ALIBABA CLOUD

阿里云

数据湖分析

Serverless Spark

文档版本：20220711

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
Courier字体	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

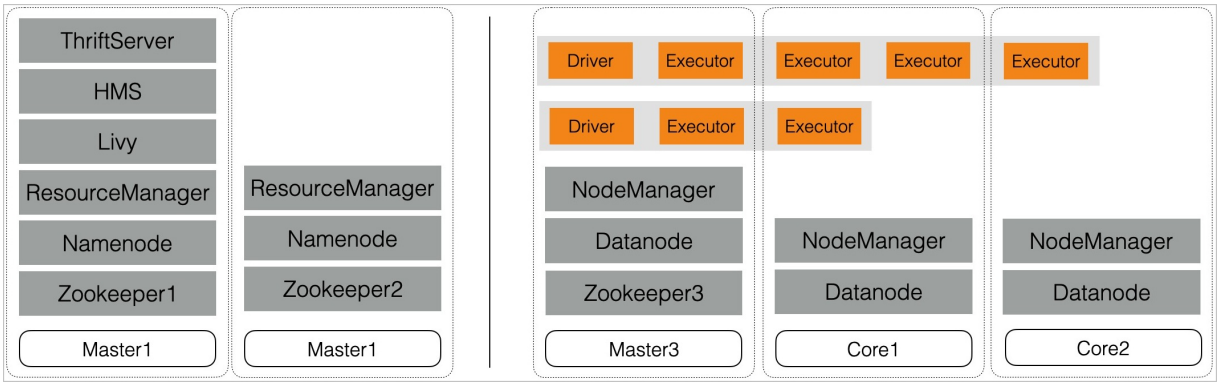
目录

1.Serverless Spark概述

DLA Spark基于云原生架构，提供面向数据湖场景的数据分析和计算。开通DLA服务后，您只需简单的配置，就可以提交Spark作业，无需关心Spark集群部署。

传统开源Spark集群版面临的挑战

Spark是大数据领域十分流行的引擎，面向数据湖场景，Spark本身内置的数据源连接器，可以很方便的扩展接口。Spark既支持使用SQL，又支持编写多种语言的DataFrame代码，兼具易用性和灵活性。Spark一站式的引擎能力，可以同时提供SQL、流、机器学习、图计算的能力。传统Spark集群版的方案架构图如下所示：

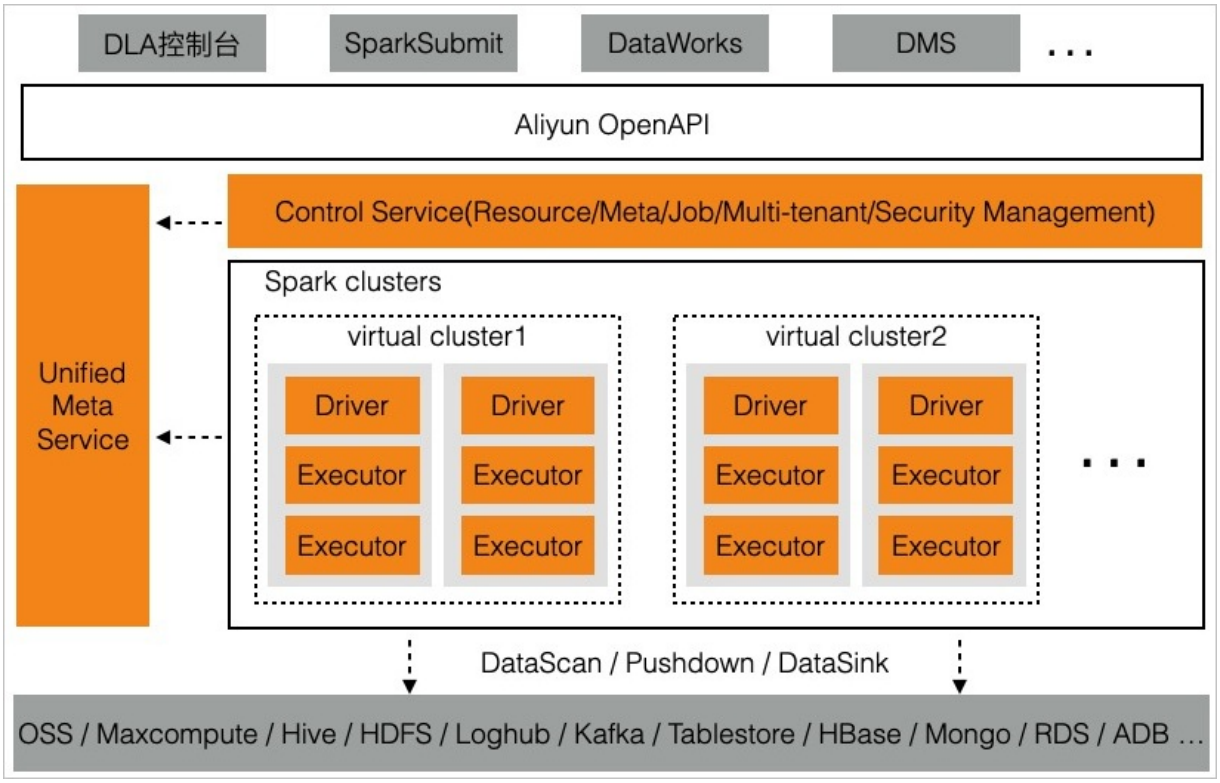


但是对于传统Spark集群版，用户首先需要部署一套开源大数据基础组件：Yarn、HDFS、Zookeeper等，可能会存在以下问题：

- **使用门槛高**：开发者需要同时熟悉多种大数据组件，才能完成开发与运维相关工作，如果遇到疑难问题，还要去深入研究社区源码。
- **运维成本高**：企业往往需要一个运维团队，以运维多套开源组件。运维团队的职责包括配置资源节点、配置和部署开源软件、监控开源组件、开源组件升级、集群扩缩容等。典型的，为了满足企业级需求，比如权限隔离、监控报警等，还需要做定制化开发。
- **资源成本高**：Spark作业负载往往具备波峰波谷的特点，在低峰期，Spark集群的空闲资源是浪费的；Spark集群的管控组件（比如，集群Master节点，Zookeeper、Hadoop等后台进程的资源开销等），是不会对用户产生价值的，属于额外开销。
- **弹性能力不足**：在业务高峰期，企业往往需要能够准确预估资源需求，并及时扩容机器。如果扩容过多，则会存在资源浪费，如果扩容过少，则会影响业务。而且集群扩容过程较复杂，时间也会较长，并且云上容易出现资源库存不足的问题。

解决方案

Serverless Spark是云原生数据湖团队基于Apache Spark打造的服务化的大数据分析与计算服务。方案架构图如下所示：



Serverless Spark将Spark、Serverless、云原生技术，深度整合到一起，相对于传统开源Spark集群版方案，具体以下优势：

- **入门门槛低**：Serverless Spark屏蔽掉了底层的基础组件，提供简单的API、脚本以及控制台使用方式。开发者只需要了解开源Spark的使用方式就可以进行大数据业务开发。
- **0运维**：用户只需通过产品接口管理Spark作业即可，无需关心服务器配置以及Hadoop集群配置，无需扩缩容等运维操作。
- **作业级细粒度的弹性能力**：Serverless Spark按照Driver和Executor的粒度创建资源，相比于集群版的计算节点，粒度要细很多，粒度细的好处是库存不足的问题会大大降低。支持秒级拉起，目前每分钟可以拉起500~1000个计算节点，可以快速响应业务资源需求。
- **更低成本**：每作业，每账单，不使用不收费。用户无需为管控资源付费，无需为低峰期空闲的计算资源付费。
- **良好的性能**：云原生数据湖团队对Spark引擎做了深度定制和优化，特别是针对云产品，比如OSS，典型场景下，性能可以提升3~5倍。

  > **说明** 关于进一步的性价比对比数据，您可以参考性价比白皮书中的测试结果。

- **企业级能力**：Serverless Spark跟Serverless Presto共用元数据，可以通过GRANT、REVOKE语句对子用户进行权限管理。提供服务化的UI服务，相较于社区HistoryServer方式，无论作业多复杂，运行时间多长，都可以秒级打开。

基本概念

- **虚拟集群（Virtual Cluster）**

Serverless Spark采用多租户模式，Spark进程运行在安全隔离的环境中，虚拟集群是资源隔离和安全隔离的单元。区别于传统实体集群，虚拟集群中没有固定的计算资源，您无需配置和维护计算节点，只需根据实际业务需要分配资源额度和配置待访问目标数据所在的网络环境。同时，虚拟集群也可以配置默认的Spark作业参数，方便您统一管理Spark作业。关于如何创建虚拟集群，请参考创建虚拟集群。

- **计算单元CU（Compute Unit）**

  CU主要是Serverless Spark计量的基本单元，1CU=1 vCPU 4GB Memory。作业结束后，DLA会按照Driver和Executor实际使用的总和CU * 时，进行计量。具体的计费信息，请参考计费方式概述。

- **资源规格（Resource Specification）**

  Serverless Spark底层采用阿里云轻量虚拟机ECI，ECI跟ECS类似，都具备规格，DLA平台对具体的ECI规格进行了屏蔽、简化，用户只需要配置small、medium、large这样的简单配置即可，平台在调度的时候会优先使用高性能计算资源。

资源规格	计算资源	消耗的CU数
c.small	1Core 2GB	0.8CU
small	1Core 4GB	1CU
m.small	1Core 8GB	1.5CU
c.medium	2Core 4GB	1.6CU
medium	2Core 8GB	2CU
m.medium	2Core 16GB	3CU
c.large	4Core 8GB	3.2CU
large	4Core 16GB	4CU
m.large	4Core 32GB	6CU
c.xlarge	8Core 16GB	6.4CU
xlarge	8Core 32GB	8CU
m.xlarge	8Core 64GB	12CU
c.2xlarge	16Core 32GB	12.8CU
2xlarge	16Core 64GB	16CU
m.2xlarge	16Core 128GB	24CU

开始使用Serverless Spark

您可以参考DLA Spark快速入门来提交您的第一个Spark作业。

如果您需要访问数据源，请参考连接数据源目录下面的文档。

如果您对时空计算有需求，请参考数据湖时空引擎Ganos。

如果您想联系我们做进一步交流，请参考专家服务。

2.专家服务

如果您想对DLA Spark有进一步的了解，或者有任何疑问，除了官网文档外，您还可以通过以下途经：

- 关注云原生数据湖的开发者社区。
- 加入我们的技术专家服务群（搜索钉钉群号『33444627』或扫下方二维码进群）。如果您在使用过程中遇到困难，请在群内提问。

3.权限配置

3.1. 快速配置子账号权限

本文主要介绍如何快速配置RAM子账号权限并提交DLA Spark作业。

前提条件

- 已创建RAM子账号。具体请参见创建RAM用户。
- 已创建DLA子账号。具体请参见管理DLA账号。

背景信息

目前DLA Spark的权限分为三部分。

- DLA控制接口的访问权限：用于控制RAM子账号是否允许登录DLA控制台，是否允许RAM子账号调用Spark作业管理相关API。具体操作请参见操作步骤1。
- DLA表的访问权限：DLA表的访问权限默认通过DLA账号进行权限管理，Spark作业通过RAM子账号提交。Spark作业如果要访问表，需要将DLA账号跟RAM账号绑定。具体操作请参见操作步骤2。
- Spark作业所依赖其他资源的访问权限：用于控制RAM子账号所提交Spark作业所依赖的JAR包、非DLA表的其他数据源，比如直接访问OSS目录等。具体操作请参见操作步骤3。

操作步骤

1. 登录RAM控制台，为RAM子账号授予DLA访问权限。具体请参见为RAM用户授权。

   当前RAM系统中已经预设了三种DLA授权策略，在**系统策略**输入框中输入DLA就可以快速选择。

   权限策略的说明如下。

权限名称	权限说明
AliyunDLAFullAccess	用于数据湖分析的管理员权限，拥有数据湖分析的所有权限，可以执行新建集群、删除集群、提交作业等操作，拥有授权给DLA服务的角色的使用权。
AliyunDLAReadOnlyAccess	用于数据湖分析的访客权限，拥有数据湖分析的只读权限，可以查看集群状态、作业状态等信息。无法修改集群的状态，也无法提交作业。
AliyunDLADeveloperAccess	用于数据湖分析的开发者权限，可以查看集群、作业的状态，提交和执行作业，无法新建和删除集群，拥有授权给DLA账户的角色的使用权。

2. DLA子账号绑定RAM子账号。具体请参见DLA子账号绑定RAM账号。
3. 单击快速授权链接为RAM子账户快速授予资源访问权限。

   该操作步骤自动帮您创建**AliyunDLASparkProcessingDataRole**角色，该角色包含用户账号下所有OSS Bucket的读写权限。

> **说明** 上述3个步骤均是必选的，否则作业会报权限错误。

验证RAM子账号权限配置

当您完成以上操作后，即可使用RAM子账号登录Data Lake Analytics管理控制台在**Serverless Spark > 作业管理**页签下，提交Spark作业验证RAM子账号权限配置是否正确。具体请参见创建和执行Spark作业和作业配置指南配置示例如下。

```
{
    "name": "SparkPi",
    "file": "local:///tmp/spark-examples.jar",
    "className": "org.apache.spark.examples.SparkPi",
    "args": [
        "100"
    ],
    "conf": {
        "spark.driver.resourceSpec": "medium",
        "spark.executor.instances": 1,
        "spark.executor.resourceSpec": "medium"
    }
}
```

> **说明** 如果您在conf中没有填写spark.dla.roleArn配置信息时，系统会默认使用AliyunDLASparkProcessingDataRole，您也可以自定义roleArn。具体请参见细粒度配置RAM子账号权限。

3.2. 细粒度配置RAM子账号权限

本文详细介绍了如何配置RAM子账号权限来使用DLA Serverless Spark提交作业。

前提条件

- 已创建RAM子账号。具体请参见创建RAM用户。
- 已创建DLA子账号。具体请参见管理DLA账号。
- DLA子账号绑定RAM子账号。具体请参见DLA子账号绑定RAM账号。

操作步骤

1. 登录RAM控制台为子账号授予访问DLA的权限。

   如何为RAM子账号赋权，请参考为RAM用户授权。

   权限策略的说明如下：

权限策略名称	权限策略说明
AliyunDLAFullAccess	用于数据湖分析的管理员权限，拥有数据湖分析的所有权限，可以执行新建集群、删除集群、提交作业等操作，停止其它账号的作业等操作。拥有授权给DLA账户的角色的使用权。
AliyunDLAReadOnlyAccess	用于数据湖分析的访客权限，拥有数据湖分析的只读权限，可以查看集群状态、作业状态， 作业日志等信息。无法修改集群的状态，也无法提交作业。

权限策略名称	权限策略说明
AliyunDLADeveloperAccess	用于数据湖分析的开发者权限，可以查看集群、作业的状态，提交和执行作业，只能终止自己提交的作业，无法新建和删除集群，拥有授权给DLA账户的角色的使用权。

> **说明** 在RAM访问控制中，您还可以根据需要自定义创建更细粒度的访问策略，比如一个子账号可以使用哪些集群。具体请参见授予RAM账号细粒度访问DLA的权限。创建完权限策略后，您便可添加该自定义策略给该RAM子账号。

2. 为了让Spark作业可以访问您的数据，需要为DLA创建一个数据访问的角色，并授权子用户可以使用该角色。

   用户角色是RAM用户控制中的概念，具体请参考RAM角色概览。

   i. 登录RAM控制台，选择**RAM角色管理->创建RAM角色->选择类型**选择**阿里云服务**，点击**下一步**。

   ii. 在**配置角色**页面，填写**角色名称**（本例中为dla-sub-user-role），**选择受信服务**，选择**数据湖分析**，点击**完成** 。

   iii. 在**创建完成**界面，点击**为角色授权**，可以为该角色授予访问您的资源的访问权限。如授予OSSFullAccess意味着这个角色可以访问您的所有OSS数据。

   > **说明** 您也可以选择自定义权限策略做更细粒度授权，具体请参考RAM Policy概述。创建完细粒度权限策略后，便可在上图中的自定义策略中给该角色添加自定义权限策略。

   iv. 在**RAM角色管理**页面，搜索刚创建好的角色，点击角色名字，查看角色基本信息。

   > **说明** 有两个信息需要注意：**最大会话时间**可以调整，最大可以调整为12小时；ARN是该Role的ID，需要记录下来，后面的步骤中会用到。

3. 授权允许子账号（本例中子账号为dla-sub-user）可以使用步骤2.iii中角色的权限策略。

   i. 登录RAM控制台，选择**权限管理->权限策略管理->创建权限策略->新建自定义权限策略**，填写策略名称（本例中为dla-sub-user-auth），**配置模式**选择**脚本配置**。

ii. 在**策略内容**中输入如下代码。

```
{
    "Statement": [
        {
            "Action": "ram:PassRole",
            "Resource": "<步骤2.d中记录的ARN的值>",
            "Effect": "Allow",
            "Condition": {
                "StringEquals": {
                    "acs:Service": "openanalytics.aliyuncs.com"
                }
            }
        }
    ],
    "Version": "1"
}
```

4. 把步骤3中创建的权限策略（本例中为dla-sub-user-auth）授权给RAM子账号（本例中子账号为dla-sub-user），对应的RAM子账号便拥有了权限策略中的权限。操作步骤如下：

   登录RAM控制台，选择**人员管理->用户->权限管理->自定义策略**，选择dla-sub-user-auth。

验证RAM子账号权限配置

RAM子账号在提交任务时需要在 `conf` 参数中增加 `spark.dla.roleArn` 。这个值就是步骤2.iv中创建的角色的**ARN**值。示例如下：

```
{
    "name": "SparkPi",
    "file": "oss://sparkRoleTest/original-spark-examples_2.11-2.4.5.jar",
    "className": "org.apache.spark.examples.SparkPi",
    "args": [
        "10"
    ],
    "conf": {
        "spark.dla.roleArn": "acs:ram::xxxxxx:role/dla-sub-user-role"
        "spark.driver.resourceSpec": "small",
        "spark.executor.instances": 2,
        "spark.executor.resourceSpec": "small"
    }
}
```

该作业在运行时就拥有了步骤2.iv中所创建角色的权限。

3.3. 配置RAM子账号跨账号访问OSS

本文主要介绍RAM子账号如何访问其他账号的OSS资源并提交Spark作业。

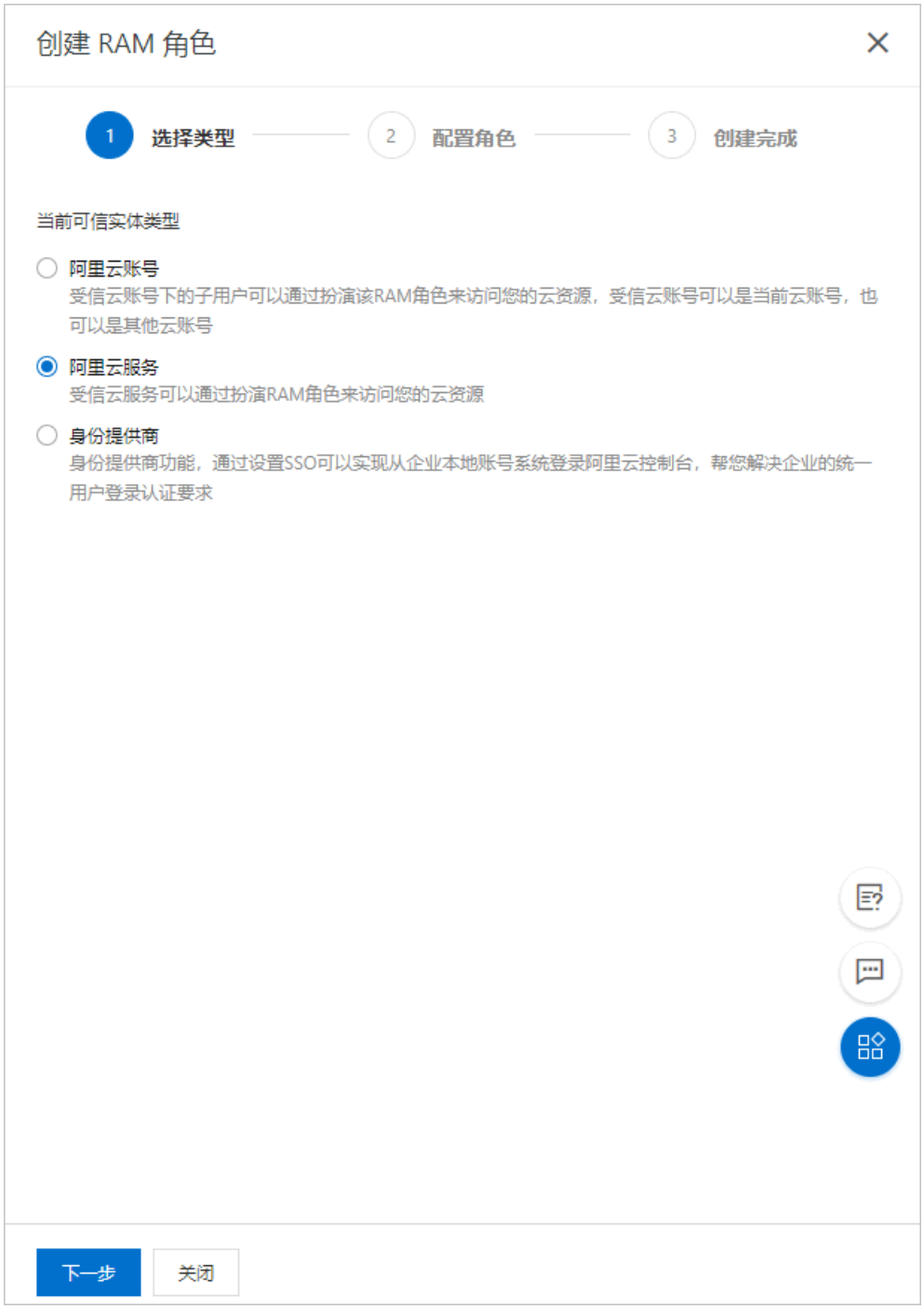
前提条件

- 您需要准备2个阿里云账号。假设提交Spark作业的是A账号，要访问B账号的OSS资源。为了您操作的便捷性，建议您使用两个浏览器来进行操作，一个浏览器登录A账号，一个浏览器登录B账号。
- A账号下的RAM子账号已经可以访问A账号下的所有资源。具体操作请参见快速配置子账号权限或细粒度

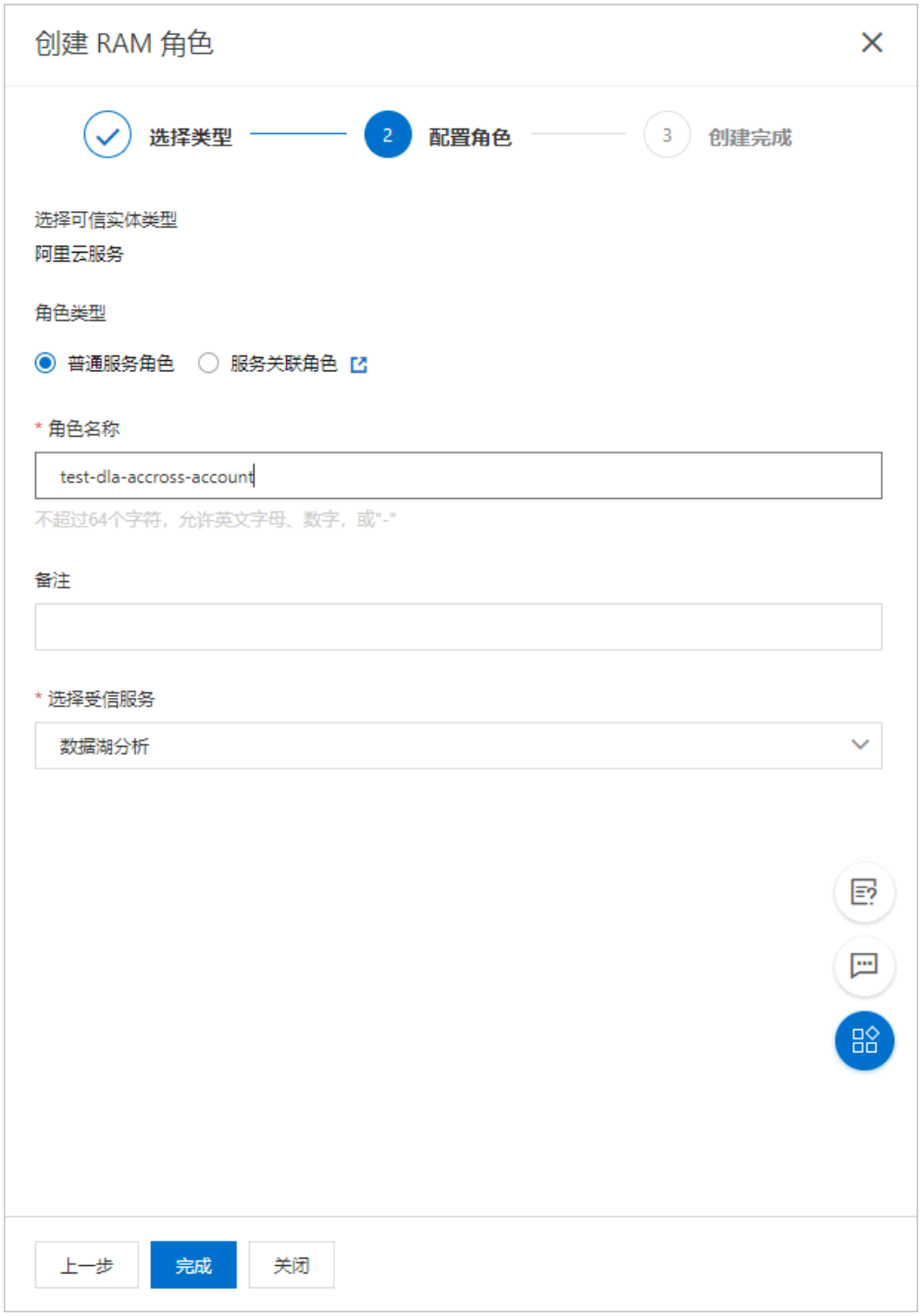
配置RAM子账号权限。

操作步骤

1. 使用B账号登录RAM控制台创建RAM角色。
   i. 登录RAM控制台，在左侧导航栏单击**RAM角色管理**。
   ii. 在**RAM角色管理**页面，单击**创建RAM角色**。
   iii. 在**创建RAM角色**页面的**选择类型**区域，选择**阿里云服务**，单击**下一步**。



iv. 在**创建RAM角色**页面的**配置角色**区域，选择**普通服务角色**，编辑**角色名称**（本例中角色名称为test-dla-accross-account），**受信服务**选择**数据湖分析**。单击**完成**。

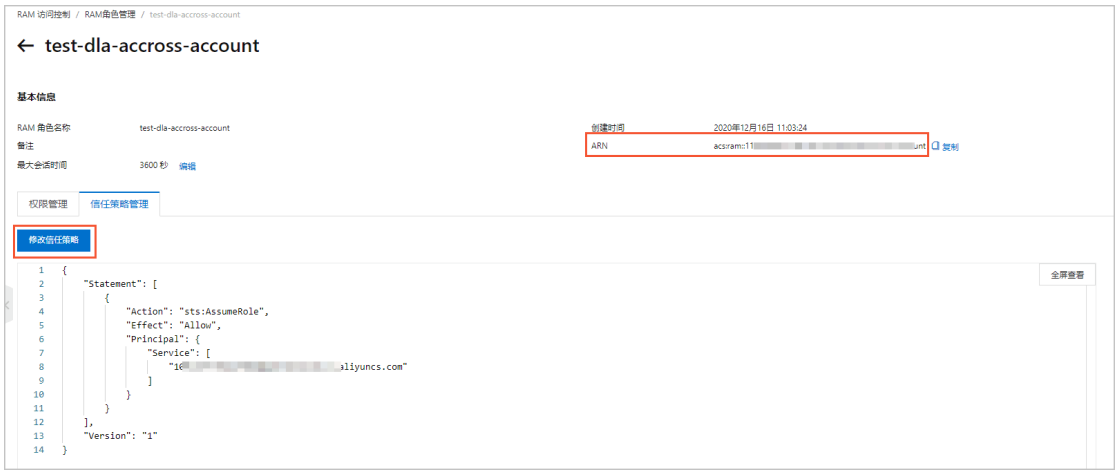


2. 使用B账号登录RAM控制台修改新创建的RAM角色（本例中角色名称为test-dla-accross-account）的权限策略，并为该角色添加访问OSS的权限。

   i. 登录RAM控制台，在左侧导航栏单击**RAM角色管理**。

ii. 在**RAM角色管理**页面的**RAM角色名称**列，定位到新创建的RAM角色（本例中角色名称为test-dla-accross-account），单击RAM角色名称链接。

iii. 在**信任策略管理**页签，单击**修改信任策略**。



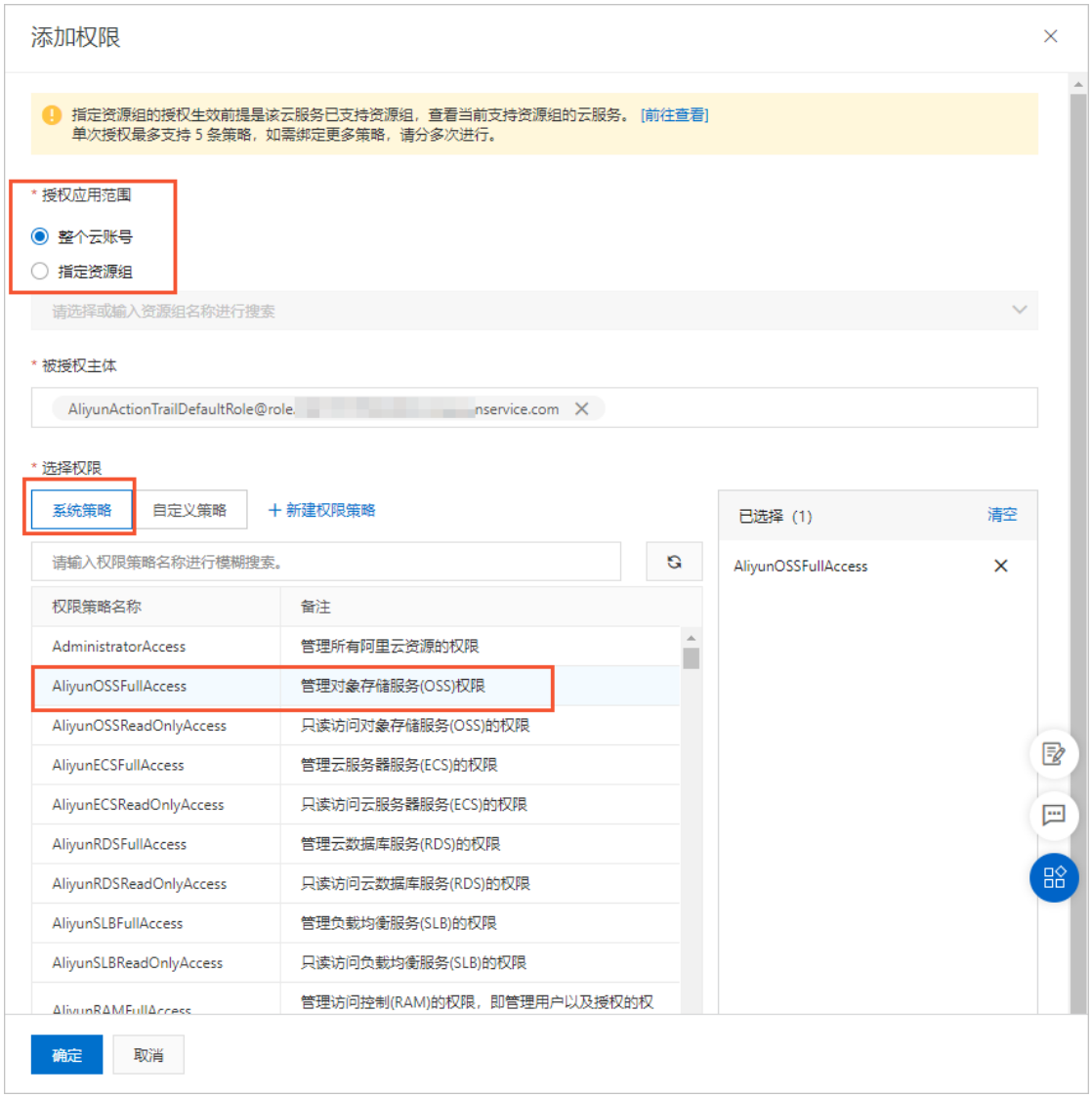
> **说明** 上述截图中B账号的**ARN**值，会在RAM角色创建成功后生成，在步骤3中会用到。

iv. 在**修改信任策略**页面，修改权限策略。如下所示：

```
{
    "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "<这里需要替换为A账号的UID>@openanalytics.aliyuncs.com"
                ]
            }
        }
    ],
    "Version": "1"
}
```

v. 权限策略修改完成后，在**权限管理**页签，单击**添加权限**。

vi. 在**添加权限**页面，**授权范围**选择**整个云账号**，**系统策略**选择**AliyunOSSFullAccess**，单击**确定**。



3. 使用A账号登录RAM控制台新建自定义权限策略。
   i. 登录RAM控制台，在左侧导航栏单击**权限管理 > 权限策略管理**。
   ii. 在**权限策略管理**页面，单击**创建权限策略**。
   iii. 在**新建自定义权限策略**页面，输入**策略名称**（本例中策略名称为test-dla-accross-b-oss），选择**脚本配置**模式，并输入以下**策略内容**。单击**确定**。

策略内容：

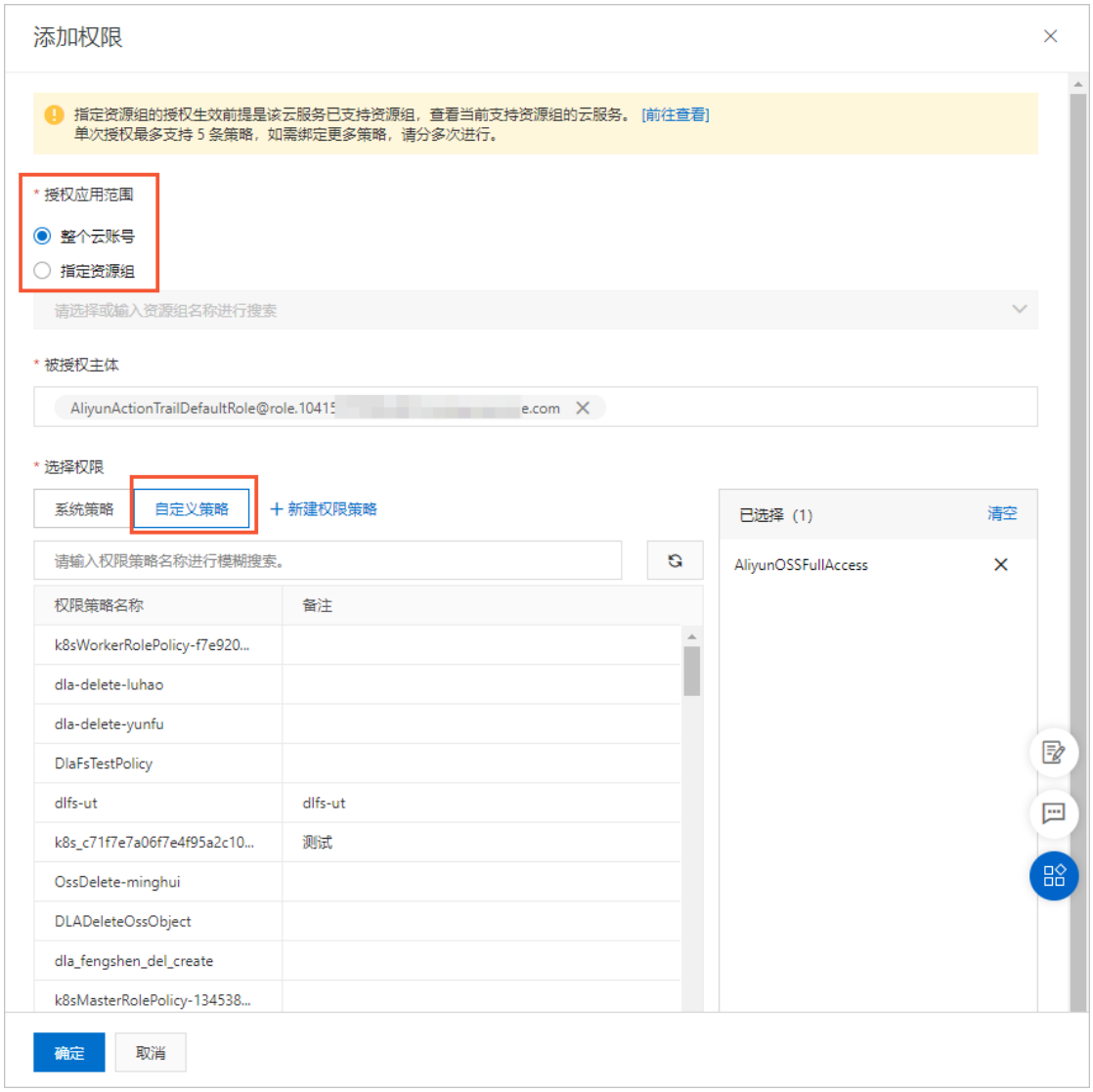
```
{
    "Statement": [
        {
            "Action": "ram:PassRole",
            "Resource": "<这里填写您在步骤2中创建的B账号的ARN>",
            "Effect": "Allow",
            "Condition": {
                "StringEquals": {
                    "acs:Service": "openanalytics.aliyuncs.com"
                }
            }
        }
    ],
    "Version": "1"
}
```

4. 使用A账号登录RAM控制台为A账号下的RAM子账号，添加步骤3创建的权限策略。
   i. 登录RAM控制台，在左侧导航栏单击**人员管理 > 用户**。
   ii. 在**用户**页面，定位到需要添加权限策略的RAM子账号，单击RAM子账号链接。

iii. 在RAM子账号详情页面，单击**权限管理**。

iv. 在**权限管理**页签，单击**添加权限**。

v. 在**添加权限**页签，**授权范围**选择**云账号全部资源**，**自定义策略**选择步骤3创建的权限策略（test-dla-accross-b-oss）。单击**确定**。



验证RAM子账号跨账号访问OSS配置

账号A的RAM子账号在提交Spark任务时，需要在*conf*参数中增加*spark.dla.roleArn*配置，*spark.dla.roleArn*的值就是步骤2中B账号的**ARN**值。示例如下：

```
{
    "name": "<作业名称>",
    "file": "<oss://path/to/your/jar>",
    "className": "<mainclass>",
    "args": [
        "作业参数1",
        "作业参数2"
    ],
    "conf": {
        "spark.dla.roleArn": "acs:ram::xxxxxx:role/test-dla-accross-account"
        "spark.driver.resourceSpec": "small",
        "spark.executor.instances": 2,
        "spark.executor.resourceSpec": "small"
    }
}
```

4.开发指南

4.1. 创建和执行Spark作业

本文介绍如何在数据湖分析控制台创建和执行Spark作业。

准备事项

- 您需要在提交作业之前先创建虚拟集群。

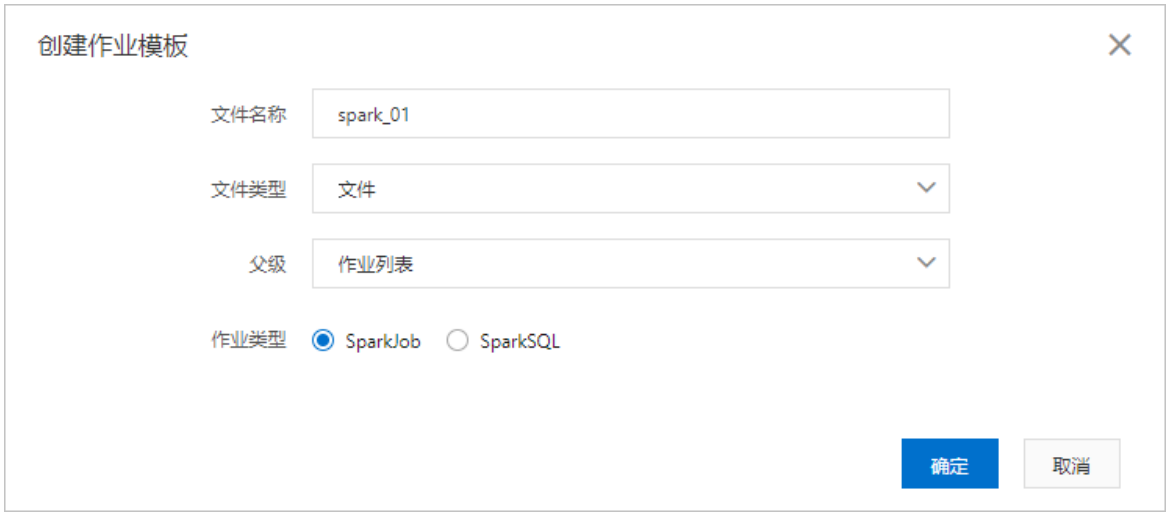
  > **说明** 创建虚拟集群时注意选择引擎类型为Spark。

- 如果您是子账号登录，需要配置子账号提交作业的权限，具体请参考细粒度配置RAM子账号权限。由于SparkPi不需要访问外部数据源，您只需要配置文档中的前两个步骤：”DLA子账号关联RAM子账号“和”为子账号授予访问DLA的权限“。

操作步骤

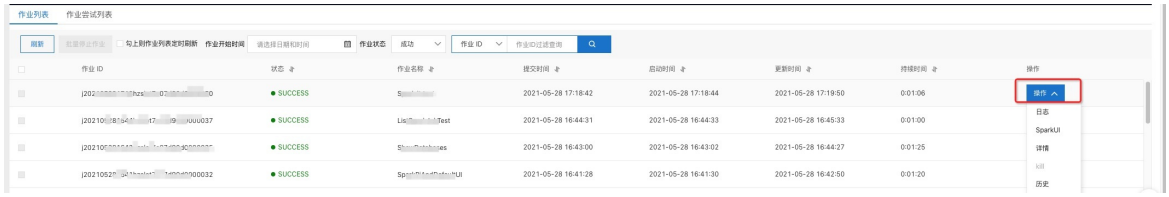
1. 登录Data Lake Analytics管理控制台。
2. 页面左上角，选择DLA所在地域。
3. 单击左侧导航栏中的**Serverless Spark -> 作业管理**。
4. 在**作业编辑**页面，单击**创建作业模板**。
5. 在**创建作业模板**页面，按照页面提示进行参数配置。

参数名称	参数说明
文件名称	设置文件或者文件夹的名字。文件名称不区分大小写。
文件类型	可以设置为文件或者文件夹。
父级	设置文件或者文件夹的上层目录。 ○ 作业列表相当于根目录，所有的作业都在作业列表下创建。 ○ 您可以在作业列表下创建文件夹，然后在文件夹下创建作业；也可以直接在作业列表根目录下创建作业。
作业类型	您可以选择为SparkJob或SparkSQL。 ○ SparkJob：Python/Java/Scala类型的Spark作业，需要填写JSON配置作业。 ○ SparkSQL：SQL类型的Spark配置， 通过set命令配置作业，详情请就参见Spark SQL。



6. 完成上述参数配置后，单击**确定**创建Spark作业。
7. 创建Spark作业后，您可以根据作业配置指南编写Spark作业。
8. Spark作业编写完成后，您可以进行以下操作：
   - 单击**保存**，保存Spark作业，便于后续复用作业。
   - 单击**执行**，执行Spark作业，作业列表实时显示作业的执行状态。
   - 单击**示例**，右侧作业编辑框显示DLA为您提供的SparkPi示例作业，单击**执行**，执行SparkPi示例。

9. （可选）在**作业列表**中，查看作业状态或对作业执行操作。



配置	说明
作业ID	Spark任务ID，由系统生成。

状态	Spark任务的运行状态。 ○ **STARTING**：任务正在提交。 ○ **RUNNING**：任务运行中。 ○ **SUCCESS**：Spark作业执行成功。 ○ **DEAD**：任务出错，可通过查看日志进行排错处理。 ○ **KILLED**：任务被主动终止。
作业名称	创建Spark作业时设置的作业名称，由name参数指定。
提交时间	当前Spark作业的提交时间。
启动时间	当前Spark作业的启动时间。
更新时间	当前Spark作业状态发生变化时的更新时间。
持续时间	运行当前Spark作业所花费的时间。
操作	操作中有5个参数，分别为： ○ **日志**，当前作业的日志，只获取最新的300行日志。 ○ **SparkUI**，当前作业的Spark Job UI 地址，如果Token过期需要单击**刷新**获取最新的地址。 ○ **详情**，当前作业提交时填写的JSON脚本。 ○ **kill**，终止当前的作业。 ○ **历史**，查看当前作业的作业尝试列表。 ○ **监控**，查看当前作业的监控数据。

10. （可选）单击**作业尝试列表**，查看所有作业的作业尝试。

> **说明**
> ○ 默认情况下，一个作业只会进行一次作业尝试。如需进行多次作业尝试，请配置作业重置参数。更多信息，请参见作业配置指南。
> ○ 在**作业尝试**列表中，选中单个作业，单击**操作 > 历史**，可以查看该作业的尝试列表。

附录

- 数据湖分析提供了开发Spark作业的Demo，您可以参考开源项目Aliyun DLA Demo。您可以直接进行下载，执行mvn进行打包。建议您参考本项目进行pom配置和开发。
- 使用DMS进行Spark作业编排和任务周期调度，请参考文档DMS任务编排调度Spark任务训练机器学习模型。
- DLA Spark作业配置，请参考文档作业配置指南。

4.2. 作业配置指南

Serverless Spark作业的描述格式为JSON格式，包含作业名称，JAR包路径以及作业配置参数等信息。本文主要介绍如何配置Serverless Spark任务格式。

Spark任务示例

本文以读取OSS数据为例，介绍Spark任务的编写方式，命令行参数格式为JSON格式。示例如下：

```
{
 "args": ["oss://${oss-buck-name}/data/test/test.csv"],
 "name": "spark-oss-test",
 "file": "oss://${oss-buck-name}/jars/test/spark-examples-0.0.1-SNAPSHOT.jar",
 "className": "com.aliyun.spark.oss.SparkReadOss",
 "conf": {
   "spark.driver.resourceSpec": "medium",
   "spark.executor.resourceSpec": "medium",
   "spark.executor.instances": 2,
   "spark.dla.connectors": "oss"
 }
}
```

上述示例描述了一个典型的离线Spark JAR任务的格式，包括任务的名字、主JAR、入口类、入口类参数以及Spark作业配置。如果开发者熟悉Spark社区用法的话，可以发现这些配置跟社区Spark-Submit工具的命令行参数类似。实际上，Serverless Spark参考了社区的用法，并跟社区的用法保持一致，包括参数名以及参数的语义。

作业参数说明

本章节对Serverless Spark作业参数进行说明。

参数名称	是否必填	示例值	使用说明
args	否	`"args": ["args0", "args1"]`	Spark任务传入的参数，多个参数之间以英文逗号（,）分隔。
name	否	`"name": "your_job_name"`	Spark任务名称。
file	Python/Java/Scala应用必填	`"file":"oss://bucket/path/to/your/jar"`	Spark任务主文件的存储位置，可以是入口类所在的JAR包或者Python的入口执行文件。 **说明** Spark任务主文件目前只支持存储在OSS中。

参数名称	是否必填	示例值	使用说明
className	Java/Scala应用必填	`"className":"com.aliyun.spark.oss.SparkReadOss"`	Java或者Scala程序入口类。如果是Python则不需要指定。
sqls	SQL应用必填	`"sqls":["select * from xxxx","show databases"]`	本关键字是区别于社区Spark的DLA平台自研功能，允许用户不提交JAR包和Python文件，直接提交SQL离线作业。该关键字跟**file, className, args**关键字不能同时使用。用户可以在一个作业中指定多条SQL语句，中间以英文逗号（,）隔开。多条SQL语句按照指定的顺序依次执行。
jars	否	`jars:["oss://bucket/path/to/jar","oss://bucket/path/to/jar"]`	Spark任务依赖的JAR包，多个JAR包之间以英文逗号（,）分隔。JAR包在作业运行时会被加入到Driver和Executor JVM的ClassPath里面。 **说明** Spark任务所依赖的所有JAR包须存储在OSS中。
files	否	`"files":["oss://bucket/path/to/files","oss://bucket/path/to/files"]`	Spark任务依赖的文件资源，文件会被下载到Driver和Executor进程的当前执行目录下。文件可以指定别名，比如 `oss://bucket/xx/yy.txt#yy` ，用户在代码中只需要使用*./yy*就可以访问文件，否则使用*./yy.txt*。多个文件中间用英文逗号（,）分隔。 **说明** • files中包含名为 `oss://<path/to>/log4j.properties` 的文件时（文件名固定为*log4j.properties*），Spark会使用该 `log4j.properties` 作为日志配置。 • Spark任务所依赖的所有文件须存储在OSS中。
archives	否	`"archives":["oss://bucket/path/to/archives","oss://bucket/path/to/archives"]`	Spark任务依赖的文件包资源，目前支持ZIP、TAR、TAR.GZ后缀。文件包会被解压到当前Spark进程的当前目录下。文件包可以指定别名，比如*oss://bucket/xx/yy.zip#yy*，用户在代码中只需要使用*./yy/zz.txt*就可以访问解压后的文件，否则使用*./yy.zip/zz.txt*访问文件（假设zz.txt是yy.zip压缩包中的文件）。多个文件包中间使用英文逗号（,）分隔。 **说明** Spark任务所依赖的所有文件包须存储在OSS中。文件包解压缩失败，任务会失败。

参数名称	是否必填	示例值	使用说明
pyFiles	Python应用可选	`"pyFiles":["oss://bucket/path/to/pyfiles","oss://bucket/path/to/pyfiles"]`	PySpark依赖的Python文件，后缀可以是ZIP、PY和EGG。如果依赖多个Python文件，建议用户使用ZIP或者EGG文件包。这些文件可以直接在Python代码中以module的方式引用。多个文件包中间使用英文逗号（,）分隔。 **说明** Spark任务所依赖的所有Python文件须存储在OSS中。
conf	否	`"conf":{"spark.xxxx":"xxx","spark.xxxx":"xxxx"}`	与开源Spark中的配置项相同，参数格式为 `key:value` 形式，多个参数之间以英文逗号（,）分隔。 若不填写conf，系统使用创建虚拟集群时设置的默认值。

DLA Spark 配置项

DLA Spark配置项跟社区Spark中的配置项基本保持一致，本章节将对其中不一致的地方以及DLA Serverless Spark平台提供的参数进行说明。

- **与社区Spark不一致的地方**

  指定Driver和Executor资源大小。

参数名称	使用说明	对应社区Spark参数
spark.driver.resourceSpec	表示spark driver的资源规格。取值： ○ **small**：表示1c4g ○ **medium**：表示2c8g ○ **large**：表示4c16g ○ **xlarge**：表示8c32g	**spark.driver.cores**以及**spark.driver.memory**
spark.executor.resourceSpec	表示spark executor的资源规格，同spark.driver.resourceSpec。	**spark.executor.cores**以及**spark.executor.memory**

- **DLA Spark相关参数**
  - Spark UI相关参数：

参数名称	默认值	参数说明
spark.dla.job.log.oss.uri	无	用于存储DLA Spark作业产生的作业日志以及SparkUI EventLog的目录，目前只支持用户的OSS路径。如果不填无法看到作业日志，作业结束后无法打开SparkUI。

- RAM账号运维Spark相关参数：

参数名称	默认值	参数说明
spark.dla.roleArn	无	在RAM系统中授予提交作业的子账号的roleArn，子账号提交作业需要填写该参数，主账号提交作业无需提交该参数。

- DLA Spark内置数据源连接器。

参数名称	默认值	参数说明
spark.dla.connectors	无	启用DLA Spark内置的连接器，连接器名称以逗号隔开，目前可选的连接器有oss、hbase1.x、tablestore。
spark.hadoop.job.oss.fileoutputcommitter.enable	false	开启parquet格式写入优化。请参考OSS。 注意 ■ 需要两个参数同时使用。 ■ 不支持与其它数据格式混用。 ■ 必须设置"spark.dla.connectors":"oss"。。
spark.sql.parquet.output.committer.class	com.aliyun.hadoop.mapreduce.lib.output.OSSFileOutputCommitter	
spark.hadoop.io.compression.codec.snappy.native	false	标识Snappy文件是否为标准Snappy文件。Hadoop默认识别的是Hadoop修改过的Snappy格式文件。设置为true时将使用标准snappy库解压，否则使用hadoop默认的snappy库解压。

- 访问用户VPC和连接用户数据源相关参数：

参数名称	默认值	参数说明
spark.dla.eni.enable	false	这个参数为true表示启用打通VPC功能。
spark.dla.eni.vswitch.id	无	开启打通VPC功能，用于弹性网卡的交换机ID。一般地，如果用户有ECS可以访问目标数据源，那么可以直接使用该ECS的交换机ID。
spark.dla.eni.security.group.id	无	开启打通VPC功能，用于弹性网卡的安全组ID。一般地，如果用户有ECS可以访问目标数据源，那么可以直接使用该ECS的安全组ID。
spark.dla.eni.extra.hosts	无	需要额外传入的IP和Host的映射关系，以便Spark能正确解析用户数据源中的域名信息。如连接用户的Hive数据源，就需要传入此参数。 注意 IP和域名之间用空格隔开。多个IP和域名用逗号隔开，如"ip0 master0, ip1 master1"。

- Spark SQL链接DLA元数据相关参数：

参数名称	默认值	参数说明
spark.sql.hive.metastore.version	1.2.1	用于指定Hive MetaStore的版本，DLA Spark扩充了社区版该参数的可取值，当设置为**dla**时，用户即可使用Spark SQL访问DLA元数据。

- PySpark相关参数：

参数名称	默认值	参数说明
spark.kubernetes.pyspark.pythonVersion	2	DLA Spark使用的Python版本，可取值为2和3。2代表使用Python2，3代表使用Python3。

- 作业重试相关参数：

参数名称	默认值	参数说明	示例

参数名称	默认值	参数说明	示例
spark.dla.job.maxAttempts	1	作业最大尝试次数，默认为1，代表作业不支持重试。 **说明** 取值范围为[1, 9999]。如果作业成功则不再继续尝试，如果作业失败，且该值大于1，会自动进行下一次尝试。	假设spark.dla.job.maxAttempts=3，则这个作业最多尝试3次。

参数名称	默认值	参数说明	示例
spark.dla.job.attemptFailuresValidityInterval	-1	作业尝试追踪的有效时间间隔，默认值为-1，代表未启用作业尝试追踪。 注意 ■ 如果作业尝试结束时间距离当前时间已经超过了指定的时间间隔，则该次尝试不会被计入失败的次数。 ■ 值如果设置的过小，很容易导致错误的作业被无限重试，因此默认情况下不建议设置该值。 ■ 值可以带单位，支持的单位有： ■ ms：毫秒，默认单位。 ■ m：分钟。 ■ h：小时。 ■ d：天。	假设spark.dla.job.attemptFailuresValidityInterval=30m，当前时间是12：40，已存在JobAttempt0的结束时间为12：00，JobAttempt1的结束时间为12：30，JobAttempt2的结束时间为12：35，则JobAttempt0不被计入尝试次数，这个作业的有效尝试只有JobAttempt1和JobAttempt2，总尝试计数为2。

参数名称	默认值	参数说明	示例

- 资源配置相关参数：

参数名称	默认值	参数说明
spark.dla.driver.cpu-vcores-ratio	1	Driver虚拟Core实际CPU Core之间的比例。 假设Driver是Medium规格（2C8G），本参数值设置为2，那么Driver进程可以按照4个Core进行并发控制，相当于spark.driver.cores=4。
spark.dla.executor.cpu-vcores-ratio	1	Executor虚拟Core实际CPU Core之间的比例。当单个Task的CPU使用率比较低时，可以通过该配置，提升CPU利用效率。 假设Executor是Medium规格（2C8G），本参数值设置为2，那么Executor进程可以按照4个Core进行并发控制，也就是同时调度4个并发任务，相当于spark.executor.cores=4。

- 监控相关参数：

参数名称	默认值	参数说明
spark.dla.monitor.enabled	true	作业级别监控开关，默认打开。 当设置为false后将不采集当前作业的监控数据 。

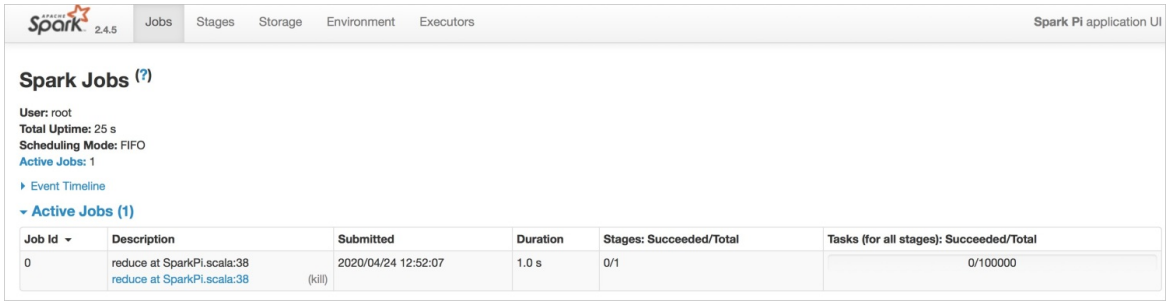
4.3. Spark UI

本文介绍如何在作业运行中和结束后查看Apache Spark web UI。

操作步骤

1. 登录Data Lake Analytics管理控制台。
2. 页面左上角，选择DLA所在地域。
3. 单击左侧导航栏中的**Serverless Spark > 作业管理**。
4. 单击目标Spark任务右侧的**操作 > SparkUI**。

5. 在浏览器中直接查看Spark UI，使用方式跟社区保持一致。



注意事项

1. 关于日志路径

为了作业结束后，也能访问Spark web UI，运行历史日志信息默认被上传到*oss://aliyun-oa-query-results-<user-parent-id>-oss-<region-id>/Spark_Tmp/*的OSS路径。如果您有自定义日志路径需求，可以通过下列配置项修改：

```
spark.dla.job.log.oss.uri":"oss://{bucket_name}/{dir_name}
```

> **说明**
>
> - `bucket_name`：所属的 bucket 必须已经创建，并与 DLA Spark 同一 Region。
> - `dir_name`：目录名称，支持多级目录。

示例如下：

```
{
    "name": "SparkPi",
    "file": "local:///tmp/spark-examples.jar",
    "className": "org.apache.spark.examples.SparkPi",
    "conf": {
        "spark.dla.job.log.oss.uri": "oss://your-bucket/your-logs-dir"
    }
}
```

2. 关于Spark UI有效期

由于Spark UI需要占用服务平台的网络资源，运行结束作业的Spark UI有效期只有三天。过期后Spark UI虽然不能打开，但是OSS上的日志数据是未被删除的，您可以根据需求通过OSS客户端查看或删除日志数据。

4.4. 配置数据源网络

本文主要介绍Serverless Spark如何配置数据源网络来访问用户VPC网络中的数据。这些数据包括RDS系列、ADB系列、PolarDB系列、MongoDB、Elasticsearch、HBase、E-MapReduce、Kafka以及用户在ECS上自建的各种数据服务等。

背景信息

Serverless Spark进程（Driver和Executor）运行在安全容器中，安全容器可以动态挂载用户VPC网络的虚拟网卡，就如同运行在用户VPC网络内的ECS上一样，可以直接访问用户的数据源网络。虚拟网卡的生命周期跟Spark进程的生命周期一致，作业结束后，所有网卡也会被释放。

Serverless Spark要挂载用户VPC网络的虚拟网卡，只需要在Spark作业配置中配置VPC网络中的安全组ID和虚拟交换机ID即可。如果用户的ECS本来就可以访问目标数据，只需要在Spark作业配置中配置该ECS所在的安全组ID和虚拟交换机ID即可。

> **说明** Spark的计算容器，Driver和每个Executor都会占用所配置的虚拟交换机下面的一个IP，提交作业前，请确保虚拟交换机网段的剩余IP数充足。

注意事项

访问服务型云产品中的数据，无需进行本文档中的配置，这些服务包括OSS、MaxCompute、TableStore、SLS等，这些服务一般都需要配置用户AK访问公共服务地址。

操作步骤

1. 准备虚拟交换机和安全组。

   DLA提供了三种方法来准备虚拟交换机和安全组。如果您的某个ECS通过VPC内网访问过目标数据源，那么推荐您直接选取该ECS的安全组和虚拟交换机，这个方法最简单，请参考下面的方法1；如果您的ECS不能访问目标数据源，您可以在数据源基础信息页面中获取相应信息，参考方法2；您也可以选择创建新的安全组和虚拟交换机，参考方法3。

   - **方法1：选取ECS的安全组和虚拟交换机**

     a. 登录ECS控制台，在ECS的实例列表中定位到该ECS实例。

     b. 在该ECS实例的**实例详情**页面查看安全组ID和虚拟交换机ID，如下图所示：

   - **方法2：使用目标数据源已有的交换机ID和安全组ID**

您可以在目标数据源的基础信息页面获取交换机ID和安全组ID，以E-MapReduce为例：

当目标数据源的基础信息页面中不存在安全组信息时，可以进入VPC控制台，进入目标数据源VPC中任意选择一个安全组即可。

- **方法3：在要访问的VPC网络内创建新的安全组和交换机**
  a. 在要访问的VPC网络内创建新的交换机和安全组，具体操作请参见创建交换机和创建安全组。
  b. 允许上个步骤创建的安全组出方向访问目标数据源。

     登录ECS控制台，打开**安全组规则**页面，配置出方向允许访问目标数据源。具体操作请参见添加安全组规则。

2. 为配置的交换机网段添加白名单。
   - 目标数据源如果是阿里云实例型产品，例如RDS、MongoDB等，您可以登录控制台去配置白名单。您可以在白名单中配置虚拟交换机IP地址段和安全组ID。以RDS为例，如下图所示：

- 目标数据源如果是用户基于ECS自建，则需要在目标数据源所在ECS的**安全组规则**页面，配置入方向允许新创建的安全组或者交换机网段访问目标数据源。具体操作请参见添加安全组规则。

> **说明** 如果您的安全组是企业安全组，默认不支持同一个安全组内的机器相互访问，需要将您选择的vSwitch所在的网段，加到此安全组的出、入方向。

3. 提交Spark作业。

   在Serverless Spark中编写Spark-Submit的脚本，具体操作请参见创建和执行Spark作业。

```
{
    "name": "SparkPi",
    "file": "local:///tmp/spark-examples.jar",
    "className": "org.apache.spark.examples.DriverSubmissio*****",
    "args": [
       "100000"
    ],
    "conf": {
        "spark.driver.resourceSpec": "small",
        "spark.executor.resourceSpec": "medium",
        "spark.executor.instances": 1,
        "spark.dla.eni.enable": "true",
        "spark.dla.eni.vswitch.id": "vsw-bp17jqw3lrrobn6y*****",
        "spark.dla.eni.security.group.id": "sg-bp163uxgt4zandx*****",
    }
}
```

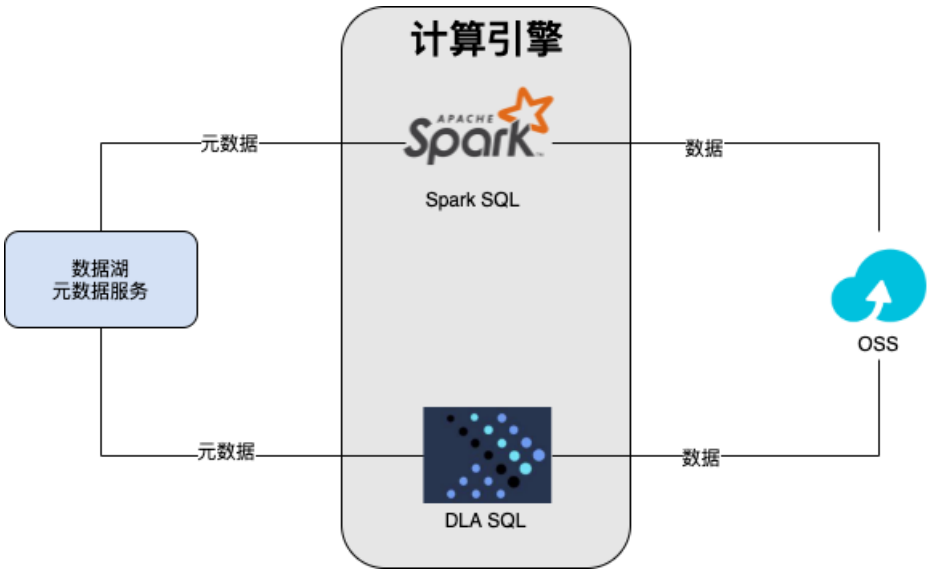
> **说明**
>
> - `spark.dla.eni.enable` 取值为 `true` 时，表示启用访问用户VPC功能，挂载用户数据源网络虚拟网卡。
> - `spark.dla.eni.vswitch.id` 和 `spark.dla.eni.security.group.id` 配置为步骤一中获取的虚拟交换机ID和安全组ID。

4.5. Spark SQL

Spark与DLA SQL引擎以及数据湖构建服务共享元数据。

Spark访问数据湖元数据服务

Spark引擎可以支持多种元数据服务，既支持访问用户自建的Hive，也支持访问DLA统一管理的数据湖元数据。DLA统一管理的数据湖元数据管理服务，同时支持多种引擎访问，实现多种引擎的元数据信息共享。在数据湖元信息发现、T+1全量同步一键建仓中创建的库表结构，可以被Spark读取并使用，Spark SQL创建或者修改的元数据也可以被其他引擎访问到。下图是Spark SQL和DLA SQL与元数据服务之间的关系。



登录DLA控制台，单击左侧导航栏的**SQL执行**，您可以看到所有数据湖中的数据库和表结构，对表进行在线分析查询，以及管理子用户对库表的权限，如下图所示。

纯SQL作业

DLA Spark支持直接在控制台写Spark SQL。无需用户打包jar包或者写python代码，更有利于数据开发人员使用Spark进行数据分析。

您需要先登录DLA控制台，在**Serverless Spark > 作业管理**菜单中创建SparkSQL类型的作业。创建SparkSQL类型的作业后，系统默认会使用DLA元数据服务。如果您想关闭DLA元数据服务，可以使用以下两种方式：

- 使用 `in-memory catalog` ，将不会使用DLA元数据服务。

```
set spark.sql.catalogImplementation = in-memory;
```

- 设置 `hive metastore version` 为1.2.1或其他版本。

```
set spark.sql.catalogImplementation = hive;
set spark.sql.hive.metastore.version = 1.2.1;
```

SparkSQL作业的文本框中，支持直接写SQL语句，**每条SQL语句以分号隔开**。

SQL语句支持下列类型命令：

- SET命令
  - 用于指定Spark的设置，一般置于整个SQL语句的最前面。
  - 每条Set命令指定一个Spark参数的值，每条SET命令用分号隔开。
  - SET命令的Key和Value均**不要加单引号或者双引号**。
- ADD JAR命令
  - 用于增加Spark SQL运行时，依赖的jar包，比如UDF的jar包，各类数据源连接器的Jar包等。Jar包目前支持OSS格式路径，一般置于整个SQL语句的最前面。
  - 每条add jar命令指定一个oss jar包路径，路径字符串不要加单引号和双引号， 每条ADD JAR命令用分号隔开。
- Spark SQL语法所支持的DDL或DML语句
  - 例如查询语句 `select` 。
  - 例如插入语句 `insert` 。
  - 例如查看数据库 `SHOW DATABASE` 。

> **说明**
>
> - SparkSQL语句的使用限制，请参考后续章节使用限制和注意事项。
> - 不在SQL语句最前面的 `SET` 命令和 `ADD JAR` 命令，将会在SQL语句运行时生效。例如，两个SELECT语句中间的 `SET` 命令将会在上一条 `SELECT` 语句执行完后， `SET` 命令才会生效。

代码中使用Spark SQL

您也可以在程序中执行SQL，操作元数据信息，以及读写表内容。下面以PySpark为例进行介绍，其他语言使用方式类似。首先，建立以下Python文件，保存为example.py，将文件上传至OSS。

```
from pyspark.sql import SparkSession

if __name__ == "__main__":
    # init pyspark context
    spark = SparkSession \
        .builder \
        .appName("Python SQL Test") \
        .getOrCreate()

    # create a database
    spark.sql(
            "create database if not exists dlatest comment 'c' location 'oss://{your bucket
name}/{path}/' WITH DBPROPERTIES(k1='v1', k2='v2')")
    # create table
    spark.sql(
            "create table dlatest.tp(col1 INT)  PARTITIONED BY (p1 STRING, p2 STRING) locat
ion 'oss://{your bucket name}/{path}/' STORED AS parquet TBLPROPERTIES ('parquet.compress'=
'SNAPPY')")
    # show structure
    print(spark.sql("show create table dlatest.tp").collect()[0])

    # insert data
    spark.sql("INSERT into dlatest.tp partition(p1='a',p2='a') values(1)")

    # show data
    spark.sql("select * from dlatest.tp").show()
```

通过以下的JSON将作业通过DLA控制台提交Spark作业。

```
{
    "name": "DLA SQL Test",
    "file": "oss://path/to/example.py",
    "conf": {
        "spark.driver.resourceSpec": "small",
        "spark.sql.hive.metastore.version": "dla",
        "spark.sql.catalogImplementation": "hive",
        "spark.dla.connectors": "oss",
        "spark.executor.instances": 1,
        "spark.dla.job.log.oss.uri": "oss://path/to/spark-logs",
        "spark.executor.resourceSpec": "small"
    }
}
```

执行成功后，可以在DLA控制台的**SQL执行**页面中找到名为 `dlatest` 的数据库，以及其下的 `tp` 表。

> **注意**
>
> DLA元数据服务对命名大小写不敏感，在引用库名和表名时忽略大小写。

使用DLA元数据服务的限制和注意事项

1. 当前在Spark中仅支持External类型数据库和表的创建和读写操作。

当前Spark连接数据湖元数据服务，只支持**外表（External Table）**的读写和创建操作。
这意味着在建立数据库时，需要显式指定 `LOCATION` 信息，类似如下的SQL语句。

```
CREATE DATABASE db1 LOCATION 'oss://test/db1/';
```

同样的,建表语句必须显式指定表的存储 `LOCATION` 信息,类似如下SQL语句。

```
CREATE TABLE table1(col1 INT) LOCATION 'oss://test/db1/table1/';
```

需要注意以下几个事项：

- 当用户在Spark中DROP一个表或者表的某个 `PARTITION` 时，并不会删除OSS上的文件。
- 当用户创建一个表时，指定的表的 `LOCATION` 必须是库的 `LOCATION` 的子文件夹。
- 当用户为添加表的 `PARTITION` 时，指定的 `PARTITION LOCATION` 必须是表的 `LOCATION` 的子文件夹。
- 当用户 `RENAME PARTITION` ` 时，并不会改变其在OSS上的路径结构。

2. 当前Spark中仅支持以OSS为存储的外表。

在当前阶段，数据湖分析 `SQL执行` 支持多种不同的存储，包括RDS、表格存储等等。当前在Spark中使用元数据服务支持读写以OSS为存储的外表。
使用Spark直接创建数据库和数据表， `LOCATION` 必须指定一个合法的OSS地址。
对于其他存储的支持，后续会陆续更新。

3. 当前禁止创建 `DEFAULT` 为名字的数据库。

由于不允许使用 `DEFAULT` 为名字的数据库，需要注意以下两个事项：

- 禁止在Spark中创建和操作名为 `DEFAULT` 的数据库。
- 在Spark中执行SQL时,操作表之前需要使用 `USE DatabaseName` SQL语句来切换到目标数据。或者显式指定某个表属于哪个数据库,例如 `SELECT * FROM db1.table1` 。

4. 当前在Spark中执行 `ALTER` 语句有部分限制。

用户可以通过类似 `ALTER DATABASE ...` 等语句修改库和表的元数据信息，当前在Spark中使用这一类语句有如下的限制。

- 当前对数据库，仅支持修改 `COMMENT` 信息，其它如 `LOCATION` 、 `PROPERTIES` 禁止修改。
- 当前仅支持修改表的 `COLUMN` 和 `PROPERTIES` 信息，如添加列、修改注解等等。请注意这里的 `COLUMN` 必须是非 `PARTITION` 列。

5. 当前在 `SQL执行` 中操作Spark表的一些限制。

当用户在Spark程序中创建了数据库db1和表table1后，如果在控制台的SQL执行中操作它们，需要显式检测其是否存在。如尝试删除此数据库时必须使用如下语句。

```
DROP DATABASE IF EXISTS db1;
```

6. 当前在Spark SQL中不支持 `GRANT` 类赋权语句。
同开源社区一致，当前用户无法通过Spark引擎执行一个GRANT语句来修改子账户赋权。

4.6. Spark UDF

本文档主要介绍了如何在Spark中管理并使用用户自定义函数UDF（User Define Function）。

使用元数据服务管理用户自定义函数UDF

- **注册UDF**

  Spark元数据支持UDF使用**Hive 1.2.1**标准来进行开发，注册UDF的示例如下：

  ```
  CREATE FUNCTION function_name AS class_name USING resource_location_list;
  ```

参数名称	参数说明
function_name	注册方法名，在注册前需要通过 `USE DatabaseName` 来指定此UDF的作用范围，或者以显式方式指定它的应用范围。
class_name	完整的class_name需要携带package信息，它的开发规范可以参考Spark和Hive的 `FUNCTION` 开发规范。
resource_location_list	这个方法使用到的JAR包或者文件放置的位置，需要显式指定依赖的是JAR还是FILE：`USING JAR 'oss://test/function.jar',FILE 'oss://test/model.csv'`。

- **查询当前数据库的所有UDF**

  ```
  USE databasename;
  SHOW USER FUNCTIONS;
  ```

  > **说明**
  >
  > 如果不加 `USER` 关键词，查询到的是Spark默认的Function，默认Function不允许被删除。

- **删除UDF**

  ```
  USE databasename;
  DROP FUNCTION functionname;
  ```

> **说明**
>
> 数据湖元数据管理不支持针对UDF的Alter语法，如果需要修改元数据的一些配置，请DROP对应的UDF后重新创建。

使用UDF

1. 实现UDF。

   初始化一个Maven管理工程，并在依赖中加入如下代码：

   ```
   <dependency>
     <groupId>org.apache.hive</groupId>
     <artifactId>hive-exec</artifactId>
     <version>1.2.1</version>
   </dependency>
   ```

   在Package的 `org.test.udf` 中实现一个 `Ten.java` ，它会为数据加10然后进行返回。

   ```
   package org.test.udf;
   import org.apache.hadoop.hive.ql.exec.UDF;
   public class Ten extends UDF {
     public long evaluate(long value) {
       return value + 10;
     }
   }
   ```

2. 注册UDF到Spark并使用。

   编译Maven工程为 `udf.jar` 并上传到OSS中，之后您可以注册这个方法并通过Serverless Spark来执行SQL访问它。

   ```
   -- here is the spark conf
   set spark.driver.resourceSpec=medium;
   set spark.executor.instances=5;
   set spark.executor.resourceSpec=medium;
   set spark.app.name=sparksqltest;
   set spark.sql.hive.metastore.version=dla;
   set spark.dla.connectors=oss;
   -- here is your sql statement
   use db;
   CREATE FUNCTION addten as 'com.aliyun.dla.udf.Ten' USING JAR 'oss://path/to/your/udf.ja
   r';
   select addten(7);
   ```

3. 检查结果

   在注册UDF后，您无需反复进行注册，可以直接调用这个方法。

   由于 `addten` 这个UDF已经被注册到了数据库 `db` 中，执行完上述的Spark作业后，可以在日志中看到输出 `17` 。

4.7. PySpark

本文展示如何提交PySpark作业以及使用自定义Virtualenv。

PySpark基本使用方式

1.开发主程序文件

您可以建立如下内容的 `example.py` 文件，示例中定义main函数可以允许PySpark找到程序的统一启动入口。

```
from __future__ import print_function
from pyspark.sql import SparkSession


# import third part file
from tools import func

if __name__ == "__main__":
    # init pyspark context
    spark = SparkSession\
        .builder\
        .appName("Python Example")\
        .getOrCreate()

    df = spark.sql("SELECT 2021")
    # print schema and data to the console
    df.printSchema()
    df.show()
```

2.执行主程序文件

1. 和Scala、Java程序开发的JAR包一样，您需要将 `example.py` 文件上传到OSS中，并在Spark的启动配置中使用 `file` 来指定这个文件为启动文件。
2. 在DLA控制台的**Serverless->作业管理**页面，使用如下示例代码配置作业。

```
{
    "name": "Spark Python",
    "file": "oss://{your bucket name}/example.py"
    "conf": {
        "spark.driver.resourceSpec": "small",
        "spark.executor.instances": 2,
        "spark.executor.resourceSpec": "small",
        "spark.kubernetes.pyspark.pythonVersion": "3"
  }
}
```

注意

- 您需要在配置时将 `{your bucket name}` 替换为您使用的OSS的Bucket名称。
- 示例中使用Python 3执行文件，和社区版的Spark相同，通过 `spark.kubernetes.pyspark.pythonVersion` 配置使用的Python版本，默认为Python 2.7。

3. 单击**执行**。

如何上传自行开发的或者第三方开发的Module

当开发Python程序时，往往会用到自行开发的或者由第三方开发的各种Module模块，这些模块可以上传并加载到PySpark的执行环境中，被主程序调用。

以计算员工的税后收入为例，步骤如下。

1. 准备测试数据

新建一个如下格式的CSV文件，命名为 `staff.csv` ，并上传到OSS中。文件反映了每个员工的信息和收入情况。

```
name,age,gender,salary
Lucky,25,male,100
Lucy,23,female,150
Martin,30,male,180
Rose,31,female,200
```

说明

如何将文件上传到OSS请参见简单上传。

2. 开发一个依赖方法

1. 创建一个文件夹 `tools` 。
2. 在 `tools` 文件夹中创建一个文件 `func.py` ，文件内容如下。

```
def tax(salary):
    """
    convert string to int
    then cut 15% tax from the salary
    return a float number

    :param salary: The salary of staff worker
    :return:
    """
    return 0.15 * int(salary)
```

3. 将 `tools` 文件夹压缩为 `tools.zip` 后上传到OSS中。压缩包的生成方式如下。

```
mgong@MGONG pyspark_demo % ls
tools
mgong@MGONG pyspark_demo %
```

> **注意**
>
> 不同操作系统平台的ZIP压缩工具会略有区别，请保证解压后可以看到顶层目录是tools文件夹。

3. 开发主程序

开发一个Spark的Python程序，将测试中的CSV从OSS中读取出来，注册为一个 `DataFrame` 。同时将依赖包中的 `tax` 方法注册为一个 `Spark UDF` ，然后使用该 `UDF` 对刚刚生成的 `DataFrame` 进行计算并打印结果。

示例代码如下， 您需要在配置时将 `{your bucket name}` 替换为您使用的OSS的Bucket名称。

```
from __future__ import print_function
from pyspark.sql import SparkSession
from pyspark.sql.functions import udf
from pyspark.sql.types import FloatType

# import third part file
from tools import func

if __name__ == "__main__":
    # init pyspark context
    spark = SparkSession\
        .builder\
        .appName("Python Example")\
        .getOrCreate()

    # read csv from oss to a dataframe, show the table
    df = spark.read.csv('oss://{your bucket}/staff.csv', mode="DROPMALFORMED",inferSchema=T
rue, header = True)
    # print schema and data to the console
    df.printSchema()
    df.show()

    # create an udf
    taxCut = udf(lambda salary: func.tax(salary), FloatType())

    # cut tax from salary and show result
    df.select("name", taxCut("salary").alias("final salary")).show()
    spark.stop()
```

将代码写入到 `example.py` 中，并上传到OSS。

4. 提交任务

在DLA控制台的**Serverless->作业管理**页面，新建一个作业，并提交以下作业信息。

```
{
    "name": "Spark Python",
    "file": "oss://{your bucket name}/example.py",
    "pyFiles": ["oss://{your bucket name}/tools.zip"],
    "conf": {
        "spark.driver.resourceSpec": "small",
        "spark.executor.instances": 2,
        "spark.executor.resourceSpec": "small",
        "spark.dla.connectors": "oss",
        "spark.kubernetes.pyspark.pythonVersion": "3"
    }
}
```

代码中主要参数说明。

参数	说明	是否必选
conf	Spark任务用到的配置参数，需要的配置项如下。 • `"spark.dla.connectors": "oss"` ：此任务需要有连接OSS的能力。 • `"spark.kubernetes.pyspark.pythonVersion": "3"` ：此任务需要使用Python 3来执行。	否

> **说明**
>
> 更多参数说明请参见作业配置指南。

PySpark使用自定义Virtualenv

当需要复杂的第三方依赖包时，可以使用Virtualenv来将本地调试环境上传到云端的Spark集群中。这种方式可以将大量复杂的系统包，如Pandas、Numpy、PyMySQL等装入隔离环境，并迁移到相同的操作系统中。您可以选择如下两种方案。

- 自行生成Virtualenv压缩包。
- 使用镜像工具生成Virtualenv压缩包。

> **说明**
>
> Virtualenv的更多信息请参见Python官方社区venv说明。

自行生成Virtualenv压缩包

1.准备Linux环境

由于 `Virtualenv` 需要相同的操作系统，当前上传到DLA Spark使用的压缩包必须在Linux环境下的进行安装。您可以采用如下方式准备Linux环境。

- 准备一台Centos7的电脑进行打包。
- 在阿里云以按量付费的方式新开一台Centos 7的ECS，使用完毕后关闭。
- 使用Centos 7的官方Docker镜像，在镜像内部打包。

2.在Linux环境下打包Python执行环境

常用的执行环境打包工具包括Virtualenv、Conda，您可以根据您的需要来选择对应的工具，并安装工具到您的Linux环境中。

> **注意**
>
> - 当前Serverless Spark支持的Python版本为3.7及以下主版本。
> - Spark运行环境为Centos 7，请使用该环境打包Venv（推荐使用Docker中的环境打包）。

以下示例使用 `Virtualenv` 生成一个执行环境压缩包 `venv.zip` ，压缩包中包含了 `scikit-spark` 的特定版本。

```
# create directory venv at current path with python3
# MUST ADD --copies !
virtualenv --copies --download --python Python3.7 venv

# active environment
source venv/bin/activate

# install third part modules
pip install scikit-spark==0.4.0

# check the result
pip list

# zip the environment
zip -r venv.zip venv
```

> **说明**
>
> 如何使用Conda生成执行环境，请参见Conda管理虚拟环境。

3.在Spark中使用Python执行环境

您可以在提交Spark作业时，使用如下的代码配置作业。其中 `spark.pyspark.python` 的参数值表示上传的压缩文件中的运行包。更多参数说明，请参见作业参数说明。

```
{
    "name": "venv example",
    "archives": [
        "oss://test/venv.zip#PY3"
    ],
    "conf": {
        "spark.driver.resourceSpec": "medium",
        "spark.dla.connectors": "oss",
        "spark.executor.instances": 1,
        "spark.dla.job.log.oss.uri": "oss://test/spark-logs",
        "spark.pyspark.python": "./PY3/venv/bin/python3",
        "spark.executor.resourceSpec": "medium"
    },
    "file": "oss://test/example.py"
}
```

> **说明**
>
> 与Spark开源社区的语义相同，`venv.zip#PY3`代表将压缩包解压到计算节点工作目录的`PY3`文件夹下，继而可以从本地访问。如果不使用`#`指定文件夹名称，则默认使用文件名称作为新建的文件夹名。

使用镜像工具生成Virtualenv压缩包

1.使用如下命令拉取镜像。

```
docker pull registry.cn-hangzhou.aliyuncs.com/dla_spark/dla-venv:0.1
```

2.将需要生成环境的`requirements.txt`文件放置在`/home/admin`文件夹中，并将此文件夹挂载到Docker中。

```
docker run -ti -v /home/admin:/tmp dla-venv:0.1 -p python3 -f /tmp/requirements.txt
```

> **说明**
>
> `requirements.txt`是Python的标准依赖包描述文件，更多信息请参见User Guide。

打包程序自动化执行，您可以看到如下日志。

```
adding: venv-20210611-095454/lib64/ (stored 0%)
  adding: venv-20210611-095454/lib64/python3.6/ (stored 0%)
  adding: venv-20210611-095454/lib64/python3.6/site-packages/ (stored 0%)
  adding: venv-20210611-095454/pyvenv.cfg (deflated 30%)
  venv-20210611-095454.zip
```

3.在`/home/admin`文件夹下找到打包好的压缩文件`venv-20210611-095454.zip`。如何使用压缩包请参见在Spark中使用Python执行环境。

4.（可选）关于Docker镜像的更多使用说明，您可以执行如下命令查看。

```
docker run -i dla-venv:0.1

Used to create venv package for Aliyun DLA
Docker with host machine volumes: https://docs.docker.com/storage/volumes/
Please copy requirements.txt to a folder and mount the folder to docker as /tmp path
Usage example: docker run -it -v /home/admin:/tmp dla-venv:0.1 -p python3 -f /tmp/requireme
nts.txt
 -p python version, could be python2 or python3
 -f path to requirements.txt, default is /tmp/requirements.txt
```

常见问题

如果在使用镜像工具生成Virtualenv压缩包时自动打包失败，您可以通过执行如下命令启动Linux Centos 7环境，并以Root权限进入环境内部进行操作。

```
docker run -ti --entrypoint bash dla-venv:0.1
```

后续操作请参见自行生成Virtualenv压缩包。

4.8. Spark MLlib

本文介绍如何在DLA Serverless Spark中运行Spark MLlib任务。

场景

本示例将在DLA Serverless Spark中通过K-Means聚类算法，将以下数据分成两个族类，然后判断测试数据是否在族类中。

```
0.0 0.0 0.0
0.1 0.1 0.1
0.2 0.2 0.2
9.0 9.0 9.0
9.1 9.1 9.1
9.2 9.2 9.2
```

前提条件

在DLA Serverless Spark中运行Spark MLlib任务前，您需要完成以下准备工作。

- 在OSS中上传测试数据，测试数据存储在rawdata.csv文件中。

操作步骤

1. 登录Data Lake Analytics管理控制台。
2. 在页面左上角，选择DLA所在地域。
3. 单击左侧导航栏中的**Serverless Spark > 作业管理**。
4. 在**作业编辑**页面，单击**创建作业**。
5. 在**创建作业**页面，按照页面提示进行参数配置。
6. 完成上述参数配置后，单击**确定**创建Spark作业。
7. 单击Spark作业名，在Spark作业编辑框中输入Spark MLlib任务内容。

```
        {
                "name": "spark-mllib-test",
                "file": "oss://${your oss bucket}/jars/test/spark-examples-0.0.1-SNAPSH
OT.jar",
                "className": "com.aliyun.spark.SparkMLlib",
                "args": ["oss://${your oss bucket}/data/rawdata.csv"],
                "conf": {
                        "spark.driver.resourceSpec": "medium",
                        "spark.executor.instances": 2,
                        "spark.executor.resourceSpec": "medium",
                         "spark.dla.connectors": "oss"
                }
        }
```

注意：如果是子账号提交，还需要配置spark.dla.roleArn参数，参考文档使用RAM子账号开发Spark作业。

示例代码

以下为主类SparkMLlib对应的源代码。

```
package com.aliyun.spark
import org.apache.spark.SparkConf
import org.apache.spark.mllib.clustering.KMeans
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.sql.SparkSession
object SparkMLlib {
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("Spark MLlib")
    val spark = SparkSession
      .builder()
      .config(conf)
      .getOrCreate()
    val rawDataPath = args(0)
    val data = spark.sparkContext.textFile(rawDataPath)
    val parsedData = data.map(s => Vectors.dense(s.split(' ').map(_.toDouble)))
    val numClusters = 2
    val numIterations = 20
    val model = KMeans.train(parsedData, numClusters, numIterations)
    for (c <- model.clusterCenters) {
      println(s"cluster center: ${c.toString}")
    }
    val cost = model.computeCost(parsedData)
    //预测数据
    println("Vectors 0.2 0.2 0.2 is belongs to clusters:" +
      model.predict(Vectors.dense("0.2 0.2 0.2".split(' ').map(_.toDouble))))
    println("Vectors 0.25 0.25 0.25 is belongs to clusters:" +
      model.predict(Vectors.dense("0.25 0.25 0.25".split(' ').map(_.toDouble))))
    println("Vectors 8 8 8 is belongs to clusters:" +
      model.predict(Vectors.dense("8 8 8".split(' ').map(_.toDouble))))
  }
}
```

上述代码的运行结果如下所示，前两个值属于族0，后面一个值属于族1。

```
Vectors 0.2 0.2 0.2 is belongs to clusters:0
Vectors 0.25 0.25 0.25 is belongs to clusters:0
Vectors 8 8 8 is belongs to clusters:1
```

4.9. Spark Streaming

本文介绍DLA Serverless Spark如何提交Spark Streaming作业以及Spark Streaming作业重试的最佳实践。

前提条件

在DLA Serverless Spark中运行Spark Streaming作业前，您需要完成以下准备工作：

- 授权DLA Serverless Spark访问用户VPC网络的权限。具体操请参见配置数据源网络。
- 在Data Lake Analytics管理控制台的**虚拟集群管理**页面中，确认您的虚拟集群使用的版本是`spark_2_4_5-dla_1_2_0`及以上。

创建Spark Streaming作业

以DLA Serverless Spark访问用户VPC网络中的阿里云消息队列Kafka为例。

1. 登录Data Lake Analytics管理控制台。
2. 在页面左上角，选择Kafka服务所在地域。
3. 单击左侧导航栏中的**Serverless Spark > 作业管理**。
4. 在**作业编辑**页面，单击**创建作业模板**。
5. 在**创建作业模板**页面，按照页面提示进行参数配置后，单击**确定**创建Spark作业。

6. 单击Spark作业名，在Spark作业编辑框中输入Spark Streaming作业内容。

```
{
    "file": "oss://path/to/xxx.jar",
    "name": "Kafka",
    "className": "com.alibabacloud.cwchan.SparkKafkaSub",
    "conf": {
        "spark.driver.resourceSpec": "medium",
        "spark.executor.instances": 5,
        "spark.executor.resourceSpec": "medium",
        "spark.dla.job.log.oss.uri": "oss://path/to/spark-logs",
        "spark.dla.eni.vswitch.id": "{vswitch-id}",
        "spark.dla.eni.security.group.id": "{security-group-id}",
        "spark.dla.eni.enable": "true"
    }
}
```

编译打包过程中，需要打包 `Spark-Kafka` 的相关依赖，如下所示：

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql-kafka-0-10_2.11</artifactId>
    <version>2.4.5</version>
</dependency>
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming-kafka-0-10_2.11</artifactId>
    <version>2.4.5</version>
</dependency>
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>0.10.2.2</version>
</dependency>
```

注意 如果是使用RAM用户提交作业，需要配置RAM用户权限，详情请参见快速配置子账号权限。

示例代码

以下是连接Kafka的核心代码片段：

```
val sparkConf: SparkConf = new SparkConf()
  .setAppName("SparkKafkaSub")
val sparkSessoin = SparkSession
  .builder()
  .config(sparkConf)
  .getOrCreate()
val df = sparkSessoin
  .readStream
  .format("kafka")
  .option("kafka.bootstrap.servers", bootstrapServers)
  .option("subscribe", topicName)
  .option("group.id", groupId)
  .load()
val query = df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")
  .writeStream
  .outputMode("append")
  .format("console")
  .start()
query.awaitTermination()
```

说明 该代码消费Kafka的消息，并打印Key-Value对。

Spark Streaming作业重试的最佳实践

对于流应用，如果想要配置作业失败之后进行自动重试，您可以在conf中配置如下参数。

```
#代表作业的尝试次数，默认为1代表不重试，5代表作业失败将会尝试五次
spark.dla.job.maxAttempts  5
#代表作业尝试的计数有效时间，1h代表有效时间为一个小时，超时将不会计入作业尝试总数，该值默认为-1，代表作业失败计数永不超时
spark.dla.job.attemptFailuresValidityInterval 1h
#以上两个参数组合起来的含义是：在任意1个1小时的时间区间内，作业尝试次数超过五次后，作业将停止尝试，否则作业会继续进行尝试
```

说明 关于作业重试功能的具体配置说明，请参见作业重试相关参数。

对于流应用而言，在重新提交作业前，通常希望能够从上一次中断消费的位置继续消费。下面介绍如何才能使得作业在重试的时候从上一次停止的消费位点继续消费。

- Spark Structured Streaming（推荐）

  对于Structured Streaming，您只需要在启动query的时候，指定checkpoint location即可，location指定为OSS路径。示例代码如下：

```
    val query = df
      .selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")
      .as[(String, String)]
      .writeStream
      .format("csv")
      .option("path", outputPath)
      .outputMode("Append")
      .option("checkpointLocation", "oss://path/to/checkpoint/dir")
      .trigger(Trigger.ProcessingTime(10, TimeUnit.SECONDS))
      .start()
```

> **说明** 关于Spark Structured Streaming的checkpoint的更多内容，请参见Structured Streaming Programming Guide。

- Spark Streaming（DStreams）

  对于DStreams，您需要按照一定的编程模式，才可以正确的从checkpoint处恢复程序，示例代码如下：

```
// Function to create and setup a new StreamingContext
def functionToCreateContext(): StreamingContext = {
  val ssc = new StreamingContext(...)   // new context
  val lines = ssc.socketTextStream(...) // create DStreams
  ...
  ssc.checkpoint(checkpointDirectory)   // set checkpoint directory
  ssc
}
// Get StreamingContext from checkpoint data or create a new one
val context = StreamingContext.getOrCreate(checkpointDirectory, functionToCreateContext _
)
// Do additional setup on context that needs to be done,
// irrespective of whether it is being started or restarted
context. ...
// Start the context
context.start()
context.awaitTermination()
```

> **说明** 关于DStreams的checkpoint的更多内容，请参见Spark Streaming Programming Guide。

Spark Streaming作业的监控与报警

对于流作业而言，DLA Spark默认为作业开启了监控与报警功能。

- 您可以通过监控查看流作业的运行状态，例如作业处理延迟，数据处理速率等，具体请参见查看Spark监控。
- 您可以通过配置报警规则实现对流作业的实时报警通知，具体请参见管理报警。

5.生态工具

5.1. Airflow调度DLA Spark作业

Airflow是比较流行的开源调度工具，可以实现各类工作负载的DAG编排与调度。您可以通过Spark-Submit和Spark-SQL命令行来实现Airflow调度Spark任务。DLA Spark提供了命令行工具包，支持通过Spark-Submit和Spark-SQL方式来提交Spark作业。您可以直接将开源Spark命令行工具包替换成DLA Spark命令行工具包，并进行简单的配置即可使用Airflow调度DLA Spark作业。

准备工作

- 安装Airflow服务。
  i. 安装Airflow服务并启动。具体操作请参见Airflow社区文档。
  ii. 安装Airflow Spark插件。执行命令如下：

```
pip3 install apache-airflow-providers-apache-spark
```

> **说明**
> - 您需要使用Python3来安装Airflow Spark插件。
> - 安装*apache-airflow-providers-apache-spark*会默认安装社区版Pyspark，需要将其卸载，执行命令如下：
>
> ```
> pip3 uninstall pyspark
> ```

- 下载DLA Spark命令行工具包并进行配置。
  i. 下载DLA Spark命令行工具包并进行配置。具体操作请参见Spark-Submit命令行工具。
  ii. 配置PATH路径，执行命令如下：

```
export PATH=$PATH:/your/dla/spark/path/bin
```

> **说明** 在启动Airflow scheduler之前需要将Spark-Submit和Spark-SQL命令加入到PATH中，否则调度任务可能会找不到Spark-Submit和Spark-SQL命令。

操作步骤

1. 编辑DLA Spark Airflow DAG的*dla_spark_demo.py*文件。如下所示：

```
from airflow.models import DAG
from airflow.providers.apache.spark.operators.spark_jdbc import SparkJDBCOperator
from airflow.providers.apache.spark.operators.spark_sql import SparkSqlOperator
from airflow.providers.apache.spark.operators.spark_submit import SparkSubmitOperator
from airflow.utils.dates import days_ago
args = {
    'owner': 'Aliyun DLA',
}
with DAG(
    dag_id='example_dla_spark_operator',
    default_args=args,
    schedule_interval=None,
    start_date=days_ago(2),
    tags=['example'],
) as dag:
    dla_spark_conf = {
        "spark.driver.resourceSpec": "medium",
        "spark.executor.resourceSpec": "medium",
        "spark.sql.hive.metastore.version": "dla",
        "spark.dla.connectors": "oss",
        "spark.hadoop.job.oss.fileoutputcommitter.enable": "true"
    }
    # [START howto_operator_spark_submit]
    submit_job = SparkSubmitOperator(
        conf = dla_spark_conf,
        application="oss://your-bucket/jar/pi.py",
        task_id="submit_job",
        verbose=True
    )
    # [END howto_operator_spark_submit]
    # [START howto_operator_spark_sql]
    sql_job = SparkSqlOperator(
        conn_id="spark_default",
        sql="SELECT * FROM yourdb.yourtable",
        conf=",".join([k+"="+v for k,v in dla_spark_conf.items()]),
        task_id="sql_job",
        verbose=True
    )
    # [END howto_operator_spark_sql]
    submit_job >> sql_job
```

2. 执行DLA Spark DAG。

   将编辑完成的*dla_spark_demo.py*文件放到Airflow安装目录的dags目录下，然后执行DLA Spark DAG。具体操作请参见Airflow社区文档。

注意事项

- DLA Spark的最小资源调度单元是容器，容器规格通过 `resourceSpec` 来定义。您可以通过配置 `spark.driver.resourceSpec` 和 `spark.executor.resourceSpec` 来指定driver和executor的容器规格。Hadoop社区通过指定driver和executor的CPU和Memory来申请资源。DLA Spark工具包兼容了Hadoop的资源配置能力，如果您指定了driver和executor的CPU和Memory，会被自动转换为大于所指定CPU和Memory的最小资源规格。例如，当 `executor_cores` =2、 `executor_memory` =5 G时，则会被转换为 `spark.executor.resourceSpec` =medium。

- 对于DLA特有的一些参数，例如 `vcName` 、 `regionId` 、 `keyId` 、 `secretId` 、 `ossUploadPath` ，您可以在DLA Spark工具包的配置文件*conf/spark-defaults.conf*中进行配置，也可以通过Airflow参数来配置。
- 由于DLA Spark访问DLA的元数据时，只支持外表，因此对于 `SparkJDBCOperator` ， `cmd_type='jdbc_to_spark'` 并且 `save_mode="overwrite"` 的方式不支持。

  > **说明** DLA Spark访问自建Hive集群的元数据时，不存在该问题。关于如何访问自建Hive元数据，请参见Hive。

- 如果您当前使用的是Airflow调度Livy的方式，目前还是需要改造成命令行的形式。DLA Spark团队正在开发Livy兼容版本，以降低迁移成本，具体请联系专家服务。

5.2. Jupyter交互式作业开发

为了支持Spark REPL功能，阿里云数据湖分析团队推出了本地安装Jupyter Lab和DLA Proxy、使用Docker快速启动环境两种方案，帮助用户将本地Jupyter Lab和阿里云DLA Spark连接在一起，从而可以利用DLA的弹性资源进行交互测试和计算。

注意事项

- DLA Spark当前支持python3，Scala 2.11的Jupyter可交互作业。
- 新版Jupyter Lab对Python的最低版本要求是Python 3.6。
- 推荐使用Docker快速启动环境方式来使用本功能。
- 交互式作业会在空闲一段时间后自动释放，默认释放时间为最后一个代码块执行完毕后1200秒。用户可以通过 `spark.dla.session.ttl` 来配置空闲多长时间后自动释放交互式作业。

本地安装Jupyter Lab和DLA Proxy

1. 安装DLA Livy Proxy。
   i. 安装Aliyun OpenAPI SDK。

   > **说明** Aliyun OpenAPI SDK的最低版本要求是 `2.0.4` 。

   ii. 使用以下命令安装Aliyun DLA Livy Proxy。

   ```
   pip install aliyun-dla-livy-proxy-0.0.5.zip
   ```

   > **说明** 您需要使用root用户安装Aliyun DLA Livy Proxy，非root用户进行安装可能无法注册命令到可执行目录中。Aliyun DLA Livy Proxy安装完成后，您可以在命令行中找到 `dlaproxy` 命令。

   iii. 启动DLA Livy Proxy。

   DLA Livy Proxy用于将阿里云DLA的接口翻译为 `SparkMagic` 需要的 `Apache Livy` 语义接口，从而在本地建立HTTP PROXY监听与转发，端口默认是 `5000` 。

```
# 查看dlaproxy命令使用方式。
$dlaproxy -h
usage: dlaproxy [-h] --vcname VCNAME -i AK -k SECRET --region REGION [--host HOST]
[--port PORT] [--loglevel LOGLEVEL]
Proxy AliYun DLA as Livy
optional arguments:
  -h, --help            show this help message and exit
  --vcname VCNAME       Virtual Cluster Name
  -i AK, --access-key-id AK
                        Aliyun Access Key Id
  -k SECRET, --access-key-secret SECRET
                        Aliyun Access Key Secret
  --region REGION       Aliyun Region Id
  --host HOST           Proxy Host Ip
  --port PORT           Proxy Host Port
  --loglevel LOGLEVEL   python standard log level
# 直接启动dla livy proxy。
dlaproxy --vcname <vcname> -i akid -k aksec --region <regionid>
```

上述代码中出现的输入参数说明如下：

参数名称	参数说明
--vcname	DLA Spark虚拟集群名称。 **说明** 您可以登录Data Lake Analytics控制台，在**虚拟集群管理 > 集群名称/实例ID > 详情**中查看虚拟集群名称。
-i	RAM用户的AccessKey ID。 **说明** 如果您已经创建了AccessKey，可以在RAM控制台上查看相关信息。关于如何查看以及创建AccessKey，请参见为RAM用户创建访问密钥。
-k	RAM用户的AccessKey Secret。 **说明** 如果您已经创建了AccessKey，可以在RAM控制台上查看相关信息。关于如何查看以及创建AccessKey，请参见为RAM用户创建访问密钥。
--region	DLA所在地域对应的Region ID，详细信息请参见地域和可用区。
--host	服务绑定的Host，默认为127.0.0.1， 仅代理本地请求。可以修改为0.0.0.0或者其它地址监听公网/内网的请求并代理，建议使用默认值。
--port	监听端口， 默认为5000， 可以修改为其它端口，建议使用默认值。
--loglevel	日志级别， 默认为INFO，可以修改为ERROR、 WARNING、 INFO或DEBUG，建议使用默认值。

2. 安装Jupyter Lab。

i. （可选）安装venv。

> **说明** 推荐将整套环境安装到Virtual Environment环境中，这样后续的安装不会破坏主账号下的公共Python环境。

ii. 使用以下命令安装Jupyter lab。

```
pip install jupyterlab #安装Jupyter Lab。
jupyter lab  #验证一下是否安装成功，成功安装的话，可以看到启动日志。
```

iii. 按照如下步骤安装Sparkmagic。

a. 安装库。

```
pip install sparkmagic
```

b. 确保ipywidgets成功运行。

```
jupyter nbextension enable --py --sys-prefix widgetsnbextension
```

c. 如果您使用的是JupyterLab，需要运行如下命令。

```
jupyter labextension install "@jupyter-widgets/jupyterlab-manager"
```

d. 使用 `pip show sparkmagic` 找到对应的文件夹，进入文件夹执行如下命令安装kernels。

```
jupyter-kernelspec install sparkmagic/kernels/sparkkernel
jupyter-kernelspec install sparkmagic/kernels/pysparkkernel
jupyter-kernelspec install sparkmagic/kernels/sparkrkernel
```

e. 修改 `~/.sparkmagic/config.json` 上的配置文件，详情请参见example_config.json。

f. 运行。

```
jupyter serverextension enable --py sparkmagic
```

安装成功后，需要手动创建配置文件 `~/.sparkmagic/config.json` ，并将 `url` 的配置指向本地开启的代理。示例如下：

```
{
  "kernel_python_credentials" : {
    "username": "",
    "password": "",
    "url": "http://127.0.0.1:5000",
    "auth": "None"
  },
  "kernel_scala_credentials" : {
    "username": "",
    "password": "",
    "url": " http://127.0.0.1:5000",
    "auth": "None"
  },
  "kernel_r_credentials": {
    "username": "",
    "password": "",
    "url": "http://localhost:5000"
  },
```

```
  "logging_config": {
    "version": 1,
    "formatters": {
      "magicsFormatter": {
        "format": "%(asctime)s\t%(levelname)s\t%(message)s",
        "datefmt": ""
      }
    },
    "handlers": {
      "magicsHandler": {
        "class": "hdijupyterutils.filehandler.MagicsFileHandler",
        "formatter": "magicsFormatter",
        "home_path": "~/.sparkmagic"
      }
    },
    "loggers": {
      "magicsLogger": {
        "handlers": ["magicsHandler"],
        "level": "DEBUG",
        "propagate": 0
      }
    }
  },
  "wait_for_idle_timeout_seconds": 15,
  "livy_session_startup_timeout_seconds": 600,
  "fatal_error_suggestion": "The code failed because of a fatal error:\n\t{}.\n\nSome t
hings to try:\na) Make sure Spark has enough available resources for Jupyter to create
a Spark context.\nb) Contact your Jupyter administrator to make sure the Spark magics l
ibrary is configured correctly.\nc) Restart the kernel.",
  "ignore_ssl_errors": false,
  "session_configs": {
    "conf": {
      "spark.dla.connectors": "oss"
    }
  },
  "use_auto_viz": true,
  "coerce_dataframe": true,
  "max_results_sql": 2500,
  "pyspark_dataframe_encoding": "utf-8",
  "heartbeat_refresh_seconds": 30,
  "livy_server_heartbeat_timeout_seconds": 0,
  "heartbeat_retry_seconds": 10,
  "server_extension_default_kernel_name": "pysparkkernel",
  "custom_headers": {},
  "retry_policy": "configurable",
  "retry_seconds_to_sleep_list": [0.2, 0.5, 1, 3, 5],
  "configurable_retry_policy_max_retries": 8
}
```

说明 示例中的 `session_configs` 是提交到DLA Spark的 `conf` 部分，如果需要加载JAR包，连接DLA的元数据服务请参考作业配置指南。

DLA Livy Proxy启动后监听的默认地址为 `127.0.0.1:5000` ，如果在启动DLA Livy Proxy时修改了默认

的Host和Port，则需要修改上述配置文件中的 `url` 部分。示例：启动时指定地址 `--host 192.168.1.3 --port 8080` ，则应当将上述配置文件中的 `url` 为 `http://192.168.1.3:8080` ，修改后如下所示：

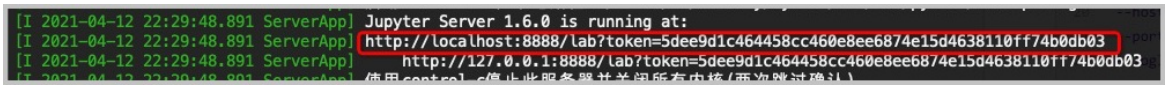
```
{
  "kernel_python_credentials" : {
    "username": "",
    "password": "",
    "url": "http://192.168.1.3:8080",
    "auth": "None"
  },
  "kernel_scala_credentials" : {
    "username": "",
    "password": "",
    "url": "http://192.168.1.3:8080",
    "auth": "None"
  },
  "kernel_r_credentials": {
    "username": "",
    "password": "",
    "url": "http://192.168.1.3:8080"
  },
 "logging_config": {
    "version": 1,
    "formatters": {
      "magicsFormatter": {
        "format": "%(asctime)s\t%(levelname)s\t%(message)s",
        "datefmt": ""
      }
    },
    "handlers": {
      "magicsHandler": {
        "class": "hdijupyterutils.filehandler.MagicsFileHandler",
        "formatter": "magicsFormatter",
        "home_path": "~/.sparkmagic"
      }
    },
    "loggers": {
      "magicsLogger": {
        "handlers": ["magicsHandler"],
        "level": "DEBUG",
        "propagate": 0
      }
    }
  },
  "wait_for_idle_timeout_seconds": 15,
  "livy_session_startup_timeout_seconds": 600,
  "fatal_error_suggestion": "The code failed because of a fatal error:\n\t{}.\n\nSome t
hings to try:\na) Make sure Spark has enough available resources for Jupyter to create
a Spark context.\nb) Contact your Jupyter administrator to make sure the Spark magics l
ibrary is configured correctly.\nc) Restart the kernel.",
  "ignore_ssl_errors": false,
  "session_configs": {
    "conf": {
```

```
        "spark.dla.connectors": "oss"
      }
    },
    "use_auto_viz": true,
    "coerce_dataframe": true,
    "max_results_sql": 2500,
    "pyspark_dataframe_encoding": "utf-8",
    "heartbeat_refresh_seconds": 30,
    "livy_server_heartbeat_timeout_seconds": 0,
    "heartbeat_retry_seconds": 10,
    "server_extension_default_kernel_name": "pysparkkernel",
    "custom_headers": {},
    "retry_policy": "configurable",
    "retry_seconds_to_sleep_list": [0.2, 0.5, 1, 3, 5],
    "configurable_retry_policy_max_retries": 8
}
```

3. 运行Jupyter Lab。

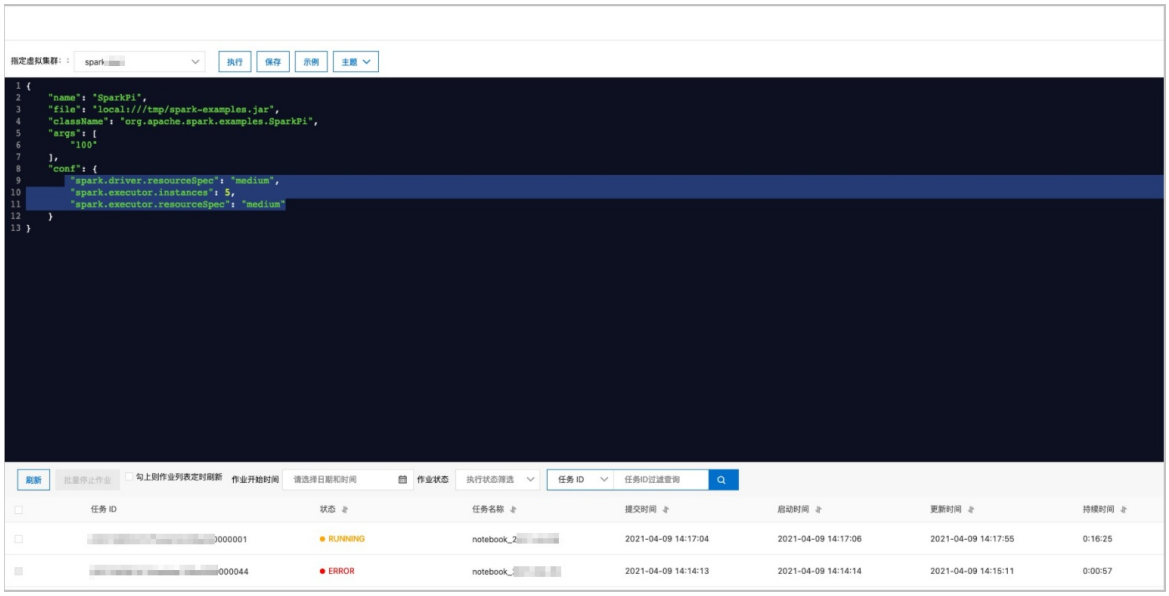
```
# 重新启动Jupyter Lab。
jupyter lab
# 启动DLA Livy Proxy。
dlaproxy --vcname vcname -i akid -k aksec --region <regionid>
```

Jupyter Lab启动日志中会打印Jupyter Lab的本地地址。如下图所示：



当系统出现提示 `Aliyun DLA Proxy is ready` ，表明DLA Livy Proxy启动成功。DLA Livy Proxy启动成功后，您就可以正常使用Jupyter Lab了。Jupyter Lab的使用文档，请参考jupyterlab官方地址。

运行Jupyter Lab任务，DLA中会自动创建DLA Spark作业，您可以登录Data Lake Analytics控制台，在**Serverless Spark > 作业管理**菜单进行查看和管理。如下图所示，名称以 `notebook_` 为开头的Spark作业即为Jupyter交互式作业。



当Jupyter Lab运行起来后，您仍然可以动态的修改配置，修改方式是在Jupyter Lab代码单元中使用 `magic` 语句来覆盖旧的配置项。执行后，Jupyter Lab会根据新的配置项重启作业。

```
# 重新启动Jupyter Lab。
jupyter lab
# 启动DLA Livy Proxy。
dlaproxy --vcname vcname -i akid -k aksec --region <regionid>
```

```
%%configure -f
{
    "conf": {
      "spark.sql.hive.metastore.version": "dla",
      "spark.dla.connectors": "oss"
    }
}
```

如果需要使用自制的依赖方法，您可以参考下述配置。

```
%%configure -f
{
    "conf": {
      ...
    },
    "pyFiles": "oss://{your bucket name}/{path}/*.zip" # module
}
```

4. 关闭Jupyter Lab作业。

   单击JupyterLab Kernel菜单栏下的**Restart Kernel**。

使用Docker快速启动环境

阿里云数据湖团队同时提供Docker镜像快速启动一个Jupyter交互式开发环境，Docker的安装和使用说明请参见Docker官方文档。

1. 安装并启动Docker后，使用如下命令拉取DLA Jupyter镜像。

   ```
   docker pull registry.cn-hangzhou.aliyuncs.com/dla_spark/dla-jupyter:0.5
   ```

2. 拉取成功后，您可以使用如下命令查看此镜像的帮助文件。

   ```
   docker run -ti registry.cn-hangzhou.aliyuncs.com/dla_spark/dla-jupyter:0.5
   Used to run jupyter lab for Aliyun DLA
   Usage example: docker run -it -p 8888:8888 dla-jupyter:0.1 -i akid -k aksec -r cn-hangh
   zou -c spark-vc -l INFO
       -i Aliyun AkId
       -k Aliyun AkSec
       -r Aliyun Region Id
       -c Aliyun DLA Virtual cluster name
       -l LogLevel
   ```

   上述代码中出现的参数和DLA Proxy的参数非常相似，具体说明如下。

参数名称	参数说明
-c	DLA Spark虚拟集群名称。 说明 您可以登录Data Lake Analytics控制台。在虚拟集群管理 > 集群名称/实例ID > 详情中查看虚拟集群名称。
-i	RAM用户的AccessKey ID。 说明 如果您已经创建了AccessKey，可以在RAM控制台上查看相关信息。关于如何查看以及创建AccessKey，请参见为RAM用户创建访问密钥。
-k	RAM用户的AccessKey Secret。 说明 如果您已经创建了AccessKey，可以在RAM控制台上查看相关信息。关于如何查看以及创建AccessKey，请参见为RAM用户创建访问密钥。
-r	DLA所在地域对应的Region ID，详细信息请参见地域和可用区。
-l	日志级别，默认为INFO，可以修改为ERROR、 WARNING、 INFO或DEBUG，建议使用默认值。

3. 输入正确的参数，在本地启动对应的实例。

```
 docker run -it -p 8888:8888 registry.cn-hangzhou.aliyuncs.com/dla_spark/dla-jupyter:0.5 -i {AkId} -k {AkSec} -r {RegionId} -c {VcName}
```

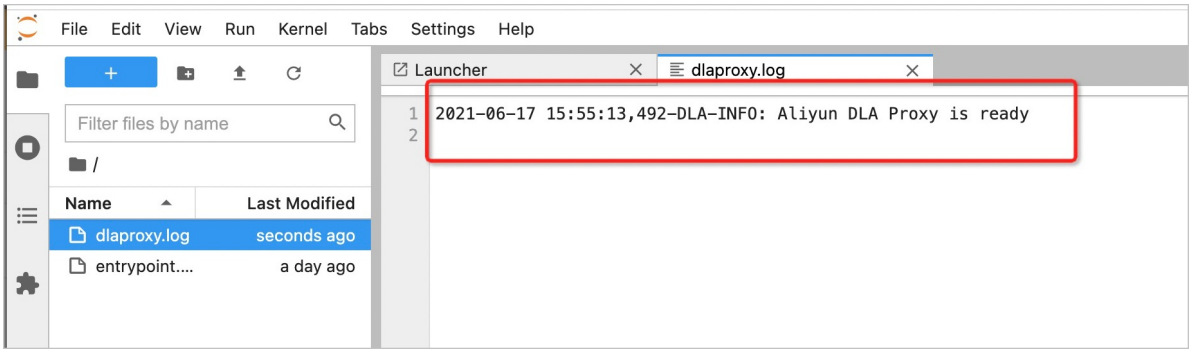
启动成功后您可以看到如下图所示的提示信息，将标注框中的网址贴入浏览器可开始使用Jupyter服务直连DLA Spark。

```
[I 2021-06-17 15:51:27.714 ServerApp] http://fa7def777f38:8888/lab?token=6d11a3b64b7a42b0cd7e241a1471233a9f3a7d5e1765242a
[I 2021-06-17 15:51:27.714 ServerApp]     http://127.0.0.1:8888/lab?token=6d11a3b64b7a42b0cd7e241a1471233a9f3a7d5e1765242a
[I 2021-06-17 15:51:27.714 ServerApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[W 2021-06-17 15:51:27.719 ServerApp] No web browser found: could not locate runnable browser.
[C 2021-06-17 15:51:27.719 ServerApp]

    To access the server, open this file in a browser:
        file:///root/.local/share/jupyter/runtime/jpserver-13-open.html
    Or copy and paste one of these URLs:
        http://fa7def777f38:8888/lab?token=6d11a3b64b7a42b0cd7e241a1471233a9f3a7d5e1765242a
        http://127.0.0.1:8888/lab?token=6d11a3b64b7a42b0cd7e241a1471233a9f3a7d5e1765242a
```

注意事项

- 在排查错误时需要观察底层日志，文件浏览框中的 `dlaproxy.log` 即为日志记录文件，正确的启动信息如下图所示。



- 在不挂载宿主机文件夹的情况下，关闭Docker后编辑的文件会丢失。Docker关闭时也会自动尝试终止所有正在运行的Spark交互式作业。此时，您可以选择如下两种方案。
  - 在关闭Docker前保证所有的文件被妥善的复制保管。
  - 将本地文件夹挂载到Docker镜像中，并将作业文件存储在对应的文件夹下。

    以Linux环境为例，将文件夹 `/home/admin/notebook` 挂载到Docker实例 `/root/notebook` 文件夹下，启动命令如下。

    ```
    docker run -it --privileged=true -p 8888:8888  -v /home/admin/notebook:/root/notebook registry.cn-hangzhou.aliyuncs.com/dla_spark/dla-jupyter:0.5 -i {AkId} -k {AkSec} -r {RegionId} -c {VcName}
    ```

    需要注意将编辑中的notebook最终另存到 `/tmp` ，关闭Docker示例后，在宿主机的 `/home/admin/notebook` 文件夹下可以看到对应的文件，下次启动Docker实例时可以接着工作。

    > **说明** 更多信息，请参见 Docker卷管理文档。

常见问题处理

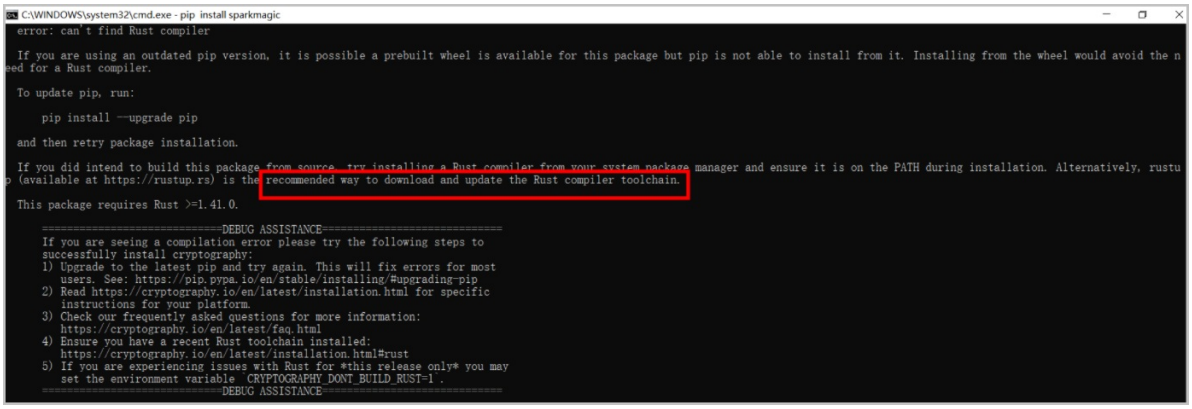
- 问题现象：Jupyter Lab启动失败，出现如下报错，如何处理？
  - `[C 09:53:15.840 LabApp] Bad config encountered during initialization:`
  - `[C 09:53:15.840 LabApp] Could not decode '\xe6\x9c\xaa\xe5\x91\xbd\xe5\x90\x8d' for unicode trait 'untitled_notebook' of a LargeFileManager instance.`

  解决方法：LANG=zn jupyter lab。
- 问题现象：出现报错 `$ jupyter nbextension enable --py --sys-prefix widgetsnbextension Enabling notebook extension jupyter-js-widgets/extension... - Validating: problems found: - require? X jupyter-js-widgets/extension` ，如何处理？

  解决方法： `jupyter nbextension install --py widgetsnbextension --user` 和 `jupyter nbextension enable widgetsnbextension --user --py` 。
- 问题现象：出现报错 `ValueError: Please install nodejs >=12.0.0 before continuing. nodejs may be installed using conda or directly from the nodejs website.` ，如何处理？

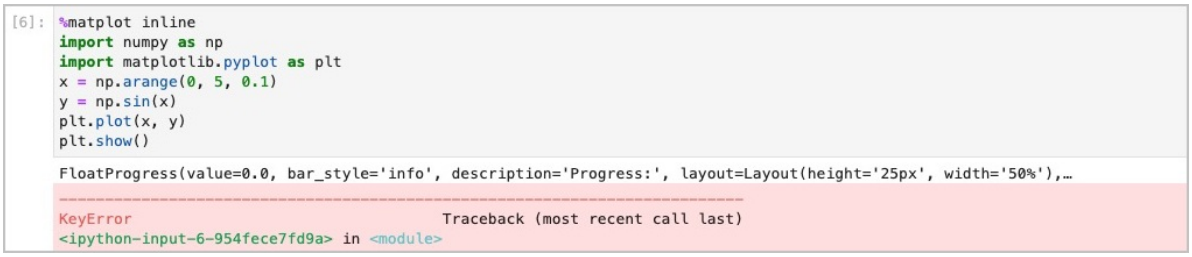
  解决方法：conda install nodejs。关于安装Conda请参考Conda官方文档。

- 问题现象：安装Sparkmagic时报错失败，如下图所示，如何处理？

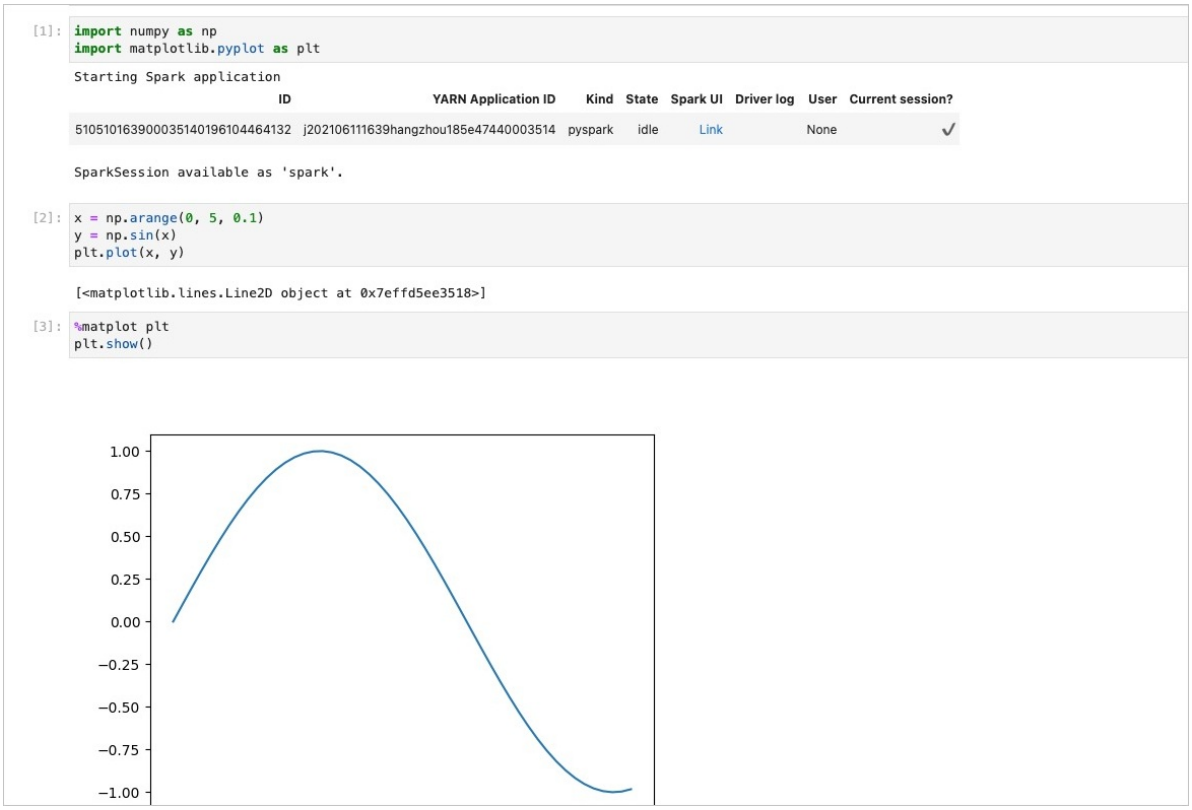


解决方法：安装Rust。

- 问题现象：无法使用matplot进行绘图，即使加入了 `%matplotlib inline` 仍然会报错如下图所示。



解决方法：当使用云端PySpark时，使用 `%matplot plt` 组合 `plt.show()` 绘制图表，效果如下图所示。



5.3. Spark-Submit命令行工具

本文主要介绍了如何操作Spark-Submit命令行工具以及相关示例。

安装工具

1. 获取SparkSubmit工具包。

   您也可以通过wget的方式获取*dla-spark-toolkit.tar.gz*。

   ```
   wget https://dla003.oss-cn-hangzhou.aliyuncs.com/dla_spark_toolkit/dla-spark-toolkit.tar.gz
   ```

2. *dla-spark-toolkit.tar.gz*下载成功后，解压工具包。

   ```
   tar zxvf dla-spark-toolkit.tar.gz
   ```

   > **说明** 该工具包支持JDK8或以上版本。

参数配置

通过命令行配置*conf/spark-defaults.conf*的常用参数。*spark-defaults.conf*目前支持下列常用参数的配置：

```
#  cluster information
# AccessKeyId
#keyId =
#  AccessKeySecret
#secretId =
#  RegionId
#regionId =
#  set vcName
#vcName =
#  set ossKeyId,  if not set will use --keyId value or keyId value
#ossKeyId =
# set ossSecretId if not set will use --secretId value or secretId value
#ossSecretId =
#  set OssUploadPath, if you need upload local resource
#ossUploadPath =
##spark conf
#  driver specifications : small(1c4g) | medium (2c8g) | large (4c16g) | xalrge (8c32g)
#spark.driver.resourceSpec =
#  executor instance number
#spark.executor.instances =
#  executor specifications : small(1c4g) | medium (2c8g) | large (4c16g) | xalrge (8c32g)
#spark.executor.resourceSpec =
#  when use ram,  role arn
#spark.dla.roleArn =
#  config dla oss connectors
#spark.dla.connectors = oss
#  config eni, if you want to use eni
#spark.dla.eni.enable = true
#spark.dla.eni.vswitch.id =
#spark.dla.eni.security.group.id =
#  config log location, need an oss path to store logs
#spark.dla.job.log.oss.uri =
#  config spark read dla table when use option -f or -e
#spark.sql.hive.metastore.version = dla
## any other user defined spark conf...
```

其中 `keyId` 、 `secretId` 、 `regionId` 、 `vcName` 必须要进行配置，参数说明如下：

参数名称	参数说明
keyId	您的阿里云AccessKeyId。
secretId	您的阿里云AccessKeySecret。
vcName	您的虚拟集群的名称。
regionId	您的虚拟集群所在的地域，地区和regionId的对照关系，请参见地域和可用区。

您可以输入如下命令查看该工具的命令行使用帮助。

```
cd /path/to/dla-spark-toolkit
./bin/spark-submit --help
```

运行上述帮助命令后，结果如下所示：

```
Info: Usage: spark-submit [options] <app jar> [app arguments]
Usage: spark-submit --list [PAGE_NUMBER] [PAGE_SIZE]
Usage: spark-submit --kill [JOB_ID]
Info:
Options:
  --keyId                           Your ALIYUN_ACCESS_KEY_ID, is same as `keyId` in conf
/spark-defaults.conf or --conf spark.dla.access.key.id=<value>, required
  --secretId                        Your ALIYUN_ACCESS_KEY_SECRET, is same as `secretId`
in conf/spark-defaults.conf or --conf spark.dla.access.secret.id=<value>, required
  --regionId                        Your Cluster Region Id, is same as `regionId` in conf
/spark-defaults.conf or --conf spark.dla.region.id=<value>,  required
  --vcName                          Your Virtual Cluster Name, is same as `vcName` in con
f/spark-defaults.conf or --conf spark.dla.vc.name=<value>, required
  --oss-keyId                       Your ALIYUN_ACCESS_KEY_ID to upload local resource to
oss,
                                    by default, the value will take from --keyId, is same
as `ossKeyId` in conf/spark-defaults.conf or --conf spark.dla.oss.access.key.id=<value>
  --oss-secretId                    Your ALIYUN_ACCESS_KEY_SECRET to upload local resourc
e to oss,
                                    default the value will take from --secretId, is same
as `ossSecretId` in conf/spark-defaults.conf or --conf spark.dla.oss.access.secret.id=<valu
e>
  --oss-endpoint                    Oss endpoint where the resource will upload. default
is http://oss-$regionId.aliyuncs.com,
                                    is same as `ossEndpoint` in conf/spark-defaults.conf
or --conf spark.dla.oss.endpoint=<value>
  --oss-upload-path                 The user oss path where the resource will upload
                                    If you want to upload a local jar package to the OSS
directory,
                                    you need to specify this parameter. It is same as `os
sUploadPath` in conf/spark-defaults.conf or --conf spark.dla.oss.upload.path=<value>
  --class CLASS_NAME                Your application's main class (for Java / Scala apps)
.
  --name NAME                       A name of your application.
  --jars JARS                       Comma-separated list of jars to include on the driver
                                    and executor classpaths.
  --conf PROP=VALUE                 Arbitrary Spark configuration property, or you can se
t conf in conf/spark-defaults.conf
  --help, -h                        Show this help message and exit.
  --driver-resource-spec            Indicates the resource specifications used by the dri
ver:
                                    small | medium | large | xlarge | 2xlarge
                                    you can also set this value through --conf spark.driv
er.resourceSpec=<value>
  --executor-resource-spec          Indicates the resource specifications used by the exe
cutor:
                                    small | medium | large | xlarge | 2xlarge
                                    you can also set this value through --conf spark.exec
utor.resourceSpec=<value>
  --num-executors                   Number of executors to launch, you can also set this
value through --conf spark.executor.instances=<value>
  --driver-memory MEM               Memory for driver (e.g. 1000M, 2G)
                                    you can also set this value through --conf spark.driv
```

```
er.memory=<value>
  --driver-cores NUM                  Number of cores used by the driver
                                      you can also set this value through --conf spark.driv
er.cores=<value>
  --driver-java-options               Extra Java options to pass to the driver
                                      you can also set this value through --conf spark.driv
er.extraJavaOptions=<value>
  --executor-memory MEM               Memory per executor (e.g. 1000M, 2G)
                                      you can also set this value through --conf spark.exec
utor.memory=<value>
  --executor-cores NUM                Number of cores per executor.
                                      you can also set this value through --conf spark.exec
utor.cores=<value>
  --properties-file                   Spark default conf file location, only local files ar
e supported, default conf/spark-defaults.conf
  --py-files PY_FILES                 Comma-separated list of .zip, .egg, or .py files to p
lace
                                      on the PYTHONPATH for Python apps
  --files FILES                       Comma-separated list of files to be placed in the wor
king
                                      directory of each executor. File paths of these files
                                      in executors can be accessed via SparkFiles.get(fileN
ame).
                                      Specially, you can pass in a custom log output format
file named `log4j.properties`
                                      Note: The file name must be `log4j.properties` to tak
e effect
  --archives                          Comma separated list of archives to be extracted into
the
                                      working directory of each executor. Support file type
s: zip、tgz、tar、tar.gz
  --status job_id                     If given, requests the status and details of the job
specified
  --verbose                           Print additional debug output
  --version, -v                       Print out the dla-spark-toolkit version.
  List Spark Job Only:
  --list                              List Spark Job, should use specify --vcName and --reg
ionId
  --pagenumber, -pn                   Set page number which want to list (default: 1)
  --pagesize, -ps                     Set page size which want to list (default: 1)
  Get Job Log Only:
  --get-log job_id                    Get job log
  Kill Spark Job Only:
  --kill job_id                       Specifies the jobid to be killed
  Spark Offline SQL options:
  -e <quoted-query-string>            SQL from command line. By default, use SubmitSparkSQL
API to submit SQL, support set command to set spark conf.
                                      you can set --disable-submit-sql to submit an SQL job
using the previous SubmitSparkJob API,
                                      which requires the user to specified the --oss-upload
-path
  -f <filename>                       SQL from files. By default, use SubmitSparkSQL API to
submit SQL, support set command to set spark conf.
                                      you can set --disable-submit-sql to submit an SQL job
```

```
using the previous SubmitSparkJob API,
                                    which requires the user to specified the --oss-upload
-path
  -d,--define <key=value>           Variable substitution to apply to spark sql
                                    commands. e.g. -d A=B or --define A=B
  --hivevar <key=value>             Variable substitution to apply to spark sql
                                    commands. e.g. --hivevar A=B
  --hiveconf <property=value>       Use value for given property, DLA spark toolkit will
add `spark.hadoop.` prefix to property
  --database <databasename>         Specify the database to use
  --enable-inner-endpoint           It means that DLA pop SDK and OSS SDK will use the en
dpoint of Intranet to access DLA,
                                    you can turn on this option when you are on Alibaba c
loud's ECS machine.
  Inner API options:
  --api-retry-times                 Specifies the number of retries that the client fails
to call the API, default 3.
  --time-out-seconds                Specifies the timeout for the API(time unit is second
(s)), which is considered a call failure.
                                    default 10s.
```

说明 Spark-Submit工具的脚本将自动读取 `conf/spark-defaults.conf` 中的配置信息。如果您通过命令行修改了 `conf/spark-defaults.conf` 中的参数取值，Spark-Submit工具将获取命令行提交的参数值。

配置兼容性说明

为了方便兼容开源社区的Spark-Submit，以下非开源社区的选项也可以通过Spark Conf进行设置。

```
--keyId             #--conf spark.dla.access.key.id=<value>
--secretId          #--conf spark.dla.access.secret.id=<value>
--regionId          #--conf spark.dla.region.id=<value>
--vcName            #--conf spark.dla.vc.name=<value>
--oss-keyId         #--conf spark.dla.oss.access.key.id=<value>
--oss-secretId      #--conf spark.dla.oss.access.secret.id=<value>
--oss-endpoint      #--conf spark.dla.oss.endpoint=<value>
--oss-upload-path   #--conf spark.dla.oss.upload.path=<value>
```

以下选项暂不支持，或者对于DLA Spark无意义，它们的值会被忽略。

```
Useless options(these options will be ignored):
  --deploy-mode
  --master
  --packages, please use `--jars` instead
  --exclude-packages
  --proxy-user
  --repositories
  --keytab
  --principal
  --queue
  --total-executor-cores
  --driver-library-path
  --driver-class-path
  --supervise
  -S,--silent
  -i <filename>
```

由于DLA Spark的Driver和Executor是运行在弹性容器上的，而弹性容器只能选取某些固定的资源规格，DLA Spark目前支持的资源规格请参见Serverless Spark概述。因此，阿里云数据湖分析团队对Spark-Submit中设置资源规格参数的选项进行了处理。开源社区Spark-Submit和DLA-Spark-Toolkit表现不一致的参数说明如下表所示：

参数名称	说明
--driver-cores/--conf spark.driver.cores	设置Driver的核数。DLA-Spark-Toolkit会选择最接近用户指定的核数的资源规格并且该资源规格的核数大于等于用户指定的核数。
--driver-memory/--conf spark.driver.memory	设置Driver的内存。DLA-Spark-Toolkit会选择最接近用户指定的内存的资源规格并且该资源规格的内存大于等于用户指定的内存。
--executor-cores/--conf spark.executor.cores	设置Executor的核数。DLA-Spark-Toolkit会选择最接近用户指定的核数的资源规格并且该资源规格的核数大于等于用户指定的核数。
--executor-memory/--conf spark.executor.memory	设置Executor的内存。DLA-Spark-Toolkit会选择最接近用户指定的内存的资源规格并且该资源规格的内存大于等于用户指定的内存。

DLA-Spark-Toolkit其他独有的参数如下表所示：

参数名称	说明
--driver-resource-spec	指定Driver的资源规格。优先级高于 `--driver-cores/--conf spark.driver.cores` ，即如果同时指定 `--driver-resource-spec` 和 `--driver-cores` ，则取 `--driver-resource-spec` 指定的资源规格。

参数名称	说明
--executor-resource-spec	指定Driver的资源规格。优先级高于 `--executor-cores/--conf spark.executor.cores` 。
--api-retry-times	DLA-Spark-Toolkit内部执行命令失败时的重试次数（除提交命令外），提交任务不是一个幂等操作，由于网络超时等原因导致的提交失败，实际上任务可能在后台成功，为防止任务重复提交，提交任务失败将不会重试。您需要自行获取已经提交的任务列表（--list），或者去DLA Spark控制台查看任务列表判断任务是否提交成功。
--time-out-seconds	DLA-Spark-Toolkit内部默认的网络超时时间，默认为10，超时命令将会失败重试。
--enable-inner-endpoint	当用户处于阿里云ECS机器上时，可以指定此选项，DLA-Spark-Toolkit将使用内网环境来访问DLA Pop SDK和OSS SDK，内网环境比公网环境稳定。
--list	用于获取作业列表，常搭配 --pagesize和 --pagenumber使用，分别用于指定列表的页数和和一页显示的作业个数，默认值分别为1，10（代表返回第一页的10个作业）。
--kill	接收一个JobID作为参数，用于终止对应的JobID作业。
--get-log	接收一个JobID作为参数，用于获取对应的JobID作业的日志。
--status	接收一个JobID作为参数，用于获取对应的JobID作业的详细情况。

如何提交作业

- 在作业管理页面，提交Spark作业需要按照JSON格式，如下所示：

```
{
    "name": "xxx",
    "file": "oss://{bucket-name}/jars/xxx.jar",
    "jars": "oss://{bucket-name}/jars/xxx.jar,oss://{bucket-name}/jars/xxx.jar"
    "className": "xxx.xxx.xxx.xxx.xxx",
    "args": [
        "xxx",
        "xxx"
    ],
    "conf": {
        "spark.executor.instances": "1",
        "spark.driver.resourceSpec": "medium",
        "spark.executor.resourceSpec": "medium",
        "spark.dla.job.log.oss.uri": "oss://{bucket-name}/path/to/log/"
    }
}
```

- 使用Spark-Submit工具，则按照如下格式提交作业：

```
./bin/spark-submit  \
--class xxx.xxx.xxx.xxx.xxx \
--verbose \
--name xxx \
--jars oss://{bucket-name}/jars/xxx.jar,oss://{bucket-name}/jars/xxx.jar
--conf spark.driver.resourceSpec=medium \
--conf spark.executor.instances=1 \
--conf spark.executor.resourceSpec=medium \
oss://{bucket-name}/jars/xxx.jar \
args0 args1
##主程序文件,--jars指定的jar包， --py-files --files均支持本地文件路径和oss文件路径。
##指定的本地资源需要使用绝对路径，spark-submit会自动上传本地文件资源至用户指定的oss目录
##用户可使用 --oss-upload-path 或者在spark-defaults.conf中设置ossUploadPath的值来指定上传到oss的目录
##资源上传时，会使用md5校验文件内容，当指定的oss目录中有相同文件名且md5相同时，将不再重复上传。
##注意您如果手动更新了oss上传目录的jar包,请删除掉对应md5文件
##格式：--jars  /path/to/local/directory/XXX.jar,/path/to/local/directory/XXX.jar
##多个文件以逗号隔开， 文件指定绝对路径
## --jars， --py-files --files 也支持指定本地目录，会上传该目录下所有的文件（不递归上传子目录内容）
## 目录路径也需要指定绝对路径 如 --jars /path/to/local/directory/,/path/to/local/directory2/
## 多个目录以逗号隔开,目录使用绝对路径
##程序输出， 可通过查看下方输出的SparkUI连接访问作业的SparkUI， 和 JobDetail查看作业提交的参数是否符合预期
Info: job status: starting
Info: job status: starting
Info: job status: starting
Info: job status: starting
Info: job status: starting
Info: job status: starting
Info: job status: starting
Info: job status: starting
Info: job status: starting
Info: job status: starting
Info: job status: starting
Info: job status: running
{
  "jobId": "",
  "jobName": "SparkPi",
  "status": "running",
  "detail": "",
  "sparkUI": "",
  "createTime": "2020-08-20 14:12:07",
  "updateTime": "2020-08-20 14:12:07",
  ...
}
Job Detail: {
  "name": "SparkPi",
  "className": "org.apache.spark.examples.SparkPi",
  "conf": {
    "spark.driver.resourceSpec": "medium",
    "spark.executor.instances": "1",
    "spark.executor.resourceSpec": "medium"
  },
```

```
  "file": "",
  "sparkUI":"https://xxx"
}
```

spark-submit作业退出码说明如下：

```
255    #代表作业执行失败
0      #代表作业执行成功
143    #代表作业被kill
```

> **说明**
> - 使用子账号AccessKeyId、AccessKeySecret提交作业请参考细粒度配置RAM子账号权限。
> - 由于Spark-Submit上传本地jar包时，需要RAM账号拥有对OSS的访问权限，您可以在用户页面为相应子账户添加AliyunOSSFullAccess权限。

如何结束Spark作业

执行如下命令即可结束Spark作业：

```
./spark-submit \
--kill <jobId>
```

结果如下：

```
## 打印结果
Info: kill job: jxxxxxxxx, response: null
```

如何查看Spark作业列表

您可以通过命令行的方式查看作业，例如指定查看任务列表中第1页，总共1个作业：

```
./bin/spark-submit \
--list --pagenumber 1 --pagesize 1
```

结果如下：

```
## 打印结果
{
  "requestId": "",
  "dataResult": {
    "pageNumber": "1",
    "pageSize": "1",
    "totalCount": "251",
    "jobList": [
      {
        "createTime": "2020-08-20 11:02:17",
        "createTimeValue": "1597892537000",
        "detail": "",
        "driverResourceSpec": "large",
        "executorInstances": "4",
        "executorResourceSpec": "large",
        "jobId": "",
        "jobName": "",
        "sparkUI": "",
        "status": "running",
        "submitTime": "2020-08-20 11:01:58",
        "submitTimeValue": "1597892518000",
        "updateTime": "2020-08-20 11:22:01",
        "updateTimeValue": "1597893721000",
        "vcName": ""
      }
    ]
  }
}
```

如何获取作业提交参数和SparkUI

执行如下命令即可获取：

```
./bin/spark-submit --status <jobId>
```

获取结果如下：

```
## 打印输出
Info: job status: success
Info:
{
  "jobId": "jxxxxxxxx",
  "jobName": "drop database if exists `",
  "status": "success",
  "detail": "xxxxxx",
  "sparkUI": "xxxxxxx",
  "createTime": "2021-05-08 20:02:28",
  "updateTime": "2021-05-08 20:04:05",
  "submitTime": "2021-05-08 20:02:28",
  "createTimeValue": "1620475348180",
  "updateTimeValue": "1620475445000",
  "submitTimeValue": "1620475348180",
  "vcName": "release-test"
}
Info: Job Detail:
set spark.sql.hive.metastore.version=dla;
set spark.dla.connectors=oss;
set spark.executor.instances=1;
set spark.sql.hive.metastore.version = dla;
set spark.dla.eni.enable = true;
set spark.dla.eni.security.group.id = xxxx ;
set spark.dla.eni.vswitch.id = xxxxx;
drop database if exists `my_hdfs_db_1` CASCADE;
```

如何获取作业日志

执行如下命令即可获取：

```
./bin/spark-submit --get-log <jobId>
```

获取结果如下：

```
##打印结果
20/08/20 06:24:57 WARN NativeCodeLoader: Unable to load native-hadoop library for your plat
form... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
20/08/20 06:24:58 INFO SparkContext: Running Spark version 2.4.5
20/08/20 06:24:58 INFO SparkContext: Submitted application: Spark Pi
20/08/20 06:24:58 INFO SecurityManager: Changing view acls to: spark
20/08/20 06:24:58 INFO SecurityManager: Changing modify acls to: spark
20/08/20 06:24:58 INFO SecurityManager: Changing view acls groups to:
20/08/20 06:24:58 INFO SecurityManager: Changing modify acls groups to:
20/08/20 06:24:58 INFO SecurityManager: SecurityManager: authentication disabled; ui acls d
isabled; users  with view permissions: Set(spark); groups with view permissions: Set(); use
rs  with modify permissions: Set(spark); groups with modify permissions: Set()
...
```

5.4. Spark-SQL命令行工具

本文主要介绍如何操作工具Spark-SQL以及相关示例。

前提条件

本工具的安装、配置说明请参考Spark-Submit命令行工具。

提交离线SQL作业

spark-sql工具，提供 `-e` 用于执行以分号隔开的多条sql语句，以及 `-f` 用于执行sql文件中的语句（每条sql语句以分号结尾）。用户可将conf字段指定的配置放入 `conf/spark-defaults.conf` 中，然后按照如下格式提交：

```
## 使用-e命令执行多条语句，每条sql语句使用`;`号隔开。--database用于指定默认的database
$ ./bin/spark-sql \
--verbose \
--database mydb \
--name offlinesql \
-e "select * from t1;insert into table t1 values(4,'test');select * from t1"
## 或者您也可以将sql语句放入到文件中，每条sql语句使用分号`;`隔开，并使用-f参数指向sql文件，仅支持本地文件，
且必须写绝对路径
$ ./bin/spark-sql \
--verbose \
--name offlinesql \
-f /path/to/your/sql/file
##输出结果如下所示
++++++++++++++++++executing sql: select * from t1
| id|name|
|  1|  zz|
|  2|  xx|
|  3|  yy|
|  4|test|
++++++++++++++++++ end ++++++++++++++++++
++++++++++++++++++executing sql: insert into table t1 values(4,'test')
||
++++++++++++++++++ end ++++++++++++++++++
++++++++++++++++++executing sql: select * from t1
| id|name|
|  1|  zz|
|  2|  xx|
|  3|  yy|
|  4|test|
## spark-sql 也支持使用变量替换，您可以使用-d key=value来指定参数的值，随后可以在sql语句中使用${key}来引用key的值。
$ ./bin/spark-sql \
--verbose \
--name variableSubstitution \
-d table=mytable \
-e "use <db>; select * from \${table}"
## 在终端 `$` 符号有特殊含义，所以需要在引用变量`table`时转义$符号，如果是在sql文件中引用变量则无需转义
##替换的sql语句如下
use <db>; select * from mytable
```

6.连接数据源

6.1. Elasticsearch

本文介绍了如何使用DLA Spark访问阿里云Elasticsearch。

前提条件

- 创建了Spark虚拟集群。具体操作请参见创建虚拟集群。
- 创建了Elasticsearch实例，并获得了私网地址和用户名密码。具体操作请参见创建阿里云Elasticsearch实例。
- 配置了Elasticsearch实例的白名单。具体操作请参见配置ES白名单。

使用Java连接Elasticsearch

1. 从Maven官方仓库下载Elasticsearch-spark-20_2.11对应版本的JAR包，本示例下载JAR包为Elasticsearch-spark-20_2.11-6.3.2.jar。
2. 将从官方仓库下载的JAR包上传到一个与Elasticsearch同地域的OSS中，如何上传请参见上传文件。
3. 在本地开发工程中添加如下依赖。

```
<dependencies>
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-core_2.11</artifactId>
        <version>2.4.5</version>
    </dependency>
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-sql_2.11</artifactId>
        <version>2.4.5</version>
    </dependency>
    <dependency>
        <groupId>org.elasticsearch</groupId>
        <artifactId>elasticsearch-spark-20_2.11</artifactId>
        <version>6.7.0</version>
    </dependency>
</dependencies>
```

> 注意 请确保pom依赖中版本与云服务对应版本保持一致，例如Elasticsearch-spark-20_2.11版本与阿里云Elasticsearch版本一致；spark-core_2.11与Spark版本一致。

4. 准备以下测试代码来连接Elasticsearch。您需要替换示例中的 `es.nodes` 为您的私网地址， `es.net.http.auth.pass` 为您自己创建Elasticsearch时设置的密码。

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.Elasticsearch.spark.rdd.api.java.JavaEsSpark;
import java.util.Map;
public class ReadES {
    public static void main(String[] args) {
        SparkConf  conf = new SparkConf().setAppName("readEs").setMaster("local[*]")
                .set("es.nodes", "es-cn-n6w1o1x0w001c****.Elasticsearch.aliyuncs.com")
                .set("es.port", "9200")
                .set("es.net.http.auth.user", "elastic")
                .set("es.net.http.auth.pass", "xxxxxx")
                .set("es.nodes.wan.only", "true")
                .set("es.nodes.discovery","false")
                .set("es.input.use.sliced.partitions","false")
                .set("es.resource", "index_name/_doc")
                .set("es.scroll.size","500");
        JavaSparkContext sc = new JavaSparkContext(conf);
        JavaPairRDD<String, Map<String, Object>> rdd = JavaEsSpark.esRDD(sc);
        for ( Map<String, Object> item : rdd.values().collect()) {
            System.out.println(item);
        }
        sc.stop();
    }
}
```

代码中出现的参数说明如下。

参数名称	说明
`es.nodes`	Elasticsearch的Master角色节点的IP地址，至少需要填入一个Master角色的节点。使用阿里云Elasticsearch时，需填写私网地址。
`es.port`	Elasticsearch服务的端口号。
`es.net.http.auth.user`	Elasticsearch服务的用户名。阿里云Elasticsearch的用户名仅支持elastic。
`es.net.http.auth.pass`	Elasticsearch服务的密码。
`es.nodes.wan.only`	是否通过公网连接云端Elasticsearch服务。使用阿里云Elasticsearch时，必须配置为 `true` 。
`es.nodes.discovery`	是否自动发现更多接入点。使用阿里云Elasticsearch时，必须配置为 `false` ，代表只是用 `es.nodes` 中的接入点。
`es.resource`	Spark读取的Elasticsearch的数据类型，格式为<index>/<type>。

注意 更多配置说明，请参见ES配置官方文档。

5. 将测试代码打包成JAR包上传至您的OSS中，本示例JAR包命名为spark-es-examples-0.0.1-SNAPSHOT-shaded.jar。
6. 登录Data Lake Analytics管理控制台。
7. 在页面左上角，选择Elasticsearch实例所在地域。
8. 单击左侧导航栏中**Serverless Spark > 作业管理**。
9. 提交作业如下。您需要分别替换 `file` 和 `jars` 中的JAR包为您自己测试代码和从官方下载的JAR包名称。

```
{
    "name": "es",
    "className": "com.aliyun.spark.ReadES",
    "conf": {
        "spark.driver.resourceSpec": "medium",
        "spark.executor.instances": 5,
        "spark.executor.resourceSpec": "medium",
        "spark.dla.eni.enable": "true",
        "spark.dla.eni.vswitch.id": "vsw-bp17jqw3lrrobn6y*****",
        "spark.dla.eni.security.group.id": "sg-bp163uxgt4zandx*****"
    },
    "file": "oss://{your bucket}/spark-es-examples-0.0.1-SNAPSHOT-shaded.jar",
    "jars": "oss://{your bucket}/Elasticsearch-spark-20_2.11-6.3.2.jar"
}
```

> **说明** 代码中的参数说明请参见作业配置指南。

使用PySpark连接Elasticsearch

1. 从Maven官方仓库下载Elasticsearch-spark-20_2.11对应版本的JAR包，本示例下载JAR包为Elasticsearch-spark-20_2.11-6.3.2.jar。
2. 将从官方仓库下载的JAR包上传到一个与Elasticsearch同地域的OSS中，如何上传请参见上传文件。
3. 建立如下内容的 `exmaple.py` 。

```
from pyspark import SparkContext, RDD
spark = SparkSession \
    .builder \
    .getOrCreate()
df = spark.read.format("org.Elasticsearch.spark.sql")\
    .option('es.nodes','es-cn-n6w1o1x0w001c****.Elasticsearch.aliyuncs.com') \
    .option('es.port','9200') \
    .option('es.net.http.auth.user', 'xxx') \
    .option('es.net.http.auth.pass', 'xxx') \
    .option("es.nodes.wan.only", "true")\
    .option("es.nodes.discovery", "false")\
    .load("index_name/_doc").show
```

代码中出现的参数说明如下。

参数名称	说明

参数名称	说明
es.nodes	Elasticsearch的Master角色节点的IP地址，至少需要填入一个Master角色的节点。使用阿里云Elasticsearch时，需填写私网地址。
es.port	Elasticsearch服务的端口号。
es.net.http.auth.user	Elasticsearch服务的用户名。阿里云Elasticsearch的用户名仅支持elastic。
es.net.http.auth.pass	Elasticsearch服务的密码。
es.nodes.wan.only	是否通过公网连接云端Elasticsearch服务。使用阿里云Elasticsearch时，必须配置为 true 。
es.nodes.discovery	是否自动发现更多接入点。使用阿里云Elasticsearch时，必须配置为 false ，代表只是用 es.nodes 中的接入点。
es.resource	Spark读取的Elasticsearch的数据类型，格式为<index>/<type>。

> **注意** 更多配置说明，请参见ES配置官方文档。

4. 将 exmaple.py 文件上传到OSS中。
5. 登录Data Lake Analytics管理控制台。
6. 在页面左上角，选择Elasticsearch实例所在地域。
7. 单击左侧导航栏中**Serverless Spark > 作业管理**。
8. 提交作业如下。您需要替换 jars 中的JAR包为您从官方下载的JAR包名称。

```
{
    "name": "es",
    "className": "com.aliyun.spark.ReadES",
    "conf": {
        "spark.driver.resourceSpec": "medium",
        "spark.executor.instances": 5,
        "spark.executor.resourceSpec": "medium",
        "spark.dla.eni.enable": "true",
        "spark.dla.eni.vswitch.id": "vsw-bp17jqw3lrrobn6y*****",
        "spark.dla.eni.security.group.id": "sg-bp163uxgt4zandx*****"
    },
    "file": "oss://{your bucket}/example.py",
    "jars": "oss://{your bucket}/Elasticsearch-spark-20_2.11-6.3.2.jar"
}
```

> **说明** 代码中的参数说明请参见作业配置指南。

6.2. AnalyticDB PostgreSQL

分析型数据库PostgreSQL版（原HybridDB for PostgreSQL）为您提供简单、快速、经济高效的PB级云端数据仓库解决方案。本文主要介绍如何通过DLA Serverless Spark访问云原生数仓AnalyticDB PostgreSQL。

前提条件

- 已经开通对象存储OSS（Object Storage Service）服务。具体操作请参考开通OSS服务。
- 已经创建云原生数仓AnalyticDB PostgreSQL实例。具体请参考创建实例。
- 在AnalyticDB PostgreSQL实例中已创建数据库和表，并插入数据。参考命令样例如下：

```
#建库语句
create database testdb
#建表语句：
CREATE TABLE "test_table"
(
 "name" varchar(32) ,
 "age" smallint ,
 "score" double precision
)
WITH (
    FILLFACTOR = 100,
    OIDS = FALSE
)
;
ALTER TABLE "test_table" OWNER TO testuser;
#插入数据语句：
INSERT INTO "test_table" VALUES('aliyun01', 101, 10.0);
INSERT INTO "test_table" VALUES('aliyun02', 102, 10.0);
INSERT INTO "test_table" VALUES('aliyun03', 103, 10.0);
INSERT INTO "test_table" VALUES('aliyun04', 104, 10.0);
INSERT INTO "test_table" VALUES('aliyun05', 105, 10.0);
```

- 准备DLA Spark访问AnalyticDB PostgreSQL实例所需的安全组ID和交换机ID。具体操作请参见配置数据源网络。
- DLA Spark访问AnalyticDB PostgreSQL实例所需的交换机IP，已添加到AnalyticDB PostgreSQL实例的白名单中。具体操作请参见设置白名单。

操作步骤

1. 准备以下测试代码和依赖包来访问AnalyticDB PostgreSQL，并将此测试代码和依赖包分别编译打包生成jar包上传至您的OSS。

   测试代码示例：

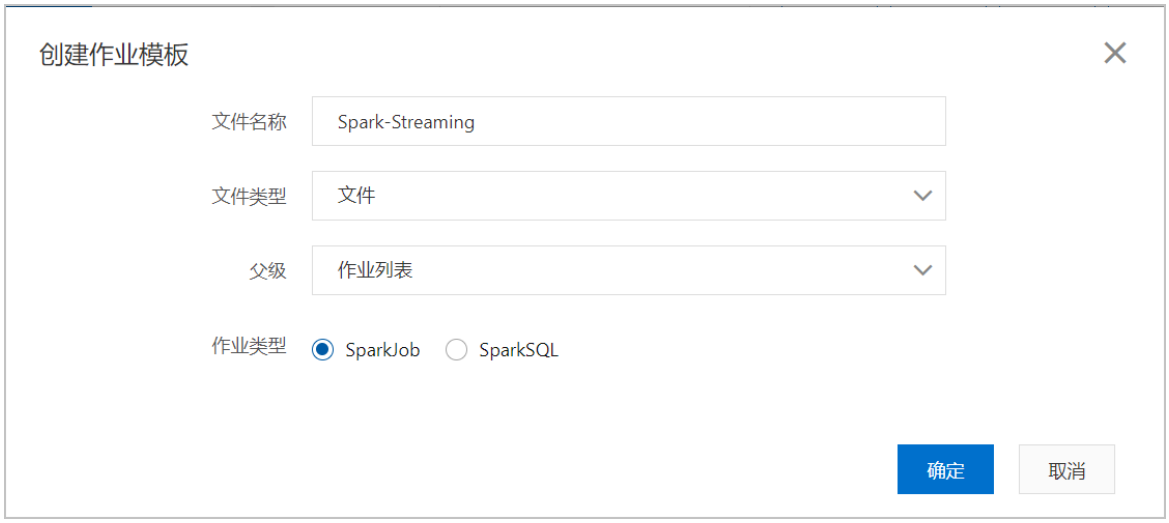
```
package com.aliyun.spark
import java.util.Properties
import org.apache.spark.sql.SparkSession
object SparkOnADBPostgreSQL {
  def main(args: Array[String]): Unit = {
    val url = args(0)
    val database = args(1)
    val jdbcConnURL = s"jdbc:postgresql://$url/$database"
    var schemaName = args(2)
```

```
    val tableName = args(3)
    val user = args(4)
    val password = args(5)
    //Spark侧的表名。
    var sparkTableName = args(6)
    val sparkSession = SparkSession
      .builder()
      .appName("scala spark on adb test")
      .getOrCreate()
    val driver = "org.postgresql.Driver"
    //如果存在的话就删除表。
    sparkSession.sql(s"drop table if exists $sparkTableName")
    //Sql方式，Spark会映射数据库中表的Schema。
    val createCmd =
      s"""CREATE TABLE ${sparkTableName} USING org.apache.spark.sql.jdbc
         |    options (
         |    driver '$driver',
         |    url '$jdbcConnURL',
         |    dbtable '$schemaName.$tableName',
         |    user '$user',
         |    password '$password'
         |    )""".stripMargin
    println(s"createCmd: \n $createCmd")
    sparkSession.sql(createCmd)
    val querySql = "select * from " + sparkTableName + " limit 1"
    sparkSession.sql(querySql).show
    //使用dataset API接口。
    val connectionProperties = new Properties()
    connectionProperties.put("driver", driver)
    connectionProperties.put("user", user)
    connectionProperties.put("password", password)
    //读取数据。
    var jdbcDf = sparkSession.read.jdbc(jdbcConnURL,
      s"$database.$schemaName.$tableName",
      connectionProperties)
    jdbcDf.select("name", "age", "score").show()
    val data =
      Seq(
        PersonADBPG("bill", 30, 170.5D),
        PersonADBPG("gate", 29, 200.3D)
      )
    val dfWrite = sparkSession.createDataFrame(data)
    //写入数据。
    dfWrite
      .write
      .mode("append")
      .jdbc(jdbcConnURL, s"$database.$schemaName.$tableName", connectionProperties)
    jdbcDf.select("name", "age").show()
    sparkSession.stop()
  }
}
case class PersonADBPG(name: String, age: Int, score: Double)
```

AnalyticDB PostgreSQL依赖的pom文件：

```
        <dependency>
            <groupId>org.postgresql</groupId>
            <artifactId>postgresql</artifactId>
            <version>42.2.5</version>
        </dependency>
```

2. 登录Data Lake Analytics管理控制台。
3. 在页面左上角，选择AnalyticDB PostgreSQL实例所在地域。
4. 单击左侧导航栏中的**Serverless Spark > 作业管理**。
5. 在**作业编辑**页面，单击**创建作业**。
6. 在**创建作业模板**页面，按照页面提示进行参数配置后，单击**确定**创建Spark作业。

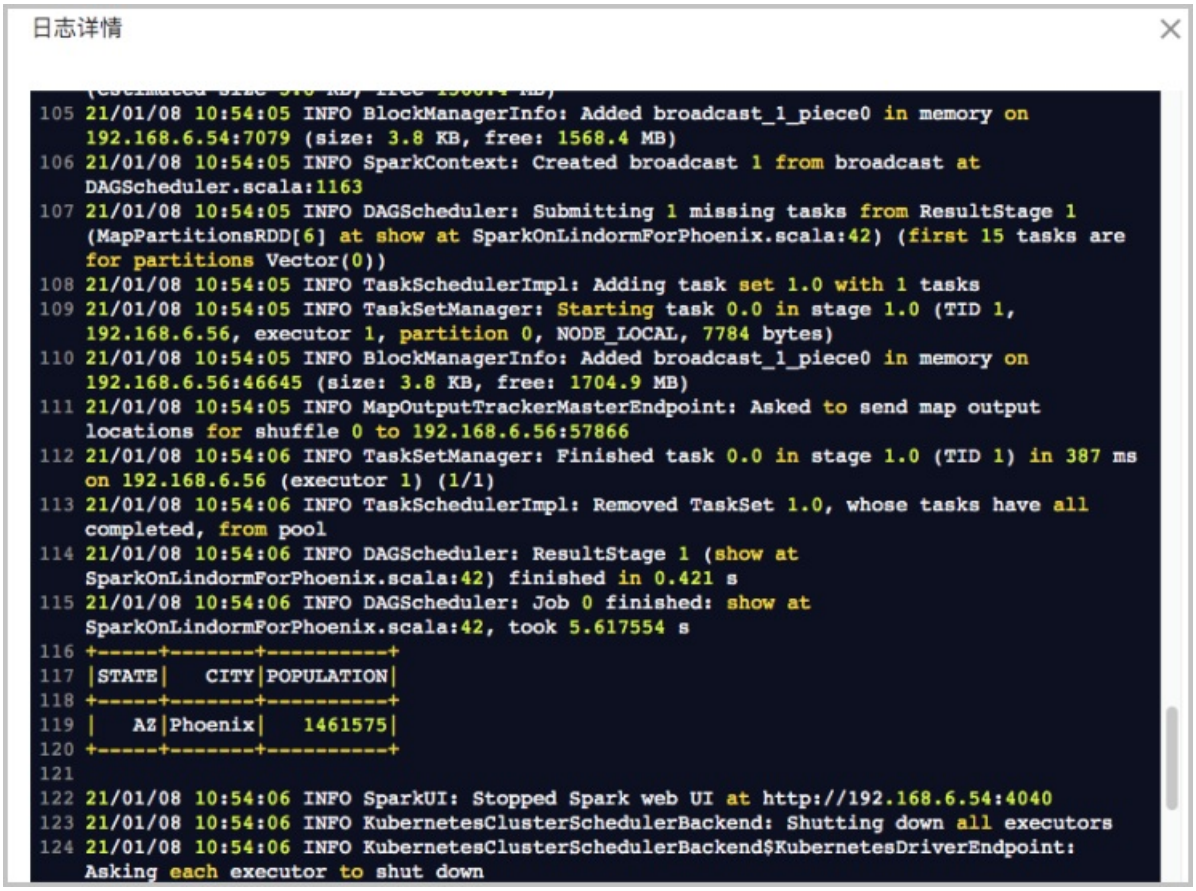


7. 单击Spark作业名，在Spark作业编辑框中输入以下作业内容，并按照以下参数说明进行参数值替换。保存并提交Spark作业。

```
{
    "args": [
        "gp-xxx-master.gpdbmaster.rds.aliyuncs.com:5432",  #AnalyticDB PostgreSQL内网地址和端口。
        "testdb",  #AnalyticDB PostgreSQL中的数据库名称。
        "public",  #AnalyticDB PostgreSQL中的schema名称。
        "test_table",  #AnalyticDB PostgreSQL中的表名。
        "xxx1",  #登录AnalyticDB PostgreSQL数据库的用户名。
        "xxx2",  #登录AnalyticDB PostgreSQL数据库的密码。
        "spark_on_adbpg_table"  #Spark中创建的映射AnalyticDB PostgreSQL表的表名。
    ],
    "file": "oss://spark_test/jars/adbpg/spark-examples-0.0.1-SNAPSHOT.jar",  #存放测试代码的OSS路径。
    "name": "adbpg-test",
    "jars": [
        "oss://spark_test/jars/adbpg/postgresql-42.2.5.jar"  #存放测试代码依赖包的OSS路径。
    ],
    "className": "com.aliyun.spark.SparkOnADBPostgreSQL",
    "conf": {
        "spark.driver.resourceSpec": "small",  #表示driver的规格，有small、medium、large、xlarge之分。
        "spark.executor.instances": 2,  #表示executor的个数。
        "spark.executor.resourceSpec": "small",  #表示executor的规格，有small、medium、large、xlarge之分。
        "spark.dla.eni.enable": "true",  #开启访问用户VPC网络的权限。当您需要访问用户VPC网络内的数据时，需要开启此选项。
        "spark.dla.eni.vswitch.id": "vsw-xxx",  #可访问的AnalyticDB PostgreSQL交换机id。
        "spark.dla.eni.security.group.id": "sg-xxx"  #可访问的AnalyticDB PostgreSQL安全组id。
    }
}
```

执行结果

作业运行成功后，在任务列表中单击**操作 > 日志**，查看作业日志。出现如下日志说明作业运行成功：



6.3. AnalyticDB MySQL

云原生数据仓库AnalyticDB MySQL版（简称ADB，原分析型数据库MySQL版），是阿里巴巴自主研发的海量数据实时高并发在线分析云计算服务，使得您可以在毫秒级针对千亿级数据进行即时的多维分析透视和业务探索。本文主要介绍如何通过DLA Serverless Spark访问云原生数据仓库AnalyticDB MySQL。

前提条件

- 已经开通对象存储OSS（Object Storage Service）服务。具体操作请参考开通OSS服务。
- 已经创建云原生数仓AnalyticDB MySQL集群。具体请参考创建集群。
- 在AnalyticDB MySQL集群中已创建数据库和表，并插入数据。参考命令样例如下：

```
#建库语句
create database testdb
#建表语句:
 CREATE TABLE testdb.test_table (
     name varchar(32) NULL,
     age INT  NULL,
     score DOUBLE NULL,
     primary key (name)
 )
  PARTITION by hash key(name) PARTITION NUM 10
#插入数据语句:
 INSERT INTO testdb.test_table VALUES('aliyun01', 1001, 10.1);
 INSERT INTO testdb.test_table VALUES('aliyun02', 1002, 10.2);
 INSERT INTO testdb.test_table VALUES('aliyun03', 1003, 10.3);
 INSERT INTO testdb.test_table VALUES('aliyun04', 1004, 10.4);
 INSERT INTO testdb.test_table VALUES('aliyun05', 1005, 10.5);
```

- 准备DLA Spark访问AnalyticDB MySQL集群所需的安全组ID和交换机ID。具体操作请参考配置数据源网络。
- DLA Spark访问AnalyticDB MySQL集群所需的交换机IP，已添加到AnalyticDB MySQL集群的白名单中。具体操作请参考设置白名单。

操作步骤

1. 准备以下测试代码和依赖包来访问AnalyticDB MySQL，并将此测试代码和依赖包分别编译打包生成jar包上传至您的OSS。

   测试代码示例：

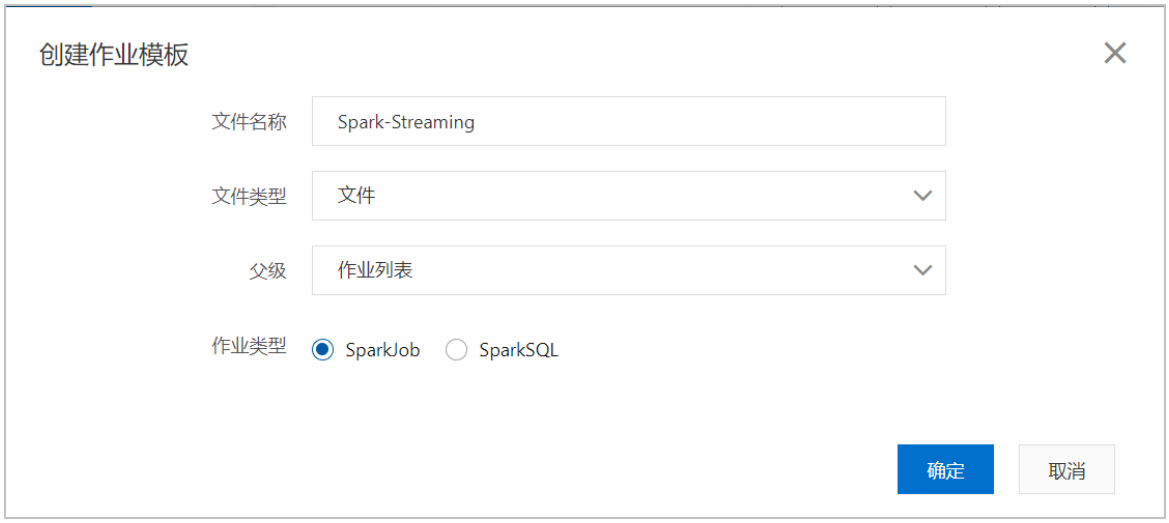
```
package com.aliyun.spark
import java.util.Properties
import org.apache.spark.sql.SparkSession
object SparkOnADBMySQL {
  def main(args: Array[String]): Unit = {
    val url = args(0)
    val database = args(1)
    val tableName = args(2)
    val user = args(3)
    val password = args(4)
    val jdbcConnURL = s"jdbc:mysql://$url/$database"
    //Spark侧的表名。
    var sparkTableName = args(5)
    val sparkSession = SparkSession
      .builder()
      .appName("scala spark on adb test")
      .getOrCreate()
    val driver = "com.mysql.cj.jdbc.Driver"
    //如果存在的话就删除表。
    sparkSession.sql(s"drop table if exists $sparkTableName")
    //Sql方式，Spark会映射数据中表的Schema。
    val createCmd =
      s"""CREATE TABLE ${sparkTableName} USING org.apache.spark.sql.jdbc
         |    options (
         |    driver '$driver',
         |    url '$jdbcConnURL',
         |    dbtable '$tableName',
```

```
          |    dbtable '$tableName',
          |    user '$user',
          |    password '$password'
          |    )""".stripMargin
    println(s"createCmd: \n $createCmd")
    sparkSession.sql(createCmd)
    val querySql = "select * from " + sparkTableName + " limit 1"
    sparkSession.sql(querySql).show
    //使用dataset API接口。
    val connectionProperties = new Properties()
    connectionProperties.put("driver", driver)
    connectionProperties.put("user", user)
    connectionProperties.put("password", password)
    //读取数据。
    var jdbcDf = sparkSession.read.jdbc(jdbcConnURL,
      s"$database.$tableName",
      connectionProperties)
    jdbcDf.select("name", "age", "score").show()
    val data =
      Seq(
        PersonADBMysql("bill", 30, 170.5D),
        PersonADBMysql("gate", 29, 200.3D)
      )
    val dfWrite = sparkSession.createDataFrame(data)
    //写入数据。
    dfWrite
      .write
      .mode("append")
      .jdbc(jdbcConnURL, s"$database.$tableName", connectionProperties)
    jdbcDf.select("name", "age").show()
    sparkSession.stop()
  }
}
case class PersonADBMysql(name: String, age: Int, score: Double)
```

AnalyticDB MySQL依赖的pom文件：

```
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.22</version>
        </dependency>
```

2. 登录Data Lake Analytics管理控制台。
3. 在页面左上角，选择AnalyticDB MySQL集群所在地域。
4. 单击左侧导航栏中的**Serverless Spark > 作业管理**。
5. 在**作业编辑**页面，单击**创建作业**。
6. 在**创建作业模板**页面，按照页面提示进行参数配置后，单击**确定**创建Spark作业。

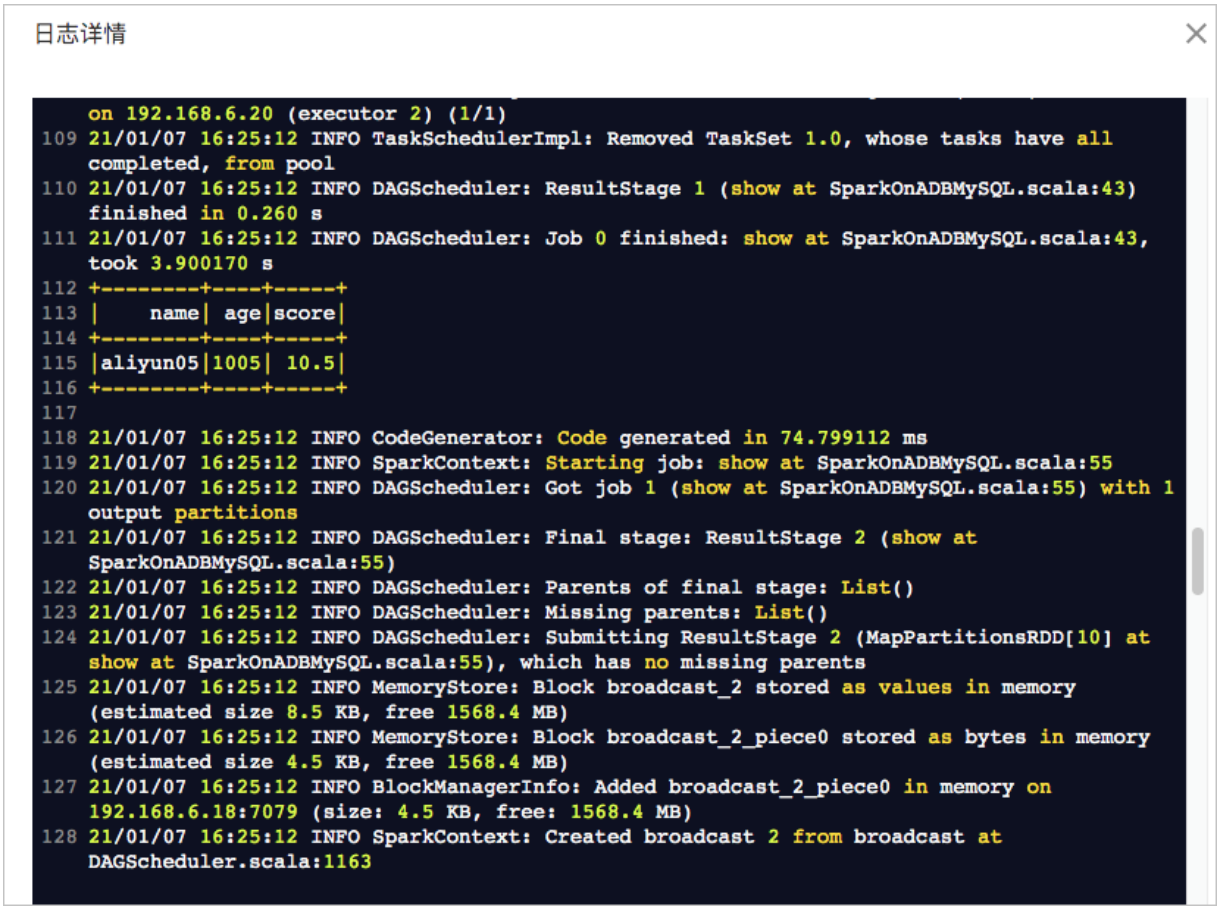


7. 单击Spark作业名，在Spark作业编辑框中输入以下作业内容，并按照以下参数说明进行参数值替换。保存并提交Spark作业。

```
{
    "args": [
        "am-xxx.ads.aliyuncs.com:3306",  #AnalyticDB MySQL的VPC地址和端口号。
        "testdb",  #AnalyticDB MySQL的数据库名称。
        "test_table",  #AnalyticDB MySQL的表名称。
        "xxx1",  #登录AnalyticDB MySQL数据库的用户名。
        "xxx2",  #登录AnalyticDB MySQL数据库的密码。
        "spark_on_adbmysql_table"  #Spark中创建映射AnalyticDB MySQL表的表名。
    ],
    "file": "oss://spark_test/jars/adbmysql/spark-examples-0.0.1-SNAPSHOT.jar",   #存放测试代码的OSS路径。
    "name": "adbmysql-test",
    "jars": [
        "oss://spark_test/jars/adbmysql/mysql-connector-java-8.0.22.jar"  #存放测试代码依赖包的OSS路径。
    ],
    "className": "com.aliyun.spark.SparkOnADBMySQL",
    "conf": {
        "spark.driver.resourceSpec": "small",  #表示driver的规格，有small、medium、large、xlarge之分。
        "spark.executor.instances": 2,  #表示executor的个数。
        "spark.executor.resourceSpec": "small", #表示executor的规格，有small、medium、large、xlarge之分。
        "spark.dla.eni.enable": "true",  #开启访问用户VPC网络的权限。当您需要访问用户VPC网络内的数据时，需要开启此选项。
        "spark.dla.eni.vswitch.id": "vsw-xxx",  #可访问AnalyticDB MySQL的交换机id。
        "spark.dla.eni.security.group.id": "sg-xxx"  #可访问AnalyticDB MySQL的安全组id。
    }
}
```

执行结果

作业运行成功后，在任务列表中单击**操作 > 日志**，查看作业日志。出现如下日志说明作业运行成功：



6.4. LogHub

本文介绍了如何使用DLA Spark Streaming访问LogHub。

前提条件

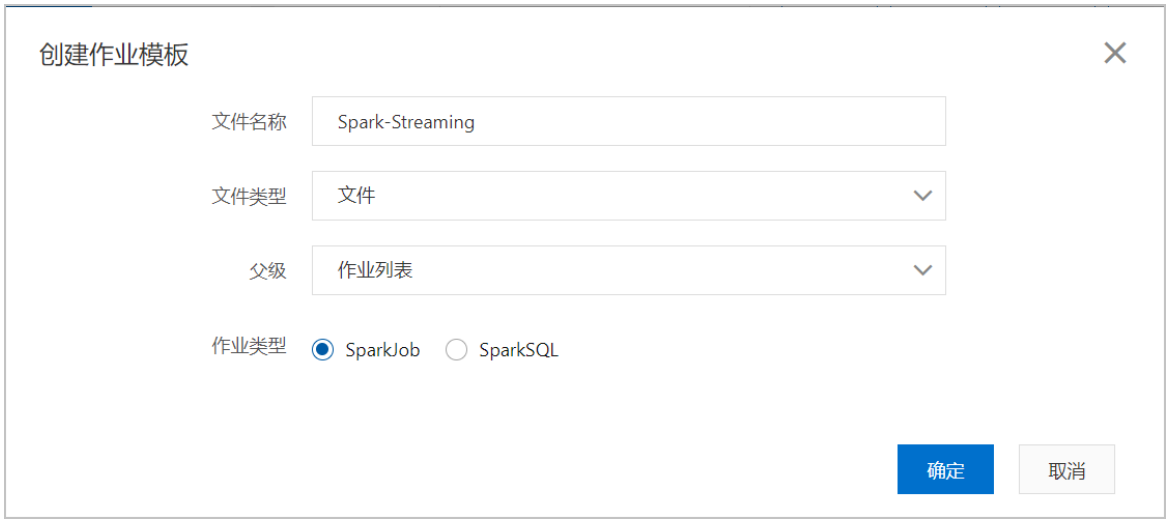
- 已经创建了Spark虚拟集群。具体操作请参见创建虚拟集群。
- 已经开通对象存储OSS（Object Storage Service）服务。具体操作请参见开通OSS服务。

操作步骤

1. 准备以下测试代码来连接LogHub，并将测试代码打包成jar包上传至您的OSS。

```
package com.aliyun.spark.streaming
import org.apache.spark.SparkConf
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.aliyun.logservice.LoghubUtils
import org.apache.spark.streaming.{Milliseconds, StreamingContext}
object SparkLogHub {
  def main(args: Array[String]): Unit = {
    if (args.length < 8) {
      System.err.println(
        """Usage: LoghubSample <sls project> <sls logstore> <loghub group name> <sls en
dpoint>
          |         <access key id> <access key secret> <batch interval seconds> <check
point dir>
        """.stripMargin)
      System.exit(1)
    }
    val loghubProject = args(0)
    val logStore = args(1)
    val loghubGroupName = args(2)
    val endpoint = args(3)
    val accessKeyId = args(4)
    val accessKeySecret = args(5)
    val batchInterval = Milliseconds(args(6).toInt * 1000)
    val checkPointDir = args(7)
    def functionToCreateContext(): StreamingContext = {
      val conf = new SparkConf().setAppName("LoghubSample")
      val ssc = new StreamingContext(conf, batchInterval)
      val loghubStream = LoghubUtils.createStream(
        ssc,
        loghubProject,
        logStore,
        loghubGroupName,
        endpoint,
        accessKeyId,
        accessKeySecret,
        StorageLevel.MEMORY_AND_DISK)
      loghubStream.checkpoint(batchInterval * 2).foreachRDD(rdd => println(rdd.count())
)
      ssc.checkpoint(checkPointDir) // set checkpoint directory
      ssc
    }
    val ssc = StreamingContext.getOrCreate(checkPointDir, functionToCreateContext _)
    ssc.start()
    ssc.awaitTermination()
  }
}
```

2. 登录Data Lake Analytics管理控制台。
3. 在页面左上角，选择LogHub所在地域。
4. 单击左侧导航栏中的**Serverless Spark > 作业管理**。
5. 在**作业编辑**页面，单击**创建作业**。
6. 在**创建作业模板**页面，按照页面提示进行参数配置后，单击**确定**创建Spark作业。



7. 单击Spark作业名，在Spark作业编辑框中输入以下Spark Streaming任务内容。保存并提交Spark作业。

```
{
    "args": [
        "<sls project>",  #sls的project名称。
        "<ls logstore>,   #sls的logstore名称。
        "<loghub group name>",  #sls的消费者组名称。
        "<sls endpoint>",   #sls的endpoint。
        "<access key id>",  #访问sls所需的accessKeyId。
        "<access key secret>",  #访问sls所需的accessKeySecret。
        "<batch interval seconds>", #Streaming的batch interval。
        "<checkpoint dir>" #checkpoint的oss路径。
    ],
    "name": "LogHub",
    "className": "com.aliyun.spark.streaming.SparkLogHub",
    "conf": {
        "spark.driver.resourceSpec": "medium",
        "spark.dla.connectors": "oss",
        "spark.executor.instances": 1,
        "spark.dla.job.log.oss.uri": "oss://</path/to/store/your/spark/log>",  #存放Spark日志的路径。
        "spark.executor.resourceSpec": "medium"
    },
    "file": "oss://path/to/spark-examples-0.0.1-SNAPSHOT-shaded.jar"
}
```

6.5. Lindorm文件引擎

本文介绍了如何使用DLA Spark访问Lindorm文件引擎。

前提条件

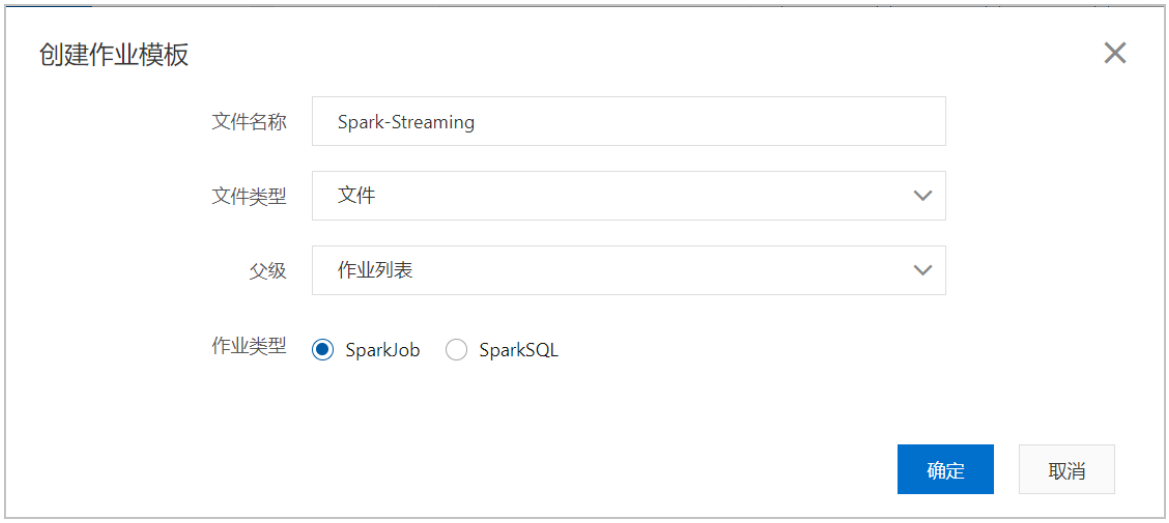
- 已经创建了Spark虚拟集群。具体操作请参见创建虚拟集群。
- 已经开通对象存储OSS（Object Storage Service）服务。具体操作请参见开通OSS服务。
- 前往Lindorm控制台，把要访问的Lindorm实例VPC网段加入到访问控制白名单中。具体操作请参见设置白名单。
- 准备DLA Spark访问Lindorm实例文件引擎所需的安全组ID和交换机ID。具体操作请参见配置数据源网络。

操作步骤

1. 准备以下测试代码来读写Lindorm文件引擎的HDFS，并将测试代码打包成*AccessLindormHDFS.py*文件上传至您的OSS。

```
from pyspark.sql import SparkSession
if __name__ == '__main__':
    def f(a):
        print(a)
    spark = SparkSession.builder.getOrCreate()
    welcome_str = "hello, dla-spark"
    #hdfs目录用于存放内容
    hdfsPath = sys.argv[1]
    #将welcome字符串存入指定的hdfs目录
    spark.sparkContext.parallelize(list(welcome_str)).saveAsTextFile(hdfsPath)
    #从指定的hdfs目录中读取内容，并打印
    print("------------------------------------------------------------")
    res = spark.sparkContext.textFile(hdfsPath).collect()
    [f(e) for e in res]
    print("------------------------------------------------------------")
```

2. 登录Lindorm控制台，定位到Lindorm实例文件引擎，一键生成配置项。具体操作请参见开通指南。
3. 登录Data Lake Analytics管理控制台。
4. 在页面左上角，选择Lindorm实例文件引擎所在地域。
5. 单击左侧导航栏中的**Serverless Spark > 作业管理**。
6. 在**作业编辑**页面，单击**创建作业模板**。
7. 在**创建作业模板**页面，按照页面提示进行参数配置后，单击**确定**创建Spark作业。



8. 单击Spark作业名，在Spark作业编辑框中输入以下作业内容，并按照以下参数说明进行参数值替换。保存并提交Spark作业。

```
{
    "name": "Lindorm",
    "args": [
        "<fs.defaultFS>/tmp/test-lindorm.txt"
    ],
    "conf": {
        "spark.driver.resourceSpec": "medium",
        "spark.executor.resourceSpec": "medium",
        "spark.executor.instances": 1,
        "spark.kubernetes.pyspark.pythonVersion": "3",
        "spark.dla.job.log.oss.uri": "oss://<存放您UI日志的OSS路径>",
        "spark.dla.eni.enable": "true",
        "spark.dla.eni.security.group.id": "<您的安全组ID>",
        "spark.dla.eni.vswitch.id": "<您的交换机ID>",
        "spark.hadoop.dfs.nameservices": "<dfs.nameservices>",
        "spark.hadoop.dfs.client.failover.proxy.provider.<dfs.nameservices>": "org.apac
he.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider",
        "spark.hadoop.dfs.ha.namenodes.<dfs.nameservices>": "nn1,nn2",
        "spark.hadoop.dfs.namenode.rpc-address.<dfs.nameservices>.nn1": "<dfs.namenode.
rpc-address.<dfs.nameservices>.nn1>",
        "spark.hadoop.dfs.namenode.rpc-address.<dfs.nameservices>.nn2": "<dfs.namenode.
rpc-address.<dfs.nameservices>.nn2>"
    },
    "file": "oss://path/to/AccessLindormHDFS.py"
}
```

参数说明

参数名称	参考值	参数说明
args	`args: <fs.defaultsFS>` 的取值来源于步骤2中一键生成的*core-site*配置项中的 `fs.defaultsFS` 的值。具体请参见开通指南。	无。
spark.driver.resourceSpec	medium	表示Driver的规格，取值： ○ small：1c4g ○ medium：2c8g ○ large：4c16g ○ xlarge：8c32g
spark.executor.resourceSpec	medium	表示Executor的规格，取值： ○ small：1c4g ○ medium：2c8g ○ large：4c16g ○ xlarge：8c32g
spark.executor.instances	1	表示Executor的个数。

参数名称	参考值	参数说明
spark.kubernetes.pyspark.pythonVersion	3	表示Python的运行版本，取值： ○ 2：Python2 ○ 3：Python3
spark.dla.job.log.oss.uri	oss://<存放您UI日志的OSS路径>	存放Spark日志的路径，只能配置OSS路径。
spark.dla.eni.enable	true	开启访问用户VPC网络的权限。当您需要访问用户VPC网络内的数据时，需要开启此选项。
spark.dla.eni.security.group.id	<您的安全组ID>	访问用户VPC网络所需要的安全组ID。
spark.dla.eni.vswitch.id	<您的交换机ID>	访问用户VPC网络所需要的交换机ID。
spark.hadoop.dfs.nameservices	取值来源于步骤2中一键生成的*hdfs-site*配置项中的`dfs.nameservices`的值。	连接Hadoop所需配置项。
spark.hadoop.dfs.client.failover.proxy.provider.<dfs.nameservices>	取值来源于步骤2中一键生成的*hdfs-site*配置项中的`dfs.client.failover.proxy.provider.<dfs.nameservices>`的值。	连接Hadoop所需配置项。
spark.hadoop.dfs.ha.namenodes.<dfs.nameservices>	取值来源于步骤2中一键生成的*hdfs-site*配置项中的`dfs.ha.namenodes.<dfs.nameservices>`的值。	连接Hadoop所需配置项。
spark.hadoop.dfs.namenode.rpc-address.<dfs.nameservices>.nn1	取值来源于步骤2中一键生成的*hdfs-site*配置项中的`dfs.namenode.rpc-address.<dfs.nameservices>.nn1`的值。	连接Hadoop所需配置项。
spark.hadoop.dfs.namenode.rpc-address.<dfs.nameservices>.nn2	取值来源于步骤2中一键生成的*hdfs-site*配置项中的`dfs.namenode.rpc-address.<dfs.nameservices>.nn2`的值。	连接Hadoop所需配置项。

常见问题

问题描述：在使用DLA Spark访问Lindorm文件引擎时，遇到如下错误。

```
py4j.protocol.Py4JJavaError: An error occurred while calling o181.saveAsTextFile.
: org.apache.hadoop.security.AccessControlException: Permission denied: user=spark,
access=WRITE, inode="/spark-test/test1/_temporary/0":root:hadoop:drwxr-xr-x
    at
org.apache.hadoop.hdfs.server.namenode.FSPermissionChecker.check(FSPermissionChecker.jav
a:319)
    at
org.apache.hadoop.hdfs.server.namenode.FSPermissionChecker.check(FSPermissionChecker.jav
a:292)
```

解决办法：您可以在Spark作业的配置项中增加如下参数，改变当前作业的运行用户为有权限的用户。

```
"spark.driver.extraJavaOptions": "-DHADOOP_USER_NAME=<用户名>",
"spark.executor.extraJavaOptions": "-DHADOOP_USER_NAME=<用户名>"
```

6.6. Lindorm宽表SQL

Lindorm宽表引擎是面向海量半结构化、结构化数据设计的分布式存储，适用于元数据、订单、账单、画像、社交、feed流、日志等场景，兼容HBase、Phoenix(SQL)。本文主要介绍如何通过DLA Serverless Spark访问Lindorm的宽表SQL（Phoenix）。

前提条件

- 已开通宽表SQL服务。具体操作请参见开通宽表SQL。
- 已经开通对象存储OSS（Object Storage Service）服务。具体操作请参见开通OSS服务。
- 准备DLA Spark访问Lindorm实例宽表引擎所需的安全组ID和交换机ID。具体操作请参见配置数据源网络。
- 前往Lindorm控制台，把要访问的Lindorm实例VPC网段加入到访问控制白名单中。具体操作请参见设置白名单。
- 在宽表SQL服务中已创建表并插入数据。假设本文档创建的表名为**us_population**。参考命令样例如下：

```
#建表语句：
CREATE TABLE IF NOT EXISTS us_population (
state CHAR(2) NOT NULL,
city VARCHAR NOT NULL,
population BIGINT
CONSTRAINT my_pk PRIMARY KEY (state, city));
#插入数据语句：
UPSERT INTO us_population VALUES('NY','New York',8143197);
UPSERT INTO us_population VALUES('CA','Los Angeles',3844829);
UPSERT INTO us_population VALUES('IL','Chicago',2842518);
UPSERT INTO us_population VALUES('TX','Houston',2016582);
UPSERT INTO us_population VALUES('PA','Philadelphia',1463281);
UPSERT INTO us_population VALUES('AZ','Phoenix',1461575);
UPSERT INTO us_population VALUES('TX','San Antonio',1256509);
UPSERT INTO us_population VALUES('CA','San Diego',1255540);
UPSERT INTO us_population VALUES('TX','Dallas',1213825);
UPSERT INTO us_population VALUES('CA','San Jose',912332);
```

操作步骤

1. 准备以下测试代码和依赖包来访问宽表SQL，并将此测试代码和依赖包分别编译打包生成jar包上传至您的OSS。

   测试代码示例：

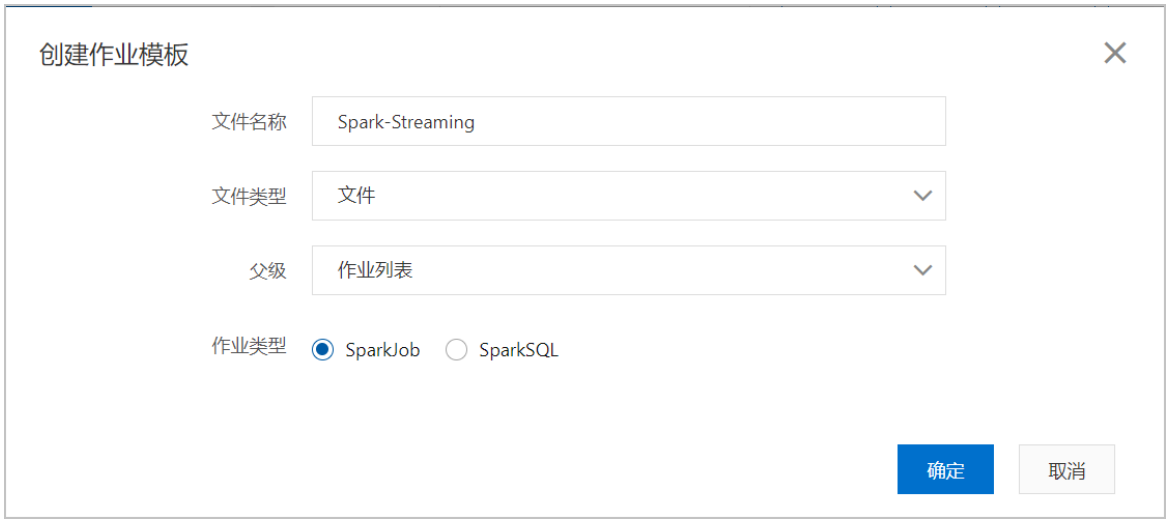
```
package com.aliyun.spark
import org.apache.spark.sql.SparkSession
object SparkOnLindormForPhoenix {
  def main(args: Array[String]): Unit = {
    //queryServerAddress为Lindorm集群宽表SQL服务访问地址，格式为：http://xxx:8765。
    val queryServerAddress = args(0)
    //宽表引擎的用户名和密码。
    val user = args(1)
    val password = args(2)
    //Phoenix侧的表名，需要在Phoenix侧提前创建。
    val phoenixTableName = args(3)
    //Spark侧的表名。
    val sparkTableName = args(4)
    val sparkSession = SparkSession
      .builder()
      .appName("scala spark on Phoenix5.x test")
      .getOrCreate()
    //如果存在的话就删除表。
    sparkSession.sql(s"drop table if exists $sparkTableName")
    val driver = "org.apache.phoenix.queryserver.client.Driver"
    val url = "jdbc:phoenix:thin:url=" + queryServerAddress + ";serialization=PROTOBUF"
    val createCmd = "CREATE TABLE " +
      sparkTableName +
      " USING org.apache.spark.sql.jdbc\n" +
      "OPTIONS (\n" +
      "  'driver' '" + driver + "',\n" +
      "  'url' '" + url + "',\n" +
      "  'user' '" + user + "',\n" +
      "  'password' '" + password + "',\n" +
      "  'dbtable' '" + phoenixTableName + "',\n" +
      "  'fetchsize' '" + 100 + "'\n" +
      ")"
    println(" createCmd: \n" + createCmd)
    sparkSession.sql(createCmd)
    val querySql = "select * from " + sparkTableName + " limit 1"
    sparkSession.sql(querySql).show
    sparkSession.stop()
  }
}
```

   宽表SQL依赖的pom文件：

```
        <dependency>
            <groupId>com.aliyun.phoenix</groupId>
            <artifactId>ali-phoenix-shaded-thin-client</artifactId>
            <version>5.2.2-HBase-2.x-SNAPSHOT</version>
        </dependency>
```

2. 登录Data Lake Analytics管理控制台。

3. 在页面左上角，选择Lindorm宽表SQL所在实例的相同地域。
4. 单击左侧导航栏中的**Serverless Spark > 作业管理**。
5. 在**作业编辑**页面，单击**创建作业**。
6. 在**创建作业模板**页面，按照页面提示进行参数配置后，单击**确定**创建Spark作业。

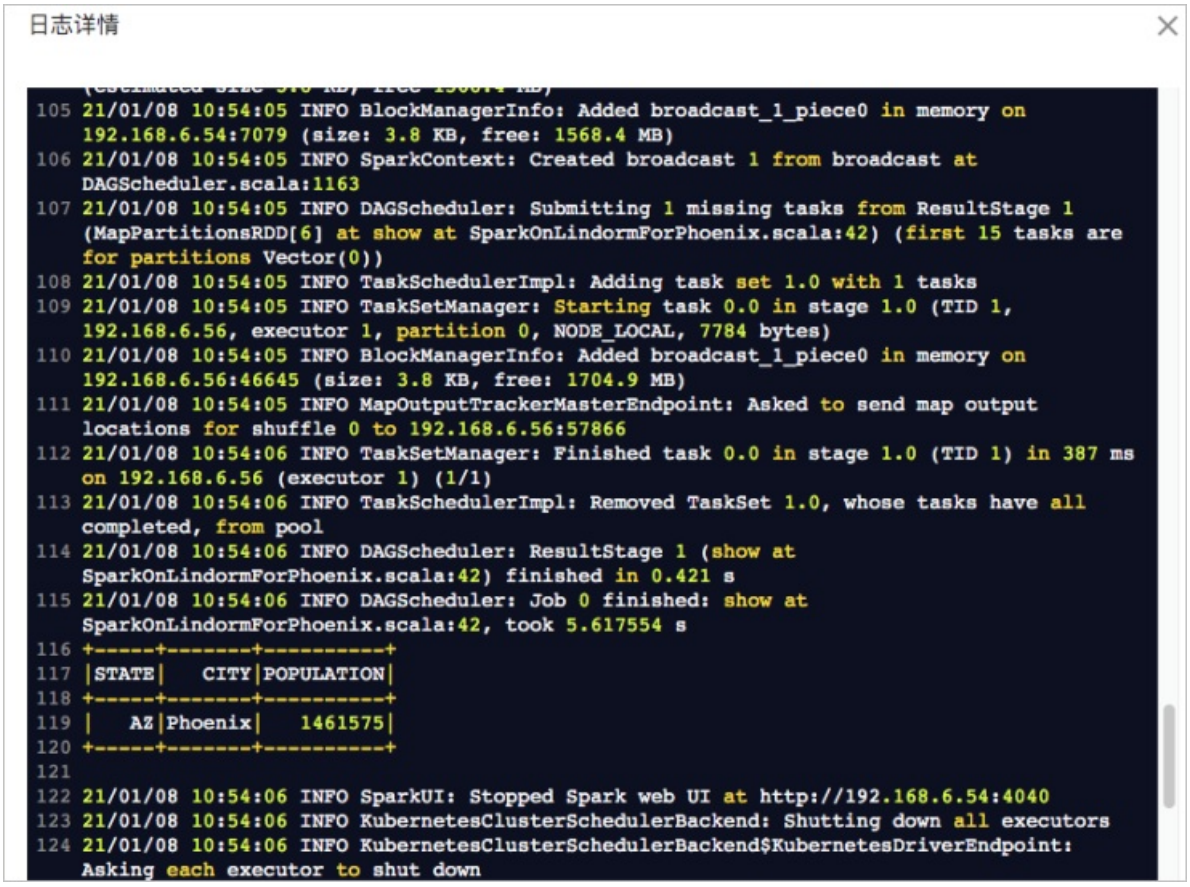


7. 单击Spark作业名，在Spark作业编辑框中输入以下作业内容，并按照以下参数说明进行参数值替换。保存并提交Spark作业。

```
{
    "args": [
        "http://ld-xxx-proxy-phoenix.lindorm.rds.aliyuncs.com:8765",  #宽表SQL的"客户端访问地址"-> "私网"。注意地址要以："http://"开头。
        "xxx1",  #访问Lindorm的用户名。
        "xxx2",  #访问Lindorm的密码。
        "us_population",  #Phoenix的表名。
        "spark_on_lindorm_phoenix"  #Spark中创建映射Phoenix表的表名。
    ],
    "file": "oss://spark_test/jars/lindorm/spark-examples-0.0.1-SNAPSHOT.jar",  ##测试代码的OSS路径。
    "name": "lindorm-for-phoenix-test",
    "jars": [
        "oss://spark_test/jars/lindorm/ali-phoenix-shaded-thin-client-5.2.2-HBase-2.x-SNAPSHOT.jar"  ##测试代码依赖包的OSS路径。
    ],
    "className": "com.aliyun.spark.SparkOnLindormForPhoenix",
    "conf": {
        "spark.driver.resourceSpec": "small",  #表示driver的规格，有small、medium、large、xlarge之分。
        "spark.executor.instances": 2,  #表示executor的个数。
        "spark.executor.resourceSpec": "small",  ##表示executor的规格，有small、medium、large、xlarge之分。
        "spark.dla.eni.enable": "true",  #开启访问用户VPC网络的权限。当您需要访问用户VPC网络内的数据时，需要开启此选项。
        "spark.dla.eni.vswitch.id": "vsw-xxx",  #可访问Lindorm的交换机id。
        "spark.dla.eni.security.group.id": "sg-xxx"  #可访问Lindorm的安全组id。
    }
}
```

执行结果

作业运行成功后，在任务列表中单击**操作 > 日志**，查看作业日志。出现如下日志说明作业运行成功：



6.7. HBase标准版2.0版本Phoenix服务

云数据库HBase是面向大数据领域的一站式NoSQL服务，适用于GB至PB级的大规模吞吐、检索、分析工作负载，是为淘宝推荐、支付宝账单、花呗风控等众多阿里巴巴核心服务提供支撑的数据库。本文主要介绍如何通过DLA Serverless Spark 对接云数据库HBase标准版2.0版本的HBase SQL服务Phoenix。

前提条件

- 已开通HBase SQL服务。具体操作请参见HBase SQL(Phoenix) 5.x 使用说明。
- 已经开通对象存储OSS（Object Storage Service）服务。具体操作请参见开通OSS服务。
- 准备DLA Spark访问HBase集群SQL服务所需的安全组ID和交换机ID。具体操作请参见配置数据源网络。
- 前往HBase管理控制台，把要访问的HBase集群VPC网段加入到访问控制白名单中。具体操作请参见设置白名单和安全组。
- 在HBase SQL服务中已创建表并插入数据。假设本文档创建的表名为us_population。参考命令样例如下：

```
#建表语句：
CREATE TABLE IF NOT EXISTS us_population (
state CHAR(2) NOT NULL,
city VARCHAR NOT NULL,
population BIGINT
CONSTRAINT my_pk PRIMARY KEY (state, city));
#插入数据语句：
UPSERT INTO us_population VALUES('NY','New York',8143197);
UPSERT INTO us_population VALUES('CA','Los Angeles',3844829);
UPSERT INTO us_population VALUES('IL','Chicago',2842518);
UPSERT INTO us_population VALUES('TX','Houston',2016582);
UPSERT INTO us_population VALUES('PA','Philadelphia',1463281);
UPSERT INTO us_population VALUES('AZ','Phoenix',1461575);
UPSERT INTO us_population VALUES('TX','San Antonio',1256509);
UPSERT INTO us_population VALUES('CA','San Diego',1255540);
UPSERT INTO us_population VALUES('TX','Dallas',1213825);
UPSERT INTO us_population VALUES('CA','San Jose',912332);
```

操作步骤

1. 准备以下测试代码和依赖包来访问HBase SQL，并将此测试代码和依赖包分别编译打包生成jar包上传至您的OSS。

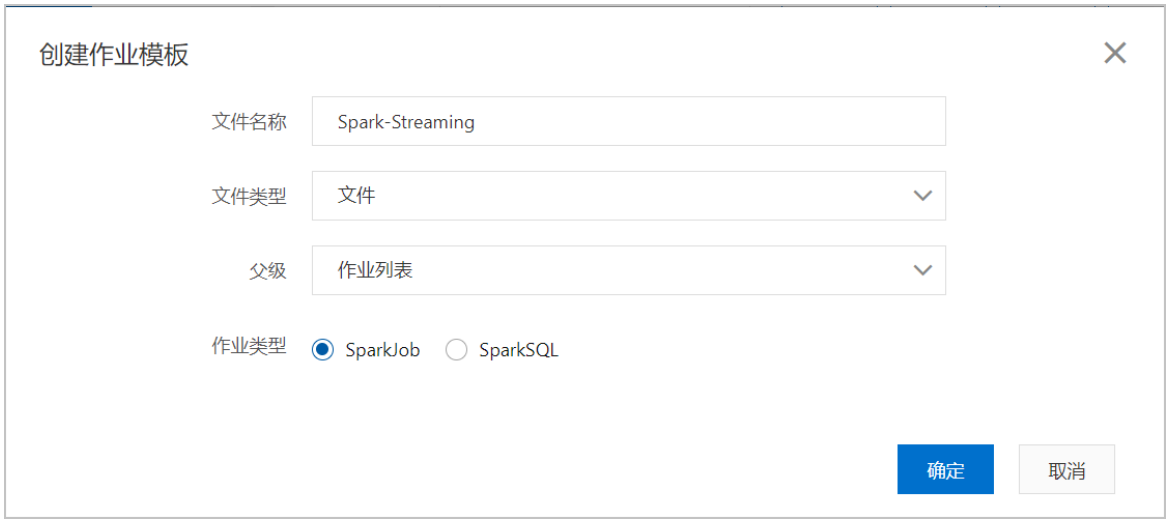
   测试代码示例：

```
package com.aliyun.spark
import org.apache.spark.sql.SparkSession
object SparkOnHBase2xForPhoenix {
  def main(args: Array[String]): Unit = {
    //queryServerAddress为HBase集群SQL服务访问地址，格式为：http://xxx:8765
    val queryServerAddress = args(0)
    //Phoenix侧的表名，需要在Phoenix侧提前创建。
    val phoenixTableName = args(1)
    //Spark侧的表名。
    val sparkTableName = args(2)
    val sparkSession = SparkSession
      .builder()
      .appName("scala spark on Phoenix5.x test")
      .getOrCreate()
    //如果存在的话就删除表
    sparkSession.sql(s"drop table if exists $sparkTableName")
    val driver = "org.apache.phoenix.queryserver.client.Driver"
    val url = "jdbc:phoenix:thin:url=" + queryServerAddress + ";serialization=PROTOBUF"
    val createCmd = "CREATE TABLE " +
      sparkTableName +
      " USING org.apache.spark.sql.jdbc\n" +
      "OPTIONS (\n" +
      "  'driver' '" + driver + "',\n" +
      "  'url' '" + url + "',\n" +
      "  'dbtable' '" + phoenixTableName + "',\n" +
      "  'fetchsize' '" + 100 + "'\n" +
      ")"
    println(" createCmd: \n" + createCmd)
    sparkSession.sql(createCmd)
    val querySql = "select * from " + sparkTableName + " limit 1"
    sparkSession.sql(querySql).show
    sparkSession.stop()
  }
}
```

HBase SQL依赖的pom文件：

```
        <dependency>
            <groupId>com.aliyun.phoenix</groupId>
            <artifactId>ali-phoenix-shaded-thin-client</artifactId>
            <version>5.2.2-HBase-2.x-SNAPSHOT</version>
        </dependency>
```

2. 登录Data Lake Analytics管理控制台。
3. 在页面左上角，选择HBase集群HBase SQL服务所在的地域。
4. 单击左侧导航栏中的**Serverless Spark > 作业管理**。
5. 在**作业编辑**页面，单击**创建作业**。
6. 在**创建作业模板**页面，按照页面提示进行参数配置后，单击**确定**创建Spark作业。

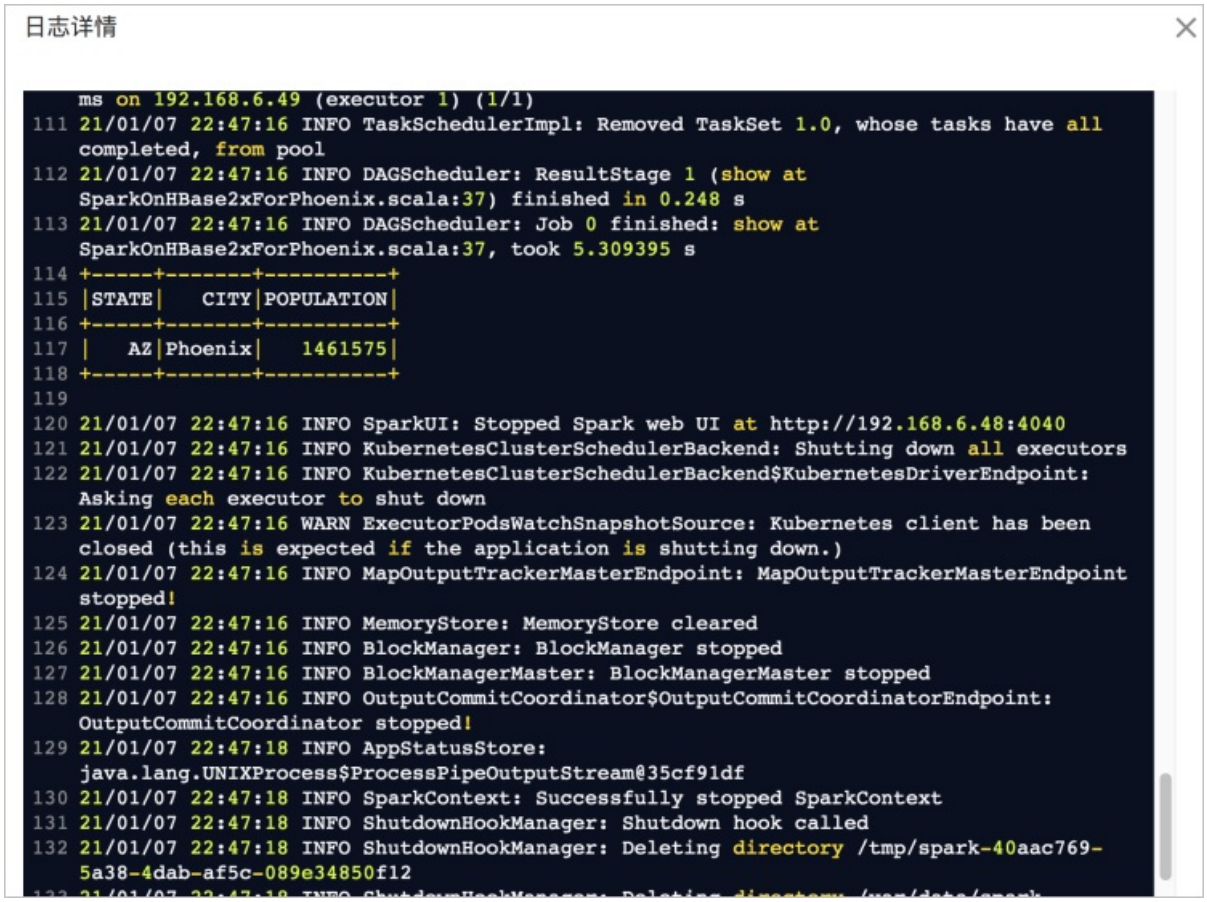


7. 单击Spark作业名，在Spark作业编辑框中输入以下作业内容，并按照以下参数说明进行参数值替换。保存并提交Spark作业。

```
{
    "args": [
        "http://hb-xxx-proxy-phoenix.hbase.rds.aliyuncs.com:8765",  #HBase SQL服务的客户端访问地址，可以是“负载均衡连接”或者“单点 QueryServer连”。
        "us_population",  #Phoenix的表名。
        "spark_on_hbase2x_phoenix"  #Spark中创建映射Phoenix表的表名。
    ],
    "file": "oss://spark_test/jars/hbase2x/spark-examples-0.0.1-SNAPSHOT.jar",  #测试代码的OSS路径。
    "name": "hbase2x-for-phoenix-test",
    "jars": [
        "oss://spark_test/jars/hbase2x/ali-phoenix-shaded-thin-client-5.2.2-HBase-2.x-SNAPSHOT.jar"  #测试代码依赖包的OSS路径。
    ],
    "className": "com.aliyun.spark.SparkOnHBase2xForPhoenix",
    "conf": {
        "spark.driver.resourceSpec": "small",  #表示driver的规格，有small、medium、large、xlarge之分。
        "spark.executor.instances": 2,  #表示executor的个数。
        "spark.executor.resourceSpec": "small",  #表示executor的规格，有small、medium、large、xlarge之分。
        "spark.dla.eni.enable": "true",  #开启访问用户VPC网络的权限。当您需要访问用户VPC网络内的数据时，需要开启此选项。
        "spark.dla.eni.vswitch.id": "vsw-xxx",  #可访问HBase 的交换机id。
        "spark.dla.eni.security.group.id": "sg-xxx"  #可访问HBase的安全组id。
    }
}
```

执行结果

作业运行成功后，在任务列表中单击**操作 > 日志**，查看作业日志。出现如下日志说明作业运行成功：



6.8. DataHub

阿里云流式数据服务DataHub流式数据（Streaming Data）的处理平台，提供对流式数据的发布（Publish），订阅（Subscribe）和分发功能，让您可以轻松构建基于流式数据的分析和应用。本文主要介绍如何通过DLA Serverless Spark访问DataHub。

前提条件

- 已经在DataHub中创建项目。本文档中假设DataHub的区域为**华南1（深圳）**，Project名称为**spark_test**，Topic名称为**topic01**。具体请参考Project操作和Topic操作。

  > **说明** 目前内置的SparkOnDataHub Connectors仅支持**TUPLE**类型的Topic。

- 已经开通对象存储OSS（Object Storage Service）服务。具体操作请参见开通OSS服务。

背景信息

为了Spark能正常消费到DataHub数据，您需要将本地准备的模拟测试数据发送到DataHub，来测试Spark和DataHub的连通性。本文档假设您下载以下模拟测试代码到本地，并执行以下命令运行jar包来发送数据到**spark_test**下的**topic01**。

```
//下载模拟测试代码到本地。
wget https://spark-home.oss-cn-shanghai.aliyuncs.com/common_test/common-test-0.0.1-SNAPSHOT
-shaded.jar
//运行jar包来发送数据到spark_test下的topic01。
java -cp /opt/jars/common-test-0.0.1-SNAPSHOT-shaded.jar com.aliyun.datahub.DatahubWrite_ja
va spark_test topic01 xxx1 xxx2 https://dh-cn-shenzhen.aliyuncs.com
```

命令参数说明：

参数名称	参数说明
spark_test	DataHub的project名称。
topic01	DataHub的topic名称。
xxx1	访问阿里云API的AccessKey ID。
xxx2	访问阿里云API的AccessKey Secret。
https://dh-cn-shenzhen.aliyuncs.com	DataHub访问域名中“华南1（深圳）”的“外网Endpoint”。

操作步骤

1. 准备以下测试代码和依赖包来访问DataHub，并将此测试代码和依赖包分别编译打包生成jar包上传至您的OSS。

   测试代码示例：

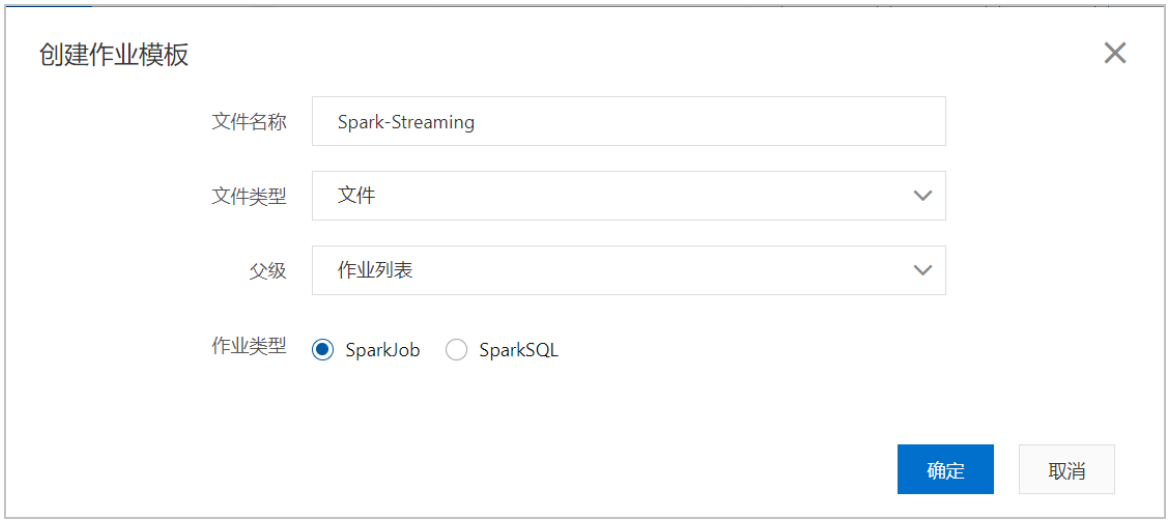
```
package com.aliyun.spark
import com.aliyun.datahub.model.RecordEntry
import org.apache.spark.SparkConf
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.aliyun.datahub.DatahubUtils
import org.apache.spark.streaming.{Milliseconds, StreamingContext}
import org.apache.spark.streaming.dstream.DStream
object SparkStreamingOnDataHub {
  def main(args: Array[String]): Unit = {
    val endpoint = args(0)
    //RAM访问控制中的AccessKeyID。
    val accessKeyId = args(1)
    //RAM访问控制中的AccessKeySecret。
    val accessKeySecret = args(2)
    //dataHub的订阅ID。
    val subId = args(3)
    //dataHub的project名称。
    val project = args(4)
    //dataHub的topic名称。
    val topic = args(5)
    val batchInterval = Milliseconds(10 * 1000)
    var checkpoint = "/tmp/SparkOnDatahubReliable_T001/"
    if (args.length >= 7) {
      checkpoint = args(6)
    }
```

```
    var shardId = "0"
    if (args.length >= 8) {
      shardId = args(7).trim
    }
    println(s"=====project=${project}===topic=${topic}===batchInterval=${batchInterval.
milliseconds / 1000}=====")
    def functionToCreateContext(): StreamingContext = {
      val conf = new SparkConf().setAppName("Test DataHub")
      //设置使用Reliable DataReceiver。
      conf.set("spark.streaming.receiver.writeAheadLog.enable", "true")
      val ssc = new StreamingContext(conf, batchInterval)
      ssc.checkpoint(checkpoint)
      var datahubStream: DStream[String] = null
      if (!shardId.isEmpty) {
        datahubStream = DatahubUtils.createStream(
          ssc,
          project,
          topic,
          subId,
          accessKeyId,
          accessKeySecret,
          endpoint,
          shardId,
          read,
          StorageLevel.MEMORY_AND_DISK_SER_2)
      } else {
        datahubStream = DatahubUtils.createStream(
          ssc,
          project,
          topic,
          subId,
          accessKeyId,
          accessKeySecret,
          endpoint,
          read,
          StorageLevel.MEMORY_AND_DISK_SER_2)
      }
      datahubStream.foreachRDD { rdd =>
        //注意，测试环境小数据量使用了rdd.collect()。真实环境请慎用。
        rdd.collect().foreach(println)
        //        rdd.foreach(println)
      }
      ssc
    }
    val ssc = StreamingContext.getActiveOrCreate(checkpoint, functionToCreateContext)
    ssc.start()
    ssc.awaitTermination()
  }
  def read(record: RecordEntry): String = {
    s"${record.getString(0)},${record.getString(1)}"
  }
}
```

DataHub依赖的pom文件：

```
<dependency>
    <groupId>com.aliyun.apsaradb</groupId>
    <artifactId>datahub-spark</artifactId>
    <version>2.9.2-public_2.4.3-1.0.4</version>
</dependency>
<dependency>
    <groupId>com.aliyun.datahub</groupId>
    <artifactId>aliyun-sdk-datahub</artifactId>
    <version>2.9.2-public</version>
</dependency>
```

2. 登录Data Lake Analytics管理控制台。
3. 在页面左上角，选择DataHub所在的地域。
4. 单击左侧导航栏中的**Serverless Spark > 作业管理**。
5. 在**作业编辑**页面，单击**创建作业**。
6. 在**创建作业模板**页面，按照页面提示进行参数配置后，单击**确定**创建Spark作业。



7. 单击Spark作业名，在Spark作业编辑框中输入以下作业内容，并按照以下参数说明进行参数值替换。保存并提交Spark作业。

```
{
    "args": [
        "http://dh-cn-shenzhen-int-vpc.aliyuncs.com",  #DataHub访问域名中"华南1(深圳)"的"外网Endpoint"。
        "xxx1",  #访问阿里云API的AccessKey ID。
        "xxx2",  #访问阿里云API的AccessKey Secret。
        "xxx3",  #DataHub中topic01的订阅ID。
        "spark_test",  #DataHub的project名称。
        "topic01"  #DataHub的topic名称。
    ],
    "file": "oss://spark_test/jars/datahub/spark-examples-0.0.1-SNAPSHOT.jar",  #测试代码的OSS路径。
    "name": "datahub-test",
    "jars": [
        //#测试代码依赖包的OSS路径。
        "oss://spark_test/jars/datahub/aliyun-sdk-datahub-2.9.2-public.jar",
        "oss://spark_test/jars/datahub/datahub-spark-2.9.2-public_2.4.3-1.0.4.jar"
    ],
    "className": "com.aliyun.spark.SparkStreamingOnDataHub",
    "conf": {
        "spark.driver.resourceSpec": "small",  #表示driver的规格，有small、medium、large、xlarge之分。
        "spark.executor.instances": 2,  #表示executor的个数。
        "spark.executor.resourceSpec": "small"  #表示executor的规格，有small、medium、large、xlarge之分。
    }
}
```

执行结果

作业运行成功后，在任务列表中单击**操作 > 日志**，查看作业日志。出现如下日志说明作业运行成功：

日志详情

```
172.18.87.40:41815 (size: 25.3 KB, free: 1704.9 MB)
49 21/01/07 17:41:26 INFO TaskSetManager: Finished task 0.0 in stage 3.0 (TID 71) in 150 ms
on 172.18.87.40 (executor 2) (1/1)
50 21/01/07 17:41:26 INFO TaskSchedulerImpl: Removed TaskSet 3.0, whose tasks have all
completed, from pool
51 21/01/07 17:41:26 INFO DAGScheduler: ResultStage 3 (collect at
SparkStreamingOnDataHub.scala:72) finished in 6.655 s
52 21/01/07 17:41:26 INFO DAGScheduler: Job 2 finished: collect at
SparkStreamingOnDataHub.scala:72, took 6.664689 s
53 name_00001,value_00001
54 name_00002,value_00002
55 name_00003,value_00003
56 name_00004,value_00004
57 21/01/07 17:41:26 INFO JobScheduler: Finished job streaming job 1610012480000 ms.0 from
job set of time 1610012480000 ms
58 21/01/07 17:41:26 INFO JobScheduler: Total delay: 6.895 s for time 1610012480000 ms
(execution: 6.835 s)
59 21/01/07 17:41:26 INFO JobGenerator: Checkpointing graph for time 1610012480000 ms
60 21/01/07 17:41:26 INFO DStreamGraph: Updating checkpoint data for time 1610012480000 ms
61 21/01/07 17:41:26 INFO DStreamGraph: Updated checkpoint data for time 1610012480000 ms
62 21/01/07 17:41:26 INFO CheckpointWriter: Submitted checkpoint of time 1610012480000 ms
to writer queue
63 21/01/07 17:41:26 INFO CheckpointWriter: Saving checkpoint for time 1610012480000 ms to
file 'file:/tmp/SparkOnDatahubReliable_T001/checkpoint-1610012480000'
64 21/01/07 17:41:26 INFO CheckpointWriter: Checkpoint for time 1610012480000 ms saved to
file 'file:/tmp/SparkOnDatahubReliable_T001/checkpoint-1610012480000', took 8279 bytes
and 3 ms
65 21/01/07 17:41:26 INFO DStreamGraph: Clearing checkpoint data for time 1610012480000 ms
66 21/01/07 17:41:26 INFO DStreamGraph: Cleared checkpoint data for time 1610012480000 ms
67 21/01/07 17:41:26 INFO ReceivedBlockTracker: Deleting batches:
68 21/01/07 17:41:26 INFO FileBasedWriteAheadLog_ReceivedBlockTracker: Attempting to clear
0 old log files in file:/tmp/SparkOnDatahubReliable_T001/receivedBlockMetadata older
```

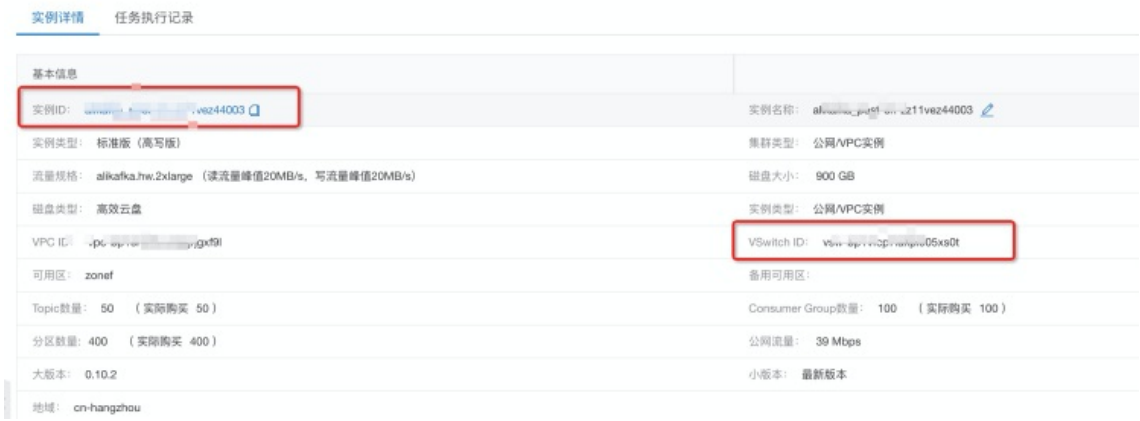
6.9. Kafka

本文介绍如何通过DLA Serverless Spark访问消息队列Kafka版。

前提条件

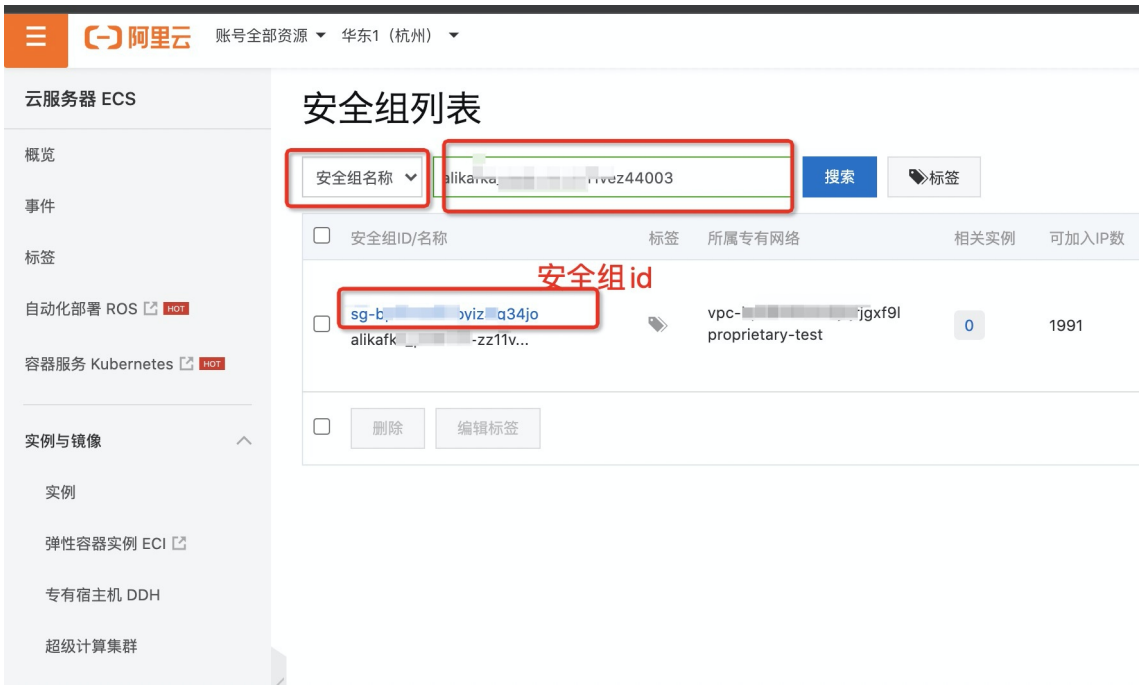
在DLA Serverless Spark中访问消息队列Kafka版前，您需要正确配置Kafka数据源网络。具体如下：

- 授权DLA Serverless Spark访问用户VPC的功能，具体内容，请参见配置数据源网络。
  i. 访问用户VPC所需要的交换机可使用Kafka服务中的交换机。在Kafka服务控制台的实例详情中您可以找到Kafka服务中的交换机。



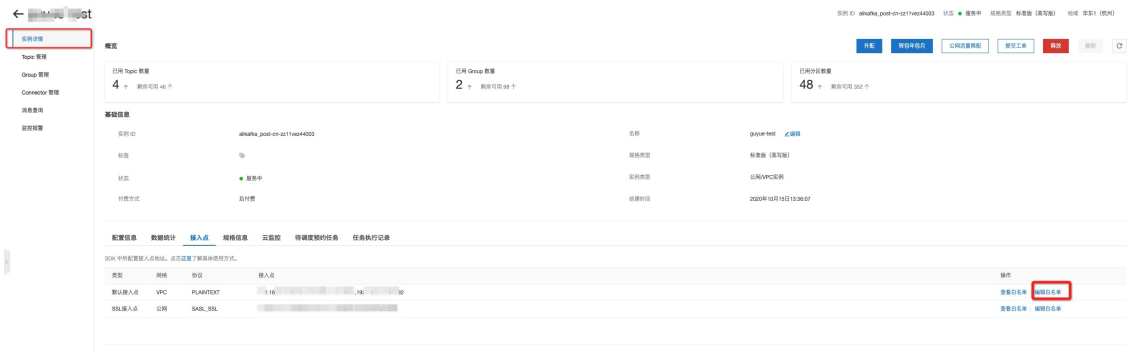
ii. 访问用户VPC所需要的安全组可以使用Kafka服务中的安全组，在**安全组列表**页面，按照安全组名称

搜索Kafka消息队列的实例ID即可找到对应的安全组。

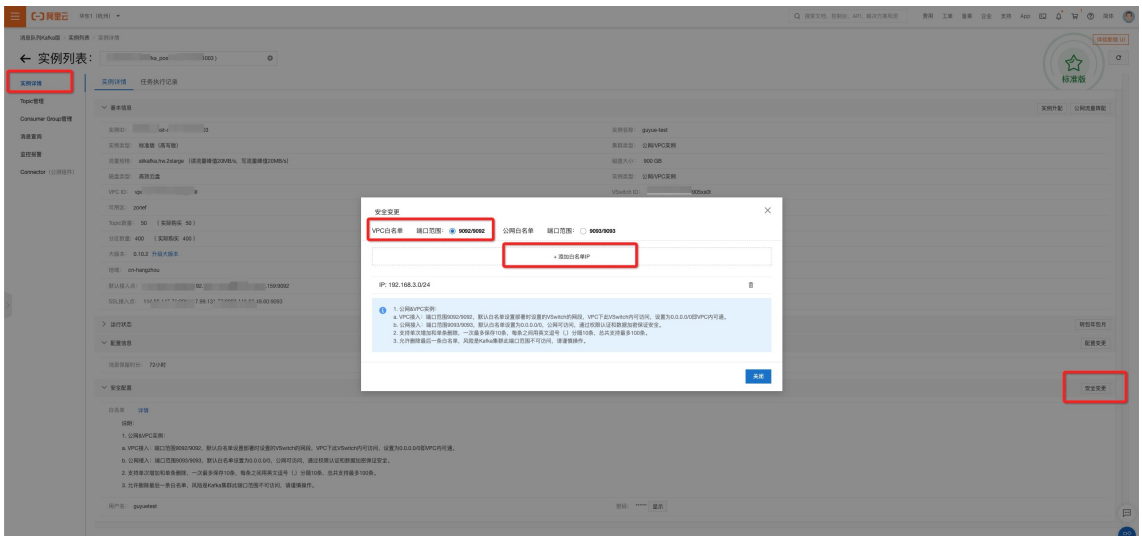


iii. 往kafka白名单中添加第一步中选中的VswitchID的网段。

新版Kafka界面加白名单



旧版Kafka UI加白名单



操作步骤

1. 登录Data Lake Analytics管理控制台。
2. 在页面左上角，选择Kafka服务所在地域。
3. 单击左侧导航栏中的**Serverless Spark > 作业管理**。
4. 在**作业编辑**页面，单击**创建作业**。
5. 在**创建作业模板**页面，按照页面提示进行参数配置后，单击**确定**创建Spark作业。

6. 单击Spark作业名，在Spark作业编辑框中输入Spark Streaming任务内容。

```
{
    "file": "oss://path/to/xxx.jar",
    "name": "Kafka",
    "className": "com.alibabacloud.cwchan.SparkKafkaSub",
    "conf": {
        "spark.driver.resourceSpec": "medium",
        "spark.executor.instances": 5,
        "spark.executor.resourceSpec": "medium",
        "spark.dla.job.log.oss.uri": "oss://path/to/spark-logs",
        "spark.dla.eni.vswitch.id": "{vswitch-id}",
        "spark.dla.eni.security.group.id": "{security-group-id}",
        "spark.dla.eni.enable": "true"
    }
}
```

编译打包过程中需要打包 `Spark-Kafka` 的相关依赖，如下所示。

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql-kafka-0-10_2.11</artifactId>
    <version>2.4.5</version>
</dependency>
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming-kafka-0-10_2.11</artifactId>
    <version>2.4.5</version>
</dependency>
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>0.10.2.2</version>
</dependency>
```

注意 如果是使用子账号提交作业，需要配置子账号权限，请参见快速配置子账号权限。

示例代码

以下是连接Kafka的核心代码片段，完整代码参考。

```
val sparkConf: SparkConf = new SparkConf()
  .setAppName("SparkKafkaSub")
val sparkSessoin = SparkSession
  .builder()
  .config(sparkConf)
  .getOrCreate()
val df = sparkSessoin
  .readStream
  .format("kafka")
  .option("kafka.bootstrap.servers", bootstrapServers)
  .option("subscribe", topicName)
  .option("group.id", groupId)
  .load()
val query = df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")
  .writeStream
  .outputMode("append")
  .format("console")
  .start()
query.awaitTermination()
```

说明 该代码消费Kafka的消息，并打印Key-Value对。

6.10. OSS

本文介绍如何通过Serverless Spark访问OSS数据源。您需要先配置访问OSS的权限，然后可以使用SQL的方式或者提交代码包（Python或者Jar包）的方式访问OSS。

操作步骤

1. 配置DLA访问OSS的权限。
   - 如果您使用的是阿里云主账号访问OSS，则默认您拥有该账号下所有OSS数据以及DLA OSS表的访问权限，无需配置，可直接使用。
   - 如果您使用RAM子账号访问OSS并提交代码包作业，需要配置代码包和Spark代码访问OSS的权限。具体操作请参见细粒度配置RAM子账号权限。
   - 如果您使用Spark SQL访问DLA OSS表的数据，需要确保您的RAM子账号关联了DLA账号，并且DLA账号拥有对应表的访问权限。如果您的RAM子账号未关联DLA账号，请进行关联操作，具体操作请参见DLA子账号绑定RAM账号。DLA账号对于表的访问权限，您可以登录DLA控制台，在**Serverless Presto > SQL执行**页面，使用 `GRANT` 或 `REVOKE` 语法进行操作。
2. 配置Spark OSS Connector。

   配置了OSS访问权限之后，您就可以使用Spark来访问OSS数据了。在Spark的作业配置文件中，您需要添加配置项 `"spark.dla.connectors" : "oss"` 。DLA平台内置了Spark OSS Connector相关的实现，默认不生效，需要配置该参数令其生效。如果您有Spark OSS Connector的其他实现方式，您不需要配置该参数，您只需提交您自己的实现Jar包，并添加相应的配置即可。
3. 访问OSS数据。

   您可以通过以下两种方式访问OSS数据：
   - 通过提交Spark SQL语句的方式来访问OSS数据，具体操作请参见Spark SQL。作业示例配置如下所示：

```
{
    "sqls": [
        "select * from `1k_tables`.`table0` limit 100",
        "insert into `1k_tables`.`table0` values(1, 'test')"
    ],
    "name": "sql oss test",
    "conf": {
        "spark.dla.connectors": "oss",
        "spark.driver.resourceSpec": "small",
        "spark.sql.hive.metastore.version": "dla",
        "spark.executor.instances": 10,
        "spark.dla.job.log.oss.uri": "oss://test/spark-logs",
        "spark.executor.resourceSpec": "small"
    }
}
```

   - 通过Java、Scala、Python代码访问OSS数据。下面以Scala为例进行说明：

```
{
  "args": ["oss://${oss-buck-name}/data/test/test.csv"],
  "name": "spark-oss-test",
  "file": "oss://${oss-buck-name}/jars/test/spark-examples-0.0.1-SNAPSHOT.jar",
  "className": "com.aliyun.spark.oss.SparkReadOss",
  "conf": {
    "spark.driver.resourceSpec": "medium",
    "spark.executor.resourceSpec": "medium",
    "spark.executor.instances": 2,
    "spark.dla.connectors": "oss"
  }
}
```

> **说明** `SparkReadOss` 对应的源码可以参考DLA Spark OSS demo。

启用OSS数据写入性能优化

当您使用自建HiveMetaStore或者DLA元数据服务访问OSS时，社区版Spark HiveClient的rename操作比较低效，DLA对此进行了优化。您只需要将参数 `spark.sql.hive.dla.metastoreV2.enable` 设置为 `true` 即可启用这项优化。示例如下：

```
{
  "args": ["oss://${oss-buck-name}/data/test/test.csv"],
  "name": "spark-oss-test",
  "file": "oss://${oss-buck-name}/jars/test/spark-examples-0.0.1-SNAPSHOT.jar",
  "className": "com.aliyun.spark.oss.WriteParquetFile",
  "conf": {
    "spark.driver.resourceSpec": "medium",
    "spark.executor.resourceSpec": "medium",
    "spark.executor.instances": 2,
    "spark.dla.connectors": "oss",
    "spark.sql.hive.dla.metastoreV2.enable": "true"
  }
}
```

OSS Connector数据写入性能优化

OSS Connector数据写入性能优化功能是DLA Spark团队基于OSS分片上传功能，针对Spark写入数据到OSS过程中大量调用OSS API导致写入性能差的问题，实现的性能优化提升。在典型场景下，性能可提升1~3倍。

您需要启用DLA Spark内置的OSS connector，并开启性能优化开关，才能使用该功能。具体配置如下：

```
spark.dla.connectors = oss;  //启用DLA Spark内置的OSS connector。
spark.hadoop.job.oss.fileoutputcommitter.enable = true;  //开启性能优化开关。
```

> **说明**
>
> - 如果启用该性能优化功能，在作业被强制Kill等情况下，可能会产生一些没有被清理的文件碎片，占用您OSS的存储空间。建议对相关OSS Bucket设置碎片生命周期规则，对过期未合并的碎片自动进行清理，建议配置周期为3天以上。具体操作请参见设置生命周期规则。
> - 该性能优化功能对RDD的 `saveAsHadoop` 前缀和 `saveAsNewAPIHadoop` 前缀的方法不生效。

使用示例：

```
{
  "args": ["oss://${oss-buck-name}/data/test/test.csv"],
  "name": "spark-oss-test",
  "file": "oss://${oss-buck-name}/jars/test/spark-examples-0.0.1-SNAPSHOT.jar",
  "className": "com.aliyun.spark.oss.WriteParquetFile",
  "conf": {
    "spark.driver.resourceSpec": "medium",
    "spark.executor.resourceSpec": "medium",
    "spark.executor.instances": 2,
    "spark.dla.connectors": "oss",
    "spark.hadoop.job.oss.fileoutputcommitter.enable": true
  }
}
```

6.11. PolarDB MySQL

PolarDB是阿里云自研的下一代关系型云数据库，100%兼容MySQL，适用于企业多样化的数据库应用场景。本文主要介绍如何通过DLA Serverless Spark访问云数据库PolarDB。

前提条件

- 已经开通对象存储OSS（Object Storage Service）服务。具体操作请参考开通OSS服务。
- 已经创建云数据库PolarDB MySQL实例。具体请参考数据库管理。
- 在PolarDB MySQL实例中已创建数据表，并插入数据。参考命令样例如下：

```
#建表语句：
CREATE TABLE `testdb`.`test_table` (
 `name` varchar(32) NULL,
 `age` INT  NULL,
 `score` DOUBLE NULL
)
#插入数据语句：
INSERT INTO `testdb`.`test_table` VALUES('aliyun01', 1001, 10.1);
INSERT INTO `testdb`.`test_table` VALUES('aliyun02', 1002, 10.2);
INSERT INTO `testdb`.`test_table` VALUES('aliyun03', 1003, 10.3);
INSERT INTO `testdb`.`test_table` VALUES('aliyun04', 1004, 10.4);
INSERT INTO `testdb`.`test_table` VALUES('aliyun05', 1005, 10.5);
```

- 准备DLA Spark访问PolarDB MySQL实例所需的安全组ID和交换机ID。具体操作请参见配置数据源网络。
- DLA Spark访问PolarDB MySQL实例所需的安全组ID或交换机ID，已添加到PolarDB MySQL实例所在集群的白名单中。具体操作请参见设置白名单。

操作步骤

1. 准备以下测试代码和依赖包来访问PolarDB MySQL，并将此测试代码和依赖包分别编译打包生成jar包上传至您的OSS。

   测试代码示例：

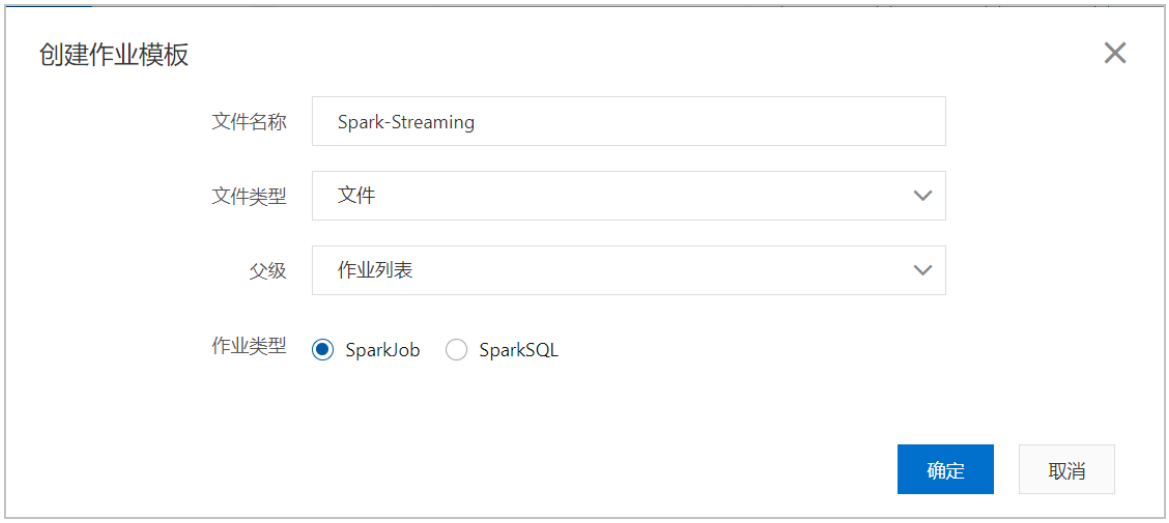
```
package com.aliyun.spark
import java.util.Properties
import org.apache.spark.sql.SparkSession
object SparkOnPOLARDB {
  def main(args: Array[String]): Unit = {
    //获取POLARDB的url、database、tableName、登录POLARDB数据库的user和password。
    val url = args(0)
    val jdbcConnURL = s"jdbc:mysql://$url"
    val database = args(1)
    val tableName = args(2)
    val user = args(3)
    val password = args(4)
    //Spark侧的表名。
    var sparkTableName = args(5)
    val sparkSession = SparkSession
      .builder()
      .appName("scala spark on POLARDB test")
      .getOrCreate()
    val driver = "com.mysql.cj.jdbc.Driver"
    //如果存在的话就删除表。
    sparkSession.sql(s"drop table if exists $sparkTableName")
    //Sql方式，Spark会映射POLARDB中表的Schema。
    val createCmd =
      s"""CREATE TABLE ${sparkTableName} USING org.apache.spark.sql.jdbc
         |    options (
         |    driver '$driver',
         |    url '$jdbcConnURL',
         |    dbtable '$database.$tableName',
         |    user '$user',
         |    password '$password'
         |    )""".stripMargin
    println(s"createCmd: \n $createCmd")
    sparkSession.sql(createCmd)
    val querySql = "select * from " + sparkTableName + " limit 1"
    sparkSession.sql(querySql).show
    //使用dataset API接口。
    val connectionProperties = new Properties()
    connectionProperties.put("driver", driver)
    connectionProperties.put("user", user)
    connectionProperties.put("password", password)
    //读取数据。
    var jdbcDf = sparkSession.read.jdbc(jdbcConnURL,
      s"$database.$tableName",
      connectionProperties)
    jdbcDf.select("name", "age", "score").show()
    val data =
      Seq(
        PersonPolardb("bill", 30, 170.5),
```

```
        PersonPolardb("gate", 29, 200.3)
      )
    val dfWrite = sparkSession.createDataFrame(data)
    //写入数据。
    dfWrite
      .write
      .mode("append")
      .jdbc(jdbcConnURL, s"$database.$tableName", connectionProperties)
    jdbcDf.select("name", "age").show()
    sparkSession.stop()
  }
}
case class PersonPolardb(name: String, age: Int, score: Double)
```

PolarDB MySQL依赖的pom文件：

```
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.22</version>
        </dependency>
```

2. 登录Data Lake Analytics管理控制台。
3. 在页面左上角，选择PolarDB MySQL实例所在地域。
4. 单击左侧导航栏中的**Serverless Spark > 作业管理**。
5. 在**作业编辑**页面，单击**创建作业**。
6. 在**创建作业模板**页面，按照页面提示进行参数配置后，单击**确定**创建Spark作业。

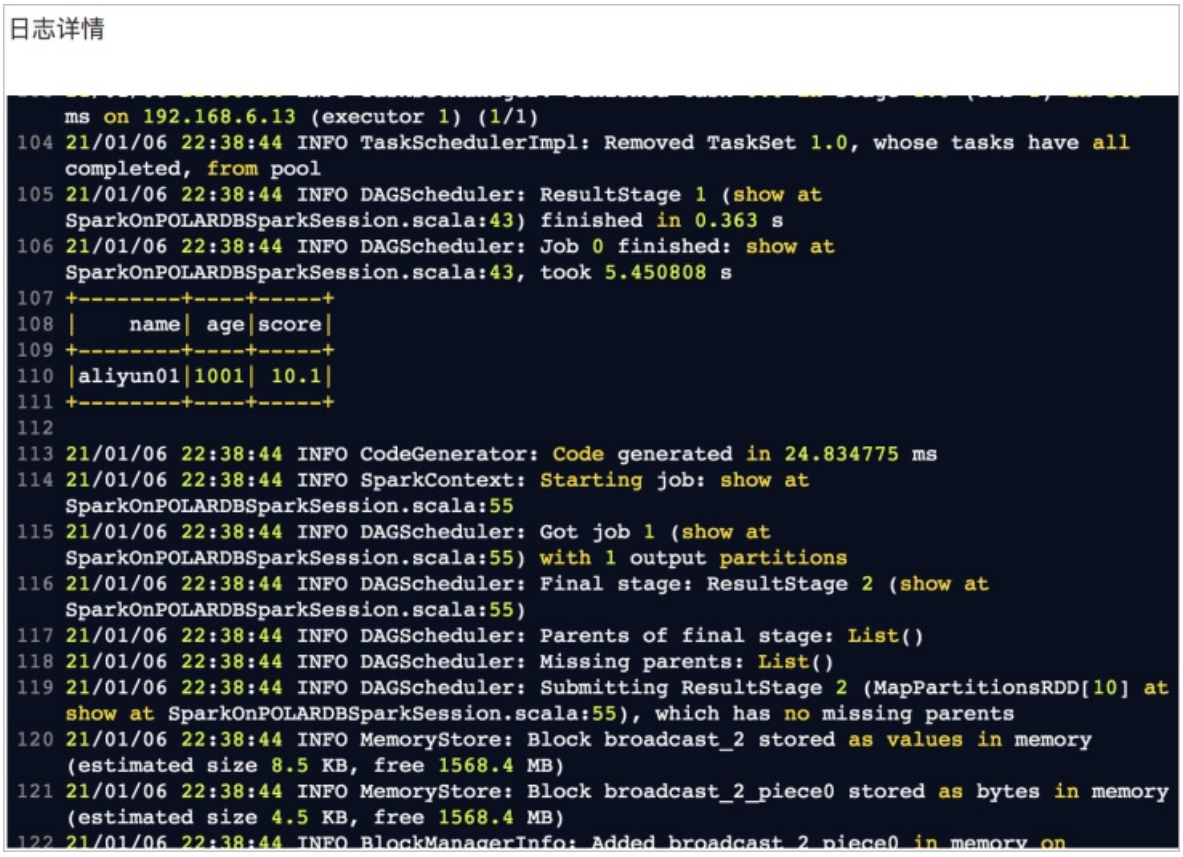


7. 单击Spark作业名，在Spark作业编辑框中输入以下作业内容，并按照以下参数说明进行参数值替换。保存并提交Spark作业。

```
{
    "args": [
        "pc-xxx.mysql.polardb.rds.aliyuncs.com:3306",  #POLARDB的私网地址和端口。
        "testdb",  #POLARDB中的数据库名。
        "test_table",  #POLARDB中的表名。
        "xxx1",  #POLARDB中的数据库登录的用户名。
        "xxx2",  #POLARDB中的数据库登录的用户名。
        "spark_on_polardb_table"  #Spark中创建映射POLARDB表的表名。
    ],
    "file": "oss://spark_test/jars/polardb/spark-examples-0.0.1-SNAPSHOT.jar",  #存放测试代码的OSS路径。
    "name": "polardb-test",
    "jars": [
        "oss://spark_test/jars/polardb/mysql-connector-java-8.0.22.jar"  #存放测试代码依赖包的OSS路径。
    ],
    "className": "com.aliyun.spark.SparkOnPOLARDB",
    "conf": {
        "spark.driver.resourceSpec": "small",
        "spark.executor.instances": 2,
        "spark.executor.resourceSpec": "small",
        "spark.dla.eni.enable": "true",
        "spark.dla.eni.vswitch.id": "vsw-xxx",   #可访问PolarDB的交换机id。
        "spark.dla.eni.security.group.id": "sg-xxx"  #可访问PolarDB的安全组id。
    }
}
```

执行结果

作业运行成功后，在任务列表中单击**操作 > 日志**，查看作业日志。出现如下日志说明作业运行成功：



6.12. PolarDB-X

PolarDB-X（原DRDS升级版）是由阿里巴巴自主研发的云原生分布式数据库，融合分布式SQL引擎DRDS与分布式自研存储X-DB，基于云原生一体化架构设计，可支撑千万级并发规模及百PB级海量存储。本文主要介绍如何通过DLA Serverless Spark访问云数据库PolarDB-X。

前提条件

- 已经开通对象存储OSS（Object Storage Service）服务。具体操作请参考开通OSS服务。
- 已经创建PolarDB-X数据库。具体请参考创建数据库。
- 在PolarDB-X数据库中已创建数据表，并插入数据。参考命令样例如下：

```
#建表语句：
CREATE TABLE `testdb_drds`.`test_table` (
 `name` varchar(32) NULL,
 `age` INT  NULL,
 `score` DOUBLE NULL
)
#插入数据语句：
INSERT INTO `testdb_drds`.`test_table` VALUES('aliyun01', 1001, 10.1);
INSERT INTO `testdb_drds`.`test_table` VALUES('aliyun02', 1002, 10.2);
INSERT INTO `testdb_drds`.`test_table` VALUES('aliyun03', 1003, 10.3);
INSERT INTO `testdb_drds`.`test_table` VALUES('aliyun04', 1004, 10.4);
INSERT INTO `testdb_drds`.`test_table` VALUES('aliyun05', 1005, 10.5);
```

- 准备DLA Spark访问PolarDB-X数据库所需的安全组ID和交换机ID。具体操作请参见配置数据源网络。
- DLA Spark访问PolarDB-X数据库所需的交换机IP，已添加到PolarDB-X数据库的白名单中。具体操作请参

见设置白名单。

操作步骤

1. 准备以下测试代码和依赖包来访问PolarDB-X，并将此测试代码和依赖包分别编译打包生成jar包上传至您的OSS。

   测试代码示例：

```
package com.aliyun.spark
import java.util.Properties
import org.apache.spark.sql.SparkSession
object SparkOnPOLARDB {
  def main(args: Array[String]): Unit = {
    //获取POLARDB的url、database、tableName、登录POLARDB数据库的user和password。
    val url = args(0)
    val jdbcConnURL = s"jdbc:mysql://$url"
    val database = args(1)
    val tableName = args(2)
    val user = args(3)
    val password = args(4)
    //Spark侧的表名。
    var sparkTableName = args(5)
    val sparkSession = SparkSession
      .builder()
      .appName("scala spark on POLARDB test")
      .getOrCreate()
    val driver = "com.mysql.cj.jdbc.Driver"
    //如果存在的话就删除表。
    sparkSession.sql(s"drop table if exists $sparkTableName")
    //Sql方式，Spark会映射POLARDB中表的Schema。
    val createCmd =
      s"""CREATE TABLE ${sparkTableName} USING org.apache.spark.sql.jdbc
         |    options (
         |    driver '$driver',
         |    url '$jdbcConnURL',
         |    dbtable '$database.$tableName',
         |    user '$user',
         |    password '$password'
         |    )""".stripMargin
    println(s"createCmd: \n $createCmd")
    sparkSession.sql(createCmd)
    val querySql = "select * from " + sparkTableName + " limit 1"
    sparkSession.sql(querySql).show
    //使用dataset API接口。
    val connectionProperties = new Properties()
    connectionProperties.put("driver", driver)
    connectionProperties.put("user", user)
    connectionProperties.put("password", password)
    //读取数据。
    var jdbcDf = sparkSession.read.jdbc(jdbcConnURL,
      s"$database.$tableName",
      connectionProperties)
    jdbcDf.select("name", "age", "score").show()
    val data =
```

```
      Seq(
        PersonPolardb("bill", 30, 170.5),
        PersonPolardb("gate", 29, 200.3)
      )
    val dfWrite = sparkSession.createDataFrame(data)
    //写入数据。
    dfWrite
      .write
      .mode("append")
      .jdbc(jdbcConnURL, s"$database.$tableName", connectionProperties)
    jdbcDf.select("name", "age").show()
    sparkSession.stop()
  }
}
case class PersonPolardb(name: String, age: Int, score: Double)
```

PolarDB-X依赖的pom文件：

```
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.22</version>
        </dependency>
```

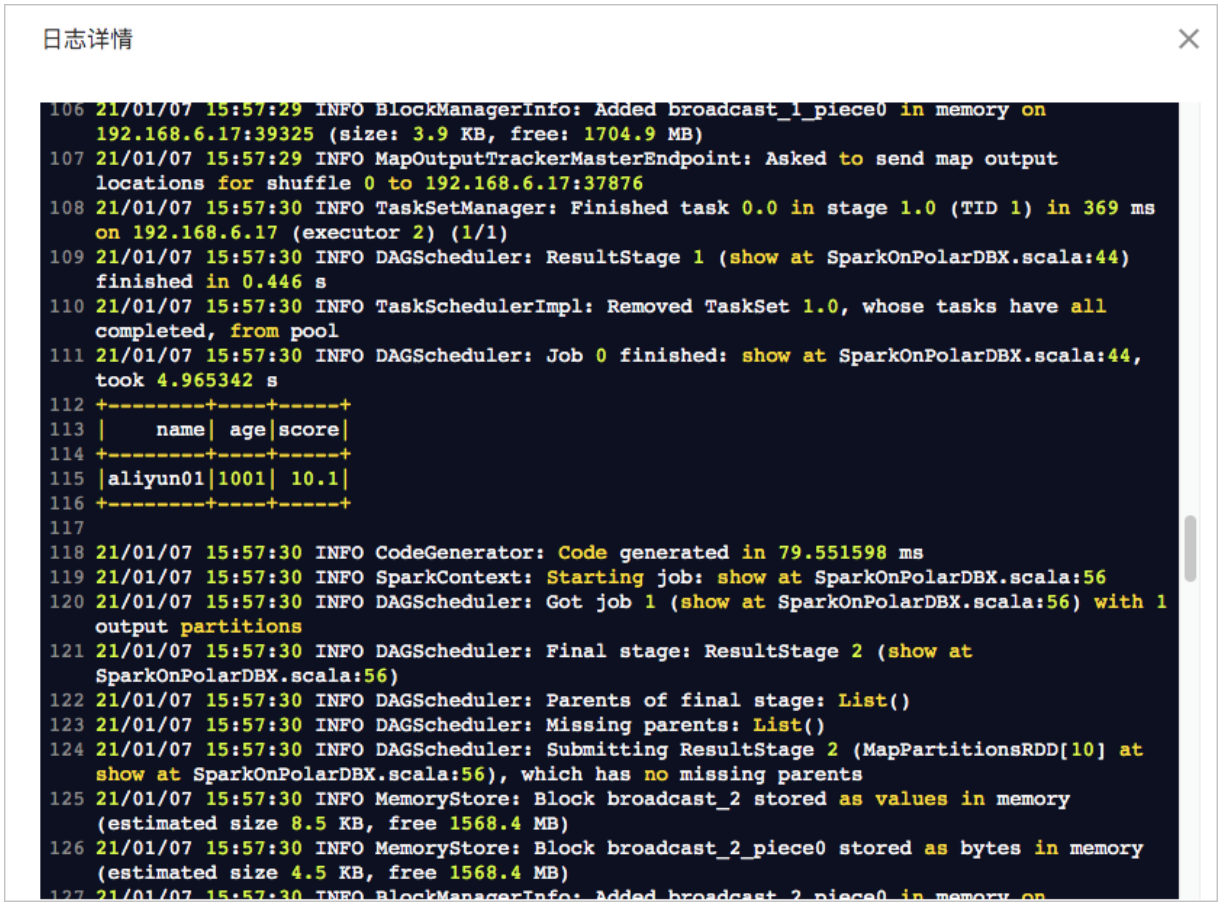
2. 登录Data Lake Analytics管理控制台。
3. 在页面左上角，选择PolarDB-X实例所在的地域。
4. 单击左侧导航栏中的**Serverless Spark > 作业管理**。
5. 在**作业编辑**页面，单击**创建作业**。
6. 在**创建作业模板**页面，按照页面提示进行参数配置后，单击**确定**创建Spark作业。

7. 单击Spark作业名，在Spark作业编辑框中输入以下作业内容，并按照以下参数说明进行参数值替换。保存并提交Spark作业。

```
{
    "args": [
        "xxx.drds.aliyuncs.com:3306",  #POLARDB-X的内网地址和端口。
        "testdb_drds",  #POLARDB-X中的数据库名称。
        "test_table",  #POLARDB-X中的数据库表名。
        "xxx1",  #登录POLARDB-X数据库的用户名。
        "xxx2",  #登录POLARDB-X数据库的密码。
        "spark_on_polardbx_table"  #Spark中创建映射POLARDB-X表的表名。
    ],
    "file": "oss://spark_test/jars/polardbx/spark-examples-0.0.1-SNAPSHOT.jar",  #存放测试代码的OSS路径。
    "name": "polardbx-test",
    "jars": [
        "oss://spark_test/jars/polardbx/mysql-connector-java-8.0.22.jar"  #存放测试代码依赖包的OSS路径。
    ],
    "className": "com.aliyun.spark.SparkOnPolarDBX",
    "conf": {
        "spark.driver.resourceSpec": "small", #表示driver的规格，有small、medium、large、xlarge之分。
        "spark.executor.instances": 2,  #表示executor的个数。
        "spark.executor.resourceSpec": "small",  #表示executor的规格，有small、medium、large、xlarge之分。
        "spark.dla.eni.enable": "true",  #开启访问用户VPC网络的权限。当您需要访问用户VPC网络内的数据时，需要开启此选项。
        "spark.dla.eni.vswitch.id": "vsw-xxx",  #可访问PolarDB-X的交换机id。
        "spark.dla.eni.security.group.id": "sg-xxx"  #可访问PolarDB-X的安全组id。
    }
}
```

执行结果

作业运行成功后，在任务列表中单击**操作 > 日志**，查看作业日志。出现如下日志说明作业运行成功：



6.13. MongoDB

云数据库MongoDB版（ApsaraDB for MongoDB）是基于飞天分布式系统和高可靠存储引擎的在线数据库服务，完全兼容MongoDB协议，提供稳定可靠、弹性伸缩的数据库服务。本文主要介绍如何通过DLA Serverless Spark访问云数据库MongoDB。

前提条件

- 已经开通对象存储OSS（Object Storage Service）服务。具体操作请参考开通OSS服务。
- 已经创建云数据库MongoDB实例。具体请参考创建实例。
- 创建MongoDB的连接准备数据。

  通过DMS连接MongoDB分片集群实例，执行以下命令在DMS上创建数据库config，并插入以下测试数据创建连接准备数据。具体示例如下：

  ```
  db.createCollection("test_collection");
  db.test_collection.insert( {"id":"id01","name":"name01"});
  db.test_collection.insert( {"id":"id02","name":"name02"});
  db.test_collection.insert( {"id":"id03","name":"name03"});
  db.test_collection.find().pretty()
  ```

- 准备DLA Spark访问MongoDB实例所需的安全组ID和交换机ID。具体操作请参见配置数据源网络。
- DLA Spark访问MongoDB实例所需的安全组ID或交换机ID，已添加到MongoDB实例的白名单中。具体操作请参见设置白名单。

操作步骤

1. 准备以下测试代码和依赖包来访问MongoDB，并将此测试代码和依赖包分别编译打包生成jar包上传至您的OSS。

   测试代码示例：

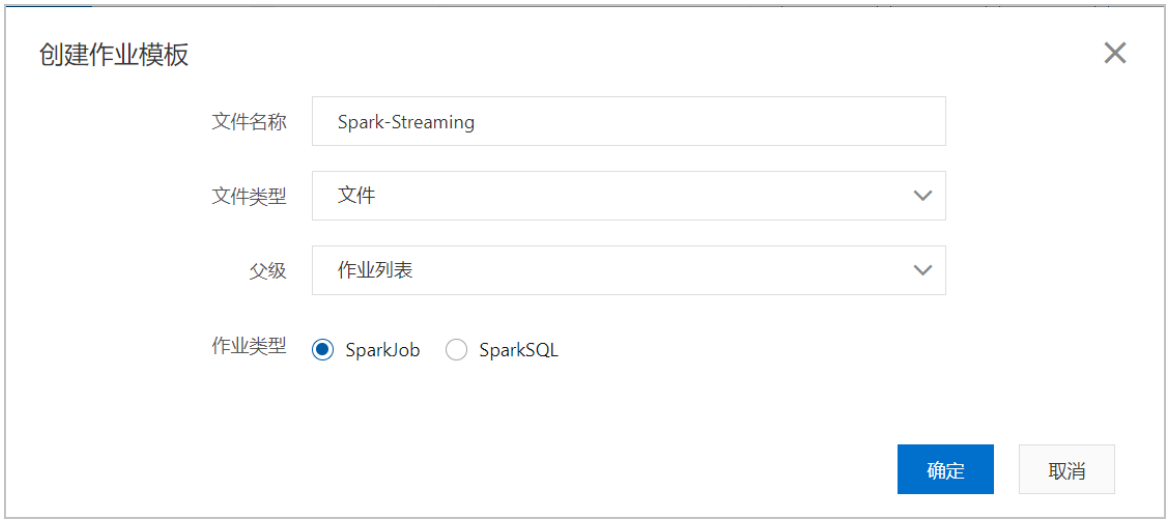
```
package com.aliyun.spark
import com.mongodb.spark.MongoSpark
import com.mongodb.spark.config.{ReadConfig, WriteConfig}
import org.apache.spark.SparkConf
import org.apache.spark.sql.SparkSession
import org.bson.Document
object SparkOnMongoDB {
  def main(args: Array[String]): Unit = {
    //获取MongoDB的connectionStringURI，database和collection。
    val connectionStringURI = args(0)
    val database = args(1)
    val collection = args(2)
    //Spark侧的表名。
    var sparkTableName = if (args.size > 3) args(3) else "spark_on_mongodb_sparksession
_test01"
    val sparkSession = SparkSession
      .builder()
      .appName("scala spark on MongoDB test")
      .getOrCreate()
    //Spark读取MongoDB数据有多种方式。
    //使用Dataset API方式：
    //设置MongoDB的参数。
    val sparkConf = new SparkConf()
      .set("spark.mongodb.input.uri", connectionStringURI)
      .set("spark.mongodb.input.database", database)
      .set("spark.mongodb.input.collection", collection)
      .set("spark.mongodb.output.uri", connectionStringURI)
      .set("spark.mongodb.output.database", database)
      .set("spark.mongodb.output.collection", collection)
    val readConf = ReadConfig(sparkConf)
    //获取Dataframe。
    val df = MongoSpark.load(sparkSession, readConf)
    df.show(1)
    //使用MongoSpark.save入库数据到MongoDB。
    val docs =
      """
        |{"id": "id105", "name": "name105"}
        |{"id": "id106", "name": "name106"}
        |{"id": "id107", "name": "name107"}
        |"""
        .trim.stripMargin.split("[\\r\\n]+").toSeq
    val writeConfig: WriteConfig = WriteConfig(Map(
      "uri" -> connectionStringURI,
      "spark.mongodb.output.database" -> database,
      "spark.mongodb.output.collection"-> collection))
    MongoSpark.save(sparkSession.sparkContext.parallelize(docs.map(Document.parse)), wr
iteConfig)
    //使用Sql的方式，SQL的方式有两种，指定Schema和不指定Schema。
```

```
    //指定Schema的创建方式，Schema中的字段必须和MongoDB中Collection的Schema一致。
    var createCmd =
    s"""CREATE TABLE ${sparkTableName} (
       |      id String,
       |      name String
       |    ) USING com.mongodb.spark.sql
       |    options (
       |    uri '$connectionStringURI',
       |    database '$database',
       |    collection '$collection'
       |    )""".stripMargin
    sparkSession.sql(createCmd)
    var querySql = "select * from " + sparkTableName + " limit 1"
    sparkSession.sql(querySql).show
    //不指定Schema的创建方式，不指定Schema，Spark会映射MOngoDB中collection的Schema。
    sparkTableName = sparkTableName + "_noschema"
    createCmd =
      s"""CREATE TABLE ${sparkTableName} USING com.mongodb.spark.sql
         |    options (
         |    uri '$connectionStringURI',
         |    database '$database',
         |    collection '$collection'
         |    )""".stripMargin
    sparkSession.sql(createCmd)
    querySql = "select * from " + sparkTableName + " limit 1"
    sparkSession.sql(querySql).show
    sparkSession.stop()
  }
}
```

MongoDB依赖的pom文件：

```
        <dependency>
            <groupId>org.mongodb.spark</groupId>
            <artifactId>mongo-spark-connector_2.11</artifactId>
            <version>2.4.2</version>
        </dependency>
        <dependency>
            <groupId>org.mongodb</groupId>
            <artifactId>mongo-java-driver</artifactId>
            <version>3.8.2</version>
        </dependency>
```

2. 登录Data Lake Analytics管理控制台。
3. 在页面左上角，选择MongoDB实例所在地域。
4. 单击左侧导航栏中的**Serverless Spark > 作业管理**。
5. 在**作业编辑**页面，单击**创建作业**。
6. 在**创建作业模板**页面，按照页面提示进行参数配置后，单击**确定**创建Spark作业。

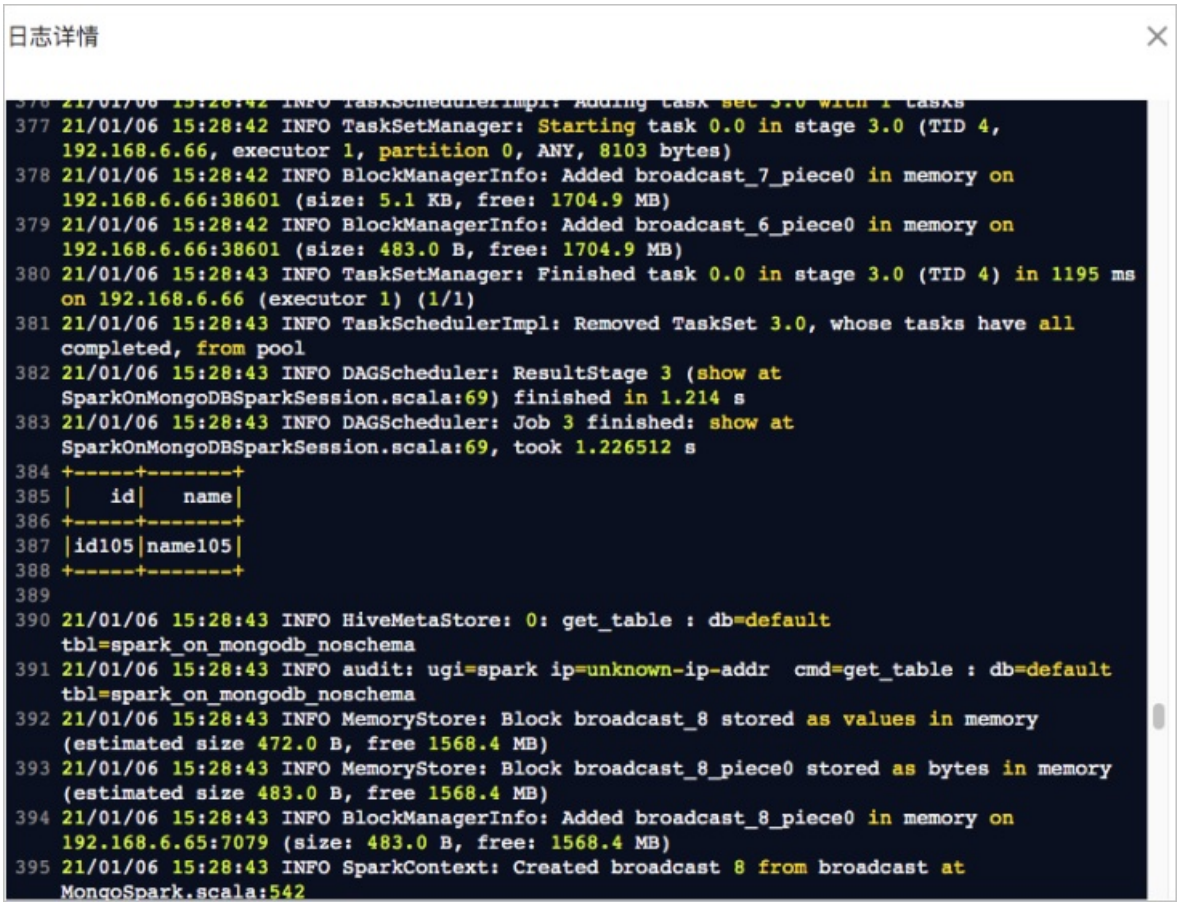


7. 单击Spark作业名，在Spark作业编辑框中输入以下作业内容，并按照以下参数说明进行参数值替换。保存并提交Spark作业。

```
{
    "args": [
        "mongodb://root:xxx@xxx:3717,xxx:3717/xxx",  #MongoDB集群中的"连接信息 (Connection StringURI)"。
        "config",  #MongoDB集群中的数据库名称。
        "test_collection",  #MongoDB集群中的collection名称。
        "spark_on_mongodb"  #Spark中创建映射MongoDB中collection的表名。
    ],
    "file": "oss://spark_test/jars/mongodb/spark-examples-0.0.1-SNAPSHOT.jar",  #存放测试软件包的OSS路径。
    "name": "mongodb-test",
    "jars": [
        "oss://spark_test/jars/mongodb/mongo-java-driver-3.8.2.jar",  ##存放测试软件依赖包的OSS路径。
        "oss://spark_test/jars/mongodb/mongo-spark-connector_2.11-2.4.2.jar"  ##存放测试软件依赖包的OSS路径。
    ],
    "className": "com.aliyun.spark.SparkOnMongoDB",
    "conf": {
        "spark.driver.resourceSpec": "small",
        "spark.executor.instances": 2,
        "spark.executor.resourceSpec": "small",
        "spark.dla.eni.enable": "true",
        "spark.dla.eni.vswitch.id": "vsw-xxx",   #可访问MongoDB的交换机id。
        "spark.dla.eni.security.group.id": "sg-xxx"  #可访问MongoDB的安全组id。
    }
}
```

执行结果

作业运行成功后，在任务列表中单击**操作 > 日志**，查看作业日志。出现如下日志说明作业运行成功：



6.14. Redis

阿里云数据库Redis版是兼容开源Redis协议标准、提供内存加硬盘混合存储的数据库服务，基于高可靠双机热备架构及可平滑扩展的集群架构，可充分满足高吞吐、低延迟及弹性变配的业务需求。本文主要介绍如何通过DLA Serverless Spark访问云数据库Redis。

前提条件

- 已经开通对象存储OSS（Object Storage Service）服务。具体操作请参考开通OSS服务。
- 已经创建云数据库Redis实例。具体请参考步骤1：创建实例。
- 准备DLA Spark访问Redis实例所需的安全组ID和交换机ID。具体操作请参见配置数据源网络。
- DLA Spark访问Redis实例所需的安全组ID或交换机ID，已添加到Redis实例的白名单中。具体操作请参见步骤2：设置白名单。

操作步骤

1. 准备以下测试代码和依赖包来访问Redis，并将此测试代码和依赖包分别编译打包生成jar包上传至您的OSS。

   测试代码示例：

```
package com.aliyun.spark
import org.apache.spark.SparkConf
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.types._
object SparkOnRedis {
```

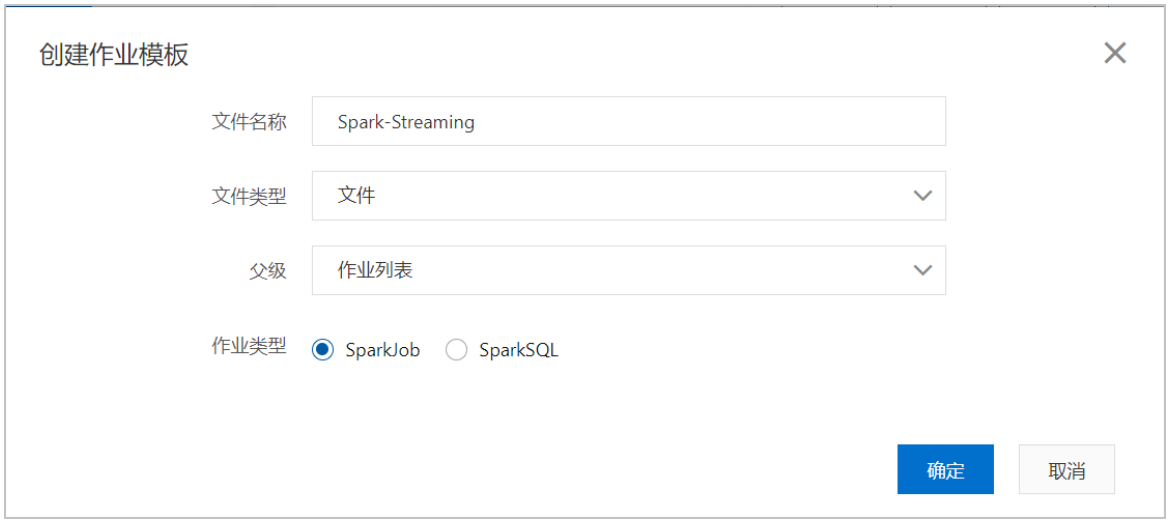
```
  def main(args: Array[String]): Unit = {
    //获取Redis的, redisHost：内网连接地址(host)，redisPort:端口号(port)，redisPassword：连接密码。
    val redisHost = args(0)
    val redisPort = args(1)
    val redisPassword = args(2)
    //redis侧的表名。
    var redisTableName = args(3)
    //spark conf中配置的redis信息。
    val sparkConf = new SparkConf()
      .set("spark.redis.host", redisHost)
      .set("spark.redis.port", redisPort)
      .set("spark.redis.auth", redisPassword)
    val sparkSession = SparkSession
      .builder()
      .config(sparkConf)
      .getOrCreate()
    //样例数据。
    val data =
      Seq(
        Person("John", 30, "60 Wall Street", 150.5),
        Person("Peter", 35, "110 Wall Street", 200.3)
      )
    //通过dataset API写入数据。
    val dfw = sparkSession.createDataFrame(data)
    dfw.write.format("org.apache.spark.sql.redis")
      .option("model", "hash")
      .option("table", redisTableName)
      .save()
    //默认方式读取redis的hash值。
    var loadedDf = sparkSession.read.format("org.apache.spark.sql.redis")
      .option("table", redisTableName)
      .load()
      .cache()
    loadedDf.show(10)
    //设置infer.schema=true，spark会检索redis的Schema。
    loadedDf = sparkSession.read.format("org.apache.spark.sql.redis")
      //        .option("table", redisTableName)
      .option("keys.pattern", redisTableName + ":*")
      .option("infer.schema", "true")
      .load()
    loadedDf.show(10)
    //指定Schema的方式。
    loadedDf = sparkSession.read.format("org.apache.spark.sql.redis")
      .option("keys.pattern", redisTableName + ":*")
      .schema(StructType(Array(
        StructField("name", StringType),
        StructField("age", IntegerType),
        StructField("address", StringType),
        StructField("salary", DoubleType)
      )))
      .load()
    loadedDf.show(10)
    sparkSession.stop()
  }
```

```
}
case class Person(name: String, age: Int, address: String, salary: Double)
```

Redis依赖的pom文件：

```
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-pool2</artifactId>
            <version>2.0</version>
        </dependency>
        <dependency>
            <groupId>com.redislabs</groupId>
            <artifactId>jedis</artifactId>
            <version>3.0.0-m1</version>
        </dependency>
        <dependency>
            <groupId>com.redislabs</groupId>
            <artifactId>spark-redis</artifactId>
            <version>2.3.1-m3</version>
        </dependency>
```

2. 登录Data Lake Analytics管理控制台。
3. 在页面左上角，选择Redis实例所在地域。
4. 单击左侧导航栏中的**Serverless Spark > 作业管理**。
5. 在**作业编辑**页面，单击**创建作业**。
6. 在**创建作业模板**页面，按照页面提示进行参数配置后，单击**确定**创建Spark作业。

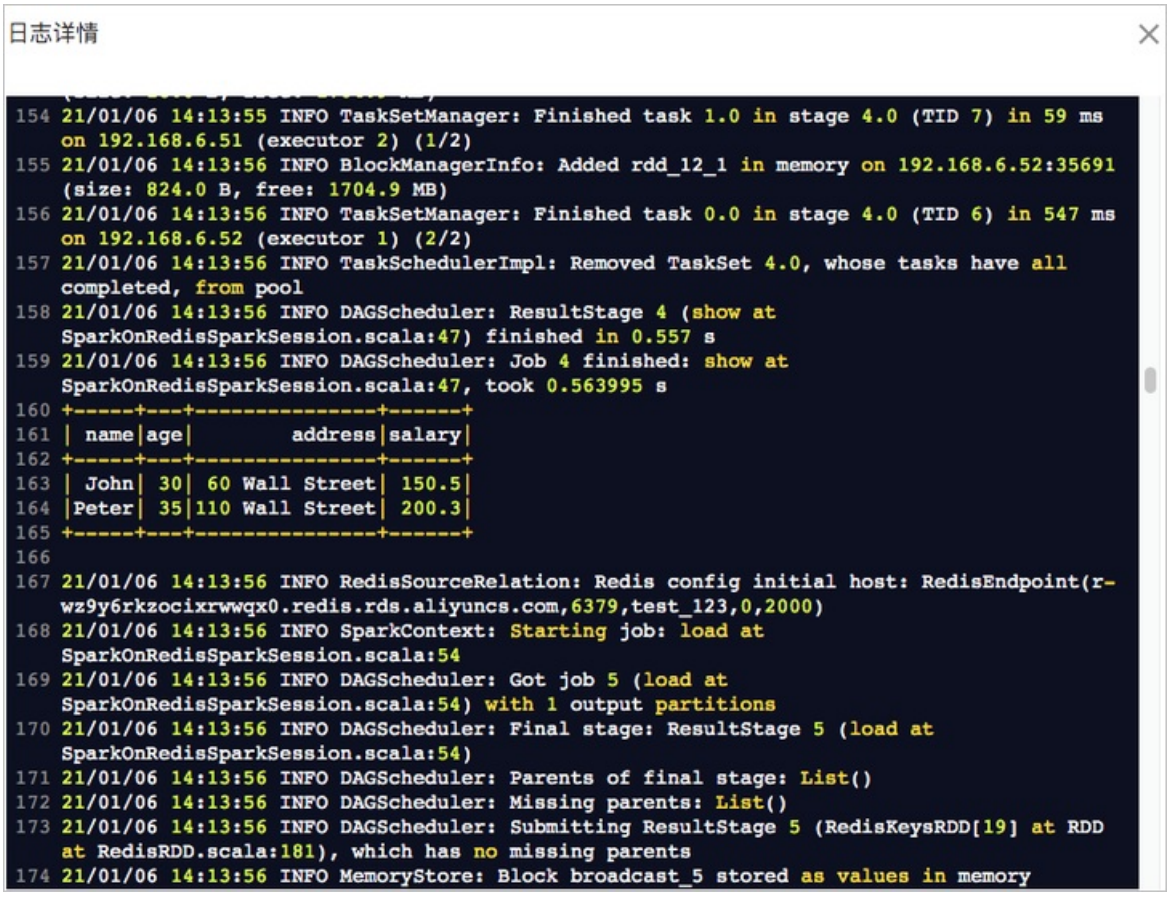


7. 单击Spark作业名，在Spark作业编辑框中输入以下作业内容，并按照以下参数说明进行参数值替换。保存并提交Spark作业。

```
{
    "args": [
        "r-xxx1.redis.rds.aliyuncs.com",  #Redis数据库"连接信息"的"内网连接地址（host）。
        "6379",  #Redis数据库"连接信息"的"端口号（port）"。
        "xxx2",  #Redis数据库登录密码。
        "spark-test"  #Redis数据库的表名。
    ],
    "file": "oss://spark_test/jars/redis/spark-examples-0.0.1-SNAPSHOT.jar",  #存放测试软件包的OSS路径。
    "name": "redis-test",
    "jars": [
        "oss://spark_test/jars/redis/spark-redis-2.3.1-m3.jar",  #存放测试软件依赖包的OSS路径。
        "oss://spark_test/jars/redis/commons-pool2-2.0.jar",  #存放测试软件依赖包的OSS路径。
        "oss://spark_test/jars/redis/jedis-3.0.0-m1.jar"  #存放测试软件依赖包的OSS路径。
    ],
    "className": "com.aliyun.spark.SparkOnRedis",
    "conf": {
        "spark.driver.resourceSpec": "small",
        "spark.executor.instances": 2,
        "spark.executor.resourceSpec": "small",
        "spark.dla.eni.enable": "true",
        "spark.dla.eni.vswitch.id": "vsw-xxx",  #可访问Redis的交换机id。
        "spark.dla.eni.security.group.id": "sg-xxx"  #可访问Redis的安全组id。
    }
}
```

执行结果

作业运行成功后，在任务列表中单击**操作 > 日志**，查看作业日志。出现如下日志说明作业运行成功：



6.15. MaxCompute

大数据计算服务（MaxCompute，原名ODPS）是一种快速、完全托管的TB/PB级数据仓库解决方案。本文主要介绍如何通过DLA Serverless Spark访问MaxCompute。

前提条件

- 已经开通对象存储OSS（Object Storage Service）服务。具体操作请参考开通OSS服务。
- 已经创建MaxCompute的项目空间。具体操作请参考创建MaxCompute项目。假设本文中项目空间为**spark_on_maxcompute**，模式为**简单模式（单环境）**。
- 已经在MaxCompute的项目空间中创建了表。假设本文中表名称为**sparktest**。建表示例如下：

```
CREATE TABLE `sparktest` (
 `a` int,
 `b` STRING
)
PARTITIONED BY (pt string);
```

- 已经为MaxCompute的项目空间添加RAM用户及对应角色。具体操作请参考添加工作空间成员并设置角色。

操作步骤

1. 准备以下测试代码和依赖包来访问MaxCompute，并将此测试代码和依赖包分别编译打包生成jar包上传至您的OSS。

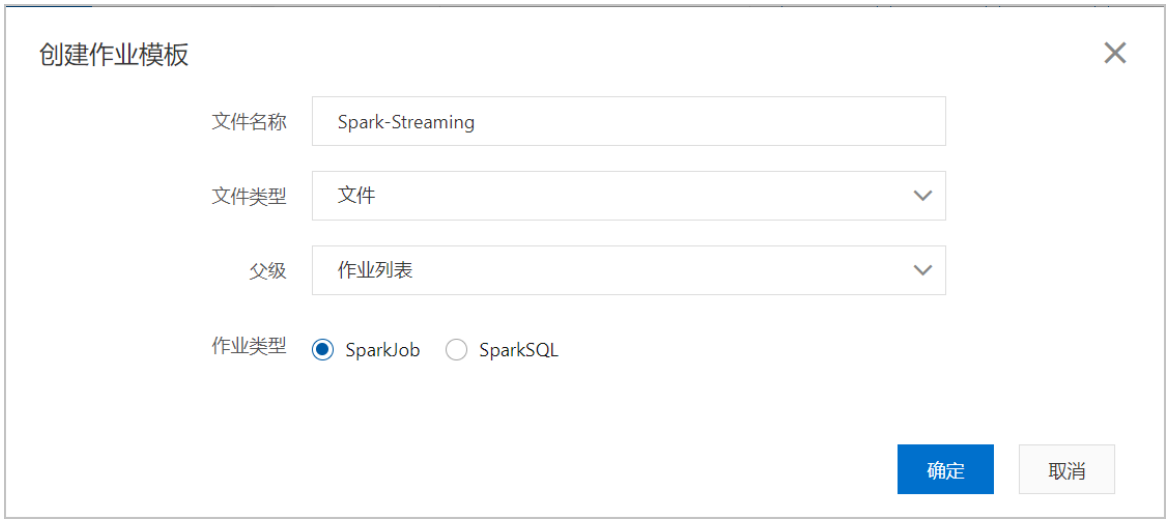
   测试代码示例：

```
package com.aliyun.spark
import org.apache.spark.sql.{SaveMode, SparkSession}
object MaxComputeDataSourcePartitionSample {
  def main(args: Array[String]): Unit = {
    val accessKeyId = args(0)
    val accessKeySecret = args(1)
    val odpsUrl = args(2)
    val tunnelUrl = args(3)
    val project = args(4)
    val table = args(5)
    var numPartitions = 1
    if(args.length > 6)
      numPartitions = args(6).toInt
    val ss = SparkSession.builder().appName("Test Odps Read").getOrCreate()
    import ss.implicits._
    val dataSeq = (1 to 1000000).map {
      index => (index, (index-3).toString)
    }.toSeq
    val df = ss.sparkContext.makeRDD(dataSeq).toDF("a", "b")
    System.out.println("*****" + table + ",before overwrite table")
    df.write.format("org.apache.spark.aliyun.odps.datasource")
      .option("odpsUrl", odpsUrl)
      .option("tunnelUrl", tunnelUrl)
      .option("table", table)
      .option("project", project)
      .option("accessKeySecret", accessKeySecret)
      .option("partitionSpec", "pt='2018-04-01'")
      .option("allowCreateNewPartition", true)
      .option("accessKeyId", accessKeyId).mode(SaveMode.Overwrite).save()
    System.out.println("*****" + table + ",after overwrite table, before read table")
    val readDF = ss.read
      .format("org.apache.spark.aliyun.odps.datasource")
      .option("odpsUrl", odpsUrl)
      .option("tunnelUrl", tunnelUrl)
      .option("table", table)
      .option("project", project)
      .option("accessKeySecret", accessKeySecret)
      .option("accessKeyId", accessKeyId)
      .option("partitionSpec", "pt='2018-04-01'")
      .option("numPartitions",numPartitions).load()
    readDF.collect().foreach(println)
  }
}
```

MaxCompute依赖的pom文件：

```
        <dependency>
            <groupId>com.aliyun.odps</groupId>
            <artifactId>odps-sdk-commons</artifactId>
            <version>0.28.4-public</version>
        </dependency>
        <dependency>
            <groupId>com.aliyun.apsaradb</groupId>
            <artifactId>maxcompute-spark</artifactId>
            <version>0.28.4-public_2.4.3-1.0-SNAPSHOT</version>
        </dependency>
```

2. 登录Data Lake Analytics管理控制台。
3. 在页面左上角，选择MaxCompute项目空间所在地域。
4. 单击左侧导航栏中的**Serverless Spark > 作业管理**。
5. 在**作业编辑**页面，单击**创建作业**。
6. 在**创建作业模板**页面，按照页面提示进行参数配置后，单击**确定**创建Spark作业。



7. 单击Spark作业名，在Spark作业编辑框中输入以下作业内容，并按照以下参数说明进行参数值替换。保存并提交Spark作业。

```
{
    "args": [
        "xxx1",  #具备访问MaxCompute权限的AccessKey ID。
        "xxx2",  #具备访问MaxCompute权限的AccessKey Secret。
        "http://service.cn.maxcompute.aliyun-inc.com/api",  #MaxCompute的VPC网络Endpoint
。
        "http://dt.cn-shenzhen.maxcompute.aliyun-inc.com",  #MaxCompute的VPC网络Tunnel E
ndpoint。
        "spark_on_maxcompute",  #MaxCompute的工作空间名称。
        "sparktest",  #MaxCompute的数据表名。
        "2"  #MaxCompute数据表的分区数。
    ],
    "file": "oss://spark_test/jars/maxcompute/spark-examples-0.0.1-SNAPSHOT.jar", #存放
测试软件包的OSS路径。
    "name": "Maxcompute-test",
    "jars": [
        ##存放测试软件依赖包的OSS路径。
        "oss://spark_test/jars/maxcompute/maxcompute-spark-0.28.4-public_2.4.3-1.0-SNAP
SHOT.jar",
        "oss://spark_test/jars/maxcompute/odps-sdk-commons-0.28.4-public.jar",
        "oss://spark_test/jars/maxcompute/odps-sdk-core-0.28.4-public.jar",
        "oss://spark_test/jars/maxcompute/mail-1.4.7.jar"
    ],
    "className": "com.aliyun.spark.MaxComputeDataSourcePartitionSample",
    "conf": {
        "spark.driver.resourceSpec": "small",
        "spark.executor.instances": 2,
        "spark.executor.resourceSpec": "small"
    }
}
```

执行结果

作业运行成功后，在任务列表中单击**操作 > 日志**，查看作业日志。出现如下日志说明作业运行成功：

日志详情

```
[999511,999508]
[999512,999509]
[999513,999510]
[999514,999511]
[999515,999512]
[999516,999513]
[999517,999514]
[999518,999515]
[999519,999516]
[999520,999517]
[999521,999518]
[999522,999519]
[999523,999520]
[999524,999521]
[999525,999522]
[999526,999523]
[999527,999524]
[999528,999525]
[999529,999526]
[999530,999527]
[999531,999528]
[999532,999529]
[999533,999530]
[999534,999531]
[999535,999532]
[999536,999533]
[999537,999534]
[999538,999535]
[999539,999536]
[999540,999537]
[999541,999538]
[999542,999539]
```

6.16. Hive

本文主要介绍如何使用DLA Spark访问用户VPC中的Hive集群。

前提条件

- 您已开通数据湖分析DLA（Data Lake Analytics）服务，如何开通，请参见开通云原生数据湖分析服务。
- 您已登录云原生数据库分析DLA控制台，在云原生数据湖分析DLA控制台上创建了**Spark虚拟集群**。

- 您已开通对象存储OSS（Object Storage Service）服务。如何开通，请参见开通OSS服务
- 准备创建Spark计算节点所需要的**交换机id**和**安全组id**，可以选择已有的交换机和安全组，也可以新建交换机和安全组。交换机和安全组需要满足以下条件：
  - **交换机需要与您的Hive服务集群在同一VPC下。**可使用您Hive集群控制台上的交换机ID。
  - **安全组需要与您的Hive服务集群在同一VPC下。**您可以前往ECS管理控制台-**网络与安全-安全组**按照专有网络（VPC）ID搜索该VPC下的安全组，任意选择一个安全组ID即可。

- 如果您的Hive服务有**白名单**控制，需要您将交换机网段加入到您Hive服务的**白名单**中。

操作步骤

1. **如果您的Hive元数据使用的是独立的RDS且表数据存放在OSS中，则可以使用下列配置并跳过后续步骤，否则请您从第二步开始配置。**

```
{
    "name": "spark-on-hive",
    "className": "com.aliyun.spark.SparkHive", #连接Hive的测试代码，按需修改名称
    "jars": [
        "oss://path/to/mysql-connector-java-5.1.47.jar"
    ],
    "conf": {
        "spark.dla.eni.vswitch.id": "<交换机id>",
        "spark.dla.eni.security.group.id": "<安全组id>",
        "spark.dla.eni.enable": "true",
        "spark.driver.resourceSpec": "medium",
        "spark.dla.connectors": "oss",
        "spark.executor.instances": 1,
        "spark.sql.catalogImplementation": "hive",
        "spark.executor.resourceSpec": "medium",
        "spark.hadoop.javax.jdo.option.ConnectionDriverName": "com.mysql.jdbc.Driver",
        "spark.hadoop.javax.jdo.option.ConnectionUserName": "<hive_user_name>", #Hive RDS的用户名
        "spark.hadoop.javax.jdo.option.ConnectionPassword": "<your_pass_word>", #Hive RDS的密码
        "spark.hadoop.javax.jdo.option.ConnectionURL": "<jdbc连接>", #Hive RDS 的jdbc链接
        "spark.dla.job.log.oss.uri": "<日志目录路径>"
    },
    "file": "<oss://主资源Jar包路径>"
}
```

> **说明** jars中指定的Jar包是MySQL的jdbc连接器，可从官方Maven仓库，并上传到oss。

2. **获取需要在DLA Spark配置的Hive相关参数。**

> **说明** 如果您无法在您的Hive服务所在的集群中执行spark作业，可以跳过这步。

我们提供了工具来读取你Hive服务所在的集群的配置，您可以按照下面的地址下载 `spark-examples-0.0.1-SNAPSHOT-shaded.jar` 并上传至OSS, 然后提交Spark作业到您的 `Hive` 服务所在集群上执行，即可在作业输出中获得访问您Hive集群所需的配置。

```
wget https://dla003.oss-cn-hangzhou.aliyuncs.com/GetSparkConf/spark-examples-0.0.1-SNAPSHOT-shaded.jar
```

- **EMR集群**用户将Jar包上传至OSS后，可以通过以下命令提交作业到EMR集群获取配置作业：

```
--class com.aliyun.spark.util.GetConfForServerlessSpark
--deploy-mode client
ossref://{path/to}/spark-examples-0.0.1-SNAPSHOT-shaded.jar
get hive hadoop
```

作业运行完毕后，可以通过SparkUI查看driver的stdout输出或者从作业详情中的提交日志中查看输出的配置。

- 云Hbase-Spark用户可以将Jar包上传至**资源管理**目录后，用以下命令提交获取配置作业：

```
--class com.aliyun.spark.util.GetConfForServerlessSpark
/{path/to}/spark-examples-0.0.1-SNAPSHOT-shaded.jar
get hive hadoop
```

等待作业完成后，通过SparkUI的driver中的stdout查看输出配置。

- 其他Hive集群，如果您在集群上未设置 `HIVE_CONF_DIR` 环境变量，则需要手动输入 `HIVE_CONF_DIR` 路径。

```
--class com.aliyun.spark.util.GetConfForServerlessSpark
--deploy-mode client
/{path/to}/spark-examples-0.0.1-SNAPSHOT-shaded.jar
get --hive-conf-dir </path/to/your/hive/conf/dir> hive hadoop
```

3. **编写访问Hive的SparkApplication。**

以下示例代码可以首先根据用户传入的表名，在用户 `default namespace` 创建一个表，该表只有一列字符串类型的数据，内容为 `hello，dla-spark` ，然后从该表读出这一列数据，并打印到stdout：

```
package com.aliyun.spark
import org.apache.spark.sql.SparkSession
object SparkHive {
  def main(args: Array[String]): Unit = {
    val sparkSession = SparkSession
      .builder()
      .appName("Spark HIVE TEST")
      .enableHiveSupport()
      .getOrCreate()
    val welcome = "hello, dla-spark"
    //Hive表名
    val tableName = args(0)
    import sparkSession.implicits._
    //将只有一行一列数据的DataFrame: df 存入到Hive, 表名为用户传进来的tableName, 列名为welcom
e_col
    val df = Seq(welcome).toDF("welcome_col")
    df.write.format("hive").mode("overwrite").saveAsTable(tableName)
    //从Hive中读取表 tableName
    val dfFromHive = sparkSession.sql(
      s"""
        |select * from $tableName
        |""".stripMargin)
    dfFromHive.show(10)
  }
}
```

4. **将SparkApplication Jar包和依赖上传至OSS中。**

   详情请参见上传文件。

   > **说明** OSS所在的region和Serverless Spark所在的region需要保持一致。

5. **在DLA Spark中提交作业并进行计算。**
   - 访问Hive，如果您集群中的HDFS是以高可用部署（即您的集群有一个以上Master节点/NameNode），详情请参见创建和执行Spark作业和作业配置指南。

```
{
    "args": [
        "hello_dla"
    ],
    "name": "spark-on-hive",
    "className": "com.aliyun.spark.SparkHive",
    "conf": {
    "spark.sql.catalogImplementation":"hive",
    "spark.dla.eni.vswitch.id": "{您的交换机ID}",
    "spark.dla.eni.security.group.id": "{您的安全组ID}",
    "spark.dla.eni.enable": "true",
    "spark.driver.resourceSpec": "medium",
    "spark.executor.instances": 1,
    "spark.executor.resourceSpec": "medium",
    "spark.dla.job.log.oss.uri": "oss://<指定您存放SparkUI日志的目录/>",
    "spark.hadoop.hive.metastore.uris":"thrift://${ip}:${port},thrift://${ip}:${port}
",
    "spark.hadoop.dfs.nameservices":"{您的nameservices名称}",
    "spark.hadoop.dfs.client.failover.proxy.provider.${nameservices}":"{您的failover p
roxy provider实现类全路径名称}",
    "spark.hadoop.dfs.ha.namenodes.${nameservices}":"{您的nameservices所属namenode列表}
",
    "spark.hadoop.dfs.namenode.rpc-address.${nameservices}.${nn1}":"namenode0所属的ip:
port",
    "spark.hadoop.dfs.namenode.rpc-address.${nameservices}.${nn2}":"namenode1所属的ip:
port"
    },
    "file": "oss://{您的Jar包所属的oss路径}"
}
```

参数说明如下：

参数	说明	备注
spark.hadoop.hive.metastore.uris	配置访问HiveMetaStore的Uri，对应${HIVE_CONF_DIR}/hive-site.xml中的hive.metastore.uris配置项。注意，一般该配置项的值都是域名：端口的形式，用户在serverless spark中配置参数的时候需要将它替换为对应IP+端口的形式。	域名和IP的映射关系，一般可以登录集群的master节点查看本机的/etc/hosts，或者在master节点，直接使用ping+域名的方式获取，您也可以采用步骤2获取对应的配置参数。
spark.dla.eni.vswitch.id	您的交换机ID。	无
spark.dla.eni.security.group.id	您的安全组ID。	无
spark.dla.eni.enable	控制开启或关闭ENI。	无
spark.hadoop.dfs.nameservices	对应hdfs-site.xml中的dfs.nameservices	无
spark.dla.job.log.oss.uri	指定您存放SparkUI日志的OSS目录	无

spark.hadoop.dfs.client.failover.proxy.provider.${nameservices}	对应hdfs-site.xml中的dfs.client.failover.proxy.provider.${nameservices}	无
spark.hadoop.dfs.ha.namenodes.${nameservices}	对应hdfs-site.xml中的dfs.ha.namenodes.${nameservices}	无
spark.hadoop.dfs.namenode.rpc-address.${nameservices}.${nn1/nn2}	对应hdfs-site.xml中的dfs.namenode.rpc-address.${nameservices}.${nn1/nn2}	注意该配置项应该写成IP：端口的形式，用户可以通过用户集群master节点中的/etc/hosts文件查看域名和IP的对应关系或者在master节点，直接使用ping+域名的方式获取，您也可以采用步骤2获取对应的配置参数。

作业运行成功后，单击**操作 > 日志**，查看作业日志。

```
20/08/23 16:20:15 INFO TaskSchedulerImpl: Removed TaskSet 2.0, whose tasks have all
completed, from pool
20/08/23 16:20:15 INFO DAGScheduler: ResultStage 2 (show at SparkHive.scala:28) finished
in 0.096 s
20/08/23 16:20:15 INFO DAGScheduler: Job 2 finished: show at SparkHive.scala:28, took
0.104908 s
+----------------+
|     welcome_col|
+----------------+
|hello, dla-spark|
+----------------+
```

- 访问Hive, 如果您集群中的HDFS是以非高可用部署的（即只有一个Master节点/NameNode）。

```
{
    "args": [
        "hello_dla"
    ],
    "name": "spark-on-hive",
    "className": "com.aliyun.spark.SparkHive",
    "conf": {
        "spark.sql.catalogImplementation":"hive",
        "spark.dla.eni.vswitch.id": "{您的交换机ID}",
        "spark.dla.eni.security.group.id": "{您的安全组ID}",
        "spark.dla.eni.enable": "true",
        "spark.driver.resourceSpec": "medium",
        "spark.executor.instances": 1,
        "spark.executor.resourceSpec": "medium",
        "spark.dla.job.log.oss.uri": "oss://<指定您存放SparkUI日志的目录/>","
        "spark.hadoop.hive.metastore.uris":"thrift://${ip}:${port},thrift://${ip}:${p
ort}",
        "spark.dla.eni.extra.hosts":"${ip0} ${hostname_0} ${hostname_1} ${hostname_n}
"
    },
    "file": "oss://{您的Jar包所属的oss路径}"
}
```

参数	说明	备注

spark.hadoop.hive.metastore.uris	配置访问HiveMetaStore的Uri，对应${HIVE_CONF_DIR}/hive-site.xml中的hive.metastore.uris配置项。注意，一般该配置项的值都是域名+端口的形式，用户在serverless spark中配置参数的时候需要将它替换为对应ip：端口的形式。	域名和IP的映射关系，一般可以登录集群的master节点查看本机的/etc/hosts, 或者在master节点，直接使用ping + 域名的方式获取，用户也可以采用步骤1获取对应的配置参数。
spark.dla.job.log.oss.uri	指定您存放SparkUI日志的OSS目录	无
spark.dla.eni.vswitch.id	您的交换机ID	无
spark.dla.eni.security.group.id	您的安全组ID	无
spark.dla.eni.enable	控制开启或关闭ENI	无
spark.dla.eni.extra.hosts	Spark解析Hive表位置时，需要额外传入IP和表格存储节点host的映射关系，以便Spark能正确解析位置的域名信息。 **注意** IP和域名之间用空格隔开。多个IP 和域名用逗号隔开，如 "ip0 master0, ip1 master1"	该值可从用户集群${Hive_CONF_DIR}/core-site.xml的fs.defaultFS获取。示例用户fs.defaultFs的值为："hdfs://master-1:9000", 则需要配置spark.dla.eni.extra.hosts的值为："${master-1的ip} master-1"。IP和域名的对应关系，您可以登录自建集群的master节点，从/etc/hosts中查看IP和域名的对应关系。您也可以从步骤2中获取相关参数。

6.17. Hadoop

本文主要介绍如何使用DLA Spark访问用户VPC中的HADOOP集群（开启kerberos认证的集群暂不支持）。

前提条件

- 您已开通数据湖分析DLA（Data Lake Analytics）服务。如何开通，请参见开通云原生数据湖分析服务。
- 您已登录云原生数据库分析DLA控制台，在云原生数据湖分析DLA控制台上创建了**Spark虚拟集群**。

- 您已开通对象存储OSS（Object Storage Service）服务。如何开通，请参见开通OSS服务。
- 准备创建Spark计算节点所需要的**交换机id**和**安全组id**，可以选择已有的交换机和安全组，也可以新建交换机和安全组。交换机和安全组需要满足以下条件。
  - **交换机需要与您的Hadoop服务集群在同一VPC下。**可使用您Hadoop集群控制台上的交换机ID。
  - **安全组需要与您的Hadoop服务集群在同一VPC下。**您可以前往ECS控制台-**网络与安全**-**安全组**按照专有网络（VPC）ID搜索该VPC下的安全组，任意选择一个安全组ID即可。

- 如果您的Hadoop服务有**白名单**控制，需要您将交换机网段加入到您Hadoop服务的**白名单**中。

> **注意** 对于Xpack-Spark用户首先联系云X-Pack Spark答疑（钉钉号：dgw-jk1ia6xzp）开通HDFS，由于HDFS的开放可能造成用户的恶意攻击，引起集群不稳定甚至造成破坏。因此XPack-Spark的HDFS功能暂时不直接开放给用户。

操作步骤

1. **获取需要在DLA Spark配置的Hadoop相关参数。**

   > **说明** 如果您的Hadoop服务所在集群无法执行spark作业，可以跳过这步。

   我们提供了工具来读取您Hadoop服务所在集群的配置，您可以按照下面的地址下载 `spark-examples-0.0.1-SNAPSHOT-shaded.jar` 并上传至OSS，然后提交Spark作业到用户的Hadoop服务所在集群上执行，即可在作业输出中获得访问Hadoop所需的配置。

   ```
   wget https://dla003.oss-cn-hangzhou.aliyuncs.com/GetSparkConf/spark-examples-0.0.1-SNAPSHOT-shaded.jar
   ```

   - **EMR用户**将Jar包上传至OSS后，可以通过以下命令提交获取配置作业：

     ```
     --class com.aliyun.spark.util.GetConfForServerlessSpark
     --deploy-mode client
     ossref://{path/to}/spark-examples-0.0.1-SNAPSHOT-shaded.jar
     get hadoop
     ```

     作业运行完毕后，可以通过SparkUI查看driver的stdout输出或者从作业详情中的提交日志中查看输出的配置。

   - **云Hbase-Spark**用户将Jar包上传至资源管理目录后，以用以下命令提交获取配置作业：

     ```
     --class com.aliyun.spark.util.GetConfForServerlessSpark
     /{path/to}/spark-examples-0.0.1-SNAPSHOT-shaded.jar
     get hadoop
     ```

     等待作业完成后，通过SparkUI的driver中的stdout查看输出配置。

   - 其他HADOOP集群，如果您在集群上未设置 `HADOOP_CONF_DIR` 环境变量，则需要手动输入 `HADOOP_CONF_DIR` 路径。

     ```
     --class com.aliyun.spark.util.GetConfForServerlessSpark
     /{path/to}/spark-examples-0.0.1-SNAPSHOT-shaded.jar
     get --hadoop-conf-dir </path/to/your/hadoop/conf/dir> hadoop
     ```

2. **编写访问HDFS的SparkApplication。**

   以下示例代码可以根据传入的HDFS目录信息，来读写HDFS目录，然后把内容展示出来：

```
package com.aliyun.spark
import org.apache.spark.sql.SparkSession
object SparkHDFS {
  def main(args: Array[String]): Unit = {
    val sparkSession = SparkSession
      .builder()
      .appName("Spark HDFS TEST")
      .getOrCreate()
    val welcome = "hello, dla-spark"
    //hdfs目录用于存放内容
    val hdfsPath = args(0)
    //将welcome字符串存入指定的hdfs目录
    sparkSession.sparkContext.parallelize(Seq(welcome)).saveAsTextFile(hdfsPath)
    //从指定的hdfs目录中读取内容，并打印
    sparkSession.sparkContext.textFile(hdfsPath).collect.foreach(println)
  }
}
```

3. **将SparkApplication jar包和依赖上传至OSS中。**

   详情请参见上传文件。

   > **说明** OSS所在的region和Serverless Spark所在的region需要保持一致。

4. **在DLA Spark中提交作业并进行计算。**

   - 如果您的HDFS服务以**非高可用的模式部署**（即只有一个Mater节点/NameNode），详情请参见创建和执行Spark作业和作业配置指南。

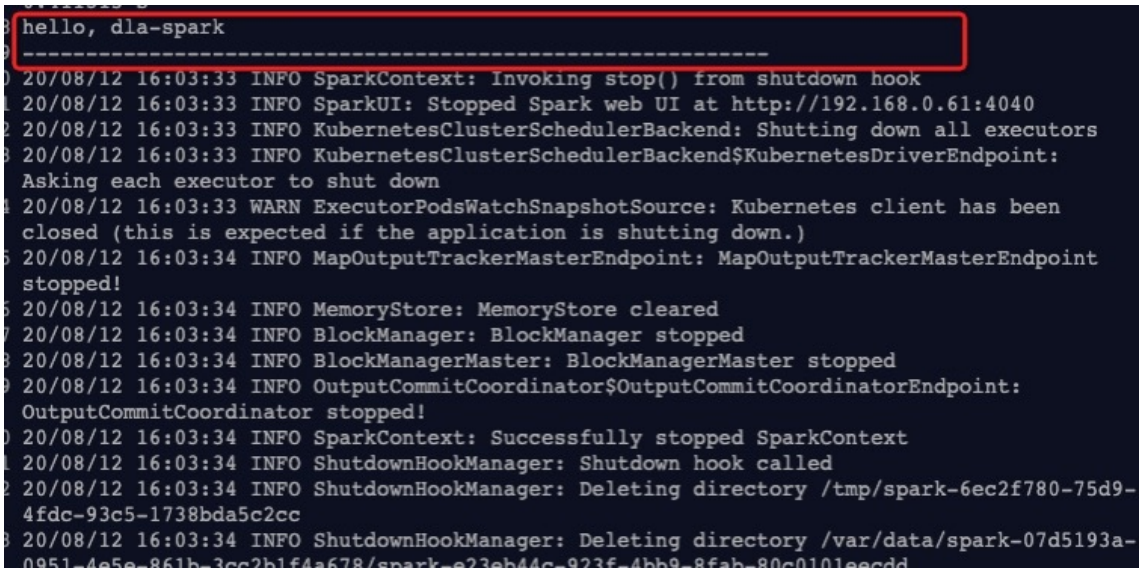
     ```
     {
         "args": [
             "${fs.defaultFS}/tmp/dla_spark_test"
         ],
         "name": "spark-on-hdfs",
         "className": "com.aliyun.spark.SparkHDFS",
         "conf": {
         "spark.dla.eni.enable": "true",
         "spark.dla.eni.vswitch.id": "{您的交换机id}",
         "spark.dla.eni.security.group.id": "{您的安全组id}",
         "spark.dla.job.log.oss.uri": "oss://<指定您存放SparkUI日志的目录/>",
         "spark.driver.resourceSpec": "medium",
         "spark.executor.instances": 1,
         "spark.executor.resourceSpec": "medium"
         },
         "file": "oss://{您的jar包所属的oss路径}"
     }
     ```

     参数说明如下：

参数	说明	备注

参数	说明	备注
fs.defaultFS	您的hdfs配置文件中core-site.xml 中的配置，注意如果fs.defaultFS配置的是机器的域名，需要转换成域名所对应的IP。典型格式 hdfs://${域名对应的ip}:9000/path/to/dir。	用户可以通过登录集群master节点，通过/etc/hosts文件查看域名和IP的对应关系，或者直接采取ping域名的方式获取，或者通过步骤1获取相应配置。
spark.dla.eni.vswitch.id	您的交换机ID。	无
spark.dla.eni.security.group.id	您的安全组ID。	无
spark.dla.eni.enable	控制开启或关闭ENI。	无

作业运行成功后，单击**操作 > 日志**，查看作业日志。



- 如果您的HDFS以**高可用模式部署**（即有一个以上Master节点/NameNode）。

```
{
    "args": [
        "${fs.defaultFS}/tmp/test"
    ],
    "name": "spark-on-hdfs",
    "className": "com.aliyun.spark.SparkHDFS",
    "conf": {
        "spark.dla.eni.enable": "true",
        "spark.dla.eni.vswitch.id": "{您的交换机id}",
        "spark.dla.eni.security.group.id": "{您的安全组id}",
        "spark.driver.resourceSpec": "medium",
        "spark.dla.job.log.oss.uri": "oss://<指定您存放SparkUI日志的目录/>",
        "spark.executor.instances": 1,
        "spark.executor.resourceSpec": "medium",
        "spark.hadoop.dfs.nameservices":"{您的nameservices名称}",
        "spark.hadoop.dfs.client.failover.proxy.provider.${nameservices}":"{您的failover proxy provider实现类全路径名称}",
        "spark.hadoop.dfs.ha.namenodes.${nameservices}":"{您的nameservices所属namenode列表}",
        "spark.hadoop.dfs.namenode.rpc-address.${nameservices}.${nn1}":"namenode0所属的ip:port",
        "spark.hadoop.dfs.namenode.rpc-address.${nameservices}.${nn2}":"namenode1所属的ip:port"
    },
    "file": "oss://{{您的jar包所属的oss路径}"
}
```

参数	说明	备注
spark.hadoop.dfs.nameservices	对应hdfs-site.xml中的dfs.nameservices	无
spark.hadoop.dfs.client.failover.proxy.provider.${nameservices}	对应hdfs-site.xml中的dfs.client.failover.proxy.provider.${nameservices}	无
spark.hadoop.dfs.ha.namenodes.${nameservices}	对应hdfs-site.xml中的dfs.ha.namenodes.${nameservices}	无
spark.hadoop.dfs.namenode.rpc-address.${nameservices}.${nn1/nn2}	对用hdfs-site.xml中的dfs.namenode.rpc-address.${nameservices}.${nn1/nn2}	注意这里应该填写namenode域名对应的ip：port，用户可以通过用户集群master节点中的/etc/hosts文件查看域名和IP的对应关系，或者通过步骤1获取相应的配置。

6.18. HBase

弹性网卡ENI（Elastic Network Interface）是一种可以绑定到专有网络VPC类型ECS实例上的虚拟网卡。通过弹性网卡，您可以实现高可用集群搭建、低成本故障转移和精细化的网络管理。本文主要介绍如何使用Serverless Spark通过ENI访问VPC中的HBase集群。

前提条件

保证Serverless Spark能够访问您的VPC，详情请参见配置数据源网络。

> 注意 ENI交换机和安全组可以使用用户集群已有的ENI交换机和安全组。

操作步骤

1. 将ENI交换机网段添加到HBase服务的白名单/安全组中。

   当您需要访问Xpack-HBase时，可在HBase集群控制台的访问控制中，将ENI交换机所在网段添加到网络白名单。详细操作请参见设置白名单和安全组。

2. 获取需要在Severless Spark配置的参数。

   Xpack-HBase用户可以登录HBase集群控制台，单击**集群列表**，然后单击相应的实例名称获取对应的`ThriftServer Access`。

3. 编写访问HBase的SparkApplication。

   - 示例代码如下所示：

```
package com.aliyun.spark
import org.apache.spark.sql.SparkSession
object SparkHbase {
  def main(args: Array[String]): Unit = {
    //HBase集群的ThriftServer访问链接地址。//HBase集群的ThriftServer访问链接地址。使用时请把此路径替换为你自己的HBase集群的ThriftServer访问地址。
    //格式为：xxx-002.hbase.rds.aliyuncs.com:2181,xxx-001.hbase.rds.aliyuncs.com:2181,xxx-003.hbase.rds.aliyuncs.com:2181
    val zkAddress = args(0)
    //HBase侧的表名，需要在HBase侧提前创建。HBase表创建可以参考使用Java Client访问。
    val hbaseTableName = args(1)
    //Spark侧的表名。
    val sparkTableName = args(2)
    val sparkSession = SparkSession
      .builder()
      //      .enableHiveSupport() //使用enableHiveSupport后通过spark jdbc查看到代码中创建的表
      .appName("scala spark on HBase test")
      .getOrCreate()
    import sparkSession.implicits._
    //如果存在的话就删除表
    sparkSession.sql(s"drop table if exists $sparkTableName")
    val createCmd =
      s"""CREATE TABLE ${sparkTableName} USING org.apache.hadoop.hbase.spark
         |    OPTIONS ('catalog'=
         |    '{"table":{"namespace":"default", "name":"${hbaseTableName}"},"rowkey":"rowkey",
         |    "columns":{
         |    "col0":{"cf":"rowkey", "col":"rowkey", "type":"string"},
         |    "col1":{"cf":"cf", "col":"col1", "type":"String"}}}',
         |    'hbase.zookeeper.quorum' = '${zkAddress}'
         |    )""".stripMargin
    println(s" the create sql cmd is: \n $createCmd")
    sparkSession.sql(createCmd)
    val querySql = "select * from " + sparkTableName + " limit 10"
    sparkSession.sql(querySql).show
  }
}
```

- HBase相关依赖可参考下方 `pom.xml` ：

```
        <dependency>
            <groupId>com.aliyun.apsaradb</groupId>
            <artifactId>alihbase-spark</artifactId>
            <version>1.1.3_2.4.3-1.0.4</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>com.aliyun.hbase</groupId>
            <artifactId>alihbase-client</artifactId>
            <version>1.1.3</version>
            <scope>provided</scope>
            <exclusions>
                <exclusion>
                    <groupId>io.netty</groupId>
                    <artifactId>netty-all</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>com.aliyun.hbase</groupId>
            <artifactId>alihbase-protocol</artifactId>
            <version>1.1.3</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>com.aliyun.hbase</groupId>
            <artifactId>alihbase-server</artifactId>
            <version>1.1.3</version>
            <scope>provided</scope>
            <exclusions>
                <exclusion>
                    <groupId>io.netty</groupId>
                    <artifactId>netty-all</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
```

4. 将SparkApplication JAR包和依赖上传至OSS中。

   详情请参见上传文件。

   > **说明** OSS所在的region和Serverless Spark所在的region需要保持一致。

5. 在Serverless Spark中提交作业并进行计算。

   i. 通过HBase集群的HBase Shell准备数据。

   ```
   bin/hbase shell
   hbase(main):001:0> create 'mytable', 'cf'
   hbase(main):001:0> put 'mytable', 'rowkey1', 'cf:col1', 'this is value'
   ```

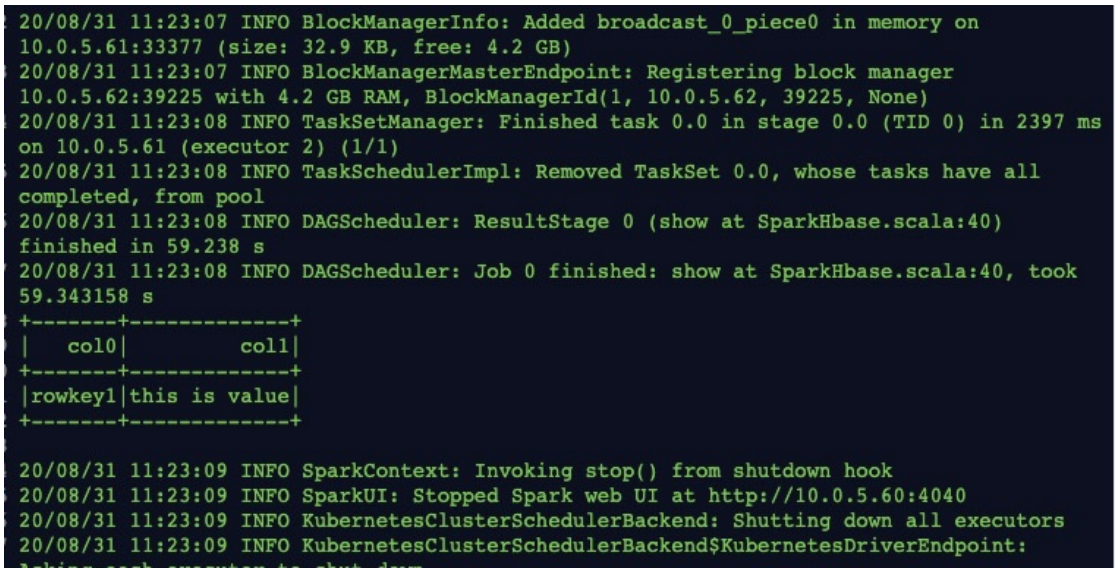
   ii. 在Serverless Spark控制台提交访问HBase的Spark作业。详情请参见创建和执行Spark作业。

```
{
    "args": [
        "xxx:2181,xxx1:2181,xxx2:2181",
        "mytable",
        "spark_on_hbase_job"
    ],
    "name": "spark-on-hbase",
    "className": "com.aliyun.spark.SparkHbase",
    "conf": {
    "spark.dla.eni.vswitch.id": "{您的交换机ID}",
    "spark.dla.eni.security.group.id": "{您的安全组ID}",
    "spark.driver.resourceSpec": "medium",
    "spark.dla.eni.enable": "true",
    "spark.dla.connectors": "hbase",
    "spark.executor.instances": 2,
    "spark.executor.resourceSpec": "medium"
    },
    "file": "oss://{您的JAR包所属的OSS路径}"
}
```

作业参数说明：

参数	说明	备注
xxx:2181,xxx1:2181,xxx2:2181	HBase集群中的ZK链接地址。	可按照第二步方法在HBase控制台获取
mytable	HBase集群中表，本实例使用的HBase表：mytable,使用HBase Shell创建表准备数据	无
spark_on_hbase_job	Spark中创建映射HBase表的表名。	无
spark.dla.connectors	将Serverless Spark内置的读取HBase表格相关JAR包包含到classpath中。	如果用户作业JAR包中不包含读取HBase相关依赖，则需要配置该参数，如果用户作业JAR包中包含了读取HBase的相关依赖，则无需配置该参数。
spark.dla.eni.vswitch.id	您的交换机ID。	无
spark.dla.eni.security.group.id	您的安全组ID。	无
spark.dla.eni.enable	控制开启或关闭ENI。	无

作业运行成功后，单击**操作 > 日志**，查看作业日志。



> **说明** 如果用户需要上传自定义的HBase Connector JAR包，则无需设置 `spark.dla.connectors` 为HBase，直接在 `ConfigJson` 中使用 `jars:["<oss://path/to/your/hbase/connector/jar>"]` 上传HBase依赖JAR包即可。

6.19. Cassandra

云数据库Cassandra是基于开源Apache Cassandra，融合阿里云数据库DBaaS能力的分布式NoSQL数据库。本文主要介绍如何通过DLA Serverless Spark访问云数据库Cassandra。

前提条件

- 已经开通对象存储OSS（Object Storage Service）服务。具体操作请参考开通OSS服务。
- 已经创建云数据库Cassandra实例。具体请参考使用cqlsh访问Cassandra。
- 已获取到Cassandra的**私网连接点**、**CQL端口**、**数据库用户名**、**数据库密码**。具体请参考多语言SDK访问（公网&内网）。
- 在Cassandra实例中已创建数据表，并插入数据。具体请参考使用cqlsh访问Cassandra。参考命令样例如下：

```
CREATE KEYSPACE spark WITH replication = {'class': 'SimpleStrategy', 'replication_factor'
: 1};
use spark;
CREATE TABLE spark_test (first_name text , last_name text, PRIMARY KEY (first_name)) ;
INSERT INTO spark_test (first_name, last_name) VALUES ('hadoop', 'big data basic platefor
m');
INSERT INTO spark_test (first_name, last_name) VALUES ('spark', 'big data compute engine'
);
INSERT INTO spark_test (first_name, last_name) VALUES ('kafka', 'streaming data plateform
');
INSERT INTO spark_test (first_name, last_name) VALUES ('mongodb', 'document database');
INSERT INTO spark_test (first_name, last_name) VALUES ('es', 'serarch egnine');
INSERT INTO spark_test (first_name, last_name) VALUES ('flink', 'streaming plateform');
```

- 准备DLA Spark访问Cassandra实例所需的安全组ID和交换机ID。具体操作请参见配置数据源网络。

操作步骤

1. 准备以下测试代码和依赖包来访问Cassandra，并将此测试代码和依赖包分别编译打包生成jar包上传至您的OSS。

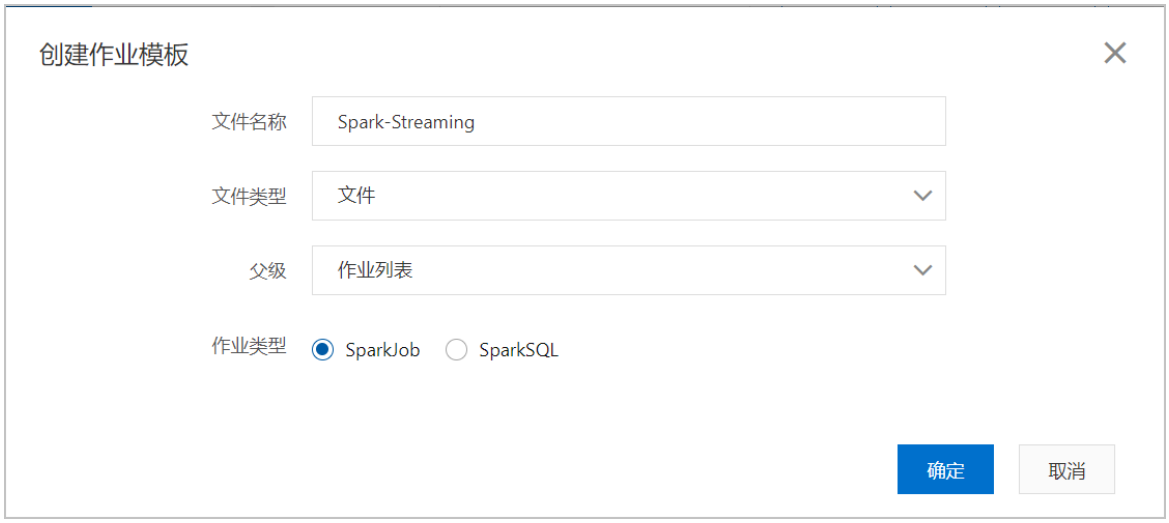
   测试代码示例：

```
import org.apache.spark.sql.SparkSession
object SparkCassandra {
  def main(args: Array[String]): Unit = {
    //cassandra的私网连接点
    val cHost = args(0)
    //Cassandra的CQL端口
    val cPort = args(1)
    //Cassandra的数据库用户名和密码
    val cUser = args(2)
    val cPw = args(3)
    //Cassandra的keystone和table
    val cKeySpace = args(4)
    val cTable = args(5)
    val spark = SparkSession
      .builder()
      .config("spark.cassandra.connection.host", cHost)
      .config("spark.cassandra.connection.port", cPort)
      .config("spark.cassandra.auth.username", cUser)
      .config("spark.cassandra.auth.password", cPw)
      .getOrCreate();
    val cData1 = spark
      .read
      .format("org.apache.spark.sql.cassandra")
      .options(Map("table" -> cTable, "keyspace" -> cKeySpace))
      .load()
    print("=======start to print the cassandra data======")
    cData1.show()
  }
}
```

   Cassandra依赖的pom文件：

```
        <dependency>
            <groupId>com.datastax.spark</groupId>
            <artifactId>spark-cassandra-connector_2.11</artifactId>
            <version>2.4.2</version>
        </dependency>
```

2. 登录Data Lake Analytics管理控制台。
3. 在页面左上角，选择Cassandra实例所在地域。
4. 单击左侧导航栏中的**Serverless Spark > 作业管理**。
5. 在**作业编辑**页面，单击**创建作业**。
6. 在**创建作业模板**页面，按照页面提示进行参数配置后，单击**确定**创建Spark作业。



7. 单击Spark作业名，在Spark作业编辑框中输入以下作业内容，并按照以下参数说明进行参数值替换。保存并提交Spark作业。

```
{
    "args": [
        "cds-xxx-1-core-001.cassandra.rds.aliyuncs.com",  #Cassandra的私网连接点，私网连接点可能会有两个，任选其一即可。
        "9042",  #Cassandra的CQL端口。
        "cassandra",  #Cassandra的数据库用户名。
        "test_1234",  #Cassandra的数据库密码。
        "spark",   #Cassndra的keyspace。
        "spark_test"  #Cassandra的表名。
    ],
    "file": "oss://spark_test/jars/cassandra/spark-examples-0.0.1-SNAPSHOT.jar",  #测试代码的OSS路径。
    "name": "Cassandra-test",
    "jars": [
        "oss://spark_test/jars/cassandra/spark-cassandra-connector_2.11-2.4.2.jar"  #测试代码依赖包的OSS路径。
    ],
    "className": "com.aliyun.spark.SparkCassandra",
    "conf": {
        "spark.driver.resourceSpec": "small",  #表示driver的规格，有small、medium、large、xlarge之分。
        "spark.executor.instances": 2,  #表示executor的个数。
        "spark.executor.resourceSpec": "small",   #表示executor的规格，有small、medium、large、xlarge之分。
        "spark.dla.eni.enable": "true",  #开启访问用户VPC网络的权限。当您需要访问用户VPC网络内的数据时，需要开启此选项。
        "spark.dla.eni.vswitch.id": "vsw-xxx",  #可访问Cassandra 的交换机id。
        "spark.dla.eni.security.group.id": "sg-xxx"  #可访问Cassandra的安全组id。
    }
}
```

执行结果

作业运行成功后，在任务列表中单击**操作 > 日志**，查看作业日志。出现如下日志说明作业运行成功：

```
130 20/12/29 12:54:48 INFO DAGScheduler: ResultStage 2 (show at SparkCassandra.scala:34)
    finished in 0.126 s
131 20/12/29 12:54:48 INFO DAGScheduler: Job 2 finished: show at SparkCassandra.scala:34,
    took 0.131639 s
132 =======start to print the cassandra data=======+----------+--------------------+
133 |first_name|          last_name|
134 +----------+--------------------+
135 |        es|      serarch egnine|
136 |    hadoop|big data basic pl...|
137 |   mongodb|   document database|
138 |     flink| streaming plateform|
139 |     spark|big data compute ...|
140 |     kafka|streaming data pl...|
141 +----------+--------------------+
142
143 20/12/29 12:54:48 INFO SerialShutdownHooks: Successfully executed shutdown hook:
    Clearing session cache for C* connector
```

6.20. RDS

本文主要介绍如何使用Serverless Spark通过ENI访问VPC中的RDS。

前提条件

Serverless Spark需要访问您的VPC，Serverless Spark访问VPC的操作步骤请参见访问用户VPC。

设置RDS的白名单

详细操作步骤请参见如何设置RDS白名单。将ENI所在交换机的网段添加到RDS的白名单中，或者在RDS安全组中将ENI所在安全组添加进去。

在数据库中编写测试数据

详细操作步骤请参见通过DMS登录RDS数据库。测试数据内容如下：

```
CREATE TABLE `persons` (
  `id` int(11) DEFAULT NULL,
  `first_name` varchar(32) DEFAULT NULL,
  `laster_name` varchar(32) DEFAULT NULL,
  `age` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8
;
insert into persons VALUES(1,'a','b',5);
insert into persons VALUES(2,'c','d',6);
insert into persons VALUES(3,'e','f',7);
```

编写访问RDS的SparkApplication

SparkApplication能根据传入的RDS库和表的信息来通过spark访问这个RDS库和表，然后把内容展示出来，例如：

```
package com.aliyun.spark
import org.apache.spark.sql.SparkSession
object SparkRDS {
  def main(args: Array[String]): Unit = {
    val sparkSession = SparkSession.builder()
      .appName("rds test")
      .getOrCreate()
    val url = args(0)
    val dbtable = args(1)
    val user = args(2)
    val password = args(3)
    val jdbcDF = sparkSession.read
      .format("jdbc")
      .option("url", url)
      .option("driver", "com.mysql.jdbc.Driver")
      .option("dbtable", dbtable)
      .option("user", user)
      .option("password", password)
      .load()
    jdbcDF.show()
  }
}
```

上传文件至对象储存OSS中

进行编译打包，把SparkApplication的Jar包和MySQL驱动依赖上传到OSS中，MySQL驱动依赖的下载地址为MySQL驱动依赖下载。

详细操作步骤请参见上传文件。

提交作业

在Serverless Spark中编写Spark-Submit的脚本，详细操作步骤请参见创建和执行Spark作业。脚本内容为：

```
{
    "args": [
        "jdbc:mysql://填写你的RDS的URL",
        "persons",
        "spark",
        "填写你的密码"
    ],
    "name": "changqing-dla-test",
    "jars": [
        "oss://changqing-dla-test/mysql-connector-java.jar"
    ],
    "file": "oss://changqing-dla-test/rds_test.jar",
    "className": "com.aliyun.spark.SparkRDS",
    "conf": {
        "spark.dla.eni.enable": "true",
        "spark.dla.eni.vswitch.id": "填写你在上面选择的交换机的id",
        "spark.dla.eni.security.group.id": "填写上面选择的安全组的id",
        "spark.driver.resourceSpec": "medium",
        "spark.executor.instances": 1,
        "spark.executor.resourceSpec": "medium"
    }
}
```

作业运行成功后单击操作列表中的日志按钮，出现如下图所示内容，表示Spark访问RDS成功。

```
   in 1.676 s
260 20/08/05 19:47:27 INFO DAGScheduler: Job 0 finished: show at SparkRDS.scala:42, took
   1.757446 s
261 +---+----------+-----------+---+
262 | id|first_name|laster_name|age|
263 +---+----------+-----------+---+
264 |  1|         a|          b|  5|
265 |  2|         c|          d|  6|
266 |  3|         e|          f|  7|
267 +---+----------+-----------+---+
268
269 20/08/05 19:47:27 INFO SparkContext: Invoking stop() from shutdown hook
270 20/08/05 19:47:27 INFO SparkUI: Stopped Spark web UI at http://172.16.254.226:4040
271 20/08/05 19:47:27 INFO KubernetesClusterSchedulerBackend: Shutting down all executors
272 20/08/05 19:47:27 INFO KubernetesClusterSchedulerBackend$KubernetesDriverEndpoint: Asking
   each executor to shut down
273 20/08/05 19:47:27 WARN ExecutorPodsWatchSnapshotSource: Kubernetes client has been closed
   (this is expected if the application is shutting down.)
274 20/08/05 19:47:27 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint
   stopped!
275 20/08/05 19:47:27 INFO MemoryStore: MemoryStore cleared
276 20/08/05 19:47:27 INFO BlockManager: BlockManager stopped
277 20/08/05 19:47:27 INFO BlockManagerMaster: BlockManagerMaster stopped
278 20/08/05 19:47:27 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint:
   OutputCommitCoordinator stopped!
279 20/08/05 19:47:27 INFO SparkContext: Successfully stopped SparkContext
280 20/08/05 19:47:27 INFO ShutdownHookManager: Shutdown hook called
281 20/08/05 19:47:27 INFO ShutdownHookManager: Deleting directory /tmp/spark-69ca8aa9-7915-
   4ddf-bc71-49f8e9589d53
282 20/08/05 19:47:27 INFO ShutdownHookManager: Deleting directory /var/data/spark-72b04c72-
```

6.21. ClickHouse

本文主要介绍如何使用DLA Spark访问云ClickHouse。

前提条件

- 您已开通数据湖分析DLA（Data Lake Analytics）服务，详情请参见开通云原生数据湖分析服务并在云原生数据湖分析DLA控制台上创建了**Spark虚拟集群**。

- 您已开通对象存储OSS（Object Storage Service）服务，详情请参见开通OSS服务。
- 您已经开通云数据库ClickHouse, 详情请参见 开通ClickHouse服务。
- 准备Spark计算节点所需要的**交换机id**和**安全组id**，可以选择已有的交换机和安全组，也可以新建交换机和安全组。交换机和安全组需要满足以下条件。
  - **交换机需要与您的ClickHouse在同一VPC下。**可使用ClickHouse控制台上的交换机ID。
  - **安全组需要与您的ClickHouse在同一VPC下。**您可以前往ECS控制台-**网络与安全-安全组**按照专有网络(VPC)ID搜索该VPC下的安全组，任意选择一个安全组ID即可。
  - 将第二步选定的交换机网段加入**ClickHouse**的白名单。

操作步骤

1. **准备测试数据，命名为ck.csv上传至OSS。**

```
name,age
fox,18
tiger,20
alice,36
```

2. **准备以下读写ClickHouse的代码，下面的代码是读取OSS的CSV文件，写入到新建的ClickHouse表中，然后从ClickHouse表中读取数据，打印到控制台。**
   - **参考POM文件**

```
<dependencies>
    <dependency>
        <groupId>ru.yandex.clickhouse</groupId>
        <artifactId>clickhouse-jdbc</artifactId>
        <version>0.2.4</version>
    </dependency>
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-sql_2.11</artifactId>
        <version>2.4.5</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

   - **参考代码片段。**完整代码请参考代码。

```
package com.aliyun.spark
import java.sql.{Connection, DriverManager}
import java.util.Properties
import org.apache.spark.sql.execution.datasources.jdbc.JDBCOptions
import org.apache.spark.sql.{SaveMode, SparkSession}
object SparkWriteToCK {
  val ckProperties = new Properties()
  val url = "jdbc:clickhouse://<您的clickhouse VPC 地址>:8123/default"
  ckProperties.put("driver", "ru.yandex.clickhouse.ClickHouseDriver")
  ckProperties.put("user", "<用户名>")
  ckProperties.put("password", "<密码>")
  ckProperties.put("batchsize","100000")
  ckProperties.put("socket_timeout","300000")
  ckProperties.put("numPartitions","8")
  ckProperties.put("rewriteBatchedStatements","true")
  //创建ClickHouse表
  def createCKTable(table: String): Unit ={
    Class.forName(ckProperties.getProperty("driver"))
    var conn : Connection = null
    try {
      conn = DriverManager.getConnection(url, ckProperties.getProperty("user"), ckPro
perties.getProperty("password"))
      val stmt = conn.createStatement()
      val sql =
        s"""
           |create table if not exists default.${table}  on cluster default(
           |    `name` String,
           |    `age`  Int32)
           |ENGINE = MergeTree() ORDER BY `name` SETTINGS index_granularity = 8192;
           |""".stripMargin
      stmt.executeQuery(sql)
    } finally {
      if(conn != null)
        conn.close()
    }
  }
  def main(args: Array[String]): Unit = {
    val table = "ck_test"
    //使用jdbc建ClickHouse表
    createCKTable(table)
    val spark = SparkSession.builder().getOrCreate()
    //将csv的数据写入ClickHouse
    val csvDF = spark.read.option("header","true").csv("oss://<path/to>/ck.csv").toDF
("name", "age")
    csvDF.printSchema()
    csvDF.write.mode(SaveMode.Append).option(JDBCOptions.JDBC_BATCH_INSERT_SIZE, 1000
00).jdbc(url, table, ckProperties)
    //读ck表数据
    val ckDF = spark.read.jdbc(url, table, ckProperties)
    ckDF.show()
  }
}
```

3. **将以上代码打包编译上传至OSS。**

4. 进入DLA控制台，选择**Serverless Spark > 作业管理**，单击**创建作业模板**，选择前提条件中创建的虚拟集群。配置作业运行参数后点击**执行**。

```
{
    "file": "oss://<path/to/your/jar>",
    "name": "SparkWriteToCK",
    "className": "com.aliyun.spark.SparkWriteToCK",
    "conf": {
        "spark.driver.resourceSpec": "medium",
        "spark.executor.instances": 5,
        "spark.executor.resourceSpec": "medium",
        "spark.dla.job.log.oss.uri": "oss://<指定您存放SparkUI日志的目录/>",
        "spark.dla.connectors": "oss",
        "spark.dla.eni.enable": "true",
        "spark.dla.eni.security.group.id": "<您前提条件中选定的安全组id>",
        "spark.dla.eni.vswitch.id": "<您前提条件中选定的交换机id>"
    }
}
```

> **说明** 更多配置说明，请参考Spark配置指南。

5. 作业运行后，可在作业界面查看作业**日志**和**SparkUI**。

6.22. Tablestore

本文介绍了如何使用DLA Spark 访问Tablestore。

前提条件

- 已经创建了Spark虚拟集群。具体操作请参见创建虚拟集群。
- 已经开通对象存储OSS（Object Storage Service）服务。具体操作请参见开通OSS服务。

操作步骤

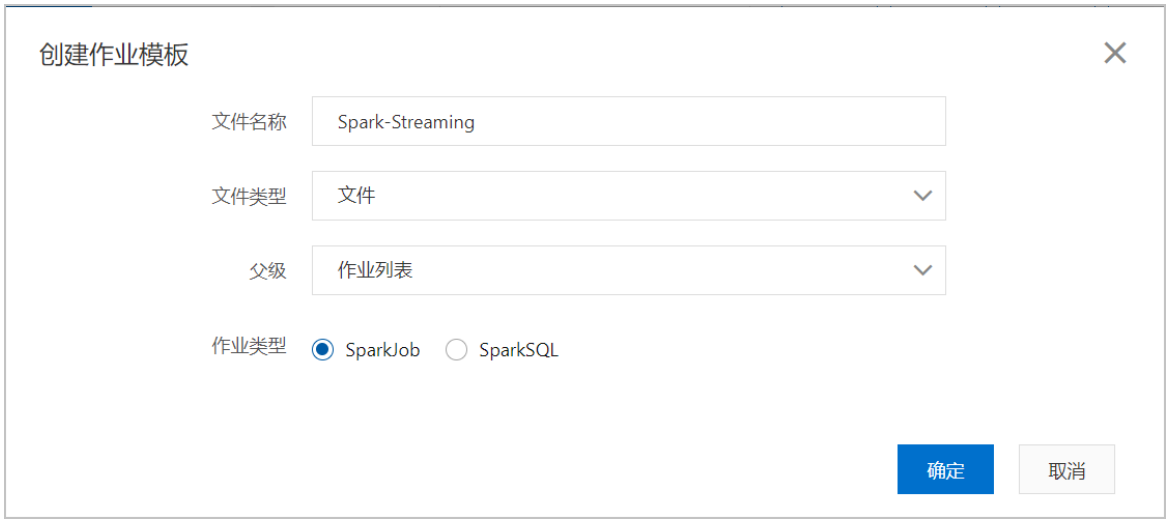
1. 准备以下测试代码来连接Tablestore，并将测试代码打包成JAR包上传至您的OSS。

```
package com.aliyun.spark
import org.apache.spark.SparkConf
import org.apache.spark.sql.types.{DataType, IntegerType, StringType, StructField, Stru
ctType}
import org.apache.spark.sql.{Row, SaveMode, SparkSession}
import scala.collection.mutable
//批计算谓词下推配置
object SparkTablestore {
  def main(args: Array[String]): Unit = {
    if (args.length < 5) {
      System.err.println(
        """Usage: SparkTablestore <instance-id> <table-name> <实例访问地址-VPC> <ACCESS_K
EY_ID>
          |         <ACCESS_KEY_SECRET>
        """.stripMargin)
      System.exit(1)
    }
    val instanceId = args(0)
```

```
val tableName = args(1)
val endpoint = args(2)
val accessKeyId = args(3)
val accessKeySecret = args(4)
//表结构，您需要在Tablestore中准备具有如下表结构的表。
val catalog =
  """
    |{
    |  "columns": {
    |    "id":{
    |      "col":"id",
    |      "type":"string"
    |    },
    |     "company": {
    |      "col": "company",
    |      "type": "string"
    |    },
    |    "age": {
    |      "col": "age",
    |      "type": "integer"
    |    },
    |
    |    "name": {
    |      "col": "name",
    |      "type": "string"
    |    }
    |  }
    |}
    |""".stripMargin
val sparkConf = new SparkConf
val options = new mutable.HashMap[String, String]()
//您的Tablestore实例名称。
options.put("instance.name", instanceId)
//您想要连接的表名。
options.put("table.name", tableName)
//您的访问的endpoint
options.put("endpoint", endpoint)
//您的ACCESS KEY。
options.put("access.key.id", accessKeyId)
//您的ACCESS KEY SECRET。
options.put("access.key.secret", accessKeySecret)
//您的表格结构，使用JSON表达式。
options.put("catalog", catalog)
//与Long类型做Range( >= > < <= )比较的谓词是否下推。
options.put("push.down.range.long", "true")
//与String类型做Range( >= > < <= )比较的谓词是否下推。
options.put("push.down.range.string", "true")
// Tablestore通道Channel在每个Spark Batch周期内同步的最大数据条数，默认10000。
options.put("maxOffsetsPerChannel", "10000")
//多元索引名，可选。
//options.put("search.index.name", "<index_name>")
//tunnel id，可选。
//options.put("tunnel.id", "<tunnel id>")
val spark = SparkSession.builder.config(sparkConf).appName("Serverless Spark Tables
```

```
tore Demo").getOrCreate
    import spark.implicits._
    val dfReader = spark.read.format("Tablestore").options(options)
    val df = dfReader.load()
    //显示表内容。
    df.show()
    df.printSchema()
    //写表。
    val schema = StructType.apply(Seq(StructField("id", StringType), StructField("compa
ny", StringType),StructField("age", IntegerType), StructField("name", StringType)))
    val newData = spark.sparkContext.parallelize(Seq(("1","ant",10,"xxx"))).map(row =>
Row(row._1, row._2, row._3, row._4))
    val newDataDF = spark.createDataFrame(newData, schema)
    newDataDF.write.format("Tablestore").options(options).save
    val dfReader1 = spark.read.format("Tablestore").options(options)
    val df1 = dfReader1.load()
    df1.show()
  }
}
```

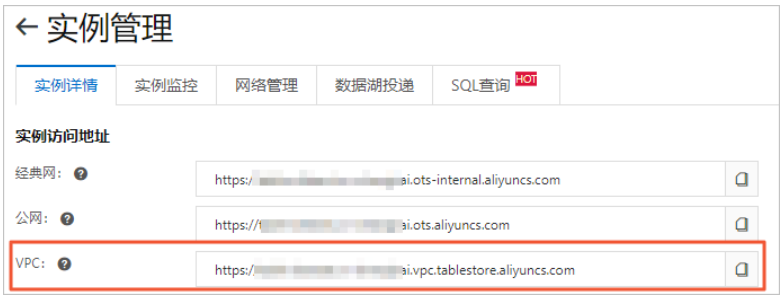
2. 登录Data Lake Analytics管理控制台。
3. 在页面左上角，选择Tablestore所在地域。
4. 单击左侧导航栏中的**Serverless Spark > 作业管理**。
5. 在**作业编辑**页面，单击**创建作业模板**。
6. 在**创建作业模板**页面，按照页面提示进行参数配置后，单击**确定**创建Spark作业。



7. 单击Spark作业名，在Spark作业编辑框中输入以下Spark Streaming任务内容。保存并提交Spark作业。

```
{
    "args": [
        "<instanceId>",
        "<tableName>",
        "<endpoint>",
        "<access key id>",
        "<access key secret>"
    ],
    "name": "Tablestore",
    "className": "com.aliyun.spark.SparkTablestore",
    "conf": {
        "spark.driver.resourceSpec": "medium",
        "spark.dla.connectors": "oss",
        "spark.executor.instances": 1,
        "spark.dla.job.log.oss.uri": "oss://</path/to/store/your/spark/log>",
        "spark.executor.resourceSpec": "medium"
    },
    "file": "oss://path/to/spark-examples-0.0.1-SNAPSHOT-shaded.jar"
}
```

上述代码中出现的参数说明如下。

参数名称	说明
instanceId	Tablestore的实例名称。
tableName	Tablestore的表名。
endpoint	Tablestore的endpoint，可在Tablestore控制台上查看，选择其中的VPC访问地址。 
access key id	访问Tablestore所需的AccessKey ID。
access key secret	访问Tablestore所需的AccessKey Secret。
spark.dla.job.log.oss.uri	存放Spark日志的路径。

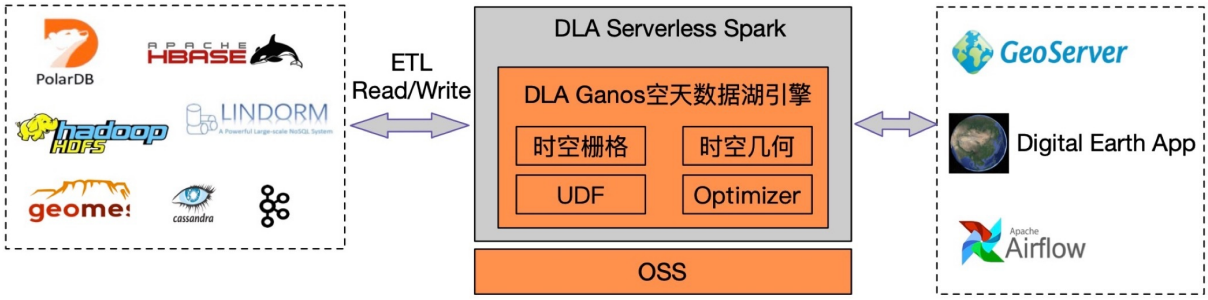
7.数据湖时空引擎Ganos

7.1. 产品简介

DLA Ganos是基于云原生数据湖分析（Data Lake Analytics，DLA）系统设计开发的，面向时空大数据存储与计算的数据引擎产品。基于DLA无服务器化（Serverless）数据湖分析服务与内置的Spark计算引擎，DLA Ganos打通了阿里云各个存储系统，如PolarDB、Lindorm(HBase)、OSS等，通过统一的时空数据模型与计算接口，实现对多源异构数据的一体化管理与计算，并支持进行异构数据源关联分析等复杂运算。此外，DLA无服务（Serverless）架构使DLA Ganos完全按需使用，只需为您运行的查询付费，资源伸缩方便，无感知升级，显著降低了运营成本。DLA详情请参见什么是云原生数据湖分析。

使用场景

DLA Ganos的最基本的应用场景就是将存储在不同的数据库系统或文件系统中的时空数据，通过ETL操作，实现数据源之间的数据流转与协同分析。如下图所示，用户通过DLA Ganos可以加载OSS上的GeoTiff文件为RDD模型，然后写入Lindorm（HBase）等存储系统实现数据归档，同时也可以同时加载多个数据源（PolarDB或Lindorm）的时空数据，进行清洗转换，并通过机器学习等工具进行分析计算，最终将分析结果写回到任意数据源中。最后，通过专业的时空数据服务发布系统（如GeoServer）将计算的结果发布为OGC标准服务供用户进行查询浏览。



产品优势

- **高性价比**

  依托于数据湖分析DLA的Serverless无服务器化架构，用户在使用DLA Ganos时无需基础设施和管理成本，不需要单独维护Spark实例，只需要申请虚拟集群后即可随时随用、按需付费。零启动时间，透明升级、QoS弹性服务等。

- **数据库体验**

  DLA Ganos基于Spark SQL设计开发了一系列针对空间数据分析的用户API，内置了大量基本时空UDF算子，用户可以像操作关系型数据库那样通过SQL处理海量时空数据，方便灵活。

- **时空数据统一建模**

  DLA Ganos基于Spark RDD设计开发了统一的时空数据模型，方便对各类时空数据进行建模。用户不再需要关注不同类型时空数据的处理，只需要将重点放在业务逻辑之中，将复杂的数据加载与模型转化任务交给DLA Ganos进行处理。

- **异构数据源**

  DLA Ganos支持多路数据源接入分析，提供了多样化、异构的数据源分析能力。客户不仅能够对阿里云OSS、PolarDB、Lindorm（HBase）中的数据进行分析，还能将这两者之间的数据进行关联性分析，解决了客户需要将不同种类的数据进行联合分析的问题。

开通DLA Ganos

1. 创建虚拟集群，详情请参见虚拟集群管理。
2. 在创建好的虚拟集群右侧单击**详情**，进入集群详情页面。
3. 集群详情页面，**集群属性 > 版本**处选择**spark_dla_ganos**即可开通DLA Ganos服务。

7.2. 地理空间分析（Geometry）

7.2.1. 基本概念

本文主要介绍时空几何的一些基本概念。

时空几何

DLA Ganos中所称的时空几何，其范畴包含以下几方面：

- 时空几何对象。
  - 矢量数据，如点、线、面状要素。
  - 在矢量数据基础上结合时间属性，组成的时空数据（或时空轨迹数据）。
- 针对时空几何对象的相关操作，如时空关系判断。

时空索引

DLA Ganos能够提供优异的查询性能，其背后的机制在于在底层NoSQL数据库中基于空间填充曲线（Space Filling Curve）模型为时空数据建立了高效的时空索引。例如在Lindorm（HBase）中，时空索引以Rowkey形式存在，详情请参见创建索引表。

时空关系

所谓的时空关系是指两个时空几何对象之间的时间&空间的相对位置。典型的时空关系包括：相交、相离、覆盖、包含等。 在现实场景中，常说的「地理围栏判断」是指一个面状要素表示的地理围栏与目标对象（点、线、面）之间的关系。如果目标对象在地理围栏之内，则称之为包含；在面状要素之外，则称之为相离。

OGC

OGC全称是开放地理空间信息联盟（Open Geospatial Consortium），是一个非盈利的国际标准组织，它制定了数据模型和相关操作的一系列标准，GIS厂商按照这个标准进行开发可保证空间数据的互操作。

GeoTools

DLA Ganos的SDK是基于GeoTools实现的。GeoTools是一个遵循OGC标准，用于处理地理空间数据的工具包，实现了OGC标准的数据模型和接口，很多地理工具都基于GeoTools开发，详情请参见https://geotools.org/

Geometry

在OGC的定义中，Geometry用来表示一个空间对象，例如空间点对象、空间线对象、空间面对象。Geometry只包含空间对象的位置信息，并不包含其附带的属性信息。 GeoTools提供了工具来帮助构建Geometry，因此在HBase Ganos开发过程中，可以较为方便的使用。

SimpleFeature

简单要素。 通俗来讲，SimpleFeature包含Geometry以及其他属性信息。通常所说的一个轨迹点就是一个SimpleFeature，包含了该轨迹点的空间位置、时间信息以及其他属性信息，其中时间信息也是作为属性信息的一部分。

CQL&ECQL

CQL全称为Common Query Language，是OGC为方便地理服务的查询而定义的查询语言。ECQL全称Extended Common Query Language，是CQL的扩展版，比CQL更强大。 一般来说，ECQL更多的是定义filter，类似于SQL语言的where子句，通过文本描述的方式来筛选出目标对象。参考「查询时空对象」一章。 在本手册中，为了简单起见，所说的CQL就是ECQL。

WKT

WKT全称Well-known text，是OGC定义的一种用文本来描述空间对象的格式。例如点就可以写成 `POINT(0,0)` ，这在查询语句中经常使用，也容易理解，CQL&ECQL中也是用WKT来表示空间对象的。WKT的详细规范请参见：http://www.opengeospatial.org/standards/wkt-crs

WKB

WKB全称Well-known Binary，是OGC定义的一种通过序列化字节来描述几何对象的格式。与WKT相比，其优点在于数据较小，适宜传输。GeoTools提供了工具可用于WKB与WKT之间的转换。

DLA Ganos时空几何模型

DLA Ganos中矢量数据模型采用OGC协议标准中SimpleFeature数据模型，其中的Geometry对象包含Point、LineString、Polygon、MultiPoint、MultiLineString和MultiPolygon等要素。Geometry以GeometryUDT的形式被Spark加载并参与运算。

Ganos目前支持的矢量数据源包括：

- PolarDB
- Lindorm（HBase）
- HDFS
- GeoMesa

7.2.2. 快速开始

本文主要介绍在DLA中如何使用时序时空引擎Ganos进行时空几何（Geometry）数据分析。

DLA Ganos中的时空几何，其范畴包含以下几个方面：

- 时空几何对象
  - 矢量数据，如点、线、面状要素。
  - 在矢量数据基础上结合时间属性，组成的时空数据或时空轨迹数据。
- 针对时空几何对象的相关操作，如空间关系判断（相交、相邻、包含等）。

DLA Ganos中矢量数据模型采用OGC协议标准中的SimpleFeature数据模型，其中的Geometry对象包含Point、LineString、Polygon、MultiPoint、MultiLineString和MultiPolygon等要素。Geometry以GeometryUDT的形式被Spark加载并参与运算。

Ganos目前支持的矢量数据源包括：

- Hive
- PolarDB

- Lindorm(HBase)
- GeoMesa
- PostGIS

操作步骤

1. 获取SDK。

   请提交工单或联系阿里云专家团队获取DLA Ganos SDK用于本机编译调试。如何获取专家服务，请参考专家服务。

2. 搭建开发环境，并准备测试数据。

   获取SDK后，就可以开始部署开发环境并构建第一个Ganos Spark任务。首先，您需要通过以下链接获取测船载自动识别系统（AIS）样例数据：demo.csv。

   ```
   sid;date;longitude;latitude;course;speed;sourceid;heading;rot;messaged;status
   205003001;2018-09-03 09:15:50;72.96675666666667;11.982751666666667;130.0;13.1;17;129.0;0.0;1;在航
   205003001;2018-09-03 11:10:17;73.27778666666667;11.70024;133.0;13.1;17;131.0;0.0;1;在航
   205003001;2018-09-03 11:12:20;73.28349333333334;11.695146666666666;133.0;13.1;17;131.0;0.0;1;在航
   205003001;2018-09-03 12:45:25;73.54346666666666;11.469733333333334;134.0;13.2;17;133.0;0.0;1;在航
   205003001;2018-09-03 14:21:37;73.75408;11.182;144.8;14.0;17;145.0;0.0;1;在航
   205003001;2018-09-03 14:21:45;73.75434666666666;11.1816;144.8;14.0;17;145.0;0.0;1;在航
   205003001;2018-09-03 16:49:06;74.06773333333334;10.7524;146.0;12.8;17;143.0;0.0;1;在航
   205003001;2018-09-03 17:55:34;74.20368;10.555546666666666;146.0;12.8;17;143.0;0.0;1;在航
   205003001;2018-09-03 22:29:56;74.76798666666667;9.754333333333333;142.0;12.6;17;141.0;0.0;1;在航
   ...
   ```

   获取测试数据后，请上传到OSS目录或服务器，生成可访问的URI，如：https://bucket名称.oss-cn-beijing.aliyuncs.com/xxx/data.csv。

   然后在IDE中新建Maven工程`dla-ganos-quickstart`，并在*pom.xml*文件中的`<dependencies>`标签中添加如下依赖：

```
<dependency>
     <groupId>com.aliyun.ganos</groupId>
     <artifactId>ganos-spark-sdk</artifactId>
     <version>1.0</version>
     <scope>system</scope>
     <systemPath>获取的DLA-Ganos-SDK包的uri</systemPath>
     </dependency>
<!-- GeoTools deps -->
<dependency>
     <groupId>org.geotools</groupId>
     <artifactId>gt-referencing</artifactId>
     <version>22.0</version>
</dependency>
<dependency>
     <groupId>org.geotools</groupId>
     <artifactId>gt-epsg-hsql</artifactId>
       <version>22.0</version>
</dependency>
<!-- provided deps -->
<dependency>
     <groupId>org.apache.spark</groupId>
     <artifactId>spark-core_2.11</artifactId>
     <version>2.4.3</version>
</dependency>
<dependency>
     <groupId>org.apache.spark</groupId>
     <artifactId>spark-sql_2.11</artifactId>
       <version>2.4.3</version>
</dependency>
<dependency>
     <groupId>org.apache.curator</groupId>
     <artifactId>curator-client</artifactId>
     <version>4.3.0</version>
</dependency>
<dependency>
     <groupId>org.apache.curator</groupId>
     <artifactId>curator-framework</artifactId>
     <version>4.3.0</version>
</dependency>
<dependency>
     <groupId>org.apache.curator</groupId>
     <artifactId>curator-recipes</artifactId>
     <version>4.3.0</version>
</dependency>
```

至此，开发环境搭建完成。

3. 数据加载与处理。

   创建*QuickStart.scala*文件，导入以下程序包：

```
import org.apache.log4j.{Level, Logger}
import com.aliyun.ganos.spark.GanosSparkKryoRegistrator
import com.aliyun.ganos.spark.jts._
import com.aliyun.ganos.spark.SparkUtils._
import com.aliyun.ganos.spark.utils._
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.{DataFrame, Row, SQLContext, SparkSession}
import org.apache.spark.sql.functions.col
import org.locationtech.geomesa.utils.io.WithStore
import org.geotools.util.factory.Hints
import org.geotools.data.DataStore
import org.opengis.feature.simple.{SimpleFeature, SimpleFeatureType}
```

然后，初始化 `SparkSession` 对象，并注册Kryo序列化工具 `GanosSparkKryoRegistrator` ：

```
val spark = SparkSession.builder
    .appName("Simple Application")
    .config("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
    .config("spark.sql.crossJoin.enabled", "true")
    .config("spark.kryo.registrator", classOf[GanosSparkKryoRegistrator].getName)
    .getOrCreate()
import spark.implicits._
//SparkSession加载JTS包用于处理时空数据。
spark.withJTS
val sc = spark.sparkContext
```

接着，通过spark的csv接口加载CSV文件，并调用 `st_makePoint` 函数构建时空对象。

> **说明** 这里使用了 `spark context` 的 `sc.addFile` 接口首先将文件下载到集群，然后才能进行访问。

```
sc.addFile("https://bucket名称.oss-cn-beijing.aliyuncs.com/xxx/data.csv")
val result = spark.read.format("csv")
    .option("header", "true")
    .option("delimiter", ";")
    .option("inferSchema", "true")
    .load(org.apache.spark.SparkFiles.get("data.csv"))
//调用st_makePoint函数构建时空对象。
val ais = result.withColumn("geom", st_makePoint(col("longitude"), col("latitude")))
ais.show            //打印DataFrame内容。
ais.printSchema     //打印DataFrame Schema。
```

输出结果如下：

```
+---------+-------------------+-----------------+------------------+------+-----+------
--+-------+---+---------+------+--------------------+
|      sid|               date|        longitude|          latitude|course|speed|source
id|heading|rot|messaged|status|                geom|
+---------+-------------------+-----------------+------------------+------+-----+------
--+-------+---+---------+------+--------------------+
|205003001|2018-09-03 09:15:50|72.96675666666667|11.982751666666667| 130.0| 13.1|
17|  129.0|0.0|        1|  在航|POINT (72.9667566...|
|205003001|2018-09-03 11:10:17|73.27778666666667|          11.70024| 133.0| 13.1|
```

```
17|  131.0|0.0|        1|  在航|POINT (73.2777866...|
|205003001|2018-09-03 11:12:20|73.28349333333334|11.695146666666666| 133.0| 13.1|
17|  131.0|0.0|        1|  在航|POINT (73.2834933...|
|205003001|2018-09-03 12:45:25|73.54346666666666|11.469733333333334| 134.0| 13.2|
17|  133.0|0.0|        1|  在航|POINT (73.5434666...|
|205003001|2018-09-03 14:21:37|         73.75408|            11.182| 144.8| 14.0|
17|  145.0|0.0|        1|  在航|POINT (73.75408 1...|
|205003001|2018-09-03 14:21:45|73.75434666666666|           11.1816| 144.8| 14.0|
17|  145.0|0.0|        1|  在航|POINT (73.7543466...|
|205003001|2018-09-03 16:49:06|74.06773333333334|           10.7524| 146.0| 12.8|
17|  143.0|0.0|        1|  在航|POINT (74.0677333...|
|205003001|2018-09-03 17:55:34|         74.20368|10.555546666666666| 146.0| 12.8|
17|  143.0|0.0|        1|  在航|POINT (74.20368 1...|
|205003001|2018-09-03 22:29:56|74.76798666666667| 9.754333333333333| 142.0| 12.6|
17|  141.0|0.0|        1|  在航|POINT (74.7679866...|
|205003001|2018-09-04 00:07:58|74.96557333333334|           9.46616| 145.0| 12.8|
17|  145.0|0.0|        1|  在航|POINT (74.9655733...|
|205003001|2018-09-04 00:35:11|75.02181333333333|           9.38648| 145.0| 12.9|
17|  145.0|0.0|        1|  在航|POINT (75.0218133...|
|205003001|2018-09-04 00:41:32|75.03477333333333|           9.36856| 145.2| 14.0|
17|  145.0|0.0|        1|  在航|POINT (75.0347733...|
|205003001|2018-09-04 00:41:51|75.03549333333333| 9.367653333333333| 145.2| 14.0|
17|  145.0|0.0|        1|  在航|POINT (75.0354933...|
|205003001|2018-09-04 00:42:02|75.03581333333334| 9.367146666666667| 145.0| 12.8|
17|  145.0|0.0|        1|  在航|POINT (75.0358133...|
|205003001|2018-09-04 00:49:42|75.05202666666666| 9.344533333333333| 144.0| 12.8|
17|  145.0|0.0|        1|  在航|POINT (75.0520266...|
|205003001|2018-09-04 02:30:35|75.25965333333333| 9.049493333333333| 146.0| 13.2|
17|  147.0|0.0|        1|  在航|POINT (75.2596533...|
|205003001|2018-09-04 02:30:45|            75.26| 9.048986666666666| 146.0| 13.2|
17|  147.0|0.0|        1|  在航|POINT (75.26 9.04...|
|205003001|2018-09-04 10:54:37|76.46938666666667| 7.623226666666667| 141.0| 13.7|
17|  139.0|0.0|        1|  在航|POINT (76.4693866...|
|205003001|2018-09-04 10:54:59|         76.47024|           7.62216| 141.0| 13.7|
17|  139.0|0.0|        1|  在航|POINT (76.47024 7...|
|205003001|2018-09-04 10:55:59|76.47261333333333|            7.6192| 142.0| 13.7|
17|  139.0|0.0|        1|  在航|POINT (76.4726133...|
+---------+-------------------+-----------------+------------------+------+-----+------
--+-------+---+---------+------+--------------------+
only showing top 20 rows
root
 |-- sid: integer (nullable = true)
 |-- date: timestamp (nullable = true)
 |-- longitude: double (nullable = true)
 |-- latitude: double (nullable = true)
 |-- course: double (nullable = true)
 |-- speed: double (nullable = true)
 |-- sourceid: integer (nullable = true)
 |-- heading: double (nullable = true)
 |-- rot: double (nullable = true)
 |-- messaged: integer (nullable = true)
 |-- status: string (nullable = true)
 |-- geom: point (nullable = true)
```

可以看到输出的Schema中新增了一列 `geom` ，类型为 `point` 。该输出结果就是通过 `st_makePoint` 方法构造出的时空对象。下一步可以对 `geom` 列进行各类时空相关的查询分析操作，如使用Spark SQL对 `geom` 字段进行时空查询过滤。

```
ais.createOrReplaceTempView("ais")
val query = spark.sql(
    "SELECT * FROM ais WHERE " +
      "st_contains(st_geomfromtext('POLYGON((73.0 8.5,75 8.5,75 13.5,73.0 13.5,73.0 8.5
))'), geom) AND " +
      "date>=to_timestamp('2018-09-03') AND date <=to_timestamp('2018-10-05') order by
date")
query.show()
println(s"count=$query.count()")
```

输出结果如下：

```
+---------+-------------------+-----------------+------------------+------+-----+------
--+-------+---+---------+------+--------------------+
|      sid|               date|        longitude|          latitude|course|speed|source
id|heading|rot|messaged|status|                geom|
+---------+-------------------+-----------------+------------------+------+-----+------
--+-------+---+---------+------+--------------------+
|205003001|2018-09-03 11:10:17|73.27778666666667|          11.70024| 133.0| 13.1|
17|  131.0|0.0|        1|  在航|POINT (73.2777866...|
|205003001|2018-09-03 11:12:20|73.28349333333334|11.695146666666666| 133.0| 13.1|
17|  131.0|0.0|        1|  在航|POINT (73.2834933...|
|205003001|2018-09-03 12:45:25|73.54346666666666|11.469733333333334| 134.0| 13.2|
17|  133.0|0.0|        1|  在航|POINT (73.5434666...|
|205003001|2018-09-03 14:21:37|         73.75408|            11.182| 144.8| 14.0|
17|  145.0|0.0|        1|  在航|POINT (73.75408 1...|
|205003001|2018-09-03 14:21:45|73.75434666666666|           11.1816| 144.8| 14.0|
17|  145.0|0.0|        1|  在航|POINT (73.7543466...|
|205003001|2018-09-03 16:49:06|74.06773333333334|           10.7524| 146.0| 12.8|
17|  143.0|0.0|        1|  在航|POINT (74.0677333...|
|205003001|2018-09-03 17:55:34|         74.20368|10.555546666666666| 146.0| 12.8|
17|  143.0|0.0|        1|  在航|POINT (74.20368 1...|
|205003001|2018-09-03 22:29:56|74.76798666666667| 9.754333333333333| 142.0| 12.6|
17|  141.0|0.0|        1|  在航|POINT (74.7679866...|
|205003001|2018-09-04 00:07:58|74.96557333333334|           9.46616| 145.0| 12.8|
17|  145.0|0.0|        1|  在航|POINT (74.9655733...|
+---------+-------------------+-----------------+------------------+------+-----+------
--+-------+---+---------+------+--------------------+
count=9
```

可以看到通过 `st_contains` 时空过滤，获取到9条符合条件的记录。

4. 登录Data Lake Analytics管理控制台提交Spark作业。具体请参见创建和执行Spark作业。

   编译工程并打包：

   ```
   mvn clean package
   ```

更多操作

您还可以对时空数据对象进行如下操作：

- 更多时空函数介绍与使用方法请参考时空几何函数参考。
- 数据入库到Lindorm（HBase）并创建时空索引请参考GeoMesa(HBase/Cassandra)。

7.2.3. 连接数据源

7.2.3.1. Hive

本实例展示如何将Hive数据加载到DLA Ganos进行分析。

Hive是Hadoop生态系统中的一个被广泛使用的数据仓库工具，主要用来进行Hadoop中的大规模数据的提取、转化、加载、查询和分析等操作。Hive数据仓库工具能将存储在HDFS系统中的结构化的数据文件映射为一张数据库表，并提供SQL查询功能，能将SQL语句转变成Map/Reduce任务来执行。

操作步骤

1. 初始化Spark。

```
 val spark: SparkSession = SparkSession.builder()
      .config("hive.metastore.uris", hiveMetastoreUris)
      .config("hive.sql.warehouse.dir", hiveWarehouseDir)
      .config("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
      .config("spark.sql.crossJoin.enabled", "true")
      .config("spark.kryo.registrator", classOf[GanosSparkKryoRegistrator].getName)
      .enableHiveSupport()
      .getOrCreate()
//SparkSession加载JTS包用于处理时空数据
spark.withJTS
import spark.implicits._
val sc = spark.sparkContext
```

2. 加载Hive数据。

```
val tableName = args(0)
//从Hive中读取表 tableName
val dfFromHive = sparkSession.sql(
s"""
    |select * from $tableName
    |""".stripMargin)
```

3. 创建时空Geometry对象。

```
val ganosDF=dfFromHive.withColumn("geom",st_makePoint(col("x"),col("y")))
ganosDF.show
```

这里“x”与“y”代表每条记录中空间对象的横纵坐标值。

4. 时空查询。

```
ganosDF.createOrReplaceTempView("testpoints")
//创建SQL查询
val points = spark.sql("select * from testpoints where st_contains(st_makeBox2d(st_poin
t(38,48), st_point(52,62)),geom)")
```

7.2.3.2. PolarDB

PolarDB是阿里巴巴自主研发的下一代关系型分布式云原生数据库。阿里云自研Ganos时空引擎提供一系列的数据类型、函数和存储过程，用于在云原生关系型分布式数据库PolarDB中对空间/时空数据进行高效的存储、索引、查询和分析计算。DLA Ganos兼容PolarDB Ganos数据访问接口，可直接加载PolarDB Ganos中的矢量数据。

本文以全球船舶轨迹AIS数据集为例，展示如何利用DLA Ganos加载PolarDB Ganos中的矢量数据。

1. **始化Spark环境：**

```
//初始化SparkSession
val spark = SparkSession.builder
    .appName("Simple Application")
    .config("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
    .config("spark.sql.crossJoin.enabled", "true")
    .config("spark.kryo.registrator", classOf[GanosSparkKryoRegistrator].getName)
    .getOrCreate()
import spark.implicits._
//SparkSession加载JTS包用于处理时空数据
spark.withJTS
val sc = spark.sparkContext
```

2. **加载数据：**

   i. Spark format方式：

```
//配置链接参数
  val dsParams: JMap[String, String] = Map(
    "host" -> "PolarDB URL地址",
    "polardb.ganos" -> "true",
    "dbtype"->"polardb",
    "port"->"PolarDB接口",
    "schema"->"public",
    "database"->"数据库名称",
    "user"->"用户名",
    "passwd"->"密码")
//加载AIS数据源
df = spark.read.format("ganos-geometry")
      .options(dsParams)
      .option("ganos.feature", "AIS")
      .load()
df.show
```

   ii. Spark DataFrame API方式：

```
val layer=spark.read.ganos.polardbGeometry(dsParams)
layer.show
```

3. **输出结果：**

__fid__	ship_id	speed	course	rot	heading	messageid	sourceid	status	dtg	geom
4	205073000	14.0	144.8	0.0	145.0	1	17	机动船在航	2018-09-03 14:21:37	POINT (73.75408 1...
11	205073000	14.0	145.2	0.0	145.0	1	17	机动船在航	2018-09-04 00:41:32	POINT (75.0347733...
12	205073000	14.0	145.2	0.0	145.0	1	17	机动船在航	2018-09-04 00:41:51	POINT (75.0354933...
14	205073000	12.8	144.0	0.0	145.0	1	17	机动船在航	2018-09-04 00:49:42	POINT (75.0520266...
20	205073000	14.3	134.2	0.0	134.0	1	17	机动船在航	2018-09-04 12:31:38	POINT (76.7091466...
29	205073000	13.3	103.0	0.0	104.0	1	20	机动船在航	2018-09-05 10:46:28	POINT (81.1187666...
34	205073000	13.5	88.0	0.0	90.0	1	0	机动船在航	2018-09-05 22:00:39	POINT (83.6038333...
39	205073000	12.1	106.0	0.0	104.0	1	0	机动船在航	2018-09-08 06:02:57	POINT (95.4343016...
44	205073000	12.7	105.0	0.0	104.0	1	0	机动船在航	2018-09-08 07:11:25	POINT (95.6616866...
50	205073000	12.7	107.0	0.0	104.0	1	0	机动船在航	2018-09-08 08:21:06	POINT (95.9018916...
56	205073000	11.5	136.4	0.0	136.0	1	0	限于吃水	2018-09-09 17:46:10	POINT (101.400416...
59	205073000	11.5	124.5	0.0	123.0	1	0	限于吃水	2018-09-09 18:01:20	POINT (101.434085...
60	205073000	11.3	127.0	0.0	125.0	1	0	限于吃水	2018-09-09 18:06:40	POINT (101.448043...
71	205073000	10.9	125.2	0.0	125.0	1	0	限于吃水	2018-09-09 18:30:40	POINT (101.508566...
75	205073000	10.8	123.4	0.0	122.0	1	0	限于吃水	2018-09-09 18:38:29	POINT (101.528061...
79	205073000	10.8	125.0	0.0	122.0	1	0	限于吃水	2018-09-09 18:52:20	POINT (101.56322 ...
82	205073000	10.8	124.8	0.0	122.0	1	0	限于吃水	2018-09-09 18:56:00	POINT (101.572435...
86	205073000	10.9	125.2	0.0	121.0	1	0	限于吃水	2018-09-09 19:05:00	POINT (101.59507 ...
92	205073000	10.9	124.2	0.0	121.0	1	0	限于吃水	2018-09-09 19:19:09	POINT (101.630333...
100	205073000	9.8	188.3	0.0	194.0	1	0	限于吃水	2018-09-09 19:39:50	POINT (101.642328...

7.2.3.3. GeoMesa(HBase/Cassandra)

GeoMesa是由locationtech开源的一套地理大数据处理工具套件。本文主要介绍如何通过DLA Ganos查询基于GeoMesa管理的HBase和Cassandra数据库。

通过GeoMesa您可以在NoSQL分布式计算存储系统上进行大规模的地理空间查询和分析。GeoMesa目前支持的NoSQL数据库包括Accumulo、HBase、Google Bigtable和Cassandra等。有关GeoMesa的详细介绍请参见：geomesa官方文档。

> **说明** 如果您想通过DLA Ganos查询基于GeoMesa管理的其他数据库，可以参见GeoMesa文档修改相关参数即可。详细代码请参见：DLA Ganos GitHub样例库。

Lindorm(HBase)

1. 初始化SparkSession

```
val spark = SparkSession.builder
    .appName("Simple Application")
    .config("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
    .config("spark.sql.crossJoin.enabled", "true")
    .config("spark.kryo.registrator", classOf[GanosSparkKryoRegistrator].getName)
    .getOrCreate()
import spark.implicits._
//SparkSession加载JTS包用于处理时空数据。
spark.withJTS
val sc = spark.sparkContext
```

2. 配置HBase连接参数，并通过SparkSQL加载数据。

```
//指定HBase连接参数，POINT为Catalog名称。
val params = Map(
      "hbase.catalog" -> "AIS",
      "hbase.zookeepers" -> "zookeeper地址",
//加载AIS数据源。
val dataFrame = spark.read
      .format("ganos-geometry")
      .options(params)
      .option("ganos.feature", "testpoints")
      .load()
 dataFrame.createOrReplaceTempView("testpoints")
 //创建SQL查询。
 val points = spark.sql("select * from testpoints where st_contains(st_makeBox2d(st_poi
nt(38,48), st_point(52,62)),geom)")
 //输出Schema与表内容。
 points.printSchema
 points.show
```

3. 输出结果

```
root
 |-- __fid__: string (nullable = false)
 |-- name: string (nullable = true)
 |-- attr: string (nullable = true)
 |-- dtg: timestamp (nullable = true)
 |-- geom: point (nullable = true)
+-------+-----+-----+-------------------+-------------+
|__fid__| name| attr|                dtg|         geom|
+-------+-----+-----+-------------------+-------------+
|      2|name2|name2|2014-01-03 08:00:01|POINT (42 52)|
|      3|name3|name3|2014-01-04 08:00:01|POINT (43 53)|
|      4|name4|name4|2014-01-05 08:00:01|POINT (44 54)|
|      5|name5|name5|2014-01-06 08:00:01|POINT (45 55)|
|      6|name6|name6|2014-01-07 08:00:01|POINT (46 56)|
|      7|name7|name7|2014-01-08 08:00:01|POINT (47 57)|
|      8|name8|name8|2014-01-09 08:00:01|POINT (48 58)|
|      9|name9|name9|2014-01-10 08:00:01|POINT (49 59)|
+-------+-----+-----+-------------------+-------------+
```

注意 通过DLA Ganos将dataFrame对象写入数据库，然后再通过DLA Ganos将DataFrame回写到HBase数据库前，必须先创建好待写入的数据表，否则将写入失败。

```
points.write.format("ganos-geometry").options(dsParams).option("ganos.feature", "testpo
ints").save()
```

Cassandra

1. 初始化SparkSession

```
val spark = SparkSession.builder
    .appName("Simple Application")
    .config("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
    .config("spark.sql.crossJoin.enabled", "true")
    .config("spark.kryo.registrator", classOf[GanosSparkKryoRegistrator].getName)
    .getOrCreate()
import spark.implicits._
//SparkSession加载JTS包用于处理时空数据。
spark.withJTS
val sc = spark.sparkContext
```

2. 配置连接参数

```
val dsParams = Map(
    "cassandra.contact.point" -> "Cassandra连接地址:9042",
    "cassandra.keyspace" -> "ganos",
    "cassandra.catalog"->"test_sft"
  )
//创建DataFrame对应到Cassandra表。
  val dataFrame = spark.read
    .format("ganos-geometry")
    .options(dsParams)
    .option("ganos.feature", "testpoints")
    .load()
 dataFrame.createOrReplaceTempView("testpoints")
 //创建SQL查询。
 val points = spark.sql("select * from testpoints where st_contains(st_makeBox2d(st_poi
nt(38,48), st_point(52,62)),geom)")
 //输出Schema与表内容。
 points.printSchema
 points.show
```

3. 输出结果

```
root
 |-- __fid__: string (nullable = false)
 |-- name: string (nullable = true)
 |-- attr: string (nullable = true)
 |-- dtg: timestamp (nullable = true)
 |-- geom: point (nullable = true)
+-------+-----+-----+-------------------+-------------+
|__fid__| name| attr|                dtg|         geom|
+-------+-----+-----+-------------------+-------------+
|      2|name2|name2|2014-01-03 08:00:01|POINT (42 52)|
|      3|name3|name3|2014-01-04 08:00:01|POINT (43 53)|
|      4|name4|name4|2014-01-05 08:00:01|POINT (44 54)|
|      5|name5|name5|2014-01-06 08:00:01|POINT (45 55)|
|      6|name6|name6|2014-01-07 08:00:01|POINT (46 56)|
|      7|name7|name7|2014-01-08 08:00:01|POINT (47 57)|
|      8|name8|name8|2014-01-09 08:00:01|POINT (48 58)|
|      9|name9|name9|2014-01-10 08:00:01|POINT (49 59)|
+-------+-----+-----+-------------------+-------------+
```

> **注意** 您通过DLA Ganos将dataFrame对象写入数据库，再通过DLA Ganos将DataFrame回写到HBase数据库前，必须先创建好待写入的表，否则将写入失败。

```
points.write.format("ganos-geometry").options(dsParams).option("ganos.feature", "testpoints").save()
```

7.2.3.4. PostGIS

PostGIS在对象关系型数据库PostgreSQL上增加了存储管理空间数据的能力。DLA Ganos兼容PostGIS数据访问接口，可直接加载PostGIS中的矢量数据。本文以全球船舶轨迹AIS数据集为例，展示如何利用DLA Ganos加载PostGIS中的矢量数据。

操作步骤

1. 初始化SparkSession：

```
//初始化SparkSession
val spark = SparkSession.builder
    .appName("Simple Application")
    .config("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
    .config("spark.sql.crossJoin.enabled", "true")
    .config("spark.kryo.registrator", classOf[GanosSparkKryoRegistrator].getName)
    .getOrCreate()
import spark.implicits._
//SparkSession加载JTS包用于处理时空数据
spark.withJTS
val sc = spark.sparkContext
```

2. 加载数据：

```
//配置链接参数
  val dsParams: JMap[String, String] = Map(
    "host" -> "PostgreSQL URL地址",
    "polardb.ganos" -> "true",
    "dbtype"->"postgis",
    "port"->"PostgreSQL接口",
    "schema"->"public",
    "database"->"数据库名称",
    "user"->"用户名",
    "passwd"->"密码")
//加载AIS数据源
df = spark.read.format("ganos-geometry")
      .options(dsParams)
      .option("ganos.feature", "AIS")
      .load()
df.show
```

3. 输出结果：

```
+-------+---------+-----+------+---+-------+---------+--------+--------+-------------------+--------------------+
|__fid__|  ship_id|speed|course|rot|heading|messageid|sourceid|  status|                dtg|                geom|
+-------+---------+-----+------+---+-------+---------+--------+--------+-------------------+--------------------+
|      4|205073000| 14.0| 144.8|0.0|  145.0|        1|      17|机动船在航|2018-09-03 14:21:37|POINT (73.75408 1...|
|     11|205073000| 14.0| 145.2|0.0|  145.0|        1|      17|机动船在航|2018-09-04 00:41:32|POINT (75.0347733...|
|     12|205073000| 14.0| 145.2|0.0|  145.0|        1|      17|机动船在航|2018-09-04 00:41:51|POINT (75.0354933...|
|     14|205073000| 12.8| 144.0|0.0|  145.0|        1|      17|机动船在航|2018-09-04 00:49:42|POINT (75.0520266...|
|     20|205073000| 14.3| 134.2|0.0|  134.0|        1|      17|机动船在航|2018-09-04 12:31:38|POINT (76.7091466...|
|     29|205073000| 13.3| 103.0|0.0|  104.0|        1|      20|机动船在航|2018-09-05 10:46:28|POINT (81.1187666...|
|     34|205073000| 13.5|  88.0|0.0|   90.0|        1|       0|机动船在航|2018-09-05 22:00:39|POINT (83.6038333...|
|     39|205073000| 12.1| 106.0|0.0|  104.0|        1|       0|机动船在航|2018-09-08 06:02:57|POINT (95.4343016...|
|     44|205073000| 12.7| 105.0|0.0|  104.0|        1|       0|机动船在航|2018-09-08 07:11:25|POINT (95.6616866...|
|     50|205073000| 12.7| 107.0|0.0|  104.0|        1|       0|机动船在航|2018-09-08 08:21:06|POINT (95.9018916...|
|     56|205073000| 11.5| 136.4|0.0|  136.0|        1|       0|  限于吃水|2018-09-09 17:46:10|POINT (101.400416...|
|     59|205073000| 11.5| 124.5|0.0|  123.0|        1|       0|  限于吃水|2018-09-09 18:01:20|POINT (101.434085...|
|     60|205073000| 11.3| 127.0|0.0|  125.0|        1|       0|  限于吃水|2018-09-09 18:06:40|POINT (101.448043...|
|     71|205073000| 10.9| 125.2|0.0|  125.0|        1|       0|  限于吃水|2018-09-09 18:30:40|POINT (101.508566...|
|     75|205073000| 10.8| 123.4|0.0|  122.0|        1|       0|  限于吃水|2018-09-09 18:38:29|POINT (101.528061...|
|     79|205073000| 10.8| 125.0|0.0|  122.0|        1|       0|  限于吃水|2018-09-09 18:52:20|POINT (101.56322 ...|
|     82|205073000| 10.8| 124.8|0.0|  122.0|        1|       0|  限于吃水|2018-09-09 18:56:00|POINT (101.572435...|
|     86|205073000| 10.9| 125.2|0.0|  121.0|        1|       0|  限于吃水|2018-09-09 19:05:00|POINT (101.59507 ...|
|     92|205073000| 10.9| 124.2|0.0|  121.0|        1|       0|  限于吃水|2018-09-09 19:19:09|POINT (101.630333...|
|    100|205073000|  9.8| 188.3|0.0|  194.0|        1|       0|  限于吃水|2018-09-09 19:39:50|POINT (101.642328...|
+-------+---------+-----+------+---+-------+---------+--------+--------+-------------------+--------------------+
```

7.2.3.5. GeoTools

DLA Ganos内置了GeoTools数据驱动。任何兼容GeoTools数据访问接口的存储系统都可以作为DLA Ganos矢量数据源，如PostGIS、GeoMesa等。本文主要介绍DLA Ganos如何加载兼容GeoTools数据访问接口的存储系统中的数据。

操作步骤

1. 初始化SparkSession：

```
//初始化SparkSession
val spark = SparkSession.builder
      .appName("Simple Application")
      .config("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
      .config("spark.sql.crossJoin.enabled", "true")
      .config("spark.kryo.registrator", classOf[GanosSparkKryoRegistrator].getName)
      .getOrCreate()
//指定HBase连接参数，POINT为Catalog名称
val params = Map(
      "hbase.catalog" -> "POINT",
      "hbase.zookeepers" -> "zookeeper地址",
      "geotools" -> "true")
//加载AIS数据源
val dataFrame = spark.read
      .format("ganos-geometry")
      .options(params)
      .option("ganos.feature", "AIS")
      .load()
dataFrame.show
```

2. 输出结果如下：

```
+---------+---------+-----+------+---+-------+---------+--------+------+-------------------+--------------------+
|  __fid__|  ship_id|speed|course|rot|heading|messageid|sourceid|status|                dtg|                geom|
+---------+---------+-----+------+---+-------+---------+--------+------+-------------------+--------------------+
|205073000|205073000| 14.0| 144.8|0.0|  145.0|        1|      17|  在航|2018-09-03 14:21:37|POINT (73.75408 1...|
|205073000|205073000| 14.0| 144.8|0.0|  145.0|        1|      17|  在航|2018-09-03 14:21:45|POINT (73.75435 1...|
|205073000|205073000| 12.8| 146.0|0.0|  143.0|        1|      17|  在航|2018-09-03 16:49:06|POINT (74.06773 1...|
|205073000|205073000| 13.1| 130.0|0.0|  129.0|        1|      17|  在航|2018-09-03 09:15:50|POINT (72.96676 1...|
|205073000|205073000| 13.1| 133.0|0.0|  131.0|        1|      17|  在航|2018-09-03 11:10:17|POINT (73.27779 1...|
|205073000|205073000| 13.1| 133.0|0.0|  131.0|        1|      17|  在航|2018-09-03 11:12:20|POINT (73.28349 1...|
|205073000|205073000| 13.2| 134.0|0.0|  133.0|        1|      17|  在航|2018-09-03 12:45:25|POINT (73.54347 1...|
|205073000|205073000| 12.8| 146.0|0.0|  143.0|        1|      17|  在航|2018-09-03 17:55:34|POINT (74.20368 1...|
|205073000|205073000| 12.6| 142.0|0.0|  141.0|        1|      17|  在航|2018-09-03 22:29:56|POINT (74.76799 9...|
|205073000|205073000| 12.8| 145.0|0.0|  145.0|        1|      17|  在航|2018-09-04 00:07:58|POINT (74.96557 9...|
|205073000|205073000| 12.9| 145.0|0.0|  145.0|        1|      17|  在航|2018-09-04 00:35:11|POINT (75.02181 9...|
|205073000|205073000| 14.0| 145.2|0.0|  145.0|        1|      17|  在航|2018-09-04 00:41:32|POINT (75.03477 9...|
|205073000|205073000| 14.0| 145.2|0.0|  145.0|        1|      17|  在航|2018-09-04 00:41:51|POINT (75.03549 9...|
|205073000|205073000| 12.8| 145.0|0.0|  145.0|        1|      17|  在航|2018-09-04 00:42:02|POINT (75.03581 9...|
|205073000|205073000| 12.8| 144.0|0.0|  145.0|        1|      17|  在航|2018-09-04 00:49:42|POINT (75.05203 9...|
|205073000|205073000| 13.2| 146.0|0.0|  147.0|        1|      17|  在航|2018-09-04 02:30:35|POINT (75.25965 9...|
|205073000|205073000| 13.2| 146.0|0.0|  147.0|        1|      17|  在航|2018-09-04 02:30:45|POINT (75.26 9.04...|
|205073000|205073000| 13.7| 141.0|0.0|  139.0|        1|      17|  在航|2018-09-04 10:54:37|POINT (76.46939 7...|
|205073000|205073000| 13.7| 141.0|0.0|  139.0|        1|      17|  在航|2018-09-04 10:54:59|POINT (76.47024 7...|
|205073000|205073000| 13.7| 142.0|0.0|  139.0|        1|      17|  在航|2018-09-04 10:55:59|POINT (76.47261 7...|
+---------+---------+-----+------+---+-------+---------+--------+------+-------------------+--------------------+
only showing top 20 rows
```

7.2.4. 时空几何函数参考

SparkSession对象后，就可以通过read方法加载数据为DataFrame对象了。Ganos Spark提供了一系列UDF时空算子实现基于SQL的时空数据查询，本文主要介绍相关时空算子函数。

SpatialConstructors空间构造函数

- **ST_GeomFromGeoHash**

  ```
  Geometry st_geomFromGeoHash(String geohash, Int prec)
  ```

  给定GeoHash编码，返回该编码对应的GeoHash值，GeoHash的精度为prec位。

- **ST_GeomFromWKT**

  ```
  Geometry st_geomFromWKT(String wkt)
  ```

  从给定WKT格式表示的空间对象描述转化为Geometry对象。

- **ST_GeomFromWKB**

  ```
  Geometry st_geomFromWKB(Array[Byte] wkb)
  ```

  从给定WKB格式表示的空间对象描述转化为Geometry对象。

- **ST_LineFromText**

  ```
  LineString st_lineFromText(String wkt)
  ```

  给定左下（lowerLeft）与右上（upperRight）点对象，生成它们所表示的空间范围的Geometry对象。

- **ST_MakeBBOX**

  ```
  Geometry st_makeBBOX(Double lowerX, Double lowerY, Double upperX, Double upperY)
  ```

  按照给定边界的坐标值生成所表示的空间范围的Geometry对象。

- **ST_MakePolygon**

  ```
  Polygon st_makePolygon(LineString shell)
  ```

生成线对象Shell所包围的区域的Polygon对象，Shell必须为闭合线对象。

- ST_MakePoint

```
Point st_makePoint(Double x, Double y)
```

给定x，y坐标，生成POINT对象

- ST_MakeLine

```
LineString st_makeLine(Seq[Point] points)
```

给定一系列POINT对象，生成它们所构成的LineString对象。

- ST_MakePointM

```
Point st_makePointM(Double x, Double y, Double m)
```

给定x,y,m坐标值，生成POINT对象。

- ST_MLineFromText

```
MultiLineString st_mLineFromText(String wkt)
```

生成wkt格式表示的MultiLineString对象。

- ST_MPolyFromText

```
MultiPolygon st_mPolyFromText(String wkt)
```

生成wkt格式表示的MultiPolygon对象。

- ST_Point

```
Point st_point(Double x, Double y)
```

等同于ST_MakePoint，即给定x，y坐标，生成POINT对象。

- ST_PointFromGeoHash

```
Point st_pointFromGeoHash(String geohash, Int prec)
```

返回由Geohash字符串geohash（base-32编码）定义的边界框的几何中心处的Point，其精度为prec位。

- ST_PointFromText

```
Point st_pointFromText(String wkt)
```

- ST_PointFromWKB

```
Point st_pointFromWKB(Array[Byte] wkb)
```

从给定WKB格式表示的空间对象描述生成Point对象。

- ST_Polygon

```
Polygon st_polygon(LineString shell)
```

从给定LineString空间对象生成Polygon对象。

- ST_PolygonFromText

```
Polygon st_polygonFromText(String wkt)
```

从给定WKT格式表示的空间对象描述生成Polygon对象。

Geometry Accessors函数

- **ST_Boundary**

```
Geometry st_boundary(Geometry geom)
```

使用 ST_Boundary 函数可确定源 ST_Geometry 的边界。

- **ST_CoordDim**

```
Int st_coordDim(Geometry geom)
```

要评估几何的维度，请使用 ST_Dimension 函数，该函数处理 ST_Geometry 要素并以整数形式返回维度。

- **ST_Dimension**

```
Int st_dimension(Geometry geom)
```

要评估几何的维度，请使用 ST_Dimension 函数，该函数处理 ST_Geometry 要素并以整数形式返回维度。

- **ST_Envelope**

```
Geometry st_envelope(Geometry geom)
```

T_Envelope 函数处理 ST_Geometry 并返回表示源 ST_Geometry 包络矩形的 ST_Geometry。

- **ST_ExteriorRing**

```
LineString st_exteriorRing(Geometry geom)
```

以 ST_LineString 形式返回 ST_Polygon 的外部环。

- **ST_GeometryN**

```
Int st_geometryN(Geometry geom, Int n)
```

评估 ST_Polygon 和索引并以 ST_LineString 形式返回第 n 个内部环。

- **ST_IsClosed**

```
Boolean st_isClosed(Geometry geom)
```

使用 ST_IsClosed 谓词函数可确定线串是否闭合；如果线串的起点与终点相交，则 ST_IsClosed 返回 TRUE。

- **ST_IsCollection**

```
Boolean st_isCollection(Geometry geom)
```

判断geom对象是否为几何对象集合。

- **ST_IsEmpty**

```
Boolean st_isEmpty(Geometry geom)
```

ST_IsEmpty 谓词函数用于确定几何是否为空。该函数分析 ST_Geometry，如果 ST_Geometry 为空，则返回 1 (TRUE)；如果不为空，则返回 0 (FALSE)

- **ST_IsRing**

```
Boolean st_isRing(Geometry geom)
```

ST_IsRing 以 ST_LineString 作为输入参数，如果是环（如 ST_LineString 是闭合的简单线串），则返回 1；否则返回 0

- **ST_IsSimple**

```
Boolean st_isSimple(Geometry geom)
```

ST_IsSimple 谓词函数用于确定 ST_LineString、ST_MultiPoint 或 ST_MultiLineString 是简单的还是非简单的。

- **ST_IsValid**

```
Boolean st_isValid(Geometry geom)
```

ST_IsSimple 谓词函数用于确定集合对象geom是否为有效的集合对象。

- **ST_NumGeometries**

```
Int st_numPoints(Geometry geom)
```

您可能想要确定多部分几何中各种几何的数目，例如 ST_MultiPoint、ST_MultiLineString 和 ST_MultiPolygon 的数目。为此，请使用 ST_NumGeometries谓词函数。此函数返回几何集合中各种元素的计数。

- **ST_NumPoints**

```
Int st_numPoints(Geometry geom)
```

评估 ST_LineString 并以整数数形式返回其序列中的点数。

- **ST_PointN**

```
Point st_pointN(Geometry geom, Int n)
```

获取 ST_LineString 和第 n 个点的索引，然后返回该点。

- **ST_X**

```
Float st_X(Geometry geom)
```

以双精度数形式返回点数据类型的 x 坐标值。

- **ST_Y**

```
Float st_y(Geometry geom)
```

以双精度数形式返回点数据类型的 y 坐标值。

Geometry Cast函数

- **ST_CastToPoint**

```
Point st_castToPoint(Geometry g)
```

将Geometry类型转换为Point类型。

- ST_CastToPolygon

```
Polygon st_castToPolygon(Geometry g)
```

将Geometry类型转换为Polygon类型。

- ST_CastToLineString

```
LineString st_castToLineString(Geometry g)
```

将Geometry类型转换为LineString类型。

- ST_ByteArray

```
Array[Byte] st_byteArray(String s)
```

将字符串类型按照UTF-8转换为Array[Byte]类型。

Geometry Editors函数

ST_Translate

```
Array[Byte] st_asBinary(Geometry geom)
```

返回将Geometry对象按照矢量（deltaX，deltaY）位移后生成的新的Geometry对象。

Geometry Outputs函数

- ST_AsBinary

```
Array[Byte] st_asBinary(Geometry geom)
```

返回Geometry的Array[Byte] 表示方式。

- ST_AsGeoJSON

```
String st_asGeoJSON(Geometry geom)
```

返回Geometry的GeoJSON表示方式。

- ST_AsLatLonText

```
String st_asLatLonText(Point p)
```

返回描述Point对象的纬度和经度的字符串描述，以度，分和秒为单位（这假设p的坐标单位是纬度和经度）。

- ST_AsText

```
String st_asText(Geometry geom)
```

返回Geometry的字符串表示方式。

- ST_GeoHash

```
String st_geoHash(Geometry geom, Int prec)
```

返回Geometry的GeoHash编码表示方式，精度位数为prec。

Spatial Relationships函数

- **ST_Contains**

```
Boolean st_contains(Geometry a, Geometry b)
```

当且仅当a的外部没有b的位置时，返回true，并且b的内部的至少一个点位于a的内部。

- **ST_Covers**

```
Boolean st_covers(Geometry a, Geometry b)
```

如果属于b的任意一点都位于a的内部，则返回true

- **ST_Crosses**

```
Boolean st_crosses(Geometry a, Geometry b)
```

如果a与b有部分相同区域，但不是全部，则返回true

- **ST_Disjoint**

```
Boolean st_disjoint(Geometry a, Geometry b)
```

等同于“NOT ST_Intersects”， 即a与b没有任何相交的部分

- **ST_Equals**

```
Boolean st_equals(Geometry a, Geometry b)
```

如果a与b完全想同，则返回true

- **ST_Intersects**

```
Boolean st_intersects(Geometry a, Geometry b)
```

如果a与b有部分会全部区域相同，则返回true

- **ST_Overlaps**

```
Boolean st_overlaps(Geometry a, Geometry b)
```

如果几何具有一些但不是所有的共同点，具有相同的尺寸，并且两个几何的内部的交点与几何本身具有相同的尺寸，则返回true

- **ST_Touches**

```
Boolean st_touches(Geometry a, Geometry b)
```

如果a与b至少有一个共同POINT对象，并且a与b内部没有任何相交部分，则返回true。

- **ST_Within**

```
Boolean st_within(Geometry a, Geometry b)
```

如果a完全位于b内部，则返回true

- **ST_Relate**

```
String st_relate(Geometry a, Geometry b)
```

返回描述两个几何的内部，边界和外部之间的交叉点的维度的“九交模型”交互矩阵模式。

- ST_RelateBool

```
Boolean st_relateBool(Geometry a, Geometry b, String mask)
```

如果“九交模型”交互矩阵掩码与从st_relate（a，b）获得的交互矩阵模式匹配，则返回true。

- ST_Area

```
Double st_area(Geometry g)
```

返回Geometry的面积

- ST_Centroid

```
Point st_centroid(Geometry g)
```

返回Geometry的几何中心

- ST_ClosestPoint

```
Point st_closestPoint(Geometry a, Geometry b)
```

返回a与b距离最近的POINT对象

- ST_Distance

```
Double st_distance(Geometry a, Geometry b)
```

以坐标参考系统为单位返回两个几何之间的2D笛卡尔距离（例如，EPSG的度数：4236）。

- ST_DistanceSphere

```
Double st_distanceSphere(Geometry a, Geometry b)
```

假设球形地球，近似两个经度/纬度几何之间的最小距离。

- ST_Length

```
Double st_length(Geometry geom)
```

以坐标参考系统为单位返回线性几何的2D路径长度或面几何的周长（例如，EPSG的度数：4236）。对于其他几何类型（例如Point），返回0.0。

- ST_LengthSphere

```
Double st_lengthSphere(LineString line)
```

使用球形地球模型近似LineString几何的2D路径长度。返回的长度以米为单位。近似值在st_lengthSpheroid的0.3%范围内，并且在计算上更有效。

Geometry Processing函数

- ST_antimeridianSafeGeom

```
Geometry st_antimeridianSafeGeom(Geometry geom)
```

如果geom跨越antimeridian，则尝试将几何转换为“antimeridian-safe”的等效形式（即输出几何由BOX（-180 -90,180,90）覆盖）。在某些情况下，此方法可能会失败，在这种情况下将返回输入几何，并将记录错误。

- ST_BufferPoint

```
Geometry st_bufferPoint(Point p, Double buffer)
```

返回覆盖Point p给定半径内所有点的几何，其中radius以米为单位

- ST_ConvexHull

```
Geometry st_convexHull(Geometry geom)
```

聚合函数。几何体的凸包表示包含聚合行中所有几何图形的最小凸面几何体。

7.3. 地理空间分析（Raster）

7.3.1. 基本概念

栅格数据（Raster Data）是将地理空间分割成有规律的网格，每一个网格称为一个单元（像元或像素），并在各单元上赋予相应的属性值来表示实体的一种数据形式。

栅格数据通常有两种类型的栅格数据：专题数据和影像数据。

- 专题数据：每个栅格像元的值可以是一个测量值或分类值，如污染物浓度、降雨量、土地的权属类型或植被类型等。
- 影像数据：又称遥感影像或遥感图像（Remote Sensing Image），是通过地面遥感、航空遥感或航天遥感平台拍摄的，记录各种地物电磁波大小的胶片或照片，包含如航空影像和遥感卫星影像等。

每一幅栅格数据均带有时间属性和空间属性，我们称之为时空栅格。时空栅格还强调了栅格数据的时态特性，比如管理时间序列的系列栅格数据。

DLA Ganos Raster

DLA Ganos Raster是DLA Ganos针对栅格数据管理和处理的时空数据引擎及配套工具集，它允许用户使用阿里云Lindorm（HBase）存储、索引、查询、分析和传输栅格数据及其相关元数据。栅格数据以数据分块（Tile或Block）的形式存储在Lindorm（HBase）中，每个分块对应于一个主键，并且该主键支持时间和空间查询。此外，Ganos Raster允许多源栅格数据（如遥感、摄影测量和专题地图）之间的融合与分析以及数据服务发布等功能（如TMS或WMTS等）。Ganos Raster可用于包括基于位置的服务、地理图像存档、环境监测和评估、地质工程和勘探、自然资源管理、国防、应急响应、电信、交通、城市规划以及国土安全等领域。

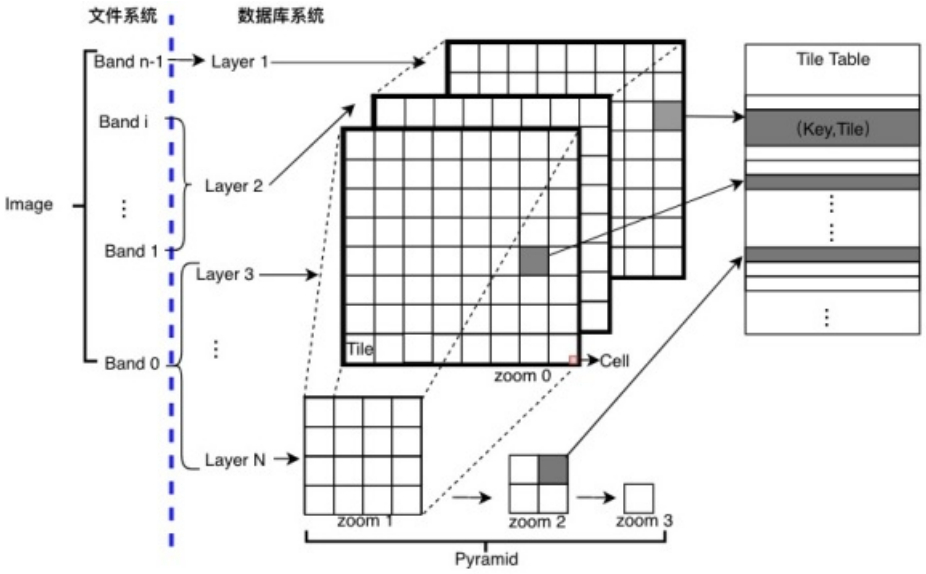
Ganos Raster数据模型

基本概念

Ganos Raster数据模型主要包括以下几个元素构成：

- Image：泛指一景遥感影像，如一个TIFF文件。
- Catalog：数据目录，相当于”数据库“的概念。Catalog是一个逻辑概念，它由一个Lindorm（HBase）数据库中所有Layer（一个Layer一个表）与一个Metadata元数据表（其中每一行代表一个Layer的元数据）构成。
- Cover或Coverage：多幅栅格数据构成的数据集合，等同于镶嵌数据集。
- Layer：由多个Tile构成的2D栅格数据图层，每个Tile拥有一个行列号。
- Tile或Block：数据分块，为一系列像素的集合。Tile为栅格数据在数据库中存储的基本单元，每个Tile包含若干Cell，一般像素大小为256或512。
- Cell或Pixel：表示Tile中的一个像素，可以拥有不同的数据类型如Byte，Short，Int，Double等。

- Key：唯一标识Tile的键值，分为SpatialKey、SpaceTimeKey、TimeKey三种。
- Pyramid：栅格金字塔，方便快速显示。每个Pyramid包含不同的层级，每个层级对应一个Layer，第0层代表原始数据。
- Metadata：栅格的存储元数据（空间范围、投影类型、像素类型等），不包含遥感平台元数据。
- Layout Definition或Layout：定义Layer中Tile的分块方式，每个像素所代表的地理范围，以及Key到真实坐标系中的映射关系等信息。
- Layout Scheme：一个金字塔中所有Layer的zoom编号及其对应的Layout Definition组成一个Layout Scheme。
- 栅格数据的文件表示以及在数据库中存储的逻辑模型如下图所示：



Band与Layer

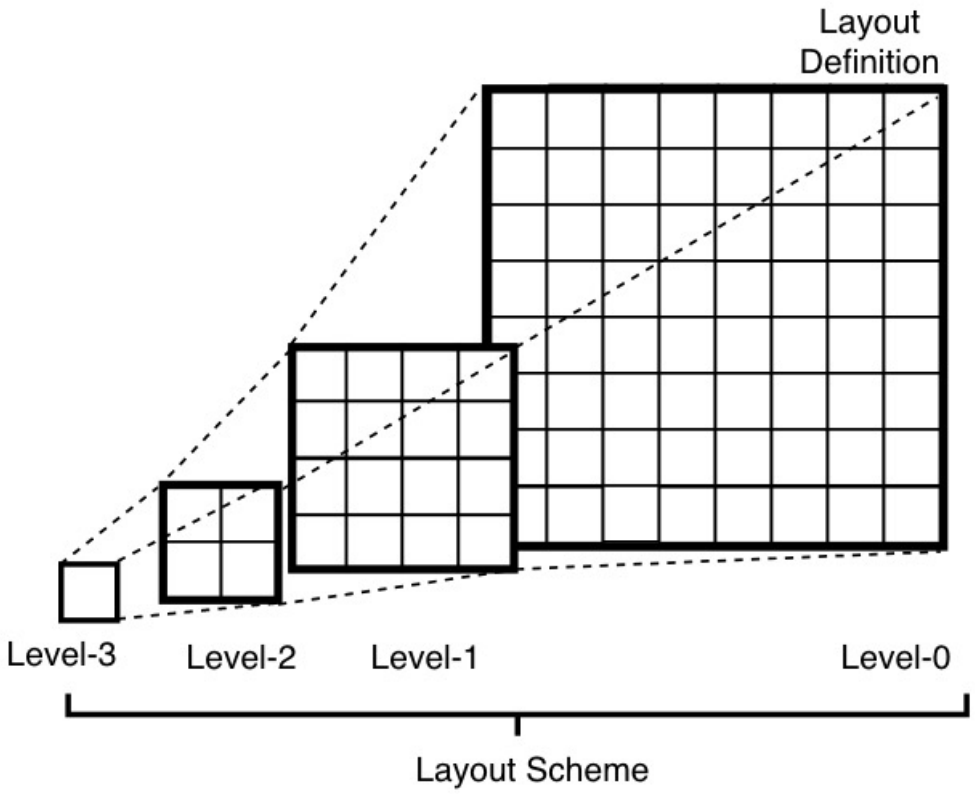
Ganos Raster采用了一种简单而高效的通用栅格数据模型来管理专题数据和遥感影像数据。一幅遥感影像（Image）由若干可以表示为2D栅格图层的波段（Band）组成，每个Band的一个像素表示为一个像素单元（Cell）。注意，为了便于存储与管理，Ganos Raster规定同一Band中的Cell必须是同质的，即具有相同的数据类型与投影参数，但是同一影像的不同波段可以是异质的。每一个Image都有对应的元数据（Metadata）信息，如图幅范围（Extent）、数据类型、投影信息、行列号等。栅格数据在数据库中以Layer的形式进行表达，每个Layer是以数据块（Tile）为基本存储单元在Ganos Raster中进行存储和管理。Tile分为单波（Tile）段与多波段（MultibandTile）两种类型，每个MultibandTile由任意数量的Tile构成。

如上图所示，Band与Layer的关系分为三种：

- 一个Band对应一个Layer： 对于一些单波段的栅格数据，如模型的输出结果，遥感影像分析结果，每个像素一般只包括一个值，在不构建金字塔模型的情况下，每个Band对应于一个Layer。
- 多个Band构成一个Layer： 如遥感影像的真彩色（RGB）合成的图层，可以MultibandTile进行表示，这种情况下，R、G、B三个波段可以合成为一个Layer进行存储。
- 一个Band包括多个Layer： 如果对栅格数据创建了金字塔模型，那么一个Band会包含多个层级的金字塔数据（zoom），而每个zoom对应了一个Layer。

金字塔模型

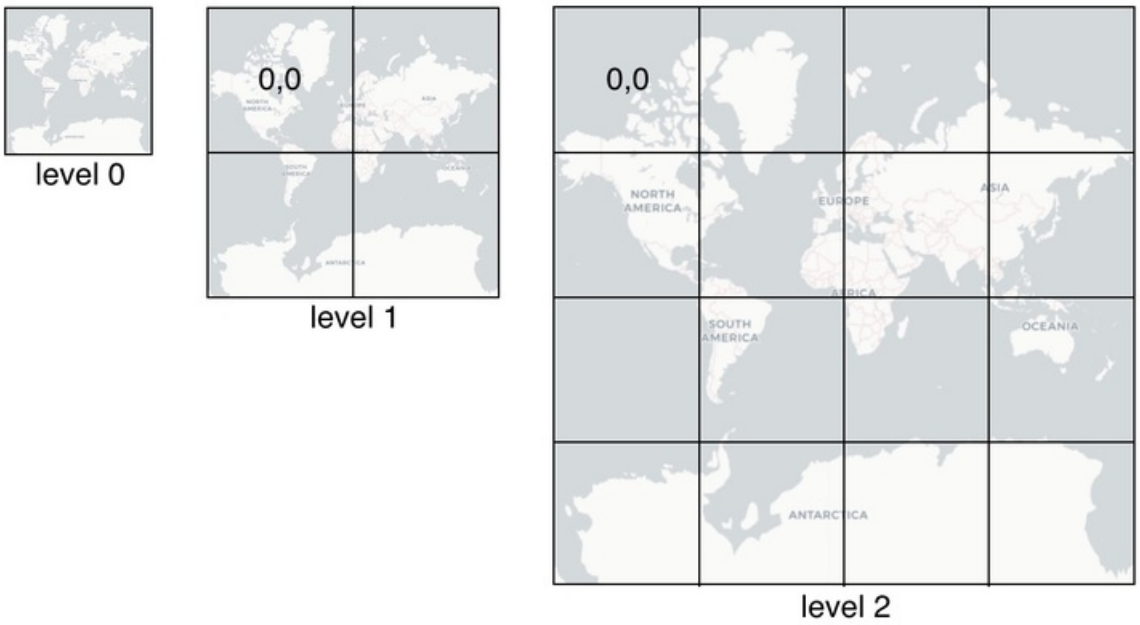
对栅格数据创建金字塔可以显著改善数据访问性能。金字塔模型是原始栅格数据集的缩减采样版本，一般包含多个缩减后的连续的采样图层。各个连续图层均以 2:1 的比例进行重采样。以下是为栅格数据集创建的4级金字塔示例：



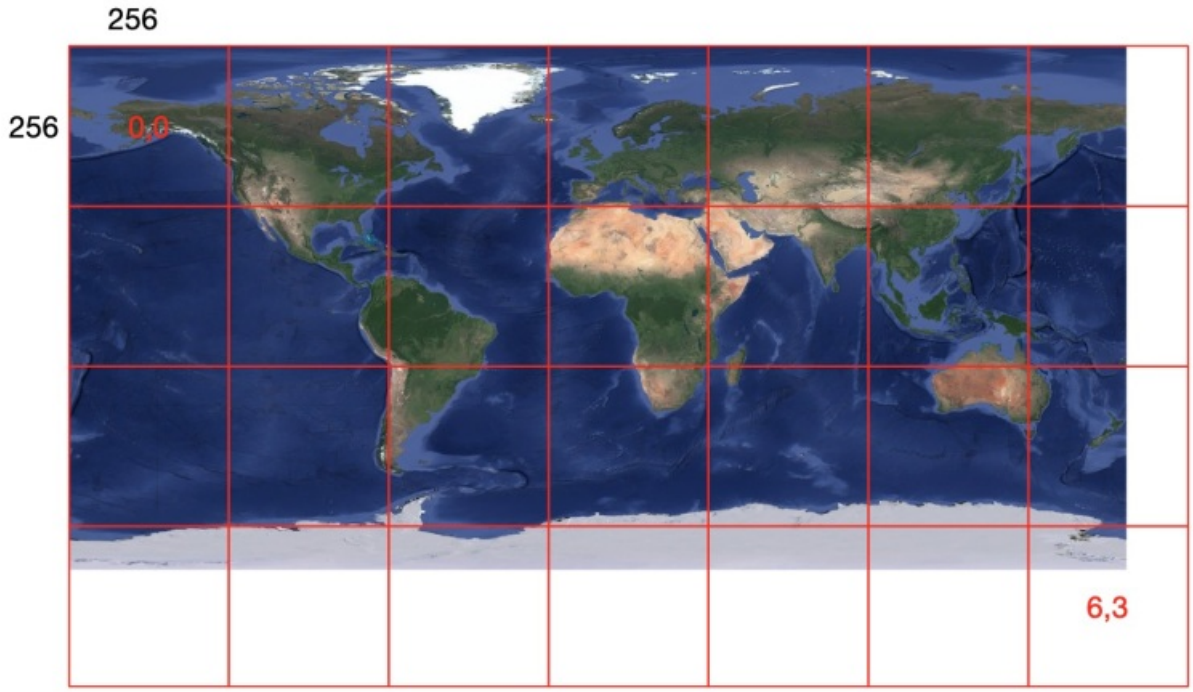
金字塔最大的优点是仅检索使用指定分辨率（取决于显示要求）的数据，从而可以极大提高栅格数据的访问速度。利用金字塔，可在绘制整个数据集时快速显示较低分辨率的数据分块。而随着放大操作的进行，各个更精细的分辨率等级将逐渐得到绘制；但性能将保持不变。数据库会根据用户的显示比例自动选择最适合的金字塔等级。每个栅格数据集只需构建一次金字塔，之后每次查看栅格数据集时都会访问这些金字塔。栅格数据集越大，创建金字塔集所花费的时间就越长。但是，这也就意味着可以为将来节省更多的时间。

Ganos Raster 数据分块模式（Layout Scheme）

Layout Definition与Layout Scheme用来定义Layer的分块方式。给定数据图幅范围和像元大小（一个像元代表的实际空间范围），Layout Scheme可以给出一个缩放层序列（Zoom）级以及各层级对应的Layout Definition。Ganos Raster支持两种数据分块模式：zoom与local。zoom方式是按照TMS的分块标准在全球范围内对数据进行分块与编码，最左上角的数据块为起始点（0,0），按照从左到右，从上到下的顺序进行增加。该方法的优点是各类型栅格数据都按照统一的时空格网进行分块，便于叠加分析与多元数据融合。此外，由于遵循了TMS标准，使用该方法分块后的数据经过渲染后可直接发布为TMS服务用于显示（如使用OpenLayers）。但该方法的缺点是数据分块速度相对较慢。以zoom方式对数据进行分块的示意图如下：



除了zoom方式外，Ganos Raster还提供了一种基于图像本地坐标系统的分块方式：local。该方式定义的起算点不再是基于全球范围，而是以数据本身的图幅范围的左上角为起始点（0,0），然后按照256x256像素窗口对数据进行分块，直到完全覆盖图像所覆盖的范围结束，多出的像素位置以NoData值填充。该方式的优点在于第0层为原始数据分辨率，保持了数据原始的数据信息，而且数据分块速度快，图像更新方便，查询效率高。但缺点是不同栅格数据没有统一的数据分块方式，不便于进行叠加分析。Ganos Raster 默认使用local方式对数据进行分块并创建金字塔。



坐标系统

Ganos Raster支持OGC CRS标准定义的坐标系统。用户可以按照EPSG坐标参照系统参数来定义栅格数据所要采用的坐标系统，目前Ganos Raster比较常用的EPSG主要有：

1. EPSG:4326即WGS84投影坐标系，它采用经纬度的方式展示，是目前最常用的坐标系统之一。
2. EPSG:3857即Mercator球形墨卡托投影坐标系，也叫做或Web墨卡托坐标系。目前Google Maps等主流的Web制图应用程序都在使用此坐标系。

想要查询具体的EPSG参数，请参见https://epsg.io/。

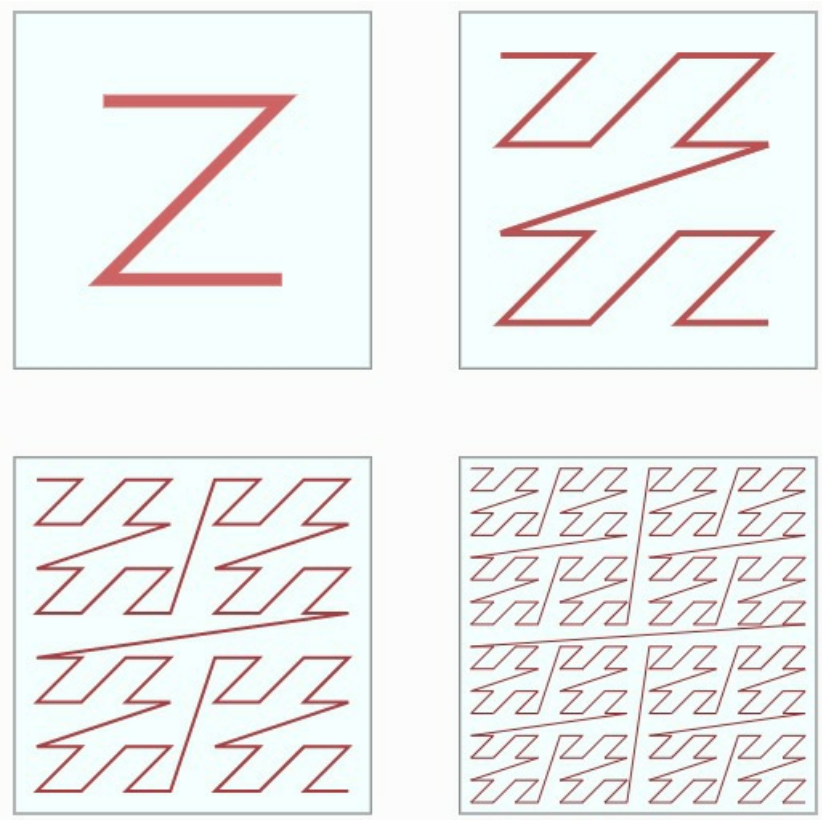
主键与索引

当栅格数据被分块为Tile之后，需要定义他们的组织方式从而方便创建数据索引。Tile在Lindorm（HBase）中以Key-Value（键值对）的形成进行存储。每个Tile的Key是由图层名、层级、时间（SpaceTimeKey）、行号、列号等属性构成。目前Lindorm（HBase） Ganos Raster提供了两种Key模型：

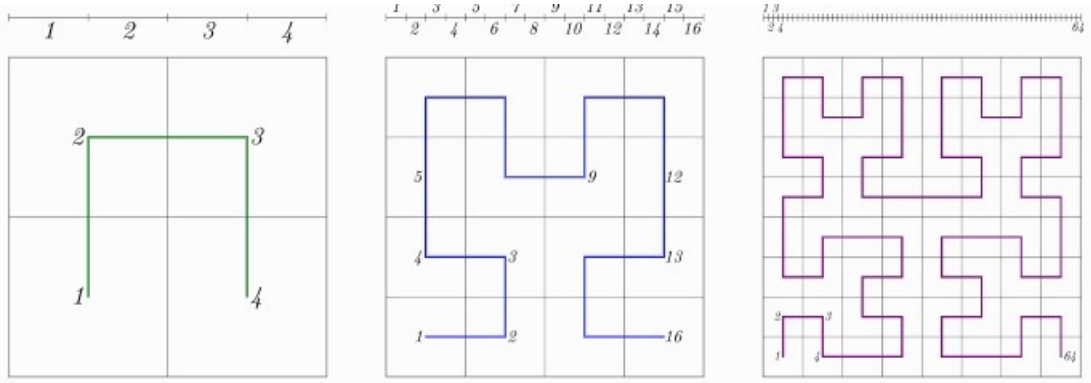
- SpatialKey：空间主键

  SpatialKey采用空间填充曲线（Space Filling Curve，后面简称SFC）对Tile进行编码和索引，Ganos Raster目前支持两种类型的空间索引，它们分别是：

  Z-Curve



Hilbert

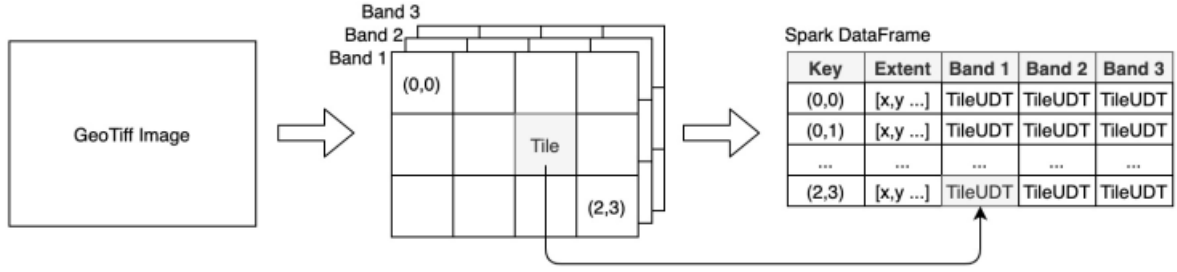


- SpaceTimeKey：时空主键

  SpaceTimeKey主要是在SpatialKey的基础上增加了时间维度，可以理解成三维的SFC。

DLA Ganos时空栅格

在DLA Ganos中，Tile为栅格数据处理的基本单元，所有栅格数据都以TileUDT的方式被Spark加载并参与计算,如下图所示：



Ganos目前支持的栅格数据源包括：

- PolarDB
- Lindorm（HBase）
- OSS
- HDFS

7.3.2. 快速开始

本文主要介绍在DLA中如何快速上手时序时空引擎Ganos。

前提条件

请提交工单获取DLA Ganos SDK用于本机编译调试。

操作步骤

1. 准备数据

   将tiff文件上传至OSS指定目录下，例如：

   ```
   oss://bucket名称/raster
   ```

2. 加载OSS数据

   i. 新建 `maven` 工程 `dla-ganos-quickstart` ，然后编辑项目的 `pom.xml` 文件：

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <project xmlns="http://maven.apache.org/POM/4.0.0"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.
   org/xsd/maven-4.0.0.xsd">
       <modelVersion>4.0.0</modelVersion>
       <groupId>com.aliyun.ganos.dla</groupId>
       <artifactId>dla-ganos-quickstart</artifactId>
       <version>1.0</version>
       <properties>
           <scala.version>2.11.12</scala.version>
           <scala.binary.version>2.11</scala.binary.version>
           <scala.xml.version>1.0.6</scala.xml.version>
           <scala.parsers.version>1.0.6</scala.parsers.version>
           <scalalogging.version>3.8.0</scalalogging.version>
           <spark.version>2.4.3</spark.version>
           <kryo.version>3.0.3</kryo.version>
       </properties>
       <dependencies>
           <dependency>
               <groupId>com.aliyun.ganos</groupId>
               <artifactId>dla-ganos-sdk</artifactId>
               <version>1.0</version>
               <scope>system</scope>
               <systemPath>
                   下载的dla-ganos-sdk-1.0.jar的路径
               </systemPath>
           </dependency>
           <dependency>
   ```

```
<dependency>
    <groupId>io.spray</groupId>
    <artifactId>spray-json_2.11</artifactId>
    <version>1.3.5</version>
</dependency>
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_${scala.binary.version}</artifactId>
    <exclusions>
        <exclusion>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-databind</artifactId>
        </exclusion>
    </exclusions>
    <version>${spark.version}</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_${scala.binary.version}</artifactId>
    <version>${spark.version}</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-catalyst_${scala.binary.version}</artifactId>
    <version>${spark.version}</version>
    <scope>provided</scope>
</dependency>
 <!-- GeoTools -->
<dependency>
    <groupId>org.geotools</groupId>
    <artifactId>gt-geojson</artifactId>
    <version>23.0</version>
</dependency>
<dependency>
    <groupId>org.geotools</groupId>
    <artifactId>gt-metadata</artifactId>
    <version>23.0</version>
</dependency>
<dependency>
    <groupId>org.geotools</groupId>
    <artifactId>gt-referencing</artifactId>
    <version>23.0</version>
</dependency>
<dependency>
    <groupId>org.geotools.jdbc</groupId>
    <artifactId>gt-jdbc-postgis</artifactId>
    <version>23.0</version>
</dependency>
<dependency>
    <groupId>org.geotools</groupId>
    <artifactId>gt-epsg-hsql</artifactId>
    <version>23.0</version>
</dependency>
```

```
        <dependency>
            <groupId>com.aliyun.oss</groupId>
            <artifactId>aliyun-sdk-oss</artifactId>
            <version>3.9.0</version>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
            <plugin>
                <groupId>net.alchim31.maven</groupId>
                <artifactId>scala-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <id>scala-compile-first</id>
                        <phase>process-resources</phase>
                        <goals>
                            <goal>add-source</goal>
                            <goal>compile</goal>
                        </goals>
                    </execution>
                    <execution>
                        <id>scala-test-compile</id>
                        <phase>process-test-resources</phase>
                        <goals>
                            <goal>testCompile</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>
                    <compilerPlugins>
                        <compilerPlugin>
                            <groupId>org.spire-math</groupId>
                            <artifactId>kind-projector_2.11</artifactId>
                            <version>0.9.4</version>
                        </compilerPlugin>
                    </compilerPlugins>
                </configuration>
            </plugin>
        </plugins>
    </build>
     <repositories>
        <repository>
            <id>osgeo</id>
            <name>OSGeo Release Repository</name>
            <url>https://repo.osgeo.org/repository/release/</url>
            <snapshots><enabled>false</enabled></snapshots>
```

```
            <releases><enabled>true</enabled></releases>
        </repository>
        <repository>
            <id>osgeo-snapshot</id>
            <name>OSGeo Snapshot Repository</name>
            <url>https://repo.osgeo.org/repository/snapshot/</url>
            <snapshots><enabled>true</enabled></snapshots>
            <releases><enabled>false</enabled></releases>
        </repository>
    </repositories>
</project>
```

ii. 创建scala文件 `OSSTest.scala` ：

```
import com.aliyun.ganos.dla._
import com.aliyun.ganos.dla.raster._
import com.aliyun.ganos.dla.oss._
import com.aliyun.ganos.dla.geometry._
import com.typesafe.config.ConfigFactory
import org.apache.log4j.{Level, Logger}
import org.apache.spark.SparkConf
import org.apache.spark.sql.SparkSession
object OSSTest extends App {
  Logger.getLogger("org").setLevel(Level.ERROR)
  Logger.getLogger("com").setLevel(Level.ERROR)
  val spark: SparkSession = {
    val session = SparkSession.builder
      .withKryoSerialization
      .config(additionalConf)
      .getOrCreate()
    session
  }
  spark.withGanosGeometry
  spark.withGanosRaster
  val uri = new java.net.URI("oss://bucket名称/raster") //指定文件夹路径
  val options = Map(
    "crs"->"EPSG:4326",
    "endpoint" -> "EndPoint地址",
    "accessKeyId" -> "用户AccessKeyID",
    "accessKeySecret" -> "用户AccessKeySecret")
  val rf = spark.read.ganos.oss(options).loadLayer(uri)
  rf.show
  def additionalConf = new SparkConf(false)
}
```

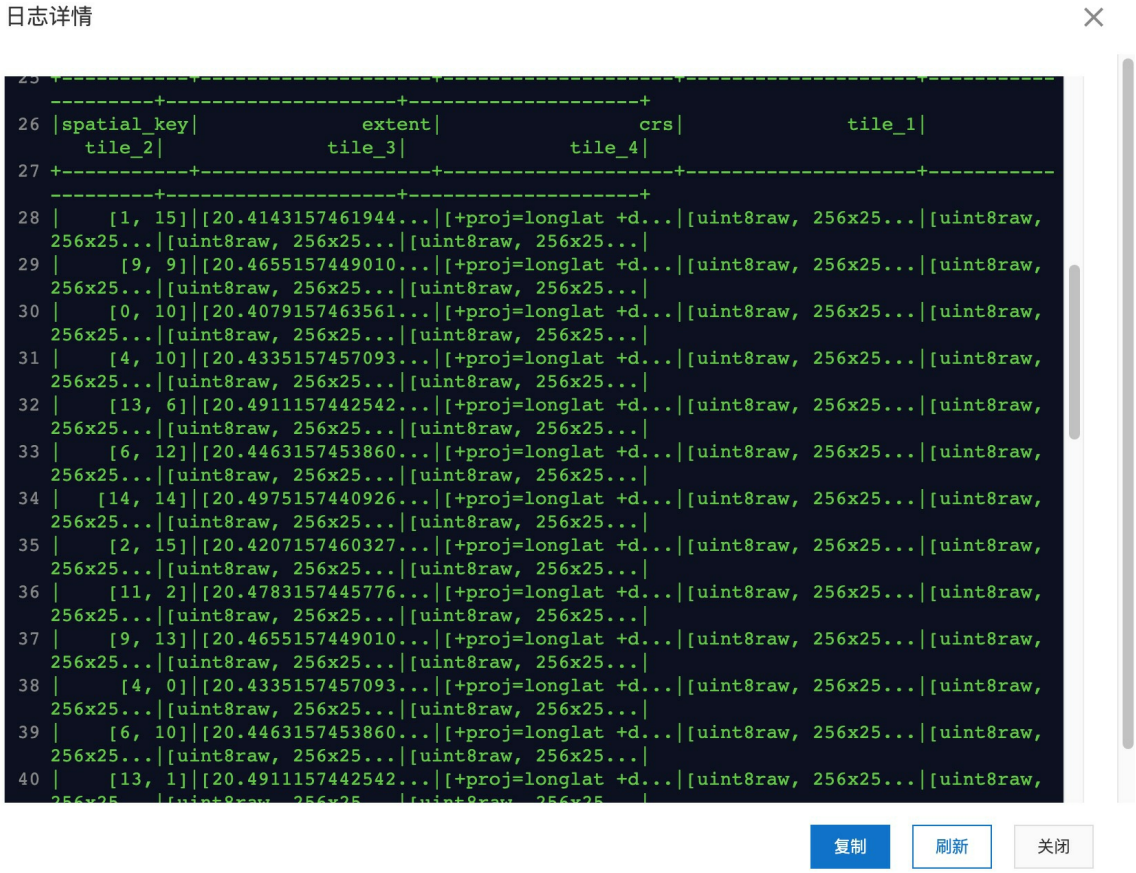
iii. 进行编译工程：

```
mvn clean package
```

3. 提交作业

登录Data Lake Analytics管理控制台提交Spark作业，详情请参见：创建和执行Spark作业。

保存运行成功后查看作业状态，可以看到原始tiff以Tile的形式加载到Spark中，然后您就可以对这些Tile进行相关操作了。



7.3.3. 连接数据源

7.3.3.1. OSS

阿里云对象存储OSS（Object Storage Service）是阿里云提供的海量、安全、低成本、高持久的云存储服务。本文主要介绍DLA Ganos如何支持加载存储在OSS中的Tiff文件为DataFrames。

操作步骤

1. 初始化Spark Session：

```
val spark: SparkSession = {
   val session = SparkSession.builder
     .master("local[*]")
     .withKryoSerialization
     .config(additionalConf)
     .getOrCreate()
   session
 }
//加载DLA Ganos Raster驱动
spark.withGanosRaster
```

2. 定义OSS连接参数并加载图层：

```
  val options = Map(
    "crs"->"EPSG:4326",
    "endpoint" -> "OSS Endpoint地址",
    "accessKeyId" ->"AccessKey Id",
    "accessKeySecret" -> "AccessKey Secret")
val uri=new java.net.URI("oss://<OSS Bucket名称>/srtm_60_05.tif") //指定OSS具体文件名称
val layer=spark.read.ganos.oss(options).loadLayer(uri)
```

3. 输出结果为：

```
+-----------+--------------------+--------------------+--------------------+--------------------+--------------------+--------------------+
|spatial_key|              extent|                 crs|            metadata|              tile_1|              tile_2|              tile_3|
+-----------+--------------------+--------------------+--------------------+--------------------+--------------------+--------------------+
|     [0, 0]|[703986.502389, 4...|[+proj=utm +zone=...|[AREA_OR_POINT ->...|ArrayTile(186,169...|ArrayTile(186,169...|ArrayTile(186,169...|
+-----------+--------------------+--------------------+--------------------+--------------------+--------------------+--------------------+
```

如果要批量加载OSS数据，只需要修改uri为OSS中的目录名称即可：

```
val uri=new java.net.URI("oss://<OSS Bucket名称>/raster") //指定OSS目录
val rf=spark.read.ganos.oss(options).loadLayer(uri)
rf.show
```

修改后的输出结果为：

```
+-----------+--------------------+--------------------+--------------------+--------------------+--------------------+--------------------+
|spatial_key|              extent|                 crs|              tile_1|              tile_2|              tile_3|              tile_4|
+-----------+--------------------+--------------------+--------------------+--------------------+--------------------+--------------------+
|    [1, 15]|[20.4143157461944...|[+proj=longlat +d...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|
|     [9, 9]|[20.4655157449010...|[+proj=longlat +d...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|
|    [0, 10]|[20.4079157463561...|[+proj=longlat +d...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|
|    [4, 10]|[20.4335157457093...|[+proj=longlat +d...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|
|    [13, 6]|[20.4911157442542...|[+proj=longlat +d...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|
|    [6, 12]|[20.4463157453860...|[+proj=longlat +d...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|
|   [14, 14]|[20.4975157440926...|[+proj=longlat +d...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|
|    [11, 2]|[20.4783157445776...|[+proj=longlat +d...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|
|     [4, 0]|[20.4335157457093...|[+proj=longlat +d...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|
|    [9, 13]|[20.4655157449010...|[+proj=longlat +d...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|
|    [2, 15]|[20.4207157460327...|[+proj=longlat +d...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|
|    [6, 10]|[20.4463157453860...|[+proj=longlat +d...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|
|    [13, 1]|[20.4911157442542...|[+proj=longlat +d...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|
|     [7, 7]|[20.4527157452243...|[+proj=longlat +d...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|
|     [0, 9]|[20.4079157463561...|[+proj=longlat +d...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|
|    [18, 8]|[20.5231157434459...|[+proj=longlat +d...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|
|   [11, 15]|[20.4783157445776...|[+proj=longlat +d...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|
|    [12, 8]|[20.4847157444159...|[+proj=longlat +d...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|
|     [9, 3]|[20.4655157449010...|[+proj=longlat +d...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|
|     [8, 9]|[20.4591157450626...|[+proj=longlat +d...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|ArrayTile(256,256...|
+-----------+--------------------+--------------------+--------------------+--------------------+--------------------+--------------------+
```

7.3.3.2. PolarDB

本文主要介绍DLA Ganos如何加载PolarDB中的栅格数据。

操作步骤

1. 初始化Spark Session

```
val spark: SparkSession = {
   val session = SparkSession.builder
     .master("local[*]")
     .withKryoSerialization
     .config(additionalConf)
     .getOrCreate()
   session
 }
//加载DLA Ganos Raster驱动。
spark.withGanosRaster
```

2. 初始化连接参数

```
val options = Map(
    "url" -> "jdbc:postgresql://121.43.235.228:5432/ganos_demodb",
    "user" -> "postgres",
    "password" -> "Ganos@2020",
    "dbtable" -> "gf",
    "numPartitions" -> "4")val options = Map(
    "url" -> "jdbc:postgresql://121.43.235.228:5432/ganos_demodb",
    "user" -> "postgres",
    "password" -> "Ganos@2020",
    "dbtable" -> "gf",
    "numPartitions" -> "4")
```

3. 加载CataLog

```
 val cat = spark.read.ganos.polardbRasterCatalog(options)
```

4. 输出结果如下

```
+---+--------------------+-------------------+----+--------+--------+--------+--------------------+--------------------+----+----+--------------------+
| id|                name|   acquisition_date|srid|celltype|numbands|toplevel|          pixelwidth|         pixelheight|rows|cols|            envelope|
+---+--------------------+-------------------+----+--------+--------+--------+--------------------+--------------------+----+----+--------------------+
|  1|GF1B_PMS_E119.4_N...|2019-01-05 18:10:...|4326|   16BUI|       4|      14|7.335940079126839E-5|7.335940079126793E-5|9664|11259|POLYGON ((118.989...|
|  8|GF1B_PMS_E119.2_N...|2019-01-02 15:25:...|4326|   16BUI|       4|      14|7.342978233994846E-5|7.342978233994792E-5|9670|11211|POLYGON ((118.814...|
|  9|GF1C_PMS_E118.9_N...|2019-01-04 12:34:...|4326|   16BUI|       4|      14|7.750144143609912E-5|7.750144143609898E-5|9641|11496|POLYGON ((118.486...|
| 21|GF1D_PMS_E118.3_N...|2019-01-02 14:42:...|4326|   16BUI|       4|      14|7.481303614424423E-5| 7.48130361442446E-5|9619|10998|POLYGON ((117.884...|
| 27|GF1C_PMS_E118.8_N...|2019-01-04 13:27:...|4326|   16BUI|       4|      14|7.734392730303184E-5|7.734392730303153E-5|9654|11449|POLYGON ((118.336...|
| 43|GF1B_PMS_E119.4_N...|2019-01-08 09:42:...|4326|   16BUI|       4|      14|7.335940079126839E-5|7.335940079126793E-5|9664|11259|POLYGON ((118.989...|
| 50|GF1B_PMS_E119.2_N...|2019-01-05 03:43:...|4326|   16BUI|       4|      14|7.342978233994846E-5|7.342978233994792E-5|9670|11211|POLYGON ((118.814...|
| 52|GF1B_PMS_E119.4_N...|2019-01-08 14:50:...|4326|   16BUI|       4|      14|7.335940079126839E-5|7.335940079126793E-5|9664|11259|POLYGON ((118.989...|
| 61|GF1B_PMS_E119.2_N...|2019-01-02 17:48:...|4326|   16BUI|       4|      14|7.342978233994846E-5|7.342978233994792E-5|9670|11211|POLYGON ((118.814...|
| 62|GF1C_PMS_E118.9_N...|2019-01-05 21:40:...|4326|   16BUI|       4|      14|7.750144143609912E-5|7.750144143609898E-5|9641|11496|POLYGON ((118.486...|
| 64|GF1C_PMS_E119.6_N...|2019-01-09 01:55:...|4326|   16BUI|       4|      14| 7.35984028741279E-5| 7.35984028741277E-5|9712|11172|POLYGON ((119.230...|
|  1|GF1B_PMS_E119.2_N...| 2016-06-22 19:10:25|4326|   16BUI|       4|      14|7.342978233994846E-5|7.342978233994792E-5|9670|11211|POLYGON ((118.814...|
+---+--------------------+-------------------+----+--------+--------+--------+--------------------+--------------------+----+----+--------------------+
```

CataLog中包含各个图层的ID和其他元数据信息，支持时空查询。

- 您可以通过指定ID和Zoom（默认z=0，即最底层金字塔）加载对应的图层：

```
val layer = spark.read.ganos.polardbRasterImage(options).load(id, zoom)
```

```
+-----------+--------------------+--------------------+--------------------+--------------------+--------------------+--------------------+
|spatial_key|              extent|                 crs|              tile_1|              tile_2|              tile_3|              tile_4|
+-----------+--------------------+--------------------+--------------------+--------------------+--------------------+--------------------+
|     [0, 0]|[118.188017696569...|[+proj=longlat +d...|[uint16, 513x513,...|[uint16, 513x513,...|[uint16, 513x513,...|[uint16, 513x513,...|
|     [0, 1]|[118.188017696569...|[+proj=longlat +d...|[uint16, 513x92, ...|[uint16, 513x92, ...|[uint16, 513x92, ...|[uint16, 513x92, ...|
|     [1, 0]|[118.820323846027...|[+proj=longlat +d...|[uint16, 201x513,...|[uint16, 201x513,...|[uint16, 201x513,...|[uint16, 201x513,...|
|     [1, 1]|[118.820323846027...|[+proj=longlat +d...|[uint16, 201x92, ...|[uint16, 201x92, ...|[uint16, 201x92, ...|[uint16, 201x92, ...|
+-----------+--------------------+--------------------+--------------------+--------------------+--------------------+--------------------+
```

> **注意** 多波段影像会自动拆分为多个单波段Tile进行展示。

- 您也可以通过图层ID集合批量加载图层：

```
val layers = spark.read.ganos.polardbRasterCover(options).load(Seq[id], zoom)
```

7.3.3.3. Lindorm（HBase）

本文主要介绍DLA Ganos如何基于GeoTrellis对Lindorm（HBase）中的栅格数据进行查询和计算。

GeoTrellis是一个基于Spark开发的高性能程序的地理数据处理引擎。GeoTrellis 能够非常快的读/写/操作栅格数据，同时也支持大量的几何操作。有关GeoTrellis的详细介绍请参见：https://geotrellis.readthedocs.io/en/latest/。

读取栅格数据

1. 初始化Spark Session：

```
val spark: SparkSession = {
   val session = SparkSession.builder
     .master("local[*]")
     .withKryoSerialization
     .config(additionalConf)
     .getOrCreate()
   session
 }
//加载DLA Ganos Raster驱动
spark.withGanosRaster
```

2. 通过geotrellisCatalog加载Catalog表：

```
val inputUri = "hbase://hb-proxy-pub-hbase实例id-001.hbase.rds.aliyuncs.com:2181?master=
localhost&attributes=GF1C"
val cat=spark.read.ganos.geotrellisCatalog(URI.create(inputUri))
cat.show
```

index	layer	format	keyClass	layerType	tileTable	valueClass	bounds	cellType	crs	extent	layoutDefinition	time
0	[[hbase://hb-prox...	hbase	geotrellis.layer....	AvroLayerType	GF1C_TILE	geotrellis.raster...	[[1, 1], [0, 0]]	uint16raw	+proj=longlat +da...	[120.209909649862...	[[120.58836695219...	1593313100051
0	[[hbase://hb-prox...	hbase	geotrellis.layer....	AvroLayerType	GF1C_TILE	geotrellis.raster...	[[5, 4], [0, 0]]	uint16raw	+proj=longlat +da...	[120.355585117979...	[[120.43129154214...	1593318640441
0	[[hbase://hb-prox...	hbase	geotrellis.layer....	AvroLayerType	GF1C_TILE	geotrellis.raster...	[[21, 18], [0, 0]]	uint16raw	+proj=longlat +da...	[120.014538864793...	[[120.02398437297...	1593333059146
0	[[hbase://hb-prox...	hbase	geotrellis.layer....	AvroLayerType	GF1C_TILE	geotrellis.raster...	[[0, 0], [0, 0]]	uint16raw	+proj=longlat +da...	[120.014538864793...	[[120.39887586043...	1593333059146
0	[[hbase://hb-prox...	hbase	geotrellis.layer....	AvroLayerType	GF1C_TILE	geotrellis.raster...	[[1, 1], [0, 0]]	uint16raw	+proj=longlat +da...	[120.209909649862...	[[120.58836695219...	1593313100051
0	[[hbase://hb-prox...	hbase	geotrellis.layer....	AvroLayerType	GF1C_TILE	geotrellis.raster...	[[5, 4], [0, 0]]	uint16raw	+proj=longlat +da...	[120.355585117979...	[[120.43129154214...	1593318640441
0	[[hbase://hb-prox...	hbase	geotrellis.layer....	AvroLayerType	GF1C_TILE	geotrellis.raster...	[[21, 18], [0, 0]]	uint16raw	+proj=longlat +da...	[120.014538864793...	[[120.02398437297...	1593333059146
0	[[hbase://hb-prox...	hbase	geotrellis.layer....	AvroLayerType	GF1C_TILE	geotrellis.raster...	[[0, 0], [0, 0]]	uint16raw	+proj=longlat +da...	[120.014538864793...	[[120.39887586043...	1593333059146
0	[[hbase://hb-prox...	hbase	geotrellis.layer....	AvroLayerType	GF1C_TILE	geotrellis.raster...	[[1, 1], [0, 0]]	uint16raw	+proj=longlat +da...	[120.209909649862...	[[120.58836695219...	1593313100051
0	[[hbase://hb-prox...	hbase	geotrellis.layer....	AvroLayerType	GF1C_TILE	geotrellis.raster...	[[5, 4], [0, 0]]	uint16raw	+proj=longlat +da...	[120.355585117979...	[[120.43129154214...	1593318640441
0	[[hbase://hb-prox...	hbase	geotrellis.layer....	AvroLayerType	GF1C_TILE	geotrellis.raster...	[[21, 18], [0, 0]]	uint16raw	+proj=longlat +da...	[120.014538864793...	[[120.02398437297...	1593333059146
0	[[hbase://hb-prox...	hbase	geotrellis.layer....	AvroLayerType	GF1C_TILE	geotrellis.raster...	[[0, 0], [0, 0]]	uint16raw	+proj=longlat +da...	[120.014538864793...	[[120.39887586043...	1593333059146
0	[[hbase://hb-prox...	hbase	geotrellis.layer....	AvroLayerType	GF1C_TILE	geotrellis.raster...	[[1, 1], [0, 0]]	uint16raw	+proj=longlat +da...	[120.209909649862...	[[120.58836695219...	1593313100051
0	[[hbase://hb-prox...	hbase	geotrellis.layer....	AvroLayerType	GF1C_TILE	geotrellis.raster...	[[5, 4], [0, 0]]	uint16raw	+proj=longlat +da...	[120.355585117979...	[[120.43129154214...	1593318640441
0	[[hbase://hb-prox...	hbase	geotrellis.layer....	AvroLayerType	GF1C_TILE	geotrellis.raster...	[[21, 18], [0, 0]]	uint16raw	+proj=longlat +da...	[120.014538864793...	[[120.02398437297...	1593333059146
0	[[hbase://hb-prox...	hbase	geotrellis.layer....	AvroLayerType	GF1C_TILE	geotrellis.raster...	[[0, 0], [0, 0]]	uint16raw	+proj=longlat +da...	[120.014538864793...	[[120.39887586043...	1593333059146
1	[[hbase://hb-prox...	hbase	geotrellis.layer....	AvroLayerType	GF1C_TILE	geotrellis.raster...	[[2, 2], [0, 0]]	uint16raw	+proj=longlat +da...	[120.209909649862...	[[120.28858719772...	1593313100051
1	[[hbase://hb-prox...	hbase	geotrellis.layer....	AvroLayerType	GF1C_TILE	geotrellis.raster...	[[10, 9], [0, 0]]	uint16raw	+proj=longlat +da...	[120.355585117979...	[[120.35617198948...	1593318640441
1	[[hbase://hb-prox...	hbase	geotrellis.layer....	AvroLayerType	GF1C_TILE	geotrellis.raster...	[[43, 37], [0, 0]]	uint16raw	+proj=longlat +da...	[120.014538864793...	[[120.02398437297...	1593333059146
1	[[hbase://hb-prox...	hbase	geotrellis.layer....	AvroLayerType	GF1C_TILE	geotrellis.raster...	[[1, 1], [0, 0]]	uint16raw	+proj=longlat +da...	[120.014538864793...	[[120.39887586043...	1593333059146

Catalog中包含各个图层的layer和其他元数据信息，支持属时空查询：

```
val layer = cat.where($"layer.id.zoom" === "2").select(raster_layer).collect
```

- 您可以通过catalog得到图层Layer并加载数据：

```
 val lots = layer.map(spark.read.ganos.geotrellis.loadLayer).map(_.toDF).reduce(_ uni
on _)
lots.show
```

spatial_key	extent	crs	tile_1	tile_2	tile_3	tile_4
[1, 15]	[20.4143157461944...	[+proj=longlat +d...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...
[9, 9]	[20.4655157449010...	[+proj=longlat +d...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...
[0, 10]	[20.4079157463561...	[+proj=longlat +d...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...
[4, 10]	[20.4335157457093...	[+proj=longlat +d...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...
[13, 6]	[20.4911157442542...	[+proj=longlat +d...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...
[6, 12]	[20.4463157453860...	[+proj=longlat +d...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...
[14, 14]	[20.4975157440926...	[+proj=longlat +d...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...
[11, 2]	[20.4783157445776...	[+proj=longlat +d...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...
[4, 0]	[20.4335157457093...	[+proj=longlat +d...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...
[9, 13]	[20.4655157449010...	[+proj=longlat +d...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...
[2, 15]	[20.4207157460327...	[+proj=longlat +d...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...
[6, 10]	[20.4463157453860...	[+proj=longlat +d...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...
[13, 1]	[20.4911157442542...	[+proj=longlat +d...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...
[7, 7]	[20.4527157452243...	[+proj=longlat +d...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...
[0, 9]	[20.4079157463561...	[+proj=longlat +d...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...
[18, 8]	[20.5231157434459...	[+proj=longlat +d...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...
[11, 15]	[20.4783157445776...	[+proj=longlat +d...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...
[12, 8]	[20.4847157444159...	[+proj=longlat +d...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...
[9, 3]	[20.4655157449010...	[+proj=longlat +d...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...
[8, 9]	[20.4591157450626...	[+proj=longlat +d...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...	ArrayTile(256,256...

- 也可以通过loadLayers接口批量加载图层：

```
val df = spark.read.ganos.geotrellis.loadLayers(layers)
```

输出栅格数据

您可以将DF模型输出到Lindorm表中：

```
lazy val layer = Layer(scratchDir.toUri, LayerId("test-layer", 4))
df.write.ganos.geotrellis.asLayer(layer).save()
```

7.3.4. 时空栅格函数参考

Tile属性操作

- st_dimensions

  获取Tile的长宽像素值。

  ```
  Struct[Int, Int] st_dimensions(Tile tile)
  ```

- st_cell_type

  回去Tile单元格数据类型。单元格数据类型可以通过st_convert_cell_type来改变。

  ```
  Struct[String] st_cell_type(Tile tile)
  ```

- st_tile

  从ProjectedRasterTile中获取完全加载的Tile。

  ```
  Tile st_tile(ProjectedRasterTile proj_raster)
  ```

  注意　ProjectedRasterTile为Tile的子类，是内置了空间范围与空间参考的Tile。

- st_extent

  获取Tile的空间范围。

  ```
  Struct[Double xmin, Double xmax, Double ymin, Double ymax] st_extent(ProjectedRasterTile
  proj_raster)
  Struct[Double xmin, Double xmax, Double ymin, Double ymax] st_extent(RasterSource proj_ra
  ster)
  ```

- st_crs

  获取空间参考CRS。从st_mk_crs支持的形式的字符串列中获取表示ProjectedRasterTile或RasterSource类型瓦片列的坐标参考系统的CRS结构。

  ```
  Struct st_crs(ProjectedRasterTile proj_raster)
  Struct st_crs(RasterSource proj_raster)
  Struct st_crs(String crs_spec)
  ```

- st_proj_raster

  通过指定的Tile，Extent和CRS列构造ProjectedRasterTile对象。

  ```
  ProjectedRasterTile st_proj_raster(Tile tile, Extent extent, CRS crs)
  ```

- st_mk_crs

  根据指定CRS表述生成CRS对象。

```
Struct st_mk_crs(String crsText)
```

这里crsText可以是如下类型：

- EPSG code: 格式为EPSG:Int
- Proj4 字符串: +proj <proj参数>
- WKT 字符串，嵌入 EPSG 代码: GEOGCS["<name>", <datum>, <prime meridian>, <angular unit> {,<twin axes>} {,<authority>}]

示例为：SELECT rf_mk_crs('EPSG:4326')

- **st_convert_cell_type**

  对Tile单元格类型进行类型转换

```
Tile st_convert_cell_type(Tile tile_col, CellType cell_type)
Tile st_convert_cell_type(Tile tile_col, String cell_type)
```

- **st_interpret_cell_type_as**

  根据指定的cell_type更改tile_col的单元格值的解释方式。在Python中，您可以将CellType对象传递给cell_type。

```
Tile st_interpret_cell_type_as(Tile tile_col, CellType cell_type)
Tile st_interpret_cell_type_as(Tile tile_col, String cell_type)
```

- **st_resample**

  更改Tile尺寸，参数factor为缩放比例，1.0等于原始Tile的列和行数；少于1对Tile进行向降采样；大于1对Tile进行升采样。传递shape_tile作为第二个参数将输出与shape_tile相同的行列号的tile。所有重采样都是通过最邻近方法进行。

```
Tile st_resample(Tile tile, Double factor)
Tile st_resample(Tile tile, Int factor)
Tile st_resample(Tile tile, Tile shape_tile)
```

矢量计算

- **st_extent**

  获取指定空间要素的外包框范围。

  - 函数定义：

```
Struct[Double xmin, Double xmax, Double ymin, Double ymax] st_extent(Geometry geom)
```

  - 示例：

```
val boxed = df.select(GEOMETRY_COLUMN, st_extent(GEOMETRY_COLUMN) as "extent")
```

- **st_reproject**

  将矢量geometry类型从origin_crs转化为destination_crs，所有crs必须以坐标系统国际标准格式表示，如Pro4j、EPSG或WKT等。

  - 函数定义：

```
Geometry st_reproject(Geometry geom, String origin_crs, String destination_crs)
```

- 示例：

```
SELECT st_reproject(ll, '+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs', 'EPSG:3857'
)
AS wm4 from geom
```

- **st_geometry**

  将extent对象转换为Geometry对象

  - 函数定义：

```
Geometry st_geometry(Struct[Double xmin, Double xmax, Double ymin, Double ymax] extent)
```

  - 示例：

```
val geom = df.select(st_geometry(st_extent(GEOMETRY_COLUMN)))
```

- **st_xz2_index**

  计算空间参考为WGS84/EPSG:4326的Geometry、Extent或ProjectedRasterTile对象的XZ2索引值。该函数可以用于数据计算的区间分块。

  - 函数定义：

```
Long st_xz2_index(Geometry geom, CRS crs)
Long st_xz2_index(Extent extent, CRS crs)
Long st_xz2_index(ProjectedRasterTile proj_raster)
```

  - 示例：

```
val indexes = df.select(st_xz2_index($"extent", serialized_literal(crs))).collect()
```

- **st_z2_index**

  计算空间参考为WGS84/EPSG:4326的Geometry、Extent或ProjectedRasterTile对象的Z2索引值。计算过程中，空间的对象的空间范围extent会被首先提取出来，然后就算extent中心点的Z2索引值。该函数可以用于数据计算的区间分块。在读取raster数据过程中会自动调用该函数进行计算。

  - 函数定义：

```
Long st_z2_index(Geometry geom, CRS crs)
Long st_z2_index(Extent extent, CRS crs)
Long st_z2_index(ProjectedRasterTile proj_raster)
```

  - 示例：

```
val crs: CRS = CRS.fromName("EPSG:4326")
val indexes = df.withColumn("xz2",st_z2_index($"extent", serialized_literal(crs)))
                                .withColumn("z2",st_z2_index($"extent", serialized_lit
eral(crs)))
indexes.show
```

- 输出结果：

```
+-----------+--------------------+-----------+-------------------+
|spatial_key|              extent|        xz2|                 z2|
+-----------+--------------------+-----------+-------------------+
|     [0, 0]|[118.336550697428...|77880735068|3919866226478294259|
|     [1, 0]|[118.356350742817...|77880736433|3919866232129151218|
|     [0, 1]|[118.336550697428...|77880732338|3919865820477424883|
|     [1, 1]|[118.356350742817...|77880733703|3919865826128281842|
|     [2, 0]|[118.376150788207...|77880736433|3919866249224287474|
|     [3, 0]|[118.395950833596...|77880740529|3919866437950910706|
|     [2, 1]|[118.376150788207...|77880733703|3919865843223418098|
|     [3, 1]|[118.395950833596...|77880737799|3919866031950041330|
|     [0, 2]|[118.336550697428...|77880614918|3919865683711796473|
|     [1, 2]|[118.356350742817...|77880616283|3919865689362653432|
|     [0, 3]|[118.336550697428...|77880615260|3919859775873615097|
|     [1, 3]|[118.356350742817...|77880616284|3919859781524472056|
|     [2, 2]|[118.376150788207...|77880616283|3919865706457789688|
|     [3, 2]|[118.395950833596...|77880620379|3919865895184412920|
|     [2, 3]|[118.376150788207...|77880616625|3919859798619608312|
|     [3, 3]|[118.395950833596...|77880620380|3919859987346231544|
|     [4, 0]|[118.415750878986...|77880740529|3919866454330795250|
|     [5, 0]|[118.435550924375...|77880741894|3919866505651495154|
|     [4, 1]|[118.415750878986...|77880737799|3919866048329925874|
|     [5, 1]|[118.435550924375...|77880739164|3919866099650625778|
+-----------+--------------------+-----------+-------------------+
```

Mask操作

- st_mask

  对tile做Mask操作，参数mask里面含有NoData信息。

```
Tile st_mask(Tile tile, Tile mask, bool inverse)
```

- st_mask_by_value

  对tile做Mask操作，当参数mask_tile中的值等于mask_value时，则值设定为NoData

```
Tile st_mask_by_value(Tile data_tile, Tile mask_tile, Int mask_value, bool inverse)
```

- st_mask_by_values

  返回一个Tile，其中数值来自data_tile，且mask_tile中等于mask_values中值的像素位置设为NoData

```
Tile st_mask_by_values(Tile data_tile, Tile mask_tile, Array mask_values)
Tile st_mask_by_values(Tile data_tile, Tile mask_tile, list mask_values)
```

Tile构造函数

- st_make_zeros_tile

  创建一个行列数分别为tile_rows和tile_columns的tile，填充值为0，数据类型为可选，默认单元格类型为float64。有关cell_type参数的信息，请参见有关单元格类型的讨论。所有参数都是文字表达式形式，而非列表达式。

```
Tile rf_make_zeros_tile(Int tile_columns, Int tile_rows, [CellType cell_type])
Tile rf_make_zeros_tile(Int tile_columns, Int tile_rows, [String cell_type_name])
```

- st_make_ones_tile

  创建一个行列数分别为tile_rows和tile_columns的tile，填充值为1，数据类型为可选，默认单元格类型为float64。有关cell_type参数的信息，请参见有关单元格类型的讨论。所有参数都是文字表达式形式，而非列表达式。

```
Tile st_make_ones_tile(Int tile_columns, Int tile_rows, [CellType cell_type])
Tile st_make_ones_tile(Int tile_columns, Int tile_rows, [String cell_type_name])
```

- st_make_constant_tile

  创建一个行列数分别为tile_rows和tile_columns的tile，填充值通过参数constant指定，数据类型为可选，默认单元格类型为float64。有关cell_type参数的信息，请参见有关单元格类型的讨论。所有参数都是文字表达式形式，而非列表达式。Tile st_make_constant_tile(Numeric constant, Int tile_columns, Int tile_rows, [CellType cell_type])Tile st_make_constant_tile(Numeric constant, Int tile_columns, Int tile_rows, [String cell_type_name])

```
Tile st_make_constant_tile(Numeric constant, Int tile_columns, Int tile_rows,  [CellType
cell_type])
Tile st_make_constant_tile(Numeric constant, Int tile_columns, Int tile_rows,  [String ce
ll_type_name])
```

- st_rasterize

  将Geometry类型的对象geom转换为Tile对象。在该Tile中，geom与tile_bounds相交部分将被赋予参数value定义的值。返回的图块的形状尺寸为tile_columns乘tile_rows。在geom之外的值将被分配一个NoData值。返回的图块的单元格类型为int32。请注意，输出Tile单元格的类型为Int。

```
Tile st_rasterize(Geometry geom, Geometry tile_bounds, Int value, Int tile_columns, Int t
ile_rows)
```

> **说明** 参数tile_columns和tile_rows是文字表达式，而不是列表达式。其他是列表达式。

- st_assemble_tile

  从给定的位置索引的单元格数据列中，创建大小为numRows和numCols的Tile图块。此函数与st_explode_tiles相反。预定用途是与groupby一起使用，每组产生一行带有新的Tile。有关可选cell_type参数的信息，请参见有关单元格类型的讨论。默认值为float64。

```
Tile st_assemble_tile(Int colIndex, Int rowIndex, Numeric cellData, Int numCols, Int numR
ows, [CellType cell_type])
Tile st_assemble_tile(Int colIndex, Int rowIndex, Numeric cellData, Int numCols, Int numR
ows, [String cell_type_name])
```

代数运算

- st_add

  波段加法运算

```
st_add(Tile tile1, Tile rhs)
st_add(Tile tile1, Int rhs)
st_add(Tile tile1, Double rhs)
```

- st_substract

  波段减法运算

```
st_substract(Tile tile1, Tile rhs)
st_substract(Tile tile1, Int rhs)
st_substract(Tile tile1, Double rhs)
```

- st_multiply

```
st_multiply(Tile tile1, Tile rhs)
st_multiply(Tile tile1, Int rhs)
st_multiply(Tile tile1, Double rhs)
```

- st_divide

```
st_divide(Tile tile1, Tile rhs)
st_divide(Tile tile1, Int rhs)
st_divide(Tile tile1, Double rhs)
```

- st_round

```
Tile st_round(Tile tile)
```

- st_exp

```
Tile st_exp(Tile tile)
```

- st_exp2

```
Tile st_exp2(Tile tile)
```

- st_exp10

```
Tile st_exp10(Tile tile)
```

- st_abs

```
Tile st_abs(Tile tile)
```

- st_log

```
Tile st_log(Tile tile)
```

- st_log10

```
Tile st_log10(Tile tile)
```

- st_sqrt

```
Tile st_sqrt(Tile tile)
```

- st_normalized_difference

  计算两个波段的归一化指数：(tile1-tile2)/(tile1+tile2)

```
Tile st_normalized_difference(Tile tile1,Tile tile2)
```

- st_rescale

  重新缩放单元格的值，使最小值为零，最大值为1，其他值将线性插值到该范围内。如果指定min与max，则min参数将变为0，max将变为1，超出范围的值将设置为0或1。如果未指定min和max，则使用Tile最小值和最大值。

```
Tile st_rescale(Tile tile)
Tile st_rescale(Tile tile, Double min, Double max)
```

- st_standardize

  使用mean和stddev标准化Tile，如果mean和stddev没有指定，则分别为0和1

```
st_standardize(Tile tile)
st_standardize(Tile tile, Double mean, Double stddev)
```

Tile聚合统计函数

- st_tile_mean

  计算Tile的平均值

```
Double st_tile_mean(Tile tile)
```

- st_tile_max

  计算Tile中最大值

```
Double st_tile_max(Tile tile)
```

- st_tile_min

  计算Tile中做小值

```
Double st_tile_min(Tile tile)
```

- st_tile_sum

  计算Tile中像素值之和

```
Double st_tile_sum(Tile tile)
```

- st_tile_stats

  在Tile上聚合并返回单元格值的统计信息，包括：Cell数量，NoData数量，最小值，最大值，均值和方差。最小值，最大值，均值和方差计算过程中忽略NoData。

```
Struct[Long, Long, Double, Double, Double, Double] st_tile_stats(Tile tile)
```

  示例为：

```
spark.sql("SELECT rf_tile_stats(rf_make_ones_tile(5, 5, 'float32')) as tile_stats").print
Schema()
```

- st_tile_approx_histogram

  统计Tile的直方图信息

```
Struct[Array[Struct[Double, Long]]] st_tile_approx_histogram(Tile tile)
```

- st_agg_extent

  根据各Tile的Extent聚合成整体的Extent

```
Extent st_agg_extent(Extent extent)
```

- st_agg_reprojected_extent

  根据各Tile的Extent聚合成整体的Extent，并进行重投影

```
Extent st_agg_reprojected_extent(Extent extent, CRS source_crs, String dest_crs)
```

Tile转换

- st_explode_tiles

  在Tile列中为每个单元格创建一行。可以传入多个Tile列，并且返回的DataFrame对象中每个输入将具有一个数字列。也将有column_index和row_index的列。与rf_assemble_tile功能相反。使用此功能时，请确保对行具有唯一的标识符，以便成功反转操作。

```
Int, Int, Numeric* st_explode_tiles(Tile* tile)
```

- st_tile_to_array_int

  按行优先（row-major）顺序将Tile列转换为Spark SQL Array类型。Double类型单元格将强制为Int类型。

```
Array st_tile_to_array_int(Tile tile)
```

- st_tile_to_array_double

  按行优先（row-major）顺序将tile列转换为Spark SQL Array。Int类型单元格将强制转换为浮点型。

```
Array st_tile_to_arry_double(Tile tile)
```

- st_render_ascii

  将Tile打印为ASCII纯文本。

```
String rf_render_ascii(Tile tile)
```

- rf_render_matrix

  将Tile单元格值输出为一串数字值。

```
String st_render_matrix(Tile tile)
```

- st_render_png

  RGB三个波段Tile渲染为PNG bytearray

```
Array st_render_png(Tile red, Tile green, Tile blue)
```

- st_render_color_ramp_png

  指定Tile和配色方案，输出PNG bytearray

```
Array st_render_color_ramp_png(Tile tile, String color_ramp_name)
```

color_ramp_name可包括：

- “BlueToOrange”
- “LightYellowToOrange”
- “BlueToRed”
- “GreenToRedOrange”
- “LightToDarkSunset”
- “LightToDarkGreen”
- “HeatmapYellowToRed”
- “HeatmapBlueToYellowToRedSpectrum”
- “HeatmapDarkRedToYellowWhite”
- “HeatmapLightPurpleToDarkPurpleToWhite”
- “ClassificationBoldLandUse”
- “ClassificationMutedTerrain”
- “Magma”
- “Inferno”
- “Plasma”
- “Viridis”
- “Greyscale2”
- “Greyscale8”
- “Greyscale32”
- “Greyscale64”
- “Greyscale128”
- “Greyscale256”
- RT_OverviewRaster

根据用户指定的AOI，采用双线性差值方法，按照cols和rows定义尺寸生成overview。如果Tile中包含了Extent和CRS，则tile_extent_col和tile_crs_col参数可选

```
Tile ST_OverviewRaster(Tile proj_raster_col, int cols, int rows, Extent aoi)
Tile ST_OverviewRaster(Tile tile_col, int cols, int rows, Extent aoi, Extent tile_extent_
col, CRS tile_crs_col)
```

- **st_rgb_composite**

  RGB三个波段Tile渲染为一个RGB Tile。处理过程为先将每个单元转换为0-255范围的无符号byte类型，然后合并所有三个通道输出32-bit无符号整数。

  ```
  Tile st_rgb_composite(Tile red, Tile green, Tile blue)
  ```

DataFrame API接口

部分功能不支持SQL函数方式，采用DataFrame API方式直接调用。

- **stitch**

  进行拼图操作，将多个Tile合并为一个Tile

  ```
  def stitch(extentCol: Column, tileCol: Column): Tile
  ```

使用案例：

```
val result = spark.sql("SELECT rows,cols,envelope,st_ndvi(red,nir) as ndvi FROM gf2")
result.stitch($"envelope", $"ndvi").renderPng(ndviColorMap).write("polardb_ndvi.png")
```

- transformToZoomLayer

  单波段时空拼图，并转换为zoom方式输出为rdd模型。

```
def transformToZoomLayer(rowCol: Column,
                         colCol: Column,
                         extentCol: Column,
                         tileCol: Column)
:(Int, RDD[(SpatialKey, Tile)] with Metadata[TileLayerMetadata[SpatialKey]])
```

- transformToMultibandZoomLayer

  多波段时空拼图，并转换为zoom方式输出为rdd模型。

```
def transformToMultibandZoomLayer(rowCol: Column,
                                  colCol: Column,
                                  extentCol: Column,
                                  tileCol: Column)
:(Int, RDD[(SpatialKey, MultibandTile)] with Metadata[TileLayerMetadata[SpatialKey]])
```

使用案例：

```
val result = spark.sql("SELECT rows,cols,envelope,st_ndvi(red,nir) as ndvi FROM gf2")
val (zoom, rdd1): (Int, RDD[(SpatialKey, Tile)] with Metadata[TileLayerMetadata[SpatialKe
y]]) = result.stitchToZoomLayer($"rows",$"cols",$"envelope", $"ndvi")
val path = "result" //按图层输出到指定文件夹
  val layoutScheme = ZoomedLayoutScheme(WebMercator, tileSize = 256)
  Pyramid.upLevels(rdd1.asInstanceOf[RDD[(SpatialKey, Tile)] with Metadata[TileLayerMetad
ata[SpatialKey]]], layoutScheme, zoom, NearestNeighbor) { (rdd, z) =>
    val layerId = LayerId("", z)
    println(layerId)
    new File("result/" + z).mkdir()
    rdd.collect().foreach(elem => {
      elem._2.renderPng(ndviColorMap).write(path + "/" + z + "/" + elem._1.col + "_" + el
em._1.row + ".png")
    })
  }
```

7.3.5. 应用案例

您可以在GitHub上获取时空数据分析的典型案例，以便快速熟悉DLA Ganos的各项功能。

栅格代数运算

栅格代数运算是指使用数学运算符对栅格数据进行加减乘除等代数计算的操作。例如，您可以应用简单的数学运算（例如加法或乘法）来更新栅格像元值，或者对多个栅格数据图层执行叠加计算（Overlay）等。栅格代数运算中最常见的类型是像素单元函数，即栅格单元直接堆叠在一起进行计算。该功能适用于分辨率相同的栅格图层之间的叠加计算，详情请参见栅格代数运算和Code。

Masking

掩膜（Masking）是栅格处理中的常见操作，它将某些单元格设置为NoData值。掩膜操作一般有两个目的，首先是为了从栅格处理中删除低质量的观测值，实现去噪的目的；另一个是按照给定的形状将多边形剪切到指定形状。详情请参见Masking和Code。

自定义UDF

用户可以通过自定义UDF来扩展DLA Ganos的功能，从而更加方便地与业务系统对接。这里还是以NDVI为例，来展示如何基于DLA Ganos实现用户自定义的UDF算子。详情请参见自定义UDF和Code。

多源异构栅格Join

在复杂的业务系统中，往往需要对多个数据源进行联邦分析。这些数据具有不同的投影信息与分辨率，所以在分析之前需要对数据进行重投影与重采样等转换操作。在许多使用情况下，这种转换操作都是基于某一组遥感影像数据进行的。在DLA Ganos中，可以对多源栅格数据类型DataFrame执行RasterJoin操作。该操作将基于CRS将每个DataFrame中的Tile列执行空间连接操作。默认情况下是左连接，并使用交运算符，右侧的所有Tile列会匹配左侧的Tile列的CRS、范围和分辨率等。 详情请参见多源异构栅格Join和Code。

OSS与Lindorm数据源

DLA Ganos可以用来构建ETL工具，实现数据在不同数据库之间的流转。如用户可以将数据上传到OSS，然后进行重投影、拼接、创建金字塔并写入Lindorm（HBase）等。相关代码链接请参见Code。

机器学习

本节我们展示如何基于DLA Ganos和SparkML进行机器学习等操作。遥感科学中最常见的一类机器学习操作是监督分类，又称训练分类法。监督分类是用被确认类别的样本像元去识别其他未知类别像元的过程。它就是在分类之前通过目视判读和野外调查，对遥感图像上某些样区中影像地物的类别属性有了先验知识，对每一种类别选取一定数量的训练样本，计算机计算每种训练样区的统计或其他信息，同时用这些种子类别对判决函数进行训练，使其符合于对各种子类别分类的要求，随后用训练好的判决函数去对其他待分数据进行分类。详情请参见机器学习和Code。

7.4. 应用案例与最佳实践

7.4.1. Lindorm（HBase）数据入库与ETL

矢量数据入库

Lindorm（HBase）矢量数据导入，请参见快速入门。

栅格数据入库

1. **Pipeline技术**

   Pipeline模型是DLA Ganos基于GeoTrellis开源项目开发的用于栅格数据快速加载、处理和入库的ETL技术，详情请参见https://pdal.io/pipeline.html

   Pipeline模型包含了一系列功能模块：如读取数据（Load），转换（Transform），保存数据（Save）等。DLA Ganos Pipeline模型一般表示为一个JSON对象，其主要对象称为pipeline，该对象是要执行的步骤数组（还有一些JSON对象，我们将其称为Stage Objects）。DLA Ganos整个入库操作的Pipeline流程与相关参数全部通过一个JSON对象进行定义，一个简单的JSON脚本如下所示：

```
[
  {
    "uri" : "OSS资源URI",
    "type" : "singleband.spatial.read.oss"
  },
  {
    "resample_method" : "nearest-neighbor",
    "type" : "singleband.spatial.transform.tile-to-layout"
  },
  {
    "crs" : "EPSG:3857",
    "scheme" : {
      "crs" : "epsg:3857",
      "tileSize" : 256,
      "resolutionThreshold" : 0.1
    },
    "resample_method" : "nearest-neighbor",
    "type" : "singleband.spatial.transform.buffered-reproject"
  },
  {
    "end_zoom" : 0,
    "resample_method" : "nearest-neighbor",
    "type" : "singleband.spatial.transform.pyramid"
  },
  {
    "name" : "mask",
    "uri" : "oss://geotrellis-test/colingw/pipeline/",
    "key_index_method" : {
      "type" : "zorder"
    },
    "scheme" : {
      "crs" : "epsg:3857",
      "tileSize" : 256,
      "resolutionThreshold" : 0.1
    },
    "type" : "singleband.spatial.write"
  }
]
```

2. **入库流程**

   i. 导入相关依赖。

```
import geotrellis.layer._
import geotrellis.spark.pipeline._
import geotrellis.spark.pipeline.json._
import geotrellis.spark._
import geotrellis.spark.store.kryo.KryoRegistrator
import org.apache.spark.{SparkConf, SparkContext}
import scala.util.{Failure, Try}
```

ii. 初始化Spark环境。

```
val conf =
  new SparkConf ()
    .setMaster ("local[*]")
    .setAppName ("Spark Tiler")
    .set ("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
    .set ("spark.kryo.registrator", classOf[KryoRegistrator].getName)
conf.set ("spark.kryoserializer.buffer.max", "2047m")
implicit val sc = new SparkContext (conf)
```

iii. 定义Pipeline JSON描述。

以下示例是一个简单的pipeline模型，该模型定义的操作如下：

- 定义导入文件的URI与加载驱动。
- 数据分块模式（tile-to-layout）。
- 数据转换与冲投影等操作。
- 数据写入地址（Lindorm）。

该模型的详细配置如下所示：

```
 val pipeline: String =
    """
      |[
      |  {
      |    "uri" : "OSS资源URI",
      |    "time_tag" : "TIFFTAG_DATETIME",
      |    "time_format" : "yyyy:MM:dd HH:mm:ss",
      |    "type" : "singleband.spatial.read.hadoop"
      |  },
      |  {
      |    "resample_method" : "nearest-neighbor",
      |    "type" : "singleband.spatial.transform.tile-to-layout"
      |  },
      |  {
      |    "crs" : "EPSG:3857",
      |    "scheme" : {
      |      "crs" : "EPSG:3857",
      |      "tileSize" : 256,
      |      "resolutionThreshold":0.1
      |    },
      |    "resample_method" : "nearest-neighbor",
      |    "type" : "singleband.spatial.transform.buffered-reproject"
      |  },
      |  {
      |    "end_zoom" : 0,
      |    "resample_method" : "nearest-neighbor",
      |    "type" : "singleband.spatial.transform.pyramid"
      |  },
      |  {
      |    "name" : "srtm",
      |    "uri" : "hbase://localhost:2181?master=localhost&attributes=attributes&l
ayers=srtm-tms-layers",
      |    "pyramid" : true,
      |    "key_index_method" : {
      |      "type" : "zorder"
      |    },
      |    "scheme" : {
      |      "tileCols" : 256,
      |      "tileRows" : 256
      |    },
      |    "type" : "singleband.spatial.write"
      |  }
      |]
    """.stripMargin
```

iv. 运行Pipeline模型。

```
//首先解析JSON描述的Pipeline模型，生成表达式集合：
val list: List[PipelineExpr] = pipeline.pipelineExpr match {
    case Right (r)  => r
    case Left (e)  => throw e
}
//执行pipeline模型：
val erasedNode = list.erasedNode
  Try {
    erasedNode.eval[Stream[ (Int, TileLayerRDD[SpatialKey]) ]]
  } match {
    case Failure (e)  => println ("run failed as expected") ; throw e
    case _ =>
  }
```

配置文件参考

数据加载objects

```
{
   "uri" : "{oss| file | hdfs | ...}://...",
   "time_tag" : "TIFFTAG_DATETIME", // optional field
   "time_format" : "yyyy:MM:dd HH:mm:ss", // optional field
   "type" : "{singleband | multiband}.{spatial | temporal}.read.{oss | hadoop}"
}
```

参数说明如下：

Key	Value
uri	栅格数据源URI
time_tag	数据集元数据中的时间标签名称
type	操作类型

> **说明** 这里只有两种类型的读取器可用：通过Hadoop API从S3或从Hadoop支持的文件系统中读取。

数据写入objects

```
{
   "name" : "layerName",
   "uri" : "{oss| file | hdfs | ...}://...",
   "key_index_method" : {
      "type" : "{zorder | hilbert}",
      "temporal_resolution": 1 // optional, if set - temporal index is used
   },
   "scheme" : {
      "crs" : "epsg:3857",
      "tileSize" : 256,
      "resolutionThreshold" : 0.1
   },
   "type" : "{singleband | multiband}.{spatial | temporal}.write"
}
```

参数说明如下：

Key	Value
uri	栅格数据源URI
name	图层名称
key_index_method	从空间键（Satial Key）生成索引的键索引方法
key_index_method.type	填充曲线类型：zorder, row-major, hilbert
key_index_method. tmporal_resolution	时间分辨率（单位：毫秒ms）
scheme	目标layout scheme
scheme.crs	目标scheme的crs参数
scheme.tileSize	layout scheme 数据块Tile尺寸
scheme.resolutionThreshold	用户定义的布局方案的分辨率（可选字段）

> **说明** 这里只有两种类型的读取器可用：通过Hadoop API从OSS或Hadoop支持的文件系统中读取。

数据转换objects

- Tile To Layout

```
{
   "resample_method" : "nearest-neighbor",
   "type" : "{singleband | multiband}.{spatial | temporal}.transform.tile-to-layout"
}
```

> **说明** 将RDD[（{ProjectedExtent | TemporalProjectedExtent}, {Tile | MultibandTile}）] 转换为RDD[（{SpatialKey | SpaceTimeKey}, {Tile | MultibandTile}）]模型

参数说明如下：

Key	Options
resample_method	重采样方法：nearest-neighborbilinearcubic-convolutioncubic-splinelanczos

- ReTile To Layout

```
{
   "layout_definition": {
      "extent": [0, 0, 1, 1],
      "tileLayout": {
         "layoutCols": 1,
         "layoutRows": 1,
         "tileCols": 1,
         "tileRows": 1
      }
    },
   "resample_method" : "nearest-neighbor",
   "type" : "{singleband | multiband}.{spatial | temporal}.transform.retile-to-layout"
}
```

> **说明** 将 RDD[({SpatialKey | SpaceTimeKey}, {Tile | MultibandTile})] 对象按照用户配置的layout definition规则进行重新分块。

- Buffered Reproject

```
{
   "crs" : "EPSG:3857",
   "scheme" : {
      "crs" : "epsg:3857",
      "tileSize" : 256,
      "resolutionThreshold" : 0.1
   },
   "resample_method" : "nearest-neighbor",
   "type" : "{singleband | multiband}.{spatial | temporal}.transform.buffered-reproject"
}
```

> **说明** 将 RDD[({SpatialKey | SpaceTimeKey}, {Tile | MultibandTile})] 对象按照用户配置的layout scheme参数转换为目标CRS 数据分块。

参数说明如下：

Key	Options
crs	目标scheme的crs参数
tileSize	layout scheme 数据块Tile尺寸
resolutionThreshold	用户定义的布局方案的分辨率（可选字段）

Key	Options
resample_method	重采样方法：nearest-neighborbilinearcubic-convolutioncubic-splinelanczos

- Per Tile Reproject

```
{
   "crs" : "EPSG:3857",
   "scheme" : {
      "crs" : "epsg:3857",
      "tileSize" : 256,
      "resolutionThreshold" : 0.1
   },
   "resample_method" : "nearest-neighbor",
   "type" : "{singleband | multiband}.{spatial | temporal}.transform.per-tile-reproject"
}
```

> **说明** 将 RDD[({ProjectedExtent | TemporalProjectedExtent}, {Tile | MultibandTile})] 对象按照用户配置的layout scheme参数转换为目标CRS 数据分块。

参数说明如下：

Key	Options
scheme	目标layout scheme
scheme.crs	目标scheme的crs参数
scheme.tileSize	layout scheme 数据块Tile尺寸
scheme. resolutionThreshold	用户定义的布局方案的分辨率（可选字段）
resample_method	重采样方法：nearest-neighborbilinearcubic-convolutioncubic-splinelanczos

- Pyramid

```
{
   "end_zoom" : 0,
   "resample_method" : "nearest-neighbor",
   "type" : "{singleband | multiband}.{spatial | temporal}.transform.pyramid"
}
```

> **说明** 将RDD[({SpatialKey | SpaceTimeKey}, {Tile | MultibandTile})] 对象创建金字塔，直到end_zoom 定义层级为止，返回类型为Stream[RDD[({SpatialKey | SpaceTimeKey}, {Tile | MultibandTile})]].

关于Layout Scheme

LA Ganos 支持两种Layout Scheme模式：

- ZoomedLayoutScheme

  匹配TMS金字塔

  > 注意 ZoomedLayoutScheme需要知道从CRS获取的世界范围，以便构建TMS金字塔布局。这可能会导致重新采样输入栅格以匹配TMS级别的分辨率。

- FloatingLayoutScheme

  匹配输入栅格的原始分辨率。

  > 注意 FloatingLayoutScheme将发现本机分辨率和范围，并按给定的图块大小对其进行分区，而无需重新采样。

7.4.2. 自定义UDF计算NDVI

除了DLA Ganos本身提供的UDF算子之外，您也可以通过自定义UDF来扩展DLA Ganos的功能，从而更加方便地与业务系统对接。

操作步骤

本文以计算归一化植被指数（NDVI）为例，来展示如何基于DLA Ganos实现用户自定义的UDF算子。

1. **初始化SparkSession与DLA Ganos**

```
implicit val spark = SparkSession
  .builder()
  .master("local[*]")
  .appName(getClass.getName)
  .withKryoSerialization
  .getOrCreate()
  .withGanosRaster
import spark.implicits._
```

2. **加载栅格数据图层**

```
def readTiff(name: String) =
    SinglebandGeoTiff(IOUtils.toByteArray(getClass.getResourceAsStream(s"/$name")))
def redBand = readTiff("L8-B4-Elkton-VA.tiff").projectedRaster.toLayer("red_band")
def nirBand = readTiff("L8-B5-Elkton-VA.tiff").projectedRaster.toLayer("nir_band")
```

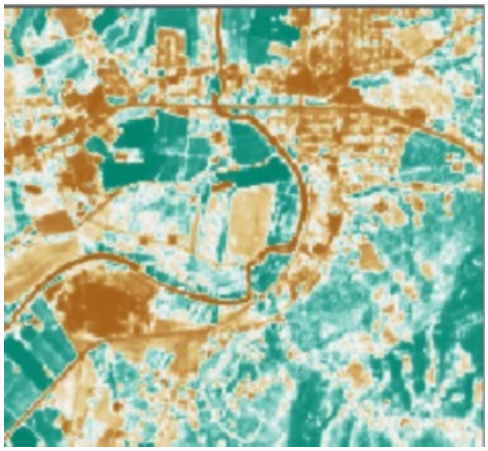
3. **自定义UDF算子**

```
val ndvi = udf((red: Tile, nir: Tile) => {
   val redd = red.convert(DoubleConstantNoDataCellType)
   val nird = nir.convert(DoubleConstantNoDataCellType)
   (nird - redd) / (nird + redd)
 })
```

4. **计算NDVI并输出结果**

```
val df = redBand.spatialJoin(nirBand).withColumn("ndvi", ndvi($"red_band", $"nir_band")
).asLayer
val pr = df.toRaster($"ndvi", 233, 214)
val brownToGreen = ColorRamp(
    RGB(166, 97, 26),
    RGB(223, 194, 125),
    RGB(245, 245, 245),
    RGB(128, 205, 193),
    RGB(1, 133, 113)
  ).stops(128)
val colors = ColorMap.fromQuantileBreaks(pr.tile.histogramDouble(), brownToGreen)
pr.tile.color(colors).renderPng().write("ndvi.png")
```

结果输出如下图所示：



8.性价比白皮书

8.1. 测试环境

本次测试采用3种不同的测试场景，针对开源自建的Hadoop+Spark集群与阿里云云原生数据湖分析DLA Spark在执行Terasort基准测试的性能做了对比分析。本文档主要介绍了3种不同测试场景下的测试环境配置要求。

环境配置要求

测试环境总体要求：

- 自建Hadoop+Spark集群的网络环境为VPC网络。
- 自建Hadoop+Spark集群和DLA Spark在同一个地域。
- 自建Spark集群请使用Spark 2.4.5版本，自建Hadoop请使用2.7.3版本。

3种不同测试场景下的测试环境配置要求：

- **场景一：1 TB测试数据下DLA Spark+OSS与自建Hadoop+Spark集群性能对比**

  场景说明：每天跑一次Terasort 1 TB基准测试，连续运行一个月，自建Hadoop+Spark集群用包年包月来进行计费，DLA Spark+OSS按量来进行计费。对比自建Hadoop+Spark集群和DLA Spark+OSS的费用价格，以及它们运行完Terasort基准测试的耗时，来进行性能对比分析。

  DLA Spark+OSS配置如下：

配置名称	规格要求	数量
Driver	medium（2核8 GB）	1个
Executor	medium（2核8 GB）	19个
OSS	无	2 TB的存储空间

  自建Hadoop+Spark集群配置如下：

配置名称	规格要求	数量
Master	4核16 GB（机型为ecs.g5.xlarge）	2个
Slave	8核32 GB，4个500 GB的高效云盘（机型为ecs.g6.2xlarge）	5个

> 说明
> - 进行TeraSort基准测试预计需要使用：输入1 TB+shuffle大约1 TB+输出1 TB。
> - 自建Hadoop+Spark集群的存储采用一个总容量为5 TB的典型配置。磁盘大小为 4*500 GB*5=10 TB，其中高效云盘采用双备份的HDFS配置，因此可用大小为5 TB。一般情况下集群磁盘使用率不能太高，一般建议不要超过80%，否则系统可能会因为空间不足，引发各类稳定性问题。
> - DLA Spark按需使用存储空间和计算资源。其中shuffle不占用OSS存储空间，输入和输出各占用1 TB，共需要占用2 TB的OSS存储空间。

- **场景二：10 TB测试数据下DLA Spark+OSS与自建Hadoop+Spark性能对比**

  场景说明：每天跑一次Terasort 1 TB基准测试，连续运行一个月，自建Hadoop+Spark集群用包年包月来进行计费，DLA Spark+OSS按量来进行计费。对比自建Hadoop+Spark集群和DLA Spark+OSS的费用价格，以及它们运行完Terasort基准测试的耗时，来进行性能对比分析。

  DLA Spark+OSS配置如下：

配置名称	规格要求	数量
Driver	medium（2核8 GB）	1个
Executor	medium（2核8 GB），200 GB的ESSD数据盘	39个
OSS	无	30 TB的存储空间

  自建Hadoop+Spark集群配置如下：

配置名称	规格要求	数量
Master	4核16 GB（机型为ecs.g5.xlarge）	2个
Slave	16核64 GB，8个5.5 TB的本地盘（机型为ecs.d1ne.4xlarge）	5个

> 说明
> - 进行TeraSort基准测试预计需要使用：输入10 TB+shuffle大约 10 TB+输出10 TB。
> - 自建Hadoop+Spark集群的存储采用的是大数据量场景下的典型配置，采用本地盘D1机型，成本相对于云盘更便宜。由于本地盘机型要求的空间比较大，16核64 GB只能配置44 TB的本地盘，一般本地盘采用3备份的HDFS配置，所以可用的存储空间为5.5 TB*8*5/3=73 TB。
> - DLA给每个Executor配置了一个200 G的ESSD盘，用于存放Executor shuffle的数据，从而不占用OSS的存储空间。
> - 本测试环境需要的OSS实际存储空间为20 TB，但由于考虑到Hadoop集群的本地盘通常不能占满，为了与自建Hadoop进行对比测试，采用的OSS存储空间为30 TB。

- **场景三：1 TB测试数据下DLA Spark+用户自建Hadoop集群与自建Hadoop+Spark性能对比**

  场景说明：使用自建Spark和DLA Spark分别访问自建Hadoop集群，运行Terasort 1 TB基准测试，对它们的耗时进行对比分析。

DLA Spark+OSS配置如下：

配置名称	规格要求	数量
Driver	medium（2核8 GB）	1个
Executor	medium（2核8 GB）	39个

自建Hadoop+Spark集群配置如下：

配置名称	规格要求	数量
Master	4核8 GB	2个
Slave	8核32 GB，4个500 GB的高效云盘	5个

> 说明
> - DLA Spark可以和自建Hadoop配合使用，实现为用户的Hadoop集群加弹性的目的。
> - 本测试中用户自建的Spark集群和DLA Spark都采用40核160 GB的配置。

8.2. 测试方法

本次测试采用3种不同的测试场景，针对开源自建的Hadoop+Spark集群与阿里云云原生数据湖分析DLA Spark在执行Terasort基准测试的性能做了对比分析。您可以按照本文介绍自行测试对比，快速了解云原生数据湖分析（DLA）Spark引擎的性价比数据。

背景信息

本文档将分别针对3种测试场景进行Terasort基准测试。3种测试场景的场景说明和配置要求，请参考环境配置要求。

准备工作

1. 下载Terasort测试相关的Jar包。

   在使用Spark进行Terasort基准测试时，需要准备测试数据和Terasort测试程序，DLA提供了一个Jar包，里面包含了可以生成Terasort测试数据的Spark应用和进行Terasort基准测试的Spark应用。

2. 将下载下来的Jar包上传到您的OSS上。

   在后续测试过程中，DLA Spark需要使用这个Jar包来生成Terasort测试数据以及进行Terasort基准测试，因此需要把该Jar包上传到您的OSS上。

操作步骤

场景一：1 TB测试数据下DLA Spark+OSS与自建Hadoop+Spark集群性能对比

1. 准备测试数据
   - 在OSS上生成1 TB Terasort测试数据

     登录Data Lake Analytics管理控制台，在**Serverless Spark > 作业管理**页签下，提交运行生成1 TB Terasort测试数据的Spark作业。示例如下：

```
{
    "args": [
        "1000g",
        "oss://<bucket-name>/<输出文件夹路径>",  #您的Terasort测试数据存放的OSS路径，例如oss://test-bucket/terasort/input/1T。
        "true"
    ],
    "file": "<Jar包的OSS路径>",  #上文中您Jar包上传的OSS路径，例如oss://test/performance/dla-spark-perf.jar。
    "name": "TeraGen-1T",
    "className": "com.aliyun.dla.perf.terasort.TeraGen",
    "conf": {
        "spark.dla.connectors": "oss",
        "spark.hadoop.job.oss.fileoutputcommitter.enable": "true",
        "spark.hadoop.mapreduce.fileoutputcommitter.algorithm.version": 2,
        "spark.driver.resourceSpec": "medium",
        "spark.executor.resourceSpec": "medium",
        "spark.default.parallelism": "2000",
        "spark.executor.memoryOverhead": 2000,
        "spark.executor.instances": 19
    }
}
```

- 在自建Hadoop上生成1 TB Terasort测试数据

  使用spark-submit命令向自建Spark集群中提交运行生成1 TB Terasort测试数据的Spark程序。示例如下：

```
./bin/spark-submit \
--class com.aliyun.dla.perf.terasort.TeraGen \
--executor-cores 2 \
--executor-memory 6G \
--num-executors 19 \
--driver-memory 8G \
--driver-cores 2 \
--name terasort-sort-1000g \
--conf yarn.nodemanager.local-dirs=/mnt/disk1/yarn（配置地址到您挂载的数据盘上） \
--conf spark.hadoop.mapreduce.fileoutputcommitter.algorithm.version=2 \
--conf spark.yarn.executor.memoryOverhead=2000 \
--conf spark.default.parallelism=2000 \
/dla-spark-perf.jar（填写测试jar包的路径）1000g hdfs://test/terasort/input/1T（填写测试数据的输出路径）
```

2. 运行测试程序
   - 在DLA Spark上运行测试程序

     登录Data Lake Analytics管理控制台，在**Serverless Spark > 作业管理**页签下，提交运行Terasort基准测试的Spark作业。示例如下：

```
{
    "args": [
        "--input",
        "<上述步骤生成的oss测试数据的路径>", #例如oss://test-bucket/terasort/input/1T。
        "--output",
        "<经过Terasort测试程序运行完成后的输出文件夹路径>", #例如oss://test-bucket/terasor
t/output/1T。
        "--optimized",
        "true",
        "--shuffle-part",
        "2000"
    ],
    "file": "<您上传的测试所需要的jar文件的oss路径>", #例如oss://test/performance/dla-spa
rk-perf.jar。
    "name": "Terasort-1T",
    "className": "com.aliyun.dla.perf.terasort.TeraSort",
    "conf": {
        "spark.dla.connectors": "oss",
        "spark.hadoop.job.oss.fileoutputcommitter.enable": "true",
        "spark.hadoop.mapreduce.fileoutputcommitter.algorithm.version": 2,
        "spark.driver.resourceSpec": "medium",
        "spark.executor.resourceSpec": "medium",
        "spark.default.parallelism": "2000",
        "spark.executor.memoryOverhead": 2000,
        "spark.executor.instances": 19
    }
}
```

- 在自建Spark集群上运行测试程序

  使用以下spark-submit命令向自建Spark集群中提交运行Terasort基准测试的Spark程序。

```
./bin/spark-submit \
--class com.aliyun.dla.perf.terasort.TeraSort \
--driver-memory 8G \
--driver-cores 2 \
--executor-cores 2 \
--executor-memory 6G \
--num-executors 19 \
--name terasort-sort-1000g \
--conf yarn.nodemanager.local-dirs=/mnt/disk1/yarn \
--conf spark.hadoop.mapreduce.fileoutputcommitter.algorithm.version=2 \
--conf spark.default.parallelism=2000 \
--conf spark.yarn.executor.memoryOverhead=2000 \
/dla-spark-perf.jar（填写测试jar包的路径） \
--input hdfs://test/terasort/input/1T（更改为测试数据地址）--output hdfs://test/terasort
/output/1t/（更改为您定义的输出地址）--optimized false --shuffle-part 2000
```

3. 记录测试结果

   记录DLA Spark和自建Spark运行Terasort基准测试的耗时。

场景二：10 TB测试数据下DLA Spark+OSS与自建Hadoop+Spark性能对比

1. 准备测试数据
   - 在OSS上生成10 TB Terasort测试数据

- 在OSS上生成10 TB Terasort测试数据

  登录Data Lake Analytics管理控制台，在**Serverless Spark > 作业管理**页签下，提交运行生成10 TB Terasort测试数据的Spark作业。示例如下：

```
{
    "args": [
        "10000g",
        "oss://<bucket-name>/<输出文件夹路径>",  #您的Terasort测试数据存放的OSS路径，例如oss://test-bucket/terasort/input/10T。
        "true"
    ],
    "file": "<Jar包的OSS路径>",  #上文中您Jar包上传的OSS路径，例如oss://test/performance/dla-spark-perf.jar。
    "name": "TeraGen-1T",
    "className": "com.aliyun.dla.perf.terasort.TeraGen",
    "conf": {
        "spark.dla.connectors": "oss",
        "spark.hadoop.job.oss.fileoutputcommitter.enable": "true",
        "spark.hadoop.mapreduce.fileoutputcommitter.algorithm.version": 2,
        "spark.driver.resourceSpec": "medium",
        "spark.executor.resourceSpec": "medium",
        "spark.default.parallelism": "20000",
        "spark.executor.memoryOverhead": 2000,
        "spark.executor.instances": 39
    }
}
```

- 在自建Hadoop上生成10 TB Terasort测试数据

  使用spark-submit命令向自建Spark集群中提交运行生成10 TB Terasort测试数据的Spark程序。示例如下：

```
./bin/spark-submit \
--class com.aliyun.dla.perf.terasort.TeraGen \
--executor-cores 2 \
--executor-memory 6G \
--num-executors 39 \
--driver-memory 8G \
--driver-cores 2 \
--name terasort-sort-1000g \
--conf yarn.nodemanager.local-dirs=/mnt/disk1/yarn（配置地址到您挂载的数据盘上） \
--conf spark.hadoop.mapreduce.fileoutputcommitter.algorithm.version=2 \
--conf spark.yarn.executor.memoryOverhead=2000 \
--conf spark.default.parallelism=20000 \
/dla-spark-perf.jar（填写测试jar包的路径） 1000g hdfs://test/terasort/input/1T（填写测试数据的输出路径）
```

2. 运行测试程序

- 在DLA Spark上运行测试程序

  登录Data Lake Analytics管理控制台，在**Serverless Spark > 作业管理**页签下，提交运行Terasort基准测试的Spark作业。示例如下：

```
{
    "args": [
        "--input",
        "<上述步骤生成的oss测试数据的路径>", #例如oss://test-bucket/terasort/input/10T。
        "--output",
        "<经过Terasort测试程序运行完成后的输出文件夹路径>", #例如oss://test-bucket/terasor
t/output/10T。
        "--optimized",
        "true",
        "--shuffle-part",
        "2000"
    ],
    "file": "<您上传的测试所需要的jar文件的oss路径>", #例如oss://test/performance/dla-spa
rk-perf.jar。
    "name": "Terasort-10T",
    "className": "com.aliyun.dla.perf.terasort.TeraSort",
    "conf": {
        "spark.dla.connectors": "oss",
        "spark.hadoop.job.oss.fileoutputcommitter.enable": "true",
        "spark.hadoop.mapreduce.fileoutputcommitter.algorithm.version": 2,
        "spark.driver.resourceSpec": "medium",
        "spark.executor.resourceSpec": "medium",
        "spark.default.parallelism": "2000",
        "spark.executor.memoryOverhead": 2000,
        "spark.executor.instances": 39
    }
}
```

- 在自建Spark集群上运行测试程序

  使用以下spark-submit命令向自建Spark集群中提交运行Terasort基准测试的Spark程序。

```
./bin/spark-submit \
--class com.aliyun.dla.perf.terasort.TeraSort \
--driver-memory 8G \
--driver-cores 2 \
--executor-cores 2 \
--executor-memory 6G \
--num-executors 19 \
--name terasort-sort-1000g \
--conf yarn.nodemanager.local-dirs=/mnt/disk1/yarn \
--conf spark.default.parallelism=20000 \
--conf spark.yarn.executor.memoryOverhead=2000 \
/dla-spark-perf.jar（填写测试jar包的路径）\
--input hdfs://test/terasort/input/10T（更改为测试数据地址）--output hdfs://test/terasor
t/output/10t/（更改为您定义的输出地址）--optimized false --shuffle-part 20000
```

3. 记录测试结果

   记录DLA Spark和自建Spark运行Terasort基准测试的耗时。

场景三：1 TB测试数据下DLA Spark+用户自建Hadoop集群与自建Hadoop+Spark性能对比

1. 准备测试数据

在自建Hadoop上生成1 TB Terasort测试数据。使用spark-submit命令向自建Spark集群中提交运行生成1 TB Terasort测试数据的Spark程序。示例如下：

```
./bin/spark-submit \
--class com.aliyun.dla.perf.terasort.TeraGen \
--executor-cores 2 \
--executor-memory 6G \
--num-executors 19 \
--driver-memory 8G \
--driver-cores 2 \
--name terasort-sort-1000g \
--conf yarn.nodemanager.local-dirs=/mnt/disk1/yarn（配置地址到您挂载的数据盘上）\
--conf spark.hadoop.mapreduce.fileoutputcommitter.algorithm.version=2 \
--conf spark.yarn.executor.memoryOverhead=2000 \
--conf spark.default.parallelism=2000 \
/dla-spark-perf.jar（填写测试jar包的路径）1000g hdfs://test/terasort/input/1T（填写测试数据的输出路径）
```

2. 运行测试程序
   - 在DLA Spark上运行测试程序

     > **说明** 通过DLA Spark访问自建Hadoop集群需要配置打通VPC网络，具体DLA Spark连接VPC网络下的HDFS相关参数的含义和配置步骤，请参见Hadoop。

     登录Data Lake Analytics管理控制台，在**Serverless Spark > 作业管理**页签下，提交运行Terasort基准测试的Spark作业。示例如下：

```
{
    "args": [
        "--input",
        "<上述步骤生成的oss测试数据的路径>", #例如hdfs://test/terasort/input/1T。
        "--output",
        "<经过Terasort测试程序运行完成后的输出文件夹路径>", #例如hdfs://test/terasort/outp
ut/1t/。
        "--optimized",
        "false",
        "--shuffle-part",
        "2000"
    ],
    "file": "<您上传的测试所需要的jar文件的oss路径>", #例如oss://test/performance/dla-spa
rk-perf.jar。
    "name": "TeraSort-HDFS",
    "className": "com.aliyun.dla.perf.terasort.TeraSort",
    "conf": {
        "spark.dla.eni.enable": "true",
        "spark.dla.eni.vswitch.id": "vsw-xxxxx",
        "spark.dla.eni.security.group.id": "sg-xxxx",
        "spark.hadoop.mapreduce.fileoutputcommitter.algorithm.version": 2,
        "spark.driver.resourceSpec": "medium",
        "spark.hadoop.dfs.namenode.rpc-address.<nameservices>.nn2": "xxxx2:8020",
        "spark.hadoop.dfs.namenode.rpc-address.<nameservices>.nn1": "xxxx1:8020",
        "spark.hadoop.dfs.ha.automatic-failover.enabled.<nameservices>": "true",
        "spark.hadoop.dfs.namenode.http-address.<nameservices>.nn1": "xxxx1:50070",
        "spark.executor.resourceSpec": "medium",
        "spark.hadoop.dfs.nameservices": "<nameservices>",
        "spark.hadoop.dfs.client.failover.proxy.provider.<nameservices>": "org.apache
.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider",
        "spark.hadoop.dfs.namenode.http-address.<nameservices>.nn2": "xxxx2:50070",
        "spark.hadoop.dfs.ha.namenodes.<nameservices>": "nn1,nn2",
        "spark.executor.memoryOverhead": 2000,
        "spark.default.parallelism": "2000",
        "spark.executor.instances": 19
    }
```

- 在自建Spark集群上运行测试程序

  使用以下spark-submit命令向自建Spark集群中提交运行Terasort基准测试的Spark程序。

```
./bin/spark-submit \
--class com.aliyun.dla.perf.terasort.TeraSort \
--driver-memory 8G \
--driver-cores 2 \
--executor-cores 2 \
--executor-memory 6G \
--num-executors 19 \
--name terasort-sort-1000g \
--conf yarn.nodemanager.local-dirs=/mnt/disk1/yarn \
--conf spark.hadoop.mapreduce.fileoutputcommitter.algorithm.version=2 \
--conf spark.default.parallelism=2000 \
--conf spark.yarn.executor.memoryOverhead=2000 \
/dla-spark-perf.jar（填写测试jar包的路径） \
--input hdfs://test/terasort/input/1T（更改为测试数据地址） --output hdfs://test/terasort
/output/1t/（更改为您定义的输出地址） --optimized false --shuffle-part 2000
```

3. 记录测试结果

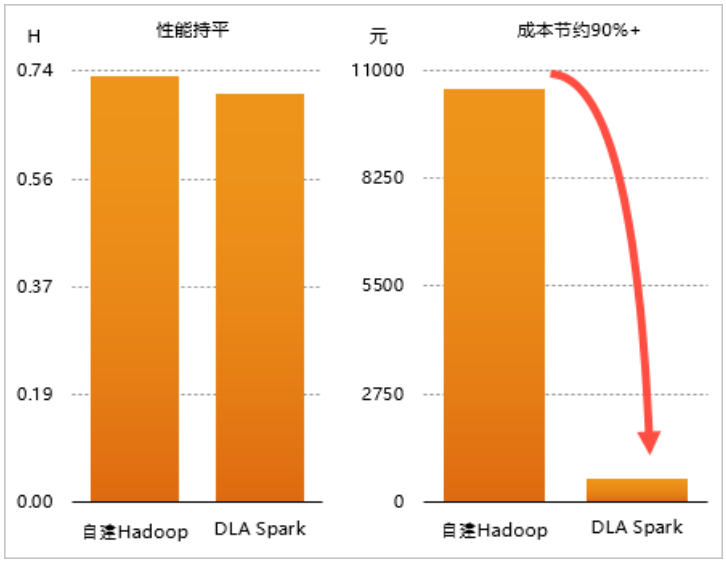
   记录DLA Spark和自建Spark运行Terasort基准测试的耗时。

8.3. 测试结果

本次测试采用3种不同的测试场景，针对开源自建的Hadoop+Spark集群与阿里云云原生数据湖分析DLA Spark在执行Terasort基准测试的性能做了对比分析。本文档主要展示了开源自建Spark和DLA Spark在3种测试场景下的测试结果及性能对比分析。

1 TB测试数据下DLA Spark+OSS与自建Hadoop+Spark集群性能对比结果

集群类型	运行Terasort基准测试集耗时（h）	费用价格（元）
DLA Spark+OSS	0.701	577.42
自建Hadoop+Spark	0.733	10543.04

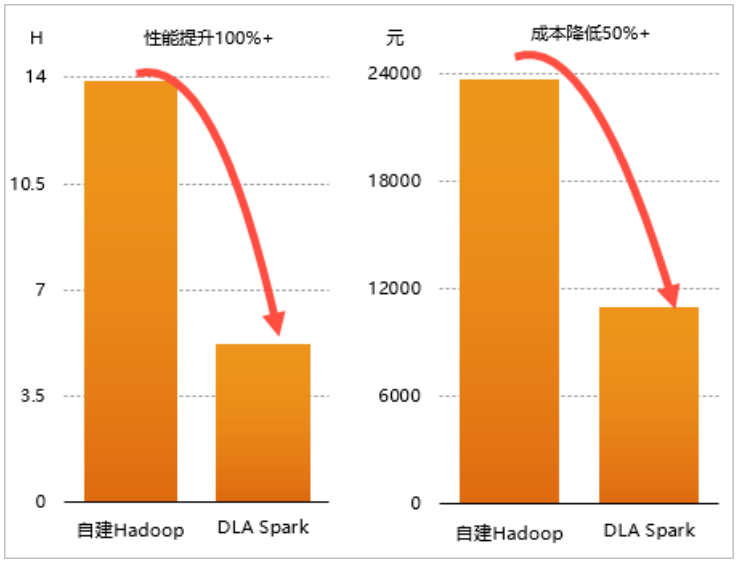


通过上述耗时和价格对比结果可以看出，作业性能上DLA Spark跟自建Spark基本持平，但是性价比差异非常大，DLA Spark能节约90%的成本，会有9~10倍的性价比提升。对于中小客户来说，业务比较简单，集群的使用空闲率较高，使用DLA Spark会极大的降低成本。

需要强调的是，DLA Spark完全按需使用存储和计算资源，对OSS访问实现了深度定制优化，性能相比于优化前提升1倍左右，与Spark访问HDFS性能持平。

10 TB测试数据下DLA Spark+OSS与自建Hadoop+Spark性能对比结果

集群类型	运行Terasort基准测试集耗时（h）	价格（元）
DLA Spark+OSS	5.2	10989.4
自建Hadoop+Spark	13.9	23660.24

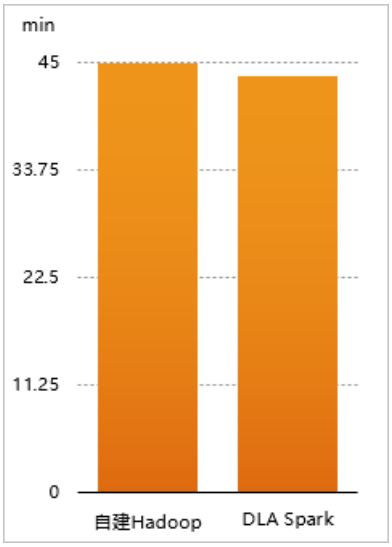


通过上述耗时和价格对比结果可以看出，性能上DLA Spark提升了1倍，成本反而降低了一半，性价比提升4倍。

在分析性能时发现，在10 TB场景下，本地盘的存储和shuffle之间会有IO带宽上的明显争抢，而Serverless Spark计算节点自带essd云盘，与shuffle盘完全独立，能较高的提升性能。

1 TB测试数据下DLA Spark+用户自建Hadoop集群与自建Hadoop+Spark性能对比结果

集群类型	运行Terasort基准测试集耗时（min）
DLA Spark+OSS	43.5
自建Hadoop+Spark	44.8



您可以将自建Hadoop和DLA Spark混合使用，自建Hadoop集群在高峰期需要更多的计算资源。DLA Spark可以直接跟您的VPC网络打通，直接使用内网的带宽，计算性能相对于本地计算并没有降低。DLA Spark完全弹性的模式，1分钟内可以拉起500~1000个计算节点，可以很好满足您对弹性计算的需求。