数据湖计算

实践教程

产品文档

腾讯云

版权所有：腾讯云计算（北京）有限责任公司

【版权声明】

©2013-2024 腾讯云版权所有

本文档著作权归腾讯云单独所有，未经腾讯云事先书面许可，任何主体不得以任何形式复制、修改、抄袭、传播全部或部分本文档内容。

【商标声明】

及其它腾讯云服务相关的商标均为腾讯云计算（北京）有限责任公司及其关联公司所有。本文档涉及的第三方主体的商标，依法由权利人所有。

【服务声明】

本文档意在向客户介绍腾讯云全部或部分产品、服务的当时的整体概况，部分产品、服务的内容可能有所调整。您所购买的腾讯云产品、服务的种类、服务标准等应由您与腾讯云之间的商业合同约定，除非双方另有约定，否则，腾讯云对本文档内容不做任何明示或默示的承诺或保证。

文档目录

实践教程
DLC 原生表
DLC 原生表核心能力

最近更新时间：2024-07-31 17:34:35

概述

DLC 原生表（Iceberg）是基于 Iceberg 湖格式打造的性能强、易用性高、操作使用简单的表格式，用户可在该基础上完整数据探索，建设 Lakehouse 等应用。用户首次在使用 DLC 原生表（Iceberg）时需要按照如下5个主要步骤进行：

1. 开通 DLC 托管存储。
2. 购买引擎。
3. 创建数据库表。结合使用场景，选择创建 append 或者 upsert 表，并携带优化参数。
4. 配置数据优化。结合表类型，选择独立的优化引擎，配置优化选项。
5. 导入数据到 DLC 原生表。DLC 支持多种数据写入，如 insert into/merge into/upsert，支持多种导入方式，如 spark/presto/flink/inlong/oceanus。

Iceberg 原理解析

DLC 原生表（Iceberg）采用 Iceberg 表格式作为底层存储，在兼容开源 Iceberg 能力的基础上，还做了存算分离性能、易用性增强。

Iceberg 表格式通过划分数据文件和元数据文件管理用户的数据。

数据层（data layer）：由一系列 data file 组成，用于存放用户表的数据，data file 支持 parquet、avro、orc 格式，DLC 中默认为 parquet 格式。

由于 iceberg 的快照机制，用户在删除数据时，并不会立即将数据从存储上删除，而是写入新的 delete file，用于记录被删除的数据，根据使用的不同，delete file 分为 position delete file 和 equality delete file。

position delete 是位置删除文件，记录某个 data file 的某一行被删除的信息。

equality delete 为等值删除文件，记录某个 key 的值被删除，通常用在 upsert 写入场景。delete file 也属于 data file 的一种类型。

元数据层（metadata layer）：由一系列的 manifest、manifest list、metadata 文件组成，manifest file 包含一系列的 data file 的元信息，如文件路径、写入时间、min-max 值、统计值等信息。

manifest list 由 manifest file组成，通常一个 manifest list 包含一个快照的 manifest file。

metadata file 为 json 格式，包含一些列的 manifest list 文件信息和表的元信息，如表 schema、分区、所有快照等。

每当表状态发生变化时，都会产生一个新的 metadata file 覆盖原有 metadata 文件，且该过程有 Iceberg 内核的原子性保证。

版权所有：腾讯云计算（北京）有限责任公司

原生表使用场景

DLC 原生表（iceberg）作为 DLC lakehouse 主推格式，从使用场景上，可分为 Append 场景表和 Upsert 表，Append 场景表采用的V1格式，Upsert 表采用的V2格式。

Append 场景表：该场景表仅支持 Append，Overwrite，Merge into 方式写入。

Upsert 场景表：相比 Append，写入能力相比多一种 Upsert 写入模式。

原生表使用场景及特点汇总如下表描述。

表类型	使用场景及建议	特点
原生表（iceberg）	1. 用户有实时写入需求，包括 append/merge into/upsert 场景需求，不限于 inlong/oceans/自建 flink 实时写入。 2. 用户不想直接管理存储相关的运维，交给 DLC 存储托管。 3. 用户不想对 Iceberg 表格式进行运维，交给 DLC 进行调优和运维。 4. 希望使用 DLC 提供的自动数据优化能力，持续对数据进行优化。	1. Iceberg 表格式。 2. 使用前需要开通托管存储。 3. 数据存储存储在 DLC 提供的托管存储上。 4. 不需要指定 external、location 等信息。 5. 支持开启 DLC 智能数据优化。

为更好的管理和使用 DLC 原生表（Iceberg），您创建该类型的表时需要携带一些属性，这些属性参考如下。用户在创建表时可以携带上这些属性值，也可以修改表的属性值，详细的操作请参见 DLC 原生表操作配置。

属性值	含义	配置指导
format-version	iceberg 表版本，取值范围1、2，默认取值为1	如果用户写入场景有 upsert，该值必须设置为2
write.upsert.enabled	是否开启 upsert，取值为 true；不设置则为不开启	如果用户写入场景有 upsert，必须设置为 true
write.update.mode	更新模式	merge-on-read 指定为 MOR 表，缺省为 COW
write.merge.mode	merge 模式	merge-on-read 指定为 MOR 表，缺省为 COW
write.parquet.bloom-filter-enabled.column.{col}	开启 bloom，取值为 true 表示开启，缺省不开启	upsert 场景必须开启，需要根据上游的主键进行配置；如上游有多个主键，最多取前两个；开启后可提升 MOR 查询和小文件合并性能
write.distribution-mode	写实模式	建议取值为 hash，当取值为 hash 时，当数据写入时会自行进行 repartition，缺点是影响部分写入性能
write.metadata.delete-after-	开始 metadata 文	强烈建议设置为 true，开启后 iceberg 在产生快照时会

commit.enabled	件自动清理	自动清理历史的 metadata 文件，可避免大量的 metadata 文件堆积
write.metadata.previous-versions-max	设置默认保留的 metadata 文件数量	默认值为100，在某些特殊的情况下，用户可适当调整该值，需要配合 write.metadata.delete-after-commit.enabled 一起使用
write.metadata.metrics.default	设置列 metrices 模型	必须取值为 full

原生表核心能力

存储托管

DLC 原生表（Iceberg）采用了存储数据托管模式，用户在使用原生表（Iceberg）时，需要先开通托管存储，将数据导入到 DLC 托管的存储空间。用户采用 DLC 存储托管，将获得如下收益。
数据更安全：Iceberg 表数据分为元数据和数据两个部分，一点某系文件被破坏，将导致整个表查询异常（相比于 Hive 可能是损坏的文件数据不能查询），存储托管在 DLC 可减少用户在不理解 Iceberg 的情况下对某些文件进行破坏。
性能：DLC 存储托管默认采用 chdfs 作为存储，相比于普通 COS，性能有较大的提升。
减少存储方面的运维：用户采用存储托管后，可不用再开通及运维对象存储，能减少存储的运维。
数据优化：DLC 元生表（Iceberg）采用存储托管模式，DLC 提供的数据优化能针对元生表持续优化。

ACID 事务

Iceberg 写入支持在单个操作中删除和新增，且不会部分对用户可见，从而提供原子性写入操作。
Iceberg 使用乐观并发锁来确保写入数据不会导致数据不一致，用户只能看到读视图中已经提交成功的数据。
Iceberg 使用快照机制和可序列化隔离级确保读取和写入是隔离的。
Iceberg 确保事务是持久化的，一旦移交成功就是永久性的。

写入

写入过程遵循乐观并发控制，写入者首先假设当前表版本在提交更新之前不会发生变更，对表数据进行更新/删除/新增，并创建新版本的元数据文件，之后尝试替换当前版本的元数据文件到新版本上来，但是在替换过程中，Iceberg 会检查当前的更新是否是基于当前版本的快照进行的，
如果不是则表示发生了写冲突，有其他写入者先更新了当前 metadata，此时写入必须基于当前的 metadata 版本从新更新，之后再次提交更新，整个提交替换过程由元数据锁保证操作的原子性。

读取

Iceberg 读取和写入是独立的过程，读者始终只能看到已经提交成功的快照，通过获取版本的 metadata 文件，获取的快照信息，从而读取当前表的数据，由于在未完成写入时，并不会更新 metadata 文件，从而确保始终从已经完成的操作中读取数据，无法从正在写入的操作中获取数据。

版权所有：腾讯云计算（北京）有限责任公司

冲突参数配置

DLC 托管表（Iceberg）在写入并发变高时将会触发写入冲突，为降低冲突频率，用户可从如下方面对业务进行合理的调整。
进入合并的表结构设置，如分区，合理规划作业写入范围，减少任务写入时间，在一定程度上降低并发冲突概率。
作业进行一定程度的合并，减小写入并发量。
DLC 还支持一些列冲突并发重试的参数设置，在一定程度可提供重试操作的成功率，减小对业务的影响，参数含义及配置指导如下。

属性值	系统默认值	含义	配置指导
commit.retry.num-retries	4	提交失败后的重试次数	发生重试时，可尝试提大次数
commit.retry.min-wait-ms	100	重试前的最小等待时间，单位为毫秒	当时冲突十分频繁，如等待一段时间后依然冲突，可尝试调整该值，加大重试之间的间隔
commit.retry.max-wait-ms	60000（1 min）	重试前的最大等待时间，单位为毫秒	结合commit.retry.min-wait-ms一起调整使用
commit.retry.total-timeout-ms	1800000（30 min）	整个重试提交的超时时间	-

隐藏式分区

DLC 原生表（Iceberg）隐藏分区是将分区信息隐藏起来，开发人员只需要在建表的时候指定分区策略，Iceberg 会根据分区策略维护表字段与数据文件之间的逻辑关系，在写入和查询时无需关注分区布局，Iceberg 在写入数据是根据分区策略找到分区信息，并将其记录在元数据中，查询时也会更具元数据记录过滤到不需要扫描的文件。DLC 原生表（Iceberg）提供的分区策略如下表所示。

转换策略	描述	原始字段类型	转换后类型
identity	不转换	所有类型	与原类型一致
bucket[N, col]	hash分桶	int, long, decimal, date, time, timestamp, timestamptz, string, uuid, fixed, binary	int
truncate[col]	截取固定长度	int, long, decimal, string	与原类型一致
year	提取字段 year 信息	date, timestamp, timestamptz	int
month	提取字段 mouth 信息	date, timestamp, timestamptz	int

day	提取字段 day 信息	date, timestamp, timestamptz	int
hour	提取字段 hour 信息	timestamp, timestamptz	int

元数据查询和存储过程

DLC 原生表（Iceberg）可调用存储过程语句查询各类型表信息，如文件合并、快照过期等，如下表格提供部分常用的查询方法。

场景	CALL 语句	执行引擎
查询 history	select * from `DataLakeCatalog` . `db` . `sample$history`	DLC spark SQL 引擎、presto 引擎
查询快照	select * from `DataLakeCatalog` . `db` . `sample$snapshots`	DLC spark SQL 引擎、presto 引擎
查询 data 文件	select * from `DataLakeCatalog` . `db` . `sample$files`	DLC spark SQL 引擎、presto 引擎
查询 manifests	select * from `DataLakeCatalog` . `db` . `sample$manifests`	DLC spark SQL 引擎、presto 引擎
查询分区	select * from `DataLakeCatalog` . `db` . `sample$partitions`	DLC spark SQL 引擎、presto 引擎
回滚指定快照	CALL DataLakeCatalog. `system` .rollback_to_snapshot('db.sample', 1)	DLC spark SQL 引擎
回滚到某个时间点	CALL DataLakeCatalog. `system` .rollback_to_timestamp('db.sample', TIMESTAMP '2021-06-30 00:00:00.000')	DLC spark SQL 引擎
设置当前快照	CALL DataLakeCatalog. `system` .set_current_snapshot('db.sample', 1)	DLC spark SQL 引擎
合并文件	CALL DataLakeCatalog. `system` .rewrite_data_files(table => 'db.sample', strategy => 'sort', sort_order => 'id DESC NULLS LAST,name ASC NULLS FIRST')	DLC spark SQL 引擎
快照过期	CALL DataLakeCatalog. `system` .expire_snapshots('db.sample', TIMESTAMP '2021-06-30 00:00:00.000', 100)	DLC spark SQL 引擎
移除孤立文件	CALL DataLakeCatalog. `system` .remove_orphan_files(table => 'db.sample', dry_run => true)	DLC spark SQL 引擎

重新元数据	CALL DataLakeCatalog. `system` .rewrite_manifests('db.sample')	DLC spark SQL 引擎

数据优化

优化策略

DLC 原生表（Iceberg）提供的具备继承关系优化策略，用户可将策略配置在数据目录、数据库和数据表上。具体配置操作请参见 开启数据优化。

数据目录配置优化策略：该数据目录下的所有库下的所有的原生表（Iceberg）默认复用该数据目录优化策略。

数据库配置优化策略：该数据库下的所有原生表（Iceberg）默认复用该数据库优化策略。

数据表配置优化策略：该配置仅针对配置的原生表（Iceberg）生效。

用户通过上面的组合配置，可实现针对某库某表定制化优化策略，或者某些表关闭策略。

DLC 针对优化策略还提供高级参数配置，如用户对Iceberg熟悉可根据实际场景定制化高级参数，如下图所示。

DLC 针对高级参数设置了默认值。DLC 会将文件尽可能合并到128M大小，快照过期时间为2天，保留过期的5个快照，快照过期和清理孤立文件的执行周期分别为600分钟和1440分钟。

针对 upsert 写入场景，DLC 默认还提供合并阈值，该部分参数由 DLC 提供，在超过5min的时间新写入的数据满足其中一个条件将会触发小文件合并，如表所示。

参数	含义	取值
AddDataFileSize	写入新增数据文件数量	20
AddDeleteFileSize	写入新增Delete文件数据量	20
AddPositionDeletes	写入新增Position Deletes记录数量	1000
AddEqualityDeletes	写入新增Equality Deletes记录数量	1000

优化引擎

DLC 数据优化通过执行存储过程完成对数据的优化，因此需要数据引擎用于执行存储过程。当前 DLC 支持使用 Spark SQL 引擎作为优化引擎，在使用时需注意如下几点：

数据优化的 Spark SQL 引擎与业务引擎分开使用，可避免数据优化任务与业务任务相互抢占资源，导致任务大量排队和业务受阻。
生产场景建议优化资源64CU起，除非量特性表，如果小于10张表且单表数据量超过2G，建议资源开启弹性，防止突发流量，建议采用包年包月集群，防止提交任务时集群不可用导致优化任务失败。

参数定义

数据库、数据表数据优化参数设置在库表属性上，用户可以通过创建库、表时携带数据优化参数（DLC 原生表提供的可视化创建库表用户可配置数据优化）；用户也可通过 ALTER DATABASE/TABLE 对表数据进行表更，修改数据优化参数详细的操作请参见 DLC 原生表操作配置。

属性值	含义	默认值	取值说明
smart-optimizer.inherit	是否继承上一级策略	default	none：不继承；default：继承
smart-optimizer.written.enable	是否开启写入优化	disable	disable：不开启；enable：开启。默认不开启
smart-optimizer.written.advance.compact-enable	（可选）写入优化高级参数，是否开始小文件合并	enable	disable：不开启；enable：开启。
smart-optimizer.written.advance.delete-enable	（可选）写入优化高级参数，是否开始数据清理	enable	disable：不开启；enable：开启。
smart-optimizer.written.advance.min-input-files	（可选）合并最小输入文件数量	5	当某个表或分区下的文件数目超过最小文件个数时，平台会自动检查并启动文件优化合并。文件优化合并能有效提高分析查询性能。最小文件个数取值较大时，资源负载越高，最小文件个数取值较小时，执行更灵活，任务会更频繁。建议取值为5。
smart-optimizer.written.advance.target-file-size-bytes	（可选）合并目标大小	134217728 (128 MB)	文件优化合并时，会尽可能将文件合并成目标大小，建议取值128M。
smart-	（可选）	2	快照存在时间超过该值时，平台会将该

optimizer.written.advance.before-days	快照过期时间，单位天		快照标记为过期的快照。快照过期时间取值越长，快照清理的速度越慢，占用存储空间越多。
smart-optimizer.written.advance.retain-last	（可选）保留过期快照数量	5	超过保留个数的过期快照将会被清理。保留的过期快照个数越多，存储空间占用越多。建议取值为5。
smart-optimizer.written.advance.expired-snapshots-interval-min	（可选）快照过期执行周期	600（10 hour）	平台会周期性扫描快照并过期快照。执行周期越短，快照的过期会更灵敏，但是可能消耗更多资源。
smart-optimizer.written.advance.remove-orphan-interval-min	（可选）移除孤立文件执行周期	1440（24 hour）	平台会周期性扫描并清理孤立文件。执行周期越短，清理孤立文件会更灵敏，但是可能消耗更多资源。

优化类型

当前 DLC 提供写入优化和数据清理两种类型，写入优化对用户写入的小文件进行合并更大的文件，从而提供查询效率；数据清理则是清理历史过期快照的存储空间，节约存储成本。

写入优化

小文件合并：将业务侧写入的小文件合并为更大的文件，提升文件查询效率；处理写入的deletes文件和data文件合并，提升MOR查询效率。

数据清理

快照过期：执行删除过期的快照信息，释放历史数据占据的存储空间。

移除孤立文件：执行移除孤立文件，释放无效文件占据的存储空间。

根据用户的使用场景，在优化类型上有一定差异，如下所示。

优化类型	建议开启场景
写入优化	upsert 写场景：必须开启 merge into 写场景：必须开启 append 写入场景：按需开启
数据清理	upsert 写场景：必须开启 merge into 写入场景：必须开启 append 写入场景：建议开启，并结合高级参数及历史数据回溯需求配置合理的过期删除时间

DLC 的写入优化不仅完成小文件的合并，还可以提供手动构建索引，用户需要提供索引的字段及规则，之后 DLC 将产生对应的存储过程执行语句，从而完成索引的构建。该能在 upsert 场景和结合小文件合并同时进行，完成小文件合并的时候即可完成索引构建，大大提高索引构建能力。

该功能目前处于测试阶段，如需要使用，可 联系我们 进行配置。

优化任务

DLC 优化任务产生有时间和事件两种方式。

时间触发

时间触发是优化高级参数配置的执行时间，周期性地触发检查是否需要优化，如对应治理项满足条件后，将会产生对应的治理任务。当前时间触发的周期至少是60min，通常用在清理快照和移除孤立文件。
时间触发针对小文件合并类型的优化任务仍然有效，当时触发周期默认为60min。
V1表（需后端开启）情况，每60min触发一次小文件合并。
V2表情况，防止表写入慢，长时间达不到事件触发条件，当时间触发V2进行小文件合并时，需要满足上一次小文件合并时间间隔超过1小时。
当快照过期和移除孤立文件任务执行失败或者超时时，在下一个检查周期会再次执行，检查周期为60min。

事件触发

事件触发发生了表 Upsert 写入场景，主要是 DLC 数据优化服务后台会监控用户表数据的 Upsert 表写入的情况，当写的达到对应的条件时，触发产生治理任务。事件触发用在小文件并场景，特别是 flink upsert 实时写入场景，数据写入快，频繁产生小文件合并任务。
如数据文件阈值20，deletes 文件阈值20，则写入20个文件或者20个 deletes 文件，并同时满足相同任务类型之间的产生的间隔默认最小时间5min时，就会触发产生小文件合并。

生命周期

DLC 原生表的生命周期（Lifecycle），指表（分区）数据从最后一次更新的时间算起，在经过指定的时间后没有变动，则此表（分区）将被自动回收。DLC 元数据表的生命周期执行时，只是产生新的快覆盖过期的数据，并不会立即将数据从存储空间上移除，数据真正从存储上移除需要依赖于元数据表数据清理（快照过期和移除孤立文件），因此生命周期需要和数据清理一起使用。
注意：
生命周期目前处于邀测阶段，如需要开通，欢迎 联系我们。
生命周期移除分区时只是从当前快照中逻辑移除该分区，但是被移除的文件并不会立即从存储系统中删除，被移除的文件只有等到快照过期才会从存储系统上删除。

参数定义

数据库、数据表生命周期参数设置在库表属性上，用户可以通过创建库、表时携带生命周期参数（DLC 原生表提供的可视化创建库表用户可配置生命周期）；用户也可通过 ALTER DATABASE/TABLE 对表数据进行表更改，修改生命周期参数详细的操作请参见 DLC 原生表操作配置。

属性值	含义	默认值	取值说明
smart-optimizer.lifecycle.enable	是否开启生命周期	disable	disable：不开启；enable：开启。默认不开启
smart-	生命周期实现周	30	当 smart-optimizer.lifecycle.enable 取值为

optimizer.lifecycle.expiration	期，单位：天		enable 时生效，需大于1

结合 WeData 管理原生表生命周期

如果用户分区表的按照天分区，如分区值为yyyy-MM-dd或者yyyyMMdd的分区值，可配合WeData完成数据生命周期管理。

数据导入

DLC 原生表（Iceberg）支持多种方式的数据导入，根据数据源不同，可参考如下方式进行导入。

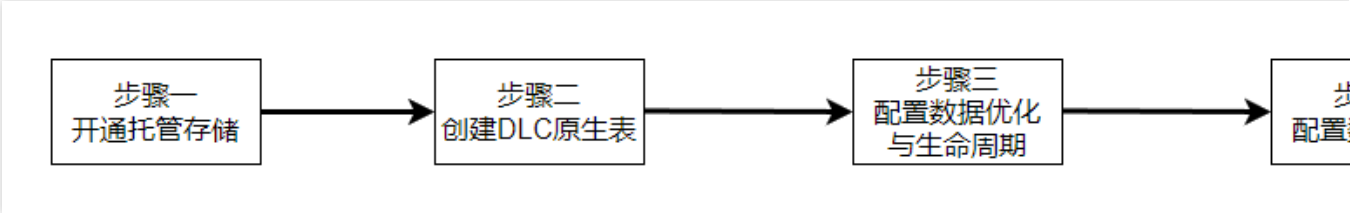
数据位置	导入建议
数据在用户自己的 COS 桶上	通过在 DLC 建立外部表，之后通过 Insert into/overwrite 的方式导入
数据在用户本地（或者其他执行机上）	用户需要将数据上传到用户自己的 COS 桶上，之后建立 DLC 外部表，通过 insert into/overwrite 的方式导入
数据在用户 mysql	用户可通过 flink/inlong/oceans 方式将数据导入，详细的数据入湖操作方式请参见 DLC原生表（Iceberg）入湖实践。
数据在用户自建 hive 上	用户通过建立联邦 hive 数据目录，之后通过 insert into/overwrite 的方式导入

DLC 原生表操作配置

最近更新时间：2024-07-31 17:34:51

概述

用户在使用 DLC 原生表（Iceberg）时，可以参考如下流程进行原生表创建和完成相关的配置。



步骤一：开启托管存储

说明：

托管存储需要 DLC 管理员开启。

开启托管存储需要在控制台上操作。详情请参见 托管存储配置。如果使用元数据加速桶，需注意权限配置，详情请参见 元数据加速桶的绑定。需注意共享引擎无法访问元数据加速桶。

步骤二：创建 DLC 原生表

创建原生表有两种方式。

1. 通过控制台界面可视化建表。
2. 通过 SQL 建表。

说明：

创建 DLC 原生表之前，需要先创建数据库。

通过控制台界面建表

DLC 提供数据管理模块进行建表，具体操作请参见数据管理。

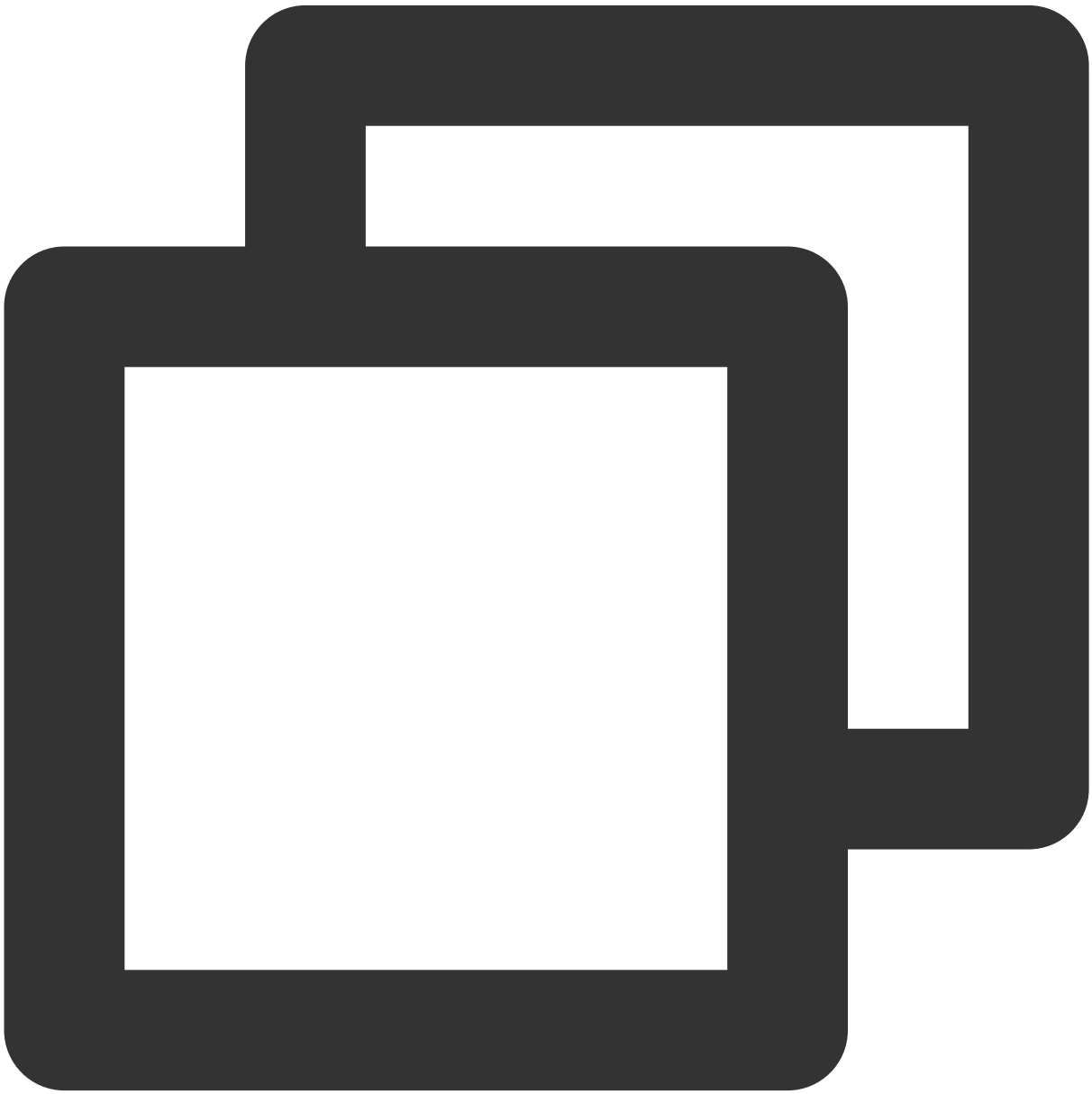
通过 SQL 建表

SQL 建表时用户自己编写 CREATE TABLE SQL 语句进行创建，DLC 原生表（Iceberg）创建不需要指定表描述，不需要指定 location，不需要指定表格式。但根据使用场景，需要自行补充一些高级参数，参数通过 TBLPROPERTIES 的方式带入。

如果您在创建表的时候没有参数，或者要修改某些属性值，可通过 alter table set tblproperties 的方式进行修改，执行 alter table 修改之后，重启上游的导入任务即可完成属性值修改或者增加。

以下提供 Append 场景和 Upsert 场景的典型建表语句，用户在使用时可在该语句的基础上结合实际情况进行调整。

Append 场景建表

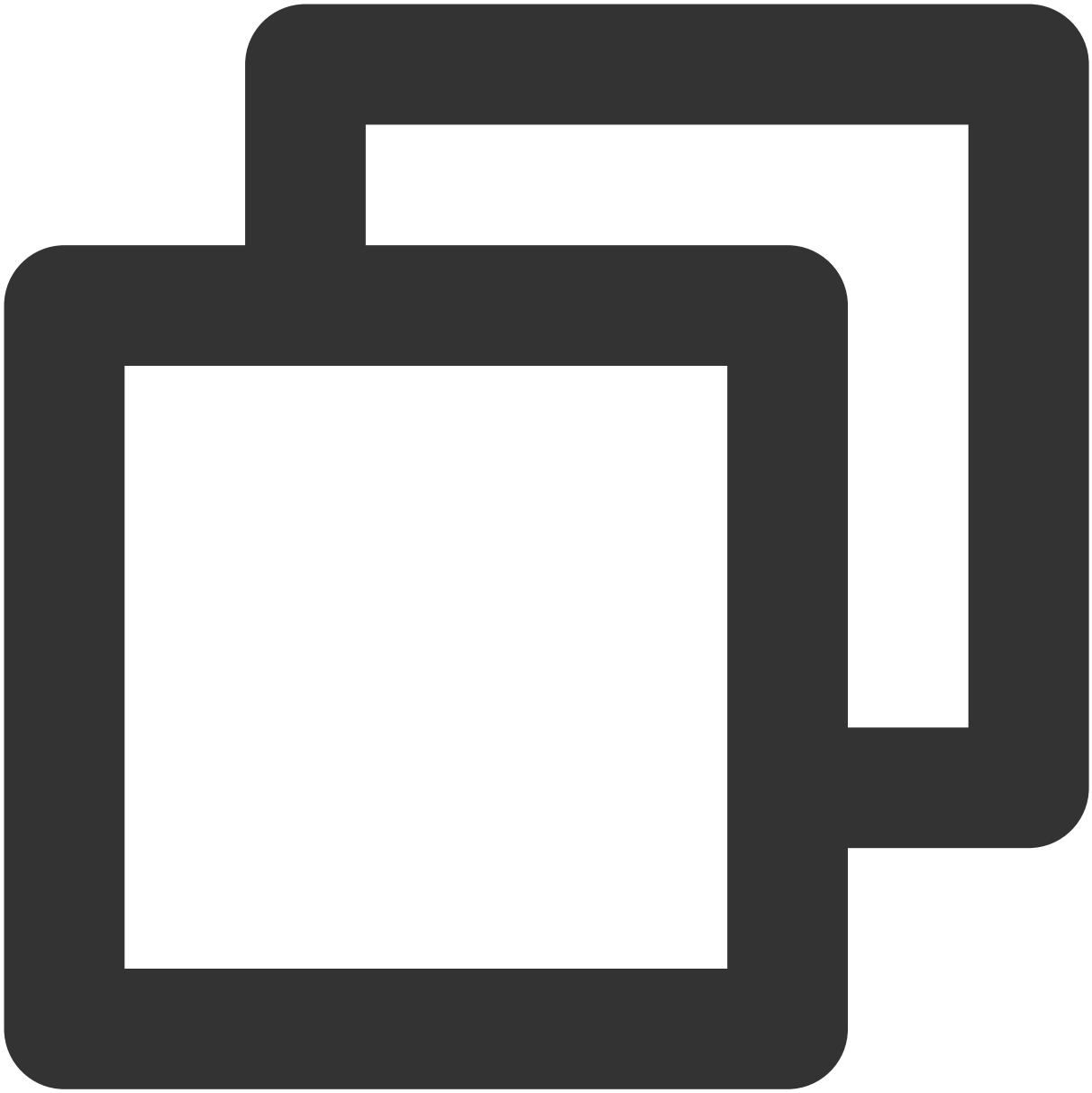


```
CREATE TABLE IF NOT EXISTS `DataLakeCatalog`.`axitest`.`append_case` (`id` int, `n
```

```
  PARTITIONED BY (`pt`)
  TBLPROPERTIES (
     'format-version' = '1',
     'write.upsert.enabled' = 'false',
     'write.distribution-mode' = 'hash',
     'write.metadata.delete-after-commit.enabled' = 'true',
     'write.metadata.previous-versions-max' = '100',
     'write.metadata.metrics.default' = 'full',
     'smart-optimizer.inherit' = 'default'
  );
```

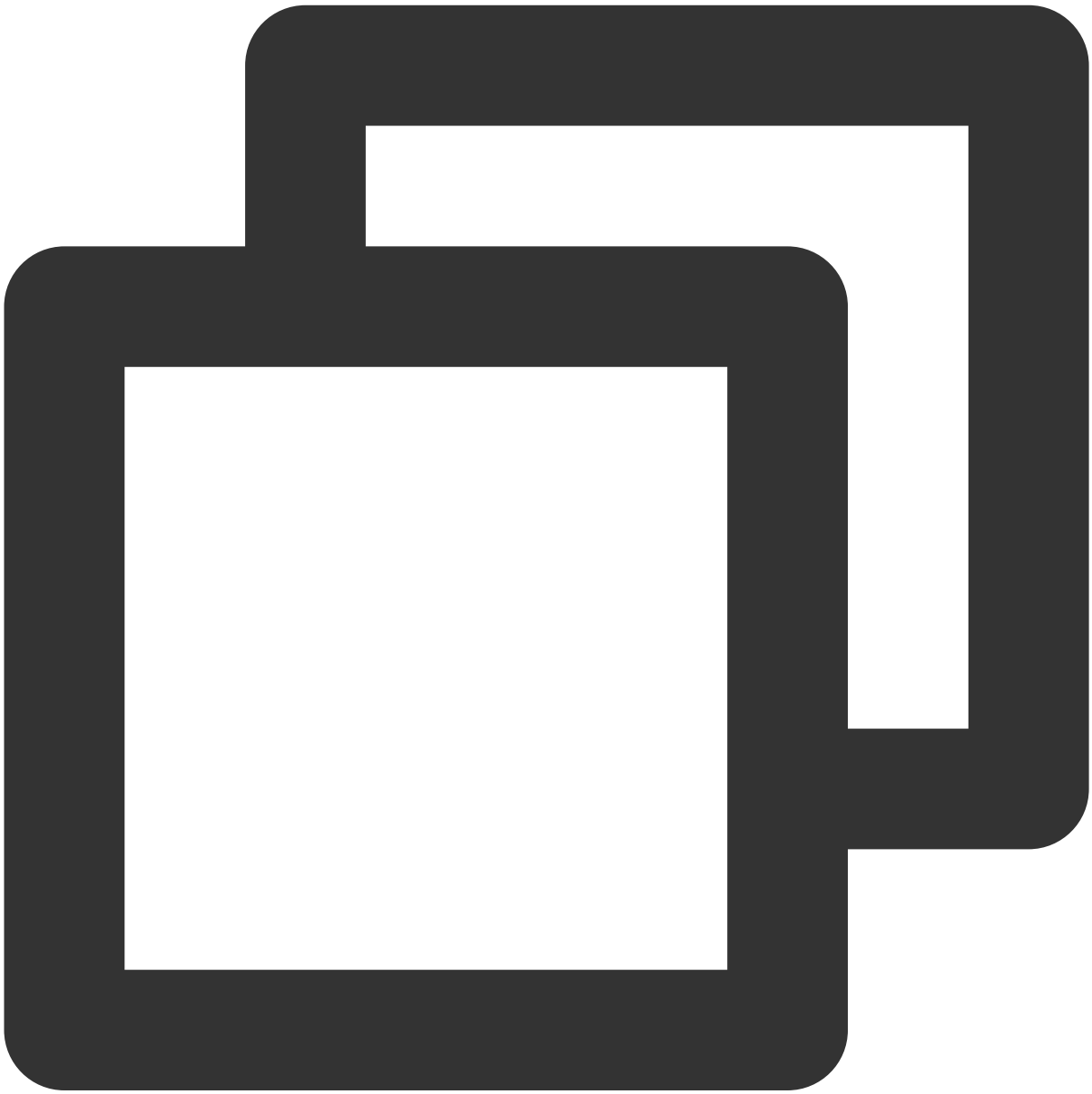
Upsert 场景建表

Upsert 场景建表需要指定 version为2，设置 write.upsert.enabled 属性为 true，且要根据 upsert 的键值设置 bloom，如果用户有多个主键，一般取前两个键值设置 bloom。如果 upsert 表为非分区场景，且 upsert 更新频繁，数据量大，可根据主键做一定的分桶打散。
非分区表及分区表样例参考如下。



```
// 分区表
CREATE TABLE IF NOT EXISTS `DataLakeCatalog`.`axitest`.`upsert_case` (`id` int, `na
PARTITIONED BY (bucket(4, `id`))
TBLPROPERTIES (
    'format-version' = '2',
    'write.upsert.enabled' = 'true',
    'write.update.mode' = 'merge-on-read',
    'write.merge.mode' = 'merge-on-read',
    'write.parquet.bloom-filter-enabled.column.id' = 'true',
    'dlc.ao.data.govern.sorted.keys' = 'id',
    'write.distribution-mode' = 'hash',
```

```
    'write.metadata.delete-after-commit.enabled' = 'true',
    'write.metadata.previous-versions-max' = '100',
    'write.metadata.metrics.default' = 'full',
    'smart-optimizer.inherit' = 'default'
);
```

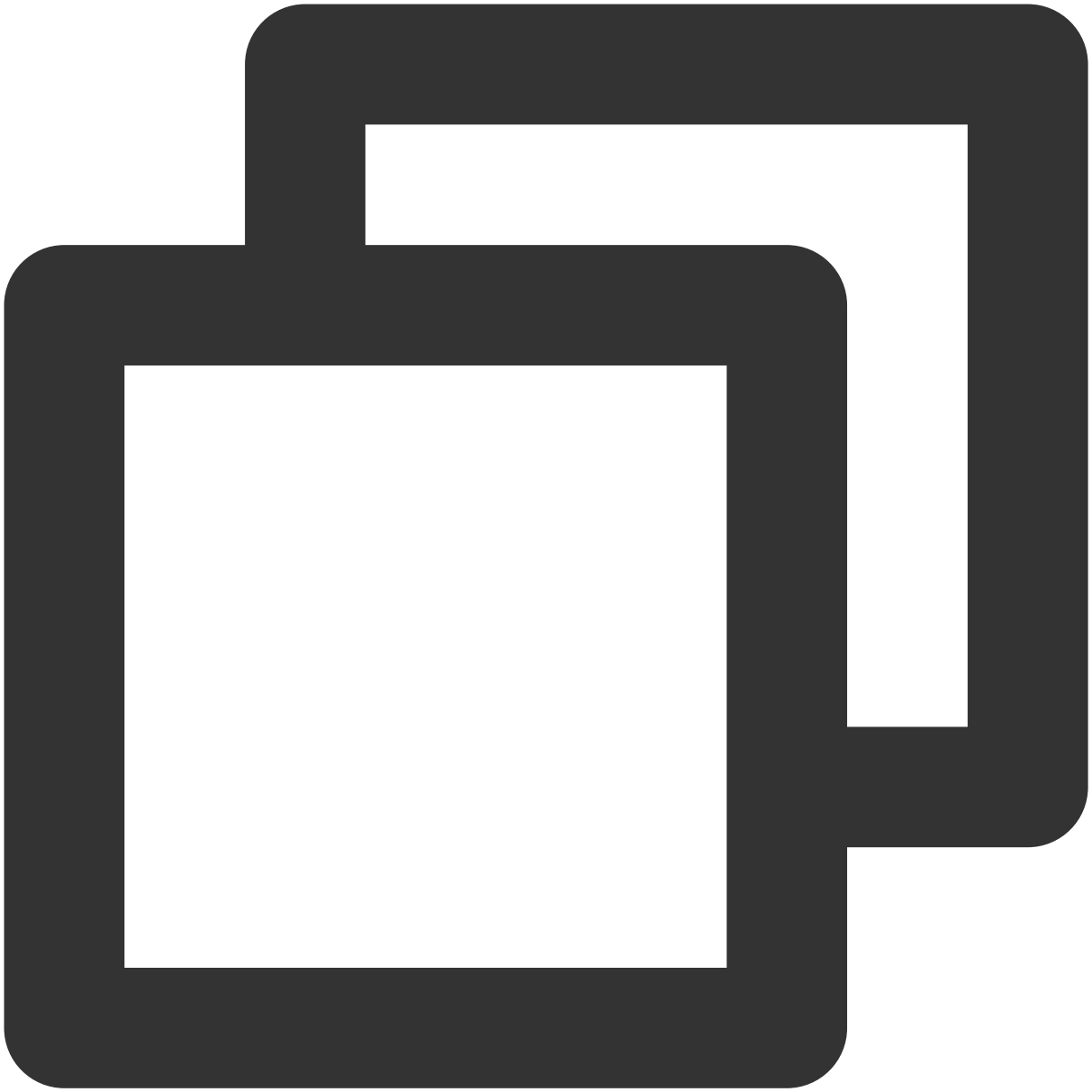


```
// 非分区表
CREATE TABLE IF NOT EXISTS `DataLakeCatalog`.`axitest`.`upsert_case` (`id` int, `na
TBLPROPERTIES (
    'format-version' = '2',
    'write.upsert.enabled' = 'true',
```

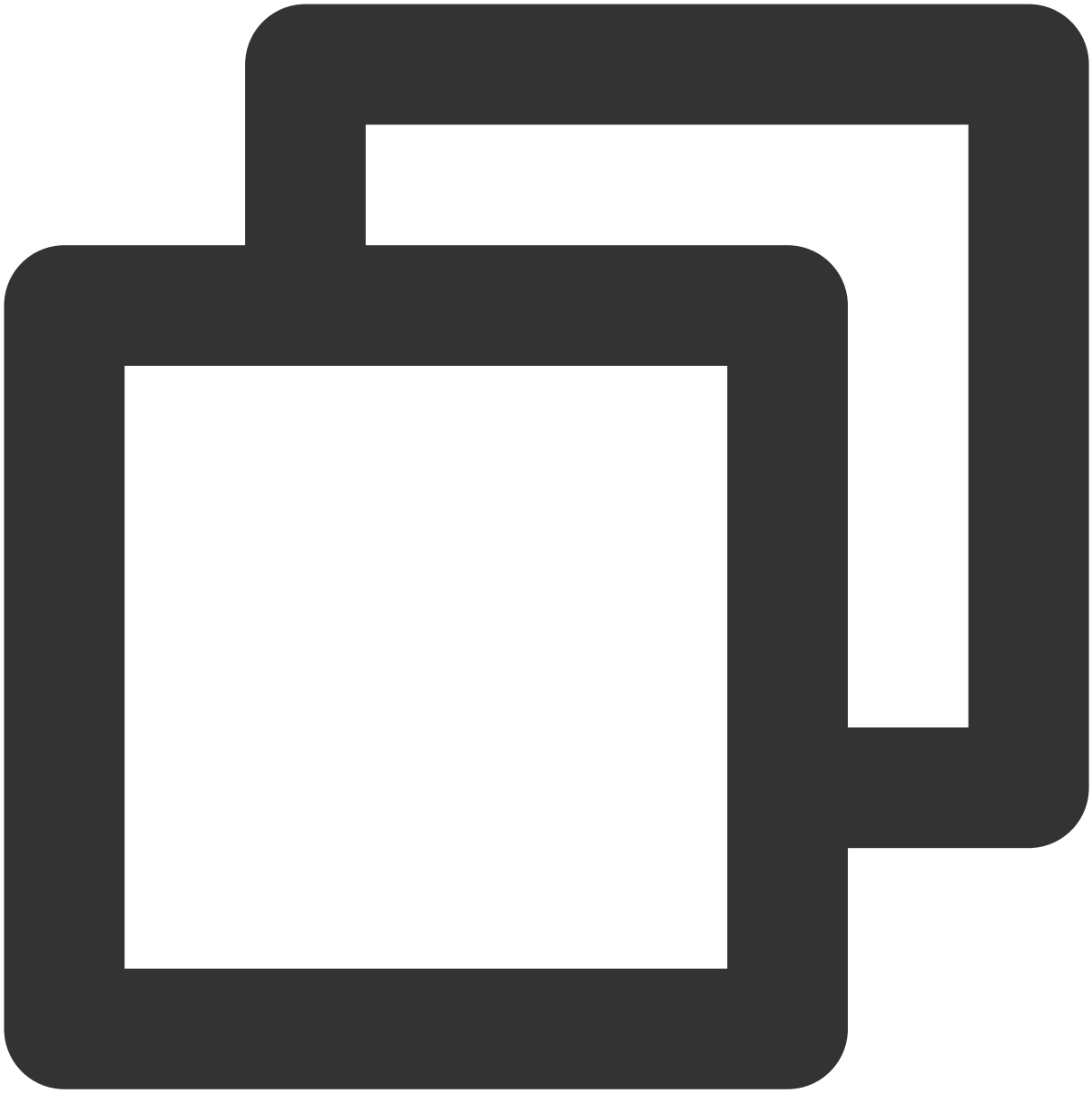
```
    'write.update.mode' = 'merge-on-read',
    'write.merge.mode' = 'merge-on-read',
    'write.parquet.bloom-filter-enabled.column.id' = 'true',
    'dlc.ao.data.govern.sorted.keys' = 'id',
    'write.distribution-mode' = 'hash',
    'write.metadata.delete-after-commit.enabled' = 'true',
    'write.metadata.previous-versions-max' = '100',
    'write.metadata.metrics.default' = 'full',
    'smart-optimizer.inherit' = 'default'
);
```

修改表属性

如果用户在创建表的时候没有携带相关属性值，可通过 alter table 将相关属性值进行修改、添加和移除，如下所示。如涉及到表属性值的变更都可以通过改方式。特别的 Iceberg format-version 字段不能修改，另外如果用户表已经有 inlong/oceans/flink 实时导入，修改后需要重启上游导入业务。



```
// 修改冲突重试次数为10

ALTER TABLE `DataLakeCatalog`.`axitest`.`upsert_case` SET TBLPROPERTIES('commit.ret
```



```
// 取消name字段bloom设置

ALTER TABLE `DataLakeCatalog`.`axitest`.`upsert_case` UNSET TBLPROPERTIES('write.pa
```

步骤三：数据优化与生命周期配置

数据优化与生命周期配置有两种方式。

1. 通过控制台界面可视化配置。

2. 通过 SQL 进行配置。

通过控制台界面配置

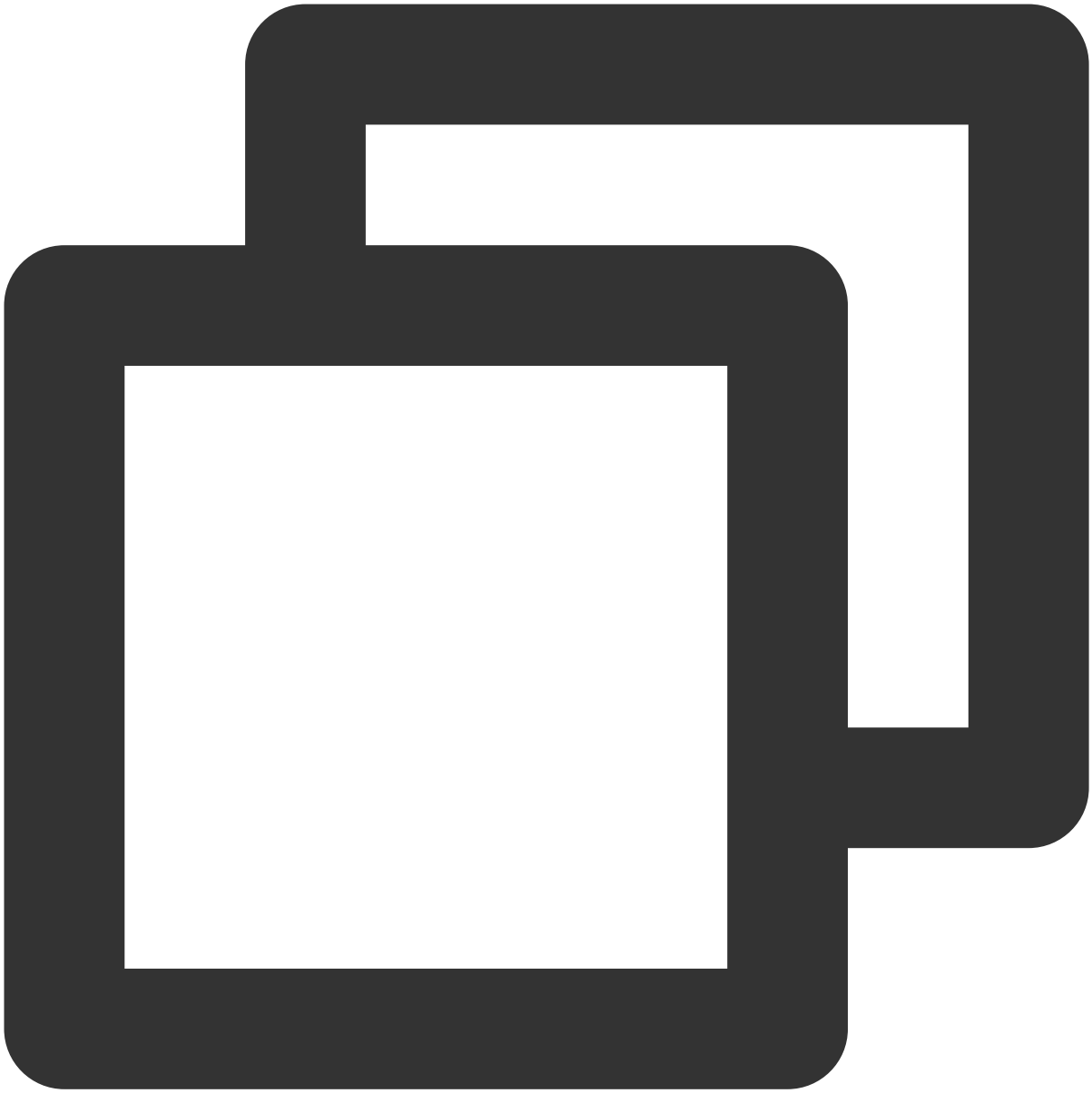
DLC 提供数据管理模块进行配置，具体操作请参见 开启数据优化。

通过 SQL 配置

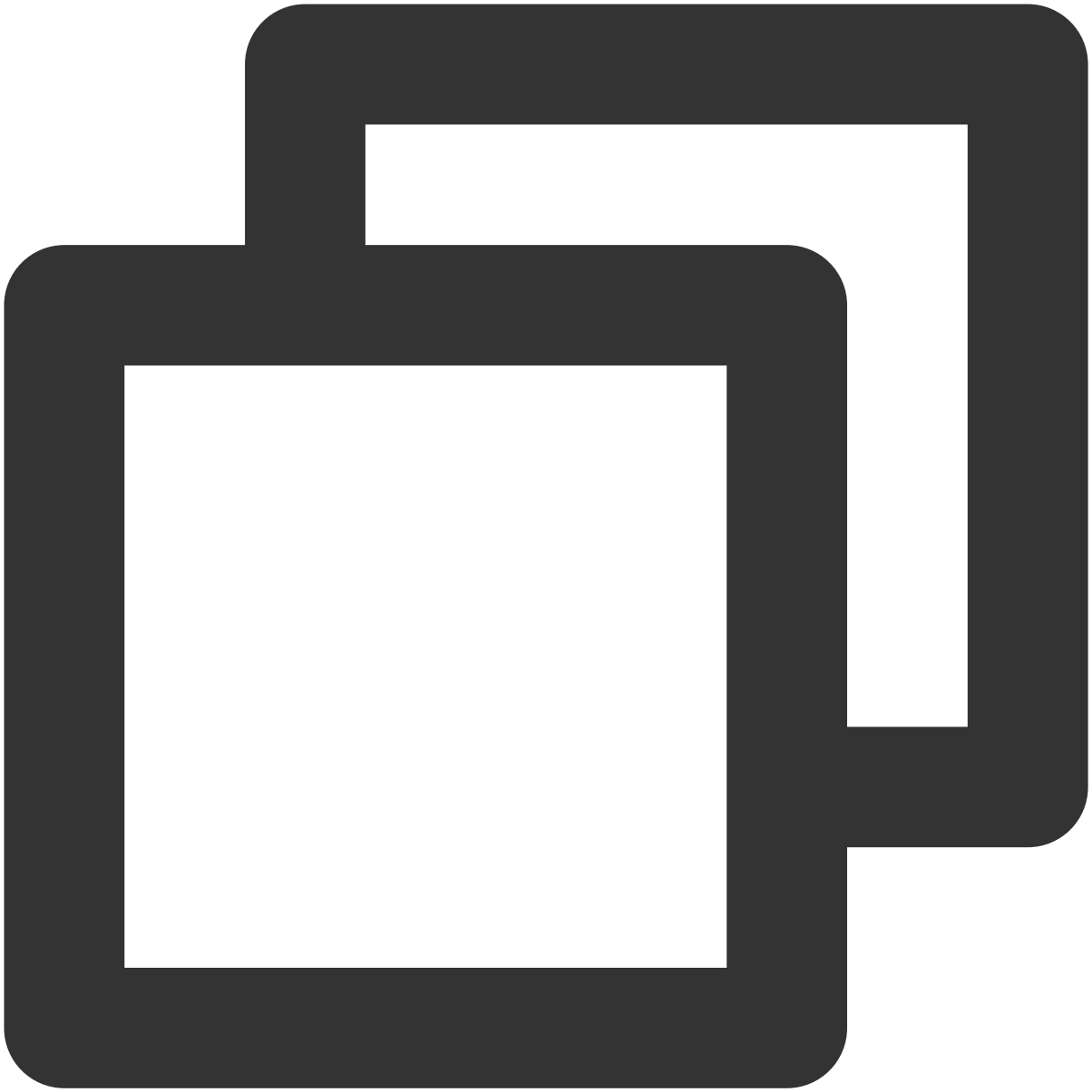
DLC 定义了详细的属性用于管理数据优化与生命周期，您可以结合业务特点灵活配置数据管理与生命周期，详细的数据优化和生命周期配置值请参见 开启数据优化。

配置数据库

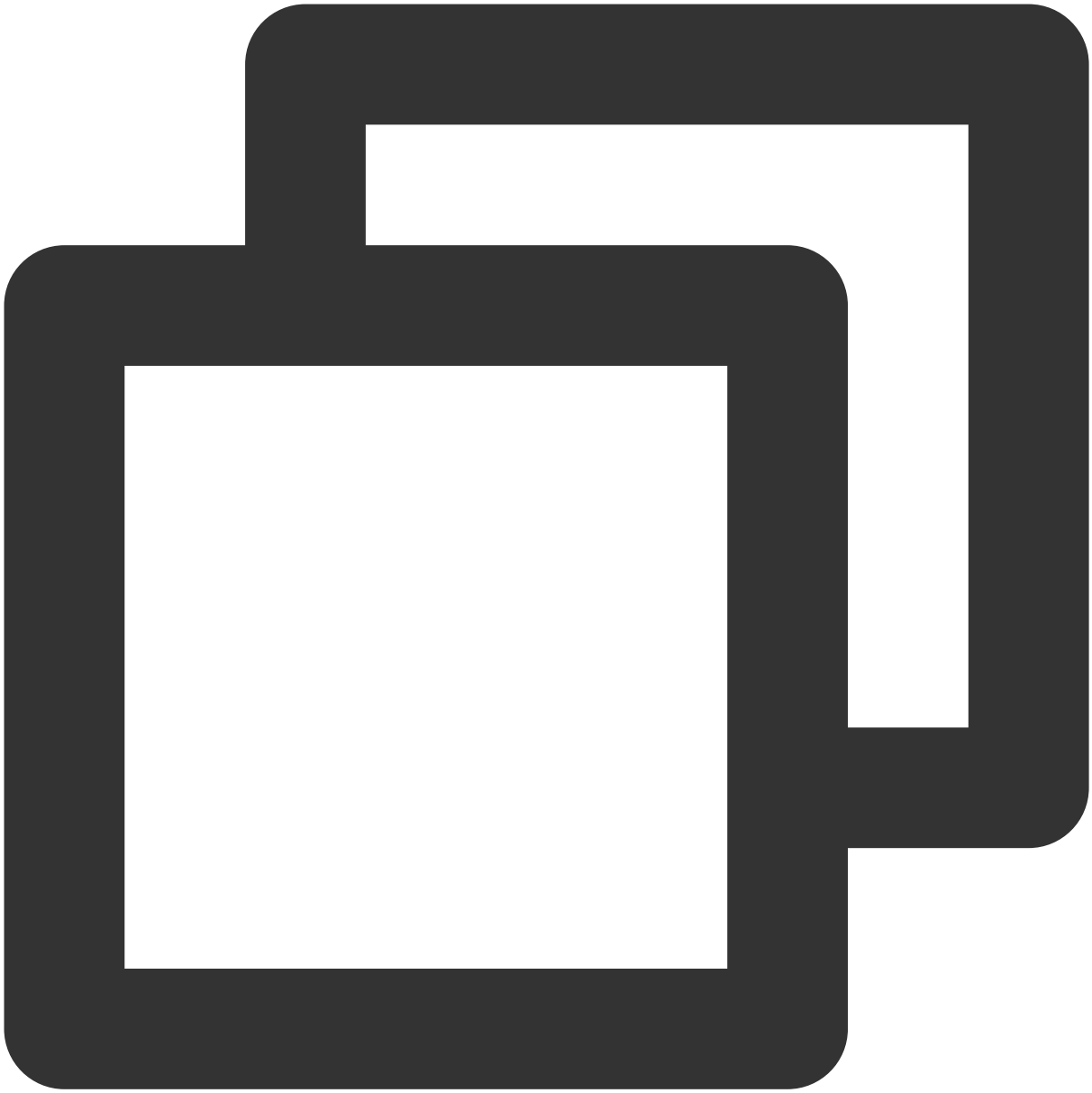
数据库的数据优化和生命周期可通过变更 DBPROPERTIES 进行，如下所示。



```
// 针对my_database表开启写入优化，且不继承数据目录策略

ALTER DATABASE DataLakeCatalog.my_database SET  DBPROPERTIES ('smart-optimizer.inhe
```



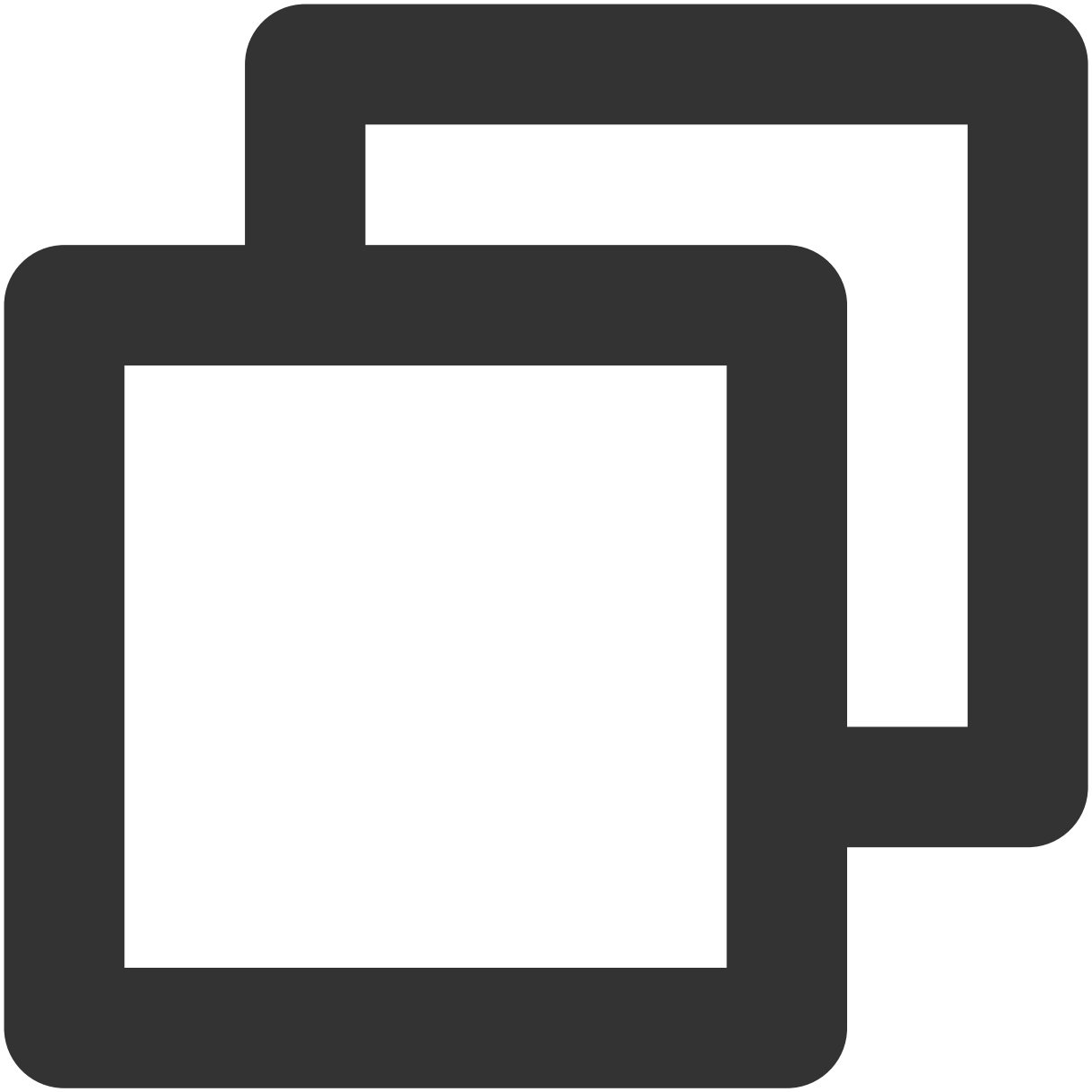
```
// 设置my_database继承数据目录策略

ALTER DATABASE DataLakeCatalog.my_database SET  DBPROPERTIES ('smart-optimizer.inhe
```



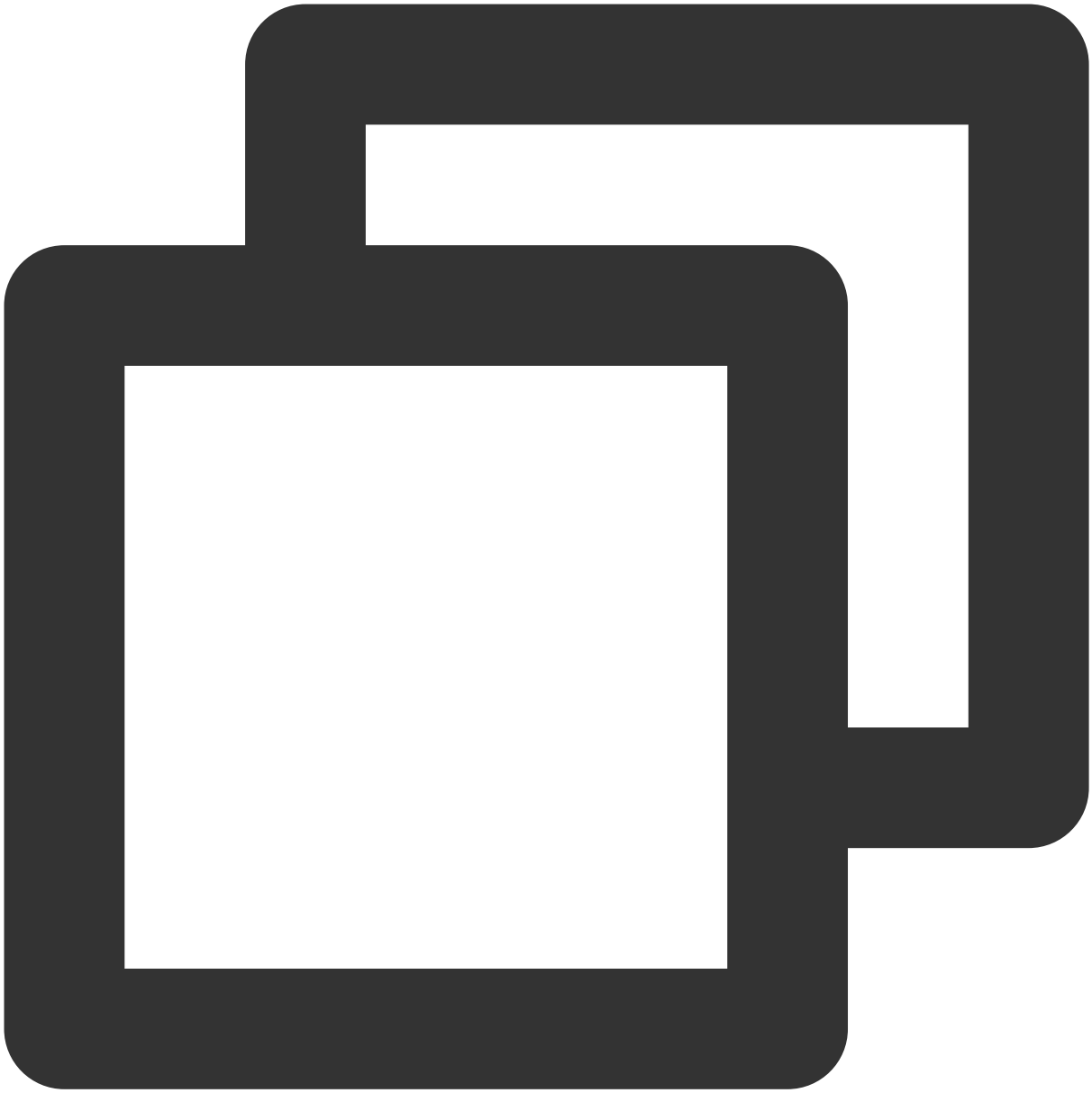
```
// 针对my_database表关闭生命周期，且不继承数据目录策略

ALTER DATABASE DataLakeCatalog.my_database SET  DBPROPERTIES ('smart-optimizer.inhe
```

配置数据表

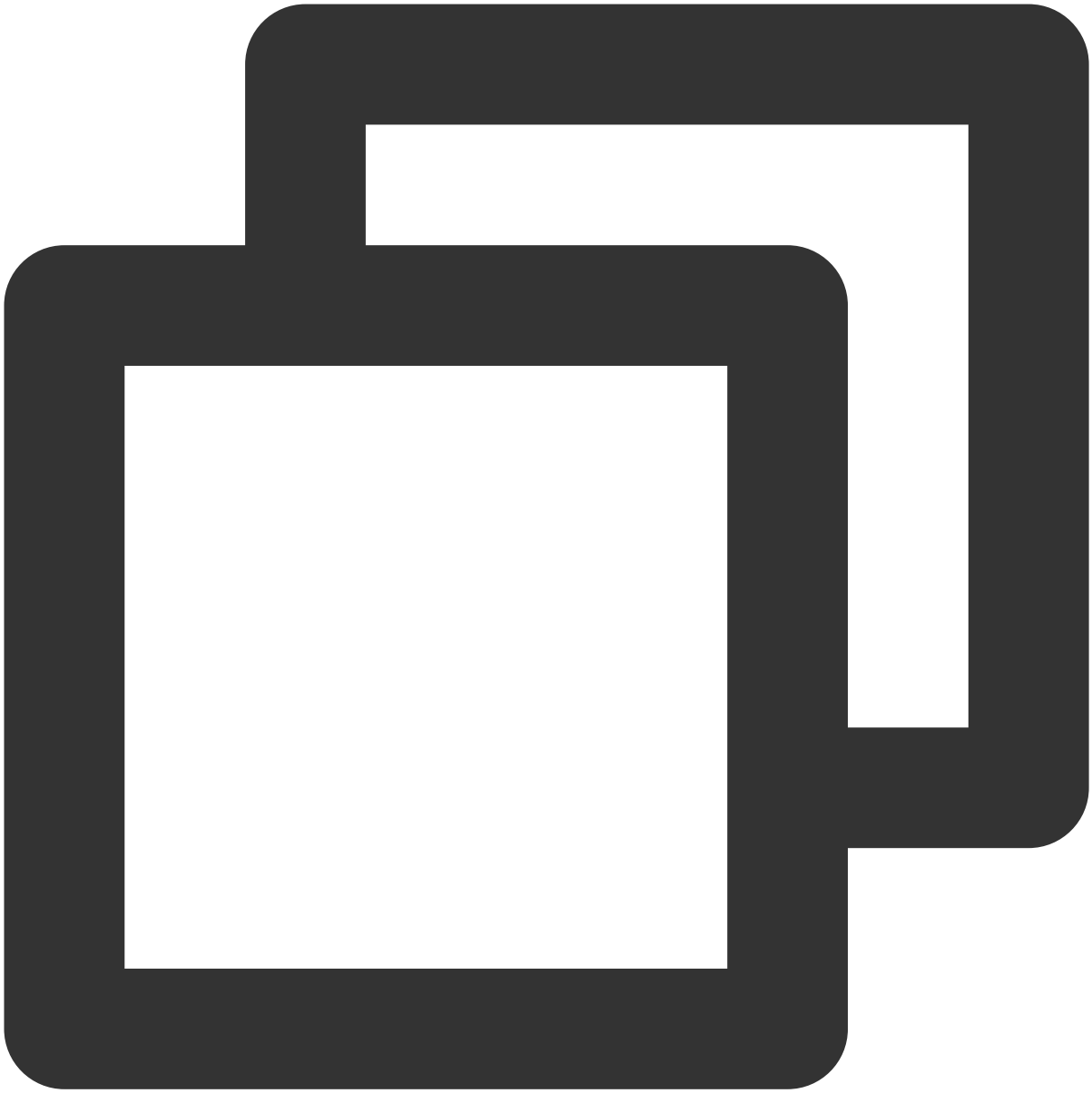
数据表的数据优化和生命周期通过变更 TBLPROPERTIES 进行，如下所示。



```
// 针对upsert_cast表关闭写入优化，且不继承数据库策略

ALTER TABLE `DataLakeCatalog`.`axitest`.`upsert_case` SET TBLPROPERTIES('smart-opti
```



```
// 设置upsert_cast表继承数据库策略

ALTER TABLE `DataLakeCatalog`.`axitest`.`upsert_case` SET TBLPROPERTIES('smart-opti
```



```
// 针对upsert_cast表开启生命周期并设置生命周期时间为7天，且不继承数据库策略

ALTER TABLE `DataLakeCatalog`.`axitest`.`upsert_case` SET TBLPROPERTIES('smart-opti
```

步骤四：数据入湖到原生表

DLC 原生表支持多种多种方式的数据写入，结合您的数据写入方式，具体操作请参见 DLC 原生表入湖实践。

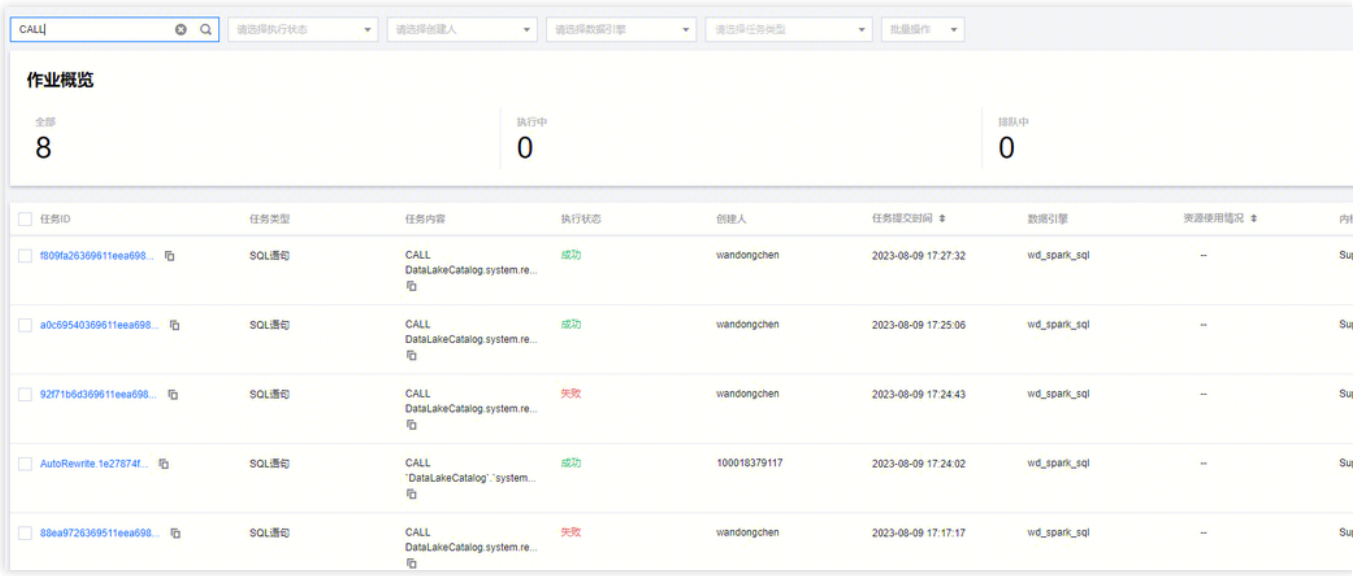
步骤五：查看数据优化任务

您可以在 DLC 控制台**数据运维**菜单，进入**历史任务**页面查看数据治理任务。可以"CALL"、"Auto"、库名称和表名称等关键字查询任务。

说明：

查看系统数据优化任务的用户需要具备 DLC 管理员权限。

任务 ID 为 Auto 开头的任务都是自动产生的数据优化任务。如下图所示。



您也可以点击**查看详情**，查询任务的基本信息和运行结果。


运行详情
基本信息
运行结果
查询统计
任务ID
AutoRewrite.1e27874f-0118-448f-b31d-127d59d6f545
任务类型
SQL语句
查询语句
CALL `DataLakeCatalog`.`system`.`rewrite_data_files`(
`table` => 'ods.aps_test_20230809',
`options` => map(
'delete-file-threshold',
'1',
'max-concurrent-file-group-rewrites',
'20',
'min-input-files',
'5',
'target-file-size-bytes',
'134217728'
)
)
执行状态
成功
创建人
100018379117
任务提交时间
2023-08-09 17:24:02
内核版本
SuperSQL-S 1.0
任务结束时间
2023-08-09 17:24:42
数据引擎
wd_spark_sql
任务耗时
4.6s
数据扫描量
0B
数据条数
1条

DLC 原生表入湖实践

最近更新时间：2024-07-31 17:35:06

应用场景

CDC（Change Data Capture）是变更数据捕获的缩写，可以将源数据库中的增量变更近似实时同步到其他数据库或应用程序。DLC 支持通过 CDC 技术将源数据库的增量变更同步到 DLC 原生表，完成源数据入湖。

前置条件

正确开通 DLC，已完成用户权限配置，开通托管存储。
正确创建 DLC 数据库。
正确配置 DLC 数据库数据优化，详细配置请参考开启数据优化。

InLong 数据入湖

通过 DataInLong 可将源数据同步到 DLC。

Oceanus 流计算数据入湖

通过 Oceanus 可将源数据同步到 DLC。

自建 Flink 数据入湖

通过 Flink 可将源数据同步到 DLC。本示例展示将源 Kafka 的数据同步到 DLC，完成数据入湖。

环境准备

依赖集群：Kafka 2.4.x，Flink 1.15.x, Hadoop3.x。
Kafka、Flink 集群建议购买 EMR 集群。

整体操作过程

详细操作流程可参考如下图：

步骤1：上传依赖 Jar ：上传同步所需的 Kakfa、DLC 连接 Jar 包和 Hadoop 相关依赖 Jar。

步骤2：创建 Kafka Topic：创建 Kafka 生产消费的 Topic。

步骤3：DLC 新建目标表：DLC 数据管理新建目标表。

步骤4：提交任务：Flink 集群下提交同步任务。

步骤5：发送消息数据和查询同步结果：Kafka 集群发送消息数据和 DLC 上查看数据同步结果。

步骤1：上传依赖 Jar

1. 下载依赖 Jar

相关依赖 Jar 建议上传与Flink对应版本的Jar，例如 Flink 为 Flink1.15.x，则建议下载 flink-sql-connect-kafka-1.15.x.jar。相关文件参考附件。

Kafka 相关依赖：flink-sql-connect-kafka-1.15.4.jar

DLC 相关依赖：sort-connector-iceberg-dlc-1.6.0.jar

Hadoop3.x 相关依赖：api-util-1.0.0-M20.jar、guava-27.0-jre.jar、hadoop-mapreduce-client-core-3.2.2.jar。

2. 登录 Flink 集群，将准备好的 Jar 上传到 flink/ib 目录下。

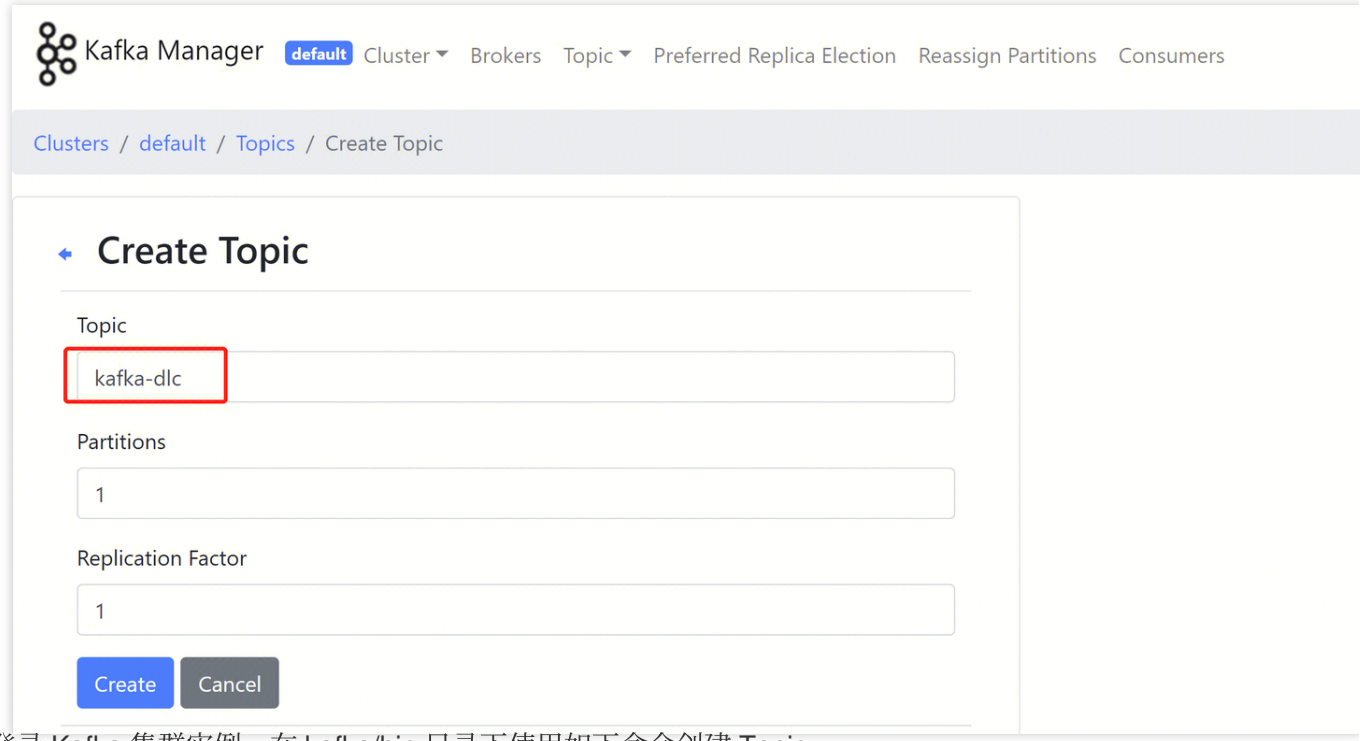
步骤2：创建 Kafka Topic

登录 Kafka Manager，单击 **default 集群**，单击 **Topic > Create**。

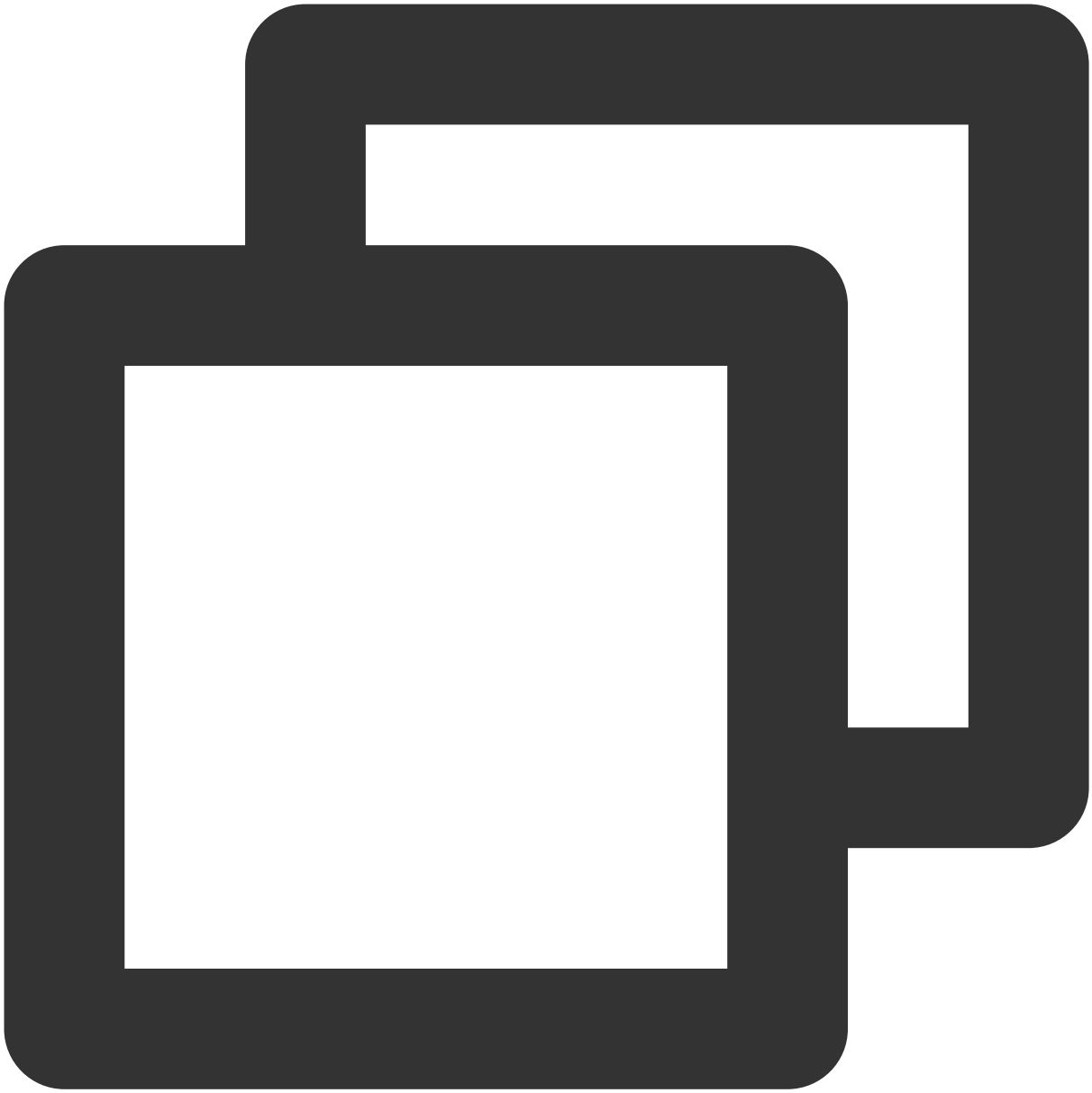
Topic 名称：本示例输入为 kafka_dlc

分区数：1

副本数：1



或者登录 Kafka 集群实例，在 kafka/bin 目录下使用如下命令创建 Topic。



```
./kafka-topics.sh --bootstrap-server ip:port --create  --topic kafka-dlc
```

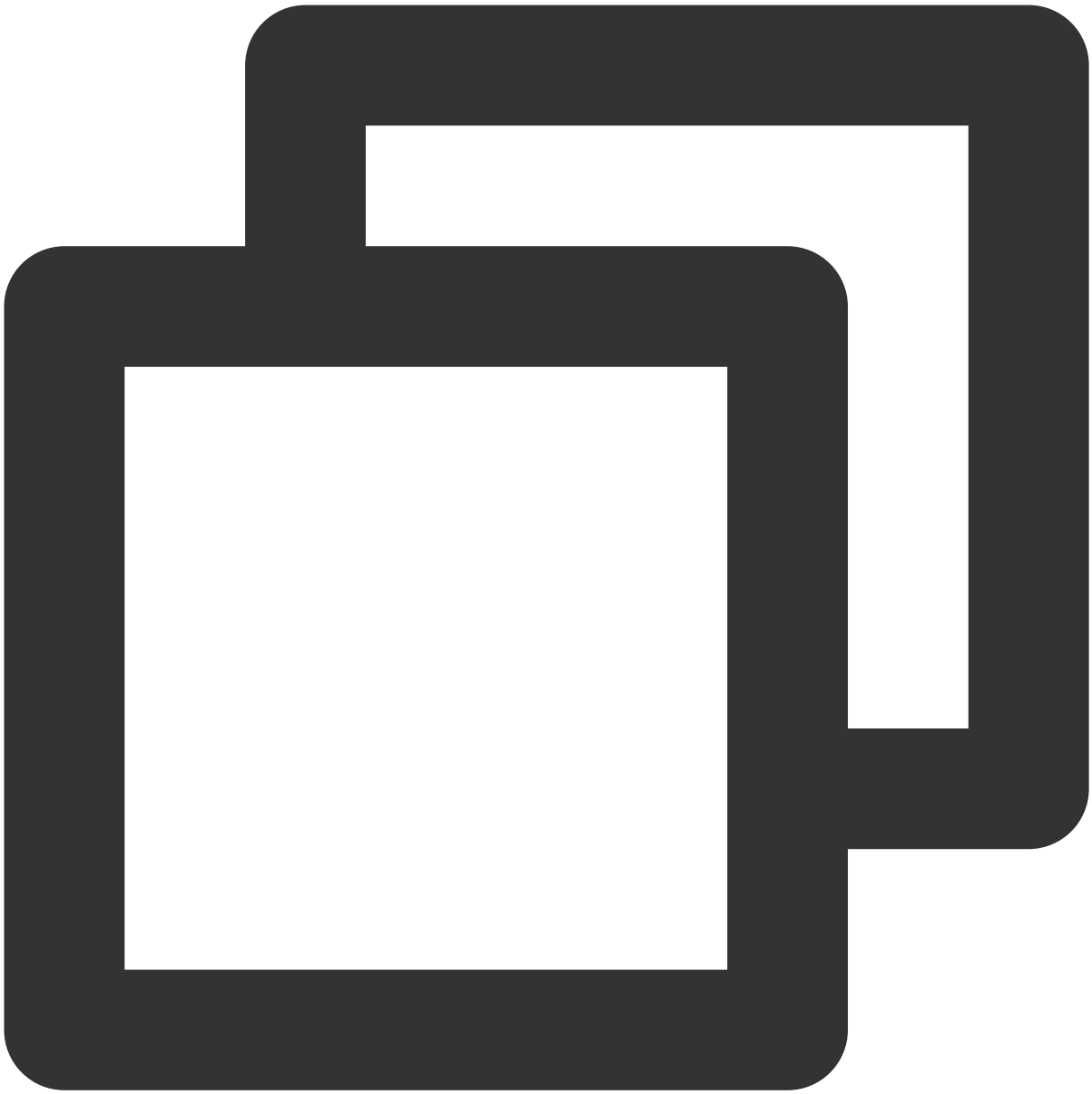
步骤3：DLC 新建目标表

新建目标表详情可参考 DLC 原生表操作配置。

步骤4：提交任务

Flink 同步数据的方式有2种，Flink SQL写入模式 和 Flink Stream API，以下会介绍2种同步方式。
提交任务前，需要新建保存 checkpoint 数据的目录，通过如下命令新建数据目录。

新建 hdfs /flink/checkpoints 目录：

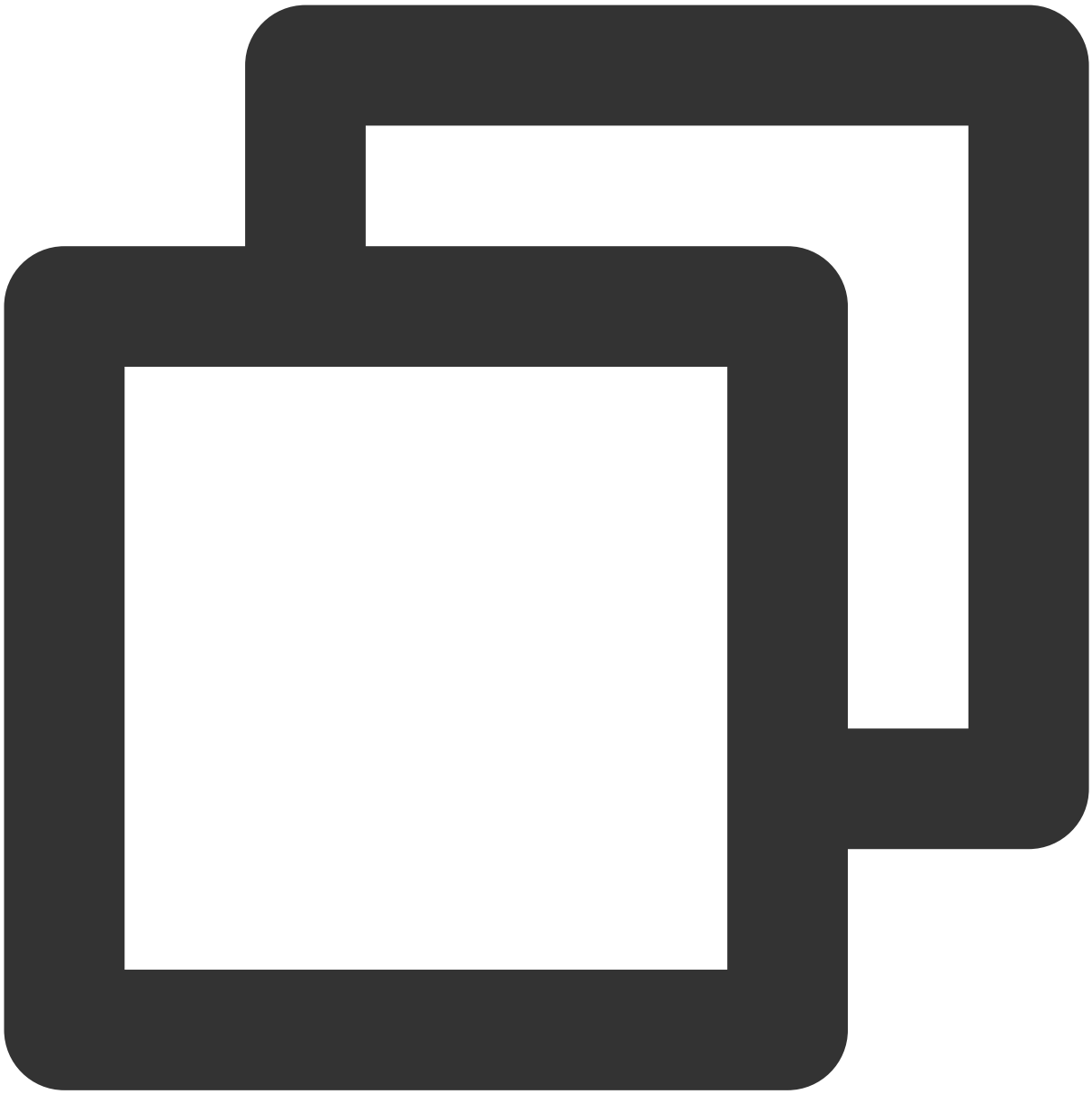


```
hadoop fs -mkdir /flink
hadoop fs -mkdir /flink/checkpoints
```

Flink SQL 同步模式

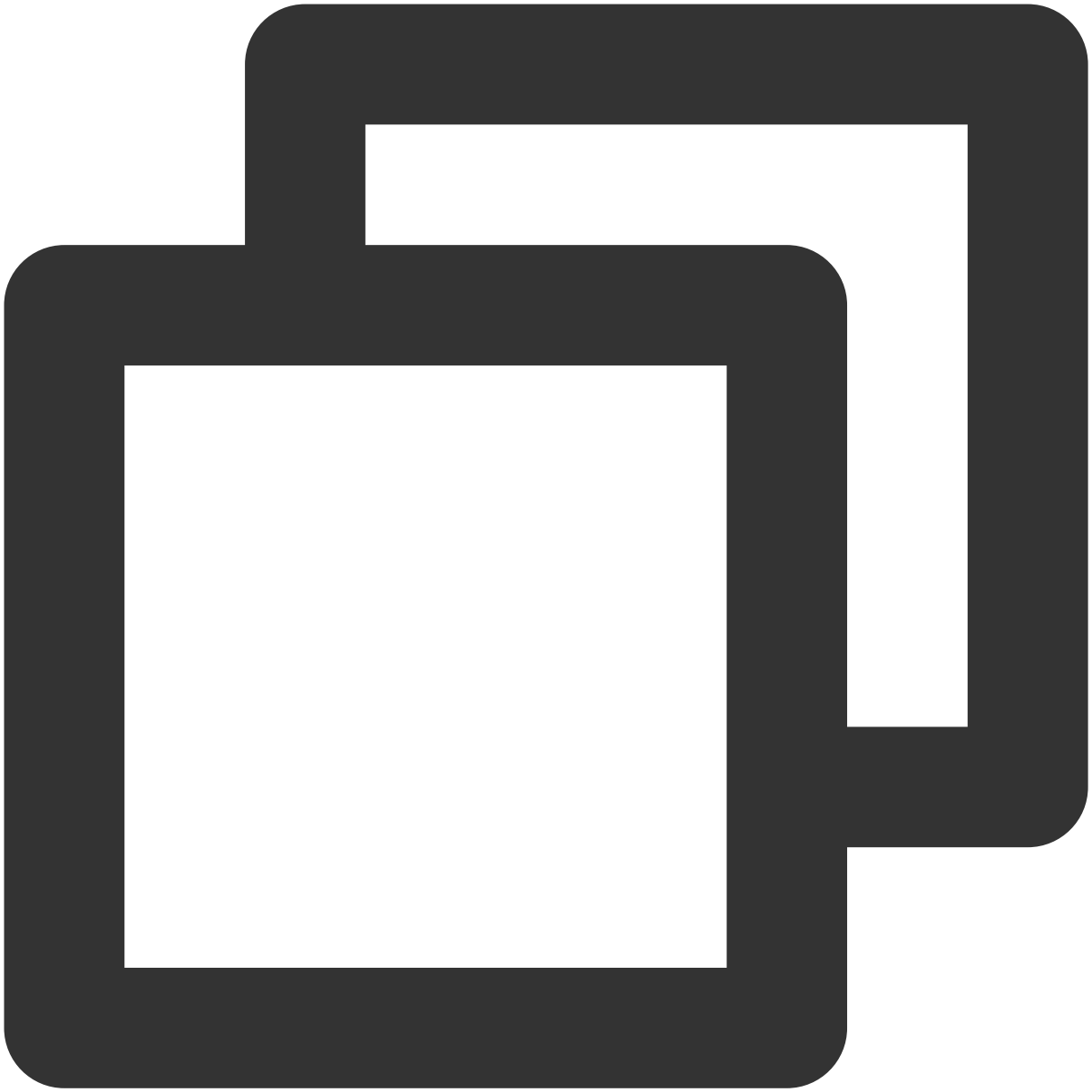
1. 通过 IntelliJ IDEA 新建一个名称为“flink-demo”的 Maven 项目。
2. 在 pom 中添加相关依赖，依赖详情请参考 完整样例代码参考 > 示例1。
3. Java 同步代码：核心代码如下步骤展示，详细代码请参考 完整样例代码参考 > 示例2。

创建执行环境和配置 checkpoint：



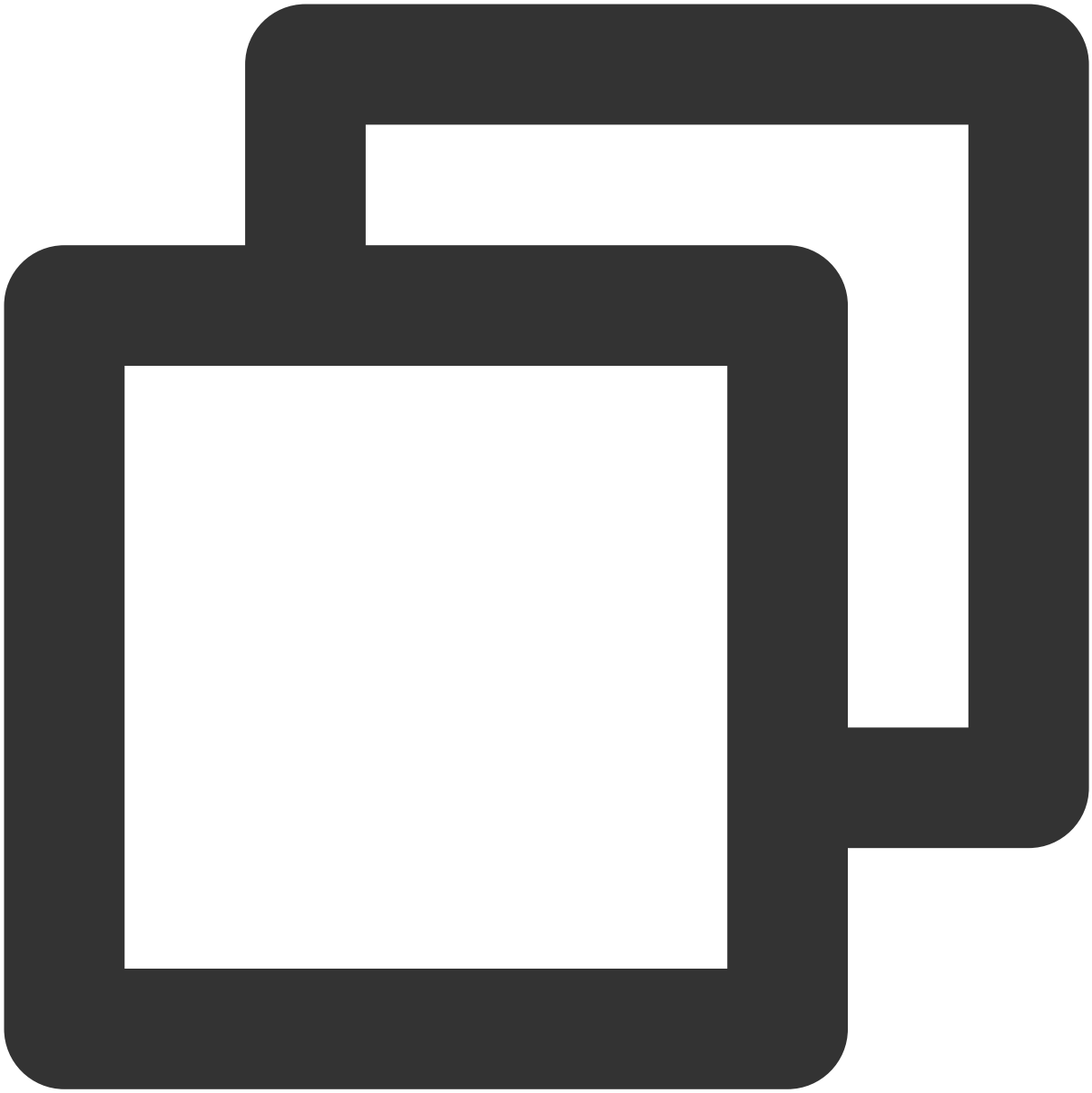
```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment
env.setParallelism(1);
env.enableCheckpointing(60000);
env.getCheckpointConfig().setCheckpointStorage("hdfs:///flink/checkpoints");
env.getCheckpointConfig().setCheckpointTimeout(60000);
env.getCheckpointConfig().setTolerableCheckpointFailureNumber(5);
env.getCheckpointConfig().enableExternalizedCheckpoints(CheckpointConfig.Externaliz
```

执行 Source SQL：



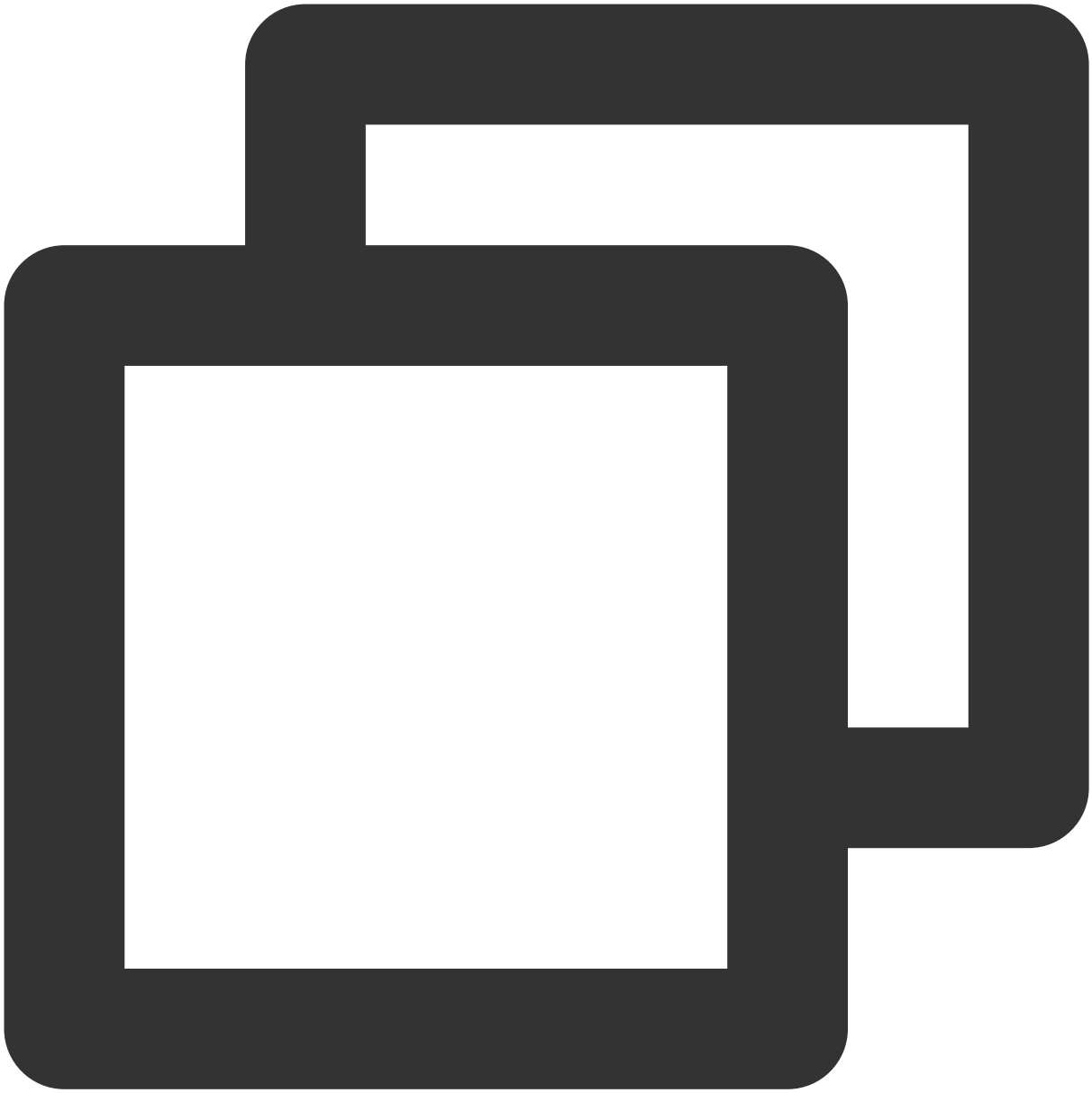
```
tEnv.executeSql(sourceSql);
```

执行 同步 SQL：



```
tEnv.executeSql(sql)
```

4. 通过 IntelliJ IDEA 对 flink-demo 项目编译打包，在项目 target 文件夹下生成 JAR 包 flink-demo-1.0-SNAPSHOT.jar。
5. 登录 Flink 集群其中的一个实例，上传 flink-demo-1.0-SNAPSHOT.jar 到 /data/jars/ 目录（没有目录则新建）。
6. 登录 Flink 集群其中的一个实例，在 flink/bin 目录下执行如下命提交同步任务。

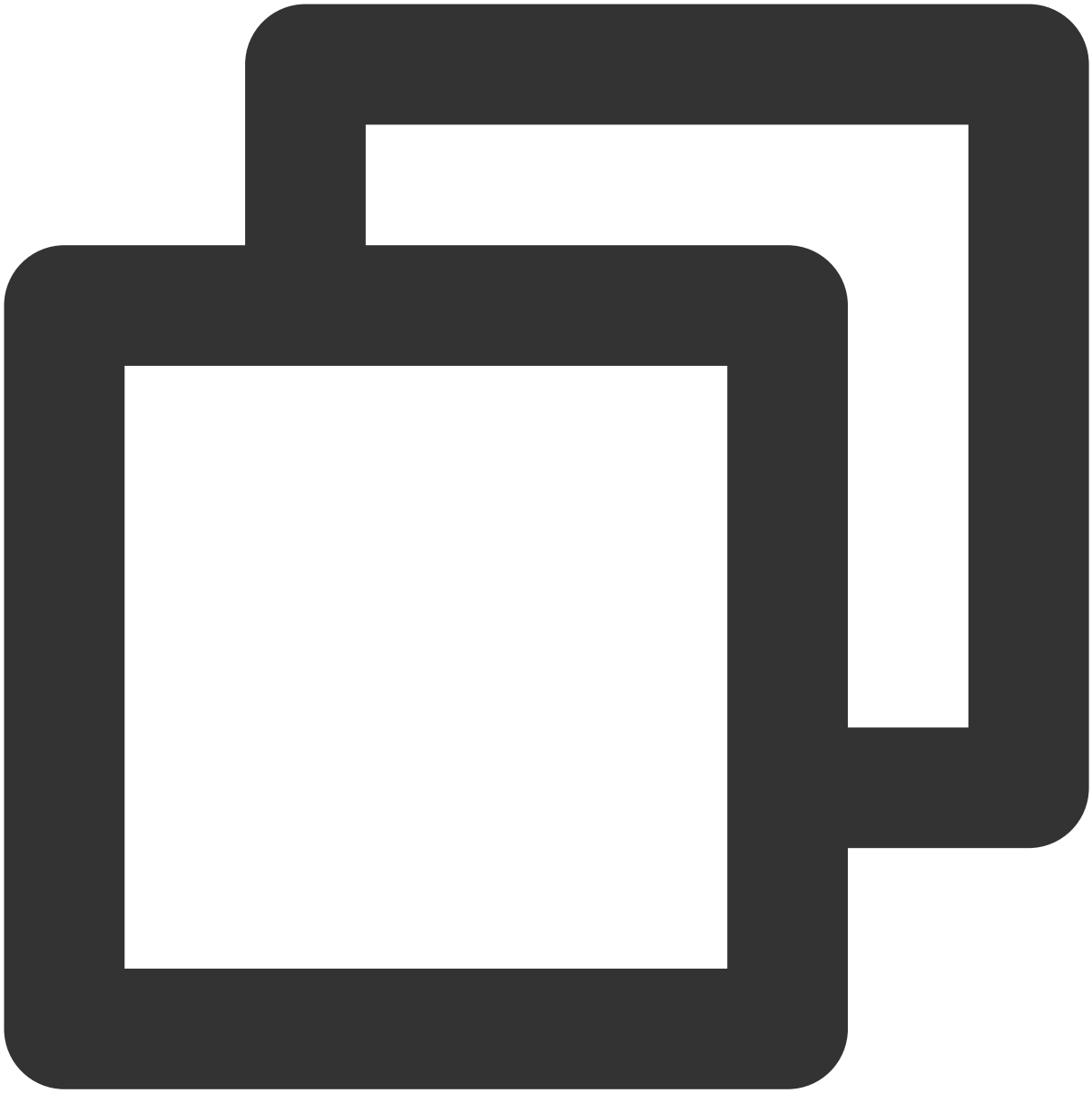


```
./flink run  --class com.tencent.dlc.iceberg.flink.AppendIceberg /data/jars/flink-d
```

Flink Stream API 同步模式

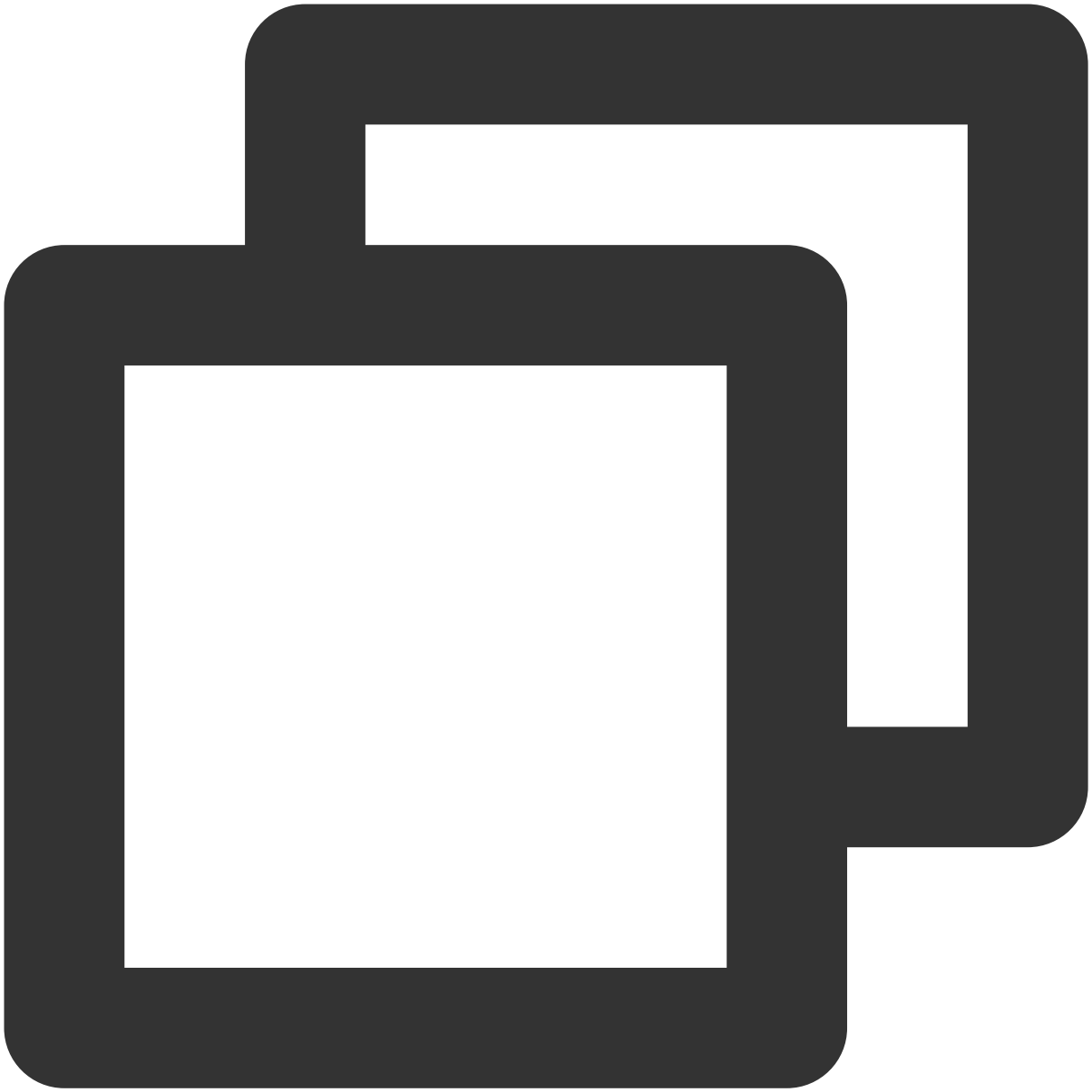
1. 通过 IntelliJ IDEA 新建一个名称为“flink-demo”的 Maven 项目。
2. 在 pom 中添加相关依赖：完整样例代码参考> 示例3。
3. Java 核心代码如下步骤展示，详细代码请参考 完整样例代码参考 > 示例4。

创建执行环境 StreamTableEnvironment，配置 checkpoint：



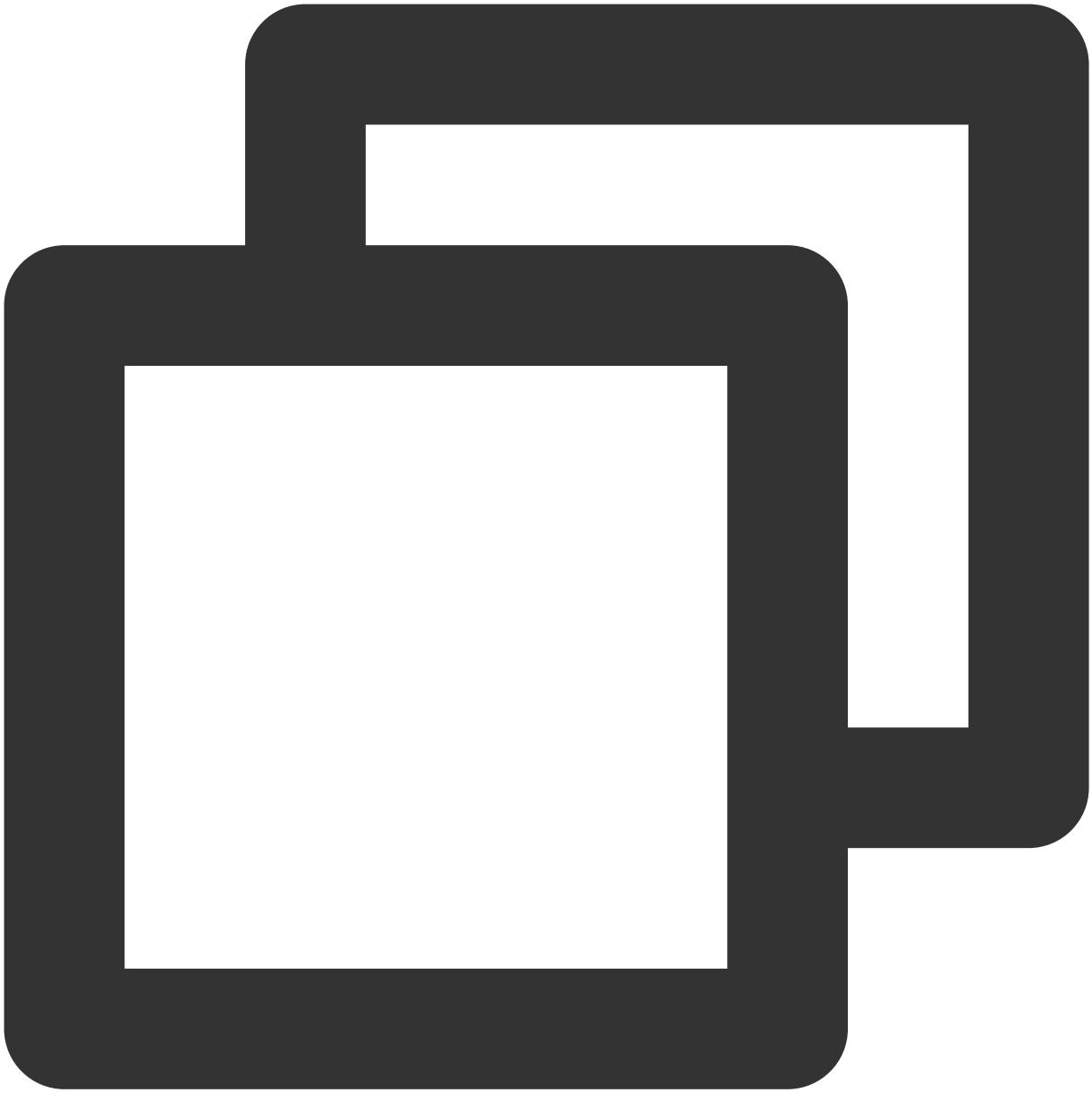
```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment
env.setParallelism(1);
env.enableCheckpointing(60000);
env.getCheckpointConfig().setCheckpointStorage("hdfs:///data/checkpoints");
env.getCheckpointConfig().setCheckpointTimeout(60000);
env.getCheckpointConfig().setTolerableCheckpointFailureNumber(5);
env.getCheckpointConfig().enableExternalizedCheckpoints(CheckpointConfig.Externaliz
```

获取 Kafka 输入流：



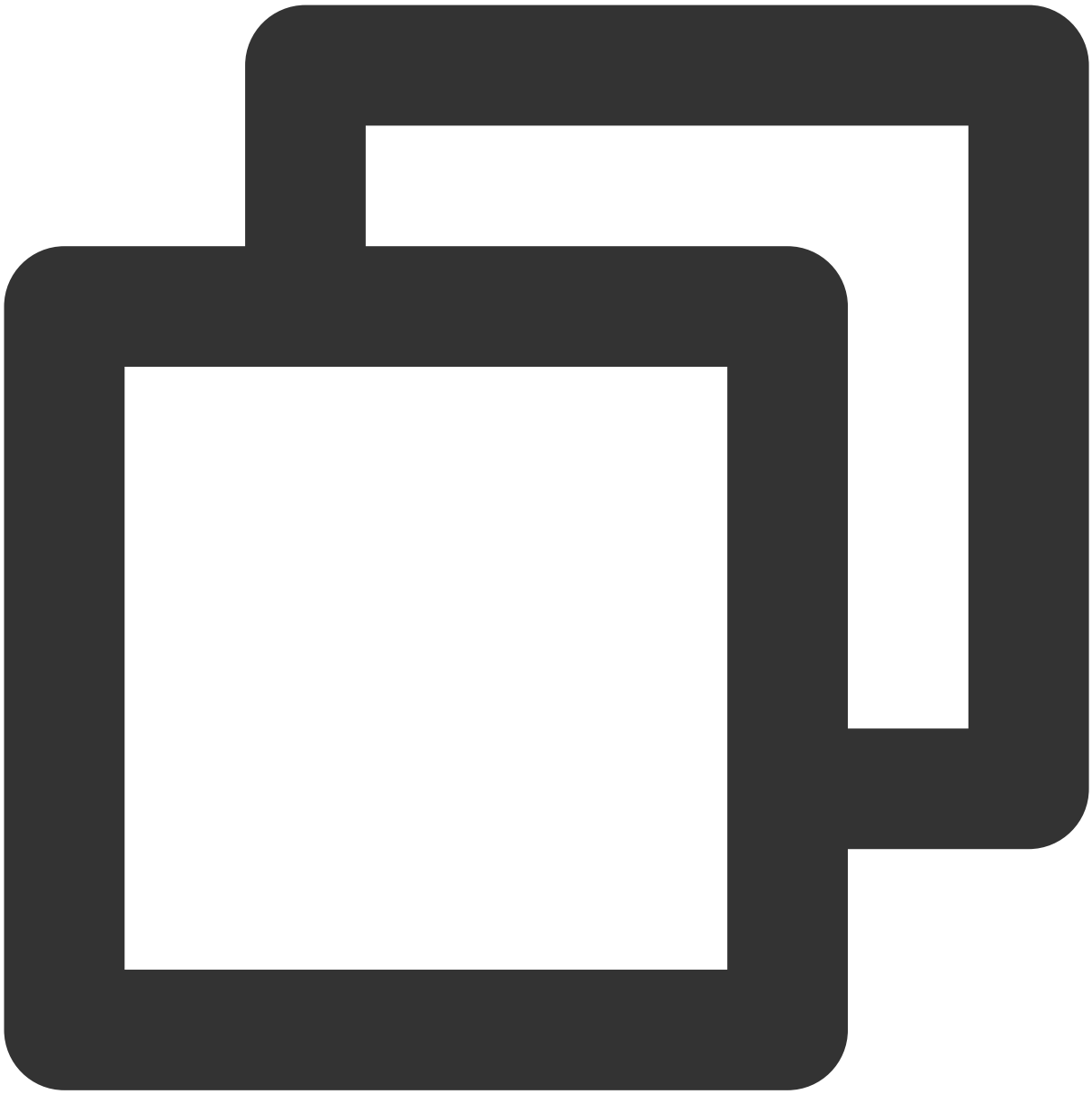
```
KafkaToDLC dlcSink = new KafkaToDLC();
DataStream<RowData> dataStreamSource = dlcSink.buildInputStream(env);
```

配置 Sink：



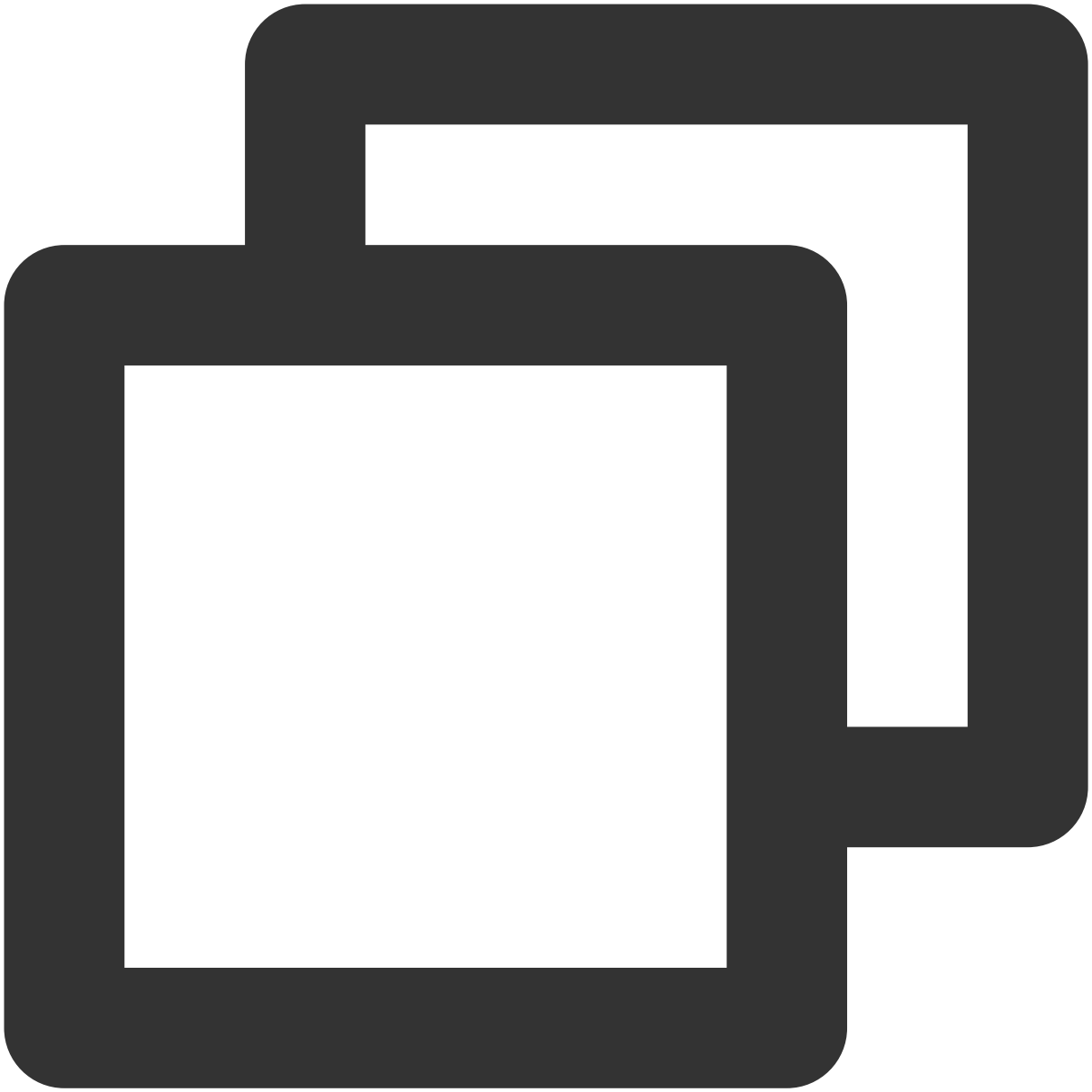
```
FlinkSink.forRowData(dataStreamSource)
          .table(table)
          .tableLoader(tableLoader)
          .equalityFieldColumns(equalityColumns)
          .metric(params.get(INLONG_METRIC.key()), params.get(INLONG_AUDIT.key()))
          .action(actionsProvider)
          .tableOptions(Configuration.fromMap(options))
          //默认为false,追加数据。如果设置为true 就是覆盖数据
          .overwrite(false)
          .append();
```

执行同步 SQL：



```
env.execute("DataStream Api Write Data To Iceberg");
```

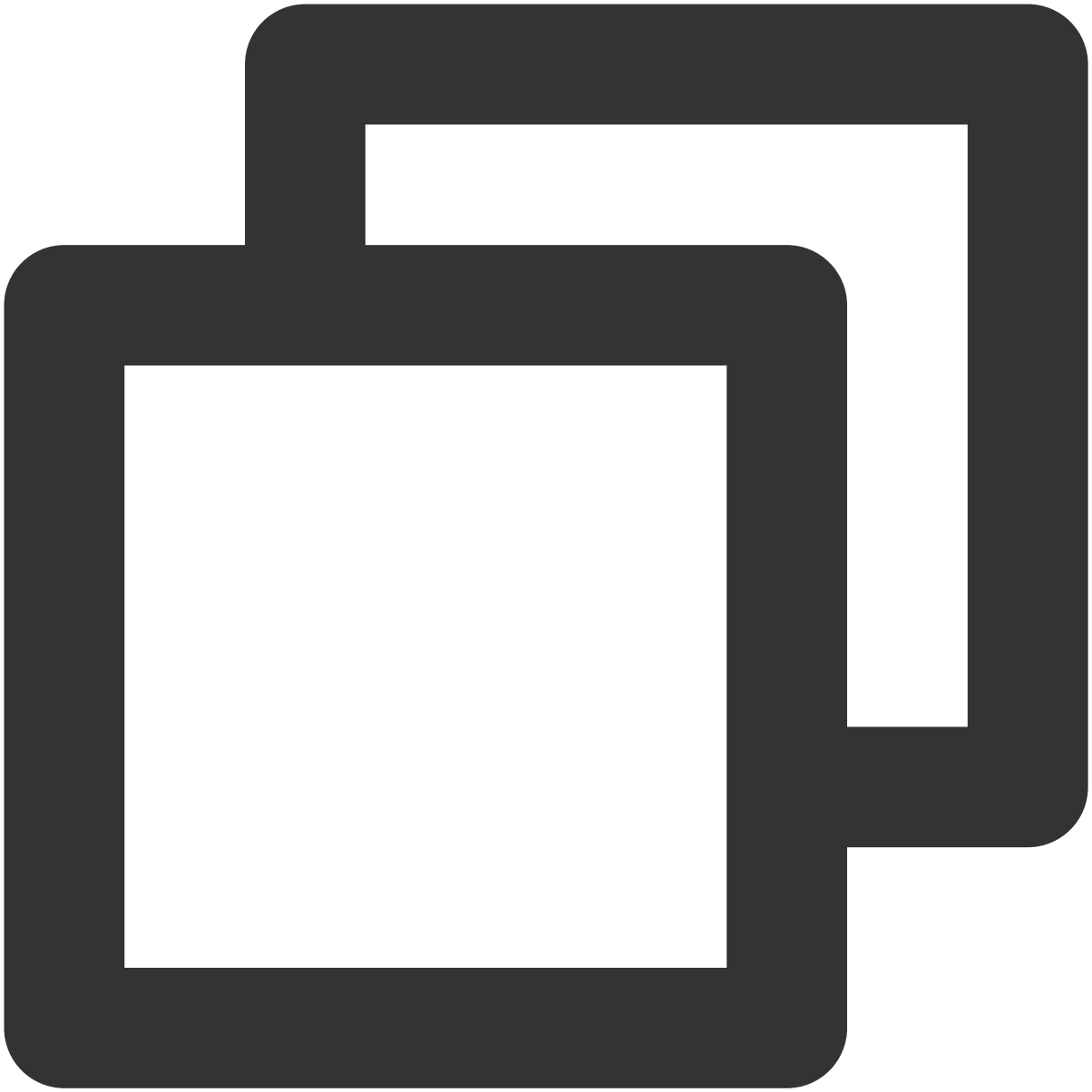
4. 通过 IntelliJ IDEA 对 flink-demo 项目编译打包，在项目 target 文件夹下生成 JAR 包 flink-demo-1.0-SNAPSHOT.jar。
5. 登录 Flink 集群其中的一个实例，上传 flink-demo-1.0-SNAPSHOT.jar 到 /data/jars/ 目录（没有目录则新建）。
6. 登录 Flink 集群其中的一个实例，在 flink/bin 目录下执行如下命令提交任务。



```
./flink run  --class com.tencent.dlc.iceberg.flink.AppendIceberg /data/jars/flink-d
```

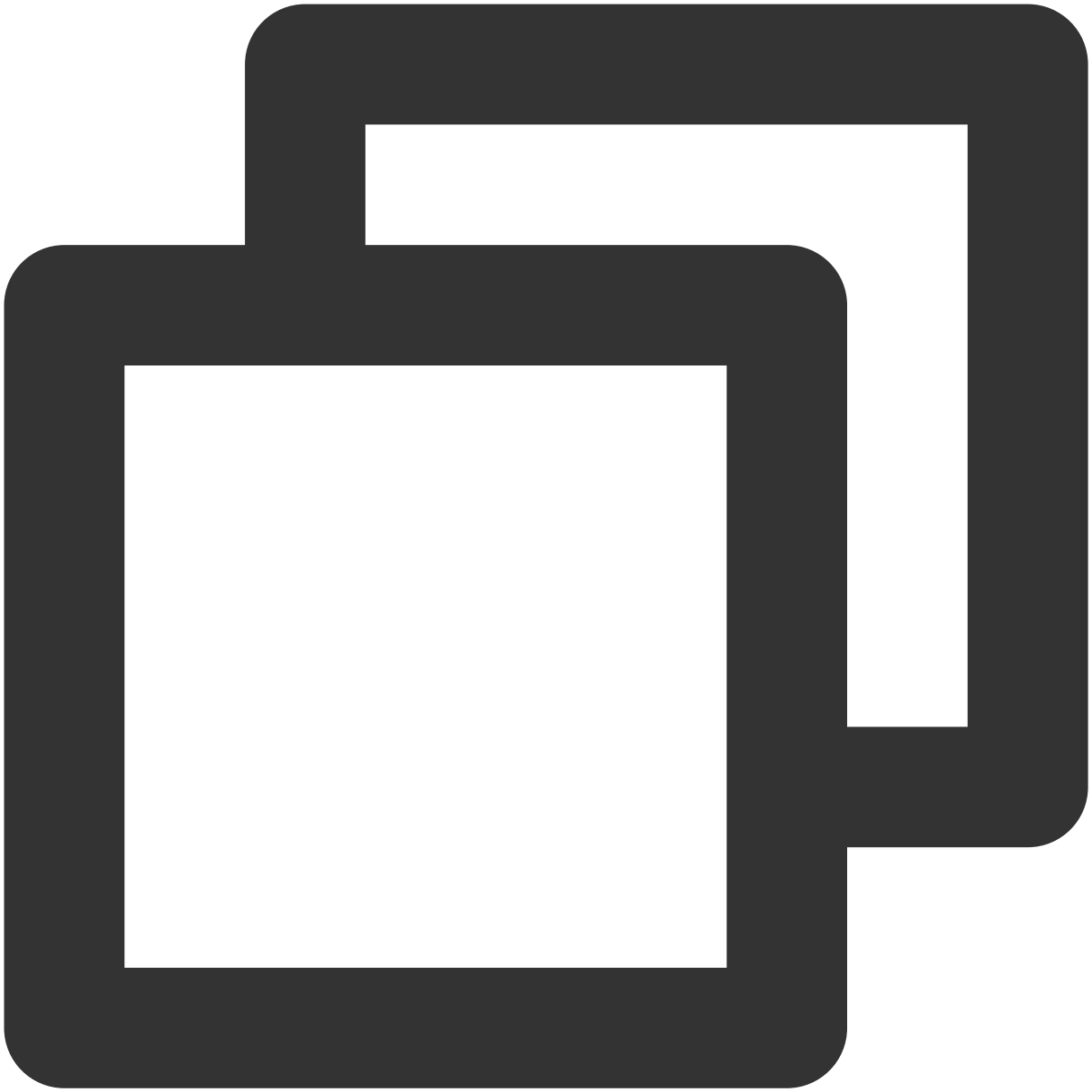
步骤5：发送消息数据和查询同步结果

1. 登录 Kafka 集群实例，在 kafka/bin 目录 用如下命令，发送消息数据。



```
./kafka-console-producer.sh --broker-list 122.152.227.141:9092 --topic kafka-dlc
```

数据信息如下：



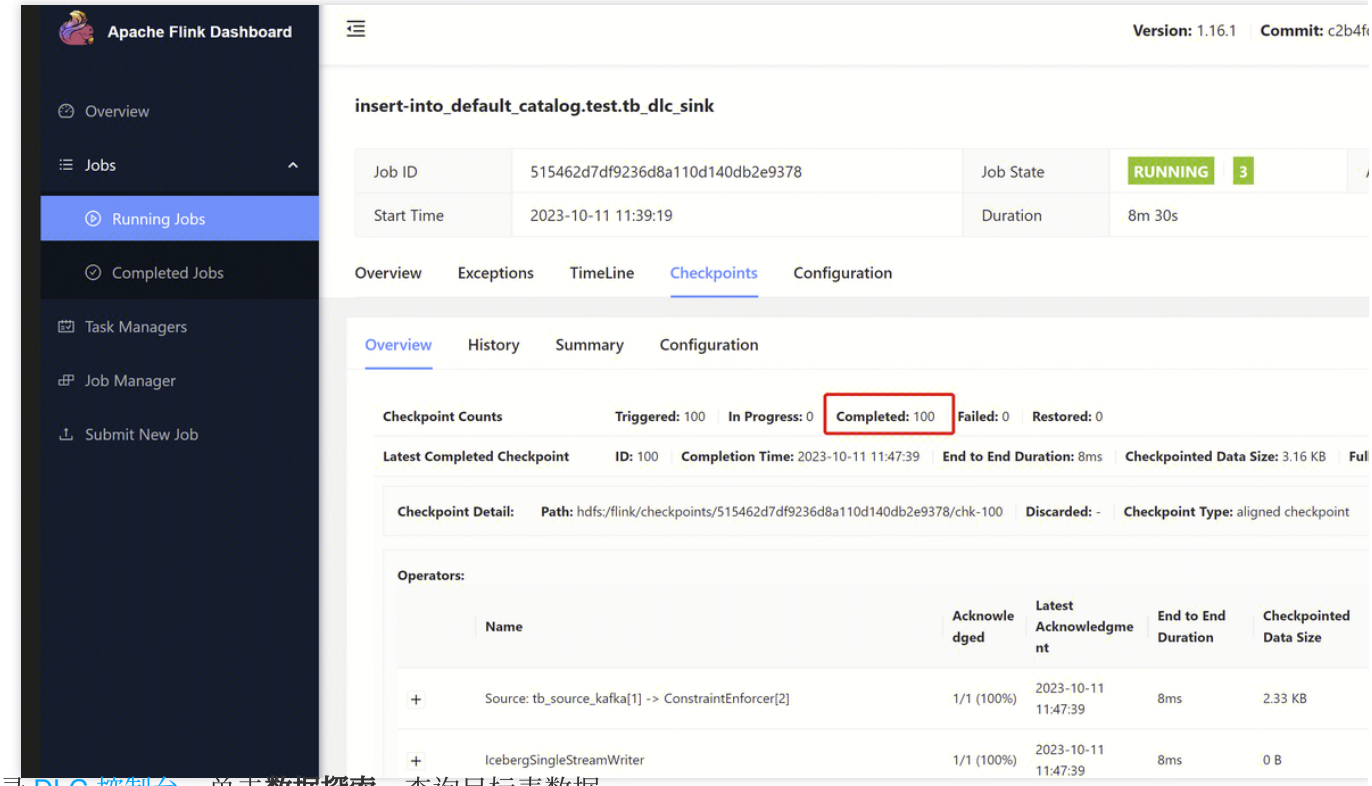
```
{"id":1,"name":"Zhangsan","age":18}
{"id":2,"name":"Lisi","age":19}
{"id":3,"name":"Wangwu","age":20}
{"id":4,"name":"Lily","age":21}
{"id":5,"name":"Lucy","age":22}
{"id":6,"name":"Huahua","age":23}
{"id":7,"name":"Wawa","age":24}
{"id":8,"name":"Mei","age":25}
{"id":9,"name":"Joi","age":26}
{"id":10,"name":"Qi","age":27}
{"id":11,"name":"Ky","age":28}
```

版权所有：腾讯云计算（北京）有限责任公司

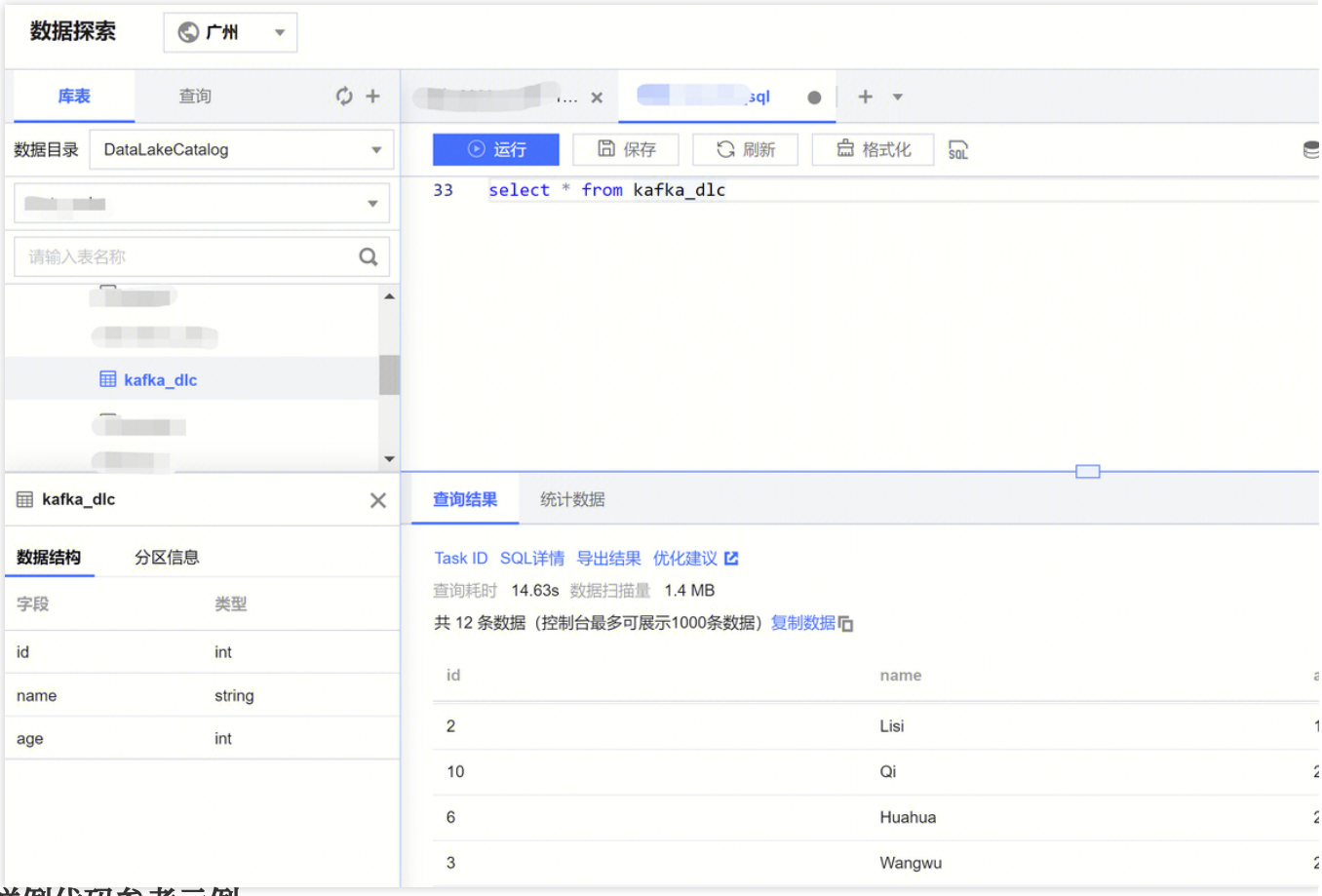
```
{"id":12,"name":"Mark","age":29}
```

2. 查询同步结果

打开 Flink Dashboard，单击 **Running Job > 运行Job > Checkpoint > Overview，**查看 Job 同步结果。



3. 登录 DLC 控制台，单击**数据探索，**查询目标表数据。

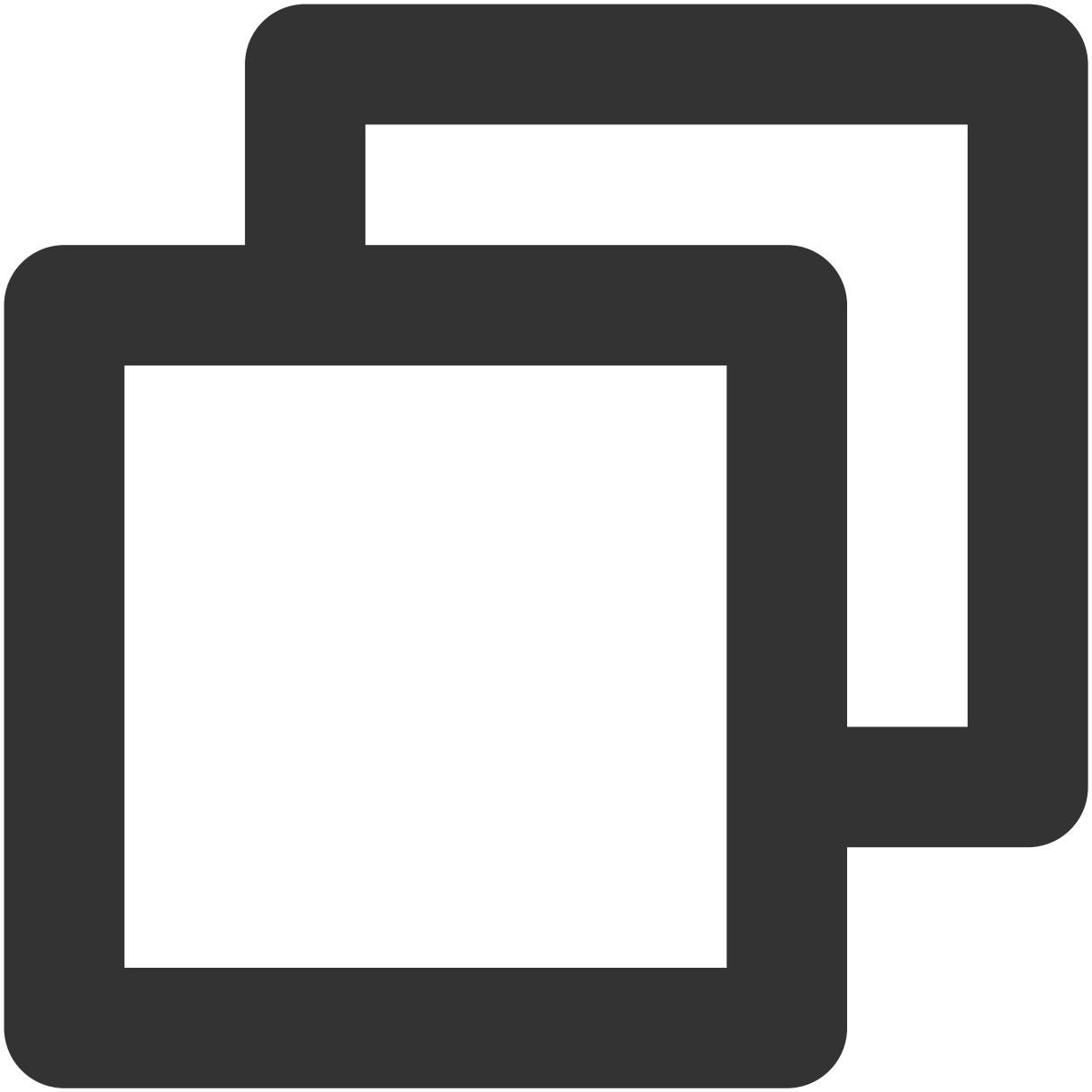


完整样例代码参考示例

说明：

示例中带“****”的数据请替换成开发中实际的数据。

示例1

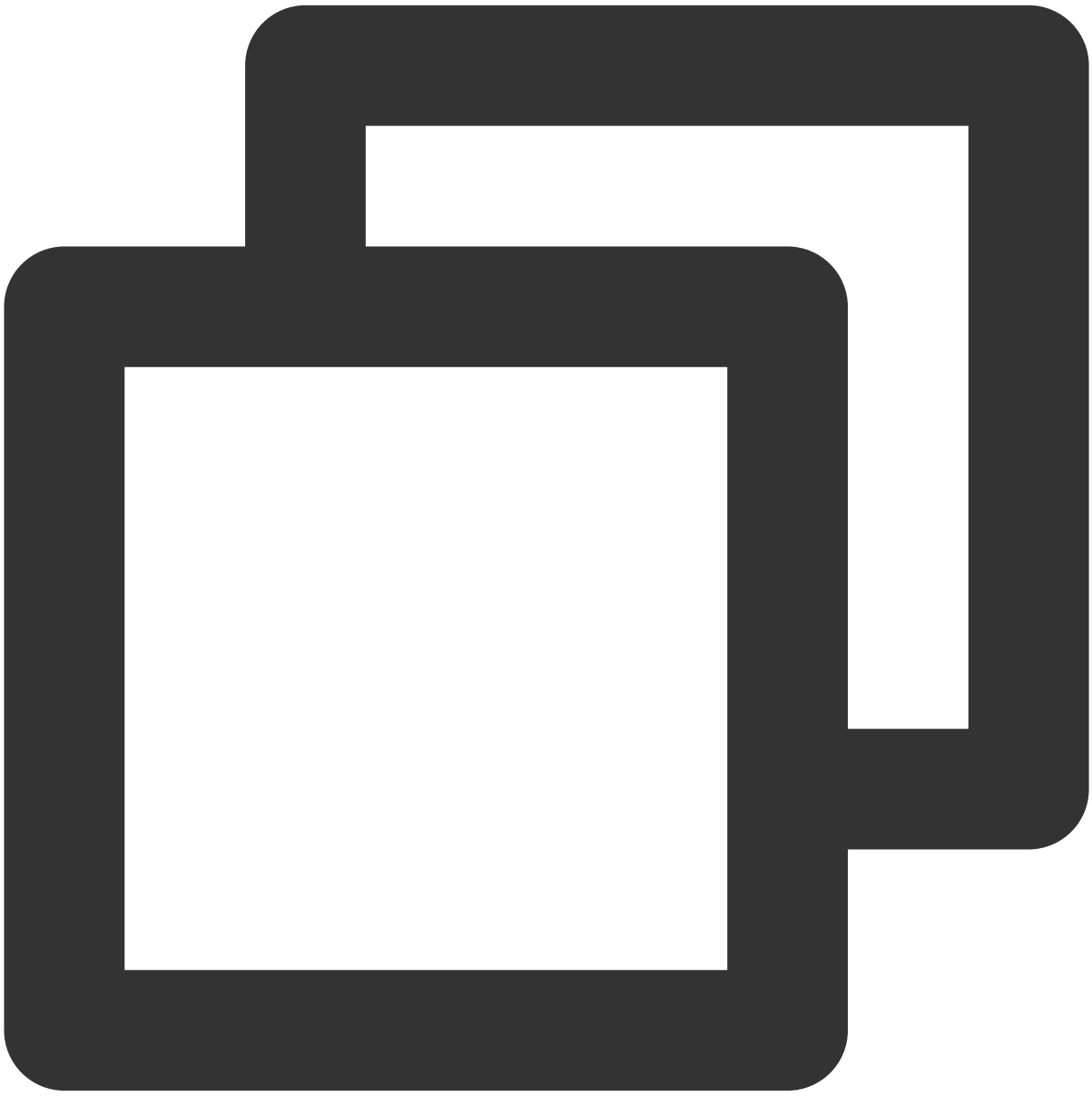


```
<properties>
  <flink.version>1.15.4</flink.version>
  <cos.lakefs.plugin.version>1.0</cos.lakefs.plugin.version>
</properties>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
```

```
    </dependency>
    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-java</artifactId>
      <version>${flink.version}</version>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-clients</artifactId>
      <version>${flink.version}</version>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-streaming-java</artifactId>
      <version>${flink.version}</version>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-connector-kafka</artifactId>
      <version>${flink.version}</version>
    </dependency>

    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-table-planner_2.12</artifactId>
      <version>${flink.version}</version>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-json</artifactId>
      <version>${flink.version}</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>com.qcloud.cos</groupId>
      <artifactId>lakefs-cloud-plugin</artifactId>
      <version>${cos.lakefs.plugin.version}</version>
      <exclusions>
        <exclusion>
```

```
            <groupId>com.tencentcloudapi</groupId>
            <artifactId>tencentcloud-sdk-java</artifactId>
          </exclusion>
        </exclusions>
      </dependency>
    </dependencies>
```

示例2

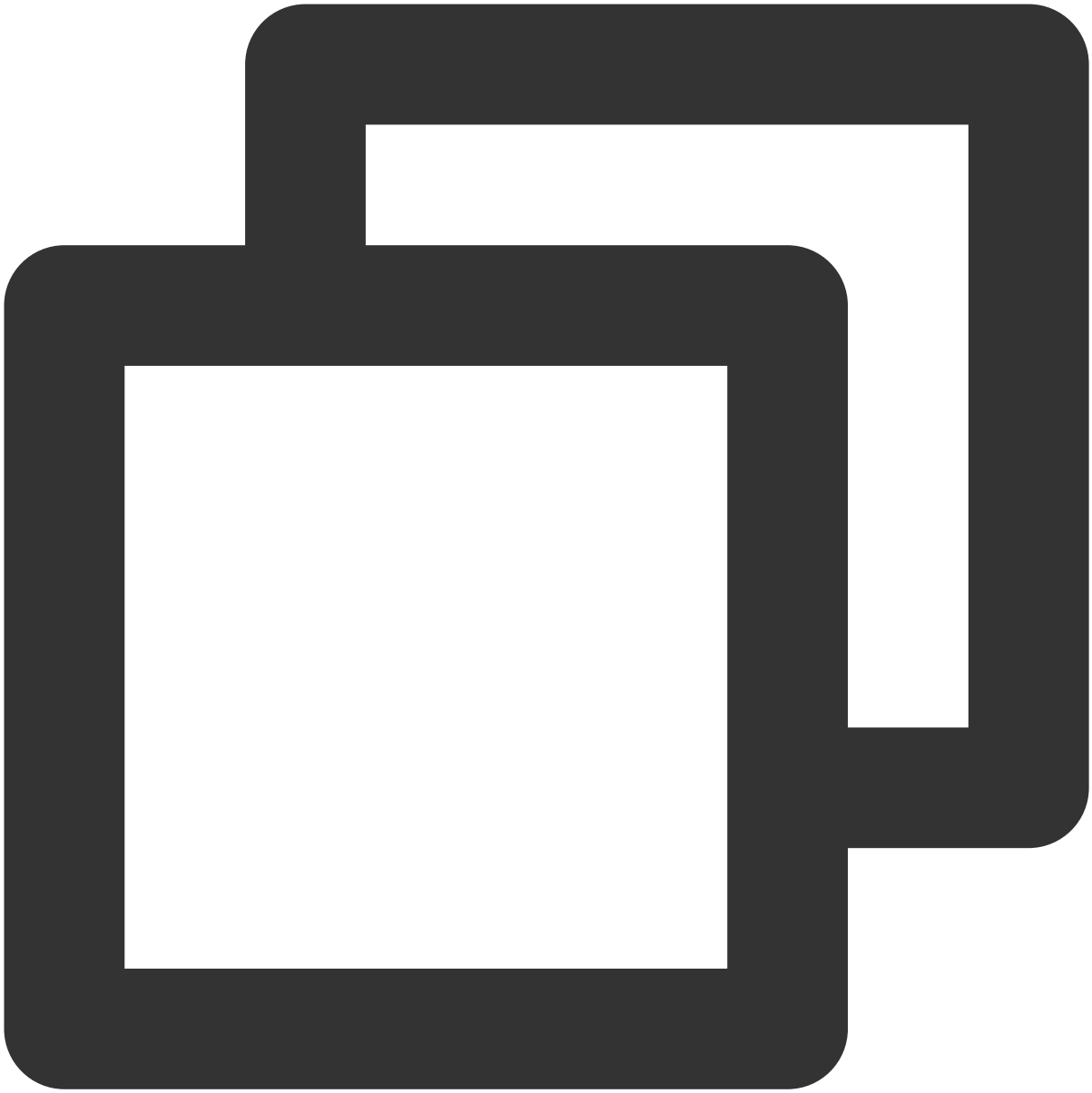


```
public class AppendIceberg {

    public static void main(String[] args) {
```

```
        // 创建执行环境 和 配置checkpoint
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnv
        env.setParallelism(1);
        env.enableCheckpointing(60000);
        env.getCheckpointConfig().setCheckpointStorage("hdfs:///flink/checkpoints")
        env.getCheckpointConfig().setCheckpointTimeout(60000);
        env.getCheckpointConfig().setTolerableCheckpointFailureNumber(5);
        env.getCheckpointConfig()
                .enableExternalizedCheckpoints(CheckpointConfig.ExternalizedCheckpo
        EnvironmentSettings settings = EnvironmentSettings
                .newInstance()
                .inStreamingMode()
                .build();
        StreamTableEnvironment tEnv = StreamTableEnvironment.create(env, settings);

        // 创建输入表
        String sourceSql = "CREATE TABLE tb_kafka_sr ( \\n"
                + "   id INT, \\n"
                + "   name STRING, \\n"
                + "   age INT \\n"
                + ") WITH ( \\n"
                + "  'connector' = 'kafka', \\n"
                + "  'topic' = 'kafka_dlc', \\n"
                + "  'properties.bootstrap.servers' = '10.0.126.***:9092', \\n" //
                + "  'properties.group.id' = 'test-group', \\n"
                + "  'scan.startup.mode' = 'earliest-offset', \\n"  // 从可能的最早偏
                + "  'format' = 'json' \\n"
                + ");";
        tEnv.executeSql(sourceSql);

        // 创建输出表
        String sinkSql = "CREATE TABLE tb_dlc_sk ( \\n"
                + "  id INT PRIMARY KEY NOT ENFORCED, \\n"
                + "  name STRING,\\n"
                + "  age INT\\n"
                + ") WITH (\\n"
                + "  'qcloud.dlc.managed.account.uid' = '1000***79117',\\n" //用户Ui
                + "  'qcloud.dlc.secret-id' = 'AKIDwjQvBHCsKYXL3***pMdkeMsBH8lAJEt'
                + "  'qcloud.dlc.region' = 'ap-***',\\n" // 数据库表地域信息
                + "  'qcloud.dlc.user.appid' = 'AKIDwjQvBHCsKYXL3***pMdkeMsBH8lAJEt
                + "  'qcloud.dlc.secret-key' = 'kFWYQ5WklaCYgbLtD***cyAD7sUyNiVP',\
                + "  'connector' = 'iceberg-inlong', \\n"
                + "  'catalog-database' = 'test_***', \\n" // 目标数据库
                + "  'catalog-table' = 'kafka_dlc', \\n" // 目标数据表
                + "  'default-database' = 'test_***', \\n" //默认数据库
                + "  'catalog-name' = 'HYBRIS', \\n"
                + "  'catalog-impl' = 'org.apache.inlong.sort.iceberg.catalog.hybri
```

```
                + "  'uri' = 'dlc.tencentcloudapi.com', \\n"
                + "  'fs.cosn.credentials.provider' = 'org.apache.hadoop.fs.auth.Dl
                + "  'qcloud.dlc.endpoint' = 'dlc.tencentcloudapi.com', \\n"
                + "  'fs.lakefs.impl' = 'org.apache.hadoop.fs.CosFileSystem', \\n"
                + "  'fs.cosn.impl' = 'org.apache.hadoop.fs.CosFileSystem', \\n"
                + "  'fs.cosn.userinfo.region' = 'ap-guangzhou', \\n" // 使用到的COS
                + "  'fs.cosn.userinfo.secretId' = 'AKIDwjQvBHCsKYXL3***pMdkeMsBH8l
                + "  'fs.cosn.userinfo.secretKey' = 'kFWYQ5WklaCYgbLtD***cyAD7sUyNi
                + "  'service.endpoint' = 'dlc.tencentcloudapi.com', \\n"
                + "  'service.secret.id' = 'AKIDwjQvBHCsKYXL3***pMdkeMsBH8lAJEt', \
                + "  'service.secret.key' = 'kFWYQ5WklaCYgbLtD***cyAD7sUyNiVP', \\n
                + "  'service.region' = 'ap-***', \\n"  // 数据库表地域信息
                + "  'user.appid' = '1305424723', \\n"
                + "  'request.identity.token' = '1000***79117', \\n"
                + "  'qcloud.dlc.jdbc.url'='jdbc:dlc:dlc.internal.tencentcloudapi.c
                + ");";
        tEnv.executeSql(sinkSql);
        // 执行计算并输出
        String sql = "insert into tb_dlc_sk select * from tb_kafka_sr";
        tEnv.executeSql(sql);
    }

}
```

示例3

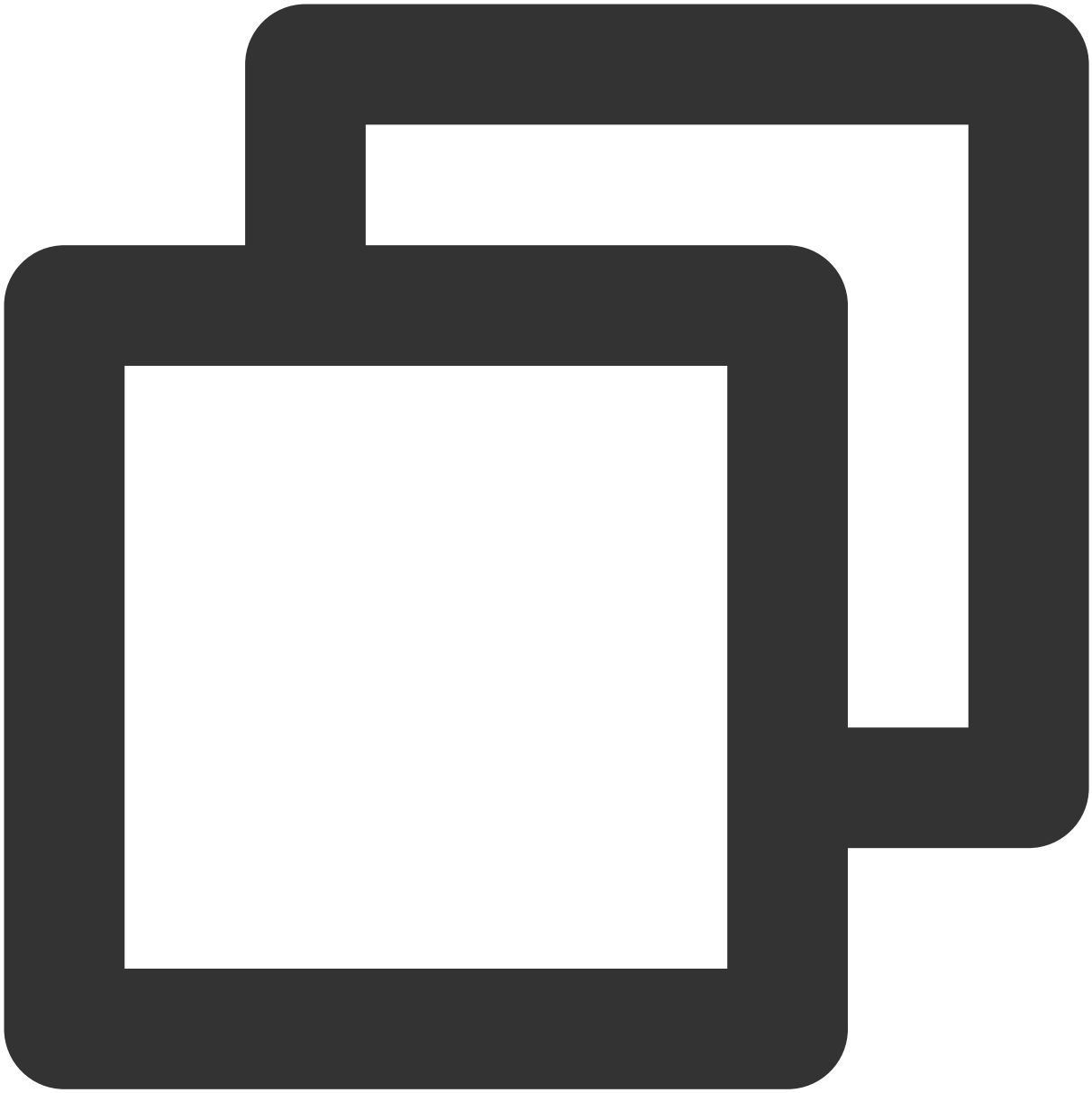


```
<properties>
  <flink.version>1.15.4</flink.version>
</properties>

<dependencies>
  <dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>2.0.22</version>
    <scope>provided</scope>
  </dependency>
```

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-java</artifactId>
  <version>${flink.version}</version>
  <scope>provided</scope>
</dependency>

<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-clients</artifactId>
  <version>${flink.version}</version>
  <scope>provided</scope>
</dependency>

<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-streaming-java</artifactId>
  <version>${flink.version}</version>
  <scope>provided</scope>
</dependency>

<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-kafka</artifactId>
  <version>${flink.version}</version>
  <scope>provided</scope>
</dependency>

<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-planner_2.12</artifactId>
  <version>${flink.version}</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-json</artifactId>
  <version>${flink.version}</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.apache.inlong</groupId>
  <artifactId>sort-connector-iceberg-dlc</artifactId>
  <version>1.6.0</version>
  <scope>system</scope>
  <systemPath>${project.basedir}/lib/sort-connector-iceberg-dlc-1.6.0.jar</syst
```

```
    </dependency>
    <dependency>
      <groupId>org.apache.kafka</groupId>
      <artifactId>kafka-clients</artifactId>
      <version>${kafka-version}</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
      <version>1.7.25</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>1.7.25</version>
    </dependency>
  </dependencies>
```

示例4



```
public class KafkaToDLC {

    public static void main(String[] args) throws Exception {
        final MultipleParameterTool params = MultipleParameterTool.fromArgs(args);
        final Map<String, String> options = setOptions();
        //1.执行环境 StreamTableEnvironment,配置checkpoint
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnv
        env.setParallelism(1);
        env.enableCheckpointing(60000);
        env.getCheckpointConfig().setCheckpointStorage("hdfs:///data/checkpoints");
        env.getCheckpointConfig().setCheckpointTimeout(60000);
```

```
        env.getCheckpointConfig().setTolerableCheckpointFailureNumber(5);
        env.getCheckpointConfig()
                .enableExternalizedCheckpoints(CheckpointConfig.ExternalizedCheckpo
        env.getConfig().setGlobalJobParameters(params);

        //2.获取输入流
        KafkaToDLC dlcSink = new KafkaToDLC();
        DataStream<RowData> dataStreamSource = dlcSink.buildInputStream(env);

        //3.创建Hadoop配置、Catalog配置
        CatalogLoader catalogLoader = FlinkDynamicTableFactory.createCatalogLoader(
        TableLoader tableLoader = TableLoader.fromCatalog(catalogLoader,
                TableIdentifier.of(params.get(CATALOG_DATABASE.key()), params.get(C
        tableLoader.open();
        Table table = tableLoader.loadTable();
        ActionsProvider actionsProvider = FlinkDynamicTableFactory.createActionLoad
                Thread.currentThread().getContextClassLoader(), options);
        //4.创建Schema
        Schema schema = Schema.newBuilder()
                .column("id", DataTypeUtils.toInternalDataType(new IntType(false)))
                .column("name", DataTypeUtils.toInternalDataType(new VarCharType())
                .column("age", DataTypeUtils.toInternalDataType(new DateType(false)
                .primaryKey("id")
                .build();
        List<String> equalityColumns = schema.getPrimaryKey().get().getColumnNames(
        //5.配置Slink
        FlinkSink.forRowData(dataStreamSource)
                //这个 .table 也可以不写，指定tableLoader 对应的路径就可以。
                .table(table)
                .tableLoader(tableLoader)
                .equalityFieldColumns(equalityColumns)
                .metric(params.get(INLONG_METRIC.key()), params.get(INLONG_AUDIT.ke
                .action(actionsProvider)
                .tableOptions(Configuration.fromMap(options))
                //默认为false,追加数据。如果设置为true 就是覆盖数据
                .overwrite(false)
                .append();
        //6.执行同步
        env.execute("DataStream Api Write Data To Iceberg");
    }

    private static Map<String, String> setOptions() {
        Map<String, String> options = new HashMap<>();
        options.put("qcloud.dlc.managed.account.uid", "1000***79117"); //用户Uid
        options.put("qcloud.dlc.secret-id", "AKIDwjQvBHCsKYXL3***pMdkeMsBH8lAJEt");
        options.put("qcloud.dlc.region", "ap-***"); // 数据库表地域信息
        options.put("qcloud.dlc.user.appid", "AKIDwjQvBHCsKYXL3***pMdkeMsBH8lAJEt")
```

```
        options.put("qcloud.dlc.secret-key", "kFWYQ5WklaCYgbLtD***cyAD7sUyNiVP"); /
        options.put("connector", "iceberg-inlong");
        options.put("catalog-database", "test_***"); // 目标数据库
        options.put("catalog-table", "kafka_dlc"); // 目标数据表
        options.put("default-database", "test_***"); //默认数据库
        options.put("catalog-name", "HYBRIS");
        options.put("catalog-impl", "org.apache.inlong.sort.iceberg.catalog.hybris.
        options.put("uri", "dlc.tencentcloudapi.com");
        options.put("fs.cosn.credentials.provider", "org.apache.hadoop.fs.auth.DlcC
        options.put("qcloud.dlc.endpoint", "dlc.tencentcloudapi.com");
        options.put("fs.lakefs.impl", "org.apache.hadoop.fs.CosFileSystem");
        options.put("fs.cosn.impl", "org.apache.hadoop.fs.CosFileSystem");
        options.put("fs.cosn.userinfo.region", "ap-guangzhou"); // 使用到的COS的地域信
        options.put("fs.cosn.userinfo.secretId", "AKIDwjQvBHCsKYXL3***pMdkeMsBH8lAJ
        options.put("fs.cosn.userinfo.secretKey", "kFWYQ5WklaCYgbLtD***cyAD7sUyNiVP
        options.put("service.endpoint", "dlc.tencentcloudapi.com");
        options.put("service.secret.id", "AKIDwjQvBHCsKYXL3***pMdkeMsBH8lAJEt"); //
        options.put("service.secret.key", "kFWYQ5WklaCYgbLtD***cyAD7sUyNiVP"); // 用
        options.put("service.region", "ap-***");  // 数据库表地域信息
        options.put("user.appid", "1305***23");
        options.put("request.identity.token", "1000***79117");
        options.put("qcloud.dlc.jdbc.url",
                "jdbc:dlc:dlc.internal.tencentcloudapi.com?task_type,SparkSQLTask&d
        return options;
    }

    /**
     * 创建输入流
     *
     * @param env
     * @return
     */
    private DataStream<RowData> buildInputStream(StreamExecutionEnvironment env) {
        //1.配置执行环境
        EnvironmentSettings settings = EnvironmentSettings
                .newInstance()
                .inStreamingMode()
                .build();
        StreamTableEnvironment sTableEnv = StreamTableEnvironment.create(env, setti
        org.apache.flink.table.api.Table table = null;
        //2.执行SQL，获取数据输入流
        try {
            sTableEnv.executeSql(createTableSql()).print();
            table = sTableEnv.sqlQuery(transformSql());
            DataStream<Row> rowStream = sTableEnv.toChangelogStream(table);
            DataStream<RowData> rowDataDataStream = rowStream.map(new MapFunction<R
                @Override
```

```
                public RowData map(Row rows) throws Exception {
                    GenericRowData rowData = new GenericRowData(3);
                    rowData.setField(0, rows.getField(0));
                    rowData.setField(1, (String) rows.getField(1));
                    rowData.setField(2, rows.getField(2));
                    return rowData;
                }
            });
            return rowDataDataStream;
        } catch (Exception e) {
            throw new RuntimeException("kafka to dlc transform sql execute error.",
        }
    }

    private String createTableSql() {
        String tableSql = "CREATE TABLE tb_kafka_sr ( \\n"
                + "   id INT, \\n"
                + "   name STRING, \\n"
                + "   age INT \\n"
                + ") WITH ( \\n"
                + "  'connector' = 'kafka', \\n"
                + "  'topic' = 'kafka_dlc', \\n"
                + "  'properties.bootstrap.servers' = '10.0.126.30:9092', \\n"
                + "  'properties.group.id' = 'test-group-10001', \\n"
                + "  'scan.startup.mode' = 'earliest-offset', \\n"
                + "  'format' = 'json' \\n"
                + ");";
        return tableSql;
    }

    private String transformSql() {
        String transformSQL = "select * from tb_kafka_sr";
        return transformSQL;
    }
}
```

DLC 原生表常见 FAQ

最近更新时间：2024-07-31 17:35:21

为什么 Upsert 写入的 DLC 原生表（Iceberg）一定要开启数据优化？

1. DLC 原生表（Iceberg）采用的 MOR（Merge On Read）表，上游 Upsert 写入时，针对 update 的数据，会先写 delete file 标记某记录已经被删除，然后再写 data file 新增改记录。
2. 如果不提交进行合并，作业引擎在读取数据时，需要将读取原来的数据，该记录的 delete file 和新增的 data file，将三者进行合并得到最新的数据，这会导致作业需要大量的资源和时间来进行。数据优化中的小文件合并是提前将上述的文件读取回来合并，并写成新的 data file，使得作业引擎不用再进行数据文件合并而直接读取最新的文件。
3. DLC 元数据（Iceberg）采用了的快照机制，写入流程中即便是产生了新的快照也不会将历史快照清理，这依赖于数据优化的快照过期能力，将产生时间较久远的快照过期移除，从而达到释放存储空间效果，避免无用的历史数据占用存储空间。

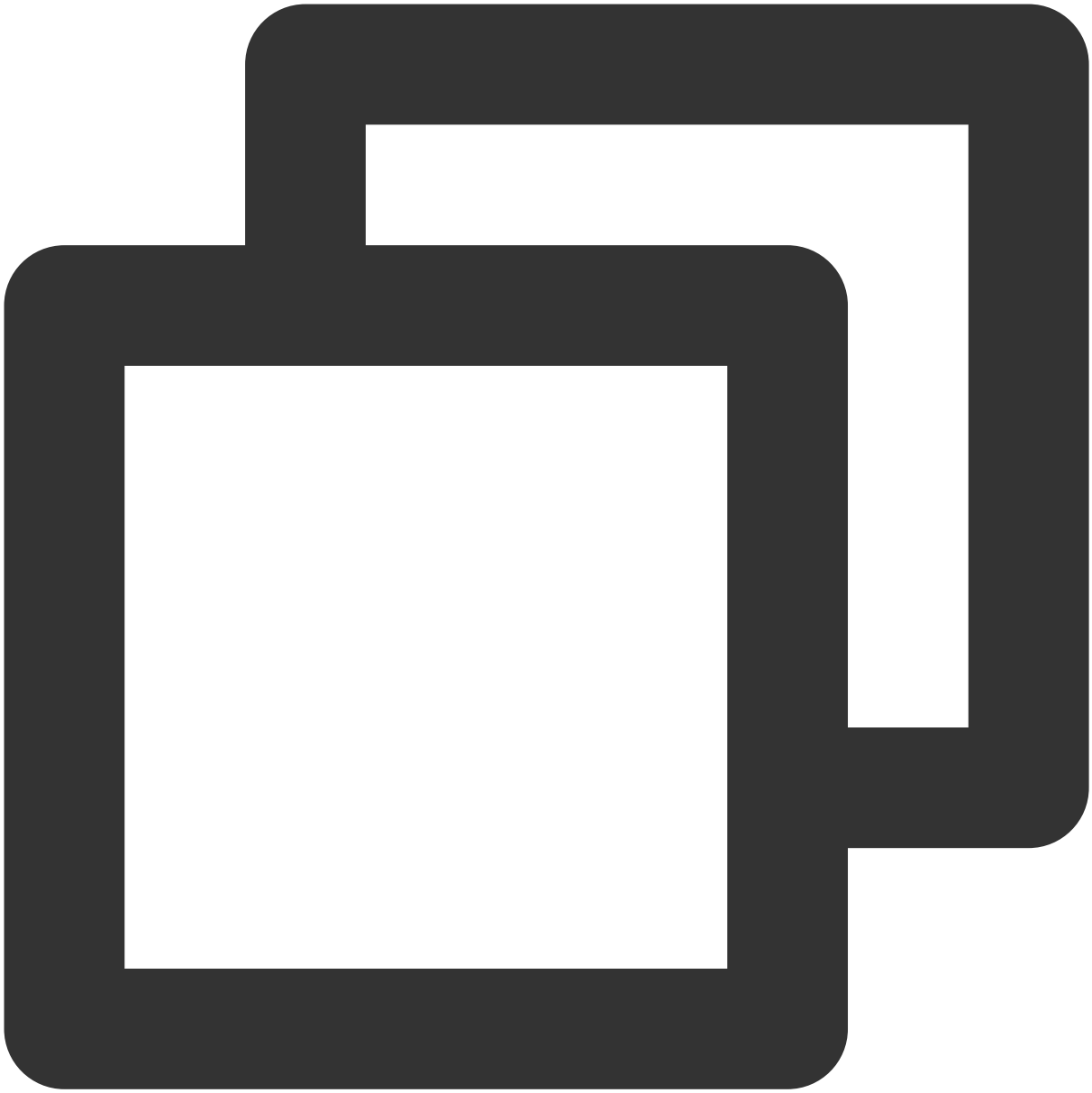
数据优化任务出现执行超时的任务怎么处理？

系统针对数据优化任务默认设置运行超时时间（默认2小时），避免某个任务长时间占用资源而导致其他任务无法执行，当超时时间到期时该优化任务会被系统取消，根据任务类型的不同，可参考如下流程处理。
1. 如果是小文件合并任务超时，当出现连续多次超时识别的，则是数据存在了堆积，当前资源已经无法满足该表的合并，则可以临时扩展资源（或者设置该表只有优化资源为独立的资源），将历史堆积数据处理完毕后再设置回来。
2. 如果是小文件合并任务，偶尔出现任务执行超时，则是治理资源有些不足，可以适当地对数据资源扩容，并持续观察后续多个周期的治理任务是否还存在超时。当某些表偶尔出现小文件合并超时，短期内并不会对查询性能造成影响，但是不处理可能会发展为连续超时失败，到达该阶段后将影响查询性能。 DLC 默认针对小文件合并开启了分段提交，当执行超时，已经完成的部分任务仍然有机会提交成功，提交成功的合并仍然有效。
3. 如果是快照过期执行超时，快照过期执行分为两个阶段，第一个阶段从元数据中移除快照，该过程执行快照，通常不会在该阶段超时，第二个阶段将被移除快照的数据文件从存储上删除，该阶段需要逐一比较删除文件，当待删除的文件较多时，可能会出现超时。该类型的任务超时可以忽略，任务超时被移除的文件在系统会被当做孤立文件而被后续的移除孤立文件清理。
4. 如果是孤立文件执行超时，孤立文件与移除孤立文件类似，只要扫描到被删除的文件仍然是有效的，由于移除孤立文件是周期性地扫描执行，当本次任务超时后，后续的周期会继续扫描执行。

为什么 Iceberg 在 insert 写完数据后会偶尔在极短的时间内读到旧的快照？

1. Iceberg 提供了默认缓存 Catalog 的能力，默认30秒，极端情况下如果两次查询相同表间隔特别短且不在同一个 session 中执行时，在缓存没有过期和获取更新之前，有极低的概率将会查询到上一个旧快照。
2. 该参数 Iceberg 社区建议开启，DLC 在早期版本也是默认开启，该参数是为了加速任务执行，减少查询过程中对元数据的访问。 但是在极端情况下，如果两个任务读写间隔特别近，可能会出出现上述描述的情况。
3. DLC 在新版本的引擎中已经默认关闭，在结合用户场景，用户在2024年1月份之前购买的引擎如果用户需要保证数据查询到强一致，可通过如下手动关闭该参数，配置方法参考，修改引擎参数：

版权所有：腾讯云计算（北京）有限责任公司



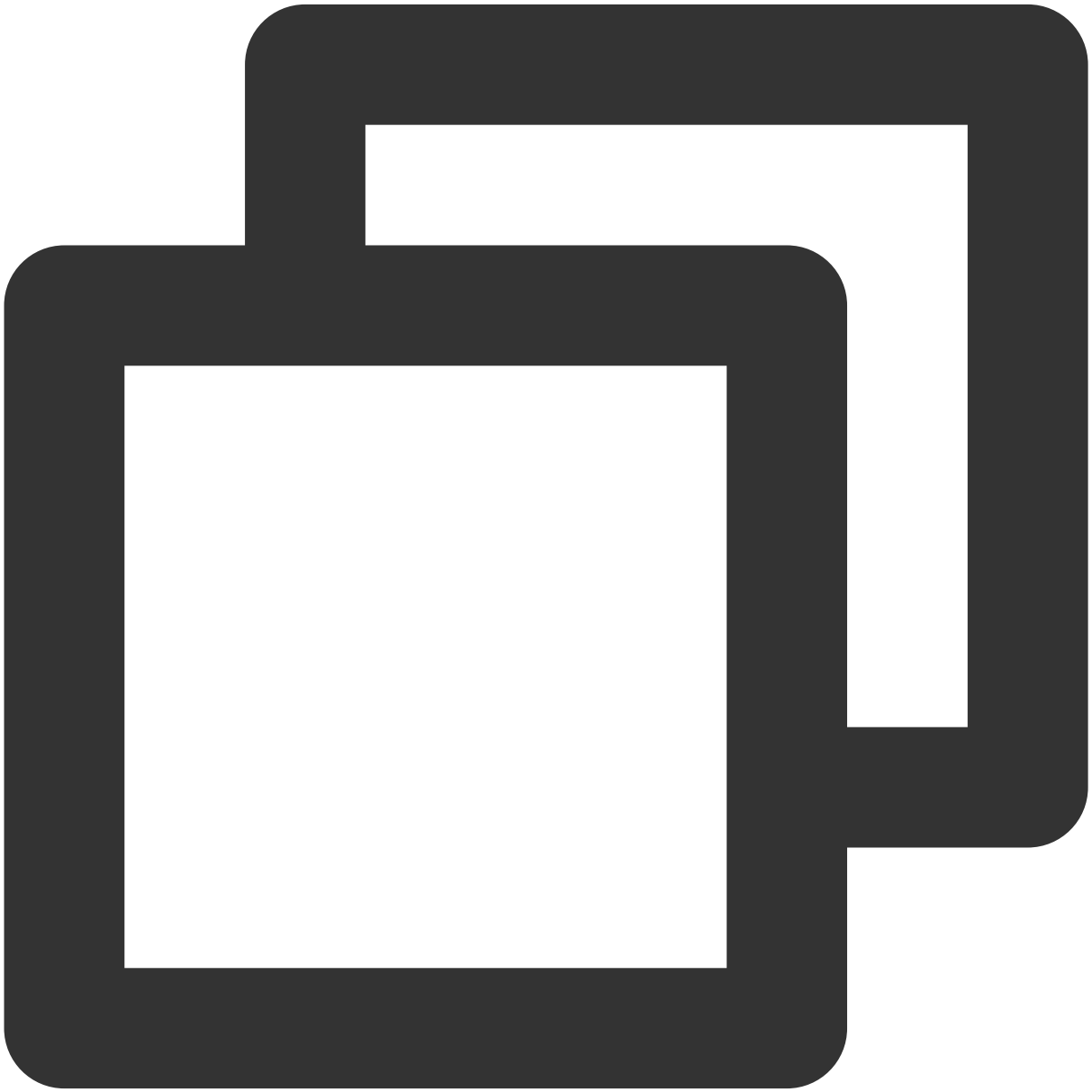
```
"spark.sql.catalog.DataLakeCatalog.cache-enabled": "false"
"spark.sql.catalog.DataLakeCatalog.cache.expiration-interval-ms": "0"
```

为什么建议 DLC 元原生表（Iceberg）需要进行分区？

1. 数据优化首先按照分区进行job划分，如果原生表（Iceberg）没有分区，大多数情况会改表的小文件合并都只有一个 job 执行，无法并行合并，很大程度降低小文件合并效率。
2. 如果表上游无分区字段，如何分区呢？此时可考虑 Iceberg 的 bucket 分桶，详细描述请参见 DLC 原生表核心能力。

DLC 原生表（Iceberg）写冲突如何处理？

1. Iceberg 为保证 ACID，在 commit 时要检查当前的视图是否有变化，如果有变化则判断为发生了冲突，之后回退到 commit 操作，合并当前视图，然后重新提交。
2. 系统提供了默认的冲突重试次数和时间，当发生多次 commit 操作还是发生了冲突，则将写入失败。默认冲突参数请参见 DLC 原生表核心能力。
3. 当冲突发生了，用户可以调整重试次数和重试时间。如下示例将冲突重试次数调整为10次，更多的参数含义请参见 DLC 原生表核心能力。



```
// 修改冲突重试次数为10
```

```
ALTER TABLE `DataLakeCatalog`.`axitest`.`upsert_case` SET TBLPROPERTIES('commit.ret
```

DLC 原生表（Iceberg）已经删除了，为什么存储空间容量还没有释放？

DLC 原生表（Iceberg）drop table 时元数据立即删除，数据是异步删除，先是将数据移动到回收站目录，延迟一天后数据才会从存储上移除。