AI DataLake

开发指南

文档版本	01
发布日期	2026-09-15

华为技术有限公司

商标声明


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

注意

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

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

华为技术有限公司

地址： 深圳市龙岗区坂田华为总部办公楼 邮编：518129

网址： https://www.huawei.com

客户服务邮箱： support@huawei.com

客户服务电话： 4008302118

安全声明

漏洞处理流程

华为公司对产品漏洞管理的规定以“漏洞处理流程”为准，该流程的详细内容请参见如下网址：
https://www.huawei.com/cn/psirt/vul-response-process
如企业客户须获取漏洞信息，请参见如下网址：
https://securitybulletin.huawei.com/enterprise/cn/security-advisory

目 录

1 Iceberg 开发指南

1.1 Iceberg 简介

1.1.1 Iceberg 功能介绍

Apache Iceberg是一种面向海量分析数据集的开放表格式，通过高性能表格式为计算引擎添加表功能，使用方式与SQL表完全一致。

Iceberg专为超大规模表而构建，已在实际业务环境中得到应用，单个表可容纳数十PB数据，且即使如此庞大的表也无需分布式SQL引擎即可读取。

Iceberg支持以下功能：

- 快速扫描规划：读取表或查找文件操作无需分布式SQL引擎即可完成。
- 高级过滤：通过分区和列级统计信息，利用表元数据对数据文件进行裁剪。
- Iceberg专为解决最终一致性云对象存储中的正确性问题而设计。兼容所有云存储，在HDFS中通过避免列表操作和重命名来减少NameNode拥塞。
- 可序列化隔离：表变更具有原子性，部分或未提交的变更对读取者不可见。
- 多个并发写入器使用乐观并发控制，即使发生写入冲突也会重试以确保兼容的更新操作成功完成。

主要特点

- **可扩展性**：Iceberg可轻松扩展到支持海量的数据表格和庞大的数据集合。
- **查询性能**：Iceberg的元数据管理和查询优化功能可以提高数据查询的性能。
- **数据版本管理**：Iceberg提供了可靠的数据版本管理功能，可以帮助用户对数据进行版本控制和回溯。

- **易于使用**：Iceberg提供了简单易用的API和命令行工具，使得用户可以轻松地创建、管理和查询数据表格。
- **灵活的分区策略**：Iceberg支持灵活的分区策略，可以根据不同的数据集合进行分区管理。
- **多版本数据支持**：Iceberg支持多版本数据，可以帮助用户对数据进行版本管理和回溯。

基本概念

- Table（表格）：Iceberg的最基本的概念是Table，它是一个数据表格的抽象表示。Table包含了表格的元数据信息、数据存储位置、分区策略等信息。
- Partition（分区）：Partition是将Table中的数据按照指定的规则划分为多个子集的过程。Partition可以基于数据的某些特征进行划分，例如按照时间、地理位置、产品类型等进行分区。
- Metadata（元数据）：Iceberg的元数据是指描述Table中的数据结构、分区策略、数据版本等信息的数据。元数据存储在持久化的存储介质中，例如HDFS、S3等。
- Snapshot（快照）：Snapshot是指Table在某个时间点上的数据视图，它包含了Table当前版本的数据和元数据信息。
- Manifest（清单）：Manifest是指Table中数据文件的清单列表，它包含了每个数据文件的元数据信息（例如文件路径、大小、分区信息等）。

文件组织方式

Iceberg将数据分为元数据管理层、数据存储层。

- 元数据管理层：
  - Metadata文件为JSON格式。存储当前版本的元数据信息，所有快照信息。
  - Manifest List文件，即snapshot文件或清单列表文件，为Avro格式。一次commit生成一个快照文件，每行存储一个manifest file的路径、其存储的数据文件的分区范围，增加、删除了几个数据文件等信息，在查询时提供过滤信息，加快速度。
  - Manifest File文件，为Avro格式。存储多个数据文件的信息列表，每行是一个数据文件的详细描述，包括状态、路径、分区信息、列级别的统计信息（最大最小值、空值数等）、文件大小以及文件里数据行数等。其中列级别的统计信息在扫描表数据时可过滤掉不必要的文件。
- 数据存储层：默认为Parquet文件格式。

1.1.2 Iceberg 表属性参数介绍

创建和修改Iceberg表时，可以通过“TBLPROPERTIES”配置相关参数的值。

Read 属性参数

表 1-1 Read 属性参数介绍

参数	默认值	参数说明
read.split.target-size	134217728（128 MB）	合并数据输入拆分时的目标大小。
read.split.metadata-target-size	33554432（32 MB）	合并元数据输入拆分时的目标大小。
read.split.planning-lookback	10	合并输入拆分时考虑的区间 (bin) 数量。
read.split.open-file-cost	4194304（4 MB）	打开文件的预估成本，在合并拆分时用作最小权重。
read.parquet.vectorization.enabled	false	用于控制是否使用Parquet向量化读取。
read.parquet.vectorization.batch-size	5000	Parquet向量化读取的批处理大小。
read.orc.vectorization.enabled	false	用于控制是否使用ORC向量化读取。
read.orc.vectorization.batch-size	5000	ORC向量化读取的批处理大小。

Write 属性参数

表 1-2 Write 属性参数介绍

参数	默认值	参数说明
write.format.default	parquet	表的默认文件格式，支持parquet、avro和orc格式。
write.delete.format.default	parquet（默认与“write.format.default”的值相同）	表的默认删除文件格式，支持parquet、avro和orc格式。
write.parquet.row-group-size-bytes	134217728（128 MB）	用于设置Parquet行组大小。
write.parquet.page-size-bytes	1048576（1 MB）	用于设置Parquet页大小。
write.parquet.page-row-limit	20000	用于设置Parquet页行限制。

参数	默认值	参数说明
write.parquet.dict-size-bytes	2097152（2 MB）	用于设置Parquet字典页大小。
write.parquet.compression-codec	zstd	用于设置Parquet压缩编解码器，支持设置为zstd、lz4、gzip、snappy或uncompressed。
write.parquet.compression-level	null	用于设置Parquet的压缩级别。
write.parquet.bloom-filter-enabled.column.col1	-	用于控制在写入Parquet文件时是否为特定列“col1”启用布隆过滤器。
write.parquet.bloom-filter-max-bytes	1048576（1 MB）	用于设置布隆过滤器位集的最大字节数。
write.parquet.bloom-filter-fpp.column.col1	0.01	用于设置在写入Parquet文件时，“col1”的布隆过滤器的误报概率，取值范围为0.0~1.0。
write.avro.compression-codec	gzip	用于指定在写入Avro文件时使用的压缩编解码器，支持设置为gzip（默认为9级Deflate）、zstd、snappy或uncompressed。
write.avro.compression-level	null	用于设置Avro压缩级别。
write.orc.stripe-size-bytes	67108864（64 MB）	用于定义默认的ORC条带大小，单位为字节。
write.orc.block-size-bytes	268435456（256 MB）	用于定义ORC文件的默认文件系统块大小，单位为字节。
write.orc.compression-codec	zlib	用于设置ORC压缩编解码器，支持设置为zstd、lz4、lzo、zlib、snappy或none。
write.orc.compression-strategy	speed	用于设置ORC压缩策略，支持speed、compression。
write.orc.bloom.filter.columns	-	需要创建布隆过滤器的列名的逗号分隔列表。
write.orc.bloom.filter.fpp	0.05	用于设置布隆过滤器的误报概率，取值范围为0.0~1.0。
write.location-provider.impl	null	可选的LocationProvider自定义实现。
write.metadata.compression-codec	none	元数据压缩编解码器，支持none或gzip。

参数	默认值	参数说明
write.metadata.metrics.max-inferred-column-defaults	100	定义收集指标的顶级列的最大数量。对于具有嵌套字段的表，存储的指标数量可能高于此限制。
write.metadata.metrics.default	truncate(16)	表中所有列的默认指标模式，支持none、counts、truncate(*length*)或full。
write.metadata.metrics.column.col1	-	“col1”列的指标模式，允许按列调整，支持设置为none、counts、truncate(*length*)或full。
write.target-file-size-bytes	536870912（512 MB）	用于控制生成文件的大小，单位为字节。
write.delete.target-file-size-bytes	67108864（512 MB）	用于控制生成的删除文件的大小，单位为字节。
write.distribution-mode	none（不同引擎该参数默认值不同）	定义写入数据的分布模式， ● none：不打乱行。 ● hash：按分区键进行哈希分布。 ● range：如果表有SortOrder，则按分区键或排序键进行范围分布。
write.delete.distribution-mode	hash	定义写入删除数据的分布模式。
write.update.distribution-mode	hash	定义写入更新数据的分布模式。
write.merge.distribution-mode	none	定义写入合并数据的分布模式。
write.wap.enabled	false	是否启用写-审计-发布（write-audit-publish）写入数据。
write.summary.partition-limit	0	如果更改的分区数量小于此限制，则在快照摘要中包含分区级别的摘要统计信息。
write.metadata.delete-after-commit.enabled	false	控制是否在提交后删除最旧的被跟踪的版本元数据文件。
write.metadata.previous-versions-max	100	提交后保留的先前版本元数据文件的最大数量，超过此数量后将删除。
write.object-storage.enabled	false	是否启用对象存储位置提供器，该提供器会向文件路径添加哈希组件。

参数	默认值	参数说明
write.object-storage.partitioned-paths	true	用于控制Iceberg是否在写入对象存储时使用分区路径存储。
write.data.path	*表位置* + /data	用于设置数据文件的基本位置。
write.metadata.path	*表位置* + /metadata	用于设置元数据文件的基本位置。
write.delete.mode	copy-on-write	用于设置删除命令使用的模式，支持copy-on-write和merge-on-read（v2v3支持）。
write.delete.isolation-level	serializable	用于设置删除命令的隔离级别，支持serializable和snapshot。
write.update.mode	copy-on-write	用于设置更新命令使用的模式，支持copy-on-write和merge-on-read（v2v3支持）。
write.update.isolation-level	serializable	用于设置更新命令的隔离级别，支持serializable和snapshot。
write.merge.mode	copy-on-write	用于设置合并命令使用的模式，支持copy-on-write和merge-on-read（v2v3支持）。
write.merge.isolation-level	serializable	用于设置合并命令的隔离级别，支持serializable和snapshot。

表行为属性参数

表 1-3 表行为属性参数介绍

参数	默认值	参数说明
commit.retry.num-retries	4	提交失败前的重试次数。
commit.retry.min-wait-ms	100	重试提交前等待的最小时间（毫秒）。
commit.retry.max-wait-ms	60000（1 分钟）	重试提交前等待的最大时间（毫秒）。
commit.retry.total-timeout-ms	1800000（30 分钟）	提交的总重试超时时间（毫秒）。
commit.status-check.num-retries	3	在连接丢失后，检查提交是否成功的次数，超过后因未知提交状态而失败。

参数	默认值	参数说明
commit.status-check.min-wait-ms	1000（1秒）	重试状态检查前等待的最小时间（毫秒）。
commit.status-check.max-wait-ms	60000（1 分钟）	重试状态检查前等待的最大时间（毫秒）。
commit.status-check.total-timeout-ms	1800000（30 分钟）	提交状态检查必须成功的总超时时间（毫秒）。
commit.manifest.target-size-bytes	8388608（8 MB）	合并清单文件时的目标大小。
commit.manifest.min-count-to-merge	100	合并前需累积的清单文件最小数量。
commit.manifest-merge.enabled	true	控制是否在写入时自动合并清单。
history.expire.max-snapshot-age-ms	432000000（5 天）	过期快照时，表及其所有分支上保留的快照的默认最大年龄。
history.expire.min-snapshots-to-keep	1	过期快照时，表及其所有分支上保留的快照的默认最小数量。
history.expire.max-ref-age-ms	Long.MAX_VALUE（永久）	对于除主分支之外的快照引用，过期快照时保留的快照引用的默认最大年龄，主分支永不过期。

其他参数

参数	默认值	参数说明
format-version	2	元数据格式版本，决定表支持的功能和写入能力。 ● v1：只支持Append写入，不支持行级删除和修改操作。 ● v2（默认值）：支持行级Delete/Update/Merge操作，兼容增量更新与删除和修改操作。 ● v3：当前仅在Spark引擎支持variant，支持行级血缘跟踪和二进制删除特性。
compatibility.snapshot-id-inheritance.enabled	false	允许在未指定快照ID的情况下提交快照；若format-version大于 1，该配置强制生效
write.spark.fanout.enabled	false	Spark引擎特定参数，在Spark中是否启用扇出写入器（fanout writer），启用后不需要数据聚类，但会使用更多内存。

参数	默认值	参数说明
write.spark.accept-any-schema	false	Spark引擎特定参数，Spark 写入时允许数据 schema 与表 schema 不完全匹配，会自动兼容字段新增 / 顺序调整。
spark.sql.native-read	false	Spark引擎特定参数，开启后会使用Iceberg Native读
spark.sql.native-write	false	Spark引擎特定参数，开启后会使用Iceberg Native写
checkencoding	low	Aura引擎特定参数，用于控制读取字符串时的UTF-8编码校验行为，取值范围：no，不做编码检验；low，做编码校验，并且替换非法字符；high，做编码校验，有非法字符时报错。

目录配置参数

参数	默认值	参数说明
catalog-impl	org.apache.iceberg.lakeformation.LakeFormationCatalog	计算引擎所使用的自定义目录实现类。
io-impl	null	目录所使用的自定义文件I/O实现类。
warehouse	null	数据仓库根路径。
uri	null	统一资源标识符，例如Hive元数据服务地址。
clients	2	客户端连接池大小。
cache-enabled	true	是否开启目录条目缓存。
cache.expiration-interval-ms	30000	目录条目本地缓存有效期（单位：毫秒）；设为 0 表示关闭缓存，负数表示永久缓存。

1.1.3 Iceberg Catalog 介绍

Iceberg Catalog是Iceberg的顶层组件，负责管理所有Iceberg表的元数据和元数据操作。 Catalog管理表的架构和元数据，提供了创建、查询和修改表的接口，是用户进行Iceberg表操作的入口点。用户可以通过它找到每个表当前元数据文件的位置，是读取和写入Iceberg表的关键组件。

当前AI DataLake版本支持使用LakeFormation Catalog作为Iceberg表的Catalog组件。

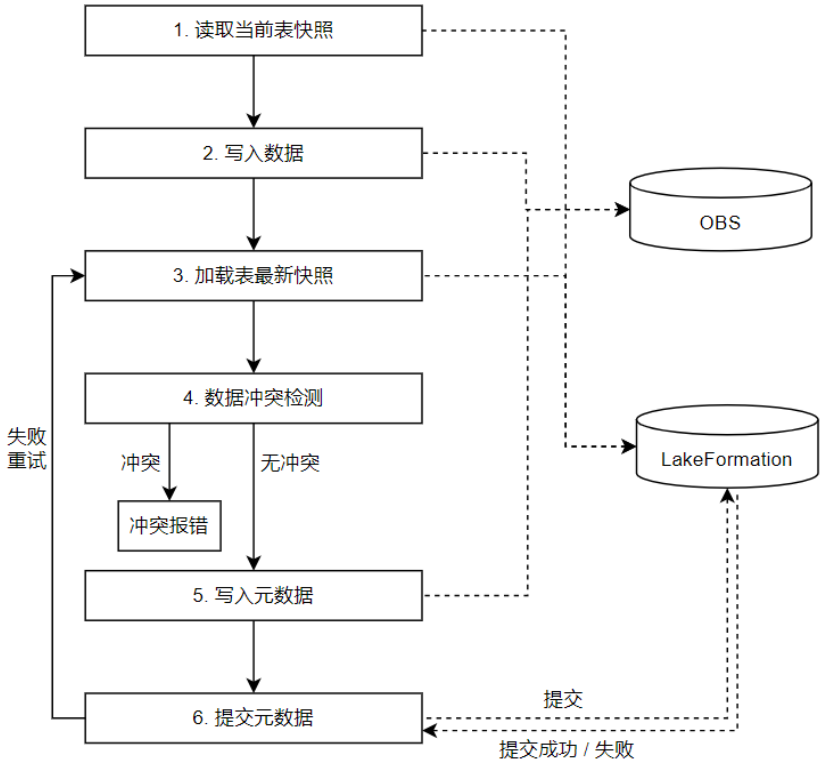
LakeFormation Catalog

LakeFormation Catalog依赖LakeFormation元数据服务，由LakeFormation管理最新的快照信息。

LakeFormation Catalog采用乐观锁（OCC）机制，保证了多租场景下并发写的数据一致性。

当前在AI DataLake上创建的Iceberg表均为LakeFormation Catalog表。

下图简述了LakeFormation Catalog的处理并发提交流程。



1. 从LakeFormation上读取当前表快照信息，包括最新的MetaData File Path、SnapshotId等。
2. 依据当前快照，将新的数据写入到表中。
3. 加载表最新快照。
4. 检测数据冲突，如果发生数据冲突，则语句执行失败，否则尝试事务提交。
5. 写入元数据，包括Manifest Files、Manifest List和MetaData File。
6. 将最新的MetaData File Path和SnapshotId等提交给LakeFormation。如果提交失败，则从步骤3重试。

1.1.4 Iceberg 表 Schema 演进介绍

Iceberg支持原地表演进，不需要新建表和重写数据，通过修改元数据就可以完成表演进。

Schema 演进介绍

Iceberg支持以下Schema演进：

- 添加：向表或嵌套结构中添加新列。
- 删除：从表或嵌套结构中移除现有列。
- 重命名：重命名现有列或嵌套结构中的字段。
- 更新：扩大列、结构字段、Map键、Map值或列表元素的类型。需注意，Map键不支持添加或删除会改变相等性的结构字段。
- 重新排序：更改列或嵌套结构中字段的顺序。

Iceberg的模式更新是元数据更改，因此执行更新不需要重写任何数据文件。

数据类型支持的演进范围

- 基本数据类型支持的演进范围如表1-4所示。

  表 1-4 基本数据类型支持的演进范围

原始类型	v1、v2有效类型提升	注意事项
unknown	-	-
int	long	-
date	-	不允许提升到timestamptz或timestamptz_ns。超出提升类型范围的值会导致任务运行失败。
float	double	-
decimal(P, S)	decimal(P', S)（P' > P）	仅扩大精度。

- struct类型支持的演进范围如下：
  - 删除struct中的字段。
  - struct添加新字段。
  - 重命名struct中的字段。
  - 重新排序struct中的字段。
  - 演进struct中的字段（参考表1-4）。

1.1.5 Iceberg 表支持的数据类型介绍

基本数据类型

Iceberg支持的基本数据类型请参见表1-5。

⚠ 注意

- 不含时区的时间戳（Timestamp）值表示一个日期和一天中的时间，与时区无关，时间值独立于时区调整，例如2017-11-16 17:10:34总是被读取为2017-11-16 17:10:34，即具体的时间值（不带时区）。
- 含时区的时间戳（Timestamptz）值表示时间线上的一个点，值以UTC存储，并且不保留源时区，例如2017-11-16 17:10:34 PST被存储/读取为2017-11-17 01:10:34 UTC，并且这些值被认为是相同的。
- 字符串必须存储为UTF-8编码的字节数组。

表 1-5 基本数据类型介绍

基本类型	说明
boolean	表示真或假。
int	表示32位有符号整数。 可以提升为long类型。
long	表示64位有符号整数。
float	表示32位IEEE 754浮点数。 可以提升为double类型。
double	表示64位IEEE 754浮点数。
decimal(P,S)	定点小数，P表示精度，S表示小数点后的位数。 小数点后的位数是固定的，精度必须小于或等于38。
date	表示日历日期（不含时区或时间）。
timestamp	表示时间戳，微秒精度（不含时区）。
timestamp_tz	表示时间戳，微秒精度（含时区）。
string	表示任意长度的字符序列，使用UTF-8编码。
binary	表示任意长度字节数组。
geometry	平面几何类型，存储点、线、面等二维空间数据。
geography	地理坐标类型，基于经纬度的球面地理位置数据。
timestamp_ns	时间戳，纳秒精度，无时区。
timestamp_tz_ns	时间戳，纳秒精度，带时区。
unknown	未知类型（仅用于schema未确定时的占位），只能为NULL，不写入数据文件。

嵌套类型

- Struct：表示一个类型化值的元组。元组中的每个字段都有名称和一个在表模式中唯一的整数ID。每个字段可以为可选或必需，即值是否可以为null。字段可以为任意类型，且可以有一个可选的注释或文档字符串，也可以有默认值。
- List： 表示具有某种元素类型的值的集合。元素字段有一个在表模式中唯一的整数ID。元素可以为可选或必需，可以为任意类型。
- Map：表示键值对的集合，具有键类型和值类型。键字段和值字段都有一个在表模式中唯一的整数ID。映射键是必需的，而映射值可以为可选或必需。映射键和映射值可以为任意类型，包括嵌套类型。

半结构化类型

Variant： 是一种存储半结构化数据的值。变体中的结构和数据类型在表或数据文件的不同行之间不一定一致。变体类型和二进制编码在Parquet项目中定义，目前支持v1。对Variant的支持在Iceberg v3中添加。

- 变体类似于JSON，但具有更广泛的基本值集合，包括日期、时间戳、带时区的时间戳、二进制和小数。
- 变体值可能包含嵌套类型：
  - 数组 (Array)： 是变体值的有序集合。
  - 对象 (Object)： 是字段的集合，每个字段包含一个字符串键和一个变体值。
- 作为一种半结构化类型，变体与Iceberg其他数据类型的区别为：
  - 变体数组类似于列表，但可以包含任何变体值，而不是固定的元素类型。
  - 变体对象类似于结构，但可以包含由名称标识的可变字段，并且字段值可以是任何变体值，而不是固定的字段类型。

1.1.6 Iceberg 隐式分区和分区演进介绍

隐式分区是Iceberg内置的自动分区机制，由表自动根据原始字段（例如时间戳）通过内置函数计算分区，用户无需手动维护分区列，使得查询与物理存储解耦，可直接修改分区规则而不用迁移数据。

典型使用场景：

- 日志、埋点、时序数据按天/小时/月分区。
- 多维度分区（时间 + 级别/类型/业务线）。
- 表需要修改分区策略但无需重写数据。
- 多人协作，避免手动分区字段误写错。

Iceberg隐式分区的核心是自动根据原始字段计算分区、无需用户维护物理分区列，既能通过分区裁剪大幅提升查询效率，又能实现存储与逻辑解耦，支持分区方案灵活演进且无需进行数据迁移。

隐式分区转换规则

隐式分区转换规则请参见**表1-6**，其中：

- 所有转换对于null输入值都必须返回null。
- void转换可用于替换现有分区字段中的转换，以便在v1表中有效地删除该字段，详细说明请参见**分区演进**。

- bucket[N]：用于对源值应用哈希函数（通常是Iceberg的特定实现），然后对N取模，得到一个介于0到N-1之间的桶号。

  不同的数据类型（例如int和string）即使逻辑值相同（例如34和"34"），也会产生不同的哈希值，因此不能直接进行类型升级。

- truncate[W]
  - 对于数字类型（int、long、decimal）：将值截断为不超过该值、且为W倍数的最大数值。例如，truncate[10]会将17转换为10。
  - 对于字符串和二进制类型：截断到前W个字符/字节。例如，truncate[3]会将**"iceberg"**转换为**"ice"**。

表 1-6 隐式分区转换规则

转换名称	说明	源类型	转换结果类型
identity	源值，未经修改。	除geometry、geography、variant外的任何类型。	源类型
bucket[N]	值的哈希值，对N取模。	int、long、decimal、date、time、timestamp、timestamptz、timestamp_ns、timestamptz_ns、string、uuid、fixed、binary	int
truncate[W]	值被截断到宽度W。	int、long、decimal、string、binary	源类型
year	提取日期或时间戳的年份。	date、timestamp、timestamptz、timestamp_ns、timestamptz_ns	int
month	提取日期或时间戳的月份。	date、timestamp、timestamptz、timestamp_ns、timestamptz_ns	int
day	提取日期或时间戳的日期。	date、timestamp、timestamptz、timestamp_ns、timestamptz_ns	int
hour	提取时间戳的小时。	timestamp、timestamptz、timestamp_ns、timestamptz_ns	int
void	总是产生null。	任何数据类型	源类型或int

分区演进

表分区可以通过添加、删除、重命名或重新排序分区规范字段来进行演进。

更改分区规范会产生一个新的规范，该规范由一个唯一的规范ID标识。新规范会被添加到表的分区规范列表中，并且可以被设置为表的默认规范。

演进规则：

- 保持字段ID稳定：在演化规范时，更改不应导致分区字段ID发生变化。分区字段ID在清单文件中被用作分区元组的字段ID。
- v2表：在v2中，必须为每个分区字段显式跟踪其分区字段ID。新的ID根据表元数据中最后分配的分区ID来分配。
- v1表（兼容性）：在v1中，分区字段ID没有被显式跟踪，但在参考实现中是按顺序从1000开始分配的。由于具有相同ID的分区字段可能包含不同的数据类型，这种分配方式在基于来自多个规范的清单文件读取元数据表时会产生问题。为了与旧版本兼容，建议v1表的分区演化遵循以下规则：
  - 不要重新排序分区字段。
  - 不要删除分区字段，应使用void转换替换该字段的转换方式来“软删除”。
  - 只能在先前分区规范的末尾添加新的分区字段。

1.2 Iceberg on Aura

1.2.1 使用 Iceberg 前准备

设置 GUC

可选GUC参数**enable_meta_scan**，用于优化查询的性能。

该参数默认打开，但在表数据量很小的情况下，关闭该参数时，查询性能可能比打开时更高，请基于实际情况打开或关闭该GUC。

代码示例如下：

```
SHOW enable_meta_scan;
SET enable_meta_scan=off;
SET enable_meta_scan=on;
或
SET enable_meta_scan=true;
SET enable_meta_scan=false;
```

1.2.2 Iceberg 配置与类型

Aura 类型和 Iceberg 类型对应关系

表 1-7 Aura 类型对应的 Iceberg 类型

Aura	Iceberg
smallint	integer
tinyint	integer

Aura	Iceberg
integer	integer
bigint	long
numeric	decimal
decimal	decimal
real	float
float4	float
double precision	double
float8	double
boolean	boolean
char	string
varchar	string
text	string
date	date
timestamp without time zone	timestamp without time zone
bytea	binary
array	list
struct	struct

表 1-8 Iceberg 类型对应的 Aura 类型

Iceberg	Aura
boolean	boolean
integer	integer
long	long
float	float
double	double
date	date
time	不支持
timestamp with time zone	不支持
timestamp without time zone	timestamp without time zone

Iceberg	Aura
string	string
uuid	不支持
fixed	不支持
binary	bytea
decimal	decimal
struct	struct
list	array
map	不支持
nanosecond timestamp	不支持
nanosecond timestamp with time zone	不支持
unknown	不支持
variant	不支持
geometry	不支持
geography	不支持

1.2.3 Iceberg on Aura DDL 语法说明

创建 Iceberg 表

通过CREATE TABLE语法创建Iceberg表。与其他格式的表相比，创建Iceberg表没有特别的参数需要指定，只需要指定STORE AS ICEBERG即可。具体语法请参见**CREATE TABLE**。

示例：

```
CREATE TABLE iceberg_ext(
col1 int,
col2 varchar(20),
...
)PARTITION BY (col3 bigint)
TABLEPROPERTIES (
'write.metadata.delete-after-commit.enabled'='true',
'write.metadata.previous-versions-max' = '10'
) STORE AS ICEBERG;
```

其中write.metadata.delete-after-commit.enabled控制是否在提交事务后删除旧版本元数据文件，默认值为false。当该参数设为true时，可以通过write.metadata.previous-versions-max设定保留多少个metadata。

使用这两个参数需要谨慎，虽然删除元数据文件可以节省存储空间，但需要确保不会影响数据的一致性和可用性。

清空 Iceberg 表

通过TRUNCATE TABLE语法清空表中的数据，具体语法请参见**TRUNCATE**。

示例：

```
TRUNCATE TABLE iceberg_ext;
```

删除 Iceberg 表

通过DROP TABLE语法删除Iceberg表，会同时删除表的元数据及数据（Managed表删除后可以从LakeFormation控制台恢复表的元数据及数据）。具体语法，请参见**DROP TABLE**。

示例：

```
DROP TABLE iceberg_ext;
```

修改表中的列

通过ALTER TABLE语法的ADD COLUMN、RENAME COLUMN TO、ALTER COLUMN、DROP COLUMNS能力，可以对iceberg表的列进行加列、删列、重命名列、更新列和重新排序列等schema evolution操作。

说明

iceberg通过唯一编号fieldId为列进行编号，在写入数据时会将fieldId写入文件的schema中，在读取文件的schema时通过fieldId将元数据的列与文件中的列映射起来，能够实现schema evolution后的读取。对于非iceberg写入的文件，schema中不存在fieldId，读取文件时会以列名为依据进行映射，进行schema evolution操作后，如果存在复用列名的情况，可能会导致类型不匹配而读取失败或者读取到了以前的数据。

示例：

建iceberg分区表：

```
CREATE TABLE t_iceberg(a int, b array<struct<x:int, y:int>>) partition by (c int) store as iceberg;
```

- 添加列：

  iceberg表支持添加一列或者多列到表中或者复杂类型字段中，需要指定列名和类型，可以指定注释和位置。当添加到复杂类型字段中时，不能指定注释。

  ```
  -- 添加到表中
  alter table t_iceberg add column d int, e float comment 'this is e', f array<int> after a, g struct<x:int,y:int> comment 'this is g' before b;
  -- 添加到复杂类型字段中，需要使用element来指定array的子类型
  alter table t_iceberg add column b.element.z int after x;
  ```

- 重命名列：

  iceberg表支持对表列或者复杂类型字段进行重命名。

  ```
  alter table t_iceberg rename column a to newa;
  alter table t_iceberg rename column b.element.x to newx;
  ```

- 更新列：

  iceberg表支持更新表列的类型、注释和位置，支持更新复杂类型字段的类型和位置。

  ```
  alter table t_iceberg alter column a type bigint comment 'this is a' after c, c type bigint comment 'this is c' after a, b.element.y type bigint after first;
  ```

  当前仅支持从低精度类型更新到高精度类型：

  – 整数类型：从smallint更新到int或者bigint，从int更新到bigint。

  - 浮点类型：从float4更新到float8。
  - numeric类型：从低precision更新到高precision，不支持更新scale。
  - 字符串类型：varchar或者char的长度变长。

  注意

  浮点类型的存储是不精确的，从float4更新到float8后，数据的小数部分精度会变化，结果会发生变化。

- 删除列

  iceberg支持删除表列或者复杂类型字段。不能删除分区列，不能删除表的所有列或者复杂类型的所有字段。

  ```
  alter table t_iceberg drop columns (a, b.x);
  ```

修改表属性

通过ALTER TABLE语法的SET/UNSET TABLEPROPERTIES能力，可以对表的属性值进行修改。

示例：

```
ALTER TABLE iceberg_ext SET TABLEPROPERTIES ('write.metadata.delete-after-commit.enabled' = 'false');
ALTER TABLE iceberg_ext UNSET TABLEPROPERTIES ('write.metadata.delete-after-commit.enabled');
```

1.2.4 查询和写入 Iceberg 表

查询 Iceberg

可以查询Iceberg表中的数据。具体语法可参考**SELECT**。

说明

Iceberg v3表支持查询系统列_rowid,_last_updated_sequence_number。

注意事项：

当前仅支持查询最新全量数据。

示例：

```
SELECT * FROM iceberg_ext order by col1;
```

写入 Iceberg 表

可以向Iceberg表中添加一行或多行数据。具体语法可参考**INSERT**。

注意事项：

- 如果频繁插入小数据，可能会出现小文件问题，建议定期重写data/manifest文件以保障存储、查询效率。
- Manifest文件在DML语句事务提交时，引擎会自动尝试触发重写，以减少manifest小文件个数，防止查询性能劣化。
- 当前仅支持数据文件以Parquet格式写入。

示例：

```
INSERT INTO iceberg_ext VALUES (1, 'a', 1.1234567, 100.11, '2025-03-03', '2025-03-03 16:00:00');
INSERT OVERWRITE INTO iceberg_ext VALUES (1, 'b', 1.1234567, 100.11, '2025-03-03', '2025-03-03
16:00:00');
INSERT INTO iceberg_ext (col1, col2, col3, col4, col5, col6) SELECT col1, col2, col3, col4, col5, col6 FROM
iceberg_table WHERE col1 > 18;
```

删除 Iceberg 表数据

可以对Iceberg表中的一行或多行数据执行删除操作。具体语法可参考**DELETE**。

注意事项：

- 当前仅支持merge-on-read方式的delete操作。
- 如果有并发删除、更新时，delete时建议指定分区条件以减少并发冲突。
- 如果对分区表做整分区删除时，delete会通过标记元数据状态来加速删除操作，是否可以走优化可以通过explain delete查看关键字“Do speed up for delete”，如果启用了删除优化，删除返回的行数可能大于实际删除的行数。

示例：

```
DELETE FROM iceberg_ext where a = 1;
```

更新 Iceberg 表数据

可以对Iceberg表中的一行或多行数据执行更新操作。具体语法可参考**UPDATE**。

注意事项：

- 当前仅支持merge-on-read方式的update操作。
- 如果有并发删除、更新时，update时建议指定分区条件以减少并发冲突。

示例：

```
UPDATE iceberg_ext set b = 1 where a = 1;
```

合并 Iceberg 表数据

可以将Iceberg表与另外一个表进行Merge操作。具体语法可参考**MERGE INTO**。

注意事项：

当前仅支持merge-on-read方式的Merge操作。

示例：

```
MERGE into t_iceberg p1 using t_iceberg p2 on (p1.a = p2.c) when matched then delete;
```

1.2.5 表达式分区

在处理大规模数据集时，数据分区是提高查询性能的关键技术之一。然而，传统的分区方式需要用户手动定义分区逻辑，这不仅增加了用户的负担，还可能导致分区策略不够灵活。表达式分区用于支持Iceberg的隐藏分区功能，其核心设计是将“分区逻辑”从用户视角隐藏，交给Iceberg元数据自动管理。如何确保数据分区既高效又灵活？通过使用表达式分区，用户可以无需关心复杂的分区逻辑，Iceberg将自动根据数据特征和查询模式优化分区策略，从而提高查询性能和数据管理效率。

约束限制

- 最多支持4个分区列。
- 不支持void表达式。
- 所有分区表达式不支持复杂数据类型。
- truncate/bucket表达式不支持二进制数据。
- INSERT INTO PARTITION功能仅支持identity表达式，不支持其他表达式。
- Iceberg约束：同一个表列，不能出现在year/month/day/hour两个时间表达式中。 但是，可以同时出现在时间和bucket/truncate表达式中。
- Iceberg约束：对分区键有默认命名规则，不允许与表的列名称冲突。
- 分区演进：不支持将分区表的分区键删除为0个。
- 分区演进：仅支持对分区表执行分区键的更新操作，非分区表不支持。
- 分区演进：不支持对分区键进行命名和重命名功能。

支持的分区表达式

Aura支持的分区表达式以及对应的数据类型如**表1-9**所示。

表 1-9 Aura 支持的分区表达式以及对应的数据类型说明

表达式名称	支持的数据类型	计算结果的类型	说明
identity	smallint，int，bigint，long，numeric，date，timestamp，text，varchar，bpchar	原数据类型	不支持float，double，boolean，bytea。
bucket[N]	smallint，int，bigint，long，numeric，date, timestamp，text，bpchar，varchar	int	• 先计算输出数值的hash值，再求模。 计算结果为 [0, N)范围之内。 • 哈希算法采用的是MurmurHash3算法。
truncate[W]	smallint，int，bigint，long，numeric，text，bpchar，varchar	原数据类型	• 不支持bytea类型。 • 详情参见**分区表达式：truncate**。
year	date，timestamp	int	例：year('2025-10-10') 计算结果为'2025'年。
month	date，timestamp	int	例：month('2025-10-10') 计算结果为'2025-10'月。
day	date、timestamp	int	例：day('2025-10-10') 计算结果为'2025-10-10'日。
hour	timestamp	int	例：hour('2025-10-10 10:10:10.000') 计算结果为'2025-10-10 10:00:00'时。

表达式名称	支持的数据类型	计算结果的类型	说明
void	不支持	不支持	本质上是将所有数值映射到null值上。

所有表达式对于null值的处理结果必须返回null值。

identity表达式对输入的数值不做任何处理，直接输出原数值。

year表达式计算的是相对1970年、以年为单位的数值。month表达式计算的是相对1970-01-01、以月为单位的数值。day表达式计算的是相对1970-01-01、以天为单位的数值。hour表达式计算的是相对1970-01-01 00:00:00、以小时为单位的数值。Iceberg在记录分区值结果时，并不会直接记录整数数值，而是将它转换为可阅读的时间字符串。例如，Iceberg会将year('2025-10-10') 表达式计算出来的分区值记录为'2025'年，而不是整数55。

Aura提供了SQL函数fq_iceberg_year()、fq_iceberg_month()、fq_iceberg_day()、fq_iceberg_hour()、fq_iceberg_truncate()、fq_iceberg_bucket()用于实现对应分区表达式year、month、day、hour、truncate、bucket的计算功能。

分区表达式：truncate

对于不同的数据类型，truncate表达式的计算方法不同，可以使用函数fq_iceberg_truncate() 进行运算。

对于整数类型，truncate(W, integer_value)返回的是不超过integer_value、且为W倍数的最大数值。 示例如下：

```
select fq_iceberg_truncate (10, -2) ;
fq_iceberg_truncate
---------------------
                -10
(1 row)

select fq_iceberg_truncate (10, 0) ;
fq_iceberg_truncate
---------------------
                  0
(1 row)

select fq_iceberg_truncate (10, 9) ;
fq_iceberg_truncate
--------------------
                  0
(1 row)

select fq_iceberg_truncate (10, 10) ;
fq_iceberg_truncate
---------------------
                 10
(1 row)

select fq_iceberg_truncate (10, 19) ;
fq_iceberg_truncate
--------------------
                 10
(1 row)
```

对于字符串类型，truncate(W, string_value)返回的是string_value字符串中前缀长度为W的前缀子字符串。注意，长度以字符为单位。 示例如下：

```
select *, length(t)  from fq_iceberg_truncate (2, '') as t;
iceberg_truncate | length
------------------+--------
                 |      0
(1 row)

select *, length(t)  from fq_iceberg_truncate (2, 'a') as t;
iceberg_truncate | length
------------------+--------
a                |      1
(1 row)

select *, length(t)  from fq_iceberg_truncate (2, 'tt') as t;
iceberg_truncate | length
------------------+--------
tt               |      2
(1 row)

select *, length(t)  from fq_iceberg_truncate (2, 'cow') as t;
iceberg_truncate | length
------------------+--------
co               |      2
(1 row)

 select *, length(t)  from fq_iceberg_truncate (2, '你好中国!') as t;
 iceberg_truncate | length
------------------+--------
 你好             |      2
(1 row)
```

对于numeric类型，truncate(W, num_value)返回的是不超过num_value、且小数点右移scale位后为W倍数的最大数值。示例如下：

```
select fq_iceberg_truncate (50, 10.64::numeric(15,3));
 fq_iceberg_truncate
---------------------
              10.600
(1 row)

select fq_iceberg_truncate (50, 10.66::numeric(15,3));
 fq_iceberg_truncate
---------------------
              10.650
(1 row)

select fq_iceberg_truncate (50, -10.66::numeric(15,3));
 fq_iceberg_truncate
---------------------
             -10.700
(1 row)
```

创建表达式分区表

CREATE TABLE语法详情，请参考**CREATE TABLE**。

创建一个表达式分区的Iceberg managed表SQL语句如下：

```
set current_schema to test;
CREATE TABLE iceberg_table(id bigint, data text, category text, ts timestamp, val numeric) partition by
(bucket(16, val), truncate(10, id), c3 int comment 'this is c3 field', year(ts) ) store as iceberg;
CREATE TABLE
select * from pg_get_partition_keys('test', 'iceberg_table');
     table     |  partition_key
---------------+-----------------
 iceberg_table | bucket(16,val)
 iceberg_table | truncate(10,id)
 iceberg_table | c3
 iceberg_table | year(ts)
(4 rows)
```

在示例中，c3是一个identity分区键。此外，还创建了bucket、truncate、year表达式的三个分区键。Iceberg会对分区键进行内部命名，其命名规则如下表所述。

表 1-10 不同表达式分区键在 Iceberg 内部的默认命名规则

表达式	默认命名规则	样例
year(source_column)	<source_column>_year	year(ts) -> 'ts_year'
month(source_column)	<source_column>_month	month(ts) -> 'ts_month'
day(source_column)	<source_column>_day	day(ts) -> 'ts_day'
hour(source_column)	<source_column>_hour	hour(ts) -> 'ts_hour'
bucket(N, source_column)	<source_column>_bucket	bucket(ts) -> 'ts_bucket'
truncate(W, source_column)	<source_column>_trunc	truncate(ts) -> 'ts_trunc'
identity(source_column)	<source_column>	identity(ts) -> 'ts'

表的列名称不能够与分区键的名称冲突，否则会报错、建表失败。

```
CREATE TABLE iceberg_table(id bigint, data text, category text, ts timestamp, val numeric) partition by
(bucket(16, val),  c3 int comment 'this is c3 field', year(ts), ts_year bigint ) store as iceberg;
ERROR:  column "ts_year" specified more than once
```

Iceberg不允许时间类型的同一列同时出现在year、month、day、hour多个表达式中，但是允许该列出现在truncate、bucket其他表达式中。

```
CREATE TABLE iceberg_table(id bigint, data text, category text, ts timestamp, val numeric) partition by
(bucket(16, val),  c3 int comment 'this is c3 field', year(ts), month(ts) ) store as iceberg;
ERROR:  java.lang.IllegalArgumentException: Cannot add redundant partition: 1002: ts_year: year(4)
conflicts with 1003: ts_month: month(4).

CREATE TABLE iceberg_table(id bigint, data text, category text, ts timestamp, val numeric) partition by
(bucket(16, val),  c3 int comment 'this is c3 field', year(ts), bucket(7, ts) ) store as iceberg;
CREATE TABLE
```

Aura支持创建分区键的最大数量是4，否则会报错、建表失败。

```
CREATE TABLE iceberg_table(id bigint, data text, category text, ts timestamp, val numeric) partition by
(bucket(16, val), truncate(10, id), c3 int comment 'this is c3 field', year(ts) ) store as iceberg;
ERROR:  partitioned table should have no more than 4 partition keys
```

Aura不支持将复杂类型作为分区键，否则会报错、建表失败。

```
CREATE TABLE iceberg_table(id bigint, data text, category text, ts timestamp, val numeric) partition by
(bucket(16, val), truncate(10, id), c3 struct<c: text, d: int>, year(ts) ) store as iceberg;
ERROR:  Complex type cannot be served as a value-partitioning column.

CREATE TABLE iceberg_table(a int , b struct<c: text, d: int>) PARTITION BY(bucket(5, b.c), truncate(3, b.d))
store as ICEBERG;
ERROR:  syntax error at or near "."
LINE 1: ... struct<c: text, d: int>) PARTITION BY(bucket(5, b.c), trunc...
                                                             ^
```

Aura支持创建表达式分区的Iceberg外表，需要满足额外条件：

- 与OBS上Iceberg表的分区键数量相同。

- 与OBS上Iceberg表的分区键定义完全一致，包括分区键的顺序、分区键的表达式、分区键操作的表列。

后续小节将以iceberg_test_part_expr表为样例，其表定义如下：

```
set current_schema to test;
CREATE TABLE iceberg_test_part_expr(id bigint, data text, category text, ts timestamp) partition by (c3 int, truncate(4, data), year(ts)) store as iceberg;
CREATE TABLE
```

导入数据

实现表达式分区的Iceberg表支持数据的单条插入和批量导入功能。

```
INSERT INTO iceberg_test_part_expr values(9, 'hellohello',   'shanxi', '2025-08-29', 1);
INSERT 1
INSERT INTO iceberg_test_part_expr values(9, 'hellohello',   'shanxi', '2025-08-29', 1);
INSERT 1
INSERT INTO iceberg_test_part_expr values(9, 'he',           'shanxi', '2024-09-29', 20);
INSERT 1
INSERT INTO iceberg_test_part_expr values(9, 'helloworld',   'shanxi', '2023-06-29', 6);
INSERT 1
INSERT INTO iceberg_test_part_expr values(9, 'hella',        'shanxi', '2022-05-29', 60);
INSERT 1
INSERT INTO iceberg_test_part_expr values(9, 'zhangmingmin', 'shanxi', '2025-04-29', 100);
INSERT 1
```

对于INSERT INTO PARTITION功能，Aura仅支持所有分区键为identity类型的表，并不支持其他类型表达式的分区表。

```
INSERT INTO iceberg_test_part_expr PARTITION(c3=1, data='hell', ts='2025') values (9, 'hellohello', 'shanxi', '2025-08-29 01:05:59.550463') ;
ERROR:  INSERT INTO PARTITION clause only support identity partition.
```

可以使用函数pg_get_partitions()来查看表的所有分区数据，示例如下：

```
select partition_name from pg_get_partitions('iceberg_test_part_expr');
     partition_name
--------------------------
 c3=1/data=hell/ts=2025
 c3=20/data=he/ts=2024
 c3=100/data=zhan/ts=2025
 c3=6/data=hell/ts=2023
 c3=60/data=hell/ts=2022
(5 rows)
```

基于列表达式的查询优化

在支持表达式分区之后，Aura可下推列表达式，从而进行静态分区剪枝的查询优化。示例如下：

```
explain(analyze, predicate_only on, costs off, timing off) select * from iceberg_test_part_expr where c3 = 1;
                                  QUERY PLAN
-------------------------------------------------------------------------------------------
 id |                            operation                                | A-rows
 ---+---------------------------------------------------------------------+--------
  1 | ->  Row Adapter                                                     |      2
  2 |    ->  Partitioned Vector Foreign Scan on iceberg_test_part_expr    |      2
  3 |       ->  Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr |      2

          Predicate Information (identified by plan id)
 ------------------------------------------------------------------
  2 --Partitioned Vector Foreign Scan on iceberg_test_part_expr
       Filter: (c3 = 1)
       Pushdown Predicate Filter: (c3 = 1)
       Server Type: lf
```



```
      DN read from: direct
  3 --Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr
      Filter: (c3 = 1)
      Pushdown Predicate Filter: (c3 = 1)
      Server Type: lf
      Pruning results: (Manifests total: 6, Manifests left: 2)
      DN read from: direct
(19 rows)
```

可以看到，c3=1列表达式下推后，扫描的分区数量由6个裁减为2个。

基于 SQL 函数的查询优化

在实现表达式分区之后，Aura支持白名单SQL函数的下推优化。当前主要支持的SQL函数白名单如下：

表 1-11 支持下推优化的 SQL 函数列表

分区表达式	SQL函数	下推条件
year	year(), date_trunc()	date_trunc()的时间单位是year。
month	date_trunc()	date_trunc()的时间单位是month。
day	date_trunc()	date_trunc()的时间单位是day。
hour	date_trunc()	date_trunc()的时间单位是hour。
truncate	left(), substring()	substring()必须从字符串头开始算起。

字符串函数left()和substring()下推进行静态分区剪枝的示例如下：

```
explain(analyze, predicate_only on, costs off, timing off) select * from iceberg_test_part_expr where
substring(data, 1, 3) > 'hello';
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 id |                          operation                          | A-rows
 ----+-------------------------------------------------------------+--------
  1 | ->  Row Adapter                                             |      1
  2 |    ->  Partitioned Vector Foreign Scan on iceberg_test_part_expr        |      1
  3 |       ->  Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr |      1

          Predicate Information (identified by plan id)
 ------------------------------------------------------------------
  2 --Partitioned Vector Foreign Scan on iceberg_test_part_expr
      Filter: ("substring"(data, 1, 3) > 'hello'::text)
      Server Type: lf
      DN read from: direct
  3 --Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr
      Server Type: lf
      Pruning results: (Manifests total: 6, Manifests left: 1)
      DN read from: direct
(16 rows)

explain(analyze, predicate_only on, costs off, timing off) select * from iceberg_test_part_expr where
left(data, 5) < 'hello';
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 id |                           operation                           | A-rows
----+---------------------------------------------------------------+--------
  1 | ->  Row Adapter                                               |      2
  2 |    ->  Partitioned Vector Foreign Scan on iceberg_test_part_expr        |      2
  3 |       ->  Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr |      5

          Predicate Information (identified by plan id)
-------------------------------------------------------------------
  2 --Partitioned Vector Foreign Scan on iceberg_test_part_expr
        Filter: ("left"(data, 5) < 'hello'::text)
        Server Type: lf
        DN read from: direct
  3 --Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr
        Server Type: lf
        Pruning results: (Manifests total: 6, Manifests left: 5)
        DN read from: direct
(16 rows)
```

可以看到，substring()函数下推后，分区数量由6个裁减到1个。 而left()函数下推后，分区数量由6个裁减到5个。

ts类型必须为timestamp类型才能剪枝，如果是date类型则不能剪枝。

时间类型函数date_trunc()下推进行静态分区剪枝的示例如下：

```
explain(analyze, predicate_only on, costs off, timing off) select * from iceberg_test_part_expr where
date_trunc('year', ts) > '2023-12-30 00:00:00';
                                QUERY PLAN
-------------------------------------------------------------------------------------------------------
 id |                           operation                           | A-rows
----+---------------------------------------------------------------+--------
  1 | ->  Row Adapter                                               |      4
  2 |    ->  Partitioned Vector Foreign Scan on iceberg_test_part_expr        |      4
  3 |       ->  Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr |      4

                    Predicate Information (identified by plan id)
-------------------------------------------------------------------------------------------------------
  2 --Partitioned Vector Foreign Scan on iceberg_test_part_expr
        Filter: (date_trunc('year'::text, ts) > '2023-12-30 00:00:00'::timestamp without time zone)
        Server Type: lf
        DN read from: direct
  3 --Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr
        Server Type: lf
        Pruning results: (Manifests total: 6, Manifests left: 4)
        DN read from: direct
(16 rows)

explain(analyze, predicate_only on, costs off, timing off) select * from iceberg_test_part_expr where year(ts)
< 2024;
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 id |                           operation                           | A-rows
----+---------------------------------------------------------------+--------
  1 | ->  Row Adapter                                               |      2
  2 |    ->  Partitioned Vector Foreign Scan on iceberg_test_part_expr        |      2
  3 |       ->  Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr |      2

          Predicate Information (identified by plan id)
-------------------------------------------------------------------
  2 --Partitioned Vector Foreign Scan on iceberg_test_part_expr
        Filter: (year(ts) < 2024)
        Server Type: lf
        DN read from: direct
  3 --Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr
        Server Type: lf
        Pruning results: (Manifests total: 6, Manifests left: 2)
        DN read from: direct
(16 rows)
```


可以看到，date_trunc()函数下推后，分区数量由6个裁减到4个。 而year()函数下推后，分区数量由6个裁减到2个。

分区演进

分区演进（Partition Evolution）是一种原位的表演进功能。它不需要重写已有的数据，只会影响后续新插入的数据。

删除、新增、直接覆盖分区键的示例如下：

```
ALTER TABLE iceberg_test_part_expr DROP PARTITION COLUMN c3;
ALTER TABLE iceberg_test_part_expr ADD PARTITION COLUMN c3;
ALTER TABLE iceberg_test_part_expr SET PARTITION COLUMN c3, year(ts);
select partition_key from pg_get_partition_keys('test', 'iceberg_test_part_expr');
  partition_key
-----------------
  c3
  year(ts)
(2 rows)
```

在支持分区演进之后，iceberg表的INSERT OVERWRITE将与Spark dynamic overwrite行为保持一致。也就是说，SELECT查询结果中列对应的表达式分区将会被覆盖，更多信息可以阅读链接**Spark Dynamic Overwrite**。

以下示例说明该功能影响结果：

```
create table test_overwrite_tb ( a int, b int, c text) partition by (bucket(12, b )) store as iceberg ;
insert into test_overwrite_tb values(1,1,'1111'), (2,2, '222222');
select partition, spec_id from iceberg_partitions('test_overwrite_tb');
  partition   | spec_id
--------------+---------
 {b_bucket=8} |       0
 {b_bucket=0} |       0
(2 rows)
select * from test_overwrite_tb ;
 a  | b |   c
----+---+--------
 11 | 1 | 1111
  2 | 2 | 222222
(2 rows)

alter table test_overwrite_tb add partition column bucket(12, a );
alter table test_overwrite_tb drop partition column bucket(12, b );
insert overwrite into test_overwrite_tb select 11,1, 3333;
```

最后一条INSERT OVERWRITE执行时，分区键为bucket(12,a)，这与建表时的分区键不同，因此INSERT OVERWRITE不会覆盖已有的分区数据，而是会写入一个新的分区。验证结果如下：

```
select * from test_overwrite_tb;
 a  | b |   c
----+---+--------
 11 | 1 | 1111
  2 | 2 | 222222
 11 | 1 | 3333
(3 rows)

select partition, spec_id from iceberg_partitions('test_overwrite_tb');
    partition    | spec_id
-----------------+---------
 {b_bucket=8}    |       0
 {a_bucket_12=3} |       2
 {b_bucket=0}    |       0
(3 rows)
```

此时，如果插入a=11的新记录，将会覆盖当前分区的所有数据。

```
insert overwrite into test_overwrite_tb select 11,4, 4444 ;
select * from test_overwrite_tb ;
 a  | b |   c
----+---+--------
 11 | 1 | 1111
  2 | 2 | 222222
 11 | 4 | 4444
(3 rows)
```

1.2.6 Iceberg 表服务

Aura提供一系列表服务，用户可以根据需要使用，例如清理旧快照，整理元数据、数据文件等，以提高存储、查询的效率。具体语法可参考**Iceberg表服务函数**。

示例：

- 清理旧快照
```
select * from iceberg_expire_snapshots('iceberg_test','2025-05-15 14:12:00+08');
deleted_data_files_count | deleted_position_delete_files_count | deleted_equality_delete_files_count |
deleted_manifest_files_count | deleted_manifest_lists_count | deleted_statistics_files_count
--------------------------+-------------------------------------+-------------------------------------
+------------------------------+------------------------------+------------------------------
0                              | 0                                   |
0                                         | 0                                  |
3                                  | 0
(1 row)
```
- 重写元数据文件
```
SELECT * from iceberg_rewrite_manifests('test1');
rewritten_manifests_count | added_manifests_count
---------------------------+----------------------
2                           | 1
(1 row)
```
- 清除孤立文件
```
SELECT * from iceberg_remove_orphanfiles('iceberg_test');
orphan_file_location
-----------------------------------------------------------
obs://xxxx/test.txt
obs://xxxx/test2.txt
(2 rows)
```
- 重写数据文件
```
OPTIMIZE reason_t2 REWRITE DATA WITH OPTIONS ('rewrite-all' = 'true') WHERE TABLE_SC =
'Hainan';
INSERT 2
```

1.3 Iceberg on Spark

1.3.1 Iceberg 配置与类型

Spark 类型和 Iceberg 类型对应关系

Spark和Iceberg支持不同的类型集。Iceberg会自动进行类型转换，但并非所有组合都支持，因此在设计表的列类型之前，您需要了解Iceberg中的类型转换。

Spark类型对应的Iceberg类型

此类型转换表描述了Spark类型如何转换为Iceberg类型。转换在创建Iceberg表和通过Spark写入Iceberg表时都适用。

表 1-12 Spark 类型对应的 Iceberg 类型

Spark	Iceberg
boolean	boolean
short	integer
byte	integer
integer	integer
long	long
float	float
double	double
date	date
timestamp	timestamp with timezone
timestamp_ntz	timestamp without timezone
char	string
varchar	string
string	string
binary	binary
decimal	decimal
struct	struct
array	list
map	map

说明

该表基于创建表时的转换表示。实际上，写入时支持类型更广泛。以下是写入时的一些要点：

- Iceberg数值类型（`integer`、`long`、`float`、`double`、`decimal`）在写入时支持提升。例如，您可以将Spark类型`short`、`byte`、`integer`、`long`写入Iceberg类型`long`。
- 您可以使用Spark`binary`类型写入Iceberg`fixed`类型。请注意，将执行长度断言。

Iceberg类型对应的Spark类型

此类型转换表描述了Iceberg类型如何转换为Spark类型。转换在通过Spark读取Iceberg表时适用。

表 1-13 Iceberg 类型对应的 Spark 类型

Iceberg	Spark
boolean	boolean

Iceberg	Spark
integer	integer
long	long
float	float
double	double
date	date
time	不支持
timestamp with timezone	timestamp
timestamp without timezone	timestamp_ntz
string	string
uuid	string
fixed	binary
binary	binary
decimal	decimal
struct	struct
list	array
map	map
nanosecond timestamp	不支持
nanosecond timestamp with timezone	不支持
unknown	不支持
variant	variant **说明** 支持Spark4.0+
geometry	不支持
geography	不支持

运行时配置

配置设置的优先级

Iceberg允许在不同级别指定配置。读或写操作的有效配置按以下优先级顺序确定：

1. DataSourceAPI读/写选项：在读/写操作中显式传递给.option(...)。
2. Spark会话配置：通过spark.conf.set(...)、spark-defaults.conf或spark-submit中的--conf全局设置。

3. 表属性：通过ALTER TABLE SET TBLPROPERTIES在Iceberg表上定义。
4. 默认值。

如果未在更高级别定义设置，则使用下一级作为后备。这允许在需要时启用全局默认值，保持灵活性。

Spark SQL选项

Iceberg支持使用SparkSQL配置选项设置各种全局行为。可以通过spark、SparkSession设置或Spark提交参数进行设置。例如：

```
//禁用向量化
val spark = SparkSession.builder()
.appName("IcebergExample")
.master("local[*]")
.config("spark.sql.catalog.my_catalog","org.apache.iceberg.spark.SparkCatalog")
.config("spark.sql.extensions","org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions")
.config("spark.sql.iceberg.vectorization.enabled","false")
.getOrCreate()
```

Spark选项	默认值	描述
spark.sql.iceberg.vectorization.enabled	继承表属性（Parquet的read.parquet.vectorization.enabled默认true；ORC的read.orc.vectorization.enabled默认false）	启用数据文件的向量化读取。 ● 设置为true时，Iceberg使用基于批次的列向量化（Columnar Vectorization）来加速查询，每次批量处理5000行（由read.parquet.vectorization.batch-size或read.orc.vectorization.batch-size控制）。 ● 设置为false时将使用非向量化的逐行读取器，性能较低但兼容性更好。
spark.sql.iceberg.parquet.reader-type	ICEBERG	设置Parquet读取器实现。可选值： ● ICEBERG（默认值）：使用Iceberg内置的Parquet读取器。 ● COMET：使用Apache DataFusion CometParquet读取器（在JVM中执行I/O和解压缩，但在原生层解码以提高性能；同时将Spark物理计划转换为原生执行）。

Spark选项	默认值	描述
spark.sql.iceberg.check-nullability	true	验证写入模式的可空性与表的可空性匹配。 ● 设置为true时，如果写入数据中声明为NOT NULL的字段包含null值将抛出异常。 ● 设置为false时跳过此检查，允许null值写入NOT NULL字段（可能导致后续查询出现问题）。
spark.sql.iceberg.check-ordering	true	验证写入模式列顺序与表模式顺序匹配。 ● 设置为true时，如果写入数据的列顺序与表定义不一致将抛出异常。 ● 设置为false时跳过此检查，允许按名称匹配列而非按位置匹配。
spark.sql.iceberg.planning.preserve-data-grouping	false	● 为true时，将同一分区的扫描任务放在同一读取分片中，用于存储分区连接（Storage Partitioned Joins），可优化分区内数据的局部性。 ● 设置为false时，扫描任务按默认策略分配，不保证同一分区的数据在同一分片中。
spark.sql.iceberg.aggregate-push-down.enabled	true	启用聚合函数（MAX、MIN、COUNT）的下推。 ● 设置为true时，Iceberg会尝试将聚合计算推入数据文件读取层，利用文件级统计信息（如min/max值）跳过不必要的数据读取。 ● 设置为false时，所有聚合计算在Spark层完成。

Spark选项	默认值	描述
spark.sql.iceberg.distribution-mode	动态决定： • 表有排序顺序时为range • 表有分区时为hash • 否则为none	控制写入期间的数据分布策略。可选值： • none：不重新分布数据。 • hash：按分区键进行Hash分布。适合数据在分区中均匀分布的场景 • range：按分区键或排序键进行Range分布。适合数据分布倾斜的场景 详细参考**1.3.4 向Iceberg表中写入数据**。
spark.wap.id	null	写-审计-发布（WAP）快照暂存ID。 设置后，新快照将以staged状态创建并关联此ID，需手动发布。设置为null时快照直接提交。前提需要设置表属性write.wap.enabled=true。
spark.wap.branch	null	WAP分支名称。设置后，快照将提交到指定分支而非主分支，用于隔离开发/测试环境。设置为null时快照提交到当前分支。 前提需要设置表属性write.wap.enabled=true。
spark.sql.iceberg.compression-codec	继承表属性（write.parquet.compression-codec默认gzip；write.avro.compression-codec默认gzip；write.orc.compression-codec默认zlib）	写入压缩编解码器。不同文件格式支持不同的编解码器： • Parquet支持uncompressed、snappy、gzip、lz4、zstd、brotli、lzop、bz2、zstd-no-seek。 • Avro支持null（无压缩）、deflate、snappy、bzip2、xz、lz4、zstd。 • ORC支持none、snappy、zlib、lz4、lzo、zstd。

Spark选项	默认值	描述
spark.sql.iceberg.compression-level	继承表属性（write.parquet.compression-level默认null；write.avro.compression-level默认null）	Parquet/Avro的压缩级别。通常为1-9的整数，数值越高压缩率越高但速度越慢。默认null表示使用编解码器的默认级别。仅对支持级别的编解码器（如gzip、zstd、deflate）有效。
spark.sql.iceberg.compression-strategy	继承表属性write.orc.compression-strategy（默认speed）	ORC的压缩策略。可选值： ● speed（默认值）：优先压缩速度，减少CPU占用。 ● compression：优先压缩率，减少存储空间。
spark.sql.iceberg.data-planning-mode	继承表属性read.data-planning-mode（默认AUTO）	数据文件的扫描规划模式。可选值： ● AUTO（默认值）：根据manifest文件数量自动选择local或distributed。 ● LOCAL：所有文件规划在Driver进程中执行。 ● DISTRIBUTED：文件规划分布到Spark Executor执行（要求spark.driver.maxResultSize>=256MB，否则自动回退到local）。
spark.sql.iceberg.delete-planning-mode	继承表属性read.delete-planning-mode（默认AUTO）	删除文件的扫描规划模式。可选值： ● AUTO（默认值）：自动选择。 ● LOCAL：Driver端规划。 ● DISTRIBUTED：分布式规划（要求spark.driver.maxResultSize>=256MB，否则自动回退到local）。
spark.sql.iceberg.advisory-partition-size	继承表属性write.target-file-size-bytes（默认536870912即512MB）	启用Spark自适应查询执行（AQE）时用于写入表的建议分区大小（字节）。用于调整输出文件大小，影响每个任务写入的数据量。

Spark选项	默认值	描述
spark.sql.iceberg.locality.enabled	false	报告Spark任务在执行器上的本地化信息。 ● 设置为true时，Iceberg会向Spark报告数据块的本地化位置，使Spark尽量将计算任务调度到存储数据的节点上，减少网络传输。对于HDFS等支持数据本地性的存储系统有效，对OBS/S3等对象存储无效。 ● 设置为false时，Spark随机分配任务到执行器。
spark.sql.iceberg.executor-cache.enabled	true	启用执行器端缓存。 ● 设置为true时，Iceberg在执行器端缓存删除文件（DeleteFiles）等内容，避免在Row级操作时重复读取。 ● 设置为false时，每次操作都从存储重新读取文件。
spark.sql.iceberg.executor-cache.timeout	10（10分钟）	缓存条目的超时时间（分钟）。超过此时间未访问的缓存条目将被清除。
spark.sql.iceberg.executor-cache.max-entry-size	67108864（64MB）	每个缓存条目的最大大小（字节）。超过此大小的文件将不会被缓存。
spark.sql.iceberg.executor-cache.max-total-size	134217728（128MB）	执行器缓存的最大总大小（字节）。当缓存总量超过此值时，将按LRU（最近最少使用）策略淘汰旧条目。
spark.sql.iceberg.executor-cache.locality.enabled	false	● 启用感知本地化的执行器缓存使用。 设置为true时，缓存优先在数据本地化的执行器上命中，减少跨节点缓存查找。 ● 设置为false时，缓存查找不考虑数据本地性。

Spark选项	默认值	描述
spark.sql.iceberg.merge-schema	false	启用修改表模式以匹配写入模式。 ● 设置为true时，如果写入数据包含表中不存在的新列，Iceberg会自动将这些列添加到表模式中。仅添加缺失的列，不会删除或修改已有列。 ● 设置为false时，如果写入数据包含表模式中不存在的列将抛出异常。
spark.sql.iceberg.report-column-stats	true	如果可用，向Spark的基于成本的优化器（CBO）报告Puffin表统计信息。 ● 设置为true时，Iceberg会将列统计信息（如NDV、null值计数）传递给SparkCBO用于优化查询计划。 ● 设置为false时不报告。必须同时启用SparkCBO（spark.sql.cbo.enabled=true）才能生效。

读取选项

Spark读取选项在配置DataFrame Reader时传递，如下所示：

```
//时间旅行
spark.read
.option("snapshot-id",10963874102873L)
.table("catalog.db.table")
```

Spark选项	默认值	描述
snapshot-id	null	要读取的表快照的快照ID（Long类型）。用于时间旅行（TimeTravel）功能，读取指定历史版本的数据。与as-of-timestamp互斥。设置为null时读取最新快照。

Spark选项	默认值	描述
as-of-timestamp	null	毫秒级时间戳（Long类型）。使用的快照将是此时间点之前最新的快照。用于基于时间的历史数据查询。与snapshot-id互斥，同时设置时优先使用snapshot-id。设置为null时读取最新快照。
split-size	继承表属性（read.split.target-size默认134217728即128MB；read.split.metadata-target-size默认33554432即32MB）	同时覆盖此表的read.split.target-size（数据文件分割目标大小）和read.split.metadata-target-size（元数据文件分割目标大小）。控制每个Spark任务处理的数据量。值越小产生的任务越多。
lookback	继承表属性read.split.planning-lookback（默认10）	覆盖此表的read.split.planning-lookback。在规划splits时向前回溯的文件数量，用于将小文件合并到同一个split中。值越大合并越多但规划时间越长。
file-open-cost	继承表属性read.split.open-file-cost（默认4194304即4MB）	覆盖此表的read.split.open-file-cost。打开一个文件的估算成本（字节），用于split规划时的成本优化。
vectorization-enabled	继承表属性（Parquet的read.parquet.vectorization.enabled默认true；ORC的read.orc.vectorization.enabled默认false）	覆盖此表的向量化读取开关。设置为true启用Parquet/ORC向量化读取，使用列式批量处理提升查询性能。设置为false时使用非向量化的逐行读取器，性能较低但兼容性更好。
batch-size	继承表属性（read.parquet.vectorization.batch-size默认5000；read.orc.vectorization.batch-size默认5000）	覆盖此表的向量化批次大小。即每批处理的行数。较大的批次（如10000）提高吞吐量但增加内存消耗；较小的批次（如1000）减少内存占用但降低吞吐量。

Spark选项	默认值	描述
stream-from-timestamp	null	毫秒级时间戳（Long类型），用于流式传输的起始点。如果早于最旧的已知祖先快照，则使用最旧的快照。设置为null时从最新快照开始流式读取。
streaming-max-files-per-micro-batch	Integer.MAX_VALUE（2147483647）	每个微批次的最大文件数。用于控制结构化流（StructuredStreaming）中每个批次的处理文件数量。设置为较小值可避免批次过大导致延迟增加。
streaming-max-rows-per-micro-batch	Integer.MAX_VALUE（2147483647）	每个微批次的最大行数（软限制）。

说明

streaming-max-rows-per-micro-batch选项设置了一个"软最大值"，一个批次将始终包含下一个未处理数据文件中的所有行，但如果包含额外文件会超过软最大值限制，则不会包含这些文件。

写入选项

Spark写入选项在配置DataFrameWriterV2时传递，如下所示：

```
//使用Avro而不是Parquet写入
df.writeTo("catalog.db.table")
.option("write-format","avro")
.option("snapshot-property.key","value")
.append()
```

Spark选项	默认值	描述
write-format	继承表属性write.format.default（默认parquet）	此写入操作使用的文件格式。可选值： • parquet（默认） • avro • orc
target-file-size-bytes	继承表属性write.target-file-size-bytes（默认536870912即512MB）	覆盖此表的write.target-file-size-bytes。控制每个输出数据文件的目标大小。较小的值产生更多小文件（查询时可能增加扫描开销），较大的值减少文件数但增加单文件大小（可能增加内存消耗）。

Spark选项	默认值	描述
check-nullability	true	设置字段的可空性检查。设置为true时，如果写入数据中声明为NOTNULL的字段包含null值将抛出异常。设置为false时跳过此检查，允许null值写入NOTNULL字段（可能导致后续查询行为异常）。
snapshot-property.*custom-key*	null	在快照摘要中添加具有自定义键和相应值的条目。例如snapshot-property.user=alice会在快照元数据中添加{"user":"alice"}。用于审计追踪或变更标记。（DSv2API需要snapshot-property.前缀）。
fanout-enabled	false	覆盖此表的write.spark.fanout.enabled。 ● 设置为true时，将数据按分区扇出写入，避免单个任务写入过多分区（每个分区一个写入任务），适合写入大量分区的场景，但会增加Shuffle开销和内存消耗。 ● 设置为false时，单个任务写入多个分区，可能在写入大量分区时产生内存压力。
check-ordering	true	检查输入模式和表模式是否相同。 ● 设置为true时，如果写入数据的列顺序与表定义不一致将抛出异常。 ● 设置为false时跳过此检查，允许按名称匹配列而非按位置匹配。

Spark选项	默认值	描述
isolation-level	null	Dataframe覆盖操作所需的隔离级别。可选值： ● null（默认）：不进行检查，适合幂等写入。 ● serializable：检查目标分区中的并发插入或删除，确保完全隔离。 ● snapshot：仅检查目标分区中的并发删除，允许并发插入。 表属性默认： ● write.delete.isolation-level默认serializable。 ● write.update.isolation-level默认serializable。 ● write.merge.isolation-level默认serializable。
validate-from-snapshot-id	null	如果设置了隔离级别，则为检查表中并发写入冲突的基线快照ID（Long类型）。应该是任何读取表之前的快照。可以通过Table API或Snapshots表获取。如果为null，则使用表的最旧已知快照。
compression-codec	继承表属性（Parquet的write.parquet.compression-codec默认gzip；Avro的write.avro.compression-codec默认gzip；ORC的write.orc.compression-codec默认zlib）	覆盖此表的写入压缩编解码器。 ● Parquet可选值：uncompressed、snappy、gzip、lz4、zstd、brotli、lzop、bz2、zstd-no-seek。 ● Avro可选值：null（无压缩）、deflate、snappy、bzip2、xz、lz4、zstd。 ● ORC可选值：none、snappy、zlib、lz4、lzo、zstd。

Spark选项	默认值	描述
compression-level	继承表属性（write.parquet.compression-level默认null；write.avro.compression-level默认null）	覆盖此表的Parquet和Avro表的写入压缩级别。整数类型，通常1-9，数值越高压缩率越高但速度越慢。默认null表示使用编解码器的默认级别。仅对支持级别的编解码器（如gzip、zstd、deflate）有效。
compression-strategy	继承表属性write.orc.compression-strategy（默认speed）	覆盖此表的ORC表的写入压缩策略。可选值： • speed（默认）：优先压缩速度，减少CPU占用。 • compression：优先压缩率，减少存储空间。
distribution-mode	动态决定（表有排序顺序为range，有分区为hash，否则none）	覆盖此表的写入分布模式。可选值： • none：不重新分布数据。 • hash：按分区键Hash分布。适合数据在分区中均匀分布的场景。 • range：按分区键或排序键Range分布。适合数据分布倾斜的场景。 表属性默认： write.delete.distribution-mode、write.update.distribution-mode、write.merge.distribution-mode默认均为hash 详细参考**1.3.4 向Iceberg表中写入数据**。

Spark选项	默认值	描述
delete-granularity	file	覆盖此表的写入删除粒度。可选值： • file（默认值）：按文件级别删除，生成positiondelete文件指向具体的数据文件行 • partition：按分区级别删除，生成partitiondelete文件标记整个分区为已删除（表属性write.delete.granularity默认值为partition） 影响DELETE操作的执行效率。file级别更精确但产生更多小文件；partition级别更高效但删除整个分区。

1.3.2 Iceberg on Spark DDL 语法说明

CREATE TABLE

Spark 可以在任何Iceberg Catalog中创建表，Iceberg会将Spark中的列类型转换为相应的Iceberg类型。命令如下：

```
CREATE DATABASE IF NOT EXISTS spark_catalog.db location 'obs://yourbucket/path/to/db';
CREATE TABLE spark_catalog.db.sample (id bigint NOT NULL COMMENT 'unique id', data string) USING iceberg;
```

创建表命令支持所有范围的Spark **create**子句，包括：

- PARTITIONED BY：配置分区。
- LOCATION '*(fully-qualified-uri)*'：设置表位置。
- COMMENT '*table documentation*'：设置表描述。
- TBLPROPERTIES ('key'='value', ...)：设置表配置，常见表属性参考**1.1.2 Iceberg表属性参数介绍**。

PARTITIONED BY

创建分区表的命令为：

```
CREATE TABLE spark_catalog.db.sample (id bigint, data string, category string) USING iceberg
PARTITIONED BY (category);
```

该子句还支持转换表达式来创建隐藏分区：

```
CREATE TABLE spark_catalog.db.sample (id bigint, data string, category string, ts timestamp) USING iceberg
PARTITIONED BY (bucket(16, id), days(ts), category);
```

转换表达式支持的转换范围包括：

- year(ts)：按年份进行分区。
- month(ts)：按月份进行分区。
- day(ts) or date(ts)：等同于dateint分区，即按日期整数格式分区。
- hour(ts) or date_hour(ts)：等同于dateint + 小时分区，即按日期整数格式与小时组合分区。
- bucket(*N*, *col*)：按列（col）的哈希值对N取模后的结果进行分桶分区。
- truncate(*L*, *col*)：按列（col）值截断至指定长度（L）后进行分区，具体规则如下：
  - 字符串类型：截断为指定长度（L）的字符串。
  - 整数（Integer）与长整数（Long）类型：按区间截断，例如truncate (10, i)会生成 0、10、20、30……分区区间。

CTAS 语法

当使用SparkCatalog时，Iceberg支持将CTAS（CREATE TABLE ... AS SELECT）作为原子操作执行。Iceberg同样支持在SparkSessionCatalog中执行CTAS操作，但该场景下的CTAS不具备原子性。

```
CREATE TABLE spark_catalog.db.sample USING iceberg AS SELECT ...;
```

通过CTAS操作生成的新表，不会从SELECT语句所指定的源表中集成分区规范和表属性。如果需要为该新建表配置分区规范或表属性，可在CTAS语句中通过PARTITIONED BY子句明确声明分区规范，通过TBLPROPERTIES子句自定义表属性。命令为：

```
CREATE TABLE spark_catalog.db.sample USING iceberg PARTITIONED BY (part) TBLPROPERTIES
('key'='value') AS SELECT ...;
```

RTAS 语法

当使用SparkCatalog时，Iceberg支持将RTAS（REPLACE TABLE ... AS SELECT）作为原子操作执行。在SparkSessionCatalog中同样支持RTAS操作，但该场景下的RTAS不具备原子性。

原子性表替换会基于SELECT查询的结果创建新快照，同时保留表的历史记录。使用方法为：

- 方式一：
  ```
  REPLACE TABLE spark_catalog.db.sample USING iceberg AS SELECT ...;
  ```
- 方式二：
  ```
  REPLACE TABLE spark_catalog.db.sample USING iceberg PARTITIONED BY (part) TBLPROPERTIES
  ('key'='value') AS SELECT ...;
  ```
- 方式三：
  ```
  CREATE OR REPLACE TABLE spark_catalog.db.sample USING iceberg AS SELECT ...;
  ```

DROP TABLE

Iceberg的**DROP TABLE**操作行为在0.14版本发生变更，具体差异如下：

- 0.14之前版本：执行**DROP TABLE**会从目录（Catalog）中移除表的元数据，并且删除表的实际数据内容。
- 0.14及之后版本：**DROP TABLE**仅会从目录中移除表的元数据；如果需要同时删除表的实际数据内容，需使用**DROP TABLE PURGE**命令。

  - **DROP TABLE**（仅移除元数据）
    ```
    DROP TABLE spark_catalog.db.sample;
    ```
  - **DROP TABLE PURGE**（移除元数据并删除数据）
    ```
    DROP TABLE spark_catalog.db.sample PURGE;
    ```

ALTER TABLE

在Spark 中，Iceberg完全支持**ALTER TABLE**操作，具体包括：

- 重命名表（ALTER TABLE ... RENAME TO）（当前限制，暂不支持）。
  ```
  ALTER TABLE spark_catalog.db.sample RENAME TO spark_catalog.db.new_name;
  ```
- 设置与移除表属性（ALTER TABLE ... SET/UNSET TBLPROPERTIES）：
  - 设置表属性：
    ```
    ALTER TABLE spark_catalog.db.sample SET TBLPROPERTIES ('read.split.target-size'='268435456');
    ```
    SET TBLPROPERTIES命令也可用于设置表的注释（描述信息）：
    ```
    ALTER TABLE spark_catalog.db.sample SET TBLPROPERTIES ( 'comment' = 'A table comment.');
    ```
  - 移除表属性，可使用UNSET命令：
    ```
    ALTER TABLE spark_catalog.db.sample UNSET TBLPROPERTIES ('read.split.target-size');
    ```
- 新增、删除和重命名列
  - 新增列
    - 可使用ALTER TABLE语句的ADD COLUMNS子句向Iceberg表添加列，且不允许通过添加列的方式修改映射（map）的 “键（key）” 列，仅能更新映射的值（value）。示例如下：
      - 向“spark_catalog.db.sample”表中添加struct类型的“point”列：
        ```
        ALTER TABLE spark_catalog.db.sample ADD COLUMN point struct<x: double, y: double>;
        ```
      - 向“spark_catalog.db.sample”表中添加double类型的“point.z”列：
        ```
        ALTER TABLE spark_catalog.db.sample ADD COLUMN point.z double;
        ```
      - 向“spark_catalog.db.sample”表中添加array类型的“points”列：
        ```
        ALTER TABLE spark_catalog.db.sample ADD COLUMN points array<struct<x: double, y: double>>;
        ```
      - 向“spark_catalog.db.sample”表中添加double类型的“points.element.z”列：
        ```
        ALTER TABLE spark_catalog.db.sample ADD COLUMN points.element.z double;
        ```
      - 向“spark_catalog.db.sample”表中添加map类型的“points”列：
        ```
        ALTER TABLE spark_catalog.db.sample ADD COLUMN points map<struct<x: int>, struct<a: int>>;
        ```
    - 也可通过添加FIRST或AFTER子句，在表中的任意位置添加列，示例如下：
      - 示例一：
        ```
        ALTER TABLE spark_catalog.db.sample ADD COLUMN new_column bigint AFTER {表中已有字段名（例如id，data等）};
        ```
      - 示例二：
        ```
        ALTER TABLE spark_catalog.db.sample ADD COLUMN new_column bigint FIRST;
        ```
  - 重命名列（ALTER TABLE ... RENAME COLUMN）

Iceberg允许对任意字段进行重命名。如果需要重命名字段，可使用**RENAME COLUMN**子句：

```
ALTER TABLE spark_catalog.db.sample RENAME COLUMN data TO payload;
```

- 可使用**ALTER TABLE ... DROP COLUMN**语句删除列：

```
ALTER TABLE spark_catalog.db.sample DROP COLUMN id;
```

- 新增、删除和重命名嵌套字段

  可使用**ADD COLUMN**或**DROP COLUMN**为结构体（struct）添加或删除列。

- 调整顶层列与嵌套结构体字段的顺序

  Iceberg允许使用FIRST和AFTER子句调整顶层列或结构体中的列顺序：

```
ALTER TABLE spark_catalog.db.sample ALTER COLUMN id FIRST;
```

- 拓宽int、float、decimal字段的类型（ALTER TABLE ... ALTER COLUMN）

  ALTER COLUMN用于拓宽字段类型，将字段设为可选项、设置注释以及调整字段顺序。

  Iceberg允许在类型更新安全的前提下修改列类型，安全的列类型更新包括：

  - int到bigint
  - float到double
  - decimal (P,S)到decimal (P2,S)，其中P2 > P，即精度可提高，小数位数不可变更。

  例如，执行以下命令将“measurement”列的类型变更为“double”：

```
ALTER TABLE spark_catalog.db.sample ALTER COLUMN measurement TYPE double;
```

  也可使用**ALTER COLUMN**更新列注释：

```
ALTER TABLE spark_catalog.db.sample ALTER COLUMN {double型字段} COMMENT 'unit is kilobytes per second';
```

- 非空列（required column）改为可空列（optional column）

  可通过**DROP NOT NULL**更改非空列的可空性：

```
ALTER TABLE spark_catalog.db.sample ALTER COLUMN id DROP NOT NULL;
```

- 通过启用SQL扩展，还可支持分区演进与表写入顺序设置功能。

Iceberg SQL 扩展 DDL 语法

在Spark 中使用Iceberg SQL扩展时，可使用以下命令：

```
ALTER TABLE ... ADD PARTITION FIELD;
```

- ADD PARTITION FIELD

  Iceberg支持通过**ADD PARTITION FIELD**向分区规范中添加新的分区字段：

```
ALTER TABLE spark_catalog.db.sample ADD PARTITION FIELD id;
```

  同时支持分区转换操作：

  - 示例一：

```
ALTER TABLE spark_catalog.db.sample ADD PARTITION FIELD bucket(16, id);
```

  - 示例二：

```
ALTER TABLE spark_catalog.db.sample ADD PARTITION FIELD truncate(4, data);
```

  - 示例三：

```
ALTER TABLE spark_catalog.db.sample ADD PARTITION FIELD year(ts);
```

  - 示例四：

```
ALTER TABLE spark_catalog.db.sample ADD PARTITION FIELD day(ts);
```

  添加分区字段属于元数据操作，不会更改任何现有表数据。新数据将按照新的分区方式写入，但现有数据仍保持原有的分区布局。在元数据表中，旧数据文件的新分区字段值将为null。

当表的分区方式发生变化时，动态分区覆盖行为也会随之改变，因为动态覆盖会隐式替换分区。如果需要显式执行覆盖操作，请使用新的DataFrameWriterV2 API。

- DROP PARTITION FIELD

  可使用**DROP PARTITION FIELD**移除分区字段。即使分区被移除，对应的列仍会保留在表的Schema 中。

  删除分区字段属于元数据操作，不会更改任何现有表数据。新数据将按照新的分区方式写入，但现有数据仍保持原有的分区布局。

  - 创建Iceberg表，例如：
    ```
    CREATE TABLE spark_catalog.db.sample (id bigint, data string, category string, ts timestamp, shard int) USING iceberg PARTITIONED BY (category);
    ```
  - 移除分区字段：
    - 示例一：
      ```
      ALTER TABLE spark_catalog.db.sample DROP PARTITION FIELD category;
      ```
    - 示例二：
      ```
      ALTER TABLE spark_catalog.db.sample DROP PARTITION FIELD bucket(16, id);
      ```
    - 示例三：
      ```
      ALTER TABLE spark_catalog.db.sample DROP PARTITION FIELD truncate(4, data);
      ```
    - 示例四：
      ```
      ALTER TABLE spark_catalog.db.sample DROP PARTITION FIELD year(ts);
      ```
    - 示例五：
      ```
      ALTER TABLE spark_catalog.db.sample DROP PARTITION FIELD shard;
      ```

- REPLACE PARTITION FIELD

  通过**REPLACE PARTITION FIELD**可在单次元数据更新中，用新的分区字段替换原有分区字段：

  - 示例一：
    ```
    ALTER TABLE spark_catalog.db.sample REPLACE PARTITION FIELD ts_day WITH day(ts);
    ```
  - 示例二：
    ```
    ALTER TABLE spark_catalog.db.sample REPLACE PARTITION FIELD ts_day WITH day(ts) AS day_of_ts;
    ```

- WRITE ORDERED BY

  Iceberg表可配置排序规则，部分引擎会依据该规则，在向表写入数据时自动对数据进行排序。例如，Spark中的MERGE INTO操作会使用表的排序规则。

  可使用**WRITE ORDERED BY**子句为表设置写入排序规则，示例如下：

  - 示例一：
    ```
    ALTER TABLE spark_catalog.db.sample WRITE ORDERED BY category, id;
    ```
  - 示例二：
    ```
    ALTER TABLE spark_catalog.db.sample WRITE ORDERED BY category ASC, id DESC;
    ```
  - 示例三：
    ```
    ALTER TABLE spark_catalog.db.sample WRITE ORDERED BY category ASC NULLS LAST, id DESC NULLS FIRST;
    ```

  WRITE ORDERED BY用于设置全局排序规则，会对跨任务的行进行排序，类似于在INSERT命令中使用ORDER BY，例如：
  ```
  INSERT INTO spark_catalog.db.sample SELECT id, data, category, ts FROM {其他表} ORDER BY ts, category;
  ```

  若只需在每个任务内排序（而非跨任务排序），可使用**LOCALLY ORDERED BY**：
  ```
  ALTER TABLE spark_catalog.db.sample WRITE LOCALLY ORDERED BY category, id;
  ```

可使用**UNORDERED**取消表的排序顺序：

```
ALTER TABLE spark_catalog.db.sample WRITE UNORDERED;
```

- WRITE DISTRIBUTED BY PARTITION

  WRITE DISTRIBUTED BY PARTITION用于指定每个分区由单个写入器（writer）处理，其默认实现方式为哈希分布。

  ```
  ALTER TABLE spark_catalog.db.sample WRITE DISTRIBUTED BY PARTITION;
  ```

  DISTRIBUTED BY PARTITION可与**LOCALLY ORDERED BY**结合使用，实现按分区分布数据，同时在每个任务内对行进行本地排序：

  ```
  ALTER TABLE spark_catalog.db.sample WRITE DISTRIBUTED BY PARTITION LOCALLY ORDERED BY category, id;
  ```

- SET IDENTIFIER FIELDS

  Iceberg支持通过**SET IDENTIFIER FIELDS**为表规范设置标识字段。如果Spark表配置了标识字段，该表可支持Spark SQL的更新插入（upsert）操作。

  - 示例一，需确保字段无空值：
    ```
    ALTER TABLE spark_catalog.db.sample SET IDENTIFIER FIELDS id;
    ```
  - 示例二：
    ```
    ALTER TABLE spark_catalog.db.sample SET IDENTIFIER FIELDS id, data;
    ```

  标识字段在创建或添加时，必须是非空列。后续执行的**ALTER TABLE ... SET IDENTIFIER FIELDS**语句会覆盖之前的标识字段设置。

- DROP IDENTIFIER FIELDS

  可使用**DROP IDENTIFIER FIELDS**移除标识字段，且即使标识字段被移除，对应的列仍会保留在表的Schema中。

  - 示例一，需确保字段无空值：
    ```
    ALTER TABLE spark_catalog.db.sample DROP IDENTIFIER FIELDS id;
    ```
  - 示例二：
    ```
    ALTER TABLE spark_catalog.db.sample DROP IDENTIFIER FIELDS id, data;
    ```

Branch 和 Tag DDL 语法

- CREATE BRANCH

  可通过**CREATE BRANCH**语句创建分支，该语句支持以下选项：

  - IF NOT EXISTS：如果分支已存在，操作不会失败。
  - CREATE OR REPLACE：如果分支已存在，则更新该分支。
  - 在特定快照处创建分支。
  - 创建具有指定保留期的分支：
    - 示例一：
      ```
      ALTER TABLE spark_catalog.db.sample CREATE BRANCH `audit-branch`;
      ```
    - 示例二：
      ```
      ALTER TABLE spark_catalog.db.sample CREATE BRANCH IF NOT EXISTS `audit-branch`;
      ```
    - 示例三：
      ```
      ALTER TABLE spark_catalog.db.sample CREATE OR REPLACE BRANCH `audit-branch`;
      ```

- CREATE TAG

  可通过**CREATE TAG**语句创建标签，该语句支持以下选项：

  - IF NOT EXISTS：如果标签已存在，操作不会失败。
  - CREATE OR REPLACE：如果标签已存在，则更新该标签。

  - 在特定快照（snapshot）处创建标签。
  - 创建具有指定保留期的标签：
    - 示例一：
      ```
      ALTER TABLE spark_catalog.db.sample CREATE TAG `historical-tag`;
      ```
    - 示例二：
      ```
      ALTER TABLE spark_catalog.db.sample CREATE TAG IF NOT EXISTS `historical-tag`;
      ```
    - 示例三：
      ```
      ALTER TABLE spark_catalog.db.sample CREATE OR REPLACE TAG `historical-tag`;
      ```
- REPLACE BRANCH

  可通过**REPLACE BRANCH**语句更新分支所引用的快照，同时也可更新其保留期：

  ```
  ALTER TABLE spark_catalog.db.sample REPLACE BRANCH `audit-branch` AS OF VERSION {current-snapshot-id} RETAIN 60 DAYS;
  ```
- REPLACE TAG

  可通过**REPLACE TAG**语句更新标签所引用的快照，同时也可更新其保留期：

  ```
  ALTER TABLE spark_catalog.db.sample REPLACE TAG `historical-tag` AS OF {current-snapshot-id} RETAIN 60 DAYS;
  ```
- DROP BRANCH

  可通过**DROP BRANCH**语句删除分支：

  ```
  ALTER TABLE spark_catalog.db.sample DROP BRANCH `audit-branch`;
  ```
- DROP TAG

  可通过**DROP TAG**语句删除标签：

  ```
  ALTER TABLE spark_catalog.db.sample DROP TAG `historical-tag`;
  ```

Iceberg 视图 DDL 语法

Iceberg视图的设计目标是提供引擎无关的视图定义，确保同一视图可以在不同的查询引擎中保持一致的行为和结果，增强了数据查询的灵活性和兼容性。

Iceberg视图是SQL视图的一种通用表示形式，旨在跨多个查询引擎进行解析。以下内容介绍如何在Spark中创建和管理视图。

- 创建视图

  创建一个不包含任何注释或属性的简单视图：

  ```
  CREATE VIEW <viewName> AS SELECT * FROM <tableName>;
  ```

  使用**IF NOT EXISTS**可避免因视图已存在而导致语句执行失败：

  ```
  CREATE VIEW IF NOT EXISTS <viewName> AS SELECT * FROM <tableName>;
  ```

  创建包含注释的视图，其中可包含与源表不同的列别名和列注释：

  ```
  CREATE VIEW <viewName> (ID COMMENT 'Unique ID', ZIP COMMENT 'Zipcode')  COMMENT 'View Comment' AS SELECT id, zip FROM <tableName>;
  ```
- 创建包含属性的视图：

  使用**TBLPROPERTIES**创建包含属性的视图：

  ```
  CREATE VIEW <viewName> TBLPROPERTIES ('key1' = 'val1', 'key2' = 'val2') AS SELECT * FROM <tableName>;
  ```

  查看视图的属性：

  ```
  SHOW TBLPROPERTIES <viewName>;
  ```
- 删除视图

  - 删除已存在的视图：
    ```
    DROP VIEW <viewName>;
    ```
  - 如果视图不存在时不希望语句执行失败，可使用**IF EXISTS**关键字：
    ```
    DROP VIEW IF EXISTS <viewName>;
    ```
- 更新视图

  如果需要更新视图的Schema、属性或底层SQL语句，可使用**CREATE OR REPLACE**语句覆盖原有视图：

  ```
  CREATE OR REPLACE VIEW <viewName> (updated_id COMMENT 'updated ID') TBLPROPERTIES ('key1' = 'new_val1') AS SELECT id FROM <tableName>;
  ```
- 设置或移除视图属性
  - 使用**ALTER VIEW ... SET TBLPROPERTIES**为已存在的视图设置属性：
    ```
    ALTER VIEW <viewName> SET TBLPROPERTIES ('key1' = 'val1', 'key2' = 'val2');
    ```
  - 使用**ALTER VIEW ... UNSET TBLPROPERTIES**从已存在的视图中移除属性：
    ```
    ALTER VIEW <viewName> UNSET TBLPROPERTIES ('key1', 'key2');
    ```
- 查看视图

  查看当前已设置的命名空间（通过**USE** *<namespace>*设置）中的所有视图：

  ```
  SHOW VIEWS;
  ```

  查看指定目录或命名空间中的所有可用视图：
  - 查看指定目录（catalog）中的所有视图：
    ```
    SHOW VIEWS IN <catalog>;
    ```
  - 查看指定命名空间（namespace）中的所有视图：
    ```
    SHOW VIEWS IN <namespace>;
    ```
  - 查看指定目录和命名空间组合中的所有视图：
    ```
    SHOW VIEWS IN <catalog>.<namespace>;
    ```
- 查看创建视图的语句：
  ```
  SHOW CREATE TABLE <viewName>;
  ```
- 使用**DESCRIBE**查看视图详情：
  ```
  DESCRIBE [EXTENDED] <viewName>;
  ```

1.3.3 查询 Iceberg 表数据

在Spark中使用Iceberg前，需先配置Spark Catalog，配置请参见**1.3.1 Iceberg配置与类型**。Iceberg基于Apache Spark的DataSourceV2 API实现数据源和Catalog功能。

查询表数据

在Spark中，表的标识符需包含catalog名称。查询表数据操作为：

创建表，例如：

```
CREATE TABLE spark_catalog.db.table (id bigint, data string, category string, ts timestamp, shard int) USING iceberg PARTITIONED BY (category);
```

- 查询表数据：
  ```
  SELECT * FROM spark_catalog.db.table;
  ```
- 对于历史记录、snapshots快照等元数据表，可将Iceberg表名作为命名空间来访问，例如：
  ```
  SELECT * FROM spark_catalog.db.table.files;
  ```

将表加载为 DataFrame

如果需要将表加载为DataFrame，可使用spark.table方法：

```
//spark-shell
val df = spark.table("spark_catalog.db.table")
```

结合 DataFrameReader 使用 Catalog

可通过Spark的DataFrameReader接口加载路径和表名，表的加载方式取决于标识符的指定形式。当使用**spark.read.format("iceberg").load(*table*)**或**spark.table(*table*)**时，*table*支持以下多种格式，且优先级按以下顺序排列（例如，匹配到目录时，优先级高于任何命名空间解析）：

- tablename：从当前目录的当前命名空间中加载表，即currentCatalog.currentNamespace.tablename。
- catalog.tablename：从指定目录中加载表。
- namespace.tablename：从当前目录中加载指定命名空间下的表。
- catalog.namespace.tablename：从指定目录中加载指定命名空间下的表。
- namespace1.namespace2.tablename：从当前目录中加载多级命名空间下的表。

上述列表按优先级从高到低排序，例如，如果标识符可匹配到目录，则优先按目录规则加载表，而非进行命名空间解析。

时间旅行

- SQL命令

  Spark支持在SQL查询中使用**TIMESTAMP AS OF**或**VERSION AS OF**子句实现时间旅行。其中，**VERSION AS OF**子句可接受长整数类型的快照 ID，或字符串类型的分支/标签名称。

  - 时间旅行至1986年10月26日01:21:00：
    ```
    SELECT * FROM spark_catalog.db.table TIMESTAMP AS OF '1986-10-26 01:21:00';
    ```
  - 时间旅行至快照ID 10963874102873：
    ```
    SELECT * FROM spark_catalog.db.table VERSION AS OF 10963874102873;
    ```
    查询current-snapshot-id快照ID的命令为：
    ```
    DESCRIBE EXTENDED spark_catalog.db.table;
    ```
  - 时间旅行至“audit-branch”的头部快照：
    ```
    SELECT * FROM spark_catalog.db.table VERSION AS OF 'audit-branch';
    ```
  - 时间旅行至标签“historical-snapshot”所引用的快照：
    ```
    SELECT * FROM spark_catalog.db.table VERSION AS OF 'historical-snapshot';
    ```

  还支持**FOR SYSTEM_TIME AS OF**和**FOR SYSTEM_VERSION AS OF**语法，功能与**TIMESTAMP AS OF**和**VERSION AS OF**一致，例如：

  - 示例一：
    ```
    SELECT * FROM spark_catalog.db.table FOR SYSTEM_TIME AS OF '1986-10-26 01:21:00';
    ```
  - 示例二：
    ```
    SELECT * FROM spark_catalog.db.table FOR SYSTEM_VERSION AS OF 10963874102873;
    ```
  - 示例三：
    ```
    SELECT * FROM spark_catalog.db.table FOR SYSTEM_VERSION AS OF 'audit-branch';
    ```
  - 示例四：
    ```
    SELECT * FROM spark_catalog.db.table FOR SYSTEM_VERSION AS OF 'historical-snapshot';
    ```

  时间戳也可使用Unix时间戳（秒级）传入，例如：

  ```
  SELECT * FROM spark_catalog.db.table TIMESTAMP AS OF 499162860;
  ```

  或可使用：

  ```
  SELECT * FROM spark_catalog.db.table FOR SYSTEM_TIME AS OF 499162860;
  ```

分支或标签也可通过类似元数据表的语法指定，格式为“branch_*<分支名>*”或“tag_*<标签名>*”：

```
SELECT * FROM spark_catalog.db.table.`branch_audit-branch`;
```

或可使用：

```
SELECT * FROM spark_catalog.db.table.`tag_historical-snapshot`;
```

含-等特殊字符的标识符不合法，必须使用反引号（`）转义。 通过“branch_*<分支名>*”或“tag_*<标签名>*”定位的分支或标签，不可与**VERSION AS OF**子句同时使用 。

- 时间旅行查询中的Schema选择

  各类时间旅行查询可根据定位目标的不同，使用快照自身的Schema或当前表的Schema，具体规则如下：

  - 使用快照自身的Schema：
    - 示例一：
      ```
      SELECT * FROM spark_catalog.db.table TIMESTAMP AS OF '1986-10-26 01:21:00';
      ```
    - 示例二：
      ```
      SELECT * FROM spark_catalog.db.table VERSION AS OF 10963874102873;
      ```
  - 使用当前表的Schema：
    - 示例一：
      ```
      SELECT * FROM spark_catalog.db.table VERSION AS OF 'audit-branch';
      ```
    - 示例二：
      ```
      SELECT * FROM spark_catalog.db.table.`branch_audit-branch`;
      ```
  - 使用快照自身的Schema：
    - 示例一：
      ```
      SELECT * FROM spark_catalog.db.table VERSION AS OF 'historical-snapshot';
      ```
    - 示例二：
      ```
      SELECT * FROM spark_catalog.db.table.`tag_historical-snapshot`;
      ```

- DataFrame

  在DataFrame API中，如果需要选择表的特定快照或某个时间点的快照，Iceberg支持以下四种Spark读取选项，用于定位目标快照：

  - snapshot-id：选择特定快照，需传入快照的ID。
  - as-of-timestamp：选择指定时间点的当前快照，需传入毫秒级Unix时间戳。
  - branch：选择指定分支的头部快照。
  - tag：选择指定标签关联的快照。

  例如：

  - 时间旅行至1986年10月26日01:21:00：
    ```
    spark.read.option("as-of-timestamp", "499162860000").format("iceberg").load("path/to/table")
    ```
  - 时间旅行至快照ID 10963874102873：
    ```
    spark.read.option("snapshot-id", 10963874102873L).format("iceberg").load("path/to/table")
    ```
  - 时间旅行至标签“historical-snapshot”所引用的快照：
    ```
    spark.read.option(SparkReadOptions.TAG, "historical-snapshot").format("iceberg").load("path/to/table")
    ```
  - 时间旅行至“audit-branch”的头部快照：
    ```
    spark.read.option(SparkReadOptions.BRANCH, "audit-branch").format("iceberg").load("path/to/table")
    ```

- 增量读取

如果需要增量读取表中追加的数据，可通过配置以下两个关键读取选项实现：

- start-snapshot-id：增量扫描的起始快照ID（排他性，即不包含此快照的数据）。
- end-snapshot-id：增量扫描的结束快照ID（包含性，即包含此快照的数据），此选项为可选。如果不指定，默认使用表的当前快照。

```
spark.read.format("iceberg").option("start-snapshot-id", "10963874102873").option("end-snapshot-id", "63874143573109").load("path/to/table")
```

表检查

如果需要查看表的历史记录、快照及其他元数据，Iceberg支持通过元数据表实现。

元数据表的标识方式为在原表名后拼接元数据表名称。例如，如果需要读取*db.table*表的历史记录，可通过*db.table*.**history**访问。注意需要使用三级或二级命名空间，即“*spark_catalog.db.table*.history”或“*db.table*.history”，不支持使用一级命名空间，即“*table*.history”。

- 历史记录

  可通过以下语句查看表的历史记录：

  ```
  SELECT * FROM spark_catalog.db.table.history;
  ```

- 元数据日志条目

  如果需要查看表的元数据日志条目，可通过以下语句查询：

  ```
  SELECT * from spark_catalog.db.table.metadata_log_entries;
  ```

- 快照

  如果需要查看表的所有有效快照，可通过以下语句查询：

  ```
  SELECT * FROM spark_catalog.db.table.snapshots;
  ```

- 条目

  如果需要查看表当前所有清单文件中数据文件和删除文件的条目，可通过以下语句查询：

  ```
  SELECT * FROM spark_catalog.db.table.entries;
  ```

- 文件

  如果需要查看表当前的所有数据文件，可通过以下语句查询：

  ```
  SELECT * FROM spark_catalog.db.table.files;
  ```

- 清单文件

  如果需要查看表当前的文件清单，可通过以下语句查询：

  ```
  SELECT * FROM spark_catalog.db.table.manifests;
  ```

  查询结果中的“partition_summaries”列中的字段对应清单列表中的field_summary结构体，按以下顺序排列：

  - contains_null
  - contains_nan
  - lower_bound
  - upper_bound

  “contains_nan”可能返回“null”，表示该信息在文件元数据中不可用，这通常发生在读取V1版本的表（V1版本不填充contains_nan信息）。

- 分区

  如果需要查看表当前的分区信息，可通过以下语句查询：

  ```
  SELECT * FROM spark_catalog.db.table.partitions;
  ```


对于未分区表，partitions表将不包含**partition**和**spec_id**字段。

partitions元数据表展示的是当前快照中包含数据文件或删除文件的分区。但需注意，删除文件并未实际应用，因此在某些情况下，即使分区的所有数据行都被删除文件标记为删除，该分区仍可能会被显示。

- 位置删除文件

  如果需要查看表当前快照中所有的位置删除文件，可通过以下语句查询：

  ```
  SELECT * from spark_catalog.db.table.position_deletes;
  ```

- 所有数据文件

  如果需要查看表的所有数据文件及其各自的元数据，可通过以下语句查询：

  ```
  SELECT * FROM spark_catalog.db.table.all_data_files;
  ```

- 所有删除文件

  如果需要查看表在所有快照中的删除文件及其各自的元数据，可通过以下语句查询：

  ```
  SELECT * FROM spark_catalog.db.table.all_delete_files;
  ```

- 所有条目

  如果需要查看表在所有快照中数据文件和删除文件的清单条目，可通过以下语句查询：

  ```
  SELECT * FROM spark_catalog.db.table.all_entries;
  ```

- 所有清单文件

  如果需要查看表的所有清单文件，可通过以下语句查询：

  ```
  SELECT * FROM spark_catalog.db.table.all_manifests;
  ```

  查询结果中“partition_summaries”列中的字段对应清单列表中的field_summary结构体，按以下顺序排列：

  - contains_null
  - contains_nan
  - lower_bound
  - upper_bound

  “contains_nan”可能返回“null”，表示该信息在文件元数据中不可用，通常发生在读取V1版本的表（V1版本不填充contains_nan信息）。

- 引用

  如果需要查看表的已知快照引用，可通过以下语句查询：

  ```
  SELECT * FROM spark_catalog.db.table.refs;
  ```

- 通过DataFrame检查元数据

  可通过DataFrameReader API加载元数据表：

  - 示例一：
    ```
    spark.read.format("iceberg").load("db.table.files");
    ```
  - 示例二：
    ```
    spark.read.format("iceberg").load("obs://bucket/path/to/table#files");
    ```

- 元数据表的时间旅行

  可结合时间旅行功能检查表在特定时间点的元数据：

  - 示例一：
    ```
    SELECT * FROM spark_catalog.db.table.manifests TIMESTAMP AS OF '2021-09-20 08:00:00';
    ```
  - 示例二：
    ```
    SELECT * FROM spark_catalog.db.table.partitions VERSION AS OF 10963874102873;
    ```

  通过DataFrameReader API对元数据表使用时间旅行：

```
spark.read.format("iceberg").option("snapshot-id", 10963874102873L).load("db.table.files")
```

1.3.4 向 Iceberg 表中写入数据

Iceberg采用Apache Spark的DataSourceV2 API实现数据源和目录写入功能，部分功能仅在Spark中使用Iceberg SQL扩展时可用。

表 1-14 Spark 支持 DataSourceV2 API 介绍

功能	说明	使用要求
INSERT INTO	用于向表中追加新数据的SQL语句。	需要设置**spark.sql.storeAssignmentPolicy=ANSI。**
MERGE INTO	用于行级数据更新操作的SQL语句。	需要在Iceberg SQL扩展中使用。
INSERT OVERWRITE	用于使用查询结果替换表中的数据的SQL语句。	需要设置**spark.sql.storeAssignmentPolicy=ANSI**
DELETE FROM	用于从表中删除满足条件的记录的SQL语句。	行级删除操作需要在Iceberg SQL扩展中使用。
UPDATE	用于更新查询通过过滤器匹配需要更新的行，并对这些行执行字段更新操作的SQL语句。	需要在Iceberg SQL扩展中使用。
DataFrame追加	可使用**append**将DataFrame追加到Iceberg表中。	无
DataFrame覆盖	可使用**overwritePartitions()**动态覆盖分区数据。	无
DataFrame CTAS和RTAS	可使用**create**、**replace**或**createOrReplace**操作运行CTAS或RTAS，以执行创建和替换操作。	需要在DataSourceV2 API中使用。
DataFrame merge into	可使用mergeInto来合并数据到表中。	需要在DataSourceV2 API中使用。

使用 SQL 写入数据

Spark 支持**INSERT INTO**、**MERGE INTO**、**INSERT OVERWRITE**的SQL写入语句，以及新的DataFrameWriterV2 API。

- INSERT INTO

  可使用**INSERT INTO**向表中追加新数据：

  创建表：

  ```
  CREATE TABLE spark_catalog.db.table (id bigint, data string) USING iceberg;
  ```

  写入数据：

  - 示例一：

    ```
    INSERT INTO spark_catalog.db.table VALUES (1, 'a'), (2, 'b');
    ```

- 示例二：
  ```
  INSERT INTO spark_catalog.db.table SELECT ...;
  ```

- MERGE INTO

  Spark新增了**MERGE INTO**查询，可用于行级数据更新操作。

  Iceberg通过重写包含需要更新的行的数据文件（以覆盖提交方式）来支持**MERGE INTO**。推荐使用**MERGE INTO**而非**INSERT OVERWRITE**，因为Iceberg仅需替换受影响的数据文件，且动态覆盖所覆盖的数据可能会因表分区变更而改变。

  MERGE INTO通过另一个查询提供的更新集更新目标表，目标表中每行的更新通过**ON**子句匹配：

  ```
  MERGE INTO spark_catalog.db.target t
  USING (SELECT ...) s
  ON t.id = s.id
  WHEN ...;
  ```

  目标表中匹配的行可通过**WHEN MATCHED** ... **THEN** ... 定义更新操作，可添加多个带条件的**MATCHED**子句，仅第一个匹配的表达式会被执行，例如：

  - 示例一：
    ```
    WHEN MATCHED AND s.op = 'delete' THEN DELETE
    ```
  - 示例二：
    ```
    WHEN MATCHED AND t.count IS NULL AND s.op = 'increment' THEN UPDATE SET t.count = 0
    ```
  - 示例三：
    ```
    WHEN MATCHED AND s.op = 'increment' THEN UPDATE SET t.count = t.count + 1
    ```

  源数据中未匹配到目标表的行可通过**WHEN NOT MATCHED**插入到目标表：
  ```
  WHEN NOT MATCHED THEN INSERT *
  ```

  插入操作也支持附加条件：

  ```
  WHEN NOT MATCHED AND s.event_time > still_valid_threshold THEN INSERT (id, count) VALUES (s.id, 1)
  ```

  源数据中最多只能有一条记录更新目标表对应的一行，否则会报错。

  源数据中未匹配到目标表行时，执行更新或删除操作：

  ```
  WHEN NOT MATCHED BY SOURCE THEN UPDATE SET status = 'invalid'
  ```

- INSERT OVERWRITE

  INSERT OVERWRITE可使用查询结果替换表中的数据，对于Iceberg表，覆盖操作是原子性的。

  INSERT OVERWRITE会基于Spark的分区覆盖模式和表的分区方式替换对应分区。**MERGE INTO**仅重写受影响的数据文件，行为更易理解，因此推荐优先使用**MERGE INTO**而非**INSERT OVERWRITE**。

  - 覆盖行为

    Spark默认使用静态覆盖模式，但写入Iceberg表时推荐使用动态覆盖模式。静态覆盖模式通过将**PARTITION**子句转换为过滤器来确定要覆盖的分区，但**PARTITION**子句只能引用表的列。

    动态覆盖模式可通过设置**spark.sql.sources.partitionOverwriteMode=dynamic**启用。

    以下示例为动态和静态覆盖不同行为：

    创建logs表，定义如下：

    ```
    CREATE TABLE spark_catalog.my_app.logs (uuid string NOT NULL,  level string NOT NULL, ts timestamp NOT NULL, message string) USING iceberg PARTITIONED BY (level, hours(ts));
    ```

    - 动态覆盖：

      当Spark的覆盖模式为动态时，查询结果中包含行的分区将被替换。

      例如，以下查询为从logs表中移除重复的日志事件：

```
INSERT OVERWRITE spark_catalog.my_app.logs SELECT uuid, first(level), first(ts),
first(message) FROM spark_catalog.my_app.logs WHERE cast(ts as date) = '2020-07-01'
GROUP BY uuid;
```

在动态模式下，此操作将仅替换查询结果中包含行的分区。由于所有行的日期都限制在7月1日，因此仅当天的小时分区会被替换。

- 静态覆盖：

  当Spark的覆盖模式为静态时，**PARTITION**子句会被转换为过滤器，用于从表中删除数据。若省略**PARTITION**子句，所有分区都将被替换。

  在以上示例中，对于不含**PARTITION**子句的查询，在静态模式下会删除表中所有现有行，仅写入7月1日的日志。

  如果需要仅覆盖加载的分区，需要添加与**SELECT**查询过滤器匹配的**PARTITION**子句：

  ```
  INSERT OVERWRITE spark_catalog.my_app.logs PARTITION (level = 'INFO') SELECT uuid,
  first(level), first(ts), first(message) FROM spark_catalog.my_app.logs WHERE level =
  'INFO' GROUP BY uuid;
  ```

  需注意，静态模式无法像动态模式替换小时级分区，因为**PARTITION**子句只能引用表的显式列，不能引用隐藏分区。

- DELETE FROM

  Spark 新增支持**DELETE FROM**查询，用于从表中删除数据。

  删除查询通过过滤器（**WHERE**子句）匹配需要删除的行，仅删除满足条件的记录，且操作对Iceberg表是原子性的。

  示例一：

  ```
  DELETE FROM spark_catalog.db.table WHERE ts >= '2020-05-01 00:00:00' and ts < '2020-06-01
  00:00:00';
  ```

  示例二：

  ```
  DELETE FROM spark_catalog.db.all_events WHERE session_time < (SELECT min(session_time) FROM
  spark_catalog.db.good_events);
  ```

  示例三：

  ```
  DELETE FROM spark_catalog.db.orders AS t1 WHERE EXISTS (SELECT oid FROM
  spark_catalog.db.returned_orders WHERE t1.oid = oid);
  ```

  如果删除过滤器匹配表的整个分区，Iceberg无需重写数据文件，仅通过更新元数据即可完成操作，性能极高。

  如果删除过滤器仅匹配表中的部分行，Iceberg会仅重写包含这些行的数据文件，即读取旧文件、过滤掉需删除的行、写入新文件，最后更新元数据指向新文件，最大限度减少数据处理量。

- UPDATE

  UPDATE用于更新查询通过过滤器匹配需要更新的行，并对这些行执行字段更新操作。

  示例一：

  ```
  UPDATE spark_catalog.db.table SET c1 = 'update_c1', c2 = 'update_c2' WHERE ts >= '2020-05-01
  00:00:00' and ts < '2020-06-01 00:00:00';
  ```

  示例二：

  ```
  UPDATE spark_catalog.db.all_events SET session_time = 0, ignored = true WHERE session_time <
  (SELECT min(session_time) FROM spark_catalog.db.good_events);
  ```

  示例三：

  ```
  UPDATE spark_catalog.db.orders AS t1 SET order_status = 'returned' WHERE EXISTS (SELECT oid
  FROM spark_catalog.db.returned_orders WHERE t1.oid = oid);
  ```

  对于基于输入数据的更复杂行级更新，可参考**MERGE INTO**，能更灵活地处理多条件组合的更新需求。

写入分支数据

在执行写入操作前，目标分支必须已存在，写入操作不会自动创建不存在的分支。创建分支操作可参考**Branch和Tag DDL语法**。

- 通过SQL语句写入分支

  分支写入可通过在操作中指定分支标识符**branch_yourBranch**实现。也可通过指定**spark.wap.branch**配置，在写入-审计-发布（WAP）工作流中执行分支写入。需注意，不能同时指定WAP分支和分支标识符。

  MERGE INTO分支数据：

  ```
  MERGE INTO spark_catalog.db.table.branch_audit t USING (SELECT ...) s  ON t.id = s.id  WHEN ...;
  ```

  UPDATE分支数据：

  ```
  UPDATE spark_catalog.db.table.branch_audit AS t1 SET val = 'c';
  ```

  删除分支数据：

  ```
  DELETE FROM spark_catalog.db.table.branch_audit WHERE id = 2;
  ```

  WAP写入分支：

  a. 配置**spark.wap.branch**：
  ```
  SET spark.wap.branch = audit-branch;
  ```
  b. 插入数据：
  ```
  INSERT INTO spark_catalog.db.table VALUES (3, 'c');
  ```

- 通过DataFrames写入分支

  通过DataFrames写入分支时，也需要在操作中指定分支标识符**branch_yourBranch**。

  示例一：

  ```
  val data: DataFrame = ...
  data.writeTo("spark_catalog.db.table.branch_audit").append();
  ```

  示例二：

  ```
  val data: DataFrame = ...
  data.writeTo("spark_catalog.db.table.branch_audit").overwritePartitions();
  ```

使用 DataFrames 写入数据

Spark 支持DataFrameWriterV2 API，用于通过DataFrames写入表。推荐使用v2 API，原因如下：

- 支持CTAS、RTAS和按过滤器覆盖。
- 所有操作始终按名称将列写入表。
- 支持在partitionedBy中使用隐藏分区表达式。
- 覆盖行为是显式的，要么是动态的，要么是用户提供的过滤器。
- 每个操作的行为与SQL语句对应：
  - **df.writeTo(t).create()**对应**CREATE TABLE AS SELECT**。
  - **df.writeTo(t).replace()**对应**REPLACE TABLE AS SELECT**。
  - **df.writeTo(t).append()**对应**INSERT INTO**。
  - df.writeTo(t).overwritePartitions() 等价于动态 INSERT OVERWRITE

v1 DataFrame写入API仍然支持，但不推荐使用。

说明

在Spark 3中使用v1 DataFrame API进行写入时，请务必使用saveAsTable或insertInto来通过Catalog加载表。直接使用format("iceberg")会加载一个孤立的表引用，这会导致查询所使用的表无法自动刷新。

- 追加数据

  可使用**append**将DataFrame追加到Iceberg表中：

  ```
  val data: DataFrame = ...
  data.writeTo("spark_catalog.db.table").append()
  ```

- 覆盖数据

  可使用**overwritePartitions()**动态覆盖分区：

  ```
  val data: DataFrame = ...
  data.writeTo("spark_catalog.db.table").overwritePartitions()
  ```

  使用**overwrite**并提供过滤器可显式覆盖分区：

  ```
  data.writeTo("spark_catalog.db.table").overwrite($"level" === "INFO")
  ```

- 创建表

  可使用**create**、**replace**或**createOrReplace**操作运行CTAS或RTAS：

  ```
  val data: DataFrame = ...
  data.writeTo("spark_catalog.db.table").create()
  ```

  如果已用Iceberg的SparkSessionCatalog替换默认的Spark目录（spark_catalog），请执行：

  ```
  val data: DataFrame = ...
  data.writeTo("db.table").using("iceberg").create()
  ```

  创建和替换操作支持表配置方法，如partitionedBy和tableProperty：

  ```
  data.writeTo("spark_catalog.db.table").tableProperty("write.format.default",
  "orc").partitionedBy($"level", days($"ts")).createOrReplace()
  ```

  也可以通过**location**表属性指定Iceberg表的位置：

  ```
  data.writeTo("spark_catalog.db.table").tableProperty("location", "/path/to/
  location").createOrReplace()
  ```

- 合并数据

  Spark 4.0增加了对使用DataFrameWriterV2 API执行MERGE INTO查询的支持。

  MERGE INTO查询通过来自源数据的一组更新操作来更新目标表，在这种场景下，该源数据即为一个DataFrame。

  ```
  val source: DataFrame = ...                              // 例如 读一张表作为源数据
  source.mergeInto("target", $"source.id" === $"target.id") // 第二个参数是on条件
      .whenMatched($"target.id" === 1)                        // 额外条件
      .updateAll()                                       // UPDATE SET *
      .whenMatched($"target.id" === 2)
      .delete()
      .whenNotMatched()
      .insertAll()                                       // INSERT *
      .whenNotMatchedBySource($"target.id" === 3)
      .update(Map("status" -> lit("invalid")))              // 更新列
      .merge()
  ```

- 模式合并

  在插入或更新数据时，Iceberg能够在运行时解决模式不匹配问题。如果配置正确，Iceberg将执行自动模式演进，具体如下：

  - 源表中存在新列但目标表中不存在：新列将添加到目标表，表中已存在的所有行的该列值将设置为NULL。
  - 目标表中存在某列但源表中不存在：插入数据时，目标列值将设置为NULL；更新行时，目标列值保持不变。

要启用此功能，目标表必须配置“write.spark.accept-any-schema”为“true”以接受任何模式更改：

```
ALTER TABLE spark_catalog.db.sample SET TBLPROPERTIES ('write.spark.accept-any-schema'='true');
```

写入器必须启用mergeSchema选项：

```
data.writeTo("spark_catalog.db.sample").option("mergeSchema","true").append()
```

写入分布模式

Iceberg的默认Spark写入器要求每个Spark Task中的数据需按分区值聚簇。这种分布方式可最大限度减少写入时保持打开的文件句柄数量。Iceberg 1.2.0及之后版本，默认情况下还会要求Spark对写入数据进行预排序，以适配该分布方式，通过表属性“write.distribution-mode”配置，默认值为“hash”。

以下示例为理解写入分布模式的作用：

创建表：

```
CREATE TABLE spark_catalog.db.sample (id bigint, data string, category string, ts timestamp) USING iceberg
PARTITIONED BY (days(ts), category);
```

向该表写入数据时，需确保数据按**days(ts)**和category排序（默认的hash分布模式会自动处理，无需手动排序）。例如：

```
INSERT INTO spark_catalog.db.sample
SELECT id, data, category, ts FROM another_table;
```

“write.distribution-mode”支持以下3种配置，分别对应不同的Spark数据分片和排序策略：

- none（Iceberg 1.2.0之前版本默认值）
  - 核心逻辑：不要求Spark自动执行任何Shuffle（数据重分区）或排序操作。
  - 数据要求：需手动确保数据按分区值排序（排序可在单个Spark Task内完成，也可对全量数据集全局排序）。全局排序能最大限度减少输出文件数量。
  - 特殊处理：如果不想排序，可启用Spark的**write fanout**属性，但这会导致所有文件句柄保持打开状态，直到每个写入Task完成，可能增加资源占用。
  - 适用场景：数据已提前按分区值排序（例如上游任务输出已满足顺序），无需额外Shuffle开销的场景。
- hash（Iceberg 1.2.0及之后版本默认值）
  - 核心逻辑：要求Spark通过基于哈希的交换对写入数据进行Shuffle，再执行写入。
  - 具体过程：对每一行数据，根据其分区值计算哈希值，再按哈希值将数据分配到对应的Spark Task中。如果启用Spark自适应查询计划，Task可能会进一步拆分或合并，以优化性能。
  - 优势：自动完成数据按分区值聚簇，无需手动排序，平衡性能与资源开销。
  - 适用场景：适用于大多数常规写入场景，尤其是数据未提前排序、需自动适配分区分布的情况。
- range（范围分布）
  - 核心逻辑：要求Spark通过基于范围的交换对数据进行Shuffle，写入前会对数据全局排序。
  - 具体过程（两阶段）：

i. 采样阶段：基于分区列和排序列，对写入数据进行采样，确定数据的范围分布规则。

ii. Shuffle阶段：根据采样得到的范围信息，将数据分配到不同Spark Task，每个Task处理一段唯一的数据范围，最终实现 “按分区聚簇 + 全局排序” 。

- 特点：
  - 开销高于“hash”模式，即需额外采样和全局排序。
  - 全局排序可优化读性能：如果查询时常用排序列过滤或排序，预排序的数据能减少读取时的计算开销。
- 默认触发：如果表创建时指定了sort-order（排序规则），“write.distribution-mode”会自动设置为“range”。
- 适用场景：表有明确排序需求（如按时间列排序），且读性能优先级高于写入开销的场景。

控制文件大小

使用Spark向Iceberg写入数据时，需注意：

- Spark无法写出比单个Spark Task数据量更大的文件，且单个文件不能跨越Iceberg的分区边界。意味着尽管当文件大小达到目标文件大小配置“write.target-file-size-bytes”的值时，Iceberg总会自动滚动生成新文件；但如果Spark Task的数据量未达到该阈值，文件滚动就不会触发。
- 磁盘上生成的文件大小会远小于Spark Task的数据量。因为磁盘数据采用压缩的列存储格式，而Spark中的数据则是未压缩的行存储格式。例如，一个数据量为100 MB的Spark Task，即便仅向单个Iceberg分区写入数据，生成的磁盘文件也会远小于100 MB；如果该Task需向多个分区写入数据，生成的单个文件会更小。

如果要控制每个Spark Task中的数据内容，可使用写入分布模式或手动对数据进行重分区。

如果要调整Spark Task的大小，需熟悉Spark的各类自适应查询执行（Adaptive Query Execution，即AQE）参数。当“write.distribution-mode”配置不为“none”时，在数据交换过程中，AQE会控制Spark Task的合并与拆分，使得生成大小符合“spark.sql.adaptive.advisoryPartitionSizeInBytes”配置的Task。这些配置同样会影响用户手动执行的重分区或排序操作。还需注意，该参数定义的是Spark内存中行数据的大小，而非磁盘上列存储压缩后的文件大小，因此需将其配置为比目标文件大小更大的值，内存数据大小与磁盘文件大小的比例取决于数据本身的特性。

Iceberg 幂等写入

当前任务在向Iceberg表写入数据时，如果遇到任务中断并重启，任务终止前可能已写入部分数据。若重启后的任务重复写入相同数据，会造成表内数据重复。Iceberg提供幂等写入能力，开启此能力后可避免此问题。

开启Iceberg幂等写入

在spark-sql命令中指定spark.sql.iceberg.write.idempotent.enabled，设为true即可开启幂等写入，其默认值为false。

其他配置项参数

表 1-15 其他配置项

配置项	参数	说明
幂等写入任务名称	spark.sql.iceberg.write.txnAppId	txnAppId：为任务唯一标识，字符串类型（例如任务名称"dailyETL"）。
幂等写入版本号	spark.sql.iceberg.write.txnVersion	txnVersion：数字支持自定义设置，仅限单调递增，作为事务版本号。针对同一张Iceberg 表的每一批写入数据，该值必须唯一。整数类型（例如"23423"）。
自动重置版本号	spark.sql.iceberg.write.txnVersion.autoReset.enabled	设为true，可在每次写入完成后自动重置版本号，无需再次手动设置txnVersion。

每一批新增入库数据对应的txnAppId + txnVersion组合必须唯一，且txnVersion数值需大于上一批成功写入数据的版本号。举例说明：

1. 上一批成功写入数据的参数为dailyETL:23423（txnAppId:txnVersion）。

2. 下一次写入需保持txnAppId = dailyETL，且txnVersion至少设为23424（在上一版本基础上加一）。

3. 若使用dailyETL搭配23422及更小的版本号写入，Iceberg会直接忽略该写入请求（版本号低于表中已记录的最新版本）。

4. 若使用anotherETL:23424 写入则可正常执行，因其任务标识与历史数据不同。

优先级说明

若同时配置写入参数与会话配置，以写入参数为准。

配置示例：

```
spark-sql
--conf spark.sql.iceberg.write.idempotent.enabled=true
--conf spark.sql.iceberg.write.txnAppId=dailyETL

spark-sql >
set spark.sql.iceberg.write.txnVersion=1;
insert into Test_Iceberg select * from Test_Source;   //txnVersion为1，成功写入
insert into Test_Iceberg select * from Test_Source;   //autoReset为false，txnVersion仍为1，写入失败抛出异常，相同txnVersion不能被用于其他Insert

-- AUTO RESET SAMPLE USAGE
set spark.sql.iceberg.write.txnVersion.autoReset.enabled=true;
set spark.sql.iceberg.write.txnVersion=2;

insert into Test_Iceberg select * from Test_Source;   //txnVersion为2，该txnVersion没有Insert，成功写入
insert into Test_Iceberg select * from Test_Source;   //txnVersion自动刷新为3，成功写入
```

1.3.5 Iceberg 表存储过程管理

在Spark中使用Iceberg前，需先配置Spark目录，spark_catalog和prod目录配置请参见1.3.1 Iceberg配置与类型。存储过程仅在Spark 3中使用Iceberg SQL扩展时可用。

使用方法

可通过**CALL**语句从任何已配置的Iceberg Catalog中调用存储过程，所有存储过程均位于system命名空间下。

CALL支持按名称传递参数（推荐方式）或按位置传递参数，且不支持混合使用位置参数和命名参数。

- 命名参数

  所有存储过程参数都有名称。按名称传递参数时，参数顺序可以任意，且任何可选参数都可省略。

  ```
  CALL spark_catalog.system.{存储名} (arg_name_2 => 'arg_2_value', arg_name_1 => 'arg_1_value');
  ```

- 位置参数

  按位置传递参数时，仅当末尾的参数为可选参数时，才可省略。

  ```
  CALL spark_catalog.system.{存储名} (arg_1, arg_2, ... arg_n);
  ```

快照管理

- **rollback_to_snapshot**

  可使用**rollback_to_snapshot**将表回滚到特定的快照ID。以下命令为将表**db.sample**回滚到快照ID为1的版本：

  ```
  CALL catalog_name.system.rollback_to_snapshot('db.sample', 1);
  ```

  其中，表名字段的类型为string，快照ID字段的类型为long。

  输出结果介绍请参见**表1-16**。

  表 1-16 回滚输出结果

参数	类型	说明
previous_snapshot_id	long	回滚前的当前快照ID。
current_snapshot_id	long	回滚后的新当前快照ID。

- **rollback_to_timestamp**

  可使用**rollback_to_timestamp**将表回滚到特定时间点对应的活跃快照。以下命令为将**db.sample**表回滚到特定日期和时间：

  ```
  CALL spark_catalog.system.rollback_to_timestamp('db.sample', TIMESTAMP '2021-06-30 00:00:00.000');
  ```

  其中，表名字段的类型为string，timestamp的类型为timestamp，表示回滚目标时间戳。

  输出结果介绍请参见**表1-16**。

- **set_current_snapshot**

  可使用**set_current_snapshot**设置表的当前快照ID。与回滚不同，该操作不要求目标快照必须是表当前状态的祖先快照。例如：

  将**db.sample**表的当前快照设置为ID为1的快照：

  ```
  CALL spark_catalog.system.set_current_snapshot('db.sample', 1);
  ```

  将**db.sample**表的当前快照设置为标签s1对应的快照：

  ```
  CALL spark_catalog.system.set_current_snapshot(table => 'db.sample', ref => 's1');
  ```


其中，表名字段的类型为string，快照ID字段的类型为long，“ref”为设置当前快照的快照引用（分支或标签），类型为string。且必须设置快照ID或ref，二者不可同时提供，也不可均不提供。

输出结果介绍请参见**表1-17**。

表 1-17 快照设置输出结果

参数	类型	说明
previous_snapshot_id	long	设置前的当前快照ID。
current_snapshot_id	long	设置后新的当前快照ID。

- **cherrypick_snapshot**

  可使用**cherrypick_snapshot**从指定快照中挑选变更并应用到表的当前状态。挑选操作会基于现有快照创建一个新快照，且不会修改或删除原始快照。仅支持对追加和动态覆盖类型的快照执行挑选操作。例如：

  挑选快照1的变更：

  ```
  CALL spark_catalog.system.cherrypick_snapshot('my_table', 1);
  ```

  使用命名参数挑选快照1的变更：

  ```
  CALL spark_catalog.system.cherrypick_snapshot(snapshot_id => 1, table => 'my_table' );
  ```

  其中，“table”表示待更新的表名，类型为string；“snapshot_id”为要设为当前快照的快照ID，类型为long。

  输出结果介绍请参见**表1-18**。

  表 1-18 快照变更输出结果

参数	类型	说明
source_snapshot_id	long	设置前的当前快照ID。
current_snapshot_id	long	设置后新的当前快照ID。

- **publish_changes**

  可使用**publish_changes**将暂存的WAP ID中的变更发布到表的当前状态。变更发布会基于现有快照创建一个新快照，且不会修改或删除原始快照。仅追加和动态覆盖类型的快照可成功发布。例如：

  使用WAP ID“wap_id_1”发布变更：

  ```
  CALL spark_catalog.system.publish_changes('my_table', 'wap_id_1');
  ```

  使用命名参数发布变更：

  ```
  CALL spark_catalog.system.publish_changes(wap_id => 'wap_id_2', table => 'my_table');
  ```

  其中，“table”表示待更新的表名，类型为string；“wap_id”为要从暂存环境发布的WAP ID，类型为long。

  输出结果介绍请参见**表1-19**。

表 1-19 快照变更输出结果

参数	类型	说明
source_snapshot_id	long	设置前的当前快照ID。
current_snapshot_id	long	设置后新的当前快照ID。

- **fast_forward**

  可使用**fast_forward**将一个分支的当前快照快速更新到另一个分支的最新快照。例如：

  将**main**分支的当前快照快速更新到**audit-branch**分支的最新快照中：

  ```
  CALL spark_catalog.system.fast_forward('my_table', 'main', 'audit-branch');
  ```

  其中，表字段的类型为string，分支字段的类型为string。

  输出结果介绍请参见**表1-20**。

  表 1-20 快照更新输出结果

参数	类型	说明
branch_updated	string	已完成快照快速更新的分支名称。
previous_ref	long	应用快速更新前的快照ID。
updated_ref	long	应用快速更新后的当前快照ID。

元数据管理

expire_snapshots

在Iceberg中，每次写入、更新、删除、upsert、压缩操作都会生成新快照，同时保留旧数据和元数据，以支持快照隔离和时间旅行功能。**expire_snapshots**可用于移除不再需要的旧快照及其文件。

expire_snapshots会移除旧快照以及这些旧快照唯一需要的数据文件，意味着该操作绝不会移除仍被非过期快照所需要的文件。例如：

- 移除特定日期和时间之前的快照，但保留最后100个快照：
  ```
  CALL spark_catalog.system.expire_snapshots('db.sample', TIMESTAMP '2021-06-30 00:00:00.000', 100);
  ```
- 移除快照ID为123的快照，且此快照ID不应是当前快照：
  ```
  CALL spark_catalog.system.expire_snapshots('db.sample', snapshot_ids => array(123));
  ```

相关参数介绍请参见**表1-21**。

表 1-21 expire_snapshots 参数介绍

参数	类型	说明
table	string	需要更新的表名，必须设置。

参数	类型	说明
older_than	timestamp	此时间戳之前的快照将被移除，默认值为5天前。
retain_last	int	保留的祖先快照数量，默认值为1，与“older_than”设置的时间戳无关。
max_concurrent_deletes	int	用于删除文件操作的线程池大小，默认情况下，不使用线程池。
stream_results	boolean	当设置该参数为“true”时，删除文件将通过RDD分区发送到Spark驱动程序（默认情况下，所有文件都将发送到Spark驱动程序）。 建议将该参数设置为“true”，以防止Spark驱动程序因文件过大而发生内存溢出。
snapshot_ids	array of long	要过期的快照ID数组。
clean_expired_metadata	boolean	当设置为true时，将清理不再被快照（snapshots）引用的元数据，例如分区规范（partition specs）和模式（schemas）。

如果不设置**older_than**和**retain_last**，将使用表的过期属性，仍被分支或标签引用的快照不会被移除。默认情况下，分支和标签永不过期，但可通过表属性**history.expire.max-ref-age-ms**更改其保留策略，且主分支永不过期。

输出结果介绍请参见**表1-22**。

表 1-22 expire_snapshots 操作输出结果

参数	类型	说明
deleted_data_files_count	long	此操作删除的数据文件数量。
deleted_position_delete_files_count	long	此操作删除的**位置删除文件**数量。
deleted_equality_delete_files_count	long	此操作删除的**等值删除文件**数量。
deleted_manifest_files_count	long	此操作删除的清单文件数量。
deleted_manifest_lists_count	long	此操作删除的清单列表文件数量。
deleted_statistics_files_count	long	此操作删除的统计文件（statistics files）数量。

remove_orphan_files

用于移除未在Iceberg表的任何元数据文件中引用的文件，这些文件可被视为“孤立文件”。

参数名称	是否必填	类型	描述
table	是	string	要清理的表名。
older_than	否	timestamp	移除在此时间戳之前创建的孤立文件（默认值：3天前）。
location	否	string	要查找文件的目录（默认为表的存储位置）
dry_run	否	boolean	设为true时，不实际删除文件（默认值为false）。
max_concurrent_deletes	否	int	用于删除文件操作的线程池大小（默认情况下，不使用线程池）。
file_list_view	否	string	用于查找文件的数据集（跳过目录列表）。
equal_schemes	否	map	被视为相等的文件系统方案映射。键是用逗号分隔的方案列表，值是一个方案（默认值为map('s3a,s3n','s3')）。
equal_authorities	否	map	被视为相等的文件系统权限映射。键是用逗号分隔的权限列表，值是一个权限。
prefix_mismatch_mode	否	string	当位置前缀（方案/权限）不匹配时的行为： ● ERROR：抛出异常（默认值）。 ● IGNORE：不执行任何操作DELETE-删除文件。
prefix_listing	否	boolean	当设置为true时，将通过SupportsPrefixOperations 接口使用基于前缀的文件列举（prefix-based file listing）。启用此标志时，表的FileIO实现必须支持SupportsPrefixOperations接口（默认值为false）。

输出结果

输出名称	类型	描述
orphan_file_location	String	此命令确定为孤立文件的每个文件的路径。

示例

对表执行remove_orphan_files命令的试运行，列出所有候选删除文件但不实际删除它们：

```
CALL spark_catalog.system.remove_orphan_files(table => 'db.sample', dry_run => true);
```

移除表db.sample的tablelocation/data文件夹中所有不被该表识别的文件：

```
CALL spark_catalog.system.remove_orphan_files(table => 'db.sample', location => 'tablelocation/data');
```

移除files_view视图中所有不被表db.sample识别的文件：

```
Dataset<Row> compareToFileList = spark.createDataFrame(allFiles,
FilePathLastModifiedRecord.class).withColumnRenamed("filePath",
"file_path").withColumnRenamed("lastModified", "last_modified");
String fileListViewName = "files_view";
compareToFileList.createOrReplaceTempView(fileListViewName);
CALL spark_catalog.system.remove_orphan_files(table => 'db.sample', file_list_view => 'files_view');
```

当文件除位置前缀（方案/权限）外与元数据文件中的引用匹配时，默认会抛出错误。通过将prefix_mismatch_mode设置为IGNORE，可忽略该错误并跳过文件：

```
CALL spark_catalog.system.remove_orphan_files(table => 'db.sample', prefix_mismatch_mode => 'IGNORE');
```

通过将prefix_mismatch_mode设置为DELETE，仍可删除该文件：

```
CALL spark_catalog.system.remove_orphan_files(table => 'db.sample', prefix_mismatch_mode => 'DELETE');
```

也可通过将不匹配的前缀视为相等来删除文件：

```
CALL spark_catalog.system.remove_orphan_files(table => 'db.sample', equal_schemes => map('file', 'file1'));
CALL spark_catalog.system.remove_orphan_files(table => 'db.sample', equal_authorities => map('ns1',
'ns2'));
```

列出所有符合删除条件的候选文件，并使用基于前缀的列表查询（prefix listing）方式进行扫描。

```
CALL catalog_name.system.remove_orphan_files(table => 'db.sample', prefix_listing => true);
```

rewrite_data_files

Iceberg会跟踪表中的每个数据文件。数据文件越多，清单文件中存储的元数据就越多；而小数据文件会产生不必要的元数据开销，并因文件打开成本导致查询效率降低。

Iceberg可通过rewriteDataFiles操作，利用Spark并行压缩数据文件。此操作会将小文件合并为大文件，以减少元数据开销和运行时的文件打开成本。

使用方法

参数名称	是否必填	类型	描述
table	是	string	要更新的表名。
strategy	否	string	策略名称，可选binpack（默认）或sort。

参数名称	是否必填	类型	描述
sort_order	否	string	排序方式： ● 若使用Zorder，格式为zorder(c1,c2,c3)（逗号分隔的列名列表）。 ● 其他情况，格式为逗号分隔的排序规则(ColumnName SortDirection NullOrder)，其中SortDirection可为ASC或DESC，NullOrder可为NULLSFIRST或NULLSLAST默认使用表的排序规则。
options	否	map	用于操作的选项参数。
where	否	string	作为字符串的过滤条件，用于筛选需要重写的文件。注意：所有可能包含与过滤条件匹配数据的文件都将被选中进行重写。

配置选项

名称	默认值	描述
max-concurrent-file-group-rewrites	5	同时重写的文件组最大数量。
partial-progress.enabled	false	启用在整个重写完成前提交文件组。
partial-progress.max-commits	10	若启用部分进度，此重写操作允许的最大提交次数。
partial-progress.max-failed-commits	partial-progress.max-commits的值	若启用部分进度，作业失败前允许的最大失败提交次数。
use-starting-sequence-number	true	使用压缩开始时快照的序列号，而非新生成快照的序列号。
rewrite-job-order	none	强制重写作业的顺序： ● bytes-asc：优先重写最小的作业组。 ● bytes-desc：优先重写最大的作业组。 ● files-asc：优先重写文件数量最少的作业组。 ● files-desc：优先重写文件数量最多的作业组。 ● none：按计划顺序重写作业组（无特定排序）。

名称	默认值	描述
target-file-size-bytes	536870912（512MB，默认值来自表属性write.target-file-size-bytes）	目标输出文件大小。
min-file-size-bytes	目标文件大小的75%	低于此阈值的文件将被视为需要重写，无论其他条件如何。
max-file-size-bytes	目标文件大小的180%	大于此阈值的文件将被视为需要重写，无论其他条件如何。
min-input-files	5	任何文件组的文件数量超过此值时，无论其他条件如何，都将被重写。
rewrite-all	false	强制重写所有提供的文件，覆盖其他选项。
max-file-group-size-bytes	107374182400（100GB）	单个文件组中应重写的最大数据量。整个重写操作会根据分区拆分，在分区内再根据大小拆分为文件组。这有助于拆分大型分区的重写任务，避免因集群资源限制而无法重写。
delete-file-threshold	2147483647	与数据文件关联的删除操作数量的最小值，达到此值时数据文件将被考虑重写。
delete-ratio-threshold	0.3	数据文件被纳入重写（rewriting）范畴所需达到的最低删除比例（minimum deletion ratio)。
output-spec-id	当前分区spec ID	输出分区规范的标识符。重写过程中，数据将被重新组织以匹配输出分区方式。
remove-dangling-deletes	false	重写后移除悬空的位置删除和等值删除文件。若删除文件不应用于任何活动数据文件，则视为悬空文件。启用此选项将额外生成一个提交来执行删除操作。

排序策略选项

名称	默认值	描述
compression-factor	1.0	Spark排序操作创建的shuffle分区数量（进而决定输出文件数量）基于此文件重写器中使用的输入数据文件大小。由于压缩，磁盘文件大小可能无法准确反映输出文件的实际大小。此参数允许用户调整用于估算实际输出数据大小的文件大小。大于1.0的值会生成比基于磁盘文件大小预期更多的文件；小于1.0的值会生成比基于磁盘大小预期更少的文件。
shuffle-partitions-per-file	1	每个输出文件使用的shuffle分区数量。Iceberg会使用自定义的合并操作将这些排序后的分区重新拼接成一个单一的排序文件。

ZORDER排序顺序的排序策略选项

名称	默认值	描述
var-length-contribution	8	从可变长度类型（String、Binary）的输入列中截取的字节数。
max-output-size	2147483647	ZOrder算法中间交错处理的字节量。

输出结果

输出名称	类型	描述
rewritten_data_files_count	int	此命令重写的数据文件数量。
added_data_files_count	int	此命令写入的新数据文件数量。
rewritten_bytes_count	long	此命令写入的字节数。
failed_data_files_count	int	当partial-progress.enabled为true时，重写失败的数据文件数量。
removed_delete_files_count	int	此命令删除的文件数。

示例

使用默认的bin-packing重写算法重写表db.sample中的数据文件，合并小文件并根据表的默认写入大小拆分大文件：

```
CALL spark_catalog.system.rewrite_data_files('db.sample');
```

通过按id和name对所有数据进行排序来重写表db.sample中的数据文件，使用与bin-pack相同的默认值来确定需要重写的文件：

```
CALL spark_catalog.system.rewrite_data_files(table => 'db.sample', strategy => 'sort', sort_order => 'id DESC NULLS LAST,name ASC NULLS FIRST');
```

通过对列c1和c2执行ZOrder排序来重写表db.sample中的数据文件，使用与bin-pack相同的默认值来确定需要重写的文件：

```
CALL spark_catalog.system.rewrite_data_files(table => 'db.sample', strategy => 'sort', sort_order => 'zorder(c1,c2)');
```

使用bin-pack策略重写表db.sample中任何至少有两个文件需要重写的分区中的数据文件，然后移除所有悬空的删除文件：

```
CALL spark_catalog.system.rewrite_data_files(table => 'db.sample', options => map('min-input-files', '2', 'remove-dangling-deletes', 'true'));
```

重写表db.sample中的数据文件，并选择可能包含与筛选条件（id=3且name="foo"）匹配的数据的文件进行重写：

```
CALL spark_catalog.system.rewrite_data_files(table => 'db.sample', where => 'id = 3 and name = "foo"');
```

rewrite_manifests

重写表的清单文件，以优化扫描规划效率。

清单文件中的数据文件会按分区规范中的字段进行排序。此过程通过Spark作业并行执行。

使用方法

参数名称	是否必填	类型	描述
table	是	string	待更新的表名。
use_caching	否	boolean	操作期间是否使用Spark缓存（默认值为true）。
spec_id	否	int	待重写清单文件对应的分区规范ID（默认值为当前分区规范ID）。

输出结果

输出名称	类型	描述
rewritten_manifests_count	int	此命令重写的清单文件数量。
added_manifests_count	int	此命令写入的新清单文件数量。

示例

重写表db.sample的清单文件，并使清单文件与表分区规则对齐：


```
CALL spark_catalog.system.rewrite_manifests('db.sample');
```

重写表db.sample的清单文件，并禁用Spark缓存（可用于避免执行器出现内存问题）：

```
CALL spark_catalog.system.rewrite_manifests('db.sample', false);
```

rewrite_position_delete_files

Iceberg可重写位置删除文件，该操作主要实现两大功能：

- 轻度压缩：将小型位置删除文件合并为大型文件，减少清单文件中存储的元数据体积，同时降低打开小型删除文件的开销。
- 移除悬空删除：过滤掉指向已不再活跃的数据文件的位置删除记录。执行rewrite_data_files后，指向被重写数据文件的位置删除记录未必会被标记为待移除，可能仍会被表的活跃快照元数据追踪，这一现象被称为“悬空删除”问题。

使用方法

参数名称	是否必填	类型	描述
table	是	string	待更新的表名。
options	否	map	用于此过程的配置选项。
where	否	string	用于过滤文件的字符串形式的过滤条件。

配置选项

名称	默认值	描述
max-concurrent-file-group-rewrites	5	可同时重写的文件组最大数量。
partial-progress.enabled	false	启用“部分进度提交”，允许在整个重写操作完成前提交部分文件组。
partial-progress.max-commits	10	若启用部分进度，此重写操作允许的最大提交次数。

名称	默认值	描述
rewrite-job-order	none	强制指定重写作业的执行顺序： • bytes-asc：优先重写数据量最小的文件组。 • bytes-desc：优先重写数据量最大的文件组。 • files-asc：优先重写文件数量最少的文件组。 • files-desc：优先重写文件数量最多的文件组。 • none：按作业规划顺序重写（无特定排序）。
target-file-size-bytes	67108864（64MB，默认值来自表属性write.delete.target-file-size-bytes）	输出文件的目标大小。
min-file-size-bytes	目标文件大小的75%	低于此阈值的文件，无论其他条件如何，均会被纳入重写范围。
max-file-size-bytes	目标文件大小的180%	超过此阈值的文件，无论其他条件如何，均会被纳入重写范围。
min-input-files	5	任何文件组的文件数量超过此值时，无论其他条件如何，均会被重写。
rewrite-all	false	强制重写所有提供的文件，覆盖其他筛选条件。
max-file-group-size-bytes	107374182400（100GB）	单个文件组中可重写的最大数据量。整个重写操作会按分区拆分，分区内再按大小拆分为文件组，避免因集群资源限制导致大型分区无法重写。
max-files-to-rewrite	null	此选项设置了将被重写的符合条件的文件数量上限。如果未指定此选项，所有符合条件的文件都将被重写。

输出结果

输出名称	类型	描述
rewritten_delete_files_count	int	此命令移除的旧位置删除文件数量。

输出名称	类型	描述
added_delete_files_count	int	此命令新增的位置删除文件数量。
rewritten_bytes_count	long	此命令移除的所有旧位置删除文件的总字节数。
added_bytes_count	long	此命令新增的所有位置删除文件的总字节数。

示例

重写表db.sample的位置删除文件：按默认筛选条件选择待重写文件，生成符合target-file-size-bytes目标大小的新文件，且重写过程中自动移除悬空删除记录。

```
CALL spark_catalog.system.rewrite_position_delete_files('db.sample');
```

重写表db.sample的所有位置删除文件：强制覆盖筛选条件，生成符合目标大小的新文件，同时移除悬空删除记录。

```
CALL spark_catalog.system.rewrite_position_delete_files(table => 'db.sample', options => map('rewrite-all', 'true'));
```

按文件数量筛选重写表db.sample的位置删除文件：仅重写“按大小标准需重写的文件数≥2”的分区中的位置删除文件，同时移除悬空删除记录。

```
CALL spark_catalog.system.rewrite_position_delete_files(table => 'db.sample', options => map('min-input-files','2'));
```

表迁移

快照和迁移存储过程可帮助测试现有Hive或Spark表，并将其迁移至Iceberg表。

snapshot

创建源表的轻量级临时副本用于测试，且不会修改源表本身。

新创建的快照表可进行修改或写入操作，不会影响源表；但快照表会引用源表的原始数据文件。当对快照表执行插入或覆盖操作时，新生成的文件会存储在快照表的指定位置，而非源表的存储位置。

快照表测试完成后，可通过执行DROP TABLE语句清理该表。

使用方法

参数名称	是否必填	类型	描述
source_table	是	string	待生成快照的源表名。
table	是	string	要创建的新Iceberg快照表名。
location	否	string	新快照表的存储位置（默认由目录自动分配，目前在AI DataLake表迁移场景下，禁止设置该参数）。

参数名称	是否必填	类型	描述
properties	否	map	要添加到新快照表的属性（如表级配置）。
parallelism	否	int	用于读取文件的线程数（默认值为1）。

输出结果

输出名称	类型	描述
imported_files_count	long	添加到新快照表的文件数量（即引用的源表数据文件数）。

示例

创建隔离的Iceberg快照表：引用源表db.sample，新表名为db.snap，存储位置使用目录为db.snap分配的默认位置。

```
CALL spark_catalog.system.snapshot('db.sample', 'db.snap');
```

migrate

将现有表替换为Iceberg表，并加载源表的数据文件。

源表的表结构、分区配置、表属性及存储位置会被复制到新的Iceberg表中。

若源表的任一分区使用了不支持的文件格式，迁移操作将失败。当前支持的文件格式为Avro、Parquet和ORC。源表的现有数据文件会被添加到Iceberg表的元数据中，且可通过基于源表结构生成的“名称-ID”映射关系进行读取。

若需在测试期间保持源表完好无损，可使用snapshot过程创建共享源表数据文件和结构的临时表。

默认情况下，原始表会被保留，并命名为table_BACKUP_。

使用方法

参数名称	是否必填	类型	描述
table	是	string	待迁移的源表名。
properties	否	map	要添加到新Iceberg表的属性（会覆盖源表中同名属性）。
drop_backup	否	boolean	设为true时，不会保留原始表作为备份（默认值为false）。
backup_table_name	否	string	保留的原始备份表名称（默认值为源表名_BACKUP_）。

参数名称	是否必填	类型	描述
parallelism	否	int	用于读取文件的线程数（默认值为1）。

输出结果

输出名称	类型	描述
migrated_files_count	long	追加到新Iceberg表中的数据文件数量。

示例

迁移Spark默认目录中的表并添加属性：将spark_catalog.db.sample迁移为Iceberg表，并为新表添加属性'foo'='bar'。

```
CALL spark_catalog.system.migrate('spark_catalog.db.sample', map('foo', 'bar'));
```

基础迁移（无额外属性）：将当前目录下的db.sample迁移为Iceberg表，不添加额外属性

```
CALL spark_catalog.system.migrate('db.sample');
```

add_files

尝试将Hive表或基于文件的表中的文件直接添加到指定的Iceberg表中。与迁移或快照不同，add_files可从特定分区导入文件，且不会创建新的Iceberg表。此命令会为待添加的文件生成元数据，但不会移动文件本身。该过程不会分析文件的schema以验证其是否与Iceberg表的schema匹配。操作完成后，Iceberg表会将这些文件视为自身管理的文件集一部分，这意味着后续执行过期快照时，可对这些添加的文件执行物理删除。若可使用migrate或snapshot实现需求，不建议使用此方法。

使用方法

参数名称	是否必填	类型	描述
table	是	string	待添加文件的目标Iceberg表名。
source_table	是	string	文件来源表（可为Hive/Spark注册的表，或格式为file_format.path的文件路径，如parquet./data/path）。
partition_filter	否	map	用于筛选源表分区的映射（仅导入符合筛选条件的分区文件）。
check_duplicate_files	否	boolean	是否防止已存在于目标表的文件被重复添加（默认值为true）。
parallelism	否	int	用于读取文件的线程数（默认值为1）。

输出结果

输出名称	类型	描述
added_files_count	long	此命令成功添加的文件数量。
changed_partition_count	long	受影响的分区数量（若可统计）。

示例

从指定分区导入文件：将会话目录中注册的Hive/Spark表db.src_tbl中，part_col_1等于A的分区文件，添加到Iceberg表db.tbl。

```
CALL spark_catalog.system.add_files( table => 'db.tbl', source_table => 'db.src_tbl', partition_filter => map('part_col_1', 'A') );
```

从文件路径导入全量文件：将路径path/to/table下基于Parquet格式的表文件，全量添加到Iceberg表db.tbl（不筛选分区）。

```
CALL spark_catalog.system.add_files( table => 'db.tbl',   source_table => '`parquet`.`path/to/table`' );
```

register_table

为已存在但尚未关联目录标识符的metadata.json文件创建目录条目。通过此操作，可将独立的Iceberg元数据文件在指定目录中注册为正式的表，使其能被目录识别和管理。

使用方法

参数名称	是否必填	类型	描述
table	是	string	待注册的表名。
metadata_file	是	string	待注册的metadata.json文件路径。

输出结果

输出名称	类型	描述
current_snapshot_id	long	新注册Iceberg表的当前快照ID。
total_records_count	long	新注册Iceberg表的总记录数。
total_data_files_count	long	新注册Iceberg表的总数据文件数。

示例

在spark_catalog目录中，将路径为path/to/metadata/file.json的metadata.json文件注册为表db.tbl：

```
CALL spark_catalog.system.register_table( table => 'db.tbl', metadata_file => 'path/to/metadata/file.json' );
```

元数据信息

ancestors_of

查询指定快照的父快照对应的活跃快照ID，即返回该快照的祖先快照链。

使用方法

参数名称	是否必填	类型	描述
table	是	string	需查询祖先快照的表名。
snapshot_id	否	long	用于查询父快照活跃ID的指定快照ID。

输出结果

输出名称	类型	描述
snapshot_id	long	祖先快照的ID。
timestamp	long	该祖先快照的创建时间。

示例

查询当前快照的所有祖先快照（默认行为）：

```
CALL spark_catalog.system.ancestors_of('db.tbl');
```

通过指定快照ID查询其所有祖先快照（使用位置参数）：

```
CALL spark_catalog.system.ancestors_of('db.tbl', 1);
```

通过指定快照ID查询其所有祖先快照（使用命名参数，推荐，提升可读性）：

```
CALL spark_catalog.system.ancestors_of(snapshot_id => 1, table => 'db.tbl');
```

Change Data Capture

create_changelog_view

创建一个包含指定表变更记录的视图，可通过该视图追踪表在不同快照或时间戳之间的所有数据变更。

使用方法

参数名称	是否必填	类型	描述
table	是	string	生成变更记录的源表名。

参数名称	是否必填	类型	描述
changelog_view	否	string	要创建的变更视图名称（若不指定，默认生成与源表关联的视图名）。
options	否	map	Spark读取选项映射，用于指定变更记录的时间范围（常用选项见下文）。
net_changes	否	boolean	是否仅输出“净变更”（移除中间变更，仅保留最终结果），默认值为false。当compute_updates=true时，此参数必须为false。
compute_updates	否	boolean	是否计算“更新前后镜像”（将删除+插入组合识别为更新），默认规则：-若提供identifier_columns，默认值为true；-若未提供，默认值为false。
identifier_columns	否	array	用于识别“更新操作”的标识列列表（如array('id','name')）。若compute_updates=true且未指定此参数，则使用表的当前标识字段。

通过options参数可限定变更记录的时间范围，支持两种维度（快照ID或时间戳），若不指定则默认覆盖表的全量快照：

- start-snapshot-id：排他性起始快照ID（不包含此快照的变更），默认从表的第一个快照开始（包含）；
- end-snapshot-id：包含性结束快照ID（包含此快照的变更），默认使用表的当前快照；
- start-timestamp：排他性起始时间戳（毫秒级），默认从表的第一个快照开始（包含）；
- end-timestamp：包含性结束时间戳（毫秒级），默认使用表的当前快照。

输出结果

输出名称	类型	描述
changelog_view	string	已创建的变更视图名称。

示例

创建视图tbl_changes，包含源表db.tbl在快照1（排他）到快照2（包含）之间的变更：

```
CALL spark_catalog.system.create_changelog_view(  table => 'db.tbl',  options => map('start-snapshot-id','1','end-snapshot-id', '2'));
```

创建视图my_changelog_view，包含源表db.tbl在时间戳1678335750489（排他）到1678992105265（包含）之间的变更：

```
CALL spark_catalog.system.create_changelog_view(   table => 'db.tbl',   options => map('start-timestamp','1678335750489','end-timestamp', '1678992105265'),   changelog_view => 'my_changelog_view' );
```

创建视图，通过标识列id和name将“删除+插入”识别为更新操作，包含快照1到2之间的变更：

```
CALL spark_catalog.system.create_changelog_view(   table => 'db.tbl',   options => map('start-snapshot-id','1','end-snapshot-id', '2'),   identifier_columns => array('id', 'name') )
```

视图创建后，可直接通过SQL查询变更记录：

```
SELECT * FROM tbl_changes;
SELECT * FROM tbl_changes where _change_type = 'INSERT' AND id = 3 ORDER BY _change_ordinal;
```

请注意，变更日志视图包含变更数据捕获（CDC）元数据列，这些列提供所追踪变更的额外信息。具体列如下：

- _change_type：变更类型，取值范围为INSERT（插入）、DELETE（删除）、UPDATE_BEFORE（更新前）或UPDATE_AFTER（更新后）。
- _change_ordinal：变更顺序。
- _commit_snapshot_id：变更发生时所属的快照ID。

以下是对应的结果示例，该示例显示第一个快照插入2条记录，第二个快照删除了1条记录。

id	name	_change_type	_change_ordinal	_commit_snapshot_id
1	Alice	INSERT	0	5390529835796506035
2	Bob	INSERT	0	5390529835796506035
1	Alice	DELETE	1	8764748981452218370

Net Changes

此过程可移除多快照间的中间变更，仅输出净变更。以下是创建计算净变更的变更日志视图的示例：

```
CALL spark_catalog.system.create_changelog_view(   table => 'db.tbl',   options => map('end-snapshot-id', '8764748981452218370'),   net_changes => true );
```

由于Alice在第一个快照中被插入、在第二个快照中被删除，因此上述变更日志视图在启用净变更后，仅包含以下行：

id	name	_change_type	_change_ordinal	_commit_snapshot_id
2	Bob	INSERT	0	5390529835796506035

Carry-over Rows

此过程默认移除结转行。结转行是采用写时复制模式时，执行行级操作（MERGE、UPDATE、DELETE）产生的结果。例如，某文件包含行1（id=1,name='Alice'）和行2（id=2,name='Bob'），采用写时复制模式删除行2时，需删除该文件并将行1保留到新文件中。尽管这并非对表的实际变更，但变更日志表仍会记录以下两行：

id	name	_change_type
1	Alice	DELETE
1	Alice	INSERT

若需查看结转行，可按以下方式查询Spark变更表（SparkChangelogTable）：

```
SELECT * FROM spark_catalog.db.tbl.changes;
```

Pre/Post Update Images

若进行相关配置，此过程可计算更新前后镜像。更新前后镜像由一组删除行和插入行转换而来。标识列用于判断一条插入记录与一条删除记录是否指向同一行：若两条记录的标识列值相同，则视为同一行的前后状态。您可在表架构中设置标识字段，也可将其作为过程参数传入。

以下示例展示了使用标识列（id）计算更新前后镜像的过程：标识列值相同的行删除和行插入操作，会被视为单个更新操作。具体而言，假设存在以下两行记录：

id	name	_change_type
3	Robert	DELETE
3	Dan	INSERT

在此情况下，该过程会将更新前的行标记为UPDATE_BEFORE镜像，将更新后的行标记为UPDATE_AFTER镜像，最终生成以下更新前后镜像：

id	name	_change_type
3	Robert	UPDATE_BEFORE
3	Dan	UPDATE_AFTER

表统计信息

compute_table_stats

此过程用于计算特定表的不同值数量（NDV）统计信息。默认情况下，统计信息使用表当前快照为所有列计算。该过程可选择配置为计算特定快照和/或列子集的统计信息。

参数名称	是否必填	类型	描述
table	是	string	表名。
snapshot_id	否	string	要收集统计信息的快照ID。
columns	否	array	要收集统计信息的列。

输出

输出名称	类型	描述
statistics_file	string	此命令创建的统计信息文件路径。

示例

收集表my_table最新快照的统计信息

```
CALL catalog_name.system.compute_table_stats('my_table');
```

收集表my_table中ID为snap1的快照的统计信息

```
CALL catalog_name.system.compute_table_stats(table => 'my_table', snapshot_id => 'snap1' );
```

收集表my_table中ID为snap1的快照，针对列col1和col2的统计信息

```
CALL catalog_name.system.compute_table_stats(table => 'my_table', snapshot_id => 'snap1', columns => array('col1', 'col2'));
```

分区统计信息

compute_partition_stats

此过程从最后一个具有PartitionStatisticsFile的快照开始增量计算分区统计信息，直到给定的快照（如果未指定则使用当前快照），并将计算结果写入PartitionStatisticsFile。如果之前的分区统计信息文件不存在，则执行完整计算。它还会将PartitionStatisticsFile注册到表元数据中。

参数名称	是否必填	类型	描述
table	是	string	表名。
snapshot_id	否	string	要计算分区统计信息的快照ID。默认为当前快照ID。

输出

输出名称	类型	描述
partition_statistics_file	string	此命令创建的分区统计信息文件路径。

示例

收集表my_table最新快照的分区统计信息

```
CALL catalog_name.system.compute_partition_stats('my_table');
```

收集表my_table中ID为snap1的快照的分区统计信息

```
CALL catalog_name.system.compute_partition_stats(table => 'my_table', snapshot_id => 'snap1');
```

表复制

rewrite_table_path过程用于准备将Iceberg表复制到其他位置。

创建一个Iceberg表元数据文件的暂存副本，其中每个绝对路径源前缀都被替换为指定的目标前缀。这可以作为完全或增量复制Iceberg表到新位置的起点。

此过程仅暂存重写的元数据文件并准备要复制的文件列表。实际的文件复制不包含在此过程中。

参数名称	是否必填	默认值	类型	描述
table	是	-	string	表名。
source_prefix	是	-	string	要替换的现有前缀。
target_prefix	是	-	string	source_prefix的替换前缀。
start_version	否	表元数据日志中的第一个metadata.json	string	要重写的按时间顺序第一个metadata.json的名称或路径。
end_version	否	表元数据日志中的最新metadata.json	string	要重写的按时间顺序最后一个metadata.json的名称或路径。
staging_location	否	表元数据目录下的新目录	string	新重写的元数据文件的输出位置。

操作模式

- **完整重写**：完整重写将重写所有可访问的元数据文件（包括metadata.json、manifest lists、manifests 和 position delete files）， 并在 file_list_location 中返回所有可访问的文件。这是此过程的默认操作模式。
- **增量重写**：可选地，可以提供start_version和end_version来将范围限制为增量重写。增量重写仅重写在start_version和end_version之间添加的元数据文件，并且仅在file_list_location中返回此范围内添加的文件。

输出

输出名称	类型	描述
latest_version	string	此过程重写的最新元数据文件名称。
file_list_location	string	包含源路径到目标路径映射的CSV文件路径。

文件列表

该文件包含在start_version和end_version之间添加到表中的所有文件的复制计划。

对于每个文件，它指定：

- **源路径**：表中的原始文件路径，如果文件已被重写，则为暂存位置
- **目标路径**：带有替换前缀的路径

以下示例显示了三个文件的复制计划：

```
sourcepath/datafile1.parquet,targetpath/datafile1.parquet
sourcepath/datafile2.parquet,targetpath/datafile2.parquet
stagingpath/manifest.avro,targetpath/manifest.avro
```

示例

此示例完整重写my_table的元数据路径，从OBS中的源位置到OBS中的目标位置。它将在表元数据目录下的默认暂存位置生成一组新的元数据。

```
CALL catalog_name.system.rewrite_table_path(
    table => 'db.my_table',
    source_prefix => 'obs://bucket/path/to/source_table',
    target_prefix => 'obs://bucket/prefix/db.db/my_table' );
```

此示例增量重写my_table在元数据版本v2.metadata.json和v20.metadata.json之间的元数据路径，新元数据文件写入指定的暂存位置。

```
CALL catalog_name.system.rewrite_table_path(
    table => 'db.my_table',
    source_prefix => 'obs://bucketOne/prefix/db.db/my_table',
    target_prefix => 'obs://bucketTwo/prefix/db.db/my_table',
    start_version => 'v2.metadata.json',
    end_version => 'v20.metadata.json',
    staging_location => 'obs://bucketStaging/my_table' );
```

重写完成后，第三方工具（如DistCp）可以将新创建的元数据文件和数据文件复制到目标位置。

最后，可以使用注册表方法将复制的表注册到catalog中。

1.3.6 使用 spark 流式读写 Iceberg

Iceberg使用Apache Spark的DataSourceV2 API来实现数据源和目录（catalog）功能。Spark DSv2是一个不断演进的API，在不同版本的Spark中支持程度有所不同。

流式读取

Iceberg支持在Spark结构化流处理任务中读取增量数据，可以从指定的历史时间戳开始：

```
val df = spark.readStream
    .format("iceberg")
    .option("stream-from-timestamp", Long.toString(streamStartTimestamp))
    .load("database.table_name")
```

Iceberg只支持从追加（append）快照读取数据。覆盖（overwrite）快照无法处理，默认会抛出异常。可以通过设置streaming-skip-overwrite-snapshots=true来忽略覆盖操作。同样，删除（delete）快照默认也会抛出异常，可以通过设置streaming-skip-delete-snapshots=true来忽略删除操作。

流式写入

要将流式查询的结果写入Iceberg表，请使用DataStreamWriter：

```
data.writeStream
    .format("iceberg")
    .outputMode("append")
    .trigger(Trigger.ProcessingTime(1, TimeUnit.MINUTES))
    .option("checkpointLocation", checkpointPath)
    .toTable("database.table_name")
```

Iceberg支持append和complete两种输出模式：

- append：将每个微批次的行追加到表中
- complete：用每个微批次的内容替换整个表的内容

在启动流式查询之前，请确保已创建表。请参阅SQL创建表文档了解如何创建Iceberg表。

Iceberg不支持实验性的连续处理模式，因为该模式没有提供"提交"输出的接口。

分区表

Iceberg要求在写入数据之前按分区对每个任务的数据进行排序。在Spark中，任务按Spark分区进行划分。对于批处理查询，建议进行显式排序以满足要求，但这种方法会增加额外延迟，因为重新分区和排序对于流式工作负载来说是重量级操作。为了避免额外延迟，您可以启用fanout writer来消除这一要求。

```
data.writeStream
    .format("iceberg")
    .outputMode("append")
    .trigger(Trigger.ProcessingTime(1, TimeUnit.MINUTES))
    .option("fanout-enabled", "true")
    .option("checkpointLocation", checkpointPath)
    .toTable("database.table_name")
```

Fanout writer为每个分区值打开文件，并在写入任务完成之前不会关闭这些文件。应避免在批处理写入中使用fanout writer，因为对输出行进行显式排序对于批处理工作负载来说成本较低。

流式表的维护

流式写入会快速创建新的表版本，产生大量表元数据来跟踪这些版本。通过调整提交频率、过期旧快照和自动清理元数据文件来维护元数据是非常推荐的。

调整提交频率

高频率的提交会产生数据文件、清单文件和快照，从而增加维护工作量。建议至少设置1分钟的触发间隔，并根据需要增加间隔。

过期旧快照

写入表的每个批次都会产生一个新的快照。Iceberg会跟踪表元数据中的快照，直到它们过期。频繁提交会导致快照快速累积，因此强烈建议对流式查询写入的表进行定期维护。快照过期是移除不再需要的元数据和任何数据文件的过程。默认情况下，该过程会过期超过五天的快照。

合并小文件

流式写入的数据量通常很小，这会导致表元数据跟踪大量小文件。将小文件合并成大文件可以减少表所需的元数据，并提高查询效率。Iceberg和Spark提供了rewrite_data_files存储过程。

重写清单文件

为了优化流式工作负载的写入延迟，Iceberg可以使用"快速追加"方式写入新快照，而不会自动合并清单文件。这可能会导致大量小清单文件。Iceberg可以重写清单文件数量以提高查询性能。Iceberg和Spark提供了rewrite_manifests存储过程。

1.4 Iceberg 开发常见问题

1.4.1 如何处理并发提交数据冲突

在并发写入Iceberg表时，通常会遇到两类冲突：并发提交数据冲突、乐观锁并发提交元数据冲突。

本文介绍并发提交数据冲突。

当出现以下并发作业时，可能会产生数据冲突，从而导致作业失败：

- 对同一分区数据并发执行DML操作，例如对同一分区执行Update/Delete/Insert Overwrite/Insert操作。
- Compaction与DML操作并发执行。

下文描述当前各引擎支持的Iceberg操作类型可能会遇到的并发数据冲突。

- 当执行Insert Overwrite操作时，可能会出现以下冲突：
  - data数据冲突：如果为分区表，在检测数据冲突时，识别到insert overwrite涉及的分区有其他操作提交后产生的新的data文件，则检测为数据冲突发生；如果为非分区表，在检测数据冲突时，识别到有任何其他操作提交后产生的新的data文件，则检测为数据冲突发生。
  - delete数据冲突：如果为分区表，在检测数据冲突时，识别到insert overwrite涉及的分区有其他操作删除了data文件、新增了delete文件，则检测为数据冲突发生；如果为非分区表，在检测数据冲突时，识别到有其他操作删除了data文件、新增了delete文件，则检测为数据冲突发生。
- 当执行Delete操作时，可能会出现以下冲突：

  data数据冲突：在检测数据冲突时，识别到delete file涉及的data file被其他操作删除了，则检测为数据冲突发生。
- 当执行Update操作时，可能会出现以下冲突：
  - data数据冲突：在检测数据冲突时，识别到delete file涉及的data file被其他操作删除了，则检测为数据冲突发生。

- delete数据冲突：在检测数据冲突时，识别到有其他操作对update操作涉及的分区产生了新的delete file，则检测为数据冲突发生。

表 1-23 并发冲突矩阵

本次操作/已提交操作	insert	insert overwrite	delete
insert	无冲突	无冲突	无冲突
insert overwrite	可能会发生冲突	可能会发生冲突	可能会发生冲突
delete	无冲突	可能会发生冲突	无冲突
update	无冲突	可能会发生冲突	可能会发生冲突

说明

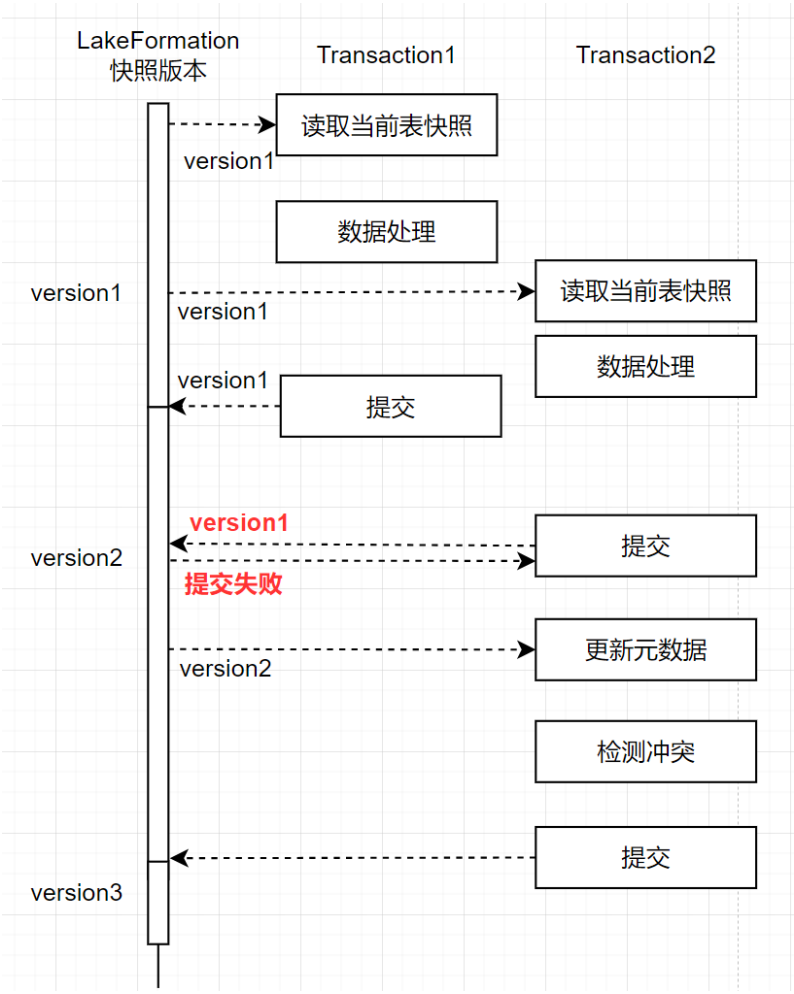
上述并发冲突矩阵是指，当本次操作开始后、提交前，有其他操作开始并完成了提交操作，则本次操作提交时是否会发生冲突。

1.4.2 如何处理乐观锁并发提交元数据冲突

在并发写入Iceberg表时，通常会遇到两类冲突：并发提交数据冲突、乐观锁并发提交元数据冲突。

本文介绍乐观锁并发提交元数据冲突。

乐观锁并发提交元数据冲突的产生原因：多个修改作业（包括Insert、Update、Delete、MergeInto以及表服务操作）同时在LakeFormation上提交导致的并发提交冲突。例如：



上图中，Transaction1和Transaction2分别为两个DML事务，Transaction1基于元数据版本version1进行数据处理和提交，并将元数据版本更新至version2，而Transaction2在Transaction1提交前已经基于元数据版本version1开始了数据处理，在首次提交时，LakeFormation拒绝提交，因为此时元数据已经更新为version2，需要由Transaction2更新元数据后，再次提交。

在这种场景下，如果并发量非常大，或者某个事务做冲突检测的时间过久，则会陷入“提交冲突->更新元数据->检测冲突->再次提交”的死循环，导致业务冲突报错。

为解决上述问题，Iceberg引入四个参数，具体参数说明请参见**表1-24**。

表 1-24 参数说明

参数名	说明	高并发场景下建议值	表服务场景下建议值
commit.retry.num-retries	最大重试次数	10	4
commit.retry.min-wait-ms	提交失败后，最小提交等待时间	100	1000
commit.retry.max-wait-ms	提交失败后，最大提交等待时间	10000	60000

参数名	说明	高并发场景下建议值	表服务场景下建议值
commit.retry.total-timeout-ms	提交过程最长执行时间。	1800000	1800000

1.4.3 使用 LakeFormation 进行表维护

开通内表服务

LakeFormation表服务配置在创建Iceberg表和编辑Iceberg表时，支持配置表服务选项（仅独享型实例支持），存量实例如需开通表服务功能需执行以下步骤：

步骤1 登录**LakeFormation管理控制台**。

步骤2 在左侧下拉框中选择待操作的LakeFormation实例。

步骤3 在“总览”页面查看实例“总览”中“是否已开通内表服务”的参数值。

如果当前实例已开通，则无“开通内表服务”按钮。

步骤4 如果需要为当前实例开通内表服务，请单击页面右上角“开通内表服务”，勾选操作影响后单击“确定”。

----结束

说明

开通内表服务后，可以创建内表和内部函数，如果需要关闭内表服务，需要彻底清除内表，删除实例时会自动关闭内表服务。

成功开通内表服务功能后，需配置表服务以开启压缩、快照及孤立文件删除功能。

开启压缩

是否将小数据文件合并成更大、更高效的文件，以优化表格性能。

勾选后，需要配置以下压缩参数：

- **压缩策略**：
  - Binpack：将小文件合并成大文件，通常目标大小为100MB以上，同时执行任何待处理的删除操作。大多数使用场景默认和推荐使用该压缩策略。
  - Sort：根据指定的列对数据进行组织，这些列在压缩过程中按层级排序，从而提高过滤操作的查询性能。当查询操作需要频繁对特定列进行过滤时，建议使用此策略。使用前需在列存表属性中使用“sort-order”表属性定义排序顺序。
  - Z-Order：通过将多个属性融合为单个标量值来优化数据组织，该标量值可用于排序，从而实现跨多个维度的高效查询。当您需要同时跨多个维度查询数据时，推荐使用此策略。使用前需在Apache Iceberg表的属性中使用“sort-order”表属性定义排序顺序。
- **最小输入文件**：触发合并的最小文件数量。最小值3，最大值100，默认值5。
- **合并后文件大小**：合并后期望的文件大小，单位：MB。最小值128，最大值2048，默认值512。

开启快照

是否通过删除元数据开销中的旧快照和使不再需要的文件过期来优化表存储。

勾选后，需要配置以下快照保留参数：

- **快照保留时间**：保存时间，单位：小时。最小值24，最大值1440，默认值72。
- **最少保留的快照数**：最少保留的Snapshot数量。最小值1，最大值100，默认值10。
- **删除关联文件**：是否删除与已过期快照关联的文件。

  快照保留策略在默认配置下会同时删除对应快照的元数据和数据文件，取消勾选“删除关联文件”后可以仅删除数据文件，保留元数据信息。
- **执行间隔**：任务执行间隔，单位：小时。最小值1，最大值168，默认值24。

开启孤立文件删除

是否定期自动清理孤立文件。

勾选后，需要配置以下孤立文件删除参数：

- **孤立文件保留时间**：孤立文件的保留时间，单位：小时。最小值24，最大值1440，默认值72。
- **执行间隔**：任务执行间隔，单位：小时。最小值1，最大值168，默认值24。

说明

- 孤立文件删除策略：会删除没有被元数据引用，被评估为孤立文件的部分。
- 为了保证孤立文件判定的准确性，请确保当前表路径下的文件全都是当前表的数据文件，否则该文件可能会被判定为孤立文件并被删除，该删除操作不可恢复。

1.4.4 使用引擎进行表维护

建议一：定期执行快照过期清理

原因：每次写/更新/删除/Upsert/压缩都会产生新快照，不清理会导致存储持续增长。

实现方式：

```
-- 清理超过7天的快照，保留最近100个
CALL spark_catalog.system.expire_snapshots(
   table => 'db.table',
   older_than => CURRENT_TIMESTAMP - INTERVAL '7' DAYS,
   retain_last => 100,
   stream_results => true
  -- 防止大数据量导致Driver OOM );
  -- 清理特定快照
CALL spark_catalog.system.expire_snapshots(
   table => 'db.table',
   snapshot_ids => ARRAY(123)
 );
  -- 清理时同时清理未使用的元数据
CALL spark_catalog.system.expire_snapshots(
   table => 'db.table',
   older_than => TIMESTAMP '2024-01-01 00:00:00',
   clean_expired_metadata => true
 );
```

注意：

- 分支和标签默认永不过期，需通过表属性history.expire.max-ref-age-ms配置。
- 建议设置stream_results=>true，避免大数据量时Driver OOM。

Aura实现方式：参考**iceberg_expire_snapshots**表服务。具体语法请参考：**Iceberg表服务函数**。

建议二：定期清理孤立文件

原因：清理不再被任何元数据文件引用的"孤儿"文件，释放存储空间。

实现方式：

```
-- 先干跑查看待清理文件（推荐生产环境先执行）
CALL spark_catalog.system.remove_orphan_files(
  table => 'db.table',
  dry_run => true
);
-- 实际清理（清理超过7天的孤立文件）
CALL spark_catalog.system.remove_orphan_files(
  table => 'db.table',
  older_than => CURRENT_TIMESTAMP - INTERVAL '7' DAYS
);
-- 清理特定目录下的孤立文件
CALL spark_catalog.system.remove_orphan_files(
  table => 'db.table',
  location => 'table_location/data'
);
```

Aura实现方式：参考**iceberg_remove_orphanfiles**表服务。具体语法请参考：**Iceberg表服务函数**。

建议三：定期执行数据文件压缩

原因：小文件过多会导致大量元数据开销和文件打开成本，影响查询性能。

实现方式：

```
-- 默认binpack策略（合并小文件）
CALL spark_catalog.system.rewrite_data_files('db.table');
-- 排序策略（按指定列排序）
CALL spark_catalog.system.rewrite_data_files(
   table => 'db.table',
   strategy => 'sort',
   sort_order => 'id DESC NULLS LAST, name ASC NULLS FIRST'
 );
-- Z-Order优化（多列交织排序，提升多列查询性能）
CALL spark_catalog.system.rewrite_data_files(
   table => 'db.table',
   strategy => 'sort',
   sort_order => 'zorder(c1,c2,c3)'
 );
-- 启用部分提交和清理dangling deletes
CALL spark_catalog.system.rewrite_data_files(
   table => 'db.table',
   options => map(
     'min-input-files', '2',
     'remove-dangling-deletes', 'true',
     'partial-progress.enabled', 'true'
   )
);
```

关键参数：

- target-file-size-bytes: 目标文件大小（默认512MB）

- min-input-files：触发重写的最小文件数（默认5）
- rewrite-job-order：排序策略（bytes-asc/desc,files-asc/desc）

Aura实现方式：

Aura中通过OPTIMIZE语法对目标表的指定数据进行重写。

```
OPTIMIZE table_name REWRITE DATA
   [ WITH OPTIONS (option_key = option_value [, ...]) ]
   [ WHERE condition ]
   [ ORDER BY target_list [ ASC | DESC ] ]
```

参数说明：

- table_name：目标表的名字。取值范围：已存在的表名。
- condition：一个返回boolean值的表达式，用于判断哪些文件需要被重写。
- target_list：数据重写的排序键。
- option_key、option_value：数据重写的配置选项。

建议四：定期重写 Manifest 文件

原因：优化查询规划性能，manifest中的数据文件按分区规范字段排序。

实现方式：

```
-- 重写manifest
CALL spark_catalog.system.rewrite_manifests('db.table');
-- 指定分区规范
CALL spark_catalog.system.rewrite_manifests(
   table => 'db.table',
   spec_id => 1
);
```

Aura实现方式：参考**iceberg_rewrite_manifests**表服务。具体语法请参考：**Iceberg表服务函数**。

建议五：定期重写 Position Delete 文件

原因：压缩小position delete文件，清理dangling deletes（指向已不存在数据文件的删除记录）。

实现方式：

```
-- 压缩并清理dangling deletes
CALL spark_catalog.system.rewrite_position_delete_files('db.table');
-- 强制重写所有文件
CALL spark_catalog.system.rewrite_position_delete_files(
   table => 'db.table',
   options => map('rewrite-all', 'true')
);
```

建议六：为重要版本创建标签

原因：便于快速回滚到特定版本，避免误操作后无法恢复。

实现方式：

```
-- 切换到指定快照
CALL spark_catalog.system.set_current_snapshot(
   table => 'db.table',
   snapshot_id => 123
```

```
);
-- 切换到标签
CALL spark_catalog.system.set_current_snapshot(
  table => 'db.table',
  ref => 'my_tag'
);
-- 回滚到指定快照
CALL spark_catalog.system.rollback_to_snapshot('db.table', 1);
-- 按时间回滚
CALL spark_catalog.system.rollback_to_timestamp(
  'db.table',
  TIMESTAMP '2024-06-30 00:00:00.000'
);
```

Aura实现方式：参考**iceberg_set_current_snapshot**、**iceberg_rollback_to_snapshot**、**iceberg_rollback_to_timestamp**表服务。具体语法请参考：**Iceberg表服务函数**。

建议七：迁移前使用快照表测试

原因：创建轻量级临时副本用于测试，不影响源表，避免迁移风险。

实现方式：

```
-- 创建快照表
CALL spark_catalog.system.snapshot('db.source_table', 'db.snapshot_table');
```

注意：快照表不是数据文件的唯一所有者，无法执行expire_snapshots物理删除文件。

建议八：正式迁移使用 migrate

原因：将现有Hive/Spark表替换为Iceberg表，保留原表作为备份。

实现方式：

```
-- 迁移并添加属性
CALL spark_catalog.system.migrate(
  'db.old_table',
  map('foo', 'bar')
);
-- 不保留备份
CALL spark_catalog.system.migrate(
  table => 'db.old_table',
  drop_backup => true
);
```

建议九：计算表统计信息

原因：收集列的NDV（不同值数量）统计信息，优化查询计划。

实现方式：

```
-- 计算全表统计
CALL spark_catalog.system.compute_table_stats('db.table');
-- 指定快照和列
CALL spark_catalog.system.compute_table_stats(
  table => 'db.table',
  snapshot_id => 'snap1',
  columns => array('col1', 'col2')
);
```

建议十：计算分区统计信息

原因：增量收集分区统计信息，提升分区裁剪效率。

实现方式：

```
-- 计算分区统计
CALL spark_catalog.system.compute_partition_stats('db.table');
-- 指定快照
CALL spark_catalog.system.compute_partition_stats(
   table => 'db.table',
   snapshot_id => 'snap1'
);
```

2 Lance 开发指南

2.1 Lance 简介

2.1.1 Lance 功能介绍

Lance是一种基于Apache Arrow生态构建的高性能列式数据存储格式，由LanceDB团队主导开发，旨在实现接近Arrow内存格式的数据访问性能，同时支持事务管理、索引和增量更新。Lance不仅适用于离线批处理分析，还重点支持在线查询、向量检索和人工智能应用，是一种面向AI与数据湖融合场景的新型存储格式。

主要特点

- Lance底层数据结构与Apache Arrow高度兼容，采用标准化的列式存储方式，支持基本类型、复杂嵌套类型以及向量数据结构。数据从磁盘加载到内存后可直接被Arrow计算引擎处理，避免了频繁的数据转换与拷贝操作。
- Lance内置多版本并发控制机制，支持数据快照管理、事务提交与回滚操作，实现读写隔离和历史版本回溯查询功能，有效保障数据一致性与可靠性。
- Lance针对高维向量数据进行了专门优化，支持向量字段的高效存储与索引构建，可用于近似最近邻搜索与语义检索等应用场景，为人工智能系统提供统一的数据底座。
- Lance支持持续追加写入与增量更新操作，并通过后台压缩与合并机制优化存储结构，提高查询效率。
- Lance核心组件基于Rust语言实现，并提供Python、Rust等多语言接口，便于在不同系统架构和应用环境中进行集成与部署。
- Lance支持本地文件系统及对象存储系统访问模式，具备范围读取、延迟加载和元数据缓存等机制，适用于云计算与数据湖环境下的大规模数据管理需求。

主要优势

- 基于Arrow内存格式和零拷贝机制，Lance在数据扫描、过滤、投影和向量检索等操作中具有接近内存级别的访问性能，显著优于传统列式存储格式。
- 与仅支持追加写入的Parquet格式不同，Lance支持对已有数据进行更新、删除与合并操作，适用于动态变化的数据管理场景。
- Lance可在同一数据表中同时存储结构化字段与向量字段，实现多模态数据的统一管理与分析处理，为推荐系统、检索系统和生物信息分析等应用提供便利。
- Lance提供面向存储引擎和查询引擎的底层接口，易于与数据库系统、分布式计算框架和机器学习平台进行深度集成，降低系统开发与维护成本。
- Lance以文件形式进行数据组织，无需额外部署独立服务，系统结构相对简单，便于在科研环境和生产系统中快速部署与应用。

组织形式

- Lance格式概述：Lance表格式是Lance数据存储体系中的核心组织形式，用于对结构化数据与向量数据进行统一管理。该表格式以Apache Arrow数据模型为基础，通过分层存储结构与元数据管理机制，实现对大规模数据的高效存储、访问与版本控制。每个Lance表在逻辑上表现为一张二维关系表，在物理上则由多个数据片段和元数据文件共同组成。
- Lance表逻辑结构：在逻辑层面，Lance表采用标准的关系型表模型，由行和列构成。每一列对应一种数据类型，每一行表示一条完整记录。表结构通过Schema进行统一定义，描述字段名称、字段类型及嵌套关系等信息。Lance表支持多种数据类型，包括整数、浮点数、字符串、二进制数据、列表、结构体以及高维向量类型，能够满足多模态数据的存储需求。
- Lance表物理存储结构：在物理层面，Lance表采用分段式存储架构，将数据划分为多个独立的数据片段（Fragment）进行管理。每个片段内部以列式方式存储数据，不同列的数据分别组织为连续块，便于高效扫描和向量化计算。每个数据片段对应独立的数据文件，同时配套存储列统计信息、偏移索引和校验信息，用于加速查询与保障数据完整性。
- Schema和元数据管理：Lance表通过集中式元数据文件维护表结构信息、数据版本信息与片段索引关系。Schema描述字段类型与布局方式，是表访问和计算的基础依据。当表结构发生变化时，Lance通过Schema演化机制支持字段新增、删除和类型扩展等操作，并通过版本控制机制保证历史数据的兼容性。
- 版本控制和快照机制：Lance表采用多版本并发控制机制，对每一次数据写入或更新操作生成新的快照版本。每个版本对应一组完整的数据片段引用关系，从而实现数据状态的持久化保存。基于快照机制，系统支持历史版本查询、数据回滚和时间点恢复等功能，有效提升数据管理的安全性与可靠性。

支持的功能

- 富有表现力的混合搜索

  Lance支持强大的混合搜索，将向量相似性搜索、全文搜索和SQL分析结合在同一数据集上。所有查询类型都由Lance规范中相应的二级索引加速。可以对嵌入向量执行语义搜索，对关键词执行BM25搜索，并应用复杂的SQL谓词，所有操作都在同一张表上通过统一接口完成。

- 极速随机访问

  与Parquet或Iceberg相比，Lance的随机访问速度快100倍。与传统格式不同，即使在整个数据集中随机访问分散的行，Lance也能保持高性能。借助高度优化的文件格式，加上表级别的高效行寻址和二级索引，可以跨多个文件即时访问单条记录，非常适合实时ML服务、随机采样和交互式应用。

- 原生多模态数据支持

  将图像、视频、音频、文本和嵌入与传统的表格数据存储在同一个统一格式中。Lance的blob编码通过惰性加载（lazy loading）高效处理大型二进制对象，而优化的向量存储则加速了相似度搜索。非常适合AI/ML工作负载，在这些负载中，需要将原始数据、机器学习特征、生成的标题和嵌入存储在一起，以用于多模态检索和生成式AI（GenAI）工作流。

- Schema演进

  大多数开放表格格式中的模式演进仅涉及元数据，速度很快。但当尝试回填现有行中的列值时，通常需要进行全表重写。Lance支持数据演进（带回填的高效模式演进），使其非常适合ML特征工程、嵌入向量和媒体内容管理。添加带有数据的新列就像向Lance表中写入新的Lance文件一样简单，无需重写整个数据集。

2.1.2 Lance 表目录介绍

Lance表目录是最初创建数据集的位置。每个Lance表都有且仅有一个根目录，它作为数据集文件的主要存储位置。Lance表根目录下包含标准子目录结构（data/、_versions/、_transactions/、_deletions/、_indices/、_refs/、tree/），用于组织数据集的文件。

```
{table_name}/
   data/
      *.lance                  -- 列数据文件
   _versions/
      *.manifest               -- Manifest 文件 (每个版本一个)
      latest_version_hint.json  -- 当前最新版本信息
   _transactions/
      *.txn                    -- 事务文件（提交协调）
   _deletions/
      *.arrow                  -- 删除向量文件 (Arrow格式)
      *.bin                    -- 删除向量文件 (位图格式)
   _indices/
      {UUID}/
         ...                   -- 索引内容 (按索引类型不同)
   _refs/
      tags/
         *.json                -- Tag 元数据
      branches/
         *.json                -- Branch 元数据
   tree/
      {branch_name}/
         ...                   -- Branch 列数据文件
```

- **数据文件**

  模式：data/{uuid-based-filename}.lance

  数据文件使用基于UUID的文件名，针对S3吞吐量进行了优化。文件名由UUID（16 字节）生成，将前3个字节转换为 24字符的二进制字符串，其余13个字节转换为26字符的十六进制字符串，最终形成50字符的文件名。二进制前缀（而非十六进制）提供了每个字符的最大熵，使S3的内部分区能够快速识别访问模式并相应地扩展，从而最大限度地减少限流。

  示例：data/10110010110101001101011 0a1b2c3d4e5f6g7h8i9j0.lance

- **清单文件列表**

  清单文件存储在 _versions/ 目录中，其命名方案支持原子提交。

  - V1：{version}.manifest - 单调递增的版本号（例如 1.manifest, 2.manifest）
  - V2：{u64::MAX - version:020}.manifest - 反向排序的字典顺序，实现了对最新版本的高效发现（例如V2第一个版本为 18446744073709551614.manifest）

_versions/latest_version_hint.json 以JSON格式记录最新提交的版本。

latest_version_hint.json文件内容如下：

```
{"version": 42}
```

它的存在是为了加速在列表操作成本高昂的存储系统上进行最新版本的发现：读取器可以读取提示，并通过HEAD请求探测更高的版本，而不是列出整个目录；如果提示缺失或过期，则回退到完整列表操作。该提示纯粹是一种优化手段。它始终可以安全删除，绝不会影响正确性，且可以被不识别它的读取器忽略。写入器可以选择不写入此文件。

- **事务文件**

  模式：_transactions/{read_version}-{uuid}.txn

  其中 read_version 是构建该事务时所依据的表版本。

  示例：_transactions/5-550e8400-e29b-41d4-a716-446655440000.txn

- **删除文件**

  模式：_deletions/{fragment_id}-{read_version}-{id}.{extension}

  删除文件使用两种扩展名：用于Arrow IPC格式（稀疏删除）的.arrow和用于Roaring位图格式（密集删除）的.bin。

  示例：_deletions/42-10-a1b2c3d4.arrow

- **索引文件**

  每个索引的内容存储在基准路径 (base path) 下的 _indices/{UUID} 目录中。将此位置称为**索引目录**。索引目录中存储的实际内容取决于索引类型。这些可以是索引实现定义的任意文件。然而，它们通常由包含索引数据结构的Lance文件组成。这允许重用现有的Lance文件格式代码来读写索引数据。

2.1.3 Lance 索引介绍

Lance将索引视为独立的、冗余的数据结构，分层构建在表行标识符之上。这使文件格式免于内置搜索结构，并允许索引格式独立于表布局演进。Lance支持三大类索引：标量索引、向量索引和系统索引。

标量索引

标量索引加速对整数、时间戳和字符串等标量数据类型的查询。Lance支持的标量索引请参考**表2-1**。

表 2-1 标量索引类型

索引类型	存储文件	加速查询	说明
Zone Map	zonemap.lance	Equals、Range、IsIn、IsNull	将数据分成固定大小的区域，维护min/max/null_count统计信息，用于谓词下推和扫描裁剪。可能包含误报，需要重新检查。
BTree	page_lookup.lance + page_data.lance	Equals、Range、IsIn、IsNull	两级结构，上层BTree缓存于内存，下层子索引存储排序值和行ID。平衡内存结构和磁盘结构。

索引类型	存储文件	加速查询	说明
Bitmap	bitmap_page_lookup.lance	Equals、Range、IsIn、IsNull	使用位数组表示值的存在或缺失，适用于低基数列，提供极快的查询性能。
Bloom Filter	bloomfilter.lance	Equals、IsIn、IsNull	概率数据结构，允许快速成员测试。空间高效，可能有误报但无误漏。使用Split Block Bloom Filter（SBBF）实现，优化SIMD操作。
Full Text Search	tokens.lance + docs.lance + invert.lance + metadata.lance	contains_tokens、match、phrase、boolean、multi_match、boost	倒排索引，将词项映射到包含它们的文档。支持BM25评分、短语查询和多种分词器。
Label List	bitmap_page_lookup. lance	array_has / array_contains、array_has_all、array_has_any	标签列表索引针对每行包含多个标签或标记的列进行了优化。它们使用底层的位图索引，针对多值列提供高效的基于集合的查询。
N-gram	ngram_postings.lance	contains	N-gram 索引将文本分解为重叠的序列（三元组），以实现高效的子字符串匹配。通过在应用 ASCII 折叠和转为小写后，对文本中所有 三元组序列进行索引，从而提供快速的文本搜索功能。
R-Tree	page_data.lance	Intersects、Contains、Within、Touches、Crosses、Overlaps、Covers、CoveredBy、IsNull	基于边界框（Bounding Boxes）构建以组织数据。该索引旨在加速基于矩形的修剪操作

向量索引

向量索引专用于高维嵌入的近似最近邻搜索（ANN）。Lance将每个向量索引分为3个部分：聚类（Clustering）、子索引（Sub-Index）和量化（Quantization）。

- 聚类：将所有向量划分为不同的不相交簇。Lance目前支持使用倒排文件（IVF）作为主要聚类机制，使用k-means聚类算法将向量分区。
- 子索引：决定向量如何组织以进行搜索。支持FLAT（精确搜索，无近似）和HNSW（分层可导航小世界图，快速近似搜索）。
- 量化：决定向量如何存储和压缩。支持Product Quantization（PQ，乘积量化）、Scalar Quantization（SQ，标量量化）、RabitQ（RQ，随机旋转二值量化）和FLAT（不量化，保留原始向量）。

索引类型通常由 {clustering}_{sub_index}_{quantization} 构成。若子索引为 FLAT，则通常省略该项，简记为 {clustering}_{quantization}。

表 2-2 向量索引类型

索引类型	名称	说明
IVF_PQ	倒排文件+乘积量化	结合IVF聚类与PQ压缩，实现高效存储和搜索
IVF_HNSW_SQ	倒排文件+HNSW+标量量化	使用IVF粗聚类和HNSW细粒度搜索，配合标量量化
IVF_SQ	倒排文件+标量量化	结合IVF聚类与标量量化，实现平衡压缩
IVF_RQ	倒排文件+RabitQ	结合IVF聚类与RabitQ，使用二值量化实现极限压缩
IVF_FLAT	倒排文件+无量化	使用IVF聚类与精确向量存储，在簇内进行精确搜索

系统索引

系统索引是支持内部表维护和行标识符解析的辅助结构，不由最终用户直接查询。例如Fragment Reuse Index支持压缩后的高效重映射。

- 碎片重用索引 (Fragment Reuse Index, FRI)用于在压缩 (compaction) 和数据集更新期间优化碎片操作。
- MemWAL索引作为所有MemWAL元数据的集中式结构。它存储配置（分片规格、需维护的索引）、合并进度以及分片状态快照。

2.1.4 Lance 表 Schema 演进介绍

Schema描述了Lance表的结构，包含所有字段、数据类型及元数据。它采用逻辑类型系统，将数据类型表示为映射到Apache Arrow的字符串。每个字段拥有唯一标识符（字段 ID），支持在不破坏引用的情况下重命名或重新排序字段，从而实现强大的Schema演进与版本跟踪。字段ID确保了即使Schema随时间变化，数据文件也能被正确解释。在创建表时，字段ID会按照深度优先的顺序，从0开始，为所有字段分配。

- 添加列：分配新的字段ID并添加到Schema。
- 删除列：从Schema中移除字段，其ID在某些系统中可以重用。
- 重命名列：更改字段名称，ID保持不变。
- 重新排序列：更改 Schema 中的字段顺序，ID 保持不变。
- 类型演进：可以更改数据类型。根据类型更改方式，这可能需要重写数据中的列。

2.1.5 Lance 表支持的数据类型介绍

表 2-3 Lance 支持的数据类型

Lance数据类型	说明
null	空类型（无值）
bool	布尔值（true/false）
int8, int16, int32, int64	有符号8位、16位、32位、64位整数
uint8, uint16, uint32, uint64	无符号8位、16位、32位、64位整数
halffloat	IEEE 754半精度浮点数（16位）
float	IEEE 754单精度浮点数（32位）
double	IEEE 754双精度浮点数（64位）
string	变长UTF-8编码字符串
large_string	变长UTF-8字符串（支持大偏移量）
binary	变长二进制数据
large_binary	变长二进制数据（支持大偏移量）
decimal:128:p:s	decimal:<位宽>:<精度>:<标度>。精度（p）为总位数（Decimal128最多38位），小数位数（s）为小数点后的位数（0 ≤ S ≤ P）
decimal:256:p:s	decimal:<位宽>:<精度>:<标度>。精度（p）为总位数（Decimal256最多76位），小数位数（s）为小数点后的位数（0 ≤ S ≤ P）
date32:day	日期（自纪元以来的天数）
date64:ms	日期（自纪元以来的毫秒数）
time32:s, time32:ms	时间（自午夜以来的秒数），时间（自午夜以来的毫秒数）
time64:us, time64:ns	时间（自午夜以来的微秒数），时间（自午夜以来的纳秒数）
timestamp:unit:tz	timestamp:<单位>:<时区>，单位可选s（秒）、ms（毫秒）、us（微秒）、ns（纳秒）。时区为IANA时区字符串（如UTC、America/New_York）或-表示无时区
duration:s, duration:ms, duration:us, duration:ns	时长（秒），时长（毫秒），时长（微秒），时长（纳秒）
struct	嵌套类型
list 或 list.struct（如果元素为 Struct）	变长值列表，变长结构体值列表

Lance数据类型	说明
large_list 或 large_list.struct	变长列表（支持大偏移量），变长结构体值列表（大偏移量）
fixed_size_list:type:size	特别适用于向量嵌入，如fixed_size_list:float:128 表示 128 个浮点数的定长列表
fixed_size_binary:size	如fixed_size_binary:16用于MD5哈希， fixed_size_binary:32用于SHA-256哈希
dict:value_type:key_type:bool	字典值的类型 dict:<值类型>:<键类型>:<是否有序>， dict:string:int16:false 表示使用 int16 键的字典编码字符串
map	键值对（目前仅支持无序键）

2.1.6 Lance Namespace 和 LakeFormation Catalog 介绍

Lance Namespace

Lance Namespace是用于逻辑隔离、层级化组织表/数据集的命名空间，提供命名唯一性与权限管控，支持多团队、多项目的数据资产分类管理。规范定义了一个标准化的接口，用于目录交互，如表发现、解析表位置以及协调提交。它通过一个名为LanceNamespace的统一接口，对LakeFormation Catalog等实现进行了抽象。

LakeFormation Catalog

LakeFormation Catalog用于管理表集合，并提供表发现、管理和事务协调功能。

LakeFormation作为成熟的湖仓治理工具，具备元数据注册、细粒度权限控制、数据访问审计等核心能力。

- 支持元数据统一存储在LakeFormation并可见。
- 支持多项关键功能，覆盖库、表、位置管理及元数据转换，table支持元数据管理、属性配置及持久化操作等特性。
- 使用LakeFormation提供的原子语义结合Lance自身的external manifest事务提交机制，实现Lance的事务提交。
- 打通Lance与LakeFormation之间所有的权限连接，包括LakeFormation用户层、OBS层和引擎层。

2.2 Lance on Aura

2.2.1 使用 Lance 前准备

在使用 Aura 对 Lance 格式的数据表执行读写等操作之前，您必须确保已获得LakeFormation的相应访问权限，以避免因权限不足导致操作失败。

关于权限的具体介绍，请参见**用户及权限管理**。

2.2.2 Lance 配置与类型

Aura 类型和 Lance 类型对应关系

表 2-4 Aura 类型对应的 Lance 类型

Aura	Lance
smallint	int16
tinyint	int8
integer	int32
bigint	int64
numeric	Decimal128
decimal	Decimal128
real	float32
float4	float32
double precision	float64
float8	float64
boolean	boolean
char	string
varchar	string
text	string
date	date
timestamp without time zone	timestamp without time zone
bytea	binary
array	list
struct	struct

2.2.3 Lance on Aura DDL 语法说明

使用限制

Lance表不支持创建partition和bucket列。

创建 Lance 表

用户使用CREATE TABLE语法创建Lance表，需指定STORE AS Lance。具体语法，请参见**CREATE TABLE**。用户创建的Lance表元数据存储于LakeFormation中，主要包含表名、表路径、表当前版本号和表参数，Lance表实际数据文件存储于OBS桶中。

示例：

```
CREATE TABLE lance_tbl(
col1 int,
col2 array<double>,
col3 bytea
)
TABLEPROPERTIES (
'enable.blob.col3'='true',
'arrow.fixed-size-list.col2' = '128'
) STORE AS Lance;
```

TABLEPROPERTIES支持配置的表参数：

表 2-5

参数	类型	说明
enable.blob.col_name	布尔值	指定col_name列为BLOB列，仅支持bytea类型列，Lance中对于BLOB的介绍请参见：**Lance BLOB格式介绍** 。
arrow.fixed-size-list.col_name	整型值	范围为1~2147483647，指定col_name使用Arrow中的FixedSizeList存储为定长向量，仅支持浮点数类型的一维数组。
max.rows.per.file	整型值	范围为1~9223372036854775807，默认值为1048576，指定单个文件写入最大行数。
max.bytes.per.file	整型值	范围为1~9223372036854775807，默认值为90GiB，指定单个文件写入最大字节数。
lance.encoding.compression	整型值	字符串，可指定为zstd、lz4、fsst三种压缩算法，其中zstd支持配置压缩等级，取值范围为0-22，如果未指定压缩等级则默认压缩等级为3，默认不启用压缩。三种压缩方式的介绍，请参见**Lance支持压缩方式介绍**。
lance.batch.readahead	整型值	范围为1~9223372036854775807，默认值为4，指定预读的Batch数目。
lance.fragment.readahead	整型值	范围为1~9223372036854775807，默认值为32，指定预读的Fragment数目。
lance.storage.version	字符串	可取值为0.1、2.0、2.1、2.2，指定存储的Lance格式版本，默认值为2.2。
lance.block.size	整型值	范围为1~34359738368，指定向存储层发起单次I/O请求的最小字节数，默认值为1MB。
metadata.cache.size	整型值	范围为1~34359738368，指定元数据缓存大小字节数，默认值为1GiB。

参数	类型	说明
io.buffer.size	整型值	范围为1~34359738368，指定I/O缓冲区大小字节数，默认值为2GiB。
index.cache.size	整型值	范围为1~34359738368，指定索引缓存大小字节数，默认值为6GiB。

清空 Lance 表

通过TRUNCATE TABLE语法清空表中的数据。具体语法，请参见**TRUNCATE**。

示例：

```
TRUNCATE TABLE lance_tbl;
```

清理 Lance 表历史版本

通过VACUUM TABLE语法清理Lance表的历史版本及不再被引用的数据文件、事务文件、索引文件，回收OBS存储空间。Lance表在执行INSERT/UPDATE/DELETE/INSERT OVERWRITE/ALTER INDEX OPTIMIZE等操作后会产生新版本，旧版本及其引用的文件会持续保留，可通过VACUUM TABLE清理。

支持的清理选项：

- older_than_seconds：整型值，范围为1~31536000，仅清理产生时间超过该秒数的旧版本。省略时Lance内核使用默认值（ 14 天 ）。
- delete_rate_limit：整型值，范围为1~2147483647，删除操作的限速（ 次/秒 ），用于避免触发OBS限流。

VACUUM TABLE返回一行清理统计结果，包含6个字段：

表 2-6 字段说明

字段	类型	说明
bytes_removed	bigint	清理释放的字节数。
old_versions	integer	清理的旧版本数。
data_files_removed	integer	清理的数据文件数。
transaction_files_removed	integer	清理的事务文件数。
index_files_removed	integer	清理的索引文件数。
deletion_files_removed	integer	清理的删除标记文件数量。

示例：

清理2秒前产生的旧版本：

```
VACUUM TABLE lance_tbl WITH (older_than_seconds = 2);
```

使用默认参数（Lance内核默认清理14天前的旧版本）：

```
VACUUM TABLE lance_tbl;
```

Lance 表数据压缩

通过Compaction操作对Lance表进行数据重写（Rewrite Data），将多个小Fragment合并为较大的Fragment，并按配置重新组织Fragment内部的Row Group，从而减少Fragment数量、降低元数据开销并优化查询性能。Lance表在多次执行 INSERT、DELETE 等操作后会产生多个Fragment以及删除标记文件（Deletion File），通过Compaction可以合并小Fragment，并根据配置选择是否物化删除（Materialize Deletions）已标记删除的数据。

支持的Compaction选项如下表所示：

表 2-7 参数说明

字段	类型	取值范围	默认值	说明
target_rows_per_fragment	bigint	[1, int32)	1048576（约100万行）	Compaction会以该值作为目标 Fragment行数，对满足条件的Fragment进行重写，使生成的新 Fragment的行数尽可能接近该值（并非严格等于，也并非所有小于该值的Fragment都一定会被单独合并）。
max_rows_per_group	bigint	[1, int32)	1024	每个Fragment内部Row Group（Arrow RecordBatch）的最大行数。当Fragment的行数超过该值时，会被划分为多个Row Group。
max_bytes_per_file	bigint	[1, int64)	write_dataset默认值	单个.lance文件允许的最大字节大小。

字段	类型	取值范围	默认值	说明
materialize_deletions	boolean	true/false	true	是否在Compaction时物化删除（Materialize Deletions）已标记删除的数据。 • 设置为true时，会将删除的数据从新的Fragment中移除。 • 设置为false时，会保留删除标记文件（Deletion File）并继续引用原有Fragment。
materialize_deletions_threshold	real	[0.0, 1.0]	0.1（10%）	当materialize_deletions=true 时，仅删除比例超过该阈值的Fragment会执行物化删除。删除比例 = 已删除行数 / Fragment总行数
num_threads	bigint	[0, int32)	CPU核心数	compaction执行时使用的线程数（控制CPU并行度）。当设置为0或未指定时，将自动使用当前机器的CPU Core数。
batch_size	bigint	[0, int32)	scanner默认值	Scanner每次读取的数据Batch大小（影响内存占用和处理速度）。
defer_index_remap	boolean	true/false	false	是否延迟执行索引重映射（Index Remapping）。设置为 true 时，Compaction不会立即更新关联索引，而是记录Fragment的映射关系，由后续操作完成索引重映射，可减少 Compaction 的执行时间

字段	类型	取值范围	默认值	说明
max_source_fragments	bigint	[0, int32)	无限制	单个Compaction Task最多允许参与合并的源Fragment数量，用于限制一次重写的数据规模。

示例：

基本用法：

```
OPTIMIZE lance_table REWRITE DATA;
```

指定目标Fragment数：

```
OPTIMIZE lance_table REWRITE DATA WITH OPTIONS ('target_rows_per_fragment' = '10000');
```

多参数配置：

```
OPTIMIZE lance_table REWRITE DATA  WITH OPTIONS (    'target_rows_per_fragment' = '10000',    'materialize_deletions' = 'true',    'materialize_deletions_threshold' = '0.2',    'max_source_fragments' = '50');
```

删除 Lance 表

通过DROP TABLE语法删除Lance表，会同时删除表的元数据及数据（Managed表删除后可以从LakeFormation控制台恢复表的元数据及数据）。具体语法，请参见**DROP TABLE**。

示例：

```
DROP TABLE lance_tbl;
```

修改表中的列

- 添加列：

  向一个指定表中添加一列或者多列，需要指定列名和数据类型，添加的列会追加到表的当前列后面。

  ```
  ALTER TABLE tbl_name ADD COLUMN col1 typ1, col2 typ2, ....;
  ```

- 删除列：

  删除一个指定表的一列或者多列，不能删除一个表的所有列。

  ```
  ALTER TABLE tbl_name DROP COLUMNS (col1, col2,...);
  ```

- 重命名列：

  重命名一个指定的列名，不能同时重命名多列。

  ```
  ALTER TABLE tbl_name RENAME COLUMN old_name TO new_name;
  ```

- 更改列类型：

  更改一个或多个指定列的数据类型为另一个类型。

  ```
  ALTER TABLE tbl_name ALTER COLUMN a TYPE new_type, b TYPE new_type, ...;
  ```

  当前仅支持从低精度类型更新到高精度类型，不支持以下范围之外的类型转换：

  - 整数类型：从smallint更新到int或者bigint，从int更新到bigint。
  - 浮点类型：从float4更新到float8。

- numeric类型：从低精度（precision）更新到高精度（precision），不支持更新小数位数（scale）。
- 字符串类型：varchar或者char的长度变长。

修改表参数

- 添加或修改参数：

  仅支持max.rows.per.file、max.bytes.per.file、lance.encoding.compression三个参数。

  ```
  ALTER TABLE tbl_name SET TABLEPROPERTIES('max.rows.per.file'='1024', 'max.bytes.per.file'='2048', 'lance.encoding.compression'='zstd');
  ```

- 移除表参数：

  仅支持max.rows.per.file、max.bytes.per.file、lance.encoding.compression三个参数。

  ```
  ALTER TABLE tbl_name UNSET TABLEPROPERTIES('max.rows.per.file', 'max.bytes.per.file', 'lance.encoding.compression');
  ```

2.2.4 查询和写入 Lance 表

查询 Lance 表

可以查询Lance表中的数据。具体语法，请参考**SELECT**。

注意事项：

当前仅支持查询最新全量数据。

示例：

```
SELECT * FROM lance_tbl;
SELECT a, b FROM lance_tbl WHERE a = 1;
```

支持基于全文索引的单词、短语检索以及相关的bool类型，content1和content2为fts索引列：

示例：

```
SELECT * FROM lance_tbl WHERE content1@@ 'Index';
SELECT * FROM lance_tbl WHERE content1 @@ 'Search' AND content2 @@ 'index';
SELECT * FROM lance_tbl WHERE content1 @@ 'Arrow' OR content2 @@ 'fox';
SELECT * FROM lance_tbl WHERE content1 @@ 'fox' AND NOT content2 @@ 'lazy';
SELECT * FROM lance_tbl WHERE (content1 @@ 'fox' AND content1 @@ 'brown') OR content2 @@ 'python';
```

写入 Lance 表

可以向Lance表中添加一行或多行数据。具体语法，请参考**INSERT**。

示例：

```
INSERT INTO lance_tbl VALUES (1, 'a', 1.1234567, 100.11, '2025-03-03', '2025-03-03 16:00:00');
INSERT OVERWRITE INTO lance_tbl VALUES (1, 'b', 1.1234567, 100.11, '2025-03-03', '2025-03-03 16:00:00');
INSERT INTO lance_tbl (col1, col2, col3, col4, col5, col6) SELECT col1, col2, col3, col4, col5, col6 FROM lance_tbl2 WHERE col1 > 18;
```

2.2.5 更新 Lance 表

轻量级更新 Lance 表数据

可以对Lance表中的一行或多行数据执行更新操作。具体语法，请参考**UPDATE**。

注意事项：

不支持update from、set或where条件中会产生多表join的update。

示例：

```
UPDATE lance_tbl SET b = 1 WHERE a = 1;
```

2.2.6 Lance 表使用索引

创建索引

可以对Lance表中的一列向量列构建索引，向量列需要被声明为fixed-size-list，请参见**创建Lance表**。

向量索引类型可以是{ivfflat, ivfpq, ivfrq, ivfsq, hnsw, hnswpq, hnswsq}中的任意一种，建索引具体语法及约束，请参考**CREATE INDEX**。

支持分布式创建向量索引。产生的索引可以为一个全局索引，也可以是多个局部索引，其中局部子索引对应表的一个划分，对应被索引的数据互不重叠，可以通过设置fq_query_vdn控制表的分片数量。若建索引时参数merge_segment=1，则生成全局索引，否则生成多个子索引。使用多个子索引有利于分布式索引查询加速。

表 2-8 向量索引算法及其适用的场景

索引类型	原理	构建开销	压缩率	查询延迟/小top	召回率	内存占用	使用场景	调优参数
ivfflat	聚类分桶(partition)，存原始向量，查询时遍历分桶中心，仅扫描部分桶，索引扫描得出与查询向量精确距离	低	无	中	极高	高	中小规模表，精度要求高，内存充足	建：nlists, sample_rate, max_iters 查：minimum_nprobes, maximum_nprobes, refine_factor

索引类型	原理	构建开销	压缩率	查询延迟/小top	召回率	内存占用	使用场景	调优参数
ivfpq	聚类分桶，存乘积量化码本及向量量化结果， 索引扫描基于量化的近似距离，可通过refine过程计算精确距离排序	中	高	较低	中	低	亿级+，精度要求低，内存受限，大topK，混合标向量查询；需要平衡速度与精度，需要增大refine_factor以提升召回率，但较高的refine_factor(>2)将会影响性能	建：nlists, sample_rate, max_iters, num_sub_vectors 查：minimum_nprobes, maximum_nprobes, refine_factor
ivfrq	聚类分桶 存残差RabitQ (1-bit)，索引扫描得出近似距离，可refine过程精排	较高	较高	中	高	中低	亿级+，精度要求中等，内存受限，大topK，混合标向量查询；通过较低的refine_factor可得到高recall的结果	建：nlists, sample_rate, max_iters, num_bits（目前仅支持1bit） 查：minimum_nprobes, maximum_nprobes, refine_factor
ivfsq	聚类分桶，存标量量化 (8-bit)，索引扫描为近似距离，可通过refine精排	中	中	中	高	中	亿级+，精度要求高，内存较充足，大topK，混合标向量查询	建：nlists, sample_rate, max_iters, num_bits 查：minimum_nprobes, maximum_nprobes, refine_factor

索引类型	原理	构建开销	压缩率	查询延迟/小top	召回率	内存占用	使用场景	调优参数
hnsw	聚类生成多层有向联通图，存原始向量，查询时通过导航图访问部分分桶中心，仅扫描部分桶，扫描为精确距离	高	无	低	极高	高 (>ivfflat)	亿级+，低延迟检索，小topK，高精度，混合查询时标量过滤器选择性较低；	建：nlists, sample_rate, max_iters, m, ef 查：ef, refine_factor, minimum_nprobes, maximum_nprobes,
hnswpq	聚类生成多层有向联通图，存乘积量化码本及向量量化结果， 索引扫描基于量化的近似距离，可通过refine过程计算精确距离排序	极高	高	低	中	中	亿级+，低延迟检索，小topK，低精度，混合查询时标量过滤器选择性较低；需要增大refine_factor以提升召回率	建：nlists, sample_rate, max_iters, m, ef, num_sub_vectors 查：ef, refine_factor, minimum_nprobes, maximum_nprobes,
hnswsq	聚类生成多层有向联通图，存标量量化 (8-bit)，索引扫描为近似距离，可通过refine精排	高	中	低	高	较高	亿级+，低延迟检索，小topK，较高精度，混合查询时标量过滤器选择性较低；	建：nlists, sample_rate, max_iters, m, ef, num_bits 查：ef, refine_factor, minimum_nprobes, maximum_nprobes,

示例:

为Lance表lance_tbl建立一个名为lance_index_01的索引，指定算法为ivfpq，索引距离函数为L2距离，索引的数据包含2类，每个向量可以被量化分解为4个子向量，将生成多个子索引。

```
CREATE INDEX IF NOT EXISTS lance_index_01 ON lance_tbl USING ivfpq (vector_column vector_l2_ops)
WITH (nlists=2, num_subvectors=4, merge_segment=0);
```

表 2-9 标量索引算法及其适用的场景

索引类型	原理	使用场景	支持的数据类型	参数
btree	按值排序分页，内存中存每页 min/max 构建查找树，查询时先通过 BTree 裁剪无关页面，再仅扫描少量候选页中的原始数据，返回精确结果。	范围查询、等值查询、IN 查询、大数据集精确过滤	整数：tinyint, smallint, int, integer, bigint, long 浮点：real, float, double precision, double 布尔：boolean 字符串：varchar(n), char(n), string, text 时间：date, timestamp 小数：decimal(p,s), numeric(p,s)	无

示例:

为Lance表lance_tbl中column_name列建立一个名为lance_idx的索引，指定算法为btree。

```
CREATE INDEX IF NOT EXISTS lance_idx ON lance_tbl USING btree (column_name);
```

表 2-10 全文索引算法及其适用的场景

索引类型	原理	使用场景	支持的数据类型	参数
inverted	按照分词器对原始句子进行分词，查询时基于查询的单词或短语，根据倒排表找到对应的原始数据，返回精确结果。	单词匹配、短语匹配。	• 文本：text • 变长字符：varchar • 定长字符：bpchar • 文本数组：text[] • 变长字符数组：varchar[] • 定长字符数组：bpchar[]	base_tokenizer、with_position、max_token_length，关于参数的详细说明，请参见**表 2-12**。

说明

在Aura中Lance格式标量索引当前只支持btree类型，全文索引只支持inverted类型。

示例:

为Lance表lance_tbl中column_name列建立一个名为lance_fts的索引，指定算法为全文索引，指定分词器为ngram（支持中文），指定包含位置（用于短语查询），指定最大token长度是20（超过将被截断）。

```
CREATE INDEX IF NOT EXISTS lance_fts ON lance_tbl USING inverted (column_name) WITH
(base_tokenizer = 'ngram', with_position = true, max_token_length = 20);
```

表 2-11 全文索引可用的分词器

分词器值	说明	特性依赖
simple	按空白符和标点分词（默认）	始终可用
whitespace	仅按空白符分词	始终可用
raw	不分词，整个文本作为单个 token	始终可用
ngram	N-Gram 分词（重叠字符序列），需配合 min_ngram_length/ max_ngram_length	始终可用
icu	ICU 基于词典的 Unicode 分词	始终可用

分词器值	说明	特性依赖
lindera/<model>	Lindera 日语形态分析（如 lindera/ipadic）	需要手动下载词典
jieba 或 jieba/<model>	Jieba 中文分词（jieba 等同于 jieba/default）	需要手动下载词典

表 2-12 参数说明

参数名	类型	默认值	推荐值	说明
base_tokenizer	String	simple	• 英文用 "simple" • 中文用 "jieba/ngram" • 日语用 "lindera/ipadic" • 模糊前缀搜索用 "ngram"	基础分词器，决定文本如何被切分为 token。可选值： • simple（按空白和标点分词） • whitespace（仅按空白分词） • raw（不分词） • ngram（N-Gram 重叠字符序列） • icu（ICU Unicode 分词） • lindera/<model>（日语形态分析） • jieba（中文分词）
with_position	bool	false	• 需要短语查询时设为 true • 仅做关键词检索时保持false	是否在倒排索引中存储 token 位置信息。true 支持短语查询，但索引体积会显著增大（可达 10 倍+）。ngram 分词器不支持此参数
max_token_length	uint32	40	• 一般场景保持默认 40 • 中文/日文等无空格分词的语言建议设为 None • 需要 contains_tokens加速时必须设为None	token文本长度上限，长度>=该值的token会被过滤掉。设为None表示不限制。设为非None值会导致contains_tokens SQL查询无法走索引加速

展示已有索引

您可以打印出当前数据库全部Lance表或某一Lance表上的全部索引，具体语法，请参考**SHOW INDEX**。

示例：

```
SHOW INDEX;
SHOW INDEX lance_tbl;
```

删除已有索引

可以删除某一Lance表上的一个已建好的索引，语法参考**DROP INDEX**。

示例：

```
DROP INDEX lance_index_01 ON lance_tbl;
```

重建或更新已有索引

注意

更新Lance索引列的数据会导致部分索引片段或者整个索引失效，需要定期执行optimize来增量重建索引。

可以基于一个已有的索引重建索引，重建的索引将使用相同的索引名及索引参数，但索引的数据是该索引列的最新的全量数据。具体语法，请参考**ALTER INDEX**。

示例：

```
ALTER INDEX IF EXISTS lance_index_01 ON lance_tbl REBUILD;
```

可以基于一个已有的索引更新索引。当有新数据插入或删除时，已经建好的索引不会立即更新，从而不能根据索引查找到最新数据，会影响查询性能（但不会影响结果集）。

为使已有索引包含最新数据，可以使用索引优化功能避免全量数据重建索引。索引优化后，将可以根据索引找到最新的全量数据。

具体语法，请参考**ALTER INDEX**。

示例：

```
ALTER INDEX IF EXISTS lance_index_01 ON lance_tbl OPTIMIZE;
```

2.3 Lance 开发常见问题

2.3.1 如何处理 Lance 表并发冲突

Lance表采用MVCC（多版本并发控制）机制：每个版本由一个不可变的manifest描述，该manifest引用fragments（数据文件）、indices（索引文件）以及可选的transaction（事务）文件。读操作读取某个特定的manifest版本，因此支持快照读取；写操作通过提交新的manifest来生成新版本，从而实现ACID语义（至少在单表或单数据集的范围内）。Lance的“事务”（transaction）是一次写入、更新、删除或建索引等变更提交（commit）后，生成一个新的表版本。

Lance 事务提交流程

- 基于某个基线版本（base version）构造事务，例如追加新的fragments、生成新的索引引用等（事务本质是“相对基线manifest的变更”）。
- 写入事务相关的新文件，先把新fragments/ 删除文件 / 索引等写到存储中（这些文件本身是不可变的）。
- 原子提交manifest，最后一步尝试“原子地”写出一个新的manifest文件作为新版本的入口；如果并发写导致原子提交失败，会进入冲突检测/重试/可能的rebase。

事务冲突

当两个或多个用户都基于同一个base版本提交时，就会出现冲突窗口：各方均试图将“下一版本”指向各自的新manifest。Lance会在提交阶段检测到这一点，并尝试用事务文件/冲突规则做判断与（有条件的）自动合并或rebase；否则返回提交冲突。

- Lance事务操作类型

表 2-13 Lance 事务操作类型说明

事务名称	事务描述	涉及语句（如不涉及标注为N/A）
Append	将新的分片添加到表中，不会修改现有数据。	INSERT
Delete	使用删除向量将行标记为已删除。	N/A
Overwrite	创建或完全用新数据、新schema和新配置覆盖该表。	INSERT OVERWRITE、TRUNCATE
CreateIndex	添加、替换或删除二级索引（向量索引、标量索引、全文检索索引）。	CREATE INDEX/DROP INDEX/REBUILD INDEX/ OPTIMIZE INDEX
Rewrite	对数据进行重组，不改变数据的语义。包括压缩、去重、重排等操作。	N/A
Merge	向表中添加新列，从而修改表结构。必须更新所有分片以包含这些新列。	ADD COLUMN、ALTER COLUMN TYPE
Project	从表中删除列，从而修改表结构。这是一个仅涉及元数据的操作；数据文件不会被修改。	DROP COLUMN、ALTER COLUMN RENAME

事务名称	事务描述	涉及语句（如不涉及标注为N/A）
Restore	将表恢复到之前的版本。	N/A
ReserveFragments	预分配分片ID，以便在将来的重写操作中使用。	N/A
Clone	创建表的浅拷贝或深拷贝。浅拷贝是仅包含元数据的拷贝，这些拷贝通过base_paths引用原始数据文件。	N/A
Update	修改行数据值，而不会添加或删除行。	UPDATE
UpdateConfig	修改表配置、表元数据、schema元数据或字段元数据，而不改变数据本身。	ALTER SET/UNSET
DataReplacement	将特定列区域的数据替换为新的数据文件。	N/A
UpdateMemWalState	更新MemWal索引（基于预写日志的索引）的状态。	N/A
UpdateBases	向表中添加新的基础路径，从而能够引用其他位置的数据文件。	N/A

- 事务冲突类型

表 2-14 事务冲突类型说明

冲突类型	描述	当前机制
Rebasable Conflict	事务可以进行修改以适应并发更改，同时保持其语义意图。事务会进行转换以适应并发修改，然后在提交层自动重试提交。	提交层重试提交，如果重试次数消耗完毕则报错。
Retryable Conflict	事务无法进行合并，会返回一个可重试冲突错误，应用层应该重新更新数据后进行重试提交。	直接报错。

冲突类型	描述	当前机制
Incompatible Conflict	事务之间存在根本性冲突，重试会违反操作的假设，或产生与预期语义不同的结果。提交操作失败，并返回不可重试的错误。	直接报错。

- Lance事务冲突矩阵：

当前操作 \ 历史操作	Delete	Update	CreateIndex	Rewrite	Overwrite	Append	DataReplament	Merge	Restore	ReserveFragments	Project	UpdateConfig	UpdateMemWalState	Clone	UpdateBases
Delete	?	?	√	?	×	√	?	×	×	√	√	√	×	√	√
Update	?	?	√	?	×	√	?	×	×	√	√	√	?	√	√
CreateIndex	√	√	?	?	×	√	?	√	×	√	√	√	×	√	√
Rewrite	?	?	?	?	×	√	?	×	×	√	√	√	√	√	√
Overwrite	√	√	√	√	?	√	√	√	√	√	√	?	×	√	√
Append	√	√	√	√	×	√	√	√	×	√	√	√	×	√	√
DataReplament	√	√	?	?	×	√	?	√	×	√	√	√	×	√	√
Merge	×	×	√	×	×	×	×	×	×	√	×	√	×	√	√
Restore	√	√	√	√	√	√	√	√	√	√	√	√	×	√	√
ReserveFragments	√	√	√	√	×	√	√	√	×	√	√	√	√	√	√
Project	√	√	√	√	×	√	√	×	×	√	×	√	×	√	√
UpdateConfig	√	√	√	√	?	√	√	√	√	√	√	?	√	√	√
UpdateMemWalState	×	?	√	√	×	×	×	×	×	√	×	√	?	×	√
Clone	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
UpdateBases	√	√	√	√	√	√	√	√	√	√	√	√	√	√	?

注：标“√”代表两类事务为可合并的冲突，“×”表示两类事务为无法解决的冲突，“？”表示两类事务为需要重试的冲突。

示例：

两个Append事务并发执行时（即同时向该表INSERT数据），互为可合并的冲突，Lance会尝试去合并两个Append的数据以适应并发修改，然后自动重试提交，如果在重试次数消耗完成前合并成功则两个事务并发执行成功且未产生冲突，如果重试次数消耗完仍无法合并则报错。

3 Spark UDF 开发指南

3.1 在 Spark SQL 作业中使用 UDF

操作场景

AI DataLake支持用户使用Hive UDF（User Defined Function，用户定义函数）进行数据查询等操作，UDF只对单行数据产生作用，适用于一进一出的场景。

约束限制

自定义函数中引用static类或接口时，必须要加上“try catch”异常捕获，否则可能会造成包冲突，导致函数功能异常。

环境准备

在进行UDF开发前，请准备以下开发环境。

表 3-1 UDF 开发环境

准备项	说明
操作系统	Windows系统，支持Windows7以上版本。
安装JDK	JDK使用21及以上版本。

准备项	说明
安装和配置IntelliJ IDEA	IntelliJ IDEA为进行应用开发的工具，版本要求使用2019.1或其他兼容版本。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。

开发流程

UDF函数开发流程参考如下：

图 3-1 UDF 开发流程

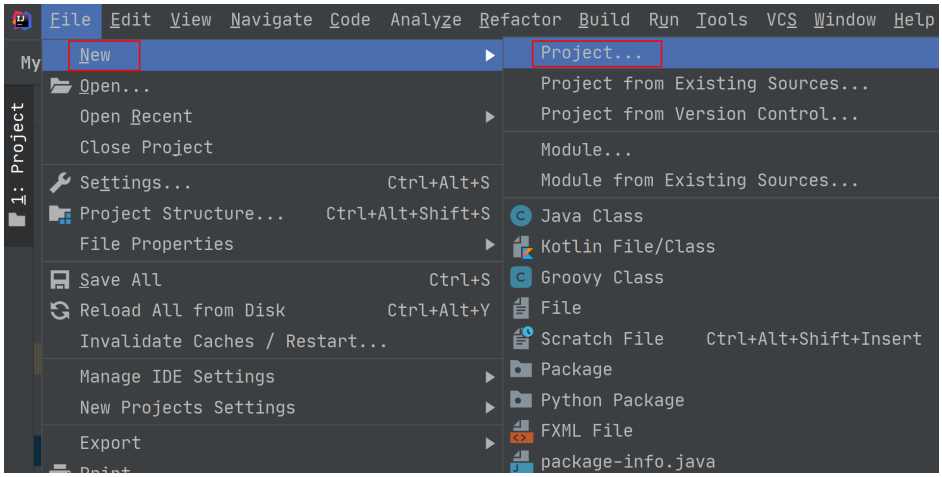
表 3-2 开发流程说明

序号	阶段	操作界面	说明
1	新建Maven工程，配置pom文件	IntelliJ IDEA	参考**操作步骤**说明，编写UDF函数代码。
2	编写UDF函数代码		
3	调试，编译代码并导出Jar包		
4	上传Jar包到OBS	OBS控制台	将生成的UDF函数Jar包文件上传到OBS目录下。
5	创建UDF函数	DataArts Studio控制台	在DataArts Studio控制台的“脚本开发”界面，创建UDF函数。
6	验证和使用UDF函数	DataArts Studio控制台	在DataArts Studio控制台的“脚本开发”界面，Spark SQL作业中使用创建的UDF函数。

操作步骤

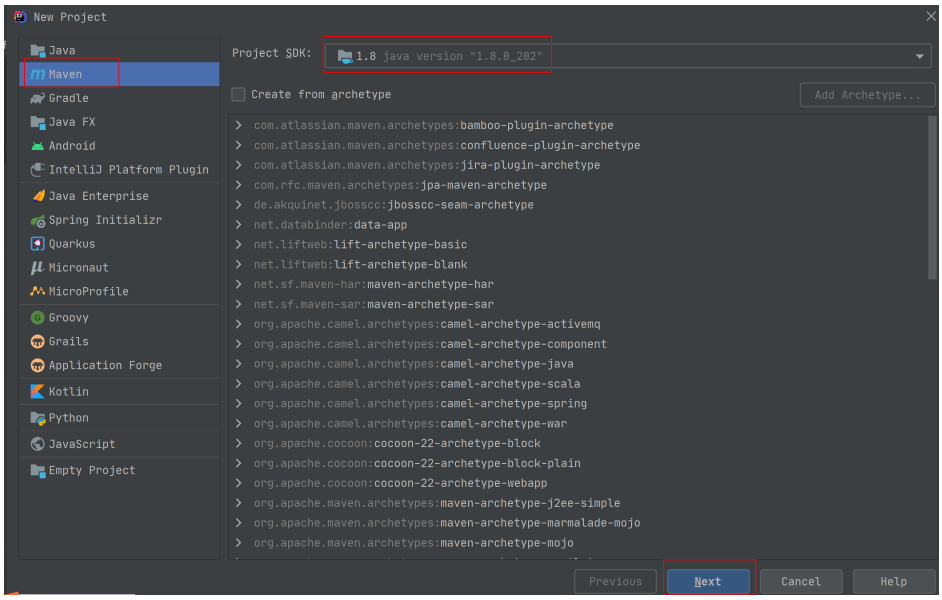
1. 新建Maven工程，配置pom文件。以下通过IntelliJ IDEA 2020.2工具操作演示。
   a. 打开IntelliJ IDEA，选择“File > New > Project”。

图 3-2 新建 Project



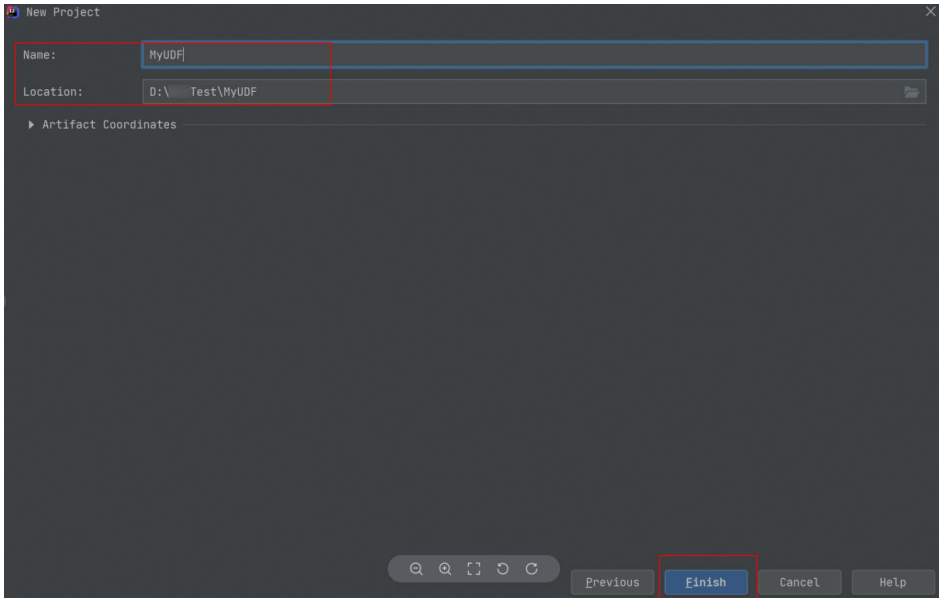
b. 选择Maven，Project SDK选择1.8，单击“Next”。

图 3-3 选择 Maven



c. 定义样例工程名和配置样例工程存储路径，单击“Finish”完成工程创建。

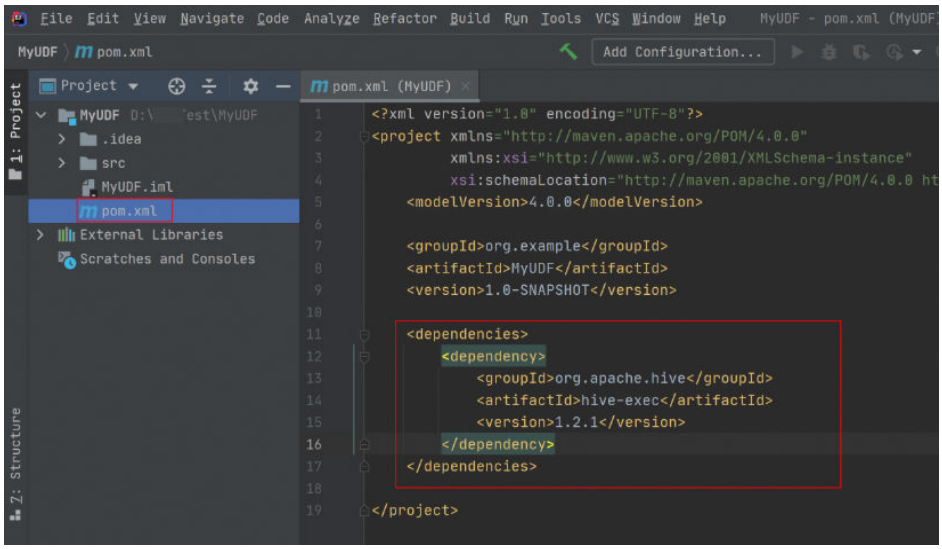
图 3-4 创建工程



d. 在pom.xml文件中添加如下配置。

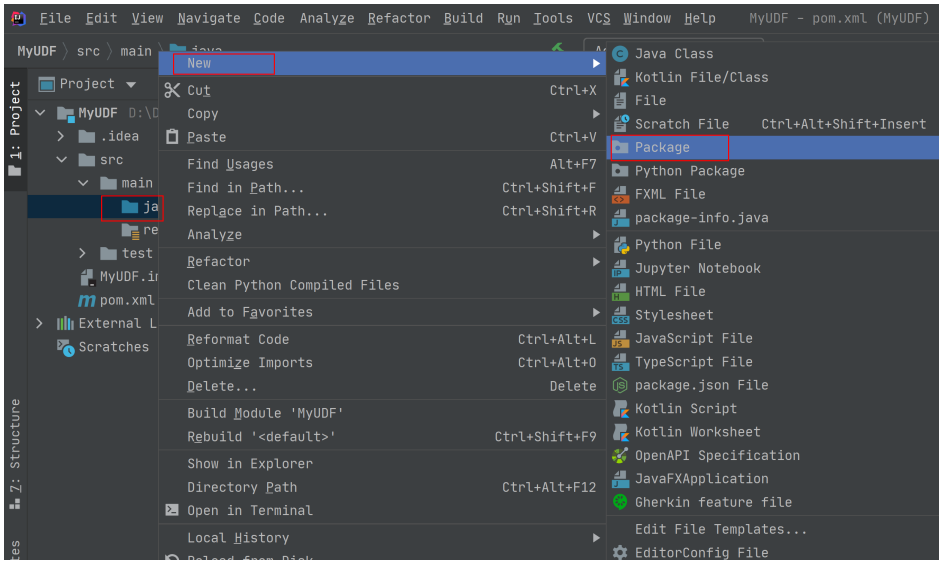
```
<dependencies>
        <dependency>
            <groupId>org.apache.hive</groupId>
            <artifactId>hive-exec</artifactId>
            <version>1.2.1</version>
        </dependency>
</dependencies>
```

图 3-5 pom 文件添加配置



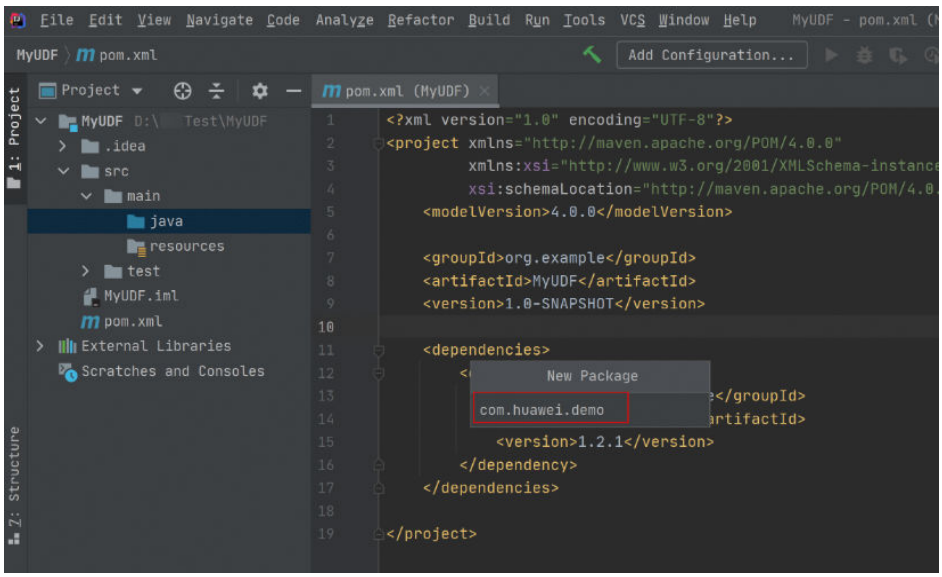
e. 在工程路径的“src > main > java”文件夹上鼠标右键，选择“New > Package”，新建Package和类文件。

图 3-6 新建 Package 和类文件



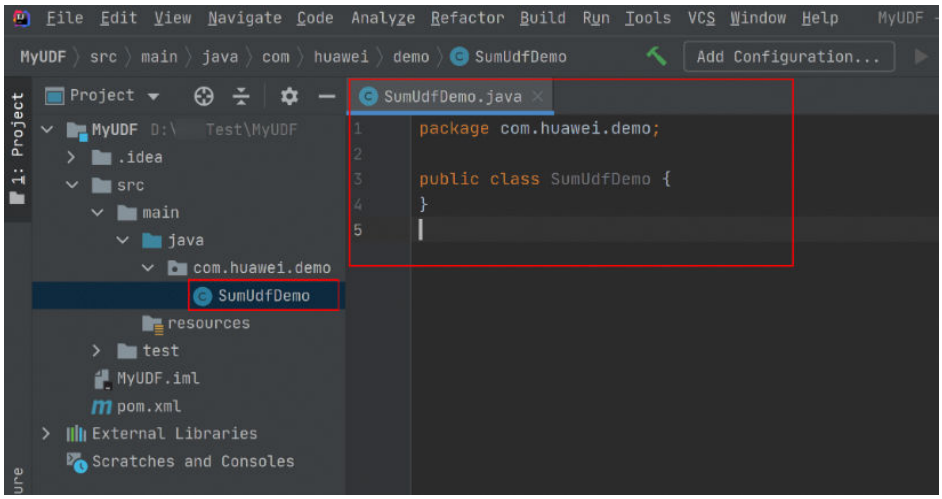
Package根据需要定义，本示例定义为：“com.huawei.demo”，完成后回车。

图 3-7 自定义 Package



在包路径下新建Java Class文件，本示例定义为：SumUdfDemo。

图 3-8 新建 Java Class 文件



2. 编写UDF函数代码。UDF函数实现，主要注意以下几点：

   a. 自定义UDF需要继承org.apache.hadoop.hive.ql.exec.UDF。

   b. 需要实现evaluate函数，evaluate函数支持重载。

   详细UDF函数实现，可以参考如下样例代码：

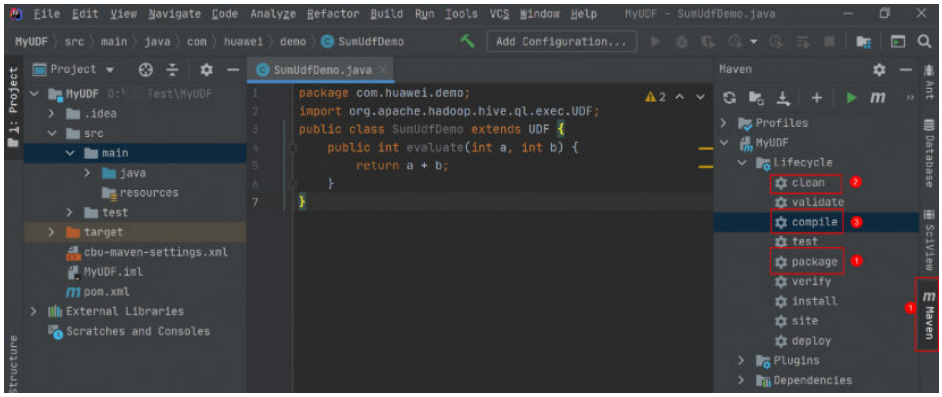
```
package com.huawei.demo;
import org.apache.hadoop.hive.ql.exec.UDF;
 public class SumUdfDemo extends UDF {
   public int evaluate(int a, int b) {
    return a + b;
 }
}
```

3. 编写调试完成代码后，通过IntelliJ IDEA工具编译代码并导出Jar包。

   a. 单击工具右侧的“Maven”，参考下图分别单击“clean”、“compile”对代码进行编译。

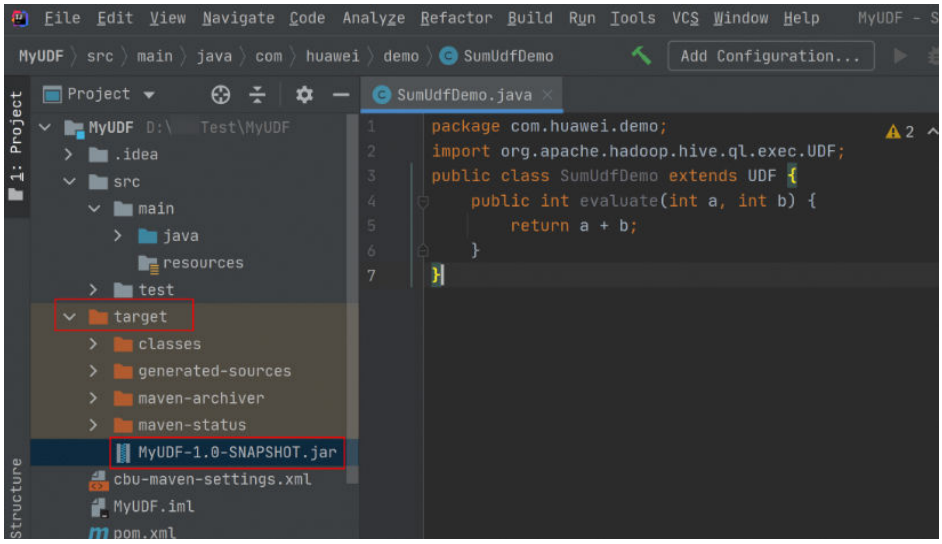
   编译成功后，单击“package”对代码进行打包。

图 3-9 编译打包



打包成功后，生成的Jar包会放到target目录下，以备后用。本示例将会生成到：“D:\AIDataLakeTest\MyUDF\target”下名为“MyUDF-1.0-SNAPSHOT.jar”。

图 3-10 生成 Jar 包



4. 登录OBS控制台，将生成的Jar包文件上传到OBS路径下。

   说明

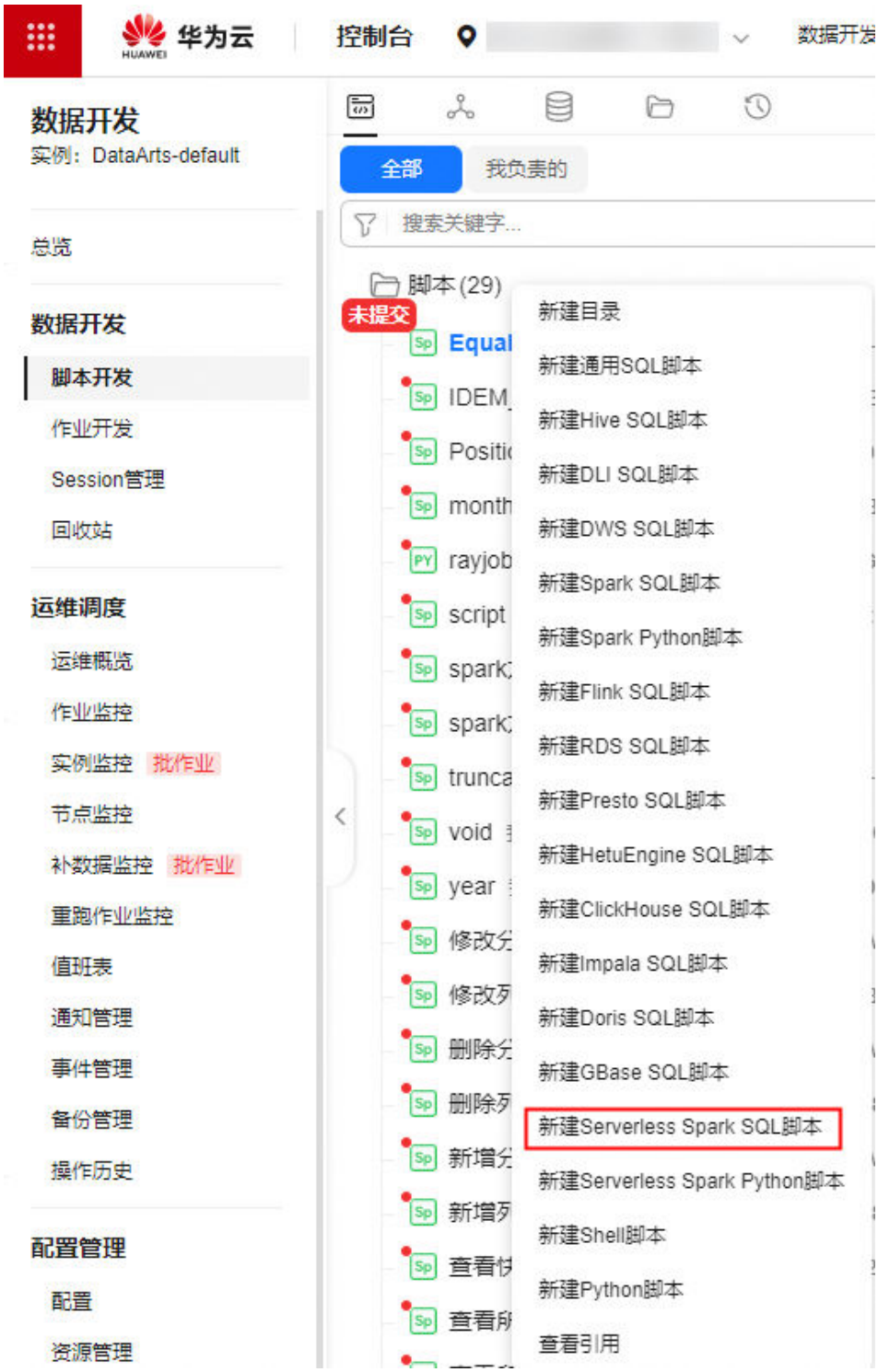
   Jar包文件上传的OBS桶所在的区域需与AI DataLake的工作空间区域相同，不可跨区域执行操作。

5. 创建UDF函数。

   a. 在**AI DataLake管理控制台**左下角，单击“DataArts Studio”。

   b. 在“脚本开发”页面，新建Serverless Spark SQL脚本后，选择端点，Catalog和数据库。

图 3-11 新建 Serverless Spark SQL 脚本



c. 在SQL编辑区域，输入创建UDF函数的命令。

```
CREATE FUNCTION TestSumUDF AS 'com.huawei.demo.SumUdfDemo' using jar 'obs://aidatalake-test-obs01/MyUDF-1.0-SNAPSHOT.jar';
```

d. 单击“运行”。

6. 使用UDF函数。

在查询语句中使用5中创建的UDF函数:

```
SELECT TestSumUDF(1,2);
```

7. （可选）删除UDF函数。

   如果不再使用UDF函数，可执行以下语句删除该函数:

   ```
   DROP FUNCTION TestSumUDF;
   ```

3.2 在 Spark SQL 作业中使用 UDAF

操作场景

AI DataLake支持用户使用Hive UDAF（User Defined Aggregation Function，用户定义聚合函数）可对多行数据产生作用，通常与groupBy联合使用；等同于SQL中常用的SUM()，AVG()，也是聚合函数。

约束限制

自定义函数中引用static类或接口时，必须要加上“try catch”异常捕获，否则可能会造成包冲突，导致函数功能异常。

环境准备

在进行UDAF开发前，请准备以下开发环境。

表 3-3 UDAF 开发环境

准备项	说明
操作系统	Windows系统，支持Windows7以上版本。
安装JDK	JDK使用21及以上版本（访问**Java官网**）。
安装和配置IntelliJ IDEA	**IntelliJ IDEA**为进行应用开发的工具，版本要求使用2019.1或其2019.1往后的版本。
安装Maven	开发环境的基本配置（**下载**并**安装 Maven**）。用于项目管理，贯穿软件开发生命周期。

开发流程

UDAF函数开发流程参考如下：

图 3-12 UDAF 开发流程

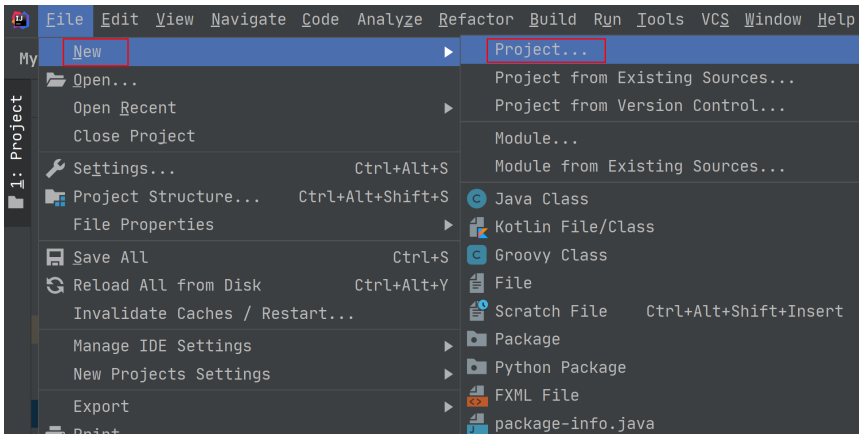
表 3-4 开发流程说明

序号	阶段	操作界面	说明
1	新建Maven工程，配置pom文件	IntelliJ IDEA	参考**操作步骤**说明，编写UDAF函数代码。
2	编写UDAF函数代码		
3	调试，编译代码并导出Jar包		
4	上传Jar包到OBS	OBS控制台	将生成的UDAF函数Jar包文件上传到OBS目录下。
5	创建UDAF函数	DataArts Studio控制台	在DataArts Studio控制台的“脚本开发”界面，创建UDAF函数。
6	验证和使用UDAF函数	DataArts Studio控制台	在DataArts Studio控制台的“脚本开发”界面，Spark SQL作业中使用创建的UDAF函数。

操作步骤

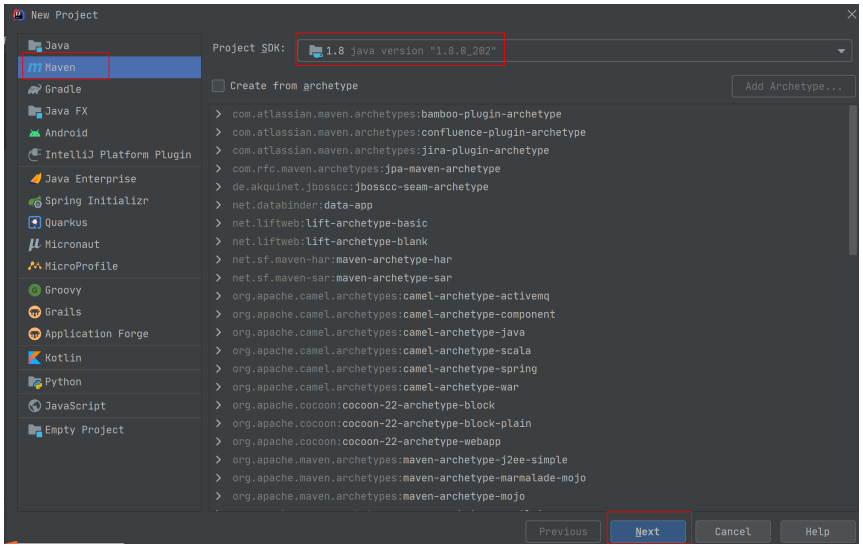
1. 新建Maven工程，配置pom文件。以下通过IntelliJ IDEA 2020.2工具操作演示。
   a. 打开IntelliJ IDEA，选择“File > New > Project”。

      图 3-13 新建 Project

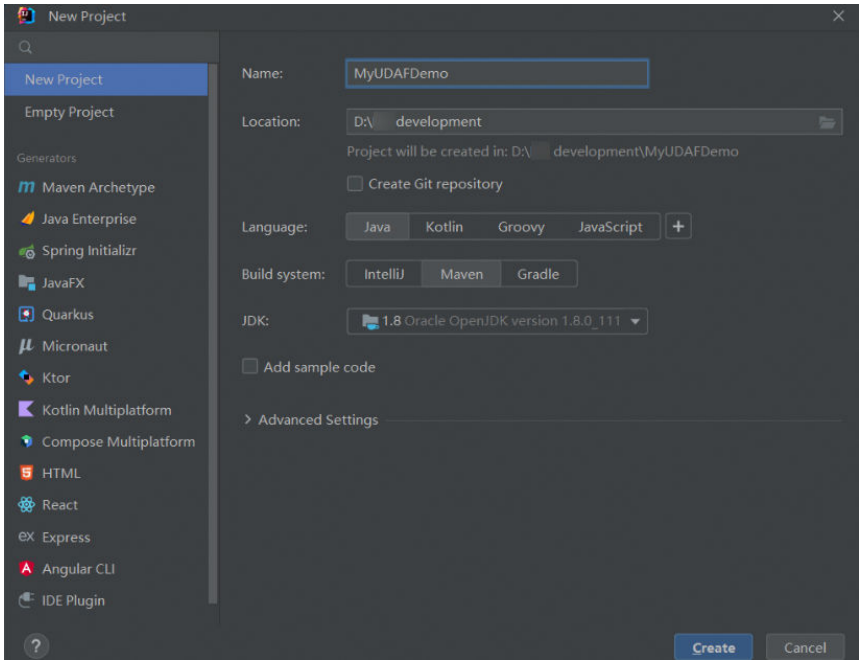
   b. 选择Maven，Project SDK选择1.8，单击“Next”。

图 3-14 配置 Project SDK



c. 定义样例工程名和配置样例工程存储路径，单击“Create"，下一步单击弹窗中的“Finish”完成工程创建。

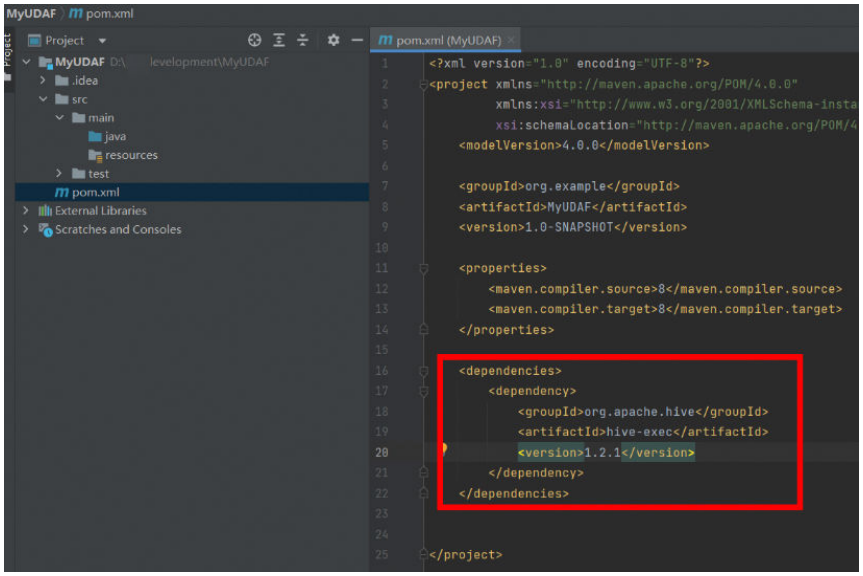
图 3-15 完成 Project 创建



d. 在pom.xml文件中添加如下配置。

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

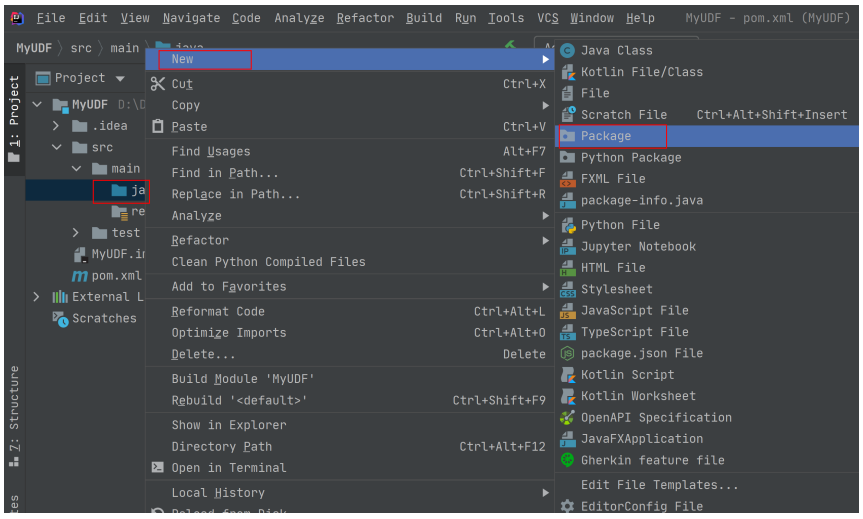
图 3-16 pom 文件中添加配置



e. 在工程路径的“src > main > java”文件夹上鼠标右键，选择“New > Package”，新建Package和类文件。

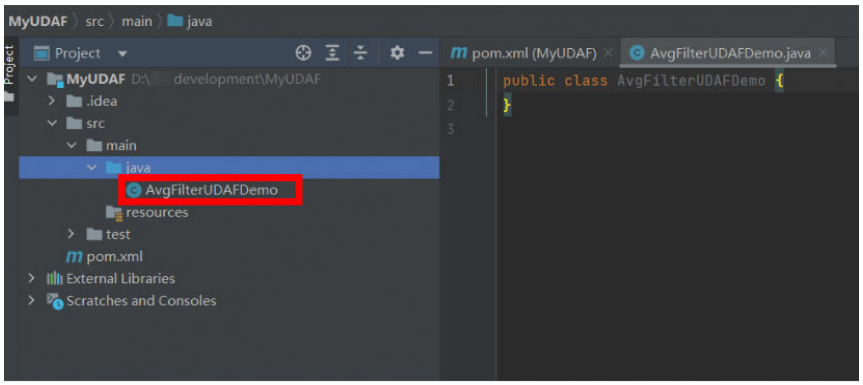
Package根据需要定义，本示例定义为：“com.aidatalake.demo”

图 3-17 新建 Package



在包路径下新建Java Class文件，本示例定义为：AvgFilterUDAFDemo。

图 3-18 创建类



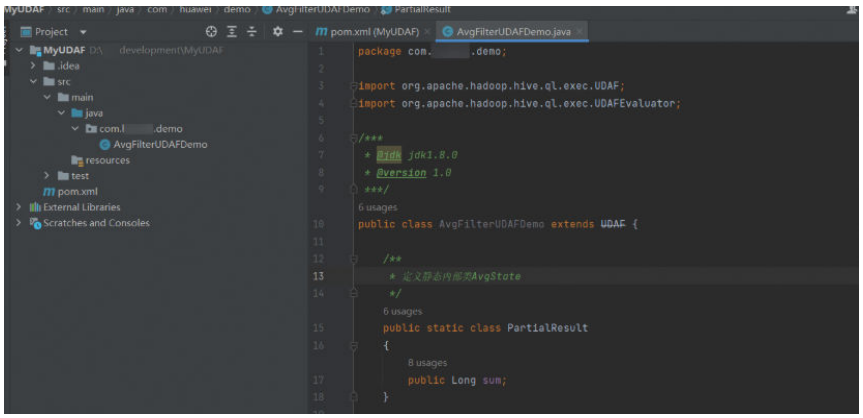
2. 编写UDAF函数代码。UDAF函数实现，主要注意以下几点：
   - 自定义UDAF需要继承org.apache.hadoop.hive.ql.exec.UDAF和org.apache.hadoop.hive.ql.exec.UDAFEvaluator类。函数类需要继承UDAF类，计算类Evaluator实现UDAFEvaluator接口。
   - Evaluator需要实现UDAFEvaluator的**init**、**iterate**、**terminatePartial**、**merge**、**terminate**这几个函数。
     - **init**函数实现接口UDAFEvaluator的init函数。
     - **iterate**接收传入的参数，并进行内部的迭代。
     - **terminatePartial**无参数，其为iterate函数遍历结束后，返回遍历得到的数据，terminatePartial类似于 hadoop的Combiner。
     - **merge**接收terminatePartial的返回结果。
     - **terminate**返回最终的聚集函数结果。

   详细UDAF函数实现，可以参考如下样例代码：

```
package com.aidatalake.demo;

import org.apache.hadoop.hive.ql.exec.UDAF;
import org.apache.hadoop.hive.ql.exec.UDAFEvaluator;

/***
 * @jdk jdk1.8.0
 * @version 1.0
 ***/
public class AvgFilterUDAFDemo extends UDAF {

    /**
     * 定义静态内部类AvgFilter
     */
    public static class PartialResult
    {
        public Long sum;
    }

    public static class VarianceEvaluator implements UDAFEvaluator {

        //初始化PartialResult对象
        private AvgFilterUDAFDemo.PartialResult partial;

        //创建VarianceEvaluator无参构造函数
        public VarianceEvaluator(){

            this.partial = new AvgFilterUDAFDemo.PartialResult();
```

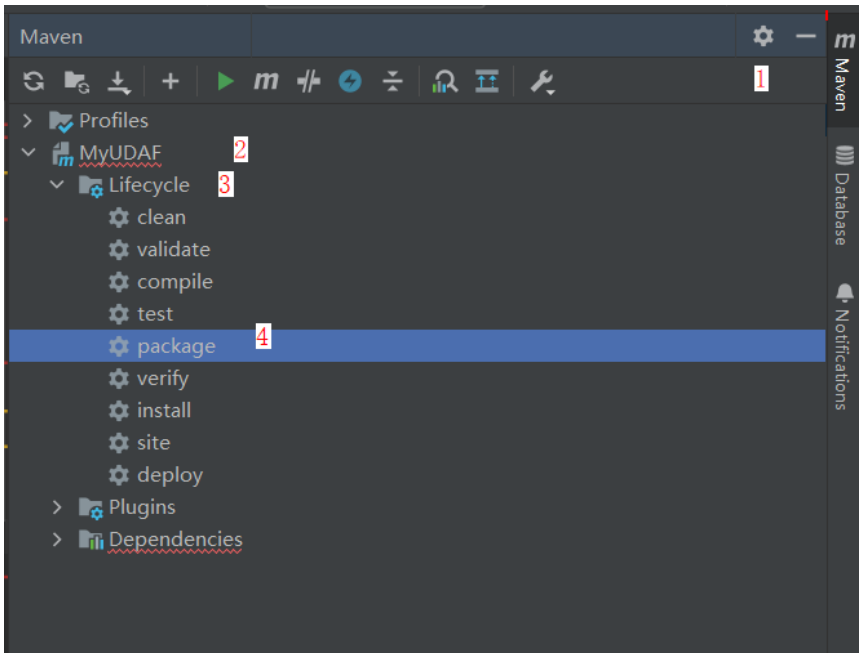
```
        init();
    }

    /**
     * init函数类似于构造函数，用于UDAF的初始化
     */
    @Override
    public void init() {

        //设置sum初始值
        this.partial.sum = 0L;
    }

    /**
     * iterate接收传入的参数，并进行内部的轮转。
     * @param x
     * @return
     */
    public void iterate(Long x) {
        if (x == null) {
            return;
        }
        AvgFilterUDAFDemo.PartialResult tmp9_6 = this.partial;
        tmp9_6.sum = tmp9_6.sum | x;
    }

    /**
     * terminatePartial无参数，其为iterate函数遍历结束后，返回轮转数据，
     * terminatePartial类似于hadoop的Combiner
     * @return
     */
    public AvgFilterUDAFDemo.PartialResult terminatePartial()
    {
        return this.partial;
    }

    /**
     * merge接收terminatePartial的返回结果，进行数据merge操作
     * @param
     * @return
     */
    public void merge(AvgFilterUDAFDemo.PartialResult pr)
    {
        if (pr == null) {
            return;
        }
        AvgFilterUDAFDemo.PartialResult tmp9_6 = this.partial;
        tmp9_6.sum = tmp9_6.sum | pr.sum;
    }

    /**
     * terminate返回最终的聚集函数结果
     * @return
     */
    public Long terminate()
    {
        if (this.partial.sum == null) {
            return 0L;
        }
        return this.partial.sum;
    }
  }
}
```

图 3-19 编写 UDAF 函数代码



3. 编写调试完成代码后，通过IntelliJ IDEA工具编译代码并导出Jar包。
   a. 单击工具右侧的“Maven”，参考下图分别单击“clean”、“compile”对代码进行编译。
      编译成功后，单击“package”对代码进行打包。

图 3-20 导出 jar 包



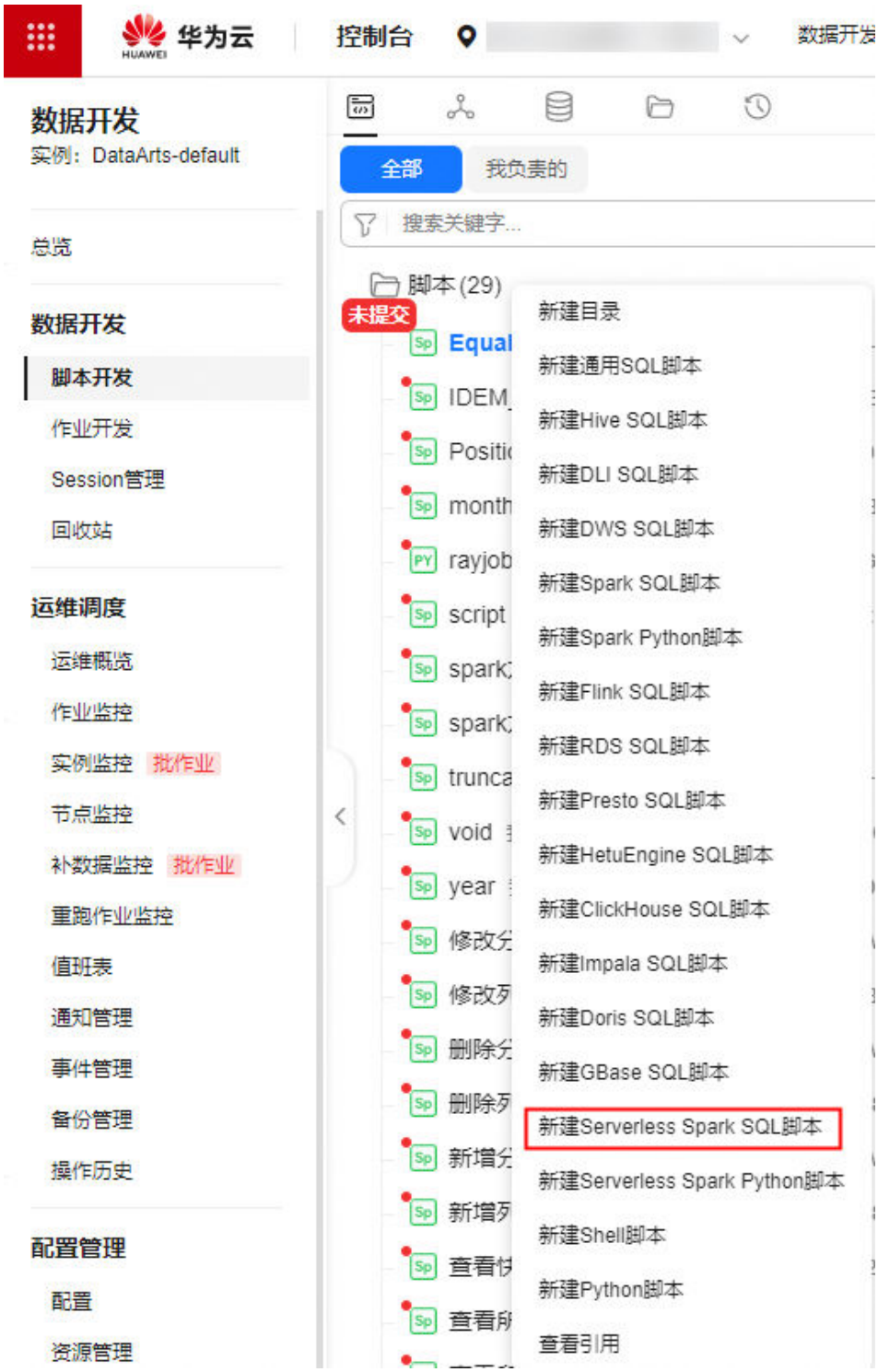
   b. 打包成功后，生成的Jar包会放到target目录下，以备后用。本示例将会生成到：“D:\AIDataLakeTest\MyUDAF\target”下名为“MyUDAF-1.0-SNAPSHOT.jar”。
4. 登录OBS控制台，将生成的Jar包文件上传到OBS路径下。

   说明

   Jar包文件上传的OBS桶所在的区域需与AI DataLake的工作空间区域相同，不可跨区域执行操作。
5. 创建UDAF函数。
   a. 在**AI DataLake管理控制台**左下角，单击“DataArts Studio”。

b. 在“脚本开发”页面，新建Serverless Spark SQL脚本后，选择端点，Catalog和数据库。

图 3-21 新建 Serverless Spark SQL 脚本



c. 在SQL编辑区域，输入创建UDAF函数的命令。

```
CREATE FUNCTION AvgFilterUDAFDemo AS 'com.aidatalake.demo.AvgFilterUDAFDemo' using jar 'obs://aidatalake-test-obs01/MyUDAF-1.0-SNAPSHOT.jar';
```

或

```
CREATE OR REPLACE FUNCTION AvgFilterUDAFDemo AS
'com.aidatalake.demo.AvgFilterUDAFDemo' using jar 'obs://aidatalake-test-obs01/MyUDAF-1.0-SNAPSHOT.jar';
```

说明

如果该用户开启了自定义函数热加载功能，注册语句会发生变化。

d. 单击“运行”。

6. 使用UDAF函数。

在查询语句中使用创建的UDAF函数:

```
select AvgFilterUDAFDemo(real_stock_rate) AS show_rate FROM dw_ad_estimate_real_stock_rate limit 1000;
```

7. （可选）删除UDAF函数。

如果不再使用UDAF函数，可执行以下语句删除该函数:

```
Drop FUNCTION AvgFilterUDAFDemo;
```

3.3 在 Spark SQL 作业中使用 UDTF

操作场景

AI DataLake支持用户使用Hive UDTF（User-Defined Table-Generating Functions）自定义表值函数，UDTF用于解决一进多出业务场景，即其输入与输出是一对多的关系，读入一行数据，输出多个值。

约束限制

自定义函数中引用static类或接口时，必须要加上“try catch”异常捕获，否则可能会造成包冲突，导致函数功能异常。

环境准备

在进行UDTF开发前，请准备以下开发环境。

表 3-5 UDTF 开发环境

准备项	说明
操作系统	Windows系统，支持Windows7以上版本。
安装JDK	JDK使用21及以上版本。
安装和配置IntelliJ IDEA	IntelliJ IDEA为进行应用开发的工具，版本要求使用2019.1或其他兼容版本。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。

开发流程

UDTF函数开发流程参考如下：

图 3-22 UDTF 开发流程

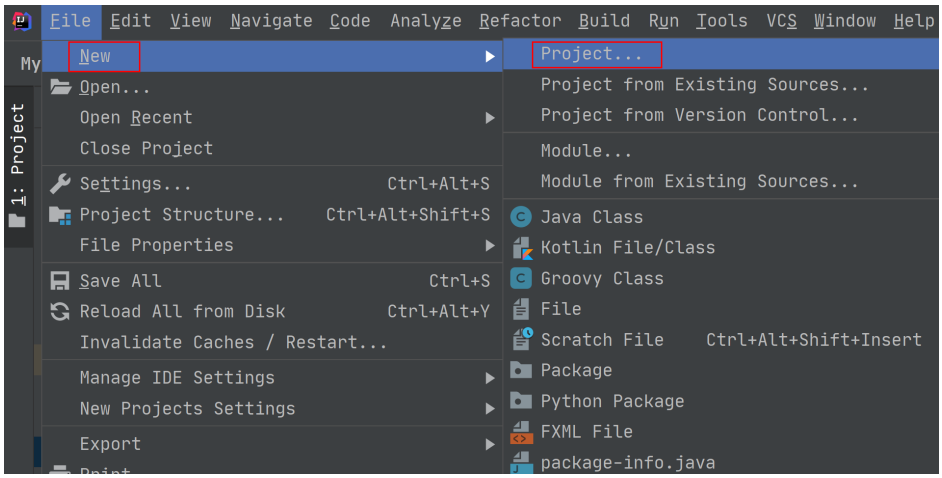
表 3-6 开发流程说明

序号	阶段	操作界面	说明
1	新建Maven工程，配置pom文件	IntelliJ IDEA	参考**操作步骤**说明，编写UDTF函数代码。
2	编写UDTF函数代码		
3	调试，编译代码并导出Jar包		
4	上传Jar包到OBS	OBS控制台	将生成的UDTF函数Jar包文件上传到OBS目录下。
5	创建UDTF函数	DataArts Studio控制台	在DataArts Studio控制台的“脚本开发”界面，创建UDTF函数。
6	验证和使用UDTF函数	DataArts Studio控制台	在DataArts Studio控制台的“脚本开发”界面，Spark SQL作业中使用创建的UDTF函数。

操作步骤

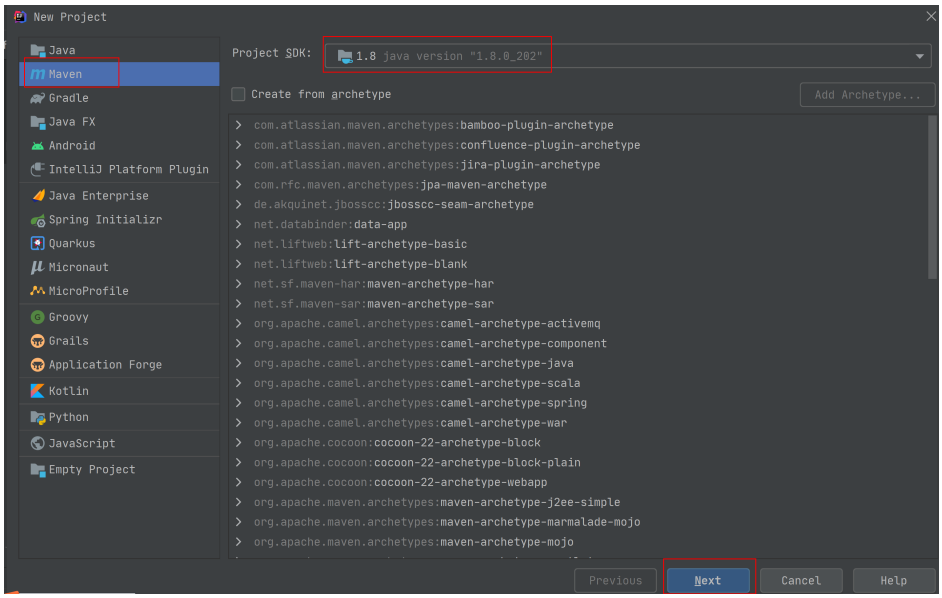
1. 新建Maven工程，配置pom文件。以下通过IntelliJ IDEA 2020.2工具操作演示。
   a. 打开IntelliJ IDEA，选择“File > New > Project”。

图 3-23 新建 Project



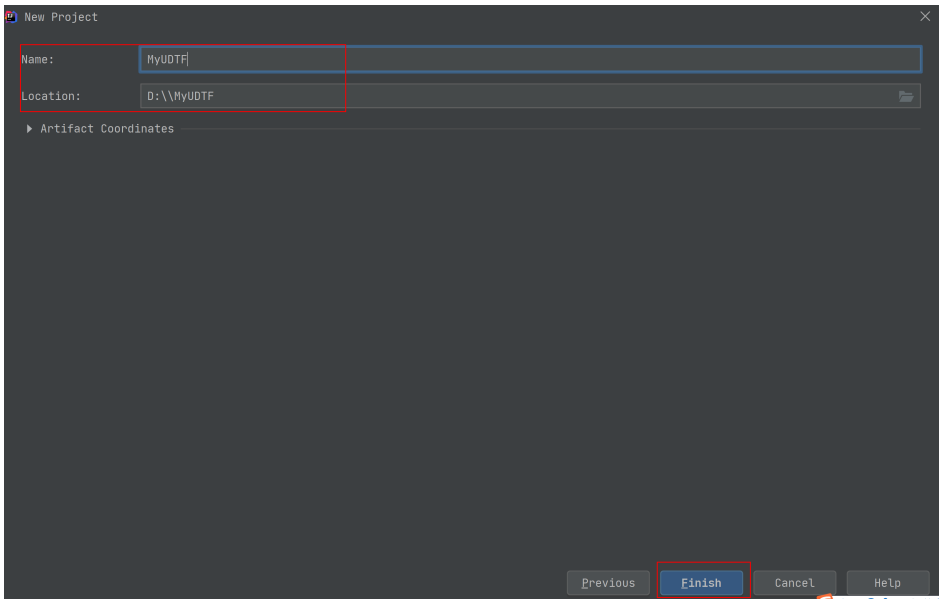
b. 选择Maven，Project SDK选择1.8，单击“Next”。

图 3-24 选择 Maven



c. 定义样例工程名和配置样例工程存储路径，单击“Finish”完成工程创建。

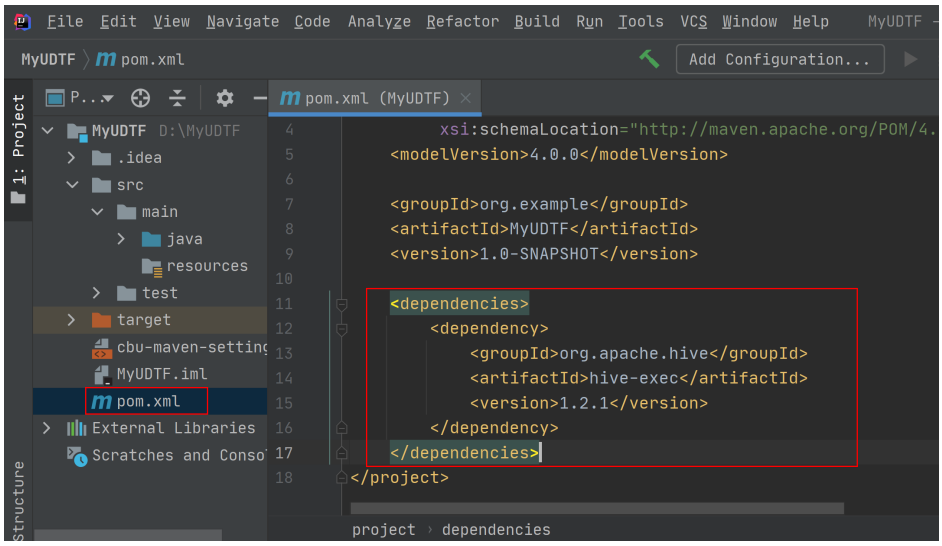
图 3-25 创建工程



d. 在pom.xml文件中添加如下配置。

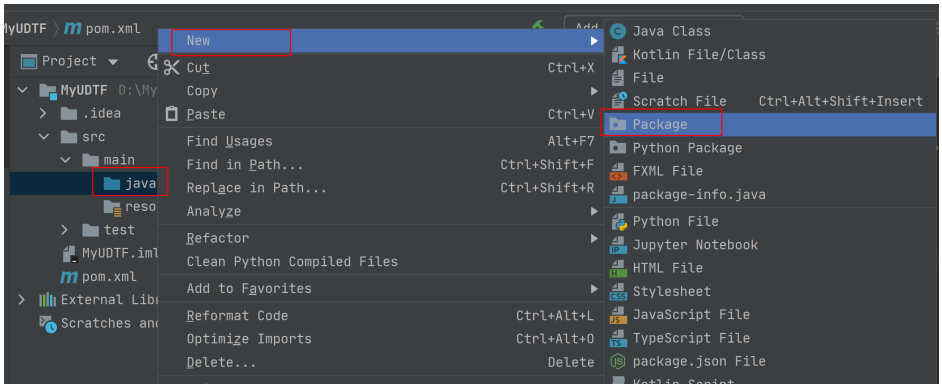
```
<dependencies>
        <dependency>
            <groupId>org.apache.hive</groupId>
            <artifactId>hive-exec</artifactId>
            <version>1.2.1</version>
        </dependency>
</dependencies>
```

图 3-26 pom 文件中添加配置



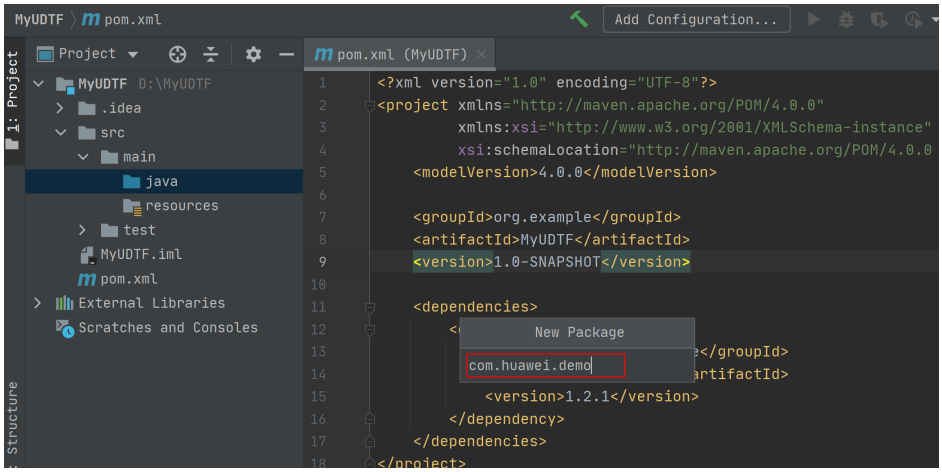
e. 在工程路径的“src > main > java”文件夹上鼠标右键，选择“New > Package”，新建Package和类文件。

图 3-27 新建 Package 和类文件



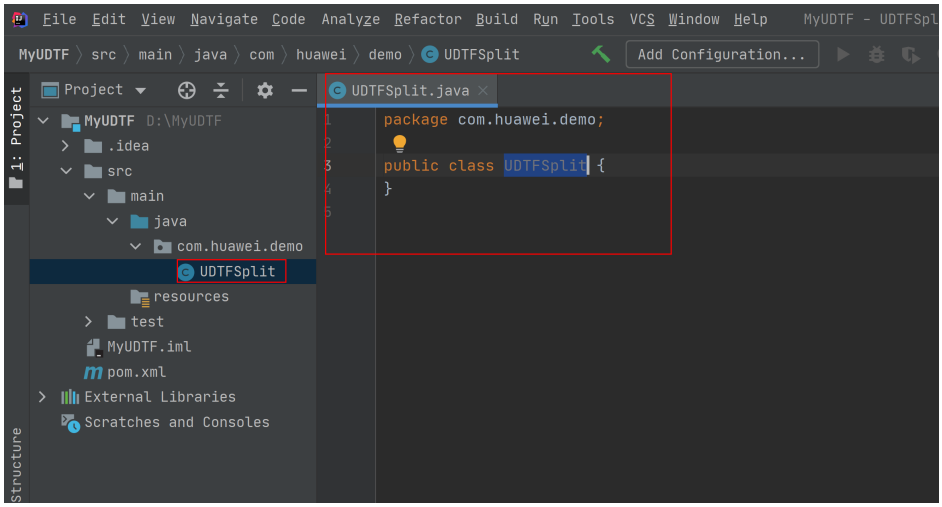
Package根据需要定义，本示例定义为：“com.huawei.demo”，完成后回车。

图 3-28 自定义 Package



在包路径下新建Java Class文件，本示例定义为：UDTFSplit。

图 3-29 新建 Java Class 文件



2. 编写UDTF函数代码。完整样例代码请参考**样例代码**。

   UDTF的类需要继承“org.apache.hadoop.hive.ql.udf.generic.GenericUDTF”，实现initialize，process，close三个方法。

   a. UDTF首先会调用initialize方法，此方法返回UDTF的返回行的信息，如，返回个数，类型等。

   b. 初始化完成后，会调用process方法，真正处理在process函数中，在process中，每一次forward()调用产生一行。

      如果产生多列可以将多个列的值放在一个数组中，然后将该数组传入到forward()函数。

```
public void process(Object[] args) throws HiveException {
        // TODO Auto-generated method stub
        if(args.length == 0){
            return;
        }
        String input = args[0].toString();
        if(StringUtils.isEmpty(input)){
            return;
        }
        String[] test = input.split(";");
        for (int i = 0; i < test.length; i++) {
            try {
                String[] result = test[i].split(":");
                forward(result);
            } catch (Exception e) {
                continue;
            }
        }

    }
```

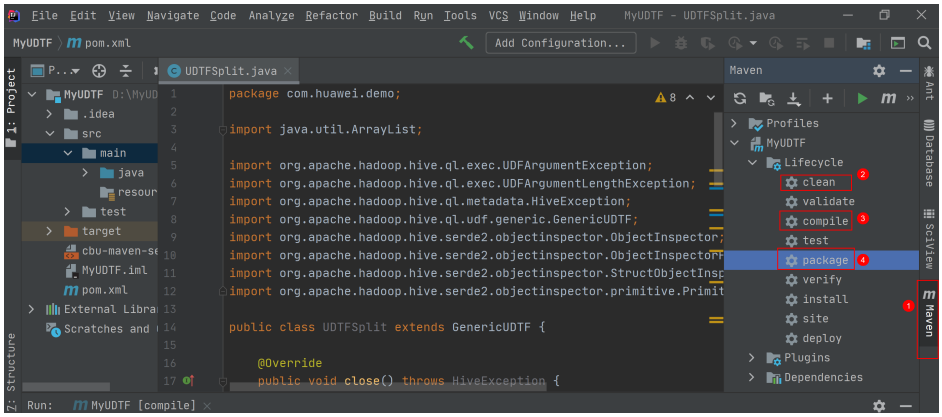
   c. 最后调用close方法，对需要清理的方法进行清理。

3. 编写调试完成代码后，通过IntelliJ IDEA工具编译代码并导出Jar包。

   a. 单击工具右侧的“Maven”，参考下图分别单击“clean”、“compile”对代码进行编译。

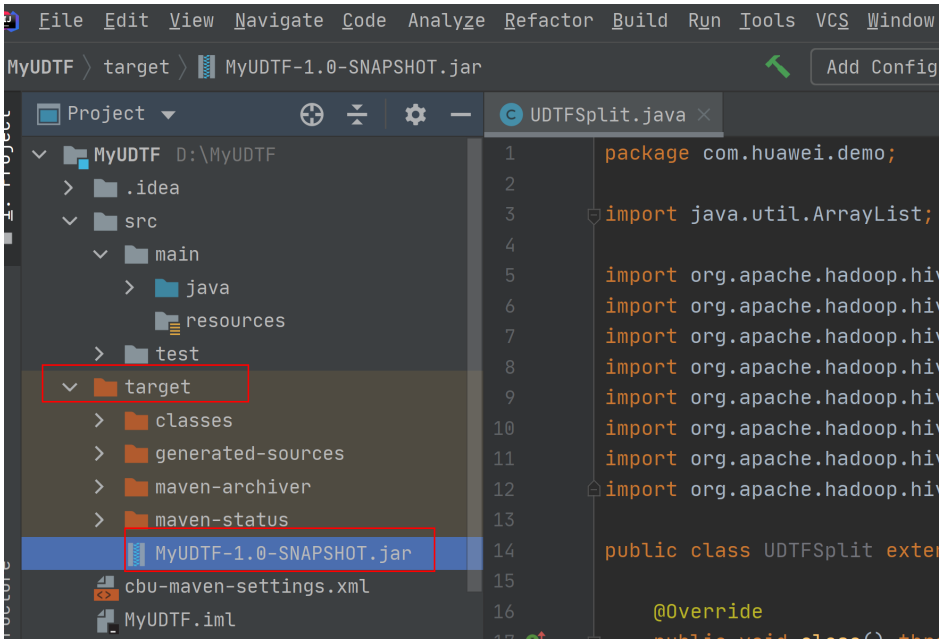
      编译成功后，单击“package”对代码进行打包。

图 3-30 编译打包



打包成功后，生成的Jar包会放到target目录下，以备后用。本示例将会生成到：“D:\MyUDTF\target”下名为“MyUDTF-1.0-SNAPSHOT.jar”。

图 3-31 生成 Jar 包



4. 登录OBS控制台，将生成的Jar包文件上传到OBS路径下。

   说明

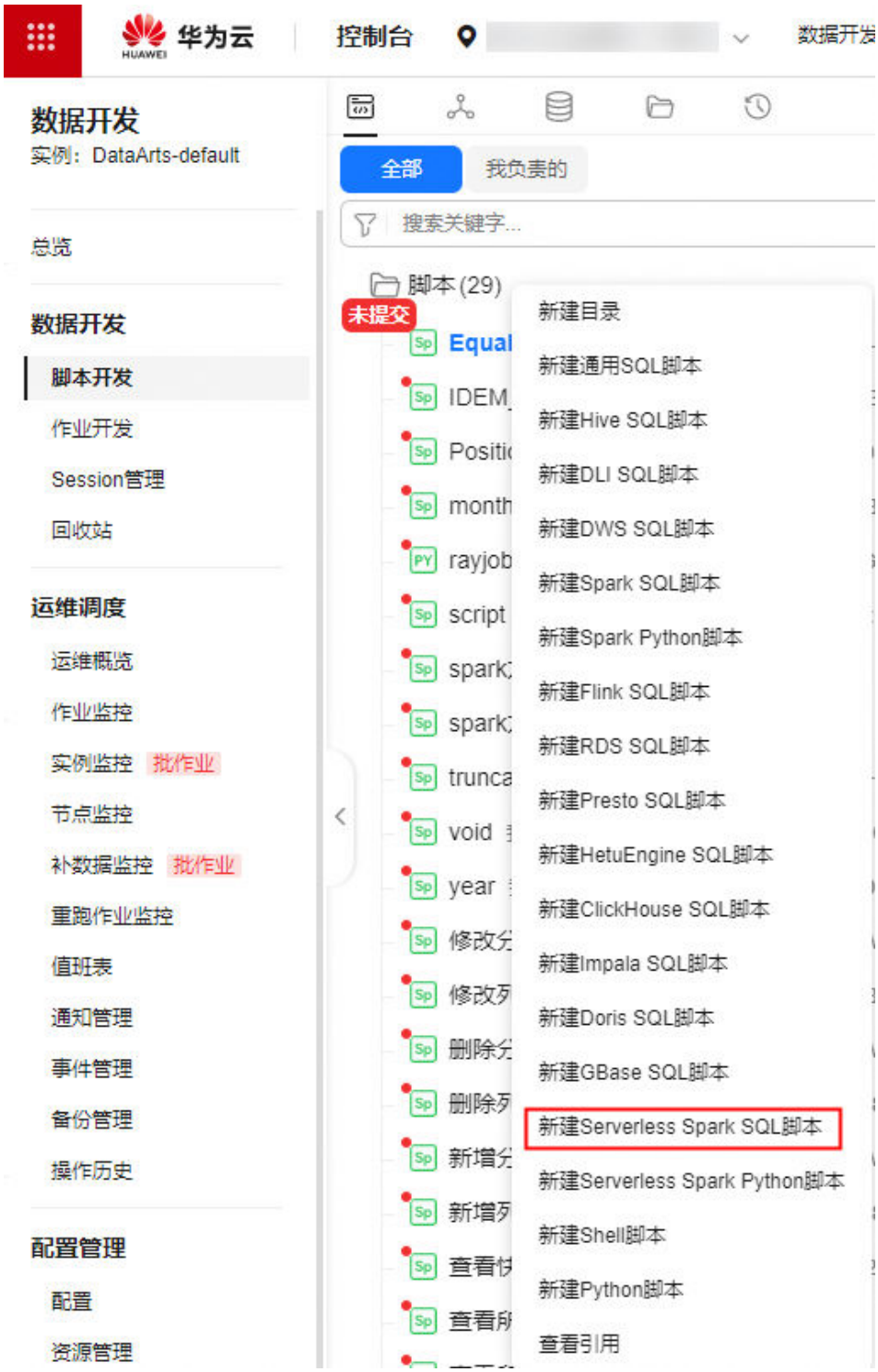
   Jar包文件上传的OBS桶所在的区域需与AI DataLake的工作空间区域相同，不可跨区域执行操作。

5. 创建UDTF函数。

   a. 在AI DataLake管理控制台左下角，单击“DataArts Studio”。

   b. 在“脚本开发”页面，新建Serverless Spark SQL脚本后，选择端点，Catalog和数据库。

图 3-32 新建 Serverless Spark SQL 脚本



c. 在SQL编辑区域，输入创建UDTF函数的命令。

```
CREATE FUNCTION mytestsplit AS 'com.huawei.demo.UDTFSplit' using jar 'obs://aidatalake-test-obs01/MyUDTF-1.0-SNAPSHOT.jar';
```

d. 单击“运行”。

6. 验证和使用创建的UDTF函数。

   在查询语句中使用创建的UDTF函数，如：

```
select mytestsplit('abc:123\;efd:567\;utf:890');
```

7. （可选）删除UDTF函数。

   如果不再使用该Function，可执行以下语句删除UDTF函数:

   ```
   Drop FUNCTION mytestsplit;
   ```

样例代码

UDTFSplit.java完整的样例代码参考如下所示：

```
package com.huawei.demo;
import java.util.ArrayList;

import org.apache.commons.lang.StringUtils;
import org.apache.hadoop.hive.ql.exec.UDFArgumentException;
import org.apache.hadoop.hive.ql.exec.UDFArgumentLengthException;
import org.apache.hadoop.hive.ql.metadata.HiveException;
import org.apache.hadoop.hive.ql.udf.generic.GenericUDTF;
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspector;
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspectorFactory;
import org.apache.hadoop.hive.serde2.objectinspector.StructObjectInspector;
import org.apache.hadoop.hive.serde2.objectinspector.primitive.PrimitiveObjectInspectorFactory;

public class UDTFSplit extends GenericUDTF {

    @Override
    public void close() throws HiveException {
        // TODO Auto-generated method stub

    }

    @Override
    public void process(Object[] args) throws HiveException {
        // TODO Auto-generated method stub
        if(args.length == 0){
            return;
        }
        String input = args[0].toString();
        if(StringUtils.isEmpty(input)){
            return;
        }
        String[] test = input.split(";");
        for (int i = 0; i < test.length; i++) {
            try {
                String[] result = test[i].split(":");
                forward(result);
            } catch (Exception e) {
                continue;
            }
        }

    }

    @Override
    public StructObjectInspector initialize(ObjectInspector[] args) throws UDFArgumentException {
        if (args.length != 1) {
            throw new UDFArgumentLengthException("ExplodeMap takes only one argument");
        }
        if (args[0].getCategory() != ObjectInspector.Category.PRIMITIVE) {
            throw new UDFArgumentException("ExplodeMap takes string as a parameter");
        }

        ArrayList<String> fieldNames = new ArrayList<String>();
        ArrayList<ObjectInspector> fieldOIs = new ArrayList<ObjectInspector>();
        fieldNames.add("col1");
        fieldOIs.add(PrimitiveObjectInspectorFactory.javaStringObjectInspector);
        fieldNames.add("col2");
        fieldOIs.add(PrimitiveObjectInspectorFactory.javaStringObjectInspector);
```



```
        return ObjectInspectorFactory.getStandardStructObjectInspector(fieldNames, fieldOIs);
    }

}
```

3.4 在 Spark SQL 作业中使用 Remote UDF

Using Remote UDF in Spark SQL Jobs

3.4.1 概述

功能描述

Remote UDF（Remote User-Defined Function，远程用户自定义函数）是指将UDF的执行逻辑从主计算引擎剥离，主引擎通过标准接口（如 HTTP、gRPC、Thrift）调用该服务完成计算。Remote UDF将计算逻辑与主引擎分离、实现资源解耦，充分提升Remote UDF复用性与灵活性。

本节内容以FunctionGraph作为Remote UDF 的执行载体，介绍开发Remote UDF到计算引擎调用的操作步骤。

了解更多：

- **UDF与Remote UDF的对比**
- **UDF、UDAF和UDTF的对比**

方案优势

Remote UDF（User Defined Functions）的优势在于风险隔离、资源解耦与成本优化。

- 风险隔离：当关键生产环境需要执行UDF时，直接修改UDF存在执行风险。

  通过Remote UDF将计算逻辑卸载到函数工作流FunctionGraph，既能保持业务功能，又能通过无服务器架构实现风险隔离。
- 资源解耦：对于大模型推理等计算密集型任务，为避免在Spark等集群中长期占用昂贵资源，兼顾功能需求、稳定性和成本效益，将计算密集型操作剥离至FunctionGraph执行
- 成本优化：Remote UDF将default队列还原为纯数据通道，消除UDF引发的风险，使其能为用户提供更经济，更安全的核心数据流服务。

约束与限制

Remote UDF只对单行数据产生作用，适用于单进单出的场景。

环境准备

使用FunctionGraph开发调试Remote UDF时，您需要先安装IDEA并配置Maven，下载依赖JAR包并配置FunctionGraph服务，做好Remote UDF开发前准备工作。

表 3-7 UDF 开发环境

准备项	说明
操作系统	Windows系统，支持Windows7以上版本。
安装JDK	JDK使用1.8版本。
安装和配置IntelliJ IDEA	IntelliJ IDEA为进行应用开发的工具，版本要求使用2019.1或其他兼容版本。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。
下载依赖JAR包	依赖引入下载Jar包：spark-function-sdk_2.13-jar-with-dependencies.jar 获取方式：从https://repo.huaweicloud.com/repository/maven/huaweicloudsdk/地址下载。

开发流程

Remote UDF函数开发流程参考如下：

图 3-33 开发流程

表 3-8 开发流程说明

序号	阶段	操作界面	说明
1	新建Maven工程，配置Project structure	IntelliJ IDEA	编写Remote UDF函数代码。
2	编写Remote UDF函数代码		
3	调试，编译代码并导出Jar包		
4	创建FunctionGraph函数，并上传Jar包	OBS控制台	将生成的UDF函数Jar包文件上传到已创建的FunctionGraph函数中。
5	创建Remote UDF函数	DataArts Studio控制台	在DataArts Studio控制台的“脚本开发”界面，创建UDF函数。

序号	阶段	操作界面	说明
6	验证和使用Remote UDF函数	DataArts Studio控制台	在DataArts Studio控制台的“作业开发”界面，Spark SQL作业中使用创建的UDF函数。

UDF 与 Remote UDF 的对比

- UDF：适合对性能要求高且熟悉查询引擎开发语言的场景。执行效率高，但开发灵活性和安全性相对较低。
- Remote UDF：适合需要调用外部服务、使用多种语言开发或对开发语言灵活性要求高的场景。具有较高的安全性和开发灵活性。

表 3-9 UDF 与 Remote UDF 的对比

特性	UDF	Remote UDF
执行方式	在查询引擎本地执行	通过调用函数托管服务执行
安全性	若UDF代码不稳定，可能影响查询引擎进程	执行失败仅影响UDF自身，不会导致整体作业进程。
性能	执行效率高	但在向量化和批处理模式下，性能与本地UDF相当
开发灵活性	与查询引擎代码耦合度高，开发限制较多	灵活，可调用任意其他服务或程序库，满足多样业务需求
开发成本	需熟悉查询引擎的开发环境和语言，开发成本较高	一次性开发投入，开发成本相对较低
适用场景	对性能要求极高的场景，且开发团队熟悉查询引擎的开发语言	对开发语言灵活性要求高、需要调用外部服务或程序库的场景

UDF、UDAF 和 UDTF 的对比

- UDF：创建一个计算两个数字之和的函数。
- UDAF：创建一个计算平均值的函数。
- UDTF：创建一个函数，用于将一个包含多个值的字符串拆分成多行。

表 3-10 UDF、UDAF 和 UDTF 的对比

特性	UDF（User-Defined Function）	UDAF（User-Defined Aggregate Function）	UDTF（User-Defined Table-Generating Function）
输入参数	一行数据的一个或多个字段	多行数据的多个字段	一行或多行数据的一个或多个字段
输出结果	单个标量值	单个聚合结果	多行数据，每行可以有多个字段
主要用途	简单的数据转换和计算，如字符串处理、数学运算等	数据聚合计算，如求和、平均值、计数等	数据展开和复杂结构解析，如拆分字符串、展开数组等

- **UDF示例**：

  创建一个函数，将输入的数字加1。

  ```
  CREATE FUNCTION add_one AS 'com.example.AddOneUDF';
  SELECT add_one(age) FROM users;
  ```

  假设users表中有一列age，值为25，该函数会输出26。

- **UDAF示例**：

  计算某一列的最大值。

  ```
  CREATE FUNCTION max AS 'com.example.MaxUDAF';
  SELECT max(age) FROM users;
  ```

  假设users表中有一列age，值为25, 30, 35，该函数会输出35。

- **UDTF示例**：

  将一个JSON字符串解析为多行数据。

  ```
  CREATE FUNCTION parse_json AS 'com.example.ParseJsonUDTF';
  SELECT parse_json(json_column) FROM users;
  ```

  假设users表中有一列json_column，值为'{"name":"Alice","age":25}, {"name":"Bob","age":30}'，该函数会输出两行，每行包含解析后的name和age字段。

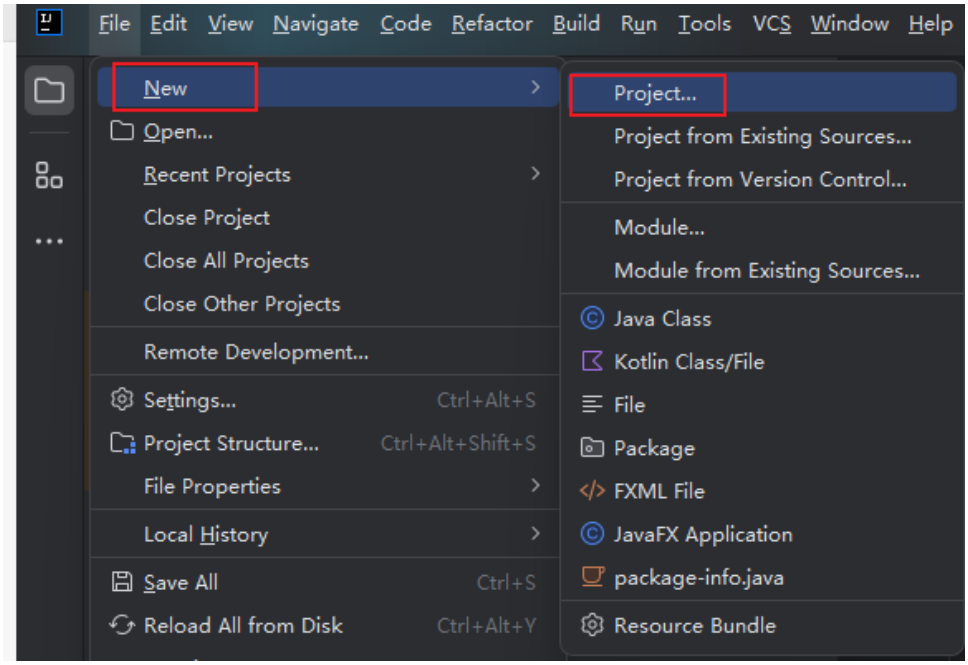
3.4.2 新建 Maven 工程，配置 Project structure

新建 Maven 工程，配置 Project structure

本例使用IntelliJ IDEA 2024.3.2.2工具操作演示

1. 打开IntelliJ IDEA，选择“File > New > Project”。

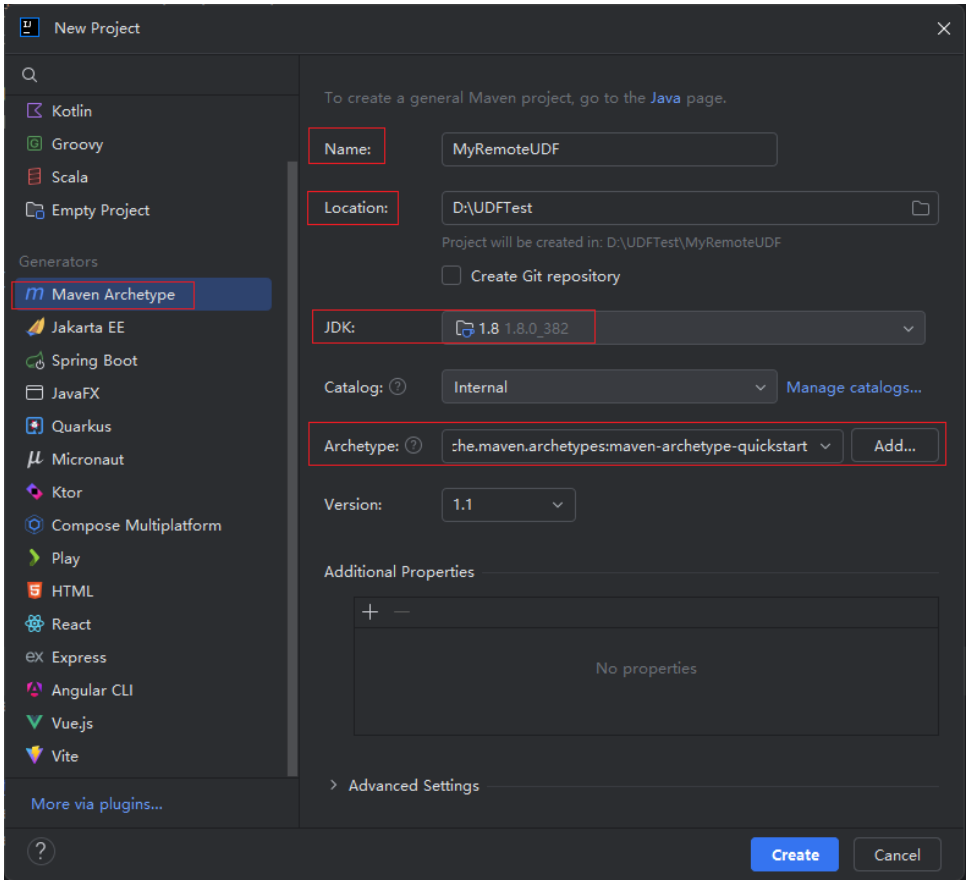
图 3-34 新建 Project



2. 选择Maven Archetype，配置以下信息：
   - Name：自定义项目名称。本例设置项目名称为MyRemoteUDF。
   - Location：选择项目存储路径。本例项目保存路径为D:\UDFTest。
   - JDK选择1.8版本的JDK。
   - Archetype选择‘org.apache.maven.archetypes:maven-archetype-quickstart’。

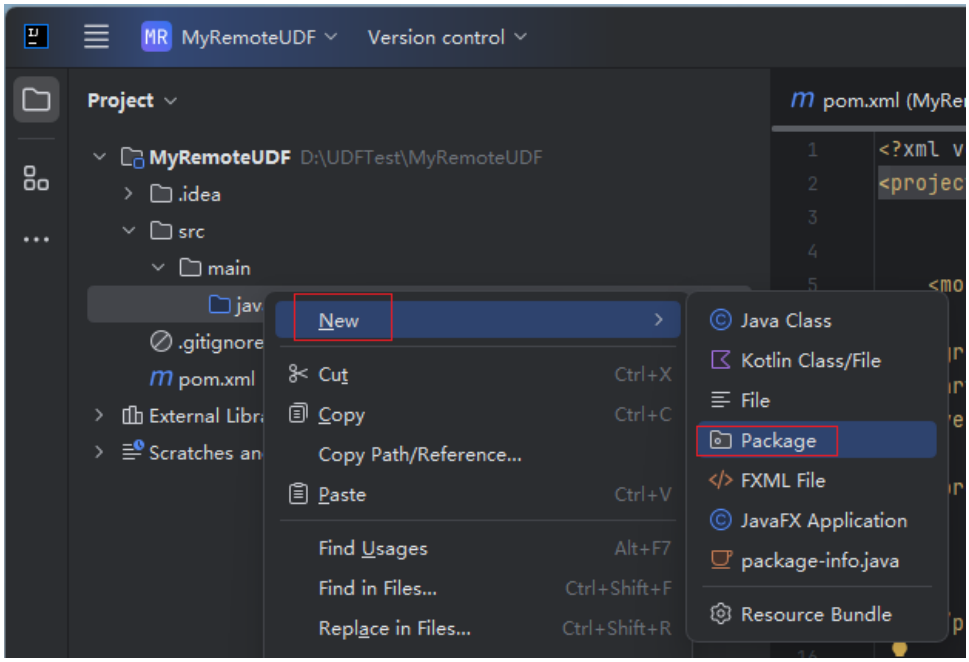
   单击“Create”创建项目。

图 3-35 新建 Maven 项目



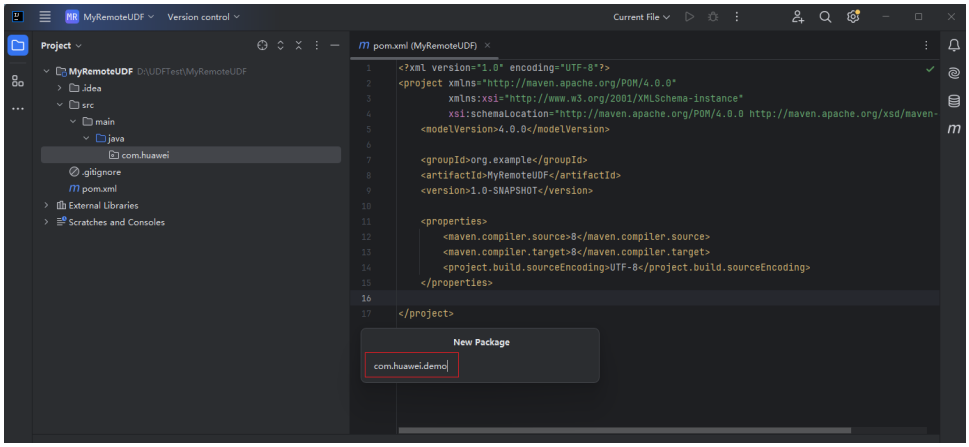
3. 在工程路径的“src > main > java”文件夹单击鼠标右键，选择“New > Package”，新建Package。

图 3-36 新建 Package



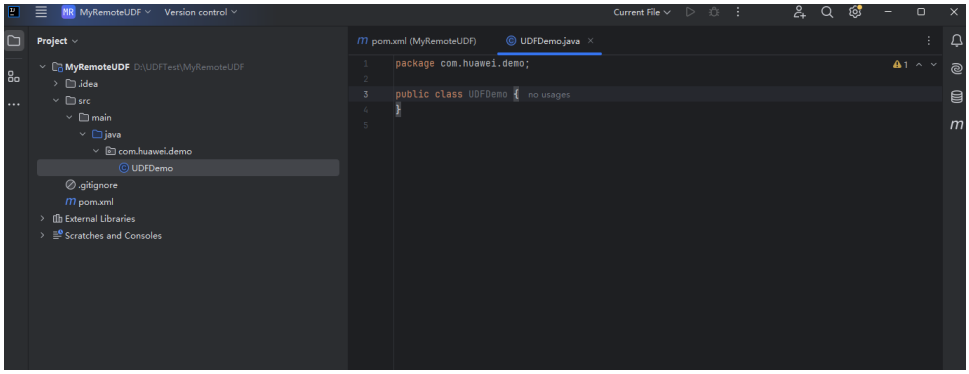
按需定义Package名称，本示例定义为：“com.huawei.demo”，完成后回车。

图 3-37 定义 Package 名称



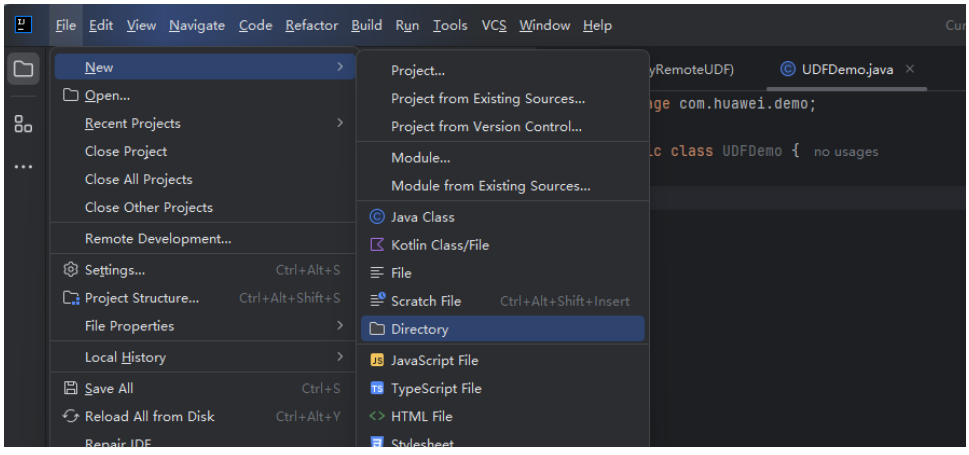
4. 在包路径下新建Java Class文件，本示例定义为：UDFDemo。

图 3-38 新建 Java Class 文件



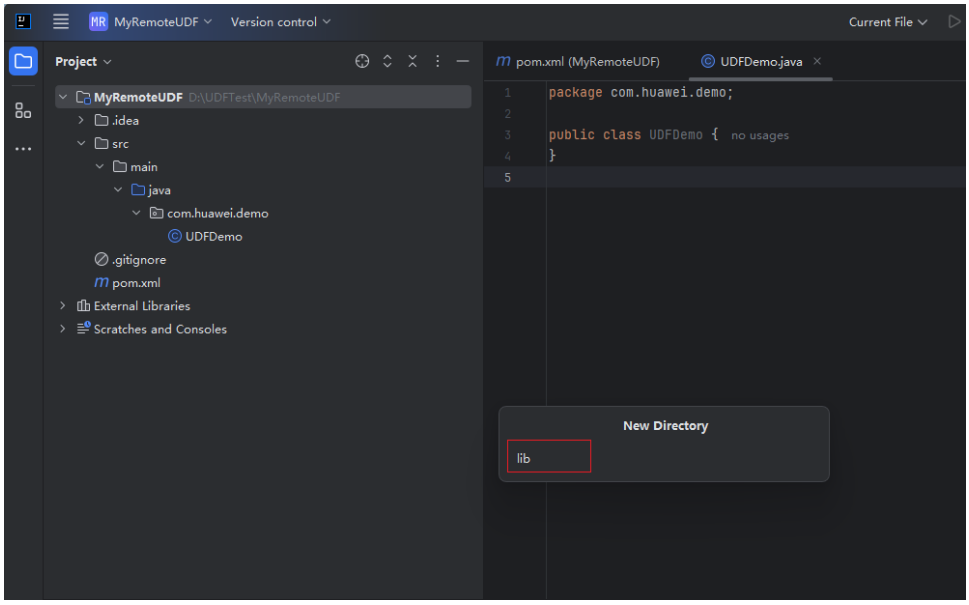
5. 添加JAR包存放路径。
   单击“File > New > Directory”，新建文件夹，用于存放示例JAR包。

图 3-39 添加 JAR 包存放路径



文件夹名称本示例定义为lib，回车确定。

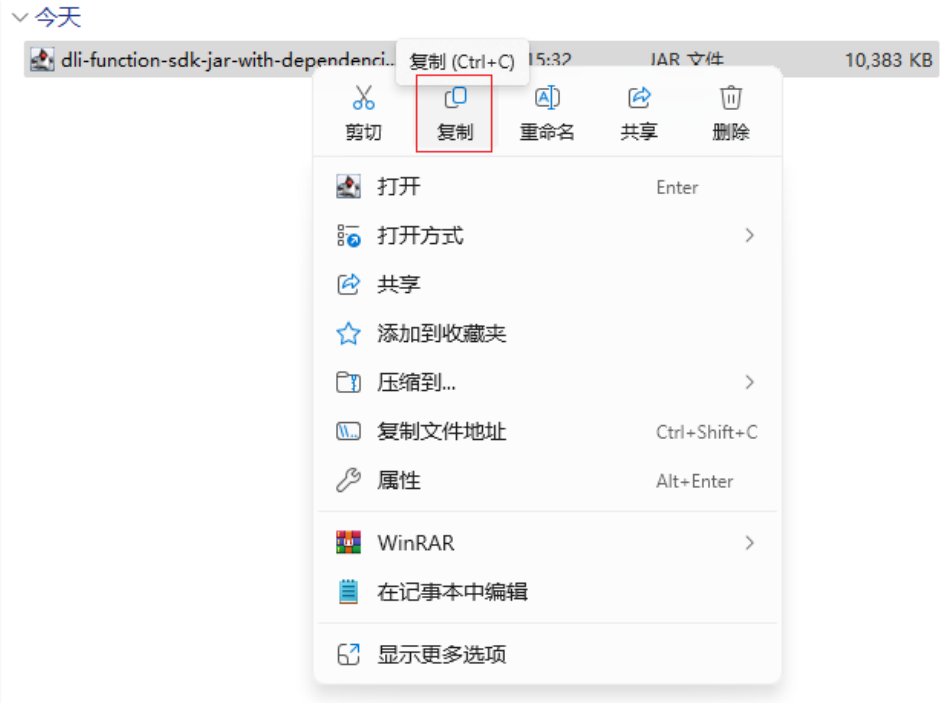
图 3-40 定义文件夹名称



6. 导入JAR包。
   a. 下载JAR包

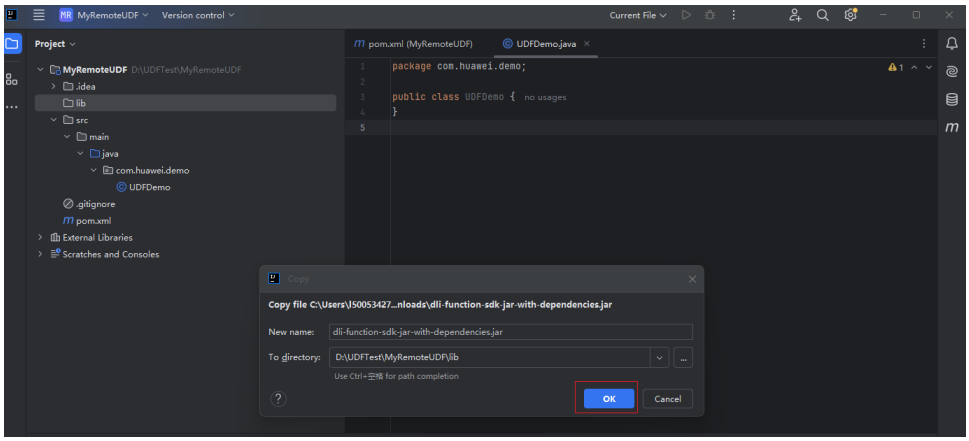
      从https://repo.huaweicloud.com/repository/maven/huaweicloudsdk/com/huawei/mrs地址下载JAR包。
   b. 下载结束后右键单击此JAR包，单击复制。

图 3-41 复制 JAR 包



7. 找到项目lib目录，将其粘贴到lib目录下，在弹出的对话框中选择OK。

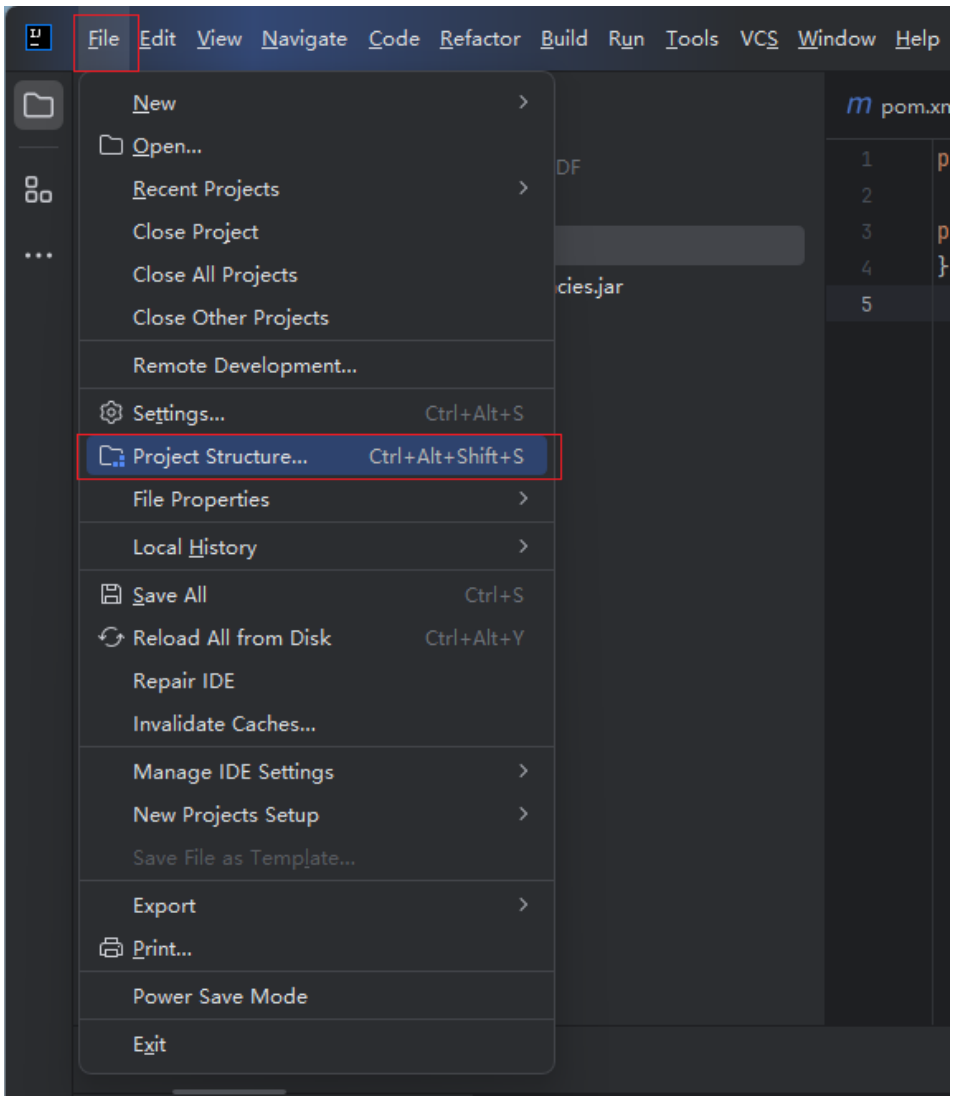
图 3-42 在 lib 目录下粘贴 JAR 包



8. 配置“Project Structure”。

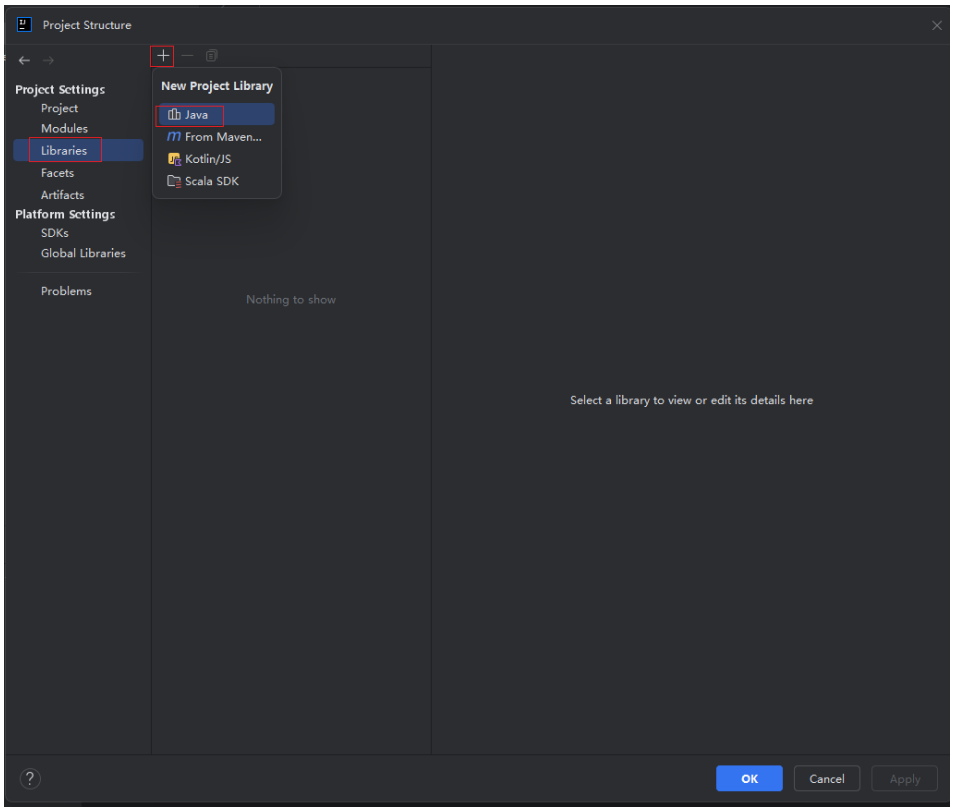
   a. 单击“File > Project Structure”。

图 3-43 单击 Project Structure



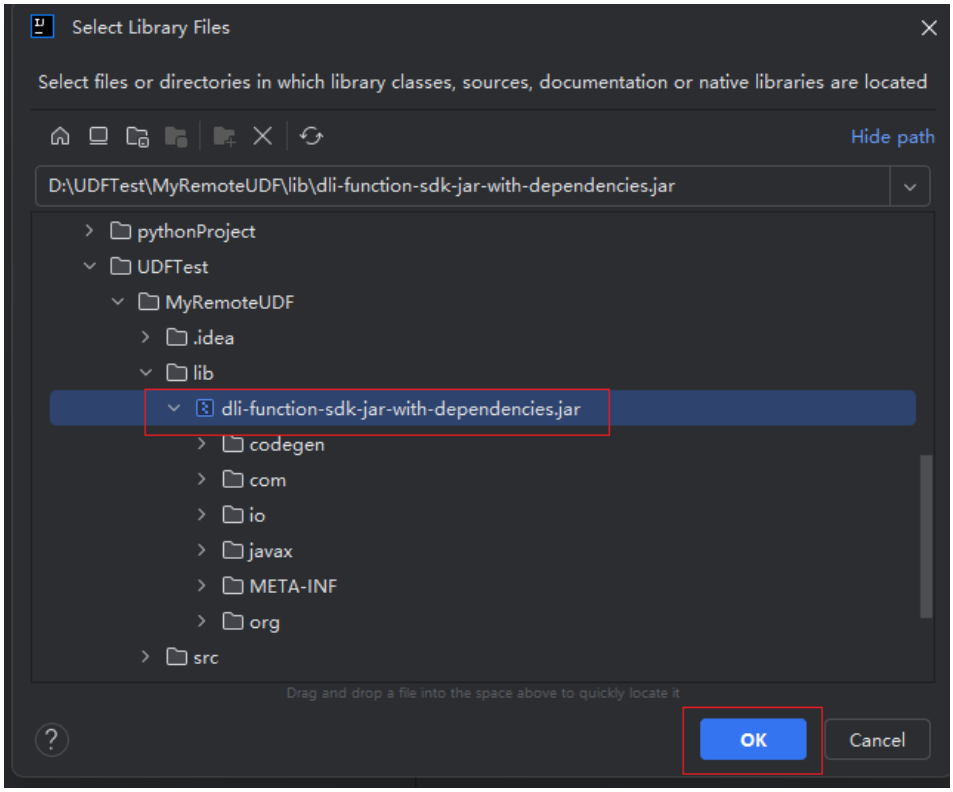
b. 单击Libraries，然后单击“+”，单击JAVA。

图 3-44 单击 JAVA



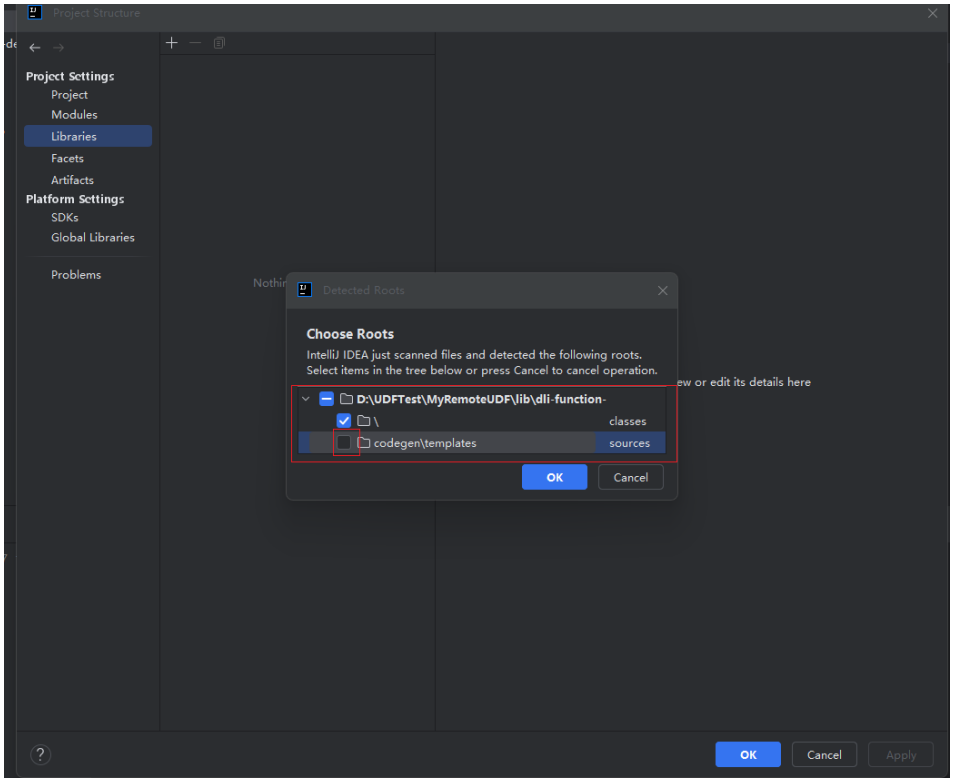
c. 选择lib目录下添加的jar包，单击OK。

图 3-45 选择 lib 目录下添加的 jar 包



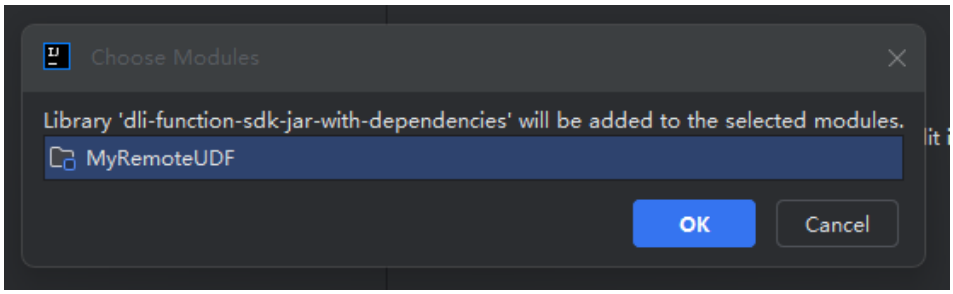
d. 选择根目录，只选取第一项，之后单击OK。

图 3-46 选择根目录



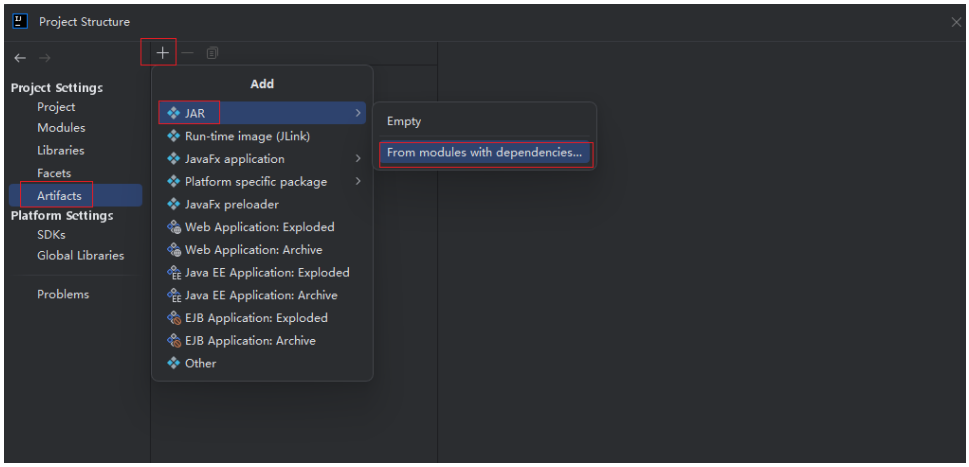
e. 按照提示单击“OK”。

图 3-47 单击“OK”



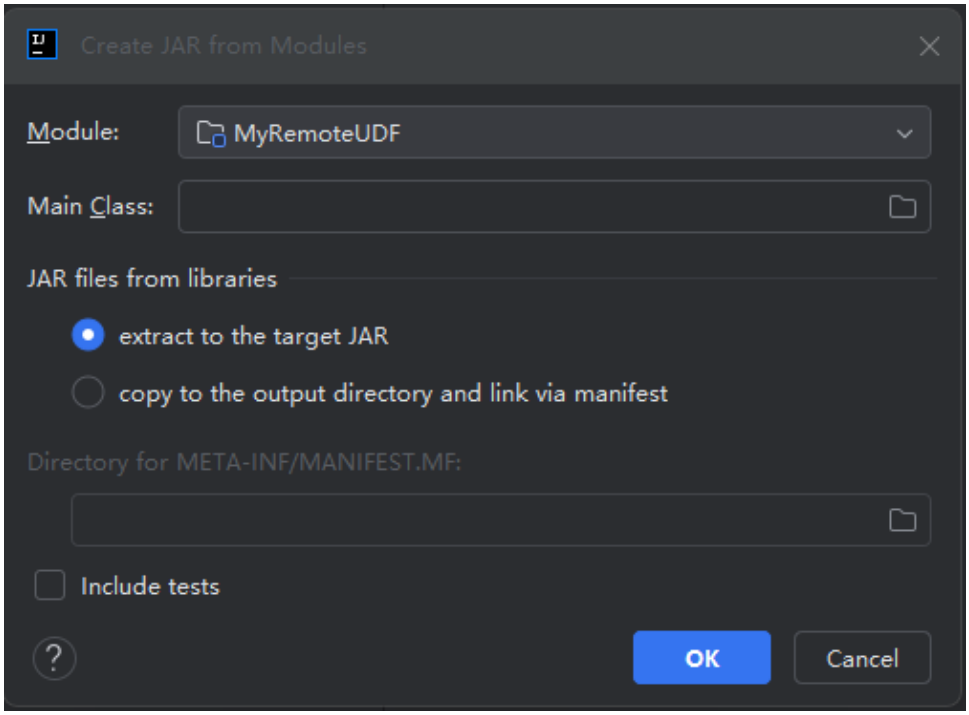
9. 单击Artifacts，并依次单击“+”，JAR，选择From modules with dependencies。

图 3-48 选择 From modules with dependencies



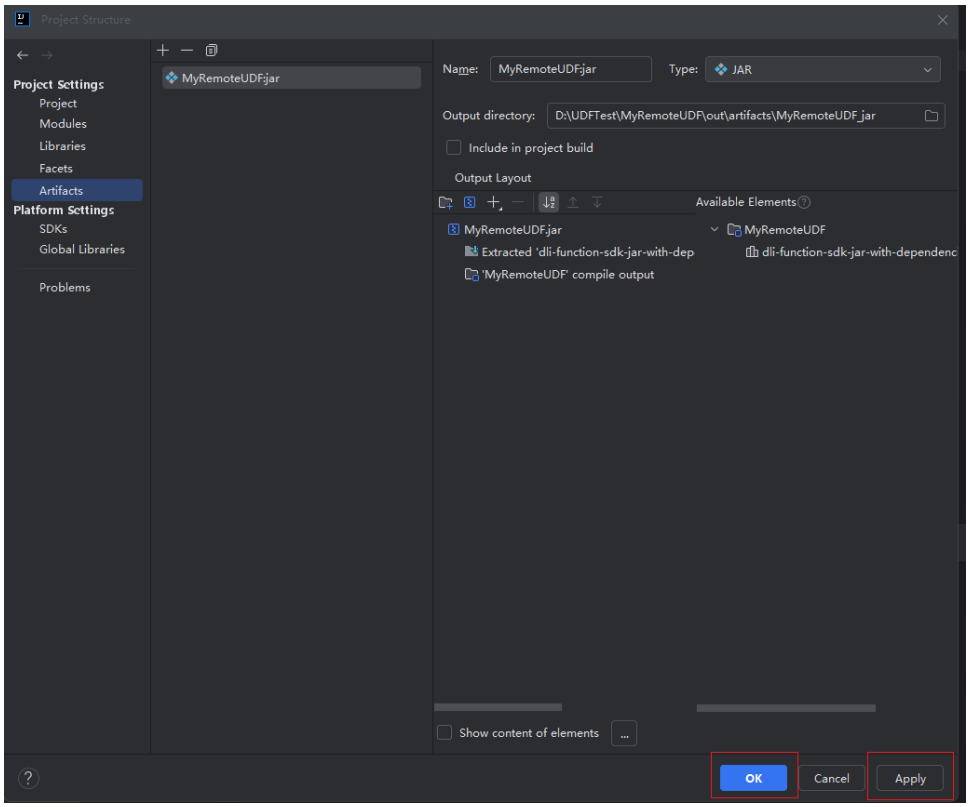
10. 在JAR files from libraries中开启第一项，单击OK。

图 3-49 单击“OK”



11. 依次单击Apply和OK。

图 3-50 Artifacts 配置



3.4.3 编写 UDF 函数代码并导出 Jar 包

步骤 1：编写 UDF 函数代码

UDF函数实现，主要注意以下几点：

- 自定义UDF需要继承UDFHandler。
- 在方法上添加@TypeResolve注解，用于表明函数的入参和出参。
  入参和出参通过“->”分隔。入参和出参类型（resolveType）以及对应的SQL和Java数据类型请参考表3-11。

  表 3-11 resolveType 类型

SQL type	Java type	resolveType
BooleanType	Boolean.class	BOOLEAN
ByteType	Byte.class	BYTE
ShortType	Short.class	SHORT
IntegerType	Integer.class	INT
LongType	Long.class	LONG
FloatType	Float.class	FLOAT
DoubleType	Double.class	DOUBLE
StringType	String.class	STRING
BinaryType	byte[].class	BINARY
DecimalType.Fixed(precision, scale)	BigDecimal.class	Decimal
DateType	LocalDate.class	DATE
TimestampType	ZonedDateTime.class	TIMESTAMP
TimestampNTZType	LocalDateTime.class	TIMESTAMP_NTZ
YearMonthIntervalType	Period.class	INTERVAL_YEAR_MONTH
DayTimeIntervalType	Duration.class	INTERVAL_DAY_TIME
List	List.class	LIST
Map	Map.class	MAP
Struct	Map.class	STRUCT

- 添加@deterministic注解：
  表明函数是确定性的，即在给定相同的输入时，总是返回相同的结果。

详细UDF函数实现，可以参考如下样例代码：

- 功能描述：实现两个Integer类型整数的加法运算，并返回计算结果
- 示例代码：

```
package com.huawei.demo;

import com.huawei.spark.function.types.FunctionType;
import com.huawei.spark.function.types.TypeResolve;
import com.huawei.spark.function.handlers.UDFHandler;

@FunctionType(deterministic = true)
public class UDFDemo extends UDFHandler {
    @FunctionType(deterministic = false)
    @TypeResolve("INT, INT->INT")
    public Integer testIntegerTypeAdd(Integer col1, Integer col2) {
        return col1 + col2;
    }
}
```

- 参数说明：

  col1：第一个整型参数（Integer对象类型）

  col2：第二个整型参数（Integer对象类型）

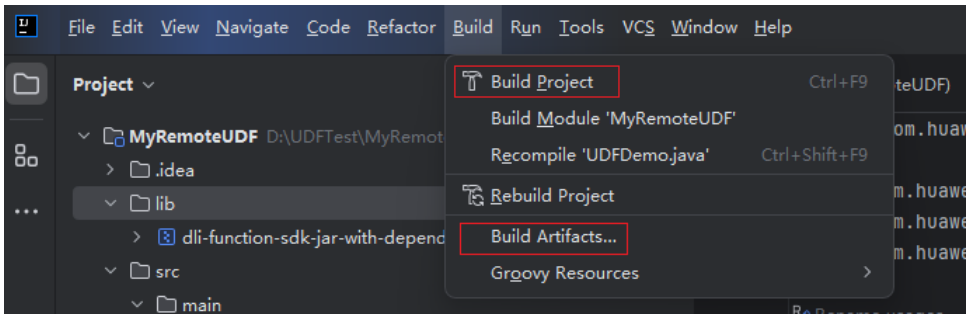
- 返回值：返回col1与col2的算术和

步骤 2：导出 Jar 包

编写调试完成代码后，通过IntelliJ IDEA工具编译代码并导出Jar包。

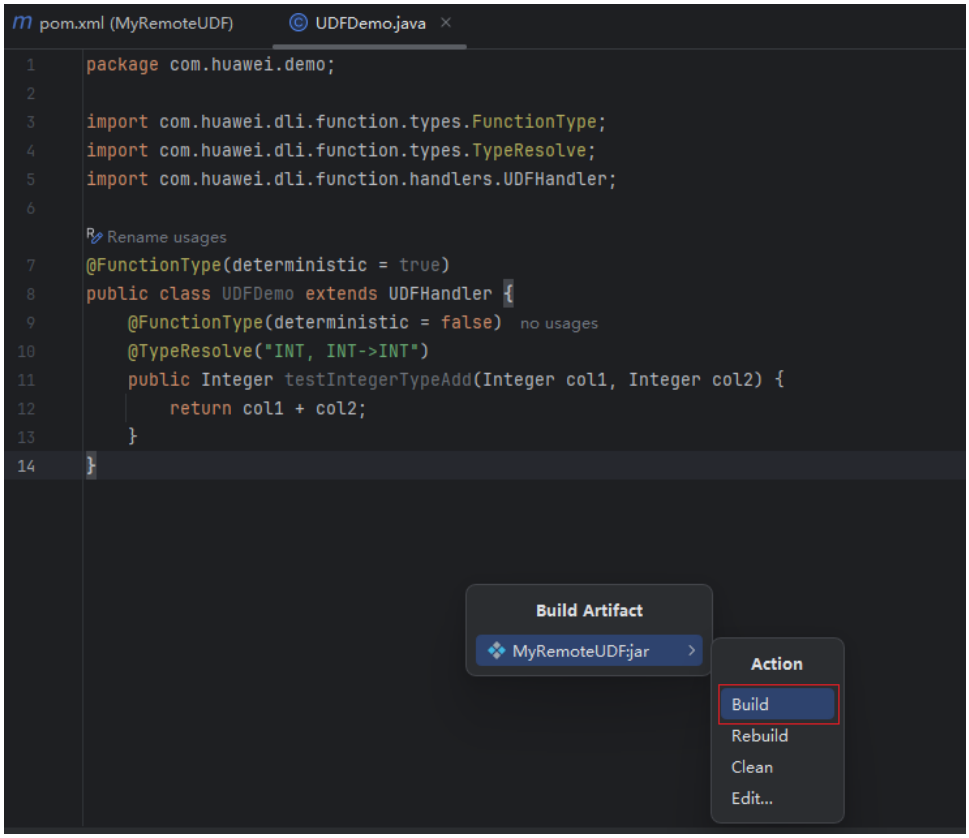
1. 单击“Build”，参考下图分别单击“Build Project ”、“Build Artifacts...”分别对代码进行编译和打包。

   图 3-51 编译和打包



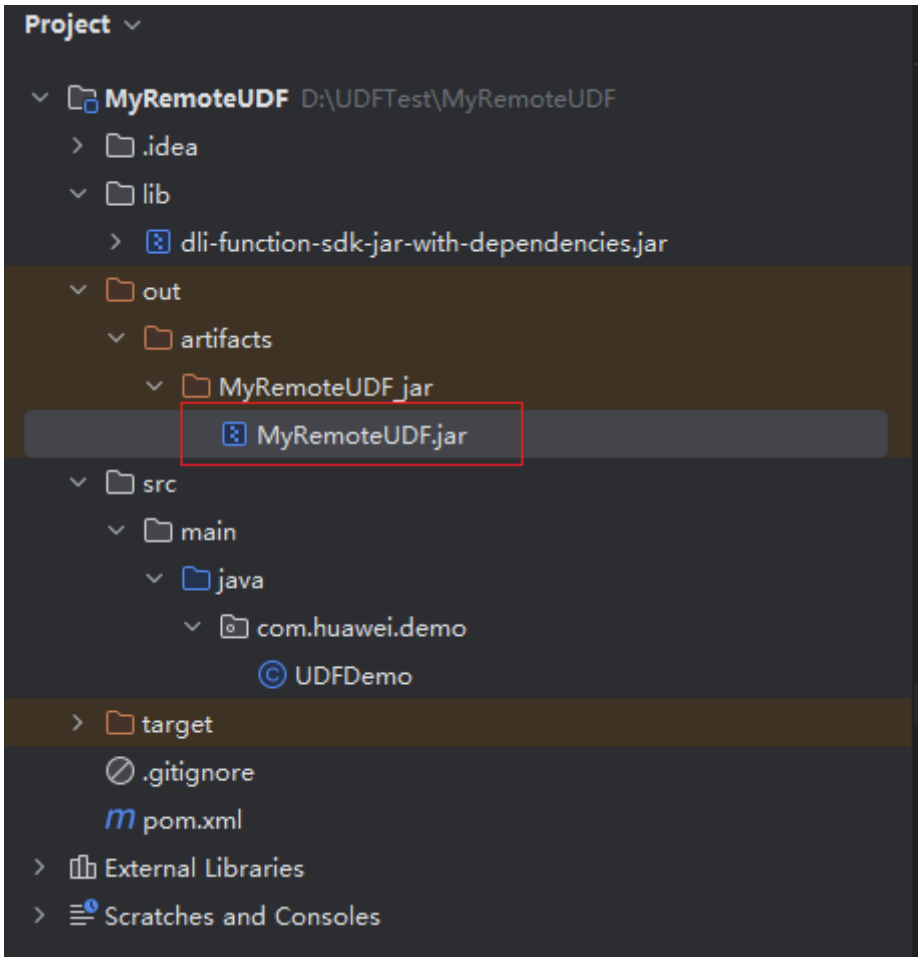
2. 在弹出的对话框中单击Build。

图 3-52 编译和打包



3. 完成上述步骤之后，可以在项目目录当中看到out文件夹，展开out目录。
   其中的MyRemoteUDF.jar就是之后要上传到函数工作流FunctionGraph当中的jar包。

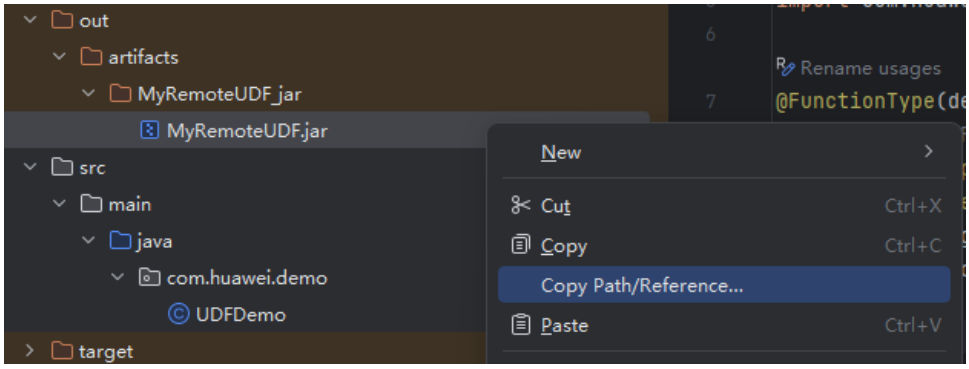
图 3-53 查看 MyRemoteUDF.jar



4. 右键单击MyRemoteUDF.jar，选择Copy Path/Reference，直接复制文件所在目录，方便之后上传JAR包使用。

   本示例将会生成到：“D:\UDFTest\MyRemoteUDF\out\artifacts\MyRemoteUDF_jar\”目录下，文件名为“MyRemoteUDF.jar”。

图 3-54 复制文件所在目录



3.4.4 创建 FunctionGraph 函数

前提条件：

了解**FunctionGraph业务使用流程**。

准备工作

在创建FunctionGraph函数之前，您需要获取这两个配置项。

表 3-12 配置项说明

配置项	配置参数
functiongraph Endpoint	spark.hadoop.fs.functiongraph.endpoint=functiongraph.XXYYZZ.huawei.com
用户项目ID	spark.hadoop.fs.user.projectId=xxxxxxxxxc9782fxxxxxxxxxxxxxxx

说明

提交Spark SQL或Spark Job作业时，需要设置以上两个配置项。

步骤 1：创建函数

1. 登录函数工作流FunctionGraph管理控制台。
2. 在左侧列表中单击“函数 > 函数列表”，单击“创建函数”

图 3-55 创建函数

3. 单击“创建空白函数”，并配置函数的基本信息。

   详细参数说明请参考**函数属性**。

   - 函数类型：本例为事件函数。
   - 运行时环境：Java 8。
   - 委托：当函数工作流服务需与其他云服务协同工作时，需要创建云服务委托，授权FunctionGraph使用这些云服务，并在FunctionGraph中为函数配置委托权限，以实现与其他云服务的协同工作。
   - FunctionGraph单次调用数据量有**约束与限制**。函数如果超过4MB，建议配置访问OBS的委托权限。具体操作请参考**相关操作：配置FunctionGraph访问其他云服务的委托**。

图 3-56 创建函数

步骤 2：函数设置

1. **设置函数执行入口**
   a. 函数创建完成后，在列表单击函数名称进入函数详情页面。
   b. 选择“设置 > 常规设置”，在“函数执行入口”中设置UDF的执行入口。
   UDF的执行入口格式为[包名].[类名].[执行函数名]，
      - [包名].[类名]为编写的UDF函数所在类，
      - 执行函数名固定为handleRequest。

      本示例设置为：com.huawei.demo.UDFDemo.handleRequest，
   c. 设置完成之后单击保存按钮。
2. **设置执行超时时间和内存**
   a. 函数创建完成后，在列表单击函数名称进入函数详情页面。
   b. 选择“设置 > 常规设置”，在“函数执行入口”中设置执行时间和内存。
   根据UDF复杂情况配置执行**超时时间和内存大小**。内存大小与**计费**关联。
   本示例选择执行时间为15秒，内存为1024MB。
   c. 设置完成之后单击保存按钮。

图 3-57 常规设置

3. **设置类隔离信息**

   a. 函数创建完成后，在列表单击函数名称进入函数详情页面。

   b. 选择“设置 > 高级设置”，配置“类隔离选项”。

      类隔离开启后将使用单独的类加载器加载上传的代码和依赖，解决依赖和运行时的冲突，如果无依赖冲突问题无需打开。

   本示例中关闭类隔离选项，设置完成后单击保存按钮。

图 3-58 设置类隔离

步骤 3：上传 Jar 包

1. 在FunctionGraph管理控制台，函数列表中单击函数名称进入函数详情页面。
2. 单击"代码"，在右侧上传代码下拉框当中选择JAR文件，将之前导出的JAR包上传Upload the exported jar.

图 3-59 上传 JAR 文件

3. 上传成功之后，可以在函数详情页面复制URN（Uniform Resource Name）。URN是函数的唯一标识函数，在下一步调用函数时需要用到。

图 3-60 复制 URN

相关操作：配置 FunctionGraph 访问其他云服务的委托

1. 登录管理控制台。
2. 单击右上方登录的用户名，在下拉列表中选择“统一身份认证”。
3. 在左侧导航栏中，单击“委托”。
4. 在“委托”页面，单击“创建委托”。
5. 在“创建委托”页面，设置如下参数：
   - 委托名称：按需填写。
   - 委托类型：选择“云服务”。
   - 云服务：（“委托类型”选择“云服务”时出现此参数项。）在下拉列表中选择“FunctionGraph"。
   - 持续时间：选择“永久”。
   - 描述：非必选，可以填写“拥有OBS OperateAccess权限的委托”。

图 3-61 创建委托

6. 配置完委托的基本信息后，单击“立即授权”。
7. 进入“选择策略”页面，在右方搜索框中搜索OBS Administrator权限并勾选。勾选完成后单击“下一步”。
8. 单击“下一步”，选择委托的授权范围。

   了解更多授权操作说明请参考**创建用户组并授权**。

   - 所有资源：授权后，IAM用户可以根据权限使用账号中所有资源，包括企业项目、区域项目和全局服务资源。
   - 全局服务资源：全局服务部署时不区分区域，访问全局级服务，不需要切换区域，全局服务不支持基于区域项目授权。如对象存储服务（OBS）、内容分发网络（CDN）等。授权后，用户根据权限使用全局服务的资源。
   - 指定区域项目资源：授权后，IAM用户根据权限使用所选区域项目中的资源，未选择的区域项目中的资源，该IAM用户将无权访问。
   - 指定企业项目资源：授权后，IAM用户根据权限使用所选企业项目中的资源。如企业项目A包含资源B，资源B部署在北京四和上海二，IAM用户所在用户组关联企业项目A后，北京四和上海二的资源B用户都可访问，不在企业项目A内的其他资源，该IAM用户将无权访问。

   本例选择的是OBS策略，因此选择全局服务资源。
9. 单击“确定”，完成授权。
10. 授权后需等待15-30分钟才可生效。

    授权生效后返回函数基本信息界面，选择对应的委托，权限策略当中会显示已授权的策略。

3.5 在 Spark SQL 作业中使用 Remote UDAF

3.5.1 概述

功能描述

Remote UDAF（Remote User-Defined Aggregate Function）为用户自定义聚合函数，适用于多进一出业务场景。即其输入与输出是多对一的关系，将多条输入记录聚合成一个输出值。

本节内容以FunctionGraph作为Remote UDAF 的执行载体，介绍开发Remote UDAF到计算引擎调用的操作步骤。

方案优势

Remote UDAF的优势在于风险隔离、资源解耦与成本优化。

- 风险隔离：当关键生产环境需要执行UDAF时，虽然技术上支持，但存在执行风险。

  通过Remote UDAF将计算逻辑卸载到函数工作流FunctionGraph，既能保持业务功能，又能通过无服务器架构实现风险隔离。
- 资源解耦：对于大模型推理等计算密集型任务，为避免在Spark等集群中长期占用昂贵资源，兼顾功能需求、稳定性和成本效益，将计算密集型操作剥离至FunctionGraph执行
- 成本优化：Remote UDAF将default队列还原为纯数据通道，消除UDAF引发的风险，使其能为用户提供更经济，更安全的核心数据流服务。

约束与限制

Remote UDAF只对单行数据产生作用，适用于单进单出的场景。

环境准备

使用FunctionGraph开发调试Remote UDAF时，您需要先安装IDEA并配置Maven，下载依赖JAR包并配置FunctionGraph服务，做好Remote UDAF开发前准备工作。

表 3-13 UDAF 开发环境

准备项	说明
操作系统	Windows系统，支持Windows7以上版本。
安装JDK	JDK使用1.8版本。
安装和配置IntelliJ IDEA	IntelliJ IDEA为进行应用开发的工具，版本要求使用2019.1或其他兼容版本。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。
下载依赖JAR包	依赖引入下载Jar包：spark-function-sdk_2.13-jar-with-dependencies.jar 获取方式：从https://repo.huaweicloud.com/repository/maven/huaweicloudsdk/地址下载。

UDAF 与 Remote UDAF 的对比

- UDAF：适合对性能要求高且熟悉查询引擎开发语言的场景。执行效率高，但开发灵活性和安全性相对较低。
- Remote UDAF：适合需要调用外部服务、使用多种语言开发或对开发语言灵活性要求高的场景。具有较高的安全性和开发灵活性。

表 3-14 UDAF 与 Remote UDAF 的对比

特性	UDAF	Remote UDAF
执行方式	在查询引擎本地执行	通过调用函数托管服务执行
安全性	若UDAF代码不稳定，可能影响查询引擎进程	执行失败仅影响UDAF自身，不会导致整体作业进程。
性能	执行效率高	但在向量化和批处理模式下，性能与本地UDAF相当
开发灵活性	与查询引擎代码耦合度高，开发限制较多	灵活，可调用任意其他服务或程序库，满足多样业务需求
开发成本	需熟悉查询引擎的开发环境和语言，开发成本较高	一次性开发投入，开发成本相对较低
适用场景	对性能要求极高的场景，且开发团队熟悉查询引擎的开发语言	对开发语言灵活性要求高、需要调用外部服务或程序库的场景

UDF、UDAF 和 UDTF 的对比

- UDF：创建一个计算两个数字之和的函数。
- UDAF：创建一个计算平均值的函数。
- UDTF：创建一个函数，用于将一个包含多个值的字符串拆分成多行。

表 3-15 UDF、UDAF 和 UDTF 的对比

特性	UDF（User-Defined Function）	UDAF（User-Defined Aggregate Function）	UDTF（User-Defined Table-Generating Function）
输入参数	一行数据的一个或多个字段	多行数据的多个字段	一行或多行数据的一个或多个字段
输出结果	单个标量值	单个聚合结果	多行数据，每行可以有多个字段
主要用途	简单的数据转换和计算，如字符串处理、数学运算等	数据聚合计算，如求和、平均值、计数等	数据展开和复杂结构解析，如拆分字符串、展开数组等

- **UDF示例**：
  创建一个函数，将输入的数字加1。
  ```
  CREATE FUNCTION add_one AS 'com.example.AddOneUDF';
  SELECT add_one(age) FROM users;
  ```
  假设users表中有一列age，值为25，该函数会输出26。
- **UDAF示例**：
  计算某一列的最大值。
  ```
  CREATE FUNCTION max AS 'com.example.MaxUDAF';
  SELECT max(age) FROM users;
  ```
  假设users表中有一列age，值为25, 30, 35，该函数会输出35。
- **UDTF示例**：
  将一个JSON字符串解析为多行数据。
  ```
  CREATE FUNCTION parse_json AS 'com.example.ParseJsonUDTF';
  SELECT parse_json(json_column) FROM users;
  ```
  假设users表中有一列json_column，值为'{"name":"Alice","age":25}, {"name":"Bob","age":30}'，该函数会输出两行，每行包含解析后的name和age字段。

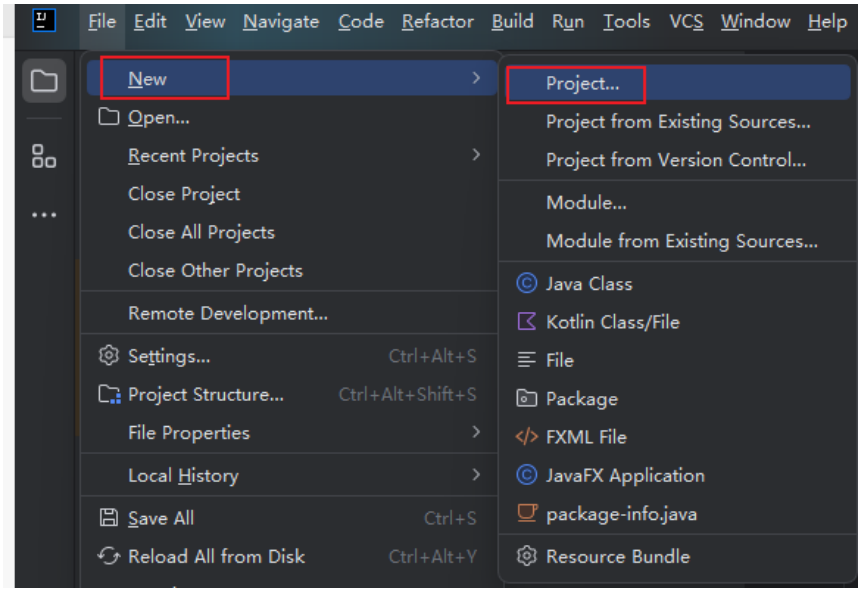
3.5.2 新建 Maven 工程，配置 Project structure

新建 Maven 工程，配置 Project structure

本例使用IntelliJ IDEA 2024.3.2.2工具操作演示

1. 打开IntelliJ IDEA，选择“File > New > Project”。

   图 3-62 新建 Project

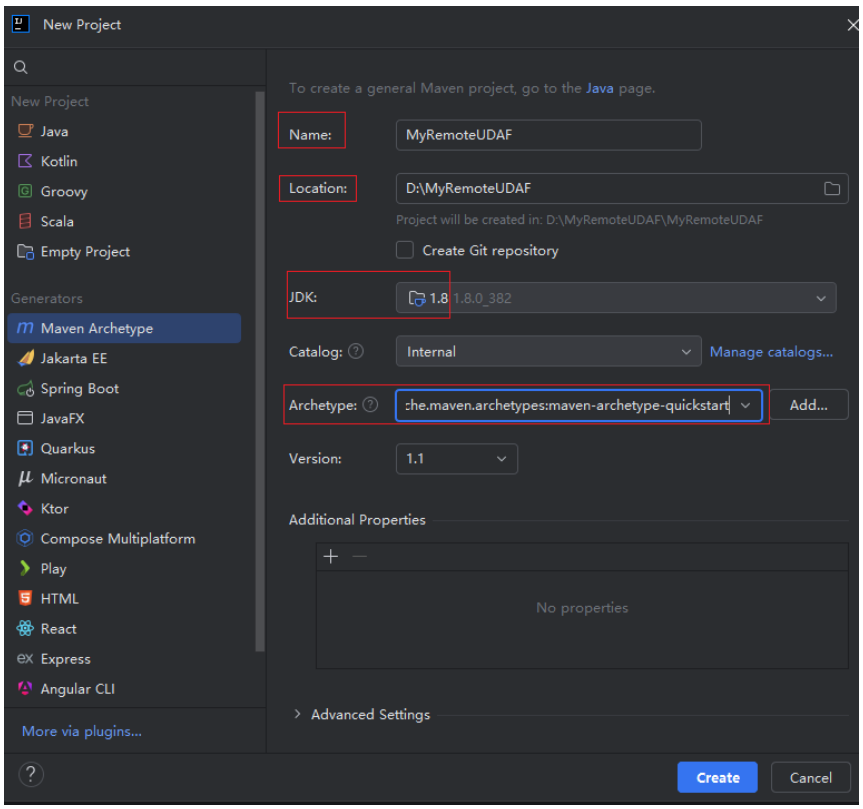
   

2. 选择Maven Archetype，配置以下信息：
   - Name：自定义项目名称。本例设置项目名称为MyRemoteUDAF。
   - Location：选择项目存储路径。本例项目保存路径为D:\UDAFTest。
   - JDK选择1.8版本的JDK。

– Archetype选择‘org.apache.maven.archetypes:maven-archetype-quickstart’。

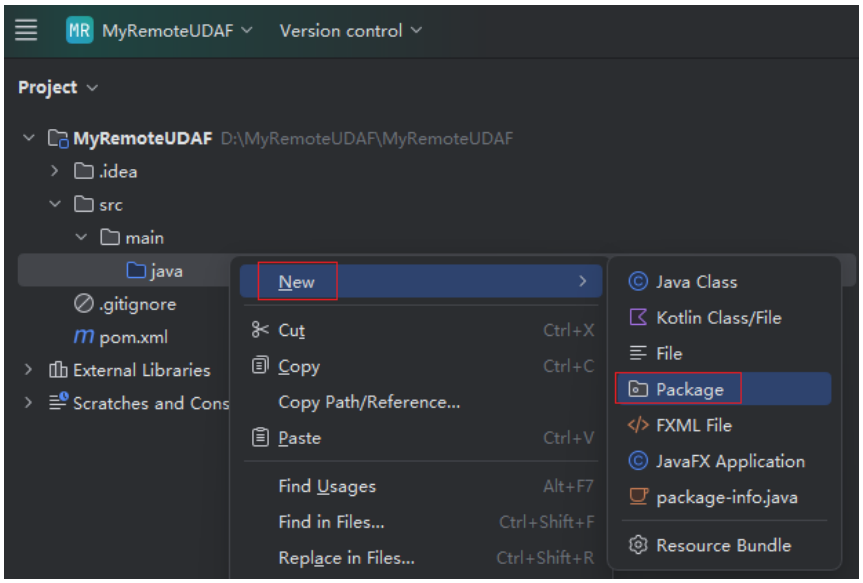
单击“Create”创建项目。

图 3-63 新建 Maven 项目



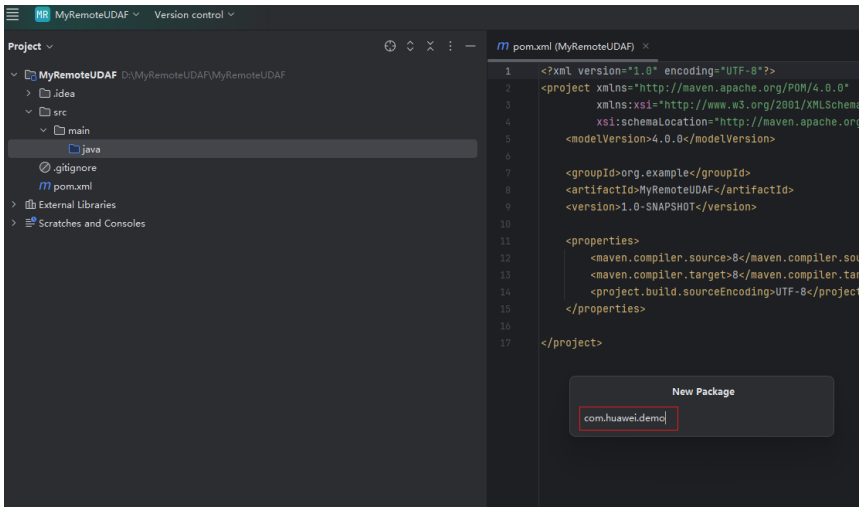
3. 在工程路径的“src > main > java”文件夹单击鼠标右键，选择“New > Package”，新建Package。

图 3-64 新建 Package



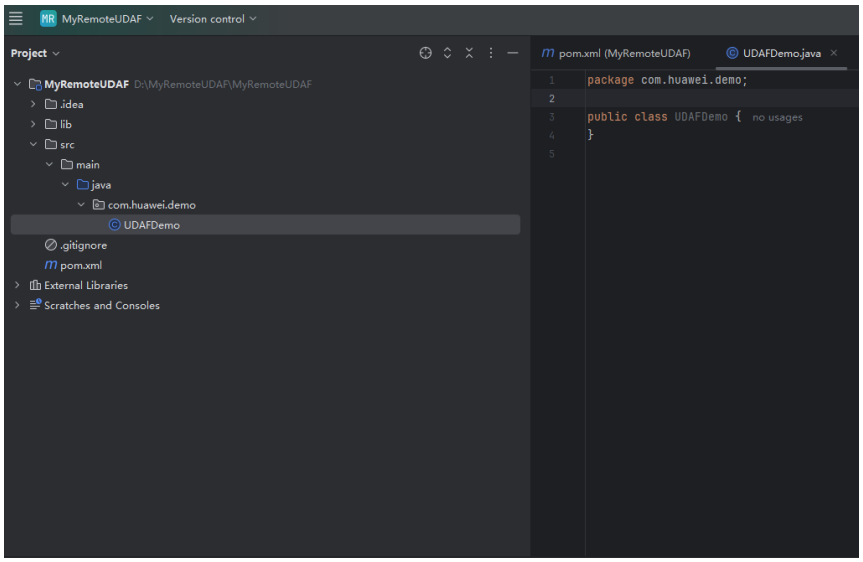
按需定义Package名称，本示例定义为：“com.huawei.demo”，完成后回车。

图 3-65 定义 Package 名称



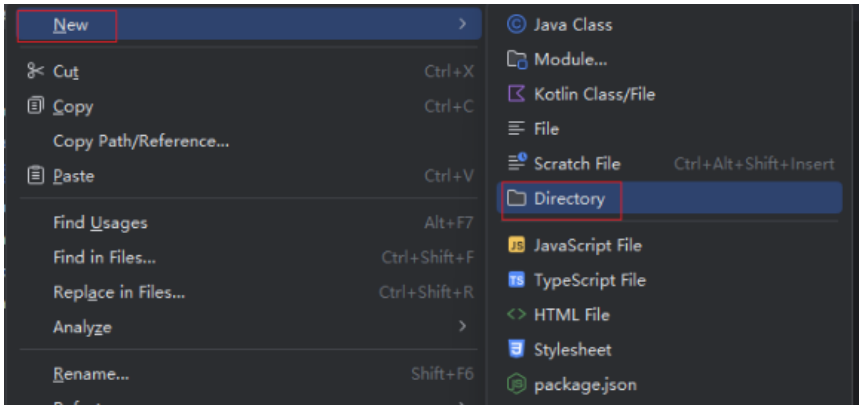
4. 在包路径下新建Java Class文件，本示例定义为：UDAFDemo。

图 3-66 新建 Java Class 文件



5. 添加JAR包存放路径。
   单击“File > New > Directory”，新建文件夹，用于存放示例JAR包。

图 3-67 添加 JAR 包存放路径



文件夹名称本示例定义为：“lib”，回车确定。

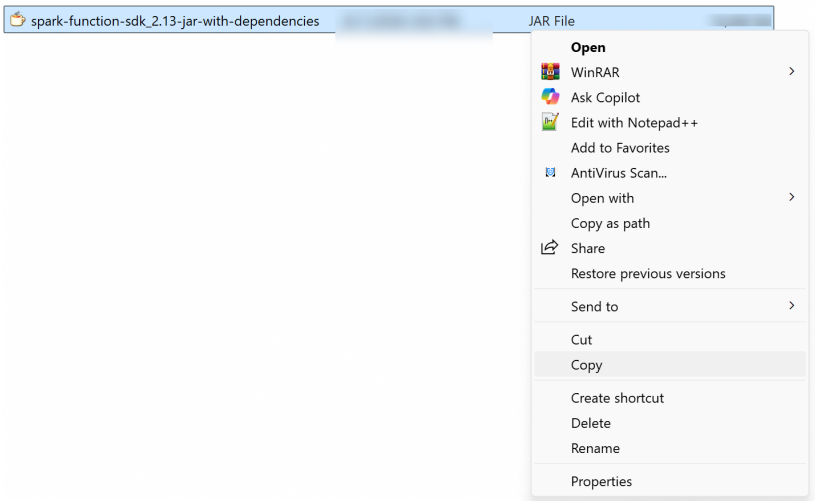
6. 导入JAR包。

   a. 下载JAR包

      从https://repo.huaweicloud.com/repository/maven/huaweicloudsdk/com/huawei/mrs地址下载JAR包。

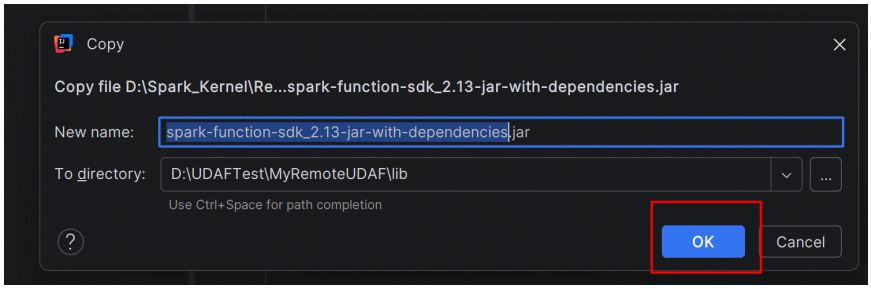
   b. 下载结束后右键单击此JAR包，单击复制。

图 3-68 复制 JAR 包



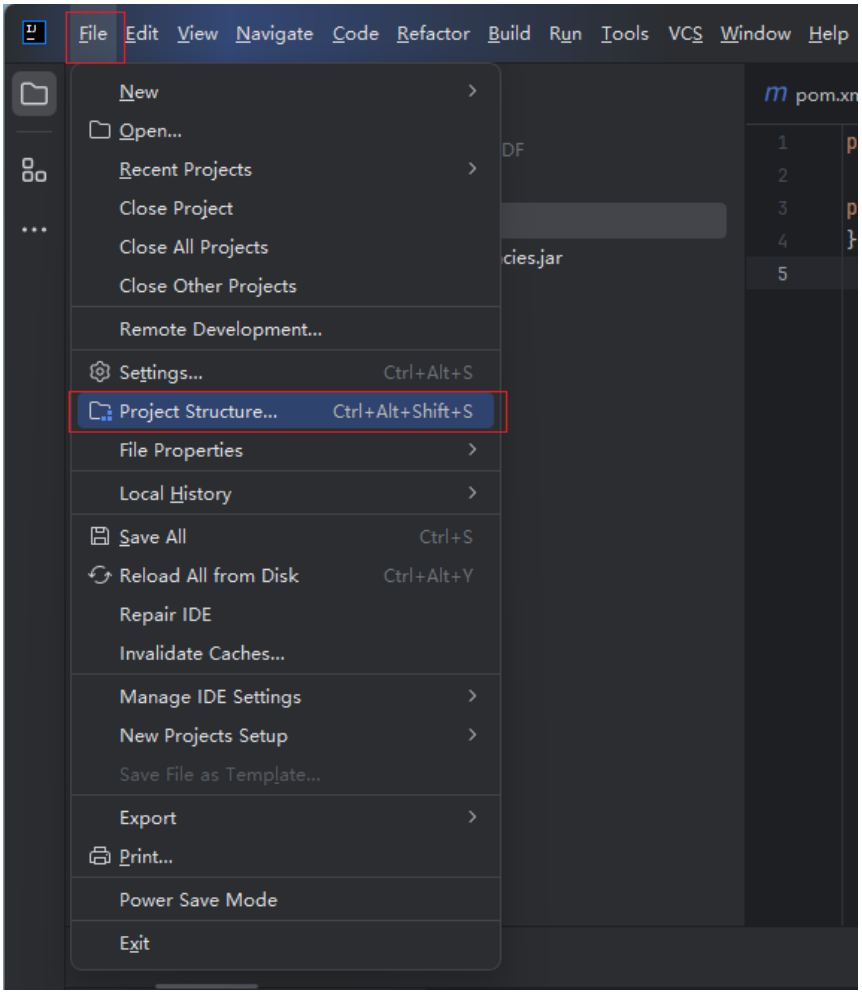
7. 找到项目lib目录，将其粘贴到lib目录下，在弹出的对话框中选择OK。

图 3-69 在 lib 目录下粘贴 JAR 包



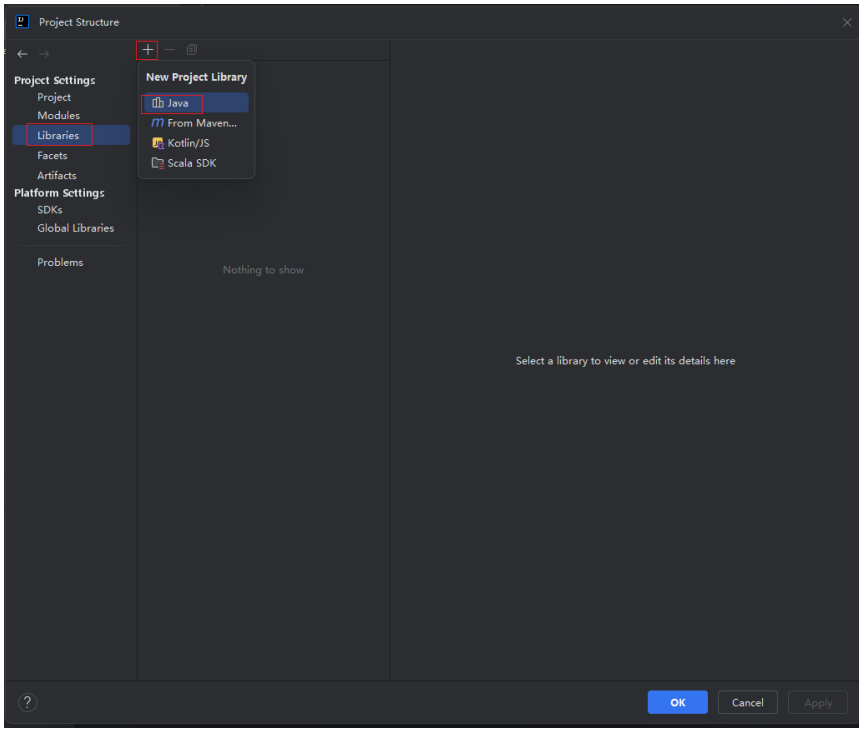
8. 配置“Project Structure”。
   a. 单击“File > Project Structure”。

图 3-70 单击 Project Structure



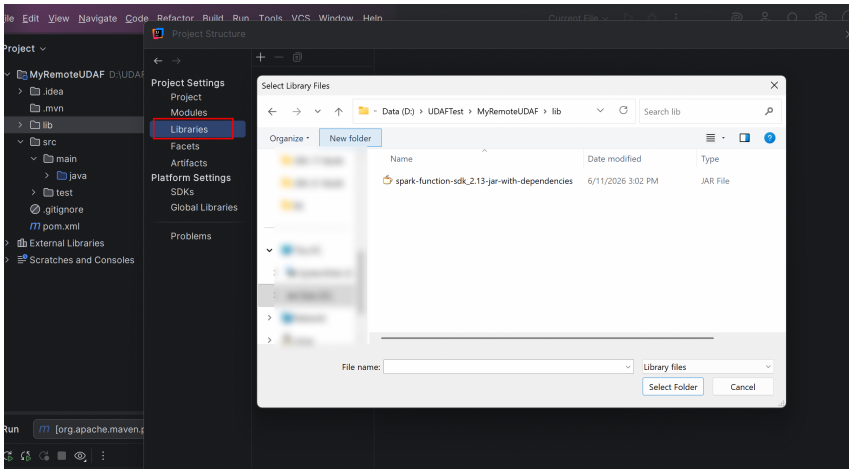
   b. 单击Libraries,然后单击“+”，单击JAVA。

图 3-71 单击 JAVA



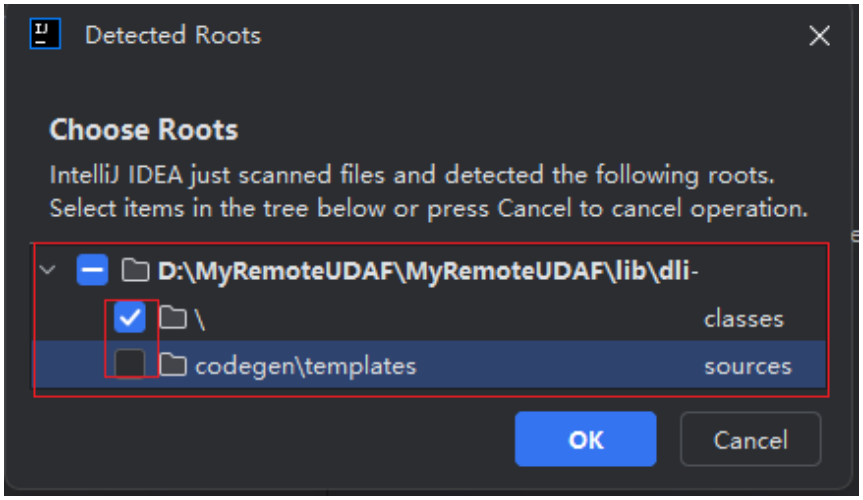
c. 选择lib目录下添加的jar包，单击OK。

图 3-72 选择 lib 目录下添加的 jar 包



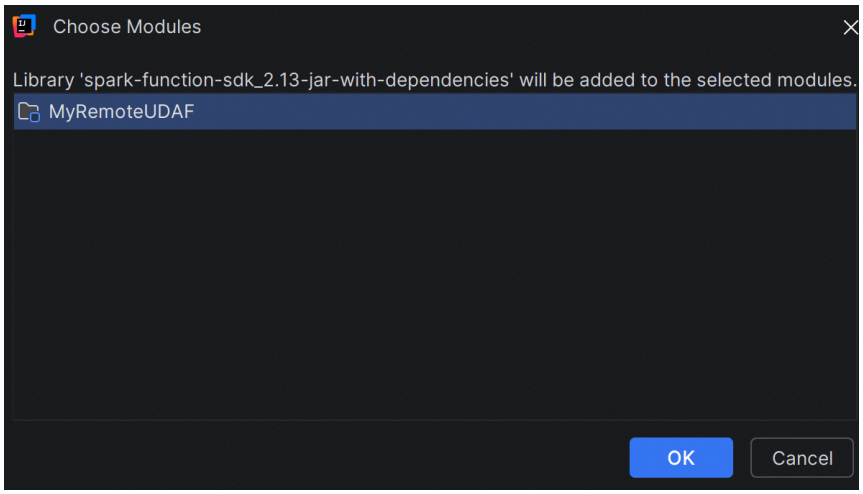
d. 选择根目录，只选取第一项，之后单击OK。

图 3-73 选择根目录



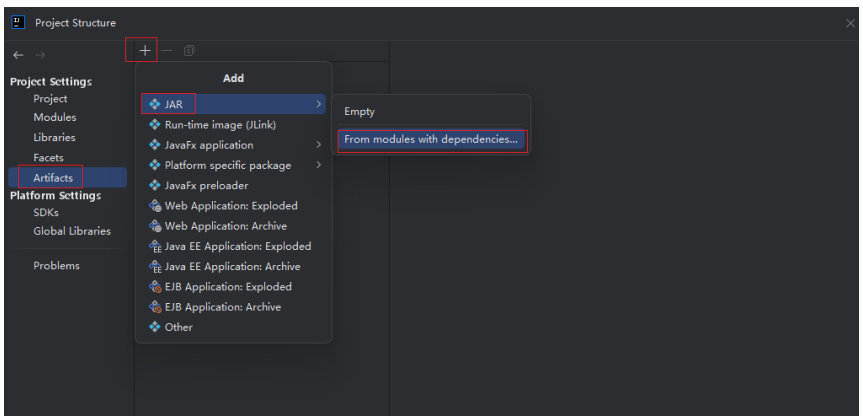
e. 按照提示单击“OK”。

图 3-74 单击“OK”



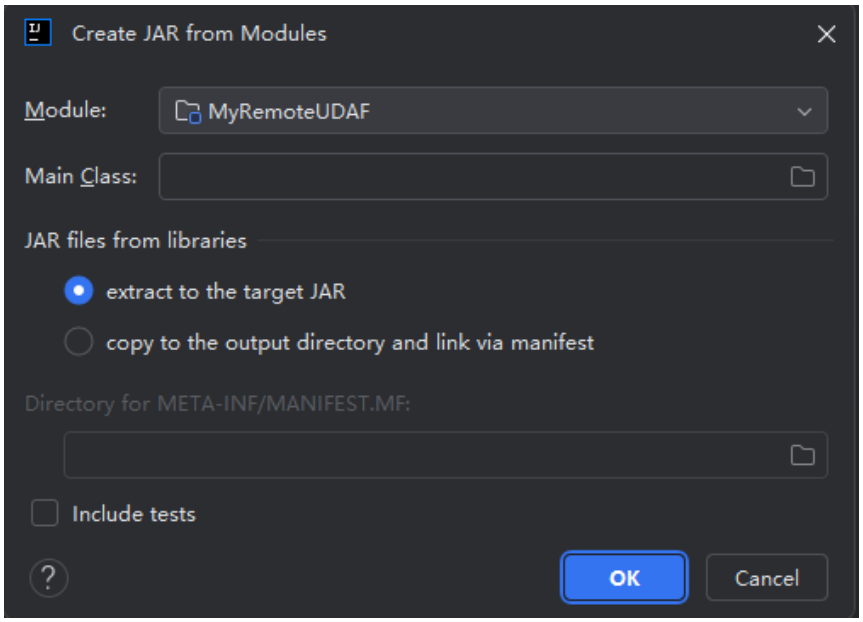
9. 单击Artifacts，并依次单击“+”，JAR，选择From modules with dependencies。

图 3-75 选择 From modules with dependencies



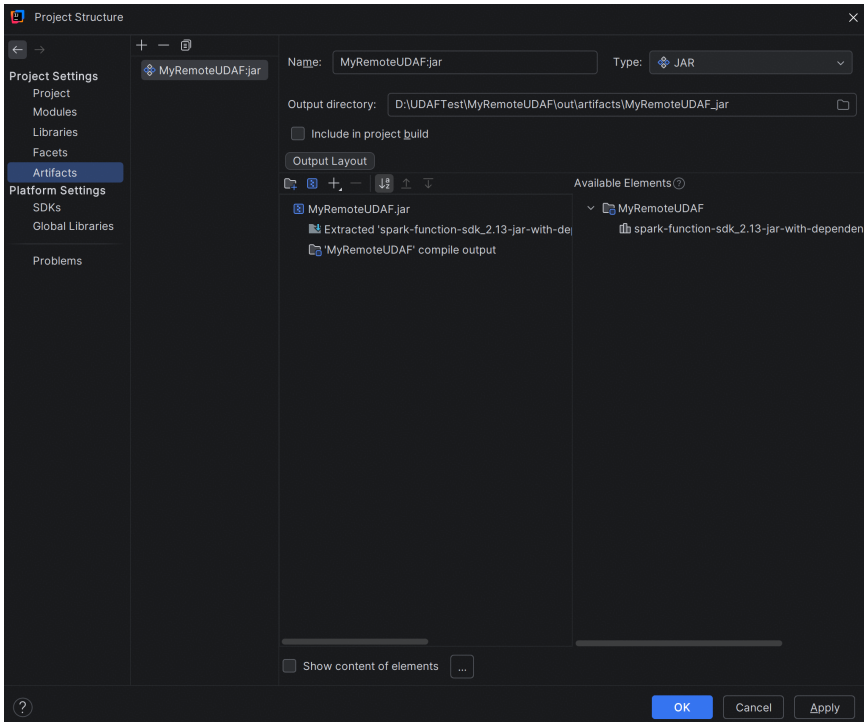
10. 在JAR files from libraries中开启第一项，单击OK。

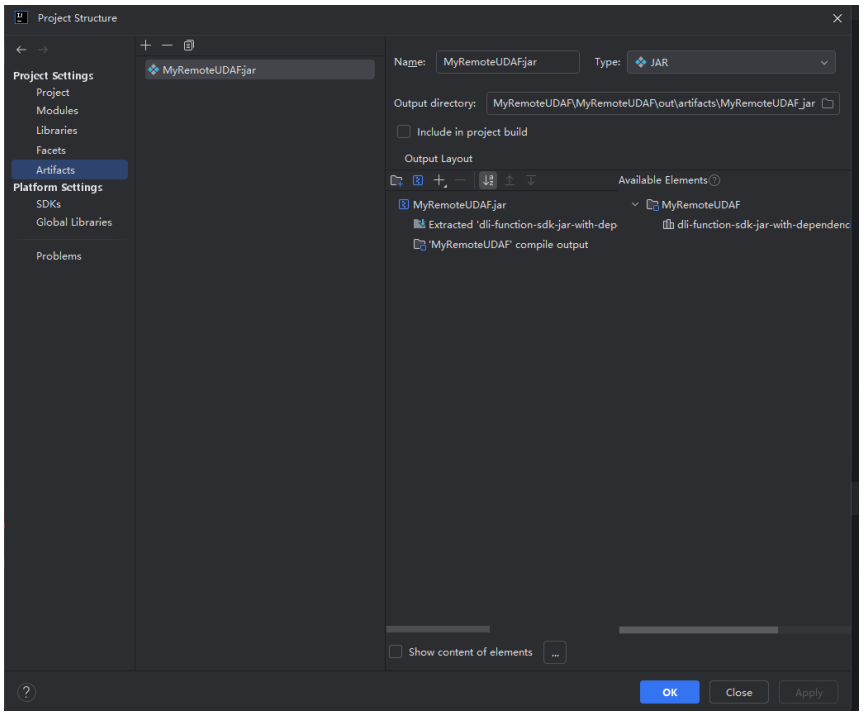
图 3-76 单击“OK”



11. 依次单击Apply和OK。

图 3-77 Artifacts 配置





3.5.3 编写 UDAF 函数代码并导出 Jar 包

步骤 1：编写 UDAF 函数代码

UDAF函数实现，主要注意以下几点：

- 自定义UDAF需继承UDAFHandler类，在类内编写UDAF函数。并实现如下方法：

```
public abstract Writable newBuffer();
public abstract void iterate(Writable buffer, Object[] args) throws FuncException;
public abstract void merge(Writable buffer, Writable partial) throws FuncException;
public abstract Object terminate(Writable buffer) throws FuncException;
```

- 在方法上添加@TypeResolve注解，用于表明函数的入参和出参。

  入参和出参通过'->'分隔。入参和出参类型（ resolveType ）以及对应的SQL和Java数据类型请参考表3-16。

表 3-16 resolveType 类型

SQL type	Java type	resolveType
BooleanType	Boolean.class	BOOLEAN
BooleanType	Boolean.class	BOOLEAN
ByteType	Byte.class	BYTE
ShortType	Short.class	SHORT
IntegerType	Integer.class	INT
LongType	Long.class	LONG
FloatType	Float.class	FLOAT

SQL type	Java type	resolveType
DoubleType	Double.class	DOUBLE
StringType	String.class	STRING
BinaryType	byte[].class	BINARY
DecimalType.Fixed(precision, scale)	BigDecimal.class	Decimal
DateType	LocalDate.class	DATE
TimestampType	ZonedDateTime.class	TIMESTAMP
TimestampNTZType	LocalDateTime.class	TIMESTAMP_NTZ
YearMonthIntervalType	Period.class	INTERVAL_YEAR_MONTH
DayTimeIntervalType	Duration.class	INTERVAL_DAY_TIME
List	List.class	LIST
Map	Map.class	MAP
Struct	Map.class	STRUCT

- 添加@deterministic注解：

  表明函数是确定性的，即在给定相同的输入时，总是返回相同的结果。

  详细UDAF函数实现，可以参考如下样例代码：

  - 功能描述：

    本函数用于计算数据的平均值，设计流程为：“初始化 > 迭代计算 > 合并结果 > 最终计算”。

  - 示例代码

```
package com.huawei.demo;

import com.huawei.spark.function.FuncException;
import com.huawei.spark.function.handlers.UDAFHandler;
import com.huawei.spark.function.requests.Writable;
import com.huawei.spark.function.types.TypeResolve;

import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

// get average
@TypeResolve("int->double")
public class UDAFDemo extends UDAFHandler {
    private static class Buffer implements Writable {
        private double sum = 0.d;

        private int count = 0;

        public Buffer() {
        }
```

```
        @Override
        public void writeExternal(ObjectOutput out) throws IOException {
            out.writeDouble(sum);
            out.writeInt(count);
        }

        @Override
        public void readExternal(ObjectInput in) throws IOException {
            sum = in.readDouble();
            count = in.readInt();
        }
    }

    public Writable newBuffer() {
        return new com.huawei.demo.UDAFDemo.Buffer();
    }

    public void iterate(Writable buffer, Object[] args) throws FuncException {
        com.huawei.demo.UDAFDemo.Buffer buf = (com.huawei.demo.UDAFDemo.Buffer) buffer;
        Integer arg = (Integer) args[0];
        buf.sum += arg;
        buf.count++;
    }

    public void merge(Writable buffer, Writable partial) throws FuncException {
        com.huawei.demo.UDAFDemo.Buffer buf = (com.huawei.demo.UDAFDemo.Buffer) buffer;
        com.huawei.demo.UDAFDemo.Buffer par = (com.huawei.demo.UDAFDemo.Buffer) partial;
        buf.sum += par.sum;
        buf.count += par.count;
    }

    public Object terminate(Writable buffer) throws FuncException {
        com.huawei.demo.UDAFDemo.Buffer buf = (com.huawei.demo.UDAFDemo.Buffer) buffer;
        if (buf.count == 0) {
            return 0;
        }
        return buf.sum / buf.count;
    }
}
```

```
package com.huawei.demo;

import com.huawei.spark.function.FuncException;
import com.huawei.spark.function.handlers.UDAFHandler;
import com.huawei.spark.function.requests.Writable;
import com.huawei.spark.function.types.TypeResolve;

import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

@TypeResolve("long->double")
public class UDAFDemo extends UDAFHandler {
    private static class Buffer implements Writable {

        private double sum = 0;

        private int count = 0;

        public Buffer() {
        }

        @Override
        public void writeExternal(ObjectOutput out) throws IOException {
            out.writeDouble(sum);
            out.writeInt(count);
        }

        @Override
        public void readExternal(ObjectInput in) throws IOException {
```

```
            sum = in.readDouble();
            count = in.readInt();
        }
    }

    public Writable newBuffer() {
        return new Buffer();
    }

    public void iterate(Writable buffer, Object[] args) throws FuncException {
        Buffer buf = (Buffer) buffer;
        if(null != args[0]) {
            Long arg = (Long) args[0];
            buf.sum += arg;
            buf.count++;
        }
    }

    public void merge(Writable buffer, Writable partial) throws FuncException {
        Buffer buf = (Buffer) buffer;
        Buffer par = (Buffer) partial;
        buf.sum += par.sum;
        buf.count += par.count;
    }

    public Object terminate(Writable buffer) throws FuncException {
        Buffer buf = (Buffer) buffer;
        if (buf.count == 0) {
            return 0;
        }
        return buf.sum / buf.count;
    }
}
```

- 详细说明：
  - **newBuffer()初始化缓冲区**

    Buffer 类实现了 Writable 接口，用于存储聚合过程中的中间结果，主要作用是暂存计算过程中的中间值，支持序列化和反序列化。

    包含两个核心变量：

    - sum：累加总和。
    - count：累加计数，记录参与计算的元素个数。

    实现了两个序列化方法：

    - writeExternal：将 sum 和 count 写入输出流。
    - readExternal：从输入流读取 sum 和 count。
- **iterate()：迭代处理输入数据**

  作用：处理每一条输入数据，更新缓冲区的中间状态。

  参数说明：
  - buffer：当前的缓冲区。
  - args：输入参数数组，此处 args[0] 是待计算的整数。

  逻辑：将输入的整数累加到 sum，同时将计数 count 加 1。
- **merge()：合并分布式计算的中间结果**

  作用：在分布式计算场景下，合并多个子任务的中间结果。

  参数说明：

  - buffer：主缓冲区，用于最终保存合并后的结果。
  - partial：其他子任务的缓冲区，用于待合并的部分结果。

  逻辑：将子任务的 sum 和 count 分别累加到主缓冲区中。

- **terminate()：计算最终结果**

  作用：在所有数据处理完成后，根据缓冲区的中间状态计算最终的平均值。

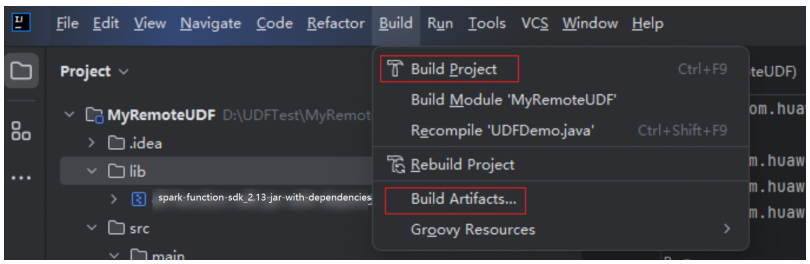
  逻辑：用总累加和 sum 除以总计数 count，返回平均值；若没有数据（count=0），返回 0 避免除零异常。

步骤 2：导出 Jar 包

编写调试完成代码后，通过IntelliJ IDEA工具编译代码并导出Jar包。

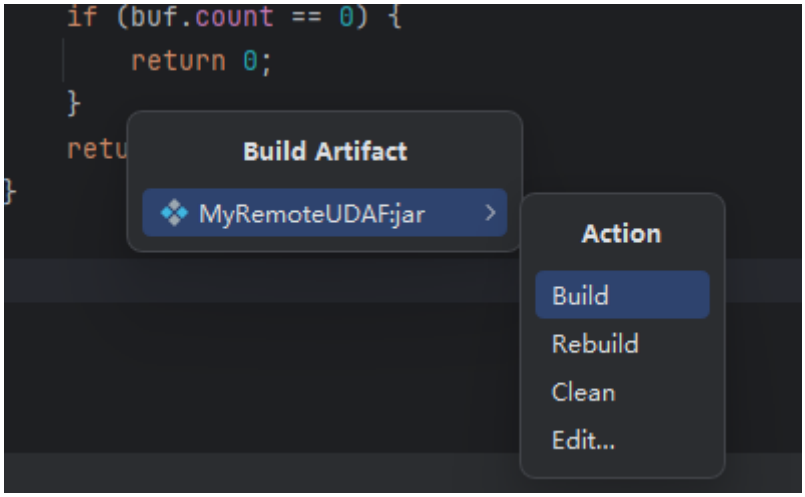
1. 单击“Build”，参考下图分别单击“Build Project ”、“Build Artifacts...”分别对代码进行编译和打包。

   图 3-78 编译和打包

2. 在弹出的对话框中单击Build。

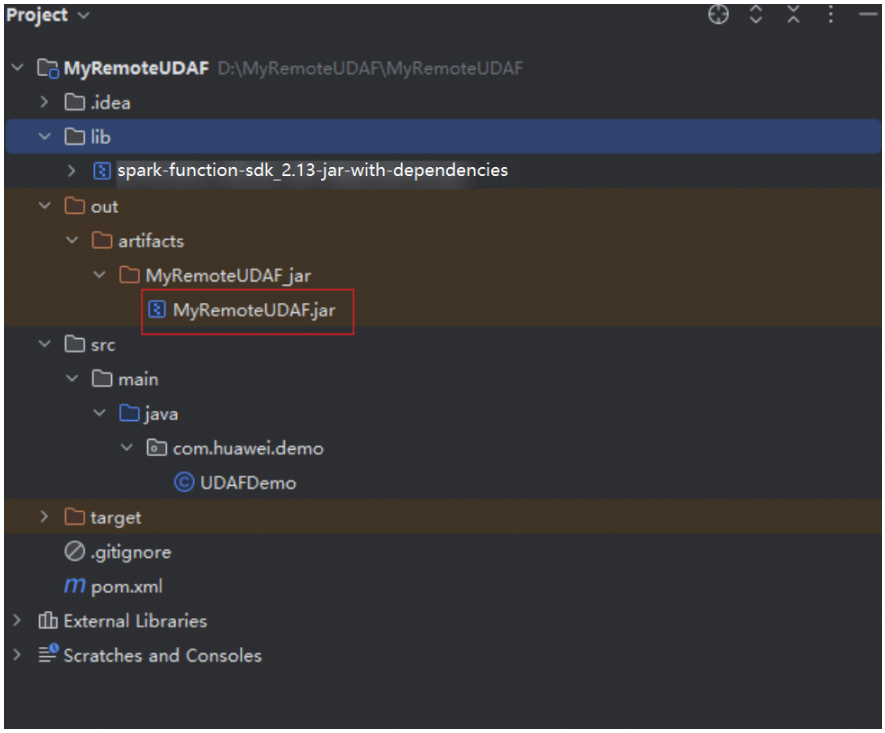
   图 3-79 编译和打包

3. 完成上述步骤之后，可以在项目目录当中看到out文件夹，展开out目录。

   其中的MyRemoteUDAF.jar就是之后要上传到函数工作流FunctionGraph当中的jar包。

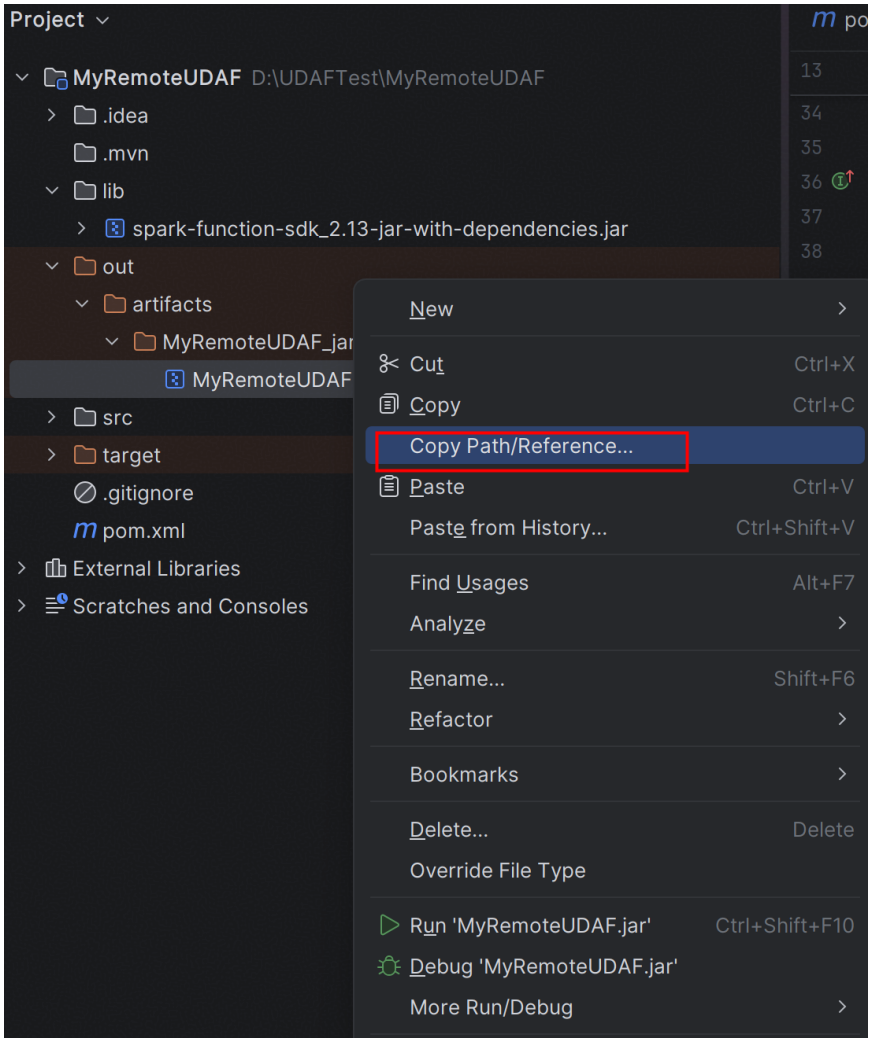
图 3-80 查看 MyRemoteUDAF.jar



4. 右键单击MyRemoteUDAF.jar，选择Copy Path/Reference，直接复制文件所在目录，方便之后上传JAR包使用。

   本示例将会生成到：“D:\UDAFTest\MyRemoteUDAF\out\artifacts\MyRemoteUDAF_jar\”目录下，文件名为“MyRemoteUDAF.jar”。

图 3-81 复制文件所在目录



3.5.4 创建 FunctionGraph 函数

了解**FunctionGraph业务使用流程**。

步骤 1：创建函数

1. 登录函数工作流FunctionGraph管理控制台。
2. 在左侧列表中单击“函数 > 函数列表”，单击“创建函数”

图 3-82 创建函数

3. 单击“创建空白函数”，并配置函数的基本信息。
详细参数说明请参考**函数属性**。

- 函数类型：本例为事件函数。
- 运行时环境：Java11。
- 委托：当函数工作流服务需与其他云服务协同工作时，需要创建云服务委托，授权FunctionGraph使用这些云服务，并在FunctionGraph中为函数配置委托权限，以实现与其他云服务的协同工作。

  FunctionGraph单次调用数据量有**约束与限制**。函数如果超过4MB，建议配置访问OBS的委托权限。具体操作请参考**相关操作：配置FunctionGraph访问其他云服务的委托**。

图 3-83 创建函数

步骤 2：函数设置

1. **设置函数执行入口**

   a. 函数创建完成后，在列表单击函数名称进入函数详情页面。

   b. 选择“设置 > 常规设置”，在“函数执行入口”中设置UDAF的执行入口。

   UDAF的执行入口格式为[包名].[类名].[执行函数名]，

   - [包名].[类名]为编写的UDAF函数所在类，
   - 执行函数名固定为handleRequest。

   本示例设置为：com.huawei.demo.UDAFDemo.handleRequest，

   c. 设置完成之后单击保存按钮。

2. **设置执行超时时间和内存**

   a. 函数创建完成后，在列表单击函数名称进入函数详情页面。

   b. 选择“设置 > 常规设置”，在“函数执行入口”中设置执行时间和内存。

   根据UDAF复杂情况配置执行**超时时间和内存大小**。内存大小与**计费**关联。

   本示例选择执行时间为15秒，内存为1024MB。

   c. 设置完成之后单击保存按钮。

图 3-84 常规设置

3. **设置类隔离信息**

   a. 函数创建完成后，在列表单击函数名称进入函数详情页面。

   b. 选择“设置 > 高级设置”，配置“类隔离选项”。

      类隔离开启后将使用单独的类加载器加载上传的代码和依赖，解决依赖和运行时的冲突，如果无依赖冲突问题无需打开。

   本示例中关闭类隔离选项，设置完成后单击保存按钮。

图 3-85 设置类隔离

步骤 3：上传 Jar 包

1. 在FunctionGraph管理控制台，函数列表中单击函数名称进入函数详情页面。
2. 单击"代码"，在右侧上传代码下拉框当中选择JAR文件，将之前导出的JAR包上传
3. 上传成功之后，可以在函数详情页面复制URN（Uniform Resource Name）。URN是函数的唯一标识函数，在下一步调用函数时需要用到。

图 3-86 复制 URN

相关操作：配置 FunctionGraph 访问其他云服务的委托

1. 登录管理控制台。
2. 单击右上方登录的用户名，在下拉列表中选择“统一身份认证”。
3. 在左侧导航栏中，单击“委托”。
4. 在“委托”页面，单击“创建委托”。
5. 在“创建委托”页面，设置如下参数：
   - 委托名称：按需填写。
   - 委托类型：选择“云服务”。
   - 云服务：（“委托类型”选择“云服务”时出现此参数项。）在下拉列表中选择“FunctionGraph"。
   - 持续时间：选择“永久”。
   - 描述：非必选，可以填写“拥有OBS OperateAccess权限的委托”。

图 3-87 创建委托

6. 配置完委托的基本信息后，单击“立即授权”。
7. 进入“选择策略”页面，在右方搜索框中搜索OBS Administrator权限并勾选。勾选完成后单击“下一步”。
8. 单击“下一步”，选择委托的授权范围。

   了解更多授权操作说明请参考**创建用户组并授权**。

   - 所有资源：授权后，IAM用户可以根据权限使用账号中所有资源，包括企业项目、区域项目和全局服务资源。
   - 全局服务资源：全局服务部署时不区分区域，访问全局级服务，不需要切换区域，全局服务不支持基于区域项目授权。如对象存储服务（OBS）、内容分发网络（CDN）等。授权后，用户根据权限使用全局服务的资源。
   - 指定区域项目资源：授权后，IAM用户根据权限使用所选区域项目中的资源，未选择的区域项目中的资源，该IAM用户将无权访问。
   - 指定企业项目资源：授权后，IAM用户根据权限使用所选企业项目中的资源。如企业项目A包含资源B，资源B部署在北京四和上海二，IAM用户所在用户组关联企业项目A后，北京四和上海二的资源B用户都可访问，不在企业项目A内的其他资源，该IAM用户将无权访问。

   本例选择的是OBS策略，因此选择全局服务资源。
9. 单击“确定”，完成授权。
10. 授权后需等待15-30分钟才可生效。

    授权生效后返回函数基本信息界面，选择对应的委托，权限策略当中会显示已授权的策略。

3.5.5 在 Spark SQL 作业中使用 Remote UDAF

3.5.5.1 SQL 创建函数

功能描述

创建使用Remote UDAF自定义函数应用于Spark作业开发当中。

语法格式

```
CREATE FUNCTION [db_name.]function_name AS class_name USING FUNCTIONGRAPH 'urn'
```

或

```
CREATE OR REPLACE FUNCTION [db_name.]function_name AS class_name USING FUNCTIONGRAPH 'urn'
```

注意事项

无

关键字

urn：在FunctionGraph函数列表页面，选择您需要复制URN的函数，在操作列中单击复制URN。

示例

创建本示例函数integer_add，执行的指令如下：

```
CREATE FUNCTION UDAFDemo AS 'ExampleUDAFHandler' USING FUNCTIONGRAPH 'urn:fss:cn-north-7:330e068af1334c9782f4226acc00a2e2:function:default:test:latest'
```

选择对应的数据库与队列，输入SQL指令后单击执行按钮。

3.5.5.2 SQL 调用函数

功能描述

调用指定函数。

语法格式

```
SELECT function_name(column_name) FROM table_name;
```

注意事项

无

关键字

function_name：即在FunctionGraph创建的自定义函数名称。

示例

调用本示例函数，返回一个平均数

```
SELECT remote_udaf(real_stock_rate)  FROM dw_ad_estimate_real_stock_rate limit 1000;
```

3.5.5.3 SQL 删除函数

功能描述

删除指定函数。

语法格式

```
DROP [TEMPORARY] FUNCTION [IF EXISTS] [db_name.] function_name;
```

注意事项

无

关键字

- TEMPORARY：所删除的函数是否为临时函数。
- IF EXISTS：所删除的函数不存在时使用，可避免系统报错。

示例

```
DROP FUNCTION remote_udaf;
```

3.5.5.4 显示所有函数

功能描述

查看当前工程下所有的函数。

语法格式

```
SHOW [USER|SYSTEM|ALL] FUNCTIONS ([LIKE] regex | [db_name.] function_name);
```

注意事项

无

关键字

LIKE：此限定符仅为兼容性而使用，没有任何实际作用。

示例

```
SHOW FUNCTIONS;
```

3.5.5.5 显示函数详情

功能描述

查看指定函数的相关信息。

语法格式

```
DESCRIBE FUNCTION [EXTENDED] [db_name.] function_name;
```

注意事项

无

关键字

EXTENDED：显示扩展使用信息。

示例

```
DESCRIBE FUNCTION remote_udaf;
```

3.6 在 Spark SQL 作业中使用 Remote UDTF

3.6.1 概述

功能描述

Remote UDTF（Remote User-Defined Table-Generating Function）为用户自定义表值函数，适用于单进多出业务场景。即其输入与输出是一对多的关系，读入一行数据，输出多个值可视为一张表。

本节内容以FunctionGraph作为Remote UDTF 的执行载体，介绍开发Remote UDTF到计算引擎调用的操作步骤。

方案优势

Remote UDTF的优势在于风险隔离、资源解耦与成本优化。

- 风险隔离：当关键生产环境需要执行UDTF时，虽然技术上支持，但存在执行风险。

  通过Remote UDTF将计算逻辑卸载到函数工作流FunctionGraph，既能保持业务功能，又能通过无服务器架构实现风险隔离。
- 资源解耦：对于大模型推理等计算密集型任务，为避免在Spark等集群中长期占用昂贵资源，兼顾功能需求、稳定性和成本效益，将计算密集型操作剥离至FunctionGraph执行
- 成本优化：Remote UDTF将default队列还原为纯数据通道，消除UDTF引发的风险，使其能为用户提供更经济，更安全的核心数据流服务。

约束与限制

Remote UDTF只对单行数据产生作用，适用于单进单出的场景。

环境准备

使用FunctionGraph开发调试Remote UDTF时，您需要先安装IDEA并配置Maven，下载依赖JAR包并配置FunctionGraph服务，做好Remote UDTF开发前准备工作。

表 3-17 UDTF 开发环境

准备项	说明
操作系统	Windows系统，支持Windows7以上版本。
安装JDK	JDK使用1.8版本。
安装和配置IntelliJ IDEA	IntelliJ IDEA为进行应用开发的工具，版本要求使用2019.1或其他兼容版本。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。
下载依赖JAR包	依赖引入下载Jar包：spark-function-sdk_2.13-jar-with-dependencies.jar 获取方式：从https://repo.huaweicloud.com/repository/maven/huaweicloudsdk/地址下载。

UDTF 与 Remote UDTF 的对比

- UDTF：适合对性能要求高且熟悉查询引擎开发语言的场景。执行效率高，但开发灵活性和安全性相对较低。
- Remote UDTF：适合需要调用外部服务、使用多种语言开发或对开发语言灵活性要求高的场景。具有较高的安全性和开发灵活性。

表 3-18 UDTF 与 Remote UDTF 的对比

特性	UDTF	Remote UDTF
执行方式	在查询引擎本地执行	通过调用函数托管服务执行
安全性	若UDTF代码不稳定，可能影响查询引擎进程	执行失败仅影响UDTF自身，不会导致整体作业进程。
性能	执行效率高	但在向量化和批处理模式下，性能与本地UDTF相当
开发灵活性	与查询引擎代码耦合度高，开发限制较多	灵活，可调用任意其他服务或程序库，满足多样业务需求
开发成本	需熟悉查询引擎的开发环境和语言，开发成本较高	一次性开发投入，开发成本相对较低
适用场景	对性能要求极高的场景，且开发团队熟悉查询引擎的开发语言	对开发语言灵活性要求高、需要调用外部服务或程序库的场景

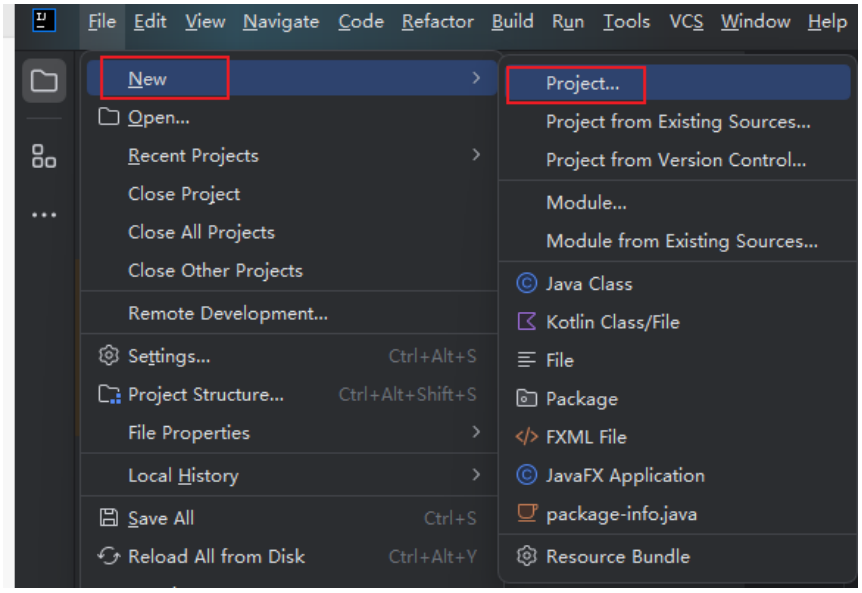
3.6.2 新建 Maven 工程，配置 Project structure

新建 Maven 工程，配置 Project structure

本例使用IntelliJ IDEA 2024.3.2.2工具操作演示

1. 打开IntelliJ IDEA，选择“File > New > Project”。

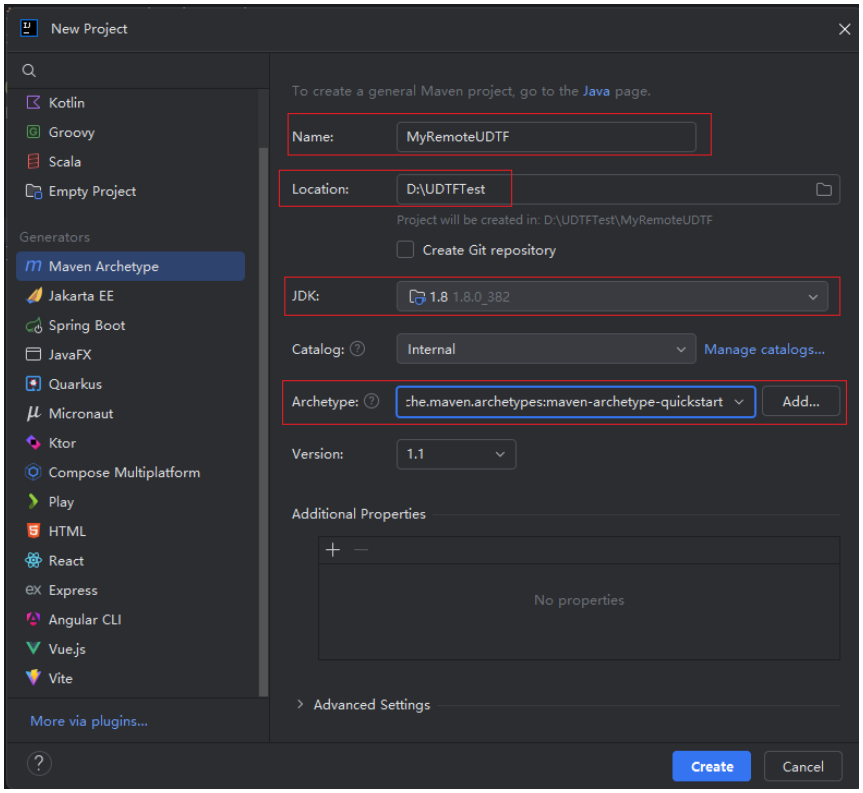
图 3-88 新建 Project



2. 选择Maven Archetype，配置以下信息：
   - Name：自定义项目名称。本例设置项目名称为MyRemoteUDTF。
   - Location：选择项目存储路径。本例项目保存路径为D:\UDTFTest。
   - JDK选择21版本的JDK。
   - Archetype选择‘org.apache.maven.archetypes:maven-archetype-quickstart’。

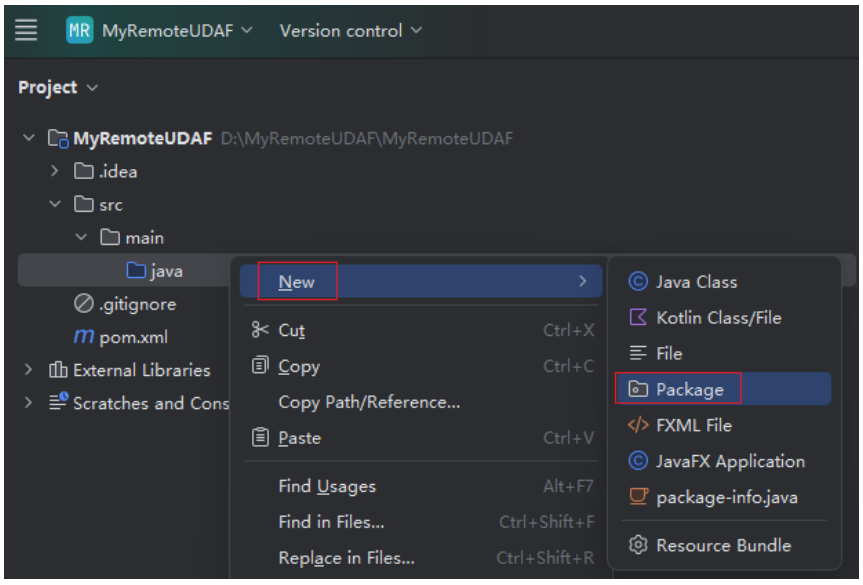
   单击“Create”创建项目。

图 3-89 新建 Maven 项目



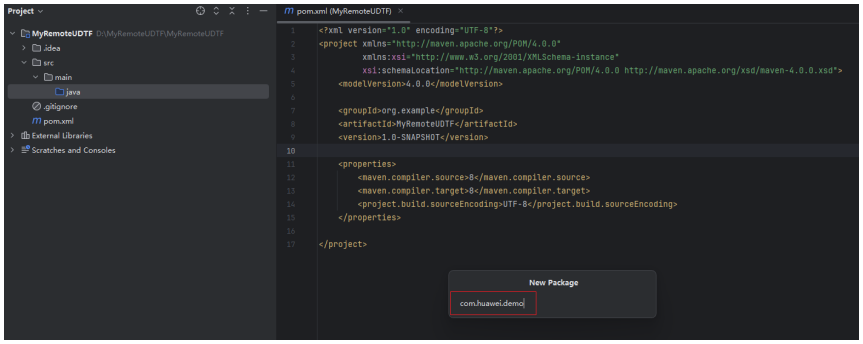
3. 在工程路径的“src > main > java”文件夹单击鼠标右键，选择“New > Package”，新建Package。

图 3-90 新建 Package



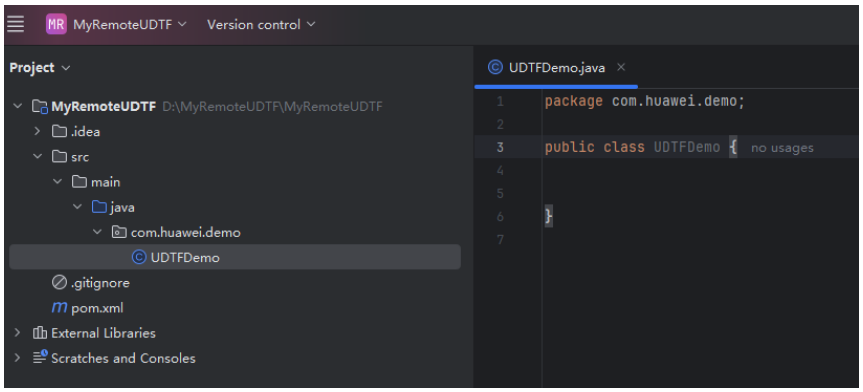
按需定义Package名称，本示例定义为：“com.huawei.demo”，完成后回车。

图 3-91 定义 Package 名称



4. 在包路径下新建Java Class文件，本示例定义为：UDTFDemo。

图 3-92 新建 Java Class 文件

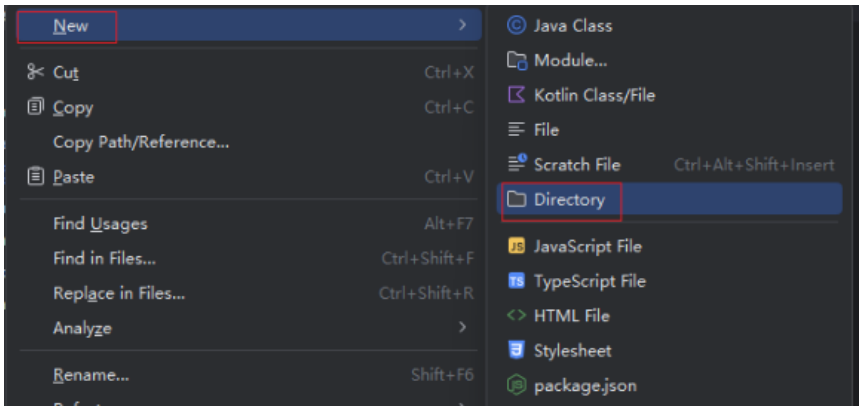


5. 添加JAR包存放路径。

   单击“File > New > Directory”，新建文件夹，用于存放示例JAR包。

   文件夹名称本示例定义为：“lib”，回车确定。

图 3-93 添加 JAR 包存放路径



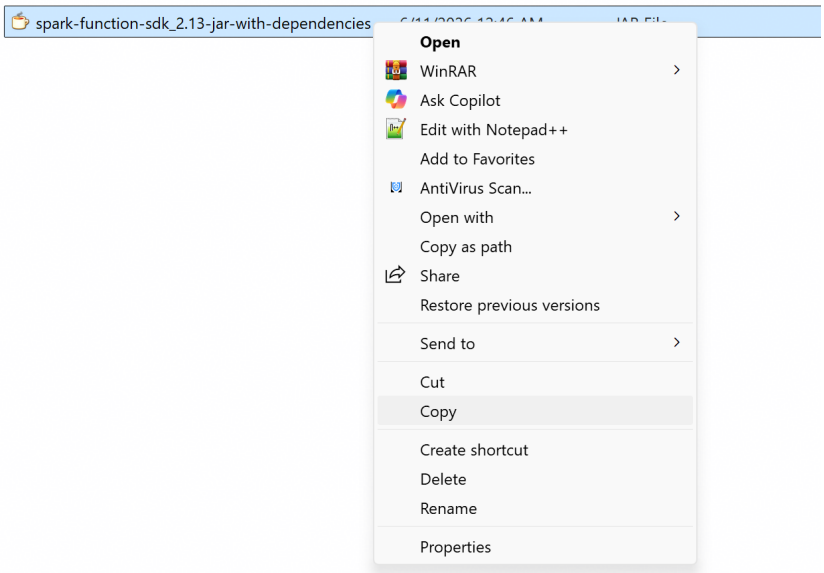
6. 导入JAR包。

   a. 下载JAR包

      从https://repo.huaweicloud.com/repository/maven/huaweicloudsdk/com/huawei/mrs地址下载JAR包。

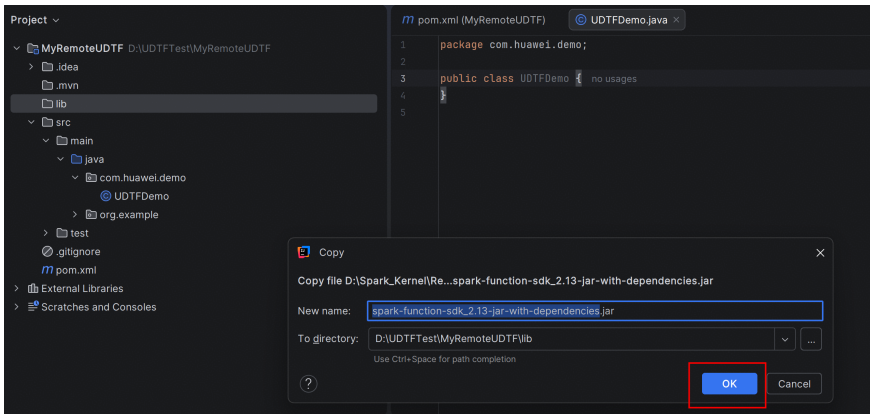
   b. 下载结束后右键单击此JAR包，单击复制。

图 3-94 复制 JAR 包



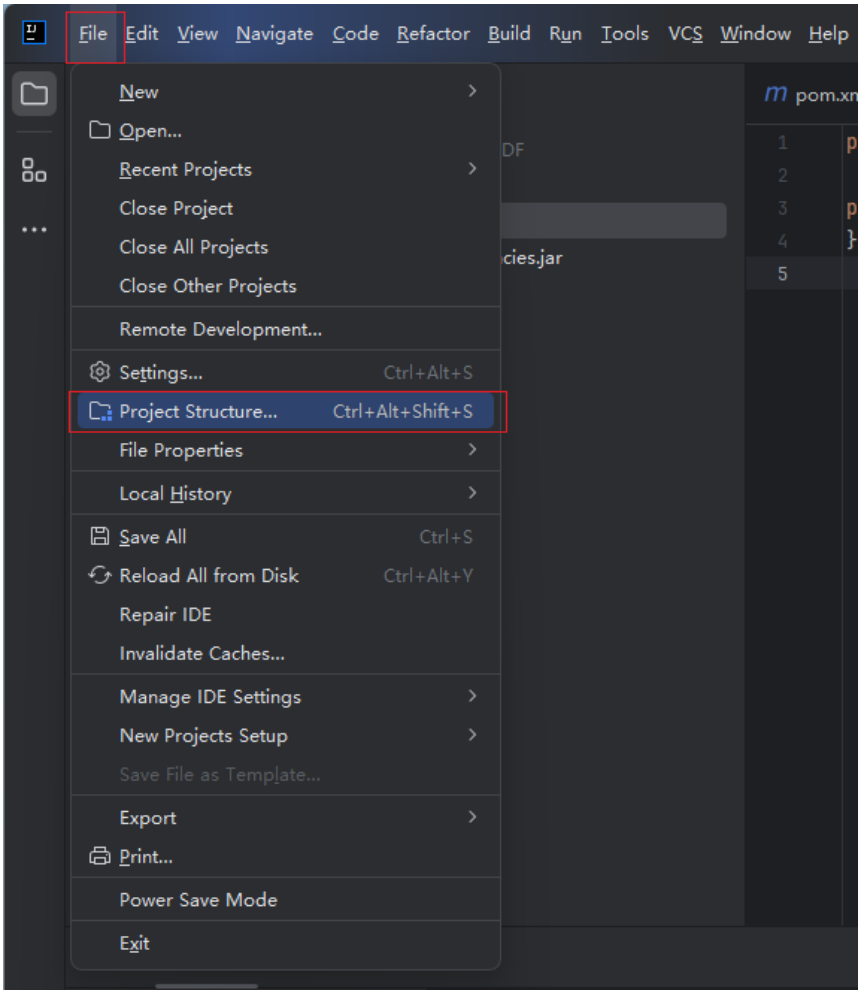
7. 找到项目lib目录，将其粘贴到lib目录下，在弹出的对话框中选择OK。

图 3-95 在 lib 目录下粘贴 JAR 包



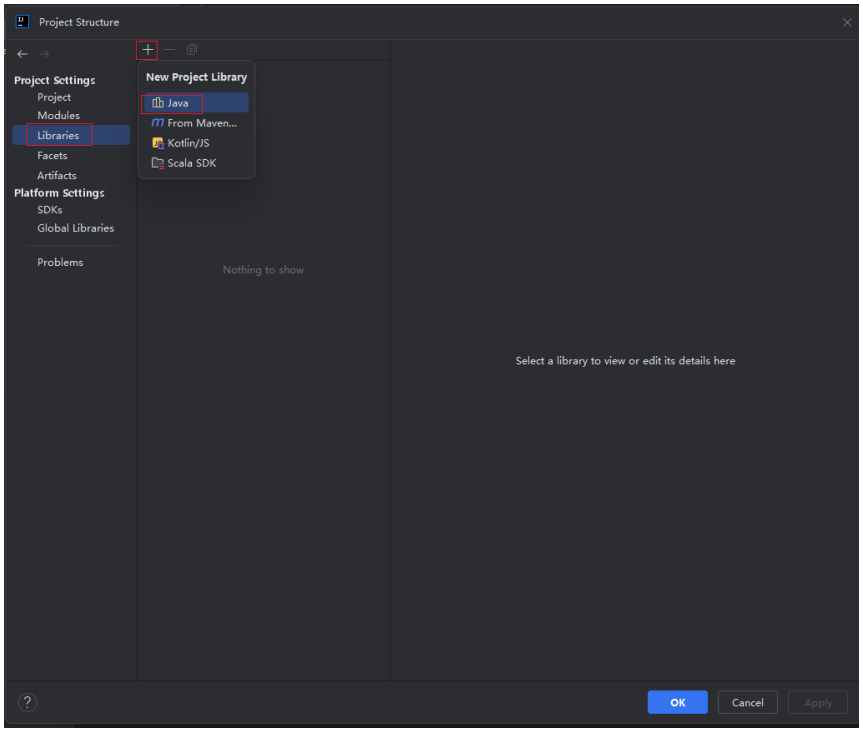
8. 配置“Project Structure”。
   a. 单击“File > Project Structure”。

图 3-96 单击 Project Structure



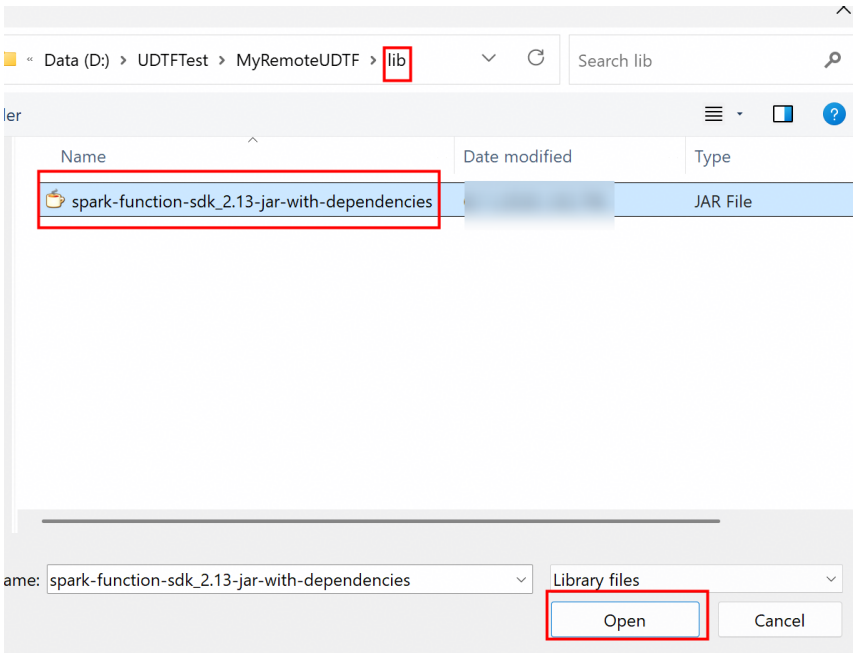
b. 单击Libraries，然后单击“+”，单击JAVA。

图 3-97 单击 JAVA



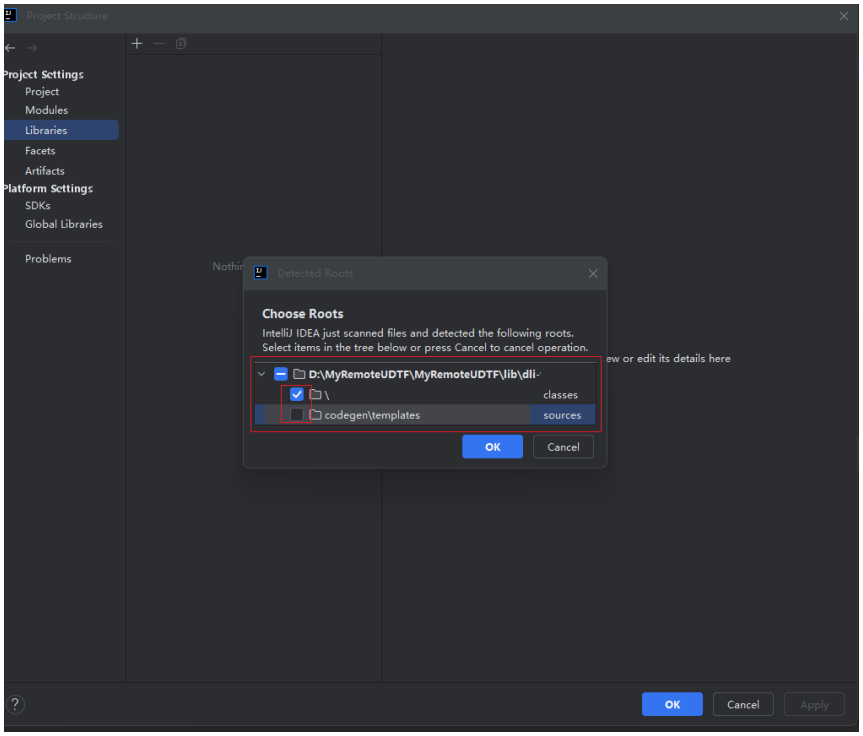
c. 选择lib目录下添加的jar包，单击OK。

图 3-98 选择 lib 目录下添加的 jar 包



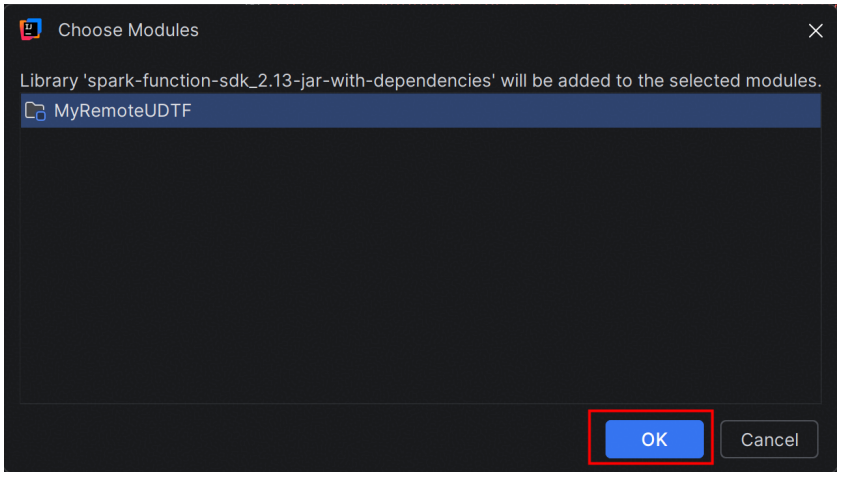
d. 选择根目录，只选取第一项，之后单击OK。

图 3-99 选择根目录



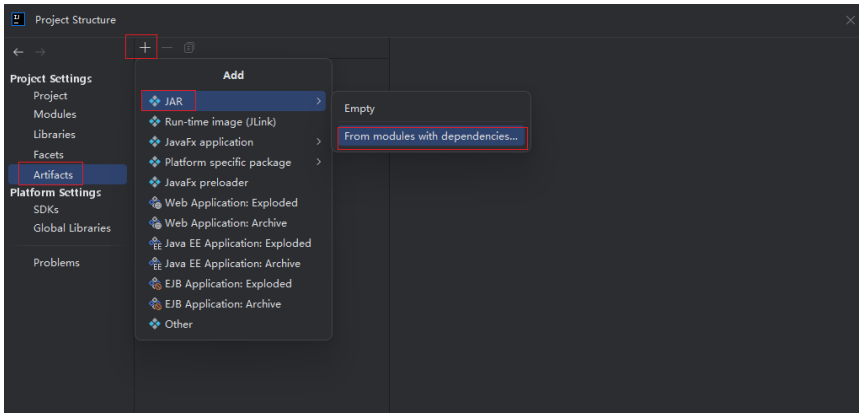
e. 按照提示单击“OK”。

图 3-100 单击“OK”



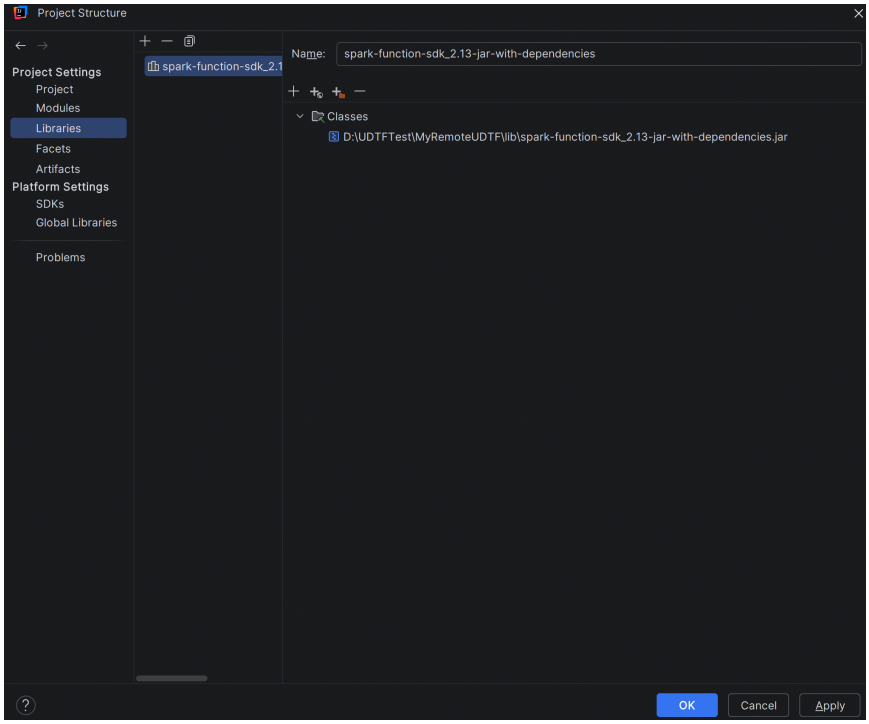
9. 单击Artifacts，并依次单击“+”，JAR，选择From modules with dependencies。

图 3-101 选择 From modules with dependencies



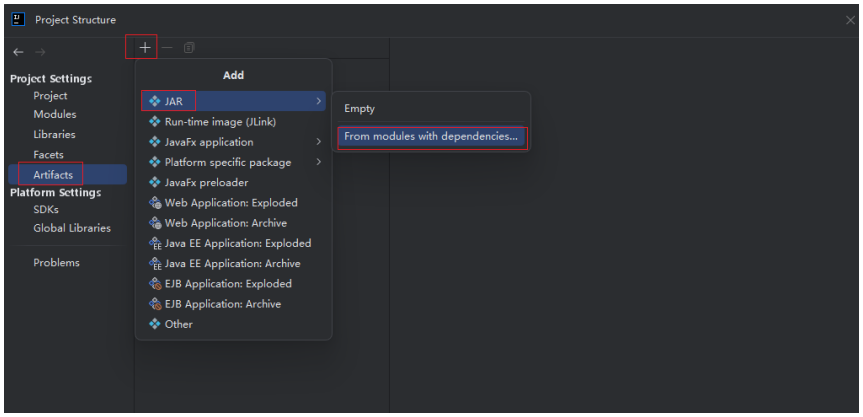
添加完成后的界面：

图 3-102 添加完成后的界面



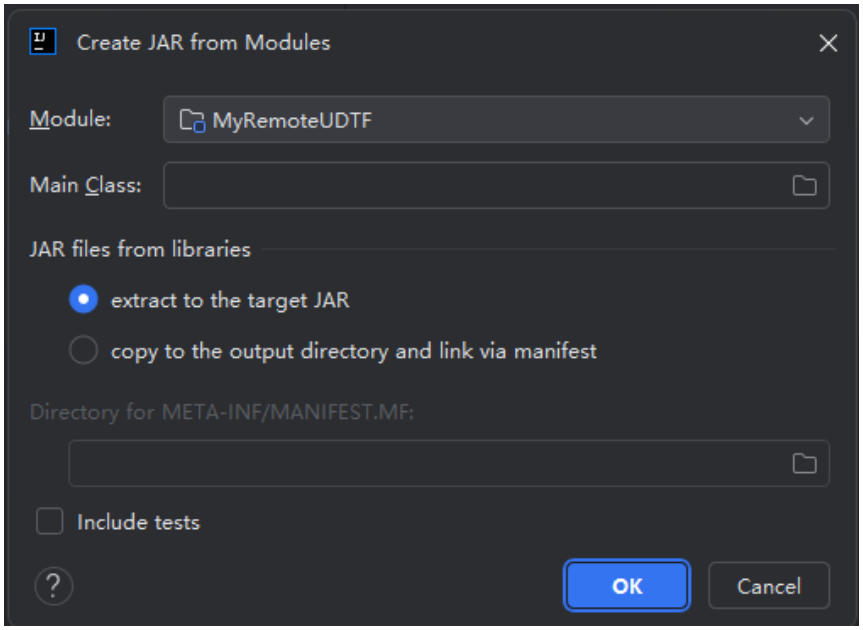
10. 单击Artifacts，并依次单击“+”，JAR，选择From modules with dependencies。

图 3-103 单击 Artifacts



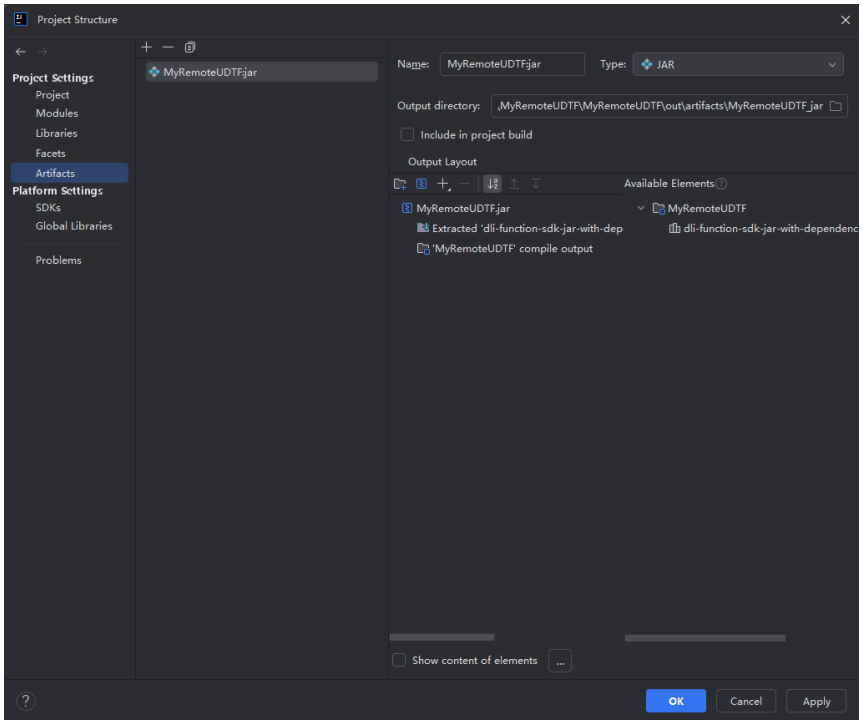
11. 在JAR files from libraries中开启第一项，单击OK。

图 3-104 单击“OK”



12. 依次单击Apply和OK。

图 3-105 Artifacts 配置



3.6.3 编写 UDTF 函数代码并导出 Jar 包

步骤 1：编写 UDTF 函数代码

UDTF函数实现，主要注意以下几点：

- 自定义UDTF需要继承UDTFHandler。
- 在方法上添加@TypeResolve注解，用于表明函数的入参和出参。

  入参和出参通过'->'分隔。入参和出参类型（ resolveType ）以及对应的SQL和Java数据类型请参考**表3-19**。

表 3-19 resolveType 类型

SQL type	Java type	resolveType
BooleanType	Boolean.class	BOOLEAN
ByteType	Byte.class	BYTE
ShortType	Short.class	SHORT
IntegerType	Integer.class	INT
LongType	Long.class	LONG
FloatType	Float.class	FLOAT
DoubleType	Double.class	DOUBLE
StringType	String.class	STRING

SQL type	Java type	resolveType
BinaryType	byte[].class	BINARY
DecimalType.Fixed(precision, scale)	BigDecimal.class	Decimal
DateType	LocalDate.class	DATE
TimestampType	ZonedDateTime.class	TIMESTAMP
TimestampNTZType	LocalDateTime.class	TIMESTAMP_NTZ
YearMonthIntervalType	Period.class	INTERVAL_YEAR_MONTH
DayTimeIntervalType	Duration.class	INTERVAL_DAY_TIME
List	List.class	LIST
Map	Map.class	MAP
Struct	Map.class	STRUCT

- 添加@deterministic注解：

  spark调度参数，deterministic支持类级别和函数级别，函数级别高于类级别。

  详细UDTF函数实现，可以参考如下样例代码：

  - 功能描述：

    对字符串进行分隔并生成分隔后的结果集。它接收两个字符串参数：body和split。

    函数的主要功能是将body字符串按照split指定的分隔符进行分隔，并将分隔后的每个子字符串与一个固定的版本标签（version，值为"tag"）组合成一个新的对象数组。

    最终函数返回一个二维对象数组，其中每个元素都是一个包含分隔后的子字符串和版本标签的数组。

  - 示例代码

```
package com.huawei.demo;

import com.huawei.spark.function.handlers.UDTFHandler;
import com.huawei.spark.function.types.FunctionType;
import com.huawei.spark.function.types.TypeResolve;


public class UDTFDemo extends UDTFHandler {
    @FunctionType(deterministic = true)
    @TypeResolve("string,string->string,string")
    public Object[][] splitString(Object[] args) {
        String body = (String) args[0];
        String split = (String) args[1];
        String[] value = body.split(split);
        Object[][] result = new Object[value.length][];
        String version = "tag";
        for (int i = 0; i < value.length; i++) {
            result[i] = new Object[] {value[i], version};
        }
```

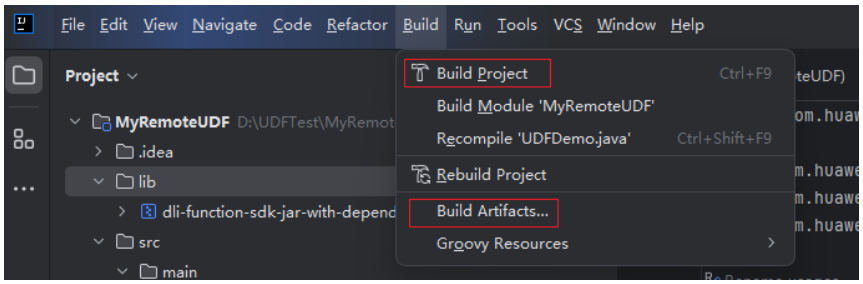
```
        return result;
    }
}
package com.huawei.demo;
import com.huawei.spark.function.handlers.UDTFHandler;
import com.huawei.spark.function.types.FunctionType;
import com.huawei.spark.function.types.TypeResolve;
public class UDTFDemo extends UDTFHandler {
    public String testUDTF(String a) {
        return a;
    }
    @TypeResolve("string, string->string")
    @FunctionType(deterministic = true)
    public Object[][] process(Object[] args) {
        String data = (String) args[0];
        String del = (String) args[1];
        String[] split = data.split(del);
        Object[][] result = new Object[split.length][];
        for (int i = 0; i < split.length; i++) {
            result[i] = new Object[] {split[i]};
        }
        return result;
    }
}
```

步骤 2：导出 Jar 包

编写调试完成代码后，通过IntelliJ IDEA工具编译代码并导出Jar包。

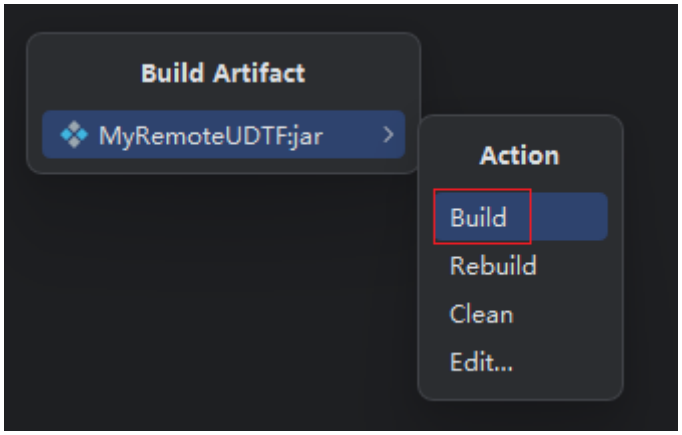
1. 单击“Build”，参考下图分别单击“Build Project ”、“Build Artifacts...”分别对代码进行编译和打包。

图 3-106 编译和打包



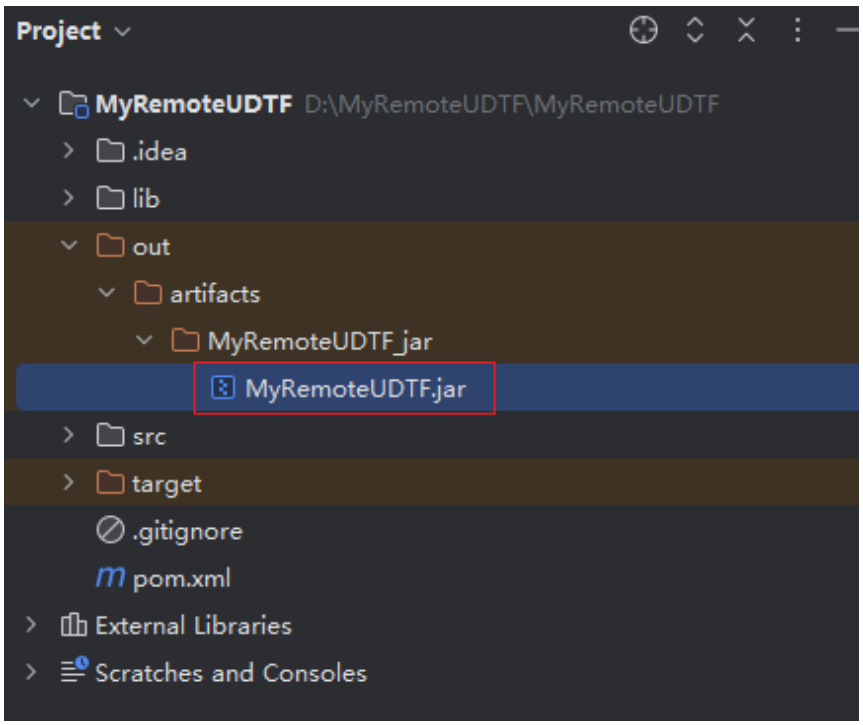
2. 在弹出的对话框中单击Build。

图 3-107 编译和打包



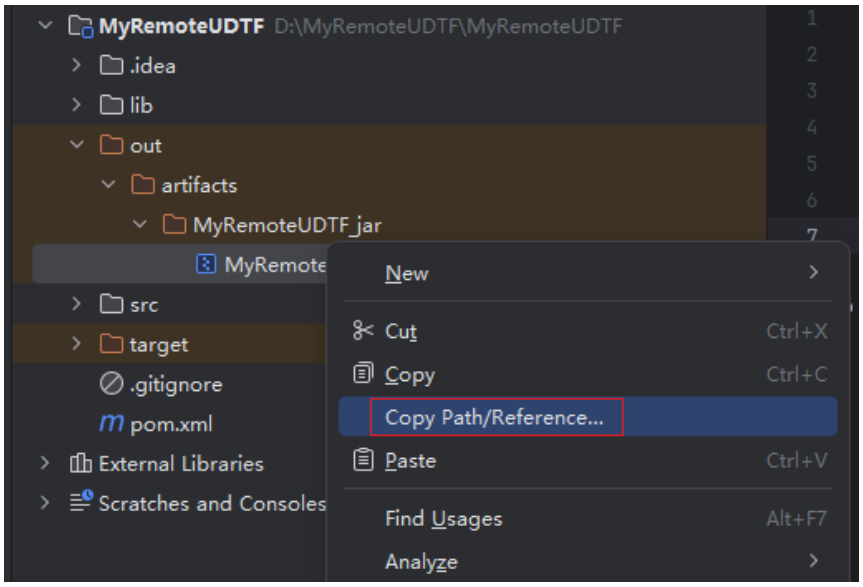
3. 完成上述步骤之后，可以在项目目录当中看到out文件夹，展开out目录。
   其中的MyRemoteUDTF.jar就是之后要上传到函数工作流FunctionGraph当中的jar包。

图 3-108 查看 MyRemoteUDTF.jar



4. 右键单击MyRemoteUDTF.jar，选择Copy Path/Reference，直接复制文件所在目录，方便之后上传JAR包使用。
   本示例将会生成到：“D:\UDTFTest\MyRemoteUDTF\out\artifacts\MyRemoteUDTF_jar\”目录下，文件名为“MyRemoteUDTF.jar”。

图 3-109 复制文件所在目录



3.6.4 创建 FunctionGraph 函数

了解**FunctionGraph业务使用流程**。

步骤 1：创建函数

1. 登录函数工作流FunctionGraph管理控制台。
2. 在左侧列表中单击“函数 > 函数列表”，单击“创建函数”

图 3-110 创建函数

3. 单击“创建空白函数”，并配置函数的基本信息。

   详细参数说明请参考**函数属性**。

   - 函数类型：本例为事件函数。
   - 运行时环境：Java 8。
   - 委托：当函数工作流服务需与其他云服务协同工作时，需要创建云服务委托，授权FunctionGraph使用这些云服务，并在FunctionGraph中为函数配置委托权限，以实现与其他云服务的协同工作。

     FunctionGraph单次调用数据量有**约束与限制**。函数如果超过4MB，建议配置访问OBS的委托权限。具体操作请参考**相关操作：配置FunctionGraph访问其他云服务的委托**。

图 3-111 创建函数

步骤 2：函数设置

1. **设置函数执行入口**

a. 函数创建完成后，在列表单击函数名称进入函数详情页面。
b. 选择“设置 > 常规设置”，在“函数执行入口”中设置UDTF的执行入口。
UDTF的执行入口格式为[包名].[类名].[执行函数名]，
- [包名].[类名]为编写的UDTF函数所在类，
- 执行函数名固定为handleRequest。

本示例设置为：com.huawei.demo.UDTFDemo.handleRequest，
c. 设置完成之后单击保存按钮。

2. **设置执行超时时间和内存**
a. 函数创建完成后，在列表单击函数名称进入函数详情页面。
b. 选择“设置 > 常规设置”，在“函数执行入口”中设置执行时间和内存。
根据UDTF复杂情况配置执行超时时间和内存大小。内存大小与计费关联。
本示例选择执行时间为15秒，内存为1024MB。
c. 设置完成之后单击保存按钮。

图 3-112 常规设置

3. **设置类隔离信息**
a. 函数创建完成后，在列表单击函数名称进入函数详情页面。
b. 选择“设置 > 高级设置”，配置“类隔离选项”。
类隔离开启后将使用单独的类加载器加载上传的代码和依赖，解决依赖和运行时的冲突，如果无依赖冲突问题无需打开。

本示例中关闭类隔离选项，设置完成后单击保存按钮。

图 3-113 设置类隔离

步骤 3：上传 Jar 包

1. 在FunctionGraph管理控制台，函数列表中单击函数名称进入函数详情页面。
2. 单击"代码"，在右侧上传代码下拉框当中选择JAR文件，将之前导出的JAR包上传
3. 上传成功之后，可以在函数详情页面复制URN（Uniform Resource Name）。URN是函数的唯一标识函数，在下一步调用函数时需要用到。

图 3-114 复制 URN

相关操作：配置 FunctionGraph 访问其他云服务的委托

1. 登录管理控制台。
2. 单击右上方登录的用户名，在下拉列表中选择“统一身份认证”。
3. 在左侧导航栏中，单击“委托”。
4. 在“委托”页面，单击“创建委托”。
5. 在“创建委托”页面，设置如下参数：
   - 委托名称：按需填写。
   - 委托类型：选择“云服务”。
   - 云服务：（“委托类型”选择“云服务”时出现此参数项。）在下拉列表中选择“FunctionGraph"。
   - 持续时间：选择“永久”。
   - 描述：非必选，可以填写“拥有OBS OperateAccess权限的委托”。

图 3-115 创建委托

6. 配置完委托的基本信息后，单击“立即授权”。
7. 进入“选择策略”页面，在右方搜索框中搜索OBS Administrator权限并勾选。勾选完成后单击“下一步”。
8. 单击“下一步”，选择委托的授权范围。

   了解更多授权操作说明请参考**创建用户组并授权**。

   - 所有资源：授权后，IAM用户可以根据权限使用账号中所有资源，包括企业项目、区域项目和全局服务资源。
   - 全局服务资源：全局服务部署时不区分区域，访问全局级服务，不需要切换区域，全局服务不支持基于区域项目授权。如对象存储服务（OBS）、内容分发网络（CDN）等。授权后，用户根据权限使用全局服务的资源。
   - 指定区域项目资源：授权后，IAM用户根据权限使用所选区域项目中的资源，未选择的区域项目中的资源，该IAM用户将无权访问。
   - 指定企业项目资源：授权后，IAM用户根据权限使用所选企业项目中的资源。如企业项目A包含资源B，资源B部署在北京四和上海二，IAM用户所在用户组关联企业项目A后，北京四和上海二的资源B用户都可访问，不在企业项目A内的其他资源，该IAM用户将无权访问。

   本例选择的是OBS策略，因此选择全局服务资源。
9. 单击“确定”，完成授权。
10. 授权后需等待15-30分钟才可生效。

    授权生效后返回函数基本信息界面，选择对应的委托，权限策略当中会显示已授权的策略。

3.6.5 在 Spark SQL 作业中使用 Remote UDTF

3.6.5.1 SQL 创建函数

功能描述

创建使用Remote UDTF自定义函数应用于Spark作业开发当中。

语法格式

```
CREATE FUNCTION [db_name.]function_name AS class_name USING FUNCTIONGRAPH 'urn'
```

或

```
CREATE OR REPLACE FUNCTION [db_name.]function_name AS class_name USING FUNCTIONGRAPH 'urn'
```

注意事项

无

关键字

urn：在FunctionGraph函数列表页面，选择您需要复制URN的函数，在操作列中单击复制URN。

示例

创建本示例函数remote_udtf，执行的指令如下：

```
CREATE FUNCTION remote_udtf AS 'splitString' USING FUNCTIONGRAPH 'urn:fss:cn-north-7:330e068af1334c9782f4226acc00a2e2:function:default:test:latest'
```

选择对应的数据库与队列，输入SQL指令后单击执行按钮。

图 3-116 创建函数

3.6.5.2 SQL 调用函数

功能描述

调用指定函数。

语法格式

```
select function_name;
```

注意事项

无

关键字

function_name：即在FunctionGraph创建的自定义函数名称。

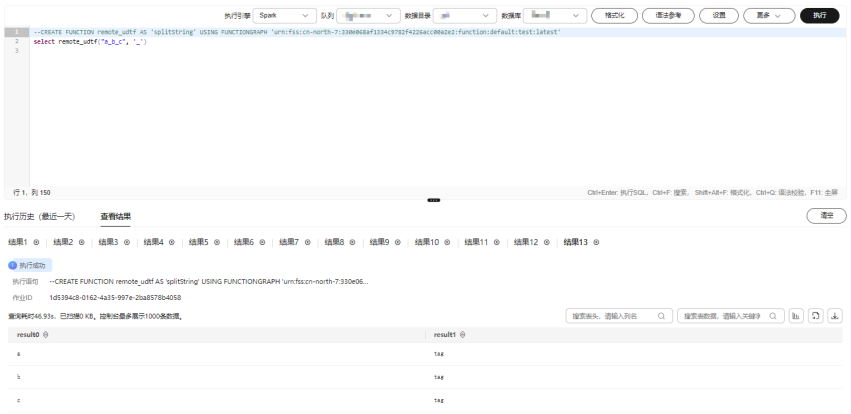
示例

调用本示例函数remote_udtf。

```
select remote_udtf("a_b_c", '_') ;
```

选择对应的数据库与队列，输入SQL指令后单击执行按钮。

图 3-117 调用函数



3.6.5.3 SQL 删除函数

功能描述

删除指定函数。

语法格式

```
DROP [TEMPORARY] FUNCTION [IF EXISTS] [db_name.] function_name;
```

注意事项

无

关键字

- TEMPORARY：所删除的函数是否为临时函数。
- IF EXISTS：所删除的函数不存在时使用，可避免系统报错。

示例

```
DROP FUNCTION remote_udtf;
```

3.6.5.4 显示所有函数

功能描述

查看当前工程下所有的函数。

语法格式

```
SHOW [USER|SYSTEM|ALL] FUNCTIONS ([LIKE] regex | [db_name.] function_name);
```

注意事项

无

关键字

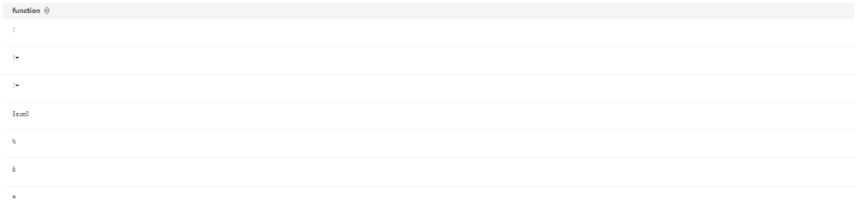
LIKE：此限定符仅为兼容性而使用，没有任何实际作用。

示例

```
SHOW FUNCTIONS;
```

运行结果：

图 3-118 显示所有函数



3.6.5.5 显示函数详情

功能描述

查看指定函数的相关信息。

语法格式

```
DESCRIBE FUNCTION [EXTENDED] [db_name.] function_name;
```

注意事项

UDTF不可以嵌套使用。

关键字

EXTENDED：显示扩展使用信息。

示例

```
DESCRIBE FUNCTION remote_udtf;
```

运行结果：

图 3-119 显示所有函数详情