消息队列 CKafka 版

实践教程

腾讯云

文档目录

实践教程
集群资源评估
高 CPU 场景下集群带宽实践教程

最近更新时间：2024-10-15 15:05:41

适用场景

- CKafka 专业版。
- 集群带宽规格 > 3000 MB。
- 某些特殊业务场景下，带宽使用率低但是 CPU 使用率高，需要结合 CPU 使用率进行集群带宽扩容。

扩容策略

建议单节点的 CPU 使用率不高于 60%，如果高于 60%，需要考虑当前集群的 CPU 使用率进行线性扩容。

示例

集群现状

- 集群带宽规格 3000 MB，CPU 平均使用率 80%，带宽使用率 50%。
- 查看节点级的 CPU 使用率信息，可参见：高级监控（专业版）

扩容目标

- 业务流量增加 50%。
- CPU 使用率降低至 60%。

扩容方案

- 扩容系数 = 80% / 60% * (1+50%) = 2
- 预留 buffer = 30%
- 扩容后规格 = 3000 MB * 扩容系数 * 预留 buffer = 3000 MB * 2 * (1+30%) = 7800 MB

集群容量规划实践教程

最近更新时间：2024-09-10 16:15:51

在使用消息队列 CKafka 时，规格主要是带宽和存储，同时还包括可用区分布、分区数等，这些指标一定程度上决定了集群的负载能力。然而，在实际的运行过程中，由于业务场景的差异性，集群的实际负荷可能会受到多种因素的影响，例如：消息大小、消息是否有压缩、消息的收发比例、Topic 的副本数及其关键属性等。所以，单纯的将集群带宽、存储占比等作为集群是否扩容的唯一判断指标，是不够全面的。
为了更好地保障业务的稳定运行、合理地规划和管理集群容量，目前在高级监控提供了**集群负载**的指标。该指标可以帮助您以更简单的方式，获取当前集群的负荷情况，从而为评估当前 CKafka 集群是否需要扩容作参考。

适用场景

- CKafka 专业版。
- 某些特殊业务场景下，带宽使用率低但是集群负载高，需要结合集群负载指标进行集群带宽扩容。

指标查看路径

查看节点级的集群整体负载情况，具体可参见 查询高级监控（专业版）。

参考策略

为了确保您生产业务的稳定性和 CKafka 集群的处理性能，建议根据集群的部署方式和负载情况，合理规划集群容量。如果集群负载超过以下参考值，建议及时增加集群带宽规格。

- 单可用区部署

  当集群部署在单个可用区时，建议集群负载最大值保持在 70% 左右。
- 多可用区部署

  当集群部署在多个可用区时，需要考虑一定的冗余，以便于当某个 AZ 发生非预期异常时，剩余可用的 AZ 可以正常负荷业务运行。例如：
  - 2 AZ 部署：当单 AZ 不可用时，集群剩余一半节点，结合 70% 的使用率，建议集群常态负载保持在 35% 以下。
  - 3 AZ 部署：当单 AZ 不可用时，集群剩余 2/3 节点，结合 70% 的使用率，建议集群常态负载保持在 47% 以下。

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。

客户端实践教程
生产消费实践教程

最近更新时间：2024-12-10 16:22:23

本文主要介绍消息队列 CKafka 版生产和消费消息的最佳实践，帮助您降低消费消息出错的可能性。

生产消息

Topic 使用推荐

配置要求：**推荐节点的整倍数副本，减少数据倾斜问题，同步复制最小同步副本数为2**，且同步副本数不能等于 Topic 副本数，否则宕机1个副本会导致无法生产消息。
创建方式：支持选择是否开启 CKafka 自动创建 Topic 的开关。选择开启后，表示生产或消费一个未创建的 Topic 时，会自动创建一个包含3个分区和2个副本的 Topic。

分区数估计

为实现尽可能的数据均衡，分区数建议为节点数的整倍数，同时在知道预估流量的基础上，按照1MB/s一个分区设置分区数，如100MB的 Topic 吞吐，建议设置分区数为100。

失败重试

分布式环境下，由于网络等原因，消息偶尔会出现发送失败的情况，其原因可能是消息已经发送成功但是 ACK 机制失败或者是消息确实没有发送成功。
您可以根据业务需求，设置以下重试参数：

参数	说明
retries	重试次数，默认值为3，就对于数据丢失零容忍的应用而言，请考虑设置为：Integer.MAX_VALUE（有效且最大）。
retry.backoff.ms	重试间隔，建议设置为1000。

这样将能够应对 Broker 的 Leader 分区出现无法立刻响应 Producer 请求的情况。

异步发送

发送接口是异步的，如果您想接收发送的结果，可用使用 send 方法中的Callback接口进行获取发送结果。

一个 Producer 对应一个应用

Producer 是线程安全的，且可以往任何 Topic 发送消息。通常情况下，建议一个应用对应一个 Producer。

Acks

Kafka 的 ACK 机制，指 Producer 的消息发送确认机制，在 Kafka 的 0.10.x 版本上，其设置值是 Acks，而在 0.8.x 版本上，则为 request.required.acks，Acks 的设置将直接影响到 Kafka 集群的吞吐量和消息可靠性。
Acks 的参数说明如下：

参数	说明
acks=0	无需服务端的 Response。性能较高、丢数据风险较大。
acks=	服务端主节点写成功即返回 Response。性能中等、丢数据风险中等、主节点宕机可能导致数据丢失。

1	
acks=all	服务端主节点写成功且 ISR 中的节点同步成功才返回 Response。性能较差、数据较为安全、主节点和备节点都宕机才会导致数据丢失。

一般建议选择 acks=1，重要的服务可以设置 acks=all。

Batch

一般情况下，消息队列 CKafka 的 Topic 会有多个分区，Producer 客户端在向服务端发送消息时，需要先确认往哪个 Topic 的哪个分区发送。在给同一个分区发送多条消息时，Producer 客户端会将相关消息打包成一个 Batch，批量发送到服务端。Producer 客户端在处理 Batch 时，是有额外开销的。一般情况下，小 Batch 会导致 Producer 客户端产生大量请求，造成请求队列在客户端和服务端的排队，并造成相关机器的 CPU 升高，从而整体推高了消息发送和消费延迟。一个合适的 Batch 大小，可以减少发送消息时客户端向服务端发起的请求次数，在整体上提高消息发送的吞吐和延迟。
Batch 参数说明如下：

参数	说明
batch.size	发往每个分区（Partition）的消息缓存量（消息内容的字节数之和，不是条数）。达到设置的数值时，就会触发一次网络请求，然后 Producer 客户端把消息批量发往服务器。
linger.ms	每条消息在缓存中的最长时间。若超过这个时间，Producer 客户端就会忽略 batch.size 的限制，立即把消息发往服务器。
buffer.memory	所有缓存消息的总体大小超过这个数值后，就会触发把消息发往服务器，此时会忽略 batch.size 和 linger.ms 的限制。buffer.memory 的默认数值是 32MB，对于单个 Producer 而言，可以保证足够的性能。

说明：
如果您在同一个 JVM 中启动多个 Producer，那么每个 Producer 都有可能占用 32MB 缓存空间，此时便有可能触发 OOM（Out of Memory），此时您需要考虑 buffer.memory 的大小，避免触发 OOM。
您可以根据具体业务需求进行参数设置值的调整。Producer 客户端什么时候把消息批量发送至服务器是由 batch.size 和 linger.ms 共同决定的。您可以根据具体业务需求进行调整。为了提升发送的性能，保障服务的稳定性， 建议您设置 batch.size=16384 和 linger.ms=1000。

Key 和 Value

消息队列 CKafka 的消息有 Key（消息标识）和 Value（消息内容）两个字段。
为了便于追踪，请为消息设置一个唯一的 Key。您可以通过 Key 追踪某消息，打印发送日志和消费日志，了解该消息的生产和消费情况。
如果消息发送量较大，建议不要设置 Key，并使用黏性分区策略。

黏性分区

只有发送到相同分区的消息，才会被放到同一个 Batch 中，因此决定一个 Batch 如何形成的一个因素是消息队列 Kafka Producer 端设置的分区策略。消息队列 Kafka Producer 允许通过设置 Partitioner 的实现类来选择适合自己业务的分区。在消息指定 Key 的情况下，消息队列 Kafka Producer 的默认策略是对消息的Key进行哈希，然后根据哈希结果选择分区，保证相同 Key 的消息会发送到同一个分区。
在消息没有指定 Key 的情况下，消息队列 Kafka 2.4版本之前的默认策略是循环使用主题的所有分区，将消息以轮询的方式发送到每一个分区上。但是，这种默认策略 Batch 的效果会比较差，在实际使用中，可能会产生大量的小 Batch，从而使得实际的延迟增加。鉴于该默认策略对无 Key 消息的分区效率低问题，消息队列 Kafka 在2.4版本引入了黏性分区策略（Sticky Partitioning Strategy）。
黏性分区策略主要解决无 Key 消息分散到不同分区，造成小 Batch问题。其主要策略是如果一个分区的 Batch 完成后，就随机选择另一个分区，然后后续的消息尽可能地使用该分区。这种策略在短时间内看，会将消息发送到同一个分区，如果拉长整个运行时间，消息还是可以均匀地发布到各个分区上的。这样可以避免消息出现分区倾斜，同时还可以降低延迟，提升服务整体性能。
如果您使用的消息队列 Kafka Producer 客户端是2.4及以上版本，默认的分区策略就采用黏性分区策略。如果您使用的 Producer 客户端版本小于2.4，可以根据黏性分区策略原理，自行实现分区策略，然后通过参数 partitioner.class 设置指定的分区策略。
关于黏性分区策略实现，您可以参见如下 Java 版代码实现。该代码的实现逻辑主要是根据一定的时间间隔，切换一次分区。

```
public class MyStickyPartitioner implements Partitioner {

    // 记录上一次切换分区时间。
    private long lastPartitionChangeTimeMillis = 0L;
    // 记录当前分区。
    private int currentPartition = -1;
    // 分区切换时间间隔，可以根据实际业务选择切换分区的时间间隔。
    private long partitionChangeTimeGap = 100L;

    public void configure(Map<String, ?> configs) {}

    /**
     * Compute the partition for the given record.
     *
     * @param topic The topic name
     * @param key The key to partition on (or null if no key)
     * @param keyBytes serialized key to partition on (or null if no key)
     * @param value The value to partition on or null
     * @param valueBytes serialized value to partition on or null
     * @param cluster The current cluster metadata
     */
    public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[]
valueBytes, Cluster cluster) {

        // 获取所有分区信息。
        List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
        int numPartitions = partitions.size();

        if (keyBytes == null) {
            List<PartitionInfo> availablePartitions =
cluster.availablePartitionsForTopic(topic);
            int availablePartitionSize = availablePartitions.size();

            // 判断当前可用分区。
            if (availablePartitionSize > 0) {
                handlePartitionChange(availablePartitionSize);
                return availablePartitions.get(currentPartition).partition();
            } else {
                handlePartitionChange(numPartitions);
                return currentPartition;
            }
        } else {
            // 对于有key的消息，根据key的哈希值选择分区。
            return Utils.toPositive(Utils.murmur2(keyBytes)) % numPartitions;
        }
    }

    private void handlePartitionChange(int partitionNum) {
        long currentTimeMillis = System.currentTimeMillis();

        // 如果超过分区切换时间间隔，则切换下一个分区，否则还是选择之前的分区。
        if (currentTimeMillis - lastPartitionChangeTimeMillis >= partitionChangeTimeGap
            || currentPartition < 0 || currentPartition >= partitionNum) {
            lastPartitionChangeTimeMillis = currentTimeMillis;
```

```
            currentPartition = Utils.toPositive(ThreadLocalRandom.current().nextInt()) %
partitionNum;
        }
    }

    public void close() {}

}
```

分区顺序

单个分区（Partition）内，消息是按照发送顺序储存的，是基本有序的。每个主题下面都有若干分区，如果消息被分配到不同的分区中，不同 Partition 之间不能保证顺序。
如果需要进行消息具有消费顺序性，可以在生产端指定这一类消息的 key，这类消息都用相同的 key 进行消息发送，CKafka 就会根据 key 哈希取模选取其中一个分区进行存储，由于一个分区只能由一个消费者进行监听消费，此时消息就具有消息消费的顺序性了。

CKafka 顺序消息场景实践教程

顺序消息场景

在 CKafka 中，确保消息顺序性的主要手段依赖于其分区（Partition）设计以及消息 Key 的使用。客户端所涉及的顺序消息使用场景可分为两类：一是全局顺序场景，二是分区顺序场景。针对这两种场景，CKafka的实践教程如下：

1. 全局顺序：为保证全局顺序，您在 Ckafka 控制台，需设置 Topic 分区为1，副本数客户可以根据具体使用场景和可用性要求平衡成本指定，建议设置为2。

   > ⚠ **注意：**
   > 全局顺序由于单分区存在吞吐上限，因此整体吞吐不会太高，单分区吞吐指标请参见 使用限制 。

2. 分区顺序：为保证分区顺序，您在 Ckafka 控制台，可以根据预估 Topic 的业务流量，除以单分区流量，取整后获得分区数，同时为避免数据倾斜，分区数尽量向节点整倍数取整，从而确定最终合理的分区数。单分区的吞吐量可参见：使用限制 。在发送 Kafka 消息时候，需要指定 Key，Kafka 会根据 Key 计算出一个哈希值，确保具有相同 Key 的消息会被发送到同一个分区，从而确保这些消息在分区内部是有序的。

   > ⚠ **注意：**
   > 尽可能让业务 Key 分散，如果生产消息都指定同一个 Key，那么分区顺序会退化为全局顺序，从而降低整体的写入吞吐。

参数实践教程

由于顺序消息，要求消息有序，不重复，默认的 Kafka 生产者发送参数当遇到网络抖动，Kafka Broker 节点变化，分区 Leader 选举等场景，容易出现消息重复，乱序问题，因此顺序场景，必须对 Kafka 生产者参数进行特别设置，关键参数设置如下：

- **enable.idempotence**

  enable.idempotence表示是否开启幂等功能。顺序场景建议开启幂等功能，应对上述场景出现的分区消息乱序，消息重复等问题。建议Kafka的Producer设置：enable.idempotence 为true。需要注意，该功能要确保kafka的Broker版本大于等于0.11，即Kafka versions >= 0.11，同时：从 Kafka 3.0开始包括3.0，Kafka的producer 默认具有 enable.idempotence=true 和 acks=all ，而对于Kafka版本>=0.11同时Kafka<3.0的版本，默认是关闭幂等的，因此建议顺序场景建议显式指定该参数值确保开启幂等。

- **acks**

  在开启幂等后，acks需要显式指定为all，如果不指定为all的话，会无法通过参数校验从而报错。

- **max.in.flight.requests.per.connection**

  默认情况下，Kafka 生产者会尝试尽快发送记录，max.in.flight.requests.per.connection表示对于一个connection，同时发送最大请求数，默认值是5。Kafka在0.11版本之后包括0.11，小于1.1的版本，即（Kafka >= 0.11 & < 1.1），KafkaBroker没有针对该方面优化，需要设置max.in.flight.requests.per.connection为1，在Kafka>=1.1后，针对幂等场景的吞吐进行优化，在Broker端会维持一个队列对5个并发批次的消息的顺序进行顺序校验，允许max.in.flight.requests.per.connection设置5，但不能大于5。
  因此建议：
  - Kafka >= 0.11 & < 1.1：显式设置max.in.flight.requests.per.connection为1。

- Kafka>=1.1：显式设置max.in.flight.requests.per.connection可以为1<=max.in.flight.requests.per.connection<=5；建议设置为5。

- **retries**

  在顺序场景下，建议指定重试参数，retries在不同版本，有不同的默认行为，在Kafka <= 2.0，默认为0；Kafka >= 2.1，默认为Integer.MAX_VALUE，即2147483647；建议顺序场景，显式设置为Integer.MAX_VALUE。

总结

在顺序场景中，需要开启的生产者参数示例如下：

Kafka >= 0.11 & < 1.1：

```
// create Producer properties
Properties properties = new Properties();
properties.setProperty("enable.idempotence", "true");
properties.setProperty("acks", "all");
properties.setProperty("max.in.flight.requests.per.connection", "1");
properties.setProperty("retries", Integer.toString(Integer.MAX_VALUE));
```

Kafka>=1.1：

```
// create Producer properties
Properties properties = new Properties();
properties.setProperty("enable.idempotence", "true");
properties.setProperty("acks", "all");
properties.setProperty("max.in.flight.requests.per.connection", "5");
properties.setProperty("retries", Integer.toString(Integer.MAX_VALUE));
```

数据倾斜

Kafka Broker数据倾斜问题通常是由于分区分布不均匀或者生产者发送数据的key分布不均匀导致的，会引发几类问题：

1. 整体流量没有限流，但是节点局部限流；
2. 某些节点负载过快，导致整体kafka使用率不高，影响整体吞吐。

针对该类问题可以通过以下方式进行优化：

3. 使用合理分区数，分区数保障为节点数的整倍数。
4. 合理的分区策略，例如：RoundRobin（轮询）、Range（范围）和Sticky（粘性）或者自定义的分区策略，均衡发送消息。
5. 查是否使用Key进行发送，如果使用了Key进行发送，尽量设计策略让Key更加分区均衡。

消费消息

消费消息基本流程

1. Poll 数据。
2. 执行消费逻辑。
3. 再次 poll 数据。

负载均衡

每个 Consumer Group 可以包含多个 Consumer，并将参数 group.id 设置成相同的值，属于同一个 Consumer Group 的 Consumer 会负责消费订阅的 Topic。

例如：Consumer Group A 订阅了 Topic A，并开启三个消费实例 C1、C2、C3，则发送到 Topic A 的每条消息最终只会传给 C1、C2、C3 的某一个。CKafka 默认会均匀地把消息传给各个消费实例，以做到消费负载均衡。

CKafka 负载均衡的内部原理是：把订阅的 Topic 的分区，平均分配给各个 Consumer。因此，Consumer 的个数不要大于分区的数量，否则会有消费实例分配不到任何分区而处于空跑状态，尽量保证消费者数量能被分区总数整除。除了第一次启动上线之外，后续消费实例发生重启、增加、减少，分区数发生增加等变更时，都会触发一次重均衡。

频繁出现 Rebalance

如果频繁出现 Rebalance，可能有以下几种可能：

1. 消费者消费处理耗时很长
2. 消费某一个异常消息导致消费者阻塞或者失败
3. 心跳超时会引发 Rebalance
4. v0.10.2之前版本的客户端：Consumer 没有独立线程维持心跳，而是把心跳维持与 poll 接口耦合在一起。其结果就是，如果用户消费出现卡顿，就会导致Consumer 心跳超时，引发 Rebalance。v0.10.2及之后版本的客户端：如果消费时间过慢，超过一定时间（max.poll.interval.ms设置的值，默认5分钟）未进行 poll 拉取消息，则会导致客户端主动离开队列，而引发 Rebalance。

可以通过优化消费处理提高消费速度和参数调整等方法解决：

1. 消费端需要和 Broker 版本保持一致。
2. 参考以下说明调整参数值：
   - session.timeout.ms：v0.10.2之前的版本可适当提高该参数值，需要大于消费一批数据的时间，但不要超过30s，建议设置为25s；而v0.10.2及其之后的版本，保持默认值10s即可。
   - max.poll.records：降低该参数值，建议远远小于<单个线程每秒消费的条数> * <消费线程的个数> * <max.poll.interval.ms> / 1000的值。
   - max.poll.interval.ms：该值要大于<max.poll.records> / (<单个线程每秒消费的条数> * <消费线程的个数>)的值。
3. 尽量提高客户端的消费速度，消费逻辑另起线程进行处理，针对耗时进行监控。
4. 减少 Group 订阅 Topic 的数量，一个 Group 订阅的 Topic 最好不要超过5个，建议一个 Group 只订阅一个 Topic。

订阅关系

> **说明：**
> 同一个 Consumer Group 内，建议客户端订阅的 Topic 保持一致，即一个 Consumer Group 订阅一个 Topic，避免给排查问题带来更多复杂度。

Consumer Group 订阅多个 Topic

一个 Consumer Group 可以订阅多个Topic，多个 Topic 的消息被 Cosumer Group 中的 Consumer 均匀消费。例如 Consumer Group A 订阅了 Topic A、Topic B、Topic C，则这三个 Topic 中的消息，被 Consumer Group 中的 Consumer 均匀消费。
Consumer Group 订阅多个 Topic 的示例代码如下：

```
String topicStr = kafkaProperties.getProperty("topic");
String[] topics = topicStr.split(",");
for (String topic: topics) {
subscribedTopics.add(topic.trim());
}
consumer.subscribe(subscribedTopics);
```

Topic 被多个 Consumer Group 订阅

一个 Topic 可以被多个 Consumer Group 订阅，且各个 Consumer Group 独立消费Topic下的所有消息。例如 Consumer Group A 订阅了 Topic A，Consumer Group B也订阅了Topic A，则发送到Topic A的每条消息，不仅会传一份给Consumer Group A的消费实例，也会传一份给Consumer Group B的消费实例，且这两个过程相互独立，相互没有任何影响。

一个 Consumer Group 对应一个应用

建议一个 Consumer Group 对应一个应用，即不同的应用对应不同的代码。如果您需要将不同的代码写在同一个应用中，请准备多份不同的 kafka.properties。例如：kafka1.properties、kafka2.properties。

消费位点 Offset

每个 Topic 会有多个分区，每个分区会统计当前消息的总条数，这个称为最大位点 MaxOffset。
消息队列 CKafka 的 Consumer 会按顺序依次消费分区内的每条消息，记录已经消费了的消息条数，称为消费位点 ConsumerOffset。
剩余的未消费的条数（也称为消息堆积量）=MaxOffset-ConsumerOffset。

offset 提交

消息队列 CKafka 的 Consumer 有两个相关参数：

- enable.auto.commit：默认值为true。
- auto.commit.interval.ms： 默认值为5000，即5s。

这两个参数组合的结果为：每次 poll 数据前会先检查上次提交位点的时间，如果距离当前时间已经超过参数 auto.commit.interval.ms 规定的时长，则客户端会启动位点提交动作。

因此，如果将 enable.auto.commit 设置为 true，则需要在每次 poll 数据时，确保前一次 poll 出来的数据已经消费完毕，否则可能导致位点跳跃。

如果想自己控制位点提交，请把 enable.auto.commit 设为 false，并调用 commit(offsets) 函数自行控制位点提交。

> **注意：**
> 尽量避免提交位点请求过于频繁，否则容易导致 Broker CPU 很高，影响正常的服务。例如自动提交位点设置 auto.commit.interval.ms 为100ms，手动提交位点，在高吞吐场景下，每消费一条消息提交一个位点。

重置 offset

以下两种情况，会发生消费位点重置：

- 当服务端不存在曾经提交过的位点时（例如客户端第一次上线）。
- 当从非法位点拉取消息时（例如某个分区最大位点是10，但客户端却从11开始拉取消息）。

Java 客户端可以通过 auto.offset.reset 来配置重置策略，主要有三种策略：

- latest：从最大位点开始消费。
- earliest：从最小位点开始消费。
- none：不做任何操作，即不重置。

> **说明：**
> 建议设置成 latest，而不要设置成 earliest，避免因位点非法时从头开始消费，从而造成大量重复。
> 如果是您自己管理位点，可以设置成 none。

拉取消息

消费过程是由客户端主动去服务端拉取消息的，在拉取大消息时需要控制拉取速度，注意以下参数设置：

- max.poll.records：如果单条消息超过 1MB，建议设置为1。
- max.partition.fetch.bytes：设置比单条消息的大小略大一点。
- fetch.max.bytes：设置比单条消息的大小略大一点。

通过公网消费消息时，通常会因为公网带宽的限制导致连接被断开，此时需要注意控制拉取速度，修改配置：

- fetch.max.bytes：建议设置成公网带宽的一半（注意该参数的单位是bytes，公网带宽的单位是bits）。
- max.partition.fetch.bytes：建议设置成fetch.max.bytes的三分之一或者四分之一。

拉取大消息

消费过程是由客户端主动去服务端拉取消息的，在拉取大消息时，需要注意控制拉取速度，注意修改配置：

- max.poll.records：每次Poll获取的最大消息数量。如果单条消息超过1 MB，建议设置为1。
- fetch.max.bytes：设置比单条消息的大小略大一点。
- max.partition.fetch.bytes：设置比单条消息的大小略大一点。

拉取大消息的核心是逐条拉取的。

消息重复和消费幂等

消息队列 CKafka 消费的语义是 at least once， 也就是至少投递一次，保证消息不丢失，但是无法保证消息不重复。在出现网络问题、客户端重启时均有可能造成少量重复消息，此时应用消费端如果对消息重复比较敏感（例如订单交易类），则应该做消息幂等。

以数据库类应用为例，常用做法为：

发送消息时，传入 key 作为唯一流水号 ID。
消费消息时，判断 key 是否已经消费过，如果已经消费过了，则忽略，如果没消费过，则消费一次。
当然，如果应用本身对少量消息重复不敏感，则不需要做此类幂等检查。

消费失败

消息队列 CKafka 是按分区逐条消息顺序向前推进消费的，如果消费端拿到某条消息后执行消费逻辑失败，例如应用服务器出现了脏数据，导致某条消息处理失败，等待人工干预，那么有以下两种处理方式：
失败后一直尝试再次执行消费逻辑。这种方式有可能造成消费线程阻塞在当前消息，无法向前推进，造成消息堆积。
由于消息队列 CKafka 没有处理失败消息的设计，实践中通常会打印失败的消息或者存储到某个服务（例如创建一个 Topic 专门用来放失败的消息），然后定时检查失败消息的情况，分析失败原因，根据情况处理。

消费延迟

消费过程是由客户端主动去服务端拉取消息。一般情况下，如果客户端能够及时消费，则不会产生较大延迟。若产生了较大延迟，请先关注是否有堆积，并注意提高消费速度。

消费堆积

通常造成消息堆积的原因是：

- 消费速度跟不上生产速度，此时应该提高消费速度。
- 消费端产生了阻塞。
- 消费端拿到消息后，执行消费逻辑，通常会执行一些远程调用，如果这个时候同步等待结果，则有可能造成一直等待，消费进程无法向前推进。

消费端应该尽量避免堵塞消费线程，如果存在等待调用结果的情况，建议设置等待的超时时间，超时后作为消费失败进行处理。

提高消费速度

增加 Consumer 实例个数提高并行处理能力，如果消费者和分区数已经1:1，可以考虑增加分区数（注意：对于 flink 自动维护分区的场景不会自动感知新增分区后可能需要修改相关代码后重启）。 可以在进程内直接增加（需要保证每个实例对应一个线程），也可以部署多个消费实例进程。

> **说明：**
> 实例个数超过分区数量后就不再能提高速度，将会有消费实例不工作。
> 增加消费线程。
> 1. 定义一个线程池。
> 2. Poll 数据。
> 3. 把数据提交到线程池进行并发处理。
> 4. 等并发结果返回成功后，再次 poll 数据执行。

套接字缓冲区（socket buffers）

在 Kafka 的 0.10.x 版本中，参数 receive.buffer.bytes 的默认值为 64KB。而在 Kafka 的 0.8.x 版本中，参数 socket.receive.buffer.bytes 的默认值为 100KB。
这两个默认值对于高吞吐量的环境而言都太小了，特别是如果 Broker 和 Consumer 之间的网络带宽延迟积（bandwidth-delay product）大于局域网（local areanetwork，LAN）时。
对于延迟为1ms或更多的高带宽的网络（例如 10Gbps 或更高），建议将套接字缓冲区设置为8或16MB。
如果您的内存不足，也至少考虑设置为 1MB。您也可以设置为 -1，它会让底层操作系统根据网络的实际情况，去调整缓冲区的大小。
但是，对于需要启动“热”分区的 Consumers 来说，自动调整可能不会那么快。

消息广播

CKafka 目前没有消息广播的语义，可以通过创建不同的 Group 来模拟实现。

消息过滤

CKafka 自身没有消息过滤的语义。实践中可以采取以下两个办法：

- 如果过滤的种类不多，可以采取多个 Topic 的方式达到过滤的目的。
- 如果过滤的种类多，则最好在客户端业务层面自行过滤。

实践中请根据业务具体情况进行选择，也可以综合运用上面两种办法。

消费某些分区不消费

消费者在消费过程中，可能遇到消费者在线，但是某些分区的位点一致不前进，可能原因如下：

1. 遇到一条异常消息，可能是超大消息，格式异常，导致消费者拉取消息时候，转换成业务位点。
2. 使用公网带宽，带宽较小，拉取大消息时候直接把带宽打满，导致在超时时间内拉取不到消息。
3. 消费者假死，导致不去拉取。

解决方式：

关掉消费者，在 CKafka 控制台设置位点，跳过某些异常消息，或者优化消费代码，然后重启消费者消费。

Confluent Go SDK

最近更新时间：2025-02-19 16:06:13

背景

TDMQ CKafka 是一个分布式流处理平台，用于构建实时数据管道和流式应用程序。它具备高吞吐量、低延迟、可伸缩性和容错性等特性。

- Sarama：Shopify 开发的一个 Kafka 库，提供了生产者、消费者、分区消费者等功能。该库的性能较好，社区支持也较为活跃。
- Confluent-Kafka-Go：由 Confluent 开发的 Kafka 库，提供了高级 API，易于使用。该库基于 librdkafka C 库，性能非常优秀，但安装和使用略显复杂。

本文着重介绍上述 Confluent Go 客户端的关键参数、实践教程以及常见问题。

生产者实践

版本选择

在使用 Confluent Go SDK 时，可以通过配置参数 "bootstrap.servers" 来指定 Kafka 集群的地址，而 Broker 的版本则可以通过"api.version.request"参数来设置，这样 Confluent Go SDK 会在启动时自动检测 Broker 的版本。

```
config := &kafka.ConfigMap{
    "bootstrap.servers": "localhost",
    "api.version.request": true,
}
```

生产者参数与调优

生产者参数

Confluent Go 是基于 librdkafka 开发的，在使用 Confluent Go 客户端写入 Kafka 的时候，需要配置的参数会透传 librdkafka，主要涉及如下关键参数，相关的参数和默认值如下：

```
package main

import (
    "fmt"
    "github.com/confluentinc/confluent-kafka-go/kafka"
)

func main() {
    config := &kafka.ConfigMap{
        "bootstrap.servers":                "localhost:9092",
        "acks":                             -1,          //ack方式，默认值为-1
        "client.id":                        "rdkafka",   //客户端ID
        "compression.type":                 "none",      //指定压缩方式
        "compression.level":                -1,          //压缩等级
        "batch.num.messages":               10000,       //默认一个批次最多聚合10000条消息，构成
MessageSet整批发送，提高性能
        "batch.size":                       1000000,     //构成MessageSet整批大小限制，默认限制最多不
超过1000000字节
        "queue.buffering.max.ms":           5,           //在构造消息批次(MessageSets)传输到Broker之
前，默认延迟5ms攒批消息
        "queue.buffering.max.messages":     100000,      //Producer攒批发送中，总的消息数不能超过
100000
```

```
        "queue.buffering.max.kbytes":       1048576,     //Producer攒批发送中，MessageSets
        "message.send.max.retries":         2147483647, //重试次数，默认2147483647
        "retry.backoff.ms":                 100,        //重试间隔时间，默认100ms
        "socket.timeout.ms":                60000,      //会话超时时间，默认60s

    }

    producer, err := kafka.NewProducer(config)
    if err != nil {
        panic(fmt.Sprintf("Failed to create producer: %s", err))
    }

    // 使用producer发送消息等操作...

    // 关闭producer
    producer.Close()
}
```

参数说明调优

关于 acks 参数优化

acks 参数用于控制生产者发送消息时的确认机制。该参数的默认值为-1，表示消息发送给 Leader Broker 后，Leader 确认以及相应的 Follower 消息都写入完成后才返回。acks 参数还有以下可选值：0，1，-1。在跨可用区场景，以及副本数较多的 Topic，acks 参数的取值会影响消息的可靠性和吞吐量。

- 在一些在线业务消息的场景下，吞吐量要求不大，可以将 acks 参数设置为-1，确保消息被所有副本接收和确认后才返回，从而提高消息的可靠性。
- 在日志采集等大数据或者离线计算的场景下，要求高吞吐（即每秒写入 Kafka 的数据量）的情况下，可以将 acks 设置为1，提高吞吐。

关于 buffering 参数优化（缓存）

默认情况下，传输同等数据量的情况下，多次请求和一次请求的网络传输，一次请求传输能有效减少相关计算和网络资源，提高整体写入的吞吐量。因此，可以通过这个参数设置优化客户端发送消息的吞吐能力。对于 Confluent kafka Go，默认提供5ms的攒批时间积攒消息。如果消息较小，可以适当增加queue.buffering.max.ms的时间。

关于压缩参数优化

Confluent Go 支持如下压缩参数：none, gzip, snappy, lz4, zstd。
在 Confluent Kafka Go 客户端中，支持以下几种压缩算法：

- none：不使用压缩算法。
- gzip：使用 GZIP 压缩算法。
- snappy：使用 Snappy 压缩算法。
- lz4：使用 LZ4 压缩算法。
- zstd：使用 ZSTD 压缩算法。

要在 Producer 客户端中使用压缩算法，需要在创建生产者时设置 compression.type 参数。例如，要使用LZ4压缩算法，可以将 compression.type 设置为 lz4，虽然压缩算法的 CPU 压缩，和 CPU 解压缩，发生客户端，是一种用计算换带宽的优化方式，但是由于 Broker 针对压缩消息存在校验行为会付出额外的计算成本，尤其是 Gzip 压缩，服务端的压缩计算成本会比较大，在某种程度上可能会出现得不偿失的情况，反而因为计算的增加导致 Broker 消息处理能力偏低，导致带宽吞吐更低。这种情况建议可以使用如下方式进行使用：

1. 在 Producer 端对消息数据独立压缩，生成压缩包数据：messageCompression，同时在消息的 key 存储压缩方式：

```
{"Compression","CompressionLZ4"}
```

2. 在 Producer 端将 messageCompression 当成正常消息发送。

3. 在 Consumer 端读取消息 key，获取使用的压缩方式，独立进行解压缩。

创建生产者实例

如果应用程序需要更高的吞吐量，则可以使用异步生产者，以提高消息的发送速度。同时，可以使用批量发送消息的方式，以减少网络开销和 IO 消耗。如果应用程序需要更高的可靠性，则可以使用同步生产者，以确保消息发送成功。同时，可以使用 ACK 确认机制和事务机制，以确保消息的可靠性和一致性。具体的参数调优参考生产者参数与调优。

```
package main

import (
    "fmt"
    "github.com/confluentinc/confluent-kafka-go/kafka"
)

func main() {
    // 配置Kafka Producer
    p, err := kafka.NewProducer(&kafka.ConfigMap{
        "bootstrap.servers": "localhost:9092",
        "acks":              "1",
        "compression.type":  "none",
        "batch.num.messages": "1000",
    })
    if err != nil {
        fmt.Printf("Failed to create producer: %s\n", err)
        return
    }

    // 发送消息
    for i := 0; i < 10; i++ {
        topic := "test-topic"
        value := fmt.Sprintf("hello world %d", i)
        message := &kafka.Message{
            TopicPartition: kafka.TopicPartition{Topic: &topic, Partition: kafka.PartitionAny},
            Value:          []byte(value),
        }
        p.Produce(message, nil)
    }

    // 关闭Kafka Producer
    p.Flush(15 * 1000)
    p.Close()
}
```

消费者实践

消费者参数与调优

消费者参数

```
package main

import (
```

```
    "fmt"
    "github.com/confluentinc/confluent-kafka-go/kafka"
)

func main() {
    // 配置Kafka Consumer
    c, err := kafka.NewConsumer(&kafka.ConfigMap{
        "bootstrap.servers": "localhost:9092",
        "group.id":          "test-group",
        "auto.offset.reset":"earliest",
        "fetch.min.bytes":1, //最小拉取字节数
        "fetch.max.bytes":52428800,//最大拉取字节数
        "fetch.wait.max.ms":"500",   //如果没有最新消费消息默认等待500ms
        "enable.auto.commit":true, //是否支持自动提交位点，默认true
        "auto.commit.interval.ms":5000,//自动提交位点间隔，默认5s
        "max.poll.interval.ms": 300000,//Consumer 在两次 poll 操作之间的最大延迟。默认5分钟
        "session.timeout.ms": 45000,//session时间，默认45s
        "heartbeat.interval.ms": 3000,//心跳时间，默认3s
    })
    if err != nil {
        fmt.Printf("Failed to create consumer: %s\n", err)
        return
    }

    // 订阅主题
    c.SubscribeTopics([]string{"test-topic"}, nil)

    // 手动提交位点
    for {
        ev := c.Poll(100)
        if ev == nil {
            continue
        }

        switch e := ev.(type) {
        case *kafka.Message:
            fmt.Printf("Received message: %s\n", string(e.Value))
            c.CommitMessage(e)
        case kafka.Error:
            fmt.Printf("Error: %v\n", e)
        }
    }

    // 关闭Kafka Consumer
    c.Close()
}
```

参数说明与调优

1. max.poll.interval.ms 是 Kafka Consumer 的一个配置参数，它用于指定 Consumer 在两次 poll 操作之间的最大延迟。这个参数的主要作用是控制 Consumer 的 liveness，也就是判断 Consumer 是否还活着。如果 Consumer 在 max.poll.interval.ms 指定的时间内没有进行 poll 操作，那么 Kafka 认为这个 Consumer 已经挂掉，会触发 Consumer 的 rebalance 操作。这个参数的设置需要根据实际的消费速度来调整。如果设置得太小，可能会导致 Consumer 频繁地触发 rebalance 操作，增加了 Kafka 的负担；如果设置得太大，可能会导致 Consumer 在出现问题时不能及时被 Kafka 检测到，从而影响了消息的消费。
2. 一般消费主要是 rebalance 时间频繁和消费线程阻塞问题，参考以下说明参数优化：

2.1 session.timeout.ms：v0.10.2之前的版本可适当提高该参数值，需要大于消费一批数据的时间，但不要超过30s，建议设置为25s；而v0.10.2及其之后的版本，保持默认值10s即可。

2.2 heartbeat.interval.ms：默认3s，设置该值 需要小于session.timeout.ms/3。

2.3 max.poll.interval.ms：默认5分钟，如果分区数和消费者较多，建议适当调大该值。该值要大于<max.poll.records> / (<单个线程每秒消费的条数> * <消费线程的个数>)的值。

> ⚠ **注意：**
> 如果消息处理是同步处理，即拉取消息、处理、再拉取下一个消息，需要做如下改造：
> - 根据需求调大 max.poll.interval.ms 时间。
> - 针对处理时间大于 max.poll.interval.ms 请求处理时间进行监控，采样打印超时时间。

3. 针对自动提交位点请求，建议 auto.commit.interval.ms 时间不要低于1000ms，因为频率过高的位点请求会导致 Broker CPU 很高，影响其他正常服务的读写。

创建消费者实例

Confluent Go 提供订阅的模型创建消费者，其中在提交位点方面，提供手动提交位点和自动提交位点两种方式。

自动提交位点

自动提交位点：消费者在拉取消息后会自动提交位点，无需手动操作。这种方式的优点是简单易用，但是可能会导致消息重复消费或丢失。

```
package main

import (
    "fmt"
    "github.com/confluentinc/confluent-kafka-go/kafka"
)

func main() {
    // 配置Kafka Consumer
    c, err := kafka.NewConsumer(&kafka.ConfigMap{
        "bootstrap.servers": "localhost:9092",
        "group.id":          "test-group",
        "auto.offset.reset": "earliest",
        "enable.auto.commit": true,     //是否启用自动提交位点。设置为true，表示启用自动提交位点。
        "auto.commit.interval.ms": 5000, //自动提交位点的间隔时间。设置为5000毫秒（即5秒），表示每5秒自动提交一次位点。
        "max.poll.interval.ms": 300000,//Consumer在一次poll操作中最长的等待时间。设置为300000毫秒（即5分钟），表示Consumer在一次poll操作中最多等待5分钟
        "session.timeout.ms": 10000,//指定Consumer与broker之间的会话超时时间,设置10秒
        "heartbeat.interval.ms": 3000, //指定Consumer发送心跳消息的间隔时间。设置为3000毫秒（即3秒）
    })
    if err != nil {
        fmt.Printf("Failed to create consumer: %s\n", err)
        return
    }

    // 订阅主题
    c.SubscribeTopics([]string{"test-topic"}, nil)

    // 自动提交位点
    for {
```

```
        ev := c.Poll(100)
        if ev == nil {
            continue
        }

        switch e := ev.(type) {
        case *kafka.Message:
            fmt.Printf("Received message: %s\n", string(e.Value))
        case kafka.Error:
            fmt.Printf("Error: %v\n", e)
        }
    }

    // 关闭Kafka Consumer
    c.Close()
```

手动提交位点

手动提交位点：消费者在处理完消息后需要手动提交位点。这种方式的优点是可以精确控制位点的提交，避免消息重复消费或丢失。但是需要注意，手动提交位点如果太频繁会导致 Broker CPU 很高，影响性能，随着消息量增加，CPU 消费会很高，影响正常 Broker 的其他功能，因此建议间隔一定消息提交位点。

```
package main

import (
    "fmt"
    "github.com/confluentinc/confluent-kafka-go/kafka"
)

func main() {
    // 配置Kafka Consumer
    c, err := kafka.NewConsumer(&kafka.ConfigMap{
        "bootstrap.servers": "localhost:9092",
        "group.id":          "test-group",
        "auto.offset.reset": "earliest",
        "enable.auto.commit": false,
        "max.poll.interval.ms": 300000,
        "session.timeout.ms": 10000,
        "heartbeat.interval.ms": 3000,
    })
    if err != nil {
        fmt.Printf("Failed to create consumer: %s\n", err)
        return
    }

    // 订阅主题
    c.SubscribeTopics([]string{"test-topic"}, nil)

    // 手动提交位点
    for {
        ev := c.Poll(100)
        if ev == nil {
            continue
        }
```

```
        switch e := ev.(type) {
        case *kafka.Message:
            fmt.Printf("Received message: %s\n", string(e.Value))
            c.CommitMessage(e)
        case kafka.Error:
            fmt.Printf("Error: %v\n", e)
        }
    }

    // 关闭Kafka Consumer
    c.Close()
```

Sarama Go

最近更新时间：2024-12-20 15:06:42

背景

TDMQ CKafka 是一个分布式流处理平台，用于构建实时数据管道和流式应用程序。它具备高吞吐量、低延迟、可伸缩性和容错性等特性。

- Sarama：Shopify 开发的一个 Kafka 库，提供了生产者、消费者、分区消费者等功能。该库的性能较好，社区支持也较为活跃。
- Confluent-Kafka-Go：由 Confluent 开发的 Kafka 库，提供了高级 API，易于使用。该库基于 librdkafka C 库，性能非常优秀，但安装和使用略显复杂。

本文着重介绍上述 Sarama Go 客户端的关键参数、最佳实践以及常见问题。

生产者实践

版本选择

在选择 Sarama 客户端版本时，需要确保所选版本与 Kafka broker 版本兼容。Sarama 库支持多个 Kafka 协议版本，可以通过设置 config.Version 来指定使用的协议版本。常见的 Kafka 协议版本与 Sarama 库版本的对应关系如下，目前最新版本请参见 Sarama 版本 。

Kafka 版本	Sarama 库版本	Sarama 协议版本常量
0.8.2.x	>= 1.0.0	sarama.V0_8_2_0
0.9.0.x	>= 1.0.0	sarama.V0_9_0_0
0.10.0.x	>= 1.0.0	sarama.V0_10_0_0
0.10.1.x	>= 1.0.0	sarama.V0_10_1_0
0.10.2.x	>= 1.0.0	sarama.V0_10_2_0
0.11.0.x	>= 1.16.0	sarama.V0_11_0_0
1.0.x	>= 1.16.0	sarama.V1_0_0_0
1.1.x	>= 1.19.0	sarama.V1_1_0_0
2.0.x	>= 1.19.0	sarama.V2_0_0_0
2.1.x	>= 1.21.0	sarama.V2_1_0_0
2.2.x	>= 1.23.0	sarama.V2_2_0_0
2.3.x	>= 1.24.0	sarama.V2_3_0_0
2.4.x	>= 1.27.0	sarama.V2_4_0_0
2.5.x	>= 1.28.0	sarama.V2_5_0_0
2.6.x	>= 1.29.0	sarama.V2_6_0_0
2.7.x	>= 1.29.0	sarama.V2_7_0_0
2.8.x以及以上	建议使用>=1.42.1	sarama.V2_8_0_0-sarama.V3_6_0_0

上述列出的 Sarama 库版本是支持对应 Kafka 协议版本的最低版本。为了获得最佳性能和使用新功能，建议使用 Sarama 的最新版本。在使用最新版本时，客户可以通过设置 config.Version 来指定与您的 Kafka broker 兼容的协议版本。设置方式如下，务必先设置版本后使用，否则会有预期外的不兼容问题：

```
config := sarama.NewConfig()
config.Version = sarama.V2_7_0_0 // 根据实际Kafka版本设置协议版本
```

生产者参数与调优

生产者参数

在使用 Sarama go 客户端写入 kafka 时候，需要配置如下关键参数，相关的参数和默认值如下：

```
config := sarama.NewConfig()
sarama.MaxRequestSize = 100 * 1024 * 1024  //请求最大大小，默认100MB，可以调整，写入大于100MB的消息会直接报错
sarama.MaxResponseSize = 100 * 1024 * 1024 //响应最大大小，默认100MB，可以调整，获取大于100MB的消息会直接报错

config.Producer.RequiredAcks = sarama.WaitForLocal // 默认值为sarama.WaitForLocal(1)

config.Producer.Retry.Max = 3                     // 生产者重试的最大次数，默认为3
config.Producer.Retry.Backoff = 100 * time.Millisecond // 生产者重试之间的等待时间，默认为100毫秒

config.Producer.Return.Successes = false          //是否返回成功的消息，默认为false
config.Producer.Return.Errors = true             // 返回失败的消息，默认值为true

config.Producer.Compression = CompressionNone   //对消息是否压缩后发送，默认CompressionNone不压缩
config.Producer.CompressionLevel = CompressionLevelDefault // 指定压缩等级，在配置了压缩算法后生效

config.Producer.Flush.Frequency = 0 //producer缓存消息的时间，默认缓存0毫秒
config.Producer.Flush.Bytes = 0        // 达到多少字节时，触发一次broker请求，默认为0，直接发送，存在天然上限值MaxRequestSize，因此默认最大100MB
config.Producer.Flush.Messages = 0       // 达到多少条消息时，强制，触发一次broker请求，这个是上限值，MaxMessages < Messages
config.Producer.Flush.MaxMessages = 0       // 最大缓存多少消息，默认为0，有消息立刻发送，MaxMessages设置大于0时，必须设置 Messages,且需要保证：MaxMessages > Messages

config.Producer.Timeout = 5 * time.Second        // 超时时间

config.Producer.Idempotent = false                    //是否需要幂等，默认false
config.Producer.Transaction.Timeout = 1 * time.Minute // 事务超时时间默认1分钟
config.Producer.Transaction.Retry.Max = 50             //事务重试时间
config.Producer.Transaction.Retry.Backoff = 100 * time.Millisecond
config.Net.MaxOpenRequests = 5                        //默认值5，一次发送请求的数量
config.Producer.Transaction.ID = "test"               //事务ID

config.ClientID = "your-client-id"               // 客户端ID
```

参数说明调优

关于 RequiredAcks 参数优化

RequiredAcks 参数用于控制生产者发送消息时的确认机制。该参数的默认值为 WaitForLocal，表示消息发送给 Leader Broker 后，Leader 确认消息写入后即返回。RequiredAcks 参数还有以下可选值：

- NoResponse: 不等待任何确认，直接返回。
- WaitForLocal: 等待 Leader 副本确认写入后返回。

- WaitForAll: 等待 Leader 副本以及相关的 Follower 副本确认写入后返回。

由上可知，在跨可用区场景，以及副本数较多的 Topic，RequiredAcks 参数的取值会影响消息的可靠性和吞吐量。因此：

- 在一些在线业务消息的场景下，吞吐量要求不大，可以将 RequiredAcks 参数设置为 WaitForAll，则可以确保消息被所有副本接收和确认后才返回，从而提高消息的可靠性。
- 在日志采集等大数据或者离线计算的场景下，要求高吞吐（即每秒写入 Kafka 的数据量）的情况下，可以将 RequiredAcks 设置为 WaitForLocal，提高吞吐。

关于 Flush 参数优化（缓存）

默认情况下，传输同等数据量的情况下，多次请求和一次请求的网络传输，一次请求传输能有效减少相关计算和网络资源，提高整体写入的吞吐量。因此，可以通过这个参数设置优化客户端发送消息的吞吐能力。在高吞吐场景下，可以配合计算和设置：
其中 Bytes 建议设置为16K，对齐 Kafka 标准 Java SDK 定义， 预估一条消息为1K（1024）个字节，因此得出如下 Messages 和 MaxMessages 的写入参数：
其中 Frequency 的计算方式为：预估流量为 16MB，分区数为16个分区，此时单分区每秒写入流量为：16 * 1024 * 1024 / 16 = 1 * 1024 * 1024 = 1MB，单个分区每秒 1MB 的流量。假设按照 16K 一个请求,数据量发送，那么在 1s 内要实现 1MB 的流量传输 1*1024*1024/16/1024 = 64个请求，因此 Frequency <= 15.62ms(1000/64)。
实际上，由于业务流量不是持续生产的，在低峰期，可能出现即时达到 16ms，也缓存不了太多的数据，因此在高吞吐的情况下，可以将条件简化，以 Bytes 为准，Frequency 可以适当调大，例如能接受 500ms 的延时增加，那么就可以设置为 500ms，因为此时如果命中数据量大于等于 Bytes，会按照 Bytes 的条件发送请求。

```
config.Producer.Flush.Frequency = 16 //producer缓存消息的时间， 默认缓存100毫秒,如果发送的流量较小，这里可以进一步增加延时时间。
config.Producer.Flush.Bytes = 16*1024        // 达到多少字节时，触发一次broker请求，默认为0，直接发送，存在天然上限值MaxRequestSize，因此默认最大100MB
config.Producer.Flush.Messages = 17       // 达到多少条消息时，强制，触发一次broker请求，这个是上限值，MaxMessages 需要小于 Messages
config.Producer.Flush.MaxMessages = 16   // 16条，实际上因为消息大小不严格1024字节，Messages和MaxMessages 建议配置值更大或者直接使用Int的最大值，
                                         //因为命中Frequency，Bytes，MaxMessages < Messages任何一个条件都会触发flush
```

关于事务参数优化

```
config.Producer.Idempotent = true                        //是否需要幂等，在事务场景下需要设置为true
config.Producer.Transaction.Timeout = 1 * time.Minute // 事务超时时间默认1分钟
config.Producer.Transaction.Retry.Max = 50               //事务重试时间
config.Producer.Transaction.Retry.Backoff = 100 * time.Millisecond
config.Net.MaxOpenRequests = 5                          //默认值5，一次发送请求的数量
config.Producer.Transaction.ID = "test"                  //事务ID
```

需要强调，事务因为要保障消息的 exactly once 语义，因此会额外付出更多的计算资源，所以 config.Net.MaxOpenRequests 的选取必须小于等于5，Broker 端的 ProducerStateManager 实例会缓存每个 PID 在每个 Topic-Partition 上发送的最近 5 个 batch 数据，如果客户在事务的基础上还需要保持一定的吞吐，因此可以设置该值为5，同时适当增加事务超时时间，容忍高负载下一些网络抖动带来的时延问题。

关于压缩参数优化

Sarama Go 支持如下压缩参数：

```
config.Producer.Compression = CompressionNone  //对消息是否压缩后发送，默认CompressionNone不压缩
```

```
config.Producer.CompressionLevel = CompressionLevelDefault //指定压缩等级，在配置了压缩算法后生效
```

在Sarama Kafka Go客户端中，支持以下几种压缩配置：
1. sarama.CompressionNone：不使用压缩。
2. sarama.CompressionGZIP：使用 GZIP 压缩。
3. sarama.CompressionSnappy：使用 Snappy 压缩。
4. sarama.CompressionLZ4：使用 LZ4 压缩。
5. sarama.CompressionZSTD：使用 ZSTD 压缩。
要在 Sarama Kafka Go 客户端中使用压缩消息，需要在创建生产者时设置 config.Producer.Compression 参数。例如，要使用 LZ4 压缩算法，可以将config.Producer.Compression 设置为 sarama.CompressionLZ4 ，虽然压缩消息的压缩和解压缩，发生客户端，是一种用计算换带宽的优化方式，但是由于Broker 针对压缩消息存在校验行为会付出额外的计算成本，尤其是 gzip 压缩，Broker 对其校验计算成本会比较大，在某种程度上可能会出现得不偿失的情况，反而因为计算的增加导致Broker消息处理能力偏低，导致带宽吞吐更低。在低吞吐或者低规格服务下，不建议使用压缩消息。如果还是需要压缩消息，这种情况建议可以使用如下方式进行使用：
1. 在 Producer 端对消息数据独立压缩，生成压缩包数据：messageCompression，同时在消息的 key 存储压缩方式：

```
{"Compression","CompressionLZ4"}
```

2. 在Producer端将messageCompression当成正常消息发送。
3. 在 Consumer 端读取消息key，获取使用的压缩方式，独立进行解压缩。

关于压缩参数内存优化

客户使用 Go 语言 Sarama 客户端并采用 LZ4 压缩算法压缩消息，LZ4 压缩消息在服务端需要解压缩进行校验，Sarama 客户端的 LZ4 压缩默认参数配置相较于官方 Java 客户端会申请更多内存，导致内存消耗过快。由于内存申请时间增长，Kafka 在处理生产请求时会导致大量处理线程在等待同分区锁的释放，容易引发请求队列持续打满，造成生产耗时增高，CPU 利用率增高，影响 Kafka 集群的生产消费。因此建议客户优化如下配置：
Go Sarama 客户端：LZ4 消息解压内存默认申请值为 4 MB，请求数翻倍时对共享集群冲击较大，因此建议客户配置为：64 KB。Java 客户端：LZ4 消息解压内存默认申请值 64 KB。
示例如下：

```
lz4.DefaultBlockSizeOption = lz4.BlockSizeOption(lz4.Block64Kb)
```

创建生产者实例

如果应用程序需要更高的吞吐量，则可以使用异步生产者，以提高消息的发送速度。同时，可以使用批量发送消息的方式，以减少网络开销和 IO 消耗。如果应用程序需要更高的可靠性，则可以使用同步生产者，以确保消息发送成功。同时，可以使用 ACK 确认机制和事务机制，以确保消息的可靠性和一致性。具体的参数调优参考生产者参数与调优。

同步生产者

在 Sarama Kafka Go 客户端中，有两种类型的生产者：同步生产者和异步生产者。它们的主要区别在于发送消息的方式和处理消息结果的方式。同步生产者：同步生产者在发送消息时会阻塞当前线程，直到消息发送完成并收到服务器的确认。因此，同步生产者的吞吐量较低，但是可以立即知道消息是否发送成功。示例如下：

```
package main

import (
    "fmt"
    "log"

    "github.com/Shopify/sarama"
)
```

```
func main() {
    config := sarama.NewConfig()
    config.Producer.RequiredAcks = sarama.WaitForLocal
    config.Producer.Return.Errors = true

    brokers := []string{"localhost:9092"}
    producer, err := sarama.NewSyncProducer(brokers, config)
    if err != nil {
        log.Fatalf("Failed to create producer: %v", err)
    }
    defer producer.Close()

    msg := &sarama.ProducerMessage{
        Topic: "test",
        Value: sarama.StringEncoder("Hello, World!"),
    }

    partition, offset, err := producer.SendMessage(msg)
    if err != nil {
        log.Printf("Failed to send message: %v", err)
    } else {
        fmt.Printf("Message sent to partition %d at offset %d\n", partition, offset)
    }
}
```

异步生产者

异步生产者：异步生产者在发送消息时不会阻塞当前线程，而是将消息放入一个内部的发送队列，然后立即返回。因此，异步生产者的吞吐量较高，但是需要通过回调函数来处理消息结果。

```
package main

import (
    "fmt"
    "log"

    "github.com/Shopify/sarama"
)

func main() {
    config := sarama.NewConfig()
    config.Producer.RequiredAcks = sarama.WaitForLocal
    config.Producer.Return.Errors = true

    brokers := []string{"localhost:9092"}
    producer, err := sarama.NewAsyncProducer(brokers, config)
    if err != nil {
        log.Fatalf("Failed to create producer: %v", err)
    }
    defer producer.Close()

    msg := &sarama.ProducerMessage{
        Topic: "test",
        Value: sarama.StringEncoder("Hello, World!"),
    }
```

```
    producer.Input() <- msg

    select {
    case success := <-producer.Successes():
        fmt.Printf("Message sent to partition %d at offset %d\n", success.Partition,
success.Offset)
    case err := <-producer.Errors():
        log.Printf("Failed to send message: %v", err)
    }
}
```

消费者实践

版本选择

在选择 Sarama 客户端版本时，需要确保所选版本与 Kafka broker 版本兼容。Sarama 库支持多个 Kafka 协议版本，可以通过设置 config.Version 来指定使用的协议版本。

```
config := sarama.NewConfig()
config.Version = sarama.V2_8_2_0
```

消费者参数与调优

```
config := sarama.NewConfig()
config.Consumer.Group.Rebalance.Strategy = sarama.NewBalanceStrategyRange //消费者分配分区的默认方式
config.Consumer.Offsets.Initial = sarama.OffsetNewest    //在没有提交位点情况下，使用最新的位点还是最老
的位点，默认是最新的消息位点
config.Consumer.Offsets.AutoCommit.Enable = true        //是否支持自动提交位点，默认支持
config.Consumer.Offsets.AutoCommit.Interval = 1 * time.Second //自动提交位点时间间隔，默认1s
config.Consumer.MaxWaitTime = 250 * time.Millisecond //在没有最新消费消息时候，客户端等待的时间，默认
250ms
config.Consumer.MaxProcessingTime = 100 * time.Millisecond
config.Consumer.Fetch.Min = 1                       //消费请求中获取的最小消息字节数,Broker将等待至少这么
多字节的消息然后返回。默认值为1，不能设置0，因为0会导致在没有消息可用时消费者空转。
config.Consumer.Fetch.Max = 0                       //消费请求最大的字节数。默认为0，表示不限制
config.Consumer.Fetch.Default = 1024 * 1024       //消费请求的默认消息字节数（默认为1MB），需要大于实例的
大部分消息，否则Broker会花费大量时间计算消费数据是否达到这个值的条件
config.Consumer.Return.Errors = true

config.Consumer.Group.Rebalance.Strategy = sarama.NewBalanceStrategyRange // 设置消费者组在进行
rebalance时所使用的策略为NewBalanceStrategyRange，默认NewBalanceStrategyRange
config.Consumer.Group.Rebalance.Timeout = 60 * time.Second    // 设置rebalance操作的超时时间，默认60s
config.Consumer.Group.Session.Timeout = 10 * time.Second  // 设置消费者组会话的超时时间为,默认为10s
config.Consumer.Group.Heartbeat.Interval = 3 * time.Second  // 心跳超时时间，默认为3s
config.Consumer.MaxProcessingTime = 100 * time.Millisecond //消息处理的超时时间，默认100ms，
```

参数说明与调优

一般消费主要是rebalance时间频繁和消费线程阻塞问题，参考以下说明参数优化：

- config.Consumer.Group.Session.Timeout：v0.10.2之前的版本可适当提高该参数值，需要大于消费一批数据的时间，但不要超过30s，建议设置为25s；而v0.10.2及其之后的版本，保持默认值10s 即可。

- config.Consumer.Group.Heartbeat.Interval：默认3s，设置该值 需要小于Consumer.Group.Session.Timeout/3。
- config.Consumer.Group.Rebalance.Timeout：默认60s，如果分区数和消费者较多，建议适当调大该值。
- config.Consumer.MaxProcessingTime：该值要大于<max.poll.records> / (<单个线程每秒消费的条数> * <消费线程的个数>)的值。

> ⚠ **注意：**
> - 根据需求调大 MaxProcessingTime 时间。
> - 针对处理时间大于 MaxProcessingTime 请求处理时间进行监控，采样打印超时时间。

创建消费者实例

Sarama 提供订阅的模型创建消费者，其中在提交位点方面，提供手动提交位点和自动提交位点两种方式。

自动提交位点

自动提交位点：消费者在拉取消息后会自动提交位点，无需手动操作。这种方式的优点是简单易用，但是可能会导致消息重复消费或丢失。

```
package main

import (
    "context"
    "fmt"
    "log"
    "os"
    "os/signal"
    "sync"
    "time"

    "github.com/Shopify/sarama"
)

func main() {
    config := sarama.NewConfig()
    config.Version = sarama.V2_1_0_0
    config.Consumer.Offsets.Initial = sarama.OffsetOldest
    config.Consumer.Offsets.AutoCommit.Enable = true
    config.Consumer.Offsets.AutoCommit.Interval = 1 * time.Second

    brokers := []string{"localhost:9092"}
    topic := "test-topic"

    client, err := sarama.NewConsumerGroup(brokers, "test-group", config)
    if err != nil {
        log.Fatalf("unable to create kafka consumer group: %v", err)
    }
    defer client.Close()

    ctx, cancel := context.WithCancel(context.Background())
    signals := make(chan os.Signal, 1)
    signal.Notify(signals, os.Interrupt)

    var wg sync.WaitGroup
    wg.Add(1)
```

```
    go func() {
        defer wg.Done()

        for {
            err := client.Consume(ctx, []string{topic}, &consumerHandler{})
            if err != nil {
                log.Printf("consume error: %v", err)
            }

            select {
            case <-signals:
                cancel()
                return
            default:
            }
        }
    }()

    wg.Wait()
}

type consumerHandler struct{}

func (h *consumerHandler) Setup(sarama.ConsumerGroupSession) error {
    return nil
}

func (h *consumerHandler) Cleanup(sarama.ConsumerGroupSession) error {
    return nil
}

func (h *consumerHandler) ConsumeClaim(sess sarama.ConsumerGroupSession, claim
sarama.ConsumerGroupClaim) error {
    for msg := range claim.Messages() {
        fmt.Printf("Received message: key=%s, value=%s, partition=%d, offset=%d\n",
string(msg.Key), string(msg.Value), msg.Partition, msg.Offset)
        sess.MarkMessage(msg, "")
    }
    return nil
}
```

手动提交位点

手动提交位点：消费者在处理完消息后需要手动提交位点。这种方式的优点是可以精确控制位点的提交，避免消息重复消费或丢失。但是需要注意，手动提交位点如果太频繁会导致 Broker CPU 很高，影响性能，随着消息量增加，CPU 消费会很高，影响正常 Broker 的其他功能，因此建议间隔一定消息提交位点。

```
package main

import (
    "context"
    "fmt"
    "log"
    "os"
    "os/signal"
```

```
    "sync"

    "github.com/Shopify/sarama"
)

func main() {
    config := sarama.NewConfig()
    config.Version = sarama.V2_1_0_0
    config.Consumer.Offsets.Initial = sarama.OffsetOldest
    config.Consumer.Offsets.AutoCommit.Enable = false

    brokers := []string{"localhost:9092"}
    topic := "test-topic"

    client, err := sarama.NewConsumerGroup(brokers, "test-group", config)
    if err != nil {
        log.Fatalf("unable to create kafka consumer group: %v", err)
    }
    defer client.Close()

    ctx, cancel := context.WithCancel(context.Background())
    signals := make(chan os.Signal, 1)
    signal.Notify(signals, os.Interrupt)

    var wg sync.WaitGroup
    wg.Add(1)

    go func() {
        defer wg.Done()

        for {
            err := client.Consume(ctx, []string{topic}, &consumerHandler{})
            if err != nil {
                log.Printf("consume error: %v", err)
            }

            select {
            case <-signals:
                cancel()
                return
            default:
            }
        }
    }()

    wg.Wait()
}

type consumerHandler struct{}

func (h *consumerHandler) Setup(sarama.ConsumerGroupSession) error {
    return nil
}

func (h *consumerHandler) Cleanup(sarama.ConsumerGroupSession) error {
```

```
    return nil
}

func (h *consumerHandler) ConsumeClaim(sess sarama.ConsumerGroupSession, claim
sarama.ConsumerGroupClaim) error {
    for msg := range claim.Messages() {
        fmt.Printf("Received message: key=%s, value=%s, partition=%d, offset=%d\n",
string(msg.Key), string(msg.Value), msg.Partition, msg.Offset)
        sess.MarkMessage(msg, "")
        sess.Commit()
    }
    return nil
}
```

Sarama Go 生产消费常见问题

1. 配置了手动提交位点，但是位点在控制台查询消费组时候没有原因。

   无论配置了手动提交位点还是自动提交位点，都需要先进行标记，sess.MarkMessage(msg, "")，表示该消息已经被消费完，然后才能提交位点。
2. Sarama Go 作为消费者的一些问题，Sarama Go 版本客户端存在以下已知问题：

   2.1 当Topic新增分区时，Sarama Go客户端无法感知并消费新增分区，需要客户端重启后，才能消费到新增分区。

   2.2 当Sarama Go客户端同时订阅两个以上的Topic时，有可能会导致部分分区无法正常消费消息。

   2.3 当Sarama Go客户端的消费位点重置策略设置为Oldest(earliest)时，如果客户端宕机或服务端版本升级，由于Sarama Go客户端自行实现OutOfRange机制，有可能会导致客户端从最小位点开始重新消费所有消息。

   2.4 对于该问题：Confluent Go客户端的Demo地址，请访问 kafka-confluent-go-demo 。
3. 出现报错：Failed to produce message to topic。

   原因可能为版本没有对齐，此时客户先确定kafka Broker的版本，然后指定版本：

```
config := sarama.NewConfig()
config.Version = sarama.V2_1_0_0
```

Java SDK

最近更新时间：2024-10-14 16:30:01

背景

TDMQ CKafka 是一个分布式流处理平台，用于构建实时数据管道和流式应用程序。它提供了高吞吐量、低延迟、可伸缩性和容错性等特性。
Kafka Clients：Kafka 自带的客户端，通过 Java 实现，是 Kafka 生产和消费标准协议的客户端。
本文着重介绍上述 Java 客户端的关键参数，实践教程以及常见问题。

生产者实践

版本选择

Kafka 客户端和集群之间的兼容性非常重要，通常情况下，较新版本的客户端可以兼容较旧版本的集群，但反之则不一定成立。一般情况下，CKafka实例 Broker 在部署后是明确的，因此可以直接根据 Broker 的版本选择相匹配的客户端的版本。
Java 生态中，广泛使用 Spring Kafka，其中 Spring Kafka 版本和 Kafka Broker 版本的对应关系，可以参见 Spring 官方网址的 版本对应关系 。

生产者参数与调优

生产者参数

在使用 Kafka Client 客户端写入 Kafka 时候，需要配置如下关键参数，相关的参数和默认值如下：

```
import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.serialization.StringSerializer;

import java.util.Properties;
import java.util.concurrent.ExecutionException;

public class KafkaProducerExample {

    private static final String BOOTSTRAP_SERVERS = "localhost:9092";
    private static final String TOPIC = "test-topic";

    public static void main(String[] args) throws ExecutionException, InterruptedException {
        // 创建Kafka生产者配置
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS); //Kafka集群的地址列表，格式为host1:port1,host2:port2。生产者会使用这个列表来找到集群并建立连接。
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

        // 设置生产者关键参数以及默认值
        props.put(ProducerConfig.ACKS_CONFIG, "1");//acks,默认值为1,消息确认的级别。0表示不等待确认;1表示等待Leader副本写入;all或者-1表示等待所有副本写入。
        props.put(ProducerConfig.BATCH_SIZE_CONFIG, 16384);//batch.size,批量发送的大小,单位为字节。生产者会将多个消息打包成一个批次发送，以提高性能。默认大小16384字节。
        props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 33554432);//buffer.memory,生产者用于缓存待发送消息的内存大小，单位为字节。默认33554432，也就是32MB
        props.put(ProducerConfig.CLIENT_ID_CONFIG, "");//client.id,客户端ID。这个ID可以用于在服务端日志中识别消息来源。
        props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "none");//compression.type消息压缩类型。默认none不压缩，可选值为none、gzip、snappy、lz4、zstd。
```

```
        props.put(ProducerConfig.CONNECTIONS_MAX_IDLE_MS_CONFIG,
540000);//connections.max.idle.ms连接的最大空闲时间，单位为毫秒。超过这个时间的空闲连接会被关闭。默认540s
        props.put(ProducerConfig.DELIVERY_TIMEOUT_MS_CONFIG, 120000);//delivery.timeout.ms消息的
最大投递时间，单位为毫秒。超过这个时间的未确认消息会被认为发送失败。默认120s
        props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, false);//enable.idempotence,是否启用
幂等性。如果启用，生产者会确保每个消息只被发送一次，即使在网络错误或重试的情况下。
        props.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG, "");//interceptor.classes拦截器类列
表。生产者会在发送消息前后调用这些拦截器。
        props.put(ProducerConfig.LINGER_MS_CONFIG, 0);//linger.ms延迟发送的时间，单位为毫秒。生产者会
等待一段时间以便将更多消息打包成一个批次发送。
        props.put(ProducerConfig.MAX_BLOCK_MS_CONFIG, 60000);//max.block.ms,生产者在获取元数据或缓存
空间时的最大阻塞时间，单位为毫秒。
        props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION,
5);//max.in.flight.requests.per.connection,每个连接上的最大未确认请求数。
        props.put(ProducerConfig.MAX_REQUEST_SIZE_CONFIG, 1048576);//max.request.size,请求的最大大
小，单位为字节
        props.put(ProducerConfig.METADATA_MAX_AGE_CONFIG, 300000);//metadata.max.age.ms元数据的最
大寿命，单位为毫秒。超过这个时间的元数据会被刷新。
        props.put(ProducerConfig.METRIC_REPORTER_CLASSES_CONFIG, "");//metric.reporters度量报告器
类列表。生产者会使用这些报告器来报告度量信息。
        props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG,
"org.apache.kafka.clients.producer.RoundRobinPartitioner");//partitioner.class分区器类。生产者会使
用这个分区器来决定每个消息发送到哪个分区。
        props.put(ProducerConfig.RECEIVE_BUFFER_CONFIG, 32768);//receive.buffer.bytes接收缓冲区的
大小，单位为字节。
        props.put(ProducerConfig.SEND_BUFFER_CONFIG, 131072);//send.buffer.bytes发送缓冲区的大小，
单位为字节。
        props.put(ProducerConfig.RECONNECT_BACKOFF_MAX_MS_CONFIG,
1000);//reconnect.backoff.max.ms重连最大间隔时间，单位为毫秒。
        props.put(ProducerConfig.RECONNECT_BACKOFF_MS_CONFIG, 50);//reconnect.backoff.ms重连间隔
时间，单位毫秒
        props.put(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG, 30000);//request.timeout.ms请求的超时
时间，单位为毫秒。
        props.put(ProducerConfig.RETRIES_CONFIG, 2147483647);//retries发送失败时的重试次数。
        props.put(ProducerConfig.RETRY_BACKOFF_MS_CONFIG, 100);//retry.backoff.ms重试的间隔时间，单
位为毫秒。
        props.put(ProducerConfig.TRANSACTION_TIMEOUT_CONFIG, 60000);//transaction.timeout.ms事务
的超时时间，单位为毫秒
        props.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, null);//transactional.id事务ID。如果设置
了这个参数，生产者会启用事务功能。
        props.put(ProducerConfig.CLIENT_DNS_LOOKUP_CONFIG, "default");//client.dns.lookupDNS查找
策略。可选值为default、use_all_dns_ips、resolve_canonical_bootstrap_servers_only。

        // 创建生产者
        KafkaProducer<String, String> producer = new KafkaProducer<>(props);

        // 发送消息
        for (int i = 0; i < 100; i++) {
            String key = "key-" + i;
            String value = "value-" + i;

            // 创建消息记录
            ProducerRecord<String, String> record = new ProducerRecord<>(TOPIC, key, value);

            // 发送消息
```

```
            producer.send(record, new Callback() {
                @Override
                public void onCompletion(RecordMetadata metadata, Exception exception) {
                    if (exception == null) {
                        System.out.println("消息发送成功: key=" + key + ", value=" + value);
                    } else {
                        System.err.println("消息发送失败: " + exception.getMessage());
                    }
                }
            });
        }

        // 关闭生产者
        producer.close();
    }
}
```

参数说明调优

关于 acks 参数优化

acks 参数用于控制生产者发送消息时的确认机制。该参数的默认值为1，表示消息发送给 Leader Broker 后，Leader 确认消息写入后即返回。acks参数还有以下可选值：

- 0: 不等待任何确认，直接返回。
- 1: 等待 Leader 副本确认写入后返回。
- -1或者 all: 等待 Leader 副本以及相关的 Follower 副本确认写入后返回。

由上可知，在跨可用区场景，以及副本数较多的 Topic，acks 参数的取值会影响消息的可靠性和吞吐量。因此：

- 在一些在线业务消息的场景下，吞吐量要求不大，可以将 acks 参数设置为-1，确保消息被所有副本接收和确认后才返回，从而提高消息的可靠性，但是会牺牲写入吞吐和性能，时延会增加。
- 在日志采集等大数据或者离线计算的场景下，要求高吞吐（即每秒写入 Kafka 的数据量）的情况下，可以将 acks 设置为1，提高吞吐量。

关于 Batch 相关参数优化

默认情况下，传输同等数据量的情况下，多次请求和一次请求的网络传输，一次请求传输能有效减少相关计算和网络资源，提高整体写入的吞吐量。因此，可以通过这个参数设置优化客户端发送消息的吞吐能力。在高吞吐场景下，可以配合计算和设置：

- batch.size：默认16K。
- linger.ms：默认为0，可以适当增加耗时，如设置100ms，尽可能聚合更多消息批量发送消息。
- buffer.memory：默认32MB，对于大流量 Producer，在堆内存充足情况可以设置更大，如设置256MB。

关于事务参数优化

```
props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, false);//enable.idempotence,是否启用幂等性。如果启用，生产者会确保每个消息只被发送一次，即使在网络错误或重试的情况下。                                  //是否需要幂等，在事务场景下需要设置为true
props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, 5);//max.in.flight.requests.per.connection,每个连接上的最大未确认请求数。在事务场景不要超过5
props.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, null);//transactional.id事务ID。如果设置了这个参数，生产者会启用事务功能。
props.put(ProducerConfig.TRANSACTION_TIMEOUT_CONFIG, 60000);//transaction.timeout.ms事务的超时时间，单位为毫秒,可以适当延长事务时间。
```

需要强调，事务因为要保障消息的 exactly once 语义，因此会额外付出更多的计算资源。
对于事务场景，可是适当增加事务超时时间，容忍高吞吐场景下，写入延时带来的抖动。

关于压缩参数优化

Kafka Java Client 支持如下压缩参数：

```
props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "none");//compression.type消息压缩类型。默认none
不压缩，可选值为none、gzip、snappy、lz4、zstd。
```

目前支持以下几种压缩配置：

- none：不使用压缩算法。
- gzip：使用 GZIP 压缩算法。
- snappy：使用 Snappy 压缩算法。
- lz4：使用 LZ4 压缩算法。
- zstd：使用 ZSTD 压缩算法。

要在 Kafka Java 客户端中使用压缩消息，需要在创建生产者时设置 compression.type 参数。例如，要使用 LZ4 压缩算法，可以将 compression.type 设置为lz4。
Kafka 压缩消息是一种用计算换带宽的优化方式，虽然 Kafka 压缩消息的压缩和解压缩，发生在客户端，但是由于Broker 针对压缩消息存在校验行为会付出额外的计算成本，尤其是 gzip 压缩，服务端对该压缩消息校验的计算成本会非常大，在某种程度上可能会出现得不偿失的情况，因为计算的增加导致 Broker CPU 利用率很高，降低了其他请求的处理能力，导致整体性能更低。这种情况建议可以使用如下方式规避 Broker 的校验：
1. 在 Producer 端对消息数据独立压缩，生成压缩包数据：messageCompression，同时在消息的 key 存储压缩方式：

```
{"Compression","lz4"}
```

2. 在 Producer 端将 messageCompression 当成正常消息发送。
3. 在 Consumer 段读取消息 key，获取使用的压缩方式，独立进行解压缩。

创建生产者实例

如果应用程序需要更高的吞吐量，可以使用异步发送，以提高消息的发送速度。同时，可以使用批量发送消息的方式，以减少网络开销和 IO 消耗。
如果应用程序需要更高的可靠性，可以使用同步发送，以确保消息发送成功。同时，可以使用 ACK 确认机制和事务机制，以确保消息的可靠性和一致性。具体的参数调优参考生产者参数与调优。

同步发送

在 Kafka Java Client 客户端中，同步发送的示例代码如下：

```
import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.serialization.StringSerializer;

import java.util.Properties;
import java.util.concurrent.ExecutionException;

public class KafkaProducerSyncExample {

    private static final String BOOTSTRAP_SERVERS = "localhost:9092";
    private static final String TOPIC = "test-topic";

    public static void main(String[] args) {
        // 创建Kafka生产者配置
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
```

```
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());

        // 设置生产者参数
        props.put(ProducerConfig.ACKS_CONFIG, "all");
        props.put(ProducerConfig.RETRIES_CONFIG, 3);

        // 创建生产者
        KafkaProducer<String, String> producer = new KafkaProducer<>(props);

        // 同步发送消息
        for (int i = 0; i < 10; i++) {
            String key = "sync-key-" + i;
            String value = "sync-value-" + i;

            // 创建消息记录
            ProducerRecord<String, String> record = new ProducerRecord<>(TOPIC, key, value);

            try {
                // 发送消息并等待结果
                RecordMetadata metadata = producer.send(record).get();
                System.out.println("同步发送成功: key=" + key + ", value=" + value);
            } catch (InterruptedException | ExecutionException e) {
                System.err.println("同步发送失败: " + e.getMessage());
            }
        }

        // 关闭生产者
        producer.close();
    }
}
```

异步发送

异步发送：异步发送消息时不会阻塞当前线程，生产者的吞吐量较高，但是需要通过回调函数来处理消息结果，示例如下：

```
import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.serialization.StringSerializer;

import java.util.Properties;

public class KafkaProducerAsyncExample {

    private static final String BOOTSTRAP_SERVERS = "localhost:9092";
    private static final String TOPIC = "test-topic";

    public static void main(String[] args) {
        // 创建Kafka生产者配置
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
```

```
        // 设置生产者参数
        props.put(ProducerConfig.ACKS_CONFIG, "all");
        props.put(ProducerConfig.RETRIES_CONFIG, 3);

        // 创建生产者
        KafkaProducer<String, String> producer = new KafkaProducer<>(props);

        // 异步发送消息
        for (int i = 0; i < 10; i++) {
            String key = "async-key-" + i;
            String value = "async-value-" + i;

            // 创建消息记录
            ProducerRecord<String, String> record = new ProducerRecord<>(TOPIC, key, value);

            // 发送消息并设置回调函数
            producer.send(record, new Callback() {
                @Override
                public void onCompletion(RecordMetadata metadata, Exception exception) {
                    if (exception == null) {
                        System.out.println("异步发送成功: key=" + key + ", value=" + value);
                    } else {
                        System.err.println("异步发送失败: " + exception.getMessage());
                    }
                }
            });
        }

        // 关闭生产者
        producer.close();
    }
}
```

消费者实践

消费者参数

```
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;

import java.time.Duration;
import java.util.Collections;
import java.util.Properties;

public class KafkaConsumerDemo {

    public static void main(String[] args) {
        Properties properties = new Properties();
```

```
        properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092"); //
"bootstrap.servers"，Kafka集群的地址，没有默认值
        properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName()); // "key.deserializer"，消息键的反序列化方式，没有默认值
        properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName()); // "value.deserializer"，消息值的反序列化方式，没有默认值
        properties.put(ConsumerConfig.GROUP_ID_CONFIG, "test-group"); // "group.id"，消费者组ID，
没有默认值
        properties.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "latest"); //
"auto.offset.reset"，位点不存在时的处理方式，"latest"表示从最新的消息开始消费，默认值为"latest"
        properties.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true"); //
"enable.auto.commit"，是否自动提交位点，默认值为"true"
        properties.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "5000"); //
"auto.commit.interval.ms"，自动提交位点的间隔时间，单位为毫秒，默认值为"5000"
        properties.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, "10000"); //
"session.timeout.ms"，消费者组成员的会话超时时间，单位为毫秒，默认值为"10000"
        properties.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, "500"); // "max.poll.records"，单
次poll的最大消息数，默认值为"500"
        properties.put(ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG, "300000"); //
"max.poll.interval.ms"，两次poll操作间的最大允许间隔时间，单位为毫秒，默认值为"300000"
        properties.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, "1"); // "fetch.min.bytes"，服务器
返回的最小数据，如果设置大于1，服务器会等待直到累计的数据量大于这个值，默认值为"1"
        properties.put(ConsumerConfig.FETCH_MAX_BYTES_CONFIG, "52428800"); //
"fetch.max.bytes"，服务器返回的最大数据量，单位为字节，默认值为"52428800"
        properties.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, "500"); // "fetch.max.wait.ms",
服务器等待满足fetch.min.bytes条件的最大时间，单位为毫秒，默认值为"500"
        properties.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, "3000"); //
"heartbeat.interval.ms"，心跳间隔时间，单位为毫秒，默认值为"3000"
        properties.put(ConsumerConfig.CLIENT_ID_CONFIG, "my-client-id"); // "client.id"，客户端
ID，没有默认值
        properties.put(ConsumerConfig.REQUEST_TIMEOUT_MS_CONFIG, "30000"); //
"request.timeout.ms"，客户端请求超时时间，单位为毫秒，默认值为"30000"

        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(properties);
        consumer.subscribe(Collections.singletonList("test-topic"));

        try {
            while (true) {
                ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
                for (ConsumerRecord<String, String> record : records) {
                    System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(),
record.key(), record.value());
                }
            }
        } finally {
            consumer.close();
        }
    }
}
```

参数调优

1. 在使用 Kafka 消费者时，我们可以通过调整一些参数来优化性能。以下是一些常见的参数调优方案：
    - fetch.min.bytes：如果不确定消息最低大小，这个参数建议设置为1，如果明确消息最小值，可以设置该值为最小消息大小。

- max.poll.records：这个参数可以根据应用的处理能力进行调整。如果您的应用可以处理更多的记录，可以将这个参数设置为更大的值，以减少 poll操作的次数。
- auto.commit.interval.ms：这个参数可以根据您的应用的需求进行调整，一般自动提交位点场景，建议保持默认值5000ms。注意，过于频繁的位点提交会影响性能，额外占用 Broker 的计算资源。
- client.id：可以为每个消费者设置一个唯一的 ID，以便在监控和日志中区分不同的消费者。

以上是一些常见的参数调优方案，但具体的最佳设置可能会根据您的应用的特性和需求有所不同。在调优参数时，请记住始终进行性能测试，以确保您的设置可以达到预期的效果。

2. 对于 rebalance 时间频繁和消费线程阻塞问题，参考以下说明参数优化：

- session.timeout.ms：v0.10.2之前的版本可适当提高该参数值，需要大于消费一批数据的时间，但不要超过30s，建议设置为25s；而v0.10.2及其之后的版本，保持默认值10s即可。
- max.poll.records：降低该参数值，建议远远小于<单个线程每秒消费的条数> * <消费线程的个数> *<max.poll.interval.ms>的积。
- max.poll.interval.ms：该值要大于<max.poll.records> / (<单个线程每秒消费的条数> * <消费线程的个数>)的值。

创建消费者实例

Kafka Java Client 提供订阅的模型创建消费者，其中在提交位点方面，提供手动提交位点和自动提交位点两种方式。

自动提交位点

自动提交位点：消费者在拉取消息后会自动提交位点，无需手动操作。这种方式的优点是简单易用，但是可能会导致消息重复消费或丢失。注意，自动提交位点时间间隔 auto.commit.interval.ms 不要设置太短，否则容易导致 Broker CPU 偏高，影响其他请求处理。

```
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.serialization.StringDeserializer;

import java.time.Duration;
import java.util.Collections;
import java.util.Properties;

public class KafkaConsumerAutoCommitExample {

    private static final String BOOTSTRAP_SERVERS = "localhost:9092";
    private static final String TOPIC = "test-topic";
    private static final String GROUP_ID = "test-group";

    public static void main(String[] args) {
        // 创建Kafka消费者配置
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        props.put(ConsumerConfig.GROUP_ID_CONFIG, GROUP_ID);
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        // 开启自动提交位点
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, true);
        props.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, 5000); // 自动提交间隔，单位：5000
毫秒

        // 创建消费者
        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
```

```
        // 订阅主题
        consumer.subscribe(Collections.singletonList(TOPIC));

        // 消费消息
        try {
            while (true) {
                ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
                for (ConsumerRecord<String, String> record : records) {
                    System.out.printf("消费消息：topic=%s, partition=%d, offset=%d, key=%s,
value=%s%n",
                            record.topic(), record.partition(), record.offset(), record.key(),
record.value());
                }
            }
        } finally {
            // 关闭消费者
            consumer.close();
        }
    }
}
```

手动提交位点

手动提交位点：消费者在处理完消息后需要手动提交位点。这种方式的优点是可以精确控制位点的提交，避免消息重复消费或丢失。但是需要注意，手动提交位点如果太频繁会导致 Broker CPU 很高，影响性能，随着消息量增加，CPU 消费会很高，影响正常 Broker 的其他功能，因此建议间隔一定消息提交位点。

```
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;

import java.time.Duration;
import java.util.Collections;
import java.util.Properties;

public class KafkaConsumerManualCommitExample {

    public static void main(String[] args) {
        Properties properties = new Properties();
        properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        properties.put(ConsumerConfig.GROUP_ID_CONFIG, "test-group");
        properties.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        properties.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "false");

        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(properties);
```

```
        consumer.subscribe(Collections.singletonList("test-topic"));

        int count = 0;
        try {
            while (true) {
                ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
                for (ConsumerRecord<String, String> record : records) {
                    System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(),
record.key(), record.value());
                    count++;
                    if (count % 10 == 0) {
                        consumer.commitSync();
                        System.out.println("Committed offsets.");
                    }
                }
            }
        } finally {
            consumer.close();
        }
    }
}
```

Assign 消费

Kafka Java Client 的 assign 消费模式允许消费者直接指定订阅的分区，而不是通过订阅主题来自动分配分区。这种模式适用于需要手动控制消费分区的场景，例如：为了实现特定的负载均衡策略，或者在某些情况下跳过某些分区。一般流程为使用 assign 方法来手动指定消费者消费的分区，通过seek 方法来设置开始消费的位点，然后执行消费逻辑，使用示例如下：

```
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.TopicPartition;

import java.time.Duration;
import java.util.Arrays;
import java.util.Properties;

public class KafkaConsumerAssignAndSeekApp {
    public static void main(String[] args) {
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
        props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
        props.put("enable.auto.commit", "false");

        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

        String topic = "my-topic";
        TopicPartition partition0 = new TopicPartition(topic, 0);
        TopicPartition partition1 = new TopicPartition(topic, 1);
        consumer.assign(Arrays.asList(partition0, partition1));
```

```
        // 设置消费的起始位点
        long startPosition0 = 10L;
        long startPosition1 = 20L;
        consumer.seek(partition0, startPosition0);
        consumer.seek(partition1, startPosition1);

        try {
            while (true) {
                ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
                for (ConsumerRecord<String, String> record : records) {
                    System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(),
record.key(), record.value());
                }
                consumer.commitSync(); // 手动提交位点
            }
        } finally {
            consumer.close();
        }
    }
}
```

Kafka Java Client 生产消费常见问题

Kafka Java Producer 无法成功发送消息

- 首先排查 Kafka 集群的 IP 和端口能够正常连接，若不能请先解决通信问题。
- 其次检查是否正确配置接入点，版本是否和 Broker 版本匹配，可以参考实践教程的发送 demo。

Kafka Python SDK

最近更新时间：2024-10-14 18:00:01

背景

CKafka 的 Python 客户端有以下几个主要的库：

- kafka-python：这是一个纯 Python 实现的 Kafka 客户端，支持 Kafka 0.8.2及更高版本。它提供了生产者、消费者和管理 Kafka 集群的 API。这个库易于使用，但性能可能不如基于 librdkafka 的客户端。

  安装方法：`pip install kafka-python`
- confluent-kafka-python：这个库是基于高性能的 C 库 librdkafka 实现的。它支持Kafka 0.9及更高版本，并提供了生产者、消费者和管理 Kafka 集群的 API。这个库性能更好，但可能需要安装额外的依赖。

  安装方法：`pip install confluent-kafka`
- aiokafka：这是一个基于 kafka-python 的异步 Kafka 客户端，使用 asyncio 库。这个库适用于需要异步编程的场景。

  安装方法：`pip install aiokafka`
- pykafka：这是一个支持 Kafka 0.8.x 版本的 Python 客户端。它提供了生产者、消费者和管理 Kafka 集群的 API。这个库已经不再积极维护，但仍然适用于需要支持较旧版本的 Kafka 的场景。

  安装方法：`pip install pykafka`

在选择 Python Kafka 客户端时，请根据您的应用需求和 Kafka 版本选择合适的库。对于大多数场景，推荐使用 kafka-python 或 confluent-kafka-python，因为它们支持较新的 Kafka 版本并且功能更完善。如果您的应用需要异步编程，可以考虑使用 aiokafka。
本文重点介绍 kafka-python 的使用方式，官网文档参见 kafka-python 。

生产者实践

版本选择

在使用 kafka-python 时，需要先安装 kafka-python 库。可以使用以下命令进行安装：
`pip install kafka-python`

生产者参数与调优

生产者参数

Kafka Python 涉及如下关键参数，相关的参数和默认值如下：

```
from kafka import KafkaProducer

producer = KafkaProducer(
    bootstrap_servers='localhost:9092', # 用于初始化连接到Kafka集群的broker列表，默认值为'localhost:9092'
    client_id=None, # 自定义客户端ID，用于在Kafka服务端日志中识别客户端，默认值为None
    key_serializer=None, # 用于将消息键序列化为字节的可调用对象，默认值为None
    value_serializer=None, # 用于将消息值序列化为字节的可调用对象，默认值为None
    compression_type=None, # 消息压缩类型，可选值为'gzip', 'snappy', 'lz4'或None，表示不压缩，默认值为None
    retries=0, # 重新发送失败的消息的次数，默认值为0
    batch_size=16384, # 用于批处理消息的大小，单位为字节，默认值为16384
    linger_ms=0, # 在批处理消息之前等待更多消息的最长时间，单位为毫秒，默认值为0
    partitioner=None, # 用于确定消息分区的可调用对象，默认值为None
    buffer_memory=33554432, # 用于缓冲待发送消息的内存总量，单位为字节，默认值为33554432
    connections_max_idle_ms=540000, # 空闲连接的最长保持时间，单位为毫秒，默认值为540000
    max_block_ms=60000, # 在达到缓冲区内存限制时，send()方法阻塞的最长时间，单位为毫秒，默认值为60000
    max_request_size=1048576, # 发送到broker的请求的最大字节数，默认值为1048576
    metadata_max_age_ms=300000, # 元数据在本地缓存中的最长存活时间，单位为毫秒，默认值为300000
```

```
    retry_backoff_ms=100, # 两次重试之间的等待时间，单位为毫秒，默认值为100
    request_timeout_ms=30000, # 客户端等待请求响应的最长时间，单位为毫秒，默认值为30000
    receive_buffer_bytes=32768, # 用于接收数据的网络缓冲区大小，单位为字节，默认值为32768
    send_buffer_bytes=131072, # 用于发送数据的网络缓冲区大小，单位为字节，默认值为131072
    acks='all', # 消息确认机制，可选值为'0', '1', 或'all'，默认值为'all'
    transactional_id=None, # 事务ID，用于标识生产者参与事务的唯一标识，默认值为None
    transaction_timeout_ms=60000, # 事务超时时间，单位为毫秒，默认值为60000
    enable_idempotence=False, # 是否启用幂等性，默认值为False
    security_protocol='PLAINTEXT', # 安全协议类型，可选值为'PLAINTEXT', 'SSL', 'SASL_PLAINTEXT',
'SASL_SSL'，默认值为'PLAINTEXT'
```

参数说明调优

关于 acks 参数优化

acks 参数用于控制生产者发送消息时的确认机制。该参数的默认值为-1，表示消息发送给 Leader Broker 后，Leader 确认以及相应的 Follower 消息都写入完成后才返回。acks 参数还有以下可选值：0，1，-1。在跨可用区场景，以及副本数较多的 Topic，acks 参数的取值会影响消息的可靠性和吞吐量。因此：

- 在一些在线业务消息的场景下，吞吐量要求不大，可以将 acks参 数设置为-1，则可以确保消息被所有副本接收和确认后才返回，从而提高消息的可靠性。
- 在日志采集等大数据或者离线计算的场景下，要求高吞吐（即每秒写入 Kafka 的数据量）的情况下，可以将 acks 设置为1，提高吞吐量。

关于 buffer_memory 参数优化（缓存）

默认情况下，传输同等数据量的情况下，多次请求和一次请求的网络传输，一次请求传输能有效减少相关计算和网络资源，提高整体写入的吞吐量。因此，可以通过这个参数设置优化客户端发送消息的吞吐能力。对于 Kafka Python Client，默认提供 linger_ms 为0ms 的攒批时间积攒消息，此处可以优化，适当增加值，例如设置100ms，进行聚合多个请求批量发送消息，提高吞吐。如果带宽较高，且单机内存充足，建议调大 buffer_memory 提高吞吐。

关于压缩参数优化

Kafka Python Client 支持如下压缩参数：none, gzip, snappy, lz4。

- none：不使用压缩。
- gzip：使用 GZIP 压缩。
- snappy：使用 Snappy 压缩。
- lz4：使用 LZ4 压缩。

要在 Producer 客户端中使用压缩消息，需要在创建生产者时设置 compression_type 参数。例如，要使用 LZ4 压缩算法，可以将 compression_type 设置为 lz4，虽然压缩消息的压缩和解压缩，发生客户端，是一种用计算换带宽的优化方式，但是由于 Broker 针对压缩消息存在校验行为会付出额外的计算成本，在低流量情况，不建议使用压缩，尤其是 gzip 压缩，Broker 校验计算成本会比较大，在某种程度上可能会出现得不偿失的情况，反而因为计算的增加导致 Broker 处理其他请求的能力偏低，导致带宽吞吐更低。这种情况建议可以使用如下方式：
在 Producer 端对消息数据独立压缩，生成压缩包数据：messageCompression，同时在消息的 key 存储压缩方式：
`{"Compression","lz4"}`

- 在 Producer 端将 messageCompression 当成正常消息发送。
- 在 Consumer 端读取消息 key，获取使用的压缩方式，独立进行解压缩。

创建生产者实例

如果应用程序需要更高的可靠性，则可以使用同步生产者，以确保消息发送成功。同时，可以使用 ACK 确认机制和事务机制，以确保消息的可靠性和一致性。具体的参数调优参考生产者参数与调优。如果应用程序需要更高的吞吐量，则可以使用异步生产者，以提高消息的发送速度。同时，可以使用批量发送消息的方式，以减少网络开销和 IO 消耗。示例如下：

```
from kafka import KafkaProducer
import sys
```

```
# 参数配置
BOOTSTRAP_SERVERS = 'localhost:9092'
TOPIC = 'test_topic'
SYNC = True
ACKS = '1'  # leader副本确认写入即可
LINGER_MS = 500  # 延迟500ms发送
BATCH_SIZE = 16384  # 消息批次大小16KB

def create_producer(servers, acks, linger_ms, batch_size):
    return KafkaProducer(bootstrap_servers=servers, acks=acks, linger_ms=linger_ms,
batch_size=batch_size)

def send_message_sync(producer, topic, message):
    future = producer.send(topic, message)
    result = future.get(timeout=10)
    print(f"Sent message: {message} to topic: {topic}, partition: {result.partition}, offset:
{result.offset}")

def send_message_async(producer, topic, message):
    def on_send_success(record_metadata):
        print(f"Sent message: {message} to topic: {topic}, partition:
{record_metadata.partition}, offset: {record_metadata.offset}")

    def on_send_error(excp):
        print(f"Error sending message: {message} to topic: {topic}", file=sys.stderr)
        print(excp, file=sys.stderr)

    future = producer.send(topic, message)
    future.add_callback(on_send_success).add_errback(on_send_error)

def main():
    producer = create_producer(BOOTSTRAP_SERVERS, ACKS, LINGER_MS, BATCH_SIZE)
    messages = ['Hello Kafka', 'Async vs Sync', 'Demo']

    if SYNC:
        for message in messages:
            send_message_sync(producer, TOPIC, message.encode('utf-8'))
    else:
        for message in messages:
            send_message_async(producer, TOPIC, message.encode('utf-8'))

    producer.flush()

if __name__ == '__main__':
    main()
```

消费者实践

消费者参数与调优

消费者参数

```
from kafka import KafkaConsumer
```

```
# 创建一个KafkaConsumer对象，用于连接Kafka集群并消费消息
consumer = KafkaConsumer(
    'topic_name', # 要订阅的主题列表
    bootstrap_servers=['localhost:9092'],     # Kafka集群的接入点
    group_id=None,     # 消费者组ID，用于将消费者分组，要加入动态分区分配（如果启用）并用于获取和提交偏移量的消费者组的名称。如果为 None，则禁用自动分区分配（通过组协调器）和偏移量提交。
    client_id='kafka-python-{version}',#客户端Id,默认是kafka-python-{version}
    api_version=None, #指定要使用的 Kafka API 版本。如果设置为 None，客户端将尝试通过API请求来启动不同版本的功能
    enable_auto_commit=True,     # 是否自动提交消费位置，默认为True
    auto_commit_interval_ms=5000,     # 自动提交消费位置的间隔，默认为5秒（5000毫秒）
    auto_offset_reset='latest',     # 消费者在读取的分区中的消费位置的策略，默认为'latest'（从最新的位置开始消费）
    fetch_min_bytes=1,     # 消费者在读取分区时的最小字节数，默认为1字节
    fetch_max_wait_ms=500,     # 在没有新的消费数据，默认等待500ms
    fetch_max_bytes=52428800,     # 消费者在读取分区时的最大字节数，默认为52428800字节（50MB）
    max_poll_interval_ms=300000     # 消参数默认值为300000毫秒（5分钟）。如果消费者在5分钟内没有发送心跳信号，它将被认为已经失去连接，并将被从消费者组中移除。在这种情况下，其他消费者将接管被移除的消费者的分区并发起重平衡。
    retry_backoff_ms=100,     # 重试间隔时间，默认为100毫秒
    reconnect_backoff_max_ms=1000,     # 重新连接到多次连接失败的Broker的间隔时间最大值（以毫秒为单位）。如果连接失败，将在每次连续连接失败时呈指数增加，直至达到此最大值。一旦达到最大值，重新连接尝试将以该固定速率定期继续。
    request_timeout_ms=305000,     # 客户端请求超时（以毫秒为单位）
    session_timeout_ms=10000,     # session_timeout_ms (int) - 使用 Kafka 组管理工具时用于检测故障的超时时间。消费者定期发送心跳给Broker表明其活跃度。如果在此会话超时到期之前Broker没有收到心跳，则Broker将改消费组从组中删除该消费者并启动重新平衡。
    heartbeat_interval_ms=3000,     # 使用 Kafka 的组管理工具时，向消费者协调器发出心跳之间的预期时间（以毫秒为单位）。心跳用于确保消费者的会话保持活动状态，并在新消费者加入或离开组时促进重新平衡。该值必须设置为低于 session_timeout_ms，但通常不应高于该值的 1/3。
    receive_buffer_bytes=32768,#读取数据时使用的 TCP 接收缓冲区 (SO_RCVBUF) 的大小。默认值：无（依赖于系统默认值），默认为32768。
    send_buffer_bytes=131072# 发送数据时使用的 TCP 发送缓冲区 (SO_SNDBUF) 的大小。默认值：无（依赖于系统默认值），131072。
)

for message in consumer:
    print(f"Topic: {message.topic}, Partition: {message.partition}, Offset: {message.offset}, Value: {message.value}")
```

参数说明与调优

1. max_poll_interval_ms 是 Kafka Python Consumer 的一个配置参数，它用于指定 Consumer 在两次 poll 操作之间的最大延迟。这个参数的主要作用是控制 Consumer 的 存活，也就是判断 Consumer 是否还活着。如果 Consumer 在 max_poll_interval_ms 指定的时间内没有进行 poll 操作，那么 Kafka 认为这个 Consumer 已经挂掉，会触发 Consumer 的 rebalance 操作。这个参数的设置需要根据实际的消费速度来调整。如果设置得太小，可能会导致 Consumer 频繁地触发 rebalance 操作，增加了 Kafka 的负担；如果设置得太大，可能会导致 Consumer 在出现问题时不能及时被 Kafka 检测到，从而影响了消息的消费。建议在高吞吐下可以时长增加该值的设置。
2. 针对自动提交位点请求，建议 auto_commit_interval_ms 时间不要低于1000ms，因为频率过高的位点请求会导致 Broker CPU 很高，影响其他正常服务的读写。

创建消费者实例

Kafka Python 提供订阅的模型创建消费者，其中在提交位点方面，提供手动提交位点和自动提交位点两种方式。

自动提交位点

自动提交位点：消费者在拉取消息后会自动提交位点，无需手动操作。这种方式的优点是简单易用，但是可能会导致消息重复消费或丢失。建议间隔5s提交位点。

```
# auto_commit_consumer_interval.py
from kafka import KafkaConsumer
from time import sleep

consumer = KafkaConsumer(
    'your_topic_name',
    bootstrap_servers=['localhost:9092'],
    group_id='auto_commit_group',
    auto_commit_interval_ms=5000  # 设置自动提交位点的间隔为5000毫秒（5秒）
)

for message in consumer:
    print(f"Topic: {message.topic}, Partition: {message.partition}, Offset: {message.offset},
Value: {message.value}")
    sleep(1)
```

手动提交位点

手动提交位点：消费者在处理完消息后需要手动提交位点。这种方式的优点是可以精确控制位点的提交，避免消息重复消费或丢失。但是需要注意，手动提交位点如果太频繁会导致 Broker CPU 很高，影响性能，随着消息量增加，CPU 消费会很高，影响正常 Broker 的其他功能，因此建议间隔一定消息提交位点。

```
# manual_commit_consumer.py
from kafka import KafkaConsumer
from kafka.errors import KafkaError
from time import sleep

consumer = KafkaConsumer(
    'your_topic_name',
    bootstrap_servers=['localhost:9092'],
    group_id='manual_commit_group',
    enable_auto_commit=False
)

count = 0
for message in consumer:
    print(f"Topic: {message.topic}, Partition: {message.partition}, Offset: {message.offset},
Value: {message.value}")
    count += 1

    if count % 10 == 0:
        try:
            consumer.commit()
        except KafkaError as e:
            print(f"Error while committing offset: {e}")

    sleep(1)
```

librdkafka SDK

最近更新时间：2024-10-14 15:47:12

背景

TDMQ CKafka 是一个分布式流处理平台，用于构建实时数据管道和流式应用程序。它提供了高吞吐量、低延迟、可伸缩性和容错性等特性。本文着重介绍上述 librdkafka 客户端的关键参数和最佳实践，以及常见问题。

生产者实践

版本选择

在使用 librdkafka 时，librdkafka 会自动根据 Kafka 集群的版本选择适当的协议版本进行通信，由于 kafka 的版本迭代更新较快，通常情况下，使用最新的 librdkafka 版本可以获得最佳的兼容性和性能。

生产者参数与调优

生产者参数

librdkafka 主要涉及如下关键参数，相关的参数和默认值如下：

```
rd_kafka_conf_t *conf = rd_kafka_conf_new();

// Kafka集群的地址，多个地址用逗号分隔，默认为空
rd_kafka_conf_set(conf, "bootstrap.servers", "localhost:9092", NULL, 0);

// 发送消息的最大尝试次数，包括第一次尝试，默认为2
rd_kafka_conf_set(conf, "message.send.max.retries", "2", NULL, 0);

// 重试之间的回退时间（以毫秒为单位），默认为100
rd_kafka_conf_set(conf, "retry.backoff.ms", "100", NULL, 0);

// 客户端请求超时时间（以毫秒为单位），默认为5000
rd_kafka_conf_set(conf, "request.timeout.ms", "5000", NULL, 0);

// 客户端发送缓冲区大小（以字节为单位），默认为131072
rd_kafka_conf_set(conf, "queue.buffering.max.kbytes", "131072", NULL, 0);

// 客户端发送缓冲区中消息的最大数量，默认为100000
rd_kafka_conf_set(conf, "queue.buffering.max.messages", "100000", NULL, 0);

// 客户端发送缓冲区中消息的最大总大小（以字节为单位），默认为1000000
rd_kafka_conf_set(conf, "queue.buffering.max.total.bytes", "1000000", NULL, 0);

// 客户端发送缓冲区的linger时间（以毫秒为单位），默认为0
rd_kafka_conf_set(conf, "queue.buffering.max.ms", "0", NULL, 0);

// 是否启用消息压缩，默认为0（不启用）
rd_kafka_conf_set(conf, "compression.codec", "none", NULL, 0);

// 消息压缩级别，默认为0（自动选择）
rd_kafka_conf_set(conf, "compression.level", "0", NULL, 0);

// 客户端的ID，默认为rdkafka
```

```
rd_kafka_conf_set(conf, "client.id", "rdkafka", NULL, 0);

// 生产者的最大并发请求数，即未收到broker响应的请求数，默认为1000000
rd_kafka_conf_set(conf, "max.in.flight.requests.per.connection", "1000000", NULL, 0);

// 客户端与Kafka集群的连接最大重试次数，默认为3次
rd_kafka_conf_set(conf, "broker.address.ttl", "3", NULL, 0);

// 客户端与Kafka集群的连接重试间隔（以毫秒为单位），默认为1000
rd_kafka_conf_set(conf, "reconnect.backoff.ms", "1000", NULL, 0);

// 客户端与Kafka集群的连接重试最大间隔（以毫秒为单位），默认为10000
rd_kafka_conf_set(conf, "reconnect.backoff.max.ms", "10000", NULL, 0);

// 客户端API版本的回退时间（以毫秒为单位），默认为10000
rd_kafka_conf_set(conf, "api.version.request.timeout.ms", "10000", NULL, 0);

// 安全协议，默认为plaintext
rd_kafka_conf_set(conf, "security.protocol", "plaintext", NULL, 0);

// 其他SSL和SASL相关参数，请参考librdkafka官方文档

// 创建生产者实例
rd_kafka_t *producer = rd_kafka_new(RD_KAFKA_PRODUCER, conf, NULL, 0);
```

参数说明调优

关于 acks 参数优化

acks 参数用于控制生产者发送消息时的确认机制。该参数的默认值为-1，表示消息发送给 Leader Broker 后，Leader 确认以及相应的 Follower 消息都写入完成后才返回。acks 参数还有以下可选值：0，1，-1。在跨可用区场景，以及副本数较多的 Topic，acks 参数的取值会影响消息的可靠性和吞吐量。因此：

- 在一些在线业务消息的场景下，吞吐量要求不大，可以将 acks 参数设置为-1，则可以确保消息被所有副本接收和确认后才返回，从而提高消息的可靠性。
- 在日志采集等大数据或者离线计算的场景下，要求高吞吐（即每秒写入 Kafka 的数据量）的情况下，可以将 acks 设置为1，提高吞吐。

关于 buffering 参数优化（缓存）

默认情况下，传输同等数据量的情况下，多次请求和一次请求的网络传输，一次请求传输能有效减少相关计算和网络资源，提高整体写入的吞吐量。因此，可以通过这个参数设置优化客户端发送消息的吞吐能力。对 librdkafka，默认提供5ms的攒批时间积攒消息。如果消息较小，可以适当增加 queue.buffering.max.ms 的时间。

关于压缩参数优化

librdkafka 支持如下压缩参数：none, gzip, snappy, lz4, zstd。
在 librdkafka客户端中，支持以下几种压缩算法：

- none：不使用压缩算法。
- gzip：使用 GZIP 压缩算法。
- snappy：使用 Snappy 压缩算法。
- lz4：使用 LZ4 压缩算法。
- zstd：使用 ZSTD 压缩算法。

要在 Producer 客户端中使用压缩算法，需要在创建生产者时设置 compression.type 参数。例如，要使用 LZ4 压缩算法，可以将 compression.type 设置为 lz4，虽然压缩算法的 CPU 压缩和 CPU解压缩，发生客户端，是一种用计算换带宽的优化方式，但是由于 Broker

针对压缩消息存在校验行为会付出额外的计算成本，尤其是 Gzip 压缩，服务端的压缩计算成本会比较大，在某种程度上可能会出现得不偿失的情况，反而因为计算的增加导致 Broker 消息处理能力偏低，导致带宽吞吐更低。这种情况建议可以使用如下方式进行使用：
在 Producer 端对消息数据独立压缩，生成压缩包数据：messageCompression，同时在消息的 key 存储压缩方式：
`{"Compression","CompressionLZ4"}`

- 在 Producer 端将 messageCompression 当成正常消息发送。
- 在 Consumer 端读取消息 key，获取使用的压缩方式，独立进行解压缩。

创建生产者实例

如果应用程序需要更高的吞吐量，则可以使用异步生产者，以提高消息的发送速度。同时，可以使用批量发送消息的方式，以减少网络开销和 IO 消耗。如果应用程序需要更高的可靠性，则可以使用同步生产者，以确保消息发送成功。同时，可以使用 ACK 确认机制和事务机制，以确保消息的可靠性和一致性。具体的参数调优参考生产者参数与调优。

```
#include <stdio.h>
#include <string.h>
#include <librdkafka/rdkafka.h>

// 生产者消息发送回调
void dr_msg_cb(rd_kafka_t *rk, const rd_kafka_message_t *rkmessage, void *opaque) {
    if (rkmessage->err) {
        fprintf(stderr, "Message delivery failed: %s\n", rd_kafka_err2str(rkmessage->err));
    } else {
        fprintf(stderr, "Message delivered (%zd bytes, partition %"PRId32")\n",
                rkmessage->len, rkmessage->partition);
    }
}

int main() {
    rd_kafka_conf_t *conf = rd_kafka_conf_new();

    // 设置Kafka集群的地址
    rd_kafka_conf_set(conf, "bootstrap.servers", "localhost:9092", NULL, 0);

    // 设置ack等于1，表示leader副本收到消息后即认为发送成功
    rd_kafka_conf_set(conf, "acks", "1", NULL, 0);

    // 设置生产者消息发送回调
    rd_kafka_conf_set_dr_msg_cb(conf, dr_msg_cb);

    // 创建生产者实例
    char errstr[512];
    rd_kafka_t *producer = rd_kafka_new(RD_KAFKA_PRODUCER, conf, errstr, sizeof(errstr));
    if (!producer) {
        fprintf(stderr, "Failed to create producer: %s\n", errstr);
        return 1;
    }

    // 创建主题实例
    rd_kafka_topic_t *topic = rd_kafka_topic_new(producer, "test", NULL);
    if (!topic) {
        fprintf(stderr, "Failed to create topic: %s\n",
rd_kafka_err2str(rd_kafka_last_error()));
        rd_kafka_destroy(producer);
        return 1;
    }
```

```
    // 发送消息
    const char *message = "Hello, Kafka!";
    if (rd_kafka_produce(
            topic,
            RD_KAFKA_PARTITION_UA,
            RD_KAFKA_MSG_F_COPY,
            (void *)message,
            strlen(message),
            NULL,
            0,
            NULL) == -1) {
        fprintf(stderr, "Failed to produce to topic %s: %s\n", rd_kafka_topic_name(topic),
rd_kafka_err2str(rd_kafka_last_error()));
    }

    // 等待所有消息发送完成
    while (rd_kafka_outq_len(producer) > 0) {
        rd_kafka_poll(producer, 1000);
    }

    // 销毁主题实例
    rd_kafka_topic_destroy(topic);

    // 销毁生产者实例
    rd_kafka_destroy(producer);

    return 0;
}
```

消费者实践

消费者参数与调优

消费者参数

```
    rd_kafka_conf_t *conf = rd_kafka_conf_new();

    // 设置Kafka集群的地址
    rd_kafka_conf_set(conf, "bootstrap.servers", "localhost:9092", NULL, 0);

    // 设置消费组ID，默认为空
    rd_kafka_conf_set(conf, "group.id", "mygroup", NULL, 0);

    // 设置消费者的自动提交间隔（以毫秒为单位），默认为5000
    rd_kafka_conf_set(conf, "auto.commit.interval.ms", "5000", NULL, 0);

    // 设置消费者的自动提交开关，默认为true
    rd_kafka_conf_set(conf, "enable.auto.commit", "true", NULL, 0);

    // 设置消费者的自动偏移量重置策略，默认为latest
    rd_kafka_conf_set(conf, "auto.offset.reset", "latest", NULL, 0);

    // 设置客户端的ID，默认为rdkafka
    rd_kafka_conf_set(conf, "client.id", "rdkafka", NULL, 0);
```

```
    // 创建消费者实例
    char errstr[512];
    rd_kafka_t *consumer = rd_kafka_new(RD_KAFKA_CONSUMER, conf, errstr, sizeof(errstr));
```

参数说明与调优

针对自动提交位点请求，建议 auto.commit.interval.ms 时间不要低于1000ms，因为频率过高的位点请求会导致 Broker CPU 很高，影响其他正常服务的读写。

创建消费者实例

提供订阅的模型创建消费者，其中在提交位点方面，提供手动提交位点和自动提交位点两种方式。

自动提交位点

自动提交位点：消费者在拉取消息后会自动提交位点，无需手动操作。这种方式的优点是简单易用，但是可能会导致消息重复消费或丢失。

```
#include <stdio.h>
#include <string.h>
#include <librdkafka/rdkafka.h>

// 消费者消息处理回调
void msg_consume(rd_kafka_message_t *rkmessage, void *opaque) {
    if (rkmessage->err) {
        fprintf(stderr, "%% Consume error for topic \"%s\" [%"PRId32"] "
                "offset %"PRId64": %s\n",
                rd_kafka_topic_name(rkmessage->rkt),
                rkmessage->partition, rkmessage->offset,
                rd_kafka_message_errstr(rkmessage));
    } else {
        printf("%% Message received on topic %s [%"PRId32"] at offset %"PRId64": %.*s\n",
               rd_kafka_topic_name(rkmessage->rkt),
               rkmessage->partition,
               rkmessage->offset, (int)rkmessage->len, (const char *)rkmessage->payload);
    }
}

int main() {
    rd_kafka_conf_t *conf = rd_kafka_conf_new();

    // 设置Kafka集群的地址
    rd_kafka_conf_set(conf, "bootstrap.servers", "localhost:9092", NULL, 0);

    // 设置消费组ID
    rd_kafka_conf_set(conf, "group.id", "mygroup", NULL, 0);

    // 设置消费者的自动提交开关，默认为true
    rd_kafka_conf_set(conf, "enable.auto.commit", "true", NULL, 0);

    // 设置消费者的自动提交间隔（以毫秒为单位），默认为5000
    rd_kafka_conf_set(conf, "auto.commit.interval.ms", "5000", NULL, 0);

    // 创建消费者实例
    char errstr[512];
    rd_kafka_t *consumer = rd_kafka_new(RD_KAFKA_CONSUMER, conf, errstr, sizeof(errstr));
```

```
    if (!consumer) {
        fprintf(stderr, "Failed to create consumer: %s\n", errstr);
        return 1;
    }

    // 订阅主题
    rd_kafka_topic_partition_list_t *topics = rd_kafka_topic_partition_list_new(1);
    rd_kafka_topic_partition_list_add(topics, "test", RD_KAFKA_PARTITION_UA);
    if (rd_kafka_subscribe(consumer, topics) != RD_KAFKA_RESP_ERR_NO_ERROR) {
        fprintf(stderr, "Failed to subscribe to topic: %s\n",
rd_kafka_err2str(rd_kafka_last_error()));
        rd_kafka_topic_partition_list_destroy(topics);
        rd_kafka_destroy(consumer);
        return 1;
    }
    rd_kafka_topic_partition_list_destroy(topics);

    // 消费消息
    while (1) {
        rd_kafka_message_t *rkmessage = rd_kafka_consumer_poll(consumer, 1000);
        if (rkmessage) {
            msg_consume(rkmessage, NULL);
            rd_kafka_message_destroy(rkmessage);
        }
    }

    // 取消订阅
    rd_kafka_unsubscribe(consumer);

    // 销毁消费者实例
    rd_kafka_destroy(consumer);

    return 0;
}
```

手动提交位点

手动提交位点：消费者在处理完消息后需要手动提交位点。这种方式的优点是可以精确控制位点的提交，避免消息重复消费或丢失。但是需要注意，手动提交位点如果太频繁会导致 Broker CPU 很高，影响性能，随着消息量增加，CPU 消费会很高，影响正常 Broker 的其他功能，因此建议间隔一定消息提交位点。

```
#include <stdio.h>
#include <string.h>
#include <librdkafka/rdkafka.h>

// 消费者消息处理回调
void msg_consume(rd_kafka_message_t *rkmessage, void *opaque) {
    if (rkmessage->err) {
        fprintf(stderr, "%% Consume error for topic \"%s\" [%"PRId32"] "
                "offset %"PRId64": %s\n",
                rd_kafka_topic_name(rkmessage->rkt),
                rkmessage->partition, rkmessage->offset,
                rd_kafka_message_errstr(rkmessage));
    } else {
        printf("%% Message received on topic %s [%"PRId32"] at offset %"PRId64": %.*s\n",
```

```
                rd_kafka_topic_name(rkmessage->rkt),
                rkmessage->partition,
                rkmessage->offset, (int)rkmessage->len, (const char *)rkmessage->payload);
    }
}

int main() {
    rd_kafka_conf_t *conf = rd_kafka_conf_new();

    // 设置Kafka集群的地址
    rd_kafka_conf_set(conf, "bootstrap.servers", "localhost:9092", NULL, 0);

    // 设置消费组ID
    rd_kafka_conf_set(conf, "group.id", "mygroup", NULL, 0);

    // 关闭消费者的自动提交开关
    rd_kafka_conf_set(conf, "enable.auto.commit", "false", NULL, 0);

    // 创建消费者实例
    char errstr[512];
    rd_kafka_t *consumer = rd_kafka_new(RD_KAFKA_CONSUMER, conf, errstr, sizeof(errstr));
    if (!consumer) {
        fprintf(stderr, "Failed to create consumer: %s\n", errstr);
        return 1;
    }

    // 订阅主题
    rd_kafka_topic_partition_list_t *topics = rd_kafka_topic_partition_list_new(1);
    rd_kafka_topic_partition_list_add(topics, "test", RD_KAFKA_PARTITION_UA);
    if (rd_kafka_subscribe(consumer, topics) != RD_KAFKA_RESP_ERR_NO_ERROR) {
        fprintf(stderr, "Failed to subscribe to topic: %s\n",
rd_kafka_err2str(rd_kafka_last_error()));
        rd_kafka_topic_partition_list_destroy(topics);
        rd_kafka_destroy(consumer);
        return 1;
    }
    rd_kafka_topic_partition_list_destroy(topics);

    // 消费消息并手动提交位点
    int message_count = 0;
    while (1) {
        rd_kafka_message_t *rkmessage = rd_kafka_consumer_poll(consumer, 1000);
        if (rkmessage) {
            msg_consume(rkmessage, NULL);

            // 每隔10条消息手动提交位点
            if (++message_count % 10 == 0) {
                if (rd_kafka_commit_message(consumer, rkmessage, 0) !=
RD_KAFKA_RESP_ERR_NO_ERROR) {
                    fprintf(stderr, "Failed to commit offset for message: %s\n",
rd_kafka_err2str(rd_kafka_last_error()));
                } else {
                    printf("Offset %"PRId64" committed\n", rkmessage->offset);
                }
            }
```

```
            rd_kafka_message_destroy(rkmessage);
        }
    }

    // 取消订阅
    rd_kafka_unsubscribe(consumer);

    // 销毁消费者实例
    rd_kafka_destroy(consumer);

    return 0;
}
```

tRpc Go SDK

最近更新时间：2025-03-27 17:43:02

背景

TDMQ CKafka 是一个分布式流处理平台，用于构建实时数据管道和流式应用程序。它提供了高吞吐量、低延迟、可伸缩性和容错性等特性。本文着重介绍 tRpc-Go-Kafka 客户端的关键参数和实践教程，以及常见问题。

调优实践

tRPC-GO-Kafka 封装开源 Kafka SDK，利用 tRPC-Go 拦截器等功能，接入 tRPC-Go 生态。因此实践教程参见 Sarama Go 。

常见问题

生产者问题

1. **使用 CKafka 生产消息时，出现错误 `Message contents does not match its CRC` 。**

   ```
   err:type:framework, code:141, msg:kafka client transport SendMessage: kafka server: Message contents does not match its CRC.
   ```

   插件默认启用了 gzip 压缩，在 target 上加上参数 `compression=none` 关闭压缩即可。

   ```
   target: kafka://ip1:port1?compression=none
   ```

2. **生产时同一用户需要有序，如何配置?**

   客户端增加参数 `partitioner` ，可选 random（默认），roundrobin，hash（按 key 分区）。

   ```
   target: kafka://ip1:port1?clientid=xxx&partitioner=hash
   ```

3. **如何异步生产?**

   客户端增加参数 `async=1`

   ```
   target: kafka://ip1:port1,ip2:port2?clientid=xxx&async=1
   ```

4. **如何使用异步生产写数据回调?**

   需要在代码中重写异步生产写数据的成功/失败的回调函数，例如：

   ```
   import (
     "git.code.oa.com/vicenteli/trpc-database/kafka"
   )

   func init() {
     // 重写默认的异步生产写数据错误回调
     kafka.AsyncProducerErrorCallback = func(err error, topic string, key, value []byte, headers
   []sarama.RecordHeader) {
       // do something if async producer occurred error.
     }

     // 重写默认的异步生产写数据成功回调
     kafka.AsyncProducerSuccCallback = func(topic string, key, value []byte, headers
   []sarama.RecordHeader) {
       // do something if async producer succeed.
     }
   }
   ```

消费者问题

1. **如果消费时 Handle 返回了非 nil 会发生什么?**

   会休眠 3s 后重新消费，不建议这么做，因为返回错误会导致无限重试消费，失败的应该由业务做重试逻辑。

2. **使用 ckafka 消费消息时，出现错误 `client has run out of available brokers to talk to` 。**

   ```
   kafka server transport: consume fail:kafka: client has run out of available brokers to talk to (Is your cluster reachable?)
   ```

   优先检查 brokers 是否可达，然后检查支持的 kafka 客户端版本，尝试在配置文件 address 中加上参数例如 version=0.10.2.0

   ```
   address: ip1:port1?topics=topic1&group=my-group&version=0.10.2.0
   ```

3. **当多个生产者生产时，部分生产者建立连接失败会影响正常的生产者超时?**

   请更新至 v0.2.18。低版本插件在创建生产者时先加锁，再建立连接，建立连接结束后释放锁。如果存在一部分异常生产者建立连接耗时很长，就会导致其他正常生产请求在获取生产者的时候加锁失败，最终超时。此行为在 v0.2.18 已经修复。

4. **消费消息时，提示 `The provider group protocol type is incompatible with the other members` 。**

   ```
   kafka server transport: consume fail:kafka server: The provider group protocol type is incompatible with the other members.
   ```

   同一消费者组的客户端重分组策略不一样，可修改参数 `strategy` ，可选：sticky(默认)，range，roundrobin。

   ```
   address: ip1:port1?topics=topic12&group=my-group&strategy=range
   ```

5. **如何注入自定义配置（远端配置）?**

   如果需要代码中指定配置，先在 `trpc_go.yaml` 中配置 `fake_address` ，然后配合 `kafka.RegisterAddrConfig` 方法注入，`trpc_go.yaml` 配置如下：

   ```
   address: fake_address
   ```

   在服务启动前，注入自定义配置：

   ```
   func main() {
     s := trpc.NewServer()
     // 使用自定义 addr，需在启动 server 前注入
     cfg := kafka.GetDefaultConfig()
     cfg.Brokers = []string{"127.0.0.1:9092"}
     cfg.Topics = []string{"test_topic"}
     kafka.RegisterAddrConfig("fake_address", cfg)
     kafka.RegisterKafkaConsumerService(s.Service("fake_address"), &Consumer{})

     s.Serve()
   }
   ```

6. **如何获取底层 sarama 的上下文信息?**

   通过 `kafka.GetRawSaramaContext` 可以获取底层 `sarama ConsumerGroupSession` 和 `ConsumerGroupClaim`。但是此处暴露这两个接口只是方便用户做监控日志，应该只使用其读方法，调用任何写方法在这里都是未定义行为，可能造成未知结果。

   ```
   // RawSaramaContext 存放 sarama ConsumerGroupSession 和 ConsumerGroupClaim
   // 导出此结构体是为了方便用户实现监控，提供的内容仅用于读，调用任何写方法属于未定义行为
   type RawSaramaContext struct {
       Session  sarama.ConsumerGroupSession
       Claim sarama.ConsumerGroupClaim
   }
   ```

   使用实例：

   ```
   func (Consumer) Handle(ctx context.Context, msg *sarama.ConsumerMessage) error {
       if rawContext, ok := kafka.GetRawSaramaContext(ctx); ok {
           log.Infof("InitialOffset: %d", rawContext.Claim.InitialOffset())
       }
   ```

```
    // ...
    return nil
}
```

Kakfa Client Rebalance 技术详解

最近更新时间：2025-07-03 14:02:01

Client Rebalance 简介

什么是 Client Rebalance

在 Kafka 中，Client Rebalance 是指消费者组（Consumer Group）内的消费者实例重新分配订阅 partition 的过程。当组内成员变化、partition 数变化或订阅关系变更时，Kafka 会触发 Client Rebalance 以确保 partition 在客户端消费者的所有权尽可能均匀分布。

为什么需要 Client Rebalance

消费者组间的 client 实现动态负载均衡，其存在的本质原因是为了确保 partition 与消费者实例的分配关系能够随集群状态动态调整。
场景 1：扩容 client 实例
场景 2：缩容 client 实例
场景 3：client 实例故障
场景 4：topic 扩容
场景 5：topic 删除

Partition 分配算法

既然 Rebalance 的目标是为了让 partition 在众多的 client 中均匀分配，那么下面介绍一下 partition 的分配算法。

RangeAssignor

将 topic 的 partition 按顺序分组，依次分配给消费者，适用于单 topic 的分配场景。

核心原理

1. 单 topic 排序与分组
    - 对每个 topic 的 partition 按编号升序排列（如 topic1 有 partition 0-5），按消费者数量计算每组 partition 数：partition 数 / 消费者数 = 基础 partition数，余数部分由前余数个消费者各多分配 1 个。
    - 例：topic 有 5 个 partition，2 个消费者（C1、C2），计算为：
        - C1：partition 0, 1, 2（5/2 = 2 余 1，前 1 个消费者多拿 1 个）
        - C2：partition 3, 4
2. 多 topic 分配逻辑
    - 对每个 topic 独立执行上述分配，可能导致同一消费者在不同topic中分配到过量partition。

优缺点

优点：
- 按顺序分配，便于理解和调试。
- Rebalance 时分配逻辑稳定，便于预测。

缺点：
- 分配可能不均匀：当 partition 数不能被消费者数整除时，前余数个消费者会多拿 partition。
- 不适合多 topic，多 topic 场景下可能导致 “数据倾斜”：如消费者 A 负责 topic1 的 3 个 partition 和 topic2 的 3 个 partition，而消费者 B 仅负责 2 个partition。

RoundRobinAssignor

将所有 topic 的 partition 统一排序，按轮询方式依次分配给消费者，追求全局均衡。

核心原理

1. 全局 partition 排序

- 将所有订阅 topic 的 partition 按 topic + partition 升序排列（如 topic1-0, topic1-1, topic2-0, topic2-1 ）。

2. 轮询分配机制
   - 按消费者列表顺序（按消费者 ID 升序）依次分配每个 partition，类似 “发牌” 模式。
   - 例：3 个消费者（C1, C2, C3）分配 6 个 partition，结果为：
     - C1: partition 0, 3
     - C2: partition 1, 4
     - C3: partition 2, 5
3. 配置
   - `partition.assignment.strategy` 需显式配置为 `org.apache.kafka.clients.consumer.RoundRobinAssignor` 。

优缺点

优点：
- 全局分配更均匀，避免单 topic Range 算法的倾斜问题。
- 支持跨 topic 的负载均衡，适合多 topic 消费场景。

缺点：
- Rebalance 时可能触发大量 partition 迁移：当消费者数量变化时，所有 partition 需重新轮询分配，开销较大。

StickyAssignor

优先保留历史 partition 分配关系，仅在必要时调整 partition，减少 rebalance 导致的迁移开销

核心原理

1. “粘性” 分配策略
   - rebalance时，分配器会优先将之前分配的partition保留给原消费者，仅当partition必须重新分配时（如消费者离开）才调整。
   - 例：消费者 C1 原有partition 0-2，新增消费者 C3 时，C1 可能仅释放partition 2，保留 0-1。
2. 与 Cooperative 协议的结合
   - 在 Kafka 2.4 + 版本中，CooperativeStickyAssignor支持增量rebalance，分阶段迁移partition，避免全量迁移partition。

优缺点

优点：
- 大幅减少 rebalance 时的 partition 迁移量，降低应用中断时间。

缺点：
- 实现复杂度高，需跟踪历史分配状态（通过 `ownedPartitions` 字段）。
- 自定义分配器需显式实现粘性逻辑，否则可能退化为旧协议。
- 若历史分配存在倾斜，粘性策略可能延续该倾斜（需手动干预或重启重置）。

对比三种均衡算法

维度	RangeAssignor	RoundRobinAssignor	StickyAssignor
分配粒度	单 topic 内的 partition 范围分组	全局 partition 轮询	优先保留历史分配，最小化迁移
均衡性	单 topic 内可能不均衡，多 topic 易倾斜	全局均衡性最佳	历史分配优先，可能延续旧倾斜
rebalance开销	高（全量重新分配）	高（全量重新分配）	低（仅必要 partition 迁移）
状态感知	无状态	无状态	有状态（依赖历史分配记录）
适用场景	单 topic、partition 数少、静态	多 topic、追求全局均衡、	有状态应用、频繁 rebalance、需减少迁移的

	消费者组	动态扩缩容	场景
Kafka 版本支持	全版本	全版本	旧版 StickyAssignor（≤2.3），新版 CooperativeStickyAssignor（≥2.4）

总结

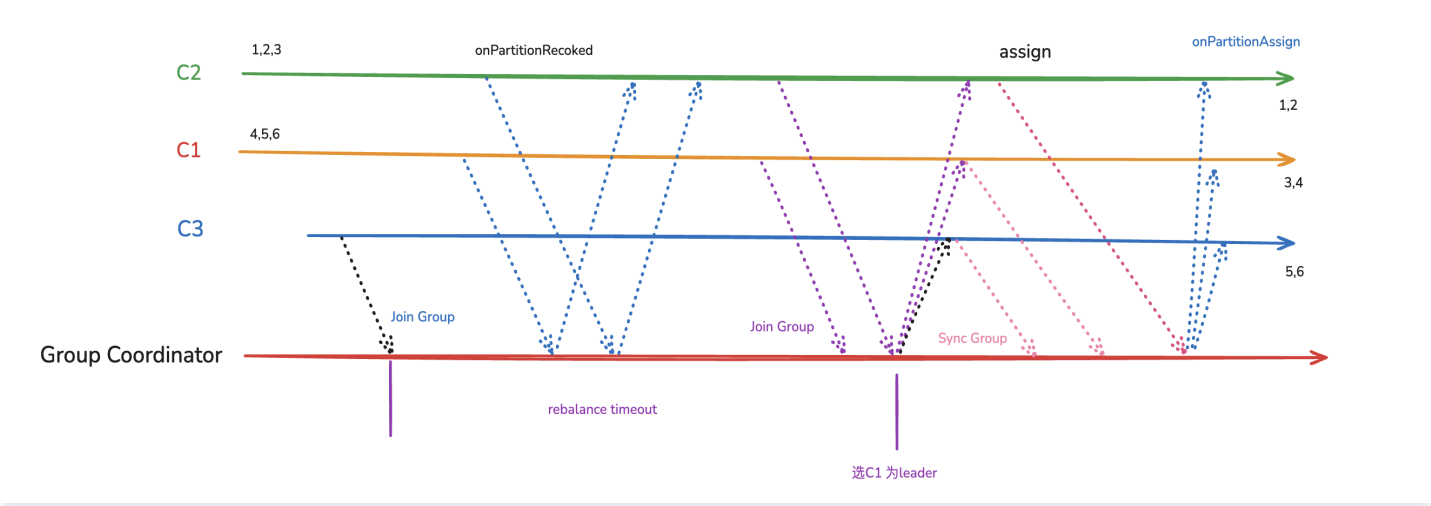
- RangeAssignor：适用于单 topic、消费者数量固定、对分配均匀性要求不高的场景
- RoundRobinAssignor：适用于多 topic 消费、消费者数量动态变化、追求全局均衡的场景
- StickyAssignor：优先用于 Kafka Streams、Kafka Connect 等有状态组件，减少 rebalance 导致的状态重建。

Rebalance 触发场景

- consumer 成员变化
  - consumer 挂掉
  - consumer 扩容
  - consumer 缩容
- topic 变化
  - topic 创建
  - topic 删除
  - partition 扩容

Rebalance 协议流程解析

Eager Rebalance 协议流程



ConsumerRebalanceListener

- onPartitionsRevoked（在发送 join group 请求之前调用 ）
- onPartitionsAssigned（在收到 sync group 请求之后调用）

ConsumerPartitionAssigner

- assign (client leader 调用该接口实现 partition 分配算法)

Eager Rebalance存在缺陷

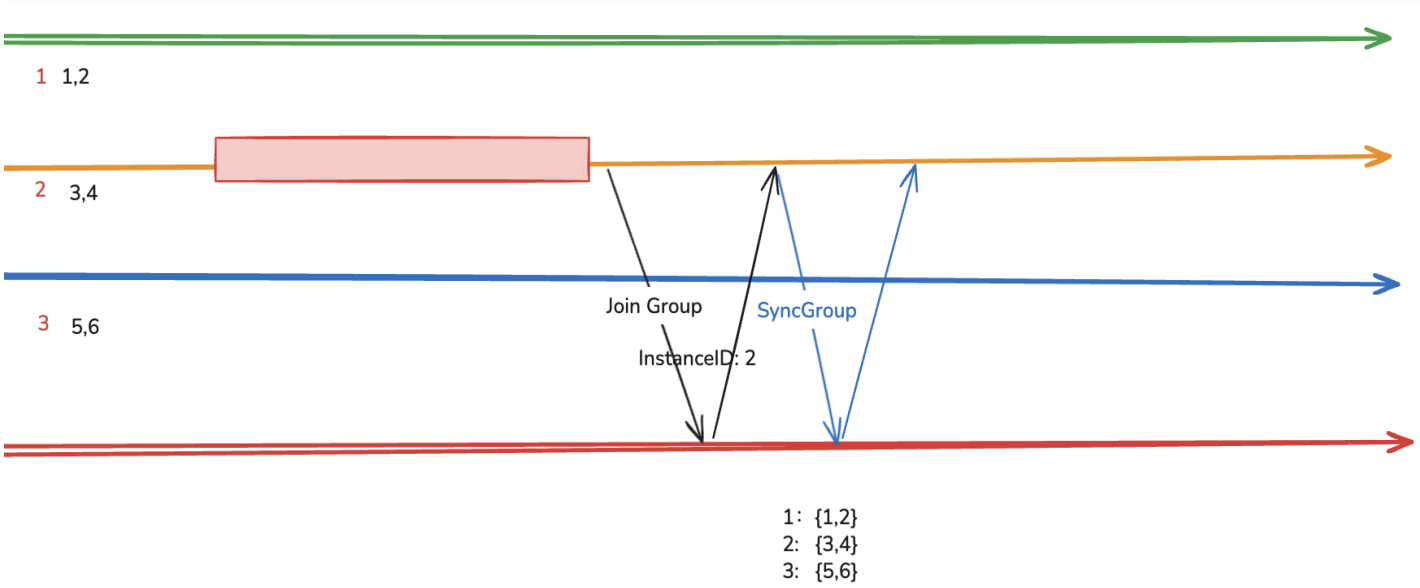
1. Stop the world：消费断流，尤其在 partition 和 client 数量比较大的情况下比较明显。
2. partition 无效迁移：例如在单实例重启场景中 partition 迁移。

优化Eager Rebalance协议

Static Membership Rebalance协议

优化思路：降低触发 rebalance 的概率，提高触发 rebalance 的要求，主要优化单实例重启的场景。
优化方式：

1. 为每个实例增加一个 `instance.id` 的参数，在 k8s 上的一个实例重启之后该 `instance.id` 保持不变。
2. 调整 `session.timeout` 参数，尽量调大 `session.timeout` 以实现实例在重启时 coordinator 感知不到实例没有发送心跳。
3. 新启动的实例携带重启之前的 `instance.id`，coordinator 找到对应的 partitions 信息，返回给新启动的实例。

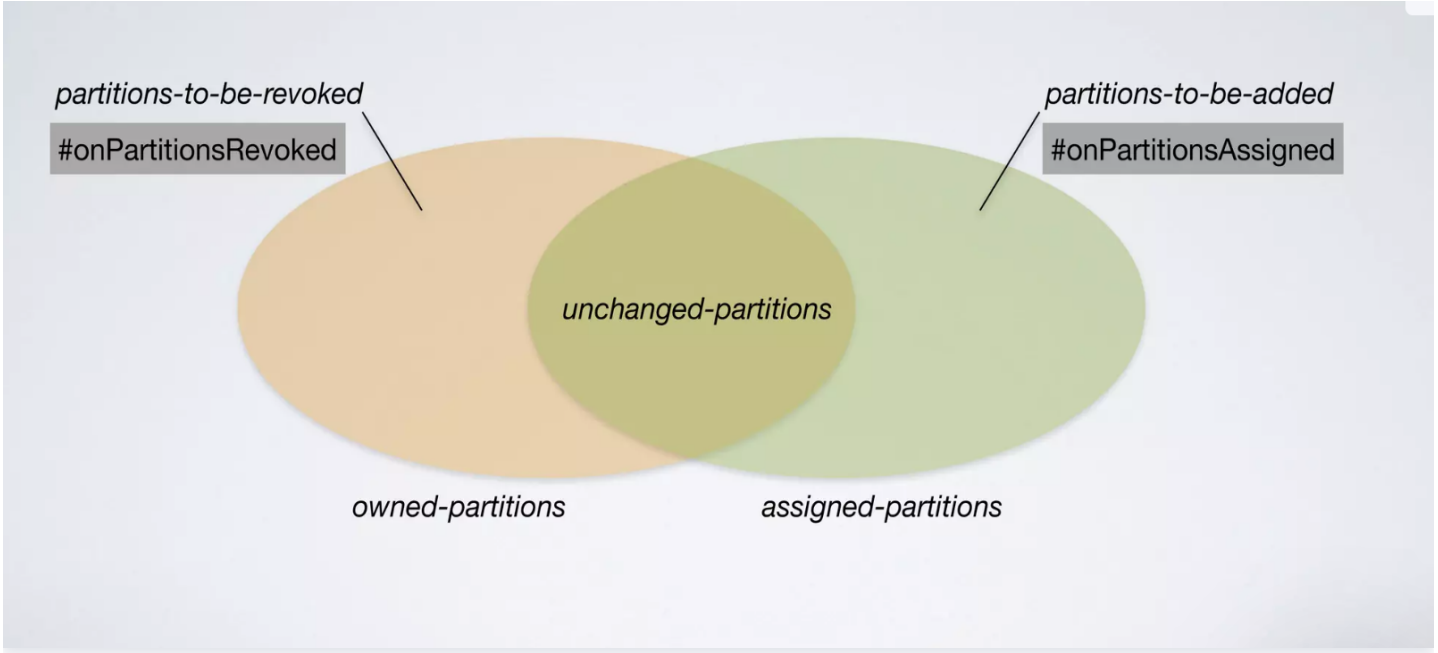


优化后触发 rebalance 的场景：

1. client 扩容
2. client leader 重新 rejoin
3. client 实例挂掉的时间超过 session timeout
4. client缩容

Cooperative Rebalance协议

优化思路：触发 rebalance 的条件保持不变，优化 rebalance 流程，尽量保持 partition 不发生迁移。



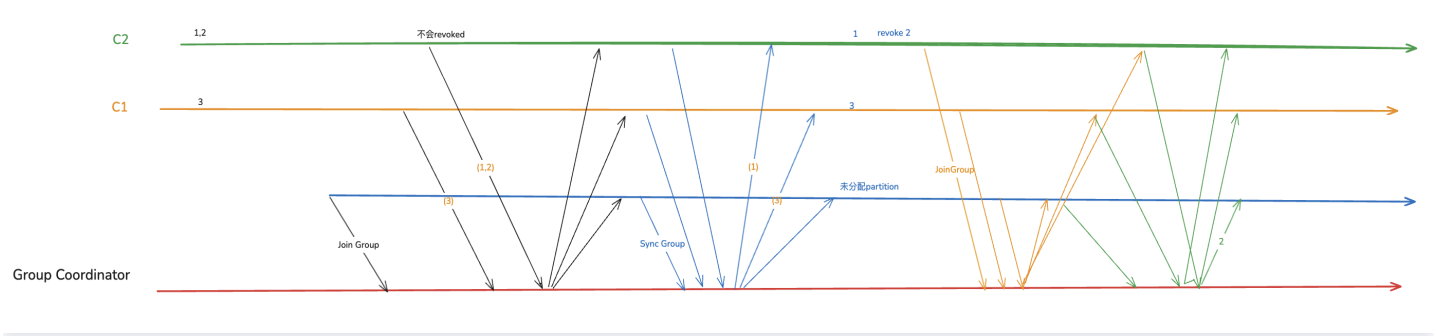
在一次 rebalance 过程中，对于一个 client 的 partition 来说，可以认为处理三种状态：

- 需要被 client 撤销
- 需要被 client 加入
- 保持不变

所以在 rebalance 过程中，对于保持不变的 partition 让其仍继续消费，然后先撤销 partition，再增加 partition。

优化方式：

1. 在 rebalance 之前不再撤销正在消费 partition。
2. 渐进式多轮 rebalance，“撤销旧 partition” 和 “分配新 partition” 两个阶段，消费者可在第一阶段继续处理已保留的 partition，减少服务中断。



从 Eager Rebalance 协议升级到 Cooperative Rebalance 协议

需要 consumer 修改配置重启2次：

1. 引入新协议同时保留旧的协议

   1.1 修改 consumer 配置

```
partition.assignment.strategy =
org.apache.kafka.clients.consumer.CooperativeStickyAssignor,org.apache.kafka.clients.consu
mer.RangeAssignor
```

> ⚠ **注意：**

> 新分配器的配置要放在列表第一位

1.2 重启 consumer，新实例发送的 joinGroup 请求包含新协议，但由于旧协议仍在列表中，消费者组会选择所有分配器共同支持的协议（此时仍为eager，因 RangeAssignor 仅支持 eager）。旧实例继续使用 eager 协议，新实例准备支持 cooperative 协议，但协议尚未切换。

2. 移除协议

从配置中删除旧的协议

```
partition.assignment.strategy = org.apache.kafka.clients.consumer.CooperativeStickyAssignor
```

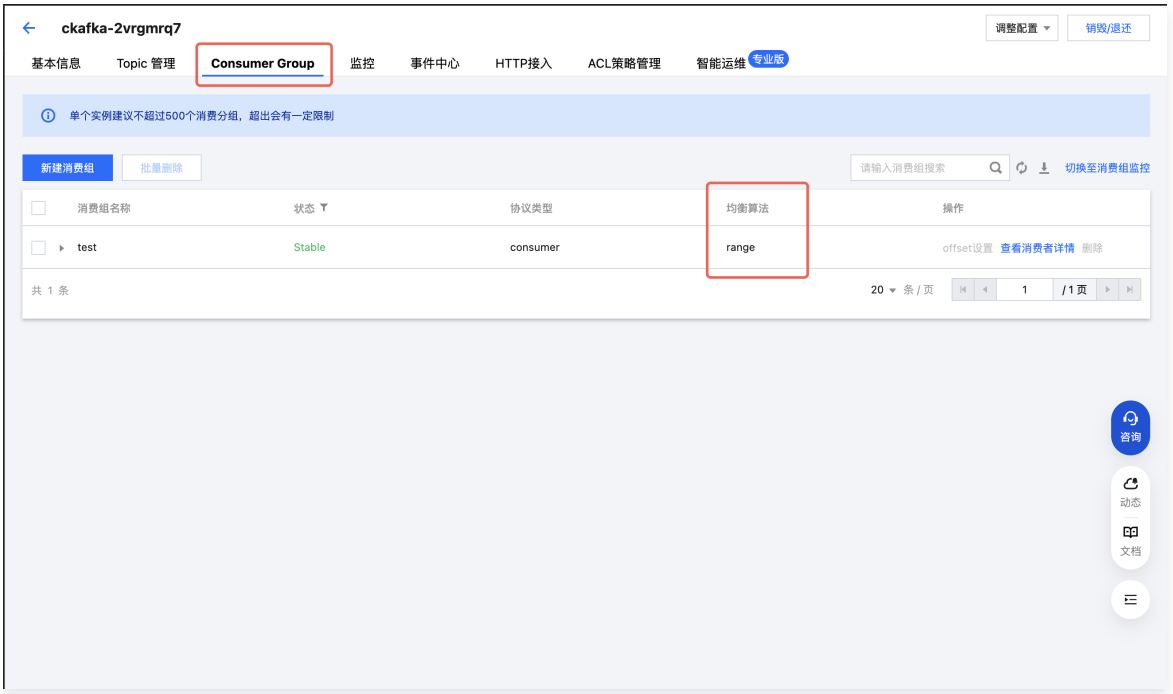
对比 Static Membership 和 Cooperative Rebalance 协议

	Static Membership	Cooperative	
优点	1. 流程简单 2. 从根本上不触发 rebalance 3. 兼容 StickyAssignor	1. 不依赖单独的唯一 id 2. 感知历史分配结果，降低了 rebalance 期间抖动	2个协议不矛盾，可以同时使用
缺点	1. 需要提供instance.id 2. 调大 session timeout，可能导致容器挂掉时感知不明显 3. 不生效时退化为全局 rebalance	1. 流程复杂 2. 需要使用 CooperativeStickyAssignor，依赖粘性分配算法	

Client Rebalance 在腾讯云的实践

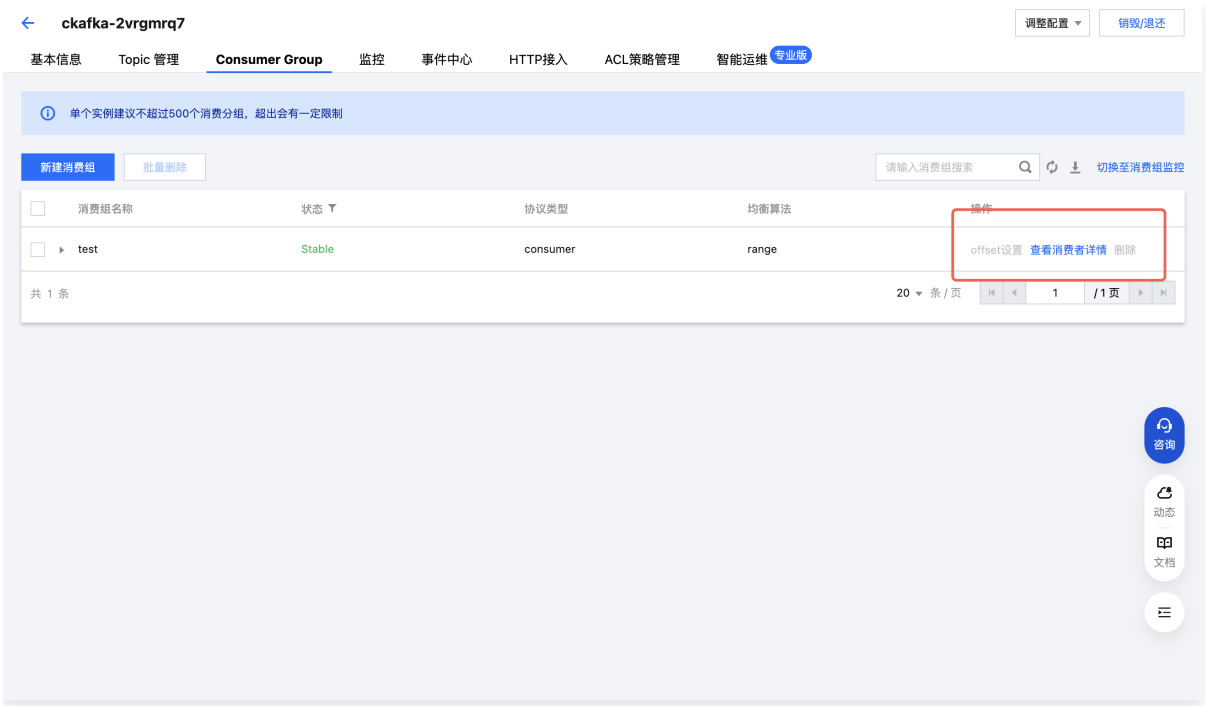
查看当前 Partition 的均衡算法

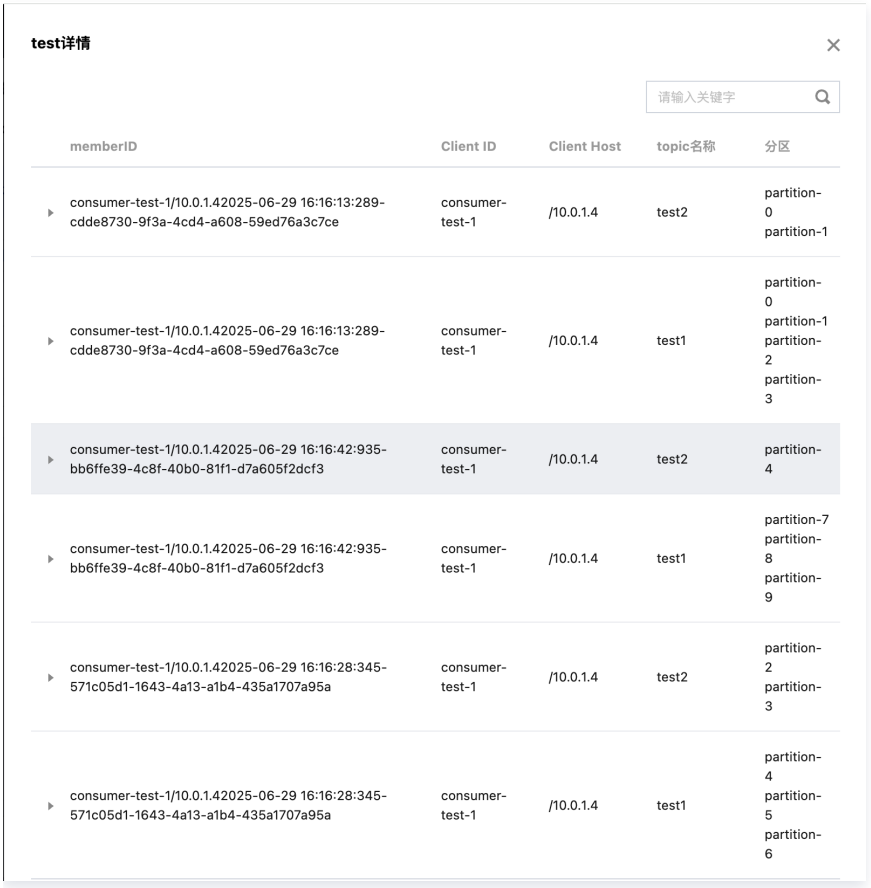
您可以通过进入实例详情，查看 Consumer Group 页签，在列表中查看客户端均衡算法。



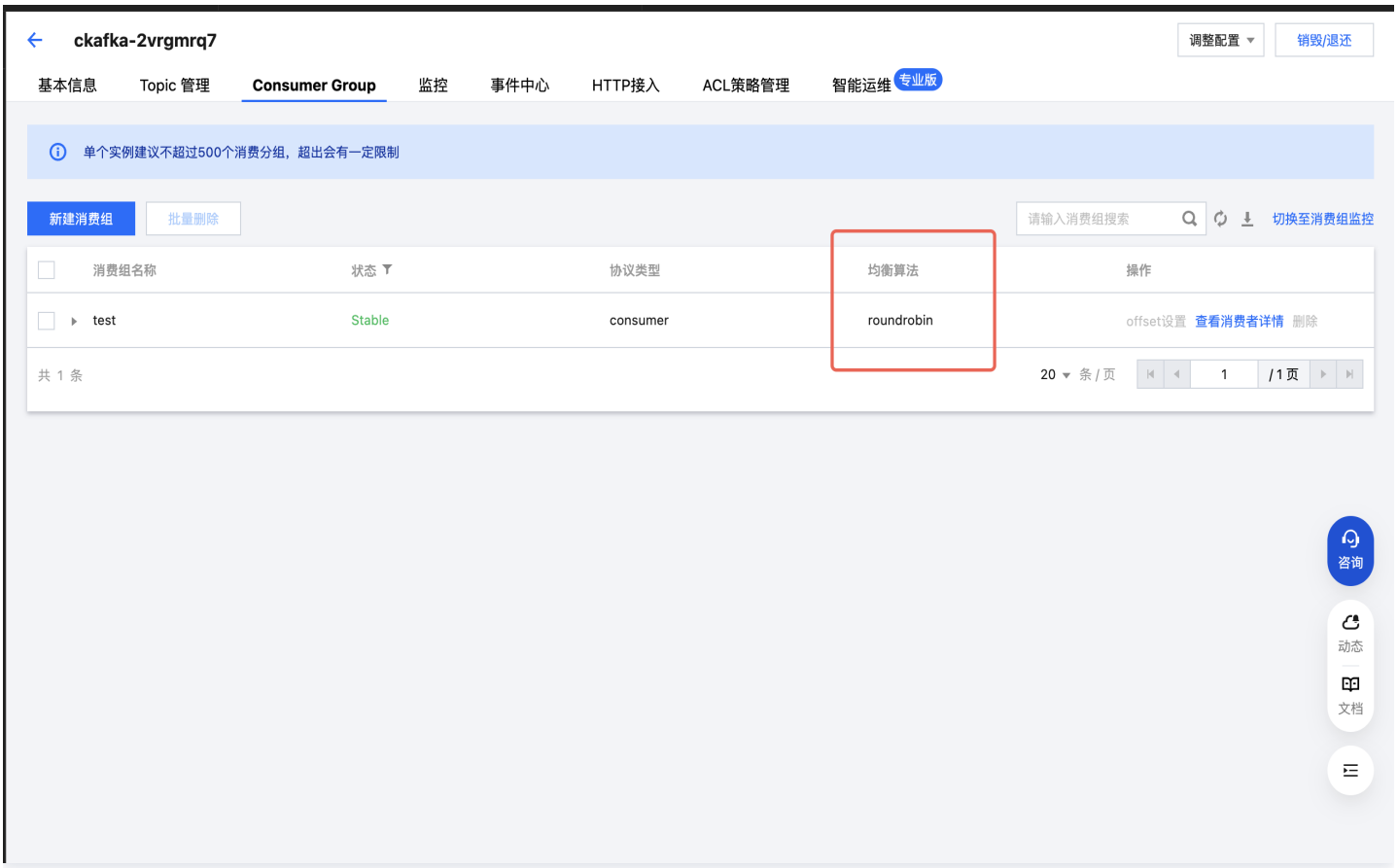
查看 partition 在 client 间的分配结果

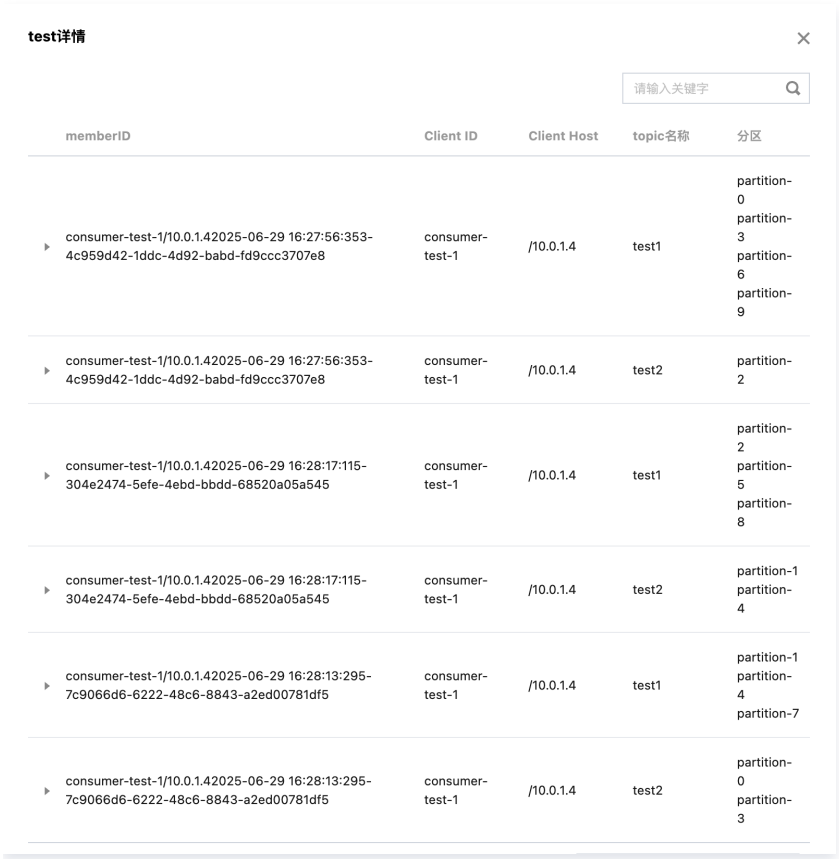
前提：在代码中 `KafkaConsumer` 对象一定要调用 subscribe 函数。
当前使用 group test 来消费 2 个 topic，一个 topic: test1 有 10 个 partition，另外一个 topic: test2 有 5 个 partition。





按照 range 的分配算法后，发现第一个消费者消费 6 个 partition，第二个消费者消费 4 个 partition，第三个消费者消费 5 个 partition，如果改成RoundRobinAssignor 均衡算法后，结果如下：





发现 3 个 client 之间分配的 partition 数量更平均一些，每个 client 都分配5个 partition。

参考资料

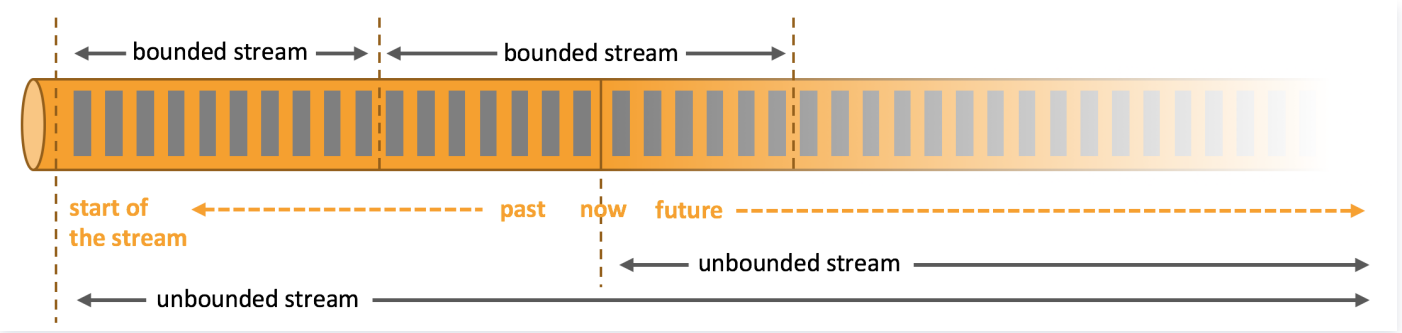
- Introduce static membership protocol to reduce consumer rebalances
- Kafka Consumer Incremental Rebalance Protocol

生态对接
Flink 接入 CKafka

最近更新时间：2024-10-14 14:35:18

Flink 简介

Apache Flink 是一个可以处理流数据的实时处理框架，用于在无界和有界数据流上进行有状态的计算。Flink 能在所有常见集群环境中运行，并能以内存速度和任意规模进行计算。

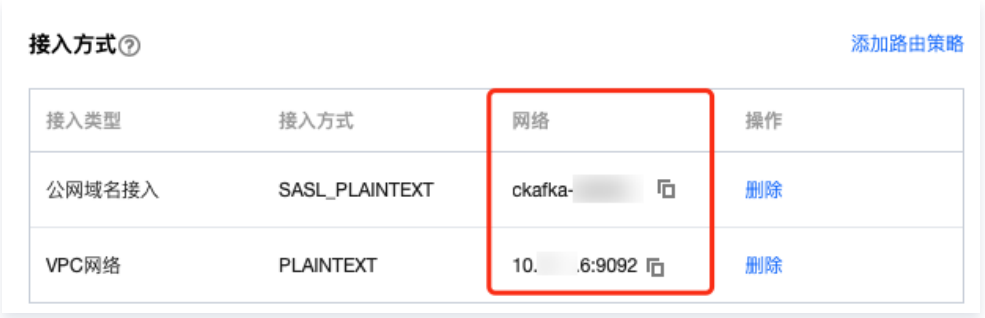


Apache Flink 擅长处理无界和有界数据集。Flink runtime 能够通过对时间和状态的精确控制处理无界数据流，也能够使用为固定大小数据集设计的算法和数据结构对有界数据集进行处理，并达到出色的性能。
应用程序可能会使用来自各种数据源（如消息队列或分布式日志，如 Apache Kafka 或 Kinesis）的实时数据。Flink 提供了 Apache Kafka 连接器，用于从 Kafka topic 中读取或者向其中写入数据，可提供一次精确的处理语义。

操作步骤

步骤1：获取 CKafka 实例接入地址

1. 登录 CKafka 控制台。
2. 在左侧导航栏选择**实例列表**，单击实例的“ID”，进入实例基本信息页面。
3. 在实例的基本信息页面的**接入方式**模块，可获取实例的接入地址，接入地址是生产消费需要用到的 bootstrap-server。



步骤2：创建 Topic

1. 在实例基本信息页面，选择顶部**Topic管理**页签。
2. 在 Topic 管理页面，单击**新建**，创建一个名为 test 的 Topic，接下来将以该 Topic 为例介绍如何消费。

步骤3：添加 Maven 依赖

pom.xml 配置如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>Test-CKafka</artifactId>
    <version>1.0-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-clients</artifactId>
            <version>0.10.2.2</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-simple</artifactId>
            <version>1.7.25</version>
            <scope>compile</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-java</artifactId>
            <version>1.6.1</version>
        </dependency>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-streaming-java_2.11</artifactId>
            <version>1.6.1</version>
        </dependency>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-connector-kafka_2.11</artifactId>
            <version>1.7.0</version>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.3</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

步骤4：消费 CKafka 中的消息

您可以单击以下页面，查看消费消息的两种方式。通过控制台或打印的日志即可查看消费结果。

通过 VPC 方式消费

```
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import java.util.Properties;

public class CKafkaConsumerDemo {
        public static void main(String args[]) throws Exception {
                StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
                Properties properties = new Properties();
                //公网接入域名地址，即公网路由地址，在实例详情页的接入方式模块获取。
                properties.setProperty("bootstrap.servers", "IP:PORT");
                //消费者组id。
                properties.setProperty("group.id", "testConsumerGroup");
                DataStream<String> stream = env
                                .addSource(new FlinkKafkaConsumer<>("topicName", new
SimpleStringSchema(), properties));
                stream.print();
                env.execute();
        }
}
```

通过公网域名方式消费

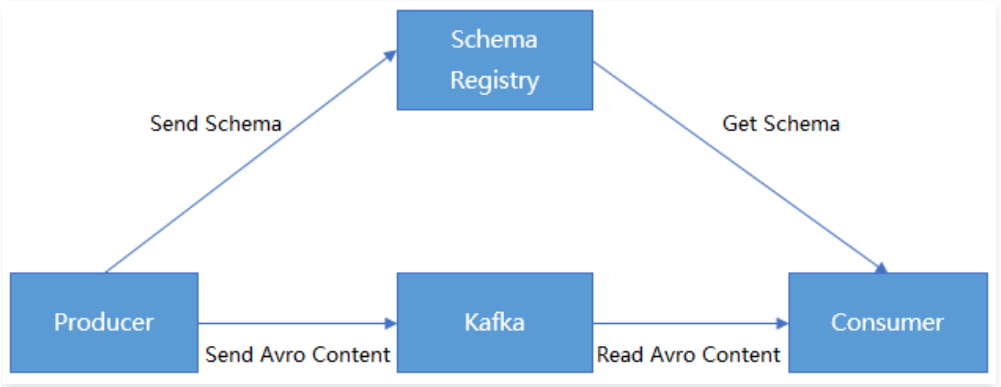
```
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import java.util.Properties;

public class CKafkaConsumerDemo {
        public static void main(String args[]) throws Exception {
                StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
                Properties properties = new Properties();
                //公网接入域名地址，即公网路由地址，在实例详情页的接入方式模块获取。
                properties.setProperty("bootstrap.servers", "IP:PORT");
                //消费者组id。
                properties.setProperty("group.id", "testConsumerGroup");
                properties.setProperty("security.protocol", "SASL_PLAINTEXT");
                properties.setProperty("sasl.mechanism", "PLAIN");
                //用户名和密码，注：用户名是需要拼接，并非控制台的用户名：instanceId#username。
                properties.setProperty("sasl.jaas.config",
```

```
"org.apache.kafka.common.security.plain.PlainLoginModule
required\nusername=\"yourinstanceId#yourusername\"\npassword=\"yourpassword\";");
                properties.setProperty("sasl.kerberos.service.name","kafka");
                DataStream<String> stream = env
                                .addSource(new FlinkKafkaConsumer<>("topicName", new
SimpleStringSchema(), properties));
                stream.print();
                env.execute();
        }
}
```

Schema Registry 接入 CKafka

最近更新时间：2024-12-10 16:22:23

无论是使用传统的 Avro API 自定义序列化类与反序列化类，还是使用 Twitter 的 Bijection 类库实现 Avro 的序列化与反序列化，两种方法有相同的缺点：在每条 Kafka 记录里都嵌入了 Schema，从而导致记录的大小成倍增加。但是不管怎样，在读取记录时仍然需要用到整个 Schema，所以要先找到 Schema。
CKafka 提供了数据共用一个 Schema 的方法：将 Schema 中的内容注册到 Confluent Schema Registry，Kafka Producer 和 Kafka Consumer 通过识别 Confluent Schema Registry 中的 schema 内容进行序列化和反序列化。



前提条件

- 下载 Download JDK 8 。
- 下载 Confluent oss 4.1.1 。
- 已 创建实例 。

操作步骤

步骤1：获取实例接入地址并开启自动创建 Topic

1. 登录 CKafka 控制台 。
2. 在左侧导航栏选择**实例列表**，单击实例的“ID”，进入实例基本信息页面。
3. 在实例的基本信息页面的**接入方式**模块，可获取实例的接入地址。

4. 在**自动创建 Topic**模块开启自动创建 Topic。

> **说明**
> 启动 oss 会创建 schemas 主题，所以实例中需要开启自动创建主题。

步骤2：准备 Confluent 配置

1. 修改 oss 配置文件中的 server 地址等信息。
   PLAINTEXT 接入方式，配置信息如下：

```
kafkastore.bootstrap.servers=PLAINTEXT://xxxx
kafkastore.topic=schemas
debug=true
```

SASL_PLAINTEXT 接入方式，配置信息如下：

```
kafkastore.bootstrap.servers=SASL_PLAINTEXT://ckafka-xxxx.ap-
xxx.ckafka.tencentcloudmq.com:50004
kafkastore.security.protocol=SASL_PLAINTEXT
kafkastore.sasl.mechanism=PLAIN
kafkastore.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required
username='ckafka-xxxx#xxxx' password='xxxx';
kafkastore.topic=schemas
debug=true
```

说明

bootstrap.servers：接入网络，在 CKafka 控制台 的实例详情页面**接入方式**模块的网络列复制。

2. 执行如下命令启动 Schema Registry。

```
bin/schema-registry-start etc/schema-registry/schema-registry.properties
```

运行结果如下：

```
        kafkastore.init.timeout.ms = 60000
 (io.confluent.kafka.schemaregistry.rest.SchemaRegistryConfig:179)
[2019-07-09 16:33:24,889] INFO Logging initialized @523ms (org.eclipse.jetty.util.log:186)
[2019-07-09 16:33:25,607] INFO Initializing KafkaStore with broker endpoints: PLAINTEXT://172.     .8:9     (
io.confluent.kafka.schemaregistry.storage.KafkaStore:103)
[2019-07-09 16:33:26,052] INFO Creating schemas topic _schemas (io.confluent.kafka.schemaregistry.storage.K
afkaStore:186)
[2019-07-09 16:33:26,424] INFO Initialized last consumed offset to -1 (io.confluent.kafka.schemaregistry.st
orage.KafkaStoreReaderThread:138)
[2019-07-09 16:33:26,437] INFO [kafka-store-reader-thread-_schemas]: Starting (io.confluent.kafka.schemareg
istry.storage.KafkaStoreReaderThread:66)
[2019-07-09 16:33:27,152] INFO Wait to catch up until the offset of the last message at 0 (io.confluent.kaf
ka.schemaregistry.storage.KafkaStore:277)
[2019-07-09 16:33:27,221] INFO Joining schema registry with Kafka-based coordination (io.confluent.kafka.sc
hemaregistry.storage.KafkaSchemaRegistry:209)
[2019-07-09 16:33:27,316] INFO Finished rebalance with master election result: Assignment{version=1, error=
0, master='sr-1-/172.26.0.11-2019-07-09 16:33:27:278-f5aa186c-a6c2-4c93-95dd-                ', masterIdentity=
version=1,host=VM_0_11_centos,port=8081,scheme=http,masterEligibility=true} (io.confluent.kafka.schemaregis
try.masterelector.kafka.KafkaGroupMasterElector:232)
[2019-07-09 16:33:27,336] INFO Wait to catch up until the offset of the last message at 1 (io.confluent.kaf
ka.schemaregistry.storage.KafkaStore:277)
[2019-07-09 16:33:27,458] INFO Adding listener: http://0.0.0.0:8081 (io.confluent.rest.Application:190)
```

步骤3：收发消息

现有 schema 文件，其中内容如下：

```
{
    "type": "record",
    "name": "User",
    "fields": [
        {"name": "id", "type": "int"},
        {"name": "name",  "type": "string"},
        {"name": "age", "type": "int"}
    ]
}
```

1. 注册 schema 到对应 Topic（注册 Topic 名为 test）。
   下面的脚本是直接在 Schema Registry 部署的环境中使用 curl 命令调用对应 API 实现注册的一个示例：

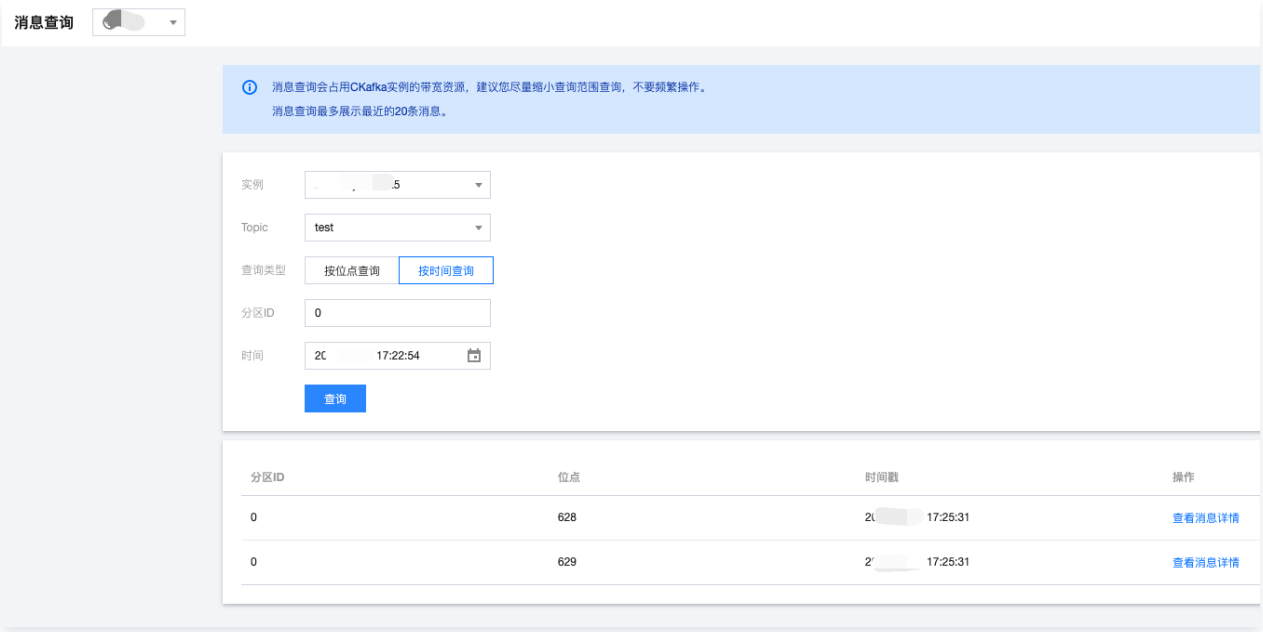
```
curl -X POST -H "Content-Type: application/vnd.schemaregistry.v1+json" \
--data '{"schema": "{\"type\": \"record\", \"name\": \"User\", \"fields\": [{\"name\":
\"id\", \"type\": \"int\"}, {\"name\": \"name\",  \"type\": \"string\"}, {\"name\": \"age\",
\"type\": \"int\"}]}"}' \
http://127.0.0.1:8081/subjects/test/versions
```

2. Kafka Producer 发送数据：

```
package schemaTest;
import java.util.Properties;
import java.util.Random;
import org.apache.avro.Schema;
import org.apache.avro.generic.GenericData;
import org.apache.avro.generic.GenericRecord;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;
public class SchemaProduce {
     public static final String USER_SCHEMA = "{\"type\": \"record\", \"name\": \"User\", " +
                "\"fields\": [{\"name\": \"id\", \"type\": \"int\"}, " +
                "{\"name\": \"name\",  \"type\": \"string\"}, {\"name\": \"age\", \"type\":
\"int\"}]}";
        public static void main(String[] args) throws Exception {
            Properties props = new Properties();
            // 添加CKafka实例的接入地址
          props.put("bootstrap.servers", "xx.xx.xx.xx:xxxx");
            props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
            // 使用 Confluent 实现的 KafkaAvroSerializer
            props.put("value.serializer",
"io.confluent.kafka.serializers.KafkaAvroSerializer");
            // 添加 schema 服务的地址，用于获取 schema
            props.put("schema.registry.url", "http://127.0.0.1:8081");
            Producer<String, GenericRecord> producer = new KafkaProducer<>(props);
            Schema.Parser parser = new Schema.Parser();
            Schema schema = parser.parse(USER_SCHEMA);
            Random rand = new Random();
            int id = 0;
            while(id < 100) {
```

```
                id++;
                String name = "name" + id;
                int age = rand.nextInt(40) + 1;
                GenericRecord user = new GenericData.Record(schema);
                user.put("id", id);
                user.put("name", name);
                user.put("age", age);
                ProducerRecord<String, GenericRecord> record = new ProducerRecord<>("test",
user);
                producer.send(record);
                Thread.sleep(1000);
            }
            producer.close();
        }
}
```

运行一段时间后，在 CKafka 控制台 的 **topic 管理**页面，选择对应的 Topic ，单击**更多 > 消息查询**，查看刚刚发送的消息。

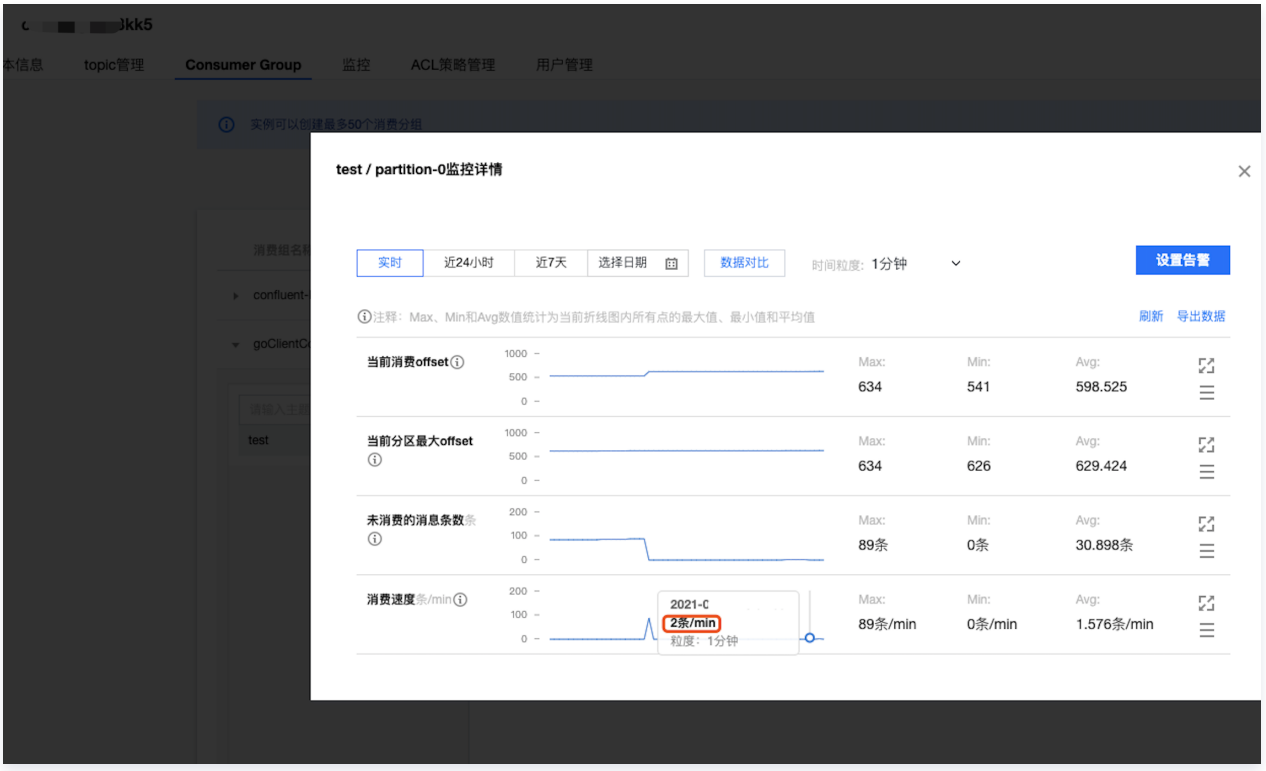


3. Kafka Consumer 消费数据：

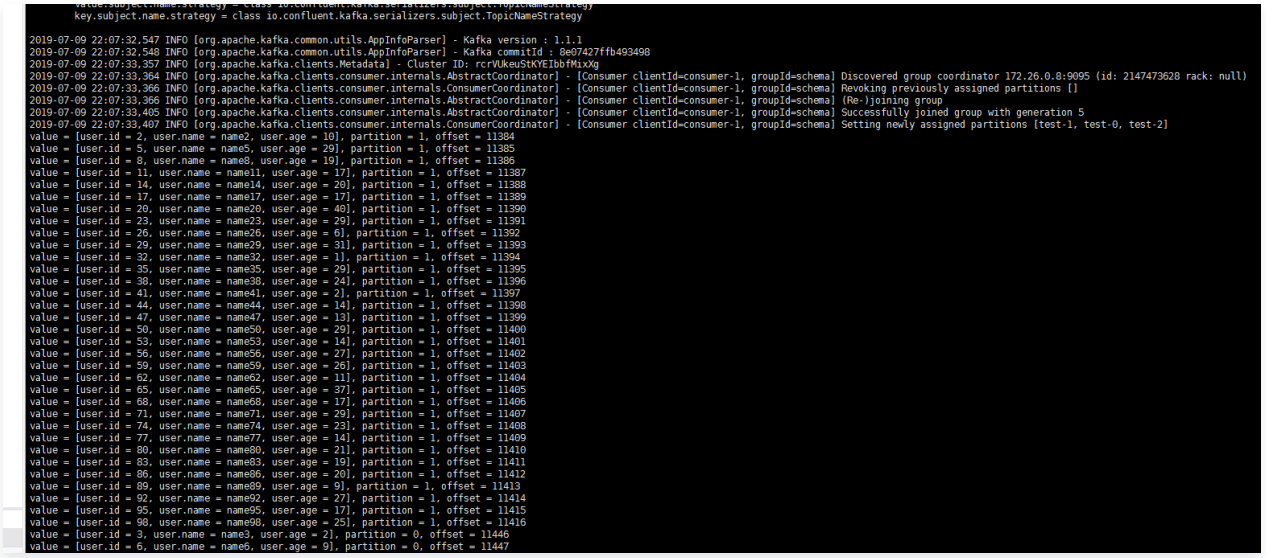
```
package schemaTest;
import java.util.Collections;
import java.util.Properties;
import org.apache.avro.generic.GenericRecord;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
public class SchemaProduce {
    public static void main(String[] args) throws Exception {
        Properties props = new Properties();
        props.put("bootstrap.servers", "xx.xx.xx.xx:xxxx"); //CKafka实例的接入地址
        props.put("group.id", "schema");
        props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
        // 使用Confluent实现的KafkaAvroDeserializer
        props.put("value.deserializer",
"io.confluent.kafka.serializers.KafkaAvroDeserializer");
```

```
        // 添加schema服务的地址，用于获取schema
        props.put("schema.registry.url", "http://127.0.0.1:8081");
        KafkaConsumer<String, GenericRecord> consumer = new KafkaConsumer<>(props);
        consumer.subscribe(Collections.singletonList("test"));
        try {
            while (true) {
                ConsumerRecords<String, GenericRecord> records = consumer.poll(10);
                for (ConsumerRecord<String, GenericRecord> record : records) {
                    GenericRecord user = record.value();
                    System.out.println("value = [user.id = " + user.get("id") + ", " +
"user.name = "
                            + user.get("name") + ", " + "user.age = " + user.get("age") + "],
"
                            + "partition = " + record.partition() + ", " + "offset = " +
record.offset());
                }
            }
        } finally {
            consumer.close();
        }
    }
}
```

在 CKafka 控制台 的**Consumer Group**页面，选择 schema消费组名称，在主题名称输入 Topic 名称，单击**查看消费者详情**，查看消费详情。



启动消费者进行消费，下图为消费日志截图：



Spark Streaming 接入 CKafka

最近更新时间：2024-10-10 18:11:21

Spark Streaming 是 Spark Core 的一个扩展，用于高吞吐且容错地处理持续性的数据，目前支持的外部输入有 Kafka、Flume、HDFS/S3、Kinesis、Twitter 和 TCP socket。

Spark Streaming 将连续数据抽象成 DStream（Discretized Stream），而 DStream 由一系列连续的 RDD（弹性分布式数据集）组成，每个 RDD 是一定时间间隔内产生的数据。使用函数对 DStream 进行处理其实即为对这些 RDD 进行处理。

使用 Spark Streaming 作为 Kafka 的数据输入时，可支持 Kafka 稳定版本与实验版本：

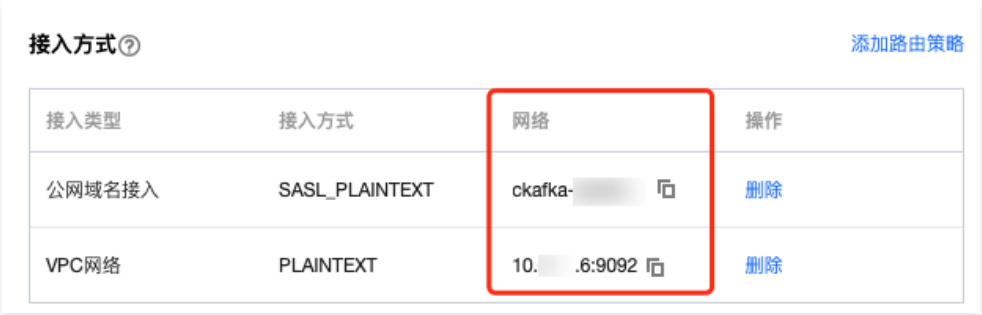
Kafka Version	spark-streaming-kafka-0.8	spark-streaming-kafka-0.10
Broker Version	0.8.2.1 or higher	0.10.0 or higher
Api Maturity	Deprecated	Stable
Language Support	Scala、Java、Python	Scala、Java
Receiver DStream	Yes	No
Direct DStream	Yes	Yes
SSL / TLS Support	No	Yes
Offset Commit Api	No	Yes
Dynamic Topic Subscription	No	Yes

目前 CKafka 兼容 0.9及以上的版本，本次实践使用 0.10.2.1 版本的 Kafka 依赖。
此外，EMR 中的 Spark Streaming 也支持直接对接 CKafka，详见 SparkStreaming 对接 CKafka 服务 。

操作步骤

步骤1：获取 CKafka 实例接入地址

1. 登录 CKafka 控制台。
2. 在左侧导航栏选择**实例列表**，单击实例的“ID”，进入实例基本信息页面。
3. 在实例的基本信息页面的**接入方式**模块，可获取实例的接入地址，接入地址是生产消费需要用到的 bootstrap-server。



步骤2：创建 Topic

1. 在实例基本信息页面，选择顶部**Topic管理**页签。
2. 在 Topic 管理页面，单击**新建**，创建一个名为 test 的 Topic，接下来将以该 Topic 为例介绍如何生产消费。

步骤3：准备云服务器环境

Centos6.8 系统

package	version
sbt	0.13.16
hadoop	2.7.3
spark	2.1.0
protobuf	2.5.0
ssh	CentOS 默认安装
Java	1.8

具体安装步骤参见 配置环境。

步骤4：对接 CKafka

向 CKafka 中生产消息

这里使用 0.10.2.1 版本的 Kafka 依赖。

1. 在 `build.sbt` 添加依赖：

```
name := "Producer Example"
version := "1.0"
scalaVersion := "2.11.8"
libraryDependencies += "org.apache.kafka" % "kafka-clients" % "0.10.2.1"
```

2. 配置 `producer_example.scala`：

```
import java.util.Properties
import org.apache.kafka.clients.producer._
object ProducerExample extends App {
 val  props = new Properties()
 props.put("bootstrap.servers", "172.16.16.12:9092") //实例信息中的内网 IP 与端口

 props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer")
 props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer")

 val producer = new KafkaProducer[String, String](props)
 val TOPIC="test"  //指定要生产的 Topic
 for(i<- 1 to 50){
        val record = new ProducerRecord(TOPIC, "key", s"hello $i") //生产 key
是"key",value 是 hello i 的消息
        producer.send(record)
 }
 val record = new ProducerRecord(TOPIC, "key", "the end "+new java.util.Date)
 producer.send(record)
 producer.close() //最后要断开
}
```

更多有关 ProducerRecord 的用法请参见 ProducerRecord 文档。

从 CKafka 消费消息

DirectStream

1. 在 `build.sbt` 添加依赖：

```
name := "Consumer Example"
version := "1.0"
scalaVersion := "2.11.8"
libraryDependencies += "org.apache.spark" %% "spark-core" % "2.1.0"
libraryDependencies += "org.apache.spark" %% "spark-streaming" % "2.1.0"
libraryDependencies += "org.apache.spark" %% "spark-streaming-kafka-0-10" % "2.1.0"
```

2. 配置 `DirectStream_example.scala`：

```
import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.kafka.common.TopicPartition
import org.apache.spark.streaming.kafka010._
import org.apache.spark.streaming.kafka010.LocationStrategies.PreferConsistent
import org.apache.spark.streaming.kafka010.ConsumerStrategies.Subscribe
import org.apache.spark.streaming.kafka010.KafkaUtils
import org.apache.spark.streaming.kafka010.OffsetRange
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import collection.JavaConversions._
import Array._
```

```
object Kafka {
    def main(args: Array[String]) {
        val kafkaParams = Map[String, Object](
            "bootstrap.servers" -> "172.16.16.12:9092",
            "key.deserializer" -> classOf[StringDeserializer],
            "value.deserializer" -> classOf[StringDeserializer],
            "group.id" -> "spark_stream_test1",
            "auto.offset.reset" -> "earliest",
            "enable.auto.commit" -> "false"
        )

        val sparkConf = new SparkConf()
        sparkConf.setMaster("local")
        sparkConf.setAppName("Kafka")
        val ssc = new StreamingContext(sparkConf, Seconds(5))
        val topics = Array("spark_test")

        val offsets : Map[TopicPartition, Long] = Map()

        for (i <- 0 until 3){
            val tp = new TopicPartition("spark_test", i)
            offsets.updated(tp , 0L)
        }
        val stream = KafkaUtils.createDirectStream[String, String](
            ssc,
            PreferConsistent,
            Subscribe[String, String](topics, kafkaParams)
        )
        println("directStream")
        stream.foreachRDD{ rdd=>
            //输出获得的消息
            rdd.foreach{iter =>
                val i = iter.value
                println(s"${i}")
            }
            //获得offset
            val offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges
            rdd.foreachPartition { iter =>
                val o: OffsetRange = offsetRanges(TaskContext.get.partitionId)
                println(s"${o.topic} ${o.partition} ${o.fromOffset} ${o.untilOffset}")
            }
        }

        // Start the computation
        ssc.start()
        ssc.awaitTermination()
    }
}
```

RDD

1. 配置 `build.sbt`（配置同上，单击查看）。
2. 配置 `RDD_example`：

```
import org.apache.kafka.clients.consumer.ConsumerRecord
```

```
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.spark.streaming.kafka010._
import org.apache.spark.streaming.kafka010.LocationStrategies.PreferConsistent
import org.apache.spark.streaming.kafka010.ConsumerStrategies.Subscribe
import org.apache.spark.streaming.kafka010.KafkaUtils
import org.apache.spark.streaming.kafka010.OffsetRange
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import collection.JavaConversions._
import Array._
object Kafka {
    def main(args: Array[String]) {
        val kafkaParams = Map[String, Object](
            "bootstrap.servers" -> "172.16.16.12:9092",
            "key.deserializer" -> classOf[StringDeserializer],
            "value.deserializer" -> classOf[StringDeserializer],
            "group.id" -> "spark_stream",
            "auto.offset.reset" -> "earliest",
            "enable.auto.commit" -> (false: java.lang.Boolean)
        )
        val sc = new SparkContext("local", "Kafka", new SparkConf())
        val java_kafkaParams : java.util.Map[String, Object] = kafkaParams
        //按顺序向 partition 拉取相应 offset 范围的消息，如果拉取不到则阻塞直到超过等待时间或者新生产
消息达到拉取的数量
        val offsetRanges = Array[OffsetRange](
            OffsetRange("spark_test", 0, 0, 5),
            OffsetRange("spark_test", 1, 0, 5),
            OffsetRange("spark_test", 2, 0, 5)
        )
        val range = KafkaUtils.createRDD[String, String](
            sc,
            java_kafkaParams,
            offsetRanges,
            PreferConsistent
        )
        range.foreach(rdd=>println(rdd.value))
        sc.stop()
    }
}
```

更多 `kafkaParams` 用法参见 kafkaParams 文档。

配置环境

安装 sbt

1. 在 sbt 官网 上下载 sbt 包。
2. 解压后在 sbt 的目录下创建一个 sbt_run.sh 脚本并增加可执行权限，脚本内容如下：

```
#!/bin/bash
SBT_OPTS="-Xms512M -Xmx1536M -Xss1M -XX:+CMSClassUnloadingEnabled -XX:MaxPermSize=256M"
java $SBT_OPTS -jar `dirname $0`/bin/sbt-launch.jar "$@"
```

```
chmod u+x ./sbt_run.sh
```

3. 执行以下命令。

```
./sbt-run.sh sbt-version
```

若能看到 sbt 版本说明可以正常运行。

安装 protobuf

1. 下载 protobuf 相应版本。
2. 解压后进入目录。

```
./configure
make && make install
```

需要预先安装 gcc-g++，执行中可能需要 root 权限。

3. 重新登录，在命令行中输入下述内容。

```
protoc --version
```

4. 若能看到 protobuf 版本说明可以正常运行。

安装 Hadoop

1. 访问 Hadoop 官网 下载所需要的版本。
2. 增加 Hadoop 用户。

```
useradd -m hadoop -s /bin/bash
```

3. 增加管理员权限。

```
visudo
```

4. 在 `root ALL=(ALL) ALL` 下增加一行。`hadoop ALL=(ALL) ALL` 保存退出。
5. 使用 Hadoop 进行操作。

```
su hadoop
```

6. SSH 无密码登录。

```
cd ~/.ssh/                     # 若没有该目录，请先执行一次ssh localhost
ssh-keygen -t rsa              # 会有提示，都按回车就可以
cat id_rsa.pub >> authorized_keys  # 加入授权
chmod 600 ./authorized_keys    # 修改文件权限
```

7. 安装 Java。

```
sudo yum install java-1.8.0-openjdk java-1.8.0-openjdk-devel
```

8. 配置 ${JAVA_HOME}。

```
vim /etc/profile
```

在文末加上下述内容：

```
export JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.121-0.b13.el6_8.x86_64/jre
export PATH=$PATH:$JAVA_HOME
```

根据安装情况修改对应路径。

9. 解压 Hadoop，进入目录。

```
./bin/hadoop version
```

若能显示版本信息说明能正常运行。

10. 配置单机伪分布式（可根据需要搭建不同形式的集群）。

```
vim /etc/profile
```

在文末加上下述内容：

```
export HADOOP_HOME=/usr/local/hadoop
export PATH=$HADOOP_HOME/bin:$PATH
```

根据安装情况修改对应路径。

11. 修改 `/etc/hadoop/core-site.xml` 。

```
<configuration>
    <property>
        <name>hadoop.tmp.dir</name>
        <value>file:/usr/local/hadoop/tmp</value>
        <description>Abase for other temporary directories.</description>
    </property>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://localhost:9000</value>
    </property>
</configuration>
```

12. 修改 `/etc/hadoop/hdfs-site.xml` 。

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
    <property>
        <name>dfs.namenode.name.dir</name>
        <value>file:/usr/local/hadoop/tmp/dfs/name</value>
    </property>
    <property>
```

```
        <name>dfs.datanode.data.dir</name>
        <value>file:/usr/local/hadoop/tmp/dfs/data</value>
    </property>
</configuration>
```

13. 修改 `/etc/hadoop/hadoop-env.sh` 中的 JAVA_HOME 为 Java 的路径。

```
export JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.121-0.b13.el6_8.x86_64/jre
```

14. 执行 NameNode 格式化。

```
./bin/hdfs namenode -format
```

显示 `Exitting with status 0` 则表示成功。

15. 启动 Hadoop。

```
./sbin/start-dfs.sh
```

成功启动会存在 `NameNode` 进程，`DataNode` 进程，`SecondaryNameNode` 进程。

安装 Spark

访问 Spark 官网 下载所需要的版本。
因为之前安装了 Hadoop，所以选择使用 `Pre-build with user-provided Apache Hadoop` 。

> **说明**
> 本示例同样使用 `hadoop` 用户进行操作。

1. 解压进入目录。
2. 修改配置文件。

```
cp ./conf/spark-env.sh.template ./conf/spark-env.sh
vim ./conf/spark-env.sh
```

在第一行添加下述内容：

```
export SPARK_DIST_CLASSPATH=$(/usr/local/hadoop/bin/hadoop classpath)
```

根据 hadoop 安装情况修改路径。

3. 运行示例。

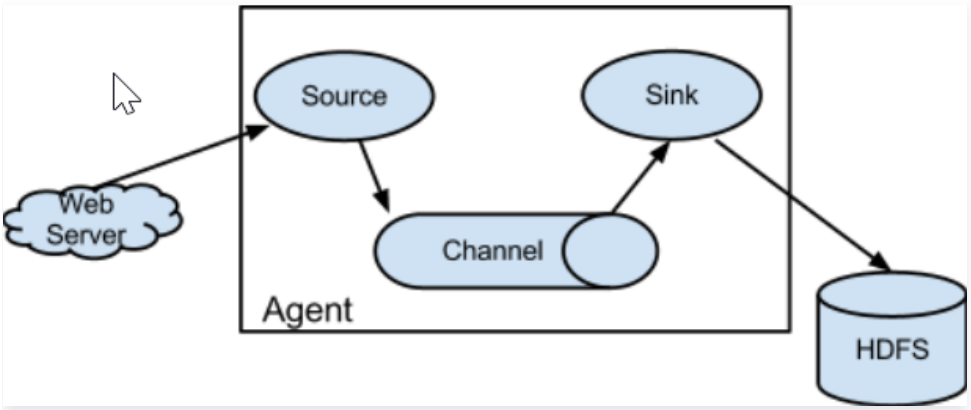
```
bin/run-example SparkPi
```

若成功安装可以看到程序输出 π 的近似值。

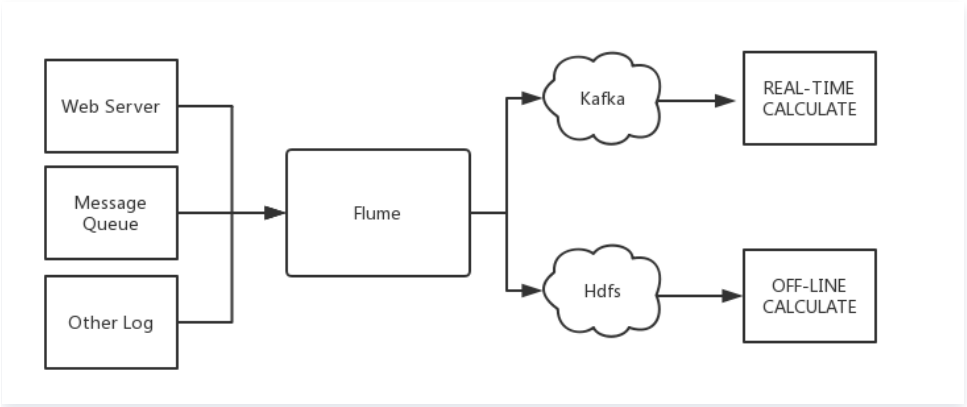
Flume 接入 CKafka

最近更新时间：2024-10-14 17:49:21

Apache Flume 是一个分布式、可靠、高可用的日志收集系统，支持各种各样的数据来源（如 HTTP、Log 文件、JMS、监听端口数据等），能将这些数据源的海量日志数据进行高效收集、聚合、移动，最后存储到指定存储系统中（如 Kafka、分布式文件系统、Solr 搜索服务器等）。
Flume 基本结构如下：



Flume 以 agent 为最小的独立运行单位。一个 agent 就是一个 JVM，单个 agent 由 Source、Sink 和 Channel 三大组件构成。



Flume 与 Kafka
把数据存储到 HDFS 或者 HBase 等下游存储模块或者计算模块时需要考虑各种复杂的场景，例如并发写入的量以及系统承载压力、网络延迟等问题。Flume 作为灵活的分布式系统具有多种接口，同时提供可定制化的管道。
在生产处理环节中，当生产与处理速度不一致时，Kafka 可以充当缓存角色。Kafka 拥有 partition 结构以及采用 append 追加数据，使 Kafka 具有优秀的吞吐能力；同时其拥有 replication 结构，使 Kafka 具有很高的容错性。
所以将 Flume 和 Kafka 结合起来，可以满足生产环境中绝大多数要求。

Flume 接入开源 Kafka

准备工作

- 下载 Apache Flume （1.6.0以上版本兼容 Kafka）
- 下载 Kafka工具包 （0.9.x以上版本，0.8已经不支持）
- 确认 Kafka 的 Source、 Sink 组件已经在 Flume 中。

接入方式

Kafka 可作为 Source 或者 Sink 端对消息进行导入或者导出。

Kafka Source

配置 kafka 作为消息来源，即将自己作为消费者，从 Kafka 中拉取数据传入到指定 Sink 中。主要配置选项如下：

配置项	说明
channels	自己配置的 Channel
type	必须为：org.apache.flume.source.kafka.KafkaSource
kafka.bootstrap.servers	Kafka Broker 的服务器地址
kafka.consumer.group.id	作为 Kafka 消费端的 Group ID
kafka.topics	Kafka 中数据目标 Topic
batchSize	每次写入 Channel 的大小
batchDurationMillis	每次写入最大间隔时间

示例：

```
tier1.sources.source1.type = org.apache.flume.source.kafka.KafkaSource
tier1.sources.source1.channels = channel1
tier1.sources.source1.batchSize = 5000
tier1.sources.source1.batchDurationMillis = 2000
tier1.sources.source1.kafka.bootstrap.servers = localhost:9092
tier1.sources.source1.kafka.topics = test1, test2
tier1.sources.source1.kafka.consumer.group.id = custom.g.id
```

更多内容请参见 Apache Flume 官网 。

Kafka Sink

配置 Kafka 作为内容接收方，即将自己作为生产者，推到 Kafka Server 中等待后续操作。主要配置选项如下：

配置项	说明
channel	自己配置的 Channel
type	必须为：org.apache.flume.sink.kafka.KafkaSink
kafka.bootstrap.servers	Kafka Broker 的服务器
kafka.topics	Kafka 中数据来源 Topic
kafka.flumeBatchSize	每次写入的 Bacth 大小
kafka.producer.acks	Kafka 生产者的生产策略

示例：

```
a1.sinks.k1.channel = c1
a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.k1.kafka.topic = mytopic
a1.sinks.k1.kafka.bootstrap.servers = localhost:9092
a1.sinks.k1.kafka.flumeBatchSize = 20
```

```
a1.sinks.k1.kafka.producer.acks = 1
```

更多内容请参见 Apache Flume 官网。

Flume 接入 CKafka

使用 CKafka 作为 Sink

步骤1：获取 CKafka 实例接入地址

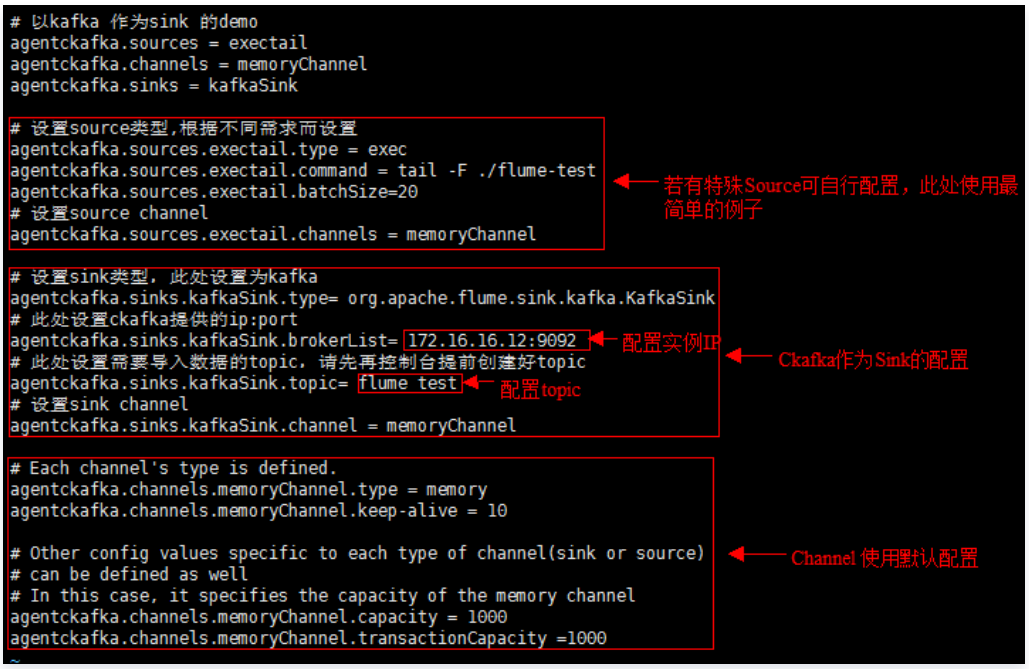
1. 登录 CKafka 控制台。
2. 在左侧导航栏选择**实例列表**，单击实例的“ID”，进入实例基本信息页面。
3. 在实例的基本信息页面的**接入方式**模块，可获取实例的接入地址。

步骤2：创建 Topic

1. 在实例基本信息页面，选择顶部 **Topic 管理**页签。
2. 在 Topic 管理页面，单击**新建**，创建一个名为 flume_test 的 Topic。

步骤3：配置 Flume

1. 下载 Apache Flume 工具包并解压。
2. 编写配置文件 flume-kafka-sink.properties，以下是一个简单的 Java 语言 Demo（配置在解压目录的 conf 文件夹下），若无特殊要求则将自己的实例 IP 与 Topic 替换到配置文件当中即可。本例使用的 source 为 tail -F flume-test，即文件中新增的信息。



代码示例如下：

```
# 以kafka作为sink的demo
agentckafka.source = exectail
agentckafka.channels = memoryChannel
agentckafka.sinks = kafkaSink

# 设置source类型，根据不同需求而设置。若有特殊source可自行配置，此处使用最简单的例子
agentckafka.sources.exectail.type = exec
agentckafka.sources.exetail.command = tail -F ./flume.test
agentckafka.sources.exectail.batchSize = 20
# 设置source channel
agentckafka.sources.exectail.channels = memoryChannel

# 设置sink类型，此处设置为kafka
agentckafka.sinks.kafkaSink.type = org.apache.flume.sink.kafka.KafkaSink
# 此处设置ckafka提供的ip:port
agentckafka.sinks.kafkaSink.brokerList = 172.16.16.12:9092 # 配置实例IP
# 此处设置需要导入数据的topic，请先在控制台提前创建好topic
agentckafka.sinks.kafkaSink.topic = flume test #配置topic
# 设置sink channel
agentckafka.sinks.kafkaSink.channel = memoryChannel

# Channel使用默认配置
# Each channel's type is defined.
agentckafka.channels.memoryChannel.type = memory
agentckafka.channels.memoryChannel.keep-alive = 10

# Other config values specific to each type of channel(sink or source) can be defined as
well
# In this case, it specifies the capacity of the memory channel
agentckafka.channels.memoryChannel.capacity = 1000
agentckafka.channels.memoryChannel.transactionCapacity = 1000
```

3. 执行如下命令启动 Flume。

```
./bin/flume-ng agent -n agentckafka -c conf -f conf/flume-kafka-sink.properties
```

4. 写入消息到 flume-test 文件中，此时消息将由 Flume 写入到 CKafka。

```
[root@VM_16_17_centos apache-flume-1.7.0-bin]# cat flume-test
ckafka
```

5. 启动 CKafka 客户端进行消费。

```
./kafka-console-consumer.sh --bootstrap-server xx.xx.xx.xx:xxxx --topic flume_test --
from-beginning --new-consumer
```

> **说明**
> bootstrap-server 填写刚创建的 CKafka 实例的接入地址，topic 填写刚创建的 Topic 名称。

可以看到刚才的消息被消费出来。

```
[root@VM_16_17_centos bin]# ./kafka-console-consumer.sh --bootstrap-server 172.16.16.12:9092 --topic flume_test --from-beginning --new-consumer
ckafka
```

使用 CKafka 作为 Source

步骤1：获取 CKafka 实例接入地址

1. 登录 CKafka 控制台 。
2. 在左侧导航栏选择**实例列表**，单击实例的“ID”，进入实例基本信息页面。
3. 在实例的基本信息页面的**接入方式**模块，可获取实例的接入地址。

步骤2：创建 Topic

1. 在实例基本信息页面，选择顶部 **Topic 管理**页签。
2. 在 Topic 管理页面，单击**新建**，创建一个名为 flume_test 的 Topic。

步骤3：配置 Flume

1. 下载 Apache Flume 工具包并解压 。

2. 编写配置文件 flume-kafka-source.properties，以下是一个简单的 Demo（配置在解压目录的 conf 文件夹下）。若无特殊要求则将自己的实例 IP 与 Topic 替换到配置文件当中即可。此处使用的 sink 为 logger。

```
# 以kafka 作为source 的demo
agentckafka.sources = kafkaSource
agentckafka.channels = memoryChannel
agentckafka.sinks = loggerSink

# 设置source类型,此处设置为kafka
agentckafka.sources.kafkaSource.type= org.apache.flume.source.kafka.KafkaSource
# 此处设置ckafka提供的ip:port
agentckafka.sources.kafkaSource.kafka.bootstrap.servers= 172.16.16.12:9092
# 此处设置需要导出数据的topic，请先再控制台提前创建好topic
agentckafka.sources.kafkaSource.kafka.topics= flume_test
# 设置找不到offset数据时的处理方式
agentckafka.sources.kafkaSource.kafka.consumer.auto.offset.reset= earliest
# 设置source channel
agentckafka.sources.kafkaSource.channels = memoryChannel

# 设置sink
agentckafka.sinks.loggerSink.type = logger
# 设置sink channel
agentckafka.sinks.loggerSink.channel = memoryChannel

# Each channel's type is defined.
agentckafka.channels.memoryChannel.type = memory
agentckafka.channels.memoryChannel.keep-alive = 10

# Other config values specific to each type of channel(sink or source)
# can be defined as well
# In this case, it specifies the capacity of the memory channel
agentckafka.channels.memoryChannel.capacity = 1000
agentckafka.channels.memoryChannel.transactionCapacity =1000
```

实例IP
刚才创建的topic
Souece配置
若有特殊Sink可自行配置
Channel使用默认配置

3. 执行如下命令启动 Flume。

```
./bin/flume-ng agent -n agentckafka -c conf -f conf/flume-kafka-source.properties
```

4. 查看 logger 输出信息（默认路径为 `logs/flume.log`）。

```
Component type: SOURCE, name: kafkaSource started
- Event: { headers:{timestamp=1501136891423, topic=flume_test, partition=0} body: 63 6B 61 66 6B 61
```

ckafka

Kafka Connect 接入 CKafka

最近更新时间：2024-10-14 10:53:42

Kafka Connect 目前支持两种执行模式：standalone 和 distributed。

以 standalone 模式启动 connect

通过以下命令以 standalone 模式启动 connect：

```
bin/connect-standalone.sh config/connect-standalone.properties connector1.properties
[connector2.properties ...]
```

接入 CKafka 与接入开源 Kafka 没有区别，仅需要修改 bootstrap.servers 为申请实例时分配的 IP。

以 distributed 模式启动 connect

通过以下命令以 distributed 模式启动 connect：

```
bin/connect-distributed.sh config/connect-distributed.properties
```

该模式下，kafka connect 会将 offsets、configs 和 task status 信息存储在 kafka topic 中，存储的 topic 在 connect-distributed 中的以下字段配置：

```
config.storage.topic
offset.storage.topic
status.storage.topic
```

这三个 topic 需要手动创建，才能保证创建的属性符合 connect 的要求。

- config.storage.topic 需要保证只有一个 partition，多副本且为 compact 模式。
- offset.storage.topic 需要有多个 partition，多副本且为 compact 模式。
- status.storage.topic 需要有多个 partition，多副本且为 compact 模式。

配置 bootstrap.servers 为申请实例时分配的 IP；
配置 group.id，用于标识 connect 集群，需要与消费者分组区分开来。

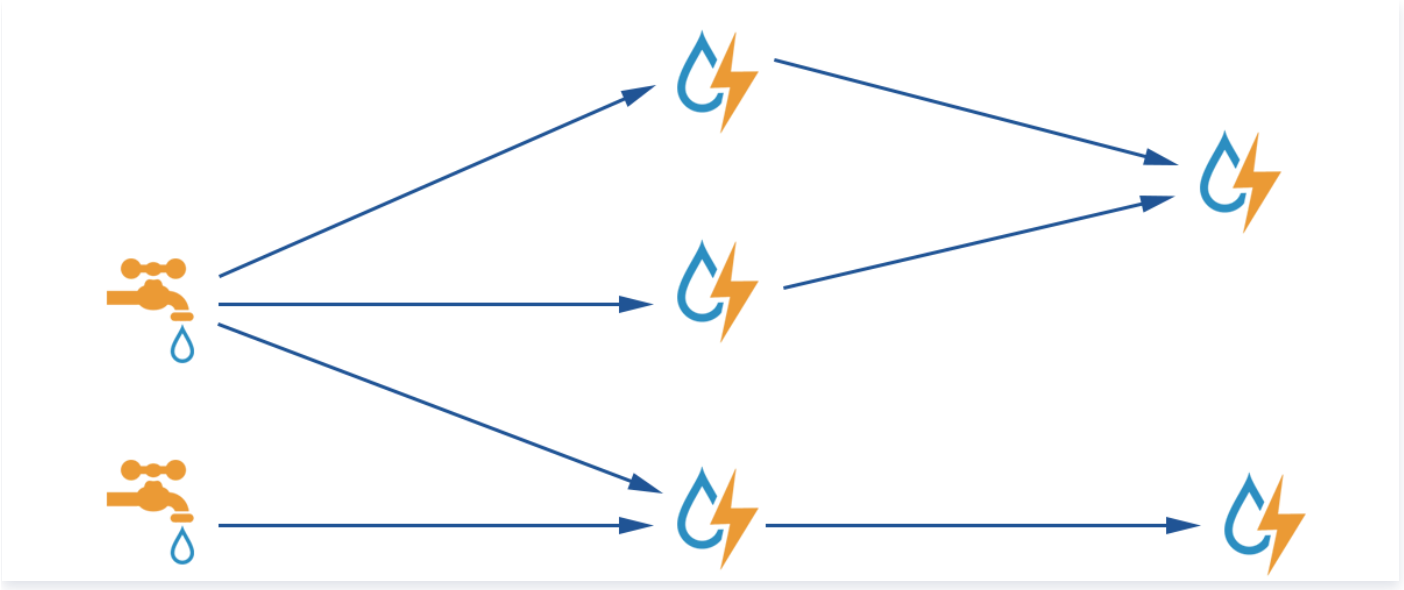
Storm 接入 CKafka

最近更新时间：2024-10-09 14:31:52

Storm 是一个分布式实时计算框架，能够对数据进行流式处理和提供通用性分布式 RPC 调用，可以实现处理事件亚秒级的延迟，适用于对延迟要求比较高的实时数据处理场景。

Storm 工作原理

在 Storm 的集群中有两种节点，控制节点 `Master Node` 和工作节点 `Worker Node` 。`Master Node` 上运行 `Nimbus` 进程，用于资源分配与状态监控。`Worker Node` 上运行 `Supervisor` 进程，监听工作任务，启动 `executor` 执行。整个 Storm 集群依赖 `zookeeper` 负责公共数据存放、集群状态监听、任务分配等功能。
用户提交给 Storm 的数据处理程序称为 `topology` ，它处理的最小消息单位是 `tuple` ，一个任意对象的数组。 `topology` 由 `spout` 和 `bolt` 构成， `spout` 是产生 `tuple` 的源头， `bolt` 可以订阅任意 `spout` 或 `bolt` 发出的 `tuple` 进行处理。



Storm with CKafka

Storm 可以把 CKafka 作为 `spout` ，消费数据进行处理；也可以作为 `bolt` ，存放经过处理后的数据提供给其它组件消费。

测试环境

Centos6.8系统

package	version
maven	3.5.0
storm	2.1.0
ssh	5.3
Java	1.8

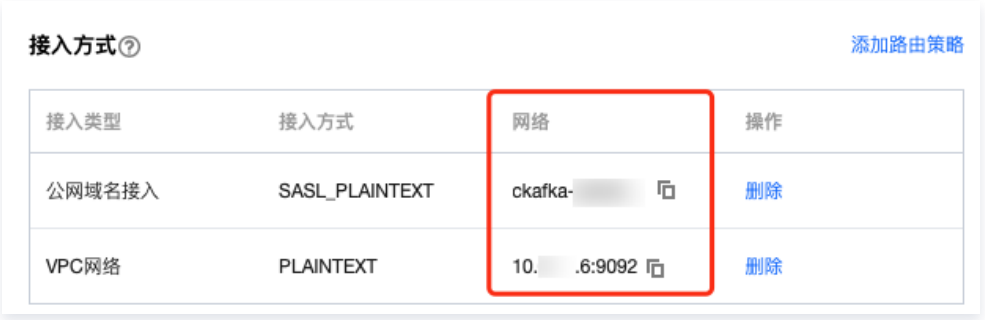
前提条件

- 下载并安装 JDK 8。具体操作，请参见 Download JDK 8 。
- 下载并安装 Storm，参见 Apache Storm downloads 。
- 已 创建 CKafka 实例 。

操作步骤

步骤1：获取 CKafka 实例接入地址

1. 登录 CKafka 控制台 。
2. 在左侧导航栏选择**实例列表**，单击实例的“ID”，进入实例基本信息页面。
3. 在实例的基本信息页面的**接入方式**模块，可获取实例的接入地址。



步骤2：创建 Topic

1. 在实例基本信息页面，选择顶部**Topic管理**页签。
2. 在 Topic 管理页面，单击**新建**，创建一个 Topic。

步骤3：添加 Maven 依赖

pom.xml 配置如下：

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>storm</groupId>
  <artifactId>storm</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>storm</name>
     <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.apache.storm</groupId>
            <artifactId>storm-core</artifactId>
            <version>2.1.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.storm</groupId>
            <artifactId>storm-kafka-client</artifactId>
            <version>2.1.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.kafka</groupId>
```

```
            <artifactId>kafka_2.11</artifactId>
            <version>0.10.2.1</version>
            <exclusions>
                <exclusion>
                    <groupId>org.slf4j</groupId>
                    <artifactId>slf4j-log4j12</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <artifactId>maven-assembly-plugin</artifactId>
                <configuration>
                    <descriptorRefs>
                        <descriptorRef>jar-with-dependencies</descriptorRef>
                    </descriptorRefs>
                    <archive>
                        <manifest>
                            <mainClass>ExclamationTopology</mainClass>
                        </manifest>
                    </archive>
                </configuration>
                <executions>
                    <execution>
                        <id>make-assembly</id>
                        <phase>package</phase>
                        <goals>
                            <goal>single</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

步骤4：生产消息

使用 spout/bolt

topology 代码：

```
//TopologyKafkaProducerSpout.java
import org.apache.storm.Config;
import org.apache.storm.LocalCluster;
import org.apache.storm.StormSubmitter;
import org.apache.storm.kafka.bolt.KafkaBolt;
import org.apache.storm.kafka.bolt.mapper.FieldNameBasedTupleToKafkaMapper;
import org.apache.storm.kafka.bolt.selector.DefaultTopicSelector;
import org.apache.storm.topology.TopologyBuilder;
import org.apache.storm.utils.Utils;

import java.util.Properties;

public class TopologyKafkaProducerSpout {
    //申请的ckafka实例ip:port
    private final static String BOOTSTRAP_SERVERS = "xx.xx.xx.xx:xxxx";
    //指定要将消息写入的topic
    private final static String TOPIC = "storm_test";
    public static void main(String[] args) throws Exception {
        //设置producer属性
        //函数参见：https://kafka.apache.org/0100/javadoc/index.html?
org/apache/kafka/clients/consumer/KafkaConsumer.html
        //属性参见：http://kafka.apache.org/0102/documentation.html
        Properties properties = new Properties();
        properties.put("bootstrap.servers", BOOTSTRAP_SERVERS);
        properties.put("acks", "1");
        properties.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        properties.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

        //创建写入kafka的bolt，默认使用fields("key" "message")作为生产消息的key和message，也可以在
FieldNameBasedTupleToKafkaMapper()中指定
        KafkaBolt kafkaBolt = new KafkaBolt()
                .withProducerProperties(properties)
                .withTopicSelector(new DefaultTopicSelector(TOPIC))
                .withTupleToKafkaMapper(new FieldNameBasedTupleToKafkaMapper());
        TopologyBuilder builder = new TopologyBuilder();
        //一个顺序生成消息的spout类，输出field是sentence
        SerialSentenceSpout spout = new SerialSentenceSpout();
        AddMessageKeyBolt bolt = new AddMessageKeyBolt();
        builder.setSpout("kafka-spout", spout, 1);
        //为tuple加上生产到ckafka所需要的fields
        builder.setBolt("add-key", bolt, 1).shuffleGrouping("kafka-spout");
        //写入ckafka
        builder.setBolt("sendToKafka", kafkaBolt, 8).shuffleGrouping("add-key");

        Config config = new Config();
        if (args != null && args.length > 0) {
            //集群模式，用于打包jar，并放到storm运行
            config.setNumWorkers(1);
```

```
            StormSubmitter.submitTopologyWithProgressBar(args[0], config,
builder.createTopology());
        } else {
            //本地模式
            LocalCluster cluster = new LocalCluster();
            cluster.submitTopology("test", config, builder.createTopology());
            Utils.sleep(10000);
            cluster.killTopology("test");
            cluster.shutdown();
        }

    }
}
```

创建一个顺序生成消息的 spout 类：

```
import org.apache.storm.spout.SpoutOutputCollector;
import org.apache.storm.task.TopologyContext;
import org.apache.storm.topology.OutputFieldsDeclarer;
import org.apache.storm.topology.base.BaseRichSpout;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Values;
import org.apache.storm.utils.Utils;

import java.util.Map;
import java.util.UUID;

public class SerialSentenceSpout extends BaseRichSpout {

    private SpoutOutputCollector spoutOutputCollector;

    @Override
    public void open(Map map, TopologyContext topologyContext, SpoutOutputCollector
spoutOutputCollector) {
        this.spoutOutputCollector = spoutOutputCollector;
    }

    @Override
    public void nextTuple() {
        Utils.sleep(1000);
        //生产一个UUID字符串发送给下一个组件
        spoutOutputCollector.emit(new Values(UUID.randomUUID().toString()));
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer) {
        outputFieldsDeclarer.declare(new Fields("sentence"));
    }
}
```

为 `tuple` 加上 key、message 两个字段，当 key 为 null 时，生产的消息均匀分配到各个 partition，指定了 key 后将按照 key 值 hash 到特定 partition 上：

```
//AddMessageKeyBolt.java
```

```
import org.apache.storm.topology.BasicOutputCollector;
import org.apache.storm.topology.OutputFieldsDeclarer;
import org.apache.storm.topology.base.BaseBasicBolt;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Tuple;
import org.apache.storm.tuple.Values;

public class AddMessageKeyBolt extends BaseBasicBolt {

    @Override
    public void execute(Tuple tuple, BasicOutputCollector basicOutputCollector) {
        //取出第一个filed值
        String messae = tuple.getString(0);
//        System.out.println(messae);
        //发送给下一个组件
        basicOutputCollector.emit(new Values(null, messae));
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer) {
        //创建发送给下一个组件的schema
        outputFieldsDeclarer.declare(new Fields("key", "message"));
    }
}
```

使用 trident

使用 trident 类生成 topology：

```
//TopologyKafkaProducerTrident.java
import org.apache.storm.Config;
import org.apache.storm.LocalCluster;
import org.apache.storm.StormSubmitter;
import org.apache.storm.kafka.trident.TridentKafkaStateFactory;
import org.apache.storm.kafka.trident.TridentKafkaStateUpdater;
import org.apache.storm.kafka.trident.mapper.FieldNameBasedTupleToKafkaMapper;
import org.apache.storm.kafka.trident.selector.DefaultTopicSelector;
import org.apache.storm.trident.TridentTopology;
import org.apache.storm.trident.operation.BaseFunction;
import org.apache.storm.trident.operation.TridentCollector;
import org.apache.storm.trident.tuple.TridentTuple;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Values;
import org.apache.storm.utils.Utils;

import java.util.Properties;

public class TopologyKafkaProducerTrident {
    //申请的ckafka实例ip:port
    private final static String BOOTSTRAP_SERVERS = "xx.xx.xx.xx:xxxx";
    //指定要将消息写入的topic
    private final static String TOPIC = "storm_test";
    public static void main(String[] args) throws Exception {
        //设置producer属性
```

```
        //函数参见: https://kafka.apache.org/0100/javadoc/index.html?
org/apache/kafka/clients/consumer/KafkaConsumer.html
        //属性参见: http://kafka.apache.org/0102/documentation.html
        Properties properties = new Properties();
        properties.put("bootstrap.servers", BOOTSTRAP_SERVERS);
        properties.put("acks", "1");
        properties.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        properties.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        //设置Trident
        TridentKafkaStateFactory stateFactory = new TridentKafkaStateFactory()
                .withProducerProperties(properties)
                .withKafkaTopicSelector(new DefaultTopicSelector(TOPIC))
                //设置使用fields("key", "value")作为消息写入  不像FieldNameBasedTupleToKafkaMapper有
默认值
                .withTridentTupleToKafkaMapper(new FieldNameBasedTupleToKafkaMapper("key",
"value"));
        TridentTopology builder = new TridentTopology();
        //一个批量产生句子的spout,输出field为sentence
        builder.newStream("kafka-spout", new TridentSerialSentenceSpout(5))
                .each(new Fields("sentence"), new AddMessageKey(), new Fields("key", "value"))
                .partitionPersist(stateFactory, new Fields("key", "value"), new
TridentKafkaStateUpdater(), new Fields());

        Config config = new Config();
        if (args != null && args.length > 0) {
            //集群模式，用于打包jar，并放到storm运行
            config.setNumWorkers(1);
            StormSubmitter.submitTopologyWithProgressBar(args[0], config, builder.build());
        } else {
            //本地模式
            LocalCluster cluster = new LocalCluster();
            cluster.submitTopology("test", config, builder.build());
            Utils.sleep(10000);
            cluster.killTopology("test");
            cluster.shutdown();
        }

    }

    private static class AddMessageKey extends BaseFunction {

        @Override
        public void execute(TridentTuple tridentTuple, TridentCollector tridentCollector) {
            //取出第一个filed值
            String messae = tridentTuple.getString(0);
            //System.out.println(messae);
            //发送给下一个组件
            //tridentCollector.emit(new Values(Integer.toString(messae.hashCode()), messae));
            tridentCollector.emit(new Values(null, messae));
        }
    }
}
```

创建一个批量生成消息的 spout 类：

```
//TridentSerialSentenceSpout.java
import org.apache.storm.Config;
import org.apache.storm.task.TopologyContext;
import org.apache.storm.trident.operation.TridentCollector;
import org.apache.storm.trident.spout.IBatchSpout;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Values;
import org.apache.storm.utils.Utils;

import java.util.Map;
import java.util.UUID;

public class TridentSerialSentenceSpout implements IBatchSpout {

    private final int batchCount;

    public TridentSerialSentenceSpout(int batchCount) {
        this.batchCount = batchCount;
    }

    @Override
    public void open(Map map, TopologyContext topologyContext) {

    }

    @Override
    public void emitBatch(long l, TridentCollector tridentCollector) {
        Utils.sleep(1000);
        for(int i = 0; i < batchCount; i++){
            tridentCollector.emit(new Values(UUID.randomUUID().toString()));
        }
    }

    @Override
    public void ack(long l) {

    }

    @Override
    public void close() {

    }

    @Override
    public Map<String, Object> getComponentConfiguration() {
        Config conf = new Config();
        conf.setMaxTaskParallelism(1);
        return conf;
    }

    @Override
    public Fields getOutputFields() {
        return new Fields("sentence");
```

```
    }
}
```

步骤5：消费消息

使用 spout/bolt

```
//TopologyKafkaConsumerSpout.java
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.storm.Config;
import org.apache.storm.LocalCluster;
import org.apache.storm.StormSubmitter;
import org.apache.storm.kafka.spout.*;
import org.apache.storm.task.OutputCollector;
import org.apache.storm.task.TopologyContext;
import org.apache.storm.topology.OutputFieldsDeclarer;
import org.apache.storm.topology.TopologyBuilder;
import org.apache.storm.topology.base.BaseRichBolt;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Tuple;
import org.apache.storm.tuple.Values;
import org.apache.storm.utils.Utils;

import java.util.HashMap;
import java.util.Map;

import static org.apache.storm.kafka.spout.FirstPollOffsetStrategy.LATEST;

public class TopologyKafkaConsumerSpout {
    //申请的ckafka实例ip:port
    private final static String BOOTSTRAP_SERVERS = "xx.xx.xx.xx:xxxx";
    //指定要将消息写入的topic
    private final static String TOPIC = "storm_test";

    public static void main(String[] args) throws Exception {
        //设置重试策略
        KafkaSpoutRetryService kafkaSpoutRetryService = new KafkaSpoutRetryExponentialBackoff(
                KafkaSpoutRetryExponentialBackoff.TimeInterval.microSeconds(500),
                KafkaSpoutRetryExponentialBackoff.TimeInterval.milliSeconds(2),
                Integer.MAX_VALUE,
                KafkaSpoutRetryExponentialBackoff.TimeInterval.seconds(10)
        );
        ByTopicRecordTranslator<String, String> trans = new ByTopicRecordTranslator<>(
                (r) -> new Values(r.topic(), r.partition(), r.offset(), r.key(), r.value()),
                new Fields("topic", "partition", "offset", "key", "value"));
        //设置consumer参数
        //函数参见
http://storm.apache.org/releases/1.1.0/javadocs/org/apache/storm/kafka/spout/KafkaSpoutConfig.Bu
ilder.html
        //参数参见http://kafka.apache.org/0102/documentation.html
        KafkaSpoutConfig spoutConfig = KafkaSpoutConfig.builder(BOOTSTRAP_SERVERS, TOPIC)
                .setProp(new HashMap<String, Object>(){{
                    put(ConsumerConfig.GROUP_ID_CONFIG, "test-group1"); //设置group
                    put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, "50000"); //设置session超时
                    put(ConsumerConfig.REQUEST_TIMEOUT_MS_CONFIG, "60000"); //设置请求超时
```

```
                }})
                .setOffsetCommitPeriodMs(10_000) //设置自动确认时间
                .setFirstPollOffsetStrategy(LATEST) //设置拉取最新消息
                .setRetry(kafkaSpoutRetryService)
                .setRecordTranslator(trans)
                .build();

        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("kafka-spout", new KafkaSpout(spoutConfig), 1);
        builder.setBolt("bolt", new BaseRichBolt(){
            private OutputCollector outputCollector;
            @Override
            public void declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer) {

            }

            @Override
            public void prepare(Map map, TopologyContext topologyContext, OutputCollector
outputCollector) {
                this.outputCollector = outputCollector;
            }

            @Override
            public void execute(Tuple tuple) {
                System.out.println(tuple.getStringByField("value"));
                outputCollector.ack(tuple);
            }
        }, 1).shuffleGrouping("kafka-spout");

        Config config = new Config();
        config.setMaxSpoutPending(20);
        if (args != null && args.length > 0) {
            config.setNumWorkers(3);
            StormSubmitter.submitTopologyWithProgressBar(args[0], config,
builder.createTopology());
        }
        else {
            LocalCluster cluster = new LocalCluster();
            cluster.submitTopology("test", config, builder.createTopology());
            Utils.sleep(20000);
            cluster.killTopology("test");
            cluster.shutdown();
        }
    }
}
```

使用 trident

```
//TopologyKafkaConsumerTrident.java
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.storm.Config;
import org.apache.storm.LocalCluster;
import org.apache.storm.StormSubmitter;
import org.apache.storm.generated.StormTopology;
import org.apache.storm.kafka.spout.ByTopicRecordTranslator;
```

```
import org.apache.storm.kafka.spout.trident.KafkaTridentSpoutConfig;
import org.apache.storm.kafka.spout.trident.KafkaTridentSpoutOpaque;
import org.apache.storm.trident.Stream;
import org.apache.storm.trident.TridentTopology;
import org.apache.storm.trident.operation.BaseFunction;
import org.apache.storm.trident.operation.TridentCollector;
import org.apache.storm.trident.tuple.TridentTuple;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Values;
import org.apache.storm.utils.Utils;

import java.util.HashMap;

import static org.apache.storm.kafka.spout.FirstPollOffsetStrategy.LATEST;


public class TopologyKafkaConsumerTrident {
    //申请的ckafka实例ip:port
    private final static String BOOTSTRAP_SERVERS = "xx.xx.xx.xx:xxxx";
    //指定要将消息写入的topic
    private final static String TOPIC = "storm_test";

    public static void main(String[] args) throws Exception {
        ByTopicRecordTranslator<String, String> trans = new ByTopicRecordTranslator<>(
                (r) -> new Values(r.topic(), r.partition(), r.offset(), r.key(), r.value()),
                new Fields("topic", "partition", "offset", "key", "value"));
        //设置consumer参数
        //函数参见
http://storm.apache.org/releases/1.1.0/javadocs/org/apache/storm/kafka/spout/KafkaSpoutConfig.Bu
ilder.html
        //参数参见http://kafka.apache.org/0102/documentation.html
        KafkaTridentSpoutConfig spoutConfig = KafkaTridentSpoutConfig.builder(BOOTSTRAP_SERVERS,
TOPIC)
                .setProp(new HashMap<String, Object>(){{
                    put(ConsumerConfig.GROUP_ID_CONFIG, "test-group1"); //设置group
                    put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true"); //设置自动确认
                    put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, "50000"); //设置session超时
                    put(ConsumerConfig.REQUEST_TIMEOUT_MS_CONFIG, "60000"); //设置请求超时
                }})
                .setFirstPollOffsetStrategy(LATEST) //设置拉取最新消息
                .setRecordTranslator(trans)
                .build();

        TridentTopology builder = new TridentTopology();
//      Stream spoutStream = builder.newStream("spout", new
KafkaTridentSpoutTransactional(spoutConfig)); //事务型
        Stream spoutStream = builder.newStream("spout", new
KafkaTridentSpoutOpaque(spoutConfig));
        spoutStream.each(spoutStream.getOutputFields(), new BaseFunction(){
            @Override
            public void execute(TridentTuple tridentTuple, TridentCollector tridentCollector) {
                System.out.println(tridentTuple.getStringByField("value"));
                tridentCollector.emit(new Values(tridentTuple.getStringByField("value")));
            }
        }, new Fields("message"));
```

```
        Config conf = new Config();
        conf.setMaxSpoutPending(20);conf.setNumWorkers(1);
        if (args != null && args.length > 0) {
            conf.setNumWorkers(3);
            StormSubmitter.submitTopologyWithProgressBar(args[0], conf, builder.build());
        }
        else {
            StormTopology stormTopology = builder.build();
            LocalCluster cluster = new LocalCluster();
            cluster.submitTopology("test", conf, stormTopology);
            Utils.sleep(10000);
            cluster.killTopology("test");
            cluster.shutdown();stormTopology.clear();
        }
    }
}
```

步骤6：提交 Storm

使用 mvn package 编译后，可以提交到本地集群进行 debug 测试，也可以提交到正式集群进行运行。

```
storm jar your_jar_name.jar topology_name
```

```
storm jar your_jar_name.jar topology_name tast_name
```

Logstash 接入 CKafka

最近更新时间：2024-10-10 19:53:33

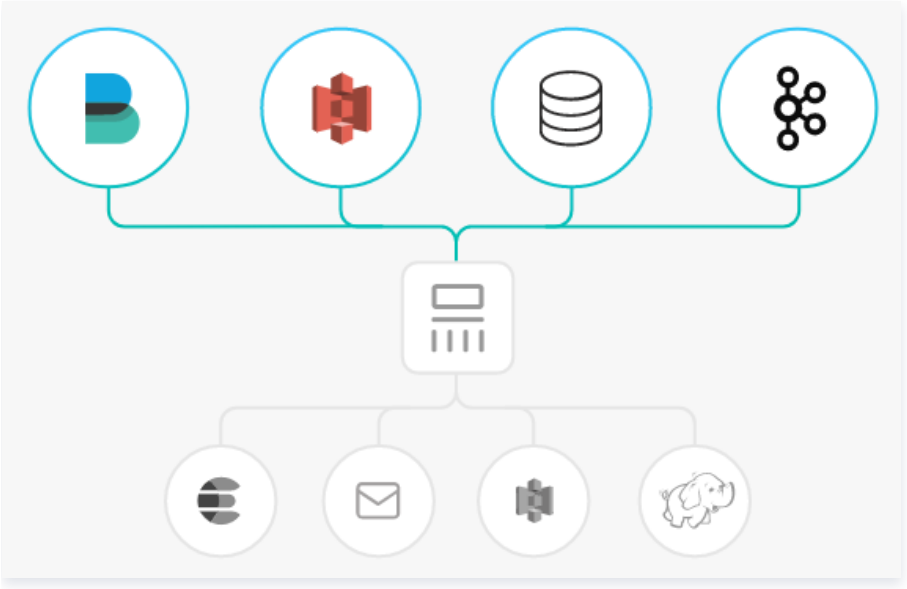
Logstash 是一个开源的日志处理工具，可以从多个源头收集数据、过滤收集的数据并对数据进行存储作为其他用途。
Logstash 灵活性强，拥有强大的语法分析功能，插件丰富，支持多种输入和输出源。Logstash 作为水平可伸缩的数据管道，与 Elasticsearch 和 Kibana 配合，在日志收集检索方面功能强大。

Logstash 工作原理

Logstash 数据处理可以分为三个阶段：inputs → filters → outputs。

1. inputs：产生数据来源，例如文件、syslog、redis 和 beats 此类来源。
2. filters：修改过滤数据， 在 Logstash 数据管道中属于中间环节，可以根据条件去对事件进行更改。一些常见的过滤器包括：grok、mutate、drop 和 clone 等。
3. outputs：将数据传输到其他地方，一个事件可以传输到多个 outputs，当传输完成后这个事件就结束。Elasticsearch 就是最常见的 outputs。

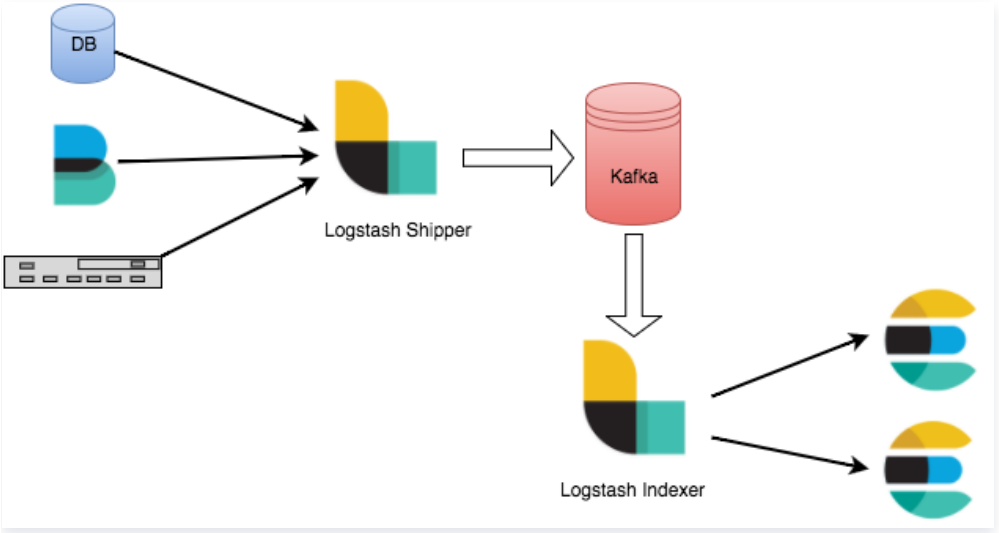
同时 Logstash 支持编码解码，可以在 inputs 和 outputs 端指定格式。



Logstash 接入 Kafka 的优势

- 可以异步处理数据：防止突发流量。
- 解耦：当 Elasticsearch 异常的时候不会影响上游工作。

> ⚠ **注意**
> Logstash 过滤消耗资源，如果部署在生产 server 上会影响其性能。



操作步骤

准备工作

- 下载并安装 Logstash，参见 Download Logstash 。
- 下载并安装 JDK 8，参见 Download JDK 8 。
- 已 创建 CKafka 实例 。

步骤1：获取 CKafka 实例接入地址

1. 登录 CKafka 控制台 。
2. 在左侧导航栏选择**实例列表**，单击实例的“ID”，进入实例基本信息页面。
3. 在实例的基本信息页面的**接入方式**模块，可获取实例的接入地址。

步骤2：创建 Topic

1. 在实例基本信息页面，选择顶部**Topic管理**页签。
2. 在 Topic 管理页面，单击**新建**，创建一个名为 logstash_test 的 Topic。

步骤3：接入 CKafka

说明

您可以单击以下页签，查看 CKafka 作为 inputs 或者 outputs 接入的具体步骤。

作为 inputs 接入

1. 执行 `bin/logstash-plugin list`，查看已经支持的插件是否含有 logstash-input-kafka。

```
[root@VM_16_17_centos bin]# ./logstash-plugin list|grep kafka
logstash-input-kafka
logstash-output-kafka
```

2. 在 .bin/ 目录下编写配置文件 input.conf。
此处将标准输出作为数据终点，将 Kafka 作为数据来源。

```
input {
    kafka {
        bootstrap_servers => "xx.xx.xx.xx:xxxx" // ckafka 实例接入地址
        group_id => "logstash_group"  // ckafka groupid 名称
        topics => ["logstash_test"] // ckafka topic 名称
        consumer_threads => 3 // 消费线程数，一般与 ckafka 分区数一致
        auto_offset_reset => "earliest"
    }
}
output {
    stdout{codec=>rubydebug}
}
```

3. 执行以下命令启动 Logstash，进行消息消费。

```
./logstash -f input.conf
```

返回结果如下：

```
[root@VM_16_17_centos bin]# ./logstash -f input.conf
ERROR StatusLogger No log4j2 configuration file found. Using default configuration: logging only errors to the console.
Sending Logstash's logs to /data/ryan/logstash-5.5.2/logs which is now configured via log4j2.properties
[2017-09-06T18:07:41,926][INFO ][logstash.pipeline        ] Starting pipeline {"id"=>"main", "pipeline.workers"=>4, "pipe
[2017-09-06T18:07:41,943][INFO ][logstash.pipeline        ] Pipeline main started
[2017-09-06T18:07:41,999][INFO ][logstash.agent           ] Successfully started Logstash API endpoint {:port=>9600}
{
    "@timestamp" => 2017-09-06T10:07:42.256Z,
      "@version" => "1",
       "message" => "2017-09-06T09:24:23.039Z localhost ckafka"
}
{
    "@timestamp" => 2017-09-06T10:07:42.256Z,
      "@version" => "1",
       "message" => "2017-09-06T09:23:28.343Z localhost logstash"
}
{
    "@timestamp" => 2017-09-06T10:07:42.256Z,
      "@version" => "1",
       "message" => "2017-09-06T09:23:15.848Z localhost test"
}
```

可以看到刚才 Topic 中的数据被消费出来。

作为 outputs 接入

1. 执行 `bin/logstash-plugin list` ，查看已经支持的插件是否含有 logstash-output-kafka。

```
[root@VM_16_17_centos bin]# ./logstash-plugin list|grep kafka
logstash-input-kafka
logstash-output-kafka
```

2. 在.bin/目录下编写配置文件 output.conf。此处将标准输入作为数据来源，将 Kafka 作为数据目的地。

```
input {
    stdin{}
}output {
    kafka {
         bootstrap_servers => "xx.xx.xx.xx:xxxx"  // ckafka 实例接入地址
         topic_id => "logstash_test" // ckafka topic 名称
         }
}
```

3. 执行如下命令启动 Logstash，向创建的 Topic 发送消息。

```
./logstash -f output.conf
```

```
[root@VM_16_17_centos bin]# ./logstash -f output.conf  <-- 启动命令
ERROR StatusLogger No log4j2 configuration file found. Using default configuration: logging only errors to the console.
Sending Logstash's logs to /data/ryan/logstash-5.5.2/logs which is now configured via log4j2.properties
log4j:WARN No appenders could be found for logger (org.apache.kafka.clients.producer.ProducerConfig).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
[2017-09-06T17:22:56,185][INFO ][logstash.pipeline        ] Starting pipeline {"id"=>"main", "pipeline.workers"=>4, "pip
[2017-09-06T17:22:56,202][INFO ][logstash.pipeline        ] Pipeline main started
The stdin plugin is now waiting for input:
[2017-09-06T17:22:56,239][INFO ][logstash.agent           ] Successfully started Logstash API endpoint {:port=>9600}
test
logstash  <-- 生产消息
ckafka
```

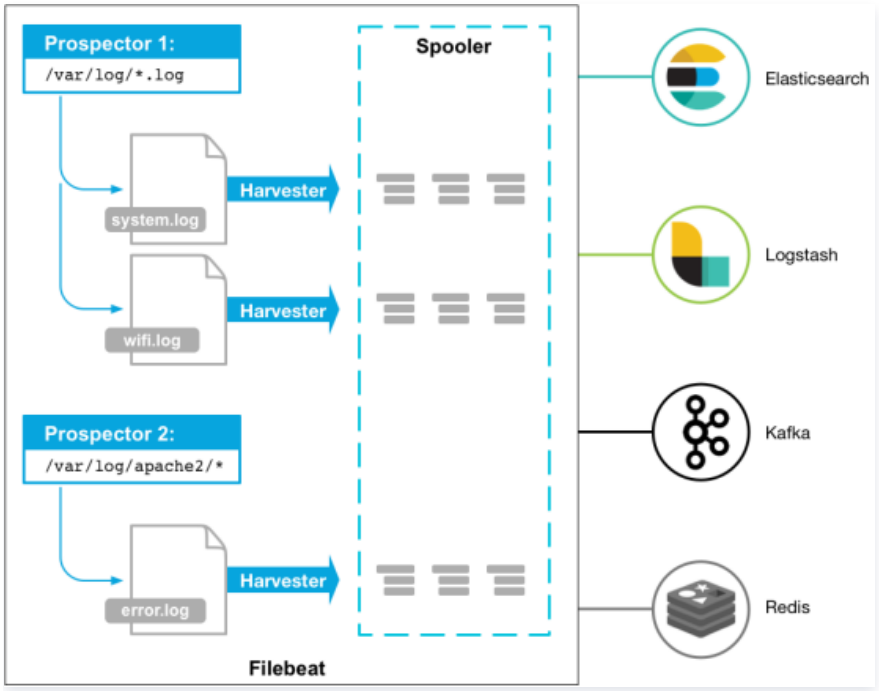
4. 启动 CKafka 消费者，检验上一步的生产数据。

```
[root@VM_16_17_centos bin]# ./kafka-console-consumer.sh --bootstrap-server 172.16.16.12:9092 --topic logstash_test --from-beginning --new-consumer
2017-09-06T09:23:28.343Z localhost logstash
2017-09-06T09:24:23.039Z localhost ckafka
2017-09-06T09:23:15.848Z localhost test
```

Filebeat 接入 CKafka

最近更新时间：2024-10-10 15:56:21

Beats 平台 集合了多种单一用途数据采集器。这些采集器安装后可用作轻量型代理，从成百上千或成千上万台机器向目标发送采集数据。



Beats 有多种采集器，您可以根据自身的需求下载对应的采集器。本文以 Filebeat（轻量型日志采集器）为例，向您介绍 Filebeat 接入 CKafka 的操作方法，及接入后常见问题的解决方法。

前提条件

- 下载并安装 Filebeat（参见 Download Filebeat ）
- 下载并安装JDK 8（参见 Download JDK 8 ）
- 已 创建 CKafka 实例

操作步骤

步骤1：获取 CKafka 实例接入地址

1. 登录 CKafka 控制台 。
2. 在左侧导航栏选择**实例列表**，单击实例的“ID”，进入实例基本信息页面。
3. 在实例的基本信息页面的**接入方式**模块，可获取实例的接入地址。

步骤2：创建 Topic

1. 在实例基本信息页面，选择顶部**Topic管理**页签。
2. 在 Topic 管理页面，单击**新建**，创建一个名为 test 的 Topic。

步骤3：准备配置文件

进入 Filebeat 的安装目录，创建配置监控文件 filebeat.yml。

```
#======= Filebeat7.x之后的版本，将 filebeat.prospectors 修改为 filebeat.inputs 即可 =======
filebeat.prospectors:

- input_type: log

# 此处为监听文件路径
  paths:
    - /var/log/messages

#=======  Outputs =========

#------------------ kafka -------------------------------------
output.kafka:
  version: 0.10.2 # 根据不同 CKafka 实例开源版本配置
  # 设置为CKafka实例的接入地址
  hosts: ["xx.xx.xx.xx:xxxx"]
  # 设置目标topic的名称
  topic: 'test'
  partition.round_robin:
    reachable_only: false

  required_acks: 1
  compression: none
  max_message_bytes: 1000000

  # SASL 需要配置下列信息，如果不需要则下面两个选项可不配置
  username: "yourinstance#yourusername"  #username 需要拼接实例ID和用户名
  password: "yourpassword"
```

步骤4：Filebeat 发送消息

1. 执行如下命令启动客户端。

```
sudo ./filebeat -e -c filebeat.yml
```

2. 为监控文件增加数据（示例为写入监听的 testlog 文件）。

```
echo ckafka1 >> testlog
echo ckafka2 >> testlog
```

```
echo ckafka3 >> testlog
```

3. 开启 Consumer 消费对应的 Topic，获得以下数据。

```
{"@timestamp":"2017-09-29T10:01:27.936Z","beat":
{"hostname":"10.193.9.26","name":"10.193.9.26","version":"5.6.2"},"input_type":"log","message
":"ckafka1","offset":500,"source":"/data/ryanyyang/hcmq/beats/filebeat-5.6.2-linux-
x86_64/testlog","type":"log"}
{"@timestamp":"2017-09-29T10:01:30.936Z","beat":
{"hostname":"10.193.9.26","name":"10.193.9.26","version":"5.6.2"},"input_type":"log","message
":"ckafka2","offset":508,"source":"/data/ryanyyang/hcmq/beats/filebeat-5.6.2-linux-
x86_64/testlog","type":"log"}
{"@timestamp":"2017-09-29T10:01:33.937Z","beat":
{"hostname":"10.193.9.26","name":"10.193.9.26","version":"5.6.2"},"input_type":"log","message
":"ckafka3","offset":516,"source":"/data/ryanyyang/hcmq/beats/filebeat-5.6.2-linux-
x86_64/testlog","type":"log"}
```

SASL/PLAINTEXT 模式

如果您需要进行 SALS/PLAINTEXT 配置，则需要配置用户名与密码。 在 Kafka 配置区域新增加 username 和 password 配置即可。

```
# SASL 需要配置下列信息，如果不需要则下面两个选项可不配置
 username: "yourinstance#yourusername"   //username 需要拼接实例ID和用户名
 password: "yourpassword"
```

常见问题

在 Filebeat 日志（默认路径 `/var/log/filebeat/filebeat` ）中，发现有大量 INFO 日志，例如：

```
2019-03-20T08:55:02.198+0800    INFO    kafka/log.go:53 producer/broker/544 starting up
2019-03-20T08:55:02.198+0800    INFO    kafka/log.go:53 producer/broker/544 state change to
[open] on wp-news-filebeat/4
2019-03-20T08:55:02.198+0800    INFO    kafka/log.go:53 producer/leader/wp-news-filebeat/4
selected broker 544
2019-03-20T08:55:02.198+0800    INFO    kafka/log.go:53 producer/broker/478 state change to
[closing] because EOF
2019-03-20T08:55:02.199+0800    INFO    kafka/log.go:53 Closed connection to broker
bitar1d12:9092
2019-03-20T08:55:02.199+0800    INFO    kafka/log.go:53 producer/leader/wp-news-filebeat/5 state
change to [retrying-3]
2019-03-20T08:55:02.199+0800    INFO    kafka/log.go:53 producer/leader/wp-news-filebeat/4 state
change to [flushing-3]
2019-03-20T08:55:02.199+0800    INFO    kafka/log.go:53 producer/leader/wp-news-filebeat/5
abandoning broker 478
2019-03-20T08:55:02.199+0800    INFO    kafka/log.go:53 producer/leader/wp-news-filebeat/2 state
change to [retrying-2]
2019-03-20T08:55:02.199+0800    INFO    kafka/log.go:53 producer/leader/wp-news-filebeat/2
abandoning broker 541
2019-03-20T08:55:02.199+0800    INFO    kafka/log.go:53 producer/leader/wp-news-filebeat/3 state
change to [retrying-2]
2019-03-20T08:55:02.199+0800    INFO    kafka/log.go:53 producer/broker/478 shut down
```

出现大量 INFO 可能是 Filebeat 版本有问题，因为 Elastic 家族的产品发版速度很频繁，而且不同大版本有很多不兼容。
例如：6.5.x 默认支持 Kafka 的版本是 0.9、0.10、1.1.0、2.0.0，而 5.6.x 默认支持的是0.8.2.0。
您需要检查配置文件中的版本配置：

```
output.kafka:
  version:0.10.2 // 根据不同 CKafka 实例开源版本配置
```

说明与注意

- 发送数据到 CKafka，不能设置压缩 compression.codec。
- 默认不支持 Gzip 压缩格式，如果需要支持，请 提交工单 申请。
  Gzip 压缩对于 CPU 的消耗较高，使用 Gzip 会导致所有的消息都是 InValid 消息。
- 使用 LZ4 压缩方法时，程序不能正常运行，可能的原因如下：
  消息格式错误。CKafka 默认版本为0.10.2，您需要使用 V1 版本的消息格式。
- 不同 Kafka Client 的 SDK 设置方式不同，您可以通过开源社区进行查询（例如 C/C++ Client 的说明），设置消息格式的版本。

日志接入
日志服务 CLS

最近更新时间：2025-03-17 14:34:24

操作场景

您可以将日志主题的数据投递到腾讯云 CKafka，然后用于您的实时流计算场景。如果您没有购买腾讯云 CKafka 实例，可以考虑使用日志服务（Cloud Log Service，CLS）自带的 Kafka 协议消费功能 。

前提条件

- 已 开通腾讯云消息队列（CKafka）。

> **注意：**
> - 如果您的 CKafka 开启了 ACL，请注意 CKafka 的版本限制：CKafka1.X版本，需在1.1.1及以上；CKafka2.X版本，需在2.4.2及以上，其他版本开启 ACL 均会导致投递失败，可通过关闭 ACL 解决投递失败的问题。
> - 当前仅支持实时日志投递，不支持历史日志的投递。
> - 不支持投递至 Ckafka 弹性 Topic。

- 确保当前操作账号拥有开通投递到 CKafka 的权限。详情请参见 CLS 访问策略模板 。

操作步骤

1. 准备和配置 CKafka Topic。在日志主题同地域下，创建一个 CKafka 实例。详情请参见 创建实例 。创建一个 Topic。详情请参见 创建 Topic 。

   您需对该 Topic 的配置做如下修改，其余参数可按需配置。
   - CleanUp.policy：选择 delete，否则会投递失败。
   - max.message.bytes：设置为12MB。

2. 登录 日志服务控制台，进入投递任务管理页面，支持以下来两种方式。
   - 在左侧导航栏中，单击**投递任务**，选择地域、日志集和日志主题，单击**投递到 CKafka** 页签。

   - 在左侧导航栏中，单击**日志主题**，选择需要配置投递到 CKafka 任务的日志主题，进入日志主题管理页面。单击**投递到 CKafka** 页签。

3. 单击右侧的**编辑**，开启投递到 CKafka 开关，选择相应的 CKafka 实例以及对应的 Topic，开始投递。您可以选择以原始内容或者 JSON 投递日志。
   - 以原始内容投递。**原始内容投递配置项说明如下：**

配置项	解释说明	规则	是否必填
目标 Ckafka Topic 归属	**当前主账号** 投递 CLS 日志至当前主账号的 CKafka Topic。 **其他主账号** 投递 CLS 日志至其他主账号的 CKafka，例如 A 账号在 CLS 将日志投递至 B 账号的 Ckafka Topic，需要 B 账号在 CAM(访问管理)侧配置访问角色，配置完成之后，由 A 账号将**角色 ARN 和外部 ID** 填写到 CLS 控制台，方可进行跨账号投递。配置角色的步骤如下： 1. 新建角色。账号 B 登录 CAM 角色管理页面 。 1.1 创建访问策略，策略名称例如：**cross_shipper**，策略语法参考如下： **说明：** 示例中的授权按照最小权限的原则，resource 配置为仅可以投递至广州地域的 CKafka：ckafka-12abcde3，请	当前主账号/其他主账号	否

> 您按照实际情况进行授权。

```
{
  "statement": [
      {
          "action": [
              "cam:GetRole"
          ],
          "effect": "allow",
          "resource": [
              "*"
          ]
      },
      {
          "action": [
              "ckafka:ListRoute",
              "ckafka:AddRoute",

"ckafka:DescribeInstanceAttributes",

"ckafka:AuthorizeToken",
              "ckafka:CreateToken",

"ckafka:DescribeTopicAttributes"
          ],
          "effect": "allow",
          "resource": [
              "qcs::ckafka:ap-
guangzhou:uin/100001234567:ckafkaId/ck
afka-12abcde3"
          ]//resource 配置为仅可以投递
至广州地域的 CKafka：ckafka-12abcde3，请
您按照实际情况进行授权
      }
  ],
  "version": "2.0"
}
```

1.2 新建角色，选择**腾讯云账户**角色载体，云账号类型选择**其他主账号**，然后输入 A 账号的 ID，例如100012345678，勾选**开启校验**并配置外部 ID，例如：**Hello123**。

1.3 配置角色策略，配置角色的访问策略，选择第一步配置好的访问策略 cross_shipper(示例)。

1.4 保存该角色，例如：uinA_writeCLS_to_CKafka。

2. 为角色配置载体。在 CAM 角色列表中找到 uinA_writeCLS_to_CKafka(示例)，单击该角色，选中下方的**角色载体 > 管理载体 > 添加产品服务 > 选中日志服务**，然后单击**更新**。

   可以看到当前角色的载体是两个：一个是 A 账号，另一个是 cls.cloud.tencent.com(CLS 日志服务)。

3. A 账号登录 CLS，填入**角色 ARN 和外部 ID**。

   该两项信息需 **B 账号来提供**：

   - B 账号在 CAM 角色列表中找到角色 uinA_writeCLS_to_CKafka(示例)，单击可查看该角色的 RoleArn，例如

	qcs::cam::uin/100001112345:roleName/uinA_writeCLS_to_CKafka。 ○ 在角色载体中可看到外部 ID，例如 Hello123。 **⚠ 注意：** • 填写角色 ARN，外部 ID 时，注意不要输入多余的空格，会导致权限校验失败。 • 跨账号投递会在 A 账号下，产生日志主题的读流量费。		
CKafka 实例	与当前日志主题同地域的 CKafka Topic 作为投递目标。 跨账号投递场景下，需用户手动填写 Ckafka 实例 ID 和 Topic 名称。	列表选择	必填
投递数据格式	选项**原始内容**，投递用户的原始日志。	列表选择	必填
数据压缩格式	不压缩\Snappy\LZ4。	列表选择	必填
投递日志预览	预览您投递的日志数据。	–	–

○ 以 JSON 投递。**JSON 投递配置项说明如下：**

配置项	解释说明	规则	是否必填
CKafka 实例	与当前日志主题同地域的 CKafka Topic 作为投递目标	列表选择	必填
投递数据格式	选项 **JSON**，以 JSON 的数据格式投递日志	列表选择	必填
JSON 格式的转义/不转义	• 转义，将 JSON 第一层节点的值转为 String，如果您的第一层节点的值是 Struct，在下游入库或者计算时，需要提前将该Struct 转为 String，可以选这个选项。示例： ○ 日志原文：{"a":"aa", "b":{"b1":"b1b1", "c1":"c1c1"}} ○ 投递到 CKafka :{"a":"aa","b":"{\"b1\":\"b1b1\", \"c1\":\"c1c1\"}"} • 不转义，不对您的 JSON 结构和层级做修改，日志格式和采集侧保持一致。示例： ○ 日志原文：{"a":"aa", "b":{"b1":"b1b1", "c1":"c1c1"}} ○ 投递到 CKafka：{"a":"aa", "b":{"b1":"b1b1", "c1":"c1c1"}} **⚠ 注意：** 当 JSON 的第一层节点中包含有数值时，投递后会自动转为 **int**、**float**。 日志原文：{"a":123, "b":"123", "c":"-123", "d":"123.45", "e":{"e1":123,"f1":"123"}} 投递到 CKafka：{"a":123,"b":123,"c":-123,"d":123.45,"e":{"e1":123,"f1":"123"}}	列表选择	必填
__TAG__元信息	将__TAG__元信息平铺或者不平铺，请按照您的实际业务场景选择。 示例：__TAG__元信息：{"__TAG__":{"fieldA":200, "fieldB":"text"}} • 平铺：{"__TAG__.fieldA":200,"__TAG__.fieldB":"text"} • 不平铺：{"__TAG__":{"fieldA":200, "fieldB":"text"}}		
数据压缩格式	支持 Snappy\ LZ4。	列表选择	必填

投递日志预览	预览您投递的日志数据。	-	-

4. 单击**确定**，启动投递到 CKafka。

> **注意：**
> 如需在投递至 CKafka 前对日志进行清洗加工过滤，请参见使用 数据加工 操作。

常见问题

提示没有读写 CKafka Topic 的权限，怎么办?

如果您直接使用 API 接口投递数据到 CKafka，可能会存在读写 CKafka Topic 的权限问题。因为，如果您在控制台使用该功能，系统会引导您完成相关授权；如果您直接调用 API 投递，则需要手动授权。具体的排查和解决方案请参见 投递任务角色授权 。

容器服务（TKE）日志

最近更新时间：2025-03-17 14:34:24

操作场景

日志采集功能是容器服务 TKE 为用户提供的集群内日志采集工具，可以将集群内服务或集群节点特定路径文件的日志发送至 腾讯云日志服务 CLS 消息队列 CKafka 版 。日志采集功能适用于需要对 Kubernetes 集群内服务日志进行存储和分析的用户。
日志采集功能需要为每个集群手动开启并配置采集规则。日志采集功能开启后，日志采集 Agent 会在集群内以 DaemonSet 的形式运行，并根据用户通过日志采集规则配置的采集源、CLS 日志主题和日志解析方式，从采集源进行日志采集，将日志内容发送到日志消费端。开启日志采集功能操作步骤可参见 文档 。

前提条件

- 请在开启前保证集群节点上有足够资源。开启日志采集功能会占用您集群的部分资源。
  - 占用 CPU 资源：0.11 - 1.1核，日志量过大时可根据情况自行调大。
  - 占用内存资源：24 - 560MB，日志量过大时可根据情况自行调大。
  - 日志长度限制：单条512K，如超过会截断。
- 若使用日志采集功能，请确认 Kubernetes 集群内节点能够访问日志消费端。且以下日志采集功能仅支持 Kubernetes 1.10 及以上版本集群。

概念

- **日志采集 Agent**：TKE 用于采集日志信息的 Agent，采用 Loglistener，在集群内以 DaemonSet 的方式运行。
- **日志规则**：用户可以使用日志规则指定日志的采集源、日志主题、日志解析方式和配置过滤器。
  - 日志采集 Agent 会监测日志采集规则的变化，变化的规则会在最多 10s 内生效。
  - 多条日志采集规则不会创建多个 DaemonSet，但过多的日志采集规则会使得日志采集 Agent 占用的资源增加。
- **日志源**：包含指定容器标准输出、容器内文件以及节点文件。
  - 在采集容器标准输出日志时，用户可选择所有容器、或指定工作负载和指定 Pod Labels 内的容器服务日志作为日志的采集源。
  - 在采集容器文件路径日志时，用户可指定工作负载或 Pod Labels 内容器的文件路径日志作为采集源。
  - 在采集节点文件路径日志时，用户可设定日志的采集源为节点文件路径日志。
- **消费端**：用户选择日志服务 CKafka 主题作为消费端。

操作步骤

开启日志采集

1. 登录 容器服务控制台 。
2. 选择左侧导航栏中的**集群**，单击目标集群的“ID”进入集群基本信息页面。
3. 在左侧导航栏选择**日志**页签，在业务日志模块右上角开启日志采集。

配置日志规则

1. 在日志采集页面顶部选择业务日志，单击**前往配置**。
2. 在新建日志采集规则页面中配置日志服务消费端，在消费端 > 类型中选择 **Kafka** 。
   支持用户选择写入 CKafka 或用户自建 Kafka ，当选择 CKafka 时，需要填写实例 ID 和实例 Topic ；当选择自建 Kafka 时，需按要求填写 Broker 地址和 Topic。

说明：

- 如果 Kafka 实例与节点不在同一个 VPC 下，会提示创建 Kafka 实例接入点后再进行日志投递。
- 在集群的 daemonSet 资源中，选择 kube-system 命名空间，找到 tke-log-agent pod 下的 kafkalistener 容器，可以查询 kafka 采集器的日志。

支持在高级设置内通过指定 Key 值将日志投递到指定分区，该功能默认不开启，日志随机投放，当开启后，带有同样 Key 值的日志，将投递到相同的分区。支持输入 TimestampKey（默认@timestamp）和指定时间戳格式。

3. 在“新建日志采集规则”页面中，选择采集类型，并配置日志源。目前采集类型支持**容器标准输出**、**容器文件路径**和**节点文件路径**。

容器标准输出

选择**容器标准输出**采集类型，并根据需求配置日志源。该类型日志源支持一次选择多个 Namespace 的工作负载。

容器文件路径

选择**容器文件路径**采集类型，并配置日志源。

采集文件路径支持文件路径和通配规则，例如当容器文件路径为 `/opt/logs/*.log`，可以指定采集路径为 `/opt/logs`，文件名为 `*.log`。

> **注意**
> “容器文件路径” 不能为符号链接或硬链接，否则会导致软链接的实际路径在采集器的容器内不存在，采集日志失败。

节点文件路径

选择**节点文件路径**采集类型。

- 采集路径支持以文件路径和通配规则的方式填写，例如当需要采集所有文件路径形式为 `/opt/logs/service1/*.log`，`/opt/logs/service2/*.log`，可以指定采集路径的文件夹为 `/opt/logs/service*`，文件名为 `*.log`。
- 您可根据实际需求自定义添加 Key-Value 形式的 metadata，metadata 将会添加到日志记录中。

> **注意：**
> - “节点文件路径” 不能为软链接或硬链接，否则会导致软链接的实际路径在采集器不存在，采集日志失败。
> - 一个节点日志文件只能被一个日志主题采集。

> **说明：**
> - 请确保 Pod 的 label 配置正确，即 Pod 的 label，而非工作负载的 label。您可以按照以下步骤查询：
>
> 1.1 登录 容器服务控制台，选择左侧导航栏中的集群。
>
> 1.2 在集群管理页面，选择集群 ID，进入集群的基本信息页面。
>
> 1.3 在工作负载中，选择具体的工作负载名称，进入工作负载详情页。
>
> 1.4 在 YAML 页签中，查看 template 下的 labels。如下图所示：

- 对于“容器的标准输出”及“容器内文件”（不包含“节点文件路径”即 hostPath 挂载），除了原始的日志内容，还会带上容器或 kubernetes 相关的元数据（例如：产生日志的容器 ID）一起上报到 CLS，方便用户查看日志时追溯来源或根据容器标识、特征（例如：容器名、labels）进行检索。

  容器或 kubernetes 相关的元数据请参考下方表格：

字段名	含义
container_id	日志所属的容器 ID。
container_name	日志所属的容器名称。
image_name	日志所属容器的镜像名称 IP。
namespace	日志所属 pod 的 namespace。
pod_uid	日志所属 pod 的 UID。
pod_name	日志所属 pod 的名字。
pod_lable_{label name}	日志所属 pod 的 label（例如一个 pod 带有两个 label：app=nginx，env=prod，则在上传的日志会附带两个 metedata：pod_label_app:nginx，pod_label_env:prod）。

4. 配置采集策略。您可以选择全量或者增量。
   - 全量：全量采集指从日志文件的开头开始采集。
   - 增量：增量采集指从距离文件末尾1M处开始采集（若日志文件小于1M，则等同于全量采集）。

> ⚠ **注意：**
> - 全量采集场景：采集规则生效延时：为确保采集到 Pod 启动过程中的所有关键日志，请配置全量采集策略。全量采集策略能够在采集规则完全生效前，无差别地收集所有日志数据，有效避免日志丢失现象。
> - 增量采集场景：采集路径为云硬盘 CBS：采集路径为云硬盘时建议配置增量采集策略，以免 Pod 重建产生重采。
> - 异常场景：采集路径为文件存储 CFS：配置全量采集会造成重采，且多个 Pod 可能同时上报相同日志，造成日志服务重复计费，建议日志源不要使用 CFS。

5. 单击**下一步**，选择日志解析方式。
   - 编码模式：支持 UTF-8 和 GBK。
   - 提取模式：支持多种类型的提取模式，详情如下：

解析模式	说明	相关文档
单行全文	一条日志仅包含一行的内容，以换行符 \n 作为一条日志的结束标记，每条日志将被解析为键值为 CONTENT 的一行完全字符串，开启索引后可通过全文检索搜索日志内容。日志时间以采集时间为准。	单行全文格式

多行全文	指一条完整的日志跨占多行，采用首行正则的方式进行匹配，当某行日志匹配上预先设置的正则表达式，就认为是一条日志的开头，而下一个行首出现作为该条日志的结束标识符，也会设置一个默认的键值 CONTENT，日志时间以采集时间为准。支持自动生成正则表达式。	多行全文格式
单行 - 完全正则	指将一条完整日志按正则方式提取多个 key-value 的日志解析模式，您需先输入日志样例，其次输入自定义正则表达式，系统将根据正则表达式里的捕获组提取对应的 key-value。支持自动生成正则表达式。	单行 - 完全正则格式
多行 - 完全正则	适用于日志文本中一条完整的日志数据跨占多行（例如 Java 程序日志），可按正则表达式提取为多个 key-value 键值的日志解析模式，您需先输入日志样例，其次输入自定义正则表达式，系统将根据正则表达式里的捕获组提取对应的 key-value。支持自动生成正则表达式。	多行-完全正则格式
JSON	JSON 格式日志会自动提取首层的 key 作为对应字段名，首层的 value 作为对应的字段值，以该方式将整条日志进行结构化处理，每条完整的日志以换行符 \n 为结束标识符。	JSON 格式
分隔符	指一条日志数据可以根据指定的分隔符将整条日志进行结构化处理，每条完整的日志以换行符 \n 为结束标识符。日志服务在进行分隔符格式日志处理时，您需要为每个分开的字段定义唯一的 key，无效字段即无需采集的字段可填空，不支持所有字段均为空。	分隔符格式
组合解析	当您的日志结构太过复杂，涉及多种解析模式，单种解析模式（如 Nginx 模式、完整正则模式、JSON 模式等）无法满足日志解析需求时，您可以使用 Loglistener 组合解析格式解析日志，此模式支持用户在控制台输入代码（json 格式）用来定义日志解析的流水线逻辑。您可添加一个或多个 Loglistener 插件处理配置，Loglistener 会根据处理配置顺序逐一执行。	组合解析格式

- 过滤器：LogListener 仅采集符合过滤器规则的日志，Key 支持完全匹配，过滤规则支持正则匹配，如仅采集 ErrorCode = 404 的日志。您可以根据需求开启过滤器并配置规则。

> **说明：**
> 一个日志主题目前仅支持一个采集配置，请保证选用该日志主题的所有容器的日志都可以接受采用所选的日志解析方式。若在同一日志主题下新建了不同的采集配置，旧的采集配置会被覆盖。

6. 单击**完成**，完成投递到 CKafka 的日志采集规则创建。

云防火墙日志

最近更新时间：2025-03-17 14:34:24

云防火墙日志日志分析可查看基于登录账号的云防火墙在过去6个月所存储的全部流量日志详情，同时日志分析支持基于检索语句的日志检索与查询，并提供报表与统计分析服务。
通过日志投递功能，您可以将云防火墙日志自动投递到指定的 CKafka 实例中。接下来将为您介绍，如何使用日志分析中的日志投递功能。

背景信息

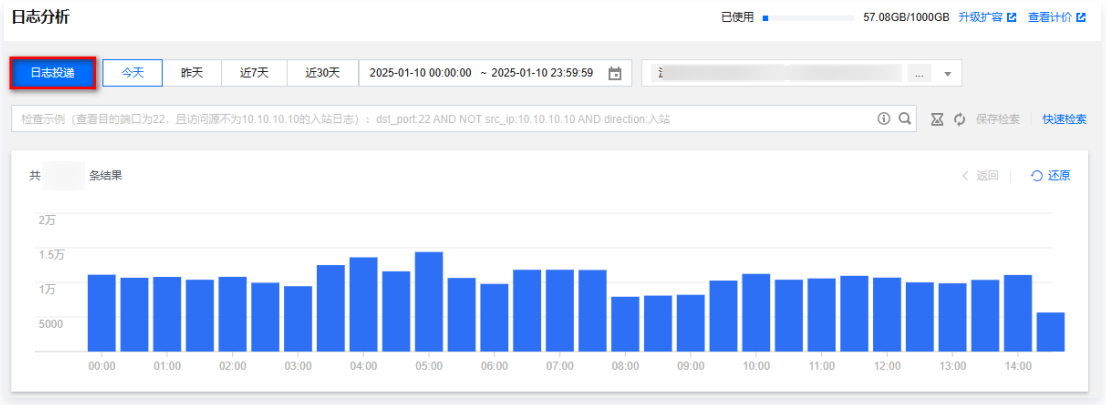
- 日志投递功能可以按照不同的云防火墙日志类型，分别投递到指定的 CKafka topic 中。
- 日志投递功能支持两种网络接入方式，分别是：公网域名接入和支撑环境接入。
  - 公网域名接入是通过公网进行日志投递的。
  - 支撑环境接入是通过腾讯云内网进行日志投递，性能更高。

前提条件

- 需已 购买腾讯云消息队列 CKafka 实例 和 云防火墙日志分析，按照云防火墙带宽来配置 Kafka 实例的带宽规格。
- 根据 消息队列 Ckafka 版文档 指引，开通“公网域名接入”或联系 腾讯云客服 开启“支撑环境接入”白名单。
- 仅支持使用一个消息队列 Ckafka 版账号进行日志投递。

配置步骤

1. 登录 云防火墙控制台，在左侧导航栏中，单击**日志分析**。
2. 在日志分析页面左上角，单击**日志投递**，默认进入投递至 Kafka 页面。



3. 在投递至 Kafka 页面，进行初始配置。

   3.1 选择网络接入方式分为：公网域名接入及支撑环境接入。

   - 方式1：选择公网域名接入，选择消息队列实例和公网域名，并输入所选消息队列实例的用户名和密码。

   - 方式2：选择支撑环境接入，支撑环境接入指您在腾讯云已选购可以与 Ckafka 结合使用的产品，并选择消息队列实例和 IP 端口。

3.2 选择网络接入方式后，可以在日志投递页面绑定 Ckafka topic。

> **说明：**
> 日志投递功能支持多种云防火墙日志类型的投递，不同类型的日志需要投递到不同的 Ckafka topic 中。每个 Ckafka topic 只能被一个云防火墙日志类型所绑定。

3.3 配置完成后，单击确定，将提示配置日志投递成功，表示日志投递已经配置成功。

4. 初始化配置完成后，即可查看日志投递详情。

- 基本信息：展示了 Ckafka 实例的基本信息。

> **注意：**
> 您需要关注“健康状态”字段，当提示“不健康”时，需要单击查看监控，查看 Ckafka 服务是否异常，或者是否配额不足。

- 日志投递开关：用于控制指定的日志类型，启动或停止日志投递任务。
  - 方式1：您可以在各项日志类型的右侧“投递开关”列，通过“开关”按钮单独控制日志投递任务。
  - 方式2：使用批量操作，目前支持全部启动和全部停止两种操作。

- 重新绑定 Ckafka topic：在日志类型的右侧的操作栏，单击编辑进行单独配置，可以重新选择指定 Ckafka 实例中，未被其他防火墙日志类型绑定的 Ckafka topic。

  > **说明：**
  > 每个 Ckafka topic 只能被一个云防火墙日志类型所绑定。

- 查看监控：在日志类型的右侧的操作栏，单击查看监控，会跳转至消息队列 Ckafka 控制台的监控页面，您可以查看网络流量、峰值带宽、消息条数、磁盘占用等情况。
- 重新配置：在日志类型列表上方，单击重新配置，将会允许您重新选择投递的消息队列实例、网络接入方式以及用户名/密码。

  > **注意：**
  > 重新配置，会中断当前的投递进程。

云安全中心

最近更新时间：2025-03-17 14:34:24

功能背景

将接入云安全中心的多款产品日志集中并归一化后通过控制台投递至消息队列，便于存储数据或联合其它系统消费数据，助力挖掘日志数据价值，满足用户日志运维诉求。启用日志投递后，将采集到的日志投递至对应的消息队列。

应用场景

日志存储

根据《中华人民共和国网络安全法》、《信息安全等级保护管理办法》等相关法律法规的规定，企业需要对网络安全事件进行记录和存储，并且日志存储时长不少于6个月。这是为了保障企业的信息安全和网络安全，防止安全事件的发生和滋生。

离线分析

将日志投递至 Kafka 后，企业可以接入其他系统进行离线分析，进一步管控原始日志，协助企业对安全事件进行深入分析和研究，发现安全事件的根本原因和漏洞，提高安全事件的处理能力和水平。

日志投递至 Kafka

在日志分析页面，您可配置云安全中心接入的不同日志类型分别投递到指定 CKafka 实例的不同 Topic 中。

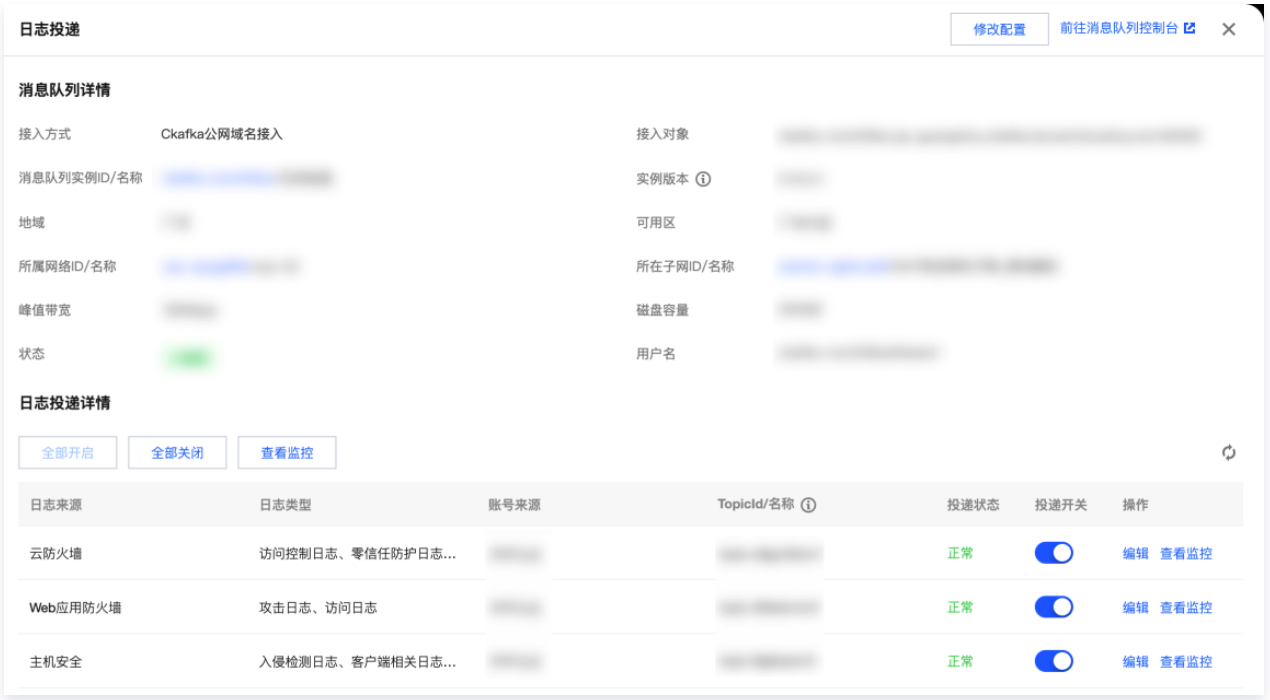
> **前提条件：**
> 为了将日志投递至消息队列，需要先购买云安全中心旗舰版，并将相关产品的日志接入云安全中心。如果需要使用 CKafka 公网域名或 CKafka 支撑环境接入两种网络接入方式之一，需要先前往创建腾讯云消息队列 CKafka 实例。

Ckafka 公网域名接入

1. 登录 云安全中心控制台，在左侧导览中，单击**日志分析**。
2. 在日志分析页面，单击**日志投递 > 投递至 kafka**。
3. 在投递至 kafka页面，云安全中心自动获取账号的腾讯云消息队列 CKafka 实例、已接入云安全中心的日志来源，选择 **CKafka 公网域名接入**，配置相关参数。

参数名称	说明
网络接入方式	CKafka 公网域名接入。
TLS 加密	选择是否开启 TLS 加密。
消息队列所属账号	投递目标所属账号。
消息队列实例	云安全中心自动获取账号的腾讯云消息队列 Ckafka 实例、选择所需消息队列实例。
公网域名接入	选择所需公网域名。
用户名	请输入所选消息队列实例的用户名。
密码	请输入所选消息队列实例的密码。
日志来源	支持选择主机安全、云防火墙、Web 应用防火墙、云安全中心、DDoS 防护、SaaS 化堡垒机、操作审计、网络蜜罐的日志。
日志类型	根据所选的日志来源不同则日志类型也有所不同。
Topic ID/名称	选择所需 Topic。
操作	• 新增：单击**新增日志投递配置**，支持新增多个日志来源。 • 删除：单击目标日志操作列的**删除**，经过二次确认后，支持删除该日志来源对应日志类型的日志投递任务。 • 编辑：如非首次配置日志投递，则支持在日志投递页面，单击**修改配置**，修改相关日志投递。

4. 确认无误后，单击**确定**，即可将采集到的日志投递至对应的消息队列。
5. 在日志投递页面，支持查看同步接入方式、接入对象、消息队列状态、用户名等消息队列详情，以及日志来源、日志类型、账号来源（多账号下）、Topic ID/名称、Topic 投递状态、投递开关等信息，允许修改消息队列、Topic 配置等信息，查看消息队列和各 Topic 状态。

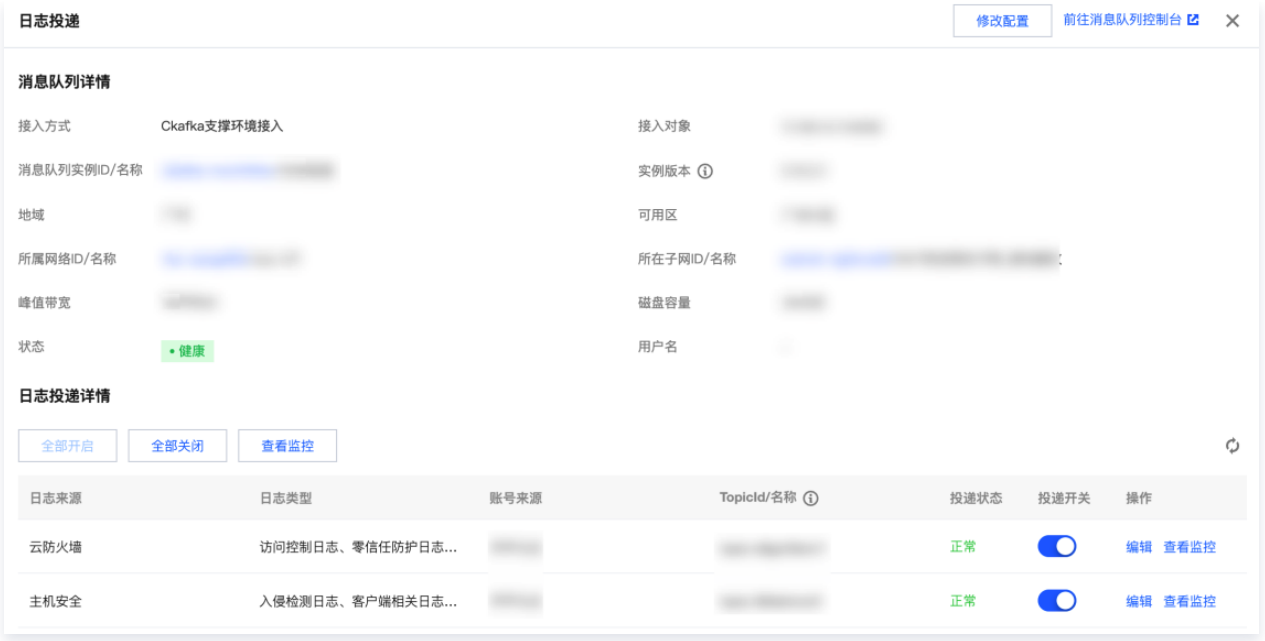


CKafka 支撑环境接入

1. 登录 云安全中心控制台，在左侧导览中，单击**日志分析**。
2. 在日志分析页面，单击**日志投递 > 投递至 kafka**。
3. 在投递至 kafka 页面，云安全中心自动获取账号的腾讯云消息队列 CKafka 实例、已接入云安全中心的日志来源，选择 **CKafka 支撑环境接入**，配置相关参数。

参数名称	说明
网络接入方式	CKafka 支撑环境接入。
TLS 加密	选择是否开启 TLS 加密。
消息队列所属账号	投递目标所属账号。
消息队列实例	云安全中心自动获取账号的腾讯云消息队列 CKafka 实例、选择所需消息队列实例。
支撑环境接入	选择所需支撑环境。
日志来源	支持选择主机安全、云防火墙、Web 应用防火墙、云安全中心、DDoS 防护、SaaS 化堡垒机、操作审计、网络蜜罐的日志。
日志类型	根据所选的日志来源不同则日志类型也有所不同。
Topic ID/名称	选择所需 Topic。
操作	• 新增：单击**新增日志投递配置**，支持新增多个日志来源。 • 删除：单击目标日志操作列的**删除**，经过二次确认后，支持删除该日志来源对应日志类型的日志投递任务。 • 编辑：如非首次配置日志投递，则支持在日志投递页面，单击**修改配置**，修改相关日志投递。

4. 确认无误后，单击**确定**，即可将采集到的日志投递至对应的消息队列。
5. 在日志投递页面，支持查看同步接入方式、接入对象、消息队列状态、用户名等消息队列详情，以及日志来源、日志类型、账号来源（多账号下）、Topic ID/名称、Topic 投递状态、投递开关等信息，允许修改消息队列、Topic 配置等信息，查看消息队列和各 Topic 状态。



其他 Kafka 公网接入

1. 登录 云安全中心控制台，在左侧导览中，单击**日志分析**。
2. 在日志分析页面，单击**日志投递 > 投递至 kafka**。
3. 在投递至 kafka 页面，选择**其他 Kafka 公网接入**，配置相关参数。

参数名称	说明
网络接入方式	其他 Kafka 公网接入。
TLS 加密	选择是否开启 TLS 加密。
公网接入	根据实际需求填写公网信息。
用户名	请输入所选消息队列实例的用户名。
密码	请输入所选消息队列实例的密码。
日志来源	支持选择主机安全、云防火墙、Web 应用防火墙、云安全中心、DDoS 防护、SaaS 化堡垒机、操作审计、网络蜜罐的日志。
日志类型	根据所选的日志来源不同则日志类型也有所不同。
Topic 名称	输入所需 Topic 名称。
操作	• 新增：单击**新增日志投递配置**，支持新增多个日志来源。 • 删除：单击目标日志操作列的**删除**，经过二次确认后，支持删除该日志来源对应日志类型的日志投递任务。

- 编辑：如非首次配置日志投递，则支持在日志投递页面，单击**修改配置**，修改相关日志投递。

4. 确认无误后，单击**确定**，即可将采集到的日志投递至对应的消息队列。
5. 在日志投递页面，支持查看同步接入方式、接入对象、消息队列状态、用户名等消息队列详情，以及日志来源、日志类型、账号来源（多账号下）、Topic 名称、Topic 投递状态、投递开关等信息，并且允许修改消息队列、Topic 配置等信息。

WAF 日志投递

最近更新时间：2025-05-23 10:27:02

功能简介

Web 应用防火墙 日志投递功能用于将日志数据投递到、CKafka（消息队列），助力挖掘日志数据价值，满足用户日志运维诉求。日志投递支持当前 WAF 引擎采集到的全部访问日志字段数据，用户只需要在 WAF 控制台进行简单配置，即可完成访问日志数据准实时投递服务。

说明：

- 如果使用日志投递时出现异常，请 联系我们 进行处理。
- 日志投递支持访问日志的 付费 投递及攻击日志免费投递（企业版及以上实例支持），需分别开启。
- 完成投递到 CKafka 的配置后，还需要参考开启日志投递相关操作，开启攻击日志/访问投递设置。
- 使用日志投递功能和使用日志服务功能不冲突，无论是否开启日志服务都可以开启和使用日志投递功能（建议根据业务需要开启）。

前提条件

- 已购买腾讯云 消息队列 Ckafka 实例，按照实际日志用量来配置 Ckafka 实例的带宽规格。

日志投递至 CKafka

1. 登录 Web 应用防火墙控制台，在左侧导航栏中，选择**访问日志 > 日志投递**。
2. 授权 WAF 投递数据至 CKafka 实例。

   2.1 在日志投递页面的投递至 Ckafka 模块中，单击**立即配置**，唤起授权弹窗。

   2.2 单击**前往授权**后，跳转至 CAM 授权页面。

   2.3 在 CAM 授权页面，单击**同意授权**，对 Web 应用防火墙进行授权 CKafka 投递数据权限。授权过程遇到相关问题，请参考 CAM 管理文档。

2.4 单击同意授权后，可以返回日志投递页面单击**立即配置**，唤起 CKafka 投递配置弹窗。

2.5 日志字段设置：支持自定义勾选保存 BOT 信息、请求内容 Request Body、自定义 Headers。

> **说明：**
> 日志字段设置支持基于全部域名、单个域名进行设置。全部域名及单个域名均配置策略时，针对单个域名配置的策略优先生效。

3. 在 CKafka 投递配置弹窗中，配置相关参数，单击**确定**即可完成配置。
   - 支撑环境：支撑环境接入指您在腾讯云已选购可以与 CKafka 结合使用的产品，并选择消息队列实例和 IP 端口。

> **说明：**
> 支持 SASL PLAINTEXT路由类型投递，选择该类型时需要输入用户名及密码进行验证。

参数名称	参数说明
地域	CKafka 支持的地域，详情请参见 CKafka 地域和可用区 。
消息队列实例	当前地域下运行中的 CKafka 实例。
Topic ID/名称	对应的 Topic ID 信息。

支撑环境接入	支撑网络的路由。

- 公网域名接入：选择公网域名接入，选择消息队列实例和公网域名，并输入所选消息队列实例的用户名和密码。

参数名称	参数说明
地域	CKafka 支持的地域，详情请参见 CKafka 地域和可用区 。
消息队列实例	当前地域下运行中的 CKafka 实例。
Topic ID/名称	对应的 Topic ID 信息。
支撑环境接入	支撑网络的路由。
用户名	SASL 用户名。
密码	SASL 密码。

4. 完成日志投递至 CKafka 后，可将所需域名开启日志投递功能，详情请参见 开启日志投递。

开启日志投递

当完成日志投递至 CKafka 后，需要将所需域名/实例开启日志投递功能。

> **说明：**
> - 攻击日志以实例维度开启投递，仅企业版及以上版本实例支持开启攻击日志投递。
> - 访问日志以域名维度开启投递，不限实例版本，均支持配置开启访问日志投递。

开启攻击日志投递

1. 登录 Web 应用防火墙控制台，在左侧导航栏中，选择**实例管理**。
2. 在实例管理页面，单击**实例名称**，唤起侧边栏。

3. 在实例详情中，单击开启攻击日志投递，即可开启当前实例的攻击日志投递。

开启访问日志投递

1. 登录 Web 应用防火墙控制台，在左侧导航栏中，选择**接入管理 > 域名接入**。
2. 在域名接入页面，选择所需域名，单击**更多 > 日志投递**。

3. 在高级设置窗口中，勾选需要投递的目标，单击**保存**，即可开启当前域名的访问日志投递。

连接器
HTTP 上报
HTTP 协议接入 Kafka

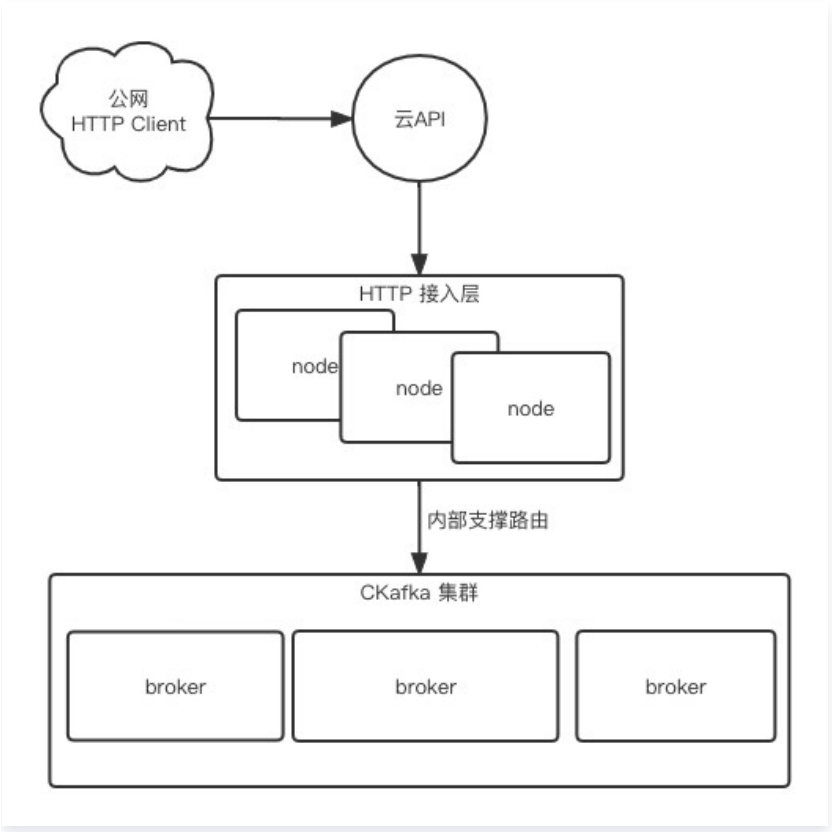
最近更新时间：2024-10-11 12:05:41

操作场景

与任何客户端-服务器应用程序一样，Kafka 通过一组明确定义的 API 提供对其功能的访问，这些 API 通过 Kafka 协议公开，是一种仅限于 Kafka 的 TCP 二进制协议。与 Kafka API 交互的最佳方式是客户端通过使用 Kafka 协议，Apache Kafka 项目仅正式支持 Java 的客户端库，但除此之外，Confluent 还正式支持 C/C++，C#，Go 和 Python 的客户端库。
一些编程语言缺乏官方支持的 Kafka 生产级客户端，而 HTTP 是一种广泛可用、普遍支持的协议，CKafka 连接器数据接入通过 HTTP 协议公开消息发送 API，以便于简化客户端复杂的配置。
本文介绍 CKafka 连接器的 HTTP 数据接入功能中的发送消息结合实际应用场景提供建议。

技术架构

HTTP 数据接入层开启后，公网的 HTTP 客户端可通过云 API 直接向 CKafka 所在的实例发送消息。示意图如下：



前提条件

已创建好数据目标 Topic。

操作步骤

创建数据接入任务

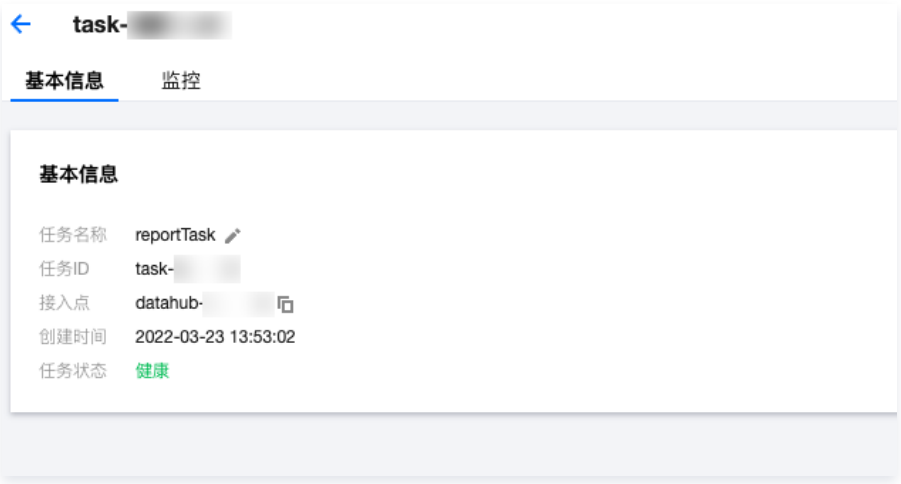
具体操作请参见 HTTP 主动上报 。

使用 SDK 发送消息

1. 参见 SDK中心：Java 在 Java 项目通过 Maven、Gradle 等方式引入数据上报 SDK。以下是配置项目的 pom.xml 文件。

```
<dependency>
    <groupId>com.tencentcloudapi</groupId>
    <artifactId>tencentcloud-sdk-java</artifactId>
    <version>3.1.430</version>
</dependency>
```

2. 单击 数据接入 的任务详情， 复制接入点信息到 SDK 中使用，用于写入数据。



3. 示例中通过 generateMsgFromUserAccess 将所有要发送的消息组装起来，复制接入点信息。

```
    List<BatchContent> batchContentList = generateMsgFromUserAccess(userId);
    // 其中 ap-xxx 为对应的云API地域简称
    CkafkaClient client = new CkafkaClient(
    new Credential("yourSecretId", "yourSecretKey"), "ap-xxx");


SendMessageRequest messageRequest = new SendMessageRequest();
// 数据接入任务接入点ID
messageRequest.setDataHubId("datahub-lzxxxxx6");
messageRequest.setMessage(batchContentList.toArray(BatchContent[]::new));

try {
  SendMessageResponse sendMessageResponse = client.SendMessage(messageRequest);
  String[] messageId = sendMessageResponse.getMessageId();
  for (String s : messageId) {
     LOGGER.info(s)
  }
} catch (TencentCloudSDKException e) {
  LOGGER.error(e.getMessage());
}
```

4. 通过 HTTP 接入层发送消息的返回值示例如下。

```
{
    "Response": {
        "MessageId": [
```

```
            "datahub-lxxxxxx6:topicDev:4:2:1648185961342:1648185961398"
        ],
        "RequestId": "3fq3na5r-xxxx-xxxx-xxxx-b2fiv0se7ded"
    }
}
```

5. 其中 MessageId 内容由一系列发送至 CKafka 实例后返回的元数据组成。如下分别为：

```
"[datahubId]:[topic名称]:[所在的topic分区数]:[所在分区的offset]:[HTTP接入层收到消息的时间]:[消息发送至Kafka的时间]"
```

查询消息

通过 CKafka 控制台 查询 HTTP 接入层发送的消息，详细操作参见 消息查询 。如下图，示例 topic 名称为 topicDev 的4号分区查询2号位点消息。

任务暂停

当您发现数据接入任务影响了正常业务时，可以暂停数据接入。

1. 在 任务列表 页面，单击目标任务的操作栏的**更多 > 暂停**，可暂停任务。
2. 出现右上角的提示，则任务暂停成功。

3. 此时通过 HTTP 接入层发送消息得到示例如下：

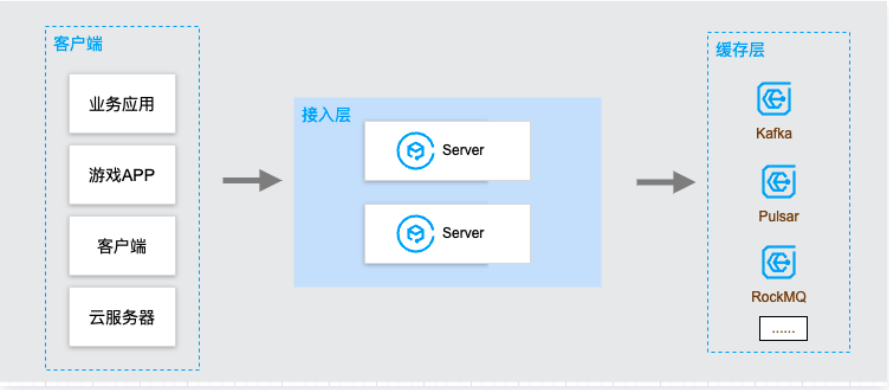
```
{
 "Response": {
     "Error": {
         "Code": "FailedOperation",
         "Message": "task status suspended [datahub-lxxxxxx6]"
     },
     "RequestId": "5f737a5b-xxxx-xxxx-xxxxx-b2fb703e7ded"
 }
}
```

统一数据上报

最近更新时间：2024-10-10 12:05:41

操作场景

CKafka 连接器是腾讯云上的数据接入和处理平台，一站式提供对数据的接入、处理和分发功能。数据在互联网业务中至关重要，而数据接入上报是整个链路中，介于数据产生和计算、存储、分析的桥梁，简单高效的数据接入是至关重要的。我们在业务上经常有一些数据（业务指标、流程信息、监控信息等）需要上报到后台进行存储、分析、计算、搜索。其处理链路通常如下：



在经典数据上报架构中，通常需要有如下几个步骤：

1. 搭建/购买存储引擎，用来存储上报的数据。
2. 开发部署数据接收的 Server、定义 API、运行服务。
3. 定义客户端和服务端的接口协议、鉴权等信息。
4. 客户端根据协议信息编写相应的代码完成数据上报。

在这四步中，Server 端的工作量是最大的，需要考虑代码逻辑开发、Server 本身扩缩容和稳定性、下游存储的扩缩容和稳定性等，而当数据量大时，Server 端问题会更加明显，需要消耗大量的人力物力来进行维护。但这部分工作通用性很高，DIP 希望可以满足这种场景，提供稳定、弹性、高可靠、高吞吐的数据接入服务。

运行原理

CKafka 连接器提供了 Java、Python、GoLang、PHP、NodeJs、C++、.Net 等各语言的 SDK 来方便客户端进行数据上报。通过3步即可将数据上报到存储引擎中如 Kafka（后续会支持 RocketMQ，Pulsar，RabbitMQ，CMQ 等多种消息队列）。具体步骤如下：

1. 在 CKafka 控制台创建接入点。
2. 通过 SDK 上报数据。
3. 数据查询。

操作步骤

1. 在 CKafka 控制台创建接入点。可参见 HTTP 上报 创建接入点。
2. 通过 SDK 进行数据上报。使用步骤可参见 数据上报 SDK 。
3. 数据查询。当数据上报到CKafka 连接器后，可以实时查询消息内容，详情请参见 消息查询 。

数据赋能

当通过 CKafka 连接器简单快速的完成数据接入后，如何让数据产生价值才是最重要的事情。CKafka 连接器提供的两个核心功能：

数据处理

CKafka 连接器提供了简单的数据 ETL 引擎，提供了可以满足大部分数据清洗需求的处理引擎，对数据进行简单的格式化和处理，以便进行后续的使用。

数据流出

当完成数据处理后，CKafka 连接器可以满足以下多种场景的数据处理需求：

- 实时搜索： 当需要对数据进行搜索时，可以将数据流式实时导出到：Elasticsearch、CLS 等搜索服务进行实时搜索。
- OLAP 分析：让需要对数据进行分析时，可以将数据流式实时导出到 ClickHouse、TDW 等引擎进行分析。
- 持久存储：当需要对数据进行持久存储时，可以将数据流式实时导出到 HDFS、COS 等持久化引擎进行存储。
- 流计算：当业务需要通过自定义代码处理数据时，可以通过标准的 Kafka 协议，使用 FLink、Spark、各语言代码对数据进行处理。

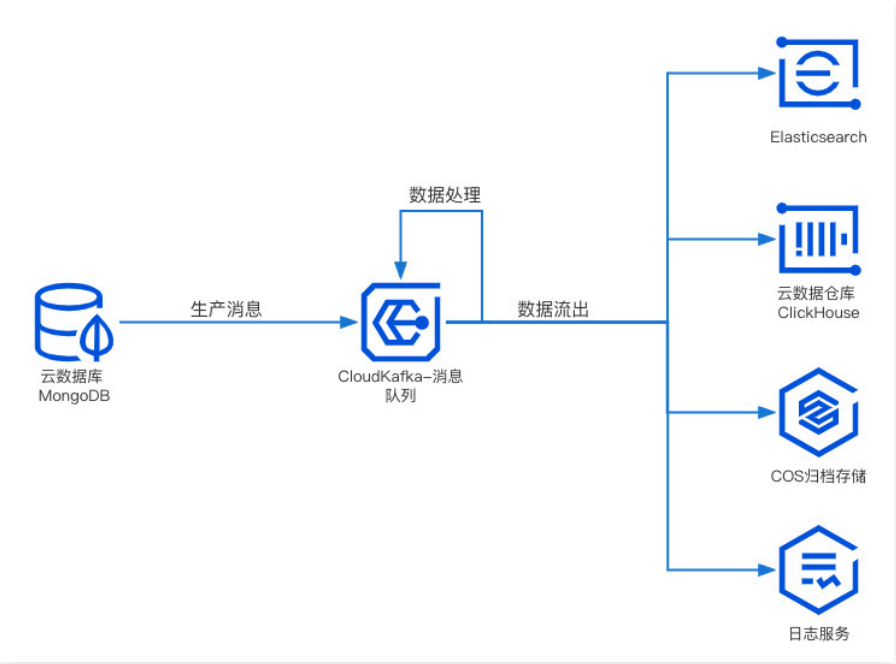
至此，数据经过数据上报、接入、处理、流出四个环节后，简单快速的满足通用的数据上报、分析类需求，用极低的成本让数据体现出价值。

数据库变更订阅查询
Mongo Stream 数据变更记录分析

最近更新时间：2024-10-15 11:19:52

操作场景

MongoDB 内有 Change Stream 作为其追踪变更的解决方案，但为了更好地对变更记录进行搜索，往往需要将变更记录同步到 Elasticsearch、日志服务（CLS）等。



本文以 MongoDB 接入 CKafka 并从 CKafka 流出到 CLS 为例，讲解如何使用 CKafka 连接器数据转储服务实现 Mongo Stream 数据变更记录分析。

运行原理

MongoDB 的数据流入配置项可参见官方 MongoDB Kafka source connector，通过设置不同的配置项，能够对接入的变更记录做相应的数据处理后再写入到 CKafka 实例的 Topic。

前提条件

- 需开启云数据库 MongoDB 服务或使用负载均衡 CLB 监听自建 MongoDB。

并且设置的安全组需要开放 TCP:27017 端口。

- 需开启 CKafka 服务。
- 需开启 CLS 服务。

操作步骤

步骤1：创建数据接入连接

1. 登录 CKafka 控制台 。
2. 在左侧导航栏单击**连接器** > **连接列表**，选择好地域后，单击**新建连接**。
3. 连接类型选择 **MongoDB**，然后单击**下一步**。
4. 填写 MongoDB 连接配置，然后单击**下一步**。

5. 等待连接校验成功后，可以在连接列表看到创建好的连接。

步骤2：创建数据接入任务

1. 在左侧导航栏单击**任务管理** > **任务列表**，选择好地域后，单击**新建任务**。
2. 任务类型选择**数据接入**，接入方式选择 **MongoDB数据订阅**，单击**下一步**。
3. **数据源**选择刚刚新建的 **MongoDB 连接**。`database` 留空表示监听所有数据库的变更。`table` 留空表示监听某个数据库的所有表的变更。`复制存量数据` 可以根据业务需求选择是否打开，然后单击**下一步**。

4. 根据业务需求配置数据目标然后单击**提交**。等待任务显示健康，即表示创建成功。

5. 当 MongoDB 数据发生变更时，可以在目标 Topic 中查看到新增的消息。

- 数据目标为 CKafka 实例的 Topic，可以在侧边栏单击**消息查询**进行查看；
- 数据目标为单独 Topic 时，可以在侧边栏单击 **Topic 列表**，然后单击 Topic 进入详情页，再单击**查看消息**。

步骤3：创建数据流出任务

1. 在左侧导航栏单击**任务管理** > **任务列表**，选择好地域后，单击**新建任务**。
2. 任务类型选择**数据流出**，数据目标选择**日志服务（CLS）**，单击**下一步**。

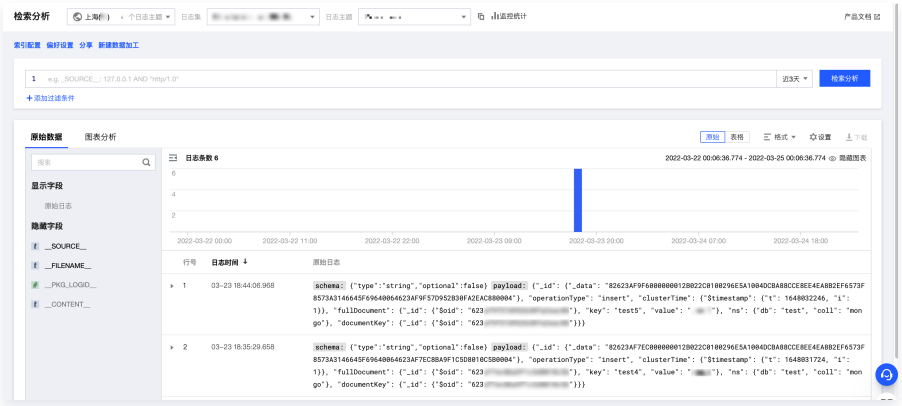
3. 填写任务详情，选取与数据接入任务相同的 CKafka 实例和 Topic，保证在消息生产后能直接进行消费。

4. 单击**提交**，等待任务显示健康，即表示创建成功。

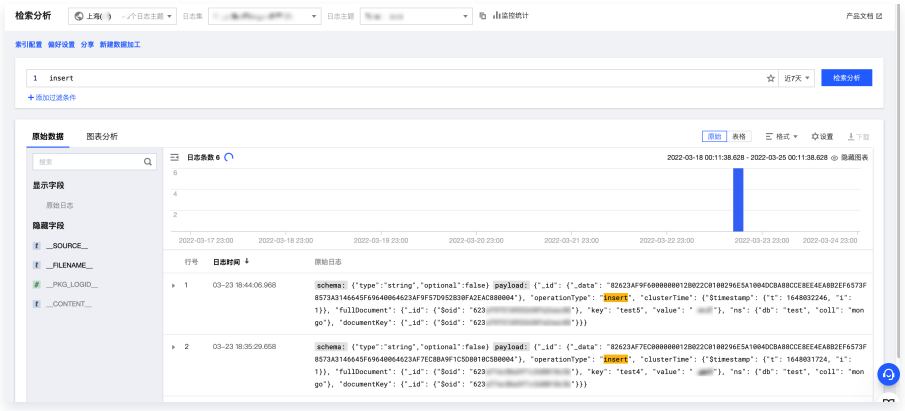
> **说明**
> 当任务在健康的状态时， Topic 有新增的消息写入，会直接被消费到指定的 CLS 日志主题中。

步骤4：查看流出数据

1. 登录 日志服务 控制台。
2. 在左侧导航栏选择**检索分析**，选择流出时填写的日志集与日志主题的“ID”，即可看到 MongoDB 的变更记录。



3. 通过关键字检索等操作，能直观得到所需要的记录。

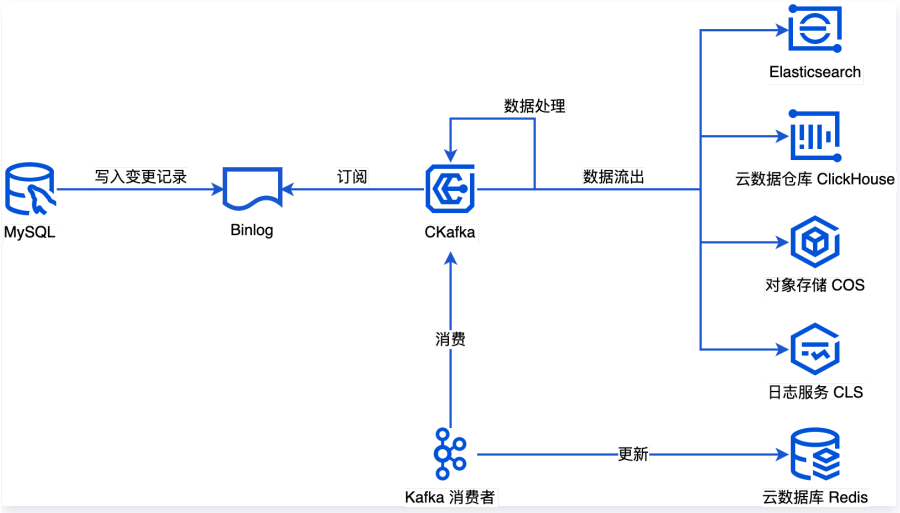


MySQL 数据变更记录分析

最近更新时间：2024-10-14 15:38:57

操作场景

MySQL Binlog 是记录 MySQL 所有数据变动的二进制日志文件，用于 MySQL 主从复制和数据恢复。此外开发者还可以将订阅 MySQL Binlog 应用在**增量索引、缓存一致性、基于数据的任务分发、记录数据变更**等场景。



本文以 MySQL Binlog 接入 CKafka，并从 CKafka 流出到 CLS 为例，讲解如何使用 CKafka 连接器实现 MySQL 数据变更记录分析。

运行原理

CKafka 连接器通过 Binlog 同步组件订阅 MySQL Binlog，然后将数据变更记录转化为 JSON 格式的消息生产到 CKafka 中。

前提条件

- 需要开通**云数据库 MySQL**开启 Binlog，并且根据业务需求修改**参数设置**。

```
binlog_format=ROW
# binlog_row_image 生产环境建议设置为 FULL，记录变更前后所有字段的值。可以快速恢复误操作的数据。
binlog_row_image=FULL
# binlog_rows_query_log_events 设为 ON 时，会记录变更时的 SQL 语句
binlog_rows_query_log_events=ON
```

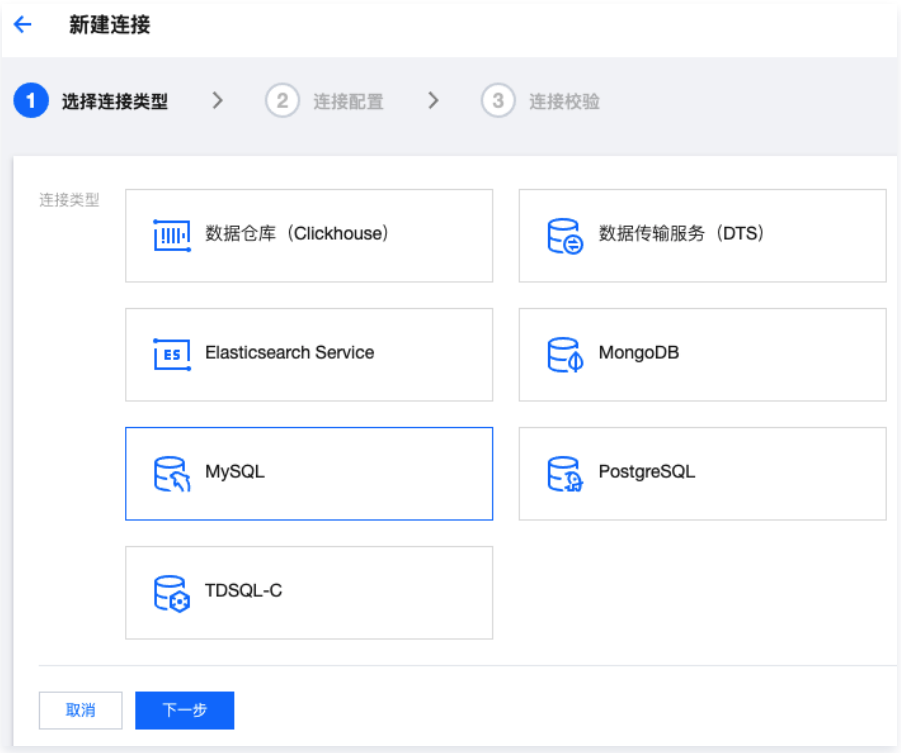
- 需开启 CKafka 服务。
- 需开启 CLS 服务。

操作步骤

步骤1：创建数据接入连接

1. 登录 CKafka 控制台 。
2. 在左侧导航栏单击**连接器** > **连接列表**，选择好地域后，单击**新建连接**。

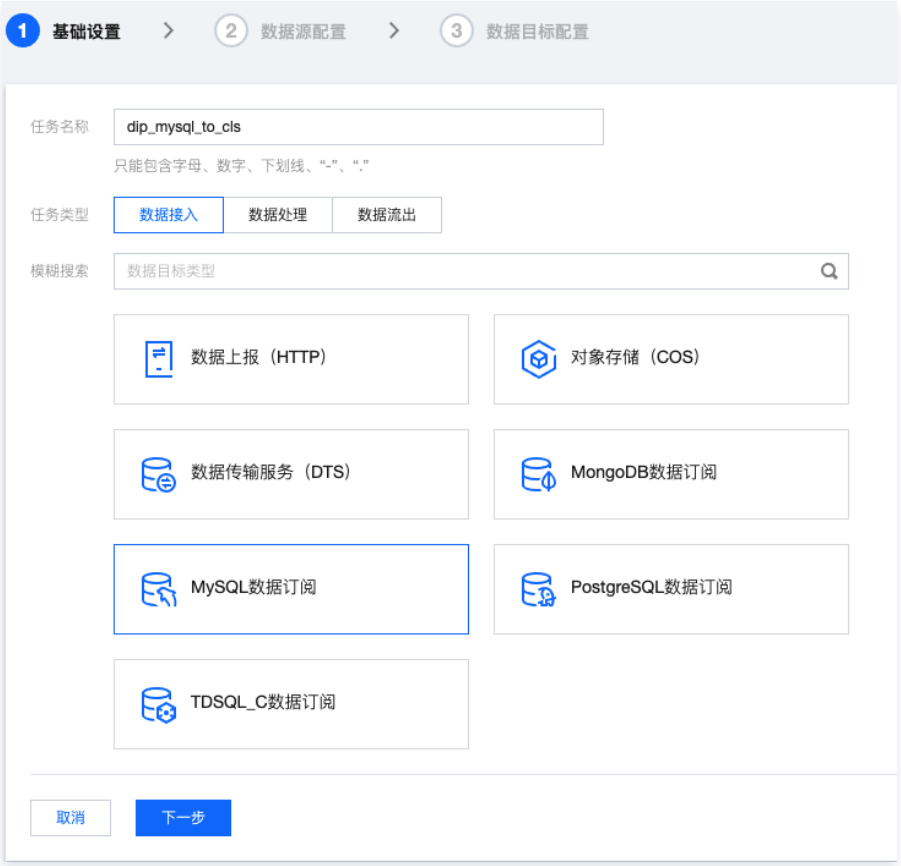
3. 连接类型选择 MySQL，然后单击**下一步**。



4. 选择有 Binlog 权限的用户，填写 MySQL 连接配置后，单击**下一步**等待连接校验完成。

步骤2：创建数据接入任务

1. 在左侧导航栏单击**任务管理** > **任务列表**，选择好地域后，单击**新建任务**。
2. 任务类型选择**数据接入**，接入方式选择 **MySQL 数据订阅**，单击**下一步**。



3. **数据源**选择刚刚新建的 **MySQL 连接**。`database` 留空表示监听所有数据库的变更。`table` 留空表示监听某个数据库的所有表的变更。`复制存量数据` 可以根据业务需求选择是否打开，然后单击**下一步**。

4. 根据业务需求配置数据目标然后点击**提交**。

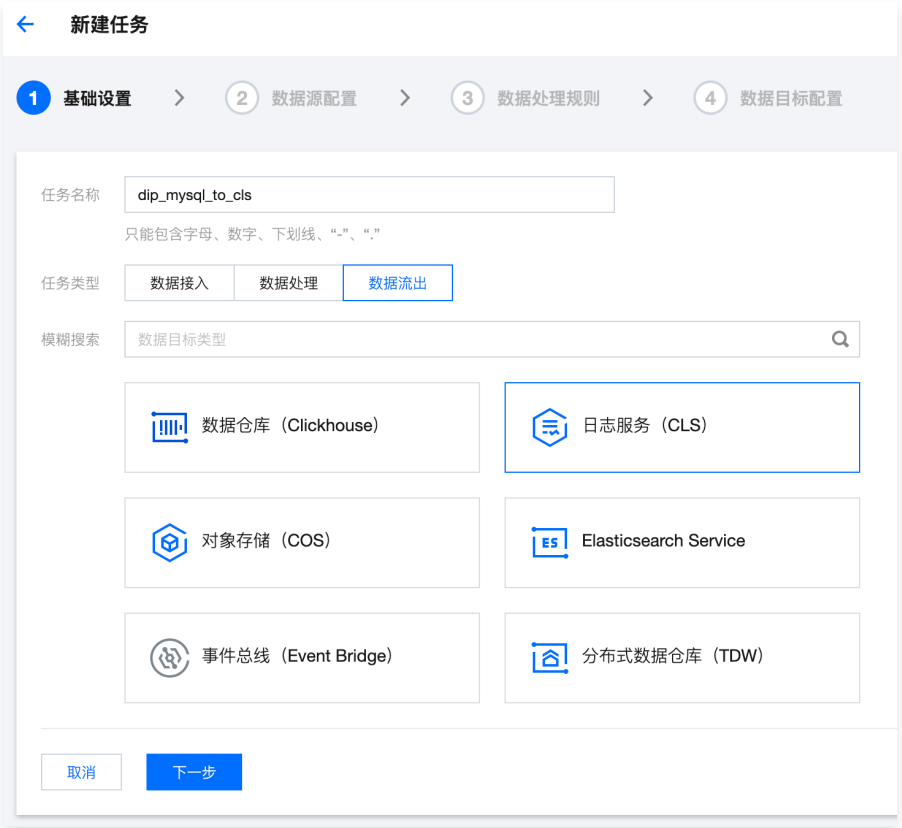
> **注意**
> 为保证 Binlog 事件的全局顺序性，请设置 Topic 分区数请设置为 1 。

5. 当 MySQL 数据发送变化时，可以在目标 Topic 中查看到新增的消息。

- 数据目标为 CKafka 实例的 Topic，可以在侧边栏点击**消息查询**进行查看；
- 数据目标为单独 Topic 时，可以在侧边栏点击 **Topic 列表**，然后点击 Topic 进入详情页，再点击**查看消息**。

步骤3：创建数据流出任务

1. 在左侧导航栏单击**任务管理** > **任务列表**，选择好地域后，单击**新建任务**。
2. 任务类型选择**数据流出**，数据目标选择**日志服务（CLS）**，单击**下一步**。



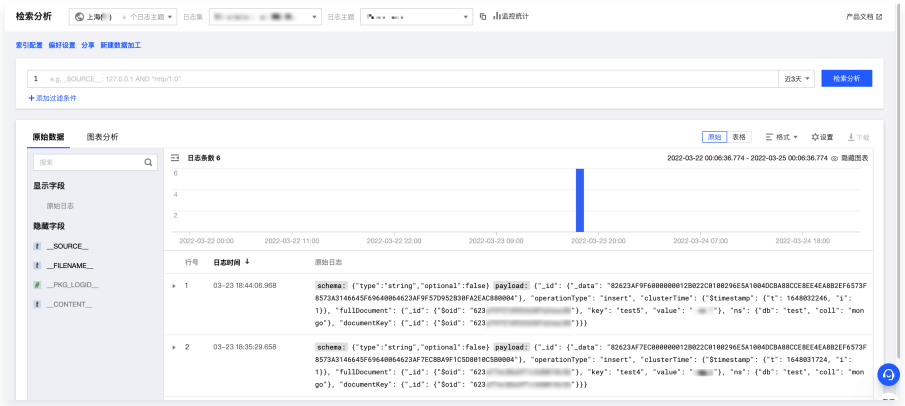
3. 填写任务详情，选取与数据接入任务相同的 CKafka 实例和 Topic，保证在消息生产后能直接进行消费。

4. 配置好数据处理规则和数据目标后，单击**提交**，等待任务显示健康，即表示创建成功。

> **说明**
> 当任务在健康的状态时， Topic 有新增的消息写入，会直接被消费到指定的 CLS 日志主题中。

步骤4：查看流出数据

1. 登录 日志服务 控制台。
2. 在左侧导航栏选择**检索分析**，选择流出时填写的日志集与日志主题的“ID”，即可看到 MySQL 的变更记录。



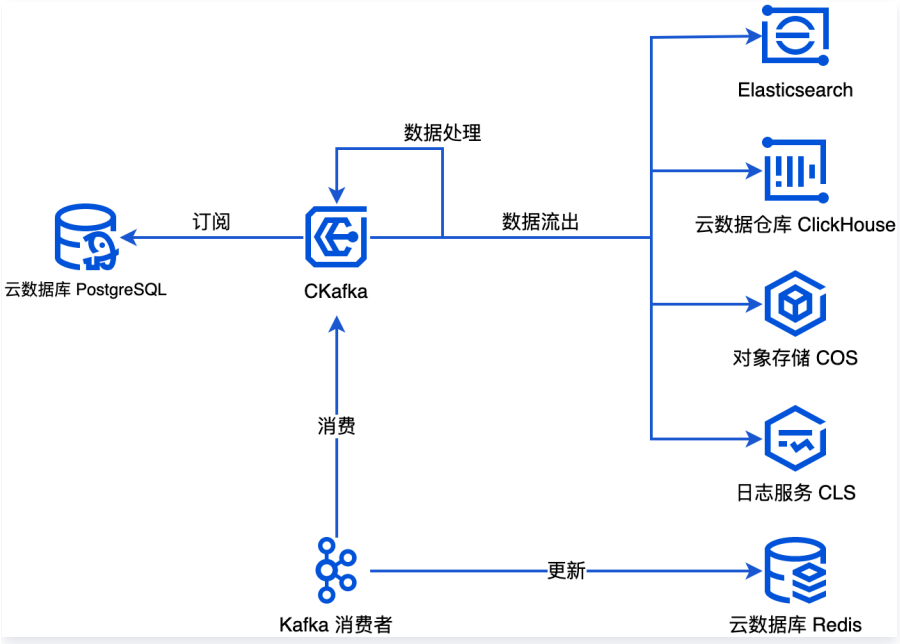
3. 通过关键字检索等操作，能直观得到所需要的记录。

PostgreSQL 数据变更记录分析

最近更新时间：2024-10-15 10:52:04

操作场景

PostgreSQL 通过 WAL（Write-Ahead Logging）预写日志机制保证事务持久性和数据完整性，开发者可以将订阅 WAL 应用在**增量索引、缓存一致性、基于数据的任务分发、记录数据变更**等场景。



本文以 PostgreSQL 接入 CKafka，并从 CKafka 流出到 CLS 为例，讲解如何使用 CKafka 连接器实现 PostgreSQL 数据变更记录分析。

运行原理

CKafka 连接器通过订阅 PostgreSQL WAL，将行级数据变更记录转化为 JSON 格式的消息生产到 CKafka 中。

前提条件

- 需要开通**云数据库 PostgrepSQL**，然后修改以下配置。

```
wal_level=logical
#
# 9.4、9.5、9.6 版本需要根据业务需求设置下面参数
# 10 及以上版本则可以使用默认值
#
max_replication_slots=10
max_wal_senders=10
```

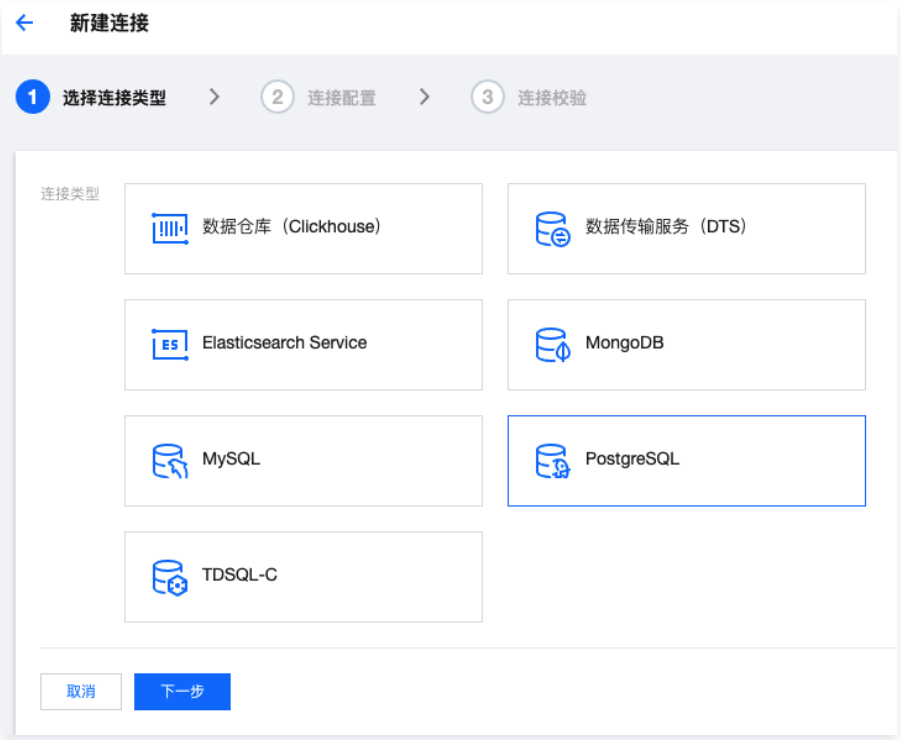
- 需开启 CKafka 服务。
- 需开启 CLS 服务。

操作步骤

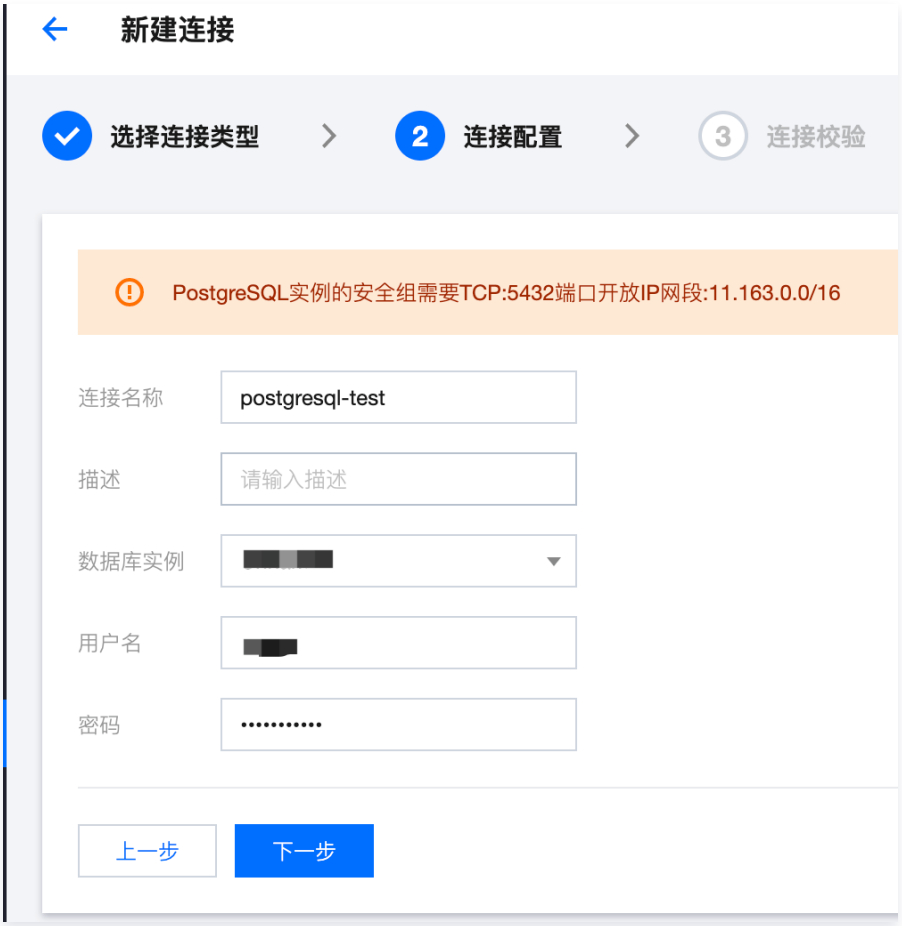
步骤1：创建数据接入连接

1. 登录 CKafka 控制台 。
2. 在左侧导航栏单击**连接器** > **连接列表**，选择好地域后，单击**新建连接**。

3. 连接类型选择 **PostgreSQL**，然后单击**下一步**。



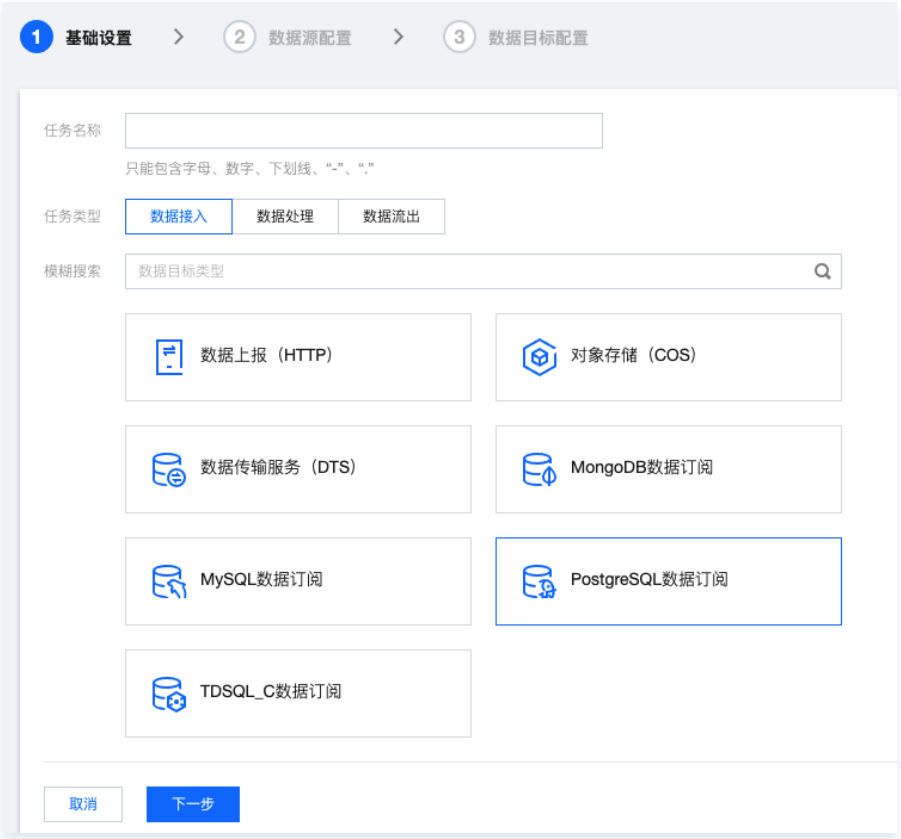
1. 选择有 **REPLICATION** 和 **LOGIN** 权限的用户，填写 PostgreSQL 连接配置，然后单击**下一步**。



步骤2：创建数据接入任务

1. 在左侧导航栏单击**任务管理** > **任务列表**，选择好地域后，单击**新建任务**。

2. 任务类型选择**数据接入**，接入方式选择**PostgreSQL 数据订阅**，单击**下一步**。



3. 数据源选择刚刚新建的 **PostgreSQL 连接**。`database` 留空表示监听所有数据库的变更。`table` 留空表示监听某个数据库的所有表的变更。`复制存量数据` 可以根据业务需求选择是否打开，然后单击**下一步**。

4. 根据业务需求配置数据目标，然后单击**提交**。

5. 当 PostgreSQL 数据发送变化时，可以在目标 Topic 中查看到新增的消息。

- 数据目标为 CKafka 实例的 Topic，可以在左侧导航栏单击**消息查询**进行查看；
- 数据目标为单独 Topic 时，可以在左侧导航栏单击 **Topic 列表**，然后单击 Topic 进入详情页，再单击**查看消息**。

步骤3：创建数据流出任务

1. 在左侧导航栏单击**任务管理** > **任务列表**，选择好地域后，单击**新建任务**。
2. 任务类型选择**数据流出**，数据目标选择**日志服务（CLS）**，单击**下一步**。

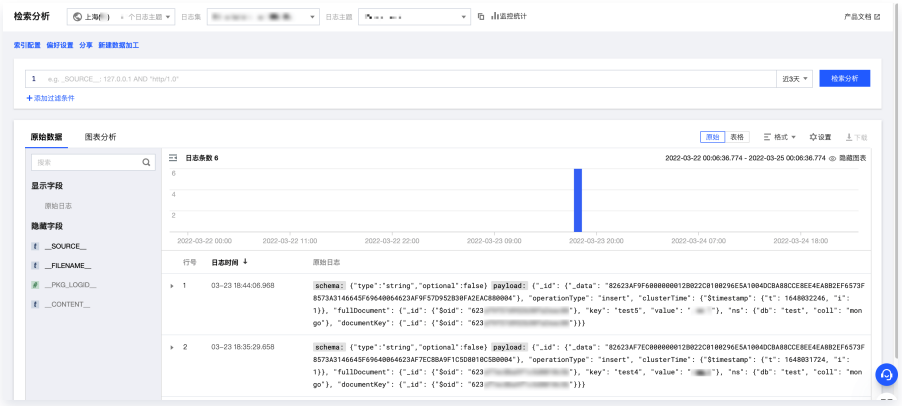
3. 填写任务详情，选取与数据接入任务相同的 CKafka 实例和 Topic，保证在消息生产后能直接进行消费。

4. 单击**提交**，等待任务显示健康，即表示创建成功。

> **说明**
> 当任务在健康的状态时， Topic 有新增的消息写入，会直接被消费到指定的 CLS 日志主题中。

步骤4：查看流出数据

1. 登录 日志服务 控制台。
2. 在左侧导航栏选择**检索分析**，选择流出时填写的日志集与日志主题的“ID”，即可看到 PostgreSQL 的变更记录。



3. 通过关键字检索等操作，能直观得到所需要的记录。

MySQL 到 Elasticsearch 实时数据同步

最近更新时间：2024-10-14 10:58:01

场景说明

本文档满足如下场景：将 MySQL 表的数据(存量+增量)实时同步数据同步到目标 ES 索引。实时同步需要同步新增，修改和删除操作。即当 MySQL 源表出现新增、修改、删除时，目标 ES 中的数据也需要发生相应的增删改。

使用限制

本种实时同步增删改的任务。一个订阅任务只能订阅一张表的数据，一个 Topic 里面只能存一个表的订阅数据。即订阅一张表的数据到 ES 需要创建一个订阅任务，一个 Topic，一个数据流出任务。

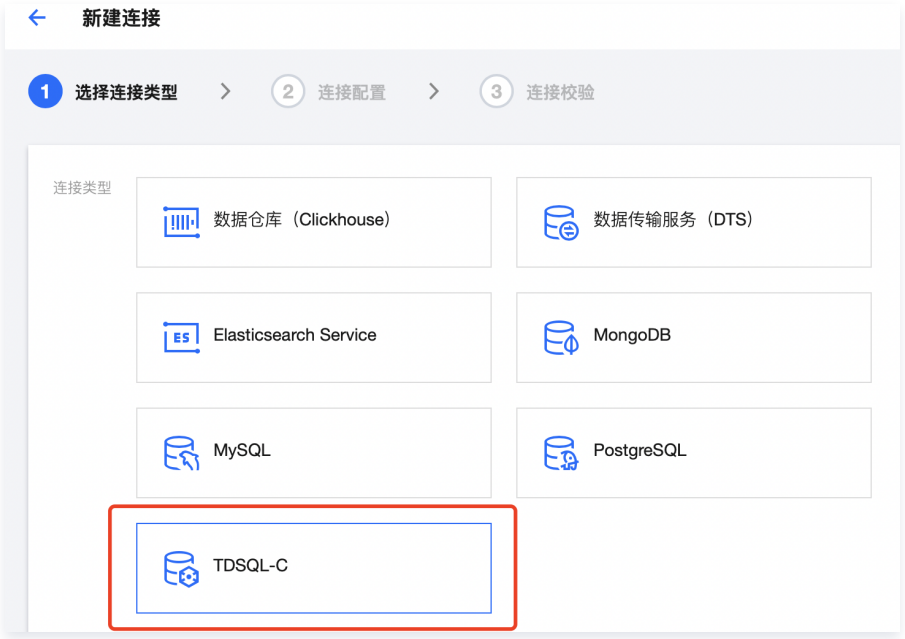
> **说明**
> 如果需要同步多张表的数据到 ES 里面。则需要创建多个 Topic，同时创建多个订阅任务和流出任务。

操作步骤

步骤1：创建连接

1. 创建 MySQL 连接

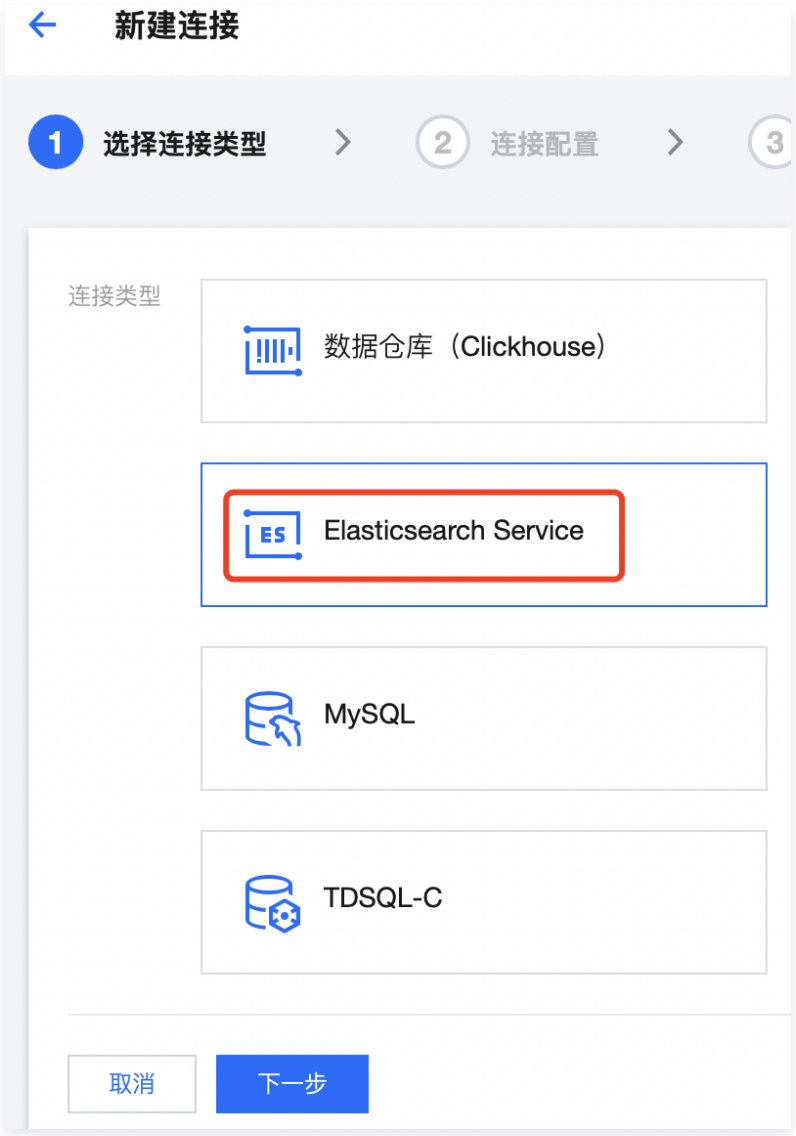
1. 单击**连接器**中的**连接列表**，单击**新建连接**，选择 TDSQL-C 数据库。



2. 填写需要同步的 MySQL 数据库的相关信息。

2. 创建 Elasticsearch 连接

1. 单击**连接器**中的 **连接列表**，单击**新建连接**，选择 Elasticsearch Service。

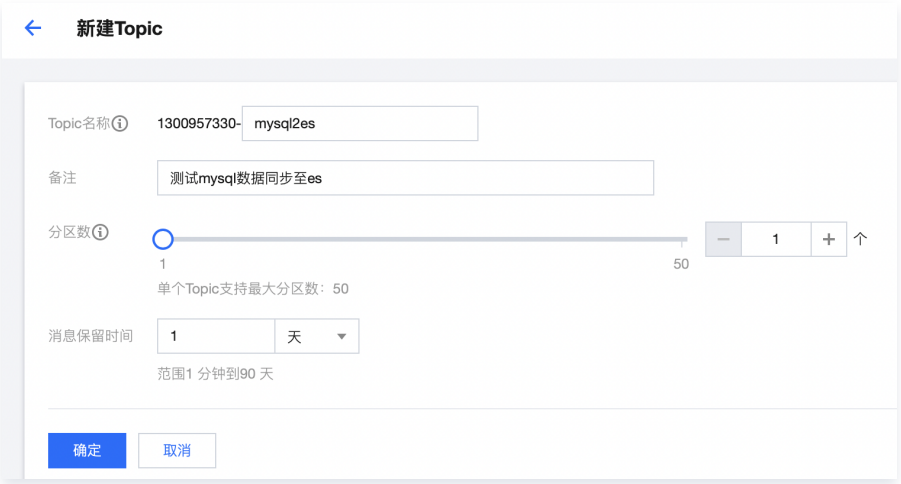

新建连接
1 选择连接类型
2 连接配置
3
连接类型
数据仓库（Clickhouse）
ES
Elasticsearch Service
MySQL
TDSQL-C
取消
下一步

2. 配置 ES 相关的参数：

步骤2：创建 Topic

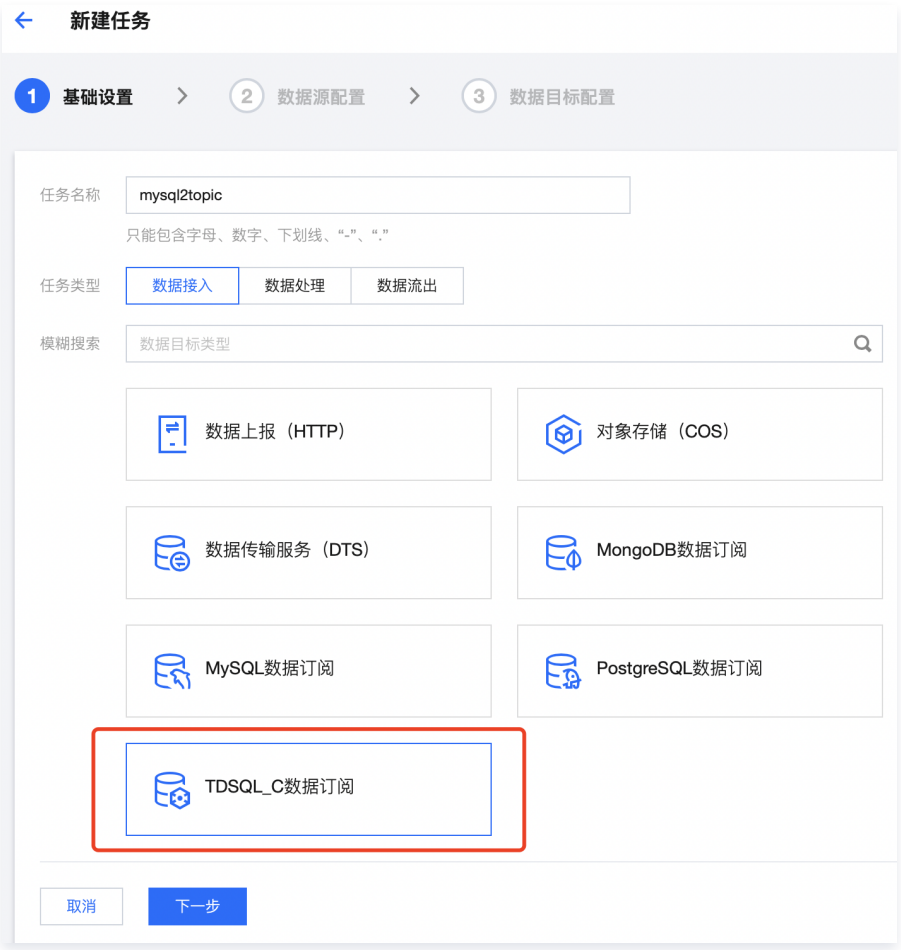
创建 Topic 有两种方式，如果已购买 kafka 实例，则直接在实例中新建 Topic 即可。否则，可以新建按量付费的 Topic（无需购买实例）。购买按量付费 Topic 操作步骤如下：

1. 进入 Ckafka 控制台，单击**弹性Topic**。
2. 单击**新建 Topic**。（**若计划将数据同步至 Ckafka 实例中的 Topic 则可跳过此步骤**）



步骤3：创建数据订阅任务

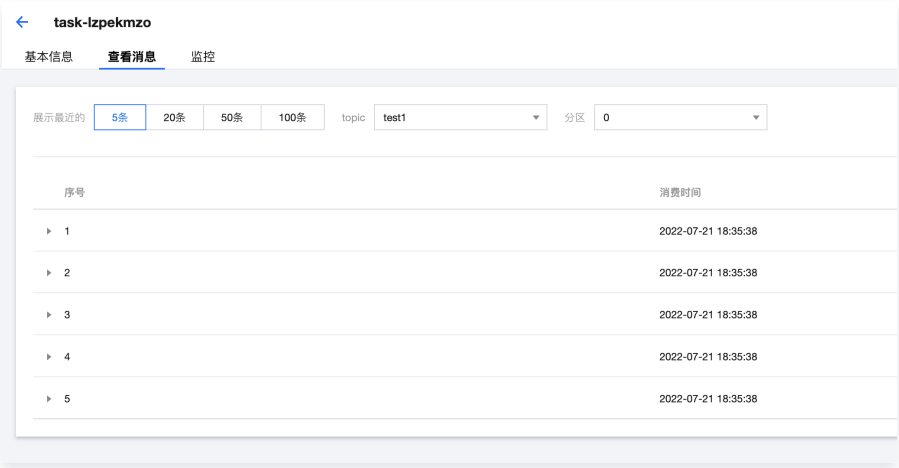
1. 进入**连接器**中的 **任务管理** > **任务列表**页面，单击**新建任务**。



2. 单击**下一步**，填写数据源配置信息：

3. 继续单击**下一步**，选择数据目标信息，即同步 MySQL 数据的 Topic，根据实际情况选择**弹性 Topic** 或 **CKafka 实例内 Topic** 即可，此处选择 步骤2 中创建的**弹性 Topic**：

4. 任务创建成功后，在任务详情 – 查看消息可以看到订阅的数据信息：



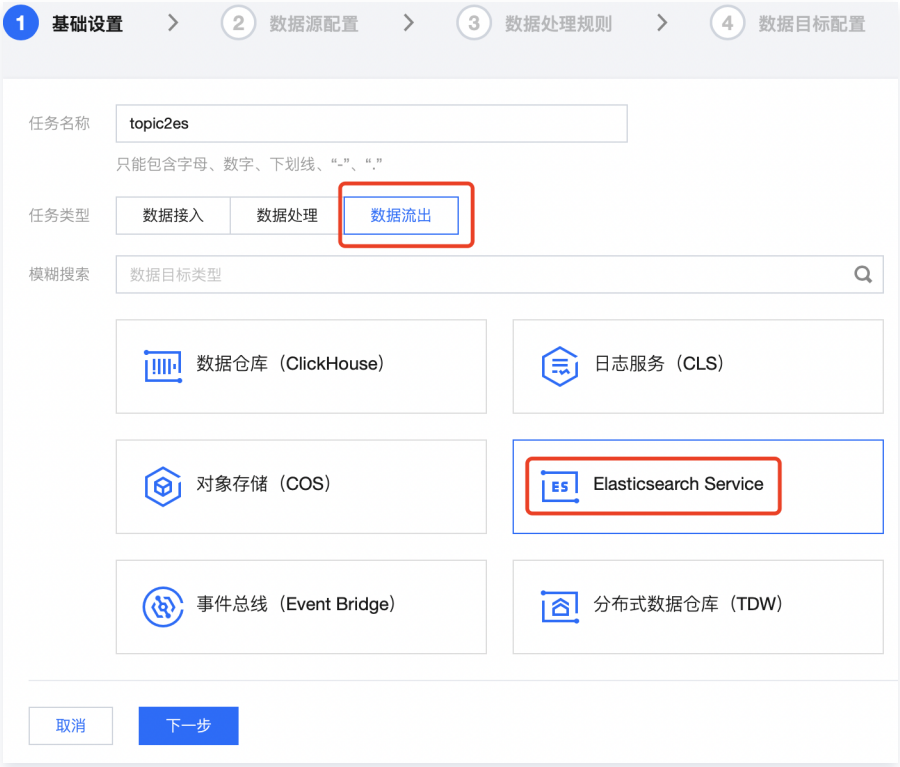
注意：

只有源表有数据存在的时候，才会订阅到消息。当源表没有数据时，可执行如下类似的insert语句，触发订阅行为，即可查询到订阅的数据：

```
insert into test values('testname',25);
```

步骤4：创建数据流出任务

1. 新建连接后，单击**任务管理 > 任务列表**，单击**新建任务**，任务类型选择**数据流出**，选择 **Elasticsearch Service**：



2. 配置数据源，选择同步了 MySQL 数据的 Topic，这里选择步骤1中的 Topic，选择 **从最开始位置开始消费**。

3. 下一步中数据处理可根据实际情况进行配置，这里不进行相关配置，使用原始消息数据。最后进行 ES 相关配置，其中主键为数据库表的主键名称。

> **注意**
> - 此模式需要开启数据库同步模式，并填写表的主键的列名，如此处主键列名为 id。
> - 若开启了数据库同步模式，数据需要是完整的 binlog 信息（包含 schema）。

4. 当数据任务运行后，即可在 ES 对应的索引中查询到相应的消息。

PostgreSQL 到 Elasticsearch 实时数据同步

最近更新时间：2024-10-14 15:38:58

场景说明

本文档满足如下场景：将 PostgreSQL 表的数据(存量+增量)实时同步数据同步到目标ES索引。实时同步需要同步新增，修改和删除操作。即当 PostgreSQL 源表出现新增、修改、删除时，目标 ES 中的数据也需要发生相应的增删改。

使用限制

本种实时同步增删改的任务。一个订阅任务只能订阅一张表的数据，一个 Topic 里面只能存一个表的订阅数据。即订阅一张表的数据到 ES 需要创建一个订阅任务，一个 Topic，一个数据流出任务。同时为了保证数据同步的有序性，一个 Topic 只支持一个分区。

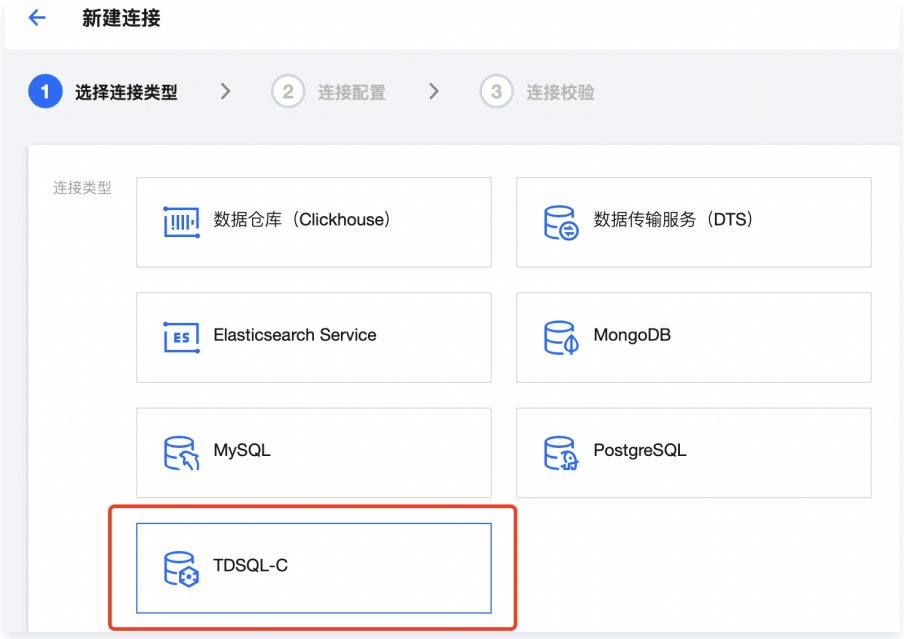
> **说明**
> 如果需要同步多张表的数据到 ES 里面。则需要创建多个 Topic，同时创建多个订阅任务和流出任务。

操作步骤

步骤1：创建连接

1. 创建 PostgreSQL 连接

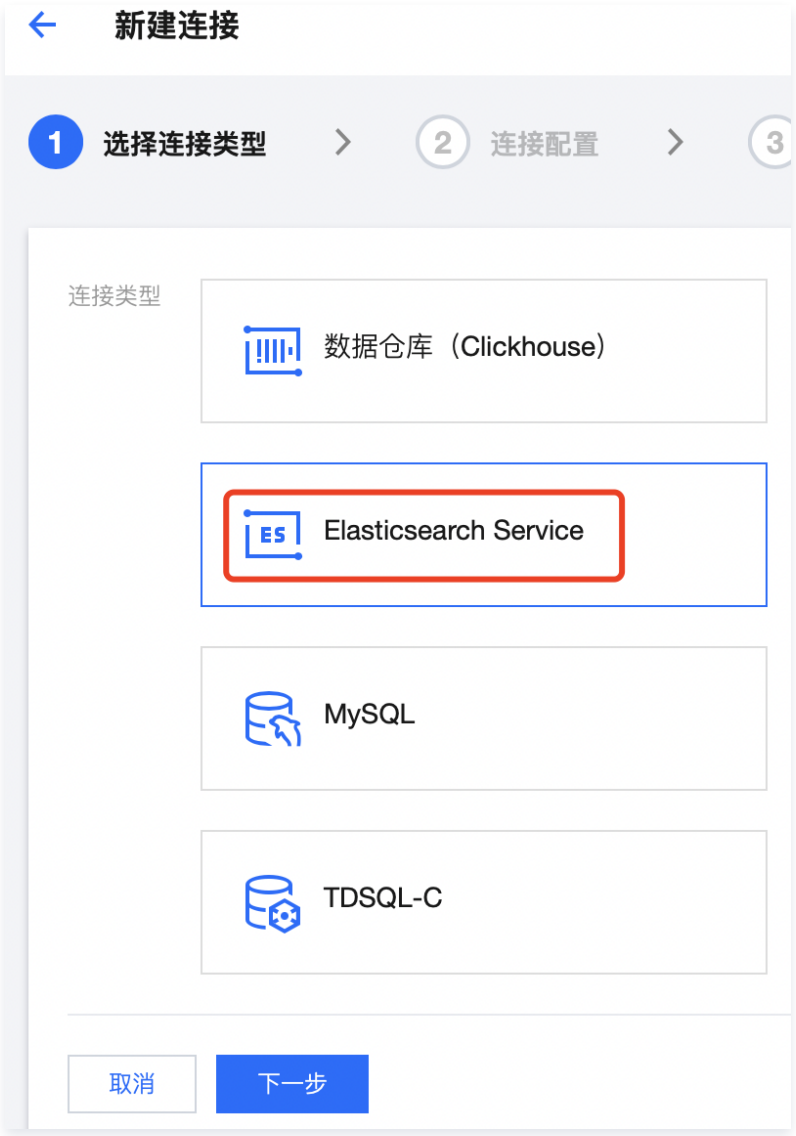
1. 单击**连接器**中的**连接列表**，单击**新建连接**，选择 TDSQL-C 数据库。



2. 填写需要同步的 PostgreSQL 数据库的相关信息。

2. 创建 Elasticsearch 连接

1. 单击**连接器**中的 **连接列表**，单击**新建连接**，选择 Elasticsearch Service。


新建连接
1 选择连接类型
2 连接配置
3
连接类型
数据仓库（Clickhouse）
ES
Elasticsearch Service
MySQL
TDSQL-C
取消
下一步

2. 配置 ES 相关的参数：

步骤2：创建 Topic

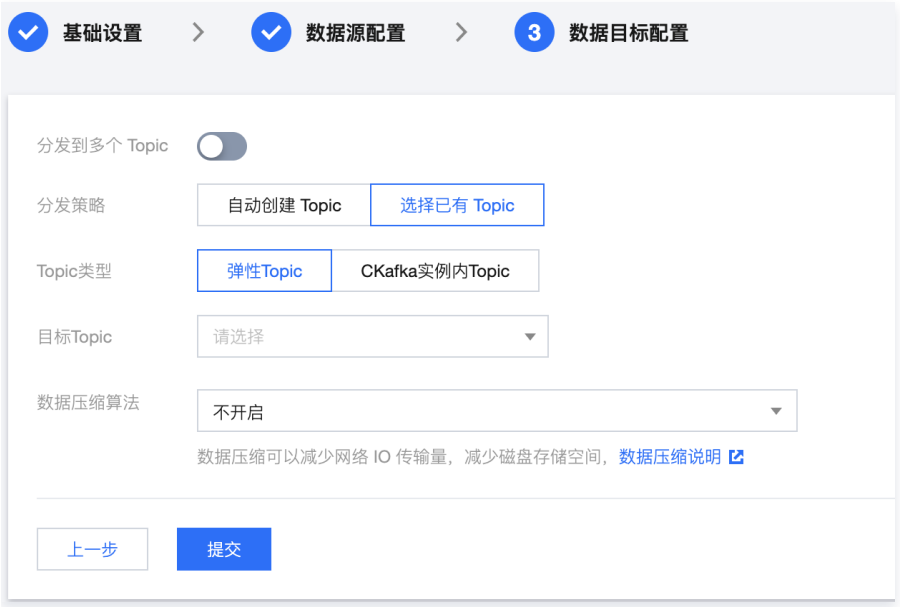
创建 Topic 有两种方式，如果已购买 kafka 实例，则直接在实例中新建 Topic 即可。否则，可以新建按量付费的 Topic（无需购买实例）。购买按量付费 Topic 操作步骤如下：
进入 Ckafka 控制台，选择**弹性 Topic**，单击**新建 Topic**。（**若计划将数据同步至 Ckafka 实例中的 topic 则可跳过此步骤**）

步骤3：创建数据订阅任务

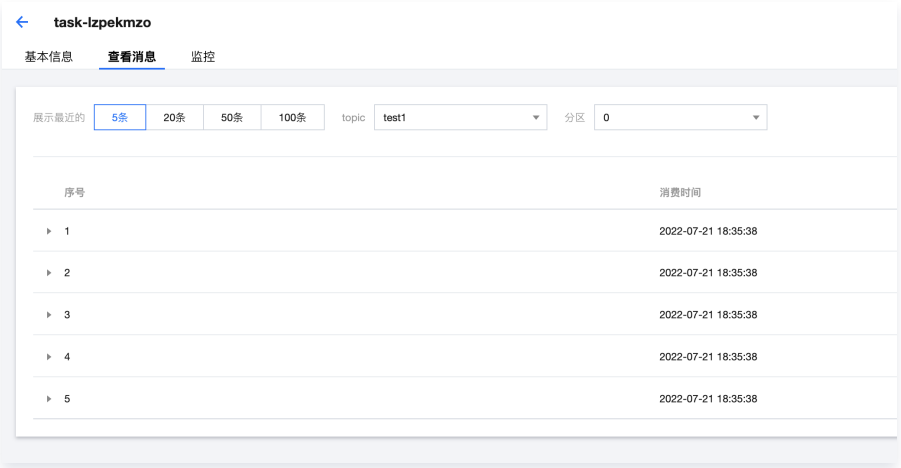
1. 单击**连接器**中的**任务管理**，单击**任务列表**，单击**新建任务**。

2. 单击**下一步**，填写数据源配置信息，数据源为 **步骤一** 中创建的 Postgresql 订阅连接：

3. 继续单击**下一步**，选择数据目标信息，即同步 PostgreSQL 数据的 topic，根据实际情况选择**弹性 Topic** 或 **CKafka 实例内 Topic** 即可，此处选择 **步骤二** 中创建的弹性 topic：



4. 任务创建成功后，在**任务详情 > 查看消息**可以看到订阅的数据信息：



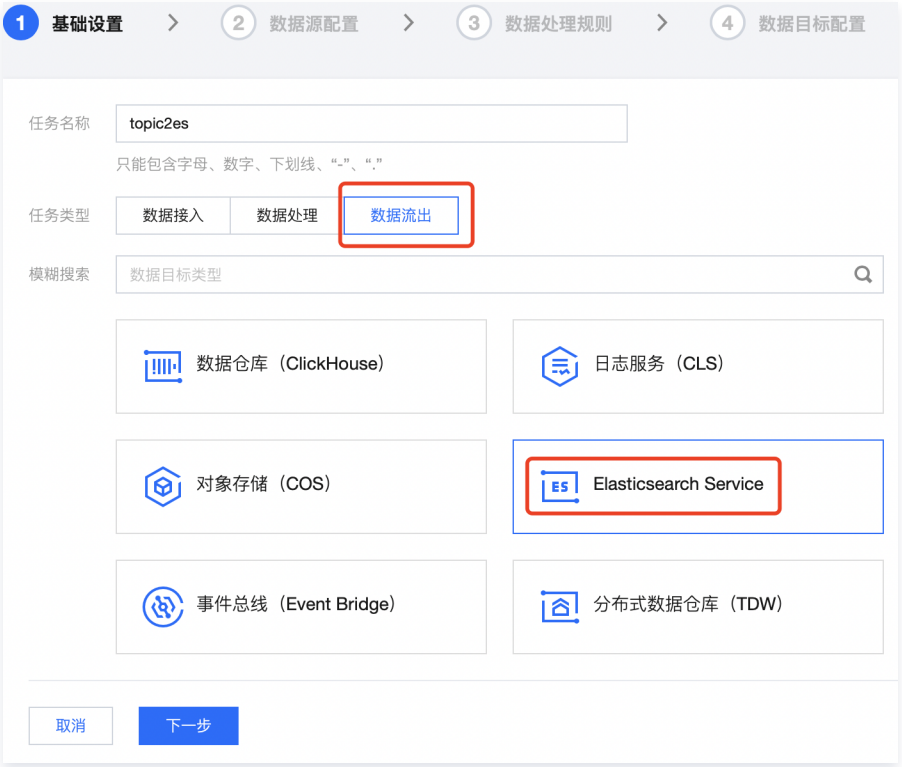
注意：

只有源表有数据存在的时候，才会订阅到消息。当源表没有数据时，可执行如下类似的 insert 语句，触发订阅行为，即可查询到订阅的数据：

```
insert into test values('testname',25);
```

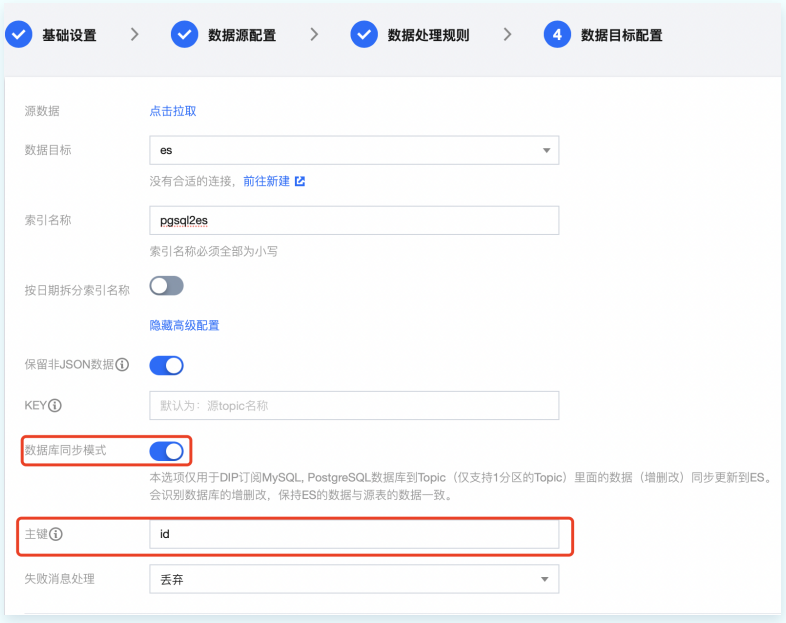
步骤4：创建数据流出任务

1. 新建连接后，单击**任务管理 > 任务列表**，单击**新建任务**，任务类型选择**数据流出**，选择 Elasticsearch Service：



2. 配置数据源，选择同步了 MySQL 数据的 topic，这里选择 步骤1 中的 topic，选择 **从最开始位置开始消费**。

3. 下一步中数据处理可根据实际情况进行配置，这里不进行相关配置，使用原始消息数据。最后进行 ES 相关配置，其中主键为数据库表的主键名称。

> **注意**
> 此模式需要开启 **数据库同步模式**，并填写表的主键的列名，如此处主键列名为 **id**。
>
> 
>

4. 当数据任务运行后，即可在 ES 对应的索引中查询到相应的消息。

替换 Canal 实现 MySQL 数据库订阅

最近更新时间：2025-05-22 10:33:43

> **注意：**
> 目前支持公测的连接类型有 Elasticsearch Service，消息队列 Kafka。数据库订阅方式的连接器已停止新增，详情参见 连接管理 。

背景

Canal 是阿里巴巴开源的一款通过解析 MySQL 数据库增量日志，达到增量数据订阅和消费目的 CDC 工具。Canal 解析 MySQL 的 binlog 后可投递到 Kafka 一类的消息中间件，供下游系统进行分析处理。
如果您正在或考虑使用 Canal 同步 MySQL 的增量变更记录到 Kafka，腾讯云 CKafka 连接器提供了兼容此场景的能力。
本文介绍 Canal 拉取 MySQL 变更记录并投递到 Kafka 的简易使用场景以及 CKafka 连接器相应的替换处理方式。

CKafka 连接器订阅 MySQL 功能列表

- 允许选择多库多表。
- 允许指定根据队列消息进行分区同步。
- 允许通过正则匹配选择多表。
- 支持存量数据的同步。
- 同时支持 DDL、DML 类型的变更。
- 兼容 Canal、debezium 等消息格式。

案例说明

前提：已有 MySQL 实例和 Kafka 实例

Canal 订阅 MySQL 变更记录到 Kafka

1. 准备一台 CVM 服务器，登录到服务器并创建 canal 目录，下载 压缩包 。
2. 下载 canal server： `canal.deployer-1.1.x.tar.gz` 并解压。

```
tar -zxvf canal.deployer-1.1.x.tar.gz
```

3. 进入解压目录，配置 conf/example/instance.properties 文件，下面展示必须配置的 mysql 实例信息以及消息中间件的 topic、partition 等信息：

```
canal.instance.master.address=xx.xx.xx.xx:3306
canal.instance.dbUsername=user
canal.instance.dbPassword=password
canal.mq.topic=canal-test-1
canal.mq.partition=0
```

4. 进入解压目录，配置 conf/canal.properties 文件，下面展示必须配置的消息中间件实例信息：

```
canal.serverMode = kafka
kafka.bootstrap.servers = xx.xx.xx.xx:9092
kafka.acks = all
kafka.compression.type = none
kafka.batch.size = 16384
kafka.linger.ms = 1
kafka.max.request.size = 1048576
```

```
kafka.buffer.memory = 33554432
kafka.max.in.flight.requests.per.connection = 1
kafka.retries = 0
```

5. 启动 canal server：

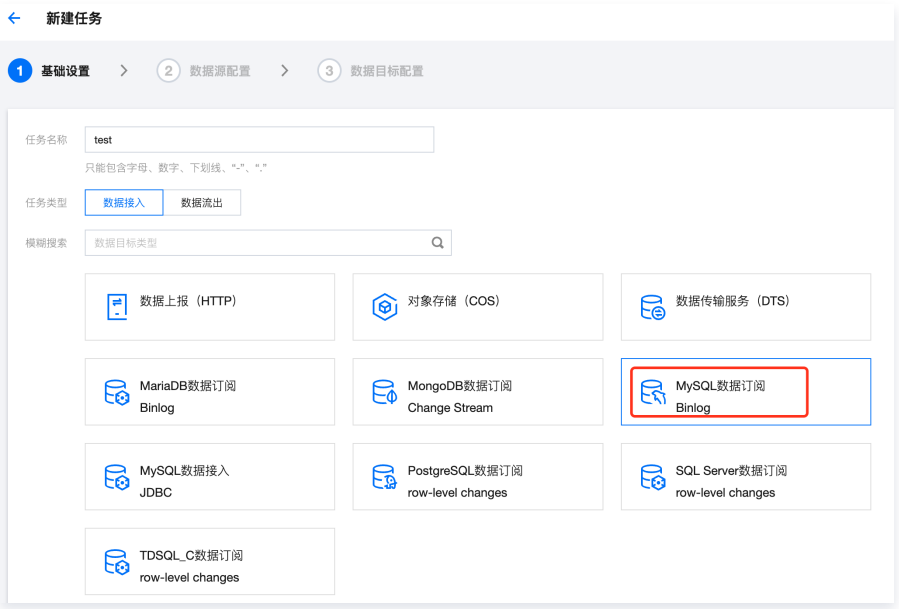
```
sh bin/startup.sh
```

启动 canal server 后，Canal 开始根据配置获取 MySQL 的增量变更消息并推送到 Kafka 集群。

CKafka 连接器 订阅 MySQL 变更记录到 Kafka

1. 新建连接，选择 MySQL 实例，CKafka 连接器同时支持腾讯云 MySQL 实例和云上自建 MySQL 实例：

2. 新建数据接入任务，选择 MySQL 数据订阅（Binlog）：

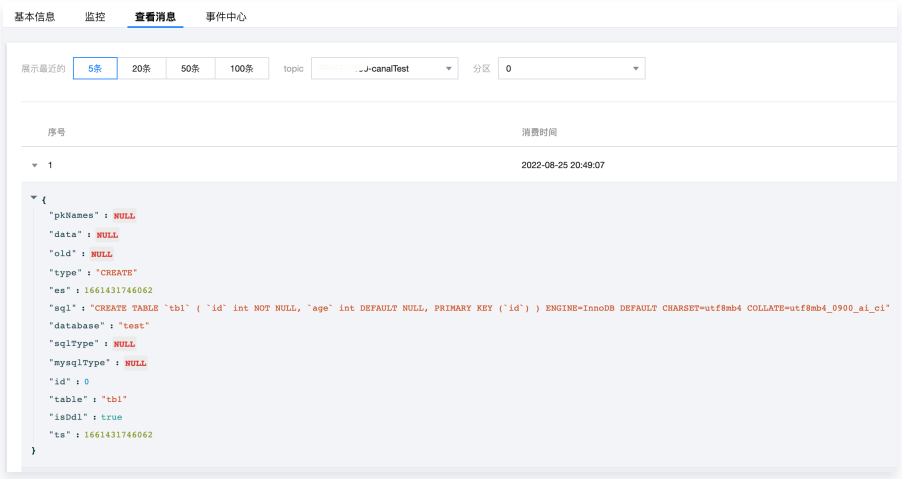


3. 数据源选择**第一步**中配置的连接，CKafka 连接器支持多种订阅库表的方式，包括全部库表、批量选择、正则匹配：

4. CKafka 连接器提供了高级配置选项，您可以根据业务需求自行进行选择配置，其中数据格式选择 Canal 格式：

5. 最后配置 Canal 的数据目标，CKafka 连接器支持将消息推送到 CKafka 实例中的 Topic 或单独创建的弹性 Topic。

6. 可在控制台查看 Topic 的落盘消息情况：



更多功能

多种上游数据源的订阅

除了订阅 MySQL 数据源，CKafka 连接器同时支持订阅其他多种上游数据源，例如 Postgresql、MongoDB、MariaDB、SQL Server 等。

可视化实时监控面板

CKafka 连接器提供了可视化的实时监控面板，可以一键更改配置信息、一键查看各种性能指标、一键查看 Topic 中落盘的消息内容，能够做到高可靠、免运维。

对标 logstash 的数据处理

CKafka 连接器提供了对标 logstash 的数据处理能力，仅需通过界面进行编辑即可创建多种数据处理规则。详情参见 数据处理规则说明 。

多种下游数据源的投递

CKafka 连接器支持多种下游数据源的消息推送功能，包括 Elastic Search、ClickHouse 等。如果您正在或考虑使用 Canal 将消息同步到 kafka 以外的其他数据源，CKafka 连接器同样提供了相应的能力。

数据简单清洗

最近更新时间：2024-10-14 15:38:58

操作场景

在使用 CKafka 连接器在进行数据流入流出服务时，可能会遇到如下情况：

- 需要对消息中的特定字段进行解析，得到相关信息。
- 需要多次迭代处理消息中某一字段。
- 需要处理为未经过结构化的原始消息后，才能继续使用。
- 需要处理多层嵌套格式的消息。

目前 CKafka 团队的技术专家经过长期的经验总结，全新推出了数据处理 V2.0 版本。该版本在完美兼容原有数据处理 V1.0 的基础上，还增加了以下特性：

- 插件丰富：数据处理支持丰富的处理预制插件，利于快速生成所需的消费格式。
- 链式处理：数据处理支持消息链式处理，即上一次的处理结果能够作为本次处理的输入参数，利于处理复杂结构。
- 可视预览：数据处理支持实时 JSON 结构化预览，利于迅速定位排查问题。
- 类型转换：数据处理支持不同数据类型间的转换，利于校验数据格式。

运行原理

整个数据处理过程如下图所示，各个组件结构说明如下：

- 数据处理组件集群中，多个 worker 共同组成一个消费者组，批量读取源 topic 中的消息，并且对于每条消息顺序执行处理操作。
- 对于每条消息，根据控制台设置的处理链顺序，依次展开消息字段的嵌套结构，并进行替换/截取/数据转换/时间格式化等操作。
- 下一条处理链读取上一条处理链结果，并且进行本轮的链式处理，并将处理结果投递到下一轮。
- 执行完最后一轮处理链后的结果，投递到设置的目标 topic 中，至此完成该条消息的数据处理操作。

前提条件

- 需开通 CKafka 服务。
- 需消息格式为 JSON 格式或字符串格式，目前暂不支持其他编码协议。

实际应用

处理字符串类型格式日志

Nginx 格式日志通常如下所示，这种 Nginx 默认格式也被称作 `combined`，可以从中解析出请求者信息，报文信息，请求状态等信息，以便于对数据进行进一步分析。

```
66.249.65.159 - - [06/Nov/2014:19:10:38 +0600] "GET /news/53f8d72920ba2744fe873ebc.html
HTTP/1.1" 404 177 "-" "Mozilla/5.0 (iPhone; CPU iPhone OS 6_0 like Mac OS X) AppleWebKit/536.26
(KHTML, like Gecko) Version/6.0 Mobile/10A5376e Safari/8536.25 (compatible; Googlebot/2.1;
+http://www.google.com/bot.html)"
```

由于 Nginx 的 `combined` 格式采用空格和 `-` 分隔符来分隔数据，因此按照下文顺序设计解析流程：

1. 首先使用 `"` **分隔符** 进行初步数据解析，此时数据处理将会自动将日志转换成 JSON 结构

```
{
    "0": "66.249.65.159 - - [06/Nov/2014:19:10:38 +0600] ",
    "1": "GET /news/53f8d72920ba2744fe873ebc.html HTTP/1.1",
    "2": " 404 177 ",
    "5": "Mozilla/5.0 (iPhone; CPU iPhone OS 6_0 like Mac OS X) AppleWebKit/536.26 (KHTML,
like Gecko) Version/6.0 Mobile/10A5376e Safari/8536.25 (compatible; Googlebot/2.1;
+http://www.google.com/bot.html)"
}
```

2. 从上文 JSON 结构中可知，键名为 `0` 和 `2` 的字段由于 `-` 和空格的影响，还存在连接的耦合数据。因此再分别对这两个键进行 `-` 和空格进行 **分隔符** 拆分。拆分后的 JSON 结果如下所示：

```
{
    "1": "GET /news/53f8d72920ba2744fe873ebc.html HTTP/1.1",
    "5": "Mozilla/5.0 (iPhone; CPU iPhone OS 6_0 like Mac OS X) AppleWebKit/536.26 (KHTML,
like Gecko) Version/6.0 Mobile/10A5376e Safari/8536.25 (compatible; Googlebot/2.1;
+http://www.google.com/bot.html)",
    "0.0": "66.249.65.159 ",
    "0.2": " [06/Nov/2014:19:10:38 +0600] ",
    "2.1": "404",
    "2.2": "177"
}
```

3. 由于时间格式前后还带有 `[]` 方括号，再使用一次 **分隔符** 进行截取，截取后的 JSON 如下所示：

```
{
    "1": "GET /news/53f8d72920ba2744fe873ebc.html HTTP/1.1",
    "5": "Mozilla/5.0 (iPhone; CPU iPhone OS 6_0 like Mac OS X) AppleWebKit/536.26 (KHTML,
like Gecko) Version/6.0 Mobile/10A5376e Safari/8536.25 (compatible; Googlebot/2.1;
+http://www.google.com/bot.html)",
    "0.0": "66.249.65.159 ",
    "0.2": "06/Nov/2014:19:10:38 +0600",
    "2.1": "404",
    "2.2": "177"
}
```

4. 由于所有字段都已经被合适拆分，最后根据对应字段属性，给相应映射字段的 `key` 设置名称即可。修改后最终结果如下所示：

```
{
    "request": "GET /news/53f8d72920ba2744fe873ebc.html HTTP/1.1",
    "http_user_agent": "Mozilla/5.0 (iPhone; CPU iPhone OS 6_0 like Mac OS X)
AppleWebKit/536.26 (KHTML, like Gecko) Version/6.0 Mobile/10A5376e Safari/8536.25
(compatible; Googlebot/2.1; +http://www.google.com/bot.html)",
    "remote_addr": "66.249.65.159 ",
    "dateTime": "06/Nov/2014:19:10:38 +0600",
    "status": "404",
    "body_bytes_sent ": "177"
}
```

处理嵌套类型格式日志

TKE 采集格式通常如下所示，TKE 采集器会将 metadata 数据放入 JSON 结构中的 `kubernetes` 字段中，采集到的日志放入 `log` 字段中。大致结构如下所示：

```
{
  "@timestamp": 1648803500.63659,
  "@filepath": "/var/log/tke-log-agent/test7/c816991f-adfe-4617-8cf3-9997aea90ded/c_tke-es-
687995d557-n29jr_default_nginx-
add90ccf49626ef42d5615a636aae74d6380996043cf6f6560d8131f21a4d8ba/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-687995d557-n29jr",
    "namespace_name": "default",
    "pod_id": "c816991f-adfe-4617-8cf3-9997aea90ded",
    "labels": {
      "k8s-app": "tke-es",
      "pod-template-hash": "687995d557",
      "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        \"172.16.0.31\"\n    ],\n    \"mac\":
\"ae:61:12:4a:c2:ba\",\n    \"default\": true,\n    \"dns\": {}\n}]"
    },
    "host": "10.0.96.47",
    "container_name": "nginx",
    "docker_id": "add90ccf49626ef42d5615a636aae74d6380996043cf6f6560d8131f21a4d8ba",
    "container_hash":
"nginx@sha256:e1211ac17b29b585ed1aee166a17fad63d344bc973bc63849d74c6452d549b3e",
    "container_image": "nginx"
  }
}
```

当日志采集后投递进 ElasticSearch 时，由于支持嵌套 JSON 结构，因此不需要对数据做出太大改动。但当需要投递到数据库时，此时就需要将数据转换成能够识别的单层 JSON 格式，因此按照下文顺序设计解析流程：

1. 使用 JSONPATH 语句处理第一层嵌套的 JSON 结构 `$.kubernetes`，解析模式选择 `JSON`。首先将嵌套 JSON 结构转换为单层 JSON 结构。测试后结果如下所示：

```
{
    "@timestamp": 1.64880350063659E9,
    "@filepath": "/var/log/tke-log-agent/test7/c816991f-adfe-4617-8cf3-9997aea90ded/c_tke-es-
687995d557-n29jr_default_nginx-
add90ccf49626ef42d5615a636aae74d6380996043cf6f6560d8131f21a4d8ba/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",
    "$.kubernetes.pod_name": "tke-es-687995d557-n29jr",
    "$.kubernetes.namespace_name": "default",
    "$.kubernetes.pod_id": "c816991f-adfe-4617-8cf3-9997aea90ded",
    "$.kubernetes.labels": {
    "k8s-app": "tke-es",
    "pod-template-hash": "687995d557",
    "qcloud-app": "tke-es"
```

```
    },
    "$.kubernetes.annotations": {
    "qcloud-redeploy-timestamp": "1648016531476",
    "tke.cloud.tencent.com/networks-status": "[{\n    \"name\": \"tke-bridge\",\n
\"interface\": \"eth0\",\n    \"ips\": [\n        \"172.16.0.31\"\n    ],\n    \"mac\":
\"ae:61:12:4a:c2:ba\",\n    \"default\": true,\n    \"dns\": {}\n}]"
    },
    "$.kubernetes.host": "10.0.96.47",
    "$.kubernetes.container_name": "nginx",
    "$.kubernetes.docker_id":
"add90ccf49626ef42d5615a636aae74d6380996043cf6f6560d8131f21a4d8ba",
    "$.kubernetes.container_hash":
"nginx@sha256:e1211ac17b29b585ed1aee166a17fad63d344bc973bc63849d74c6452d549b3e",
    "$.kubernetes.container_image": "nginx"
}
```

2. 依次处理第二层嵌套结构 `$.kubernetes.annotations` 和 `$.kubernetes.labels`。在处理链中使用 `Map` 方式选中这两个名称，即可将嵌套格式转换成单层 JSON 格式。处理后如下所示：

```
{
    "@timestamp": 1648803500.63659,
    "@filepath": "/var/log/tke-log-agent/test7/c816991f-adfe-4617-8cf3-9997aea90ded/c_tke-es-
687995d557-n29jr_default_nginx-
add90ccf49626ef42d5615a636aae74d6380996043cf6f6560d8131f21a4d8ba/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",
    "$.kubernetes.pod_name": "tke-es-687995d557-n29jr",
    "$.kubernetes.namespace_name": "default",
    "$.kubernetes.pod_id": "c816991f-adfe-4617-8cf3-9997aea90ded",
    "$.kubernetes.host": "10.0.96.47",
    "$.kubernetes.container_name": "nginx",
    "$.kubernetes.docker_id":
"add90ccf49626ef42d5615a636aae74d6380996043cf6f6560d8131f21a4d8ba",
    "$.kubernetes.container_hash":
"nginx@sha256:e1211ac17b29b585ed1aee166a17fad63d344bc973bc63849d74c6452d549b3e",
    "$.kubernetes.container_image": "nginx",
    "$.kubernetes.labels.k8s-app": "tke-es",
    "$.kubernetes.labels.pod-template-hash": "687995d557",
    "$.kubernetes.labels.qcloud-app": "tke-es",
    "$.kubernetes.annotations.qcloud-redeploy-timestamp": "1648016531476",
    "$.kubernetes.annotations.tke.cloud.tencent.com/networks-status": "[{\n    \"name\":
\"tke-bridge\",\n    \"interface\": \"eth0\",\n    \"ips\": [\n        \"172.16.0.31\"\n
],\n    \"mac\": \"ae:61:12:4a:c2:ba\",\n    \"default\": true,\n    \"dns\": {}\n}]"
}
```

3. 最后修改相应映射字段的 `key` 为所需名称，并且删去不需要的字段。此时单击 **添加处理链** 后，打开 **处理上层所有结果** 按钮，整理优化后可以参见如下所示：

```
{
    "@timestamp": 1.64880350063659E9,
    "@filepath": "/var/log/tke-log-agent/test7/c816991f-adfe-4617-8cf3-9997aea90ded/c_tke-es-
687995d557-n29jr_default_nginx-
```

```
add90ccf49626ef42d5615a636aae74d6380996043cf6f6560d8131f21a4d8ba/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",
    "pod_name": "tke-es-687995d557-n29jr",
    "namespace_name": "default",
    "pod_id": "c816991f-adfe-4617-8cf3-9997aea90ded",
    "host": "10.0.96.47",
    "container_name": "nginx",
    "docker_id": "add90ccf49626ef42d5615a636aae74d6380996043cf6f6560d8131f21a4d8ba"
}
```

说明

在使用 JSONPath 处理参数时，当 `key` 中本身带有英文句号 `.` 时，需要在路径中方括号和单引号进行隔离。
例如 `{"key1.key2":"value1"}` 中，要想取得对应字段，则需要使用 `$.['key1.key2']` 进行获取相应键值。

处理字符串序列化 JSON 格式日志

有时 JSON 格式在传输过程中，由于格式或性能等需要，需要转义成字符串格式的形式，此处把这种格式叫做 Raw JSON。需要在数据处理中重新将字符串反序列化成 JSON 格式。 此处以 MongoDB 中的 Raw JSON 格式为例，大致结构如下所示：

```
{
  "key": "  {\n    \"categories\": [\"dev\"],\n    \"created_at\": \"2020-01-05
13:42:19.324003\",\n    \"icon_url\": \"https://assets.chucknorris.host/img/avatar/chuck-
norris.png\",\n    \"id\": \"elgv2wkvt8ioag6xywykbq\",\n    \"updated_at\": \"2020-01-05
13:42:19.324003\",\n    \"url\": \"https://api.chucknorris.io/jokes/elgv2wkvt8ioag6xywykbq\",\n
\"value\": \"Chuck Norris's keyboard doesn't have a Ctrl key because nothing controls Chuck
Norris.\"\n  }\n"
}
```

如果在数据处理中就将 Raw JSON 转换成普通的 JSON 格式，就能按照字段格式直接投递到目标下游中。大致处理思路如下所示：

1. 首先设置 **解析模式** 为 JSON。这样能够将消息先预读取成内部的 MAP 格式。解析后如下所示：

```
{
    "key": "  {\n    \"categories\": [\"dev\"],\n    \"created_at\": \"2020-01-05
13:42:19.324003\",\n    \"icon_url\": \"https://assets.chucknorris.host/img/avatar/chuck-
norris.png\",\n    \"id\": \"elgv2wkvt8ioag6xywykbq\",\n    \"updated_at\": \"2020-01-05
13:42:19.324003\",\n    \"url\":
\"https://api.chucknorris.io/jokes/elgv2wkvt8ioag6xywykbq\",\n    \"value\": \"Chuck Norris's
keyboard doesn't have a Ctrl key because nothing controls Chuck Norris.\"\n  }\n"
}
```

2. 单击 **添加处理链**，使用 MAP 方式选中 key，解析方式选择 `JSON` 。这样数据处理就能将 RAW JSON 自动转换成 JSON 形式。解析后如下所示：

```
{
    "key.categories": [
    "dev"
    ],
    "key.created_at": "2020-01-05 13:42:19.324003",
    "key.icon_url": "https://assets.chucknorris.host/img/avatar/chuck-norris.png",
    "key.id": "elgv2wkvt8ioag6xywykbq",
```

```
    "key.updated_at": "2020-01-05 13:42:19.324003",
    "key.url": "https://api.chucknorris.io/jokes/elgv2wkvt8ioag6xywykbq",
    "key.value": "Chuck Norris's keyboard doesn't have a Ctrl key because nothing controls
Chuck Norris."
}
```

3. 最后单击 **添加处理链** 后，打开 **处理上层所有结果** 按钮，修改相应映射字段的 `key` 为所需名称，并且删去不需要的字段。整理优化后可以参见如下所示：

```
{
    "categories": [
    "dev"
    ],
    "created_at": "2020-01-05 13:42:19.324003",
    "icon_url": "https://assets.chucknorris.host/img/avatar/chuck-norris.png",
    "id": "elgv2wkvt8ioag6xywykbq",
    "updated_at": "2020-01-05 13:42:19.324003",
    "url": "https://api.chucknorris.io/jokes/elgv2wkvt8ioag6xywykbq",
    "value": "Chuck Norris's keyboard doesn't have a Ctrl key because nothing controls Chuck
Norris."
}
```

> **说明**
> 目前 RAW JSON 只支持解析 MAP 类型的数据。当最外层为 List 类型时，例如 `"[\"test1\",\"test2\"]"`，或者 `"[{\"key\":\"value\"}]"`，由于无法解析合适的键值，因此将会提示解析失败。

日志采集处理查询
TKE 日志

最近更新时间：2024-10-14 16:44:42

操作场景

在使用 TKE 容器服务 时，先前通常使用如下方法，来获取并查询部署的组件服务日志：

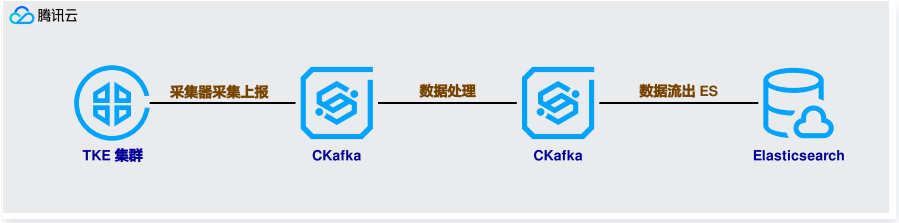
1. 从 **TKE 控制台** 登录容器节点，切换到日志所在文件夹并查看日志。
2. 从 **TKE 控制台** 查询重定向至标准输出的容器日志。
3. **SSH 终端** 登录集群节点，查询挂载在宿主机上的日志，或使用 Kubectl 登录容器查询日志。
4. 使用 **CLS 采集器** 采集日志，并在 日志服务 的控制台查询。

与以上方法相比，采用 Kafka 采集器采集日志消息投递到 CKafka，进而对消息使用 DataHub 进行数据处理后，投递到 Elasticsearch Service 进行日志解析的方法，具有以下显著优势：

- 多点灾备：Kafka 原生提供数据持久化及灾备能力，不会因单节点宕机导致日志丢失。
- 日志区分：在每条日志消息中添加 Metadata 信息 ，增加解析信息能力。
- 精细解析：连接器提供 数据处理 功能，能够结构化精细解析原始日志数据。
- 多元输出：连接器提供多种输出目标源，例如可以将 CKafka 的消息同时投递至 ElasticSearch 及 MongoDB，冷热数据分离节约成本。

运行原理

整个采集投递过程如下图所示，各个组件结构说明如下：



- TKE Kafka 采集器基于 Kubernetes 友好的开源 fluent-bit 采集器编写，监听日志文件变动并投递到目标 Kafka。
- CKafka 数据处理组件采用自研架构，通过建立内置消费者订阅源主题数据，经过消息处理器清洗后投递到目标主题中。
- 数据流出 ES 基于 Kafka 原生 Connect 定制，通过实现 Connect 插件将消息流出至指定的 Elasticsearch Service。

前提条件

- 需开通 TKE 服务。
- 需开通 CKafka 服务。
- 需开通 Elasticsearch Service 服务。

操作步骤

步骤1：启用 TKE 日志采集投递

启用日志采集

1. 登录 TKE 控制台 。
2. 在左侧导航栏选择 **运维功能管理**，点击目标实例操作栏的**设置**按钮。

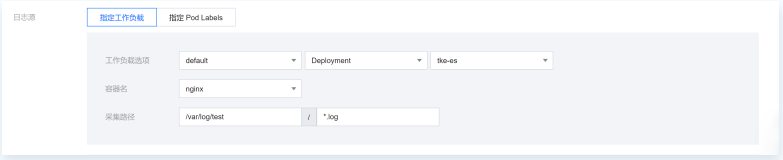
3. 勾选 **开启日志采集**，启用日志采集功能后确认。具体操作步骤可参见 配置日志采集文档 。

> **说明**
> 如果先前已经启用日志采集，需要将采集器版本更新到 `1.0.8.1` 及以上，更新后控制台展示信息如下所示：

创建日志规则

1. 在控制台选择左侧导航栏中的**运维功能管理 > 日志规则**。
2. 在**日志规则**页面上方选择地域和需要配置日志采集规则的集群，单击**新建**，随后进行日志规则的配置。

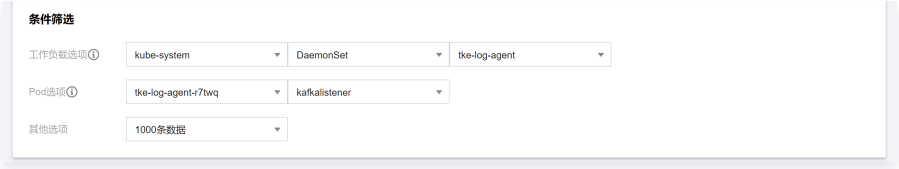
> **说明**
> 以容器内文件日志为例，下图表示创建了一个采集 nginx 容器中，路径为 `/var/log/test` 的文件夹下，所有后缀为 `.log` 的采集配置。



3. 编辑采集配置后，选择投递到 Kafka 相应的实例及主题即可。

查看采集结果

1. 以采集 Java 堆栈日志信息为例，在 TKE 控制台查看 kafkaListener 容器的标准输出。



发现有数据成功投递到目标 Kafka 的主题中，证明投递成功。

2. 查看 CKafka 实例监控，可以得到当前日志采集的速度。

3. 查看 CKafka 的目标主题 offset，可以得到当前消息投递的总数量。

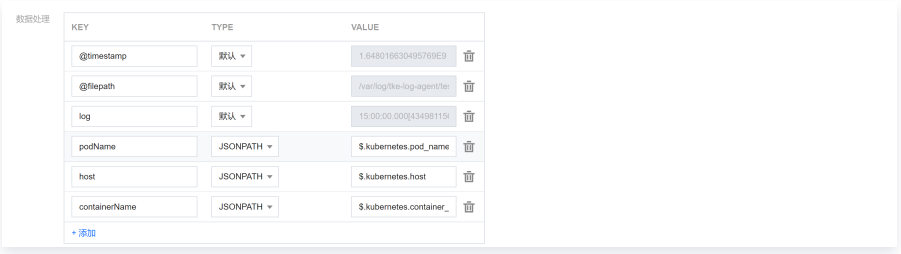
4. 在 CKafka 的 消息查询 界面，输入主题的分区数和时间，得到完整的日志采集消息，可以发现消息中已添加关于集群的 Metadata 信息。

步骤2：连接器数据简单处理

创建数据处理

CKafka 连接器能够涵盖绝大部分的数据处理场景，由于数据处理方式取决于采集的源消息格式，因此此处仅使用简单的数据处理进行展示。

1. 登录 CKafka 控制台 。
2. 在左侧导航栏选择**连接器 > 任务管理 > 任务列表**，单击 **新建任务**。
3. 任务类型选择**数据处理**。
4. 在数据处理规则界面选择 **Json 解析模式**，将所需的嵌套 Metadata 信息转换为单层 JSON 格式。

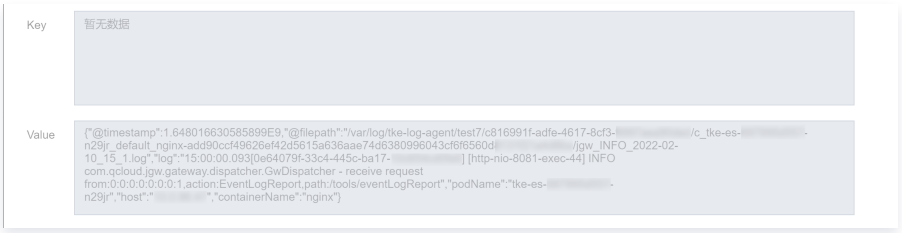


5. 创建后，当数据处理概览页面中显示当前任务为健康状态时，代表数据处理正常。

查看处理结果

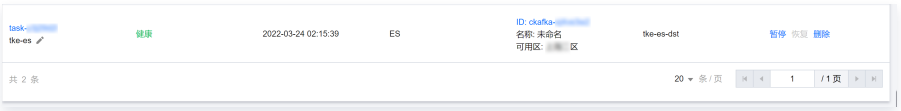
1. 在消息处理概览页面进入任务界面，点击 **监控** 按钮，可以查询得到当前消息处理的速度。

2. 在 消息查询 界面，查询目标主题消息如下图所示。从图中可见，处理后的消息，从复杂的 Metadata 中成功提取出了所需的关键部分。



步骤3：连接器数据投递

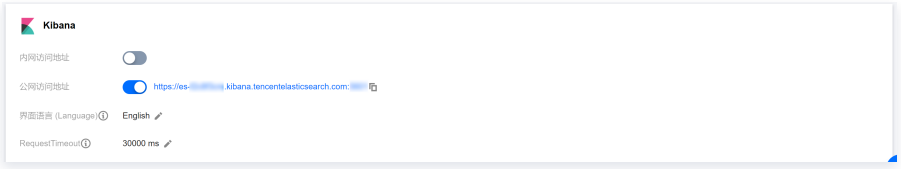
1. 登录 CKafka 控制台 。
2. 在左侧导航栏选择**连接器 > 任务管理 > 任务列表**，单击 **新建任务**。
3. 任务类型选择**数据流出**，目标类型选择 **Elasticsearch Service**，随后填写投递目标连接等信息。
4. 创建完成后，在数据流出任务列表如下图所示数据流出状态为健康时，代表任务创建成功。



5. 在数据流出任务监控界面，可以获取当前投递到 Elasticsearch 的速度。

步骤4：Elasticsearch 日志解析

1. 登录 Elasticsearch 控制台 ，打开 Kibana 的公网访问功能，以便于设置索引。



2. 打开 Kibana 界面，在左侧导航栏中依次选择 **Kibana > Discover**，进入 **Index Pattern** 界面。
3. 在索引页面创建能够匹配先前数据流出的索引，索引的数据类型由 Kibana 自动解析生成。

> **注意**
> 通过 **消息流出** 导入 Elasticsearch 中的消息，索引为消息所对应的 CKafka 实例的 **主题名称**。

4. 创建索引后，即可在 **Discover** 界面查询 Elasticsearch 解析后的日志，如下图所示。至此完成了由 TKE 日志到 Elasticsearch 整条链路的打通。

替换支撑路由（旧）

最近更新时间：2024-10-10 16:26:34

背景

为了给您提供更稳定可靠的服务，建议切换为新的支撑路由。

影响

因为需要更新　产、消费端的 bootstrap-server 地址，并且重启服务， 所以这　会影响业务的　产、消费短暂性中断。

操作步骤

1. 新建路由：新建　条　撑路由。

2. 切换路由：切换所有　产端、消费端的 CKafka 链接地址 bootstrap-server 为新创建的　撑环境路由；（　产端、消费端切换不分先后顺序，只要保证全部切换完）。
3. 重启服务：重启　产、消费端（　产、消费重启顺序　关，根据业务情况　　决策）。
4. 验证业务：持续观察　段时间业务是否稳定，建议持续观察3　时以上。
5. 删除路由：删除旧　撑路由。

回滚

验证业务阶段 如果发现异常，则需要回滚，回滚的前提是旧　撑路由未删除 具体回滚操作：

1. 所有　产、消费端替换旧　撑路由的地址。
2. 重启所有　产、消费端服务。
3. 验证业务。