**数据湖-Iceberg**

## 什么是数据湖

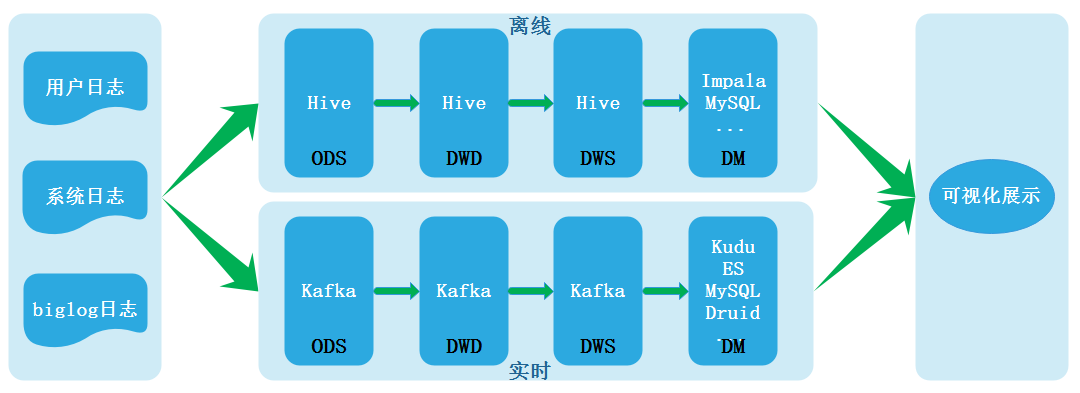
### **什么是数据湖**

数据湖是一个集中式的存储库，允许你以任意规模存储多个来源、所有结构化和非结构化数据，可以按照原样存储数据，无需对数据进行结构化处理，并运行不同类型的分析，对数据进行加工，例如：大数据处理、实时分析、机器学习，以指导做出更好地决策。

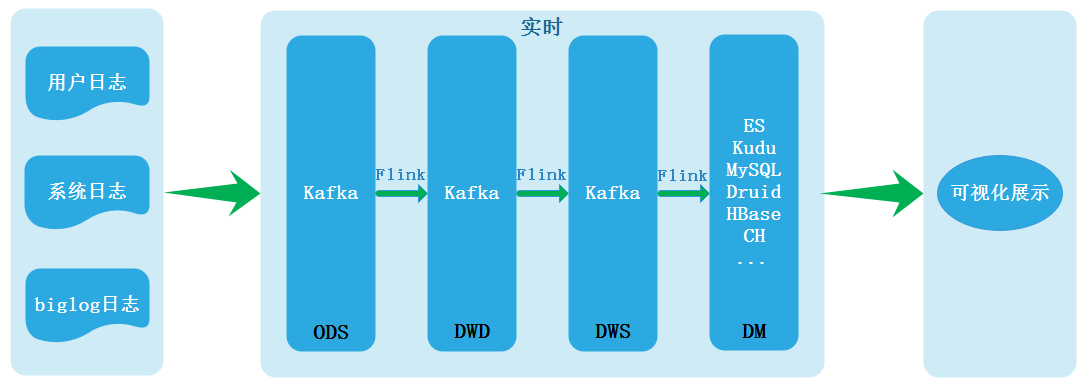
### **大数据为什么需要数据湖**

当前基于Hive的离线数据仓库已经非常成熟，在传统的离线数据仓库中对记录级别的数据进行更新是非常麻烦的，需要对待更新的数据所属的整个分区，甚至是整个表进行全面覆盖才行，由于离线数仓多级逐层加工的架构设计，数据更新时也需要从贴源层开始逐层反应到后续的派生表中去。

随着实时计算引擎的不断发展以及业务对于实时报表的产出需求不断膨胀，业界最近几年就一直聚焦并探索于实时数仓建设。根据数仓架构演变过程，在Lambda架构中含有离线处理与实时处理两条链路，其架构图如下：



正是由于两条链路处理数据导致数据不一致等一些列问题所以才有了Kappa架构，Kappa架构如下：



Kappa架构可以称为真正的实时数仓，目前在业界最常用实现就是Flink + Kafka，然而基于Kafka+Flink的实时数仓方案也有几个非常明显的缺陷，所以在目前很多企业中实时数仓构建中经常使用混合架构，没有实现所有业务都采用Kappa架构中实时处理实现。Kappa架构缺陷如下：

1. Kafka无法支持海量数据存储。对于海量数据量的业务线来说，Kafka一般只能存储非常短时间的数据，比如最近一周，甚至最近一天。
2. Kafka无法支持高效的OLAP查询，大多数业务都希望能在DWD\DWS层支持即席查询的，但是Kafka无法非常友好地支持这样的需求。
3. 无法复用目前已经非常成熟的基于离线数仓的数据血缘、数据质量管理体系。需要重新实现一套数据血缘、数据质量管理体系。
4. Kafka不支持update/upsert，目前Kafka仅支持append。

为了解决Kappa架构的痛点问题，业界最主流是采用“批流一体”方式，这里批流一体可以理解为批和流使用SQL同一处理，也可以理解为处理框架的统一，例如：Spark、Flink，但这里更重要指的是存储层上的统一，只要存储层面上做到“批流一体”就可以解决以上Kappa遇到的各种问题。数据湖技术可以很好的实现存储层面上的“批流一体”，这就是为什么大数据中需要数据湖的原因。

## Iceberg概念及特点

Apache Iceberg是一种用于大型数据分析场景的开放表格式（Table Format）。Iceberg使用一种类似于SQL表的高性能表格式，Iceberg格式表单表可以存储数十PB数据，适配Spark、Trino、PrestoDB、Flink和Hive等计算引擎提供高性能的读写和元数据管理功能，Iceberg是一种数据湖解决方案。

***注意：Trino就是原来的PrestoSQL ,2020年12月27日，PrestoSQL 项目更名为Trino，Presto分成两大分支：PrestoDB、PrestorSQL。***

Iceberg非常轻量级，可以作为lib与Spark、Flink进行集成，Iceberg官网：https://iceberg.apache.org/，Iceberg具备以下特点：

* Iceberg支持实时/批量数据写入和读取，支持Spark/Flink计算引擎。
* Iceberg支持事务ACID,支持添加、删除、更新数据。
* 不绑定任何底层存储，支持Parquet、ORC、Avro格式兼容行存储和列存储。
* Iceberg支持隐藏分区和分区变更，方便业务进行数据分区策略。
* Iceberg支持快照数据重复查询，具备版本回滚功能。
* Iceberg扫描计划很快，读取表或者查询文件可以不需要分布式SQL引擎。
* Iceberg通过表元数据来对查询进行高效过滤。
* 基于乐观锁的并发支持，提供多线程并发写入能力并保证数据线性一致。

## Iceberg数据存储格式

### Iceberg术语

* **data files（数据文件）:**

数据文件是Apache Iceberg表真实存储数据的文件，一般是在表的数据存储目录的data目录下，如果我们的文件格式选择的是parquet,那么文件是以“.parquet”结尾，例如：

00000-0-root\_20211212192602\_8036d31b-9598-4e30-8e67-ce6c39f034da-job\_1639237002345\_0025-00001.parquet就是一个数据文件。

Iceberg每次更新会产生多个数据文件(data files)。

* **Snapshot（表快照）：**

快照代表一张表在某个时刻的状态。每个快照里面会列出表在某个时刻的所有 data files 列表。data files是存储在不同的manifest files里面，manifest files是存储在一个Manifest list文件里面，而一个Manifest list文件代表一个快照。

* **Manifest list（清单列表）：**

manifest list是一个元数据文件，它列出构建表快照（Snapshot）的清单（Manifest file）。这个元数据文件中存储的是Manifest file列表，每个Manifest file占据一行。每行中存储了Manifest file的路径、其存储的数据文件（data files）的分区范围，增加了几个数文件、删除了几个数据文件等信息，这些信息可以用来在查询时提供过滤，加快速度。

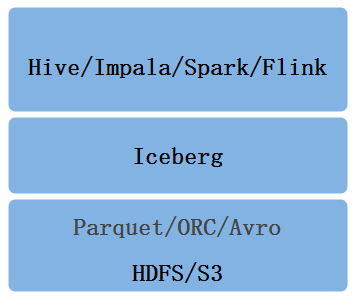
* **Manifest file（清单文件）：**

Manifest file也是一个元数据文件，它列出组成快照（snapshot）的数据文件（data files）的列表信息。每行都是每个数据文件的详细描述，包括数据文件的状态、文件路径、分区信息、列级别的统计信息（比如每列的最大最小值、空值数等）、文件的大小以及文件里面数据行数等信息。其中列级别的统计信息可以在扫描表数据时过滤掉不必要的文件。

Manifest file是以avro格式进行存储的，以“.avro”后缀结尾，例如：8138fce4-40f7-41d7-82a5-922274d2abba-m0.avro。

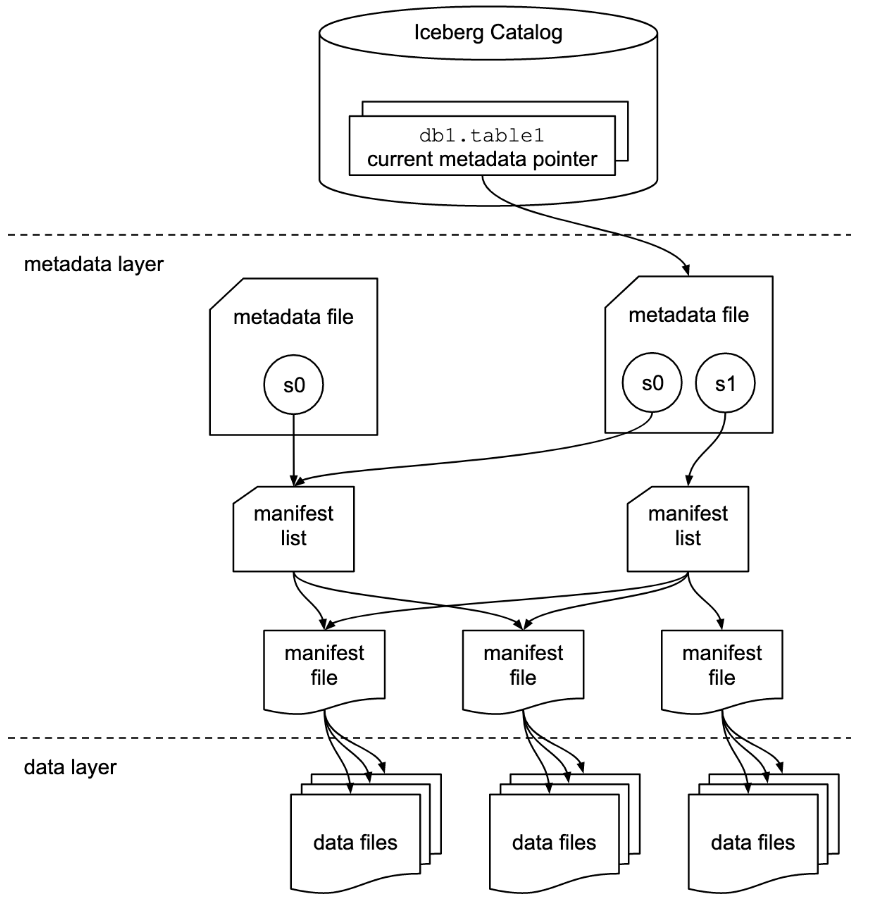
### 表格式Table Format

Apache Iceberg作为一款数据湖解决方案,是一种用于大型分析数据集的开放表格式(Table Format),表格式可以理解为元数据及数据文件的一种组织方式。Iceberg底层数据存储可以对接HDFS，S3文件系统，并支持多种文件格式，处于计算框架（Spark、Flink）之下，数据文件之上。



下面介绍下Iceberg底层文件组织方式，下图是Iceberg中表格式，s0、s1代表的是表Snapshot信息，每个表示当前操作的一个快照，每次commit都会生成一个快照Snapshot，每个Snapshot快照对应一个manifest list 元数据文件，每个manifest list 中包含多个Manifest元数据文件，manifest中记录了当前操作生成数据所对应的文件地址，也就是data file的地址。

基于snapshot的管理方式，Iceberg能够获取表历史版本数据、对表增量读取操作，data files存储支持不同的文件格式，目前支持parquet、ORC、Avro格式。



关于Iceberg表数据底层组织详细信息参照“Iceberg表数据组织与查询”小节。

## Iceberg特点详述

### Iceberg分区与隐藏分区（Hidden Partition）

Iceberg支持分区来加快数据查询。在Iceberg中设置分区后，可以在写入数据时将相似的行分组，在查询时加快查询速度。Iceberg中可以按照年、月、日和小时粒度划分时间戳组织分区。

在Hive中也支持分区，但是要想使分区能加快查询速度，需要在写SQL时指定对应的分区条件过滤数据，在Iceberg中写SQL查询时不需要再SQL中特别指定分区过滤条件，Iceberg会自动分区，过滤掉不需要的数据。

在Iceberg中分区信息可以被隐藏起来，Iceberg的分区字段可以通过一个字段计算出来，在建表或者修改分区策略之后，新的数据会自动计算所属于的分区，在查询的时候同样不用关心表的分区是什么字段，只需要关注业务逻辑，Iceberg会自动过滤不需要的分区数据。

正是由于Iceberg的分区信息和表数据存储目录是独立的，使得Iceberg的表分区可以被修改，而且不会涉及到数据迁移。

### Iceberg表演化（Table Evolution）

在Hive分区表中，如果把一个按照天分区的表改成按小时分区，那么没有办法在原有表上进行修改，需要创建一个按照小时分区的表，然后把数据加载到此表中。

Iceberg支持就地表演化，可以通过SQL的方式进行表级别模式演进，例如：更改表分区布局。Iceberg进行以上操作时，代价极低，不存在读出数据重新写入或者迁移数据这种费时费力的操作。

### 模式演化（Schema Evolution）

Iceberg支持以下几种Schema的演化：

* ADD:向表或者嵌套结构增加新列。
* Drop:从表或嵌套结构中移除列。
* Rename:重命名表中或者嵌套结构中的列。
* Update:将复杂结构（Struct、Map<Key,Value>,list）中的基本类型扩展类型长度，比如：tinyint修改成int。
* Reorder:改变列的顺序，也可以改变嵌套结构中字段的排序顺序。

注意：

Iceberg Schema的改变只是元数据的操作改变，不会涉及到重写数据文件。Map结构类型不支持Add和Drop字段。

Iceberg保证Schema演化是没有副作用的独立操作，不会涉及到重写数据文件，具体如下：

* 增加列时不会从另一个列中读取已存在的数据
* 删除列或者嵌套结构中的字段时，不会改变任何其他列的值。
* 更新列或者嵌套结构中字段时，不会改变任何其他列的值。
* 改变列或者嵌套结构中字段顺序的时候，不会改变相关联的值。

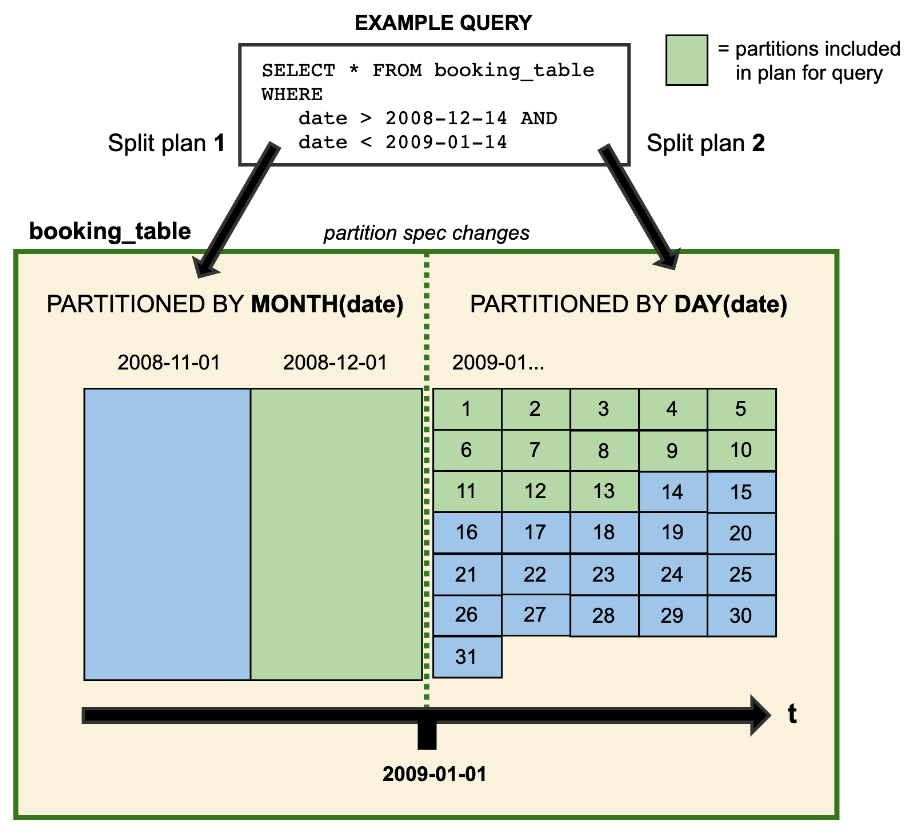
Iceberg实现以上的原因使用唯一的id来追踪表中的每一列，当添加一个列时，会分配新的ID，因此列对应的数据不会被错误使用。

### 分区演化（partition Evolution）

Iceberg分区可以在现有表中更新，因为Iceberg查询流程并不和分区信息直接关联。

当我们改变一个表的分区策略时, 对应修改分区之前的数据不会改变, 依然会采用老的分区策略, 新的数据会采用新的分区策略, 也就是说同一个表会有两种分区策略, 旧数据采用旧分区策略, 新数据采用新新分区策略, 在元数据里两个分区策略相互独立,不重合.

因此，在我们写SQL进行数据查询时, 如果存在跨分区策略的情况, 则会解析成两个不同执行计划, 如Iceberg官网提供图所示:



图中booking\_table表2008年按月分区, 进入2009年后改为按天分区, 这两中分区策略共存于该表中。得益于Iceberg的隐藏分区(Hidden Partition), 针对上图中的SQL查询, 不需要在SQL中特别指定分区过滤条件（是按照月还是按照天）, Iceberg会自动分区, 过滤掉不需要的数据。

### 列顺序演化（Sort Order Evolution）

Iceberg可以在一个已经存在的表上修改排序策略。修改了排序策略之后, 旧数据依旧采用老排序策略不变。往Iceberg里写数据的计算引擎总是会选择最新的排序策略， 但是当排序的代价极其高昂的时候, 就不进行排序了。

## Iceberg数据类型

Iceberg表支持以下数据类型：

|  |  |  |
| --- | --- | --- |
| **类型** | **描述** | **注意点** |
| **boolean** | 布尔类型，true或者false |  |
| **int** | 32位有符号整形 | 可以转换成long类型 |
| **long** | 64位有符号整形 |  |
| **float** | 单精度浮点型 | 可以转换成double类型 |
| **double** | 双精度浮点型 |  |
| **decimal(P,S)** | decimal(P,S) | P代表精度，决定总位数,S代表规模，决定小数位数。P必须小于等于38。 |
| **date** | 日期，不含时间和时区 |  |
| **time** | 时间，不含日期和时区 | 以微秒存储，1000微秒 = 1毫秒 |
| **timestamp** | 不含时区的timestamp | 以微秒存储，1000微秒 = 1毫秒 |
| **timestamptz** | 含时区的timestamp | 以微秒存储，1000微秒 = 1毫秒 |
| **string** | 任意长度的字符串类型 | UTF-8编码 |
| **fixed(L)** | 长度为L的固定长度字节数组 |  |
| **binary** | 任意长度的字节数组 |  |
| **struct<...>** | 任意数据类型组成的一个结构化字段 |  |
| **list<E>** | 任意数据类型组成的List |  |
| **map<K,V>** | 任意类型组成的K,V的Map |  |

## Hive与Iceberg整合

Iceberg就是一种表格式，支持使用Hive对Iceberg进行读写操作，但是对Hive的版本有要求，如下：

|  |  |  |
| --- | --- | --- |
| **操作** | **Hive 2.x** | **Hive 3.1.2** |
| **CREATE EXTERNAL TABLE** | **√** | **√** |
| **CREATE TABLE** | **√** | **√** |
| **DROP TABLE** | **√** | **√** |
| **SELECT** | **√** | **√** |
| **INSERT INTO** | **√** | **√** |

这里基于Hive3.1.2版本进行Hive操作Iceberg表讲解。

### 开启Hive支持Iceberg

1. **下载iceberg-hive-runtime.jar**

想要使用Hive支持查询Iceberg表，首先需要下载“iceberg-hive-runtime.jar”，Hive通过该Jar可以加载Hive或者更新Iceberg表元数据信息。下载地址：https://iceberg.apache.org/#releases/：



将以上jar包下载后，上传到Hive服务端和客户端对应的lib目录下。另外在向Hive中Iceberg格式表插入数据时需要到“libfb303-0.9.3.jar”包，将此包也上传到Hive服务端和客户端对应的lib目录下。

1. **配置hive-site.xml**

在Hive客户端$HIVE\_HOME/conf/hive-site.xml中添加如下配置：

|  |
| --- |
| <property>  <name>iceberg.engine.hive.enabled</name>  <value>true</value>  </property> |

### Hive中操作Iceberg格式表

从Hive引擎的角度来看，在运行环境中有Catalog概念（catalog主要描述了数据集的位置信息，就是元数据），Hive与Iceberg整合时，Iceberg支持多种不同的Catalog类型，例如:Hive、Hadoop、第三方厂商的AWS Glue和自定义Catalog。在实际应用场景中，Hive可能使用上述任意Catalog，甚至跨不同Catalog类型join数据，为此Hive提供了org.apache.iceberg.mr.hive.HiveIcebergStorageHandler（位于包iceberg-hive-runtime.jar）来支持读写Iceberg表，并通过在Hive中设置“iceberg.catalog.<catalog\_name>.type”属性来决定加载Iceberg表的方式，该属性可以配置：hive、hadoop，其中“<catalog\_name>”是自己随便定义的名称，主要是在hive中创建Iceberg格式表时配置iceberg.catalog属性使用。

在Hive中创建Iceberg格式表时，根据创建Iceberg格式表时是否指定iceberg.catalog属性值，有以下三种方式决定Iceberg格式表如何加载（数据存储在什么位置）。

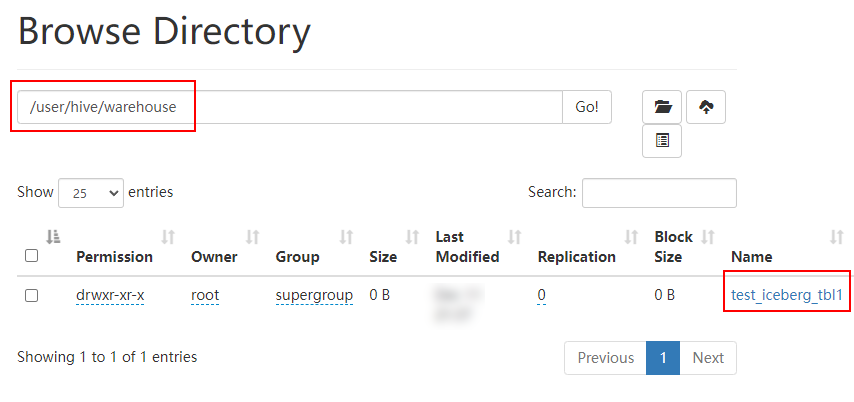
1. **如果没有设置iceberg.catalog属性，默认使用HiveCatalog来加载**

这种方式就是说如果在Hive中创建Iceberg格式表时，不指定iceberg.catalog属性，那么数据存储在对应的hive warehouse路径下。

在Hive客户端node3节点进入Hive，操作如下：

|  |
| --- |
| **#在Hive中创建iceberg格式表**  create table test\_iceberg\_tbl1(  id int ,  name string,  age int)  partitioned by (dt string)  stored by 'org.apache.iceberg.mr.hive.HiveIcebergStorageHandler';  **#在Hive中加载如下两个包，在向Hive中插入数据时执行MR程序时需要使用到**  hive> add jar /software/hive-3.1.2/lib/iceberg-hive-runtime-0.12.1.jar;  hive> add jar /software/hive-3.1.2/lib/libfb303-0.9.3.jar;  **#向表中插入数据**  hive> insert into test\_iceberg\_tbl1 values (1,"zs",18,"20211212");  **#查询表中的数据**  hive> select \* from test\_iceberg\_tbl1;  *OK*  *1 zs 18 20211212* |

在Hive默认的warehouse目录下可以看到创建的表目录：



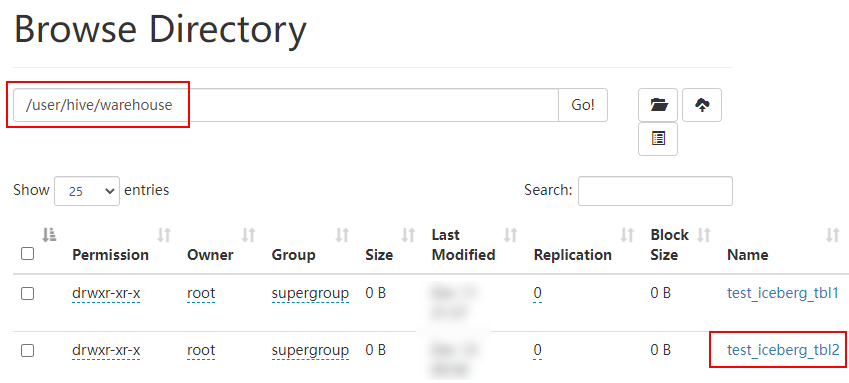
1. **如果设置了iceberg.catalog对应的catalog名字，就用对应类型的catalog加载**

这种情况就是说在Hive中创建Iceberg格式表时，如果指定了iceberg.catalog属性值，那么数据存储在指定的catalog名称对应配置的目录下。

在Hive客户端node3节点进入Hive，操作如下：

|  |
| --- |
| **#注册一个HiveCatalog叫another\_hive**  hive> set iceberg.catalog.**another\_hive**.type=hive;  **#在Hive中创建iceberg格式表**  create table test\_iceberg\_tbl2(  id int,  name string,  age int  )  partitioned by (dt string)  stored by 'org.apache.iceberg.mr.hive.HiveIcebergStorageHandler'  tblproperties ('iceberg.catalog'='another\_hive');  **#在Hive中加载如下两个包，在向Hive中插入数据时执行MR程序时需要使用到**  hive> add jar /software/hive-3.1.2/lib/iceberg-hive-runtime-0.12.1.jar;  hive> add jar /software/hive-3.1.2/lib/libfb303-0.9.3.jar;  **#插入数据，并查询**  hive> insert into test\_iceberg\_tbl2 values (2,"ls",20,"20211212");  hive> select \* from test\_iceberg\_tbl2;  *OK*  *2 ls 20 20211212* |

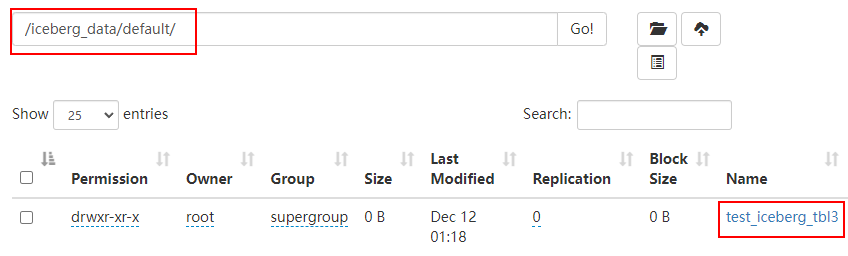
以上方式指定“iceberg.catalog.**another\_hive**.type=hive”后，实际上就是使用的hive的catalog，这种方式与第一种方式不设置效果一样，创建后的表存储在hive默认的warehouse目录下。也可以在建表时指定location 写上路径，将数据存储在自定义对应路径上。



除了可以将catalog类型指定成hive之外，还可以指定成hadoop，在Hive中创建对应的iceberg格式表时需要指定location来指定iceberg数据存储的具体位置，这个位置是**具有一定格式规范**的自定义路径。在Hive客户端node3节点进入Hive，操作如下：

|  |
| --- |
| **#注册一个HadoopCatalog叫hadoop**  hive> set iceberg.catalog.**hadoop**.type=hadoop;  **#使用HadoopCatalog时，必须设置“iceberg.catalog.<catalog\_name>.warehouse”指定warehouse路径**  hive> set iceberg.catalog.**hadoop**.warehouse=hdfs://mycluster/iceberg\_data;  **#在Hive中创建iceberg格式表,这里创建成外表**  create external table test\_iceberg\_tbl3(  id int,  name string,  age int  )  partitioned by (dt string)  stored by 'org.apache.iceberg.mr.hive.HiveIcebergStorageHandler'  **location 'hdfs://mycluster/iceberg\_data/default/test\_iceberg\_tbl3'**  tblproperties ('iceberg.catalog'='hadoop');  **注意：以上location指定的路径必须是“iceberg.catalog.hadoop.warehouse”指定路径的子路径,格式必须是${iceberg.catalog.hadoop.warehouse}/${当前建表使用的hive库}/${创建的当前iceberg表名}**  **#在Hive中加载如下两个包，在向Hive中插入数据时执行MR程序时需要使用到**  hive> add jar /software/hive-3.1.2/lib/iceberg-hive-runtime-0.12.1.jar;  hive> add jar /software/hive-3.1.2/lib/libfb303-0.9.3.jar;  **#插入数据，并查询**  hive> insert into test\_iceberg\_tbl3 values (3,"ww",20,"20211213");  hive> select \* from test\_iceberg\_tbl3;  *OK*  *3 ww 20 20211213* |

在指定的“iceberg.catalog.**hadoop**.warehouse”路径下可以看到创建的表目录：



1. **如果iceberg.catalog属性设置为“location\_based\_table”,可以从指定的根路径下加载Iceberg 表**

这种情况就是说如果HDFS中已经存在iceberg格式表，我们可以通过在Hive中创建Icerberg格式表指定对应的location路径映射数据。，在Hive客户端中操作如下：

|  |
| --- |
| CREATE TABLE test\_iceberg\_tbl4 (  id int,  name string,  age int,  dt string  )STORED BY 'org.apache.iceberg.mr.hive.HiveIcebergStorageHandler'  LOCATION 'hdfs://mycluster/spark/person'  TBLPROPERTIES ('iceberg.catalog'='location\_based\_table');  **注意：指定的location路径下必须是iceberg格式表数据，并且需要有元数据目录才可以。不能将其他数据映射到Hive iceberg格式表。** |

注意：由于Hive建表语句分区语法“Partitioned by”的限制,如果使用Hive创建Iceberg格式表，目前只能按照Hive语法来写，底层转换成Iceberg标识分区，这种情况下不能使用Iceberge的分区转换，例如：days(timestamp)，如果想要使用Iceberg格式表的分区转换标识分区，需要使用Spark或者Flink引擎创建表。

## Iceberg表数据组织与查询

### 下载avro-tools jar包

由于后期需要查看avro文件内容，我们可以通过avro-tool.jar来查看avro数据内容。可以在以下网站中下载avro-tools对应的jar包，下载之后上传到node5节点上：

“https://mvnrepository.com/artifact/org.apache.avro/avro-tools”。

查看avro文件信息可以直接执行如下命令，可以将avro中的数据转换成对应的json数据。

|  |
| --- |
| [root@node5 ~]# java -jar /software/avro-tools-1.8.1.jar tojson snap-\*-wqer.avro |

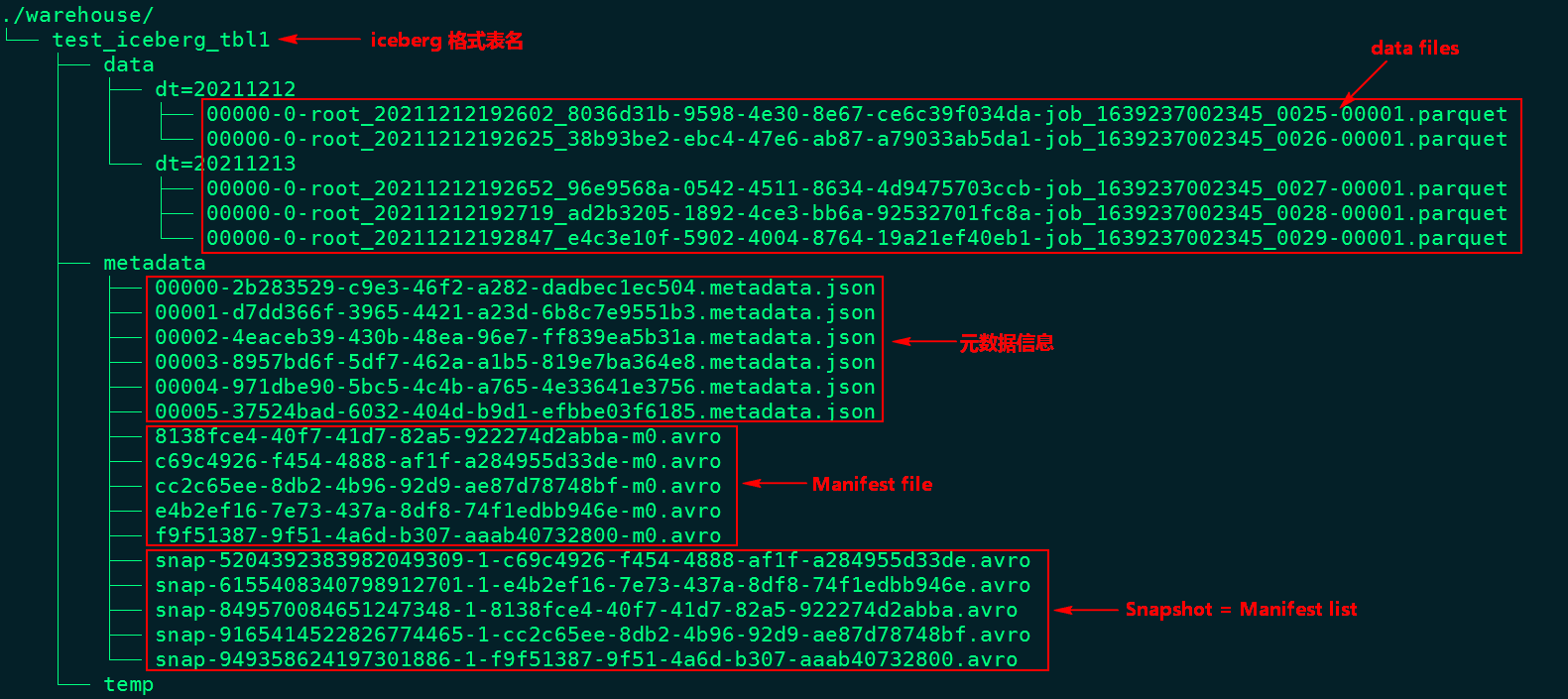
### 在Hive中创建Iceberg表并插入数据

在Hive中创建Iceberg格式表，并插入如下数据：

|  |
| --- |
| **#在Hive中创建iceberg格式表**  create table test\_iceberg\_tbl1(  id int ,  name string,  age int)  partitioned by (dt string)  stored by 'org.apache.iceberg.mr.hive.HiveIcebergStorageHandler';  **#插入如下数据**  insert into test\_iceberg\_tbl1 values (1,"zs",21,"20211212");  insert into test\_iceberg\_tbl1 values (2,"ls",22,"20211212");  insert into test\_iceberg\_tbl1 values (3,"ww",23,"20211213");  insert into test\_iceberg\_tbl1 values (4,"ml",24,"20211213");  insert into test\_iceberg\_tbl1 values (5,"tq",25,"20211213"); |

### 查看Iceberg底层数据存储

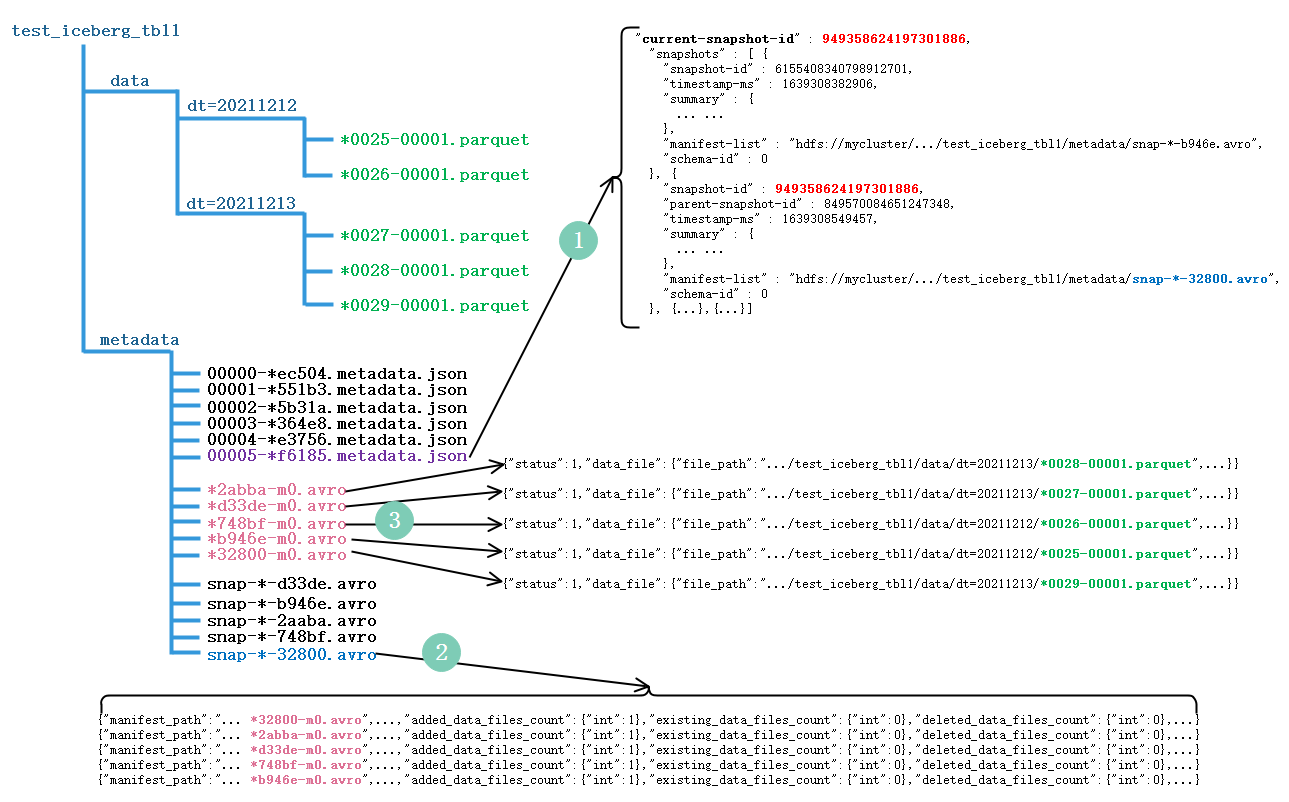
下图为Iceberg表“test\_iceberg\_tbl1”在HDFS中存储的数据组织图：



通过上图我们可以看到有5个Snapshot快照,以上5个Snapshot实际上就是对应了5个Manifest list清单列表。

1. **查询最新快照数据**

为了了解Iceberg如何查询最新数据，可以参照下面这张图来详细了解底层实现。



查询Iceberg表数据时，首先获取最新的metadata信息，这里先获取到“00000-\*ec504.metadata.json”元数据信息，解析当前元数据文件可以拿到当前表的快照id:“949358624197301886”以及这张表的所有快照信息，也就是json信息中snapshots数组对应的值。根据当前表的快照id值可以获取对应的snapshot对应的avro文件信息：“snap-\*-32800.avro”，我们可以找到当前快照对应的路径，看到其包含的Manifest 清单文件有5个："\*32800-m0.avro"、"\*2abba-m0.avro"、"\*d33de-m0.avro"、"\*748bf-m0.avro"、"\*b946e-m0.avro"，读取该Iceberg格式表最新数据就是读取这几个文件中描述对应的parquet数据文件即可。

我们可以看到“snap-\*-32800.avro”快照文件中不仅有包含的manifest路径信息，还有“added\_data\_files\_count”、“existing\_data\_files\_count”、“deleted\_data\_files\_count”三个属性，Iceberg 根据 deleted\_data\_files\_count 大于 0 来判断对应的manifest清单文件里面是不是被删除的数据,如果一个manifest清单文件该值大于0代表数据删除，读数据时就无需读这个manifest清单文件对应的数据文件。

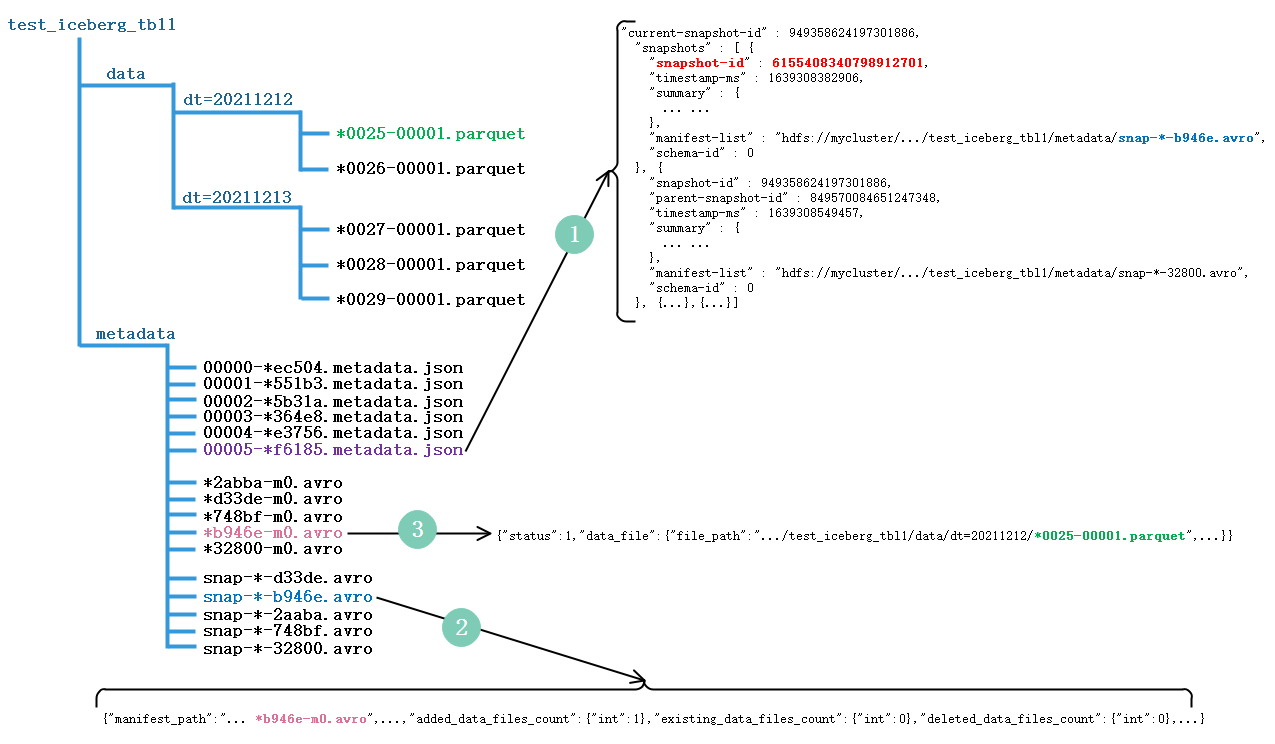
根据Manifest list找到了各个对应的manifest 清单文件，每个文件中描述了对应parquet文件存储的位置信息，可以看到在对应的avro文件中有“status”属性，该属性为1代表对应的parquet文件为新增文件，需要读取，为2代表parquet文件被删除。

1. **查询某个快照的数据**

Apache Iceberg支持查询历史上任何时刻的快照，在查询时需要指定snapshot-id属性即可，这个只能通过Spark/Flink来查询实现，例如在Spark中查询某个快照数据如下：

|  |
| --- |
| spark.read.option("snapshot-id",6155408340798912701L).format("iceberg").load("path") |

查询某个快照数据的原理如下图所示（以查询快照id为“6155408340798912701”的数据为例）：



通过上图可以看出，实际上读取历史快照数据和读取最新数据不同之处就是找到的snapshot-id不同而已，原理都是一样。

1. **根据时间戳查看某个快照的数据**

Apache iceberg还支持通过as-of-timestamp参数执行时间戳来读取某个快照的数据，同样也是通过Spark/Flink来读取，Spark读取代码如下：

|  |
| --- |
| spark.read.option("as-of-timestamp","时间戳").format("iceberg").load("path") |

实际上通过时间戳找到对应数据文件的原理与通过snapshot-id找到数据文件原理一样，在\*.metadata.json文件中，除了有“current-snapshot-id”、“snapshots”属性外还有“snapshot-log”属性，该属性对应的值如下：



我们可以看到其中有个 timestamp-ms 属性和 snapshot-id 属性，并且是按照 timestamp-ms 升序的。在 Iceberg 内部实现中，它会将 as-of-timestamp 指定的时间和 snapshot-log 数组里面每个元素的 timestamp-ms 进行比较，找出最后一个满足 timestamp-ms <= as-of-timestamp 对应的 snapshot-id，原理同上，通过snapshot-id再找到要读取的数据文件。

## Spark3.1.2与Iceberg0.12.1整合

Spark可以操作Iceberg数据湖，这里使用的Iceberg的版本为0.12.1，此版本与Spark2.4版本之上兼容。由于在Spark2.4版本中在操作Iceberg时不支持DDL、增加分区及增加分区转换、Iceberg元数据查询、insert into/overwrite等操作，建议使用Spark3.x版本来整合Iceberg0.12.1版本，这里我们使用的Spark版本是3.1.2版本。

### 向pom文件导入依赖

在Idea中创建Maven项目，在pom文件中导入以下关键依赖：

|  |
| --- |
| *<!-- 配置以下可以解决 在jdk1.8环境下打包时报错 “-source 1.5 中不支持 lambda 表达式” -->* <**properties**>  <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  <**maven.compiler.source**>1.8</**maven.compiler.source**>  <**maven.compiler.target**>1.8</**maven.compiler.target**> </**properties**>  <**dependencies**>  *<!-- Spark-core -->* <**dependency**>  <**groupId**>org.apache.spark</**groupId**>  <**artifactId**>spark-core\_2.12</**artifactId**>  <**version**>3.1.2</**version**>  </**dependency**>   *<!-- Spark与Iceberg整合的依赖包-->* <**dependency**>  <**groupId**>org.apache.iceberg</**groupId**>  <**artifactId**>iceberg-spark3</**artifactId**>  <**version**>0.12.1</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.iceberg</**groupId**>  <**artifactId**>iceberg-spark3-runtime</**artifactId**>  <**version**>0.12.1</**version**>  </**dependency**>   *<!-- avro格式 依赖包 -->* <**dependency**>  <**groupId**>org.apache.avro</**groupId**>  <**artifactId**>avro</**artifactId**>  <**version**>1.10.2</**version**>  </**dependency**>   *<!-- parquet格式 依赖包 -->* <**dependency**>  <**groupId**>org.apache.parquet</**groupId**>  <**artifactId**>parquet-hadoop</**artifactId**>  <**version**>1.12.0</**version**>  </**dependency**>   *<!-- SparkSQL -->* <**dependency**>  <**groupId**>org.apache.spark</**groupId**>  <**artifactId**>spark-sql\_2.12</**artifactId**>  <**version**>3.1.2</**version**>  </**dependency**>  *<!-- SparkSQL ON Hive-->* <**dependency**>  <**groupId**>org.apache.spark</**groupId**>  <**artifactId**>spark-hive\_2.12</**artifactId**>  <**version**>3.1.2</**version**>  </**dependency**>  *<!--&lt;!&ndash;mysql依赖的jar包&ndash;&gt;-->  <!--<dependency>-->  <!--<groupId>mysql</groupId>-->  <!--<artifactId>mysql-connector-java</artifactId>-->  <!--<version>5.1.47</version>-->  <!--</dependency>-->  <!--SparkStreaming-->* <**dependency**>  <**groupId**>org.apache.spark</**groupId**>  <**artifactId**>spark-streaming\_2.12</**artifactId**>  <**version**>3.1.2</**version**>  </**dependency**>  *<!-- SparkStreaming + Kafka -->* <**dependency**>  <**groupId**>org.apache.spark</**groupId**>  <**artifactId**>spark-streaming-kafka-0-10\_2.12</**artifactId**>  <**version**>3.1.2</**version**>  </**dependency**>  *<!--&lt;!&ndash; 向kafka 生产数据需要包 &ndash;&gt;-->  <!--<dependency>-->  <!--<groupId>org.apache.kafka</groupId>-->  <!--<artifactId>kafka-clients</artifactId>-->  <!--<version>0.10.0.0</version>-->  <!--&lt;!&ndash; 编译和测试使用jar包，没有传递性 &ndash;&gt;-->  <!--&lt;!&ndash;<scope>provided</scope>&ndash;&gt;-->  <!--</dependency>-->  <!-- StructStreaming + Kafka -->* <**dependency**>  <**groupId**>org.apache.spark</**groupId**>  <**artifactId**>spark-sql-kafka-0-10\_2.12</**artifactId**>  <**version**>3.1.2</**version**>  </**dependency**>   *<!-- Scala 包-->* <**dependency**>  <**groupId**>org.scala-lang</**groupId**>  <**artifactId**>scala-library</**artifactId**>  <**version**>2.12.14</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.scala-lang</**groupId**>  <**artifactId**>scala-compiler</**artifactId**>  <**version**>2.12.14</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.scala-lang</**groupId**>  <**artifactId**>scala-reflect</**artifactId**>  <**version**>2.12.14</**version**>  </**dependency**>   <**dependency**>  <**groupId**>log4j</**groupId**>  <**artifactId**>log4j</**artifactId**>  <**version**>1.2.12</**version**>  </**dependency**>  <**dependency**>  <**groupId**>com.google.collections</**groupId**>  <**artifactId**>google-collections</**artifactId**>  <**version**>1.0</**version**>  </**dependency**>  </**dependencies**> |

### SparkSQL设置catalog配置

以下操作主要是SparkSQL操作Iceberg,同样Spark中支持两种Catalog的设置：hive和hadoop,Hive Catalog就是iceberg表存储使用Hive默认的数据路径，Hadoop Catalog需要指定Iceberg格式表存储路径。

在SparkSQL代码中通过以下方式来指定使用的Catalog：

|  |
| --- |
| **val** spark: SparkSession = SparkSession.*builder*().master(**"local"**).appName(**"SparkOperateIceberg"**)  *//指定hive catalog, catalog名称为hive\_prod* .config(**"spark.sql.catalog.hive\_prod"**, **"org.apache.iceberg.spark.SparkCatalog"**)  .config(**"spark.sql.catalog.hive\_prod.type"**, **"hive"**)  .config(**"spark.sql.catalog.hive\_prod.uri"**, **"thrift://node1:9083"**)  .config(**"iceberg.engine.hive.enabled"**, **"true"**)  *//指定hadoop catalog，catalog名称为hadoop\_prod* .config(**"spark.sql.catalog.hadoop\_prod"**, **"org.apache.iceberg.spark.SparkCatalog"**)  .config(**"spark.sql.catalog.hadoop\_prod.type"**, **"hadoop"**)  .config(**"spark.sql.catalog.hadoop\_prod.warehouse"**, **"hdfs://mycluster/sparkoperateiceberg"**)  .getOrCreate() |

### 使用Hive Catalog管理Iceberg表

使用Hive Catalog管理Iceberg表默认数据存储在Hive对应的Warehouse目录下，在Hive中会自动创建对应的Iceberg表，SparkSQL 相当于是Hive客户端，需要额外设置“iceberg.engine.hive.enabled”属性为true，否则在Hive对应的Iceberg格式表中查询不到数据。

|  |
| --- |
| * **创建表**   *//创建表 ，hive\_pord：指定catalog名称。default:指定Hive中存在的库。test：创建的iceberg表名。* spark.sql(  **"""  | create table if not exists hive\_prod.default.test(id int,name string,age int) using iceberg  """**.stripMargin)  ***注意：***   1. ***创建表时，表名称为：${catalog名称}.${Hive中库名}.${创建的Iceberg格式表名}***   ***2）表创建之后，可以在Hive中查询到对应的test表，创建的是Hive外表，在对应的Hive warehouse 目录下可以看到对应的数据目录。***     * **插入数据**   *//插入数据* spark.sql(  **"""  |insert into hive\_prod.default.test values (1,"zs",18),(2,"ls",19),(3,"ww",20)  """**.stripMargin)   * **查询数据**   *//查询数据* spark.sql(  **"""  |select \* from hive\_prod.default.test  """**.stripMargin).show()  ***结果如下：***    ***在Hive对应的test表中也能查询到数据：***     * **删除表**   *//删除表，删除表对应的数据不会被删除* spark.sql(  **"""  |drop table hive\_prod.default.test  """**.stripMargin)  ***注意：删除表后，数据会被删除，但是表目录还是存在，如果彻底删除数据，需要把对应的表目录删除。*** |

### 使用Hadoop Catalog管理Iceberg表

使用Hadoop Catalog管理表，需要指定对应Iceberg存储数据的目录。

|  |
| --- |
| * **创建表**   *//创建表 ，hadoop\_prod：指定Hadoop catalog名称。default:指定库名称。test：创建的iceberg表名。*  spark.sql(  **"""  | create table if not exists hadoop\_prod.default.test(id int,name string,age int) using iceberg  """**.stripMargin)  ***注意：***   1. ***创建表名称为：${Hadoop Catalog名称}.${随意定义的库名}.${Iceberg格式表名}***   ***2）创建表后，会在hadoop\_prod名称对应的目录下创建该表***     * **插入数据**   *//插入数据* spark.sql(  **"""  |insert into hadoop\_prod.default.test values (1,"zs",18),(2,"ls",19),(3,"ww",20)  """**.stripMargin)   * **查询数据**   spark.sql(  **"""  |select \* from hadoop\_prod.default.test  """**.stripMargin).show()     * **创建对应的Hive表映射数据**   在Hive表中执行如下建表语句：  CREATE TABLE hdfs\_iceberg (  id int,  name string,  age int  )  STORED BY 'org.apache.iceberg.mr.hive.HiveIcebergStorageHandler'  LOCATION 'hdfs://mycluster/sparkoperateiceberg/default/test'  TBLPROPERTIES ('iceberg.catalog'='location\_based\_table');  在Hive中查询“hdfs\_iceberg”表数据如下：     * **删除表**   spark.sql(  **"""  |drop table hadoop\_prod.default.test  """**.stripMargin)  ***注意：删除iceberg表后，数据被删除，对应的库目录存在。*** |

### Spark与Iceberg整合DDL操作

这里使用Hadoop Catalog 来演示Spark 与Iceberg的DDL操作。

#### CREATE TABLE 创建表

Create table 创建Iceberg表，创建表不仅可以创建普通表还可以创建分区表，再向分区表中插入一批数据时，必须对数据中分区列进行排序，否则会出现文件关闭错误，代码如下：

|  |
| --- |
| **val** spark: SparkSession = SparkSession.*builder*().master(**"local"**).appName(**"SparkOperateIceberg"**)  *//指定hadoop catalog，catalog名称为hadoop\_prod* .config(**"spark.sql.catalog.hadoop\_prod"**, **"org.apache.iceberg.spark.SparkCatalog"**)  .config(**"spark.sql.catalog.hadoop\_prod.type"**, **"hadoop"**)  .config(**"spark.sql.catalog.hadoop\_prod.warehouse"**, **"hdfs://mycluster/sparkoperateiceberg"**)  .getOrCreate()  *//创建普通表* spark.sql(  **"""  | create table if not exists hadoop\_prod.default.normal\_tbl(id int,name string,age int) using iceberg  """**.stripMargin)  *//创建分区表,以 loc 列为分区字段* spark.sql(  **"""  |create table if not exists hadoop\_prod.default.partition\_tbl(id int,name string,age int,loc string) using iceberg partitioned by (loc)  """**.stripMargin)  *//向分区表中插入数据时，****必须对分区列排序****，否则报错：java.lang.IllegalStateException: Already closed files for partition:xxx* spark.sql(  **"""  |insert into table hadoop\_prod.default.partition\_tbl values (1,"zs",18,"beijing"),(3,"ww",20,"beijing"),(2,"ls",19,"shanghai"),(4,"ml",21,"shagnhai")  """**.stripMargin) spark.sql(**"select \* from hadoop\_prod.default.partition\_tbl"**).show()  ***查询结果如下：*** |

创建Iceberg分区时，还可以通过一些转换表达式对timestamp列来进行转换，创建**隐藏分区**，常用的转换表达式有如下几种：

* years(ts):按照年分区

|  |
| --- |
| *//创建分区表 partition\_tbl1 ,指定分区为year* spark.sql(  **"""  |create table if not exists hadoop\_prod.default.partition\_tbl1(id int ,name string,age int,regist\_ts timestamp) using iceberg  |partitioned by (years(regist\_ts))  """**.stripMargin)  *//向表中插入数据，注意，插入的数据需要提前排序，必须排序，只要是相同日期数据写在一起就可以 //(1,'zs',18,1608469830) --"2020-12-20 21:10:30" //(2,'ls',19,1634559630) --"2021-10-18 20:20:30" //(3,'ww',20,1603096230) --"2020-10-19 16:30:30" //(4,'ml',21,1639920630) --"2021-12-19 21:30:30" //(5,'tq',22,1608279630) --"2020-12-18 16:20:30" //(6,'gb',23,1576843830) --"2019-12-20 20:10:30"* spark.sql(  **"""  |insert into hadoop\_prod.default.partition\_tbl1 values  |(1,'zs',18,cast(1608469830 as timestamp)),  |(3,'ww',20,cast(1603096230 as timestamp)),  |(5,'tq',22,cast(1608279630 as timestamp)),  |(2,'ls',19,cast(1634559630 as timestamp)),  |(4,'ml',21,cast(1639920630 as timestamp)),  |(6,'gb',23,cast(1576843830 as timestamp))  """**.stripMargin)  *//查询结果* spark.sql(  **"""  |select \* from hadoop\_prod.default.partition\_tbl1  """**.stripMargin).show()  ***数据结果如下：***    ***在HDFS中是按照年进行分区：*** |

* months(ts):按照“年-月”月级别分区

|  |
| --- |
| *//创建分区表 partition\_tbl2 ,指定分区为months,会按照“年-月”分区* spark.sql(  **"""  |create table if not exists hadoop\_prod.default.partition\_tbl2(id int ,name string,age int,regist\_ts timestamp) using iceberg  |partitioned by (months(regist\_ts))  """**.stripMargin)  *//向表中插入数据，注意，插入的数据需要提前排序，必须排序，只要是相同日期数据写在一起就可以 //(1,'zs',18,1608469830) --"2020-12-20 21:10:30" //(2,'ls',19,1634559630) --"2021-10-18 20:20:30" //(3,'ww',20,1603096230) --"2020-10-19 16:30:30" //(4,'ml',21,1639920630) --"2021-12-19 21:30:30" //(5,'tq',22,1608279630) --"2020-12-18 16:20:30" //(6,'gb',23,1576843830) --"2019-12-20 20:10:30"* spark.sql(  **"""  |insert into hadoop\_prod.default.partition\_tbl2 values  |(1,'zs',18,cast(1608469830 as timestamp)),  |(5,'tq',22,cast(1608279630 as timestamp)),  |(2,'ls',19,cast(1634559630 as timestamp)),  |(3,'ww',20,cast(1603096230 as timestamp)),  |(4,'ml',21,cast(1639920630 as timestamp)),  |(6,'gb',23,cast(1576843830 as timestamp))  """**.stripMargin)  *//查询结果* spark.sql(  **"""  |select \* from hadoop\_prod.default.partition\_tbl2  """**.stripMargin).show()  ***数据结果如下：***    ***在HDFS中是按照“年-月”进行分区：*** |

* days(ts)或者date(ts):按照“年-月-日”天级别分区

|  |
| --- |
| *//创建分区表 partition\_tbl3 ,指定分区为 days,会按照“年-月-日”分区* spark.sql(  **"""  |create table if not exists hadoop\_prod.default.partition\_tbl3(id int ,name string,age int,regist\_ts timestamp) using iceberg  |partitioned by (days(regist\_ts))  """**.stripMargin)  *//向表中插入数据，注意，插入的数据需要提前排序，必须排序，只要是相同日期数据写在一起就可以 //(1,'zs',18,1608469830) --"2020-12-20 21:10:30" //(2,'ls',19,1634559630) --"2021-10-18 20:20:30" //(3,'ww',20,1603096230) --"2020-10-19 16:30:30" //(4,'ml',21,1639920630) --"2021-12-19 21:30:30" //(5,'tq',22,1608279630) --"2020-12-18 16:20:30" //(6,'gb',23,1576843830) --"2019-12-20 20:10:30"* spark.sql(  **"""  |insert into hadoop\_prod.default.partition\_tbl3 values  |(1,'zs',18,cast(1608469830 as timestamp)),  |(5,'tq',22,cast(1608279630 as timestamp)),  |(2,'ls',19,cast(1634559630 as timestamp)),  |(3,'ww',20,cast(1603096230 as timestamp)),  |(4,'ml',21,cast(1639920630 as timestamp)),  |(6,'gb',23,cast(1576843830 as timestamp))  """**.stripMargin)  *//查询结果* spark.sql(  **"""  |select \* from hadoop\_prod.default.partition\_tbl3  """**.stripMargin).show()  ***数据结果如下：***    ***在HDFS中是按照“年-月-日”进行分区：*** |

* hours(ts)或者date\_hour(ts):按照“年-月-日-时”小时级别分区

|  |
| --- |
| *//创建分区表 partition\_tbl4 ,指定分区为 hours,会按照“年-月-日-时”分区* spark.sql(  **"""  |create table if not exists hadoop\_prod.default.partition\_tbl4(id int ,name string,age int,regist\_ts timestamp) using iceberg  |partitioned by (hours(regist\_ts))  """**.stripMargin)  *//向表中插入数据，注意，插入的数据需要提前排序，必须排序，只要是相同日期数据写在一起就可以 //(1,'zs',18,1608469830) --"2020-12-20 21:10:30" //(2,'ls',19,1634559630) --"2021-10-18 20:20:30" //(3,'ww',20,1603096230) --"2020-10-19 16:30:30" //(4,'ml',21,1639920630) --"2021-12-19 21:30:30" //(5,'tq',22,1608279630) --"2020-12-18 16:20:30" //(6,'gb',23,1576843830) --"2019-12-20 20:10:30"* spark.sql(  **"""  |insert into hadoop\_prod.default.partition\_tbl4 values  |(1,'zs',18,cast(1608469830 as timestamp)),  |(5,'tq',22,cast(1608279630 as timestamp)),  |(2,'ls',19,cast(1634559630 as timestamp)),  |(3,'ww',20,cast(1603096230 as timestamp)),  |(4,'ml',21,cast(1639920630 as timestamp)),  |(6,'gb',23,cast(1576843830 as timestamp))  """**.stripMargin)  *//查询结果* spark.sql(  **"""  |select \* from hadoop\_prod.default.partition\_tbl4  """**.stripMargin).show()  ***数据结果如下：***    ***在HDFS中是按照“年-月-日-时”进行分区：*** |

Iceberg支持的时间分区目前和将来只支持UTC,UTC是国际时，UTC+8就是国际时加八小时，是东八区时间,也就是北京时间，所以我们看到上面分区时间与数据时间不一致。

除了以上常用的时间隐藏分区外，Iceberg还支持bucket(N,col)分区，这种分区方式可以按照某列的hash值与N取余决定数据去往的分区。truncate(L,col)，这种隐藏分区可以将字符串列截取L长度，相同的数据会被分到相同分区中。

#### CREATE TAEBL ... AS SELECT

Iceberg支持“create table .... as select ”语法，可以从查询语句中创建一张表，并插入对应的数据，操作如下：

1. **创建表hadoop\_prod.default.mytbl,并插入数据**

|  |
| --- |
| **val** spark: SparkSession = SparkSession.*builder*().master(**"local"**).appName(**"SparkOperateIceberg"**)  *//指定hadoop catalog，catalog名称为hadoop\_prod* .config(**"spark.sql.catalog.hadoop\_prod"**, **"org.apache.iceberg.spark.SparkCatalog"**)  .config(**"spark.sql.catalog.hadoop\_prod.type"**, **"hadoop"**)  .config(**"spark.sql.catalog.hadoop\_prod.warehouse"**, **"hdfs://mycluster/sparkoperateiceberg"**)  .getOrCreate()  *//创建普通表* spark.sql(  **"""  | create table hadoop\_prod.default.mytbl(id int,name string,age int) using iceberg  """**.stripMargin)  *//向表中插入数据* spark.sql(  **"""  |insert into table hadoop\_prod.default.mytbl values (1,"zs",18),(3,"ww",20),(2,"ls",19),(4,"ml",21)  """**.stripMargin)  *//查询数据* spark.sql(**"select \* from hadoop\_prod.default.mytbl"**).show() |

1. **使用“create table ... as select”语法创建表mytal2并查询**

|  |
| --- |
| spark.sql(  **"""  |create table hadoop\_prod.default.mytbl2 using iceberg as select id,name,age from hadoop\_prod.default.mytbl  """**.stripMargin) spark.sql(  **"""  |select \* from hadoop\_prod.default.mytbl2  """**.stripMargin).show()  ***结果如下：*** |

#### REPLACE TABLE ... AS SELECT

Iceberg支持“replace table .... as select ”语法，可以从查询语句中重建一张表，并插入对应的数据，操作如下：

1. **创建表“hadoop\_prod.default.mytbl3”，并插入数据、展示**

|  |
| --- |
| spark.sql(  **"""  |create table hadoop\_prod.default.mytbl3 (id int,name string,loc string,score int) using iceberg  """**.stripMargin) spark.sql(  **"""  |insert into table hadoop\_prod.default.mytbl3 values (1,"zs","beijing",100),(2,"ls","shanghai",200)  """**.stripMargin) spark.sql(  **"""  |select \* from hadoop\_prod.default.mytbl3  """**.stripMargin).show |

1. **重建表“hadoop\_prod.default.mytbl3”，并插入对应数据**

|  |
| --- |
| spark.sql(  **"""  |replace table hadoop\_prod.default.mytbl2 using iceberg as select \* from hadoop\_prod.default.mytbl3  """**.stripMargin)  spark.sql(  **"""  |select \* from hadoop\_prod.default.mytbl2  """**.stripMargin).show() |

#### DROP TABLE

删除iceberg表时直接执行:“drop table xxx”语句即可，删除表时，表数据会被删除，但是库目录存在。

|  |
| --- |
| *//删除表* spark.sql(  **"""  |drop table hadoop\_prod.default.mytbl  """**.stripMargin) |

#### ALTER TABLE

Iceberg的alter操作在Spark3.x版本中支持，alter一般包含以下操作：

* 添加、删除列

添加列操作：ALTER TABLE ... ADD COLUMN

删除列操作：ALTER TABLE ... DROP COLUMN

|  |
| --- |
| *//1.创建表test，并插入数据、查询*  spark.sql(  **"""  |create table hadoop\_prod.default.test(id int,name string,age int) using iceberg  """**.stripMargin) spark.sql(  **"""  |insert into table hadoop\_prod.default.test values (1,"zs",18),(2,"ls",19),(3,"ww",20)  """**.stripMargin)  spark.sql(  **"""  | select \* from hadoop\_prod.default.test  """**.stripMargin).show()  *//2.添加字段,给 test表增加 gender 列、loc列* spark.sql(  **"""  |alter table hadoop\_prod.default.test add column gender string,loc string  """**.stripMargin)  *//3.删除字段，给test 表删除age 列* spark.sql(  **"""  |alter table hadoop\_prod.default.test drop column age  """**.stripMargin)  *//4.查看表test数据* spark.sql(  **"""  |select \* from hadoop\_prod.default.test  """**.stripMargin).show()  ***最终表展示的列少了age列，多了gender、loc列：*** |

* 重命名列

重命名列语法：ALTER TABLE ... RENAME COLUMN，操作如下:

|  |
| --- |
| *//5.重命名列* spark.sql(  **"""  |alter table hadoop\_prod.default.test rename column gender to xxx  |  """**.stripMargin) spark.sql(  **"""  |select \* from hadoop\_prod.default.test  """**.stripMargin).show()  ***最终表展示的列 gender列变成了xxx列：*** |

#### ALTER TABLE 分区操作

alter 分区操作包括增加分区和删除分区操作，这种分区操作在Spark3.x之后被支持，spark2.4版本不支持，并且使用时，必须在spark配置中加入spark.sql.extensions属性，其值为：org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions，在添加分区时还支持分区转换，语法如下：

* 添加分区语法：ALTER TABLE ... ADD PARTITION FIELD
* 删除分区语法：ALTER TABLE ... DROP PARTITION FIELD

具体操作如下：

1. **创建表mytbl，并插入数据**

|  |
| --- |
| **val** spark: SparkSession = SparkSession.*builder*().master(**"local"**).appName(**"SparkOperateIceberg"**)  *//指定hadoop catalog，catalog名称为hadoop\_prod* .config(**"spark.sql.catalog.hadoop\_prod"**, **"org.apache.iceberg.spark.SparkCatalog"**)  .config(**"spark.sql.catalog.hadoop\_prod.type"**, **"hadoop"**)  .config(**"spark.sql.catalog.hadoop\_prod.warehouse"**, **"hdfs://mycluster/sparkoperateiceberg"**)  **.config("spark.sql.extensions","org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions")**  .getOrCreate()  *//1.创建普通表* spark.sql(  **"""  | create table hadoop\_prod.default.mytbl(id int,name string,loc string,ts timestamp) using iceberg  """**.stripMargin)  *//2.向表中插入数据，并查询* spark.sql(  **"""  |insert into hadoop\_prod.default.mytbl values  |(1,'zs',"beijing",cast(1608469830 as timestamp)),  |(3,'ww',"shanghai",cast(1603096230 as timestamp))  """**.stripMargin) spark.sql(**"select \* from hadoop\_prod.default.mytbl"**).show()  ***在HDFS中数据存储和结果如下：*** |

1. **将表loc列添加为分区列，并插入数据，查询**

|  |
| --- |
| *//3.将 loc 列添加成分区,必须添加 config("spark.sql.extensions", "org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions") 配置* spark.sql(  **"""  |alter table hadoop\_prod.default.mytbl add partition field loc  """**.stripMargin)  *//4.向表 mytbl中继续插入数据，之前数据没有分区，之后数据有分区* spark.sql(  **"""  |insert into hadoop\_prod.default.mytbl values  |(5,'tq',"hangzhou",cast(1608279630 as timestamp)),  |(2,'ls',"shandong",cast(1634559630 as timestamp))  """**.stripMargin ) spark.sql(**"select \* from hadoop\_prod.default.mytbl"**).show()  ***在HDFS中数据存储和结果如下：*** |

**注意：添加分区字段是元数据操作，不会改变现有的表数据，新数据将使用新分区写入数据，现有数据将继续保留在原有的布局中。**

1. **将ts列进行转换作为分区列，插入数据并查询**

|  |
| --- |
| *//5.将 ts 列通过分区转换添加为分区列* spark.sql(  **"""  |alter table hadoop\_prod.default.mytbl add partition field years(ts)  """**.stripMargin)  *//6.向表 mytbl中继续插入数据，之前数据没有分区，之后数据有分区* spark.sql(  **"""  |insert into hadoop\_prod.default.mytbl values  |(4,'ml',"beijing",cast(1639920630 as timestamp)),  |(6,'gb',"tianjin",cast(1576843830 as timestamp))  """**.stripMargin ) spark.sql(**"select \* from hadoop\_prod.default.mytbl"**).show()  ***在HDFS中数据存储和结果如下：*** |

1. **删除分区loc**

|  |
| --- |
| *//7.删除表 mytbl 中的loc分区* spark.sql(  **"""  |alter table hadoop\_prod.default.mytbl drop partition field loc  """**.stripMargin) *//8.继续向表 mytbl 中插入数据，并查询* spark.sql(  **"""  |insert into hadoop\_prod.default.mytbl values  |(4,'ml',"beijing",cast(1639920630 as timestamp)),  |(6,'gb',"tianjin",cast(1576843830 as timestamp))  """**.stripMargin ) spark.sql(**"select \* from hadoop\_prod.default.mytbl"**).show()  ***在HDFS中数据存储和结果如下：***    **注意：由于表中还有ts分区转换之后对应的分区，所以继续插入的数据loc分区为null** |

1. **删除分区years(ts)**

|  |
| --- |
| *//9.删除表 mytbl 中的years(ts) 分区* spark.sql(  **"""  |alter table hadoop\_prod.default.mytbl drop partition field years(ts)  """**.stripMargin) *//10.继续向表 mytbl 中插入数据，并查询* spark.sql(  **"""  |insert into hadoop\_prod.default.mytbl values  |(5,'tq',"hangzhou",cast(1608279630 as timestamp)),  |(2,'ls',"shandong",cast(1634559630 as timestamp))  """**.stripMargin ) spark.sql(**"select \* from hadoop\_prod.default.mytbl"**).show()  ***在HDFS中数据存储和结果如下：*** |

### Spark与Iceberg整合查询操作

#### DataFrame API加载Iceberg中的数据

Spark操作Iceberg不仅可以使用SQL方式查询Iceberg中的数据，还可以使用DataFrame方式加载Iceberg表中的数据，可以通过spark.table(Iceberg表名)或者spark.read.format("iceberg").load("iceberg data path")来加载对应Iceberg表中的数据，操作如下：

|  |
| --- |
| **val** spark: SparkSession = SparkSession.*builder*().master(**"local"**).appName(**"test"**)  *//指定hadoop catalog，catalog名称为hadoop\_prod* .config(**"spark.sql.catalog.hadoop\_prod"**, **"org.apache.iceberg.spark.SparkCatalog"**)  .config(**"spark.sql.catalog.hadoop\_prod.type"**, **"hadoop"**)  .config(**"spark.sql.catalog.hadoop\_prod.warehouse"**, **"hdfs://mycluster/sparkoperateiceberg"**)  .config(**"spark.sql.extensions"**, **"org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions"**)  .getOrCreate()  *//1.创建Iceberg表，并插入数据* spark.sql(  **"""  |create table hadoop\_prod.mydb.mytest (id int,name string,age int) using iceberg  """**.stripMargin) spark.sql(  **"""  |insert into hadoop\_prod.mydb.mytest values (1,"zs",18),(2,"ls",19),(3,"ww",20)  """**.stripMargin) *//1.SQL 方式读取Iceberg中的数据* spark.sql(**"select \* from hadoop\_prod.mydb.mytest"**).show()  */\*\*  \* 2.使用Spark查询Iceberg中的表除了使用sql 方式之外，还可以使用DataFrame方式,建议使用SQL方式  \*/* ***//第一种方式使用DataFrame方式查询Iceberg表数据*****val** frame1: DataFrame = spark.table(**"hadoop\_prod.mydb.mytest"**) frame1.show()  ***//第二种方式使用DataFrame加载 Iceberg表数据*****val** frame2: DataFrame = spark.read.format(**"iceberg"**).load(**"hdfs://mycluster/sparkoperateiceberg/mydb/mytest"**) frame2.show() |

#### 查询表快照

每次向Iceberg表中commit数据都会生成对应的一个快照，我们可以通过查询“***${catalog名称}.${库名}.${Iceberg表}.snapshots***”来查询对应Iceberg表中拥有的所有快照，操作如下：

|  |
| --- |
| *//向表 hadoop\_prod.mydb.mytest 中再次插入以下数据* spark.sql(  **"""  |insert into hadoop\_prod.mydb.mytest values (4,"ml",18),(5,"tq",19),(6,"gb",20)  """**.stripMargin)  *//3.查看Iceberg表快照信息* spark.sql(  **"""  |select \* from hadoop\_prod.mydb.mytest.snapshots  """**.stripMargin).show(**false**)  ***结果如下：*** |

#### 查询表历史

对Iceberg表查询表历史就是查询Iceberg表快照信息内容，与查询表快照类似，通过“***${catalog名称}.${库名}.${Iceberg表}.history***”命令进行查询，操作如下：

|  |
| --- |
| *//4.查询表历史,实际上就是表快照的部分内容* spark.sql(  **"""  |select \* from hadoop\_prod.mydb.mytest.history  """**.stripMargin).show(**false**)  ***结果如下：*** |

#### 查询表data files

我们可以通过”***${catalog名称}.${库名}.${Iceberg表}.files***”命令来查询Iceberg表对应的data files 信息，操作如下：

|  |
| --- |
| *//5.查看表对应的data files* spark.sql(  **"""  |select \* from hadoop\_prod.mydb.mytest.files  """**.stripMargin).show(**false**)  ***结果如下：*** |

#### 查询Manifests

我们可以通过“***${catalog名称}.${库名}.${Iceberg表}.manifests***”来查询表对应的manifests信息，具体操作如下：

|  |
| --- |
| *//6.查看表对应的 Manifests* spark.sql(  **"""  |select \* from hadoop\_prod.mydb.mytest.manifests  """**.stripMargin).show(**false**)  ***结果如下：*** |

#### 查询指定快照数据

查询Iceberg表数据还可以指定***snapshot-id***来查询指定快照的数据，这种方式可以使用DataFrame Api方式来查询，Spark3.x版本之后也可以通过SQL 方式来查询，操作如下：

|  |
| --- |
| *//7.查询指定快照数据,快照ID可以通过读取json元数据文件获取* spark.read  .option(**"snapshot-id"**,3368002881426159310L)  .format(**"iceberg"**)  .load(**"hdfs://mycluster/sparkoperateiceberg/mydb/mytest"**)  .show()  ***结果如下：*** |

Spark3.x 版本之后，SQL指定快照语法为：

|  |
| --- |
| ***CALL ${Catalog 名称}.system.set\_current\_snapshot("${库名.表名}",快照ID)*** |

操作如下：

|  |
| --- |
| *//SQL 方式指定查询快照ID 数据* spark.sql(  **"""  |call hadoop\_prod.system.set\_current\_snapshot('mydb.mytest',3368002881426159310)  """**.stripMargin) spark.sql(  **"""  |select \* from hadoop\_prod.mydb.mytest  """**.stripMargin).show()  ***结果如下：*** |

#### 根据时间戳查询数据

Spark读取Iceberg表可以指定“***as-of-timestamp***”参数，通过指定一个毫秒时间参数查询Iceberg表中数据，iceberg会根据元数据找出***timestamp-ms <= as-of-timestamp 对应的 snapshot-id*** ，也只能通过DataFrame Api把数据查询出来，Spark3.x版本之后支持SQL指定时间戳查询数据。具体操作如下：

|  |
| --- |
| *//8.根据时间戳查询数据,时间戳指定成毫秒，iceberg会根据元数据找出timestamp-ms <= as-of-timestamp 对应的 snapshot-id ，把数据查询出来* spark.read.option(**"as-of-timestamp"**,**"1640066148000"**)  .format(**"iceberg"**)  .load(**"hdfs://mycluster/sparkoperateiceberg/mydb/mytest"**)  .show()  ***结果如下：*** |

Spark3.x 版本之后，SQL根据时间戳查询最近快照语法为：

|  |
| --- |
| ***CALL ${Catalog 名称}.system.rollback\_to\_timestamp("${库名.表名}",TIMESTAMP '日期数据')*** |

操作如下：

|  |
| --- |
| *//省略重新创建表mytest，两次插入数据*  *//SQL 方式查询指定 时间戳 快照数据* spark.sql(  **"""  |CALL hadoop\_prod.system.rollback\_to\_timestamp('mydb.mytest', TIMESTAMP '2021-12-23 16:56:40.000')  """**.stripMargin) spark.sql(  **"""  |select \* from hadoop\_prod.mydb.mytest  """**.stripMargin).show()  ***结果如下：*** |

#### 回滚快照

在Iceberg中可以回滚快照，可以借助于Java 代码实现，Spark DataFrame Api 不能回滚快照，在Spark3.x版本之后，支持SQL回滚快照。回滚快照之后，Iceberg对应的表中会生成新的Snapshot-id,重新查询，回滚生效，具体操作如下：

|  |
| --- |
| *//9.回滚到某个快照，rollbackTo(snapshot-id)，指定的是固定的某个快照ID，回滚之后，会生成新的Snapshot-id, 重新查询生效。* **val** conf = **new** Configuration() **val** catalog = **new** HadoopCatalog(conf,**"hdfs://mycluster/sparkoperateiceberg"**) catalog.setConf(conf) **val** table: Table = catalog.loadTable(TableIdentifier.*of*(**"mydb"**,**"mytest"**)) table.manageSnapshots().rollbackTo(3368002881426159310L).commit()  ***注意：回滚快照之后，在对应的Iceberg表中会生成新的Snapshot-id,再次查询后，会看到数据是回滚快照之后的数据。***  *//查询表 hadoop\_prod.mydb.mytest 数据,已经是历史数据* spark.sql(  **"""  |select \* from hadoop\_prod.mydb.mytest  """**.stripMargin).show(100)  ***结果如下：*** |

Spark3.x 版本之后，SQL回滚快照语法为：

|  |
| --- |
| ***CALL ${Catalog 名称}.system.rollback\_to\_snapshot("${库名.表名}",快照ID)*** |

操作如下：

|  |
| --- |
| *//省略重新创建表mytest，两次插入数据*  *//SQL方式回滚快照ID，操作如下：* spark.sql(  **"""  |Call hadoop\_prod.system.rollback\_to\_snapshot("mydb.mytest",5440886662709904549)  """**.stripMargin)  *//查询表 hadoop\_prod.mydb.mytest 数据,已经是历史数据* spark.sql(  **"""  |select \* from hadoop\_prod.mydb.mytest  """**.stripMargin).show(100)  ***结果如下：*** |

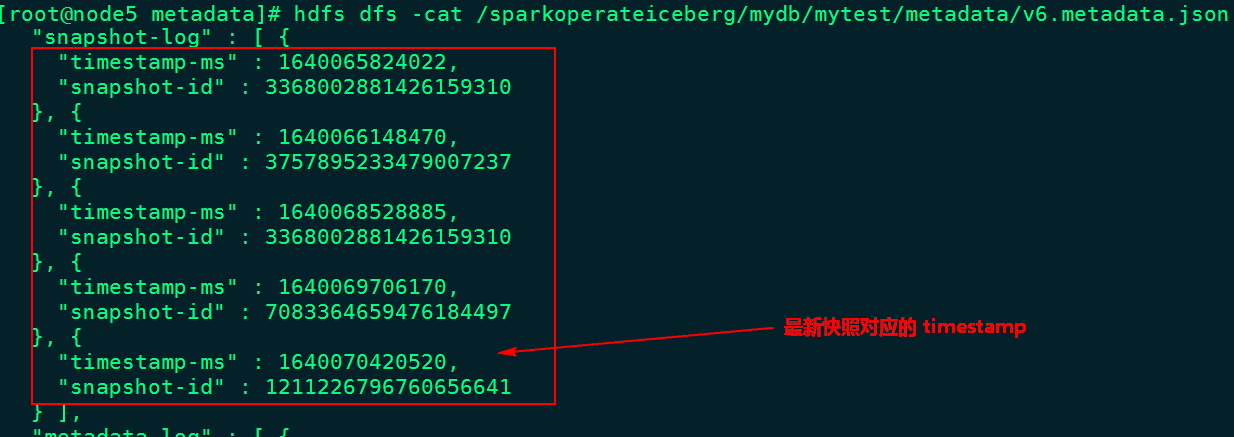
#### 合并Iceberg表的数据文件

针对Iceberg表每次commit都会生成一个parquet数据文件，有可能一张Iceberg表对应的数据文件非常多，那么我们通过Java Api 方式对Iceberg表可以进行数据文件合并，数据文件合并之后，会生成新的Snapshot且原有数据并不会被删除，如果要删除对应的数据文件需要通过“***Expire Snapshots来实现***”，具体操作如下：

|  |
| --- |
| *//10.合并Iceberg表的数据文件 // 1) 首先向表 mytest 中插入一批数据，将数据写入到表mytest中* **import** spark.implicits.\_ **val** df: DataFrame = spark.read.textFile(**"D:\\2018IDEA\_space\\Iceberg-Spark-Flink\\SparkIcebergOperate\\data\\nameinfo"**)  .map(line => {  **val** arr: Array[String] = line.split(**","**)  (arr(0).toInt, arr(1), arr(2).toInt)  }).toDF(**"id"**,**"name"**,**"age"**) df.writeTo(**"hadoop\_prod.mydb.mytest"**).append()  ***经过以上插入数据，我们可以看到Iceberg表元数据目录如下：***    ***数据目录如下：***    *//2) 合并小文件数据,Iceberg合并小文件时并不会删除被合并的文件，Compact是将小文件合并成大文件并创建新的Snapshot。如果要删除文件需要通过Expire Snapshots来实现,targetSizeInBytes 指定合并后的每个文件大小* **val** conf = **new** Configuration() **val** catalog = **new** HadoopCatalog(conf,**"hdfs://mycluster/sparkoperateiceberg"**) **val** table: Table = catalog.loadTable(TableIdentifier.*of*(**"mydb"**,**"mytest"**)) Actions.*forTable*(table).rewriteDataFiles().targetSizeInBytes(1024)*//1kb，指定生成合并之后文件大小* .execute()  ***合并小文件后，Iceberg对应表元数据目录如下：***    ***数据目录如下：*** |

#### 删除历史快照

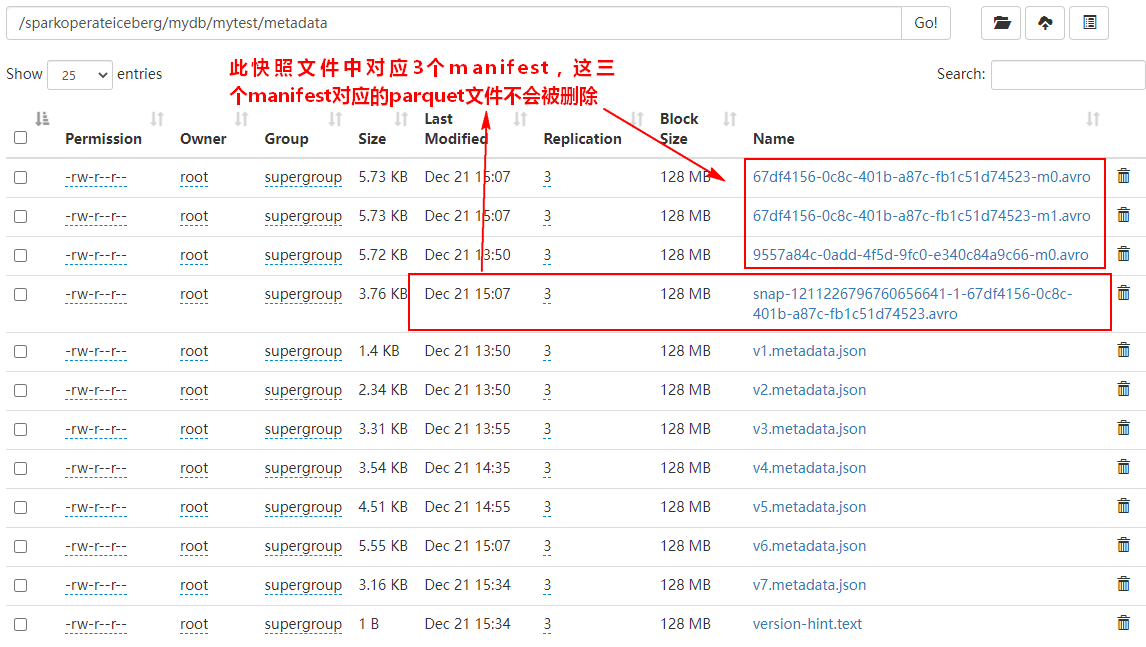
目前我们可以通过Java Api 删除历史快照，可以通过指定时间戳，当前时间戳之前的所有快照都会被删除（如果指定时间比最后一个快照时间还大，会保留最新快照数据），可以通过查看最新元数据json文件来查找要指定的时间。例如，表mytest 最新的json元数据文件信息如下:



这里删除时间为“1640070000000”之前的所有快照信息，在删除快照时，数据data目录中过期的数据parquet文件也会被删除（例如：快照回滚后不再需要的文件），代码操作如下：

|  |
| --- |
| *//11.删除历史快照,历史快照是通过ExpireSnapshot来实现的，设置需要删除多久的历史快照* **val** conf = **new** Configuration() **val** catalog = **new** HadoopCatalog(conf,**"hdfs://mycluster/sparkoperateiceberg"**) **val** table: Table = catalog.loadTable(TableIdentifier.*of*(**"mydb"**,**"mytest"**)) table.expireSnapshots().expireOlderThan(1640070000000L).commit()  ***以上代码执行完成之后，可以看到只剩下最后一个快照信息：***    ***数据目录如下：*** |

注意：删除对应快照数据时，Iceberg表对应的Parquet格式数据也会被删除，到底哪些parquet文件数据被删除决定于最后的“snap-xx.avro”中对应的manifest list数据对应的parquet数据，如下图所示：



随着不断删除snapshot，在Iceberg表不再有manifest文件对应的parquet文件也会被删除。

除了以上这种使用Java Api方式来删除表旧快照外，在Spark3.x版本之后，我们还可以使用SQL方式来删除快照方式，SQL删除快照语法为：

|  |
| --- |
| **//删除早于某个时间的快照，但保留最近N个快照**  ***CALL ${Catalog 名称}.system.expire\_snapshots("${库名.表名}",TIMESTAMP '年-月-日 时-分-秒.000',N)*** |

**注意：以上使用SQL方式采用上述方式进行操作时，SparkSQL执行会卡住，最后报错广播变量广播问题（没有找到好的解决方式，目测是个bug问题）**

每次Commit生成对应的Snapshot之外，还会有一份元数据文件“vX-metadata.json”文件产生，我们可以在创建Iceberg表时执行对应的属性决定Iceberg表保留几个元数据文件，属性如下：

|  |  |
| --- | --- |
| **Property** | **Description** |
| write.metadata.delete-after-commit.enabled | 每次表提交后是否删除旧的元数据文件 |
| write.metadata.previous-versions-max | 要保留旧的元数据文件数量 |

例如，在Spark中创建表 test ,指定以上两个属性，建表语句如下：

|  |
| --- |
| CREATE TABLE ${CataLog名称}.${库名}.${表名} (  id bigint,  name string  ) using iceberg  PARTITIONED BY (  loc string  ) **TBLPROPERTIES (**  **'write.metadata.delete-after-commit.enabled'= 'true',**  **'write.metadata.previous-versions-max' = '3'**  **)** |

### Spark与Iceberg整合写操作

#### INSERT INTO

"insert into"是向Iceberg表中插入数据，有两种语法形式："INSERT INTO tbl VALUES (1,"zs",18),(2,"ls",19)"、"INSERT INTO tbl SELECT ..."，以上两种方式比较简单，这里不再详细记录。

#### MERGE INTO

Iceberg "merge into"语法可以对表数据进行行级更新或删除，在Spark3.x版本之后支持，其原理是重写包含需要删除和更新行数据所在的data files。"merge into"可以使用一个查询结果数据来更新目标表的数据，其语法通过类似join关联方式，根据指定的匹配条件对匹配的行数据进行相应操作。"merge into"语法如下：

|  |
| --- |
| MERGE INTO tbl t  USING (SELECT ...) s  ON t.id = s.id  WHEN MATCHED AND ... THEN DELETE //删除  WHEN MATCHED AND ... THEN UPDATE SET ... //更新  WHEN MATCHED AND ... AND ... THEN UPDATE SET ... //多条件更新  WHEN NOT MATCHED ADN ... THEN INSERT (col1,col2...) VALUES(s.col1,s.col2 ...)//匹配不上向目标表插入数据 |

具体案例如下：

1. **首先创建a表和b表，并插入数据**

|  |
| --- |
| **val** spark: SparkSession = SparkSession.*builder*().master(**"local"**).appName(**"SparkOperateIceberg"**)  *//指定hadoop catalog，catalog名称为hadoop\_prod* .config(**"spark.sql.catalog.hadoop\_prod"**, **"org.apache.iceberg.spark.SparkCatalog"**)  .config(**"spark.sql.catalog.hadoop\_prod.type"**, **"hadoop"**)  .config(**"spark.sql.catalog.hadoop\_prod.warehouse"**, **"hdfs://mycluster/sparkoperateiceberg"**)  .config(**"spark.sql.extensions"**, **"org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions"**)  .getOrCreate()  *//创建一张表 a ，并插入数据* spark.sql(  **"""  |create table hadoop\_prod.default.a (id int,name string,age int) using iceberg  """**.stripMargin) spark.sql(  **"""  |insert into hadoop\_prod.default.a values (1,"zs",18),(2,"ls",19),(3,"ww",20)  """**.stripMargin)  *//创建另外一张表b ,并插入数据* spark.sql(  **"""  |create table hadoop\_prod.default.b (id int,name string,age int,tp string) using iceberg  """**.stripMargin) spark.sql(  **"""  |insert into hadoop\_prod.default.b values (1,"zs",30,"delete"),(2,"李四",31,"update"),(4,"王五",32,"add")  """**.stripMargin) |

1. **使用MERGE INTO 语法向目标表更新、删除、新增数据**

这里我们计划将b表与a表匹配id，如果b表中tp字段是"delete"那么a表中对应的id数据删除，如果b表中tp字段是"update"，那么a表中对应的id数据其他字段进行更新，如果a表与b表id匹配不上，那么将b表中的数据插入到a表中，具体操作如下：

|  |
| --- |
| *//将表b 中与表a中相同id的数据更新到表a,表a中没有表b中有的id对应数据写入增加到表a* spark.sql(  **"""  |merge into hadoop\_prod.default.a t1  |using (select id,name ,age,tp from hadoop\_prod.default.b) t2  |on t1.id = t2.id  |when matched and t2.tp = 'delete' then delete  |when matched and t2.tp = 'update' then update set t1.name = t2.name,t1.age = t2.age  |when not matched then insert (id,name,age) values (t2.id,t2.name,t2.age)  """**.stripMargin)  spark.sql(**"""select \* from hadoop\_prod.default.a """**).show()  ***最终结果如下：*** |

**注意：更新数据时，在查询的数据中只能有一条匹配的数据更新到目标表，否则将报错。**

#### INSERT OVERWRITE

"insert overwrite"可以覆盖Iceberg表中的数据，这种操作会将表中全部数据替换掉，建议如果有部分数据替换操作可以使用"merge into"操作。

对于Iceberg分区表使用"insert overwrite"操作时，有两种情况，第一种是“动态覆盖”，第二种是“静态覆盖”。

* **动态分区覆盖：**

动态覆盖会全量将原有数据覆盖，并将新插入的数据根据Iceberg表分区规则自动分区，类似Hive中的动态分区。

* **静态分区覆盖：**

静态覆盖需要在向Iceberg中插入数据时需要手动指定分区，如果当前Iceberg表存在这个分区，那么只有这个分区的数据会被覆盖，其他分区数据不受影响，如果Iceberg表不存在这个分区，那么相当于给Iceberg表增加了个一个分区。具体操作如下：

1. **创建三张表**

创建test1分区表、test2普通表、test3普通表三张表，并插入数据，每张表字段相同，但是插入数据不同。

|  |
| --- |
| *//创建 test1 分区表，并插入数据* spark.sql(  **"""  |create table hadoop\_prod.default.test1 (id int,name string,loc string)  |using iceberg  |partitioned by (loc)  """**.stripMargin)  spark.sql(  **"""  |insert into hadoop\_prod.default.test1 values (1,"zs","beijing"),(2,"ls","shanghai")  """**.stripMargin)   *//创建 test2 普通表，并插入数据* spark.sql(  **"""  |create table hadoop\_prod.default.test2 (id int,name string,loc string)  |using iceberg  """**.stripMargin)  spark.sql(  **"""  |insert into hadoop\_prod.default.test2 values (10,"x1","shandong"),(11,"x2","hunan")  """**.stripMargin)   *//创建 test3 普通表，并插入数据* spark.sql(  **"""  |create table hadoop\_prod.default.test3 (id int,name string,loc string)  |using iceberg  """**.stripMargin)  spark.sql(  **"""  |insert into hadoop\_prod.default.test3 values (3,"ww","beijing"),(4,"ml","shanghai"),(5,"tq","guangzhou")  """**.stripMargin) |

1. **使用insert overwrite 读取test3表中的数据覆盖到test2表中**

|  |
| --- |
| *//使用insert overwrite 读取test3 表中的数据覆盖到test2 普通表中* spark.sql(  **"""  |insert overwrite hadoop\_prod.default.test2  |select id,name,loc from hadoop\_prod.default.test3  """**.stripMargin)  *//查询 test2 表中的数据* spark.sql(  **"""  |select \* from hadoop\_prod.default.test2  """**.stripMargin).show()  ***Iceberg 表 test2结果如下：*** |

1. **使用insert overwrite 读取test3表数据，动态分区方式覆盖到表test1**

|  |
| --- |
| *// 使用insert overwrite 读取test3表数据 动态分区方式覆盖到表 test1* spark.sql(  **"""  |insert overwrite hadoop\_prod.default.test1  |select id,name,loc from hadoop\_prod.default.test3  """**.stripMargin)  *//查询 test1 表数据* spark.sql(  **"""  |select \* from hadoop\_prod.default.test1  """**.stripMargin).show()  ***Iceberg 表 test1结果如下：*** |

1. **静态分区方式，将iceberg表test3的数据覆盖到Iceberg表test1中**

这里可以将test1表删除，然后重新创建，加载数据，也可以直接读取test3中的数据静态分区方式更新到test1。另外，使用insert overwrite 语法覆盖静态分区方式时，查询的语句中就不要再次写入分区列，否则会重复。

|  |
| --- |
| *//删除表test1,重新创建表test1 分区表，并插入数据* spark.sql(  **"""  |drop table hadoop\_prod.default.test1  """**.stripMargin)  spark.sql(  **"""  |create table hadoop\_prod.default.test1 (id int,name string,loc string)  |using iceberg  |partitioned by (loc)  """**.stripMargin)  spark.sql(  **"""  |insert into hadoop\_prod.default.test1 values (1,"zs","beijing"),(2,"ls","shanghai")  """**.stripMargin)  spark.sql(**"select \* from hadoop\_prod.default.test1"**).show()  ***Iceberg 表 test1结果如下：***    *//注意：指定静态分区"jiangsu",静态分区下，就不要在查询 “loc" 列了，否则重复* spark.sql(  **"""  |insert overwrite hadoop\_prod.default.test1  |partition (loc = "jiangsu")  |select id,name from hadoop\_prod.default.test3  """**.stripMargin)  *//查询 test1 表数据* spark.sql(  **"""  |select \* from hadoop\_prod.default.test1  """**.stripMargin).show()  ***Iceberg 表 test1结果如下：*** |

**注意:使用insert overwrite 读取test3表数据 静态分区方式覆盖到表 test1，表中其他分区数据不受影响，只会覆盖指定的静态分区数据。**

#### DELETE FROM

Spark3.x版本之后支持"Delete from"可以根据指定的where条件来删除表中数据。如果where条件匹配Iceberg表一个分区的数据，Iceberg仅会修改元数据，如果where条件匹配的表的单个行，则Iceberg会重写受影响行所在的数据文件。具体操作如下：

|  |
| --- |
| *//创建表 delete\_tbl ,并加载数据* spark.sql(  **"""  |create table hadoop\_prod.default.delete\_tbl (id int,name string,age int) using iceberg  |"""**.stripMargin)  spark.sql(  **"""  |insert into hadoop\_prod.default.delete\_tbl values (1,"zs",18),(2,"ls",19),(3,"ww",20),(4,"ml",21),(5,"tq",22),(6,"gb",23)  """**.stripMargin)  *//根据条件范围删除表 delete\_tbl 中的数据* spark.sql(  **"""  |delete from hadoop\_prod.default.delete\_tbl where id >3 and id <6  """**.stripMargin)  spark.sql(**"select \* from hadoop\_prod.default.delete\_tbl"**).show()  ***Iceberg 表 delete\_tbl结果如下：***    *//根据条件删除表 delete\_tbl 中的一条数据* spark.sql(  **"""  |delete from hadoop\_prod.default.delete\_tbl where id = 2  """**.stripMargin)  spark.sql(**"select \* from hadoop\_prod.default.delete\_tbl"**).show()  ***Iceberg 表 delete\_tbl结果如下：*** |

#### UPDATE

Spark3.x+版本支持了update更新数据操作，可以根据匹配的条件进行数据更新操作。操作如下：

|  |
| --- |
| *//创建表 delete\_tbl ,并加载数据* spark.sql(  **"""  |create table hadoop\_prod.default.update\_tbl (id int,name string,age int) using iceberg  |"""**.stripMargin) spark.sql(  **"""  |insert into hadoop\_prod.default.update\_tbl values (1,"zs",18),(2,"ls",19),(3,"ww",20),(4,"ml",21),(5,"tq",22),(6,"gb",23)  """**.stripMargin) |

通过“update”更新表中id小于等于3的数据name列改为“zhangsan”,age列改为30，操作如下：

|  |
| --- |
| *//更新 delete\_tbl 表* spark.sql(  **"""  |update hadoop\_prod.default.update\_tbl set name = 'zhangsan' ,age = 30  |where id <=3  """**.stripMargin) spark.sql(  **"""  |select \* from hadoop\_prod.default.update\_tbl  """**.stripMargin).show()  ***Iceberg 表 update\_tbl结果如下：*** |

#### DataFrame API 写入Iceberg表

Spark向Iceberg中写数据时不仅可以使用SQL方式，也可以使用DataFrame Api方式操作Iceberg,建议使用SQL方式操作。

DataFrame创建Iceberg表分为创建普通表和分区表，创建分区表时需要指定分区列，分区列可以是多个列。创建表的语法如下:

|  |
| --- |
| df.write(tbl).create() 相当于 CREATE TABLE AS SELECT ...  df.write(tbl).replace() 相当于 REPLACE TABLE AS SELECT ...  df.write(tbl).append() 相当于 INSERT INTO ...  df.write(tbl).overwritePartitions() 相当于动态 INSERT OVERWRITE ... |

具体操作如下：

|  |
| --- |
| *//1.准备数据，使用DataFrame Api 写入Iceberg表及分区表* **val** nameJsonList = *List*[String](  **"{\"id\":1,\"name\":\"zs\",\"age\":18,\"loc\":\"beijing\"}"**,  **"{\"id\":2,\"name\":\"ls\",\"age\":19,\"loc\":\"shanghai\"}"**,  **"{\"id\":3,\"name\":\"ww\",\"age\":20,\"loc\":\"beijing\"}"**,  **"{\"id\":4,\"name\":\"ml\",\"age\":21,\"loc\":\"shanghai\"}"**)  **import** spark.implicits.\_ **val** df: DataFrame = spark.read.json(nameJsonList.toDS)  *//创建普通表df\_tbl1,并将数据写入到Iceberg表，其中DF中的列就是Iceberg表中的列* df.writeTo(**"hadoop\_prod.default.df\_tbl1"**).create()  *//查询表 hadoop\_prod.default.df\_tbl1 中的数据，并查看数据存储结构* spark.read.table(**"hadoop\_prod.default.df\_tbl1"**).show()  ***Iceberg 表 df\_tbl1结果如下：***    ***Iceberg 表 df\_tbl1存储如下：***  *//创建分区表df\_tbl2,并将数据写入到Iceberg表，其中DF中的列就是Iceberg表中的列* df.sortWithinPartitions(**$"loc"**)*//写入分区表，必须按照分区列进行排序* .writeTo(**"hadoop\_prod.default.df\_tbl2"**)  .partitionedBy(**$"loc"**)*//这里可以指定多个列为联合分区* .create() *//查询分区表 hadoop\_prod.default.df\_tbl2 中的数据，并查看数据存储结构* spark.read.table(**"hadoop\_prod.default.df\_tbl2"**).show()  ***Iceberg 分区表 df\_tbl2结果如下：***    ***Iceberg 分区表 df\_tbl2存储如下：*** |

### Structured Streaming实时写入Iceberg

目前Spark中Structured Streaming只支持实时向Iceberg中写入数据，不支持实时从Iceberg中读取数据，下面案例我们将使用Structured Streaming从Kafka中实时读取数据，然后将结果实时写入到Iceberg中。

1. **创建Kafka topic**

启动Kafka集群，创建“kafka-iceberg-topic”

|  |
| --- |
| [root@node1 bin]# ./kafka-topics.sh --zookeeper node3:2181,node4:2181,node5:2181 --create --topic kafka-iceberg-topic --partitions 3 --replication-factor 3 |

1. **编写向Kafka生产数据代码**

|  |
| --- |
| */\*\*  \* 向Kafka中写入数据  \*/* **object** WriteDataToKafka {  **def** main(args: Array[String]): Unit = {  **val** props = **new** Properties()  props.put(**"bootstrap.servers"**, **"node1:9092,node2:9092,node3:9092"**)  props.put(**"key.serializer"**, **"org.apache.kafka.common.serialization.StringSerializer"**)  props.put(**"value.serializer"**, **"org.apache.kafka.common.serialization.StringSerializer"**)   **val** producer = **new** KafkaProducer[String,String](props)  **var** counter = 0  **var** keyFlag = 0  **while**(**true**){  counter +=1  keyFlag +=1  **val** content: String = *userlogs*()  producer.send(**new** ProducerRecord[String, String](**"kafka-iceberg-topic"**, content))  *//producer.send(new ProducerRecord[String, String]("kafka-iceberg-topic", s"key-$keyFlag", content))* **if**(0 == counter%100){  counter = 0  Thread.*sleep*(5000)  }  }  producer.close()  }   **def** userlogs()={  **val** userLogBuffer = **new** StringBuffer(**""**)  **val** timestamp = **new** Date().getTime();  **var** userID = 0L  **var** pageID = 0L   *//随机生成的用户ID* userID = Random.nextInt(2000)   *//随机生成的页面ID* pageID = Random.nextInt(2000);   *//随机生成Channel* **val** channelNames = *Array*[String](**"Spark"**,**"Scala"**,**"Kafka"**,**"Flink"**,**"Hadoop"**,**"Storm"**,**"Hive"**,**"Impala"**,**"HBase"**,**"ML"**)  **val** channel = channelNames(Random.nextInt(10))   **val** actionNames = *Array*[String](**"View"**, **"Register"**)  *//随机生成action行为* **val** action = actionNames(Random.nextInt(2))   **val** dateToday = **new** SimpleDateFormat(**"yyyy-MM-dd"**).format(**new** Date())  userLogBuffer.append(dateToday)  .append(**"\t"**)  .append(timestamp)  .append(**"\t"**)  .append(userID)  .append(**"\t"**)  .append(pageID)  .append(**"\t"**)  .append(channel)  .append(**"\t"**)  .append(action)  System.*out*.println(userLogBuffer.toString())  userLogBuffer.toString()  } } |

1. **编写Structured Streaming读取Kafka数据实时写入Iceberg**

|  |
| --- |
| **object** StructuredStreamingSinkIceberg {  **def** main(args: Array[String]): Unit = {  *//1.准备对象* **val** spark: SparkSession = SparkSession.*builder*().master(**"local"**).appName(**"StructuredSinkIceberg"**)  *//指定hadoop catalog，catalog名称为hadoop\_prod* .config(**"spark.sql.catalog.hadoop\_prod"**, **"org.apache.iceberg.spark.SparkCatalog"**)  .config(**"spark.sql.catalog.hadoop\_prod.type"**, **"hadoop"**)  .config(**"spark.sql.catalog.hadoop\_prod.warehouse"**, **"hdfs://mycluster/structuredstreaming"**)  .getOrCreate() *// spark.sparkContext.setLogLevel("Error")   //2.创建Iceberg 表* spark.sql(  **"""  |create table if not exists hadoop\_prod.iceberg\_db.iceberg\_table (  | current\_day string,  | user\_id string,  | page\_id string,  | channel string,  | action string  |) using iceberg  """**.stripMargin)   **val** checkpointPath = **"hdfs://mycluster/iceberg\_table\_checkpoint"  val** bootstrapServers = **"node1:9092,node2:9092,node3:9092"** *//多个topic 逗号分开* **val** topic = **"kafka-iceberg-topic"** *//3.读取Kafka读取数据* **val** df = spark.readStream  .format(**"kafka"**)  .option(**"kafka.bootstrap.servers"**, bootstrapServers)  .option(**"auto.offset.reset"**, **"latest"**)  .option(**"group.id"**, **"iceberg-kafka"**)  .option(**"subscribe"**, topic)  .load()   **import** spark.implicits.\_  **import** org.apache.spark.sql.functions.\_   **val** resDF = df.selectExpr(**"CAST(key AS STRING)"**, **"CAST(value AS STRING)"**)  .as[(String, String)].toDF(**"id"**, **"data"**)   **val** transDF: DataFrame = resDF.withColumn(**"current\_day"**, *split*(*col*(**"data"**), **"\t"**)(0))  .withColumn(**"ts"**, *split*(*col*(**"data"**), **"\t"**)(1))  .withColumn(**"user\_id"**, *split*(*col*(**"data"**), **"\t"**)(2))  .withColumn(**"page\_id"**, *split*(*col*(**"data"**), **"\t"**)(3))  .withColumn(**"channel"**, *split*(*col*(**"data"**), **"\t"**)(4))  .withColumn(**"action"**, *split*(*col*(**"data"**), **"\t"**)(5))  .select(**"current\_day"**, **"user\_id"**, **"page\_id"**, **"channel"**, **"action"**)   *//结果打印到控制台,Default trigger (runs micro-batch as soon as it can) // val query: StreamingQuery = transDF.writeStream // .outputMode("append") // .format("console") // .start()   //4.流式写入Iceberg表* **val** query = transDF.writeStream  .format(**"iceberg"**)  .outputMode(**"append"**)  *//每分钟触发一次Trigger.ProcessingTime(1, TimeUnit.MINUTES)  //每10s 触发一次 Trigger.ProcessingTime(1, TimeUnit.MINUTES)* .trigger(Trigger.*ProcessingTime*(10, TimeUnit.*SECONDS*))  .option(**"path"**, **"hadoop\_prod.iceberg\_db.iceberg\_table"**)  .option(**"fanout-enabled"**, **"true"**)  .option(**"checkpointLocation"**, checkpointPath)  .start()   query.awaitTermination()   } } |

注意：以上代码执行时由于使用的Spark版本为3.1.2,其依赖的Hadoop版本为Hadoop3.2版本，所以需要在本地Window中配置Hadoop3.1.2的环境变量以及将对应的hadoop.dll放入window "C:\Windows\System32"路径下。

Structuerd Streaming向Iceberg实时写入数据有以下几个注意点：

* **写Iceberg表写出数据支持两种模式：append和complete，append是将每个微批数据行追加到表中。complete是替换每个微批数据内容。**
* **向Iceberg中写出数据时指定的path可以是HDFS路径，可以是Iceberg表名，如果是表名，要预先创建好Iceberg表。**
* **写出参数fanout-enabled指的是如果Iceberg写出的表是分区表，在向表中写数据之前要求Spark每个分区的数据必须排序，但这样会带来数据延迟，为了避免这个延迟，可以设置“fanout-enabled”参数为true,可以针对每个Spark分区打开一个文件，直到当前task批次数据写完，这个文件再关闭。**
* **实时向Iceberg表中写数据时，建议trigger设置至少为1分钟提交一次，因为每次提交都会产生一个新的数据文件和元数据文件，这样可以减少一些小文件。为了进一步减少数据文件，建议定期合并“data files”（参照1.9.6.9）和删除旧的快照（1.9.6.10）。**

1. **查看Iceberg中数据结果**

启动向Kafka生产数据代码，启动向Iceberg中写入数据的Structured Streaming程序，执行以下代码来查看对应的Iceberg结果：

|  |
| --- |
| *//1.准备对象* **val** spark: SparkSession = SparkSession.*builder*().master(**"local"**).appName(**"StructuredSinkIceberg"**)  *//指定hadoop catalog，catalog名称为hadoop\_prod* .config(**"spark.sql.catalog.hadoop\_prod"**, **"org.apache.iceberg.spark.SparkCatalog"**)  .config(**"spark.sql.catalog.hadoop\_prod.type"**, **"hadoop"**)  .config(**"spark.sql.catalog.hadoop\_prod.warehouse"**, **"hdfs://mycluster/structuredstreaming"**)  .getOrCreate()  *//2.读取Iceberg 表中的数据结果* spark.sql(  **"""  |select \* from hadoop\_prod.iceberg\_db.iceberg\_table  """**.stripMargin).show() |

## Flink1.13.5与Iceberg0.12.1整合

目前Flink支持使用DataStream API 和SQL API 方式实时读取和写入Iceberg表，建议大家使用SQL API 方式实时读取和写入Iceberg表。

Iceberg 支持的Flink版本为1.11.x版本以上，目前经过测试Iceberg版本与Flink的版本对应关系如下：

* Flink1.11.x版本与Iceberg0.11.1版本匹配。
* Flink1.12.x~Flink1.1.x 版本与Iceberg0.12.1版本匹配，SQL API有一些bug。
* Flink1.14.x版本与Iceberg0.12.1版本能整合但是有一些小bug，例如实时读取Iceberg中的数据有bug。

以下Flink与Iceberg整合使用的Flink版本为1.13.5，Iceberg版本为0.12.1版本。后期使用SQL API 操作时使用的Flink版本为1.11.6,Iceberg版本为0.11.1版本。

### DataStream API 操作Iceberg

#### DataStream API 实时写入Iceberg表

DataStream Api方式操作Iceberg方式目前仅支持Java Api。使用DataStream API 实时写入Iceberg表具体操作如下：

1. **首先在Maven中导入以下依赖：**

|  |
| --- |
| <**properties**>  <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  <**maven.compiler.source**>1.8</**maven.compiler.source**>  <**maven.compiler.target**>1.8</**maven.compiler.target**>  *<!-- flink 1.12.x -1.13.x 版本与Iceberg 0.12.1 版本兼容 ，不能与Flink 1.14 兼容-->* <**flink.version**>1.13.5</**flink.version**>  *<!--<flink.version>1.12.1</flink.version>-->  <!--<flink.version>1.14.2</flink.version>-->  <!-- flink 1.11.x 与Iceberg 0.11.1 合适-->  <!--<flink.version>1.11.6</flink.version>-->* <**hadoop.version**>3.2.2</**hadoop.version**> </**properties**>  <**dependencies**>  <**dependency**>  <**groupId**>com.alibaba.ververica</**groupId**>  <**artifactId**>ververica-connector-iceberg</**artifactId**>  <**version**>1.13-vvr-4.0.7</**version**>  </**dependency**>  *<!-- Flink 操作Iceberg 需要的Iceberg依赖 -->* <**dependency**>  <**groupId**>org.apache.iceberg</**groupId**>  <**artifactId**>iceberg-flink-runtime</**artifactId**>  <**version**>0.12.1</**version**>  *<!--<version>0.11.1</version>-->* </**dependency**>   *<!-- java 开发Flink 所需依赖 -->* <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-java</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-streaming-java\_2.11</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-clients\_2.11</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-streaming-scala\_2.11</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>   *<!-- Flink Kafka连接器的依赖 -->* <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-connector-kafka\_2.11</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-connector-base</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>   *<!-- 读取hdfs文件需要jar包-->* <**dependency**>  <**groupId**>org.apache.hadoop</**groupId**>  <**artifactId**>hadoop-client</**artifactId**>  <**version**>${hadoop.version}</**version**>  </**dependency**>   *<!-- Flink SQL & Table-->* <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table-runtime-blink\_2.11</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table-common</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table-api-java</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table-api-java-bridge\_2.11</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table-planner\_2.11</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table-planner-blink\_2.11</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>junit</**groupId**>  <**artifactId**>junit</**artifactId**>  <**version**>4.11</**version**>  <**scope**>test</**scope**>  </**dependency**>   *<!-- log4j 和slf4j 包,如果在控制台不想看到日志，可以将下面的包注释掉-->* <**dependency**>  <**groupId**>org.slf4j</**groupId**>  <**artifactId**>slf4j-log4j12</**artifactId**>  <**version**>1.7.25</**version**>  <**scope**>test</**scope**>  </**dependency**>  <**dependency**>  <**groupId**>log4j</**groupId**>  <**artifactId**>log4j</**artifactId**>  <**version**>1.2.17</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.slf4j</**groupId**>  <**artifactId**>slf4j-api</**artifactId**>  <**version**>1.7.25</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.slf4j</**groupId**>  <**artifactId**>slf4j-nop</**artifactId**>  <**version**>1.7.25</**version**>  <**scope**>test</**scope**>  </**dependency**>  <**dependency**>  <**groupId**>org.slf4j</**groupId**>  <**artifactId**>slf4j-simple</**artifactId**>  <**version**>1.7.5</**version**>  </**dependency**> </**dependencies**> |

1. **编写代码使用DataStream API将Kafka数据写入到Iceberg表**

|  |
| --- |
| **import** com.google.common.collect.ImmutableMap; **import** org.apache.flink.api.common.eventtime.WatermarkStrategy; **import** org.apache.flink.api.common.functions.MapFunction; **import** org.apache.flink.api.common.serialization.SimpleStringSchema; **import** org.apache.flink.connector.kafka.source.KafkaSource; **import** org.apache.flink.connector.kafka.source.enumerator.initializer.OffsetsInitializer; **import** org.apache.flink.streaming.api.datastream.DataStreamSource; **import** org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.table.data.GenericRowData; **import** org.apache.flink.table.data.RowData; **import** org.apache.hadoop.conf.Configuration; **import** org.apache.iceberg.\*; **import** org.apache.iceberg.catalog.Catalog; **import** org.apache.iceberg.catalog.TableIdentifier; **import** org.apache.iceberg.flink.TableLoader; **import** org.apache.flink.table.data.StringData; **import** org.apache.iceberg.flink.sink.FlinkSink; **import** org.apache.iceberg.hadoop.HadoopCatalog; **import** org.apache.iceberg.types.Types; **import** java.util.Map;  */\*\*  \* 使用DataStream Api 向Iceberg 表写入数据  \*/* **public class** StreamAPIWriteIceberg {  **public static void** main(String[] args) **throws** Exception {  StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  *//1.必须设置checkpoint ,Flink向Iceberg中写入数据时当checkpoint发生后，才会commit数据。* env.enableCheckpointing(5000);   *//2.读取Kafka 中的topic 数据* KafkaSource<String> source = KafkaSource.<String>*builder*()  .setBootstrapServers(**"node1:9092,node2:9092,node3:9092"**)  .setTopics(**"flink-iceberg-topic"**)  .setGroupId(**"my-group-id"**)  .setStartingOffsets(OffsetsInitializer.*latest*())  .setValueOnlyDeserializer(**new** SimpleStringSchema())  .build();  DataStreamSource<String> kafkaSource = env.fromSource(source, WatermarkStrategy.*noWatermarks*(), **"Kafka Source"**);   *//3.对数据进行处理，包装成RowData 对象，方便保存到Iceberg表中。* SingleOutputStreamOperator<RowData> dataStream = kafkaSource.map(**new** MapFunction<String, RowData>() {  @Override  **public** RowData map(String s) **throws** Exception {  System.***out***.println(**"s = "**+s);  String[] split = s.split(**","**);  GenericRowData row = **new** GenericRowData(4);  row.setField(0, Integer.*valueOf*(split[0]));  row.setField(1, StringData.*fromString*(split[1]));  row.setField(2, Integer.*valueOf*(split[2]));  row.setField(3, StringData.*fromString*(split[3]));  **return** row;  }  });   *//4.创建Hadoop配置、Catalog配置和表的Schema，方便后续向路径写数据时可以找到对应的表* Configuration hadoopConf = **new** Configuration();  Catalog catalog = **new** HadoopCatalog(hadoopConf,**"hdfs://mycluster/flink\_iceberg/"**);   *//配置iceberg 库名和表名* TableIdentifier name =  TableIdentifier.*of*(**"icebergdb"**, **"flink\_iceberg\_tbl"**);   *//创建Icebeng表Schema* Schema schema = **new** Schema(  Types.NestedField.*required*(1, **"id"**, Types.IntegerType.*get*()),  Types.NestedField.*required*(2, **"nane"**, Types.StringType.*get*()),  Types.NestedField.*required*(3, **"age"**, Types.IntegerType.*get*()),  Types.NestedField.*required*(4, **"loc"**, Types.StringType.*get*()));   *//如果有分区指定对应分区，这里“loc”列为分区列，可以指定unpartitioned 方法不设置表分区 // PartitionSpec spec = PartitionSpec.unpartitioned();* PartitionSpec spec = PartitionSpec.*builderFor*(schema).identity(**"loc"**).build();   *//指定Iceberg表数据格式化为Parquet存储* Map<String, String> props =  ImmutableMap.*of*(TableProperties.***DEFAULT\_FILE\_FORMAT***, FileFormat.***PARQUET***.name());  Table table = **null**;   *// 通过catalog判断表是否存在，不存在就创建，存在就加载* **if** (!catalog.tableExists(name)) {  table = catalog.createTable(name, schema, spec, props);  }**else** {  table = catalog.loadTable(name);  }   TableLoader tableLoader = TableLoader.*fromHadoopTable*(**"hdfs://mycluster/flink\_iceberg/icebergdb/flink\_iceberg\_tbl"**, hadoopConf);   *//5.通过DataStream Api 向Iceberg中写入数据* FlinkSink.*forRowData*(dataStream)  *//这个 .table 也可以不写，指定tableLoader 对应的路径就可以。* .table(table)  .tableLoader(tableLoader)  *//默认为false,追加数据。如果设置为true 就是覆盖数据*  .overwrite(**false**)  .build();   env.execute(**"DataStream Api Write Data To Iceberg"**);  } } |

以上代码有如下几个注意点：

* **需要设置Checkpoint，Flink向Iceberg中写入Commit数据时，只有Checkpoint成功之后才会Commit数据，否则后期在Hive中查询不到数据。**
* **读取Kafka数据后需要包装成RowData或者Row对象，才能向Iceberg表中写出数据。写出数据时默认是追加数据，如果指定overwrite就是全部覆盖数据。**
* **在向Iceberg表中写数据之前需要创建对应的Catalog、表Schema,否则写出时只指定对应的路径会报错找不到对应的Iceberg表。**
* **不建议使用DataStream API 向Iceberg中写数据，建议使用SQL API。**

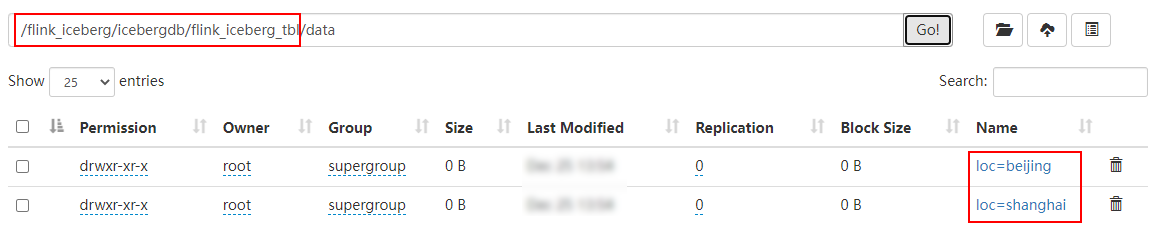
1. **在Kafka 中创建代码中指定的“flink-iceberg-topic”并启动代码生产数据**

|  |
| --- |
| ***# 在Kafka 中创建 flink-iceberg-topic topic***  [root@node1 bin]# ./kafka-topics.sh --zookeeper node3:2181,node4:2181,node5:2181 --create --topic flink-iceberg-topic --partitions 3 --replication-factor 3 |

创建好以上topic之后，启动代码，然后向topic中生产以下数据：

|  |
| --- |
| [root@node1 bin]#./kafka-console-producer.sh --topic flink-iceberg-topic --broker-list node1:9092,node2:9092,node3:9092  *1,zs,18,beijing*  *2,ls,19,shanghai*  *3,ww,20,beijing*  *4,ml,21,shanghai* |

可以看到在HDFS 对应的路径中保存了对应的数据：



1. **通过Hive查看保存到Iceberg中的数据**

启动Hive、Hive Metastore 在Hive中创建映射Iceberg的外表：

|  |
| --- |
| CREATE TABLE **flink\_iceberg\_tbl** (  id int,  name string,  age int,  loc string  )  STORED BY '**org.apache.iceberg.mr.hive.HiveIcebergStorageHandler**'  LOCATION '**hdfs://mycluster/flink\_iceberg/icebergdb/flink\_iceberg\_tbl**'  TBLPROPERTIES (**'iceberg.catalog'='location\_based\_table'**);  ***注意：虽然loc是分区列，创建时忽略分区列就可以，此外映射表的路径要保持与保存Iceberg数据路径一致。*** |

通过Hive查询对应的Iceberg表中的数据，结果如下：

|  |
| --- |
| hive> ***select \* from flink\_iceberg\_tbl;***  OK  *2 ls 19 shanghai*  *3 ww 20 beijing*  *1 zs 18 beijing*  *4 ml 21 shanghai* |

#### DataStream API 批量/实时读取Iceberg表

DataStream API 读取Iceberg表又分为批量读取和实时读取。通过方法“streaming(true/false)”来控制。

* **批量/全量读取：**

|  |
| --- |
| **import** org.apache.flink.api.common.functions.MapFunction; **import** org.apache.flink.streaming.api.datastream.DataStream; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.table.data.RowData; **import** org.apache.hadoop.conf.Configuration; **import** org.apache.iceberg.flink.TableLoader; **import** org.apache.iceberg.flink.source.FlinkSource;  */\*\*  \* 使用DataStream Api 批量/实时 读取Iceberg 数据  \*/* **public class** StreamAPIReadIceberg {  **public static void** main(String[] args) **throws** Exception {  StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();   *//1.配置TableLoader* Configuration hadoopConf = **new** Configuration();  TableLoader tableLoader = TableLoader.*fromHadoopTable*(**"hdfs://mycluster/flink\_iceberg/icebergdb/flink\_iceberg\_tbl"**, hadoopConf);   *//2.从Iceberg中读取全量/增量读取数据* DataStream<RowData> batchData = FlinkSource.*forRowData*().env(env)  .tableLoader(tableLoader)  *//默认为false,整批次读取，设置为true 为流式读取* .streaming(**false**)  .build();   batchData.map(**new** MapFunction<RowData, String>() {  @Override  **public** String map(RowData rowData) **throws** Exception {  **int** id = rowData.getInt(0);  String name = rowData.getString(1).toString();  **int** age = rowData.getInt(2);  String loc = rowData.getString(3).toString();  **return** id+**","**+name+**","**+age+**","**+loc;  }  }).print();   env.execute(**"DataStream Api Read Data From Iceberg"**);   } }  ***结果如下：*** |

* **实时读取**

|  |
| --- |
| *//当配置 streaming参数为true时就是实时读取*  DataStream<RowData> batchData = FlinkSource.*forRowData*().env(env)  .tableLoader(tableLoader)  *//默认为false,整批次读取，设置为true 为流式读取* **.streaming(true)**  .build(); |

修改以上代码并启动，向Hive 对应的Iceberg表“flink\_iceberg\_tbl”中插入2条数据：

|  |
| --- |
| ***#在向Hive的Iceberg表中插入数据之前需要加入以下两个包：***  add jar /software/hive-3.1.2/lib/iceberg-hive-runtime-0.12.1.jar;  add jar /software/hive-3.1.2/lib/libfb303-0.9.3.jar;  ***#向Hive 中Iceberg 表插入两条数据***  hive> insert into flink\_iceberg\_tbl values (5,'s1',30,'guangzhou'),(6,'s2',31,'tianjin');  ***#插入完成之后，可以看到Flink 控制台实时读取到对应数据*** |

#### 指定基于快照实时增量读取数据

以上案例我们发现Flink将表中所有数据都读取出来，我们也可以指定对应的snapshot-id 决定基于哪些数据增量读取数据。

|  |
| --- |
| DataStream<RowData> batchData = FlinkSource.*forRowData*().env(env)  .tableLoader(tableLoader)  *//基于某个快照实时增量读取数据，快照需要从元数据中获取* .startSnapshotId(4226332606322964975L)  *//默认为false,整批次读取，设置为true 为流式读取* .streaming(**true**)  .build();  ***结果只读取到指定快照往后的数据，如下：*** |

#### 合并data files

Iceberg提供Api将小文件合并成大文件，可以通过Flink 批任务来执行。Flink中合并小文件与Spark中小文件合并完全一样。

代码如下：

|  |
| --- |
| **import** org.apache.flink.api.java.ExecutionEnvironment; **import** org.apache.hadoop.conf.Configuration; **import** org.apache.iceberg.Table; **import** org.apache.iceberg.actions.RewriteDataFilesActionResult; **import** org.apache.iceberg.catalog.Catalog; **import** org.apache.iceberg.catalog.TableIdentifier; **import** org.apache.iceberg.flink.TableLoader; **import** org.apache.iceberg.flink.actions.Actions; **import** org.apache.iceberg.hadoop.HadoopCatalog;  */\*\*  \* 可以通过提交Flink批量任务来合并Data Files 文件。  \*/* **public class** RewrietDataFiles {  **public static void** main(String[] args) {   ExecutionEnvironment env = ExecutionEnvironment.*getExecutionEnvironment*();   *//1.配置TableLoader* Configuration hadoopConf = **new** Configuration();   *//2.创建Hadoop配置、Catalog配置和表的Schema，方便后续向路径写数据时可以找到对应的表* Catalog catalog = **new** HadoopCatalog(hadoopConf,**"hdfs://mycluster/flink\_iceberg/"**);   *//3.配置iceberg 库名和表名并加载表* TableIdentifier name =  TableIdentifier.*of*(**"icebergdb"**, **"flink\_iceberg\_tbl"**);  Table table = catalog.loadTable(name);   *//4..合并 data files 小文件* RewriteDataFilesActionResult result = Actions.*forTable*(table)  .rewriteDataFiles()  *//默认 512M ，可以手动通过以下指定合并文件大小，与Spark中一样。* .targetSizeInBytes(536870912L)  .execute();  } } |

### SQL API操作Iceberg

Flink SQL 在操作Iceberg时，对应的版本为Flink 1.11.x 与Iceberg0.11.1版本，目前，Flink1.14.2版本与Iceberg0.12.1版本对于SQL API 来说兼容有问题，所以这里使用Flink1.11.6版本与Iceberg0.11.1版本来演示Flink SQL API 操作Iceberg。

#### SQL API 创建Iceberg表并写入数据

1. **创建新项目，导入如下maven依赖包**

|  |
| --- |
| <**properties**>  <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  <**maven.compiler.source**>1.8</**maven.compiler.source**>  <**maven.compiler.target**>1.8</**maven.compiler.target**>  *<!-- flink 1.11.x 与Iceberg 0.11.1 合适-->* <**flink.version**>1.11.6</**flink.version**>  <**hadoop.version**>3.2.2</**hadoop.version**> </**properties**>  <**dependencies**>  *<!-- Flink 操作Iceberg 需要的Iceberg依赖 -->* <**dependency**>  <**groupId**>org.apache.iceberg</**groupId**>  <**artifactId**>iceberg-flink-runtime</**artifactId**>  <**version**>0.11.1</**version**>  </**dependency**>   *<!-- java 开发Flink 所需依赖 -->* <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-java</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-streaming-java\_2.11</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-clients\_2.11</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-streaming-scala\_2.11</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>   *<!-- Flink Kafka连接器的依赖 -->* <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-connector-kafka\_2.11</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-csv</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>   *<!-- 读取hdfs文件需要jar包-->* <**dependency**>  <**groupId**>org.apache.hadoop</**groupId**>  <**artifactId**>hadoop-client</**artifactId**>  <**version**>${hadoop.version}</**version**>  </**dependency**>   *<!-- Flink SQL & Table-->* <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table-runtime-blink\_2.11</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table-common</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table-api-java</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table-api-java-bridge\_2.11</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table-planner\_2.11</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table-planner-blink\_2.11</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>    <**dependency**>  <**groupId**>junit</**groupId**>  <**artifactId**>junit</**artifactId**>  <**version**>4.11</**version**>  <**scope**>test</**scope**>  </**dependency**>   *<!-- log4j 和slf4j 包,如果在控制台不想看到日志，可以将下面的包注释掉-->* <**dependency**>  <**groupId**>org.slf4j</**groupId**>  <**artifactId**>slf4j-log4j12</**artifactId**>  <**version**>1.7.25</**version**>  <**scope**>test</**scope**>  </**dependency**>  <**dependency**>  <**groupId**>log4j</**groupId**>  <**artifactId**>log4j</**artifactId**>  <**version**>1.2.17</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.slf4j</**groupId**>  <**artifactId**>slf4j-api</**artifactId**>  <**version**>1.7.25</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.slf4j</**groupId**>  <**artifactId**>slf4j-nop</**artifactId**>  <**version**>1.7.25</**version**>  <**scope**>test</**scope**>  </**dependency**>  <**dependency**>  <**groupId**>org.slf4j</**groupId**>  <**artifactId**>slf4j-simple</**artifactId**>  <**version**>1.7.5</**version**>  </**dependency**> </**dependencies**> |

1. **编写Flink SQL 创建Iceberg表并写入数据**

|  |
| --- |
| StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  StreamTableEnvironment tblEnv = StreamTableEnvironment.*create*(env);   env.enableCheckpointing(1000);  *//1.创建Catalog* tblEnv.executeSql(**"CREATE CATALOG hadoop\_iceberg WITH ("** +  **"'type'='iceberg',"** +  **"'catalog-type'='hadoop',"** +  **"'warehouse'='hdfs://mycluster/flink\_iceberg')"**);   *//2.使用当前Catalog* tblEnv.useCatalog(**"hadoop\_iceberg"**);   *//3.创建数据库* tblEnv.executeSql(**"create database iceberg\_db"**);   *//4.使用数据库* tblEnv.useDatabase(**"iceberg\_db"**);  *//5.创建iceberg表 flink\_iceberg\_tbl* tblEnv.executeSql(**"create table hadoop\_iceberg.iceberg\_db.flink\_iceberg\_tbl2(id int,name string,age int,loc string) partitioned by (loc)"**);   *//6.写入数据到表 flink\_iceberg\_tbl* tblEnv.executeSql(**"insert into hadoop\_iceberg.iceberg\_db.flink\_iceberg\_tbl2 values (1,'zs',18,'beijing'),(2,'ls',19,'shanghai'),(3,'ww',20,'guangzhou')"**); |

1. **在Hive中映射Iceberg表并查询**

在Hive中执行如下命令创建对应的Iceberg表：

|  |
| --- |
| ***#在Hive中创建Iceberg表***  CREATE TABLE flink\_iceberg\_tbl2 (  id int,  name string,  age int,  loc string  )  STORED BY 'org.apache.iceberg.mr.hive.HiveIcebergStorageHandler'  LOCATION 'hdfs://mycluster/flink\_iceberg/iceberg\_db/flink\_iceberg\_tbl2'  TBLPROPERTIES ('iceberg.catalog'='location\_based\_table');  ***#在Hive中查询Iceberg表中的数据***  hive> select \* from flink\_iceberg\_tbl2;  *OK*  *3 ww 20 guangzhou*  *1 zs 18 beijing*  *2 ls 19 shanghai* |

#### SQL API 批量查询Iceberg表数据

Flink SQL API 批量查询Iceberg表数据，直接查询显示即可。代码如下：

|  |
| --- |
| StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*(); StreamTableEnvironment tblEnv = StreamTableEnvironment.*create*(env);  env.enableCheckpointing(1000);  *//1.创建Catalog* tblEnv.executeSql(**"CREATE CATALOG hadoop\_iceberg WITH ("** +  **"'type'='iceberg',"** +  **"'catalog-type'='hadoop',"** +  **"'warehouse'='hdfs://mycluster/flink\_iceberg')"**); *//2.批量读取表数据* TableResult tableResult = tblEnv.executeSql(**"select \* from hadoop\_iceberg.iceberg\_db.flink\_iceberg\_tbl2 "**);  tableResult.print();  ***结果如下：*** |

#### SQL API 实时查询Iceberg表数据

Flink SQL API 实时查询Iceberg表数据时需要设置参数“table.dynamic-table-options.enabled”为true,以支持SQL语法中的“OPTIONS”选项，代码如下：

|  |
| --- |
| StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*(); StreamTableEnvironment tblEnv = StreamTableEnvironment.*create*(env);  env.enableCheckpointing(1000);  Configuration configuration = tblEnv.getConfig().getConfiguration(); *// 支持SQL语法中的 OPTIONS 选项* configuration.setBoolean(**"table.dynamic-table-options.enabled"**, **true**);  *//1.创建Catalog* tblEnv.executeSql(**"CREATE CATALOG hadoop\_iceberg WITH ("** +  **"'type'='iceberg',"** +  **"'catalog-type'='hadoop',"** +  **"'warehouse'='hdfs://mycluster/flink\_iceberg')"**);  *//2.从Iceberg表当前快照读取所有数据，并继续增量读取数据 // streaming指定为true支持实时读取数据，monitor\_interval 监控数据的间隔，默认1s* TableResult tableResult = tblEnv.executeSql(**"select \* from hadoop\_iceberg.iceberg\_db.flink\_iceberg\_tbl2 /\*+ OPTIONS('streaming'='true', 'monitor-interval'='1s')\*/"**);  tableResult.print(); |

启动以上代码后，可以看到会将目前存在于Iceberg表中的数据读取出来，向Hive中对应的Iceberg表中插入数据，可以看到控制台实时获取数据。

|  |
| --- |
| ***#在向Hive的Iceberg表中插入数据之前需要加入以下两个包：***  add jar /software/hive-3.1.2/lib/iceberg-hive-runtime-0.12.1.jar;  add jar /software/hive-3.1.2/lib/libfb303-0.9.3.jar;  ***#向Hive 中Iceberg 表插入两条数据***  hive> insert into flink\_iceberg\_tbl2 values (4,'ml',30,'shenzhen'),(5,'tq',31,'beijing');  ***#在控制台可以看到实时新增数据*** |

#### SQL API指定基于快照实时增量读取数据

Flink SQL API 还支持基于某个snapshot-id来继续实时获取数据，代码如下：

|  |
| --- |
| StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*(); StreamTableEnvironment tblEnv = StreamTableEnvironment.*create*(env); env.enableCheckpointing(1000);  Configuration configuration = tblEnv.getConfig().getConfiguration(); *// 支持SQL语法中的 OPTIONS 选项* configuration.setBoolean(**"table.dynamic-table-options.enabled"**, **true**);  *//1.创建Catalog* tblEnv.executeSql(**"CREATE CATALOG hadoop\_iceberg WITH ("** +  **"'type'='iceberg',"** +  **"'catalog-type'='hadoop',"** +  **"'warehouse'='hdfs://mycluster/flink\_iceberg')"**);  *//2.从Iceberg 指定的快照继续实时读取数据，快照ID从对应的元数据中获取 //start-snapshot-id :快照ID* TableResult tableResult2 = tblEnv.executeSql(**"SELECT \* FROM hadoop\_iceberg.iceberg\_db.flink\_iceberg\_tbl2 /\*+ OPTIONS('streaming'='true', 'monitor-interval'='1s', 'start-snapshot-id'='3821550127947089987')\*/"**); tableResult2.print(); |

### SQL API 读取Kafka数据实时写入Iceberg表

从Kafka中实时读取数据写入到Iceberg表中，操作步骤如下：

1. **首先需要创建对应的Iceberg表**

|  |
| --- |
| StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*(); StreamTableEnvironment tblEnv = StreamTableEnvironment.*create*(env); env.enableCheckpointing(1000);  *//1.创建Catalog* tblEnv.executeSql(**"CREATE CATALOG hadoop\_iceberg WITH ("** +  **"'type'='iceberg',"** +  **"'catalog-type'='hadoop',"** +  **"'warehouse'='hdfs://mycluster/flink\_iceberg')"**);  *//2.创建iceberg表 flink\_iceberg\_tbl* tblEnv.executeSql(**"create table hadoop\_iceberg.iceberg\_db.flink\_iceberg\_tbl3(id int,name string,age int,loc string) partitioned by (loc)"**); |

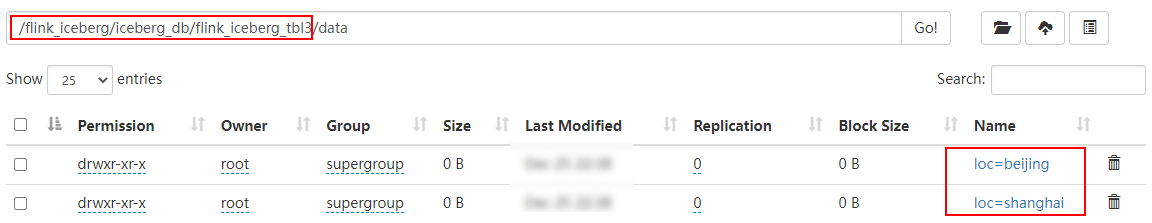
1. **编写代码读取Kafka数据实时写入Iceberg**

|  |
| --- |
| **public class** ReadKafkaToIceberg {  **public static void** main(String[] args) **throws** Exception {  StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  StreamTableEnvironment tblEnv = StreamTableEnvironment.*create*(env);  env.enableCheckpointing(1000);   */\*\*  \* 1.需要预先创建 Catalog 及Iceberg表  \*/  //1.创建Catalog* tblEnv.executeSql(**"CREATE CATALOG hadoop\_iceberg WITH ("** +  **"'type'='iceberg',"** +  **"'catalog-type'='hadoop',"** +  **"'warehouse'='hdfs://mycluster/flink\_iceberg')"**);   *//2.创建iceberg表 flink\_iceberg\_tbl // tblEnv.executeSql("create table hadoop\_iceberg.iceberg\_db.flink\_iceberg\_tbl3(id int,name string,age int,loc string) partitioned by (loc)");   //3.创建 Kafka Connector,连接消费Kafka中数据* tblEnv.executeSql(**"create table kafka\_input\_table("** +  **" id int,"** +  **" name varchar,"** +  **" age int,"** +  **" loc varchar"** +  **") with ("** +  **" 'connector' = 'kafka',"** +  **" 'topic' = 'flink-iceberg-topic',"** +  **" 'properties.bootstrap.servers'='node1:9092,node2:9092,node3:9092',"** +  **" 'scan.startup.mode'='latest-offset',"** +  **" 'properties.group.id' = 'my-group-id',"** +  **" 'format' = 'csv'"** +  **")"**);   *//4.配置 table.dynamic-table-options.enabled* Configuration configuration = tblEnv.getConfig().getConfiguration();  *// 支持SQL语法中的 OPTIONS 选项* configuration.setBoolean(**"table.dynamic-table-options.enabled"**, **true**);   *//5.写入数据到表 flink\_iceberg\_tbl3* tblEnv.executeSql(**"insert into hadoop\_iceberg.iceberg\_db.flink\_iceberg\_tbl3 select id,name,age,loc from kafka\_input\_table"**);   *//6.查询表数据* TableResult tableResult = tblEnv.executeSql(**"select \* from hadoop\_iceberg.iceberg\_db.flink\_iceberg\_tbl3 /\*+ OPTIONS('streaming'='true', 'monitor-interval'='1s')\*/"**);  tableResult.print();  } } |

启动以上代码，向Kafka topic中生产如下数据：

|  |
| --- |
| *1,zs,18,beijing*  *2,ls,19,shanghai*  *3,ww,20,beijing*  *4,ml,21,shanghai* |

我们可以看到控制台上有对应实时数据输出，查看对应的Icberg HDFS目录，数据写入成功。



### Flink兼容Iceberg目前不足

* Iceberg目前不支持Flink SQL 查询表的元数据信息，需要使用Java API 实现。
* Flink不支持创建带有隐藏分区的Iceberg表
* Flink不支持带有WaterMark的Iceberg表
* Flink不支持添加列、删除列、重命名列操作。
* Flink对Iceberg Connector支持并不完善。

## Iceberg与Hudi对比

Iceberg和Hudi都是数据湖技术，从社区活跃度上来看，Iceberg有超越Hudi的趋势。他们有以下共同点：

1. 都是构建于存储格式之上的数据组织方式
2. 提供ACID能力，提供一定的事务、并行执行能力
3. 提供行级别数据修改能力。
4. 提供一定的Schema扩展能力，例如：新增、修改、删除列操作。
5. 支持数据合并，处理小文件。
6. 支持Time travel 查询快照数据。
7. 支持批量和实时数据读写

Iceberg与Hudi之间不同点在于以下几点：

1. Iceberg支持Parquet、avro、orc数据格式，Hudi支持Parquet和Avro格式。
2. 两者数据存储和查询机制不同

Iceberg只支持一种表存储模式，就是有metadata file、manifest file和data file组成存储结构，查询时首先查找Metadata元数据进而过滤找到对应的 SnapShot对应的manifest files ,再找到对应的数据文件。Hudi支持两种表存储模式：Copy On Write（写时合并） 和Merge On Read（读时合并），查询时直接读取对应的快照数据。

1. 对于处理小文件合并时，Iceberg只支持API方式手动处理合并小文件，Hudi对于小文件合并处理可以根据配置自动的执行。
2. Spark与Iceberg和Hudi整合时，Iceberg对SparkSQL的支持目前来看更好。Spark与Hudi整合更多的是Spark DataFrame API 操作。
3. 关于Schema方面，Iceberg Schema与计算引擎是解耦的，不依赖任何的计算引擎，而Hudi的Schema依赖于计算引擎Schema。

|  |  |  |
| --- | --- | --- |
| **对比点** | **Hudi** | **Iceberg** |
| **数据组织方式** | 表格式 | 表格式 |
| **数据修改** | 支持新增、修改、删除 | 支持新增、修改、删除 |
| **小文件合并处理** | 支持，根据配置自动执行 | 支持，API手动处理合并 |
| **快照查询** | 支持 | 支持 |
| **批量实时读写** | 支持 | 支持 |
| **数据格式** | Parquet、Avro | Parquet、Avro、ORC |
| **存储查询机制** | COW表、MOR表 | 只有一种表格式存储模式，metadata file、manifest file和data file组成存储结构 |
| **Schema支持** | Hudi依赖计算引擎Schema | Iceberg Schema 与计算引擎解耦 |