消息队列 CKafka 版

通用参考

腾讯云

文档目录

通用参考
对 CKafka 进行生产和消费压力测试

最近更新时间：2024-10-11 11:22:50

测试工具

Kafka Producer 和 Consumer 的性能测试均可使用 Kafka 客户端自带的开源脚本，主要输出每秒发送消息量（MB/second）和每秒发送消息数（records/second）两项指标。

- Kafka Producer 测试脚本：`$KAFKA_HOME/bin/kafka-producer-perf-test.sh`
- Kafka Consumer 测试脚本：`$KAFKA_HOME/bin/kafka-consumer-perf-test.sh`

测试命令

> **说明**
> 以下命令中的 `ckafka vip:vport` 应替换为您实际实例分配的 IP 和端口。

生产测试命令示例：

```
bin/kafka-producer-perf-test.sh
--topic test
--num-records 123
--record-size 1000
--producer-props bootstrap.servers= ckafka vip : port
--throughput 20000
```

消费测试命令示例：

```
bin/kafka-consumer-perf-test.sh
--topic test
--new-consumer
--fetch-size 10000
--messages 1000
--broker-list bootstrap.servers=ckafka vip : port
```

测试建议

- 为了提高吞吐量，建议创建分区时数量 ⩾ 3（因后端 CKafka 集群节点数量最少是3，如只创建1个分区则分区会分布在一个 Broker 上面，影响性能）。
- 由于 CKafka 是分区级别消息有序的，因此过多的分区也会影响生产性能，根据实际压测，建议分区数不超过6。
- 为了保证压力测试的效果，需要多客户端模拟一定的并发，建议采用多台机器作为压测客户端（生产端），每台启动多个压测程序，提高并发。此外建议每1s启动一个生产者，避免同时启动所有生产者导致测试机器高负载。

常见参数配置说明

最近更新时间：2025-02-10 17:37:32

Broker 配置参数说明

当前 CKafka broker 端的一些配置如下，供参考：

```
# 消息体的最大大小，单位是字节
message.max.bytes=1000012

# 是否允许自动创建 Topic，默认是 false，当前可以通过控制台或云 API 创建
auto.create.topics.enable=false

# 是否允许调用接口删除 Topic
delete.topic.enable=true

# Broker 允许的最大请求大小为16MB
socket.request.max.bytes=16777216

# 每个 IP 与 Broker 最多建立5000个连接
max.connections.per.ip=5000

# offset 保留时间，默认为7天
offsets.retention.minutes=10080

# 没有 ACL 设置时，允许任何人访问
allow.everyone.if.no.acl.found=true

# 日志分片大小为1GB
log.segment.bytes=1073741824

# 日志滚动检查间隔5分钟，当设置保留时间小于5分钟时，也可能需要等待5分钟才会清空日志
log.retention.check.interval.ms=300000
```

> **说明**
> 其他未列出的 Broker 配置参见 开源 Kafka 默认配置 。

Topic 配置参数说明

1. 选取合适的分区数量

从生产者的角度来看，向不同的 partition 写入是完全并行的；从消费者的角度来看，并发数完全取决于 partition 的数量（如果 consumer 数量大于 partition 数量，则必有 consumer 闲置）。因此选取合适的分区数量对于发挥 CKafka 实例的性能十分重要。
partition 的数量需要根据生产和消费的吞吐来判断。理想情况下，可以通过如下公式来判断分区的数目：

```
Num =  max( T/PT , T/CT ) = T / min( PT ,  CT )
```

其中，Num 代表 partition 数量，T 代表目标吞吐量，PT 代表生产者写入单个 partition 的最大吞吐，CT 代表消费者从单个 partition 消费的最大吞吐。则 partition 数量应该等于 T/PT 和 T/CT 中较大的那一个。
在实际情况中，生产者写入单个 partition 的最大吞吐 PT 的影响因素和批处理的规模、压缩算法、确认机制、副本数等有关。消费者从单个 partition 消费的最大吞吐 CT 的影响因素和业务逻辑有关，需要在不同场景下实测得出。
通常建议 partition 的数量一定要大于等于消费者的数量来实现最大并发。如果消费者数量是 5，则 partition 的数目也应该是 ≥ 5 的。同时，过多的分区会导致生产吞吐的降低和选举耗时的增加，因此也不建议过多分区。提供如下信息供参考：

- 单个 partition 是可以实现消息的顺序写入的。
- 单个 partition 只能被同消费者组的单个消费者进程消费。
- 单个消费者进程可同时消费多个 partition，即 partition 限制了消费端的并发能力。
- partition 越多则失败后 leader 选举的耗时越长。
- offset 的粒度最细是在 partition 级别的，partition 越多，查询 offset 就越耗时。
- partition 的数量是可以动态增加的，只能增加不能减少。但增加会出现消息 rebalance 的情况。

2. 选取合适的副本

目前为了保证可用性副本数必须大于等于2，如果需要保障高可靠建议3副本。

> ⚠ 注意
> 副本数会影响生产/消费流量，如3副本则实际流量 = 生产流量 × 3。

3. 日志保留时间

Topic 的 log.retention.ms 配置通过控制台实例的保留时间统一设置。

4. 其他 Topic 级别配置说明

```
# Topic 级别最大消息大小
max.message.bytes=1000012

# 0.10.2 版本消息格式为 V1 格式
message.format.version=0.10.2-IV0

# 不在 ISR 中的 replica 允许选择为 Leader，可用性高于可靠性，存在数据丢失风险。
unclean.leader.election.enable=true

# ISR 提交生产者请求的最小副本数。如果同步状态的副本数小于该值，服务器将不再接受。request.required.acks为-1或all的写入请求。
min.insync.replicas=1
```

生产者配置指南

生产端常用参数配置如下，建议客户根据实际业务场景调整配置：

```
# 生产者会尝试将业务发送到相同的 Partition 的消息合包发送到 Broker，batch.size 设置合包的大小上限。默认为 16KB。batch.size 设太小会导致吞吐下降，设太大会导致内存使用过多。
batch.size=16384

# Kafka producer 的 ack 有 3 种机制，分别说明如下：
# -1 或 all：Broker 在 leader 收到数据并同步给所有 ISR 中的 follower 后，才应答给 Producer 继续发送下一条（批）消息。 这种配置提供了最高的数据可靠性，只要有一个已同步的副本存活就不会有消息丢失。注意：这种配置不能确保所有的副本读写入该数据才返回，可以配合 Topic 级别参数 min.insync.replicas 使用。
# 0：生产者不等待来自 broker 同步完成的确认，继续发送下一条（批）消息。这种配置生产性能最高，但数据可靠性最低（当服务器故障时可能会有数据丢失，如果 leader 已死但是 producer 不知情，则 broker 收不到消息）
# 1：生产者在 leader 已成功收到的数据并得到确认后再发送下一条（批）消息。这种配置是在生产吞吐和数据可靠性之间的权衡（如果leader已死但是尚未复制，则消息可能丢失）

# 用户不显式配置时，默认值为1。用户根据自己的业务情况进行设置
acks=1

# 控制生产请求在 Broker 等待副本同步满足 acks 设置的条件所等待的最大时间
timeout.ms=30000

# 配置生产者用来缓存消息等待发送到 Broker 的内存。用户要根据生产者所在进程的内存总大小调节
buffer.memory=33554432

# 当生产消息的速度比 Sender 线程发送到 Broker 速度快，导致 buffer.memory 配置的内存用完时会阻塞生产者 send 操作，该参数设置最大的阻塞时间
max.block.ms=60000

# 设置消息延迟发送的时间(ms)，这样可以等待更多的消息组成 batch 发送。默认为0表示立即发送。当待发送的消息达到 batch.size 设置的大小时，不管是否达到 linger.ms 设置的时间，请求也会立即发送
# 推荐用户根据实际使用场景，设置linger.ms在100~1000之间，更大的取值相对有更大的吞吐但会相应增加时延
linger.ms=100

# 设置分区缓存消息量(bytes)，达到该数值时生产者将批量的消息发送给broker。默认值为16384，过小的batch.size会增加发送请求次数可能会损耗性能和影响稳定性，用户根据实际场景，可适当增大该值。注：该值为上限值，当还未达到时若时间已经达到linger.ms时生产者会发送消息
batch.size=16384

# 生产者能够发送的请求包大小上限，默认为1MB。在修改该值时注意不能超过 Broker 配置的包大小上限16MB
```

```
max.request.size=1048576

# 压缩格式配置，目前 0.9(包含)以下版本不允许使用压缩，0.10（包含）以上不允许使用 GZip 压缩
compression.type=[none, snappy, lz4]

# 客户端发送给 Broker 的请求的超时时间，不能小于 Broker 配置的 replica.lag.time.max.ms，目前该值为10000ms
request.timeout.ms=30000

# 客户端在每个连接上最多可发送的最大的未确认请求数，该参数大于1且 retries 大于0时可能导致数据乱序。 希望消息严格有序时，建议客户将该值设置1
max.in.flight.requests.per.connection=5


# 请求发生错误时重试次数，建议将该值设置为大于0，失败重试最大程度保证消息不丢失
retries=0

# 发送请求失败时到下一次重试请求之间的时间
retry.backoff.ms=100
```

消费者配置指南

消费端常用参数配置如下，建议客户根据实际业务场景调整配置：

```
# 是否在消费消息后将 offset 同步到 Broker，当 Consumer 失败后就能从 Broker 获取最新的 offset
enable.auto.commit=true

# 当 auto.commit.enable=true 时，自动提交 Offset 的时间间隔，建议设置至少1000
auto.commit.interval.ms=5000

# 当 Broker 端没有 offset（如第一次消费或 offset 超过7天过期）时如何初始化 offset，当收到 OFFSET_OUT_OF_RANGE 错误时，如何重置 Offset
# earliest：表示自动重置到 partition 的最小 offset
# latest：默认为 latest，表示自动重置到 partition 的最大 offset
# none：不自动进行 offset 重置，抛出 OffsetOutOfRangeException 异常
auto.offset.reset=latest

# 标识消费者所属的消费分组
group.id=""

# 使用 Kafka 消费分组机制时，消费者超时时间。当 Broker 在该时间内没有收到消费者的心跳时，认为该消费者故障失败，Broker 发起重新 Rebalance 过程。目前该值的配置必须在 Broker 配置group.min.session.timeout.ms=6000和group.max.session.timeout.ms=300000之间
session.timeout.ms=10000

# 使用 Kafka 消费分组机制时，消费者发送心跳的间隔。这个值必须小于 session.timeout.ms，一般小于它的三分之一
heartbeat.interval.ms=3000

# 使用 Kafka 消费分组机制时，再次调用 poll 允许的最大间隔。如果在该时间内没有再次调用 poll，则认为该消费者已经失败，Broker 会重新发起 Rebalance 把分配给它的 partition 分配给其他消费者
max.poll.interval.ms=300000

# Fecth 请求最少返回的数据大小。默认设置为 1B，表示请求能够尽快返回。增大该值会增加吞吐，同时也会增加延迟
fetch.min.bytes=1

# Fetch 请求最多返回的数据大小，默认设置为 50MB
fetch.max.bytes=52428800

# Fetch 请求等待时间
fetch.max.wait.ms=500

# Fetch 请求每个 partition 返回的最大数据大小，默认为1MB
max.partition.fetch.bytes=1048576

# 在一次 poll 调用中返回的记录数
max.poll.records=500
```

```
# 客户端请求超时时间，如果超过这个时间没有收到应答，则请求超时失败
request.timeout.ms=305000
```

接入低版本自建 Kafka

最近更新时间：2024-10-10 17:36:02

CKafka 兼容0.9及以上的生产/消费接口（目前可以直接购买的版本包括 2.4.1、2.8.1、3.2.3 版本），如果接入低版本（例如0.8版本）的自建 Kafka，您需要对接口进行相应改造。本文将从生产端和消费端对比0.8版本 Kafka 和高版本 Kafka，并提供改造方式。

Kafka Producer

概述

Kafka 0.8.1版本中，Producer API 被重写。该客户端为官方推荐版本，其拥有更好的性能和更多的功能，社区将维护新版本的 Producer API。

新旧版本 Producer API 对比

- 新版 Producer API Demo

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:4242");
props.put("acks", "all");
props.put("retries",0);
props.put("batch.size", 16384);
props.put("linger.ms", 1);
props.put("buffer.memory", 33554432);
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
Producer<String, String> producer = new KafkaProducer<>(props);
producer.send(new ProducerRecord<String, String>("my-topic", Integer.toString(0), Integer.toString(0)));
producer.close();
```

- 旧版 Producer API Demo

```
Properties props = new Properties();
props.put("metadata.broker.list", "broker1:9092");
props.put("serializer.class", "kafka.serializer.StringEncoder");
props.put("partitioner.class", "example.producer.SimplePartitioner");
props.put("request.required.acks", "1");
ProducerConfig config = new ProducerConfig(props);
Producer<String, String> producer = new Producer<String, String>(config);
KeyedMessage<String, String> data = new KeyedMessage<String, String>("page_visits", ip, msg);
producer.send(data);
producer.close();
```

可以看出新旧版本的使用方法基本一致，只有一些参数的配置不同，改造代价不大。

兼容性说明

对于 Kafka 而言，0.8.x 版本的 Producer API 都可以顺利接入 CKafka，无需改造。推荐使用新版 Kafka Producer API。

Kafka Consumer

概述

开源 Apache Kafka 0.8版本中提供了两种消费者 API，分别为：

- High Level Consumer API（屏蔽配置细节）
- Simple Consumer API（配置细节支持指定）

Kafka 0.9.x 版本引入了 New Consumer，其融合了 Old Consumer（0.8版本）两种 Consumer API 的特性，减轻了 ZooKeeper 的负载。
因此下文给出了0.8版本 Consumer 转换为0.9版本 New Consumer 的方式。

新旧版本 Consumer API 对比

0.8版本 Consumer API

- **High Level Consumer API**（参见 Demo）
  如果您只需要数据而不考虑消息 offset 相关的处理时，High Level API 可以满足一般性消费要求。High Level Consumer API 围绕着 Consumer Group 逻辑概念展开，屏蔽 Offset 管理、具有 Broker 异常处理、Consumer 负载均衡功能。使开发者可以快速上手 Consumer 客户端。
  在使用 High Level Consumer 时需要注意以下几点：
- 如果消费线程大于 Partition 个数，某些消费线程将无法获得数据。
- 如果 Partition 个数大于线程数目，某些线程会消费多个 Partition。

- Partition 和消费者变动会影响 Rebalance。
- **Low Level Consumer API**（参见 Demo）
  如果使用者关心消息的 offset 并且希望进行重复消费或者跳读等功能、又或者希望指定某些 partition 进行消费时和确保更多消费语义时推荐使用 Low Level Consumer API。但是使用者需要自己处理 Offset 以及 Broker 的异常情况。
  在使用 Low Level Consumer 时需要注意以下几点：
- 自行跟踪维护 Offset，控制消费进度。
- 查找 Topic 相应 Partition 的 Leader，以及处理 Partition 变更情况。

0.9版本 New Consumer API

Kafka 0.9.x 版本引入了 New Consumer，其融合了 Old Consumer 两种 Consumer API 的特性，同时提供消费者的协调(高级 API)和 lower-level 访问，并构建自定义的消费策略。New Consumer 还简化了消费者客户端，引入中心 Coordinator，解决分别连接 ZooKeeper 产生的 Herd Effect 和 Split Brain 问题，同时也减轻了 ZooKeeper 的负载。

优势：

- Coordinator 引入
  当前版本的 High Level Consumer 存在 Herd Effect 和 Split Brain 的问题。将失败探测和 Rebalance 的逻辑放到一个高可用的中心 Coordinator，那么这两个问题即可解决。同时还可很大程度的减少 ZooKeeper 的负载。
- 允许自己分配 Partition
  为了保持本地每个分区的一些状态不变，所以需要将 Partition 的映射也保持不变。另外一些场景是为了让 Consumer 与地域相关的 Broker 关联。
- 允许自己管理 Offset
  可以根据自己需要去管理 Offset，实现重复、跳跃消费等语意。
- Rebalance 后触发用户指定的回调
- 非阻塞式 Consumer API

新旧版本 Consumer API 功能对比

种类	引入版本	Offset 自动保存	Offset 自行管理	自动进行异常处理	Rebalance 自动处理	Leader 自动查找	优缺点
High Level Consumer	Before 0.9	支持	不支持	支持	支持	支持	Herd Effect 和 Split Brain
Simple Consumer	Before 0.9	不支持	支持	不支持	不支持	不支持	需要处理多种异常情况
New Consumer	After 0.9	支持	支持	支持	支持	支持	成熟，当前版本推荐

Old Consumer 转换 New Consumer

- New Consumer

```
//config中主要变化是 ZooKeeper 参数被替换了
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "test");
props.put("enable.auto.commit", "true");
props.put("auto.commit.interval.ms", "1000");
props.put("session.timeout.ms", "30000");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
// 相比old consumer 而言，这里创建消费者更加简单
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("foo", "bar"));
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
       System.out.printf("offset = %d, key = %s, value = %s", record.offset(), record.key(), record.value());}
```

- Old Consumer（High Level）

```
//  old consumer 需要 ZooKeeper
Properties props = new Properties();
props.put("zookeeper.connect", "localhost:2181");
```

```
props.put("group.id", "test");
props.put("auto.commit.enable", "true");
props.put("auto.commit.interval.ms", "1000");
props.put("auto.offset.reset", "smallest");
ConsumerConfig config = new ConsumerConfig(props);
// 需要创建connector
ConsumerConnector connector = Consumer.createJavaConsumerConnector(config);
// 创建message stream
Map<String, Integer> topicCountMap = new HashMap<String, Integer>();
topicCountMap.put("foo", 1);
Map<String, List<KafkaStream<byte[], byte[]>>> streams =
 connector.createMessageStreams(topicCountMap);
// 获取数据
KafkaStream<byte[], byte[]> stream = streams.get("foo").get(0);
ConsumerIterator<byte[], byte[]> iterator = stream.iterator();
MessageAndMetadata<byte[], byte[]> msg = null;
while (iterator.hasNext()) {
     msg = iterator.next();
     System.out.println(//
             " group " + props.get("group.id") + //
             ", partition " + msg.partition() + ", " + //
              new String(msg.message()));
}
```

可以看到，改造成 New Consumer 编写更加简单，最主要的变化是将 ZooKeeper 参数的输入替代成了 Kafka 地址输入。同时，New Consumer 也增加了与 Coodinator 交互的参数配置，一般情况下使用默认配置就足够。

兼容性说明

CKafka 与开源社区高版本的 Kafka 一致，支持重写后的 New Consumer API，屏蔽了 Consumer 客户端与 Zookeeper 的交互（Zookeeper不再向用户暴露）。New Consumer 解决原有与 Zookeeper 直接交互的 Herd Effect 和 Split Brain 问题，以及融合了原有 Old Consumer 的特性，使消费环节更加可靠。

CKafka 版本选择建议

最近更新时间：2024-12-23 15:12:42

本文为您介绍腾讯云 CKafka 和社区版 Kafka 的兼容性，帮助您在使用腾讯云 CKafka 时根据业务需求选择更加适合您的版本。

概述

社区版 Kafka 目前共演进了0.7.x到3.3.x大概30个版本，从消息队列的角度可分为四个阶段：0.x、1.x、2.x、3.x。目前腾讯云针对这三个社区发展阶段提供了四个对应版本：0.10、1.1、2.4、2.8、3.2，基本覆盖了用户使用的主流 Kafka 版本。
其中1.x和2.x这两个大版本主要是对 Kafka Streams 的优化和改进，在消息引擎方面并未引入太多的重大功能特性（2.x在事务特性方面有较大改进）。Kafka Streams 在2.x版本有较大改进，如果您是这些特性的用户，请至少选择2.x的版本。3.x版本在KRaft、mirrormaker2等方面有较大改进，详情可以参考社区 release notes 。

兼容性说明

腾讯云 CKafka 完美兼容社区 Kafka，其中高版本和低版本是完全向下兼容的。例如：自建 0.10 版本的 Kafka，在云上选择0.10、1.1.1、2.4.1版本的 CKafka 均可；如果自建是高版本，不建议选择低版本（因为不确定业务是否使用高版本携带的特性）。
以下是兼容性说明：

CKafka 版本	可兼容社区版本	兼容性
0.10.2（已停售）	≤ 0.10.x	100%
1.1.1（已停售）	≤ 1.1.x	100%
2.4.1	≤ 2.4.x	100%
2.8.1	≤ 2.8.x	100%
3.2.3	≤ 3.2.x	100%

关于 CKafka2.4.1 版本说明
CKafka 上线2.4版本时，社区的稳定分支为2.4.1版，后来社区有一个开发分支2.4.2版本，进行一些修复合入后，定位为2.4.2版本。后续社区删除了2.4.2版本，最终社区上没有2.4.2版本，因此 CKafka 之前显示的2.4.2版本和现在显示的2.4.1版本对齐。

CKafka 版本选择建议

- 如果是自建 Kafka 上云，建议选择对应的大版本即可。例如：自建 Kafka 是1.1.0版本，则选择 CKafka 的1.1版本。
- 当在云上找不到对应的版本时，建议向上选择版本。例如：自建是1.0.0版本，则建议使用1.1.1版本；自建是0.11.x版本，则建议使用版本1.1.1（因为 Broker 的每个版本是向下兼容的）。

CKafka 数据可靠性说明

最近更新时间：2024-11-06 17:29:42

本文将分别通过生产端、服务端（CKafka）和消费端介绍影响消息队列 CKafka 版可靠性的因素，并提供对应的解决方法。

生产端数据丢失如何处理?

数据丢失原因

生产者将数据发送到消息队列 CKafka 版时，数据可能因为网络抖动而丢失，此时消息队列 CKafka 版未收到该数据。可能情况：

- 网络负载高或者磁盘繁忙时，生产者又没有重试机制。
- 磁盘超过购买规格的限制，例如实例磁盘规格为9000GB，在磁盘写满后未及时扩容，会导致数据无法写入到消息队列 CKafka 版。
- 突发或持续增长峰值流量超过购买规格的限制，例如实例峰值吞吐规格为100MB/s，在长时间峰值吞吐超过限制后未及时扩容，会导致数据写入消息队列 CKafka 版变慢，生产者有排队超时机制时，导致数据无法写入到消息队列 CKafka 版。

解决方法

- 生产者对自己重要的数据，开启失败重试机制。
- 针对磁盘使用，在配置实例时设置好监控和 告警策略 ，可以做到事先预防。
  遇到磁盘写满时，可以在控制台及时升配（消息队列 CKafka 版非独占实例间升配为平滑升配不停机且也可以单独升配磁盘）或者通过修改消息保留时间降低磁盘存储。
- 为了尽可能减少生产端消息丢失，您可以通过 buffer.memory 和 batch.size（以字节为单位）调优缓冲区的大小。缓冲区并非越大越好，如果由于某种原因生产者宕机了，那么缓冲区存在的数据越多，需要回收的垃圾越多，恢复就会越慢。应该**时刻注意生产者的生产消息数情况、平均消息大小**等（消息队列 CKafka 版监控中有丰富的监控指标）。
- 配置生产端 ACK
  当 producer 向 leader 发送数据时，可以通过 request.required.acks 参数以及 min.insync.replicas 设置数据可靠性的级别。
- 当 acks = 1时（默认值），生产者在 ISR 中的 leader 已成功收到数据可以继续发送下一条数据。如果 leader 宕机，由于数据可能还未来得及同步给其 follower，则会丢失数据。
- 当 acks = 0时，生产者不等待来自 broker 的确认就发送下一条消息。这种情况下数据传输效率最高，但数据可靠性最低。

> ⚠ **注意：**
> 当生产端配置 acks = 0 时，如果当前实例被限流，为了保护服务端能正常提供服务，服务端会主动关闭与客户端的连接。

- 当 acks = -1或者 all 时，生产者需要等待 ISR 中的所有 follower 都确认接收到消息后才能发送下一条消息，可靠性最高。
  即使按照上述配置 ACK，也不能保证数据不丢，例如，当 ISR 中只有 leader 时（ISR 中的成员由于某些情况会增加也会减少，最少时只剩一个 leader），此时会变成 acks = 1的情况。所以需要同时在配合 min.insync.replicas 参数（此参数可以在消息队列 CKafka 版控制台 Topic 配置开启高级配置中进行配置），min.insync.replicas 表示在 ISR 中最小副本的个数，默认值是1，当且仅当 acks = -1或者 all 时生效。

建议配置的参数值

此参数值仅供参考，实际数值需要依业务实际情况而定。

- 重试机制：`message.send.max.retries=3;retry.backoff.ms=10000;`
- 高可靠的保证：`request.required.acks=-1;min.insync.replicas=2;`
- 高性能的保证：`request.required.acks=0;`
- 可靠性+性能：`request.required.acks=1;`

服务端（CKafka）数据丢失如何处理?

数据丢失原因

- partition 的 leader 在未完成副本数 followers 的备份时就宕机，即使选举出了新的 leader 但是数据因为未来得及备份就丢失。
- 开源 Kafka 的落盘机制为异步落盘，也就是数据是先存在 PageCache 中的，当还没有正式落盘时，broker 出现断开连接或者重启或者故障时，PageCache 上的数据由于没有来得及落盘进而丢失。
- 磁盘故障导致已经落盘的数据丢失。

解决方法

- 开源 Kafka 是多副本的，官方推荐通过副本来保证数据的完整性，此时如果是多副本，同时出现多副本多 broker 同时挂掉才会丢数据，比单副本数据的可靠性高很多，所以消息队列 CKafka 版强制 Topic 是双副本，可配置3副本。
- 消息队列 CKafka 版服务配置了更合理的参数 log.flush.interval.messages 和 log.flush.interval.ms，对数据进行刷盘。
- 消息队列 CKafka 版对磁盘做了特殊处理，保证部分磁盘损坏时也不会影响数据的可靠性。

建议配置的参数值

非同步状态的副本可以选举为 leader：`unclean.leader.election.enable=false // 关闭`

消费端数据丢失如何处理?

数据丢失原因

- 还未真正消费到数据就提交 commit 了 offset，若过程中消费者挂掉，但 offset 已经刷新，消费者错过了一条数据，需要消费分组重新设置 offset 才能找回数据。
- 消费速度和生产速度相差太久，而消息保存时间太短，导致消息还未及时消费就被过期删除。

解决方法

- 合理配置参数 auto.commit.enable，等于 true 时表示自动提交。建议使用定时提交，避免频繁 commit offset。
- 监控消费者的情况，正确调整数据的保留时间。监控当前消费 offset 以及未消费的消息条数，并配置告警，防止由于消费速度过慢导致消息过期删除。

数据丢失排查方案

在本地打印分区 partition 和偏移量 offset 进行排查

打印信息代码如下：

```
Future<RecordMetadata> future = producer.send(new ProducerRecord<>(topic, messageKey, messageStr));
RecordMetadata recordMetadata = future.get();
log.info("partition: {}", recordMetadata.partition());
log.info("offset: {}", recordMetadata.offset());
```

- 如果能够打印出 partition 和 offset，则表示当前发送的消息在服务端已经被正确保存。此时可以通过消息查询的工具去查询相关位点的消息即可。
- 如果打印不出 partition 和 offset，则表示消息没有被服务端保存，客户端需要重试。

连接器
数据库变更订阅
MongoDB 数据订阅

最近更新时间：2024-10-09 17:19:41

简介

MongoDB Kafka Connector 允许监控一个 Mongo 实例内的所有数据库（database）或单个数据库，也允许监控某个数据库内的所有集合（collection）或单个集合。将 Mongo 的修改信息生成修改事件消息，以消息流的方式提交给 kafka 的 topic。客户端应用可以通过消费对应 topic 中的消息来对数据库修改事件进行处理，从而达到监控特定数据库的目的。
本文档是对 Mongo 官方文档的归纳和整理，详情参见 MongoDB Change Events 。

事件格式

以下 JSON 框架展示了所有修改事件消息中可能出现的字段：

```
{
   _id : { <BSON Object> },
   "operationType" : "<operation>",
   "fullDocument" : { <document> },
   "ns" : {
      "db" : "<database>",
      "coll" : "<collection>"
   },
   "to" : {
      "db" : "<database>",
      "coll" : "<collection>"
   },
   "documentKey" : { "_id" : <value> },
   "updateDescription" : {
      "updatedFields" : { <document> },
      "removedFields" : [ "<field>", ... ],
      "truncatedArrays" : [
         { "field" : <field>, "newSize" : <integer> },
         ...
      ]
   },
   "clusterTime" : <Timestamp>,
   "txnNumber" : <NumberLong>,
   "lsid" : {
      "id" : <UUID>,
      "uid" : <BinData>
   }
}
```

其中部分字段可能只在特定的事件类型中才会出现，下表对相应字段及其含义进行了描述。

Field	Type	Description
_id	document	一个用来唯一标识事件的 BSON 对象。 _id 对象的格式如下：{ "_data" : <BinData|hex string>} 。_data 的类型取决于 MongoDB 的版本，可通过 Resume Tokens 查看完整的_data类型介绍。
operationType	string	触发修改事件的操作类型，具体包括以下 8 种：insertdeletereplaceupdatedroprenamedropDatabaseinvalidate
fullDocument	document	表示被新增（ insert ）, 替换（ replace ）, 删除（ delete ）, 更新（ update ）操作所影响的文档。对于 insert 和 replace 操作，该字段表示新增的文档。对于 delete 操作，该字段缺省表示文档已经不存在。对于 update 操作，只有配置了 fullDocument 为 updateLookup 时才会显示。
ns	document	命名空间（ namespace ），由 database 和 collection 构成。
ns.db	string	数据库名称。

ns.coll	string	集合名称。对于 dropDatabase 操作，该字段缺省。
to	document	当操作类型为 rename 时，表示新的集合名称。该字段对其他操作是缺省的。
to.db	string	新的数据库的名称。
to.coll	string	新的集合名称。
documentKey	document	操作修改的文档的 ID。
updateDescription	document	一个用来描述被更新操作（update operation）修改的字段的文档。该字段仅当事件对应的操作为 update 时才有。
updateDescription.updatedFields	document	包含被更新操作修改的字段，字段的 value 值为更新后的值。
updateDescription.removedFields	array	包含被更新操作删除的字段。
updateDescription.truncatedArrays	array	其中记录了使用以下一个或多个基于 pipeline 的更新执行的数组截断： $addFields $set $replaceRoot $replaceWith
updateDescription.truncatedArrays.field	string	被删除的字段。
updateDescription.truncatedArrays.newSize	integer	truncated array 中的元素个数。
clusterTime	Timestamp	oplog 与事件关联的时间戳。对于涉及 多文档事务，关联的事件的 clusterTime 值是相同的。
txnNumber	NumberLong	事务 ID。仅当操作是 多文档事务 时出现。
lsid	Document	与事务关联的 session 的 ID，仅当操作是 多文档事务 时出现。

事件列表

新增事件（insert event）

```
{
   _id: { < Resume Token > },
   operationType: 'insert',
   clusterTime: <Timestamp>,
   ns: {
      db: 'engineering',
      coll: 'users'
   },
   documentKey: {
      userName: 'alice123',
      _id: ObjectId("599af247bb69cd8996xxxxxx")
   },
   fullDocument: {
      _id: ObjectId("599af247bb69cd8996xxxxxx"),
      userName: 'alice123',
      name: 'Alice'
   }
}
```

其中 documentKey 字段同时包含了 _id 和 username 字段。表示 engineering.users 集合是分片的，shard key 为 username 和 _id。

更新事件（update event）

```
{
   _id: { < Resume Token > },
   operationType: 'update',
   clusterTime: <Timestamp>,
```

```
   ns: {
      db: 'engineering',
      coll: 'users'
   },
   documentKey: {
      _id: ObjectId("58a4eb4a30c75625e0xxxxxx")
   },
   updateDescription: {
      updatedFields: {
         email: 'alice@10gen.com'
      },
      removedFields: ['phoneNumber'],
      truncatedArrays: [ {
         "field" : "vacation_time",
         "newSize" : 36
      } ]
   }
}
```

以下例子展示了 `update` event 配置了 `fullDocument : updateLookup` 选项的消息内容：

```
{
   _id: { < Resume Token > },
   operationType: 'update',
   clusterTime: <Timestamp>,
   ns: {
      db: 'engineering',
      coll: 'users'
   },
   documentKey: {
      _id: ObjectId("58a4eb4a30c75625e0xxxxxx")
   },
   updateDescription: {
      updatedFields: {
         email: 'alice@10gen.com'
      },
      removedFields: ['phoneNumber'],
      truncatedArrays: [ {
         "field" : "vacation_time",
         "newSize" : 36
      } ]
   },
   fullDocument: {
      _id: ObjectId("58a4eb4a30c75625e0xxxxxx"),
      name: 'Alice',
      userName: 'alice123',
      email: 'alice@10gen.com',
      team: 'replication'
   }
}
```

替换事件（replace event）

```
{
   _id: { < Resume Token > },
   operationType: 'replace',
   clusterTime: <Timestamp>,
   ns: {
      db: 'engineering',
      coll: 'users'
   },
   documentKey: {
      _id: ObjectId("599af247bb69cd8996xxxxxx")
   },
```

```
    fullDocument: {
       _id: ObjectId("599af247bb69cd8996xxxxxx"),
       userName: 'alice123',
       name: 'Alice'
    }
}
```

`replace` 操作是通过两步操作实现的：

- 删除原 `documentKey` 对应的文档
- 根据一样的 `documentkey` 插入新的文档

基于 `replace` 事件的 `fullDocument` 字段表示的是插入后的新文档。

删文档事件（delete event）

```
{
   _id: { < Resume Token > },
   operationType: 'delete',
   clusterTime: <Timestamp>,
   ns: {
      db: 'engineering',
      coll: 'users'
   },
   documentKey: {
      _id: ObjectId("599af247bb69cd8996xxxxxx")
   }
}
```

对于删除文档事件的消息，`fullDocument` 字段缺省。

删集合事件（drop event）

```
{
   _id: { < Resume Token > },
   operationType: 'drop',
   clusterTime: <Timestamp>,
   ns: {
      db: 'engineering',
      coll: 'users'
   }
}
```

当一个集合被删除时会触发该事件，同时会导致订阅了该集合的 connector 产生一个无效事件（invalidate event）。

改名事件（rename event）

```
{
   _id: { < Resume Token > },
   operationType: 'rename',
   clusterTime: <Timestamp>,
   ns: {
      db: 'engineering',
      coll: 'users'
   },
   to: {
      db: 'engineering',
      coll: 'people'
   }
}
```

当一个集合名称被更改时会触发该事件，同时会导致订阅了该集合的 connector 产生一个无效事件（invalidate event）。

删库事件（drop database event）

```
{
   _id: { < Resume Token > },
   operationType: 'dropDatabase',
   clusterTime: <Timestamp>,
   ns: {
      db: 'engineering'
   }
}
```

当一个数据库被删除时会触发该事件，同时会导致订阅了该集合的 connector 产生一个无效事件（invalidate event）。
在生成数据库删除事件（dropDatabase）之前，会为数据库中的每一个集合生成一个集合删除事件（drop event）。

无效事件（invalidate event）

```
{
   _id: { < Resume Token > },
   operationType: 'invalidate',
   clusterTime: <Timestamp>
}
```

- 对于订阅了一个集合（collection）的 connector，drop event，rename event 或 dropDatabase event 这类会对该集合产生影响的事件都会产生一个无效事件。
- 对于订阅了一个数据库（database）的 connector，dropDatabase event 会产生一个无效事件。

MySQL 数据订阅

最近更新时间：2025-01-17 10:31:52

简介

> **注意：**
> MySQL 数据订阅 任务已停止新增，即将下线。欢迎使用其他正在开放的任务类型。

MySQL 通过一种二进制日志（binlog）来按序记录所有提交给数据库的操作，包括对表结构的修改以及对表中数据的修改。MySQL 通过 binlog 来进行备份或恢复数据。
Debezium MySQL connector 通过读取 binlog 来生成行级（row-level）的数据库修改事件（event），包括 INSERT、UPDATE 和 DELETE，并将事件发送给 kafka 中相应的 topic 。客户端应用可以通过消费对应 topic中的消息来对数据库修改事件进行处理，从而达到监控特定数据库的目的。
支持订阅的 SQL 操作：

操作类型	支持的SQL操作
DML	INSERT、UPDATE、DELETE
DDL	CREATE DATABASE、DROP DATABASE、CREATE TABLE、ALTER TABLE、DROP TABLE、RENAME TABLE

本文档是根据 Debezium 官方文档进行整理和归纳而来。详情参见 Debezium connector for MySQL 。

事件格式

Debezium MySQL connector 针对每一次插入、更新、删除操作生成数据变更事件。每一个事件（event）在作为消息提交给 kafka 的主题(Topic)，Topic里每条消息包含 key 和 value 两部分，示例如下：

Kafka 每条消息的key 和 value 都包含 schema 和 payload 两个字段。格式如下：

```
{
 "schema": {
   ...
  },
 "payload": {
   ...
 }
}
```

key 字段说明：

Item	Field name	Description
1	schema	schema 字段描述了 key 的 payload 字段的结构，即它描述了被修改的表的主键（primary key）结构，如果表没有主键，则描述其唯一约束（unique key）的结构。
2	payload	payload 字段的结构和第一个 schema 中描述的相同，包含了被修改的行的键值。

value 字段说明：

Item	Field name	Description
1	schema	schema 字段描述了 value 的payload 字段的结构，即描述了被修改行的字段结构。这个字段通常是一个嵌套结构的字段。
2	payload	payload 字段的结构和第二个 schema 中定义的相同，它包含被修改行的真实数据。

事件消息 key

不同类型事件的消息都有一样的 key 结构，下面给出一个示例，一个修改事件的 key 包含被修改的表的主键结构以及对应行的实际主键值。

```
CREATE TABLE customers (
  id INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
  first_name VARCHAR(255) NOT NULL,
  last_name VARCHAR(255) NOT NULL,
  email VARCHAR(255) NOT NULL UNIQUE KEY
) AUTO_INCREMENT=1001;
```

每一个捕获 customers 表修改操作的事件 key 中的 schema 都相同。该操作对应的事件消息的 key 如下所示（JSON表示）：

```
{
 "schema": {
    "type": "struct",
    "name": "mysql-server-1.inventory.customers.Key",
    "optional": false,
    "fields": [
      {
        "field": "id",
        "type": "int32",
        "optional": false
      }
    ]
  },
 "payload": {
    "id": 1001
  }
}
```

Item	Field name	Description
1	schema	Schema 描述了 payload 中的结构。
2	mysql-server-1.inventory.customers.Key	schema 的名称格式为 *connector-name*.*database-name*.*table-name*.Key。在这个例子中: mysql-server-1 是生成事件的connector的名字。inventory 是对应数据库的名字。 customers 是表的名字。
3	optional	表示字段是否是可选项。
4	fields	列出了所有 payload 中包含的字段及其结构, 包括字段名、字段类型、以及是否可选。
5	payload	包含被修改行的主键。在例子中仅包含一个字段名为 id 的主键值： 1001。

DML 事件

前面介绍了一个事件消息的 key 的结构，不同类型事件的 key 结构是相同的。本节列举了不同的事件类型，介绍了这些事件类型各自的 value 结构。

新增事件（create events）

下面这个例子展示了在表中新增数据的时候 connector 生成的事件消息的 value 部分：

```
{
  "schema": {
    "type": "struct",
    "fields": [
      {
        "type": "struct",
        "fields": [
```

```
        {
          "type": "int32",
          "optional": false,
          "field": "id"
        },
        {
          "type": "string",
          "optional": false,
          "field": "first_name"
        },
        {
          "type": "string",
          "optional": false,
          "field": "last_name"
        },
        {
          "type": "string",
          "optional": false,
          "field": "email"
        }
      ],
      "optional": true,
      "name": "mysql-server-1.inventory.customers.Value",
      "field": "before"
    },
    {
      "type": "struct",
      "fields": [
        {
          "type": "int32",
          "optional": false,
          "field": "id"
        },
        {
          "type": "string",
          "optional": false,
          "field": "first_name"
        },
        {
          "type": "string",
          "optional": false,
          "field": "last_name"
        },
        {
          "type": "string",
          "optional": false,
          "field": "email"
        }
      ],
      "optional": true,
      "name": "mysql-server-1.inventory.customers.Value",
      "field": "after"
    },
    {
      "type": "struct",
      "fields": [
        {
          "type": "string",
          "optional": false,
          "field": "version"
        },
        {
          "type": "string",
          "optional": false,
          "field": "connector"
```

```
      },
      {
        "type": "string",
        "optional": false,
        "field": "name"
      },
      {
        "type": "int64",
        "optional": false,
        "field": "ts_ms"
      },
      {
        "type": "boolean",
        "optional": true,
        "default": false,
        "field": "snapshot"
      },
      {
        "type": "string",
        "optional": false,
        "field": "db"
      },
      {
        "type": "string",
        "optional": true,
        "field": "table"
      },
      {
        "type": "int64",
        "optional": false,
        "field": "server_id"
      },
      {
        "type": "string",
        "optional": true,
        "field": "gtid"
      },
      {
        "type": "string",
        "optional": false,
        "field": "file"
      },
      {
        "type": "int64",
        "optional": false,
        "field": "pos"
      },
      {
        "type": "int32",
        "optional": false,
        "field": "row"
      },
      {
        "type": "int64",
        "optional": true,
        "field": "thread"
      },
      {
        "type": "string",
        "optional": true,
        "field": "query"
      }
    ],
    "optional": false,
    "name": "io.debezium.connector.mysql.Source",
```

```
        "field": "source"
      },
      {
        "type": "string",
        "optional": false,
        "field": "op"
      },
      {
        "type": "int64",
        "optional": true,
        "field": "ts_ms"
      }
    ],
    "optional": false,
    "name": "mysql-server-1.inventory.customers.Envelope"
  },
  "payload": {
    "op": "c",
    "ts_ms": 1465491411815,
    "before": null,
    "after": {
      "id": 1004,
      "first_name": "Anne",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "source": {
      "version": "1.9.3.Final",
      "connector": "mysql",
      "name": "mysql-server-1",
      "ts_ms": 0,
      "snapshot": false,
      "db": "inventory",
      "table": "customers",
      "server_id": 0,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 154,
      "row": 0,
      "thread": 7,
      "query": "INSERT INTO customers (first_name, last_name, email) VALUES ('Anne', 'Kretchmar',
'annek@noanswer.org')"
    }
  }
}
```

Item	Field name	Description
1	schema	Schema 描述了 payload 中的结构，其中 schema 中的 fields 字段为一个数组，表示 payload 字段包含了多个字段，数组的每个元素是对 payload 中相应字段结构的描述信息。
2	field	每一个 fields 中的元素都包含一个 field 字段，该字段表示 payload 中对应字段的名称。在示例中包括 before、after、source等。
3	type	表示字段的类型，如整型（int）、字符串（string）等.
4	mysql-server-1.inventory.customers.Value	表示该字段是 mysql-server-1 连接器生成的针对 inventory 数据库的 customers 表的 value 部分信息。
	io.debezium.connector.mysql.Source	该名称和特定的 connector 绑定，由该 connector 生成的事件该名称都相同。

6	payload	包含修改事件中具体被修改的数据，包括修改前（before 字段）和修改后（after 字段）的数据，以及一些 connector 的元数据信息（source 字段）。
7	op	表示导致事件生成的修改操作的类型，例子中的 c 表示 修改操作创建了一个新的行。c = createu = updated = deleter = read (仅 snapshots)
8	source	source 字段是一个描述事件元数据的字段。它包含的一些字段可以用来与其他事件做比较，例如比较事件生成的顺序、事件是否属于同一个事务等。该字段包含以下元数据信息： Debezium versionConnector name binlog name where the event was recorded binlog positionRow within the eventIf the event was part of a snapshotName of the database and table that contain the new rowID of the MySQL thread that created the event (non-snapshot only)MySQL server ID (if available)Timestamp for when the change was made in the database
9	query	修改操作的原始 SQL 语句。

更新事件（update events）

下面这个例子展示了更新操作生成的事件的 value 部分：

```
{
  "schema": { ... },
  "payload": {
    "before": {
      "id": 1004,
      "first_name": "Anne",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": {
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "source": {
      "version": "1.9.3.Final",
      "name": "mysql-server-1",
      "connector": "mysql",
      "name": "mysql-server-1",
      "ts_ms": 1465581029100,
      "snapshot": false,
      "db": "inventory",
      "table": "customers",
      "server_id": 223344,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 484,
      "row": 0,
      "thread": 7,
      "query": "UPDATE customers SET first_name='Anne Marie' WHERE id=1004"
    },
    "op": "u",
    "ts_ms": 1465581029523
  }
}
```

其中 schema 字段和新增操作的事件相同，而 payload 部分有所不同，在新增事件中，before 字段为 null，表示没有原始数据，而更新事件中包含了更新前（before）和更新后（after）的数据。

删除事件（delete events）

下面这个例子展示了删除操作生成的事件的 value 部分：

```
{
  "schema": { ... },
  "payload": {
    "before": {
```

```
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": null,
    "source": {
      "version": "1.9.3.Final",
      "connector": "mysql",
      "name": "mysql-server-1",
      "ts_ms": 1465581902300,
      "snapshot": false,
      "db": "inventory",
      "table": "customers",
      "server_id": 223344,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 805,
      "row": 0,
      "thread": 7,
      "query": "DELETE FROM customers WHERE id=1004"
    },
    "op": "d",
    "ts_ms": 1465581902461
  }
}
```


其中 schema 字段和新增操作的事件相同，而 payload 部分有所不同，删除事件中包含了更新前（before）的数据，但更新后（after）的数据为 null，表示数据已删除。

更新主键（primary key updates）

如果一个操作修改了数据表中某行的主键，那么 connector 会生成一条**删除事件**来表示原主键对应的数据行删除，同时生成一条**新增事件**来表示插入的新主键对应的行。每一条消息的 header 都会和相应的 key 关联。官方描述如下：

- The `DELETE` event record has `__debezium.newkey` as a message header. The value of this header is the new primary key for the updated row.
- The `CREATE` event record has `__debezium.oldkey` as a message header. The value of this header is the previous (old) primary key that the updated row had.

DDL 事件

创建数据库（create database）

下面这个例子展示了**创建数据库**操作生成的事件的 value 部分：


```
{
  "source" : {
    "server" : "dip_source"
  },
  "position" : {
    "ts_sec" : 1655812326,
    "file" : "mysql-bin.000006",
    "pos" : 26063,
    "gtids" : "b24176f2-5409-11ec-80d4-b8599fe5c6ea:1-78",
    "snapshot" : true
  },
  "databaseName" : "dip_test",
  "ddl" : "CREATE DATABASE `dip_test` CHARSET utf8mb4 COLLATE utf8mb4_0900_ai_ci",
  "tableChanges" : [ ]
}
```


其中 position 的内容为记录 binlog 文件，消费偏移量等信息。ddl 字段为触发事件的 sql 语句。

删除数据库（drop database）

下面这个例子展示了**删除数据库**操作生成的事件的 value 部分：

```
{
  "source" : {
    "server" : "dip_source"
  },
  "position" : {
    "ts_sec" : 1655812326,
    "file" : "mysql-bin.000006",
    "pos" : 26063,
    "gtids" : "b24176f2-5409-11ec-80d4-b8599fe5c6ea:1-78",
    "snapshot" : true
  },
  "databaseName" : "dip_test",
  "ddl" : "DROP DATABASE IF EXISTS `dip_test`",
  "tableChanges" : [ ]
}
```

其中 position 的内容为记录 binlog 文件，消费偏移量等信息。ddl 字段为触发事件的 sql 语句。

创建表（create table）

下面这个例子展示了**创建表**操作生成的事件的 value 部分：

```
{
  "source" : {
    "server" : "dip_source"
  },
  "position" : {
    "ts_sec" : 1655812326,
    "file" : "mysql-bin.000006",
    "pos" : 26063,
    "gtids" : "b24176f2-5409-11ec-80d4-b8599fe5c6ea:1-78",
    "snapshot" : true
  },
  "databaseName" : "dip_test",
  "ddl" : "CREATE TABLE `customers` (\n  `id` int NOT NULL AUTO_INCREMENT,\n  `first_name` varchar(255) NOT NULL,\n
`last_name` varchar(255) NOT NULL,\n  `email` varchar(255) NOT NULL,\n  PRIMARY KEY (`id`),\n  UNIQUE KEY `email`
(`email`),\n  KEY `ix_id` (`id`)\n) ENGINE=InnoDB AUTO_INCREMENT=1041 DEFAULT CHARSET=utf8",
  "tableChanges" : [ {
    "type" : "CREATE",
    "id" : "\"dip_test\".\"customers\"",
    "table" : {
      "defaultCharsetName" : "utf8",
      "primaryKeyColumnNames" : [ "id" ],
      "columns" : [ {
        "name" : "id",
        "jdbcType" : 4,
        "typeName" : "INT",
        "typeExpression" : "INT",
        "charsetName" : null,
        "position" : 1,
        "optional" : false,
        "autoIncremented" : true,
        "generated" : true,
        "comment" : null,
        "hasDefaultValue" : false,
        "enumValues" : [ ]
      }, {
        "name" : "first_name",
        "jdbcType" : 12,
        "typeName" : "VARCHAR",
        "typeExpression" : "VARCHAR",
        "charsetName" : "utf8",
        "length" : 255,
```

```
        "position" : 2,
        "optional" : false,
        "autoIncremented" : false,
        "generated" : false,
        "comment" : null,
        "hasDefaultValue" : false,
        "enumValues" : [ ]
      }, {
        "name" : "last_name",
        "jdbcType" : 12,
        "typeName" : "VARCHAR",
        "typeExpression" : "VARCHAR",
        "charsetName" : "utf8",
        "length" : 255,
        "position" : 3,
        "optional" : false,
        "autoIncremented" : false,
        "generated" : false,
        "comment" : null,
        "hasDefaultValue" : false,
        "enumValues" : [ ]
      }, {
        "name" : "email",
        "jdbcType" : 12,
        "typeName" : "VARCHAR",
        "typeExpression" : "VARCHAR",
        "charsetName" : "utf8",
        "length" : 255,
        "position" : 4,
        "optional" : false,
        "autoIncremented" : false,
        "generated" : false,
        "comment" : null,
        "hasDefaultValue" : false,
        "enumValues" : [ ]
      } ]
    },
    "comment" : null
  } ]
}
```

其中 position 的内容为记录 binlog 文件，消费偏移量等信息。ddl 字段为触发事件的 sql 语句。columns 字段记录了新增表的不同字段的定义信息。

修改表（alter table）

下面这个例子展示了**修改表**操作生成的事件的 value 部分：

```
{
  "source" : {
    "server" : "1307446078-a123"
  },
  "position" : {
    "transaction_id" : null,
    "ts_sec" : 1655782153,
    "file" : "mysql-bin.000005",
    "pos" : 1218,
    "gtids" : "ddf040ad-7509-11ec-968b-0c42a1eda2e9:1-8",
    "server_id" : 183277
  },
  "databaseName" : "test",
  "ddl" : "ALTER TABLE `user` ADD COLUMN `createtime` datetime NULL DEFAULT CURRENT_TIMESTAMP",
  "tableChanges" : [ {
    "type" : "ALTER",
    "id" : "\"test\".\"user\"",
    "table" : {
```

```
      "defaultCharsetName" : "utf8",
      "primaryKeyColumnNames" : [ ],
      "columns" : [ {
        "name" : "name",
        "jdbcType" : 1,
        "typeName" : "CHAR",
        "typeExpression" : "CHAR",
        "charsetName" : "utf8",
        "length" : 20,
        "position" : 1,
        "optional" : true,
        "autoIncremented" : false,
        "generated" : false,
        "comment" : null,
        "hasDefaultValue" : true,
        "defaultValueExpression" : "",
        "enumValues" : [ ]
      }, {
        "name" : "age",
        "jdbcType" : 4,
        "typeName" : "INT",
        "typeExpression" : "INT",
        "charsetName" : null,
        "position" : 2,
        "optional" : true,
        "autoIncremented" : false,
        "generated" : false,
        "comment" : null,
        "hasDefaultValue" : true,
        "enumValues" : [ ]
      }, {
        "name" : "createtime",
        "jdbcType" : 93,
        "typeName" : "DATETIME",
        "typeExpression" : "DATETIME",
        "charsetName" : null,
        "position" : 3,
        "optional" : true,
        "autoIncremented" : false,
        "generated" : false,
        "comment" : null,
        "hasDefaultValue" : true,
        "defaultValueExpression" : "1970-01-01 00:00:00",
        "enumValues" : [ ]
      } ]
    },
    "comment" : null
  } ]
}
```


其中 position 的内容为记录 binlog 文件，消费偏移量等信息。ddl 字段为触发事件的 sql 语句。columns 字段记录了被修改的字段的信息。

删除表（drop table）

下面这个例子展示了**删除表**操作生成的事件的 value 部分：


```
{
  "source" : {
    "server" : "dip_source"
  },
  "position" : {
    "ts_sec" : 1655812326,
    "file" : "mysql-bin.000006",
    "pos" : 26063,
    "gtids" : "b24176f2-5409-11ec-80d4-b8599fe5c6ea:1-78",
```

```
    "snapshot" : true
  },
  "databaseName" : "dip_test",
  "ddl" : "DROP TABLE IF EXISTS `dip_test`.`customers`",
  "tableChanges" : [ ]
}
```

其中 position 的内容为记录 binlog 文件，消费偏移量等信息。ddl 字段为触发事件的 sql 语句。

更改表名

下面这个例子展示了**更改**操作生成的事件的 value 部分：

```
{
    "schema": {
        "type": "struct",
        "fields": ...,
        "optional": false,
        "name": "io.debezium.connector.mysql.SchemaChangeValue"
    },
    "payload": {
        "source": {
            "version": "1.9.0.Final",
            "connector": "mysql",
            "name": "task-lzpx4pdo",
            "ts_ms": 1656300979748,
            "snapshot": "false",
            "db": "testDB",
            "sequence": null,
            "table": "t_test",
            "server_id": 170993,
            "gtid": "b24176f2-5409-11ec-80d4-b8599fe5c6ea:80",
            "file": "mysql-bin.000006",
            "pos": 26411,
            "row": 0,
            "thread": null,
            "query": null
        },
        "databaseName": "testDB",
        "schemaName": null,
        "ddl": "rename table test to t_test",
        "tableChanges": [{
            "type": "ALTER",
            "id": "\"testDB\".\"t_test\"",
            "table": {
                "defaultCharsetName": "utf8",
                "primaryKeyColumnNames": ["id"],
                "columns": [{
                    "name": "id",
                    "jdbcType": -5,
                    "nativeType": null,
                    "typeName": "BIGINT",
                    "typeExpression": "BIGINT",
                    "charsetName": null,
                    "length": 20,
                    "scale": null,
                    "position": 1,
                    "optional": false,
                    "autoIncremented": true,
                    "generated": true,
                    "comment": null
                }, {
                    "name": "name",
                    "jdbcType": 12,
                    "nativeType": null,
```

```
                    "typeName": "VARCHAR",
                    "typeExpression": "VARCHAR",
                    "charsetName": "utf8",
                    "length": 20,
                    "scale": null,
                    "position": 2,
                    "optional": true,
                    "autoIncremented": false,
                    "generated": false,
                    "comment": null
                }],
                "comment": null
            }
        }]
    }
}
```

其中 schema 中包含的是对 payload 的内容格式信息，这里省略了部分内容，payload 字段 source 为元数据信息，ddl 字段为触发事件的 sql 语句。columns 为受影响的表的字段。

PostgreSQL 数据订阅

最近更新时间：2024-10-08 15:39:00

简介

Debezium PostgreSQL connector 能够抓取 PostgreSQL 数据库中的行级（row-level）修改操作，并生成相应的修改事件。Debezium PostgreSQL connector 第一次连接 PostgreSQL 服务器时，会对所有数据库生成一个快照（snapshot），然后会持续的抓取提交给数据库的包括新增（insert）、更新（update）、删除（delete）在内的行级修改操作，并生成数据修改事件，将其作为消息提交给 Kafka 的相应 topic 。客户端应用可以通过消费对应 topic中的消息来对数据库修改事件进行处理，从而达到监控特定数据库的目的。
本文档是根据 Debezium 官方文档进行整理和归纳而来。详情参见 Debezium connector for PostgreSQL 。

事件格式

Debezium PostgreSQL connector 针对每一个行级的插入、更新、删除操作生成数据修改事件。每一个事件（event）在作为消息提交给 kafka 的主题（Topic），Topic 里每条消息包含 key 和 value 两部分。示例如下：

Kafka 每条消息的key 和 value 都包含 schema 和 payload 两个字段。格式如下：

```
{
 "schema": {
   ...
  },
 "payload": {
   ...
 }
}
```

key 字段说明：

Item	Field name	Description
1	schema	schema 字段描述了key的payload 字段的结构，即它描述了被修改的表的主键（primary key）结构，如果表没有主键，则描述其唯一约束（unique key）的结构。
2	payload	payload 字段的结构和第一个schema 中描述的相同，包含了被修改的行的键值。

value 字段说明：

Item	Field name	Description
1	schema	schema 字段描述了 value 的payload 字段的结构，即描述了被修改行的字段结构。这个字段通常是一个嵌套结构的字段。
2	payload	payload 字段的结构和 schema 中定义的相同，它包含被修改行的真实数据。

事件消息 key

不同类型事件的消息都有一样的 key 结构，下面给出一个示例，一个修改事件的 key 包含被修改的表的主键结构以及对应行的实际主键值。：

```
CREATE TABLE customers (
  id SERIAL,
  first_name VARCHAR(255) NOT NULL,
```

```
  last_name VARCHAR(255) NOT NULL,
  email VARCHAR(255) NOT NULL,
  PRIMARY KEY(id)
);
```

该操作对应的事件消息的 key 如下所示（JSON 表示）：

```
{
  "schema": {
    "type": "struct",
    "name": "PostgreSQL_server.public.customers.Key",
    "optional": false,
    "fields": [
          {
              "name": "id",
              "index": "0",
              "schema": {
                  "type": "INT32",
                  "optional": "false"
              }
          }
      ]
  },
  "payload": {
      "id": "1"
  },
}
```

Item	Field name	Description
1	schema	Schema 描述了 payload 中的结构
2	PostgreSQL_server.inventory.customers.Key	schema 的名称格式为 *connector-name*.*database-name*.*table-name*.Key。在这个例子中: PostgreSQL_server 是生成事件的 connector 的名字。 inventory 是对应数据库的名字。 customers 是表的名字。
3	optional	表示字段是否是可选项。
4	fields	列出了所有 payload 中包含的字段及其结构, 包括字段名、字段类型、以及是否可选.
5	payload	包含被修改行的主键.。在例子中仅包含一个字段名为 id 的主键值： 1。

事件列表

前面介绍了一个事件消息的 key 的结构，不同类型事件的 key 结构是相同的。本节列举了不同的事件类型，介绍了这些事件类型各自的 value 结构。

新增事件（create events）

下面给出一个 Debezium PostgreSQL connector 针对数据库新增操作生成的消息：

```
{
    "schema": {
        "type": "struct",
        "fields": [
            {
                "type": "struct",
                "fields": [
                    {
                        "type": "int32",
                        "optional": false,
                        "field": "id"
                    },
                    {
                        "type": "string",
                        "optional": false,
                        "field": "first_name"
```

```
                },
                {
                    "type": "string",
                    "optional": false,
                    "field": "last_name"
                },
                {
                    "type": "string",
                    "optional": false,
                    "field": "email"
                }
            ],
            "optional": true,
            "name": "PostgreSQL_server.inventory.customers.Value",
            "field": "before"
        },
        {
            "type": "struct",
            "fields": [
                {
                    "type": "int32",
                    "optional": false,
                    "field": "id"
                },
                {
                    "type": "string",
                    "optional": false,
                    "field": "first_name"
                },
                {
                    "type": "string",
                    "optional": false,
                    "field": "last_name"
                },
                {
                    "type": "string",
                    "optional": false,
                    "field": "email"
                }
            ],
            "optional": true,
            "name": "PostgreSQL_server.inventory.customers.Value",
            "field": "after"
        },
        {
            "type": "struct",
            "fields": [
                {
                    "type": "string",
                    "optional": false,
                    "field": "version"
                },
                {
                    "type": "string",
                    "optional": false,
                    "field": "connector"
                },
                {
                    "type": "string",
                    "optional": false,
                    "field": "name"
                },
                {
                    "type": "int64",
                    "optional": false,
```

```
                    "field": "ts_ms"
                },
                {
                    "type": "boolean",
                    "optional": true,
                    "default": false,
                    "field": "snapshot"
                },
                {
                    "type": "string",
                    "optional": false,
                    "field": "db"
                },
                {
                    "type": "string",
                    "optional": false,
                    "field": "schema"
                },
                {
                    "type": "string",
                    "optional": false,
                    "field": "table"
                },
                {
                    "type": "int64",
                    "optional": true,
                    "field": "txId"
                },
                {
                    "type": "int64",
                    "optional": true,
                    "field": "lsn"
                },
                {
                    "type": "int64",
                    "optional": true,
                    "field": "xmin"
                }
            ],
            "optional": false,
            "name": "io.debezium.connector.postgresql.Source",
            "field": "source"
        },
        {
            "type": "string",
            "optional": false,
            "field": "op"
        },
        {
            "type": "int64",
            "optional": true,
            "field": "ts_ms"
        }
    ],
    "optional": false,
    "name": "PostgreSQL_server.inventory.customers.Envelope"
},
"payload": {
    "before": null,
    "after": {
        "id": 1,
        "first_name": "Anne",
        "last_name": "Kretchmar",
        "email": "annek@noanswer.org"
    },
```

```
        "source": {
            "version": "1.9.3.Final",
            "connector": "postgresql",
            "name": "PostgreSQL_server",
            "ts_ms": 1559033904863,
            "snapshot": true,
            "db": "postgres",
            "sequence": "[\"24023119\",\"24023128\"]"
            "schema": "public",
            "table": "customers",
            "txId": 555,
            "lsn": 24023128,
            "xmin": null
        },
        "op": "c",
        "ts_ms": 1559033904863
    }
}
```

Item	Field name	Description
1	schema	Schema 描述了 payload 中的结构, 其中 schema 中的 fields 字段为一个数组，表示 payload 字段包含了多个字段，数组的每个元素是对 payload 中相应字段结构的描述信息。
2	field	每一个 fields 中的元素都包含一个 field 字段，该字段表示 payload 中对应字段的名称。在示例中包括 before、after、source 等。
3	type	表示字段的类型，如整型（int）、字符串（string）等.
4	PostgreSQL_server.inventory.customers.Value	表示该字段是 PostgreSQL_server 连接器生成的针对 inventory 数据库的 customers 表的 value 部分信息。
5	io.debezium.connector.postgresql.Source	该名称和特定的 connector 绑定，由该 connector 生成的事件该名称都相同。
6	payload	包含修改事件中具体被修改的数据，包括修改前（before 字段）和修改后（after 字段）的数据，以及一些 connector 的元数据信息（source 字段）。
7	op	表示导致事件生成的修改操作的类型，例子中的 c 表示 修改操作创建了一个新的行。c = createu = updated = deleter = read (仅snapshots)t = truncatem = message
8	source	source 字段是一个描述事件元数据的字段。它包含的一些字段可以用来与其他事件做比较，如比较事件生成的顺序、事件是否属于同一个事务等。该字段包含以下元数据信息： Debezium versionConnector type and nameDatabase and table that contains the new rowStringified JSON array of additional offset information. The first value is always the last committed LSN, the second value is always the current LSN. Either value may be nullSchema nameIf the event was part of a snapshotID of the transaction in which the operation was performedOffset of the operation in the database logTimestamp for when the change was made in the database

更新事件（update events）

下面给出一个 Debezium PostgreSQL connector 针对数据库更新操作生成的消息：

```
{
    "schema": { ... },
    "payload": {
        "before": {
            "id": 1
        },
        "after": {
            "id": 1,
```

```
            "first_name": "Anne Marie",
            "last_name": "Kretchmar",
            "email": "annek@noanswer.org"
        },
        "source": {
            "version": "1.9.3.Final",
            "connector": "postgresql",
            "name": "PostgreSQL_server",
            "ts_ms": 1559033904863,
            "snapshot": false,
            "db": "postgres",
            "schema": "public",
            "table": "customers",
            "txId": 556,
            "lsn": 24023128,
            "xmin": null
        },
        "op": "u",
        "ts_ms": 1465584025523
    }
}
```

其中 schema 字段和创建操作的事件相同，而 payload 部分有所不同，在创建事件中，before 字段为 null，表示没有原始数据，而更新事件中包含了更新前（before）和更新后（after）的数据。

清空表事件（truncate events）

当一个清空表事件发生时，事件消息的 key 为 null，消息示例如下所示：

```
{
    "schema": { ... },
    "payload": {
        "source": {
            "version": "1.9.3.Final",
            "connector": "postgresql",
            "name": "PostgreSQL_server",
            "ts_ms": 1559033904863,
            "snapshot": false,
            "db": "postgres",
            "schema": "public",
            "table": "customers",
            "txId": 556,
            "lsn": 46523128,
            "xmin": null
        },
        "op": "t",
        "ts_ms": 1559033904961
    }
}
```

如果一个 TRUNCATE 语句作用于多个表，那么 connector 会给每一个被作用的表生成一个 truncate event 消息。

消息事件（message events）

该消息类型仅支持 Postgres 14+ 的 pgoutput plugin。事务型消息事件的格式示例如下：

```
{
    "schema": { ... },
    "payload": {
        "source": {
            "version": "1.9.3.Final",
            "connector": "postgresql",
            "name": "PostgreSQL_server",
            "ts_ms": 1559033904863,
            "snapshot": false,
```

```
            "db": "postgres",
            "schema": "",
            "table": "",
            "txId": 556,
            "lsn": 46523128,
            "xmin": null
        },
        "op": "m",
        "ts_ms": 1559033904961,
        "message": {
            "prefix": "foo",
            "content": "Ymfy"
        }
    }
}
```

非事务型消息的格式示例如下：

```
{
    "schema": { ... },
    "payload": {
        "source": {
            "version": "1.9.3.Final",
            "connector": "postgresql",
            "name": "PostgreSQL_server",
            "ts_ms": 1559033904863,
            "snapshot": false,
            "db": "postgres",
            "schema": "",
            "table": "",
            "lsn": 46523128,
            "xmin": null
        },
        "op": "m",
        "ts_ms": 1559033904961
        "message": {
            "prefix": "foo",
            "content": "Ymfy"
        }
    }
}
```

其中事务类型的消息事件包含事务 ID 号字段 “txId”。此外消息事件还包含一个 message 字段，其含义解释如下：

Field name	Description
message	该字段包含了消息的元数据： • prefix（text） • Content (byte array that is encoded based on the binary handling mode setting)

删除事件（delete events）

下面给出一个 Debezium PostgreSQL connector 针对数据库删除操作生成的消息：

```
{
    "schema": { ... },
    "payload": {
        "before": {
            "id": 1
        },
        "after": null,
        "source": {
            "version": "1.9.3.Final",
            "connector": "postgresql",
            "name": "PostgreSQL_server",
```

```
            "ts_ms": 1559033904863,
            "snapshot": false,
            "db": "postgres",
            "schema": "public",
            "table": "customers",
            "txId": 556,
            "lsn": 46523128,
            "xmin": null
        },
        "op": "d",
        "ts_ms": 1465581902461
    }
}
```

其中 schema 字段和创建操作的事件相同，而 payload 部分有所不同，删除事件中包含了修改前（before）的数据，但更新后（after）的数据为 null，表示数据已删除。

更新主键（primary key events）

如果一个操作修改了数据表中某行的主键，那么 connector 会生成一条 **删除事件 来表示原主键对应的数据行删除，同时生成一条新增事件**来表示插入的新主键对应的行。每一条消息的 header 都会和相应的 key 关联。官方描述如下：

- The `DELETE` event record has `__debezium.newkey` as a message header. The value of this header is the new primary key for the updated row.
- The `CREATE` event record has `__debezium.oldkey` as a message header. The value of this header is the previous (old) primary key that the updated row had.

MySQL 订阅消息官方格式说明

最近更新时间：2024-10-09 16:56:15

概述

使用 CKafka 连接器订阅 MySQL 的变更操作时，可选多种消息格式，默认采用 debezium 格式，同时提供了兼容其他消息格式的能力。本文介绍兼容 **官方自定义格式** 的消息格式说明。

“官方格式一”说明

“官方格式一”目前仅支持 DML 消息，DDL 消息格式与 canal 格式一致。

字段名称	字段说明
BINLOG_NAME	binlog 日志文件名称
BINLOG_POS	binlog 日志的 pos 位置
DATABASE	数据库名称
EVENT_SERVER_ID	暂时默认为 null
GLOBAL_ID	如果开启 GTID，则为 GTID 信息
GROUP_ID	暂时默认为 null
NEW_VALUES	• type = U，则为更新后的行信息，格式为 json • type = D，则为 null • type = I，则为新插入的行信息，格式为 json
OLD_VALUES	• type = U，则为更新前的行信息，格式为 json • type = D，则为删除的行信息，格式为 json • type = I，则为 null
TABLE	表名
TIME	日志生成时间
TYPE	日志类型： • U：update • D：delete • I：insert

DDL 格式

create database

```
{
    "data": null,
    "database": "dip_test",
    "es": 1655812326,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "CREATE DATABASE `dip_test` CHARSET utf8mb4 COLLATE utf8mb4_0900_ai_ci",
    "sqlType": null,
    "table": "",
    "ts": 1655812326,
    "type": "QUERY"
}
```

drop database

```
{
    "data": null,
    "database": "dip_test",
    "es": 1655812326,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "DROP DATABASE IF EXISTS `dip_test`",
    "sqlType": null,
    "table": "",
    "ts": 1655812326,
    "type": "QUERY"
}
```

create table

```
{
    "data": null,
    "database": "dip_test",
    "es": 1655812326,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "CREATE TABLE `customers` (
  `id` int NOT NULL AUTO_INCREMENT,
  `first_name` varchar(255) NOT NULL,
  `last_name` varchar(255) NOT NULL,
  `email` varchar(255) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `email` (`email`),
  KEY `ix_id` (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1041 DEFAULT CHARSET=utf8",
    "sqlType": null,
    "table": "customers",
    "ts": 1655812326,
    "type": "CREATE"
}
```

alter table

```
{
    "data": null,
    "database": "test",
    "es": 1655782153,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "ALTER TABLE `user` ADD COLUMN `createtime` datetime NULL DEFAULT CURRENT_TIMESTAMP",
    "sqlType": null,
    "table": "user",
    "ts": 1655782153,
    "type": "ALTER"
}
```

drop table

```
{
    "data": null,
    "database": "dip_test",
    "es": 1655812326,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "DROP TABLE IF EXISTS `dip_test`.`customers`",
    "sqlType": null,
    "table": "customers",
    "ts": 1655812326,
    "type": "ERASE"
}
```


rename table


```
{
    "data": null,
    "database": "testDB",
    "es": 1656300979748,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "rename table test to t_test",
    "sqlType": null,
    "table": "t_test",
    "ts": 1656300979748,
    "type": "RENAME"
}
```


DML 格式

insert


```
{
    "BINLOG_NAME": "mysql-bin.000003",
    "BINLOG_POS": 154,
    "DATABASE": "inventory",
    "EVENT_SERVER_ID": null,
    "GLOBAL_ID": null,
    "GROUP_ID": null,
    "NEW_VALUES": {
        "last_name": "Kretchmar",
        "id": "1004",
        "first_name": "Anne",
        "email": "annek@noanswer.org"
    },
    "OLD_VALUES": null,
    "TABLE": "customers",
    "TIME": "19700101080000",
    "TYPE": "I"
}
```


update


```
{
    "BINLOG_NAME": "mysql-bin.000003",
    "BINLOG_POS": 484,
```

```
    "DATABASE": "inventory",
    "EVENT_SERVER_ID": null,
    "GLOBAL_ID": null,
    "GROUP_ID": null,
    "NEW_VALUES": {
        "last_name": "Kretchmar",
        "id": "1004",
        "first_name": "Anne Marie",
        "email": "annek@noanswer.org"
    },
    "OLD_VALUES": {
        "last_name": "Kretchmar",
        "id": "1004",
        "first_name": "Anne",
        "email": "annek@noanswer.org"
    },
    "TABLE": "customers",
    "TIME": "20160611015029",
    "TYPE": "U"
}
```

delete

```
{
    "BINLOG_NAME": "mysql-bin.000003",
    "BINLOG_POS": 805,
    "DATABASE": "inventory",
    "EVENT_SERVER_ID": null,
    "GLOBAL_ID": null,
    "GROUP_ID": null,
    "NEW_VALUES": null,
    "OLD_VALUES": {
        "last_name": "Kretchmar",
        "id": "1004",
        "first_name": "Anne Marie",
        "email": "annek@noanswer.org"
    },
    "TABLE": "customers",
    "TIME": "20160611020502",
    "TYPE": "D"
}
```

MySQL 订阅消息 canal 格式说明

最近更新时间：2024-10-10 15:31:32

概述

使用连接器订阅 MySQL 的变更操作时，可选多种消息格式，默认采用 debezium 格式，同时提供了兼容开源 canal 等格式的能力。本文介绍兼容 canal 的消息格式说明。

canal 格式说明

兼容字段	canal原生字段	是否支持
id（默认值0）	id（canal 生成的消息 ID）	否
database(数据库名)	database(数据库名)	是
table（表名）	table（表名）	是
pkNames（默认值null）	pkNames（主键字段名）	否
isDdl（变更是否属于DDL）	isDdl（变更是否属于 DDL）	是
type（变更类型）	type（变更类型）	是
es（binlog时间戳）	es（binlog 时间戳）	是
ts（connector 同步时间，仅 DML 支持，DDL 中与 ts 一致）	ts（connector 同步时间）	是
sql（DDL 执行语句）	sql（DDL 执行语句）	是
sqlType（暂不支持，默认值 null）	sqlType（与 mysqlType 对应的数据类型编号）	否
mysqlType（暂不支持，默认值 null）	mysqlType（mysql 中的字段类型）	否
data（变更后的记录-全字段）	data（变更后的记录-全字段）	是
old（变更前的记录--全字段）	old（变更前的记录--仅变更字段）	是

DDL 格式

create database

```
{
    "data": null,
    "database": "dip_test",
    "es": 1655812326,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "CREATE DATABASE `dip_test` CHARSET utf8mb4 COLLATE utf8mb4_0900_ai_ci",
    "sqlType": null,
    "table": "",
    "ts": 1655812326,
    "type": "QUERY"
}
```

drop database

```
{
    "data": null,
    "database": "dip_test",
    "es": 1655812326,
    "id": 0,
    "isDdl": true,
```

```
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "DROP DATABASE IF EXISTS `dip_test`",
    "sqlType": null,
    "table": "",
    "ts": 1655812326,
    "type": "QUERY"
}
```

create table

```
{
    "data": null,
    "database": "dip_test",
    "es": 1655812326,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "CREATE TABLE `customers` (
  `id` int NOT NULL AUTO_INCREMENT,
  `first_name` varchar(255) NOT NULL,
  `last_name` varchar(255) NOT NULL,
  `email` varchar(255) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `email` (`email`),
  KEY `ix_id` (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1041 DEFAULT CHARSET=utf8",
    "sqlType": null,
    "table": "customers",
    "ts": 1655812326,
    "type": "CREATE"
}
```

alter table

```
{
    "data": null,
    "database": "test",
    "es": 1655782153,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "ALTER TABLE `user` ADD COLUMN `createtime` datetime NULL DEFAULT CURRENT_TIMESTAMP",
    "sqlType": null,
    "table": "user",
    "ts": 1655782153,
    "type": "ALTER"
}
```

drop table

```
{
    "data": null,
    "database": "dip_test",
    "es": 1655812326,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
```

```
    "old": null,
    "pkNames": null,
    "sql": "DROP TABLE IF EXISTS `dip_test`.`customers`",
    "sqlType": null,
    "table": "customers",
    "ts": 1655812326,
    "type": "ERASE"
}
```

rename table

```
{
    "data": null,
    "database": "testDB",
    "es": 1656300979748,
    "id": 0,
    "isDdl": true,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "rename table test to t_test",
    "sqlType": null,
    "table": "t_test",
    "ts": 1656300979748,
    "type": "RENAME"
}
```

DML 格式

insert

```
{
    "data": [
        {
            "last_name": "Kretchmar",
            "id": "1004",
            "first_name": "Anne",
            "email": "annek@noanswer.org"
        }
    ],
    "database": "inventory",
    "es": 0,
    "id": 0,
    "isDdl": false,
    "mysqlType": null,
    "old": null,
    "pkNames": null,
    "sql": "",
    "sqlType": null,
    "table": "customers",
    "ts": 1465491411815,
    "type": "INSERT"
}
```

update

```
{
    "data": [
        {
            "last_name": "Kretchmar",
            "id": "1004",
            "first_name": "Anne Marie",
            "email": "annek@noanswer.org"
```

```
        }
    ],
    "database": "inventory",
    "es": 1465581029100,
    "id": 0,
    "isDdl": false,
    "mysqlType": null,
    "old": [
        {
            "last_name": "Kretchmar",
            "id": "1004",
            "first_name": "Anne",
            "email": "annek@noanswer.org"
        }
    ],
    "pkNames": null,
    "sql": "",
    "sqlType": null,
    "table": "customers",
    "ts": 1465581029523,
    "type": "UPDATE"
}
```

delete

```
{
    "data": null,
    "database": "inventory",
    "es": 1465581902300,
    "id": 0,
    "isDdl": false,
    "mysqlType": null,
    "old": [
        {
            "last_name": "Kretchmar",
            "id": "1004",
            "first_name": "Anne Marie",
            "email": "annek@noanswer.org"
        }
    ],
    "pkNames": null,
    "sql": "",
    "sqlType": null,
    "table": "customers",
    "ts": 1465581902461,
    "type": "DELETE"
}
```

连接器订阅 MySQL 的分区分表策略

最近更新时间：2024-10-09 09:38:33

背景

CKafka 连接器支持将订阅的多个 Mysql 数据库表的变更消息推送到 Kafka 的 Topic，有两种推送形式：

1. 支持将多个表的消息推送到同一个 Topic。
2. 支持将多个表的消息推送到不同的 Topic。

当订阅数据发送到多分区的 Topic 时，订阅数据发送的分区策略如下：

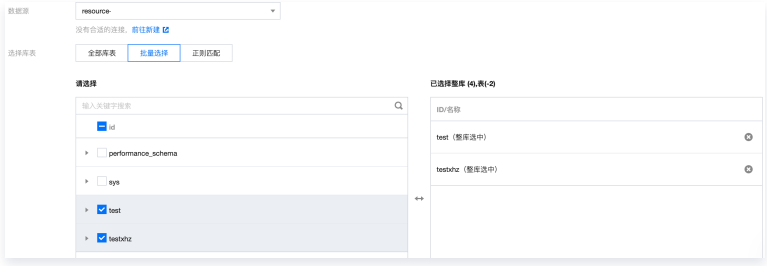
1. 默认情况下，数据将根据表的主键进行 hash，然后发送到多个分区。即同一个主键的订阅数据会发送到同一个分区，保证同一行数据的订阅变更数据是有序的。
2. 当根据主键 hash 不满足需求时，允许手动指定分区 hash 的列。即手动指定根据哪些列来进行 hash 发送至 Topic 的分区。

手动指定分区策略

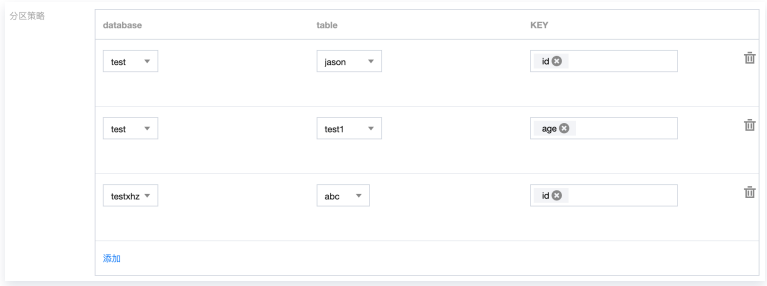
手动指定分区策略需要在新建任务-数据源配置步骤设定每个表的分区策略，默认情况下将按照主键作为 key。

操作步骤如下：

1. 在创建数据链路任务时，在数据源配置界面可选择需要订阅的多库多表。

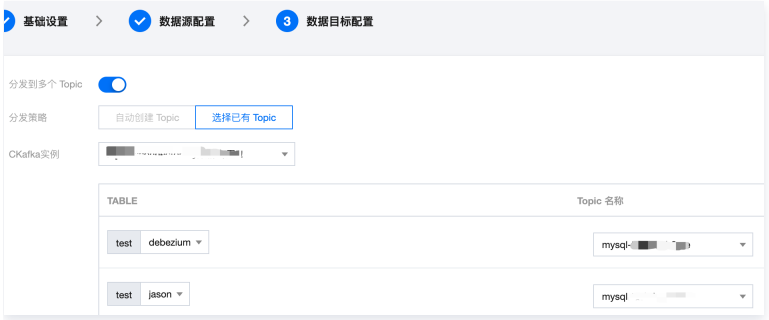


2. 根据选中的表添加分区策略，可设置表字段将字段值相同的数据变更消息投递到同一分区中。



数据订阅到多个 topic

如果希望将订阅的所有表的消息投递到多个 Topic，在新建任务-数据目标配置步骤选择分发到多个 Topic，然后配置表和 Topic 的映射规则。



> **说明**
> 当数据订阅到多个 Topic 时，会将不同表的数据发送到不同的 Topic。在 Topic 维度的分区策略时，还是会根据上述的分区策略进行发送。

连接器订阅 PostgreSQL 时用户权限设置参考

最近更新时间：2024-10-10 15:39:00

背景

在使用 CKafka 连接器订阅 Postgresql 数据库时，需要给 **连接管理** 中配置的 PostgreSQL 用户分配相应的权限。只有拥有相应权限的用户从被允许的主机访问数据库时，才能够进行消息的同步。
本文从用户权限设置以及主机访问权限设置两方面进行说明。

用户权限设置

权限赋予是分 decode 插件的，不同插件需要的权限不一样。

使用 decoderbufs 插件时用户权限设置

通过超级用户登录 PostgreSQL，新建一个角色，给角色赋予至少 `REPLICATION` 和 `LOGIN` 两种权限。
赋予权限：

```
CREATE ROLE userName REPLICATION LOGIN;
```

> **说明**
> 超级用户默认拥有必要的权限，如果是超级用户，大部分无需赋予上面权限。但出于安全考虑不建议使用超级用户。

使用 pgoutput 插件时用户权限设置

> **注意**
> pgoutput 插件需要连接管理配置的用户拥有超级用户权限。

第一步： 验证用户是否拥有超级用户权限

```
// 登录postgresql，执行 \du 命令查看用户权限
postgres=# \du
Role name   | List of roles Attributes
 admin      | Superuser
 postgres   | Superuser, Create role, Create DB, Replication, Bypass RLS
 slave      | Replication

 // 如果配置的用户不包含 Superuser 权限，需要进行授权
postgres=# ALTER USER userName WITH SUPERUSER;
```

第二步： 用户拥有超级用户权限后，按照以下步骤进行授权
connector pgoutput插件通过订阅 postgresql 节点上的 publication 来获取变更事件，您可以在启动 connector 前手动创建 publication，也可以授予配置的用户创建 publication 的权限。
赋予用户以下权限：`Replication`、`CREATE`、`SELECT`

```
CREATE ROLE userName REPLICATION LOGIN;

GRANT CREATE ON DATABASE databaseName TO userName;

GRANT SELECT ON TABLE tableName TO userName;
```

连接管理中配置的用户需要拥有订阅的表的所有者权限，可通过以下方式授予用户表的所有者权限：

1. 创建 replication group。

```
CREATE ROLE <replication_group>;
```

2. 将表的所有者加入到 replication group。

```
GRANT REPLICATION_GROUP TO <original_owner>;
```

3. 将 connector 用户添加到 replication group。

```
GRANT REPLICATION_GROUP TO <replication_user>;
```

4. 将表的所有者权限转移到 replication group。

```
ALTER TABLE <table_name> OWNER TO REPLICATION_GROUP;
```

主机访问权限设置（自建集群需要配置）

您需要配置数据库允许 connector 的主机访问，通过配置 `pg_hba.conf` 文件来设置相应的策略，`pg_hba.conf` 详细介绍可参见 pg_hba.conf 。配置文件格式如下：

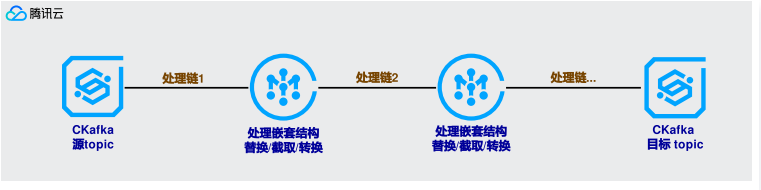
```
host    databaseName    userName     11.163.0.0/16         md5
host    databaseName    userName    11.163.0.0/16        trust
```

数据处理
数据处理规则说明

最近更新时间：2024-10-09 15:39:00

概览

在通过 CKafka 连接器处理数据流入流出任务时，通常需要对数据进行简易的清洗操作，如格式化原始数据、解析特定字段、数据格式转换等等。开发者往往需要自己搭建一套数据清洗的服务（ETL）。
Logstash 是一款免费且开放的服务器端数据处理管道，能够从多个数据源采集数据，转换数据，然后将数据发送到相应的“存储库”中。logstash 拥有丰富的过滤器插件，这使得 logstash 成为了被广泛使用的一款功能强大的数据转换工具。
然而搭建、配置、维护自己的 logstash 服务会增大开发和运维的难度，为此 Ckafka 提供了一套对标 logstash 的数据处理服务，开发者仅需通过控制台交互界面就可以新建自己的数据处理任务。数据处理服务允许用户编辑相应的数据处理规则，支持构建链式处理，同时可以预览数据处理的结果，帮助用户轻松高效的构建一套数据处理服务，满足数据清洗和转换的需求。



功能对标清单

Logstash	连接器数据处理服务	功能
Codec.json	✔	数据解析（JSON）
Filter.grok	✔	数据解析（正则匹配）
Filter.mutate.split	✔	数据解析（字符分割）
Filter.date	✔	日期格式处理
Filter.json	✔	解析内部json结构
Filter.mutate.convert	✔	数据修改（格式转换）
Filter.mutate.gsub	✔	数据修改（字符替换）
Filter.mutate.strip	✔	数据修改（去除首尾空格）
Filter.mutate.join	✔	数据修改（拼接字段）
Filter.mutate.rename	✔	字段修改（更改字段名）
Filter.mutate.update	✔	字段修改（更新字段）
Filter.mutate.replace	✔	字段修改（替换字段）
Filter.mutate.add_field	✔	字段修改（添加字段）
Filter.mutate.remove_field	✔	字段修改（删除字段）
Filter.mutate.copy	✔	字段修改（复制字段值）
Filter.mutate.merge		TODO
Filter.mutate.uppercase		TODO
Filter.mutate.lowercase		TODO

操作方法介绍

数据解析

- logstash 处理方式：

```
// Codec.json
input {
    file {
        path => "/var/log/nginx/access.log_json""
        codec => "json"
    }
}
// Filter.grok
filter {
    grok {
        match => {
            "message" => "\s+(?<request_time>\d+(?:\.\d+)?)\s+"
        }
    }
}
// Filter.mutate.split
filter {
    split {
        field => "message"
        terminator => "#"
    }
}
```

- **连接器处理方式：**
  通过选择相应的数据解析模式 ，并一键点击即可预览：

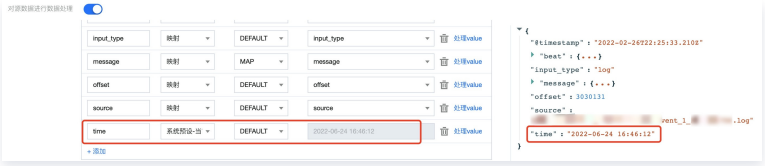
日期格式处理

- **logstash 处理方式：**

```
// Filter.date
filter {
    date {
        match => ["client_time", "yyyy-MM-dd HH:mm:ss"]
    }
}
```

- **连接器处理方式：**

  1.1 可以通过预设系统当前时间给某字段赋值：



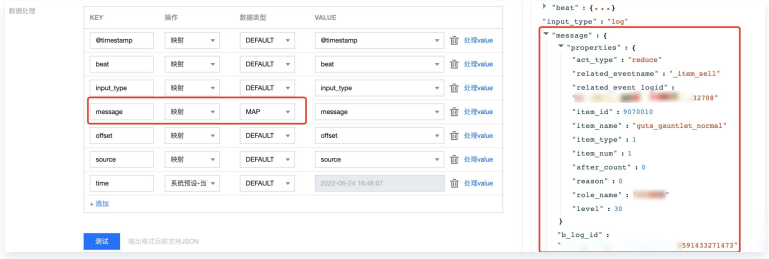
  1.2 通过 处理 value 功能来对数据内容进行处理：

解析内部 JSON 结构

- logstash 处理方式：

```
// Filter.json
filter {
    json {
        source => "message"
        target => "jsoncontent"
    }
}
```

- 连接器处理方式：
  通过对某字段选择 MAP 操作来对其进行解析，从而把特定字段解析为 JSON 格式：



数据修改

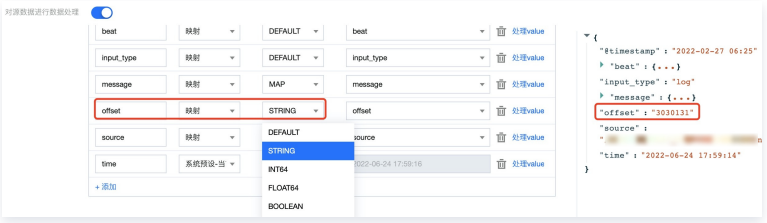
- logstash 处理方式：

```
// Filter.mutate.convert
filter {
    mutate {
        convert => ["request_time", "float"]
    }
}
// Filter.mutate.gsub
filter {
    mutate {
        gsub => ["urlparams", ",", "_"]
    }
}

// Filter.mutate.strip
filter {
    mutate {
```

```
        strip => ["field1", "field2"]
    }
}

// Filter.mutate.join
filter {
    mutate {
        join => { "fieldname" => "," }
    }
}
```

- **连接器处理方式：**
  通过选择相应的处理 value 功能一键定义规则：

1.1 通过选择数据格式一键更改相应字段的数据格式：



1.2 通过 JSONPATH 语法实现 join 的拼接功能，如是用

```
$.concat($.data.Response.SubnetSet[0].VpcId,&quot;#&quot;,$.data.Response.SubnetSet[0].SubnetId,&quot;#&quot;,$.data.Response.SubnetSet[0].CidrBlock))
```

语法拼接 Vpc 和子网的属性，并且通过 `#` 字符加以分割。

1.3 结果如下：

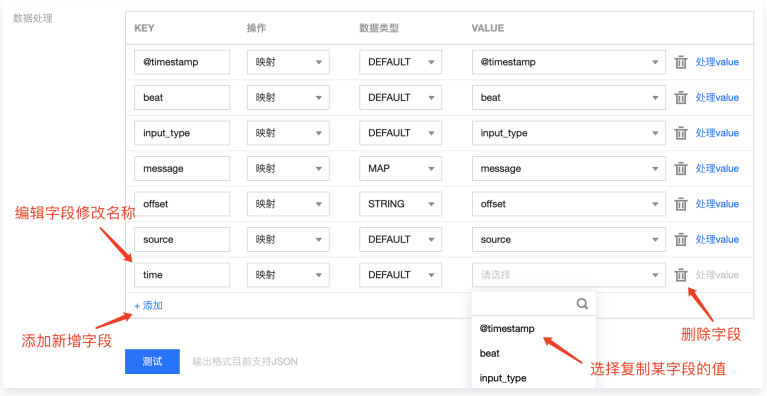
字段修改

- **logstash 处理方式：**

```
// Filter.mutate.rename
filter {
    mutate {
        rename => ["syslog_host", "host"]
    }
}
// Filter.mutate.update
filter {
    mutate {
```

```
            update => { "sample" => "My new message" }
        }
    }
    // Filter.mutate.replace
    filter {
        mutate {
            replace => { "message" => "%{source_host}: My new message" }
        }
    }
    // Filter.mutate.add_field
    filter {
        mutate {
            split => { "hostname" => "." }
            add_field => { "shortHostname" => "%{[hostname][0]}" }
        }
    }
    // Filter.mutate.remove_field
    filter {
        mutate {
            remove_field => ["field_name"]
        }
     }
    // Filter.mutate.copy
    filter {
        mutate {
             copy => { "source_field" => "dest_field" }
        }
    }
```

- 连接器处理方式：



实际案例演示

案例1：多级字段解析

- 输入 message：

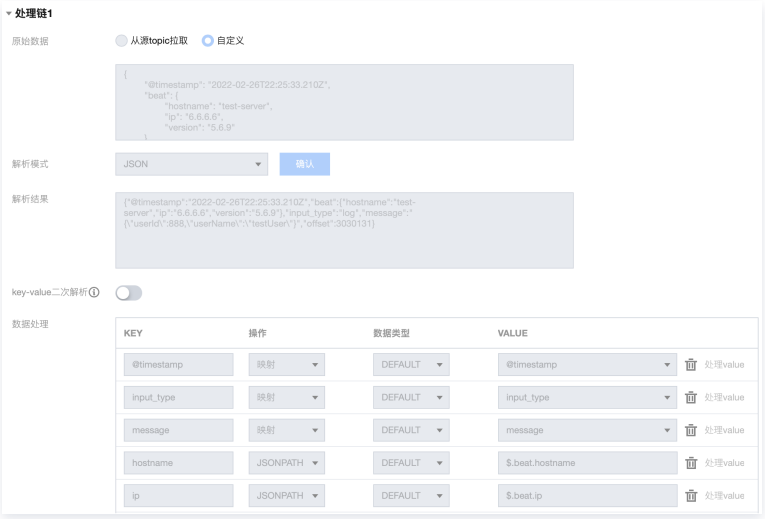
```
{
    "@timestamp": "2022-02-26T22:25:33.210Z",
    "beat": {
        "hostname": "test-server",
        "ip": "6.6.6.6",
        "version": "5.6.9"
    },
    "input_type": "log",
    "message": "{\"userId\":888,\"userName\":\"testUser\"}",
    "offset": 3030131,
}
```

- 目标 message：

```
{
    "@timestamp": "2022-02-26T22:25:33.210Z",
    "input_type": "log",
    "hostname": "test-server",
    "ip": "6.6.6.6",
    "userId": 888,
    "userName": "testUser"
}
```

- **连接器配置方法：**

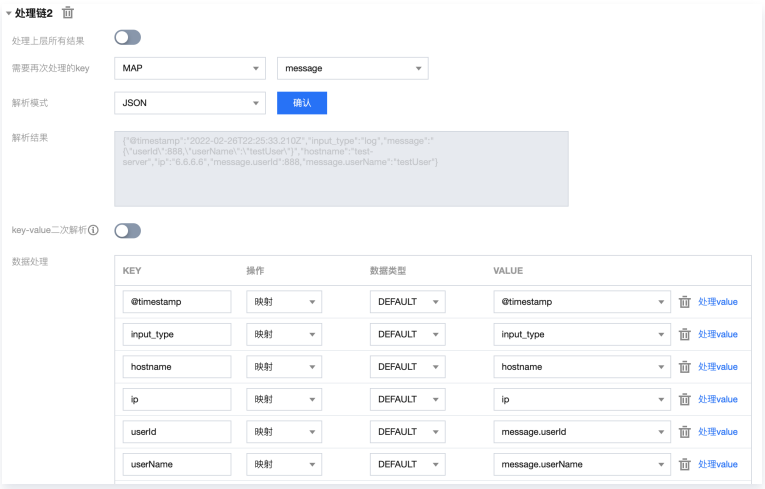
1.1 处理链 1 配置如下：



1.2 处理链 1 结果如下：

```
{
    "@timestamp": "2022-02-26T22:25:33.210Z",
    "input_type": "log",
    "message": "{\"userId\":888,\"userName\":\"testUser\"}",
    "hostname": "test-server",
    "ip": "6.6.6.6"
}
```

1.3 处理链 2 配置如下：



1.4 处理链 2 结果如下：

```
{
    "@timestamp": "2022-02-26T22:25:33.210Z",
    "input_type": "log",
```

```
    "hostname": "test-server",
    "ip": "6.6.6.6",
    "userId": 888,
    "userName": "testUser"
}
```

案例2：非 JSON 数据解析

- 输入 message：

```
region=Shanghai$area=a1$server=6.6.6.6$user=testUser$timeStamp=2022-02-26T22:25:33.210Z
```

- 目标 message：

```
{
    "region": "Shanghai",
    "area": "a1",
    "server": "6.6.6.6",
    "user": "testUser",
    "timeStamp": "2022-02-27 06:25:33",
    "processTimeStamp": "2022-06-27 11:14:49"
}
```

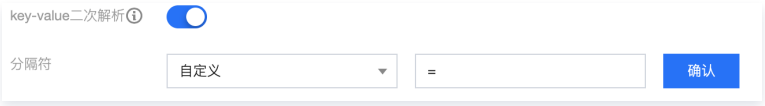
- 连接器配置方法：

1.1 使用分隔符 $ 对原始 message 进行解析

1.2 初步解析结果：

```
{
  "0": "region=Shanghai",
  "1": "area=a1",
  "2": "server=6.6.6.6",
  "3": "user=testUser",
  "4": "timeStamp=2022-02-26T22:25:33.210Z"
}
```

1.3 使用分隔符 = 对结果二次解析：



1.4 二次解析结果：

```
{
  "0": "region=Shanghai",
  "1": "area=a1",
  "2": "server=6.6.6.6",
  "3": "user=testUser",
  "4": "timeStamp=2022-02-26T22:25:33.210Z",
  "0.region": "Shanghai",
  "1.area": "a1",
  "2.server": "6.6.6.6",
  "3.user": "testUser",
```

```
    "4.timeStamp": "2022-02-26T22:25:33.210Z"
}
```

1.5 对字段进行编辑、删减，调整时间戳格式，并新增当前系统时间字段：

- 最终结果：

```
{
    "region": "Shanghai",
    "area": "a1",
    "server": "6.6.6.6",
    "user": "testUser",
    "timeStamp": "2022-02-27 06:25:33",
    "processTimeStamp": "2022-06-27 11:14:49"
}
```

正则提取

最近更新时间：2024-09-29 10:44:55

Ckafka 连接器的数据处理功能提供了根据正则表达式提取消息内容的能力，正则提取采用的是开源的正则提取包 re2 。
Java 的标准正则表达式包 `java.util.regex` 以及其他被广泛使用的正则表达式包如 PCRE、Perlre和 Python（re），都使用回溯实现策略，即当一个 pattern 出现两个替代方案 `a|b` 的时候，引擎将首先尝试匹配子模式 `a` ，如果匹配失败，它将重置输入流并尝试匹配子模式 `b` 。
如果这种匹配模式是深度嵌套的，则此策略需要对输入数据进行指数级的嵌套解析。如果输入的字符串很长，则匹配时间可以趋向无穷大。
相比之下，RE2J 算法通过使用非确定有限自动机在输入数据的单次解析中同时检查所有匹配项，从而实现在线性时间完成正则匹配。
数据处理中的正则提取适用于对长数组类型的消息进行特定字段的提取，下面展示几种常见的提取模式。

案例1：对手机号字段进行提取

输入 message：

```
{"message":
    [
        {"email":123456@qq.com,"phoneNumber":"13890000000","IDNumber":"130423199301067425"},
        {"email":123456789@163.com,"phoneNumber":"15920000000","IDNumber":"610630199109235723"},
        {"email":usr333@gmail.com,"phoneNumber":"18830000000","IDNumber":"420602198802133O1X"}
    ]
}
```

目标 message：

```
{
    "0": "\"phoneNumber\":\"13890000000\"",
    "1": "\"phoneNumber\":\"15920000000\"",
    "2": "\"phoneNumber\":\"18830000000\""
}
```

使用的正则表达式为：

```
"phoneNumber":"(13[0-9]|14[5|7]|15[0|1|2|3|5|6|7|8|9]|18[0|1|2|3|5|6|7|8|9])\d{8}"
```

案例2：对 Email 字段进行提取

输入 message：

```
{"message":
    [
        {"email":123456@qq.com,"phoneNumber":"13890000000","IDNumber":"130423199301067425"},
        {"email":123456789@163.com,"phoneNumber":"15920000000","IDNumber":"610630199109235723"},
        {"email":usr333@gmail.com,"phoneNumber":"18830000000","IDNumber":"420602198802133O1X"}
    ]
}
```

目标 message：

```
{
    "0": "\"email\":\"123456@qq.com\"",
    "1": "\"email\":\"123456789@163.com\"",
```

```
    "2": "\"email\":\"usr333@gmail.com\""
}
```

使用的正则表达式为：

```
"email":"\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*"
```

案例3：对身份证字段进行提取

输入 message：

```
{
    "@timestamp": "2022-02-26T22:25:33.210Z",
    "input_type": "log",
    "operation": "INSERT",
    "operator": "admin",
    "message": "{\"email\":\"123456@qq.com\",\"phoneNumber\":\"13890000000\",\"IDNumber\":\"130423199301067425\"},
{\"email\":\"123456789@163.com\",\"phoneNumber\":\"15920000000\",\"IDNumber\":\"610630199109235723\"},
{\"email\":\"usr333@gmail.com\",\"phoneNumber\":\"18830000000\",\"IDNumber\":\"42060219880213301X\"}"
}
```

目标 message,这里希望保留外部字段，并将message字段中的N个 IDNumber字段单独提取出来：

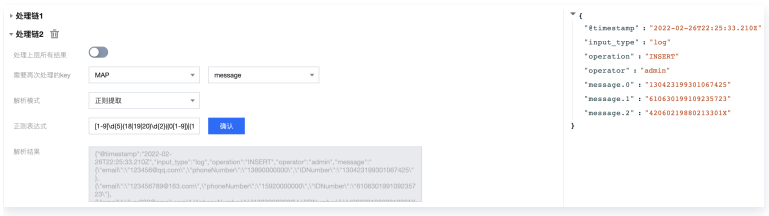
```
{
    "@timestamp": "2022-02-26T22:25:33.210Z",
    "input_type": "log",
    "operation": "INSERT",
    "operator": "admin",
    "message.0": "130423199301067425",
    "message.1": "610630199109235723",
    "message.2": "42060219880213301X"
}
```

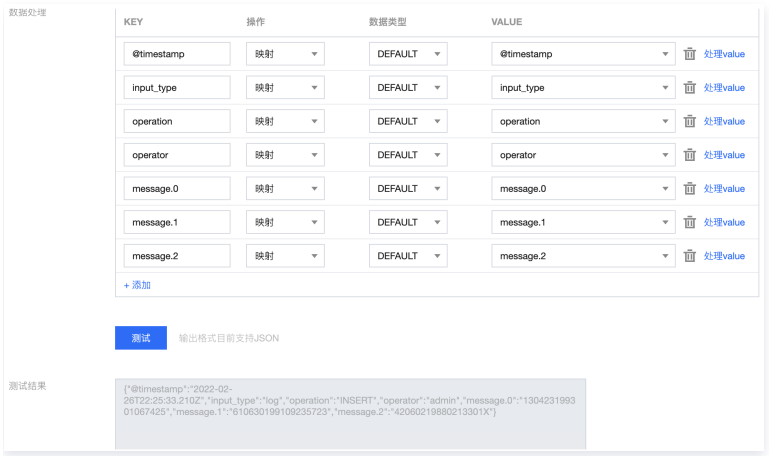
使用的正则表达式为：

```
[1-9]\d{5}(18|19|20)\d{2}((0[1-9])|(1[0-2]))(([0-2][1-9])|10|20|30|31)\d{3}[0-9Xx]
```

这里通过多个处理链进行处理，处理链1的处理结果为：

此时需要对message字段进行二次处理，处理链2的处理结果如下：





处理结果：

```
{
    "@timestamp": "2022-02-26T22:25:33.210Z",
    "input_type": "log",
    "operation": "INSERT",
    "operator": "admin",
    "message.0": "130423199301067425",
    "message.1": "610630199109235723",
    "message.2": "42060219880213301X"
}
```

这里将需要的 IDNumber 字段提取出来后，删除了原来的 message 字段，保留了外部需要的 operation 等字段，以及 message 中的 N 个需要的数据信息。

JsonPath 说明

最近更新时间：2024-10-11 15:39:01

操作场景

JSON 目前是互联网信息传递中最通用的格式协议之一。目前数据处理也主要围绕 JSON 数据格式进行解析处理。
JSONPath 是针对 JSON 格式推出的消息查询语法规范。在数据处理中，不仅能够使用简单的 JSONPath 语法，快速获取复杂嵌套 JSON 结构体的某一成员的值；还能使用 JayWay 库的扩展函数，聚合或操作某一类型的成员字段。

基础功能

基础语法

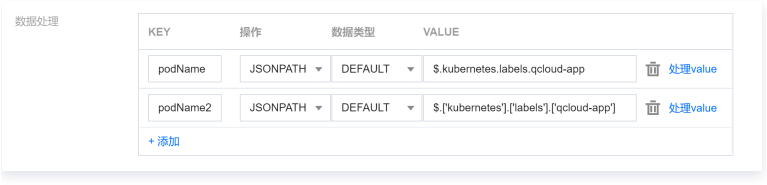
- `$` 根结点操作符，表示当前 JSON 结构体的根结点。
- `.<childName>` 点操作符或 `['<childName>']` 方括号操作符，表示选取当前作用对象名为 `childName` 的子成员。
- `..` 递归操作符，表示 **递归** 获取当前作用对象的所有子成员。
- `[<index>]` 片选操作符，表示获取当前可迭代对象的第 `index` 个子成员。

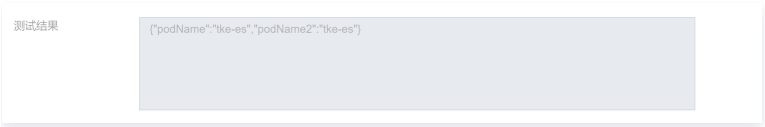
获取嵌套 JSON 特定成员变量

下图为 TKE 采集的容器标准输出日志结构：


```
{
  "@timestamp": 1648803500.63659,
  "@filepath": "/var/log/tke-log-agent/test7/xxxxxxxx-adfe-4617-8cf3-9997aea90ded/c_tke-es-xxxxxxxx57-
n29jr_default_nginx-xxxxxxxx49626ef42d5615a636aae74d6380996043cf6f6560d8131f21a4d8ba/jgw_INFO_2022-02-10_15_4.log",
  "log": "15:00:00.000[4349811564226374227] [http-nio-8081-exec-64] INFO
com.qcloud.jgw.gateway.server.topic.TopicService",
  "kubernetes": {
    "pod_name": "tke-es-xxxxxxxxxx-n29jr",
    "namespace_name": "default",
    "pod_id": "xxxxxxxx-adfe-4617-8cf3-9997aea90ded",
    "labels": {
      "k8s-app": "tke-es",
      "pod-template-hash": "xxxx95d557",
      "qcloud-app": "tke-es"
    },
    "annotations": {
      "qcloud-redeploy-timestamp": "1648016531476",
      "tke.cloud.tencent.com/networks-status": "[{\n    \"name\": \"tke-bridge\",\n    \"interface\": \"eth0\",\n
\"ips\": [\n        \"172.16.x.xx\"\n    ],\n    \"mac\": \"xx:xx:xx:4a:c2:ba\",\n    \"default\": true,\n
\"dns\": {}\n}]"
    },
    "host": "10.0.xx.xx",
    "container_name": "nginx",
    "docker_id": "xxxxxxxx49626ef42d5615a636aae74d6380996043cf6f6560d8131f21a4d8ba",
    "container_hash": "nginx@sha256:xxxxxxxx7b29b585ed1aee166a17fad63d344bc973bc63849d74c6452d549b3e",
    "container_image": "nginx"
  }
}
```


当用户需要获取当前 pod 名称，即获取 `qcloud-app` 成员字段时，可以在数据处理中使用 `$.kubernetes.labels.qcloud-app`，或 `$.['kubernetes'].['labels'].['qcloud-app']` JSONPath 语法获取。
运行结果如下所示，可以看出在测试结果中已经成功读取到 JSONPath 对应的日志：





说明

在使用 JSONPath 处理参数时，当 JSON 变量名称本身带有英文句号 `.` 等特殊符号时，就只能使用方括号操作符进行包装。

例如 `{"key1.key2":"value1"}` 的 JSON 结构体，要想取得对应成员字段，则需要使用 `$.['key1.key2']` 进行获取。

进阶功能

进阶语法

- `*` 通配操作符，表示获取当前作用对象的 **所有** 子成员。
- `*~` 内置函数，表示获取当前可迭代对象所有子对象的名称。
- `min()` 内置函数，表示获取当前可迭代对象子对象的最小值。
- `max()` 内置函数，表示获取当前可迭代对象子对象的最大值。
- `sum()` 内置函数，表示获取当前可迭代对象子对象之和。
- `concat()` 内置函数，表示连接多个对象并生成字符串。

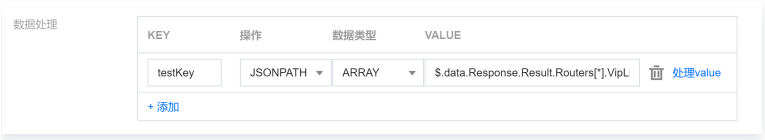
聚合特定字段数据

当 JSON 结构体中存在对象列表时，通常列表为变长形式，以下图中的请求返回日志为例：


```
{
  "data": {
    "Response": {
      "Result": {
        "Routers": [
          {
            "AccessType": 0,
            "RouteId": 81111,
            "VpcId": "vpc-xxxxxxxx",
            "VipType": 3,
            "VipList": [
              {
                "Vip": "10.0.0.189",
                "Vport": "9xxx"
              }
            ]
          },
          {
            "AccessType": 0,
            "RouteId": 81112,
            "VpcId": "vpc-r5sbavzp",
            "VipType": 3,
            "VipList": [
              {
                "Vip": "10.0.0.248",
                "Vport": "9xxx"
              }
            ]
          },
          {
            "AccessType": 0,
            "RouteId": 81113,
            "VpcId": "vpc-xxxxxxxx",
            "VipType": 3,
            "VipList": [
              {
                "Vip": "10.0.0.210",
                "Vport": "9xxx"
              }
```

```
            ]
          }
        ]
      },
      "RequestId": "20e74750-ca40-403d-9ea9-d3f63b5415d2"
    }
  },
  "code": 0
}
```


当需要聚合变长列表的成员属性时，此时无法使用处理链形式聚合处理。此时可以使用 JSONPath 中的 `*` 语法匹配列表中所有元素。
例如当需要得到所有的 VipList 中的 Vip 时，可以使用 `$.data.Response.Result.Routers[*].VipList[0].Vip` 的 JSONPath 语法获取。
运行结果如下所示，可以看出在测试结果中已经成功获取到了所有结构体中的 Vip：



合并修改结构体成员

在某些场景下，需要在数据处理中对 JSON 结构体的多个对象进行合并整合处理，以便投递到下游进行下一步操作。考虑如下格式：


```
{
  "data": {
    "Response": {
      "SubnetSet": [
        {
          "VpcId": "vpc-xxxxxxxx",
          "SubnetId": "subnet-xxxxxxxx",
          "SubnetName": "ckafka_cloud_subnet-1",
          "CidrBlock": "10.0.0.0/19",
          "Ipv6CidrBlock": "",
          "IsDefault": false,
          "IsRemoteVpcSnat": false,
          "EnableBroadcast": false,
          "Zone": "ap-changsha-ec-1",
          "RouteTableId": "rtb-xxxxxxxx",
          "NetworkAclId": "",
          "TotalIpAddressCount": 8189,
          "AvailableIpAddressCount": 8033,
          "CreatedTime": "2021-01-25 17:31:00",
          "TagSet": [],
          "CdcId": "",
          "IsCdcSubnet": 0,
          "LocalZone": false,
          "IsShare": false
        }
      ],
      "TotalCount": 1,
      "RequestId": "705c4955-0cd9-48b2-9132-79eadae2e3e6"
    }
  },
  "code": 0
}
```

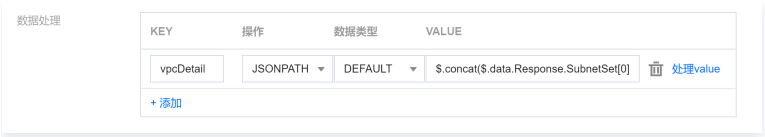

当下游不具有计算功能，需要在数据处理中聚合 Vpc 以及子网属性时，可以使用 JSONPath 中的 `concat()` 函数进行多个字段的聚合，并且在此基础上对字符串进行修改。

例如可以使用

```
$.concat($.data.Response.SubnetSet[0].VpcId,"#",$.data.Response.SubnetSet[0].SubnetId,"#",$.data.Response.SubnetSet[0].CidrBloc
```

语法拼接 VPC 和子网的属性，并且通过 `#` 字符加以分割。

运行结果如下所示，可以看出在测试结果中已经成功获取整合了 VPC 相关的资源信息：



自建集群接入说明（CLB 方式）

最近更新时间：2024-10-11 16:01:21

操作背景

通过 CKafka 连接器连接 CVM 自建的服务时，根据腾讯云网络团队制定的标准跨 VPC 资源访问方案，需要先将自建服务挂载到 CLB（负载均衡）上，才能实现跨 VPC 的资源访问。
即当在 CVM 上自建 MySQL、Mongo 服务作为数据源接入 CKafka 连接器时，需要通过挂载 CLB（负载均衡）的方式接入，本文描述相关的操作流程。

操作流程

步骤1：购买 CLB（负载均衡）实例（可选）

如果客户账号下已有内网 CLB 实例，为了节省成本，可以直接复用该 CLB 实例。也可以新建单独的 CLB 实例提供服务。下面描述新建内网负载均衡(CLB)的操作过程。

注意
购买 CLB 实例时，需要与 CVM 集群在同一地域，同一 VPC，这样 CLB 才能够正常挂载 CVM 实例。同时选择购买内网的 CLB 实例。

步骤2：配置 CLB 实例

1. 创建 TCP 监听器：进入 **CLB 控制台** > **实例管理** 页面。找到需要配置的实例，在该实例详情页面单击 **监听器管理**，单击新建 **TCP 监听器**。

此处配置的端口为访问 CLB 的端口。健康检查和会话保持可根据实际需求进行配置。

2. 绑定自建服务：新建监听器成功后，单击相应的监听器，然后单击右侧的绑定，对 CVM 实例进行绑定。

选择需要绑定的 CVM 实例，并填写服务的端口号：

> **注意：**
> 自建 MySQL 集群建议只挂载 **1台** CVM 实例（1台主节点或1台从节点），因为 MySQL 主从库的 binlog 同步存在延迟，connector 的请求被转发到不同 MySQL 服务时可能会发生读取 binlog 错误。因此自建集群建议只挂载单个 MySQL 服务。

3. 查看服务健康状态：创建完成后可看到对应的服务及健康状态。

步骤3：创建连接

进入 CKafka 控制台，单击**连接器** > **连接列表**，单击**新建连接**。

> **说明**
> 其中 CLB 实例选择挂载了 CVM 服务的实例，端口为相应的 CLB 监听端口，用户名和密码为相应的服务对应的用户名和密码。

连接器访问 CLS、COS 等服务的授权说明

最近更新时间：2024-10-11 15:39:01

操作背景

用户在使用 CKafka 连接器访问 CLS、COS 等服务时，需要授权连接器访问用户账号下 CLS、COS 等服务的权限。如果使用 CKafka 的子账号具备访问管理的策略权限（QcloudCamRoleFullAccess），在创建 CKafka 任务时勾选 **角色授权**，连接器将自动为您完成授权。否则，需要拥有管理员权限（AdministratorAccess）的用户进行相应的授权后，再使用子账号创建 连接器 任务。

需授权服务列表

需授权的服务	关联的角色	需要的策略权限
日志服务（CLS）	Datahub_QcsRole	QcloudCLSFullAccess
对象存储（COS）	Datahub_QcsRole	QcloudCOSFullAccess

授权步骤

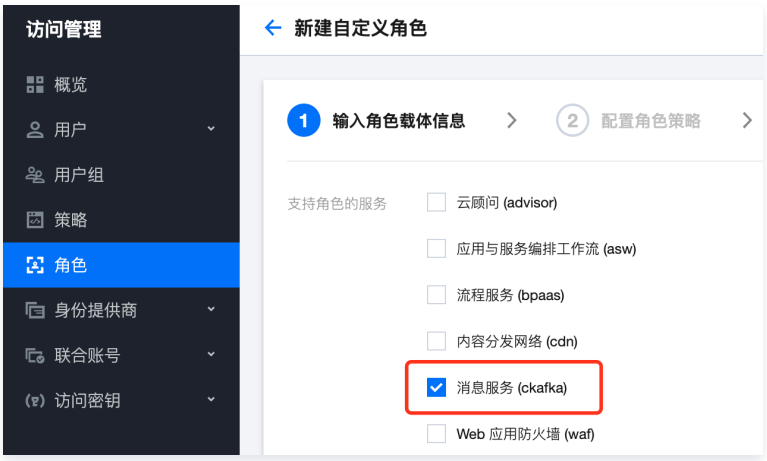
如果您创建连接器任务的子账号不具备访问管理的策略权限（QcloudCamRoleFullAccess），可能会遇到缺少 CreateRole 或 AttachRolePolicy 权限的提示。如果您的账号下还没有 Datahub_QcsRole 角色，参见 **创建角色** 进行授权。如果账号已经拥有 Datahub_QcsRole 角色，可参见 **授权角色** 进行授权。

创建角色

1. 如果提示缺少 CreateRole 策略权限，需要有管理员权限（AdministratorAccess）的用户进入 访问管理 控制台，角色页面，单击 **新建角色**。

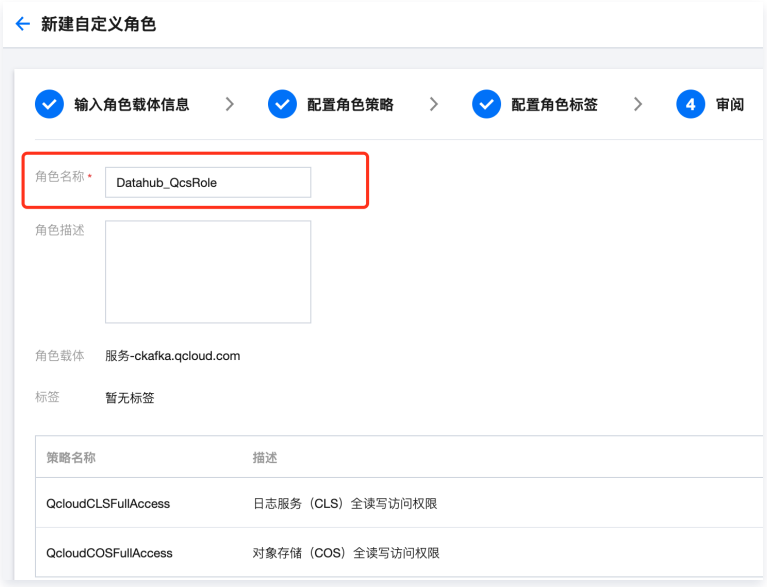
2. 在**选择角色载体**页面选择腾讯云产品服务：

3. 进入**输入角色载体信息**步骤，选择**消息服务（ckafka）**：



4. 在**配置角色策略**步骤，选择需要连接器任务访问的服务对应的策略，此处选择了 CLS、COS 对应的策略：

5. 在**配置角色标签**步骤可以给角色配置相应的标签，此处忽略。
6. 在**审阅**步骤，将角色名称命名为 **Datahub_QcsRole**：



创建角色成功后，子账号即可进行相应的连接器任务创建。

授权角色

1. 如果提示缺少 AttachRolePolicy 策略权限，需要有管理员权限（AdministratorAccess）的用户进入 访问管理 控制台，角色页面，找到服务对应的角色，此处以 Datahub_QcsRole 角色为例。

2. 单击角色名称，进入角色管理详情页面，在 **权限** 一栏单击 **关联策略**：

3. 找到您需要授权的服务相关的策略，此处以 CLS 服务为例，单击**确定**完成授权：

4. 当该角色拥有访问相应服务的权限后，子账号就可以成功创建相应的连接器任务。

什么是信令表

最近更新时间：2024-10-11 15:39:01

背景

debezium connector 最初仅在创建连接任务时会同步表的存量数据，后续新增的表无法触发存量数据的同步。为了支持新增表的存量数据同步功能，debezium 采用了“信号”的模式通知 connector 进行触发新增表的存量数据同步。

原理

需要在订阅的库中新建一张信令表，并在需要触发新表的存量数据同步时在信令表中插入相关的信息，同时 connector 需要订阅这张信令表，当 connector 收到信令表的消息时，就会触发新增表的存量数据同步功能。

注意事项

1. 由于新增了对信令表（dip_signal_taskId）消息的订阅，因此目标 Topic 中会包含信令表的消息，需要业务端进行相应的过滤处理。
2. 请确保连接管理中配置的用户拥有该数据库的创建、修改、删除表的权限（仅用于对信令表的操作）。
3. 如果使用了信令表同步新增表的存量数据，可能会存在部分增量数据出现重复的情况。如果需要避免重复的数据，可以通过暂停对需要新增的表的增删改查操作之后，再编辑连接器任务的数据源，添加该表，然后再恢复对新增表的增删改查即可，也可以通过下游做一些幂等处理。
4. 如果新建任务时没有选择同步存量数据，然后修改数据源，新增表的时候选择了同步存量数据，且在此期间没有产生任何变更消息，那么 debezium 默认会同时同步旧表和新表的存量数据（与信令表无关）。如果在修改数据源之前旧表有增量变更消息的话，那么修改数据源增加表且选择存量数据时，则只会同步新增表的存量数据（由信令表触发）。

Kafka to CKafka 实例同步任务说明
实例级数据同步可用性汇总

最近更新时间：2025-01-21 14:35:02

Kafka to CKafka 同步数据规则说明

CKafka 连接器 K2K 插件支持在控制台上进行实例之间的**元数据**，**消息数据**和**消费位点**同步，具体的规则说明如下：

- **同步元数据规则：**分为初始化同步和定时同步。
  - **初始化同步：**如果任务启动，会查看 Topic 是否新建，如果没有，则尽可能保持和原来的配置一在下游初始化新建 Topic，然后启动任务；如果下游本身就存在对应的 Topic，则不会触发初始化同步。
  - **定时同步：**任务启动之后，会按一定时间周期性将上游的部分元数据配置同步到下游，同步周期：3分钟。
- **同步消息数据规则：**将上游 CKafka 中存储的消息数据同步到下游 CKafka 对应的 Topic 中，如果开启了同步到相同分区，则消息会固定同步到下游对应的相同分区中（具体参数配置见下文详细说明）。
- **同步消费位点规则：**将上游 CKafka 中存储的消息数据同步到下游 CKafka 对应 Topic 中，同时同步相关的消费组和消费组对该 Topic 所提交的 Offset 信息，注意该 Offset 是映射后的对应关系（具体参数配置见下文详细说明）。

操作步骤

1. 登录 CKafka 控制台。
2. 在左侧导航栏单击**连接器 > 任务列表**，选择好地域后，单击**新建任务**。
3. 填写**任务名称**，任务类型选择**数据流出**，数据目标类型选择 **消息队列（Kafka）**，单击**下一步**。

4. 进行数据源配置。
   - 数据源类型：选择**整个Kafka实例**。
   - 数据源：选择创建好的 Kafka 连接。
   - 同步数据类型：选择需要同步的数据类型。
     - 只同步元数据
     - 同步元数据和消息数据
     - 同步元数据、消息数据和消费位点

只同步元数据

同步数据类型选择“只同步元数据”。

支持同步的元数据

初始化同步支持的元数据	定时同步支持的元数据
• 分区数 • 副本数 • retention.ms • cleanup.policy • min.insync.replicas • unclean.leader.election.enable • segment.ms • retention.bytes • max.message.bytes • 消费组	• 分区数（存在特殊情况） • retention.ms（存在特殊情况） • cleanup.policy • min.insync.replicas • unclean.leader.election.enable • segment.ms • retention.bytes（存在特殊情况） • max.message.bytes • 消费组

说明：

- 定时同步暂不支持同步副本数。
- 定时同步存在如下两种特殊情况：
  - 分区数只能单向增加不能减少，如果下游实例分区已经大于上游，则不会同步。
  - 出于稳定性考虑，retention.ms 和 retention.bytes 这两项元数据，目标 Topic 即下游对应元数据值为-1时（注意：-1为 Kafka 内部定义，代表无限存储），才会同步，其他情况下这两项元数据均不会定时同步。

元数据未同步情况下系统设定的默认值

> **说明：**
> 该配置上游不存在情况或者非法（非数字、null、空）会使用默认值替代。

元数据	默认值
retention.ms	604800000（7天）
cleanup.policy	delete
min.insync.replicas	1
unclean.leader.election.enable	false
segment.ms	604800000
retention.bytes	默认值取决于 Kafka 配置
max.message.bytes	1048588

各 Kafka 版本交叉同步元数据的可支持情况

Kafka 版本	0.10.2.1	1.1.1	2.4.1	2.8.1	3.2.3
0.10.2.1	支持	支持	支持	支持	支持
1.1.1	支持	支持	支持	支持	支持
2.4.1	支持	支持	支持	支持	支持
2.8.1	支持	支持	支持	支持	支持
3.2.3	支持	支持	支持	支持	支持

同步元数据和消息数据

同步数据类型选择“同步元数据和消息数据”。

元数据

支持同步的元数据

初始化同步支持的元数据	定时同步支持的元数据
• 分区数 • 副本数	• 分区数（存在特殊情况） • retention.ms（存在特殊情况）

- retention.ms
- cleanup.policy
- min.insync.replicas
- unclean.leader.election.enable
- segment.ms
- retention.bytes
- max.message.bytes
- 消费组

- cleanup.policy
- min.insync.replicas
- unclean.leader.election.enable
- segment.ms
- retention.bytes（存在特殊情况）
- max.message.bytes
- 消费组

说明：
- 定时同步暂不支持同步副本数。
- 定时同步存在如下两种特殊情况：
  - 分区数只能单向增加不能减少，如果下游实例分区已经大于上游，则不会同步。
  - 出于稳定性考虑，retention.ms 和 retention.bytes 这两项元数据，目标 Topic 即下游对应元数据值为-1（注：-1为Kafka内部定义，代表无限存储）时，才会同步，其他情况下这两项元数据均不会定时同步。

元数据未同步情况下系统设定的默认值

说明：
该配置上游不存在情况或者非法会使用默认值替代。

元数据	默认值
retention.ms	604800000（7天）
cleanup.policy	delete
min.insync.replicas	1
unclean.leader.election.enable	false
segment.ms	604800000
retention.bytes	默认值取决 Kafka 配置
max.message.bytes	1048588

各 Kafka 版本交叉同步元数据的可支持情况

Kafka 版本	0.10.2.1	1.1.1	2.4.1	2.8.1	3.2.3
0.10.2.1	支持	支持	支持	支持	支持
1.1.1	支持	支持	支持	支持	支持
2.4.1	支持	支持	支持	支持	支持
2.8.1	支持	支持	支持	支持	支持
3.2.3	支持	支持	支持	支持	支持

消息数据

各 Kafka 版本交叉同步消息数据的可支持情况：

Kafka 版本	0.10.2.1	1.1.1	2.4.1	2.8.1	3.2.3
0.10.2.1	支持	支持	支持	支持	支持
1.1.1	支持	支持	支持	支持	支持
2.4.1	支持	支持	支持	支持	支持
2.8.1	支持	支持	支持	支持	支持
3.2.3	支持	支持	支持	支持	支持

说明：
- 支持消息数据同步到相同分区。
- 消息数据在同步时，下游如果没有同名 Topic，则创建完任务之后大概会有10分钟左右的延迟才会同步消息数据。

同步元数据、消息数据和消费位点

同步数据类型选择：同步元数据、消息数据和消费位点。

元数据

支持同步的元数据

初始化同步支持的元数据	定时同步支持的元数据
• 分区数 • 副本数 • 消费组 • retention.ms • cleanup.policy • min.insync.replicas • unclean.leader.election.enable • segment.ms • retention.bytes • max.message.bytes	• 分区数 • 消费组

关于定时同步的使用限制：
- 定时同步暂不支持同步副本数。
- 定时同步存在特殊情况：分区数只能单向增加不能减少，如果下游实例分区已经大于上游，则不会同步。
- 新增的 topic 会同步一次，后续该 topic 的配置变更不会定时同步：
  - **原因：**对于 topic 级别的配置，例如：retention.ms，cleanup.policy，min.insync.replicas，unclean.leader.election.enable，segment.ms，retention.bytes，max.message.bytes配置，由于存在稳定性风险，默认情况下，只在新建连接任务的初始状态，元数据会完整同步一次，后续 topic 变更则不同步（定时同步 topic 变更的风险：客户在不知道存量任务的情况下，修改上游 topic 配置（如缩短消息时间），此时下游如果同步该变更，会导致下游在未及时消费情况下，出现大量丢失数据的情况，因此，在定时同步配置中，不对topic变更进行同步）
  - **同步重配的解决方案：**如果客户需要变更存量任务中的 topic 并且同步配置，由于上述原因，建议客户手动同步修改上下游的 topic 配置，避免出现丢数据或稳定性问题。

各 Kafka 版本交叉同步元数据的可支持情况

Kafka 版本	0.10.2.1	1.1.1	2.4.1	2.8.1	3.2.3
0.10.2.1	支持	支持	支持	支持	支持
1.1.1	支持	支持	支持	支持	支持
2.4.1	支持	支持	支持	支持	支持
2.8.1	支持	支持	支持	支持	支持
3.2.3	支持	支持	支持	支持	支持

消息数据

各 Kafka 版本交叉同步消息数据的可支持情况

Kafka 版本	0.10.2.1	1.1.1	2.4.1	2.8.1	3.2.3
0.10.2.1	支持	支持	支持	支持	支持
1.1.1	支持	支持	支持	支持	支持
2.4.1	支持	支持	支持	支持	支持
2.8.1	支持	支持	支持	支持	支持
3.2.3	支持	支持	支持	支持	支持

说明：

- 支持消息数据同步到相同分区。
- 本功能会同步上游 Topic 的消费位点，您可以选择从最开始或者最新位置开始消费。

消费位点

版本	0.10.2.1	1.1.1	2.4.1	2.8.1	3.2.3
0.10.2.1	版本不支持	版本不支持	版本不支持	版本不支持	版本不支持
1.1.1	版本不支持	支持	支持	支持	支持
2.4.1	版本不支持	支持	支持	支持	支持
2.8.1	版本不支持	支持	支持	支持	支持
3.2.3	版本不支持	支持	支持	支持	支持

说明：

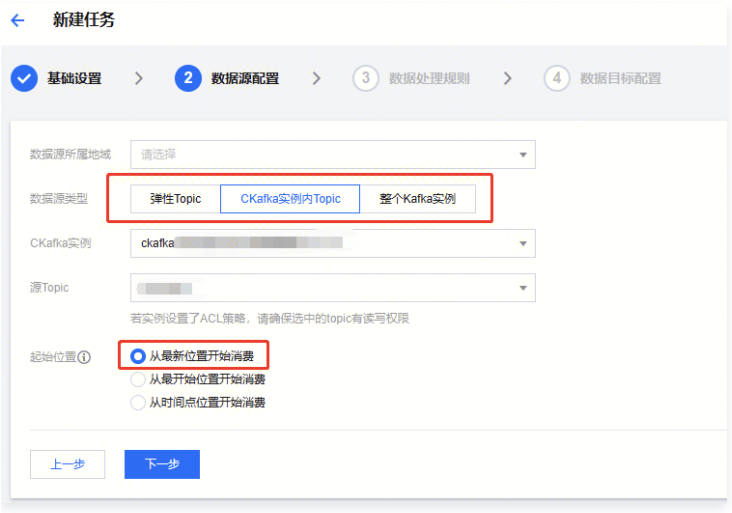
0.10.2.1 版本不支持同步消费位点，因此 K2K 同步 CKafka 实例时，上下游中若包含0.10.2.1版本，则不支持创建同步消费位点的任务。

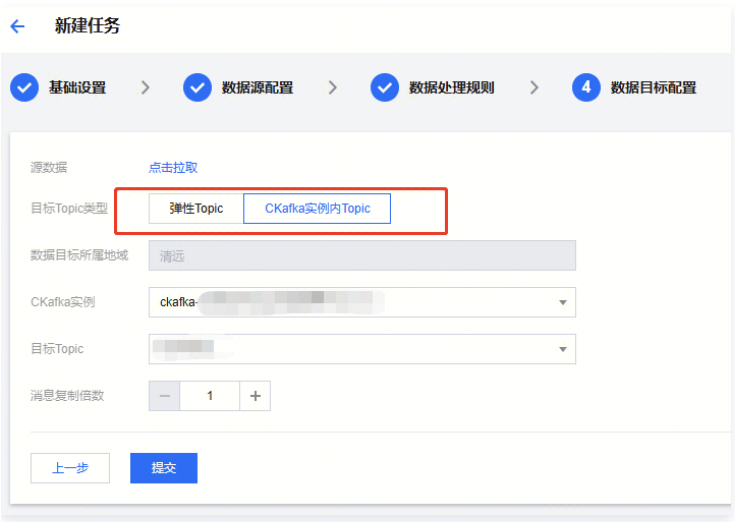
同步实例与实例内的 Topic 可用性汇总

最近更新时间：2025-01-21 14:35:02

同步实例与实例内的 Topic 可用性汇总

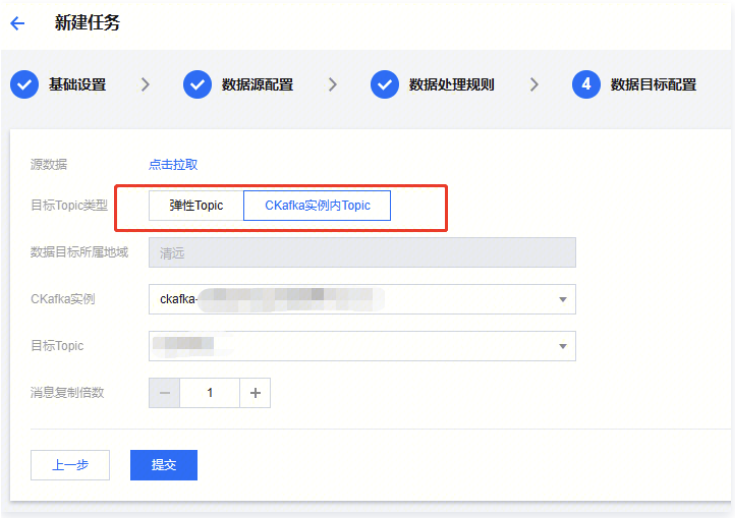
从最新位置开始消费





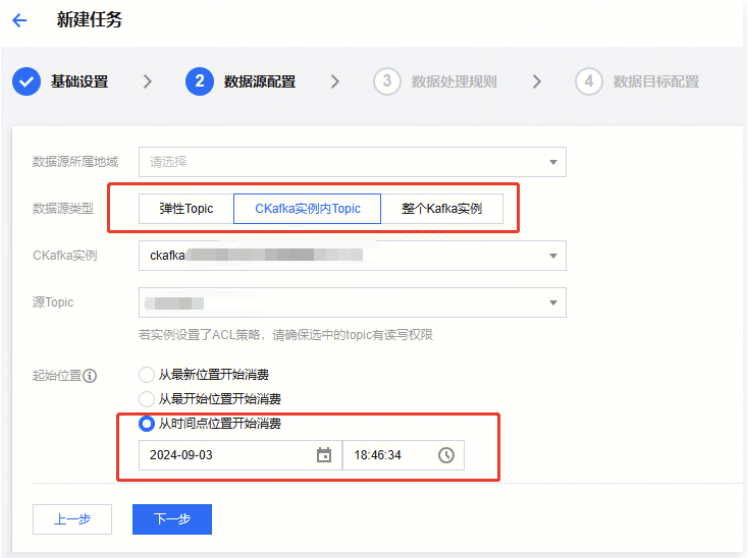
版本	0.10.2.1	1.1.1	2.4.1	2.8.1	3.2.3
0.10.2.1	支持	支持	支持	支持	支持
1.1.1	支持	支持	支持	支持	支持
2.4.1	支持	支持	支持	支持	支持
2.8.1	支持	支持	支持	支持	支持
3.2.3	支持	支持	支持	支持	支持

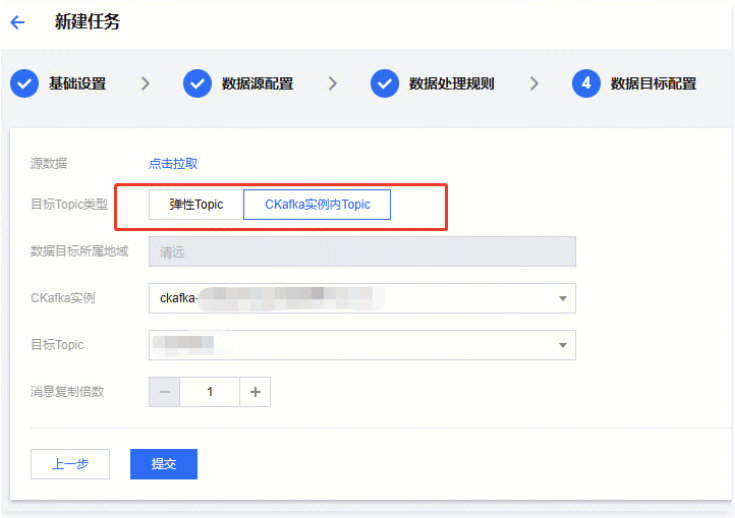
从最开始位置开始消费



版本	0.10.2.1	1.1.1	2.4.1	2.8.1	3.2.3
0.10.2.1	支持	支持	支持	支持	支持
1.1.1	支持	支持	支持	支持	支持
2.4.1	支持	支持	支持	支持	支持
2.8.1	支持	支持	支持	支持	支持
3.2.3	支持	支持	支持	支持	支持

从时间点位开始消费





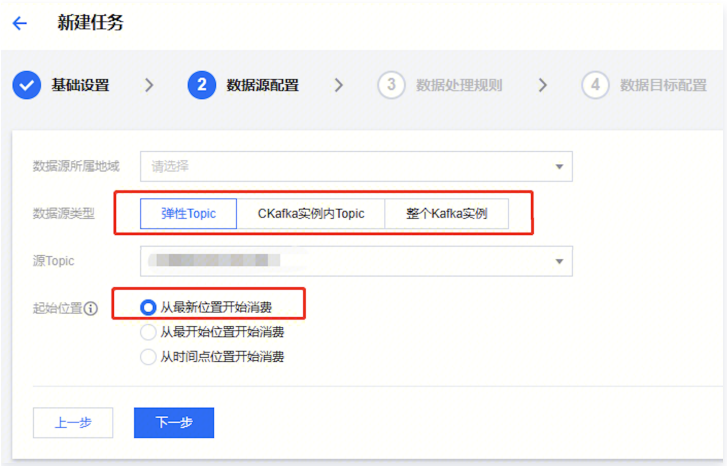
版本	0.10.2.1	1.1.1	2.4.1	2.8.1	3.2.3
0.10.2.1	支持	支持	支持	支持	支持
1.1.1	支持	支持	支持	支持	支持
2.4.1	支持	支持	支持	支持	支持
2.8.1	支持	支持	支持	支持	支持
3.2.3	支持	支持	支持	支持	支持

同步弹性 Topic 与实例内的 Topic 可用性汇总

最近更新时间：2025-01-21 14:35:02

同步弹性 Topic 与实例内的 Topic 可用性汇总

从最新位置开始消费

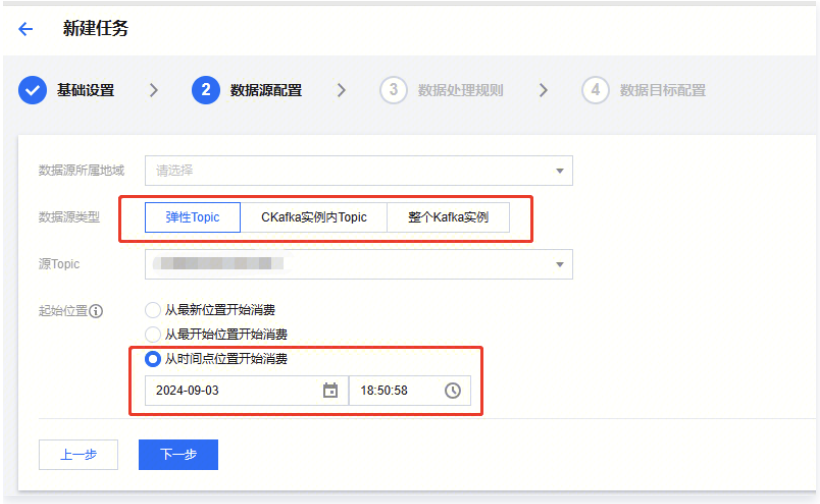


版本	0.10.2.1	1.1.1	2.4.1	2.8.1	3.2.3
0.10.2.1	支持	支持	支持	支持	支持
1.1.1	支持	/	/	/	/
2.4.1	支持	/	/	/	/
2.8.1	支持	/	/	/	/
3.2.3	支持	/	/	/	/

从最开始位置开始消费

版本	0.10.2.1	1.1.1	2.4.1	2.8.1	3.2.3
0.10.2.1	支持	支持	支持	支持	支持
1.1.1	支持	/	/	/	/
2.4.1	支持	/	/	/	/
2.8.1	支持	/	/	/	/
3.2.3	支持	/	/	/	/

从时间点位开始消费



版本	0.10.2.1	1.1.1	2.4.1	2.8.1	3.2.3
0.10.2.1	支持	支持	支持	支持	支持
1.1.1	支持	/	/	/	/
2.4.1	支持	/	/	/	/
2.8.1	支持	/	/	/	/
3.2.3	支持	/	/	/	/

CKafka to CKafka 公网实例同步任务说明

最近更新时间：2024-10-09 15:41:31

CKafka to CKafka 公网实例同步规则说明

CKafka 连接器 K2K 支持 自建Kafka（公网）到腾讯云 CKafka、自建 Kafka（公网）到自建 Kafka（公网）、腾讯云 CKafka 到自建 Kafka（公网）三种类型实例之间元数据、消息数据的同步，暂时不支持消费位点信息的同步。
公网同步包括自建 Kafka（公网）到 腾讯云 CKafka、自建 Kafka（公网）到自建 Kafka（公网）、腾讯云 CKafka 到自建 Kafka（公网）三种情况，这三种情况的可用性相同。

- 同步元数据规则：分为初始化同步和定时同步。
  - 初始化同步：如果任务启动，会查看 Topic 是否新建，如果没有，则尽可能保持和原来的配置一直去在下游初始化新建 Topic，然后启动任务。如果下游本身就存在对应 Topic，则不会触发初始化同步。
  - 定时同步：任务启动之后，会按一定时间周期性将上游的部分元数据同步到下游，同步周期：3分钟。
- 同步消息数据规则：将上游 CKafka 中存在的消息数据同步到下游 CKafka 对应 Topic 中，如果开启了同步到相同分区，消息会固定同步到下游对应的相同分区中。
- 涉及网络类型为公网的 K2K 同步不支持同步消费位点。

公网同步类型演示

自建 Kafka（公网）到 腾讯云 CKafka

腾讯云 CKafka 到自建 Kafka（公网）

自建 Kafka（公网）到自建 Kafka（公网）

仅同步元数据

支持同步的元数据

初始化同步支持的元数据	定时同步支持的元数据
• 分区数 • 副本数 • retention.ms • cleanup.policy • min.insync.replicas • unclean.leader.election.enable • segment.ms • retention.bytes • max.message.bytes • 消费组	• retention.ms（存在特殊情况） • cleanup.policy • min.insync.replicas • unclean.leader.election.enable • segment.ms • retention.bytes（存在特殊情况） • max.message.bytes • 消费组

说明：

- 定时同步暂不支持同步副本数。
- 定时同步存在如下特殊情况：retention.ms 和 retention.bytes 这两项元数据，目标 Topic 即下游对应元数据值为-1时，才会同步，其他情况下这两项元数据均不会定时同步。

元数据未同步情况下系统设定的默认值

说明：
该配置上游不存在情况或者非法会使用默认值替代。

元数据	默认值
retention.ms	604800000（7天）
cleanup.policy	delete
min.insync.replicas	1
unclean.leader.election.enable	false
segment.ms	604800000
retention.bytes	默认值取决于 Kafka 配置
max.message.bytes	1048588

同步元数据和消息数据

元数据

支持同步的元数据

初始化同步支持的元数据	定时同步支持的元数据
• 分区数 • 副本数 • retention.ms • cleanup.policy • min.insync.replicas • unclean.leader.election.enable • segment.ms • retention.bytes • max.message.bytes • 消费组	• retention.ms（存在特殊情况） • cleanup.policy • min.insync.replicas • unclean.leader.election.enable • segment.ms • retention.bytes（存在特殊情况） • max.message.bytes • 消费组

说明：
- 定时同步暂不支持同步副本数。
- 定时同步存在如下特殊情况：retention.ms 和 retention.bytes 这两项元数据，目标 Topic 即下游对应元数据值为-1时，才会同步，其他情况下这两项元数据均不会定时同步。

元数据未同步情况下系统设定的默认值

元数据	默认值
retention.ms	604800000（7天）
cleanup.policy	delete
min.insync.replicas	1
unclean.leader.election.enable	false
segment.ms	604800000
retention.bytes	默认值取决于 Kafka 配置

max.message.bytes	1048588

消息数据

支持正常传输。

> **说明：**
> - 支持消息数据同步到相同分区。
> - 消息数据在同步时，下游如果没有同名 Topic，由于公网延时较大，则创建完任务之后承诺分钟级别的延迟同步消息数据。