

- 2.1.5.6 Session APIs
- 2.1.5.7 Dataset APIs
- 2.1.5.8 Table APIs

版权所有 © 华为云计算技术有限公司

- 2.1.5.10 User-Defined Function APIs

2.1.5.4.9 直接使用已注册的 Scalar UDF

当UDF已由其他用户或脚本注册到数据库后，可以直接通过 get_function 获取已注册的UDF算子并在DataFrame中调用，无需重复注册。

场景描述

在团队协作开发中，UDF的注册和使用往往是分离的：一部分人负责编写和注册UDF，另一部分人负责在数据处理流程中调用已注册的UDF。本示例展示如何查看数据库中已注册的UDF列表，获取指定UDF算子，并在DataFrame表达式中直接使用。

约束与限制

- 使用前需确保目标UDF已注册到指定数据库中，函数名在数据库中存储为全小写。
- Scalar UDF和Class UDF均通过map方法调用，on参数传入__call__的输入列。
- Class UDF的__init__参数通过map方法的关键字参数传入（如model_file、eos）。

前提条件

- 已创建表并写入数据。具体操作，请参见2.1.5.4.3 创建表并写入数据。
- 运行时环境要求Python版本为3.11。

示例

本示例展示如何查看数据库中已注册的UDF列表，获取已注册的Scalar UDF和Class UDF算子，并在Aura DataFrame中直接调用，无需重复注册。

1. 连接Aura服务并设置UDF工作空间。
2. 查看目标数据库中已注册的UDF名称列表。
3. 若calc_final_price存在，获取该Scalar UDF算子并加载sales_line_items数据集，通过map方法计算final_price列。
4. 若spmanager存在，获取该Class UDF算子并加载text_samples数据集，通过map方法对content列执行分词（传入model_file和eos参数）。
5. 关闭Aura连接。

```
import os
from aura_frame.multimodal import ai_lake

target_database = "test"  #在LakeFormation中创建数据库，例如：test。具体操作，请参见创建LakeFormation实例并规划元数据
#连接AuraJobV2端点，具体参数说明，请参见2.1.5.4.1 建立与AuraJobV2端点的连接。
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),        # AuraJobV2端点所在的终端节点（Endpoint）
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2端点ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2端点名称
    aura_workspace_id=os.getenv("aura_workspace_id"),  # 工作空间ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog名称
    access_key=os.getenv("access_key"),              # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),              # 访问密钥（SK）
    default_database=target_database,                  # LakeFormation数据库名称
    use_single_cn_mode=True,   # 是否开启单CN模式
    logging_level=logging.INFO # 设置日志级别
```


版权所有 © 华为云计算技术有限公司

```
)
#配置UDF部署工作区，具体参数说明，请参见2.1.5.4.2 配置UDF部署工作区。
con.set_function_staging_workspace(
    obs_directory_base=os.getenv("obs_directory_base"), # 并行文件系统中UDF的存储路径
    obs_bucket_name=os.getenv("obs_bucket_name"),       # 并行文件系统名称
    obs_server=os.getenv("obs_server"),                 # OBS终端节点（Endpoint）
    access_key=os.getenv("access_key"),                 # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),                 # 访问密钥（SK）
)
try:
    # 查看已注册的所有UDF名称
    udf_names = con._conn.udf.names(database=target_database)
    print(f"已注册的UDF: {udf_names}")
    if "calc_final_price" in udf_names:
        # 获取已注册的Scalar UDF
        calc_final_price_udf = con.get_function("calc_final_price", database=target_database)
        ds = con.load_dataset("sales_line_items", database=target_database)
        # 在DataFrame中直接使用已注册的UDF
        ds = ds.map(
            fn=calc_final_price_udf,
            on=[ds.unit_price, ds.discount_rate, ds.tax_rate],
            as_col="final_price",
        )
        ds = ds.select_columns(ds.unit_price, ds.discount_rate, ds.tax_rate, ds.final_price)
        df = ds.execute()
        print(df)
    if "spmanager" in udf_names:
        # 获取已注册的Class UDF
        sp_udf = con.get_function("spmanager", database=target_database)
        ds = con.load_dataset("text_samples", database=target_database)
        # 通过map方法对content列执行分词
        ds = ds.map(
            fn=sp_udf, on=[ds.content], as_col="pieces_column",
            model_file="spiece.model", eos=True,
        )
        ds = ds.select_columns(ds.content, ds.pieces_column)
        df = ds.execute()
        print(df)
finally:
    con.close()
```

相关文档

- 2.1.5.6 Session APIs
- 2.1.5.7 Dataset APIs
- 2.1.5.8 Table APIs
- 2.1.5.10 User-Defined Function APIs

2.1.5.4.10 直接使用已注册的 UDAF

当UDAF已由其他用户或脚本注册到数据库后，可以通过udaf.get获取已注册的UDAF算子，并在aggregate中直接调用，无需重复注册。

场景描述

在团队协作中，UDAF的注册和使用往往是分离的。本示例展示如何查看数据库中已注册的UDF列表，获取已注册的UDAF算子 pythontopk，并在 category_sales 表上按 category 分组聚合，取每组 amount 最高的前3个值。

约束与限制

- 使用前需确保目标UDAF已注册到指定数据库中，函数名在数据库中存储为全小写。

版权所有 © 华为云计算技术有限公司

- UDAF通过 AggregateFnBuilder 构建聚合表达式，__init__参数通过 constructor_kwargs 传入。

前提条件

- 已创建表并写入数据。具体操作，请参见**2.1.5.4.3 创建表并写入数据**。
- 运行时环境要求Python版本为3.11。

示例

本示例展示如何查看数据库中已注册的UDF列表，获取已注册的UDAF算子pythontopk，并在category_sales表上按商品类别分组聚合，取每组金额前3的记录。

1. 连接Aura服务并设置UDF工作空间。
2. 查看目标数据库中已注册的UDF名称列表。
3. 若pythontopk存在，获取该UDAF算子。
4. 加载category_sales数据集。
5. 通过AggregateFnBuilder构建聚合表达式，按category分组取amount前3。
6. 执行聚合查询并打印结果。
7. 关闭Aura连接。

```
import os
from aura_frame.multimodal import ai_lake
from aura_frame.multimodal.function import AggregateFnBuilder

target_database = "test"  #在LakeFormation中创建数据库，例如：test。具体操作，请参见创建LakeFormation实例并规划元数据
#连接AuraJobV2端点，具体参数说明，请参见2.1.5.4.1 建立与AuraJobV2端点的连接。
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),        # AuraJobV2端点所在的终端节点（Endpoint）
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2端点ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2端点名称
    aura_workspace_id=os.getenv("aura_workspace_id"),  # 工作空间ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog名称
    access_key=os.getenv("access_key"),                # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),                # 访问密钥（SK）
    default_database=target_database,                  # LakeFormation数据库名称
    use_single_cn_mode=True,   # 是否开启单CN模式
    logging_level=logging.INFO # 设置日志级别
)
#配置UDF部署工作区，具体参数说明，请参见2.1.5.4.2 配置UDF部署工作区。
con.set_function_staging_workspace(
    obs_directory_base=os.getenv("obs_directory_base"),  # 并行文件系统中UDF的存储路径
    obs_bucket_name=os.getenv("obs_bucket_name"),        # 并行文件系统名称
    obs_server=os.getenv("obs_server"),                  # OBS终端节点（Endpoint）
    access_key=os.getenv("access_key"),                  # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),                  # 访问密钥（SK）
)
try:
    # 查看已注册的所有UDF名称
    udf_names = con._conn.udaf.names(database=target_database)
    print(f"已注册的UDF: {udf_names}")
    if "pythontopk" in udf_names:
        # 获取已注册的UDAF算子
        topk_udaf = con._conn.udaf.get(name="pythontopk", database=target_database)
        ds = con.load_dataset("category_sales", database=target_database)
        # 在aggregate中使用，通过constructor_kwargs传入__init__参数
        udaf_builder = AggregateFnBuilder(
            fn=topk_udaf, on=[ds.amount], as_col="top3",
            constructor_kwargs={"k": 3},
        )
        summary = ds.aggregate([udaf_builder], by=[ds.category])
```


版权所有 © 华为云计算技术有限公司

```
        df = summary.execute()
        print(df)
finally:
    con.close()
```

相关文档

- 2.1.5.6 Session APIs
- 2.1.5.7 Dataset APIs
- 2.1.5.8 Table APIs
- 2.1.5.10 User-Defined Function APIs

2.1.5.4.11 语音识别示例

本示例展示如何定义Scalar UDF进行语音识别，将音频二进制数据通过Whisper模型转换为文本，体现多模态数据处理中UDF的使用能力。

场景描述

在音频数据处理场景中，需要将存储在数据库表中的音频二进制数据通过语音识别模型（Whisper）转换为文本。通过Class形式的Scalar UDF，在__init__中加载Whisper模型，在__call__中对每段音频执行识别并返回文本结果。本示例从sample-files.com下载一段真实语音音频作为测试数据，写入数据库表后供UDF处理。

约束与限制

- UDF运行时需通过pip安装transformers包，请确保运行时环境网络可用。
- Scalar UDF的__init__参数仅支持传入常量。
- 注册UDF前建议先调用delete_function清理同名函数，避免重复执行时报错。

前提条件

- 已安装Aura DataFrame SDK。具体操作，请参见2.1.5.2 Aura DataFrame SDK获取与安装。
- 已创建AuraJobV2端点，并获取端点名称。具体操作，请参见创建Aura引擎端点。
- 已创建LakeFormation实例，并在该实例中创建Catalog及数据库。具体操作，请参见创建LakeFormation实例并规划元数据。
- 已创建并行文件系统。具体操作，请参见创建并行文件系统。
- 代码中涉及os.getenv()中的参数已配置到环境变量。
- 运行时环境要求Python版本为3.11。

准备工作

本示例使用Whisper语音识别模型，需要准备模型文件、音频文件和Python依赖。

1. 安装Python依赖。

```
pip install torch==2.6.0
pip install transformers==5.3.0
pip install av
pip install scipy
```

2. 安装huggingface命令行工具。

版权所有 © 华为云计算技术有限公司

```
pip install huggingface-hub
```

3. 下载测试音频文件到示例脚本同级目录，重命名为test_audio.wav：
   下载一段WAV格式的语音文件，建议10秒以内。
4. 下载Whisper base模型文件到示例脚本同级目录。
   ```
   hf download openai/whisper-base --local-dir ./whisper_base
   ```
   如果hf download命令无法使用，可通过浏览器手动下载：
   a. 在Hugging Face网站搜索openai/whisper-base，进入模型主页。
   b. 单击“Files and versions”标签页。
   c. 下载以下文件，保存到本地whisper_base目录下：
      - config.json
      - pytorch_model.bin
      - preprocessor_config.json
      - tokenizer_config.json
      - tokenizer.json
      - vocab.json
      - merges.txt
      - normalizer.json
      - special_tokens_map.json
      - added_tokens.json
      - generation_config.json
   d. 将whisper_base目录放置在与示例脚本同级目录下。

示例

本示例定义一个基于Whisper模型的语音识别Scalar UDF，将测试音频写入数据库后，通过map_batches方法调用UDF将音频二进制数据转换为文本。

1. 定义Class UDF类SpeechRecognitionUDF，在__init__中通过transformers加载Whisper语音识别模型，在__call__中用av库解码音频二进制数据并重采样到16kHz后执行识别。
2. 连接Aura服务并设置UDF工作空间。
3. 读取测试音频文件，建表并插入音频二进制数据。
4. 删除已存在的同名UDF（支持重复执行）。
5. 注册Scalar UDF到目标数据库，指定依赖包和模型文件。
6. 加载audio_data数据集。
7. 通过map_batches方法调用UDF进行语音识别，生成hyp_text列。
8. 选择相关列并执行查询打印结果。
9. 关闭Aura连接。

```
import os
import ibis
```


版权所有 © 华为云计算技术有限公司

```
import aura_frame
from aura_frame.multimodal import ai_lake



# 定义Scalar UDF：语音识别
class SpeechRecognitionUDF:
    def __init__(self, model_path: str) -> None:
        import torch
        from transformers import pipeline
        self._pipeline = pipeline(
            "automatic-speech-recognition",
            model=model_path,
            device=0 if torch.cuda.is_available() else -1,
        )
    def __call__(self, audios: bytes) -> str:
        import numpy as np
        import av
        import io
        from scipy.signal import resample_poly
        container = av.open(io.BytesIO(audios))
        stream = container.streams.audio[0]
        frames = []
        for frame in container.decode(stream):
            frames.append(frame.to_ndarray())
        if not frames:
            raise ValueError("Audio data is None.")
        audio_np = np.concatenate(frames, axis=1)
        if audio_np.ndim > 1:
            audio_np = audio_np.mean(axis=0)
        if audio_np.dtype == np.int16:
            audio_np = audio_np.astype(np.float32) / 32768.0
        elif audio_np.dtype == np.int32:
            audio_np = audio_np.astype(np.float32) / 2147483648.0
        elif audio_np.dtype != np.float32:
            audio_np = audio_np.astype(np.float32)
        # 重采样到 16kHz（Whisper 要求）
        target_sr = 16000
        if stream.rate != target_sr:
            audio_np = resample_poly(audio_np, target_sr, stream.rate)
        result = self._pipeline({"raw": audio_np, "sampling_rate": target_sr})
        return result["text"]

try:
    # 读取测试音频文件
    audio_path = "test_audio.wav"
    with open(audio_path, "rb") as f:
        audio_bytes = f.read()
    # 建表并插入音频数据
    table_name = "audio_data"
    table_schema = ibis.schema({"id": "int32", "audio_data": "binary"})
    con.sql(f'DROP TABLE IF EXISTS "{target_database}"."{table_name}";')
    con.create_table(
        table_name,
        schema=table_schema,
        database=target_database,
        table_format="orc",
        if_not_exists=True,
    )
    hex_str = audio_bytes.hex()
    con.sql(f'''
    INSERT INTO "{target_database}"."{table_name}" (id, audio_data) VALUES
        (1, '\\x{hex_str}'::bytea)
    ''')
    print("音频数据写入完成")
    # 删除已存在的同名UDF（支持重复执行）
    con.delete_function("speechrecognitionudf", database=target_database, if_it_exists=True)
    # 注册Scalar UDF：语音识别
    con.create_scalar_function(
```


版权所有 © 华为云计算技术有限公司

```
        SpeechRecognitionUDF,
        database=target_database,
        packages=["transformers==5.3.0"],
        imports=["./whisper_base"],
    )
    # 加载音频数据集
    ds = con.load_dataset("audio_data", database=target_database)
    # 使用Scalar UDF进行语音识别
    udf = con.get_function("speechrecognitionudf", database=target_database)
    ds = ds.map_batches(
        fn=udf,
        on=[ds.audio_data],
        as_col="hyp_text",
        num_cpus=2,
        memory=4096,
        arch="x86",
        model_path="./whisper_base",
    )
    ds = ds.select_columns(ds.id, ds.audio_data, ds.hyp_text)
    df = ds.execute()
    print(df)
finally:
    con.close()
```

相关文档

- 2.1.5.6 Session APIs
- 2.1.5.7 Dataset APIs
- 2.1.5.8 Table APIs
- 2.1.5.10 User-Defined Function APIs

2.1.5.4.12 视频帧描述与摘要生成示例

本示例展示如何使用Scalar UDF和UDAF对视频帧进行内容描述与摘要生成。通过av库从真实视频文件中提取帧，存入数据库表后，使用BLIP模型生成每帧的自然语言描述，并使用T5模型将多帧描述汇总为视频摘要。

场景描述

在视频内容分析场景中，需要从视频中提取关键帧，对每帧生成自然语言描述，并将多帧描述综合为连贯的视频摘要。通过Class形式的Scalar UDF，在__init__中加载BLIP模型，在__call__中对每帧执行图像描述生成。通过UDAF（用户定义聚合函数），将多帧描述聚合后使用T5模型生成摘要。本示例使用av库从真实视频文件中提取帧作为测试数据。

约束与限制

- UDF运行时需通过pip安装transformers包，请确保运行时环境网络可用。
- Scalar UDF的__init__参数仅支持传入常量。
- 注册UDF前建议先调用delete_function清理同名函数，避免重复执行时报错。

前提条件

- 已安装Aura DataFrame SDK。具体操作，请参见2.1.5.2 Aura DataFrame SDK获取与安装。
- 已创建AuraJobV2端点，并获取端点名称。具体操作，请参见创建Aura引擎端点。

版权所有 © 华为云计算技术有限公司

- 已创建LakeFormation实例，并在该实例中创建Catalog及数据库。具体操作，请参见**创建LakeFormation实例并规划元数据**。
- 已创建并行文件系统。具体操作，请参见**创建并行文件系统**。
- 代码中涉及os.getenv()中的参数已配置到环境变量。
- 运行时环境要求Python版本为3.11。

准备工作

本示例使用BLIP图像描述模型和T5文本摘要模型，需要准备模型文件、视频文件和Python依赖。

1. 安装Python依赖：
```
pip install torch==2.6.0
pip install transformers==5.3.0
pip install pillow
pip install av
```
2. 安装huggingface命令行工具：
```
pip install huggingface-hub
```
3. 下载测试视频文件到示例脚本同级目录，重命名为test_video.mp4：

   下载一段包含真实场景内容的MP4格式短视频，建议10秒以内。
4. 下载BLIP图像描述模型文件到示例脚本同级目录：
```
hf download Salesforce/blip-image-captioning-base --local-dir ./blip-image-captioning-base
```
   如果hf download命令无法使用，可通过浏览器手动下载：
   a. 在Hugging Face网站搜索 Salesforce/blip-image-captioning-base，进入模型主页。
   b. 单击“Files and versions”标签页。
   c. 下载以下文件，保存到本地 blip-image-captioning-base 目录下：
      - config.json
      - preprocessor_config.json
      - pytorch_model.bin
      - special_tokens_map.json
      - tokenizer_config.json
      - tokenizer.json
      - vocab.txt
   d. 将blip-image-captioning-base目录放置在与示例脚本同级目录下。
5. 下载T5文本摘要模型文件到示例脚本同级目录：
```
hf download google-t5/t5-small --local-dir ./t5-small
```
   如果hf download命令无法使用，可通过浏览器手动下载：
   a. 在Hugging Face网站搜索google-t5/t5-small，进入模型主页。
   b. 单击“Files and versions”标签页。
   c. 下载以下文件，保存到本地t5-small目录下：
      - config.json

版权所有 © 华为云计算技术有限公司

- pytorch_model.bin
- spiece.model
- tokenizer.json
- tokenizer_config.json
- generation_config.json

d. 将t5-small目录放置在与示例脚本同级目录下。

示例

本示例将视频帧描述与摘要生成的完整流程整合为一段可独立运行的代码，包含连接管理、Scalar UDF定义（基于BLIP模型生成帧描述）、UDAF定义（基于T5模型生成视频摘要）、函数注册、视频帧提取、帧级描述生成和视频摘要聚合的全过程。

1. 导入依赖并定义create_connect上下文管理器，封装Aura服务连接的创建与释放。
2. 定义Scalar UDF类VideoContentAnalyzer，在__init__中加载BLIP模型，在__call__中对每帧JPEG数据生成描述并分析内容特征，返回JSON格式结果。
3. 定义VideoSummaryGenerator类，加载T5模型，将多帧描述聚合后生成视频摘要。
4. 定义UDAF类VideoSummarySystem，通过accumulate收集每帧描述，finish中调用T5模型生成视频摘要。
5. 连接AuraJobV2端点并设置UDF工作空间，删除已存在的同名UDF和UDAF后注册到目标数据库。
6. 用av库从视频文件中提取最多10帧（JPEG格式），创建video_frames表写入二进制帧数据。
7. 加载video_frames数据集，通过map_batches调用Scalar UDF分析每帧生成descriptions列。
8. 通过AggregateFnBuilder调用UDAF聚合多帧描述生成视频摘要，执行查询并打印结果。

```
import ibis
import typing
import os
from contextlib import contextmanager
from aura_frame.multimodal import ai_lake
from aura_frame.multimodal.function import AggregateFnBuilder

target_database = "test"  #在LakeFormation中创建数据库，例如：test。具体操作，请参见创建LakeFormation实例并规划元数据
@contextmanager
def create_connect():
#连接AuraJobV2端点，具体参数说明，请参见2.1.5.4.1 建立与AuraJobV2端点的连接
    con = ai_lake.connect(
        aura_endpoint=os.getenv("aura_endpoint"),        # AuraJobV2端点所在的终端节点（Endpoint）
        aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2端点ID
        aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2端点名称
        aura_workspace_id=os.getenv("aura_workspace_id"),  # 工作空间ID
        lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog名称
        access_key=os.getenv("access_key"),                # 访问密钥ID（AK）
        secret_key=os.getenv("secret_key"),                # 访问密钥（SK）
        default_database=target_database,                  # LakeFormation数据库名称
        use_single_cn_mode=True,   # 是否开启单CN模式
        logging_level=logging.INFO # 设置日志级别
```


版权所有 © 华为云计算技术有限公司

```
    )
    print(f"Establish ai_lake.connect: {con}")
    try:
        yield con
    finally:
        # 关闭连接
        con.close()

class VideoContentAnalyzer:
    def __init__(self, model_name="./blip-image-captioning-base"):
        import torch
        from transformers import BlipProcessor, BlipForConditionalGeneration
        self.device = "cuda" if torch.cuda.is_available() else "cpu"
        self.processor = BlipProcessor.from_pretrained(model_name)
        self.model = BlipForConditionalGeneration.from_pretrained(
            model_name,
            torch_dtype=torch.float16 if self.device == "cuda" else torch.float32,
        ).to(self.device)
        self.model.eval()
    def __call__(self, frame: bytes, frame_index: int) -> str:
        import json
        analyse = self.analyze_frame(frame, frame_index)
        return json.dumps({"description": analyse})
    def analyze_frame(self, frame: bytes, frame_index: int):
        try:
            frame_description = self._generate_frame_description(frame)
            content_features = self._analyze_content_features(frame_description)
            analyse = {
                "description": frame_description,
                "content_type": content_features["content_type"],
                "key_objects": content_features["key_objects"],
                "scene_context": content_features["scene_context"],
            }
        except Exception:
            analyse = {}
        return analyse
    def _generate_frame_description(self, frame: bytes):
        import io
        from PIL import Image
        import torch
        pil_img = Image.open(io.BytesIO(frame)).convert("RGB")
        inputs = self.processor(images=pil_img, return_tensors="pt").to(self.device)
        with torch.no_grad():
            outputs = self.model.generate(
                **inputs,
                max_length=50,
                num_beams=5,
                early_stopping=True,
            )
        description = self.processor.decode(outputs[0], skip_special_tokens=True)
        return description.strip()
    def _analyze_content_features(self, description):
        description_lower = description.lower()
        content_categories = {
            "educational": ["tutorial", "lesson", "explain", "teach", "learn", "education"],
            "entertainment": ["funny", "comedy", "entertainment", "show", "performance"],
            "nature": ["outdoor", "nature", "landscape", "mountain", "forest", "animal"],
            "sports": ["sport", "game", "player", "team", "match", "competition"],
            "food": ["food", "cooking", "recipe", "meal", "restaurant", "kitchen"],
            "technology": ["computer", "tech", "device", "electronic", "software"],
            "people": ["person", "people", "man", "woman", "child", "group"],
        }
        detected_categories = []
        for category, keywords in content_categories.items():
            if any(keyword in description_lower for keyword in keywords):
                detected_categories.append(category)
        common_objects = [
            "person", "people", "man", "woman", "child", "car", "building",
            "tree", "house", "street", "water", "sky", "food", "animal",
```


版权所有 © 华为云计算技术有限公司

```
        ]
        key_objects = [obj for obj in common_objects if obj in description_lower]
        if any(word in description_lower for word in ["indoors", "inside", "room"]):
            scene_context = "indoor"
        elif any(word in description_lower for word in ["outdoors", "outside", "park", "street"]):
            scene_context = "outdoor"
        else:
            scene_context = "unknown"
        return {
            "content_type": detected_categories[:2] if detected_categories else ["general"],
            "key_objects": key_objects[:5],
            "scene_context": scene_context,
        }

class VideoSummaryGenerator:
    def __init__(self, model):
        from transformers import pipeline
        import torch
        self.summarizer = pipeline(
            "text-generation",
            model=model,
            device=0 if torch.cuda.is_available() else -1,
        )
    def generate_summary(self, frame_analyses):
        all_descriptions = [analysis.get("description", "") for analysis in frame_analyses]
        if not all_descriptions:
            return "Unable to analyze the video content."
        content_analysis = self._analyze_video_content(frame_analyses)
        description_text = self._create_description_text(all_descriptions, content_analysis)
        summary = self._generate_detailed_summary(description_text, content_analysis)
        return summary
    def _analyze_video_content(self, frame_analyses):
        from collections import Counter
        content_types = []
        all_objects = []
        for analysis in frame_analyses:
            content_types.extend(analysis.get("content_type", []))
            all_objects.extend(analysis.get("key_objects", []))
        common_content_types = [item for item, count in Counter(content_types).most_common(2)]
        common_objects = [item for item, count in Counter(all_objects).most_common(5)]
        return {
            "primary_categories": common_content_types,
            "dominant_objects": common_objects,
            "total_frames_analyzed": len(frame_analyses),
        }
    def _create_description_text(self, descriptions, content_analysis):
        from collections import Counter
        desc_counter = Counter(descriptions)
        representative_descs = [desc for desc, count in desc_counter.most_common(3)]
        text = "Video Content Analysis:\n"
        text += f"Primary Categories: {', '.join(content_analysis['primary_categories'])}\n"
        text += f"Key Objects: {', '.join(content_analysis['dominant_objects'][:3])}\n"
        text += "Representative Scenes:\n"
        for i, desc in enumerate(representative_descs, 1):
            text += f"{i}. {desc}\n"
        return text
    def _generate_concise_summary(self, content_analysis):
        import random
        primary_category = content_analysis["primary_categories"][0] if
content_analysis["primary_categories"] else "General content"
        main_objects = ", ".join(content_analysis["dominant_objects"][:2])
        templates = [
            f"This is a {primary_category} video that includes elements like {main_objects}.",
            f"The video presents a {primary_category} scenario, with its focus on {main_objects}.",
            f"The content centers on {primary_category}, highlighting elements such as {main_objects}.",
        ]
        return random.choice(templates)
    def _generate_detailed_summary(self, description_text, content_analysis):
        try:
```


版权所有 © 华为云计算技术有限公司

```
            summary = self.summarizer(
                "summarize: " + description_text,
                max_length=150,
                min_length=50,
                do_sample=False,
            )[0]["generated_text"]
            return summary
        except Exception:
            return self._generate_concise_summary(content_analysis)

class VideoSummarySystem:
    def __init__(self, model="./t5-small"):
        self.summary_generator = VideoSummaryGenerator(model)
        self.frame_descriptions = []
    def accumulate(self, description: str) -> None:
        import json
        data = json.loads(description)
        self.frame_descriptions.append(data.get("description", data))
    def finish(self) -> str:
        import json
        summary = self.summary_generator.generate_summary(self.frame_descriptions)
        result = {
            "video_summary": summary,
            "frames_analyzed": len(self.frame_descriptions),
            "content_categories": list(
                set(cat for analysis in self.frame_descriptions for cat in analysis.get("content_type", []))
            ),
            "key_objects": list(
                set(obj for analysis in self.frame_descriptions for obj in analysis.get("key_objects", []))
            )[:8],
            "processing_details": {
                "total_frames": len(self.frame_descriptions),
                "successful_analyses": len(self.frame_descriptions),
            },
        }
        return json.dumps(result)
    @property
    def aggregate_state(self) -> typing.Dict[str, typing.Any]:
        return {"frame_descriptions": self.frame_descriptions}
    def merge(self, other_state: typing.Dict[str, typing.Any]) -> None:
        self.frame_descriptions += other_state.get("frame_descriptions")

with create_connect() as con:
    con.set_function_staging_workspace(
        obs_directory_base=os.getenv("obs_directory_base"),
        obs_bucket_name=os.getenv("obs_bucket_name"),
        obs_server=os.getenv("obs_server"),
        access_key=os.getenv("access_key"),
        secret_key=os.getenv("secret_key")
    )
    print("finish set_function_staging_workspace")
    con.delete_function("videocontentanalyzer", database=target_database, if_it_exists=True)
    con.create_scalar_function(
        VideoContentAnalyzer,
        database=target_database,
        packages=["transformers==5.3.0"],
        imports=["./blip-image-captioning-base"],
    )
    con.delete_function("videosummarysystem", database=target_database, if_it_exists=True)
    con.create_agg_function(
        VideoSummarySystem,
        database=target_database,
        packages=["transformers==5.3.0"],
        imports=["./t5-small"],
    )
    # 用 av 从真实视频文件提取帧
    import av
    import io
    from PIL import Image
```


版权所有 © 华为云计算技术有限公司

```
video_path = "test_video.mp4"
container = av.open(video_path)
stream = container.streams.video[0]
frames = []
frame_count = 0
max_frames = 10
for frame in container.decode(stream):
    if frame_count >= max_frames:
        break
    img = frame.to_image()
    buf = io.BytesIO()
    img.save(buf, format="JPEG")
    frames.append((frame_count, buf.getvalue()))
    frame_count += 1
container.close()
print(f"从视频中提取了 {len(frames)} 帧")
# 建表并插入帧数据
table_name = "video_frames"
table_schema = ibis.schema({"frame_index": "int32", "frame_data": "binary"})
con.sql(f'DROP TABLE IF EXISTS "{target_database}"."{table_name}";')
con.create_table(
    table_name,
    schema=table_schema,
    database=target_database,
    table_format="orc",
    if_not_exists=True,
)
for frame_index, frame_bytes in frames:
    hex_str = frame_bytes.hex()
    con.sql(f'''
    INSERT INTO "{target_database}"."{table_name}" (frame_index, frame_data) VALUES
        ({frame_index}, '\\x{hex_str}'::bytea)
    ''')
print("视频帧数据写入完成")
# 加载视频帧数据集
ds = con.load_dataset("video_frames", database=target_database)
ds.show(limit=1)
# 使用 Scalar UDF 分析每个帧
udf = con.get_function("videocontentanalyzer", database=target_database)
ds = ds.map_batches(
    fn=udf,
    on=[ds.frame_data, ds.frame_index],
    as_col="descriptions",
    num_cpus=2,
    memory=4096,
    arch="x86",
    model_name="./blip-image-captioning-base",
)
# ds.select_columns([ds.frame_index, ds.descriptions]).show(1)
# 使用UDAF生成视频摘要
udaf = con.get_function("videosummarysystem", database=target_database)
agg_builder = AggregateFnBuilder(
    fn=udaf,
    on=[ds.descriptions],
    as_col="describe",
    num_cpus=1,
    memory=2048,
    arch="x86",
)
summary = ds.aggregate(agg_builder).execute()
for row in summary.itertuples():
    print(row)
```

相关文档

- **2.1.5.6 Session APIs**
- **2.1.5.7 Dataset APIs**

版权所有 © 华为云计算技术有限公司

- 2.1.5.8 Table APIs
- 2.1.5.10 User-Defined Function APIs

2.1.5.5 Aura DataFrame SDK 总览

Aura DataFrame主要包含数据处理、数据管理、输入/输出、辅助工具、触发执行五个部分：

- 数据处理：支持丰富的变换操作，如map()、flat_map()、filter()、join()、groupby()等，可对样本进行特征工程和结构化处理。
- 数据管理：提供对数据集的增删改查能力，包括insert()、update()、delete()以及索引创建与删除，支持数据生命周期管理。
- 输入/输出：具备向主流湖仓格式，如Parquet、Iceberg写入数据的能力，实现与数据湖无缝集成。
- 辅助工具：提供schema()、columns()和explain_plan()等元信息查询与执行计划分析功能，便于调试与优化。
- 触发执行：通过execute()、limit()、take()等方法触发实际计算并获取结果，支持懒加载与按需执行。

表 2-12 接口总览

操作类型	接口	描述
Session APIs	load_dataset	从指定数据库加载数据集。
	set_function_staging_workspace	配置函数部署所需的临时工作区，包括设置必要的凭证和存储信息。
	sql	用于执行SQL查询语句，并返回查询结果。
Dataset APIs	columns	获取Dataset的所有列的列名集合。
	add_column	给一个Dataset添加列。
	drop_columns	删除Dataset的列。
	aggregate	通过指定分组字段，用一组聚合函数对数据集进行聚合。
	count	计算该Dataset的总行数。
	execute	执行表达式。
	filter	过滤掉不满足条件的行。
	flat_map	将给定的函数应用于每一行，并对结果进行展平。
	map_batches	将给定的函数应用于批量数据。
	groupby	创建一个分组的Dataset。
	join	在两个Dataset之间执行join操作。
	map	对当前Dataset的每一行应用给定函数。

版权所有 © 华为云计算技术有限公司

操作类型	接口	描述
	order_by	根据一个或多个表达式对Dataset进行排序。
	rename_columns	重命名Dataset中的列。
	schema	返回Dataset的schema。
	select_columns	从Dataset的列中选取指定的列。
	show	显示查询结果生成的数据。
	take	从Dataset中返回最多limit行数据，该方法适用于数据检查和预览。
	max	计算Dataset中指定列的最大值。
	min	计算Dataset中指定列的最小值。
	mean	计算Dataset中指定列的均值。
	unique	计算Dataset中指定列的唯一值集合。
	limit	从当前Dataset中选择行，起始位置为偏移量。
	take_batch	从Dataset中返回最多batch_size行数据。
	create_async_job	根据作业id恢复作业实例。
	explain_plan	生成作业执行计划。
	explain_performance	执行查询并生成计划，计划中包含执行过程中详细的性能指标。
Table APIs	insert	用于将Dataset插入到数据库表中。
	show	用于显示表的前N行数据。
	to_dataset	将当前表对象转换为Dataset对象。
	add_columns	向表中添加一个或多个新列。
	drop_columns	用于从表中删除一个或多个列。
	alter_column_type	修改表中指定列的数据类型。
	rename_column	重命名表中的列名。
	delete	从表中删除满足指定条件的数据。
	update	更新表中满足指定条件的行数据。
	create_table	在指定数据库中创建一个新表。
	delete_table	从数据库中删除指定的Dataset。

版权所有 © 华为云计算技术有限公司

操作类型	接口	描述
	describe_table	描述指定数据库中的表。
	drop_table	从指定数据库中删除表。
	list_tables	列出指定数据库或默认数据库中的所有表。
	open_table	打开指定数据库中的表。
I/O APIs	read_text	从OBS指定的路径中读取文本文件到Dataset。
	read_json	从OBS指定的路径中读取JSON文件到Dataset。
	write_text	将文本数据写入到指定路径的OBS中。
	write_json	将JSON数据写入到指定路径的OBS中。
	write_iceberg	将数据写入指定的Iceberg表中。
	write_orc	将数据写入指定的ORC表中。
	write_parquet	将数据写入指定的Parquet表中。
User-Defined Function APIs	signature	从后端DB中返回UDF的函数签名和返回值类型。
	get	从后端DB中返回UDF函数。
	names	从后端DB中返回所有的UDF函数名称。
	unregister	从后端DB中删除指定的UDF函数。
	create_agg_function	注册一个用户自定义聚合函数（UDAF）。
	create_scalar_function	注册一个新的标量用户自定义函数（Scalar UDF）。
	create_table_function	注册一个新的用户自定义表值函数（UDTF）。
	delete_function	从指定数据库中删除一个用户自定义函数。
	describe_function	描述指定数据库中的函数。
	get_function	从指定或默认数据库中获取一个用户自定义函数。
	list_functions	列出指定数据库或默认数据库中的所有用户自定义函数。
AI APIs	ai_generate	使用大语言模型生成文本，并将生成结果作为新列添加到Dataset中。model_name需已注册。

版权所有 © 华为云计算技术有限公司

操作类型	接口	描述
	create_model	创建一个抽象本地模型，用于绑定到远程LLM服务。
	list_models	查看指定数据库或默认数据库中的所有模型。
	drop_model	删除与模型名称对应的抽象本地模型。
	describe_model	查看指定的模型的详细信息。

2.1.5.6 Session APIs

load_dataset

描述：从指定数据库加载数据集。

语法格式：

```
load_dataset(name, *, database=None)
```

输入参数：

参数	是否必选	参数类型	描述
name	是	str	要加载的Dataset的名称。
database	否	Optional[str]	要加载Dataset的数据库名称。如果未提供，则使用默认数据库。

返回值类型：Dataset。

set_function_staging_workspace

描述：配置函数部署所需的临时工作区，包括设置必要的凭证和存储信息。

语法格式：

```
set_function_staging_workspace(*, obs_server, obs_bucket_name, obs_directory_base, access_key, secret_key, security_token=None)
```

输入参数：

参数	是否必选	参数类型	描述
obs_server	是	str	OBS服务的地址。
obs_bucket_name	是	str	要使用的OBS桶的名称。

版权所有 © 华为云计算技术有限公司

参数	是否必选	参数类型	描述
obs_directory_base	是	str	OBS中用于存储工作区数据的基目录路径。
access_key	是	str	用于身份验证的访问密钥。
secret_key	是	str	用于身份验证的私密密钥。
security_token	否	Optional[str]	用于身份验证的安全令牌，如果未提供则默认为None。

返回值类型： None。

sql

描述：用于执行SQL查询语句，并返回查询结果。

语法格式：

```
sql(query, **kwargs)
```

输入参数：

参数	是否必选	参数类型	描述
query	是	str	要执行的SQL查询语句。
**kwargs	否	Any	可变关键字参数，用于传递查询的额外参数。

返回值类型： Any，返回SQL查询的结果。

2.1.5.7 Dataset APIs

columns

描述：获取Dataset的所有列的列名集合。

语法格式：

```
columns()
```

输入参数：无

返回值类型：tuple[str, ...]

示例：

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
```


版权所有 © 华为云计算技术有限公司

```
...     access_key='*',
...     secret_key='*'
...
...
... )
>>> ds = con.load_dataset("demo_table_10", database='my_schema')
>>> ds.columns
('species',
  'island',
  'bill_length_mm',
  'bill_depth_mm',  'flipper_length_mm')
```

add_column

描述：给一个Dataset添加列。

语法格式：

```
add_column(*exprs, **mutations)
```

输入参数：

参数	是否必选	参数类型	描述
exprs	否	Union[Sequence[Expr], None]	要添加为列的命名表达式序列。
mutations	否	Value	使用关键字参数的命名表达式。

返回值类型：已添加新列的Dataset。

示例：

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_6", database='my_schema')
>>> ds
| species | bill_length_mm |
| ------- | -------------- |
| string  | float64        |
| ------- | -------------- |
| Adelie  | 40.0           |
| Adelie  | 41.0           |
| Adelie  | 42.0           |
| Adelie  | 43.0           |

#Add a new column from a per-element expression
>>> ds.add_column(new_col=ds.bill_length_mm + 1)
| species | bill_length_mm | new_col  |
| ------- | -------------- | -------- |
| string  | float64        | float64  |
| ------- | -------------- | -------- |
| Adelie  | 40.0           | 41.0     |
| Adelie  | 41.0           | 42.0     |
| Adelie  | 42.0           | 43.0     |
| Adelie  | 43.0           | 44.0     |
```


版权所有 © 华为云计算技术有限公司

drop_columns

描述：删除Dataset的列。

语法格式：

```
drop_columns(*fields)
```

输入参数：

参数	是否必选	参数类型	描述
fields	是	str	被删除列的列名。

返回值类型：删除列后的Dataset。

示例：

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_12", database='my_schema')
>>> ds
| species | island    | bill_length_mm | bill_depth_mm | flipper_length_mm |
| ------- | --------- | -------------- | ------------- | ----------------- |
| string  | string    | float64        | float64       | int64             |
| ------- | --------- | -------------- | ------------- | ----------------- |
| Adelie  | Torgersen | 40.0           | 22.0          | 180               |
| Adelie  | Torgersen | 41.0           | 19.0          | 183               |
| Adelie  | Torgersen | 42.0           | 20.0          | 190               |
| Adelie  | Torgersen | 43.0           | 23.0          | 185               |

# Drop one or more columns
>>> ds.drop_columns("species")
| island    | bill_length_mm | bill_depth_mm | flipper_length_mm |
| --------- | -------------- | ------------- | ----------------- |
| string    | float64        | float64       | int64             |
| --------- | -------------- | ------------- | ----------------- |
| Torgersen | 40.0           | 22.0          | 180               |
| Torgersen | 41.0           | 19.0          | 183               |
| Torgersen | 42.0           | 20.0          | 190               |
| Torgersen | 43.0           | 23.0          | 185               |
```

aggregate

描述：通过指定分组字段，用一组聚合函数对数据集进行聚合。

语法格式：

```
aggregate(metrics=(), /, *, by=(), having=(), **kwargs)
```

输入参数：

版权所有 © 华为云计算技术有限公司

参数	是否必选	参数类型	描述
metrics	否	Union[Sequence[AggregateFnBuilder], None]	聚合表达式，这类表达式可以是任何能生成标量值的表达式，其中包括聚合函数，例如sum。
by	否	Union[Sequence[Value], None]	分组表达式，一般设为Dataset的列名集合。
having	否	Union[Sequence[BooleanValue], None]	聚合后过滤器。其数据维度应该与metrics相同，且having（子句）的输出类型为布尔型。
kwargs	否	-	命名的聚合表达式。

返回值类型：聚合后的Dataset。

示例：

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_13", database='my_schema')
>>> ds
| fruit  | price |
| ------ | ----- |
| apple  | 0.50  |
| apple  | 0.50  |
| banana | 0.25  |
| orange | 0.33  |
>>> ds.aggregate(
...     by=["fruit"],
...     total_cost=ds.price.sum(),
...     avg_cost=ds.price.mean(),
...     having=ds.price.sum() < 0.5,
... ).order_by("fruit")
| fruit  | total_cost | avg_cost |
| ------ | ---------- | -------- |
| string | float64    | float64  |
| ------ | ---------- | -------- |
| banana | 0.25       | 0.25     |
| orange | 0.33       | 0.33     |
```

count

描述：计算该Dataset的总行数。

语法格式：

```
count()
```

输入参数：无

返回值类型：int

版权所有 © 华为云计算技术有限公司

示例：

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_11", database='my_schema')
>>> ds
| a     |
| ----- |
| string|
| ----- |
| foo   |
| bar   |
| baz   |
>>> ds.count()
┌──┐
│ 3│
└──┘
```

execute

描述：执行表达式。

语法格式：

```
execute(*, limit='default', params=None,  block=False, **kwargs)
```

输入参数：

参数	是否必选	参数类型	描述
limit	否	Union[int, str, None]	限定表达式执行结果的总行数。
params	否	Union[Mapping[Value, Any], None]	标量参数表达式到值的映射。
block	否	bool	表示是否异步执行，若为True，就返回AsyncJob对象。
kwargs	否	-	关键字参数

返回值类型：Union[pandas.DataFrame, pandas.Series, AsyncJob ,Any]

示例：

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_8", database='my_schema')
>>> ds.execute()
     a     b
```


版权所有 © 华为云计算技术有限公司

```
0    1    c
1    1    a
2    2    a
[3 rows x 2 columns]
```

filter

描述：过滤掉不满足条件的行。

语法格式：

```
filter(expr=None, fn=None, on=None, num_cpus=None, memory=None, arch=None, device=None,
model=None, num_models=None, concurrency=None, **fn_constructor_kwargs)
```

输入参数：

参数	是否必选	参数类型	描述
expr	否	Union[list[ir.BooleanValue], Sequence[ir.BooleanValue], IfAnyAll, None]	表达式字符串，需为有效的Python表达式。
on	否	Union[str, list[str]]	用于应用函数的列。
fn	否	Union[str, Callable, None]	可调用的函数，或函数名称。
num_cpus	否	Union[float, None]	CPU的数量。
memory	否	Union[int, None]	内存大小，单位为MB。
arch	否	Union[str, None]	架构类型，参数值为x86或arm。
device	否	Union[str, None]	AI处理单元的类型名，参数值为NPU或GPU。
model	否	Union[str, None]	AI处理单元型号，例如910B、A100等。
num_models	否	Union[float, None]	AI处理单元的数量，仅NPU和GPU支持配置。
concurrency	否	Union[int, Tuple[int, int], None]	函数执行时的并发度。
fn_constructor_kwargs	否	-	构造参数，只有当fn为可调用类时，才能提供此项。

返回值类型：过滤掉不满足条件的行后的Dataset。

示例：

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
```


版权所有 © 华为云计算技术有限公司

```
...     ...
... )
>>> ds = con.load_dataset("demo_table_3", database='my_schema')
>>> ds = ds.filter(expr=[ds.a < 2])
>>> ds
| a     | b     |
| ----- | ----- |
| int64 | string|
| ----- | ----- |
| 1     | c     |
| 1     | a     |

# Filtered by User-defined functions
# my_schema.id_filter:inputting an int value, returning false if value is larger than 2.
>>> ds = con.load_dataset("demo_table_2", database='my_schema')
>>> ds = ds.filter(
...     fn='my_schema.id_filter',
...     on='a',
... )
>>> ds
| a     | b     |
| ----- | ----- |
| int64 | string|
| ----- | ----- |
| 1     | c     |
| 1     | a     |
```

flat_map

描述：将给定的函数应用于每一行，并对结果进行展平。

语法格式：

```
flat_map(fn, on, as_col=None, num_cpus=None, memory=None, arch=None, device=None, model=None, num_models=None, concurrency=None, **fn_constructor_kwargs)
```

输入参数：

参数	是否必选	参数类型	描述
fn	是	Union[str, Callable]	可调用的函数或其名称，例如myschema.myfunction。
on	是	Union[str, list[str],]	要应用函数的列或列的集合。
as_col	否	Union[str, list[str], None]	输出新列的名称。as_col的长度必须与fn返回的结构体中字段数量一致。
num_cpus	否	Union[float, None]	CPU的数量。
memory	否	Union[int, None]	内存大小，单位为MB。
arch	否	Union[str, None]	架构类型，参数值为x86或arm。
device	否	Union[str, None]	AI处理单元的类型名，参数值为NPU或GPU。
model	否	Union[str, None]	AI处理单元型号，例如910B、A100等。

版权所有 © 华为云计算技术有限公司

参数	是否必选	参数类型	描述
num_models	否	Union[float, None]	AI处理单元的数量，仅NPU和GPU支持配置。
concurrency	否	Union[int, Tuple[int, int], None]	函数执行时的并发度。
fn_constructor_kwargs	否	-	构造参数，只有当fn为可调用类时，才能提供此项。

返回值类型：

Dataset ： 调用函数后包含名为as_col的新列的数据集。

示例：

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_2", database='my_schema')
>>> ds.execute()
      a     b     c
0     1     4     1
[1 rows x 3 columns]

>>> ds = ds.flat_map(
...     fn='my_schema.py_generate_series',
...     on=['a', 'b', 'c'],
...     as_col=['outCol'])
>>> ds
| a     | b     | c     | outCol |
| ----- | ----- | ----- | ------ |
| int64 | int64 | int64 | int64  |
| ----- | ----- | ----- | ------ |
| 1     | 4     | 1     | 1      |
| 1     | 4     | 1     | 2      |
| 1     | 4     | 1     | 3      |
| 1     | 4     | 1     | 4      |

# Access table columns using the ds.attribute (dot notation) style
>>> ds = ds.flat_map(
...     fn='my_schema.py_generate_series',
...     on=[ds.a, ds.b, ds.c],
...     as_col='udtf_col')
>>> ds
| a     | b     | c     | udtf_col |
| ----- | ----- | ----- | -------- |
| int64 | int64 | int64 | int64    |
| ----- | ----- | ----- | -------- |
| 1     | 4     | 1     | 1        |
| 1     | 4     | 1     | 2        |
| 1     | 4     | 1     | 3        |
| 1     | 4     | 1     | 4        |
```

map_batches

描述：将给定的函数应用于批量数据。

版权所有 © 华为云计算技术有限公司

语法格式：

```
map_batches(fn, on, as_col=None, num_cpus=None, memory=None, arch=None, device=None, model=None, num_models=None, concurrency=None, **fn_constructor_kwargs)
```

输入参数：

参数	是否必选	参数类型	描述
fn	是	Union[str, Callable]	可调用的函数或其名称，例如myschema.myfunction。
on	是	Union[str, list[str],]	要应用函数的列或列的集合。
as_col	否	Union[str, list[str], None]	输出新列的名称。as_col的长度必须与fn返回的结构体中字段数量一致。
num_cpus	否	Union[float, None]	CPU的数量。
memory	否	Union[int, None]	内存大小，单位为MB。
arch	否	Union[str, None]	架构类型，参数值为x86或arm。
device	否	Union[str, None]	AI处理单元的类型名，取值为NPU或GPU。
model	否	Union[str, None]	AI处理单元型号，例如910B、A100等。
num_models	否	Union[float, None]	AI处理单元的数量，仅NPU和GPU支持配置。
concurrency	否	Union[int, Tuple[int, int], None]	函数执行时的并发度。
fn_constructor_kwargs	否	-	构造参数，只有当fn为可调用类时，才能提供此项。

返回值类型：

Dataset：调用函数后包含名为as_col的新列的数据集。

groupby

描述：创建一个分组的Dataset。

语法格式：

```
groupby(*by, **key_exprs)
```

输入参数：

版权所有 © 华为云计算技术有限公司

参数	是否必选	参数类型	描述
by	否	Union[str, Iterable[str], None]	分组表达式。
key_exprs	否	Union[str, Iterable[str]]	命名的分组表达式。

返回值类型：一个分组后的Dataset。

示例：

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_14", database='my_schema')
>>> ds
| fruit  | price |
| ------ | ----- |
| string | float |
| ------ | ----- |
| apple  | 0.50  |
| apple  | 0.50  |
| banana | 0.25  |
| orange | 0.33  |
```

join

描述：在两个Dataset之间执行join操作。

语法格式：

```
join(ds, /, on=(), *, join_type='inner', left_suffix='left_', right_suffix='right_')
```

输入参数：

参数	是否必选	参数类型	描述
ds	是	Dataset	右侧要连接的数据集。
on	否	Union[str, Sequence[Union[str, BooleanColumn, Literal[True], Literal[False], tuple[str, str]]]]	连接条件，详见示例。
join_type	否	JoinKind	连接方法，如 "inner" 或 "left"。
left_suffix	否	str	用于重命名左侧表中重叠列的格式字符串后缀（例如 "left_"）。

版权所有 © 华为云计算技术有限公司

参数	是否必选	参数类型	描述
right_suffix	否	str	用于重命名右侧表中重叠列的格式字符串后缀（例如 "right_"）。

返回值类型：join后的Dataset。

示例：

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds1 = con.load_dataset("demo_table_15", database='my_schema')
>>> ds2 = con.load_dataset("demo_table_16", database='my_schema')

# Equality inner join on the shared id column. Note the _right suffix added to all overlapping columns from the right # table (in this case only the "id" column).
>>> ds1.join(ds2, "id", join_type="inner")
|   id   |  fruit  |  price  |     color     |
| ------ | ------- | ------- | ------------- |
| int64  | string  | float64 | string        |
| ------ | ------- | ------- | ------------- |
|      2 |   apple |    0.25 | Yellow        |
|      3 |  banana |    0.33 | Orange        |

# Explicit equality join using the default join_type value of "inner". Note join_type there is no _right suffix added # to the movieId column since this is an inner join and the movieId column is part of the join condition.
>>> ds1.join(ds2, ds1.id == ds2.id)
|   id   |  fruit  |  price  |     color     |
|--------|---------|---------|---------------|
| int64  | string  | float64 | string        |
|--------|---------|---------|---------------|
|      2 |   apple |    0.25 | Yellow        |
|      3 |  banana |    0.33 | Orange        |
```

map

描述：对当前Dataset的每一行应用给定函数。

语法格式：

```
map(fn, on, as_col=None, num_cpus=None, memory=None, arch=None, device=None, model=None, num_models=None, concurrency=None, **fn_constructor_kwargs)
```

输入参数：

参数	是否必选	参数类型	描述
fn	是	Union[str, Callable]	可调用函数或其名称，如 myschema.myfunction。
on	是	Union[str, list[str],]	要应用函数的列或列的集合。

版权所有 © 华为云计算技术有限公司

参数	是否必选	参数类型	描述
as_col	否	Union[str, list[str], None]	输出新列的名称。as_col的长度必须与fn返回的结构体中字段数量一致。
num_cpus	否	Union[float, None]	CPU的数量。
memory	否	Union[int, None]	内存大小，单位为MB。
arch	否	Union[str, None]	架构类型，取值为x86或arm。
device	否	Union[str, None]	AI处理单元的类型名，取值为NPU或GPU。
model	否	Union[str, None]	AI处理单元型号，例如910B、A100等。
num_models	否	Union[float, None]	AI处理单元的数量，仅NPU和GPU支持配置。
concurrency	否	Union[int, Tuple[int, int], None]	函数执行时的并发度。
fn_constructor_kwargs	否	-	构造参数，只有当fn为可调用类时，才能提供此项。

返回值类型：

Dataset：调用函数后包含新列（列名为as_col）的数据集。

示例：

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table", database='my_schema')
>>> ds = ds.map(
...     fn='my_schema.multiply_2',
...     on=['col_a'],
...     as_col='col_c'
... )
>>> ds
| col_a | col_b | col_c |
|-------|-------|-------|
| int64 | string | int64 |
|-------|-------|-------|
|     1 | c     | 2     |
|     1 | a     | 2     |
|     2 | a     | 4     |
```

order_by

描述：根据一个或多个表达式对Dataset进行排序。

版权所有 © 华为云计算技术有限公司

语法格式：

```
order_by(key, descending=False)
```

输入参数：

参数	是否必选	参数类型	描述
key	否	Union[str, List[str], None]	用于排序的表达式。
descending	否	Union[bool, List[bool]]	是否按降序排序（DESC），默认为False。

返回值类型：排序后的Dataset。

示例：

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_15", database='my_schema')
>>> ds
| a    | b     | c    |
|------|-------|------|
| 3    | a     | 4    |
| 2    | B     | 6    |
| 1    | c     | 5    |
| 3    | D     | 7    |

# Sort by b. Default is ascending. Note how capital letters come before lowercase
>>> ds.order_by("b")
| a    | b     | c    |
|------|-------|------|
| int64 | string | int64 |
|------|-------|------|
| 2    | B     | 6    |
| 3    | D     | 7    |
| 3    | a     | 4    |
| 1    | c     | 5    |

# Sort in descending order
>>> ds.order_by(ds.b.desc())
| a    | b     | c    |
|------|-------|------|
| int64 | string | int64 |
|------|-------|------|
| 1    | c     | 5    |
| 3    | a     | 4    |
| 3    | D     | 7    |
| 2    | B     | 6    |

# Sort by multiple columns/expressions
>>> ds.order_by(["a", ds.c.desc()])
| a    | b     | c    |
|------|-------|------|
| int64 | string | int64 |
|------|-------|------|
| 1    | c     | 5    |
| 2    | B     | 6    |
```


版权所有 © 华为云计算技术有限公司

```
| 3    | D     | 7    |
| 3    | a     | 4    |
```

rename_columns

描述：重命名Dataset中的列。

语法格式：

```
rename_columns(names, /, **substitutions)
```

输入参数：

参数	是否必选	参数类型	描述
names	是	Mapping[str, str]	映射（字典），以 {新列名: 旧列名} 的形式指定重命名规则，未在映射中出现的列将保留原名。
substitutions	否	str	显式重命名的列，以关键字参数形式指定，格式为新名=旧名。

返回值类型：重命名后的新Dataset。

示例：

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_8", database='my_schema')
>>> ds
| a     | b      |
|-------|--------|
| int64 | string |
|-------|--------|
| 1     | c      |
| 1     | a      |
| 2     | a      |
>>> ds.rename_columns({"rename_col": "a"})
| rename_col | b      |
|------------|--------|
| int64      | string |
|------------|--------|
| 1          | c      |
| 1          | a      |
| 2          | a      |
```

schema

描述：返回Dataset的schema。

语法格式：

```
schema()
```

输入参数：无。

返回值类型：Schema对象。

版权所有 © 华为云计算技术有限公司

示例：

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_7", database='my_schema')
>>> ds.schema()
ibis.Schema {
  species            string
  island             string
  bill_length_mm     float64
  bill_depth_mm      float64
  flipper_length_mm  int64
}
```

select_columns

描述：从Dataset的列中选取指定的列。

语法格式：

```
select_columns(*exprs, **named_exprs)
```

输入参数：

参数	是否必选	参数类型	描述
exprs	是	Union[Value, str, Iterable[Union[Value, str]]]	列表达式、列名字符串，或由列表达式和列名字符串组成的列表。
named_exprs	否	Union[Value, str]	列表达式（支持命名参数形式）。

返回值类型：包含选定列的Dataset。

示例：

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_5", database='my_schema')
>>> ds.select_columns("species", "bill_length_mm")
| species    | bill_length_mm |
|------------|----------------|
| string     | float64        |
|------------|----------------|
| Adelie     | 40.0           |
| Adelie     | 41.0           |
| Adelie     | 42.0           |
| Adelie     | 43.0           |

# Selection by zero-indexed column position
>>> ds.select_columns(ds[0], ds[4])
| species    | flipper_length_mm |
```


版权所有 © 华为云计算技术有限公司

```
|------------|--------------------|
| string     | int64              |
|------------|--------------------|
| Adelie     | 180                |
| Adelie     | 183                |
| Adelie     | 190                |
| Adelie     | 185                |
```

show

描述：显示查询结果生成的数据。

语法格式：

```
show(limit=10)
```

输入参数：

参数	是否必选	参数类型	描述
limit	否	int	限制输出的行数，默认为10。

返回值类型：None。

take

描述：从Dataset中返回最多limit行数据，该方法适用于数据检查和预览。

语法格式：

```
take(limit)
```

输入参数：

参数	是否必选	参数类型	描述
limit	是	int	要返回的最大行数。

返回值类型: list[dict[str, Any]] ： 包含数据集中最多limit行的列表，每行以字典形式表示。

示例：

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_9", database='my_schema')
>>> ds.take(3)
[{'a': 1}, {'a': 1}, {'a': 2}]
```

max

描述：计算Dataset中指定列的最大值。

版权所有 © 华为云计算技术有限公司

语法格式：

```
max(on=None)
```

输入参数：

参数	是否必选	参数类型	描述
on	否	Union[str, list[str], None]	指定要计算最大值的列。

返回值类型：Dataset

min

描述：计算Dataset中指定列的最小值。

语法格式：

```
min(on=None)
```

输入参数：

参数	是否必选	参数类型	描述
on	否	Union[str, list[str], None]	指定要计算最小值的列。

返回值类型：Dataset

mean

描述：计算Dataset中指定列的均值。

语法格式：

```
mean(on=None)
```

输入参数：

参数	是否必选	参数类型	描述
on	否	Union[str, list[str], None]	指定要计算均值的列。

返回值类型：Dataset。

unique

描述：计算Dataset中指定列的唯一值集合。

语法格式：

版权所有 © 华为云计算技术有限公司

```
unique(col)
```

输入参数：

参数	是否必选	参数类型	描述
col	是	str	指定要计算唯一值的列。

返回值类型：list。

limit

描述：从当前Dataset中选择行，起始位置为偏移量。

语法格式：

```
limit(n, /, *, offset=0)
```

输入参数：

参数	是否必选	参数类型	描述
n	否	Union[int, None]	要包含的行数。如果为None，则从偏移量开始选择整个数据集。
offset	否	int	起始跳过的行数。

返回值类型：

Dataset ： 从偏移量开始的n行数据。

示例：

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_4", database='my_schema')
>>> ds
| a     | b      |
|-------|--------|
| int64 | string |
|-------|--------|
| 1     | c      |
| 1     | a      |
| 2     | a      |
>>> ds.limit(2)
| a     | b      |
|-------|--------|
| int64 | string |
|-------|--------|
| 1     | c      |
| 1     | a      |
| 2     | a      |
# use None with offset to slice starting from a particular row
```


版权所有 © 华为云计算技术有限公司

```
>>> ds.limit(None, offset=1)
| a      | b      |
|--------|--------|
| int64  | string |
|--------|--------|
| 1      | a      |
| 2      | a      |
```

take_batch

描述： 从Dataset中返回最多batch_size行数据。

语法格式:

```
take_batch(batch_size=20, *, batch_format='default')
```

输入参数:

参数	是否必选	参数类型	描述
batch_size	否	int	batch包含的行数（默认为20）。
batch_format	否	Optional[str]	batch的格式。 ● "default"返回pyarrow.Table。 ● "pandas"返回pandas.DataFrame。 ● "numpy"返回字典形式的numpy数组。

返回值类型：按指定格式输出数据集合。

create_async_job

描述：根据作业id恢复作业实例。

语法格式:

```
create_async_job(statement_id)
```

输入参数:

参数	是否必选	参数类型	描述
statement_id	是	str	作业唯一标识符。

返回值：AsyncJob对象。

explain_plan

描述：生成作业执行计划。

语法格式:

版权所有 © 华为云计算技术有限公司

```
explain_plan(verbose=True)
```

输入参数：

参数	是否必选	参数类型	描述
verbose	否	bool	如果为True，提供查询计划的详细说明；如果为False，则仅提供简单的执行计划。

返回值类型：str。

explain_performance

描述：执行查询并生成计划，计划中包含执行过程中详细的性能指标。

语法格式：

```
explain_performance()
```

输入参数：无。

返回值类型：str，详细的执行计划。

2.1.5.8 Table APIs

insert

描述：用于将Dataset插入到数据库表中。

语法格式：

```
insert(data, overwrite=False)
```

输入参数：

参数	是否必选	参数类型	描述
data	是	dataset.Dataset	要插入的Dataset对象，必须是aura_frame.multimodal.dataset.Dataset类型。
overwrite	否	bool	是否覆盖已存在的数据，默认为False。

返回值类型：None。

show

描述：用于显示表的前N行数据。

语法格式：

```
show(limit=10)
```

输入参数：

版权所有 © 华为云计算技术有限公司

参数	是否必选	参数类型	描述
limit	否	int	限制输出的行数，默认为10。

返回值类型：None。

to_dataset

描述：将当前表对象转换为Dataset对象。

语法格式：

```
to_dataset()
```

输入参数：无

返回值类型：Dataset

add_columns

描述：向表中添加一个或多个新列。

语法格式：

```
add_columns(new_cols)
```

输入参数：

参数	是否必选	参数类型	描述
new_cols	是	Dict[str, str]	要添加的列的字典，键为列名（str），值为列类型（str）。

返回值类型：None。

drop_columns

描述：用于从表中删除一个或多个列。

语法格式：

```
drop_columns(*columns)
```

输入参数：

参数	是否必选	参数类型	描述
*columns	是	str	要删除的列名，可以是多个字符串参数。

返回值类型：None。

版权所有 © 华为云计算技术有限公司

alter_column_type

描述：修改表中指定列的数据类型。

语法格式：

```
alter_column_type(column_name, new_type)
```

输入参数：

参数	是否必选	参数类型	描述
column_name	是	str	要修改类型的列名。
new_type	是	str	新的数据类型名称。

返回值类型：None。

rename_column

描述：重命名表中的列名。

语法格式：

```
rename_column(column_name, new_name)
```

输入参数：

参数	是否必选	参数类型	描述
column_name	是	str	当前列名。
new_name	是	str	新列名。

返回值类型：None。

delete

描述：从表中删除满足指定条件的数据。

语法格式：

```
delete(where)
```

输入参数：

参数	是否必选	参数类型	描述
where	是	str	SQL WHERE子句：用于指定删除条件。

返回值类型：None。

版权所有 © 华为云计算技术有限公司

update

描述：更新表中满足指定条件的行数据。

语法格式：

```
update(values_expr, where)
```

输入参数：

参数	是否必选	参数类型	描述
values_expr	是	Dict[str, ValueExpression]	键为列名（str），值为用于更新该列的表达式（ValueExpression）。
where	否	Union[ir.BooleanValue, list[ir.BooleanValue], IfAnyAll, None]	用于过滤要更新的行的条件。如果为None，则更新所有行。

返回值类型：None。

create_table

描述：在指定数据库中创建一个新表。

语法格式：

```
create_table(name, /, *, schema, table_format, database=None, temp=False, external=False, overwrite=False, if_not_exists=False, partition_by=None, table_properties=None, location=None)
```

输入参数：

参数	是否必选	参数类型	描述
name	是	str	表的名称。
schema	是	ibis.Schema	表的定义。
table_format	是	str	表的存储格式。
database	否	Optional[str]	要创建表的数据库名称。如果未指定，则使用默认数据库。
temp	否	bool	如果为True，则创建临时表。
external	否	bool	如果为True，则创建外部表。
overwrite	否	bool	如果为True，则在表已存在时覆盖原表。
if_not_exists	否	bool	如果为True，则仅在表不存在时创建表。
partition_by	否	Optional[ibis.Schema]	用于对表进行分区的架构。

版权所有 © 华为云计算技术有限公司

参数	是否必选	参数类型	描述
table_properties	否	Optional[dict]	表的其他属性。
location	否	Optional[str]	表数据存储的位置。

返回值类型: Dataset。

delete_table

描述：从数据库中删除指定的Dataset。

语法格式:

```
delete_table(name, /, *, database=None, if_it_exists=False)
```

输入参数:

参数	是否必选	参数类型	描述
name	是	str	要删除的表的名称。
database	否	Optional[str]	要删除表的数据库名称。如果未提供，则使用默认数据库。
if_it_exists	否	bool	如果设置为True，则仅在表存在时删除，避免因不存在而报错。

返回值类型 ： str。

describe_table

描述：描述指定数据库中的表。

语法格式:

```
describe_table(name, /, *, database=None)
```

输入参数:

参数	是否必选	参数类型	描述
name	是	str	函数名称。
database	否	Optional[str]	表所在的数据库名称。

返回值类型：str，表的元数据信息。

版权所有 © 华为云计算技术有限公司

drop_table

描述：从指定数据库中删除一个表。

语法格式：

```
drop_table(name, *, database=None, force=False)
```

输入参数：

参数	是否必选	参数类型	描述
name	是	str	要删除的表的名称。
database	否	Optional[str]	要删除表的数据库名称。如果未提供，则使用默认数据库。
force	否	bool	如果设置为True，则即使表包含数据也会强制删除。默认为False。

返回值类型 ： None。

list_tables

描述：列出指定数据库或默认数据库中的所有表。

语法格式：

```
list_tables(database=None)
```

输入参数：

参数	是否必选	参数类型	描述
database	否	Optional[str]	要列出表的数据库名称。如果为None，则使用默认数据库。

返回值类型：list[tuple]，包含表名称的元组列表。

open_table

描述：打开指定数据库中的表。

语法格式：

```
open_table(name, *, database=None)
```

输入参数：

版权所有 © 华为云计算技术有限公司

参数	是否必选	参数类型	描述
name	是	str	要打开的表的名称。
database	否	Optional[str]	包含该表的数据库名称。如果未提供，则使用默认数据库。

返回值类型： Table。

2.1.5.9 I/O APIs

write_iceberg

描述：将数据写入指定的Iceberg表中。

语法格式：

```
write_iceberg(target_name, database, create_if_not_exists=False, overwrite=False, is_external=False, schema=None, explain = None, verbose = True)
```

输入参数：

参数	是否必选	参数类型	描述
target_name	是	str	目标表的名称。
database	是	str	表所属的数据库。
create_if_not_exists	否	bool	如果为True，当表不存在时将自动创建该表。
overwrite	否	bool	如果为True，将覆盖表中现有的数据。
is_external	否	bool	如果为True，将表视为外部表。
schema	否	任意类型	表的结构定义，可选参数。
explain	否	字符串	可以设置为plan或者performance，分别表示单纯打印执行计划和打印执行的详细信息。
verbose	否	bool	如果为True，explain为plan时，会增加打印的信息。

返回值类型：str。

write_orc

描述：将数据写入指定的ORC表中。

语法格式：

版权所有 © 华为云计算技术有限公司

```
write_orc(target_name, database, create_if_not_exists=False, overwrite=False, is_external=False,
schema=None, explain = None, verbose = True)
```

输入参数:

参数	是否必选	参数类型	描述
target_name	是	str	目标表的名称。
database	是	str	表所属的数据库。
create_if_not_exists	否	bool	如果为True，当表不存在时将自动创建该表。
overwrite	否	bool	如果为True，将覆盖表中现有的数据。
is_external	否	bool	如果为True，将表视为外部表。
schema	否	任意类型	表的结构定义，可选参数。
explain	否	字符串	可以设置为plan或者performance，分别表示单纯打印执行计划和打印执行的详细信息。
verbose	否	bool	如果为True，explain为plan时，会增加打印的信息。

返回值类型：str。

write_parquet

描述：将数据写入指定的parquet表中。

语法格式:

```
write_parquet(target_name, database, create_if_not_exists=False, overwrite=False, is_external=False,
schema=None, explain = None, verbose = True)
```

输入参数:

参数	是否必选	参数类型	描述
target_name	是	str	目标表的名称。
database	是	str	表所属的数据库。
create_if_not_exists	否	bool	如果为True，当表不存在时将自动创建该表。
overwrite	否	bool	如果为True，将覆盖表中现有的数据。
is_external	否	bool	如果为True，将表视为外部表。

版权所有 © 华为云计算技术有限公司

参数	是否必选	参数类型	描述
schema	否	任意类型	表的结构定义，可选参数。
explain	否	字符串	可以设置为plan或者performance，分别表示单纯打印执行计划和打印执行的详细信息。
verbose	否	bool	如果为True，explain为plan时，会增加打印的信息。

返回值类型：str。

write_text

描述：将文本数据写入到指定路径的OBS中。

语法格式：

```
write_text(path, field_delim, max_rows_per_file=None)
```

输入参数：

参数	是否必选	参数类型	描述
path	是	str	OBS中存放文本的路径。
field_delim	是	str	指定TEXT数据文件行数据的字段分隔符，不能是'\ \.abcdefghijklmnopqrstuvwxyz0123456789'。
max_rows_per_file	否	int	指定一个文件写入最大行数。

返回值类型：None。

write_json

描述：将JSON数据写入到指定路径的OBS中。

语法格式：

```
write_json(path, max_rows_per_file=None, ignore_null_fields=False, nested_separator=None)
```

输入参数：

版权所有 © 华为云计算技术有限公司

参数	是否必选	参数类型	描述
path	是	str	OBS中存放JSON数据的路径。
max_rows_per_file	否	int	指定一个文件写入最大行数。
ignore_null_fields	否	bool	写入JSON对象时是否忽略JSON value为NULL的JSON key。
nested_separator	否	str	设置嵌套JSON key拼接分隔符，不能指定为# ，比如：设置'_'后，{"a":{"b":1}}映射到列a_b，列a_b值为1；默认不支持JSON嵌套读写。

返回值类型：None。

read_text

描述：从OBS指定的路径中读取TEXT文件到指定schema的Dataset中。

语法格式：

```
read_text(path, schema, field_delim='\t', drop_empty_lines=True, fill_missing_fields=False,
ignore_extra_data=False)
```

输入参数：

参数	是否必选	参数类型	描述
path	是	str	OBS中存放文本文件的目录路径。
schema	是	dict	读取文本文件的列信息。
field_delim	否	str	指定TEXT数据文件行数据的字段分隔符，不能是'\ \.abcdefghijklmnopqrstuvwxyz0123456789'。
drop_empty_lines	否	bool	是否丢弃文本中的空行或单行的\n 、\r、\t。
fill_missing_fields	否	bool	当行按照field_delim进行分隔时，是否在分隔结果小于列数时报错。
ignore_extra_data	否	bool	当行按照field_delim进行分隔时，是否在分隔结果多于列数时报错。

返回值类型：dataset，dataset是text文本集合。

版权所有 © 华为云计算技术有限公司

read_json

描述：从OBS指定的路径中按照schema读取指定JSON key的JSON value。

语法格式：

```
read_json(path, schema, skip_empty_files=False, nested_separator=None)
```

输入参数：

参数	是否必选	参数类型	描述
path	是	str	OBS中存放JSON文件的目录路径。
schema	是	dict	读取指定key的列信息。
skip_empty_files	否	bool	是否跳过JSON内容为'{}'的内容。
nested_separator	否	str	设置嵌套JSON key拼接分隔符，不能指定为#，比如：设置'_'后，{"a": {"b":1}}映射到列a_b，列a_b值为1；默认不支持JSON嵌套读写。

返回值类型：dataset，dataset是JSON对象集合。

2.1.5.10 User-Defined Function APIs

UDF 显式注册语法

显式注册的含义是用户需要手动在Python代码中侵入式添加注册逻辑代码，需要用户使用backend...register/register_from_file来实现，调用即注册。显式注册依赖于已经获得backend会话对象才能进行。

推荐使用显式注册的场景：如果用户希望明确控制注册时间，允许侵入式添加注册逻辑，或对同一个Backend连接下的UDF注册和使用分离有要求。

一个典型的场景是1个开发团队负责UDF的注册，多个团队负责UDF的使用，注册团队和使用团队之间的Python脚本不互通。

表 2-13 UDF 显式注册语法

UDF类型	UDF类型（二级）	注册类型（三级）	代码入口	参考
udf, udaf, udtf	python	直接注册	backend.[udf | udaf | udtf].python.register(<注册函数>, <注册参数>)	Python/Pyarrow/Pandas UDF注册参数
		从文件注册	backend.[udf | udaf | udtf].python.register_from_file(<文件路径>, <函数名>, <注册参数>)	Python/Pyarrow/Pandas UDF注册参数

版权所有 © 华为云计算技术有限公司

UDF类型	UDF类型（二级）	注册类型（三级）	代码入口	参考
	builtin	直接注册	backend.[udf | udaf | udtf].builtin.register(<注册函数>, <注册参数>)	Builtin UDF注册参数
		从文件注册	backend.[udf | udaf | udtf].builtin.register_from_file(<文件路径>, <函数名>, <注册参数>)	Builtin UDF注册参数
	pyarrow	直接注册	backend.udf.pyarrow.register(<注册函数>, <注册参数>)	Python/Pyarrow/Pandas UDF注册参数
		从文件注册	backend.udf.pyarrow.register_from_file(<文件路径>, <函数名>, <注册参数>)	Python/Pyarrow/Pandas UDF注册参数
	pandas	直接注册	backend.udf.pandas.register(<注册函数>, <注册参数>)	Python/Pyarrow/Pandas UDF注册参数
		从文件注册	backend.udf.pandas.register_from_file(<文件路径>, <函数名>, <注册参数>)	Python/Pyarrow/Pandas UDF注册参数

UDF 隐式注册语法

隐式注册的含义是依赖Python运行时自动发现并注册UDF。用户不需要在Python代码中侵入式添加注册逻辑代码，而是使用@装饰器修饰原始Python函数，然后在DataFrame中使用被装饰的原始Python函数的标识符，即可完成注册。隐式注册在使用@装饰器时不需要获得backend会话对象，后续在ibis DataFrame中获取backend会话对象。

推荐使用隐式注册的场景：如果用户希望无侵入式地注册UDF，且对同一个Backend连接下的UDF注册和使用分离无要求。

一个典型的场景是1个用户的Python脚本中直接写了整个UDF的注册和使用全流程。

表 2-14 UDF 隐式注册语法

UDF类型	UDF类型（二级）	代码入口	参考
udf, udaf, udtf	python	@aura.[udf | udaf | udtf].python(<注册参数>)	Python/Pyarrow/Pandas UDF注册参数
	builtin	@aura.[udf | udaf | udtf].builtin(<注册参数>)	Builtin UDF注册参数

版权所有 © 华为云计算技术有限公司

UDF类型	UDF类型（二级）	代码入口	参考
	pyarrow	@aura.udf.pyarrow(<注册参数>)	Python/Pyarrow/Pandas UDF注册参数
	pandas	@aura.udf.pandas(<注册参数>)	Python/Pyarrow/Pandas UDF注册参数

对于隐式注册，实际注册动作的发生时间根据DataFrame操作模式Lazy、Eager有所区别。

参考上文提及的ibis官方文档，对于DataFrame的操作分为Eager、Lazy两种模式，由配置项ibis.options.interactive控制，默认为False，即所有的DataFrame都默认是Lazy模式。对于两种DataFrame执行模式，UDF注册的发生时间不同，详情说明如下：

表 2-15 DataFrame 执行模式

ibis.options.interactive	DataFrame执行模式	UDF注册时间	UDF使用时间
False	Lazy	整个DataFrame调用execute方法时	整个DataFrame调用execute方法时
True	Eager	第一次在DataFrame中使用	每一次在DataFrame中使用

Python/Pyarrow/Pandas UDF 注册参数

注册Python/Pyarrow/Pandas UDF的作用是将一个原始的Python函数或类注册进数据库中。

不管是显式注册还是隐式注册，不管是Scalar UDF、UDAF、UDTF，对于注册Python、Pyarrow、Pandas类型的UDF，目前都接受用户传入以下参数：

表 2-16 Python/Pyarrow/Pandas UDF 注册参数

注册参数	含义	类型	默认值
name	指定UDF实际数据库中存储名称。	str | None	None
database	指定UDF所在的LakeFormation数据库。	str | None	None
fn	指定UDF原始的Python函数。	Callable	None

版权所有 © 华为云计算技术有限公司

注册参数	含义	类型	默认值
signature	指定UDF函数签名和返回值类型。	aura_frame.ibis.common.annotations.Signature | None	None
replace（目前不可用）	指定UDF是否支持替换。	bool	False
temporary（目前不可用）	指定UDF是否会话级别的生命周期。	bool	False
if_not_exist（目前不可用）	指定UDF是否跳过已存在报错。	bool	False
strict	指定UDF是否自动过滤NULL值。	bool	True
volatility	指定UDF稳定性。	VolatilityType.VOLATILE | VolatilityType.STABLE | VolatilityType.IMMUTABLE	VolatilityType.VOLATILE
runtime_version（目前不可用）	指定UDF执行的Python版本。	str	sys.version_info
runtime_env	指定UDF依赖的Python模块和安装方式（venv），后续用于取代packages	Union[str, Mapping[str, Any]] | None	None
imports	指定UDF依赖的外部代码文件。	List[str]	None
packages	指定UDF依赖的Python模块。	List[Union[str, module]] | None	None
register_type	指定UDF的注册形式。	RegisterType.INLINE | RegisterType.STAGED	RegisterType.INLINE
comment	指定UDF的用户注释。	str | None	None

Python/Pyarrow/Pandas UDF 注册参数注意事项

- 对于import参数，只支持用户传入当前Python函数或类所在的.py文件同级目录或子目录下的文件路径。
- 对于fn参数，如果fn不在当前注册UDF的.py文件中，那么需要同时在imports参数中添加fn定义的文件路径，例如：

```
from process import outer
from aura_frame.multimodal import ai_lake
```


版权所有 © 华为云计算技术有限公司

```
con = ai_lake.connect(
        aura_endpoint=os.getenv("aura_endpoint"),
        aura_endpoint_name=os.getenv("aura_endpoint_name"),
        aura_workspace_id=os.getenv("aura_workspace_id"),
        lf_catalog_name=os.getenv("lf_catalog_name"),
        access_key=os.getenv("access_key"),
        secret_key=os.getenv("secret_key"),
        default_database=target_database,
        use_single_cn_mode=True
)

# 注册UDAF
udf = con.udaf.python.register(
    outer(), #从外部引入的fn
    imports=["process.py"] #为fn添加文件路径
)
```

- 对于signature参数，目前允许用户传入，用户传入的优先级高于自动推断。当用户不传入时，进入参数/返回值类型自动推断逻辑，详情请参见**signature参数的类型推断**。
  - 对于注册Pyarrow UDF，无论用户是否传入signature参数，都需要依赖PyarrowVector，例如：

```
import aura_frame as aura
import pyarrow as pa
import pyarrow.compute as pc
from aura_frame.multimodal import ai_lake

# === 用户传入signature，需要依赖PyarrowVector ===
def calculate_sum(
    prices: pa.ChunkedArray,
    quantities: pa.ChunkedArray,
) -> pa.ChunkedArray:
    return pc.multiply(prices, quantities)

con = ai_lake.connect(
        aura_endpoint=os.getenv("aura_endpoint"),
        aura_endpoint_name=os.getenv("aura_endpoint_name"),
        aura_workspace_id=os.getenv("aura_workspace_id"),
        lf_catalog_name=os.getenv("lf_catalog_name"),
        access_key=os.getenv("access_key"),
        secret_key=os.getenv("secret_key"),
        default_database=target_database,
        use_single_cn_mode=True
)

# 注册UDF
con.create_scalar_function(
        calculate_sum,
        database=target_database
    )

# === 用户不传入signature，也需要依赖PyarrowVector ===
def calculate_sum(
    prices: aura.PyarrowVector[float],
    quantities: aura.PyarrowVector[int],
) -> aura.PyarrowVector[float]:
    return aura.PyarrowVector[float](pc.multiply(prices, quantities))

con = ai_lake.connect(
        aura_endpoint=os.getenv("aura_endpoint"),
        aura_endpoint_name=os.getenv("aura_endpoint_name"),
        aura_workspace_id=os.getenv("aura_workspace_id"),
        lf_catalog_name=os.getenv("lf_catalog_name"),
        access_key=os.getenv("access_key"),
        secret_key=os.getenv("secret_key"),
        default_database=target_database,
        use_single_cn_mode=True
```


版权所有 © 华为云计算技术有限公司

```
)

# 注册UDF
con.create_scalar_function(
        calculate_sum,
        database=target_database
    )
```

- 对于注册Pandas UDF，无论用户是否传入signature参数，都需要依赖PandasVector，例如：

```
import aura_frame as aura
import pandas as pd
from aura_frame.multimodal import ai_lake

# === 用户传入signature，需要依赖PandasVector ===
def calculate_sum(
    prices: pd.Series,
    quantities: pd.Series,
) -> pd.Series:
    return pd.Series(prices * quantities, dtype=pd.Float64Dtype())

con = ai_lake.connect(
        aura_endpoint=os.getenv("aura_endpoint"),
        aura_endpoint_name=os.getenv("aura_endpoint_name"),
        aura_workspace_id=os.getenv("aura_workspace_id"),
        lf_catalog_name=os.getenv("lf_catalog_name"),
        access_key=os.getenv("access_key"),
        secret_key=os.getenv("secret_key"),
        default_database=target_database,
        use_single_cn_mode=True
)

# 注册UDF
con.create_scalar_function(
        calculate_sum,
        database=target_database
    )

# === 用户不传入signature，也需要依赖PandasVector ===
def calculate_sum(
    prices: aura.PandasVector[float],
    quantities: aura.PandasVector[int],
) -> aura.PandasVector[float]:
    return aura.PandasVector[float](prices * quantities)

con = ai_lake.connect(
        aura_endpoint=os.getenv("aura_endpoint"),
        aura_endpoint_name=os.getenv("aura_endpoint_name"),
        aura_workspace_id=os.getenv("aura_workspace_id"),
        lf_catalog_name=os.getenv("lf_catalog_name"),
        access_key=os.getenv("access_key"),
        secret_key=os.getenv("secret_key"),
        default_database=target_database,
        use_single_cn_mode=True

)

# 注册UDF
con.create_scalar_function(
        calculate_sum,
        database=target_database
    )
```

- 对于volatility参数，3个枚举类型的含义是：
  - VolatilityType.VOLATILE：函数结果可能在任何时候都变化。
  - VolatilityType.STABLE：函数对于固定输入其结果在一次扫描里不变。
  - VolatilityType.IMMUTABLE：函数对于相同的输入总是输出相同的结果。

版权所有 © 华为云计算技术有限公司

volatility参数不会影响函数下推执行，对于IMMUTABLE/STABLE/VOLATILE类型的函数，Python UDF函数都可以下推到DN执行。

- 对于packages参数，如果用户没有指定：
  - 注册Pyarrow UDF会自动使用后端环境的pyarrow版本。
  - 注册Pandas UDF会自动使用后端环境的pandas版本。

  为了考虑到安装Python第三方依赖的隔离性，packages参数中指定的包会被传给RuntimeEnv进行安装，默认的安装方式是venv。

Builtin UDF 注册参数

注册Builtin UDF的作用是获得数据库已存在的函数的句柄，无实际注册的操作。

不管是显式注册还是隐式注册，不管是Scalar UDF、UDAF、UDTF，对于注册Builtin类型的UDF，目前都接受用户传入以下参数：

表 2-17 Builtin UDF 注册参数

注册参数	含义	类型	默认值
name	指定UDF实际数据库存储名称	str | None	None
database	指定UDF所在的LakeFormation数据库	str | None	None
fn	指定UDF原始的Python函数	Callable	None
signature	指定UDF函数签名和返回值类型	ibis.common.annotations.Signature | None	None

Builtin UDF 注册参数注意事项

对于signature参数，目前允许用户传入，用户传入的优先级高于自动推断。当用户不传入时，进入参数/返回值类型自动推断逻辑，详情请参见**signature参数的类型推断**。

signature 参数类型的推断

对于signature参数，允许用户传入参数/返回值类型，也允许用户不传入。

- 如果用户传入signature参数，不需要原始Python函数使用类型注解（type hints）语法，此时可以支持及时操作式的注册UDF。
- 如果用户不传入signature参数，推荐对原始Python函数使用类型注解（type hints）语法，此时不能支持及时操作式的注册UDF。

两种方式的区别总结如下：

版权所有 © 华为云计算技术有限公司

表 2-18 signature 参数说明

signature参数	含义	需要原始Python函数使用类型注解语法	支持REPL及时操作
用户不传值	自动推断（推荐）	否，但是推荐使用	否
用户传值	指定传值	否	是

此处的及时操作指的是读取–求值–输出循环（Read–Eval–Print Loop, REPL），通常见于Python用户交互式终端（Terminal）中。

类型注解（type hints）语法由Python 3.5引入（PEP 484），在函数定义中，类型注解通过在参数名后加冒号（:）和类型，以及在参数列表末尾使用箭头（->）指定返回类型来实现，示例如下：

```
def greet(name: str) -> str:
    return f"Hello, {name}"
from typing import List, Dict, Optional

def process_data(data: List[int]) -> Dict[str, Optional[int]]:
    return {"max": max(data) if data else None}
```

对于Python/Pyarrow/Pandas UDF，注册时要求强数据类型，所有的参数/返回值都需要指定类型。如果用户不能通过原始Python函数的类型注解语法来指明，那么就需要用户主动使用signature参数指定ibis DataType。

对于Builtin UDF，注册时不要求强数据类型（因为数据库中已经注册了该UDF函数）。如果用户不能通过原始Python函数的类型注解语法来指明，那么推荐用户只写出参数名称，不写出类型；如果用户后续用到Builtin UDF的返回值（非Top SELECT UDF），那么需要指明函数返回值类型，必要时需要用户主动使用signature参数指定ibis DataType，如果不需要（Top SELECT UDF）则允许用户不写出函数返回值类型。

对于用户不传入signature参数，依赖自动推断时的总结如下：

表 2-19 signature 自动推断

注册UDF类型	参数类型	返回值类型
Python/ Pyarrow/ Pandas UDF	需要类型注解（type hints）语法指定。	需要类型注解（type hints）语法指定。
Builtin UDF	可以只写出参数名称，不写出类型。	后续使用返回值时需要类型注解（type hints）语法指定，否则不需要。

对于用户不传入signature参数，自动推断的情况，底层实现原理是inspect.signature。目前，接受用户传入以下参数/返回值类型：

版权所有 © 华为云计算技术有限公司

表 2-20 接受的参数/返回值类型

Python类型	ibis DataType类型	对应Aura类型
DataType	DataType	-
type(None)	null	NULL
bool	Boolean	BOOLEAN
bytes	Binary	BYTEA
str	String	TEXT
numbers.Integral	Int64	BIGINT
numbers.Real	Float64	DOUBLE PRECISION
decimal.Decimal	Decimal	DECIMAL
datetime.datetime	Timestamp	TIMESTAMP/ TIMESTAMPTZ
datetime.date	Date	TIMESTAMP
datetime.time	Time	TIME
datetime.timedelta	Interval	INTERVAL
uuid.UUID	UUID	UUID
class	Struct	STRUCT
typing.Sequence, typing.Array	Array	ARRAY
typing.Mapping, typing.Map	Map	HSTORE
aura_frame.PyarrowVector[T]	T	T
aura_frame.PandasVector[T]	T	T

注意事项：

- Python内置int类型属于numbers.Integral的子类。
- Python内置float类型属于numbers.Real的子类。

上表中没有列出的Python类型，都是目前暂时不支持的自动转化类型。

对于用户不传入signature参数，同时也没有写出Python类型注解（ type hints ）语法的参数/返回值，目前自动推断采取如下的方式处理：

版权所有 © 华为云计算技术有限公司

表 2-21 特殊参数类型处理

参数类型	生成匹配Pattern	Pattern效果
POSITIONAL_ONLY, KEYWORD_ONLY, POSITIONAL_OR_KEYWORD	ValueOf(None)	免除__signature__.validate。
VAR_POSITIONAL	TupleOf(pattern=pattern)	for-loop执行pattern。
VAR_KEYWORD	DictOf(key_pattern=InstanceOf(str), value_pattern=pattern)	for-loop执行pattern。
Return	ValueOf(Unknown)	提供UnknownScalar, UnknownColumn作为UDF返回值向上返回。

inspect.signature对于参数类型（Parameter.kind）的分类如下：

表 2-22 inspect.signature 参数类型

参数类型	含义	示例代码	满足条件的参数
POSITIONAL_ONLY	仅限位置参数。	def func(a, /, b): pass	a
KEYWORD_ONLY	仅限关键字参数。	def func(a, *, b): pass	b
POSITIONAL_OR_KEYWORD	位置或关键字参数。	def func(a, b): pass	a, b
VAR_POSITIONAL	可变位置参数。	def func(*args): pass	args
VAR_KEYWORD	可变关键字参数。	def func(**kwargs): pass	kwargs

UDF 直接操作语法

在注册和使用分离的场景下，为使用者提供Scalar UDF、UDAF、UDTF的直接操作语法，使用者只需要知道UDF名称（name）、所在的数据库名称（database），就可以直接操作使用UDF。以下操作依赖Backend会话对象的udf属性进行。

signature

描述：从后端DB中返回UDF的函数签名和返回值类型。

版权所有 © 华为云计算技术有限公司

语法格式：

```
signature(name, database=None)
```

输入参数：

参数	是否必选	参数类型	描述
name	是	str	UDF函数名称。
database	否	str	UDF函数所属的数据库名称。

返回值类型：aura_frame.ibis.common.annotations.Signature - 已注册的UDF函数签名和返回值类型。

get

描述：从后端DB中返回UDF函数。

语法格式：

```
get(name, database=None)
```

输入参数：

参数	是否必选	参数类型	描述
name	是	str	UDF函数名称。
database	否	str	UDF函数所属的数据库名称。

返回值类型：Callable[..., ibis.expr.types.Value] - 已注册的UDF函数。

names

描述：从后端DB中返回所有的UDF函数名称。

语法格式：

```
names(database=None)
```

输入参数：

参数	是否必选	参数类型	描述
database	否	str	UDF函数所属的数据库名称。

返回值类型：List[str] - 所有已注册的UDF函数名称。

版权所有 © 华为云计算技术有限公司

unregister

描述：从后端DB中删除指定的UDF函数。

语法格式：

```
unregister(name, database=None)
```

输入参数：

参数	是否必选	参数类型	描述
name	是	str	UDF函数名称。
database	否	str	UDF函数所属的数据库名称。

返回值类型：None

UDF WITH ARGUMENTS 使用语法

不管是使用**UDF显式注册语法**，**UDF隐式注册语法**后返回的UDF算子，还是通过**UDF直接操作语法**返回的已注册UDF，目前都支持用户通过with_arguments方法传入参数，参数总体来说分为两类：

- 特殊含义的参数名，这些参数用来配置UDF运行时的资源、并发和执行上限时间，例如concurrency、timeout、dpu、apu，具体情况可见**表2-30**。所有的UDF类型目前都支持用户使用with_arguments传入配置参数。
- 一般的参数名，这些参数用来配置UDF初始化时的状态，只有Class UDF、Class UDTF、UDAF这些用户在__init__方法中定义了可选参数的情况下可以传入Scalar标量值。实现一次__init__初始化内部状态，后续多次使用。

表 2-23 WITH ARGUMENTS 语法适用情况

参数名	适用UDF类型	含义
特殊含义的参数名，如concurrency、timeout、dpu、apu	所有UDF类型，包括Scalar UDF、Class UDF、Vectorized UDF、UDTF、UDAF	配置UDF运行时的资源、并发和执行上限时间。
一般的参数名，由Python Class的__init__方法定义	Class UDF类型，包括Class UDF、Class UDTF、UDAF	配置UDF初始化状态，适合有内部缓存、初始化参数、模型对象等情况。

所有经过with_arguments方法传入的参数值都是Scalar标量，所有参数值整体作为一个**kwargs（Python字典类型）传入。

create_agg_function

描述：注册一个用户自定义聚合函数（UDAF）。

版权所有 © 华为云计算技术有限公司

语法格式：

```
create_agg_function(udf, /, input_type=InputType.PYTHON, *, database, name=None, signature=None, imports=None, packages=None, runtime_env=None, comment=None, **kwargs)
```

输入参数：

参数	是否必选	参数类型	描述
udf	是	Callable	要注册的用户自定义聚合函数。
input_type	否	InputType	函数的输入类型，可选值包括PYTHON、BUILTIN、PYARROW、PANDAS。
database	是	str	函数所属的数据库名称。
name	否	str	函数的名称，默认为None。
signature	否	Signature	函数的签名定义，可选。
imports	否	[Union[str, Iterable[str]]]	函数所需的导入模块，可以是单个字符串或字符串列表。
packages	否	[Union[str, Iterable[str]]]	函数所需的依赖包，可以是单个包名或包名列表。
runtime_env	否	[Union[str, Mapping[str, Any]]]	函数所需的依赖包和安装方法，可以是JSON格式的字符串或者Python字典。
comment	否	str	函数的注释说明，可选。
**kwargs	否	-	其他可选的关键字参数。

返回值类型：

Callable ： 注册后的函数。

create_scalar_function

描述：注册一个新的标量用户自定义函数（ Scalar UDF ）。

语法格式：

```
create_scalar_function(udf, /, input_type=InputType.PYTHON, *, database, name=None, imports=None, packages=None, runtime_env=None, comment=None, **kwargs)
```

输入参数：

参数	是否必选	参数类型	描述
udf	是	Callable	要注册的用户自定义标量函数。

版权所有 © 华为云计算技术有限公司

参数	是否必选	参数类型	描述
input_type	否	InputType	函数的输入类型，可选值包括PYTHON、BUILTIN、PYARROW、PANDAS。
database	是	str	函数所属的数据库名称。
name	否	str	函数的名称，默认为None。
imports	否	[Union[str, Iterable[str]]]	函数所需的导入模块，可以是单个字符串或字符串列表。
packages	否	[Union[str, Iterable[str]]]	函数所需的依赖包，可以是单个包名或包名列表。
runtime_env	否	[Union[str, Mapping[str, Any]]]	函数所需的依赖包和安装方法，可以是JSON格式的字符串或者Python字典。
comment	否	Optional[str]	函数的注释说明，可选。
**kwargs	否	-	其他可选的关键字参数。

返回值类型：Callable。

create_table_function

描述：注册一个新的用户自定义表值函数（UDTF）。

语法格式：

```
create_table_function(udf, /, input_type=InputType.PYTHON, *, database, name=None, signature=None, imports=None, packages=None, runtime_env=None, comment=None, **kwargs)
```

输入参数：

参数	是否必选	参数类型	描述
udf	是	Callable	要注册的用户自定义表值函数。
input_type	否	InputType	函数的输入类型，可选值包括PYTHON、BUILTIN。
database	是	str	函数所属的数据库名称。
name	否	str	函数的名称，默认为None。
signature	否	Signature	函数的签名定义，可选。
imports	否	[Union[str, Iterable[str]]]	函数所需的导入模块，可以是单个字符串或字符串列表。

版权所有 © 华为云计算技术有限公司

参数	是否必选	参数类型	描述
packages	否	[Union[str, Iterable[str]]]	函数所需的依赖包，可以是单个包名或包名列表。
runtime_env	否	[Union[str, Mapping[str, Any]]]	函数所需的依赖包和安装方法，可以是JSON格式的字符串或者Python字典。
comment	否	str	函数的注释说明，可选。
**kwargs	否	-	其他可选的关键字参数。

返回值类型：

Callable： 注册后的函数。

delete_function

描述：从指定数据库中删除一个用户自定义函数。

语法格式：

```
delete_function(name, /, *, database=None, if_it_exists=False)
```

输入参数：

参数	是否必选	参数类型	描述
name	是	str	要删除的函数名称。
database	否	str	要删除函数的数据库名称。如果未指定，则使用默认数据库。
if_it_exists	否	bool	如果为True，则仅在函数存在时执行删除操作，避免因函数不存在而引发错误。

返回值类型：str。

describe_function

描述：描述指定数据库中的函数。

语法格式：

```
describe_function(name, /, *, database=None)
```

输入参数：

版权所有 © 华为云计算技术有限公司

参数	是否必选	参数类型	描述
name	是	str	函数名称。
database	否	str	函数所在的数据库名称。

返回值类型：str，函数的元数据信息。

get_function

描述：从指定或默认数据库中获取一个用户自定义函数。

语法格式：

```
get_function(name, *, database=None)
```

输入参数：

参数	是否必选	参数类型	描述
func_name	是	str	要获取的用户自定义函数的名称。
database	否	Optional[str]	要获取函数的数据库名称。如果未提供，则使用默认数据库。

返回值类型 ： Callable，可调用的函数。

list_functions

描述：列出指定数据库或默认数据库中的所有用户自定义函数。

语法格式：

```
list_functions(database=None)
```

输入参数：

参数	是否必选	参数类型	描述
database	否	str	要列出函数的数据库名称。如果未提供，则使用默认数据库。

返回值类型：pandas.DataFrame。

版权所有 © 华为云计算技术有限公司

2.1.5.11 AI APIs

ai_generate

描述： 使用大语言模型生成文本，并将生成结果作为新列添加到Dataset中。model_name需已注册。

语法格式:

```
ai_generate(prompt, on, as_col, model_name, model_parameters=None)
```

输入参数:

参数	是否必选	参数类型	描述
prompt	是	str	文本生成的提示词模板。输入列的内容将被追加到此提示词后。
on	是	ir.Column	包含待处理输入数据的Dataset列。
as_col	是	str	存储生成文本的新列名称。
model_name	是	str	用于文本生成的大语言模型名称。
model_parameters	否	str	大语言模型的额外参数，以JSON格式字符串提供。默认为None。

返回值类型： 包含新生成列的Dataset。

示例:

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*',
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_text", database='my_schema')
>>> ds
| id | text_content          |
| -- | --------------------- |
| int| string                |
| -- | --------------------- |
| 1  | You're a good man.    |
| 2  | You're a bad man.     |

# 使用带模型参数的AI生成
>>> ds.ai_generate(
...     prompt="Is this a positive review? Just reply with 'Yes' or 'No'.",
...     on=ds.text_content,
...     as_col="judgement",
...     model_name="gpt-4",
...     model_parameters='{"temperature": 0.7, "max_tokens": 100}'
... )
| id | text_content          | judgement        |
| -- | --------------------- | ---------------- |
| int| string                | string           |
| -- | --------------------- | ---------------- |
| 1  | You're a good man.    | Yes              |
| 2  | You're a bad man.     | No               |
```


版权所有 © 华为云计算技术有限公司

create_model

描述：创建一个抽象本地模型，用于绑定到远程LLM服务。

语法格式：

```
create_model(model_name, base_url, api_key, remote_model_name, description=None, database=None)
```

输入参数：

参数	是否必选	参数类型	描述
model_name	是	str	抽象本地模型的名称。
base_url	是	str	远程LLM服务的基础URL。
api_key	是	str	连接远程LLM服务所需的API密钥。
remote_model_name	是	str	远程LLM服务的模型名称。
description	否	str	可选参数，抽象本地模型的描述信息。
database	否	str	可选参数，模型所属的数据库名称。

返回值类型：无。

list_models

描述：查看指定数据库或默认数据库中的所有模型。

语法格式：

```
list_models(database=None)
```

输入参数：

参数	是否必选	参数类型	描述
database	否	str	要查看的模型所在的数据库名称。如果为None，则使用默认数据库。

返回值类型：list[tuple] ，为包含模型名称元组的列表。

drop_model

描述：删除与模型名称对应的抽象本地模型。

语法格式：

```
drop_model(model_name)
```


版权所有 © 华为云计算技术有限公司

输入参数：

参数	是否必选	参数类型	描述
model_name	是	str	抽象本地模型的名称。

返回值类型：无

describe_model

描述：查看指定的模型的详细信息。

语法格式：

```
describe_model(model_name, database=None)
```

输入参数：

参数	是否必选	参数类型	描述
model_name	是	str	要查看的模型名称。
database	否	str	预留参数，模型所在的数据库名称，默认为None。

返回值类型：查看的模型的详细信息。

2.1.5.12 用户自定义函数 UDF

2.1.5.12.1 UDF 概述

用户自定义函数（User-Defined Function，简称UDF）是指由用户根据特定需求自定义的函数，用于扩展数据库或数据处理系统的功能。UDF可以执行各种操作，如数据转换、复杂计算、数据聚合等，这些操作可能超出了系统提供的内置函数的能力。

UDF的主要用途包括：

- 扩展功能：UDF允许用户扩展数据库或数据处理系统的功能，执行内置函数不支持的操作。
- 复杂计算：UDF可用于执行复杂的数学或逻辑计算，这些计算可能涉及多个步骤和条件。
- 数据转换：UDF可用于数据清洗和转换，如格式化日期、转换字符串或编码数据。
- 业务逻辑封装：UDF可将业务逻辑封装在函数中，使得数据处理更加模块化和可重用。
- 性能优化：在某些情况下，通过UDF将复杂的计算逻辑移到数据库服务器上执行，可以减少数据传输和提高性能。

版权所有 © 华为云计算技术有限公司

说明

在大数据处理中，如果需要在DataFrame上执行自定义函数UDF来处理复杂逻辑或调用第三方库，但执行UDF的容器或集群只预装了基础Python库，您可以在UDF中使用pip install package命令安装最新版本的依赖，从而灵活使用各种Python库。如果您需要安装指定版本的依赖，需加具体版本号，示例命令为pip install pandas==2.2.2。

约束限制

- UDF编程语言仅支持Python。
- UDF的参数不支持设置默认值，如果创建函数时参数或返回值带有精度，不进行精度检测。
- 不支持函数重载。

UDF 类型

在Aura DataFrame SDK中支持了Scalar UDF、Class UDF、Vectorized UDF、UDTF、UDAF等不同类别UDF的注册和使用。

表 2-24 UDF 类型介绍

类型	输入	输出	描述
Scalar UDF（标量UDF）	单行输入，多参数，例如(a, b, c)	单行输出，一个标量，例如int/float/string/date	最常见形式，每次被调用时只处理一行数据。可以做简单计算、格式化、清洗、转换。例如：字符串清洗、数学计算、日期转换。类比Python的普通函数。
Class UDF（类UDF）	与Scalar UDF相同：单行输入	单行输出	是对Scalar UDF的OOP封装方式，适合有内部缓存、初始化参数、模型对象等情况。例如需要在函数内部加载一次模型、正则、配置。数据库只是在内部初始化一次，不需要每次调用都构建资源。
Vectorized UDF（向量化UDF）	一次性输入一批行，类型为pandas.Series/pyarrow.ChunkedArray	一次性输出同等数量行的向量化结构	性能最优的单行转换方式，用于批量处理。适合：向量化数学计算、文本批量处理、embedding处理、NLP分词。具有SIMD/批量执行优势，常用于Arrow、Pandas。
UDTF（User Defined Table Function，表函数）	单行输入	多行输出	一个输入可能产出多个输出行。类似Python的yield。用于拆分、展开、解包结构化数据。例如：拆分JSON数组、展开多模态数据、音频/视频分片、tokenize、爆炸式展开。

版权所有 © 华为云计算技术有限公司

类型	输入	输出	描述
UDAF（User Defined Aggregate Function，聚合函数）	多行输入，可能来自多个分区	单个输出	对一组数据做聚合的函数：累加、统计、聚类、embedding聚合。具备accumulate、merge、finish生命周期，支持分布式执行。

UDF 注册方式

注册UDF总体上来说分为两种方式：显式注册和隐式注册。

表 2-25 注册 UDF 方式

注册方式	含义	是否依赖会话对象	是否侵入式添加注册逻辑	适用场景	参考
显式注册	代码中明确指定UDF的注册信息	是	是	如果用户希望明确控制注册时间，允许侵入式添加注册逻辑，或对同一个Backend连接下的Scalar UDF注册和使用分离有要求。	**UDF显式注册语法**
隐式注册	运行时自动发现并注册UDF	否	否	如果用户希望无侵入式地注册Scalar UDF，且对同一个Backend连接下的Scalar UDF注册和使用分离无要求。	**UDF隐式注册语法**

UDF 注册类型

不管是显式注册还是隐式注册，对于不同的UDF注册类型，其含义有所不同，详情参见下表：

表 2-26 UDF 注册类型的含义

UDF注册类型	含义	是否向量化	适用场景与特点	参考
Python	将一个原始的Python函数或者类注册进数据库中。	否	逐行处理数据，适用于简单或特定的计算，但性能较低。	**Python/Pyarrow/Pandas UDF注册参数**

版权所有 © 华为云计算技术有限公司

UDF注册类型	含义	是否向量化	适用场景与特点	参考
Builtin	获得数据库已存在的函数的句柄，无实际注册的操作。	否	直接调用数据库后端已存在的函数，适用于利用数据库原生功能的场景。	**Builtin UDF注册参数**
PyArrow	将一个接受pyarrow.ChunkedArray作为输入和输出的Python函数或类注册进数据库中。	是	利用PyArrow的高性能计算能力，适用于需要处理大型数据集或进行高效计算的场景。	**Python/Pyarrow/Pandas UDF注册参数**
Pandas	将一个接受pandas.Series作为输入和输出的Python函数或类注册进数据库中。	是	利用Pandas的矢量化操作，适用于需要在Python层进行复杂数据处理的场景。	**Python/Pyarrow/Pandas UDF注册参数**

对于Scalar UDF，目前支持全部上述4种注册类型。

对于UDAF、UDTF，目前仅支持上述的python、builtin这2种注册类型。

UDF 运行时的额外参数

不管是显式注册还是隐式注册，对于所有的UDF类型，都支持在使用时通过with_arguments方法传入额外参数。这些参数包括了concurrency、min_concurrency、max_concurrency、timeout、dpu、apu，可以为UDF的执行提供细粒度的资源分配、并发度控制、时间上限等。

具体用法见**2.1.5.12.10 WITH ARGUMENTS语法**。

2.1.5.12.2 开发注意事项

运行环境约束

- UDF编程语言仅支持Python。
- UDF运行时环境的Python版本支持3.10和3.11系列，建议使用3.11.9版本。
- 使用CLOUDPICKLE序列化主函数时，确保本地Python版本为3.11.9且cloudpickle版本为 3.0.0，以避免Aura运行环境里因版本不一致导致函数体解析失败。

函数定义约束

- UDF 的参数不支持设置默认值；如果创建函数时参数或返回值带有精度，不进行精度检测。

版权所有 © 华为云计算技术有限公司

- 不支持函数重载。

函数与代码包要求

- 函数体和代码包归档路径，两者必须定义一个。
- 当业务场景比较复杂、UDF所涉及的代码较多时，推荐以文件压缩包形式注册函数：将UDF依赖的所有相关代码文件统一归档到一个压缩包，上传至OBS后，在创建函数时指定压缩包的存储路径。压缩包要求如下：
  - 必须包含文件 {FUNCTION_NAME}_source.py，该文件被视为函数的主入口文件，且内容需使用CLOUDPICKLE序列化为十六进制。
  - 主函数需要import的模块所属的代码文件，需要正确组织结构。
  - 压缩包格式仅支持zip。

    压缩包示例如下：

```
my_udf.zip/
├── my_udf_source.py
├── my_helper.py
└── other/
    └── utils.py
```

注册方式的建议

Aura提供了三种注册UDF的方式：**ibis-aura SDK显式注册、隐式注册**以及**SQL的DDL注册**，推荐用户使用ibis-aura SDK显式注册，该方式有以下两个优势：

- SDK提供的注册接口已完全遵守Aura DDL的使用规范，只要正确调用接口即可，能避免违反约束规范而导致UDF不能正常使用的问题。
- SDK提供了更多的约束检查，例如Python版本的检测，自动整理UDF代码包等。

依赖管理

- Python UDF 的三方依赖包列表需要保证lib间的兼容性，并建议所有依赖包指定稳定的版本。
- 在本地调试UDF时，建议使用华为源所提供的三方包搭建调试环境，以避免UDF代码在本地环境可运行，而在Aura里运行时出现版本无法安装、版本不兼容等问题。

存储与权限

用户只能使用OBS并行文件系统的桶，来存储UDF代码压缩包，且需要在LakeFormation上赋予IAM用户读的权限。

2.1.5.12.3 数据类型映射

Python和Aura数据类型的映射关系如下表所示：

表 2-27 Python 和 Aura 数据类型的映射关系

Python类型	Aura类型
int	BIGINT
bool	BOOLEAN

版权所有 © 华为云计算技术有限公司

Python类型	Aura类型
float	DOUBLE PRECISION
str	TEXT
bytes	BYTEA
datetime.date	DATE
datetime.time	TIME
datetime.datetime	TIMESTAMP
decimal.Decimal	NUMERIC

2.1.5.12.4 环境变量设置

env_vars允许在UDF调用时动态注入环境变量到Actor进程，无需为不同环境变量配置创建冗余UDF定义。

基本用法

以下代码片段仅展示env_vars参数的用法：

```
# Dataset API — 传入dict
ds = ds.map(
    fn="my_schema.data_loader",
    on=["url_col"],
    env_vars={"DATA_SOURCE": "obs://bucket/path", "REGION": "cn-north-4"},
)

# with_arguments — 传入dict或JSON字符串
result = my_udf(t.col_a).with_arguments(
    env_vars={"DATA_SOURCE": "obs://bucket/path"},
)
```

在UDF中通过os.environ访问：

```
def my_udf(url: str) -> str:
    import os
    data_source = os.environ.get("DATA_SOURCE", "")
    return load_from(url, base=data_source)
```

类型校验

Python侧在with_arguments() 中执行严格类型校验：

输入	行为
env_vars={"KEY": "VALUE"}	正常，key/value 均为 str
env_vars={1: "val"}	TypeError: keys must be str
env_vars={"key": 123}	TypeError: values must be str
env_vars={}	等同未设置，不生成 ENV_VARS 子句

版权所有 © 华为云计算技术有限公司

输入	行为
env_vars='{"K":"V"}'	JSON 字符串直通
env_vars='not json'	TypeError: must be valid JSON
env_vars=None	默认值，不生成 ENV_VARS 子句

Key 格式约束

env_vars的key必须符合POSIX规范^[a-zA-Z_][a-zA-Z0-9_]*$，最大长度1022字节。内核侧校验不合法时返回ERROR。

Key 示例	合法	说明
MY_KEY	是	标准POSIX key
_UNDERSCORE_KEY	是	以下划线开头
KEY123	是	包含数字但非首字符
1KEY	否	以数字开头
MY-KEY	否	包含连字符
MY.KEY	否	包含点号
MY=KEY	否	包含等号
my key	否	包含空格

安全注意事项

禁止在env_vars中传入AK/SK等敏感凭证

```
# 禁止 — 凭证明文经过 SQL 传输，有泄漏风险
my_udf(t.col).with_arguments(
    env_vars={"ACCESS_KEY_ID": "xxx", "SECRET_ACCESS_KEY": "yyy"}
)

# 禁止 — UDF_TEMP_AK/UDF_TEMP_SK是服务端凭证保留 key，用户设置会触发ERROR
my_udf(t.col).with_arguments(
    env_vars={"UDF_TEMP_AK": "xxx", "UDF_TEMP_SK": "yyy"}
)
# ERROR: env_vars key "UDF_TEMP_AK" conflicts with reserved server credentials, UDF 调用终止
```

使用内核IAM委托Token自动注入

内核通过IAM委托Token自动获取临时AK/SK，注入为UDF_TEMP_AK、UDF_TEMP_SK、UDF_TEMP_SECURITY_TOKEN环境变量。OBS Endpoint等非凭证配置通过用户env_vars传入。UDF内部直接通过os.environ读取，配合esdk-obs-python SDK使用：

```
class ObsDataReader:
    """Read data from OBS.

    Credentials (UDF_TEMP_AK/SK/TOKEN) are automatically injected by
    the kernel via IAM delegation token — no client-side setup needed.
```


版权所有 © 华为云计算技术有限公司

```
    UDF_OBS_ENDPOINT must be provided by the user via env_vars.
    """

    def __init__(self, bucket: str):
        self.bucket = bucket

    def _get_obs_config(self) -> tuple:
        """Read OBS config from env vars."""
        import os

        # Credentials — auto-injected by kernel, do NOT set these yourself
        ak = os.environ.get("UDF_TEMP_AK", "")
        sk = os.environ.get("UDF_TEMP_SK", "")
        token = os.environ.get("UDF_TEMP_SECURITY_TOKEN", "")

        # Endpoint — user-provided via env_vars
        endpoint = os.environ.get("UDF_OBS_ENDPOINT", "")

        if not ak or not sk:
            raise RuntimeError(
                "OBS credentials not available. "
                "Ensure IAM delegation token is configured on the server."
            )
        if not endpoint:
            raise RuntimeError(
                "OBS endpoint not configured. "
                "Pass it via env_vars: with_arguments(env_vars={'UDF_OBS_ENDPOINT': '...'})"
            )

        return ak, sk, token, endpoint

    def __call__(self, object_key: str) -> str:
        """Read an object from OBS and return its content as string."""
        from obs import ObsClient

        ak, sk, token, endpoint = self._get_obs_config()

        obs_client = ObsClient(
            access_key_id=ak,
            secret_access_key=sk,
            security_token=token,
            server=endpoint,
        )

        try:
            resp = obs_client.getObject(
                self.bucket, object_key, loadStreamInMemory=True
            )
            if resp.status >= 300:
                raise RuntimeError(
                    f"OBS read failed: status={resp.status}, reason={resp.reason}"
                )
            return resp.body.buffer.decode("utf-8")
        finally:
            obs_client.close()
```

使用方式：

```
import pandas as pd
from aura_frame.multimodal import ai_lake

con = ai_lake.connect(
    aura_endpoint="*",
    aura_workspace_id="*",
    lf_catalog_name="*",
    access_key="*",
    secret_key="*",
    default_database="my_schema",
    use_single_cn_mode=True,
)
# Register the UDFs
```


版权所有 © 华为云计算技术有限公司

```
try:
    con.create_scalar_function(ObsDataReader, name="obs_reader", database="my_schema")
    # --- Direct expression API ---
    t = con.from_pandas(pd.DataFrame({"object_key": [""]}))._table
    result = con.get_function("obs_reader", database="my_schema")(
        t.object_key
    ).with_arguments(
        bucket="my-data-bucket",
        env_vars={"UDF_OBS_ENDPOINT": "obs.cn-north-4.myhuaweicloud.com"},
    )
    # --- Dataset API ---
    ds = con.from_pandas(pd.DataFrame({"object_key": [""]}))
    ds = ds.map(
        fn="my_schema.obs_reader",
        on=["object_key"],
        bucket="my-data-bucket",
        as_col="content",
        env_vars={"UDF_OBS_ENDPOINT": "obs.cn-north-4.myhuaweicloud.com"},
    )
    ds.show()
finally:
    con.close()
```

服务端凭证保留key（用户env_vars中设置会触发ERROR）：

保留 key	用途
UDF_TEMP_AK	临时Access Key。
UDF_TEMP_SK	临时Secret Key。
UDF_TEMP_SECURITY_TOKEN	临时Security Token。
SHARED_WORK_DIR	会话级共享存储的路径，同一个session内的所有UDF实例都可以访问该路径，进行读写。

这些保留key由内核自动注入，用户不能手动设置。

风险说明：

- 前端env_vars以明文JSON通过SQL传输。
- 可能被审计日志、慢查询日志、错误日志捕获。
- 预置环境变量与用户key冲突时，内核返回ERROR并终止UDF调用。

覆盖系统变量风险：覆盖PATH、LD_PRELOAD、PYTHONPATH、HOME、PYTHONHOME 等可能导致非预期行为。

2.1.5.12.5 Scalar UDF

Scalar UDF是最基本的行处理函数， 它接受零个或多个输入参数，并对这一行数据进行操作，最终返回一行结果。

通常Scalar UDF适用于数学运算、复杂类型转换和自定义格式化等行处理的场景。

Scalar UDF的行为与内置函数非常相似，但其具体实现由用户自定义。

开发流程

开发Python Scalar UDF，总体流程如下：

版权所有 © 华为云计算技术有限公司

步骤1 根据用户的业务编写和调试Python代码。

代码示例如下：

```
import numpy as np

def pysum(arg1: int, arg2: int) -> int:
    printLog(arg1, arg2)
    return np.sum(np.array([arg1, arg2]))
```

步骤2 注册Python UDF。

使用ibis-aura创建会话。

```
import aura_frame as aura
from aura_frame.multimodal import ai_lake

target_database = "test"  #在LakeFormation中创建数据库，例如：test。具体操作，请参见创建LakeFormation实例并规划元数据
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),         # AuraJobV2端点所在的终端节点（Endpoint）
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2端点ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2端点名称
    aura_workspace_id=os.getenv("aura_workspace_id"), # 工作空间ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog名称
    access_key=os.getenv("access_key"),                # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),                # 访问密钥（SK）
    default_database=target_database,                  # LakeFormation数据库名称
    use_single_cn_mode=True,   # 是否开启单CN模式
    logging_level=logging.INFO # 设置日志级别
)
```

注册Python UDF。

```
try:
    udf = con.create_scalar_function(pysum, packages=["numpy"])
finally:
    con.close()
```

步骤3 查看Python UDF。

Aura提供show functions查看特定schema下的所有函数，且提供了describe function function_name方法查看特定函数的详细信息，具体的使用方式请参考**SHOW**和**DESCRIBE**。

步骤4 调用Python UDF。

使用Aura任意客户端，下发查询即可，例如：

```
select "test".pysum(3,4);
```

步骤5 查看Python UDF性能指标及调优。

Aura支持了对Python UDF运行时的性能监控，以及提供了接口，可配置UDF运行时的资源规格和并发度，具体使用方式请参考**2.1.5.12.14 Python UDF性能调优**。

----结束

示例

```
import os
import ibis
from aura_frame.multimodal import ai_lake
target_database = "test" #在LakeFormation中创建数据库，例如：test。具体操作，请参见创建LakeFormation实例并规划元数据
#连接AuraJobV2端点，具体参数说明，请参见2.1.5.4.1 建立与AuraJobV2端点的连接。
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),         # AuraJobV2端点所在的终端节点（Endpoint）
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2端点ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2端点名称
```


版权所有 © 华为云计算技术有限公司

```
    aura_workspace_id=os.getenv("aura_workspace_id"),  # 工作空间ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog名称
    access_key=os.getenv("access_key"),                # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),                # 访问密钥（SK）
    default_database=target_database,                  # 默认数据库
    use_single_cn_mode=True,                           # 是否开启单CN模式
)
#配置UDF部署工作区，具体参数说明，请参见2.1.5.4.2 配置UDF部署工作区。
con.set_function_staging_workspace(
    obs_directory_base=os.getenv("obs_directory_base"),  # 并行文件系统中UDF的存储路径
    obs_bucket_name=os.getenv("obs_bucket_name"),        # 并行文件系统名称
    obs_server=os.getenv("obs_server"),                  # OBS终端节点（Endpoint）
    access_key=os.getenv("access_key"),                  # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),                  # 访问密钥（SK）
)
# 定义 Scalar UDF：两数求和
def add_udf(lhs: float, rhs: int) -> float:
    return lhs + rhs
try:
    # 创建测试表并写入数据
    table_name = "product_orders"
    table_schema = ibis.schema({"price": "float64", "quantity": "int32"})
    con.sql(f'DROP TABLE IF EXISTS "{target_database}"."{table_name}";')
    con.create_table(
        table_name,
        schema=table_schema,
        database=target_database,
        table_format="orc",
        if_not_exists=True,
    )
    con.sql(f'''
    INSERT INTO "{target_database}"."{table_name}" (price, quantity) VALUES
        (100.0, 10),
        (200.0, 20),
        (150.0, 15)
    ''')
    # 删除已存在的同名UDF（支持重复执行）
    con.delete_function("add_udf", database=target_database, if_it_exists=True)
    # 注册 Scalar UDF
    udf = con.create_scalar_function(
        add_udf,
        database=target_database,
        comment="To test create_scalar_function",
    )
    # 使用 UDF 处理 product_orders 表数据
    ds = con.load_dataset("product_orders", database=target_database)
    ds = ds.map(fn=udf, on=[ds.price, ds.quantity], as_col="sum_column")
    ds = ds.select_columns(ds.price, ds.quantity, ds.sum_column)
    print(ds.execute())
finally:
    con.close()
```

2.1.5.12.6 Class UDF

Class UDF基于面向对象编程范式实现，主要目的是为了创建有状态的处理逻辑。函数要求实现__init__、__call__等方法。

Class UDF 注册

Class UDF注册传入的是Python类，对于__init__、__call__、 __del__ 3个实例方法有特殊的含义认定，详情请参见下表；其他的Python类和实例方法不做限制，用户可以任意添加。

版权所有 © 华为云计算技术有限公司

表 2-28 注册 Python Class 时的特殊实例方法

Python Class方法	是否必须	参数	含义	适用场景
__init__(self, *args, **kwargs)	否	通过with_arguments方法传入参数，只允许传入标量Scalar。	UDF的构造方法。	初始化UDF属性（如保存参数、打开文件、建立连接等）。
__call__(self, *args, **kwargs)	是	通过UDF算子传入参数，可以传入标量Scalar和列名Column。	UDF的业务逻辑方法入口。	UDF处理数据、执行计算。
__del__(self)	否	不支持传入参数。	UDF的析构方法。	UDF资源清理（如关闭文件、断开网络连接等）。

注意事项：__init__的生命周期是每个batch调用一次，即Aura内核每处理一个batch，都会重新初始化一次变量。

示例

本示例定义一个带初始化参数的Class UDF，在__init__中接收姓名参数，在__call__中结合城市名生成欢迎语，将其注册到数据库后通过map方法对city_list表的每行数据生成问候消息。

1. 定义Class UDF类Hello，在__init__中接收firstname和lastname参数，在__call__中结合城市名生成欢迎语。
2. 连接Aura服务并设置UDF工作空间。
3. 创建city_list表（包含city列），并插入5条城市数据。
4. 删除已存在的同名UDF（支持重复执行）。
5. 注册Class UDF到目标数据库。
6. 加载city_list数据集，通过map方法调用UDF对city列生成问候语，传入firstname和lastname参数。
7. 选择city和new_column列并执行查询打印结果。
8. 关闭Aura连接。

```
import os
import ibis
from aura_frame.multimodal import ai_lake

target_database = "test"  #在LakeFormation中创建数据库，例如：test。具体操作，请参见创建LakeFormation实例并规划元数据
#连接AuraJobV2端点，具体参数说明，请参见2.1.5.4.1 建立与AuraJobV2端点的连接。
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),         # AuraJobV2端点所在的终端节点（Endpoint）
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2端点ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2端点名称
    aura_workspace_id=os.getenv("aura_workspace_id"), # 工作空间ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog名称
    access_key=os.getenv("access_key"),               # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),               # 访问密钥（SK）
    default_database="postgres",                      # LakeFormation数据库名称
    use_single_cn_mode=True,   # 是否开启单CN模式
    logging_level=logging.INFO # 设置日志级别
```


版权所有 © 华为云计算技术有限公司

```
)
#配置UDF部署工作区，具体参数说明，请参见2.1.5.4.2 配置UDF部署工作区。
con.set_function_staging_workspace(
    obs_directory_base=os.getenv("obs_directory_base"), # 并行文件系统中UDF的存储路径
    obs_bucket_name=os.getenv("obs_bucket_name"),       # 并行文件系统名称
    obs_server=os.getenv("obs_server"),                 # OBS终端节点（Endpoint）
    access_key=os.getenv("access_key"),                 # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),                 # 访问密钥（SK）
)
# 定义 Class UDF：带初始化参数的问候生成器
class Hello:
    def __init__(self, firstname: str, lastname: str):
        self.firstname = firstname
        self.lastname = lastname
    def __call__(self, location: str) -> str:
        return f"Hello {self.firstname} {self.lastname}! Welcome to {location}"

try:
    # 创建测试表并写入数据
    table_name = "city_list"
    table_schema = ibis.schema({"city": "string"})
    con.sql(f'DROP TABLE IF EXISTS "{target_database}"."{table_name}";')
    con.create_table(
        table_name,
        schema=table_schema,
        database=target_database,
        table_format="orc",
        if_not_exists=True,
    )
    con.sql(f'''
    INSERT INTO "{target_database}"."{table_name}" (city) VALUES
        ('Beijing'),
        ('Shanghai'),
        ('New York'),
        ('London'),
        ('Tokyo')
    ''')
    # 删除已存在的同名UDF（支持重复执行）
    con.delete_function("hello", database=target_database, if_it_exists=True)
    # 注册 Class UDF
    hello_udf = con.create_scalar_function(
        Hello,
        name="hello",
        database=target_database,
    )
    # 使用 UDF 处理 city_list 表数据
    ds = con.load_dataset("city_list", database=target_database)
    ds = ds.map(
        fn=hello_udf,
        on=[ds.city],
        as_col="new_column",
        firstname="Ethan",
        lastname="Carter",
    )
    ds = ds.select_columns(ds.city, ds.new_column)
    print(ds.execute())
finally:
    con.close()
```

2.1.5.12.7 Vectorized UDF

Vectorized UDF是向量化执行的函数，是为解决传统行式UDF性能瓶颈而设计的高效函数。入参、出参通常为PyArrow或者Pandas类型。

示例

下文提供两个示例展示如何使用Vectorized UDF。

版权所有 © 华为云计算技术有限公司

- 示例一：使用PyArrow进行向量化加速。通过 `con.create_scalar_function` 显式注册PyArrow向量化UDF，框架会根据类型注解自动识别为PyArrow类型，再通过`ds.map`调用。

```
import os
import ibis
import aura_frame as aura
from aura_frame.multimodal import ai_lake
from aura_frame.multimodal.types.vectorized import PyarrowVector
import pyarrow.compute as pc

target_database = "test"  #在LakeFormation中创建数据库，例如：test。具体操作，请参见创建LakeFormation实例并规划元数据
#连接AuraJobV2端点，具体参数说明，请参见2.1.5.4.1 建立与AuraJobV2端点的连接。
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),         # AuraJobV2端点所在的终端节点（Endpoint）
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2端点ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2端点名称
    aura_workspace_id=os.getenv("aura_workspace_id"),  # 工作空间ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog名称
    access_key=os.getenv("access_key"),                # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),                # 访问密钥（SK）
    default_database="postgres",                       # LakeFormation数据库名称
    use_single_cn_mode=True,   # 是否开启单CN模式
    logging_level=logging.INFO # 设置日志级别
)
#配置UDF部署工作区，具体参数说明，请参见2.1.5.4.2 配置UDF部署工作区。
con.set_function_staging_workspace(
    obs_directory_base=os.getenv("obs_directory_base"),  # 并行文件系统中UDF的存储路径
    obs_bucket_name=os.getenv("obs_bucket_name"),        # 并行文件系统名称
    obs_server=os.getenv("obs_server"),                  # OBS终端节点（Endpoint）
    access_key=os.getenv("access_key"),                  # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),                  # 访问密钥（SK）
)
def calculate_product_pyarrow(
    prices: PyarrowVector[float],
    quantities: PyarrowVector[int],
) -> PyarrowVector[float]:
    return PyarrowVector[float](pc.multiply(prices, quantities))

try:
    # 创建测试表并写入数据
    table_name = "sales_records"
    table_schema = ibis.schema({"price": "float64", "quantity": "int32"})
    con.sql(f'DROP TABLE IF EXISTS "{target_database}"."{table_name}";')
    con.create_table(
        table_name,
        schema=table_schema,
        database=target_database,
        table_format="orc",
        if_not_exists=True,
    )
    con.sql(f'''
    INSERT INTO "{target_database}"."{table_name}" (price, quantity) VALUES
        (100.0, 10),
        (200.0, 20),
        (150.0, 15)
    ''')
    # 删除已存在的同名UDF（支持重复执行）
    con.delete_function("calculate_product_pyarrow", database=target_database, if_it_exists=True)
    # 显式注册 PyArrow 向量化 UDF
    udf_pyarrow = con.create_scalar_function(
        calculate_product_pyarrow,
        name="calculate_product_pyarrow",
        database=target_database,
        register_type=aura.udf.RegisterType.STAGED,
    )
    ds = con.load_dataset("sales_records", database=target_database)
    ds = ds.map(fn=udf_pyarrow, on=[ds.price, ds.quantity], as_col="product_column")
    ds = ds.select_columns(ds.price, ds.quantity, ds.product_column)
```


版权所有 © 华为云计算技术有限公司

```
    print("PyArrow 向量化结果:")
    print(ds.execute())
finally:
    con.close()
```

- 示例二：使用Pandas进行向量化加速。通过`@aura.udf.pandas`装饰器隐式注册Pandas向量化UDF，注册类型为`STAGED`。

```
import os
import ibis
import aura_frame as aura
from aura_frame.multimodal import ai_lake
from aura_frame.multimodal.types.vectorized import PandasVector
import pandas as pd

target_database = "test"
# 隐式注册UDF
@aura.udf.pandas(database=target_database, register_type=aura.udf.RegisterType.STAGED)
def calculate_product_pandas(
    prices: PandasVector[float],
    quantities: PandasVector[int],
) -> PandasVector[float]:
    return PandasVector[float](prices * quantities, dtype=pd.Float64Dtype())
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),         # AuraJobV2端点所在的终端节点（Endpoint）
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2端点ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2端点名称
    aura_workspace_id=os.getenv("aura_workspace_id"),  # 工作空间ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog名称
    lf_instance_id=os.getenv("lf_instance_id"),        # LakeFormation实例ID
    project_id=os.getenv("project_id"),                # 项目ID
    access_key=os.getenv("access_key"),                # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),                # 访问密钥（SK）
    default_database=target_database,                  # 默认数据库
    use_single_cn_mode=True,                           # 是否开启单CN模式
)
con.set_function_staging_workspace(
    obs_directory_base=os.getenv("obs_directory_base"),  # 并行文件系统中UDF的存储路径
    obs_bucket_name=os.getenv("obs_bucket_name"),        # 并行文件系统名称
    obs_server=os.getenv("obs_server"),                  # OBS终端节点（Endpoint）
    access_key=os.getenv("access_key"),                  # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),                  # 访问密钥（SK）
)
try:
    # 创建测试表并写入数据
    table_name = "sales_records"
    table_schema = ibis.schema({"price": "float64", "quantity": "int32"})
    con.sql(f'DROP TABLE IF EXISTS "{target_database}"."{table_name}";')
    con.create_table(
        table_name,
        schema=table_schema,
        database=target_database,
        table_format="orc",
        if_not_exists=True,
    )
    con.sql(f'''
    INSERT INTO "{target_database}"."{table_name}" (price, quantity) VALUES
        (100.0, 10),
        (200.0, 20),
        (150.0, 15)
    ''')
    # 删除已存在的同名UDF（支持重复执行）
    con.delete_function("calculate_product_pandas", database=target_database, if_it_exists=True)
    # 使用 Pandas 向量化 UDF（已在上方通过装饰器隐式注册）
    ds2 = con.load_dataset("sales_records", database=target_database)
    ds2 = ds2.map(fn=calculate_product_pandas, on=[ds2.price, ds2.quantity],
as_col="product_column")
    ds2 = ds2.select_columns(ds2.price, ds2.quantity, ds2.product_column)
    print("Pandas 向量化结果:")
    print(ds2.execute())
```


版权所有 © 华为云计算技术有限公司

```
finally:
    con.close()
```

2.1.5.12.8 UDTF

UDTF是输入一行返回多行的函数，适用于典型的多模态数据处理场景，包括视频/音频切帧，样本生成等。

UDTF整体开发流程可参考Scalar UDF，需要注意以下几点：

- Handler目前仅支持返回迭代器，推荐使用yield。
- UDTF不支持开启并发和向量化计算。
- 当Handler出现死循环，无限输出结果时，Aura会根据超时时间自动关闭程序，默认超时时间为1000s。

示例

以下示例展示UDTF的注册和使用的流程。示例使用sequence_config表（包含start_val、end_val、step_val三个int32列），通过UDTF函数py_generate_series为每一行生成从start_val到end_val、步长为step_val的整数序列。

```
import os
import ibis
import aura_frame as aura
from aura_frame.multimodal import ai_lake
import ibis.expr.datatypes as dt

target_database = "test" #在LakeFormation中创建数据库，例如：test。具体操作，请参见创建LakeFormation实例并规划元数据
#连接AuraJobV2端点，具体参数说明，请参见2.1.5.4.1 建立与AuraJobV2端点的连接。
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),        # AuraJobV2端点所在的终端节点（Endpoint）
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2端点ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2端点名称
    aura_workspace_id=os.getenv("aura_workspace_id"),  # 工作空间ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog名称
    access_key=os.getenv("access_key"),                # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),                # 访问密钥（SK）
    default_database="postgres",                       # LakeFormation数据库名称
    use_single_cn_mode=True,   # 是否开启单CN模式
    logging_level=logging.INFO # 设置日志级别
)
#配置UDF部署工作区，具体参数说明，请参见2.1.5.4.2 配置UDF部署工作区。
con.set_function_staging_workspace(
    obs_directory_base=os.getenv("obs_directory_base"),    # 并行文件系统中UDF的存储路径
    obs_bucket_name=os.getenv("obs_bucket_name"),          # 并行文件系统名称
    obs_server=os.getenv("obs_server"),                    # OBS终端节点（Endpoint）
    access_key=os.getenv("access_key"),                    # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),                    # 访问密钥（SK）
)

# 定义 UDTF：输入 start、endnum、step，生成整数序列
def py_generate_series(start: int, endnum: int, step: int):
    current = start
    while current <= endnum:
        yield {"col1": current}
        current += step

try:
    # 创建测试表并写入数据
    table_name = "sequence_config"
    table_schema = ibis.schema({"start_val": "int32", "end_val": "int32", "step_val": "int32"})
    con.sql(f'DROP TABLE IF EXISTS "{target_database}"."{table_name}";')
    con.create_table(
        table_name,
        schema=table_schema,
```


版权所有 © 华为云计算技术有限公司

```
        database=target_database,
        table_format="orc",
        if_not_exists=True,
    )
    con.sql(f'''
    INSERT INTO "{target_database}"."{table_name}" (start_val, end_val, step_val) VALUES
      (1, 5, 1),
      (10, 20, 5),
      (0, 10, 2)
    ''')
    # 删除已存在的同名UDTF（支持重复执行）
    con.delete_function("py_generate_series", database=target_database, if_it_exists=True)
    func_signature = aura.Signature(
        parameters=[
            aura.Parameter(name="start", annotation=int),
            aura.Parameter(name="endnum", annotation=int),
            aura.Parameter(name="step", annotation=int),
        ],
        return_annotation=dt.Struct({"col1": int}),
    )
    con.create_table_function(py_generate_series, database=target_database, signature=func_signature)
    ds = con.load_dataset("sequence_config", database=target_database)
    py_generate_series_handler = con.get_function("py_generate_series", database=target_database)
    ds = ds.flat_map(
        fn=py_generate_series_handler,
        on=[ds.start_val, ds.end_val, ds.step_val],
        as_col="udtf_col",
    )
    res = ds.execute()
    print(res)
finally:
    con.close()
```

2.1.5.12.9 UDAF

UDAF是输入多行返回一组聚合值的函数，适用于典型的多模态数据处理场景，包括视频/音频/图像的统计特征聚合，多模态标签总结等。

UDAF整体开发流程可以参考Scalar UDF，需要注意以下几点：

- Handler指定的必须是Python Class，且Python Class中必须包括accumulate、aggregate_state、merge、finish 4个方法。UDAF要求用户实现的accumulate、aggregate_state、merge方法是无副作用的（side-effect free），即这些方法的返回结果只取决于输入参数和聚合状态本身，而不应依赖于方法被调用的次数、顺序或外部环境。
- UDAF的finish方法不论是单CN模式，还是多DN模式执行，对于每个分组（由GROUP BY从句指定）都只会在全局调用1次。
- UDAF要求用户在定义非STRICT时，能够在accumulate方法中正确处理NULL值输入，即Python中的None值。
- UDAF不支持作为窗口函数（window function）使用，即不支持在OVER从句中使用UDAF。
- UDAF不支持开启并发和向量化计算。

UDAF 注册

UDAF注册传入的必须是Python类，对于__init__、aggregate_state、accumulate、merge、finish、 __del__ 6个实例方法有特殊的含义认定，详情请参见下表；其他的Python类和实例方法不做限制，用户可以任意添加。

版权所有 © 华为云计算技术有限公司

表 2-29 注册 UDAF 时的特殊实例方法

Python Class方法	是否必须	参数	含义	适用场景
__init__(self, *args, **kwargs)	否	通过with_arguments方法传入参数，只允许传入标量Scalar。	UDAF的构造方法。	初始化UDAF属性（如保存参数、打开文件、建立连接等）。
aggregate_state(self)	是	无。	可序列化的聚合中间态，用于跨分区传递。	accumulate与merge间共享状态。
accumulate(self, *args, **kwargs)	是	通过UDAF算子传入参数，可以传入标量Scalar和列名Column。	增量更新聚合中间态。	正常数据扫过阶段。
merge(self, other_state)	是	来自其他分区的中间态。	合并其他分区状态到当前中间态。	分布式reduce阶段，多分区聚合结果归并。
finish(self)	是	无。	输出最终聚合结果。	聚合收尾。
__del__(self)	否	不支持传入参数。	UDAF的析构方法。	UDAF资源清理（如关闭文件、断开网络连接等）。

示例

```
import os
import json
import datetime as pydt
import ibis
import ibis.expr.datatypes as dt
import pandas as pd
from typing import Any, Dict
import aura_frame as aura
from aura_frame.multimodal import ai_lake
from aura_frame.multimodal.function import AggregateFnBuilder

target_database = "test"  #在LakeFormation中创建数据库，例如：test。具体操作，请参见创建LakeFormation实例并规划元数据
#连接AuraJobV2端点，具体参数说明，请参见2.1.5.4.1 建立与AuraJobV2端点的连接。
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),        # AuraJobV2端点所在的终端节点（Endpoint）
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2端点ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2端点名称
    aura_workspace_id=os.getenv("aura_workspace_id"),  # 工作空间ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog名称
    access_key=os.getenv("access_key"),                # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),                # 访问密钥（SK）
    default_database="postgres",                       # LakeFormation数据库名称
    use_single_cn_mode=True,   # 是否开启单CN模式
    logging_level=logging.INFO # 设置日志级别
```


版权所有 © 华为云计算技术有限公司

```
)
#配置UDF部署工作区，具体参数说明，请参见2.1.5.4.2 配置UDF部署工作区。
con.set_function_staging_workspace(
    obs_directory_base=os.getenv("obs_directory_base"),
    obs_bucket_name=os.getenv("obs_bucket_name"),
    obs_server=os.getenv("obs_server"),
    access_key=os.getenv("access_key"),
    secret_key=os.getenv("secret_key"),
)
class _AggState:
    def __init__(self):
        self.sum_px_qty = 0.0
        self.sum_qty = 0.0
        self.rows_seen = 0
        self.rows_used = 0
class PythonVWAP:
    def __init__(self) -> None:
        self._state = _AggState()
    def accumulate(self, quantity: int, price: float, date: pydt.date, symbol: str) -> None:
        self._state.rows_seen += 1
        if quantity is None or price is None or date is None or symbol is None:
            return
        self._state.sum_px_qty += price * quantity
        self._state.sum_qty += quantity
        self._state.rows_used += 1
    @property
    def aggregate_state(self) -> Dict[str, Any]:
        return {
            "sum_px_qty": self._state.sum_px_qty,
            "sum_qty":    self._state.sum_qty,
            "rows_seen":  self._state.rows_seen,
            "rows_used":  self._state.rows_used,
        }
    def merge(self, other_state: Dict[str, Any]) -> None:
        self._state.sum_px_qty += other_state.get("sum_px_qty", 0.0)
        self._state.sum_qty    += other_state.get("sum_qty",    0.0)
        self._state.rows_seen  += other_state.get("rows_seen",  0)
        self._state.rows_used  += other_state.get("rows_used",  0)
    def finish(self) -> str:
        vwap = (self._state.sum_px_qty / self._state.sum_qty
                if self._state.sum_qty != 0 else None)
        result = {
            "vwap":           round(vwap, 4) if vwap is not None else None,
            "total_quantity": self._state.sum_qty,
            "total_value":    round(self._state.sum_px_qty, 2),
            "rows_seen":      self._state.rows_seen,
            "rows_used":      self._state.rows_used,
        }
        return json.dumps(result)

try:
    # 创建测试表并写入数据
    table_name = "stock_trades"
    table_schema = ibis.schema({
        "quantity": "int32",
        "price": "float64",
        "date": "date",
        "symbol": "string",
    })
    con.sql(f'DROP TABLE IF EXISTS "{target_database}"."{table_name}";')
    con.create_table(
        table_name,
        schema=table_schema,
        database=target_database,
        table_format="orc",
        if_not_exists=True,
    )
    con.sql(f'''
    INSERT INTO "{target_database}"."{table_name}" (quantity, price, date, symbol) VALUES
```


版权所有 © 华为云计算技术有限公司

```
        (100, 150.25, '2024-01-15', 'AAPL'),
        (200, 152.30, '2024-01-16', 'AAPL'),
        (150, 148.50, '2024-01-17', 'AAPL'),
        (300, 145.75, '2024-01-15', 'GOOG'),
        (250, 147.80, '2024-01-16', 'GOOG'),
        (100, 143.20, '2024-01-17', 'GOOG'),
        (500, 380.50, '2024-01-15', 'MSFT'),
        (300, 385.40, '2024-01-16', 'MSFT'),
        (200, 382.10, '2024-01-17', 'MSFT')
    ''')
    con.delete_function("pythonvwap", database=target_database, if_it_exists=True)
    con.create_agg_function(
        PythonVWAP,
        database=target_database,
        signature=aura.Signature(
            parameters=[
                aura.Parameter(name="quantity", annotation=int),
                aura.Parameter(name="price", annotation=float),
                aura.Parameter(name="date", annotation=dt.date),
                aura.Parameter(name="symbol", annotation=str),
            ],
            return_annotation=str,
        ),
        volatility=aura.udf.VolatilityType.IMMUTABLE,
        strict=False,
    )
    ds = con.load_dataset("stock_trades", database=target_database)
    vwap_handler = con.get_function("pythonvwap", database=target_database)
    udaf_builder = AggregateFnBuilder(
        fn=vwap_handler,
        on=[ds.quantity, ds.price, ds.date, ds.symbol],
        as_col="VWAP_column",
        num_cpus=0.5,
    )
    ds = ds.aggregate([udaf_builder], by=[ds.symbol])
    res = ds.execute()
    df = res.copy()
    parsed = df["VWAP_column"].apply(json.loads).apply(pd.Series)
    df = pd.concat([df.drop(columns=["VWAP_column"]), parsed], axis=1)
    print(df)
finally:
    con.close()
```

2.1.5.12.10 WITH ARGUMENTS 语法

WITH ARGUMENTS语法扩展了现有用户定义函数（UDF）的功能，主要作用是允许用户在UDF运行时设置初始状态。通过这一功能，用户可以：

- 配置运行时资源和并发度：用户可以指定UDF运行时所需的资源（如内存、CPU等）以及并发执行的实例数量，从而优化性能和资源利用率。
- 设置类的初始化参数：对于以类（class）形式定义的UDF，用户可以通过WITH ARGUMENTS语法传递初始化构造参数，从而在UDF实例化时设置特定的初始状态。

使用注意事项如下：

- WITH ARGUMENTS支持几个固定的参数配置，具体的参数信息请参考**表2-30**。对于class的初始化参数名称不能与固定参数同名，参数名称是大小写敏感的。
- 入参列表仅支持常量。
- 以class形式定义的UDF函数体，用户按需定义class的__init__和__del__成员方法，但必须定义__call__成员方法，该方法为主函数入口。
- min_concurrency和max_concurrency需要同时设置，设置成功后表示开启自适应并发度。

版权所有 © 华为云计算技术有限公司

- init_concurrency需先开启**enable_udf_global_actor_pool**，且需同时设置min_concurrency和max_concurrency，否则将报错。

表 2-30 UDF 运行时配置列表

参数名	参数描述	默认值
concurrency	UDF的运行并发度，取值范围[1,1024]。	1
min_concurrency	UDF的运行最小并发度，取值范围[1,1024]。	null
max_concurrency	UDF的运行最大并发度，取值范围[1,1024]。	null
init_concurrency	UDF运行时的预热并发度，取值范围[min_concurrency, max_concurrency]。	(min_concurrency + max_concurrency) / 2
timeout	UDF执行的超时时间，取值范围[1,3600]，单位为s。	1000
cpu	UDF运行时单个Actor的CPU资源，取值范围为[0.33, 1]之间的分数值、[1, 16]的整数值。 **需注意，系统会将输入值四舍五入，保留两位小数。**	1
memory	UDF运行时单个Actor的Memory资源，取值范围为[128, 65536]，单位为MiB，不支持小数。	4096
arch	UDF运行时的物理架构类型，取值为“arm”或“x86”。不同的apu值，arch会有不同的默认值。	• NPU：默认值为“arm” • 其他：默认值为“x86”
apu	指定异构资源的类型和规格，例如"NPU/.+/1"。	null
env_vars	运行时环境变量，dict[str, str]或JSON字符串，请参见**2.1.5.12.4 环境变量设置**。	null
scheduling_strategy	UDF Actor放置策略，SchedulingStrategy对象、对象列表或JSON字符串，请参见**2.1.5.12.13 UDF Actor放置策略**。	null

版权所有 © 华为云计算技术有限公司

2.1.5.12.11 自适应并行

场景描述

在AI数据工程中，面对大量数据处理的场景，需要通过并行调用Actor来提升数据处理的效率，进行分布式计算。但固定并行度依赖于调参经验以及多次测试反馈，为提升用户体验和减少调参时间，提供自适应并行来提高使用UDF的易用度。

约束限制

功能约束限制如下：

- 执行UDF函数时必须指定min_concurrency和max_concurrency有效值。
- 如果当前batch的行数小于计算出的并行度，实际拉起的Actor数将等于行数。否则实际拉起的actor数将等于当前计算的并行度。

当执行UDF时，同时设置min_concurrency和max_concurrency参数后，将会开启自适应并行功能。

UDF以min_concurrency并行度开始执行所涉及的数据batch。每一轮batch执行时，都会计算UDF有效计算时长占总时间的占比。当每次batch执行中，UDF有效计算占比达到80%，当前并行度将会按照2的步长增加并行度，直至最大增加至max_concurrency并行度后，后续不变化。当某一轮占比不到80%时，则下一轮并行度将保持不变。

开启自适应并行特性的UDF，各个Actor由于先后执行数据的批次不同，因此统计信息存在较大差别。对于自适应并行场景下的统计信息查看，请参考**2.1.5.12.14 Python UDF性能调优**。

2.1.5.12.12 UDF 错误处理与重试

场景描述

在AI数据工程中，UDF常涉及网络调用、外部模型推理、数据解析等易出错操作。当UDF执行遇到异常时，默认行为是直接抛出异常并中断整个查询。但在实际业务中，用户往往需要更灵活的错误处理策略，例如：

- 静默忽略：部分数据异常不影响整体结果，希望跳过错误行继续执行。
- 日志记录：记录错误信息用于事后排查，但不中断查询。
- 自动重试：网络超时、服务临时不可用等场景，希望自动重试以提升成功率。
- 按异常类型区分：不同类型的异常采用不同策略，例如超时重试、数据格式错误忽略。

UDF错误处理与重试功能允许用户在不修改源UDF代码的前提下，通过wrap_function API为已注册的UDF动态包装错误处理能力，支持忽略、日志记录、指定类型重试、退避策略等组合配置。

使用方式

1. **注册源UDF**：按照正常流程注册Python UDF（Scalar/Class/PyarrowVector/PandasVector均可）。
2. **调用wrap_function包装**：通过AuraConnection.wrap_function()为源UDF添加错误处理配置，返回一个包装后的可调用对象。

```
wrapped = con.wrap_function(
    source_udf_name="my_func",       # 源UDF名称
```


版权所有 © 华为云计算技术有限公司

```
    source_database="my_db",          # 源UDF所在数据库
    config=ErrorHandlerConfig(...),   # 错误处理配置
    wrapper_name="my_func_wrapped",   # 包装后UDF名称（可选）
    imports=["file1", "file2"],  # 包装UDF的额外导入文件
    packages=["module1", "module2"]  # 包装UDF的额外依赖包
)
```

3. **使用包装后的UDF执行查询**：将包装后的UDF用于ds.map() 或 df.select()等操作。

```
ds = con.load_dataset("my_table", database="my_db")
result_ds = ds.map(wrapped, on="input_col", as_col="result")
df = result_ds.execute()
con.close()
```

配置项说明

通过ErrorHandlerConfig配置错误处理行为，各参数说明如下：

表 2-31 ErrorHandlerConfig 参数说明

参数名	类型	说明	默认值
on_error	OnErrorAction	调用阶段错误兜底策略。 ● RAISE：抛出异常（默认）。 ● IGNORE：静默返回None。 ● LOG：记录日志并返回None。 ● RETRY：对所有异常重试。	RAISE
retry_on	list[type[Exception]]	调用阶段触发重试的异常类型白名单。匹配时进入重试逻辑，不匹配时落入on_error兜底。	None
ignore_on	list[type[Exception]]	调用阶段触发静默忽略的异常类型白名单。匹配时直接返回None，优先级最高。	None
log_on	list[type[Exception]]	调用阶段触发日志记录的异常类型白名单。匹配时记录日志并返回None。	None
max_retries	int	调用阶段最大重试次数。	0
retry_cleanup	Callable	调用阶段重试前的清理回调，签名(exception, instance) -> None。框架自动向其 __globals__ 注入is_exception_match 函数，方便按异常类型分支处理。回调自身抛出的异常会被捕获并warning，不影响重试流程。	None
init_on_error	OnErrorAction	__init__ 阶段的异常兜底策略。 ● RAISE：抛出异常（默认）。 ● RETRY：对所有异常重试。	RAISE
init_max_retries	int	__init__ 阶段最大重试次数。	0

版权所有 © 华为云计算技术有限公司

参数名	类型	说明	默认值
init_retry_on	Sequence[Type[Exception]]	__init__ 阶段触发重试的异常类型列表。	()
init_retry_cleanup	Callable	__init__ 阶段重试前的清理回调，签名 (exception, None) -> None。注意 instance参数为None（对象尚未创建）。	None
backoff	BackoffStrategy	退避策略。 ● FIXED：固定间隔。 ● LINEAR：线性递增。 ● EXPONENTIAL：指数递增。	FIXED
initial_delay_ms	int	初始延迟，单位为毫秒，作为退避计算的基数。	100
max_delay_ms	int	延迟上限，单位为毫秒。所有退避策略计算结果受此约束。	60000
log_fn	Callable	自定义日志转换函数，签名 (exception, log_context) -> dict | None。 ● 返回dict用作日志记录。 ● 返回None跳过本次日志。 ● 返回非dict或异常时则降级使用默认log_context。	None
local_log_path	str	本地错误日志文件路径。为None时走Python logger输出warning。	None
obs_log_config	ObsLogConfig	OBS日志上传配置，包含： ● obs_server：OBS服务端地址。 ● obs_bucket_name：OBS桶名。 ● obs_directory_base：OBS日志根目录路径。 ● max_buffer_size：缓冲区大小阈值（字节），默认1024，范围 [1, 10 * 1024 * 1024]。缓冲区达到此阈值时触发上传。 ● flush_interval_s：定时刷新间隔（秒），默认值600.0，即使缓冲区未满也会在此间隔后触发上传。 ● obs_log_format：日志文件格式，默认jsonl，可选jsonl、parquet。	None

错误处理优先级

当UDF执行遇到异常时，按以下优先级依次匹配：

版权所有 © 华为云计算技术有限公司

```
┌─ 优先级 1: ignore_on（最高）
│  if ignore_on and isinstance(e, ignore_on):
│      return None                    ← 静默忽略
│
├─ 优先级 2: log_on
│  if log_on and isinstance(e, log_on):
│      _log_error(e, ...)
│      return None                    ← 记录日志 + 返回 None
│
├─ 优先级 3: retry_on（指定类型重试）
│  if retry_on and isinstance(e, retry_on)
│          and retry_count < max_retries:
│      delay = _compute_delay(...)
│      sleep(delay)
│      continue                       ← 重试
│
├─ 优先级 4: on_error=RETRY（兜底全重试）
│  if on_error==RETRY and not retry_on
│          and retry_count < max_retries:
│      delay = _compute_delay(...)
│      sleep(delay)
│      continue                       ← 重试所有异常
│
├─ 优先级 5: on_error=IGNORE
│  if on_error == IGNORE:
│      return None                    ← 静默返回 None
│
├─ 优先级 6: on_error=LOG
│  if on_error == LOG:
│      _log_error(e, ...)
│      return None                    ← 记录日志 + 返回 None
│
└─ 优先级 7: raise（默认 / 重试耗尽）
   raise                              ← 抛出原始异常
```

关键规则

- ignore_on优先级最高，匹配后直接返回None，不进入重试或日志逻辑。
- log_on优先级高于retry_on，匹配后记录日志并返回None，不进入重试。
- retry_on指定类型重试；on_error=RETRY不指定类型，对所有异常重试。两者共存时retry_on优先。
- 重试耗尽后，落入on_error决定最终行为（RAISE/IGNORE/LOG）。

退避策略

重试间隔由backoff策略和initial_delay_ms计算，所有策略最终受max_delay_ms约束：

表 2-32 退避策略间隔时间说明

backoff 策略	计算公式	示例（initial=100ms, max=5000ms）
FIXED	delay = initial_delay_ms	100, 100, 100, ...
LINEAR	delay = initial_delay_ms × (attempt + 1)	100, 200, 300, 400, ...
EXPONENTIAL	delay = initial_delay_ms × 2^attempt	100, 200, 400, 800, 1600, 3200, ...

版权所有 © 华为云计算技术有限公司

所有策略最终受max_delay_ms约束：min(delay, max_delay_ms)。

约束限制

- **自定义异常限制**：不支持定义在类中的自定义异常，定义在同名模块中的同名异常会被认为是相同异常。
- **不支持包装BUILTIN函数**：wrap_function 仅支持包装Python UDF，不支持数据库内置函数（如 ABS()、UPPER() 等）。
- **WRAPPER默认为TEMPORARY**：包装后的UDF默认为临时函数，会话结束后自动销毁。
- **Vectorized UDF整批失败**：对于PyarrowVector / PandasVector 类型的UDF，当批次中任一行触发异常时，整个批次返回None。
- **敏感数据安全**：用户自定义日志字段和UDF参数中如果存在敏感数据（如账号、密码等信息），需要用户自行脱敏。

示例

示例1：指定异常类型重试

网络调用UDF，遇到TimeoutError时自动指数退避重试，重试最多3次：

```
from aura_frame import ErrorHandlerConfig, BackoffStrategy
from aura_frame.multimodal import ai_lake

con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),         # AuraJobV2端点所在的终端节点（Endpoint）
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2端点ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2端点名称
    aura_workspace_id=os.getenv("aura_workspace_id"),  # 工作空间ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog名称
    access_key=os.getenv("access_key"),                # 访问密钥ID（AK）
    secret_key=os.getenv("secret_key"),                # 访问密钥（SK）
    default_database="postgres",                       # LakeFormation数据库名称
    use_single_cn_mode=True,   # 是否开启单CN模式
    logging_level=logging.INFO # 设置日志级别
)

try:
    wrapped = con.wrap_function(
        source_udf_name="call_api",
        source_database="my_db",
        config=ErrorHandlerConfig(
            retry_on=[TimeoutError],
            max_retries=3,
            backoff=BackoffStrategy.EXPONENTIAL,
            initial_delay_ms=200,
            max_delay_ms=5000,
        ),
    )
    ds = con.load_dataset("api_input", database="my_db")
    result_ds = ds.map(wrapped, on="input", as_col="result")
    df = result_ds.execute()
finally:
    con.close()
```

示例2：多策略组合

源UDF同时支持重试、日志降级和静默忽略，异常匹配按ignore_on、log_on、retry_on、on_error(默认RAISE) 的优先级依次判断：

1. IndexError：静默忽略返回None，不记录日志、不重试。

版权所有 © 华为云计算技术有限公司

2. ValueError/TypeError：记录日志后返回None，因配置了local_log_path="/tmp/udf_errors"，写日志到该目录下。
3. ConnectionError/TimeoutError：
   a. 重试最多5次，每次重试前：记录日志（同上）->计算延迟->sleep->重试。
   b. 当5次重试全部失败后再次异常：on_error默认为RAISE，抛出原始异常，该行执行失败。
4. 其他未配置的异常（如 RuntimeError、KeyError 等）：on_error默认为RAISE，抛出原始异常，该行执行失败。

```
wrapped = con.wrap_function(
    source_udf_name="complex_model",
    source_database="my_db",
    config=ErrorHandlerConfig(
        retry_on=[ConnectionError, TimeoutError],
        log_on=[ValueError, TypeError],
        ignore_on=[IndexError],
        max_retries=5,
        backoff=BackoffStrategy.LINEAR,
        initial_delay_ms=500,
        local_log_path="/tmp/udf_errors.log",
    ),
)
```

示例3：自定义异常+自定义日志+清理回调用户UDF中可能抛出多种异常，其中某个异常如果要重试，在重试前需要做一些清理动作。

1. UDF中通过log_extra["key"] = "..."新增日志字段，通过log_fn修改日志内容或定义记录的日志字段。
2. retry_cleanup/init_retry_cleanup配置调用/初始化阶段清理回调。

```
import random
import pandas
from aura_frame.error_handler import (
    BackoffStrategy,
    ErrorHandlerConfig,
    ObsLogConfig,
    OnErrorAction,
)
from aura_frame.multimodal import ai_lake
class InvalidInputError(Exception):
    """无效输入"""
class ExceedsLength(RuntimeError):
    """字符串超过长度"""
class WriteFileError(Exception):
    """写文件失败"""
class DataTransformer:
    """调用外部 HTTP 服务进行数据转换的类 UDF"""
    def __init__(self):
        self.handler = open("/tmp/demo.txt", "a+")  # noqa: SIM115
    def __call__(self, col: str) -> int:
        import os
        log_extra = globals().get("log_extra", None)
        if isinstance(log_extra, dict):
            log_extra.update(
                {"ak": os.getenv("UDF_TEMP_AK"), "sk": os.getenv("UDF_TEMP_SK")}
            )
        if not isinstance(col, str) or not col:
            raise InvalidInputError("The input cannot be an empty string.")
        if len(col) > 10:
            raise ExceedsLength("The input string is exceeds length.")
        if random.random() < 0.5:  # 模拟 50% 的概率发生写文件失败
            raise WriteFileError("Failed to write to file.")
        try:
            self.handler.write(col + "\n")
        except Exception as e:  # noqa: BLE001
```


版权所有 © 华为云计算技术有限公司

```
            raise WriteFileError(f"Failed to write to file: {e}")
        return id(self.handler)
    def __del__(self):
        self.handler.close()
session = ai_lake.connect(
    aura_endpoint="...",
    aura_endpoint_name="...",
    aura_workspace_id="...",
    lf_catalog_name="...",
    access_key="...",
    secret_key="...",
    security_token="...",
    default_database="err_handler",
    use_single_cn_mode=True,
)
session.sql("create schema if not exists err_handler;")
session.sql("set current_schema to err_handler;")
session.set_function_staging_workspace(
    obs_directory_base="...",
    obs_bucket_name="...",
    obs_server="...",
    access_key="...",
    secret_key="...",
    security_token="...",
)
session.delete_function("data_transformer_src", if_it_exists=True)
source_udf = session.create_scalar_function(
    DataTransformer,
    name="data_transformer_src",
)
def on_retry_cleanup(exc, instance):
    """重试前根据异常类型执行不同清理逻辑"""
    if instance is None:
        return  # __init__ 阶段，对象尚未创建
    # is_exception_match 由框架自动注入到 __globals__
    if globals()["is_exception_match"](exc, WriteFileError):
        instance.handler.close()
    instance.handler = open("/tmp/demo.txt", "a+")  # noqa: SIM115
def custom_log_fn(exc, log_context):
    """自定义日志格式：脱敏 + 添加业务标签"""
    import traceback
    record = dict(log_context)
    record["exc_msg"] = traceback.extract_tb(exc.__traceback__)
    record["sk"] = "******"  # 敏感信息脱敏
    return record
# 包装源 UDF
wrapped_udf = session.wrap_function(
    source_udf_name="data_transformer_src",
    wrapper_name="data_transformer",
    config=ErrorHandlerConfig(
        # 异常分类处理
        on_error=OnErrorAction.LOG,
        ignore_on=[InvalidInputError],  # 无效输入 -> 静默跳过
        log_on=[ExceedsLength],  # 格式错误 -> 日志 + 跳过
        retry_on=[WriteFileError],  # 瞬时错误 -> 重试
        # 重试退避策略
        backoff=BackoffStrategy.EXPONENTIAL,
        # 调用阶段重试控制
        max_retries=10,  # 最多重试10次
        initial_delay_ms=100,
        max_delay_ms=5000,
        retry_cleanup=on_retry_cleanup,
        # 初始化阶段重试
        init_on_error=OnErrorAction.RETRY,
        init_max_retries=2,
        init_retry_on=[ConnectionError],
        init_retry_cleanup=on_retry_cleanup,
        # 日志定义
        log_fn=custom_log_fn,  # 自定义日志格式
```


版权所有 © 华为云计算技术有限公司

```
        local_log_path="/tmp/...", # 本地日志目录
        obs_log_config=ObsLogConfig(
            obs_server="...",
            obs_bucket_name="...",
            obs_directory_base="...",
            max_buffer_size=4096,
            flush_interval_s=300.0,
            obs_log_format="parquet",
        ),
    ),
)
# 使用包装后的UDF查询（使用方式与源UDF完全一致）
ds = session.from_pandas(pandas.DataFrame({"col": ["", "123", "1234", "12345", "123456", "1234567",
"1234567890123456"]}))
ds = ds.map(wrapped_udf, on=[ds.col])
ds.show()
session.close()
```

2.1.5.12.13 UDF Actor 放置策略

scheduling_strategy用于控制UDF Actor的放置策略，支持两种策略类型。

策略 1：InstanceSchedulingStrategy 实例共置/分散

控制同一查询的多个Actor实例的放置关系：

```
from aura_frame.ibis.expr.operations.scheduling_strategy import (
    InstanceSchedulingStrategy, InstanceAffinityMode
)

# 多实例必须同节点（共享缓存/通信）
InstanceSchedulingStrategy(mode=InstanceAffinityMode.SAME_REQUIRED)

# 多实例必须不同节点（容错/负载均衡）
InstanceSchedulingStrategy(mode=InstanceAffinityMode.DIFF_REQUIRED)

# 优先同节点，允许回退
InstanceSchedulingStrategy(mode=InstanceAffinityMode.SAME_PREFERRED)

# 优先不同节点，允许回退
InstanceSchedulingStrategy(mode=InstanceAffinityMode.DIFF_PREFERRED)
```

SAME_REQUIRED首次分发限制：首次分发时pod上不存在同查询已分发实例标签，无法满足required约束，会返回ERROR。仅在多个Actor按序分发时有效。生产推荐SAME_PREFERRED或DIFF_PREFERRED，示例如下：

```
from aura_frame.ibis.expr.operations.scheduling_strategy import (
    InstanceSchedulingStrategy,
    InstanceAffinityMode,
)

# 场景：数据预处理（分散优先）+ 模型推理（必须共置）
# Actor-A 先分发落位，Actor-B 通过相同label追踪

# Actor-A：DIFF_PREFERRED 先分发，分散到不同节点
ds = ds.map(
    fn="my_schema.preprocess",
    on=["raw_col"],
    as_col="preprocessed",
    scheduling_strategy=InstanceSchedulingStrategy(
        mode=InstanceAffinityMode.DIFF_PREFERRED,
        label="inference-pipeline",
    ),
)

# Actor-B：SAME_REQUIRED 追到Actor-A所在节点，共享预处理缓存
ds = ds.map(
    fn="my_schema.inference",
```


版权所有 © 华为云计算技术有限公司

```
    on=["preprocessed"],
    as_col="score",
    scheduling_strategy=InstanceSchedulingStrategy(
        mode=InstanceAffinityMode.SAME_REQUIRED,
        label="inference-pipeline",
    ),
)
```

关键点：

- **相同的label**：两个Actor必须使用完全相同的label字符串。
- **非REQUIRED先分发**：用DIFF_PREFERRED或SAME_PREFERRED的Actor先落位。
- **REQUIRED后分发**：用SAME_REQUIRED的Actor通过label追过去。

如果不需要强制共置，直接用SAME_PREFERRED即可，无需自定义label：

```
# 优先共置，允许回退——不需要 SAME_REQUIRED 的简单场景
scheduling_strategy=InstanceSchedulingStrategy(mode=InstanceAffinityMode.SAME_PREFERRED),
```

Label命名规范：

- 最大63个字符
- 必须以字母或数字开头和结尾
- 中间只能使用字母、数字、-、_、.
- 禁止使用保留前缀 actor_

策略 2：NodeLabelSchedulingStrategy 节点选择

基于Kubernetes节点标签，约束Actor放置位置：

```
from aura_frame.ibis.expr.operations.scheduling_strategy import (
    NodeLabelSchedulingStrategy, In, NotIn, Exists, DoesNotExist
)

# 硬约束：必须在 pool=gpu 的节点
NodeLabelSchedulingStrategy(hard={"pool": In("gpu")})

# 软约束：优先高带宽节点，不满足时回退
NodeLabelSchedulingStrategy(soft={"high_speed_net": Exists()})

# 组合使用
NodeLabelSchedulingStrategy(
    hard={"pool": In("gpu")},            # 必须满足
    soft={"high_mem": In("true")},       # 优先满足
)
```

四个操作符：

操作符	写法	含义
In	{"pool": In("gpu", "npu")}	标签值匹配其中之一
NotIn	{"env": NotIn("test")}	标签值不在指定范围
Exists	{"npu_type": Exists()}	标签key存在
DoesNotExist	{"maintenance": DoesNotExist()}	标签key不存在

版权所有 © 华为云计算技术有限公司

多 Workgroup 场景下的调度

集群中同一类型设备（如GPU）可能被划分为多个workgroup（资源池），每个workgroup在节点上以**endpointId**作为标签 key 标识，标签value为workgroup名称。当存在多个workgroup时，device="GPU" 等参数只能将Actor调度到GPU类型的节点，但无法区分具体哪个workgroup。

说明

endpointId是Aura endpoint的标识。不同集群的endpointId可能不同，用户可在租户面查看自己的endpointId。

通过NodeLabelSchedulingStrategy可以在运行时指定目标workgroup：

```
from aura_frame.ibis.expr.operations.scheduling_strategy import (
    NodeLabelSchedulingStrategy, InstanceSchedulingStrategy,
    InstanceAffinityMode, In,
)

# 指定调度到名为 "gpu-pool-a" 的 GPU workgroup，同时实例分散
# 标签 key 为 endpointId（从租户面获取），value 为 workgroup 名称
ds = ds.map(
    fn="my_schema.inference",
    on=["feature_col"],
    as_col="result",
    device="GPU",
    model="A100",
    num_models=1,
    scheduling_strategy=[
        NodeLabelSchedulingStrategy(hard={"<endpointId>": In("gpu-pool-a")}),
        InstanceSchedulingStrategy(mode=InstanceAffinityMode.DIFF_PREFERRED),
    ],
)
```

表 2-33 workgroup 调度的优先级和共存规则

来源	级别	优先级	说明
work_group_name	DDL注册级	最高	CREATE FUNCTION 中指定，内核以LabelInOperator(endpointId, {workgroup_name}) 追加为AND约束。
NodeLabelSchedulingStrategy	运行时调用级	中	WITH ARGUMENTS中指定，灵活可变。

关键规则：

- **work_group_name与scheduling_strategy共存**：work_group_name作为AND约束追加到标签列表中（key为endpointId），不会覆盖或清除 scheduling_strategy 的约束。
- **多workgroup必须显式选择**：同类型存在多个workgroup时，需要通过NodeLabelSchedulingStrategy或work_group_name指定目标，否则调度结果不可预测。

版权所有 © 华为云计算技术有限公司

表 2-34 注册时与运行时：两种设置方式的差异

维度	work_group_name	scheduling_strategy
设置时机	UDF注册时（CREATE FUNCTION）	查询调用时（WITH ARGUMENTS）
灵活性	一次注册，永久生效	每次查询可不同
生效范围	该UDF的所有调用	仅当前查询
变更成本	需重新注册UDF	修改代码即可
SQL层级	DDL级	DML级

表 2-35 选型建议

场景	推荐做法
UDF**始终**运行在同一个workgroup	注册时通过work_group_name指定，避免每次查询重复设置
UDF需要在**不同workgroup间切换**（灰度、多租户）	注册时**不设置**work_group_name，通过scheduling_strategy运行时灵活指定

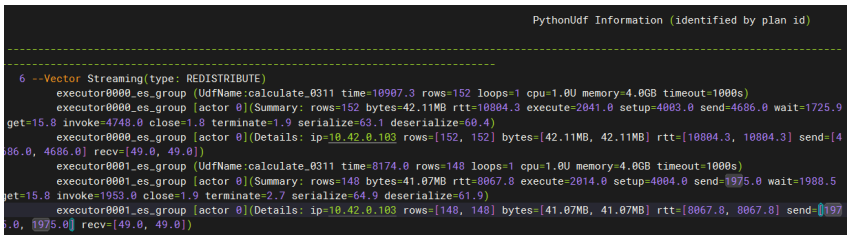
2.1.5.12.14 Python UDF 性能调优

Aura支持统计运行时的关键性能指标，根据性能指标，用户可以调整UDF运行时的资源规格和并行度。

UDF 性能监控

使用explain performance可以打印UDF Actor运行的性能指标。如图2-2所示，统计了函数calculate_0311在每个DN所对应的UDF Actor的性能情况。下文以图中三行性能数据为例进行说明。

图 2-2 Python UDF Actor 的性能指标统计



- executor0000_es_group (UdfName:calculate_0311 time=10907.3 rows=152 loops=1 cpu=1.0U memory=4.0GB timeout=1000s)中：
  - executor0000_es_group指的是DN ID。
  - UdfName表示执行UDF的名称。

版权所有 © 华为云计算技术有限公司

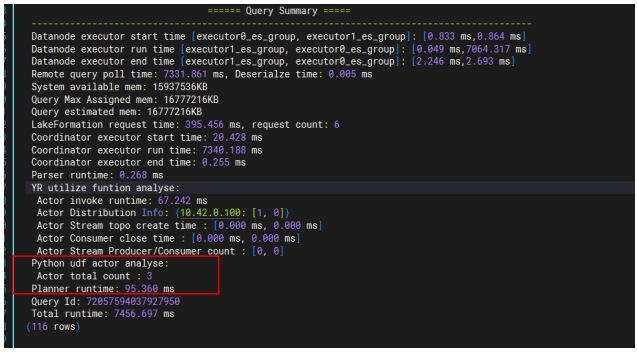
  - time描述该UDF总运行时间。
  - rows表示在该DN上处理的数据量。
  - loops表示调用UDF的次数。
  - cpu表示该UDF执行时actor所申请的cpu大小，单位为U。
  - memory表示该UDF执行时actor所申请的memory大小，单位为GB。
  - timeout表示该UDF执行时actor单次batch的最大执行时长，单位为秒。超时则会报错。
- executor0000_es_group [actor 0](Summary: rows=152 bytes=42.11MB rtt=10804.3 execute=2041.0 setup=4003.0 send=4686.0 wait=1725.9 get=15.8 invoke=4748.0 close=1.8 terminate=1.9 serialize=63.1 deserialize=60.4) 描述的是第0个UDF Actor的信息，具体如下：
  - rows表示该actor上处理的数据量。
  - bytes表示该actor处理的数据量大小。
  - rtt表示内核发送数据到actor，以及actor接收处理数据的端到端时间。
  - execute表示该actor实际计算的有效时间。
  - setup表示该actor在初始化后，准备环境和装载udf代码的时间。
  - send表示从内核发送数据到该actor的时间。
  - wait表示内核等待actor返回数据的等待时间。
  - get表示内核获取actor数据成功获取数据的时间。
  - invoke表示拉起actor的时间。
  - close表示该actor关闭计算的时间。
  - terminate表示数据系统下线actor的耗时。
  - serialize表示对数据序列化的时间。
  - deserialize表示对数据反序列化的时间。
- executor0000_es_group [actor 0](Details: ip=10.42.XX.XX rows=[152, 152] bytes=[42.11MB, 42.11MB] rtt=[10804.3, 10804.3] send=[4686.0, 4686.0] recv=[49.0, 49.0])统计信息中：
  - ip表示actor所在的ip地址。
  - rows表示actor执行的所有batch中最小行数和最大行数。
  - bytes表示actor执行的所有batch中数据大小最小值和最大值。
  - rtt表示actor执行的所有batch中端到端执行时间的最小值和最大值。
  - send表示actor执行的所有batch中从内核发送数据到该actor的最小值和最大值。
  - recv表示actor执行的所有batch中从python发送数据到内核接收的最小值和最大值。

如下图所示，Query Summary中统计了该查询中所有UDF Actor的汇总信息。

Actor total count表示当前query调用的所有actor数量。

版权所有 © 华为云计算技术有限公司

图 2-3 Python UDF Actor 数据流传输信息总览



UDF 下推优化

函数返回值中存在record类型，会导致函数无法下推到DN，提示：Function xxx can not be shipped because return record。

例如:

```
SELECT jsonb_each(data::jsonb) FROM table1;
```

如果需要支持下推, 则取消使用返回值为record类型的函数, 改为按需取值。

```
SELECT  data::jsonb ->>'car_model' AS car_model, data::jsonb ->>'color' AS color FROM  table1;
```

2.1.5.13 数据类型

2.1.5.13.1 图片

图片类型内部使用Pillow库处理图片，获取PIL Image对象后通过PIL库处理图片。支持处理的图片格式：JPEG、PNG。

表 2-36 属性（Properties）

属性名	类型	说明
filename	str	图像的文件名或路径（如果从文件加载）。
format	str	图像格式（如PNG、JPEG）。
height	int	图像高度（像素）。
width	int	图像宽度（像素）。
data	bytes	图像数据。

版权所有 © 华为云计算技术有限公司

表 2-37 方法（Methods）

方法名	参数	返回值	说明
__init__	• filename: Union[str, os.PathLike] = None • data: Union[str, PIL.Image] = None • format: Optional[str] = None	None	构造方法：从文件（filename）或数据（data）加载图像，支持bytes或Pillow对象。
to_numpy	-	np.ndarray	将图像转换为NumPy数组（float32类型）。
resize	new_shape: tuple[int, int]	Image	返回调整尺寸后的新图像。
show	-	None	调用Pillow的默认图像查看器显示图像。
to_thumbnail	size: tuple[int, int] = (100, 100)	Image	生成指定尺寸的缩略图。
crop	box: tuple[float, float, float, float]	Image	裁剪图像指定区域，返回新的Image对象。
getbands	-	tuple: [str, ...]	调用Pillow的getbands()。
getmode	-	str	返回图像mode。
to_struct	-	dict	返回图片结构化数据。

2.1.5.13.2 视频

视频内部使用PyAV处理数据。

- 支持的格式：mp4、mkv、webm、avi、mov、flv、m4v、m4a、3gp、3g2、ts。
- 支持的编码：avc1、avc3、h264、hvc1、h265、hevc、vp9、vp8、mp4v。

视频对象的属性和方法如下所示：

表 2-38 属性（Attributes）

属性名	类型	说明
filename	str	视频文件的路径名（如果从文件加载）。
format	str	视频格式（如mp4、avi）。

版权所有 © 华为云计算技术有限公司

属性名	类型	说明
data	bytes	视频数据。

表 2-39 方法（Methods）

方法名	参数	返回值	说明
__init__	filename: Union[str, os.PathLike] = None,data: Union[str, bytes] = None	None	构造函数：从文件或二进制数据加载视频。
truncate	start_time: float（开始时间，秒） end_time: float（结束时间，秒） shape: Optional[tuple[int, int]]（目标分辨率） fps: float = None（目标帧率） codec: str = "avc1"（编码格式） format: str = "mp4"（输出格式）	Video	截取视频片段，支持调整分辨率、帧率和编码格式，返回新Video对象。
write	filename: str, codec: str	None	保存视频到文件。
convert_format	format	Video	转换为不同格式，返回新的Video对象。

2.1.5.13.3 音频

音频内部使用PyAV库处理数据。支持的音频格式：mp3、aiff、flac、wav、wma、m4a、aac。音频对象的属性和方法如下所示：

表 2-40 属性（Properties）

属性名	类型	说明
filename	str	音频文件的路径或文件名（如果从文件加载）。

版权所有 © 华为云计算技术有限公司

属性名	类型	说明
format	str	音频格式（如wav、mp3）。
subtype	str	音频编码。
data	bytes	音频数据。

表 2-41 方法（Methods）

方法名	参数	返回值	说明
__init__	filename: Union[str, os.PathLike] = None data: Union[str, bytes, sf.SoundFile] = None	None	构造方法：从文件（filename）或数据（data）加载音频，支持二进制数据或soundfile对象。
truncate	start_time: float = 0 end_time: Optional[float] = None	Audio	截取音频片段（时间单位为秒），返回新Audio对象。
convert_format	format: str subtype: Optional[str] = None	Audio	转换音频格式（如wav转mp3），返回新Audio对象。
write	filename: str	None	将音频写入指定文件（自动推断格式）。
split	interval: int	list[Audio]	按指定interval秒数分割audio为片段列表。

2.1.5.13.4 向量

向量封装了EmbeddingVector类，可以通过这个类对象表示一维向量类型。

表 2-42 属性（Properties）

属性名	类型	说明
vector	list[float]	一维向量列表。

版权所有 © 华为云计算技术有限公司

属性名	类型	说明
dims	int	向量维度。

表 2-43 方法（Methods）

方法名	参数	返回值	说明
__init__	obj: dict	None	构造方法：从字典对象中初始化向量对象。
to_struct	None	dict	返回一个字典对象，包含当前向量类对象的向量和维度。
from_struct	cls, obj:dict	EmbeddingVector	输入一个字典，调用类方法返回一个类对象。

向量创建

该示例演示如何创建一个向量类型。

```
from aura_frame.multimodal.types import EmbeddingVector

vec_dict = {"vector": [1, 2, 3], "dims": 3}
vec = EmbeddingVector(vec_dict)
print(vec.vector)
print(vec.dims)
```

2.1.6 Spark SDK（Python）

2.1.6.1 Spark SDK 获取与初始化

对AIDataLake服务提供的Spark API的封装，以简化用户的开发工作。用户直接调用SDK提供的接口函数即可实现使用提交Spark SQL和Spark Job作业。本文为您介绍如何进行Spark SDK的获取与初始化。

前提条件

已完成**2.1.2 Python环境配置**。

步骤一：获取 SDK

下载SDK文件，并建议您同时下载.sha256文件，该文件可用于验证SDK包的完整性。验证方法请参考**步骤二：验证SDK包的完整性**。

- **SDK安装包链接**

版权所有 © 华为云计算技术有限公司

- **校验文件（SHA256）链接**

步骤二：验证 SDK 包的完整性

下载SDK安装包时，您需要同时下载.sha256文件，用于验证SDK包的完整性。本节介绍使用.sha256文件验证SDK包的完整性的操作步骤。

1. **查询SDK安装包的SHA - 256哈希值**
   - **在Windows系统中**
     i. 在开始菜单中搜索“cmd”打开命令窗口。
     ii. 使用cd命令切换到保存SDK安装包的文件夹目录。
     例如：SDK安装包保存在D盘的“SDK”文件夹中，输入**cd D:\SDK**。
     iii. 执行以下命令
     ```
     certutil -hashfile SDK安装包文件名 SHA256
     ```
     将“SDK安装包文件名”替换为实际的SDK安装包文件名，如aidatalake-python-sdk-1.0.0-261027.tar.gz。
     iv. 命令执行后，命令提示符会输出SDK安装包的SHA - 256哈希值。
   - **在Linux系统中**
     i. 使用cd命令切换到保存SDK安装包的文件夹目录。
     例如，如果SDK安装包保存在用户主目录下的“SDK”文件夹中，输入**cd ~/SDK**。
     ii. 输入查询命令：
     ```
     sha256sum SDK安装包文件名
     ```
     将“SDK安装包文件名”替换为实际的SDK安装包文件名，如aidatalake-python-sdk-1.0.0-261027.tar.gz 。
     iii. 命令执行后，终端会输出SDK安装包的SHA - 256哈希值。
2. **对比哈希值**
   a. 使用文本编辑器打开下载的.sha256文件。
   b. 在.sha256文件中找到SDK安装包的SHA - 256哈希值。
   通常是一个长字符串，例如“a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6xxxxxxxxxxxxxxxxxxxxxxxx”。
   c. 将**步骤1**查询得到的SDK安装包的SHA - 256哈希值与.sha256文件中的哈希值进行对比。
      - 如果两个哈希值完全一致，说明SDK安装包在下载过程中没有被篡改，文件是完整的。
      - 如果哈希值不一致，这可能意味着SDK安装包在下载过程中出现了错误，或者文件被恶意修改。
        此时，建议重新下载SDK安装包和对应的.sha256文件，然后再次进行验证。

步骤三：安装 SDK

1. 单击**aidatalake-python-sdk-x.x.x.tar.gz**下载SDK包。
2. 解压SDK安装包。
   ```
   tar -xzf aidatalake-python-sdk-x.x.x.tar.gz
   ```


版权所有 © 华为云计算技术有限公司

3. 打开命令行窗口，解压后进入目录安装。

```
cd aidatalake-python-sdk-x.x.x
python setup.py install
```

步骤四：初始化

用AIDataLake SDK工具访问AIDataLake，需要用户初始化AIDataLake客户端。用户可以使用AK/SK(Access Key ID/Secret Access Key)认证方式初始化客户端。

```
from huaweicloudsdkaidatalake.v2.aidatalake_client import AIDataLakeClient

client = AidatalakeClient(
    region="cn-north-4",
    project_id="xxxx-xxxx",
    workspace_id="xxx-xxxx",
    ak="xxxx",
    sk="xxxx",
    security_token="xxx",
    endpoint="{regionName}.myhuaweicloud.com",
    obs_endpoint="obs.{regionName}.myhuaweicloud.com",
    endpoint_name="spark-job/sql-xxx"
)
```

表 2-44 参数详情

参数	必须配置	默认值	描述
region	是	无	区域名称。
project_id	是	无	项目ID。
workspace_id	是	无	工作空间ID。
ak	是	无	访问密钥ID。
sk	是	无	访问密钥密码。
security_token	否	无	使用临时凭证时使用。
endpoint	是	无	Aidatalake URL。
obs_endpoint	否	无	OBS Endpoint 不设置时默认使用obs.{RegionName}.myhuaweicloud.com)
endpoint_name	否	无	AIDataLake Endpoint，如：fabric.{regionName}.myhuaweicloud.com

2.1.6.2 Spark SQL 作业提交

功能说明

本节操作介绍使用AI DataLake SDK提交SQL作业的操作方法。

前提条件

- 已完成配置Python SDK环境并完成了初始化，详情请参见**2.1.6.1 Spark SDK获取与初始化**。

版权所有 © 华为云计算技术有限公司

- API和SQL作业相关权限已配置，权限详情请参见**权限及授权项说明**。
- 使用的计算资源和端点为可用状态。

函数

AidatalakeClient

```
# 同步提交函数
execute_sql();

# 异步提交函数
async_execute_sql();
```

样例代码

- 异步提交SQL

```
try:
    sql_job = client.async_execute_sql(
        "SELECT 1 AS value",
        database_name="database_name",
        endpoint_name="endpoint_name",
        catalog="catalog_name",
    )
    logger.info("Job submitted. Statement ID: %s", sql_job.job_id)
    # Poll for completion
    status, msg = sql_job.get_status()
    logger.info("Initial status: %s", status)
except Exception as e:
    logger.error("Example 2 FAILED: %s", e)
```

- 同步提交SQL

```
try:
    sql_job = client.execute_sql(
        "SHOW DATABASES",
        database_name="database_name",
        endpoint_name="endpoint_name",
        catalog="catalog_name",
    )
    logger.info("Job completed. ID: %s, Status: %s", sql_job.job_id, sql_job.job_status)
except Exception as e:
    logger.error("Example 1 FAILED: %s", e)
```

- SQL查询

```
try:
    job = client.new_sql_job_builder("SELECT * FROM demo_db1.demo_tbl1 LIMIT 10") \
        .catalog_name(catalog_name) \
        .database_name(database_name) \
        .labels([{"key": "env", "value": "dev"}]) \
        .running_timeout(300) \
        .queue_timeout(60) \
        .job_timeout(600) \
        .build()
    # Submit and wait
    job.submit()
    logger.info("Job completed. ID: %s", job.job_id)
    result = job.get_result()
    if result:
        row_count = sum(1 for _ in result)
        logger.info("Rows: %d", row_count)
        if hasattr(result, 'close'):
            result.close()
except Exception as e:
    logger.error("Example 3 FAILED: %s", e)
```


版权所有 © 华为云计算技术有限公司

参数说明

表 2-45 参数详情

参数	必须配置	默认值	描述
database_name	是	无	默认LakeFormation数据库名称。
endpoint_name	是	无	SparkSQL端点名称。
catalog	是	无	默认LakeFormation Catalog名称。

2.1.6.3 Spark Job 作业提交

功能说明

本节操作介绍使用AI DataLake SDK提交Spark作业的操作方法。

前提条件

- 已完成配置Python SDK环境并完成了初始化，详情请参见**2.1.6.1 Spark SDK获取与初始化**。
- API和SQL作业相关权限已配置，权限详情请参见**权限及授权项说明**。
- 使用的计算资源和端点为可用状态。

函数

- SparkJarJob

```
# 同步提交函数
submit_and_wait();

# 异步提交函数
submit();
```

- SparkPythonJob

```
# 同步提交函数
submit_and_wait();

# 异步提交函数
submit();
```

- SparkSqlScriptingJob

```
# 同步提交函数
submit_and_wait();

# 异步提交函数
submit();
```

样例代码

- spark_jar_job类型作业

```
try:
    jar_job = client.new_jar_job_builder(
        "test-jar-job",
```


版权所有 © 华为云计算技术有限公司

```
        "org.example.WordCount",
        "obs://obs-bucket/spark-jar-test-1.0-SNAPSHOT.jar",
    ) \
        .endpoint_name("endpoint_name") \
        .catalog_name("catalog_name") \
        .main_args(["500000"]) \
        .job_agency("job_agency") \
        .driver_cpu(1000).driver_memory(4096) \
        .executor_cpu(1000).executor_memory(4096) \
        .executor_number(2) \
        .labels([{"key": "env", "value": "dev"}]) \
        .spark_config({
            "spark.dynamicAllocation.enabled": "true",
            "spark.dynamicAllocation.maxExecutors": "3",
        }) \
        .build()
    # Submit asynchronously, then wait
    jar_job.submit_and_wait()
    logger.info("JAR job completed. ID: %s, Status: %s",
                jar_job.job_id, jar_job.status)
except AidatalakeException as e:
    logger.error("Example 1 FAILED: %s", e)
```

表 2-46 参数详情

参数	必须配置	默认值	描述
endpoint_name	是	无	SparkJob端点名称。
catalog_name	否	无	默认LakeFormation Catalog名称。 如果作业内执行Spark.sql可选择设置。
main_args	否	无	主类Main参数
job_agency	否	无	作业IAM委托，用户访问OBS或Lakeformation。
spark_config	否	无	Spark参数。
driver_cpu	否	无	Driver POD毫核数。
driver_memory	否	无	Driver POD内存MB。
executor_cpu	否	无	Executor POD毫核数。
executor_memory	否	无	Executor POD内存MB。
executor_number	否	无	Executor实例个数。

- spark_python_job类型作业

```
try:
    python_job = client.new_python_job_builder(
        "test-python-job",
        "obs://obs-bucket/sleep_1.py",
```


版权所有 © 华为云计算技术有限公司

```
    ) \
        .endpoint_name("endpoint_name") \
        .catalog_name("catalog_name") \
        .main_args(["500000"]) \
        .job_agency("job_agency") \
        .dependency_py_files(["obs://obs-bucket/utils.py"]) \
        .driver_cpu(1000).driver_memory(4096) \
        .executor_cpu(1000).executor_memory(4096) \
        .executor_number(1) \
        .build()
    python_job.submit_and_wait()
    logger.info("Python job completed. ID: %s, Status: %s",
                python_job.job_id, python_job.status)
except AidatalakeException as e:
    logger.error("Example 2 FAILED: %s", e)
```

表 2-47 参数详情

参数	必须配置	默认值	描述
endpoint_name	是	无	SparkJob端点名称。
catalog_name	否	无	默认LakeFormation Catalog名称。 如果作业内执行Spark.sql可选择设置。
main_args	否	否	主类Main参数
job_agency	否	无	作业IAM委托，用户访问OBS或Lakeformation。
dependency_py_files	否	否	依赖的Python文件
driver_cpu	否	无	Driver POD豪核数。
driver_memory	否	无	Driver POD 内存MB。
executor_cpu	否	无	Executor POD豪核数。
executor_memory	否	无	Executor POD 内存MB。
executor_number	否	无	Executor实例个数。

- spark_sql_scripting_job类型作业

```
try:
    sql_scripting_job = client.new_sql_scripting_job_builder(
        "test-sql-scripting-job",
    ) \
        .sql_scripting_file("obs://obs-bucket/test_sql.sql") \
        .sql_scripting_result_to_obs(True) \
        .endpoint_name("endpoint_name") \
        .catalog_name("catalog_name_sql") \
        .job_agency("job_agency") \
        .driver_cpu(1000).driver_memory(4096) \
        .executor_cpu(1000).executor_memory(4096) \
```


版权所有 © 华为云计算技术有限公司

```
        .executor_number(1) \
        .build()
    sql_scripting_job.submit_and_wait()
    logger.info("SQL Scripting job completed. ID: %s, Status: %s",
                sql_scripting_job.job_id, sql_scripting_job.status)
    # Read results from OBS
    result = sql_scripting_job.get_result()
    row_count = 0
    for row in result:
        row_count += 1
        if row_count <= 10:
            logger.info("  Row %d: %s", row_count, row)
    logger.info("Total rows: %d", row_count)
    if hasattr(result, 'close'):
        result.close()
except AidatalakeException as e:
    logger.error("Example 3 FAILED: %s", e)
```

表 2-48 参数详情

参数	必须配置	默认值	描述
sql_scripting_file	是	无	SqlScripting OBS脚本路径。
endpoint_name	是	无	SparkJob端点名称。
catalog_name	否	无	默认LakeFormation Catalog名称。 如果作业内执行Spark.sql可选择设置。
job_agency	否	无	作业IAM委托，用户访问OBS或Lakeformation。
driver_cpu	否	无	Driver POD豪核数。
driver_memory	否	无	Driver POD 内存MB。
executor_cpu	否	无	Executor POD豪核数。
executor_memory	否	无	Executor POD 内存MB。
executor_number	否	无	Executor实例个数。

2.1.6.4 Spark 连接数据库并查询

功能说明

本节介绍如何使用AI DataLake DB-API连接数据库并进行查询。

版权所有 © 华为云计算技术有限公司

前提条件

- 已完成配置Python SDK环境并完成了初始化，详情请参见**2.1.6.1 Spark SDK获取与初始化**。
- API和SQL作业相关权限已配置，权限详情请参见**权限及授权项说明**。
- 使用的计算资源和端点为可用状态。

函数

cursor

```
# 同步提交函数
execute();

# 获取SQL查询结果函数
fetchall();
```

样例代码

- 创建连接

```
conn = connect(
    region="your_region",
    project_id="your_project_id",
    workspace_id="your_workspace_id",
    ak="your_access_key",
    sk="your_secret_key",
    endpoint="aidatalake_endpoint",
    database="default",
)
```

- 执行SQL

```
try:
    cursor = conn.cursor()
    cursor.execute("SELECT 1 AS value, 'hello' AS greeting")
    row = cursor.fetchone()
    logger.info("Row: %s", row)
    cursor.close()
```

```
try:
    cursor = conn.cursor()
    cursor.execute(
        "SELECT * FROM demo_table WHERE id = ? AND name = ?",
        [42, "test"],
    )
    rows = cursor.fetchall()
    logger.info("Fetched %d rows", len(rows))
    cursor.close()
```

```
cursor = conn.cursor()

# 获取查询结果
cursor.execute(
    "SELECT * FROM demo_table WHERE id = :id AND name = :name",
    {"id": 42, "name": "test"},
)
for row in cursor:
    logger.info("Row: %s", row)

# List catalogs
catalogs = conn.list_catalogs()
logger.info("Catalogs: %s", catalogs)

# List schemas (databases)
schemas = conn.list_schemas(catalog="default")
logger.info("Schemas in 'default': %s", schemas)
```


版权所有 © 华为云计算技术有限公司

参数说明

表 2-49 参数详情

参数	必须配置	默认值	描述
region	是	无	区域名称。
project_id	是	无	项目ID。
ak	是	无	访问密钥ID。
sk	是	无	访问密钥密码。
endpoint	是	无	Ai DataLake的URL。
database	否	无	LakeFormation数据库名字。
workspace_id	是	无	工作空间ID。
catalog	否	无	LakeFormation Catalog名字。
security_token	否	无	使用临时凭证时使用。
obs_endpoint	否	无	OBS Endpoint。 不设置时默认使用obs.{RegionName}.myhuaweicloud.com)
endpoint_name	否	无	AI DataLake Endpoint。 例如：fabric.{regionName}.myhuaweicloud.com。

2.2 Java SDK 参考

2.2.1 Java 环境配置

操作场景

在安装和使用Java SDK前，确保您已经完成开发环境的基本配置。

Java SDK要求使用JDK1.8或更高版本。考虑到后续版本的兼容性，推荐使用1.8版本。

在Java运行环境配置好的情况下，打开windows的命令行，执行命令**Java -version**，可以检查版本信息。

操作步骤

1. 安装JDK。从**Oracle官网**下载并安装JDK1.8版本安装包。
2. 配置环境变量，在“控制面板”选择“系统”属性，单击“环境变量”。
3. 选择“系统变量”，新建 “JAVA_HOME 变量”，路径配置为JDK安装路径，例如：“D:\Java\jdk1.8.0_45”。
4. 编辑 “Path 变量”，在“变量值”中增加“%JAVA_HOME%\bin;”。
5. 新建 “CLASSPATH 变量”，在“变量值”中填写 “.;%JAVA_HOME%\lib;%JAVA_HOME%\lib\tools.jar”。

版权所有 © 华为云计算技术有限公司

6. 检验是否配置成功，运行cmd ，输入 ***java -version***。运行结果，请参见图2-4，显示版本信息，则说明安装和配置成功。

图 2-4 检验配置是否成功

```
C:\Users\          >java -version
java version "1.8.0_172"
Java(TM) SE Runtime Environment (build 1.8.0_172-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.172-b11, mixed mode)
```

2.2.2 Spark SDK（Java）

2.2.2.1 Spark SDK 的获取与初始化

对AIDataLake服务提供的Spark API的封装，以简化用户的开发工作。用户直接调用SDK提供的接口函数即可实现使用提交Spark SQL和Spark Job作业。本文为您介绍如何进行Spark SDK的获取与初始化。

前提条件

已完成**2.2.1 Java环境配置**。

步骤一：获取 SDK

下载SDK文件，并建议您同时下载.sha256文件，该文件可用于验证SDK包的完整性。验证方法请参考**步骤二：验证SDK包的完整性**。

- **SDK安装包链接**
- **校验文件（SHA256）链接**

步骤二：验证 SDK 包的完整性

下载SDK安装包时，您需要同时下载.sha256文件，用于验证SDK包的完整性。本节介绍使用.sha256文件验证SDK包的完整性的操作步骤。

1. **查询SDK安装包的SHA - 256哈希值**
   - **在Windows系统中**
     i. 在开始菜单中搜索“cmd”打开命令窗口。
     ii. 使用cd命令切换到保存SDK安装包的文件夹目录。
     例如：SDK安装包保存在D盘的“SDK”文件夹中，输入**cd D:\SDK**。
     iii. 执行以下命令
     ```
     certutil -hashfile SDK安装包文件名 SHA256
     ```
     将“SDK安装包文件名”替换为实际的SDK安装包文件名，如aidatalake-spark-sdk-1.0-261030.tar.gz。
     iv. 命令执行后，命令提示符会输出SDK安装包的SHA - 256哈希值。
   - **在Linux系统中**
     i. 使用cd命令切换到保存SDK安装包的文件夹目录。
     例如，如果SDK安装包保存在用户主目录下的“SDK”文件夹中，输入**cd ~/SDK**。

版权所有 © 华为云计算技术有限公司

ii. 输入查询命令：
```
sha256sum SDK安装包文件名
```
将“SDK安装包文件名”替换为实际的SDK安装包文件名，如aidatalake-spark-sdk-1.0-261030.tar.gz。

iii. 命令执行后，终端会输出SDK安装包的SHA - 256哈希值。

2. **对比哈希值**

a. 使用文本编辑器打开下载的.sha256文件。

b. 在.sha256文件中找到SDK安装包的SHA - 256哈希值。

通常是一个长字符串，例如“a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6xxxxxxxxxxxxxxxxxxxxxxxx”。

c. 将**步骤1**查询得到的SDK安装包的SHA - 256哈希值与.sha256文件中的哈希值进行对比。

- 如果两个哈希值完全一致，说明SDK安装包在下载过程中没有被篡改，文件是完整的。
- 如果哈希值不一致，这可能意味着SDK安装包在下载过程中出现了错误，或者文件被恶意修改。

  此时，建议重新下载SDK安装包和对应的.sha256文件，然后再次进行验证。

步骤三：安装 SDK

1. 单击**aidatalake-spark-sdk-x.x.x.jar**下载SDK包。
2. 执行以下命令将jar包安装到本地Maven仓库。使用如下命令安装SDK。
```
mvn install:install-file -Dfile=aidatalake-spark-sdk-x.x.x.jar -DgroupId=com.huawei.aidatalake -DartifactId=aidatalake-spark-sdk -Dversion=x.x.x -Dpackaging=jar
```
3. 在pom.xml中添加依赖。
```
<dependency>
    <groupId>com.huawei.aidatalake</groupId>
    <artifactId>aidatalake-spark-sdk</artifactId>
    <version>x.x.x</version>
</dependency>
```

步骤四：初始化客户端

使用AIDataLake SDK工具访问AIDataLake，需要用户初始化AIDataLake客户端。用户可以使用AK/SK(Access Key ID/Secret Access Key)认证方式初始化客户端。

```
// 初始化客户端上下文
JobClientContext jobContext = new JobClientContext();
jobContext.setRegionName("cn-north-4");
jobContext.setProjectId("xxxx-xxxx");
jobContext.setWorkspaceId("xxx-xxxx");
jobContext.setLakeformationInstanceId("xxxx-xxxx");
jobContext.setEndpointName("xxxx");
// 用ststoken凭证
jobContext.setCredentialProvider(StaticCredentialProvider.create("{accesskey}", "{secretKey}",
"{securityToken}", "{projectId}"));
jobClientContext.setAiDataLakeEndpoint("xxx.xxxx");
jobClientContext.setLakeformationInstanceId("xxx-xxxx");
jobClientContext.setObsEndpoint("xxx.xxxx");

// 初始化AIDataLakeSparkSqlClient spark-sql客户端
AIDataLakeSparkSqlClient sparkSqlClient = new AIDataLakeSparkSqlClient(jobContext);

// 初始化AIDataLakeSparkJobClient spark-job客户端
```


版权所有 © 华为云计算技术有限公司

```
AIDataLakeSparkJobClient sparkJobClient = new AIDataLakeSparkJobClient(jobContext);

// 初始化AIDataLakeLakeFormationClient schema元数据查询客户端
AIDataLakeLakeFormationClient lakeformationClient = new AIDataLakeLakeFormationClient(jobContext);
```

表 2-50 参数详情

参数	必须配置	默认值	描述
RegionName	是	无	区域名称。
ProjectId	是	无	项目ID。
WorkspaceId	是	无	工作空间ID。
LakeformationInstanceId	是	无	工作空间绑定的LakeFormation实例ID。
AccessKey	是	无	访问密钥ID。
SecretKey	是	无	访问密钥密码。
SecurityToken	否	无	使用临时凭证时使用。
EndPointName	是	无	AIDataLake Endpoint, 如：fabric.{regionName}.myhuaweicloud.com
ObsEndpoint	否	无	OBS Endpoint(不设置默认使用: obs.{RegionName}.myhuaweicloud.com)

2.2.2.2 Spark SQL 作业提交

功能说明

本节操作介绍使用AI DataLake SDK提交SQL作业的操作方法。

前提条件

- 已完成配置Java SDK环境并完成了初始化，详情请参见**2.2.2.1 Spark SDK的获取与初始化**
- API和SQL作业相关权限已配置，权限详情请参见**权限及授权项说明**。
- 使用的计算资源和端点为可用状态。

函数

SparkSqlJob

```
# 同步提交函数
submit();
```


版权所有 © 华为云计算技术有限公司

```
# 异步提交函数
asyncSubmit();
```

样例代码

- 异步提交SQL

```
public static void asyncSubmit(AIDataLakeSparkSqlClient sparkSqlClient) throws AIDataLakeException
{
    String databaseName = "DATABASE_NAME";
    String catalogName = "CATALOG_NAME";
    String sqlStatement = "SELECT xxx";
    Map<String, String> sparkConfig = new HashMap<>();
    // spark conf starting with "spark."
    sparkConfig.put("spark.sql.shuffle.partitions", "200");
    SparkSqlJob sqlJob = sparkSqlClient.newSparkSqlJobBuilder(sqlStatement)
                    .catalogName(catalogName)
                    .databaseName(databaseName)
                    .sparkConfig(sparkConfig)
                    .build();
    sqlJob.asyncSubmit();
}
```

- 同步提交SQL

```
public static void submit(AIDataLakeSparkSqlClient sparkSqlClient) throws AIDataLakeException {
    String databaseName = "DATABASE_NAME";
    String catalogName = "CATALOG_NAME";
    String sqlStatement = "SELECT xxx";
    Map<String, String> sparkConfig = new HashMap<>();
    // spark conf starting with "spark."
    sparkConfig.put("spark.sql.shuffle.partitions", "200");
    SparkSqlJob sqlJob = sparkSqlClient.newSparkSqlJobBuilder(sqlStatement)
                    .catalogName(catalogName)
                    .databaseName(databaseName)
                    .sparkConfig(sparkConfig)
                    .build();
    sqlJob.submit();
}
```

- SQL查询

```
public static void submit(AIDataLakeSparkSqlClient sparkSqlClient) throws AIDataLakeException {
    String databaseName = "DATABASE_NAME";
    String catalogName = "CATALOG_NAME";
    String sqlStatement = "SELECT xxx";
    String statementId;
    Map<String, String> sparkConfig = new HashMap<>();
    // spark conf starting with "spark."
    sparkConfig.put("spark.sql.shuffle.partitions", "200");
    SparkSqlJob sqlJob = sparkSqlClient.newSparkSqlJobBuilder(sqlStatement)
                    .catalogName(catalogName)
                    .databaseName(databaseName)
                    .sparkConfig(sparkConfig)
                    .build();
    sqlJob.submit();
    statementId = sqlJob.getStatementId();
    ResultSet resultSet = sparkSqlClient.getResultsWithStatementId(statementId).getResultSet();
    while (resultSet.hasNext()) {
        Row row = resultSet.read();
        List<Column> schema = row.getSchema();
        List<Object> record = row.getRecord();
        for (int i = 0; i < schema.size(); i++) {
            DataType.TypeName typeName =
DataType.TypeName.fromName(schema.get(i).getType().getName());
            switch (typeName) {
                case STRING:
                    String strV = (String) record.get(i);
                    break;
                case DATE:
                    Date dtV = (Date) record.get(i);
                    break;
```


版权所有 © 华为云计算技术有限公司

```
        }
      }
    }
}
```

参数说明

表 2-51 参数详情

参数	必须配置	默认值	描述
databaseName	是	无	默认LakeFormation数据库名称。
catalogName	是	无	默认LakeFormation Catalog名称。
sqlStatement	是	无	待提交的SQL语句。

2.2.2.3 Spark Job 作业提交

功能说明

本节操作介绍使用AI DataLake SDK提交Spark作业的操作方法。

前提条件

- 使用的计算资源和端点为可用状态。
- API和SQL作业相关权限已配置，权限详情请参见**权限及授权项说明**。

函数

- SparkJarJob

```
# 同步提交函数
submit();

# 异步提交函数
asyncSubmit();
```

- SparkSqlScriptingJob

```
# 同步提交函数
submit();

# 异步提交函数
asyncSubmit();
```

- SparkPythonJob

```
# 同步提交函数
submit();

# 异步提交函数
asyncSubmit();
```


版权所有 © 华为云计算技术有限公司

样例代码

- 同步提交SparkJar作业

```
public static void submit(AIDataLakeSparkJobClient sparkJobClient) throws AIDataLakeException {
    String jobName = "JOB_NAME";
    String endpointName = "ENDPOINT_NAME";
    String catalogName = "CATALOG_NAME";
    // set spark configs
    Map<String, String> config = new HashMap<>();
    config.put("key", "value");

    SparkJarJob sparkJob = sparkJobClient.newSparkJarJobBuilder(jobName, "com.example.Main",
"obs://app.jar")
                    .endpointName(endpointName)
                    .jobAgency("agency")
                    .catalogName(catalogName)
                    .mainArgs(Arrays.asList("arg1"))
                    .dependencyJars(Arrays.asList("obs://lib.jar"))
                    .sparkConfig(config)
                    .driverCpu(2000)
                    .driverMemory(4096)
                    .executorCpu(4000)
                    .executorMemory(8192)
                    .executorNumber(4)
                    .build();
    sparkJob.submit();
}
```

表 2-52 参数详情

参数	必须配置	默认值	描述
catalogName	否	无	默认LakeFormation Catalog名称。 如果作业内执行Spark.sql可选择设置。
endpointName	是	无	SparkSQL端点名称。
mainArgs	否	无	主类Main参数
dependencyJars	否	无	依赖的Jars
jobAgency	否	无	作业IAM委托，用户访问OBS或Lakeformation。
sparkConfig	否	无	Spark参数。
driverCpu	否	无	Driver POD豪核数。
driverMemory	否	无	Driver POD内存MB。
executorCpu	否	无	Executor POD豪核数。
executorMemory	否	无	Executor POD内存MB。

版权所有 © 华为云计算技术有限公司

参数	必须配置	默认值	描述
executorNumber	否	无	Executor实例个数。

- 异步提交SparkSqlScript作业

```
public static void submit(AIDataLakeSparkJobClient sparkJobClient) throws AIDataLakeException {
    String jobName = "JOB_NAME";
    String endpointName = "ENDPOINT_NAME";
    String catalogName = "CATALOG_NAME";
    // set spark configs
    Map<String, String> config = new HashMap<>();
    config.put("key", "value");

    SparkSqlScriptingJob sparkScriptingJob = sparkJobClient.newSparkSqlScriptingJobBuilder(jobName)
                .endpointName(endpointName)
                .sqlScriptingFileUri("obs://xxx")
                .catalogName(catalogName)
                .jobAgency("agency")
                .sparkConfig(config)
                .driverCpu(2000)
                .driverMemory(4096)
                .executorCpu(4000)
                .executorMemory(8192)
                .executorNumber(4)
                .sqlScriptingResultToObs(true)
                .build();
    sparkJob.asyncSubmit();
}
```

表 2-53 参数详情

参数	必须配置	默认值	描述
catalogName	否	无	默认LakeFormation Catalog名称。 如果作业内执行Spark.sql可选择设置。
endpointName	是	无	SparkSQL端点名称。
jobAgency	否	无	作业IAM委托，用户访问OBS或Lakeformation。
sqlScriptingFileUri	是	无	SqlScripting OBS脚本路径。
sparkConfig	否	无	Spark参数。
driverCpu	否	无	Driver POD豪核数。
driverMemory	否	无	Driver POD内存MB。
executorCpu	否	无	Executor POD豪核数。

版权所有 © 华为云计算技术有限公司

参数	必须配置	默认值	描述
executorMemory	否	无	Executor POD内存MB。
executorNumber	否	无	Executor实例个数。

- 同步提交SparkPython作业

```
public static void submit(AIDataLakeSparkJobClient sparkJobClient) throws AIDataLakeException {
    String jobName = "JOB_NAME";
    String endpointName = "ENDPOINT_NAME";
    String catalogName = "CATALOG_NAME";

    return sparkJobClient.newSparkPythonJobBuilder(jobName, "obs://xxx/job.py")
            .endpointName(endpointName)
            .catalogName(catalogName)
            .jobAgency("agency")
            .sparkConfig(config)
            .driverCpu(1000)
            .driverMemory(1024)
            .driverDisk(20)
            .executorCpu(1000)
            .executorMemory(1024)
            .build();
    sparkJob.submit();
}
```

表 2-54 参数详情

参数	必须配置	默认值	描述
catalogName	否	无	默认LakeFormation Catalog名称。 如果作业内执行Spark.sql可选择设置。
endpointName	是	无	SparkSQL端点名称。
jobAgency	否	无	作业IAM委托，用户访问OBS/Lakeformation
sparkConfig	否	无	Spark参数
driverCpu	否	无	Driver POD豪核数
driverMemory	否	无	Driver POD 内存MB
executorCpu	否	无	Executor POD豪核数
executorMemory	否	无	Executor POD 内存MB

版权所有 © 华为云计算技术有限公司

参数	必须配置	默认值	描述
executorNumber	否	无	Executor实例个数

2.2.2.4 Spark 库表元数据查询

功能说明

本节操作介绍使用AI DataLake SDK查询库表等元数据。

前提条件

LakeFormationAPI和相关权限已配置，权限详情可参考**权限和授权项**。

函数

AIDataLakeLakeFormationClient

```
# 列举Catalogs
listCatalogs();

# 列举Databases
listDatabases("yourCatalogName");

# 列举Tables
listTables("yourCatalogName", "yourDBName");
```

参数	必须配置	默认值	描述
yourCatalogName	否	无	数据目录名称。
yourDBName	是	无	数据库名称。

样例代码

- 查询Catalog

```
public static void listCatalogs(AIDataLakeLakeFormationClient lakeformationClient) {
    Catalogs catalogs = lakeformationClient.listCatalogs();
    for (Catalog item : catalogs) {
        System.out.println(item.getCatalogName());
    }
    Catalog catalog = catalogs.getCatalog("yourCatalogName")
}
```

- 查询数据库

```
public static void listDatabases(AIDataLakeLakeFormationClient lakeformationClient) {
    Databases databases = lakeformationClient.listDatabases("yourCatalogName");
    for (Database item : databases) {
        System.out.println(item.getDatabaseName());
    }
    Database database = databases.getDatabase("yourDBName")
}
```


版权所有 © 华为云计算技术有限公司

- 查询表

```
public static void listTables(AIDataLakeLakeFormationClient lakeformationClient) throws
AIDataLakeException {
    Tables tables = lakeformationClient.listTables("yourCatalogName", "yourDBName");
    for (Table item : tables) {
        System.out.println(item.getTableName());
    }
    // get table
    Table table = tables.getTable("yourTableName");
    // filter table
    TableFilter tableFilter = new TableFilter();
    tableFilter.setTblPattern("table_name_xxx");
    tableFilter.setTblPattern("table_type_xxx");
    Iterable<Table> tableA = tables.iterableWithDatabaseAndFilter("yourDBName", tableFilter);
    // get table meta
    TableMeta tableMeta = tables.getTableMeta("yourTableName");
}
```


版权所有 © 华为云计算技术有限公司