TDMQ for CKafka

Practical Tutorial

Product Documentation

Contents

Practical Tutorial
Practical Tutorial of CKafka Client
Practical Tutorial of Production and Consumption

Last updated：2024-07-04 16:00:59

This document describes the practical tutorial of producing and consuming messages in TDMQ for CKafka to reduce the possibilities of errors in message consumption.

Messages Production

Recommendations for Topic Use

Configuration requirements: **It is recommended to use an integral multiple of nodes as replicas to reduce the data skew problem. The minimum number of in-sync replicas should be 2**, and the number of in-sync replicas cannot be equal to the number of topic replicas; otherwise, the failure of 1 replica will result in the inability to produce messages.
Creation method: You can choose whether to enable the switch for CKafka's automatic topic creation. After it is enabled, a topic with 3 partitions and 2 replicas will be automatically created when a topic that has not been created is produced or consumed.

Estimating Number of Partitions

To achieve as balanced data distribution as possible, it is recommended that the number of partitions is an integral multiple of the number of nodes. Also, based on the estimated traffic, set the number of partitions at 1 MB/s per partition. For example, for a topic with a throughput of 100 MB, it is recommended to set the number of partitions to 100.

Retry upon Failure

In a distributed environment, due to network and other issues, messages may occasionally fail to be sent. This is probably because the message has been sent successfully but the ACK mechanism failed, or because the message indeed has not been sent successfully.
You can set the following retry parameters based on your business needs:

Parameter	Description
retries	Number of retries, with the default value being 3. For applications that have zero tolerance

	for data loss, you can consider setting it to Integer.MAX_VALUE (valid and maximum).
retry.backoff.ms	Retry interval. We recommend you to set it to 1000.

By doing so, you will be able to deal with the issue where the broker's leader partition can't respond to producer requests immediately.

Asynchronous Sending

The sending API is asynchronous. If you want to receive the result of sending, you can access the sending results using the callback API provided in the send method.

One Producer Corresponding to One App

A producer is thread-safe and can send messages to any topics. Generally, we recommend that one application correspond to one producer.

Acks

Kafka's ACK mechanism refers to the producer's mechanism for acknowledgment of message sending. It is set to Acks on version Kafka 0.10.x or request.required.acks on version 0.8.x. The setting of Acks directly affects the throughput of the Kafka cluster and the reliability of messages.
The Acks parameters are described as follows:

Parameter	Description
acks=0	No response from the server is required. In this case, the performance is high, but the risk of data loss is great.
acks=1	A response will be returned after the primary server node writes successfully. In this case, the performance is moderate, and the risk of data loss is also moderate. A failure of the primary node may cause data loss.
acks=all	A response will be returned only after the primary server node writes successfully and the nodes in ISR sync successfully. The performance is poor, but the risk of data loss is small. Only a failure of both the primary and secondary nodes will cause data loss.

We recommend you to select acks=1 generally and select acks=all for important services.

Batch

A TDMQ for CKafka topic generally has multiple partitions. Before the producer client sends a message to the server, it needs to determine the topic and the partition to which the message should be sent. When sending multiple messages to the same partition, the producer client will package the messages into a batch and then send the batch to the server. Additional overheads will be incurred when the batch is processed. In general, a small batch will cause

the producer client to generate a large number of requests, which will cause the messages to queue up on the client and the server, lead to an increase in the corresponding device's CPU utilization, and thus increase the delays in message sending and consumption. A suitable batch size can reduce the number of requests sent to the server by the client during message sending, thereby improving the throughput and reducing the delay in message sending.
The batch parameters are described as follows:

Parameter	Description
batch.size	Size of cached messages sent to each partition (which is the sum of bytes of the messages rather than the number of messages). Once the set value is reached, a network request will be triggered, and the producer client will send the messages to the server in batch.
linger.ms	Maximum amount of time each message is cached. After this time elapses, the producer client will ignore the limit of the batch.size and immediately send the messages to the server.
buffer.memory	Once the total size of all cached messages exceeds this value, messages will be sent to the server, and the limits of batch.size and linger.ms will be ignored. The default value for buffer.memory is 32 MB, which ensures sufficient performance for a single Producer.

Note:
If you start multiple producers in the same JVM, it is likely that each producer will use 32 MB of cache space. In this case, OOM (Out of Memory) errors may occur, and you need to consider the value of buffer.memory in order to avoid OOM errors.
You can adjust the values of the parameters based on your specific business needs. The timing when the Producer client sends messages to the server in batches is determined by both batch.size and linger.ms, and you can adjust it based on your specific business needs. To enhance the performance of sending and ensure service stability, it is recommended to set batch.size=16384 and linger.ms=1000.

Key and Value

Each message in TDMQ for CKafka has two fields: key (message identifier) and value (message content).
For ease of tracking, set a unique key for each message, which allows you to track a message and print its sending and consumption logs to learn about its production and consumption conditions.
If you want to send a large number of messages, we recommend you to use the sticky partitioning policy instead of setting keys.

Sticky Partitioning

Only messages sent to the same partition will be placed in the same batch, so one factor that determines how a batch will be formed is the partitioning policy set by the Kafka producer. The Kafka producer allows you to choose a partition that suits your business by setting the partitioner implementation class. If a key is specified for a message, the default policy of the Kafka producer is to hash the message key and then select a partition based on the result of hashing to ensure that messages with the same key are sent to the same partition.

If no Key is specified for a message, the default policy of TDMQ for Kafka below v2.4 is to use all partitions in the topic in loops and send the message to each partition in a round robin manner. However, this default policy has a poor batch performance and may produce a large number of small batches, which increases actual delays. As it was inefficient in partitioning messages without a key, Kafka 2.4 introduced the sticky partitioning policy.
The sticky partitioning policy mainly addresses the problem of small batches caused by the distribution of messages without a key into different partitions. The main practice is to randomly select another partition and use it as much as possible for subsequent messages after the batch is completed for a partition. With this policy, messages will be sent to the same partition in the short run, but from the perspective of the entire execution, messages will be evenly sent to different partitions, which helps avoid skewed partitions while reducing delays and improving the overall service performance.
If a Kafka producer client is used on v2.4 or later, the default partitioning policy is sticky partitioning. If you use an older producer client, you can implement a partitioning policy on your own based on how the sticky partitioning policy works and then make it take effect through the partitioner.class parameter.
For more information on how to implement the sticky partitioning policy, see the following implementation of Java code. The code is implemented by switching from one partition to another at certain time intervals.

```
public class MyStickyPartitioner implements Partitioner {

    // Record the time of the last partition switch.
    private long lastPartitionChangeTimeMillis = 0L;
    // Record the current partition.
    private int currentPartition = -1;
    // Partition switch time interval, which can be selected based on your business
    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, b

        // Get the information of all partitions.
        List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
        int numPartitions = partitions.size();
```

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

            // Determine the current available partitions.
            if (availablePartitionSize > 0) {
                handlePartitionChange(availablePartitionSize);
                return availablePartitions.get(currentPartition).partition();
            } else {
                handlePartitionChange(numPartitions);
                return currentPartition;
            }
        } else {
            // For messages with a key, a partition will be selected based on the h
            return Utils.toPositive(Utils.murmur2(keyBytes)) % numPartitions;
        }
    }

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

        // After the partition switch time interval elapses, the next partition wil
        if (currentTimeMillis - lastPartitionChangeTimeMillis >= partitionChangeTim
            || currentPartition < 0 || currentPartition >= partitionNum) {
            lastPartitionChangeTimeMillis = currentTimeMillis;
            currentPartition = Utils.toPositive(ThreadLocalRandom.current().nextInt
        }
    }

    public void close() {}

}
```

Order in Partition

Within a single partition, messages are stored in the order in which they are sent. Each topic is divided into a number of partitions. If messages are distributed to different partitions, the cross-partition message order cannot be ensured. If you want messages to be consumed in the sending order, you can specify keys for such messages on the producer. If such messages are sent with the same key, CKafka will select a partition for their storage based on the hash of the key. As a partition can be listened on and consumed by only one consumer, messages will be consumed in the sending order.

Data skew

Kafka Broker data skew problems are often caused by uneven partition distribution or uneven key distribution of producer-sent data, leading to several types of issues:

1. Overall traffic is not rate-limited, but individual nodes have rate limits;
2. Some nodes are overloaded too quickly, leading to low overall Kafka usage and affecting total throughput.
To avoid such issues, you can optimize them in the following ways:
3. Use a reasonable number of partitions, ensuring that the number of partitions is an integral multiple of the number of nodes.
4. Use a reasonable partitioning policy, for example: RoundRobin, Range, and Sticky or a custom partitioning policy to deliver messages evenly.
5. Check if keys are used for sending. If so, try to design a policy to make keys more evenly distributed across partitions.

Message Consumption

Basic Message Consumption Process

1. Poll the data.
2. Execute the consumption logic.
3. Poll the data again.

Load Balancing

Each consumer group can contain multiple consumers with the same group.id value. In this way, consumers in the same consumer group consume the same subscribed topic.
For example, if consumer group A subscribes to topic A and enables three consumer instances C1, C2, and C3, then each message sent to topic A will be eventually delivered to only one of the three instances. By default, CKafka will evenly distribute messages to the consumer instances to achieve consumption load balancing.
The internal principle of CKafka load balancing is to evenly allocate the partitions of the subscribed topic to each consumer. Therefore, the number of consumers should not exceed the number of partitions; otherwise, there will be consumer instances that are not allocated any partitions and are in an idle state. Try to ensure that the number of consumers can be evenly divided by the total number of partitions. Apart from the first startup, rebalance will be triggered whenever consumer instances are restarted, increased, decreased, or the number of partitions changes.

Frequent Rebalancing

If rebalance is triggered frequently, there may be several possible causes:
1. The consumer takes a long time to process messages.
2. The consumption of a particular abnormal message causes the consumer to block or fail.
3. Heartbeat timeout triggers a rebalance.
4. For client versions before v0.10.2: The consumer did not have an independent thread to maintain the heartbeat, instead, heartbeat maintenance was coupled with the poll API. As a result, if the user experiences consumption congestion, it will lead to consumer heartbeat timeout, and trigger a rebalance. For client versions v0.10.2 and later: If

the consumption takes too long, and no messages is polled within a certain period (value set by max.poll.interval.ms, with the default value being 5 minutes), the client will actively leave the Queue, triggering a rebalance.
This can be solved by methods such as optimizing consumption processing to increase consumption speed and adjusting parameters:
1. The consumer side needs to be consistent with the broker version.
2. Adjust parameter values according to the following instructions:
session.timeout.ms: For versions before v0.10.2, increase this parameter value appropriately, making it greater than the time it takes to consume a batch of data and not exceed 30 s. It is recommended to set the value to 25 s for v0.10.2 and default value of 10 s for later versions.
max.poll.records: Reduce this parameter value. The recommended value is much less than the product of <the number of messages consumed per second per thread> <the number of consumption threads> <max.poll.interval.ms>.
max.poll.interval.ms: This value should be greater than <max.poll.records> / (<the number of messages consumed per second per thread> x <the number of consumption threads>).
3. Increase the client's consumption speed as much as possible, handle consumption logic in a separate thread, and monitor for any time-consuming operations.
4. Reduce the number of topics that a group subscribes to. It's best for a group not to subscribe to more than 5 topics, ideally subscribing to only one topic.

Subscription Relationship

We recommend that all consumer instances in the same consumer group subscribe to the same topic so as to facilitate troubleshooting.

A Consumer Group Subscribes to Multiple Topics

A consumer group can subscribe to multiple topics, and the messages in such topics will be consumed evenly by the consumers in the consumer group. For example, if consumer group A subscribes to topics A, B, and C, then messages in the three topics will be consumed evenly by the consumers in consumer group A.
Below is the sample code to make a consumer group subscribe to multiple topics:

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

A Topic is Subscribed to by Multiple Consumer Groups

A topic can be subscribed to by multiple consumer groups, and the consumers in such consumer groups independently consume all messages in the topic. For example, if both consumer groups A and B subscribe to topic A, each message sent to topic A will be delivered to both the consumer instances in consumer group A and the consumer instances in consumer group B, and the two processes are independent of each other.

One Consumer Group Corresponding to One Application

We recommended that one consumer group corresponds to one application, i.e., different applications correspond to different sets of code. If you need to write different sets of code in the same application, prepare multiple kafka.properties files, for example, kafka1.properties and kafka2.properties.

Consumer Offset

Each topic has multiple partitions, and each partition counts the total number of current messages, which is called the MaxOffset.

Consumers in TDMQ for CKafka consume each message in a partition in order, recording the number of messages consumed, known as the ConsumerOffset.

The number of remaining unconsumed messages (also known as message backlog) = MaxOffset - ConsumerOffset.

Offset Commit

There are two parameters related to TDMQ for CKafka consumers:

enable.auto.commit: The default value is true.

auto.commit.interval.ms: The default value is 5,000, i.e., 5 s.

As a result of the combination of the two parameters, before the client polls the data, it will always check the time of the last committed offset first, and if the time defined by the auto.commit.interval.ms parameter has elapsed, the client will start an offset commit.

Therefore, if enable.auto.commit is set to true, it is always necessary to ensure that the data polled last time has been consumed before data polling; otherwise, the offset may be skipped.

If you want to control offset commits by yourself, set enable.auto.commit to false and call the commit (offsets) function.

Note:

Try to avoid committing offsets too frequently; otherwise, it will cause high CPU usage on the broker, affecting normal services. For example, set auto.commit.interval.ms to 100 ms for automatic offset commit. For manually offset commit, commit an offset for every message consumed in high throughput scenarios.

Offset Reset

The ConsumerOffset will be reset in the following two scenarios:

The server has no committed offsets (for example, when the client is started for the first time).

A message is pulled from an invalid offset (for example, the MaxOffset in a partition is 10, but the client pulls a message from offset 11).

For a Java client, you can configure a resetting policy through auto.offset.reset. There are three policies:

latest: Consumption will start from the maximum offset.

earliest: Consumption will start from the minimum offset.

none: No resetting will be performed.

Note:

We recommend you to set the resetting policy to latest instead of earliest, so as to avoid starting consumption from the beginning when the offset is invalid, as that may cause a lot of repetitions.

If you manage the offset by yourself, you can set the policy to none.

Message Pull

In the consumption process, the client pulls messages from the server. When the client pulls large messages, you should control the pulling speed and pay attention to the following parameters:
max.poll.records: Set it to 1 if a single message exceeds 1 MB in size.
max.partition.fetch.bytes: Set it to a value slightly greater than the size of a single message.
fetch.max.bytes: Set it to a value slightly greater than the size of a single message.
When messages are consumed over the public network, a disconnection may often occur due to the bandwidth limit of the public network. In this case, you should control the pulling speed and pay attention to the following parameters:
fetch.max.bytes: It is recommended to set it to half of the public network bandwidth (note that the unit of this parameter is bytes, while the unit of public network bandwidth is bits)
max.partition.fetch.bytes: We recommend you to set it to half of the public network bandwidth (note that the unit of this parameter is bytes, while the unit of the public network bandwidth is bits).

Pulling Large Messages

In the consumption process, the client pulls messages from the server. When pulling large messages, you should control the pulling speed and change the configuration:
max.poll.records: The maximum number of messages polled each time. If a single message exceeds 1 MB, it is recommended to set it to 1.
fetch.max.bytes: Set it to a value slightly greater than the size of a single message.
max.partition.fetch.bytes: Set it to a value slightly greater than the size of a single message.
The essence of pulling large messages is to fetch them one by one.

Duplicate Messages and Consumption Idempotency

TDMQ for CKafka consumption uses the at-least-once semantics, that is, a message is delivered at least once, which guarantees that messages will never be lost. However, messages may be delivered more than once. Network issues and client restarts may cause a few duplicate messages. If the application consumer is sensitive to duplicate messages (such as orders and transactions), idempotency should be implemented for the messages.
Taking a database application as an example, the common practice is as follows:
When sending a message, pass in the key as the unique ID.
When consuming a message, determine whether the key has already been consumed; if so, ignore it; otherwise, consume it once.
Of course, if the application itself is not sensitive to a small number of duplicate messages, idempotency is not necessary.

Consumption Failure

In TDMQ for CKafka, messages are consumed from partitions one by one in sequence. If the consumer fails to execute the consumption logic after getting a message, for example, when dirty data is stored on the application server, the message will fail to be processed, and human intervention will be required. There are two methods for dealing with this situation:
The consumer will keep trying to execute the consumption logic upon failure. This method may cause the consumer thread to be jammed by the current message and lead to message heap.
As TDMQ for CKafka is not designed to process failed messages, in practice, it will typically print failed messages or store them in a service (such as a dedicated topic created for storing failed messages), so that you can regularly check failed messages, analyze the causes of failures, and process accordingly.

Consumption Delay

In the consumption process, the client pulls messages from the server. In general, if the client can consume messages timely, there will be no significant delays. If a high delay is detected, you should first check whether messages heap up and speed up consumption accordingly.

Consumption Heap

The common causes of message heap include the following:
Consumption is slower than production. In this case, you should speed up consumption.
Consumption is jammed.
After a message is received, the consumption end will execute the consumption logic and generally make some remote calls. If it waits for the results at the same time, the consumption process may be jammed.
It should be ensured as much as possible that the consumer will not jam the consumption threads. If it needs to wait for the call results, we recommend you to set a wait timeout period, so that the consumer will be treated as a failure after the timeout period elapses.

Speeding up Consumption

Increase the number of consumer instances to improve parallel processing capabilities. If the ratio of consumers to partitions reaches 1:1, you can increase the number of partitions. (Note: In scenarios where flink automatically maintains partitions, new partitions cannot be automatically detected. Code modifications and a restart thereafter may be required after new partitions are added.) You can increase it directly in the process (ensuring each instance corresponds to one thread), or deploy multiple consumer instance processes.
Note:
After the number of instances exceeds the number of partitions, you can't add more instances; otherwise, there will be idle consumer instances.
Increase the number of consumer threads.
1. Define a thread pool.
2. Poll the data.
3. Commit the data into the thread pool for concurrent processing.

4. Poll the data again after a successful concurrent processing result is returned.

Socket Buffers

The default value of the receive.buffer.bytes parameter in Kafka 0.10.x is 64 KB, while the default value of the socket.receive.buffer.bytes parameter in Kafka 0.8.x is 100 KB.
Both the default values are too small for high-throughput environments, especially when the bandwidth-delay product of the network between the broker and the consumer is greater than that of the local area network (LAN).
For high-bandwidth networks with a delay of 1 ms or more and (such as 10 Gbps or higher), it is recommended to set the socket buffer size to either 8 or 16 MB.
Even if you don't have enough memory, you should consider setting this parameter to at least 1 MB. You can also set it to -1, so that the underlying operating system will adjust the buffer size based on the actual network conditions.
However, for consumers that need to start hot partitions, automatic adjustment may not be that fast.

Message Broadcasting

CKafka currently does not support the semantics of message broadcasting. It can be simulated by creating different groups.

Message Filtering

CKafka does not have semantics for message filtering. In practice, you can adopt the following two methods:
If there are not many categories to filter, use multiple topics to achieve the purpose of filtering.
If there are many categories to filter, it is best to filter on the client's business layer.
In practice, choose method according to specific business circumstances, or use a combination of the two methods mentioned above.

No Consumption in Some Partitions

During the consumption process, the consumer may be online but the offsets of some partitions does not progress.
Possible causes are as follows:
1. An exception message is reported, which could be an oversized message or a format exception, causing the consumer to convert it into a business offset when pulling messages.
2. When using public network with limited bandwidth, pulling large messages immediately fills up the bandwidth, resulting in failure to pull messages within the timeout period.
3. Consumer hangs, leading to no message pulling.
Solution:
Shut down the consumer, set the offset in the CKafka console to skip some exception messages, or optimize the consumption code and restart consumer consumption thereafter.

Confluent Go SDK

Last updated：2024-07-04 16:00:59

Overview

TDMQ for CKafka is a distributed stream processing platform designed for building real-time data pipelines and streaming applications. It boasts high throughput, low latency, scalability, and fault tolerance.
Sarama: A Kafka library developed by Shopify, providing features such as producers, consumers, and partition consumers. The library performs well, and has an active community support.
Confluent-Kafka-Go: A Kafka library developed by Confluent, providing high-level APIs that are easy to use. Based on the librdkafka C library, its performance is excellent, though it can be somewhat complex to install and use.
This document describes the key parameters, practical tutorials, and FAQs of the Confluent Go client mentioned above.

Producer Practice

Version Selection

When the Confluent Go SDK is used, you can specify the address of the Kafka cluster through the configuration parameter bootstrap.servers, while the version of the Broker can be set through the api.version.request parameter, enabling the Confluent Go SDK to automatically detect the Broker's version at startup.

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

Producer Parameters and Optimization

Producer Parameters

Confluent Go is developed based on librdkafka. When writing to Kafka using the Confluent Go Client, the configuration parameters passed to librdkafka involve the following key parameters, with the relevant parameters and default values as follows:

```

package main

import (
```

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

func main() {
    config := &kafka.ConfigMap{
        "bootstrap.servers":                "localhost:9092",
          "acks":                             -1,          //ack mode, with the def
        "client.id":                        "rdkafka",   //Client ID.
        "compression.type":                 "none",     // Specify compression type
        "compression.level":                -1,         //Compression level.
        "batch.num.messages":               10000,      // By default, a batch can
        "batch.size":                       1000000,    //The limit for the total s
        "queue.buffering.max.ms":           5,          //The delay before transmit
        "queue.buffering.max.messages":     100000,     //The total number of messa
        "queue.buffering.max.kbytes":       1048576,    //MessageSets for the Produ
        "message.send.max.retries":         2147483647, //Number of retries, 2,147,
        "retry.backoff.ms":                 100,        //Retry interval, with the
        "socket.timeout.ms":                60000,      //Session timeout period, w

    }

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

    // Use producer to send messages and perform other operations...

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

Parameter Description and Optimization

acks Parameter Optimization

The acks parameter is used to control the confirmation mechanism when the producer sends messages. The default value of this parameter is -1, which means that after the message is sent to the leader broker, it is not returned until the Leader confirmation and the corresponding follower messages are all written. The acks parameter can be set to values of 0, 1, or -1. In cross-availability zone scenarios, and for topics with a higher number of replicas, the value of the acks parameter will affect the message's reliability and throughput.
In some online business message scenarios, where the throughput requirements are not high, you can set the acks parameter to -1 to ensure that the message is received and confirmed by all replicas before returning, thereby improving the message's reliability.

In scenarios involve big data, such as log collection, or offline computing, where high throughput (i.e., the volume of data written to Kafka per second) is required, you can set the acks to 1 to increase throughput.

buffering Parameter Optimization (Caching)

By default, for transmitting the same volume of data, a single request's network transmission can effectively reduce related computation and network resources compared to multiple requests, thereby improving the overall write throughput.
Therefore, you can set parameter to optimize the client's message sending throughput. For Confluent kafka Go, a default batching time of 5 ms is provided to buffer messages. If the message is small, you can increase the queue.buffering.max.ms time to an appropriate value.

Compression Parameter Optimization

Confluent Go supports the following compression parameters: none, gzip, snappy, lz4, and zstd.
In the Confluent Kafka Go client, the following compression algorithms are supported:
none: No compression algorithm.
gzip: Compress by GZIP algorithm.
snappy: Compress by Snappy algorithm.
lz4: Compress by LZ4 algorithm.
zstd: Compress by ZSTD algorithm.
To use a compression algorithm in the producer client, you need to set the compression.type parameter when creating the producer. For example, you can set compression.type to lz4 to compress by LZ4 algorithm. The compression algorithm's CPU compression and CPU decompression occur on the client side, which is a way of optimizing by trading computing power for bandwidth. However, the broker incurs extra computation costs for validating compressed messages, especially for Gzip compression, resulting in significant server's computation costs. The increased computation can lead to lower message processing capabilities for the broker and lower bandwidth throughput as a result, which may not be worth it in some cases. In such cases, the following approach is recommended:
1. Independently compress message data on the producer side to generate packed data compression: messageCompression, and store the compression method in the message's key:

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

2. On the producer side, send messageCompression as a normal message.
3. On the consumer side, read the message key to access the compression method used, and perform independent decompression.

Creating Producer Instance

If your application requires higher throughput, you can use asynchronous producer to increase the speed of message sending and utilize batch message sending to reduce network overhead and IO consumption. If your application

requires higher reliability, you can use synchronous producer to ensure successful message delivery. Additionally, the ACK acknowledgement mechanism and transaction mechanism can be employed to guarantee message reliability and consistency. For specific parameter optimization, refer to Producer Parameters and Optimization.

```
package main

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

func main() {
    // Configure 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
    }

    // Send message.
    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.Pa
            Value:          []byte(value),
        }
        p.Produce(message, nil)
    }

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

Consumer Practice

Consumer Parameters and Optimization

Consumer Parameters

```
package main

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

func main() {
    // Configure Kafka Consumer.
    c, err := kafka.NewConsumer(&kafka.ConfigMap{
        "bootstrap.servers": "localhost:9092",
        "group.id":          "test-group",
        "auto.offset.reset":"earliest",
        "fetch.min.bytes":1, // Minimum pulling byte count.
        "fetch.max.bytes":52428800, // Maximum pulling byte count.
        "fetch.wait.max.ms":"500", //If there are no new messages to consume, wait
        "enable.auto.commit":true, //Enable automatic offset commit, with the defau
        "auto.commit.interval.ms":5000, //Interval between automatic offset commit,
        "max.poll.interval.ms": 300000, // Maximum delay between two poll operation
        "session.timeout.ms": 45000, //Session time, with the default value being 4
        "heartbeat.interval.ms": 3000, //Heartbeat time, with the default value bei
    })
    if err != nil {
        fmt.Printf("Failed to create consumer: %s\\n", err)
        return
    }

    //  Subscribe to topics.
    c.SubscribeTopics([]string{"test-topic"}, nil)

    // Manually commit offset.
    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)
        }
```

```
        }

        // Close Kafka consumer.
        c.Close()
    }
```

Parameter Description and Optimization

1. max.poll.interval.ms is a configuration parameter for Kafka consumer. It specifies the maximum delay between two poll operations by the consumer. Its primary function is to control the consumer's liveness, i.e., to determine whether the consumer is still active. If the consumer does not perform a poll operation within the time specified by max.poll.interval.ms, then Kafka considers this consumer to have failed and triggers a rebalance for the consumer. Adjust this parameter according to the actual consumption speed. If it is too low, the consumer may frequently trigger rebalances, increasing the burden on Kafka; if it is too high, Kafka may not be able to promptly detect issues with the consumer, thereby affecting message consumption.
2. For general consumption, issues mainly involves frequent rebalancing and consumption thread blocking. For parameter optimization, refer to the following method:
2.1 session.timeout.ms: For versions before v0.10.2, increase this parameter value appropriately, make it greater than the time it takes to consume a batch of data and not exceed 30 s. The recommended value is 25 s; for v0.10.2 and later versions, use the default value of 10 s.
2.2 heartbeat.interval.ms: Default value is 3 s. This value should be less than session.timeout.ms / 3.
2.3 max.poll.interval.ms: Default value is 5 minutes. If there are many partitions and consumers, it is recommended to appropriately increase this value. It should be greater than <max.poll.records> / (<number of messages consumed per second per thread> x <number of consumption threads>).
Note:
If you want to process messages synchronously, that is, pull a message, process it, and then pull the next one, you need to make the following modifications:
Increase the MaxProcessingTime according to your needs.
Monitor for processing times that exceed the MaxProcessingTime, sample and print timeout durations.
3. For automatic offset commit requests, it's recommended not to set auto.commit.interval.ms below 1,000 ms, as too frequent offset requests can cause high broker CPU usage, affecting the read and write operations of other normal services.

Creating Consumer Instance

Confluent Go provides a subscription model to create consumers, and includes manual-commit offset and auto-commit offset as two offset commit methods.

Auto-Commit Offsets

Auto-commit offsets: After pulling messages, the consumer automatically commit offsets without manual intervention. The advantage of this method is it's easy to use, but it may lead to duplicate message consumption or loss.

```
package main

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

func main() {
    // Configure Kafka consumer
    c, err := kafka.NewConsumer(&kafka.ConfigMap{
        "bootstrap.servers": "localhost:9092",
        "group.id":          "test-group",
        "auto.offset.reset": "earliest",
        "enable.auto.commit": true,    //Whether to enable auto-commit offset. True
        "auto.commit.interval.ms": 5000, //Interval for auto-commit offsets. Value
        "max.poll.interval.ms": 300000, //Maximum wait time for the consumer in a s
        "session.timeout.ms": 10000, //Specify the session timeout period between t
        "heartbeat.interval.ms": 3000, //Specify the interval for the consumer to s
    })
    if err != nil {
        fmt.Printf("Failed to create consumer: %s\\n", err)
        return
    }

    // Subscribe to topics.
    c.SubscribeTopics([]string{"test-topic"}, nil)

    // Automatically commit offset.
    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)
        }
    }

    // Close Kafka Consumer.
```

```
    c.Close()
```

Manual-Commit Offsets

Manual-commit offsets: After processing messages, consumers need to manually commit offsets. The advantage of this method is that it allows for precise control over offset commit, avoiding duplicate message consumption or loss. It should be noted that manual-commit offset can lead to high broker CPU usage, affecting performance. As message volume increases, CPU consumption will be significantly high, affecting other features of the broker. Therefore, it is recommended to commit an offset after a certain number of messages.

```
package main

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

func main() {
    // Configure 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
    }

    // Subscribe to topics.
    c.SubscribeTopics([]string{"test-topic"}, nil)

    // Manually commit offsets.
    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)
        }
    }

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

Sarama Go

Last updated：2024-07-04 16:00:59

Overview

TDMQ for CKafka is a distributed stream processing platform designed for building real-time data pipelines and streaming applications. It boasts high throughput, low latency, scalability, and fault tolerance.
Sarama: A Kafka library developed by Shopify, offering features such as producers, consumers, and partition consumers. The library performs well, and has an active community support.
Confluent-Kafka-Go: A Kafka library developed by Confluent, featuring high-level APIs that are easy to use. Based on the librdkafka C library, its performance is excellent, though it can be somewhat complex to install and use.
This document mainly describes key parameters, practical tutorials, and FAQs of the aforementioned Sarama Go client.

Producer Practices

Version Selection

When the Sarama client version is selected, it is necessary to ensure that the selected version is compatible with the Kafka broker version. The Sarama library supports multiple Kafka protocol versions. Specify the protocol version to use by setting config.Version. The common correspondence between Kafka protocol versions and Sarama library versions is as follows. For the latest version, see Sarama Version.

Kafka Version	Sarama Library Version	Sarama Protocol Version Constants
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 or later	Recommended >=1.42.1	sarama.V2_8_0_0-sarama.V3_6_0_0

The Sarama library versions listed above represent the minimum versions that support the corresponding Kafka protocol versions. For optimal performance and new features, it is recommended to use the latest version of Sarama. When using the latest version, clients can specify the protocol version compatible with your Kafka broker by setting config.Version. The setting method is as follows, be sure to set the version before use, otherwise, there will be unexpected incompatibility issues:

```
config := sarama.NewConfig()
config.Version = sarama.V2_7_0_0 // Set the protocol version according to the actua
```

Producer Parameters and Optimization

Producer Parameters

When using the Sarama Go client to write to Kafka, you need to configure the following key parameters. Parameters and their default values are as follows:

```
config := sarama.NewConfig()
sarama.MaxRequestSize = 100 * 1024 * 1024  // Maximum request size, with the defaul
sarama.MaxResponseSize = 100 * 1024 * 1024 // Maximum response size, with the defau

config.Producer.RequiredAcks = sarama.WaitForLocal // The default value is sarama.W

config.Producer.Retry.Max = 3                      // Maximum retry count for produ
config.Producer.Retry.Backoff = 100 * time.Millisecond // Wait time between produce
```

```
config.Producer.Return.Successes = false         // Return successful messages, wit
config.Producer.Return.Errors = true            // Return failed messages, with the

config.Producer.Compression = CompressionNone  // Compress messages before sending,
config.Producer.CompressionLevel = CompressionLevelDefault // Specify compression l

config.Producer.Flush.Frequency = 0 //Time for caching messages by the producer, wi
config.Producer.Flush.Bytes = 0       // Triggers a broker request when reaching a
config.Producer.Flush.Messages = 0      // Forces a broker request upon reaching a
config.Producer.Flush.MaxMessages = 0      // Messages can be cached at maximum, th

config.Producer.Timeout = 5 * time.Second       // Timeout duration.

config.Producer.Idempotent = false                    // Whether idempotence is re
config.Producer.Transaction.Timeout = 1 * time.Minute // Transaction timeout durati
config.Producer.Transaction.Retry.Max = 50            // Maximum transaction retry
config.Producer.Transaction.Retry.Backoff = 100 * time.Millisecond
config.Net.MaxOpenRequests = 5                     // Default value is 5, the numbe
config.Producer.Transaction.ID = "test"            // Transaction ID.

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

Parameter Description and Optimization

RequiredAcks Parameter Optimization

The RequiredAcks parameter is used to control the acknowledgement mechanism when the producer sends messages. The default value is WaitForLocal, which means that once the message is sent to the Leader Broker and the Leader confirms the message has been written, it is returned immediately. The RequiredAcks parameter also has the following optional values:

NoResponse: Without waiting for any confirmation, return directly.

WaitForLocal: Wait for the Leader replica to confirm the write, then return.

WaitForAll: Wait for the Leader replica and the relevant Follower replicas to confirm the write, then return.

From the above, it's clear that in scenarios involving cross-availability zones, as well as in Topic with a higher number of replicas, the choice of RequiredAcks parameter value can affect the reliability and throughput of messages.

Therefore:

In scenarios involving online business messages, where throughput demands are not high, you can set the RequiredAcks parameter to WaitForAll to ensure that messages are received and confirmed by all replicas before returning and to increase reliability.

In scenarios of big data, like log collection, or offline computing, where a high throughput (i.e., data written to Kafka per second) is required, you can set the RequiredAcks to WaitForLocal to enhance throughput.

Flush Parameter Optimization (Caching)

By default, when the same amount of data is transmitted, compared to multiple requests, a single request can effectively reduce computations and network resources, thus improving the overall write throughput. Therefore, you can optimize the client's message-sending throughput by setting this parameter. In high-throughput scenarios, set parameters in conjunction with computation:
Bytes are recommended to be set at 16 K, aligned with Kafka's standard Java SDK definition, and estimated single message size is 1 K (1024) bytes, hence the following parameters for Messages and MaxMessages should be set:
The calculation method for Frequency is as follows: estimated traffic is 16 MB, with 16 partitions, then per partition write traffic per second would be: 16 x 1024 x 1024 / 16 = 1 x 1024 x 1024 = 1 MB, i.e. 1 MB of traffic per second per partition. Assuming a request size is 16 K for data transmission, to achieve 1 MB of traffic transmission in 1 s, it requires 1x1024x1024/16/1024 = 64 requests, thus Frequency <= 15.62 ms(1000/64).
In reality, since the business traffic is not continuously produced, during off-peak hours, it might still hit 16ms but not cache much data. Therefore, in high-throughput situations, the conditions can be simplified, based on Bytes, and Frequency can be appropriately increased. For example, if a delay increase of 500 ms is acceptable, then it can be set to 500 ms, because at this time, if the data volume hits greater than or equal to Bytes, requests will be sent based on the Bytes condition.

```
config.Producer.Flush.Frequency = 16 // Time for the producer to cache messages, wi
config.Producer.Flush.Bytes = 16*1024       // Triggers a broker request upon reach
config.Producer.Flush.Messages = 17      // Forces a broker request upon reaching a
config.Producer.Flush.MaxMessages = 16  // Set to 16 messages. In fact, as the mess
                                        // Because hitting any of the Frequency, By
```

Transaction Parameter Optimization

```
config.Producer.Idempotent = true                       // Whether idempotence is re
config.Producer.Transaction.Timeout = 1 * time.Minute // Default transaction timeou
config.Producer.Transaction.Retry.Max = 50             // Transaction retry duration
config.Producer.Transaction.Retry.Backoff = 100 * time.Millisecond
config.Net.MaxOpenRequests = 5                      //Number of requests sent at onc
config.Producer.Transaction.ID = "test"             // Transaction ID.
```

It is important to note that ensuring the exactly once semantics for transactions requires additional computational resources. Therefore, the selection of config.Net.MaxOpenRequests must be less than or equal to 5. The Broker's ProducerStateManager instances will cache the most recent 5 batch data sent by each PID on each Topic-Partition. If the customer wants to maintain a certain level of throughput in addition to transactions, set this value to 5 and appropriately increase the transaction timeout period to accommodate delays caused by network jitter under high load.

Compression Parameter Optimization

Sarama Go supports the following compression parameters:

```
config.Producer.Compression = CompressionNone  // Compress messages before sending,
config.Producer.CompressionLevel = CompressionLevelDefault // Specify the compressi
```

In the Sarama Kafka Go client, the following compression configurations are supported:

1. sarama.CompressionNone: Do not compress.
2. sarama.CompressionGZIP: Compress by GZIP.
3. sarama.CompressionSnappy: Compress by Snappy.
4. sarama.CompressionLZ4: Compress by LZ4.
5. sarama.CompressionZSTD: Compress by ZSTD.

To use message compression in the Sarama Kafka Go client, you should set the config.Producer.Compression parameter when creating a producer. For example, to use the LZ4 compression algorithm, set config.Producer.Compression to sarama.CompressionLZ4. As an optimization method that exchanges computation for bandwidth, message compression and decompression occurs on the client side. However, due to additional computation cost of the Broker's behavior in verifying compressed messages, especially with GZIP compression, the verification computation cost for the Broker can be significant, which is not worthwhile in some cases. Due to increased computation, message processing capabilities of the Broker may be lower, resulting in lower bandwidth throughput. Under circumstances of low throughput or low specification services, it is not recommended to use message compression. If compression is necessary, the following method is recommended:

1. Independently compress message data on the Producer side to generate packed data compression: messageCompression, and store the compression method in the message's key:

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

2. On the Producer side, send messageCompression as a normal message.
3. On the Consumer side, read the message key to access the compression method used, and decompress independently.

Creating Producer Instance

If an application requires higher throughput, use an asynchronous producer to increase the speed of message sending. At the same time, send batch messages to reduce network overhead and IO consumption. If an application demands higher reliability, a synchronous producer can be used to ensure successful message delivery. Additionally, employ the ACK acknowledgement mechanism and transaction mechanism to guarantee message reliability and consistency. For specific parameter optimization, see Producer Parameters and Optimization.

Synchronous Producer

In the Sarama Kafka Go client, there are two types of producers: synchronous and asynchronous. Their main differences lie in the method of sending messages and processing message results. Synchronous Producer: The synchronous producer blocks the current thread when sending messages, until the message has been sent and acknowledged by the server. As a result, the throughput of a synchronous producer is lower, but it allows immediate knowledge of whether the message was successfully sent. Example:

```
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, offse
    }
}
```

Asynchronous Producer

Asynchronous Producer: The asynchronous producer does not block the current thread when sending messages. Instead, it places the message into an internal sending queue and then returns immediately. Therefore, the throughput of an asynchronous producer is higher, but it requires a callback function to process the message results.

```
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.Partitio
    case err := <-producer.Errors():
        log.Printf("Failed to send message: %v", err)
    }
}
```

Consumer Practice

Version Selection

When a Sarama client version is selected, it is necessary to ensure that the chosen version is compatible with the Kafka broker version. The Sarama library supports multiple Kafka protocol versions, which can be specified by setting config.Version.

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

Consumer Parameters and Optimization

```
config := sarama.NewConfig()
config.Consumer.Group.Rebalance.Strategy = sarama.NewBalanceStrategyRange // The de
config.Consumer.Offsets.Initial = sarama.OffsetNewest // Without a committed offset
config.Consumer.Offsets.AutoCommit.Enable = true // Whether auto-commit for offsets
config.Consumer.Offsets.AutoCommit.Interval = 1 * time.Second // Auto-commit interv
config.Consumer.MaxWaitTime = 250 * time.Millisecond // Client's waiting time when
config.Consumer.MaxProcessingTime = 100 * time.Millisecond
config.Consumer.Fetch.Min = 1 // Minimum byte size of messages accessed in consumpt
config.Consumer.Fetch.Max = 0 // Maximum byte size for consumption requests. Defaul
config.Consumer.Fetch.Default = 1024 * 1024 // Default byte size of messages for co
config.Consumer.Return.Errors = true

config.Consumer.Group.Rebalance.Strategy = sarama.NewBalanceStrategyRange // Set th
config.Consumer.Group.Rebalance.Timeout = 60 * time.Second // Set the timeout durat
config.Consumer.Group.Session.Timeout = 10 * time.Second // Set the timeout duratio
config.Consumer.Group.Heartbeat.Interval = 3 * time.Second // Heartbeat timeout dur
config.Consumer.MaxProcessingTime = 100 * time.Millisecond // Timeout duration for
```

Parameter Description and Optimization

General consumption issues mainly involve frequent rebalance times and consumption thread blocking. See the following for parameter optimization:

config.Consumer.Group.Session.Timeout: For versions earlier than v0.10.2, increase this parameter to an appropriate value, making it greater than the time taken to consume a batch of data and not exceed 30 s. It is recommended to set it to 25 s. For v0.10.2 and later versions, use the default value of 10 s.

config.Consumer.Group.Heartbeat.Interval: Default is 3 s. Set this value and make sure it is less than Consumer.Group.Session.Timeout/3.

config.Consumer.Group.Rebalance.Timeout: Default is 60 s. If the number of partitions and consumers is large, it is recommended to appropriately increase this value.

config.Consumer.MaxProcessingTime: This value should be greater than the <max.poll.records> / (<number of records consumed per second per thread> x <number of consumer threads>).

Note:

Increase the MaxProcessingTime according to your needs.

Monitor for processing times that exceed the MaxProcessingTime, and log the instances of timeouts.

Creating Consumer Instance

Sarama offers a subscription model to create consumers. It provides two ways to commit offsets: manually and automatically.

Auto-Commit Offsets

Auto-commit offsets: After consumers pull messages, they automatically commit their offsets without manual intervention. The advantage of this method is it is easy to use, but it may lead to duplicate message consumption or loss.

```
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 sara
    for msg := range claim.Messages() {
        fmt.Printf("Received message: key=%s, value=%s, partition=%d, offset=%d\\n"
        sess.MarkMessage(msg, "")
    }
    return nil
}
```

Manual-Commit Offsets

Manually committing offsets: After processing messages, consumers need to manually commit their offsets. The advantage of this method is that it allows for precise control over offset committing, avoiding duplicate message consumption or loss. However, it should be noted that manual committing can lead to high Broker CPU usage,

affecting performance. As message volume increases, CPU consumption will be significantly high, affecting other features of the Broker. Therefore, it is recommended to commit offsets after a certain number of messages.

```
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 sara
    for msg := range claim.Messages() {
        fmt.Printf("Received message: key=%s, value=%s, partition=%d, offset=%d\\n"
        sess.MarkMessage(msg, "")
          sess.Commit()
    }
    return nil
}
```

Production and Consumption FAQs with Sarama Go

1. Configured manually-commit offset, but the offset doesn't show up in the console when querying the consumption group.
Whether you've configured manual-commit or auto-commit offset, you first need to mark the message, sess.MarkMessage(msg, ""), indicating that the message has been fully consumed, before committing the offset.
2. Some issues with Sarama Go as a consumer exist and the Sarama Go version client has the following known issues:
2.1 When a Topic adds a partition, the Sarama Go client cannot detect and consume the new partition. The client must be restarted to consume the newly added partition.

2.2 When the Sarama Go client subscribes to more than two Topics at the same time, it may lead to some partitions being unable to consume messages normally.
2.3 When the resetting strategy of Sarama Go client's consumption offset is set to Oldest(earliest), if the client experiences downtime or there is a server-side version upgrade, due to the OutOfRange mechanism implemented by the Sarama Go client itself, it may cause the client to start re-consuming all messages from the smallest offset.
2.4 Regarding this issue, for the Confluent Go client's demo address, see kafka-confluent-go-demo.
3. Error message: Failed to produce message to topic.
The issue may be caused by misaligned versions. In this case, the client should first check the version of the Kafka Broker, and then specify the version:

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

Java SDK

Last updated：2024-07-04 16:01:00

Overview

TDMQ for CKafka is a distributed stream processing platform used to build real-time data pipelines and streaming applications. It offers high throughput, low latency, scalability, and fault tolerance.
Kafka Clients: These are Kafka's built-in clients, implemented in Java. They serve as clients for Kafka's standard production and consumption protocols.
This document describes the key parameters, practical tutorials, and FAQs about the aforementioned Java clients.

Producer Practice

Version Selection

The compatibility between Kafka clients and clusters is very important. Generally, newer versions of clients are compatible with older versions of clusters, but the reverse may not necessarily be true. Typically, the version of the CKafka instance's broker is clear after deployment, so you can just choose the matching client version based on the broker's version.
In the Java ecosystem, Spring Kafka is widely used. The correspondence between Spring Kafka versions and Kafka Broker versions can be found on the official Spring website under Version Correspondence.
Producer parameters and optimization

Producer Parameters

When writing to Kafka using the Kafka Client, you need to configure the following key parameters. The parameters and their default values are:

```
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, InterruptedEx
```

```
        // Create Kafka producer configuration.
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS); // L
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.clas
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.cl

        // Set key parameters and default values of producers.
        props.put(ProducerConfig.ACKS_CONFIG, "1");// acks: Represents the level of
        props.put(ProducerConfig.BATCH_SIZE_CONFIG, 16384);// batch.size, the batch
        props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 33554432);// buffer.memory,
        props.put(ProducerConfig.CLIENT_ID_CONFIG, "");// client.id, the client ID.
        props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "none");// compression.ty
        props.put(ProducerConfig.CONNECTIONS_MAX_IDLE_MS_CONFIG, 540000);// connect
        props.put(ProducerConfig.DELIVERY_TIMEOUT_MS_CONFIG, 120000);// delivery.ti
        props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, false);// enable.idempo
        props.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG, "");// interceptor.cla
        props.put(ProducerConfig.LINGER_MS_CONFIG, 0);// linger.ms, the delay for s
        props.put(ProducerConfig.MAX_BLOCK_MS_CONFIG, 60000);// max.block.ms, the m
        props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, 5);// max.i
        props.put(ProducerConfig.MAX_REQUEST_SIZE_CONFIG, 1048576);// max.request.s
        props.put(ProducerConfig.METADATA_MAX_AGE_CONFIG, 300000);//metadata.max.ag
        props.put(ProducerConfig.METRIC_REPORTER_CLASSES_CONFIG, "");// metric.repo
        props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG, "org.apache.kafka.client
        props.put(ProducerConfig.RECEIVE_BUFFER_CONFIG, 32768);// receive.buffer.by
        props.put(ProducerConfig.SEND_BUFFER_CONFIG, 131072);// send.buffer.bytes,
        props.put(ProducerConfig.RECONNECT_BACKOFF_MAX_MS_CONFIG, 1000);// reconnec
        props.put(ProducerConfig.RECONNECT_BACKOFF_MS_CONFIG, 50);// reconnect.back
        props.put(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG, 30000);// request.timeo
        props.put(ProducerConfig.RETRIES_CONFIG, 2147483647);// retries, the number
        props.put(ProducerConfig.RETRY_BACKOFF_MS_CONFIG, 100);// retry.backoff.ms,
        props.put(ProducerConfig.TRANSACTION_TIMEOUT_CONFIG, 60000);// transaction.
        props.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, null);// transactional.id
        props.put(ProducerConfig.CLIENT_DNS_LOOKUP_CONFIG, "default");// client.dns

        // Create a producer.
        KafkaProducer<String, String> producer = new KafkaProducer<>(props);

        // Send the message.
        for (int i = 0; i < 100; i++) {
            String key = "key-" + i;
            String value = "value-" + i;

            // Create a message record.
            ProducerRecord<String, String> record = new ProducerRecord<>(TOPIC, key

            // Send the message.
            producer.send(record, new Callback() {
```

```
                @Override
                public void onCompletion(RecordMetadata metadata, Exception excepti
                    if (exception == null) {
                        System.out.println("Message sent successfully:key=" + key +
                    } else {
                        System.err.println("Message sending failed:" + exception.ge
                    }
                }
            });
        }

        // Close the producer.
        producer.close();
    }
}
```

Parameter Description and Optimization

acks Parameter Optimization

The acks parameter is used to control the confirmation mechanism when the producer sends messages. Its default value is 1, which means that after the message is sent to the leader broker, it returns upon the leader's confirmation of the message being written. The acks parameter also has the following optional values:
0: Do not wait for any confirmation, return directly.
1: Wait for the leader replica to confirm the write before returning.
-1 or all: Wait for the Leader replica and the relevant follower replicas to confirm the write before returning.
In cross availability zone scenarios, and for topics with a higher number of replicas, the choice of acks parameter affects the message's reliability and throughput. Therefore:
In some online business message scenarios, where throughput requirements are not high, you can set the acks parameter to -1 to ensure that the message is received and confirmed by all replicas before returning. This improves message reliability but sacrifices write throughput and performance, and the latency will increase.
In scenarios involving big data, such as log collection, or offline computing, where high throughput (i.e., the volume of data written to Kafka per second) is required, you can set the acks to 1 to improve throughput.

Batch Parameter Optimization

By default, for transmitting the same volume of data, a single request's network transmission can effectively reduce related computation and network resources compared to multiple requests, thereby improving the overall write throughput.
Therefore, this parameter can be set to optimize the client's message sending throughput capabilities. In high throughput scenarios, you can set this parameter in combination with computation:
batch.size: Default is 16 K.

linger.ms: Default is 0. You can appropriately increase the delay, such as setting it to 100 ms, to aggregate more messages for batch sending.
buffer.memory: Default is 32 MB. For high-traffic producers, you can set it larger if there is sufficient heap memory, such as setting it to 256 MB.

Transaction Parameter Optimization

```
put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, false);// enable.idempotence, determi
put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, 5);// max.in.flight.reque
props.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, null);// transactional.id, transa
props.put(ProducerConfig.TRANSACTION_TIMEOUT_CONFIG, 60000);// transaction.timeout.
```

Note that the transaction will incur additional computing resources, because it guarantee exactly once semantics.
For transaction scenarios, it is appropriate to increase the transaction timeout to tolerate jitter brought on by write latency in high throughput scenarios.

Compression Parameter Optimization

The Kafka Java Client supports the following Compression Parameters:

```
props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "none");// compression.type, the
```

Currently, it supports the following Compression configurations:
none: No compression algorithm.
gzip: Compress by GZIP algorithm.
snappy: Compress by Snappy algorithm.
lz4: Compress by LZ4 algorithm.
zstd: Compress by ZSTD algorithm.
To use compressed messages in Kafka Java Client, you need to set the compression.type parameter when creating a producer. For example, to use the LZ4 compression algorithm, you can set compression.type to lz4.
Kafka message compression is an optimization method that uses compute to save bandwidth. Although Kafka message compression and decompression occur on the client side, the broker performs verification actions on compressed messages, leading to extra computation cost. As the increased compute leads to high broker CPU usage, reducing the processing capability for other requests, the overall performance drop, especially for gzip compression. The server-side verification computation cost of such compressed messages can be very high. For some cases, the cost is not worth the benefit. We recommend the following method to avoid broker verification in such cases:
1. Independently compress message data on the producer side to generate packed data compression: messageCompression, and store the compression method in the message's key:

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

2. On the producer side, send messageCompression as a normal message.
3. On the consumer side, read the message key, access the compression method used and performs decompression independently.

Creating Producer Instance

If your application needs higher throughput, you can use asynchronous sending to increase the speed of message sending. At the same time, batch message sending can be utilized to reduce network overhead and IO consumption. If the application requires higher reliability, synchronous sending can ensure message delivery success. Meanwhile, ACK confirmation mechanism and transaction mechanism can be used to ensure the reliability and consistency of messages. For specific parameter optimization, see producer parameters and optimization.

Synchronous Sending

An example of synchronous sending in the Kafka Java Client:

```
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) {
        // Create Kafka producer configuration.
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.clas
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.cl

        // Set producer parameters.
        props.put(ProducerConfig.ACKS_CONFIG, "all");
        props.put(ProducerConfig.RETRIES_CONFIG, 3);

        // Create a producer.
        KafkaProducer<String, String> producer = new KafkaProducer<>(props);
```

```
        // Synchronously send messages.
        for (int i = 0; i < 10; i++) {
            String key = "sync-key-" + i;
            String value = "sync-value-" + i;

            // Create a message record.
            ProducerRecord<String, String> record = new ProducerRecord<>(TOPIC, key

            try {
                // Send messages and wait for results.
                RecordMetadata metadata = producer.send(record).get();
                System.out.println("Synchronous sending succeeded:key=" + key + ",
            } catch (InterruptedException | ExecutionException e) {
                System.err.println("Synchronous sending failed:" + e.getMessage());
            }
        }

        // Close the producer.
        producer.close();
    }
}
```

Asynchronous Sending

Asynchronous sending: When messages are sent asynchronously, the current thread won't be blocked, and the producer throughput is higher. However, message results need to be handled through a callback function. Example:

```
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) {
        // Create Kafka producer configuration.
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.clas
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.cl

        // Set producer parameters.
```

```
        props.put(ProducerConfig.ACKS_CONFIG, "all");
        props.put(ProducerConfig.RETRIES_CONFIG, 3);

        // Create a producer.
        KafkaProducer<String, String> producer = new KafkaProducer<>(props);

        // Send messages asynchronously.
        for (int i = 0; i < 10; i++) {
            String key = "async-key-" + i;
            String value = "async-value-" + i;

            // Create a message record.
            ProducerRecord<String, String> record = new ProducerRecord<>(TOPIC, key

            // Send the message and set the callback function.
            producer.send(record, new Callback() {
                @Override
                public void onCompletion(RecordMetadata metadata, Exception excepti
                    if (exception == null) {
                        System.out.println("Asynchronous sending succeeded:key=" +
                    } else {
                        System.err.println("Asynchronous sending failed:" + excepti
                    }
                }
            });
        }

        // Close the producer.
        producer.close();
    }
}
```

Consumer Practice

Consumer Parameters

```
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");
        properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserial
        properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeseri
        properties.put(ConsumerConfig.GROUP_ID_CONFIG, "test-group"); // "group.id"
        properties.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "latest"); // "auto
        properties.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true"); // "enabl
        properties.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "5000"); // "
        properties.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, "10000"); // "sess
        properties.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, "500"); // "max.poll
        properties.put(ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG, "300000"); // "m
        properties.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, "1"); // "fetch.min.b
        properties.put(ConsumerConfig.FETCH_MAX_BYTES_CONFIG, "52428800"); // "fetc
        properties.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, "500"); // "fetch.m
        properties.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, "3000"); // "he
        properties.put(ConsumerConfig.CLIENT_ID_CONFIG, "my-client-id"); // "client
        properties.put(ConsumerConfig.REQUEST_TIMEOUT_MS_CONFIG, "30000"); // "requ

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

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

Parameter Optimization

1. When using Kafka consumers, we can optimize performance by adjusting some parameters. Here are some common parameter optimization methods:

fetch.min.bytes: If you don't know the minimum message size, we recommend you to set this parameter to 1. You can set this value to the minimum message size if you know it.
max.poll.records: This parameter can be adjusted based on the processing capacity of the application. If your application can handle more records, you can set this value to a larger number to reduce the frequency of poll operations.
auto.commit.interval.ms: This parameter can be adjusted according to the needs of your application. Generally, for scenarios with automatic offset commits, it is recommended to set it to the default value of 5,000 ms. Note that excessively frequent offset commits can affect performance and additionally consume broker's computational resources.
client.id: You can set a unique ID for each consumer to distinguish between different consumers in monitoring and logs.
The above are some common parameter optimization methods, but the optimal settings might vary based on the features and requirements of your application. When optimize parameters, remember to always conduct performance testing to ensure the result matches your expectation.
2. For issues of frequent rebalance and consumption thread blocking, see the following parameter optimization instructions:
session.timeout.ms: For versions before v0.10.2, increase this parameter value appropriately to make it greater than the time it takes to consume a batch of data and not exceed 30 s. The recommended value is 25 s. For v0.10.2 and later versions, use the default value of 10 s.
max.poll.records: Decrease this value to make it significantly less than the product of <the number of messages consumed per second per thread> x <the number of consumption threads> x <max.poll.interval.ms> as recommended.
max.poll.interval.ms: This value should be greater than <max.poll.records> / (<the number of messages consumed per second per thread> x <the number of consumption threads>).

Creating Consumer Instance

The Kafka Java Client provides a subscription model to create consumers, where it offers two ways to commit the offset: manually and automatically.

Auto-Commit Offsets

Auto-commit offsets: Consumers automatically commit their offsets after pulling messages, eliminating the need for manual operation. This method's advantage is its simplicity and ease of use, but it may lead to duplicate message consumption or loss. Note that the auto-commit interval, auto.commit.interval.ms, should not be set too short; otherwise, it may lead to relatively high broker CPU utilization, affecting the processing of other requests.

```
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) {
        // Create Kafka consumer configuration.
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserialize
        props.put(ConsumerConfig.GROUP_ID_CONFIG, GROUP_ID);
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        // Enable auto-commit offset.
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, true);
        props.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, 5000); // Auto-com

        // Create a consumer.
        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

        // Subscribe to topics.
        consumer.subscribe(Collections.singletonList(TOPIC));

        // Consume messages.
        try {
            while (true) {
                ConsumerRecords<String, String> records = consumer.poll(Duration.of
                for (ConsumerRecord<String, String> record : records) {
                    System.out.printf("Consume message: topic=%s, partition=%d, off
                            record.topic(), record.partition(), record.offset(), re
                }
            }
        } finally {
            // Close the consumer.
            consumer.close();
        }
    }
}
```

Manual-Commit Offsets

Manual-commit offsets: After processing messages, consumers need to manually commit their offsets. The advantage of this method is that it allows for precise control over offset commit, avoiding duplicate message consumption or loss. However, it should be noted that manual commit can lead to high broker CPU usage, affecting performance. As message volume increases, CPU consumption will be significantly high, affecting other features of the broker. Therefore, it is recommended to commit offset after a certain number of messages.

```
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, StringDeserial
        properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeseri
        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.of
                for (ConsumerRecord<String, String> record : records) {
                    System.out.printf("offset = %d, key = %s, value = %s%n", record
                    count++;
                    if (count % 10 == 0) {
                        consumer.commitSync();
                        System.out.println("Committed offsets.");
                    }
```

```
                }
            }
        } finally {
            consumer.close();
        }
    }
}
```

Assign Consumption

The Kafka Java Client's assign consumption mode allows consumers to directly specify the partitions for subscription, rather than automatically assigning partitions through topic subscription. This mode is suitable for scenarios where manual control of consumed partitions is needed, such as implementing specific cloud load balancer policy, or skipping certain partitions in some cases. The general process involves using the assign method to manually specify the partitions consumed by the consumer, setting the starting offset for consumption with the seek method, and then executing the consumption logic. For example:

```
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.String
        props.put("value.deserializer", "org.apache.kafka.common.serialization.Stri
        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));

        // Set the starting offset for consumption.
        long startPosition0 = 10L;
        long startPosition1 = 20L;
```

```
        consumer.seek(partition0, startPosition0);
        consumer.seek(partition1, startPosition1);

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

Production and Consumption FAQs of Kafka Java Client

Kafka Java Producer is unable to send messages successfully

First, check if the IP and port of the Kafka cluster can be connected normally. If not, resolve the connection issues first. Next, verify the correct configuration of the access point and whether the version matches the broker version. Send demo according to the best practices.

Kafka Python SDK

Last updated：2024-07-04 16:00:59

Overview

CKafka's Python client has the following major libraries:
kafka-python: This is a Kafka client implemented in pure Python, supporting Kafka 0.8.2 and higher versions. It provides APIs for producers, consumers, and managing the Kafka cluster. This library is easy to use, but its performance may not be as good as clients based on librdkafka.
Installation method: `pip install kafka-python`
confluent-kafka-python: This library is based on the high-performance C library librdkafka. It supports Kafka 0.9 and later versions, and provides APIs for producers, consumers, and managing the Kafka cluster. This library has better performance, but might require the installation of additional dependencies.
Installation method: `pip install confluent-kafka`
aiokafka: This is an asynchronous Kafka client based on kafka-python, using the asyncio library. It is suitable for scenarios that require asynchronous programming.
Installation method: `pip install aiokafka`
pykafka: This is a Python client supporting Kafka version 0.8.x. It provides APIs for producers, consumers, and managing the Kafka cluster. This library is no longer actively maintained, but still suitable for scenarios requiring support for older Kafka versions.
Installation method: `pip install pykafka`
When choosing a Python Kafka client, select the appropriate library based on your application requirements and Kafka version. For most scenarios, kafka-python or confluent-kafka-python is recommended, as they support newer versions of Kafka and have more comprehensive features. If your application requires asynchronous programming, consider using aiokafka.
This document mainly describes how to use kafka-python. See the official documentation at kafka-python.

Producer Practices

Version Selection

To use kafka-python, you should install the kafka-python library first. Run the following command to install it:
`pip install kafka-python`

Producer Parameters and Optimization

Producer Parameters

Kafka Python involves the following key parameters. The parameters and their default values are as follows:

```
from kafka import KafkaProducer

producer = KafkaProducer(
    bootstrap_servers='localhost:9092', # Broker list used to initialize connection
    client_id=None, # Custom client ID for identification in Kafka server logs, wit
    key_serializer=None, # Callable object used for serializing message keys into b
    value_serializer=None, # Callable object used for serializing message values in
    compression_type=None, # Message compression type, valid values are 'gzip', 'sn
    retries=0, # Number of times to retry failed messages, with the default value b
    batch_size=16384, # The size of messages used for batch processing, measured in
    linger_ms=0, # Maximum waiting time for additional messages before batch proces
    partitioner=None, # Callable object used to determine the message partition, wi
    buffer_memory=33554432, # Total memory allocated for buffering messages awaitin
    connections_max_idle_ms=540000, # Maximum duration to maintain idle connections
    max_block_ms=60000, # Maximum duration to block the send() method when reaching
    max_request_size=1048576, # Maximum byte size of requests sent to the broker, w
    metadata_max_age_ms=300000, # Maximum lifespan of metadata in the local cache,
    retry_backoff_ms=100, # Waiting time between two retry attempts, in millisecond
    request_timeout_ms=30000, # Maximum waiting time for client to receive a respon
    receive_buffer_bytes=32768, # Network buffer size for receiving data, in bytes,
    send_buffer_bytes=131072, # Network buffer size for sending data, in bytes, wit
    acks='all', # Message acknowledgment mechanism, optional values are '0', '1', o
    transactional_id=None, # Transaction ID, a unique identifier for producer parti
    transaction_timeout_ms=60000, # Transaction timeout, in milliseconds, with the
    enable_idempotence=False, # Whether to enable Idempotence, with the default val
    security_protocol='PLAINTEXT', # Security protocol type, optional values are 'P
```

Parameter Description and Optimization

acks Parameter Optimization

The acks parameter controls the confirmation mechanism for producers when sending messages. The default value of this parameter is -1, indicating that the message will only be returned after being sent to the Leader Broker and confirmed by the Leader, along with the associated Follower messages being written. The acks parameter has the following optional values: 0, 1, -1. In cross-availability zone scenarios, as well as Topic with a higher number of replicas, the value of the acks parameter will affect the message's reliability and throughput. Therefore:
In some online business messaging scenarios, where required throughput is not high, you can set the acks parameter to -1. This ensures that the message is received and acknowledged by all replicas before returning, thereby increasing the message's reliability.
In scenarios involving big data, such as log collection or offline computing, where high throughput (i.e., the volume of data written to Kafka per second) is required, you can set the acks to 1 to improve throughput.

buffer_memory Parameter Optimization (Caching)

By default, for transmitting the same volume of data, a single request's can effectively reduce computation and network resources compared to multiple requests, thereby improving the overall write throughput.
Therefore, the message sending throughput of the client can be optimized by setting this parameter. For Kafka Python Client, the default linger_ms is set to 0 ms for batching time to accumulate messages, which can be optimized by appropriately increasing the value, for example, setting it to 100 ms, to aggregate and send multiple requests in batches, thus improving throughput. If bandwidth is high and memory on the machine is sufficient, it is recommended to increase buffer_memory to enhance throughput.

Compression Parameter Optimization

The Kafka Python Client supports the following compression parameters: none, gzip, snappy, lz4.
none: Do not compress.
gzip: Compress by GZIP.
snappy: Compress by Snappy.
lz4: Compress by LZ4.
To use compressed messages on the Producer client, you need to set the compression_type parameter when creating the producer. For example, to use the LZ4 compression algorithm, you can set the compression_type to lz4.
As an optimization method that exchanges computation for bandwidth, message compression and decompression occurs on the client side. However, due to additional computation cost of the Broker's behavior in verifying compressed messages, compression is not recommended in low traffic situations. For GZIP compression, especially, the verification computation cost for the Broker can be significant, which is not worthwhile in some cases. Due to increased computation, message processing capabilities of the Broker may be lower, resulting in lower bandwidth throughput. In these situations, the following compression method is recommended:
Independently compress message data on the Producer side to generate packed data compression: messageCompression, and store the compression method in the message's key:
`{"Compression","lz4"}`
On the Producer side, send messageCompression as a normal message.
On the Consumer side, read the message key, access the compression method used, and perform independent decompression.

Creating Producer Instance

If an application requires higher reliability, a synchronous producer can be used to ensure message delivery success. Furthermore, the ACK acknowledgement mechanism and transaction mechanism can be used to ensure the reliability and consistency of messages. For specific parameter optimization, see Producer Parameters and Optimization. If an application requires higher throughput, employ an asynchronous producer to increase the speed of message delivery. Additionally, use the batch message sending method to reduce network overhead and IO consumption. See the example below:

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

```
# Parameter Configuration
BOOTSTRAP_SERVERS = 'localhost:9092'
TOPIC = 'test_topic'
SYNC = True
ACKS = '1'  # leader replica acknowledgement suffices
LINGER_MS = 500  # Delay sending for 500 ms
BATCH_SIZE = 16384  # Message batch size 16 KB

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

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

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

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

Consumer Practice

Consumer Parameters and Optimization

Consumer Parameters

```
from kafka import KafkaConsumer

# Create a KafkaConsumer object for connecting to the Kafka cluster and consuming m
consumer = KafkaConsumer(
    'topic_name', # List of Topics to subscribe to
    bootstrap_servers=['localhost:9092'],    # Access point for the Kafka cluster
    group_id=None,    # Consumer Group ID used for grouping consumers, required by
    client_id='kafka-python-{version}',    # The default client ID, with the defaul
    api_version=None, # Specify the Kafka API version to use. If it is set to None,
    enable_auto_commit=True,    # Whether to automatically commit the consumer offs
    auto_commit_interval_ms=5000,    # Interval for auto-committing consumer offset
    auto_offset_reset='latest',    # Policy for consumer's consumption position in
    fetch_min_bytes=1,    # Minimum bytes for consumer to read from a partition, wi
    fetch_max_wait_ms=500,    # Waiting time when there is no more new consumption
    fetch_max_bytes=52428800,    # Maximum bytes for consumer to read from a partit
    max_poll_interval_ms=300000    # Default interval is 300,000 milliseconds (5 mi
    retry_backoff_ms=100,    # Retry interval, with the default value being 100 mil
    reconnect_backoff_max_ms=1000,    # Maximum interval for reconnection attempts
    request_timeout_ms=305000,    # Client request timeout, in milliseconds
    session_timeout_ms=10000,    # session_timeout_ms (int) - Timeout period for de
    heartbeat_interval_ms=3000,    # The expected time interval between heartbeats
    receive_buffer_bytes=32768, # Size of the TCP receive buffer (SO_RCVBUF) used w
    send_buffer_bytes=131072 # Size of the TCP send buffer (SO_SNDBUF) used when se
)

for message in consumer:
    print(f"Topic: {message.topic}, Partition: {message.partition}, Offset: {messag
```

Parameter Description and Optimization

1. max_poll_interval_ms is a configuration parameter for Kafka Python Consumer, specifying the maximum delay between two poll operations. Its primary function is to control the liveness of the Consumer, i.e., to determine if the Consumer is still active. If the Consumer does not conduct a poll operation within the time specified by max_poll_interval_ms, Kafka considers this Consumer as failed, triggering a rebalance operation for the Consumer. The setting of this parameter should be adjusted according to the actual consumption speed. Setting it too low may lead to frequent triggering of rebalance operations, increasing Kafka's load; setting it too high might prevent Kafka from timely detecting issues with the Consumer, thereby affecting message consumption. It is recommended to increase the setting of this value under high throughput conditions.

2. For auto-commit offset requests, it is advised not to set auto_commit_interval_ms lower than 1000 ms, as too frequent offset requests can cause high CPU usage on the Broker, impacting the read/write operations of other services.

Creating Consumer Instance

Kafka Python provides a subscription model for creating consumers, which supports both manual and automatic offset commits.

Auto-Commit Offsets

Committing automatic offsets: After pulling messages, consumers automatically commit offsets without manual intervention. This method is easy to use, but may lead to duplicate message consumption or loss. It is recommended to commit offsets every 5 s.

```
# 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  # Set the interval for automatic offset commits t
)

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

Manual-Commit Offsets

Manual-Commit Offsets: After processing messages, consumers need to manually commit their offsets. The advantage of this method is that it allows for precise control over offset committing, avoiding duplicate message consumption or loss. However, note that manual committing can lead to high Broker CPU usage, affecting performance. As message volume increases, CPU consumption will be significantly high, affecting other features of the Broker. Therefore, it is recommended to commit offsets after a certain number of messages.

```
# 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: {messag
    count += 1

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

    sleep(1)
```

ibrdkafka SDK

Last updated：2024-07-04 16:00:59

Overview

TDMQ for CKafka is a distributed stream processing platform used to build real-time data pipelines and streaming applications. It provides high throughput, low latency, scalability, and fault tolerance.
This document describes the key parameters, practical tutorials and FAQs of the librdkafka client mentioned above.

Producer Practice

Version Selection

When librdkafka is used, it automatically selects the appropriate protocol version for communication based on the version of the Kafka cluster. Due to the rapid iteration updates of Kafka versions, usually, we recommend the latest version of librdkafka for you to achieve the best compatibility and performance.

Producer Parameters and Optimization

Producer Parameters

The key parameters of librdkafka and their default values are as follows:

```
rd_kafka_conf_t *conf = rd_kafka_conf_new();

// Addresses of Kafka clusters, separated by commas, with the default value being e
rd_kafka_conf_set(conf, "bootstrap.servers", "localhost:9092", NULL, 0);

// The maximum number of attempts to send a message, including the first attempt an
rd_kafka_conf_set(conf, "message.send.max.retries", "2", NULL, 0);

// Backoff time between retries (in milliseconds), with the default value being 100
rd_kafka_conf_set(conf, "retry.backoff.ms", "100", NULL, 0);

// Client request timeout (in milliseconds), with the default value being 5000.
rd_kafka_conf_set(conf, "request.timeout.ms", "5000", NULL, 0);

// The buffer size for data sent by the client (in bytes), with the default value b
rd_kafka_conf_set(conf, "queue.buffering.max.kbytes", "131072", NULL, 0);

// Maximum number of messages in the client's send buffer, with the default value b
```

```
rd_kafka_conf_set(conf, "queue.buffering.max.messages", "100000", NULL, 0);

// Maximum total size of messages in the client's send buffer (in bytes), with the
rd_kafka_conf_set(conf, "queue.buffering.max.total.bytes", "1000000", NULL, 0);

// Linger time of the client's send buffer (in milliseconds), with the default valu
rd_kafka_conf_set(conf, "queue.buffering.max.ms", "0", NULL, 0);

// Whether to enable message compression, with the default value being 0 (disabled)
rd_kafka_conf_set(conf, "compression.codec", "none", NULL, 0);

// Message compression level, with the default value being 0 (auto-select).
rd_kafka_conf_set(conf, "compression.level", "0", NULL, 0);

// Client ID, with the default value being rdkafka.
rd_kafka_conf_set(conf, "client.id", "rdkafka", NULL, 0);

// Maximum concurrency request count for the producer, i.e., the number of requests
rd_kafka_conf_set(conf, "max.in.flight.requests.per.connection", "1000000", NULL, 0

// Maximum retry count for client connections to the Kafka cluster, with the defaul
rd_kafka_conf_set(conf, "broker.address.ttl", "3", NULL, 0);

// Interval between retry attempts for client connections to the Kafka cluster (in
rd_kafka_conf_set(conf, "reconnect.backoff.ms", "1000", NULL, 0);

// Maximum interval between retry attempts for client connections to the Kafka clus
rd_kafka_conf_set(conf, "reconnect.backoff.max.ms", "10000", NULL, 0);

// Backoff time for client API version (in milliseconds), with the default value be
rd_kafka_conf_set(conf, "api.version.request.timeout.ms", "10000", NULL, 0);

// Security protocol, with the default value being plaintext.
rd_kafka_conf_set(conf, "security.protocol", "plaintext", NULL, 0);

// For other SSL and SASL parameters, see the librdkafka documentation.

// Create a producer instance.
rd_kafka_t *producer = rd_kafka_new(RD_KAFKA_PRODUCER, conf, NULL, 0);
```

Parameter Description and Optimization

acks Parameter Optimization

The acks parameter controls the confirmation mechanism when the producer sends a message. The default value of this parameter is -1, which means that it returns only after the message is sent to the leader broker, the leader

confirms, and the corresponding follower messages are all written. The acks parameter also has the following optional values: 0, 1, and -1. In cross-availability zone scenarios, and for topics with a high number of replicas, the value of the acks parameter can affect the message's reliability and throughput. Therefore:
In some online business messaging scenarios where the requirement for throughput is not significant, you can set the acks parameter to -1 to ensure that the message is received and acknowledged by all replicas before returning, thus improving the message's reliability.
In scenarios involving big data, such as log collection, and offline computing, where high throughput (i.e., the volume of data written to Kafka per second) is required, you can set the acks to 1 to increase throughput.

Buffering Parameter Optimization (Caching)

By default, for transmitting the same volume of data, a single request's network transmission can effectively reduce related computation and network resources compared to multiple requests, thereby improving the overall write throughput.
Therefore, you can set this parameter to optimize the client's message sending throughput. For librdkafka, a default batching time of 5 ms is provided to buffer messages. If the messages are small, you can increase the queue.buffering.max.ms duration appropriately.

Compression Parameter Optimization

librdkafka supports the following compression parameters: none, gzip, snappy, lz4, and zstd.
The librdkafka client supports the following compression algorithms:
none: No compression algorithm.
gzip: Compress by GZIP algorithm.
snappy: Compress by Snappy algorithm.
lz4: Compress by LZ4 algorithm.
zstd: Compress by ZSTD algorithm.
To use a compression algorithm in the producer client, set the compression.type parameter when creating the producer. For example, if you want to use the LZ4 compression algorithm, set compression.type to lz4. The CPU compression and CPU decompression occur on the client side, representing a calculation-for-bandwidth optimization.
However, the broker's verification behavior for compressed messages, especially for Gzip compression, incurs additional computation costs, resulting in significant server-side computation costs. The increased computation could reduce the broker's message processing capability, leading to lower bandwidth throughput, which may not be worthwhile in some cases. In such cases, we recommend the following method:
Independently compress message data on the producer side to generate packed data compression: messageCompression, and store the compression method in the message's key:

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

On the producer side, send messageCompression as a normal message.
On the consumer side, read the message key to access the compression method used, and perform independent decompression.

Creating Producer Instance

If an application requires higher throughput, it can use an asynchronous producer to increase the speed of message sending. At the same time, batch message sending can be utilized to reduce network overhead and IO consumption. If an application demands higher reliability, a synchronous producer can be used to ensure successful message delivery. Additionally, the ACK acknowledgement mechanism and transaction mechanism can be employed to guarantee message reliability and consistency. For specific parameter optimization, see Producer Parameters and Optimization.

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

// Producer message sending callback.
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(rkmessag
    } 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();

    // Set the addresses of the Kafka cluster.
    rd_kafka_conf_set(conf, "bootstrap.servers", "localhost:9092", NULL, 0);

    // Set ack to 1, indicating that the message is considered successfully sent on
    rd_kafka_conf_set(conf, "acks", "1", NULL, 0);

    // Set the producer message sending callback.
    rd_kafka_conf_set_dr_msg_cb(conf, dr_msg_cb);

    // Create a producer instance.
    char errstr[512];
    rd_kafka_t *producer = rd_kafka_new(RD_KAFKA_PRODUCER, conf, errstr, sizeof(err
    if (!producer) {
        fprintf(stderr, "Failed to create producer: %s\\n", errstr);
        return 1;
    }

    // Create a topic instance.
    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_
        rd_kafka_destroy(producer);
        return 1;
    }

    // Send the message.
    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
    }

    // Wait for all messages to be sent.
    while (rd_kafka_outq_len(producer) > 0) {
        rd_kafka_poll(producer, 1000);
    }

    // Destroy topic instance.
    rd_kafka_topic_destroy(topic);

    // Destroy producer instance.
    rd_kafka_destroy(producer);

    return 0;
}
```

Consumer Practice

Consumer Parameters and Optimization

Consumer Parameters

```
    rd_kafka_conf_t *conf = rd_kafka_conf_new();

    // Set the addresses of the Kafka cluster.
    rd_kafka_conf_set(conf, "bootstrap.servers", "localhost:9092", NULL, 0);
```

```
    // Set consumer group ID, with the default value being empty.
    rd_kafka_conf_set(conf, "group.id", "mygroup", NULL, 0);

    // Set the consumer's automatic commit interval (in milliseconds), with the def
    rd_kafka_conf_set(conf, "auto.commit.interval.ms", "5000", NULL, 0);

    // Enable the consumer's automatic commit, with the default value being true.
    rd_kafka_conf_set(conf, "enable.auto.commit", "true", NULL, 0);

    // Set the consumer's automatic offset reset policy, with the default value bei
    rd_kafka_conf_set(conf, "auto.offset.reset", "latest", NULL, 0);

    // Set client ID, with the default value being rdkafka.
    rd_kafka_conf_set(conf, "client.id", "rdkafka", NULL, 0);

    // Create a consumer instance.
    char errstr[512];
    rd_kafka_t *consumer = rd_kafka_new(RD_KAFKA_CONSUMER, conf, errstr, sizeof(err
```

Parameter Description and Optimization

For automatic offset commit requests, it's recommended not to set auto.commit.interval.ms below 1,000 ms, as too frequent offset requests can cause high broker CPU usage, affecting the read and write operations of other normal services.

Creating Consumer Instance

Provide a subscription model for creating consumers, which offers two ways of offset commit: manual submission and automatic submission.

Auto-Commit Offsets

Auto-commit offsets: After pulling messages, consumers automatically commit their offsets without manual intervention. The advantage of this method is it's easy to use, but it may lead to duplicate message consumption or loss.

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

// Consumer message processing callback.
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": %.
               rd_kafka_topic_name(rkmessage->rkt),
               rkmessage->partition,
               rkmessage->offset, (int)rkmessage->len, (const char *)rkmessage->pay
    }
}

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

    // Set the addresses of the Kafka cluster.
    rd_kafka_conf_set(conf, "bootstrap.servers", "localhost:9092", NULL, 0);

    // Set consumer group ID.
    rd_kafka_conf_set(conf, "group.id", "mygroup", NULL, 0);

    // Enable the consumer's automatic offset commit, with the default value being
    rd_kafka_conf_set(conf, "enable.auto.commit", "true", NULL, 0);

    // Set the consumer's automatic commit interval (in milliseconds), default is 5
    rd_kafka_conf_set(conf, "auto.commit.interval.ms", "5000", NULL, 0);

    // Create Consumer Instance.
    char errstr[512];
    rd_kafka_t *consumer = rd_kafka_new(RD_KAFKA_CONSUMER, conf, errstr, sizeof(err
    if (!consumer) {
        fprintf(stderr, "Failed to create consumer: %s\\n", errstr);
        return 1;
    }

    // Subscribe to topics.
    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_
        rd_kafka_topic_partition_list_destroy(topics);
        rd_kafka_destroy(consumer);
        return 1;
    }
    rd_kafka_topic_partition_list_destroy(topics);

    // Consume messages.
    while (1) {
        rd_kafka_message_t *rkmessage = rd_kafka_consumer_poll(consumer, 1000);
```

```
        if (rkmessage) {
            msg_consume(rkmessage, NULL);
            rd_kafka_message_destroy(rkmessage);
        }
    }

    // Unsubscribe.
    rd_kafka_unsubscribe(consumer);

    // Destroy consumer instance.
    rd_kafka_destroy(consumer);

    return 0;
}
```

Manual-Commit Offsets

Manual-commit offsets: After processing messages, consumers need to manually commit their offsets. The advantage of this method is that it allows for precise control over offset commit, avoiding duplicate message consumption or loss. Not that manual commit can lead to high broker CPU usage, affecting performance. As message volume increases, CPU consumption will be significantly high, affecting other features of the broker. Therefore, it is recommended to commit offsets after a certain number of messages.

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

// Consumer message processing callback.
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": %.
               rd_kafka_topic_name(rkmessage->rkt),
               rkmessage->partition,
               rkmessage->offset, (int)rkmessage->len, (const char *)rkmessage->pay
    }
}

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

```
// Set the addresses of the Kafka cluster.
rd_kafka_conf_set(conf, "bootstrap.servers", "localhost:9092", NULL, 0);

// Set consumer group ID.
rd_kafka_conf_set(conf, "group.id", "mygroup", NULL, 0);

// Disable the consumer's automatic commit.
rd_kafka_conf_set(conf, "enable.auto.commit", "false", NULL, 0);

// Create consumer instance.
char errstr[512];
rd_kafka_t *consumer = rd_kafka_new(RD_KAFKA_CONSUMER, conf, errstr, sizeof(err
if (!consumer) {
    fprintf(stderr, "Failed to create consumer: %s\\n", errstr);
    return 1;
}

// Subscribe to topics.
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_
    rd_kafka_topic_partition_list_destroy(topics);
    rd_kafka_destroy(consumer);
    return 1;
}
rd_kafka_topic_partition_list_destroy(topics);

// Consume messages and manually commit the offset.
int message_count = 0;
while (1) {
    rd_kafka_message_t *rkmessage = rd_kafka_consumer_poll(consumer, 1000);
    if (rkmessage) {
        msg_consume(rkmessage, NULL);

        // Manually commit the offset after every 10 messages.
        if (++message_count % 10 == 0) {
            if (rd_kafka_commit_message(consumer, rkmessage, 0) != RD_KAFKA_RES
                fprintf(stderr, "Failed to commit offset for message: %s\\n", r
            } else {
                printf("Offset %"PRId64" committed\\n", rkmessage->offset);
            }
        }

        rd_kafka_message_destroy(rkmessage);
    }
}
```

```
    // Unsubscribe.
    rd_kafka_unsubscribe(consumer);

    // Destroy consumer instance.
    rd_kafka_destroy(consumer);

    return 0;
}
```

tRpc Go SDK

Last updated：2024-09-10 16:08:48

Overview

TDMQ for CKafka is a distributed stream processing platform used to build real-time data pipelines and streaming applications. It offers high throughput, low latency, scalability, and fault tolerance.
This document describes the key parameters, practical tutorials and FAQs of the tRpc-Go-Kafka client.

Optimization Practice

tRpc-GO-Kafka encapsulates the open-source Kafka SDK, using features such as the tRPC-Go interceptor to integrate into the tRPC-Go ecosystem. Therefore, see Sarama Go for practical tutorial:

FAQs

Producer Issues

1. **When messages are produced via CKafka, the error** `Message contents does not match its CRC` **occurs.**

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

By default, the plugin enables gzip compression. Add the parameter `compression=none` to the target to disable compression.

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

2. **How to configure sequential production for the same user?**

Add the `partitioner` parameter to the client, with options including random (default), roundrobin, and hash (partitioning by key).

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

3. **How to execute asynchronous production?**

Add the `async=1` parameter to the client

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

4. **How to write data callback via asynchronous production?**

You need to rewrite callback functions for the success/failure of asynchronous production writing data in the code, for example:

```
func init() {
  // Rewrite the default callback for failures of asynchronous production writing d
  kafka.AsyncProducerErrorCallback = func(err error, topic string, key, value []byt
    // do something if async producer occurred error.
  }

  // Rewrite the default callback for successes of asynchronous production writing
  kafka.AsyncProducerSuccCallback = func(topic string, key, value []byte, headers [
    // do something if async producer succeed.
  }
}
```

Consumer Issues

1. **What will happen if Handle is set to return non-nil during consumption?**

It will sleep for 3 s and then retry consumption. This is not recommended, because returning an error will lead to infinite retries of consumption. Retry logic for failures should be set by business side.

2. **When consuming messages with CKafka, the error `client has run out of available brokers to talk to` occurs.**

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

First, check if the brokers are reachable, then check the supported Kafka client version, and try adding parameters in the configuration file address, for example, version=0.10.2.0

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

3. **When multiple producers are producing messages, will the failure of some producers in establishing a connection make the normal producers time out?**

Update the version to v0.2.18. For lower versions, when a producer is created, plugins lock first, then establish a connection, and unlock thereafter. If there are some abnormal producers which take a long time to establish connection, it will cause other normal production requests to fail in locking when accessing producers, result in timeout eventually. This behavior has been fixed in v0.2.18.

4. **When consuming messages, you receive a prompt: `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.`

For the same consumer group, the client re-grouping strategy is different. You can modify the `strategy` parameter, valid values including sticky (by default), range, and roundrobin.

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

5. **How to inject custom configuration (remote configuration)?**

If you need to specify your configuration in code, first configure `trpc_go.yaml` with `fake_address`, then use the `kafka.RegisterAddrConfig` method for injection. Configure `trpc_go.yaml` as follows:

```
address: fake_address
```

Before the service is started, inject custom configuration:

```
func main() {
  s := trpc.NewServer()
  // Use custom addr, which should be injected before the server starts
  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. **How to access the underlying Sarama's context information?**

```
By using kafka.GetRawSaramaContext, you can access the underlying Sarama ConsumerGr
// RawSaramaContext stores Sarama ConsumerGroupSession and ConsumerGroupClaim
// Export this structure to facilitate user monitoring. The content provided is for
type RawSaramaContext struct {
    Session  sarama.ConsumerGroupSession
    Claim sarama.ConsumerGroupClaim
}
```

Instance:

```
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
}
```

Overview

TDMQ for CKafka is a distributed stream processing platform used to build real-time data pipelines and streaming applications. It offers high throughput, low latency, scalability, and fault tolerance.
This document describes the key parameters and best practices for the tRpc-Go-Kafka client, as well as FAQs.

Optimization Practice

tRPC-GO-Kafka encapsulates the open-source Kafka SDK, using features such as the tRPC-Go interceptor to integrate into the tRPC-Go ecosystem. Therefore, for best practices, refer to Sarama Go:

FAQs

Producer Issues

1. **When CKafka is used to produce messages, the error `Message contents does not match its CRC` occurs.**

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

By default, the plugin enables gzip compression. To disable compression, add the `compression=none` parameter to the target.

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

2. **How to configure sequential production for the same user?**

Add the `partitioner` parameter to the client, with options including random (default), roundrobin, and hash (partitioning by key).

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

3. **How to execute asynchronous production?**

Add the `async=1` parameter to the client

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

4. **How to write data callback via asynchronous production?**

You need to rewrite callback functions for the success/failure of asynchronous production writing data in the code, for example:

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

func init() {
  // Rewrite the default callback for failures of asynchronous production writing d
  kafka.AsyncProducerErrorCallback = func(err error, topic string, key, value []byt
```

```
      // do something if async producer occurred error.
    }

    // Rewrite the default callback for successes of asynchronous production writing
    kafka.AsyncProducerSuccCallback = func(topic string, key, value []byte, headers [
      // do something if async producer succeed.
    }
  }
```

Consumer Issues

1. **What will happen if Handle is set to return non-nil during consumption?**

It will sleep for 3 s and then retry consumption. This is not recommended, because returning an error will lead to infinite retries of consumption. Retry logic for failures should be set by business side.

2. **When messages are consumed with CKafka, the error** `client has run out of available brokers to talk to` **occurs.**

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

First, check if the brokers are reachable, then check the supported Kafka client version, and try adding parameters in the configuration file address, for example, version=0.10.2.0.

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

3. **When multiple producers are producing messages, will the failure of some producers in establishing a connection make the normal producers time out?**

Update the version to v0.2.18. For lower versions, when creating a producer, plugins lock first, then establish a connection, and unlock thereafter. If there are some abnormal producers which take a long time to establish connection, it will cause other normal production requests to fail in locking when accessing producers, result in timeout eventually. This behavior has been fixed in v0.2.18.

4. **When messages are consumed, you receive a prompt:** `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.`

For the same consumer group, the client re-grouping strategy is different. You can modify the `strategy` parameter, valid values including: sticky (default), range, roundrobin.

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

5. **How to inject your custom configuration (remote configuration)?**

If you need to specify your configuration in code, first configure `trpc_go.yaml` with `fake_address` , then use the `kafka.RegisterAddrConfig` method for injection. Configure `trpc_go.yaml` as follows:

```
  address: fake_address
```

Before the service is started, inject custom configuration:

```
func main() {
  s := trpc.NewServer()
  // Use custom addr, which should be injected before the server is started.
  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. **How to access the underlying Sarama's context information?**

```
By using kafka.GetRawSaramaContext, you can access the underlying Sarama ConsumerGr
// RawSaramaContext stores sarama ConsumerGroupSession and ConsumerGroupClaim
// Export this structure to facilitate user monitoring. The content provided is for
type RawSaramaContext struct {
    Session  sarama.ConsumerGroupSession
    Claim sarama.ConsumerGroupClaim
}
```

Instance:

```
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
}
```

Connector Practical Tutorial
Reporting over HTTP
Connection to Kafka over HTTP

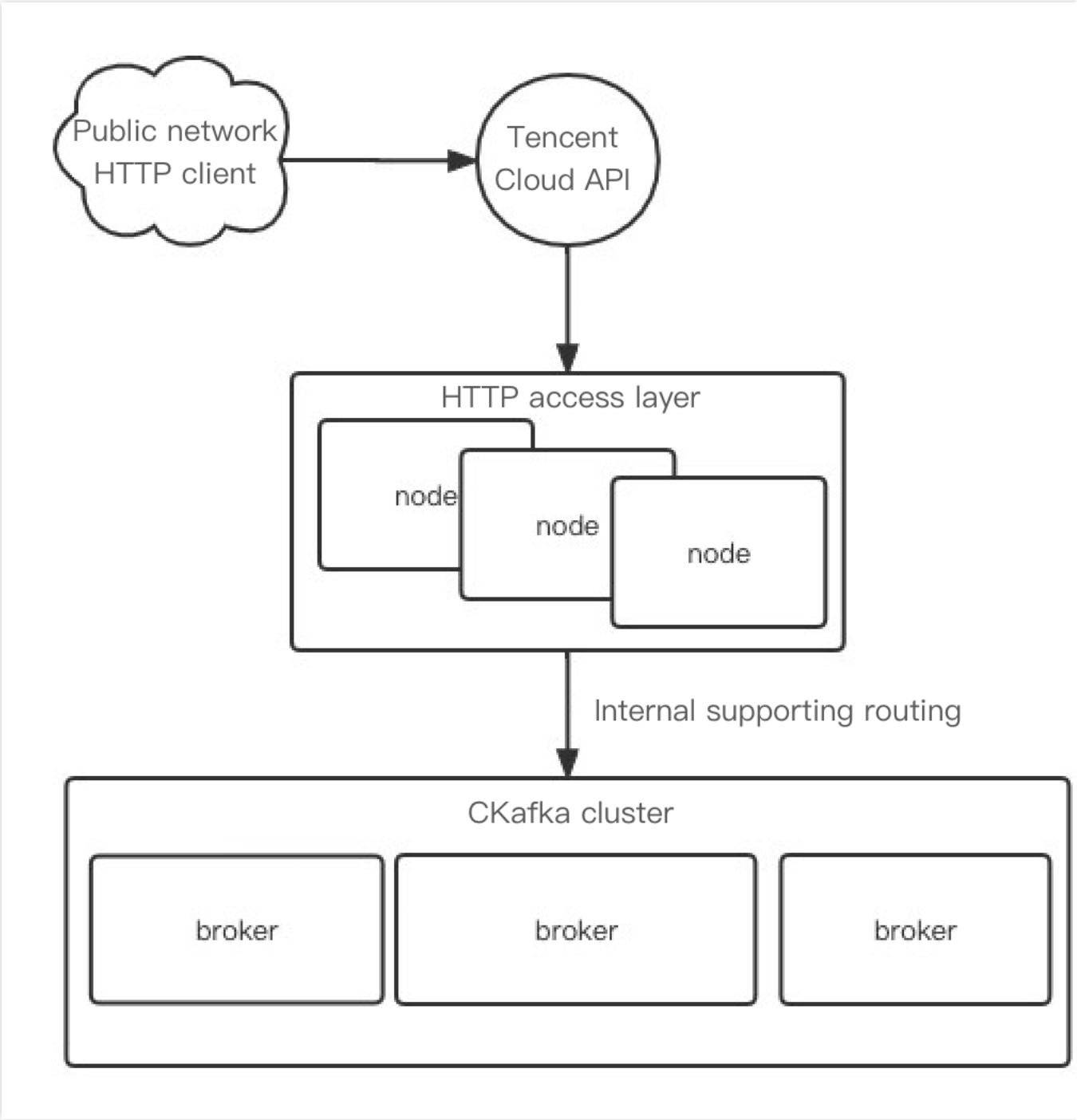
Last updated：2024-01-09 14:56:36

Overview

Apache Kafka, like any client-server application, offers access to its functionality through a well-defined set of APIs. These APIs are exposed via the Kafka wire protocol, a Kafka-specific binary protocol over TCP. The best way to interact with the Apache Kafka APIs is to make use of a client library that works with the Kafka wire protocol. The Apache Kafka project only officially supports a client library for Java, but in addition to that, Confluent officially supports client libraries for C/C++, C#, Go, and Python.
Unfortunately, some programming languages lack officially supported, production-grade client libraries for Kafka. However, HTTP is a widely available, universally supported protocol. For data access, DataHub exposes message sending APIs over the HTTP protocol to simplify client configurations.
This document describes message sending in the HTTP-based data access feature of DataHub and provides suggestions for real world cases.

Architecture

After the HTTP data access layer is enabled, an HTTP client in the public network can directly send messages to a CKafka instance through TencentCloud API as shown below:



Prerequisites

You have created the target CKafka instance and topic.

Directions

Creating data access task

For detailed directions, see Reporting over HTTP.

Sending message via SDK

1. Import the data reporting SDK through Maven or Gradle into the Java project. Below is the `pom.xml` file for project configuration:

```
<dependency>
    <groupId>com.tencentcloudapi</groupId>
    <artifactId>tencentcloud-sdk-java</artifactId>
    <version>3.1.430</version>
</dependency>
```

2. Click **Task Details** in Data Access and copy the access point information to the SDK for data writes.

3. Enter the access point information. In the sample code, `generateMsgFromUserAccess` is used to assemble all messages to be sent.

```
   List<BatchContent> batchContentList = generateMsgFromUserAccess(userId);
   // Here, `ap-xxx` is the region abbreviation of the corresponding TencentCloud A
   CkafkaClient client = new CkafkaClient(
   new Credential("yourSecretId", "yourSecretKey"), "ap-xxx");


SendMessageRequest messageRequest = new SendMessageRequest();
// Access point ID of the data access task
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. Below is a sample returned value for message sending at the HTTP access layer:

```
{
    "Response": {
        "MessageId": [
            "datahub-lxxxxxx6:topicDev:4:2:1648185961342:1648185961398"
        ],
```

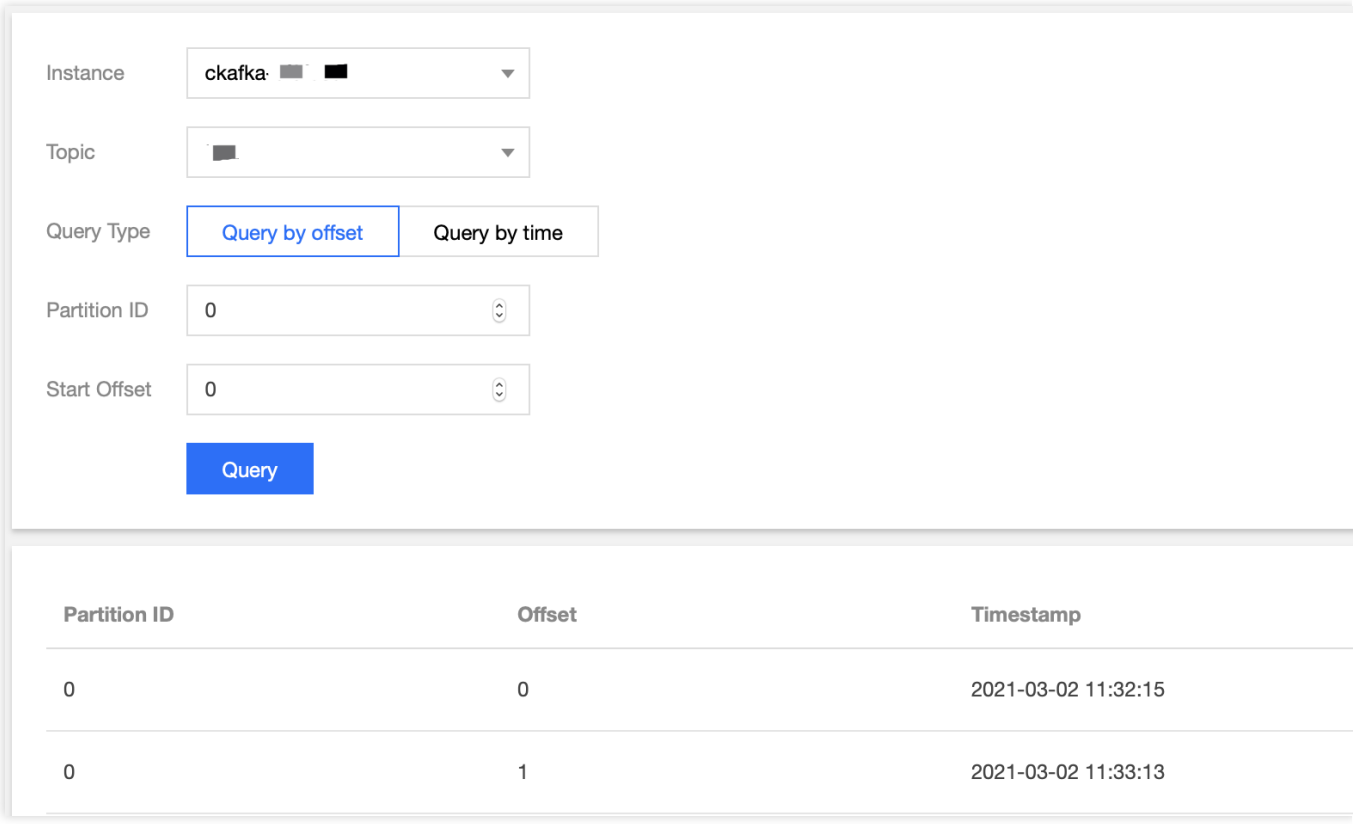
```
        "RequestId": "3fq3na5r-xxxx-xxxx-xxxx-b2fiv0se7ded"
    }
}
```

5. Here, **MessageId** consists of a series of metadata fields returned after the message is sent to the CKafka instance, as detailed below:

```
"[datahubId]:[topic name]:[topic partition number]:[topic offset]:[time when the HT
```

Querying message

You can query messages sent at the HTTP access layer in the CKafka console. For detailed directions, see Querying Message. In this example, messages at offset 2 in partition 4 in the `topicDev` topic are queried as shown below:



Pausing task

If you find that the data access task affects the normal business, you can pause the task.

1. On the Data Access page, click **Pause** in the **Operation** column of the target task to pause the task.
2. If the prompt in the top-right corner in the following figure is displayed, the task was paused successfully.
3. At this time, if you send a message at the HTTP access layer, you will receive the following response:

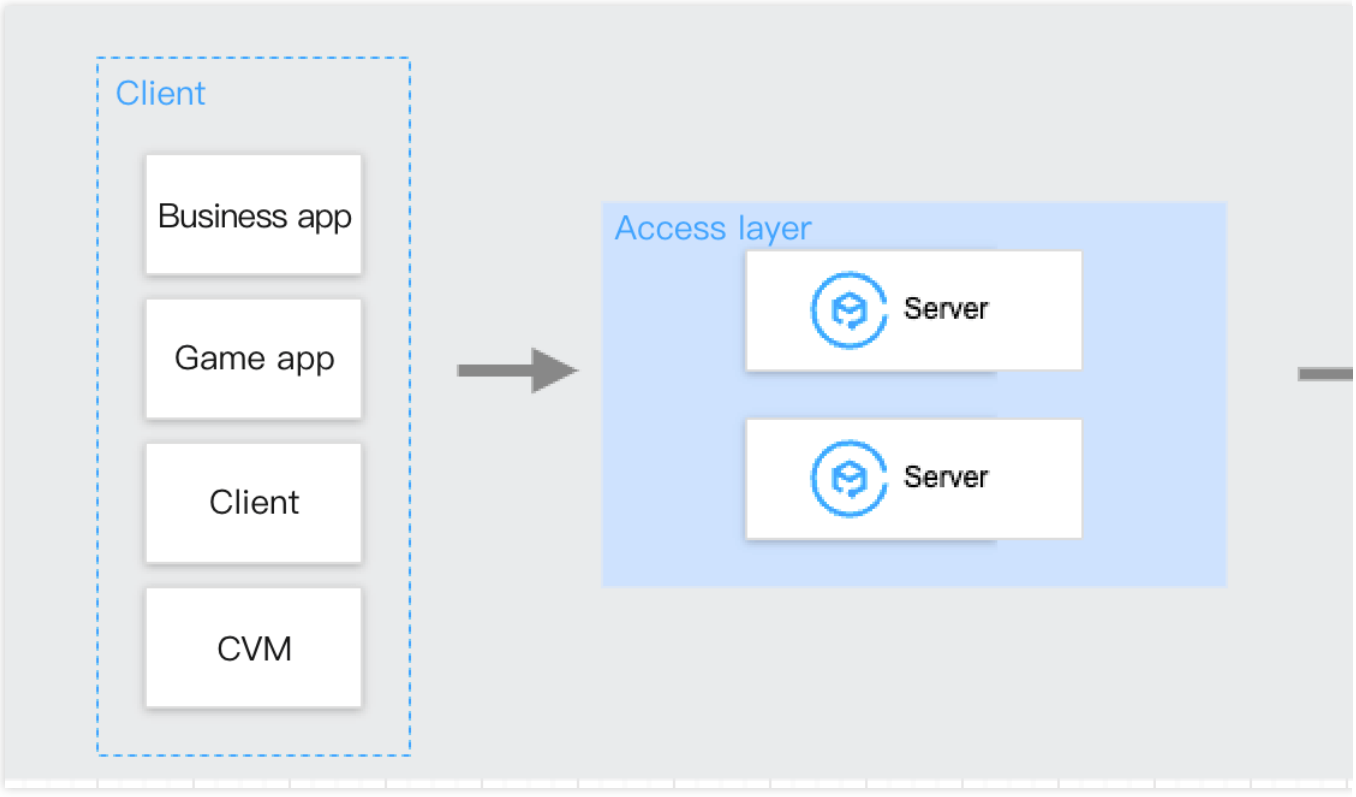
```
{
    "Response": {
        "Error": {
            "Code": "FailedOperation",
            "Message": "task status suspended [datahub-lxxxxxx6]"
        },
        "RequestId": "5f737a5b-xxxx-xxxx-xxxxx-b2fb703e7ded"
    }
}
```

Unified Data Reporting

Last updated：2024-01-09 14:56:36

Overview

DataHub is a data access and processing platform in Tencent Cloud, which provides data access, processing, and distribution features at one stop. Data is vital in internet businesses, and data access and reporting serve as the bridge between data generation, computing, storage, and analysis throughout the entire linkage. Therefore, simple yet efficient data access is critical. A business usually has data to be reported to the backend for storage, analysis, computing, and search, such as business metrics, process information, and monitoring data. The general processing linkage is as shown below:



You can set up a classic data reporting architecture generally in the following steps:

1. Build/Purchase a storage engine to store reported data.
2. Develop and deploy a server to receive data, define APIs, and run the service.
3. Define information such as API protocol and authentication on the client and server.
4. Write code based on the protocol information for the client to report data.

In the above four steps, the development and deployment workload for the server is the highest, as you need to consider code logic development as well as the scalability and stability of the server and downstream storage. In addition, when the data volume gets high, problems on the server will become more obvious, and the server needs to be maintained with a lot of manpower and resources. As tasks involved in this regard are generally universal, DataHub aims to meet the requirements in such scenario by offering a stable, elastic, high-reliability, and high-throughput data access service.

How It Works

DataHub provides SDKs for various programming languages, including Java, Python, Go, PHP, Node.js, C++, and .NET, to help the client better report data. Data can be reported to a storage engine such as CKafka in three simple steps (more Tencent Cloud message queue services like TDMQ for RocketMQ, Pulsar, RabbitMQ, and CMQ will be supported in the future).
1. Create an access point in the DataHub console.
2. Report the data via the SDK.
3. Query the data.

Directions

1. Create an access point in the DataHub console as instructed in Reporting over HTTP.
2. Report the data via the SDK as instructed in Data Reporting SDK.
3. Query the data. After the data is reported to DataHub, you can query the message content in real time as instructed in Querying Message.

Data Empowerment

After you complete data access easily and quickly through DataHub, how to make the data generate value becomes the most important thing. To address this, DataHub provides two core features:

Data processing

DataHub offers a simple data ETL engine, which can cleanse most types of data in order to simply format and process data for subsequent use.

Data distribution

After data processing is completed, DataHub can also meet the data distribution needs in various scenarios:

Real-time search: When you need to search for data, you can export real-time data streams to a search service such as Elasticsearch and CLS.
OLAP analysis: When you need to analyze data, you can export real-time data streams to an engine such as ClickHouse and TDW.
Persistent storage: When you need to persistently store data, you can export real-time data streams to a persistent storage engine such as HDFS and COS.
Stream computing: When you need to process data with custom code, you can use Flink, Spark, and code in different programming languages over the standard Kafka protocol for data processing.
After the data has gone through the four stages of reporting, access, processing, and distribution, the general data reporting and analysis needs can be easily and quickly satisfied, creating data value at ultra low costs.

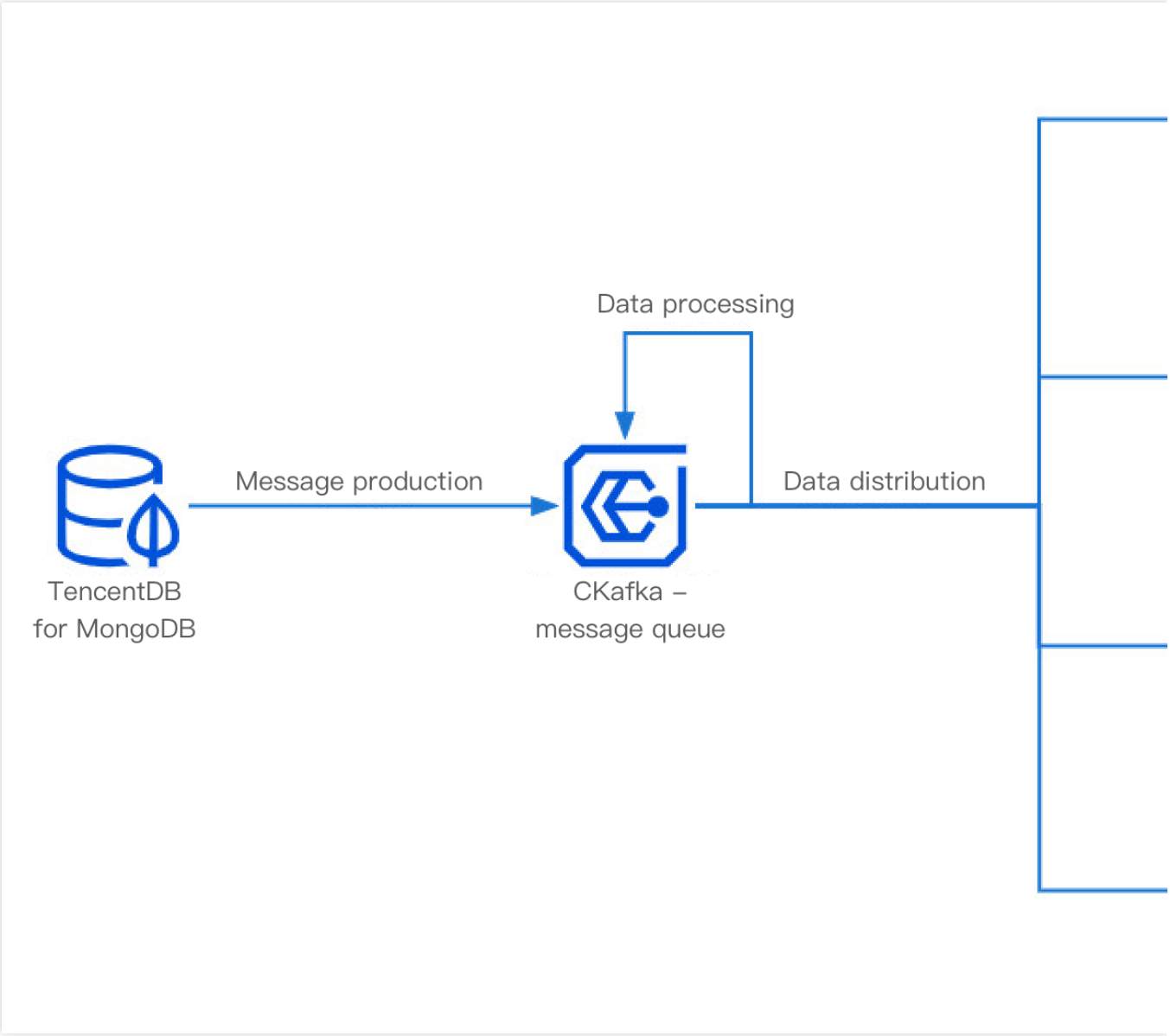
Querying Subscription to Database Change Info

Analysis of Change Logs Tracked by MongoDB Change Streams

Last updated：2024-01-09 14:56:36

Overview

MongoDB uses change streams to track changes. However, to better search for change logs, you usually need to sync them to Elasticsearch or CLS.



This document uses connecting MongoDB to CKafka and distributing CKafka data to CLS as an example to describe how to use the data dump service of DataHub to analyze change logs tracked by change streams in MongoDB.

How It Works

For more information on data access configuration items for MongoDB, see Source Connector. You can set different configuration items to perform corresponding data processing tasks on the accessed change logs and then write them to a topic in the CKafka instance.

Prerequisites

TencentDB for MongoDB has been activated, or CLB is used to listen on the self-built MongoDB instance.
The TCP:27017 port has been opened in the security group.
The CKafka service has been activated.
The CLS service has been activated.

Directions

Step 1. Create a data access task

1. Log in to the CKafka console.
2. Click **Data Access** on the left sidebar, select the region, and click **Create Task**.
3. In the pop-up window, select **Asynchronously pulled data** > **MongoDB** for **Data Source Type**.
4. Click **Next** and enter the task details.
5. Click **Submit** and wait for the task status to become **Healthy**.
6. When MongoDB data changes, you can see that there will be incremental messages in the selected topic in the CKafka instance.

Step 2. Create a data distribution task

1. Log in to the CKafka console.
2. Click **Data Distribution** on the left sidebar, select the region, and click **Create Task**.
3. Select **Cloud Log Service (CLS)** as the **Target Type** and click **Next**.
4. Enter the task details and select the same CKafka instance and topic as those used in the data access task, so that produced messages can be directly consumed.
5. Click **Submit** and wait for the task status to become **Healthy**.

Note:
When a task is in **Healthy** status and incremental messages are written to the topic, they will be directly consumed to the specified CLS log topic.

Step 3. View the distributed data

1. Log in to the CLS console.
2. Select **Search and Analysis** on the left sidebar, select the logset ID and log topic ID entered during distribution task creation, and you can view the change logs of MongoDB.
3. You can search by keyword to directly get the required logs.

Simple Data Cleansing

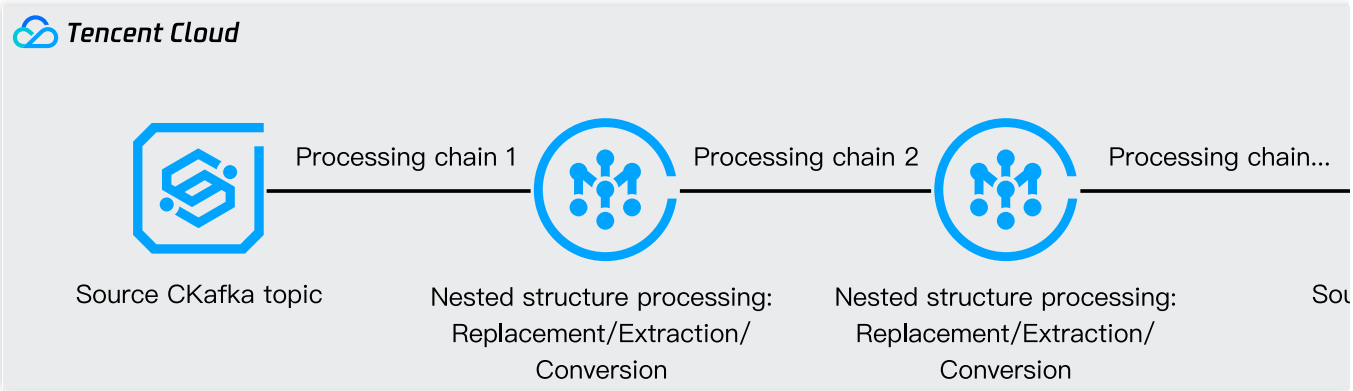
Last updated：2024-01-09 14:56:36

Overview

When using the data access and distribution services in DataHub, you may encounter the following issues:
You need to parse certain fields in messages to get the relevant information.
You need to process a field in messages multiple times in an iterative manner.
You need to convert messages to unstructured raw messages before you can use them.
You need to process messages in a multi-level nested format.
Based on the long-term experience of technical experts in the CKafka team, the brand new data processing component v2.0 is launched. It has the following new features while remaining fully compatible with v1.0:
Diverse plugins: Data processing supports various preconfigured processing plugins to help you quickly generate data in desired consumption formats.
Chain processing: Data processing supports chain message processing; that is, the previous processing result can be used as the input parameters in the current processing. This greatly facilitates processing complex data structures.
Visual preview: Data processing supports real-time structured JSON preview to help you quickly locate and troubleshoot problems.
Type conversion: Data processing can convert data between different types to make data format check easier.

How It Works

The overall data processing flow is as shown below, with each component structure detailed as follows:



In the data processing component cluster, multiple workers form a consumer group to batch read messages from the source topic and process each message in sequence.

In each message, the nested structure of message fields is expanded according to the processing chain sequence configured in the console, and operations such as replacement, extraction, data conversion, and time formatting are performed.
The current processing chain reads the result of the previous processing chain, preforms chain processing, and ships the processing result to the next chain.
The result of the last processing chain is shipped to the configured target topic. At this time, the data processing of a message is completed.

Prerequisites

The CKafka service has been activated.
The message format is JSON or string. Currently, other encoding protocols are not supported.

Actual Application

Processing string-type log

Below is a log in the default Nginx format (aka `combined` format), from which you can parse information such as requester, packet, and request status for further analysis.

```
66.249.65.159 - - [06/Nov/2014:19:10:38 +0600] "GET /news/53f8d72920ba2744fe873ebc.
```

As the `combined` format of Nginx uses spaces and `-` to separate data, design the parsing process in the following steps:
1. First, use the `"` **separator** to initially parse the data. At this time, data processing automatically converts the log to the JSON structure.

```
{
  "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 (K
}
```

2. As shown in the above JSON structure, there are still concatenated coupled data records in the fields of keys `0` and `2` as affected by `-` and spaces. Therefore, split the fields of the two keys with the space and `-` **separators**. The JSON result after splitting is as follows:

```
{
  "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 (K
  "0.0": "66.249.65.159 ",
  "0.2": " [06/Nov/2014:19:10:38 +0600] ",
  "2.1": "404",
  "2.2": "177"
}
```

3. As the time format is enclosed by square brackets “[ ]”, use a **separator** again to extract the time information. The JSON structure after extraction is as follows:

```
{
  "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 (K
  "0.0": "66.249.65.159 ",
  "0.2": "06/Nov/2014:19:10:38 +0600",
  "2.1": "404",
  "2.2": "177"
}
```

4. At this point, all fields have been split appropriately. Name the `keys` of the corresponding mapped fields based on the field attribute. The eventual modification result is as follows:

```
{
  "request": "GET /news/53f8d72920ba2744fe873ebc.html HTTP/1.1",
  "http_user_agent": "Mozilla/5.0 (iPhone; CPU iPhone OS 6_0 like Mac OS X) AppleWe
  "remote_addr": "66.249.65.159 ",
  "dateTime": "06/Nov/2014:19:10:38 +0600",
  "status": "404",
  "body_bytes_sent ": "177"
}
```

Processing nested log

Below is a sample TKE collection format. The TKE collector will place the metadata into the `kubernetes` field in the JSON structure and place the collected log into the `log` field. The overall structure is as follows:

```
{
  "@timestamp": 1648803500.63659,
  "@filepath": "/var/log/tke-log-agent/test7/c816991f-adfe-4617-8cf3-9997aea90ded/c
  "log": "15:00:00.000[4349811564226374227] [http-nio-8081-exec-64] INFO  com.qclou
```

```
  "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\
    },
    "host": "10.0.96.47",
    "container_name": "nginx",
    "docker_id": "add90ccf49626ef42d5615a636aae74d6380996043cf6f6560d8131f21a4d8ba"
    "container_hash": "nginx@sha256:e1211ac17b29b585ed1aee166a17fad63d344bc973bc638
    "container_image": "nginx"
  }
}
```

When collected logs are shipped to Elasticsearch, as the nested JSON format is supported, you don't need to make major changes on the data. However, when logs are shipped to a database, you need to convert the data to an identifiable single-level JSON format. In this case, design the parsing process in the following steps:

1. Use the **JSONPATH** statement to process the `$.kubernetes` JSON structure at the first nested level and select `JSON` as the parsing mode. This converts the nested JSON structure to a single-level JSON structure. The tested result is as follows:

```
{
  "@timestamp": 1.64880350063659E9,
  "@filepath": "/var/log/tke-log-agent/test7/c816991f-adfe-4617-8cf3-9997aea90ded/c
  "log": "15:00:00.000[4349811564226374227] [http-nio-8081-exec-64] INFO  com.qclou
  "$.kubernetes.pod_name": "tke-es-687995d557-n29jr",
  "$.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"
  },
  "$.kubernetes.annotations": {
    "qcloud-redeploy-timestamp": "1648016531476",
    "tke.cloud.tencent.com/networks-status": "[{\\n    \\"name\\": \\"tke-bridge\\"
  },
  "$.kubernetes.host": "10.0.96.47",
```

```
    "$.kubernetes.container_name": "nginx",
    "$.kubernetes.docker_id": "add90ccf49626ef42d5615a636aae74d6380996043cf6f6560d813
    "$.kubernetes.container_hash": "nginx@sha256:e1211ac17b29b585ed1aee166a17fad63d34
    "$.kubernetes.container_image": "nginx"
}
```

2. Process the `$.kubernetes.annotations` and `$.kubernetes.labels` nested structures at the second level. Use `Map` to select the two names in the processing chain to convert the nested format into a single-level JSON format. The processing result is as follows:

```
{
    "@timestamp": 1648803500.63659,
    "@filepath": "/var/log/tke-log-agent/test7/c816991f-adfe-4617-8cf3-9997aea90ded/c
    "log": "15:00:00.000[4349811564226374227] [http-nio-8081-exec-64] INFO  com.qclou
    "$.kubernetes.pod_name": "tke-es-687995d557-n29jr",
    "$.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": "add90ccf49626ef42d5615a636aae74d6380996043cf6f6560d813
    "$.kubernetes.container_hash": "nginx@sha256:e1211ac17b29b585ed1aee166a17fad63d34
    "$.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    \\"na
}
```

3. Rename the `keys` of the corresponding mapped fields and delete unnecessary fields. Click **Add Processing Chain** and **Process All Upper-Level Results**. Below is a sample result after organization and optimization:

```
{
    "@timestamp": 1.64880350063659E9,
    "@filepath": "/var/log/tke-log-agent/test7/c816991f-adfe-4617-8cf3-9997aea90ded/c
    "log": "15:00:00.000[4349811564226374227] [http-nio-8081-exec-64] INFO  com.qclou
    "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"
}
```

Note:

If the `key` contains a period (.) when you use **JSONPath** to process a parameter, you need to add square brackets and single quotation marks to the path for isolation.

For example, to get the desired fields from `{"key1.key2":"value1"}`, you need to use `$.['key1.key2']` to get the corresponding key values.

Processing serialized JSON string-type log

Sometimes, the JSON format needs to be escaped to the string format during data transfer in order to meet the format or performance requirements. This string format is called **raw JSON**, which must be deserialized to the JSON format in data processing. The following uses the raw JSON format in MongoDB as an example, and the overall structure is as shown below:

```
{
  "key": "  {\\n    \\"categories\\": [\\"dev\\"],\\n    \\"created_at\\": \\"2020-
}
```

After the raw JSON format is converted to the general JSON format during data processing, the message can be directly shipped to the target downstream service without changing the field format. The overall processing process is as follows:

1. Set **Parsing Mode** to JSON to preread the message into the internal MAP format. The parsing result is as follows:

```
{
  "key": "  {\\n    \\"categories\\": [\\"dev\\"],\\n    \\"created_at\\": \\"2020-
}
```

2. Click **Add Processing Chain**, use MAP to select the `key`, and select `JSON` as the parsing mode. Then, data processing can automatically convert the raw JSON format to the JSON format. The parsing result is as follows:

```
{
  "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 con
}
```

3. Click **Add Processing Chain** and **Process All Upper-Level Results**, rename the `keys` of the corresponding mapped fields, and delete unnecessary fields. Below is a sample result after organization and optimization:

```
{
```

```
  "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 control
}
```

Note:

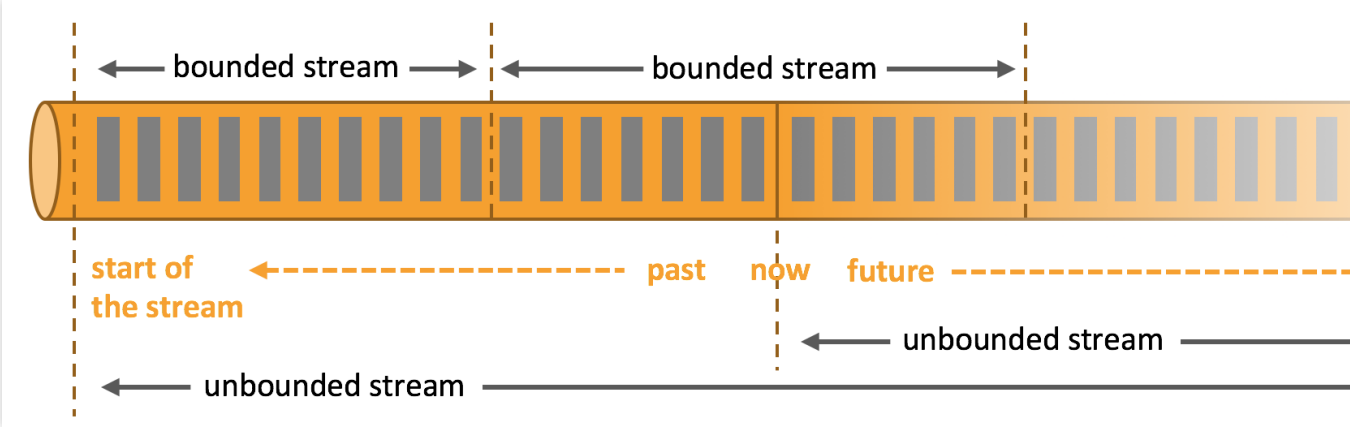
Currently, only MAP-type data in the raw JSON format can be parsed. If the first level is in List type, such as `"[\\"test1\\",\\"test2\\"]"` or `"[{\\"key\\":\\"value\\"}]"`, as it cannot be parsed into appropriate key values, a parsing failure will be reported.

Connecting Flink to CKafka

Last updated：2024-01-09 14:56:36

Overview

Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams. Flink has been designed to run in all common cluster environments, perform computations at in-memory speed and at any scale.

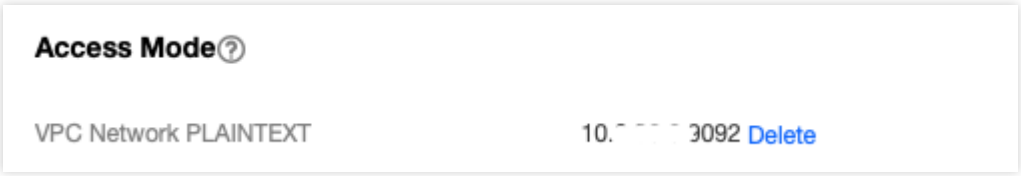


Apache Flink excels at processing unbounded and bounded data sets. Precise control of time and state enable Flink's runtime to run any kind of application on unbounded streams. Bounded streams are internally processed by algorithms and data structures that are specifically designed for fixed sized data sets, yielding excellent performance.
Apache Flink requires real-time data from various sources (such as Apache Kafka or Kinesis) in order to execute applications. Flink provides special Kafka Connectors for reading and writing data from/to Kafka topics, which offers exactly-once processing semantics.

Directions

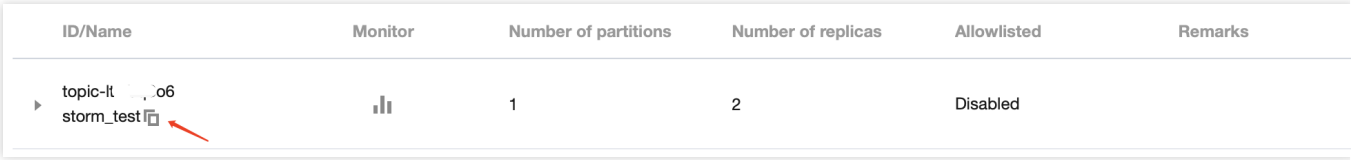
Step 1. Get the CKafka instance access address

1. Log in to the CKafka console.
2. Select **Instance List** on the left sidebar and click the **ID** of the target instance to enter its basic information page.
3. On the instance's basic information page, get the instance access address in the **Access Mode** module, which is the `bootstrap-server` required by production and consumption.



Step 2. Create a topic

1. On the instance's basic information page, select the **Topic Management** tab at the top.
2. On the topic management page, click **Create** to create a topic named `test`. This topic is used as an example below to describe how to consume messages.



Step 3. Add Maven dependencies

Configure `pom.xml` as follows:

```
<?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.
    <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>
```

Step 4. Consume CKafka messages

You can click the tabs below to view the two methods of message consumption and view consumption results in the console or through printed logs.

Consume via VPC

Consume via public domain name

```
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.getExec
```

```
                Properties properties = new Properties();
                //Domain name address for public network access, i.e., public routi
                properties.setProperty("bootstrap.servers", "IP:PORT");
                //Consumer group ID.
                properties.setProperty("group.id", "testConsumerGroup");
                DataStream<String> stream = env
                                .addSource(new FlinkKafkaConsumer<>("topicName", ne
                stream.print();
                env.execute();
        }
}
```

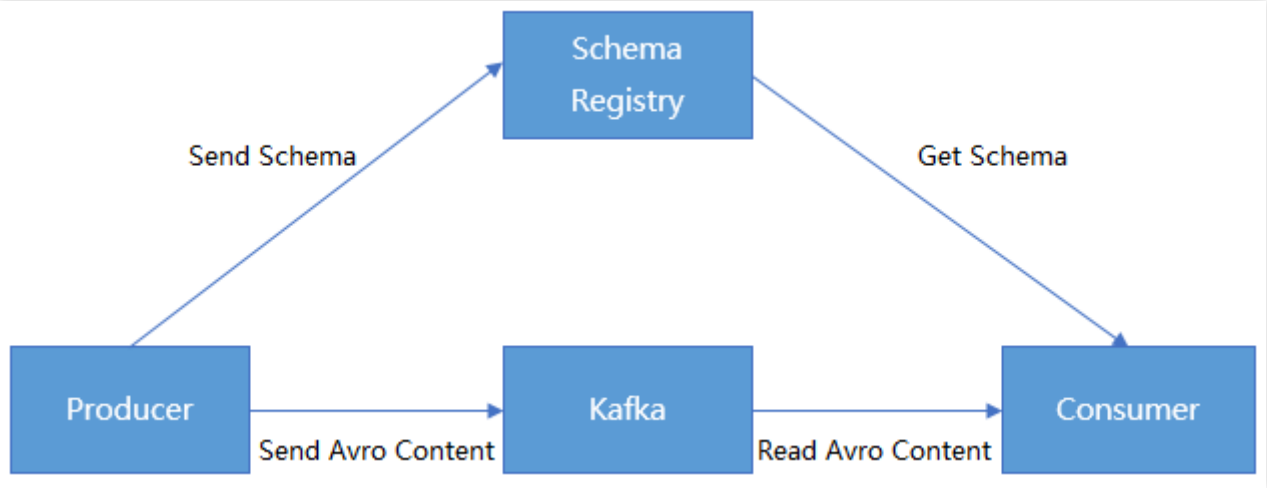
```
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import java.util.Properties;

public class CKafkaConsumerDemo {
        public static void main(String args[]) throws Exception {
                StreamExecutionEnvironment env = StreamExecutionEnvironment.getExec
                Properties properties = new Properties();
                //Domain name address for public network access, i.e., public routi
                properties.setProperty("bootstrap.servers", "IP:PORT");
                //Consumer group ID.
                properties.setProperty("group.id", "testConsumerGroup");
                properties.setProperty("security.protocol", "SASL_PLAINTEXT");
                properties.setProperty("sasl.mechanism", "PLAIN");
                //Username and password. The username is not the one on the console
                properties.setProperty("sasl.jaas.config",
                                                        "org.apache.kafka.comm
                properties.setProperty("sasl.kerberos.service.name","kafka");
                DataStream<String> stream = env
                                .addSource(new FlinkKafkaConsumer<>("topicName", ne
                stream.print();
                env.execute();
        }
}
```

Connecting Schema Registry to CKafka

Last updated：2024-07-19 14:25:46

We can serialize/deserialize classes by using Avro APIs or the Twitter Bijection class library, but the disadvantage of the two methods is that the Kafka record size will multiply as each record must be embedded with a schema. However, the schema is required for reading the records.
CKafka makes it possible for data to share one schema by registering the content of the schema in Confluent Schema Registry. Kafka producers and consumers can implement serialization/deserialization by identifying the schema content in Confluent Schema Registry.



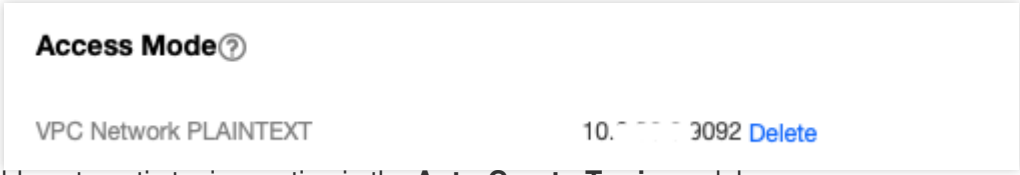
Prerequisites

You have downloaded JDK 8.
You have downloaded Confluent OSS 4.1.1.
You have created an instance as instructed in Creating Instance.

Directions

Step 1. Obtain the instance access address and enable automatic topic creation

1. Log in to the CKafka console.
2. Select **Instance List** on the left sidebar and click the **ID** of the target instance to enter its basic information page.
3. On the instance's basic information page, get the instance access address in the **Access Mode** module.



4. Enable automatic topic creation in the **Auto-Create Topic** module.

Note:

Automatic topic creation must be enabled as a topic named `schemas` will be automatically created when OSS is started.

Step 2. Prepare Confluent configurations

1. Modify the server address and other information in the OSS configuration file.

The configuration information of the PLAINTEXT access method is as follows:

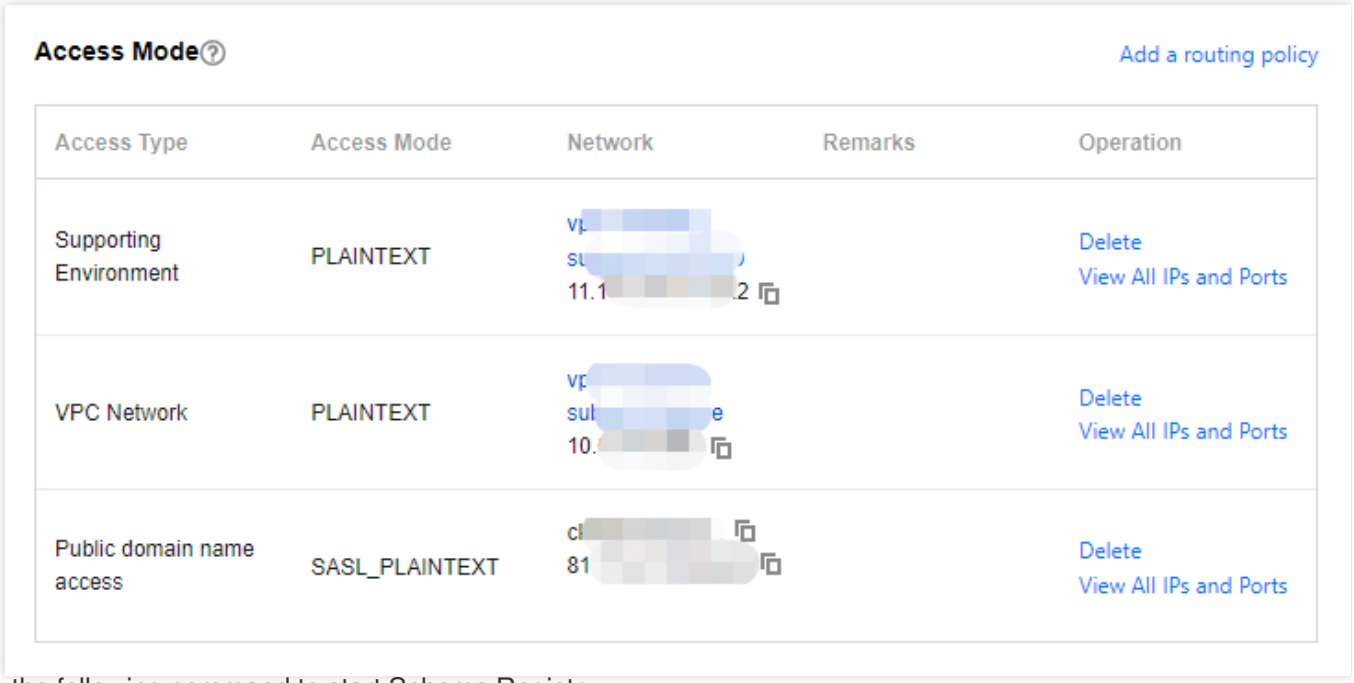
```
kafkastore.bootstrap.servers=PLAINTEXT://xxxx
kafkastore.topic=schemas
debug=true
```

The configuration information of the SASL_PLAINTEXT access method is as follows:

```
kafkastore.bootstrap.servers=SASL_PLAINTEXT://ckafka-xxxx.ap-xxx.ckafka.tencentclou
kafkastore.security.protocol=SASL_PLAINTEXT
kafkastore.sasl.mechanism=PLAIN
kafkastore.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule
kafkastore.topic=schemas
debug=true
```

Note：

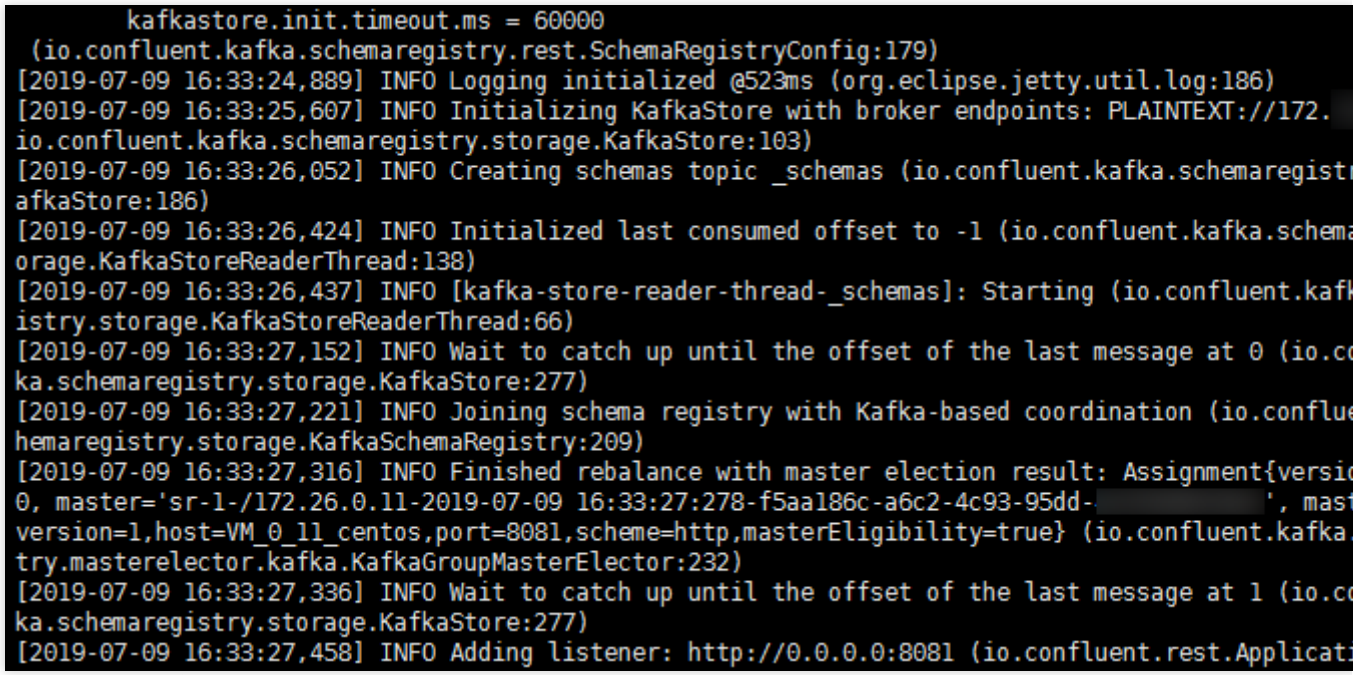
bootstrap.servers: Access the network and copy from the network column of the Access Method section on the instance details page in the CKafka Console.



2. Run the following command to start Schema Registry.

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

The execution result is as follows:



Step 3. Receive/Send messages

Below is the content of the schema file:

```
{
    "type": "record",
    "name": "User",
```

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

1. Register the schema in the topic named `test` .

The script below is an example of registering a schema by calling an API with the `curl` command in the environment deployed in Schema Registry.

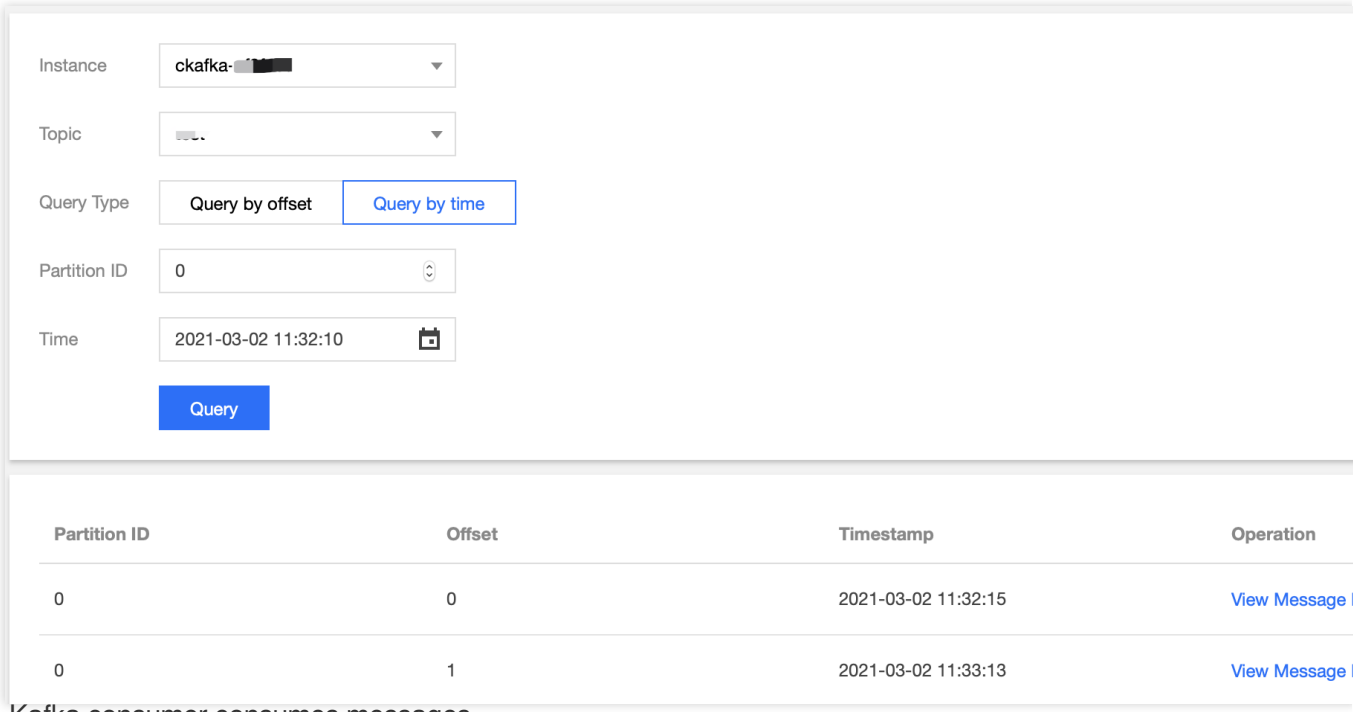
```
curl -X POST -H "Content-Type: application/vnd.schemaregistry.v1+json" \\
--data '{"schema": "{\\"type\\": \\"record\\", \\"name\\": \\"User\\", \\"field
http://127.0.0.1:8081/subjects/test/versions
```

2. The Kafka producer sends messages.

```
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\\
                "\\"fields\\": [{\\"name\\": \\"id\\", \\"type\\": \\"int\\"}, " +
                "{\\"name\\": \\"name\\",  \\"type\\": \\"string\\"}, {\\"name\\":
        public static void main(String[] args) throws Exception {
            Properties props = new Properties();
            // Add the access address of the CKafka instance
          props.put("bootstrap.servers", "xx.xx.xx.xx:xxxx");
            props.put("key.serializer", "org.apache.kafka.common.serialization.Stri
            // Use the Confluent `KafkaAvroSerializer`
            props.put("value.serializer", "io.confluent.kafka.serializers.KafkaAvro
            // Add the schema service address to obtain the 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<>
                producer.send(record);
                Thread.sleep(1000);
            }
            producer.close();
        }
}
```

After running the script for a while, go to the CKafka console, select the **Topic Management** tab on the instance details page, select the topic, and click **More** > **Message Query** to view the message just sent.

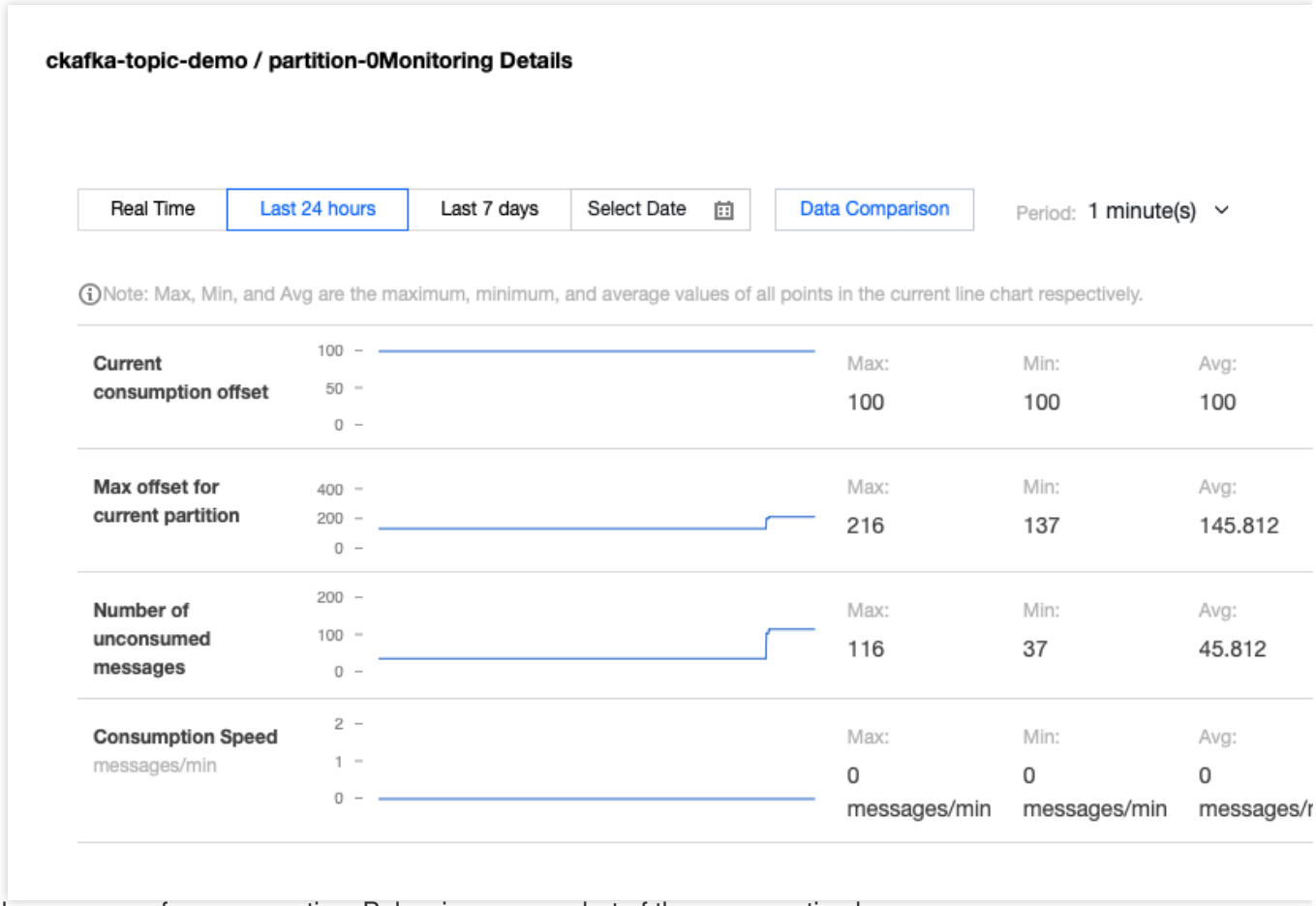


3. The Kafka consumer consumes messages.

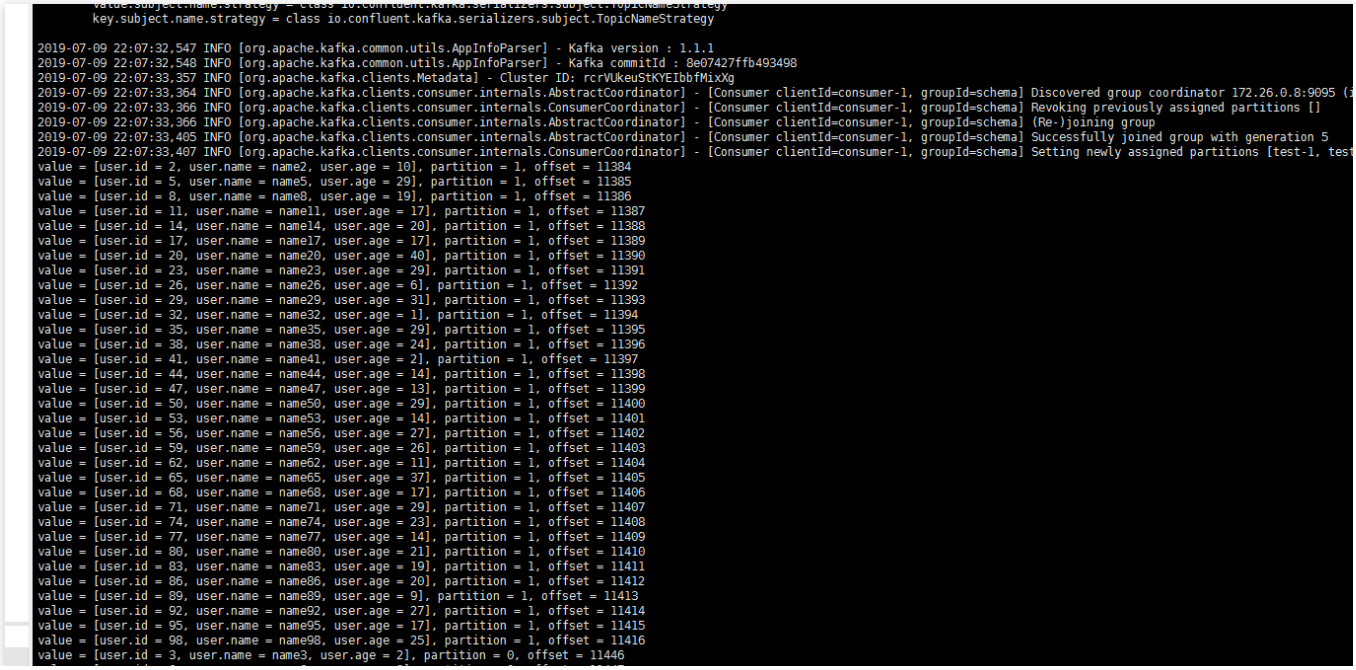
```
package schemaTest;
import java.util.Collections;
import java.util.Properties;
import org.apache.avro.generic.GenericRecord;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
public class SchemaProduce {
    public static void main(String[] args) throws Exception {
        Properties props = new Properties();
```

```
        props.put("bootstrap.servers", "xx.xx.xx.xx:xxxx"); // Access address of th
        props.put("group.id", "schema");
        props.put("key.deserializer", "org.apache.kafka.common.serialization.String
        // Use the Confluent `KafkaAvroDeserializer`
        props.put("value.deserializer", "io.confluent.kafka.serializers.KafkaAvroDe
        // Add the schema service address to obtain the 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.get("name") + ", " + "user.age = " + user.get("a
                            + "partition = " + record.partition() + ", " + "offset
                }
            }
        } finally {
            consumer.close();
        }
    }
}
```

On the **Consumer Group** tab page in the CKafka console, select the consumer group named `schema`, enter the topic name, and click **View Consumer Details** to view the consumption details.



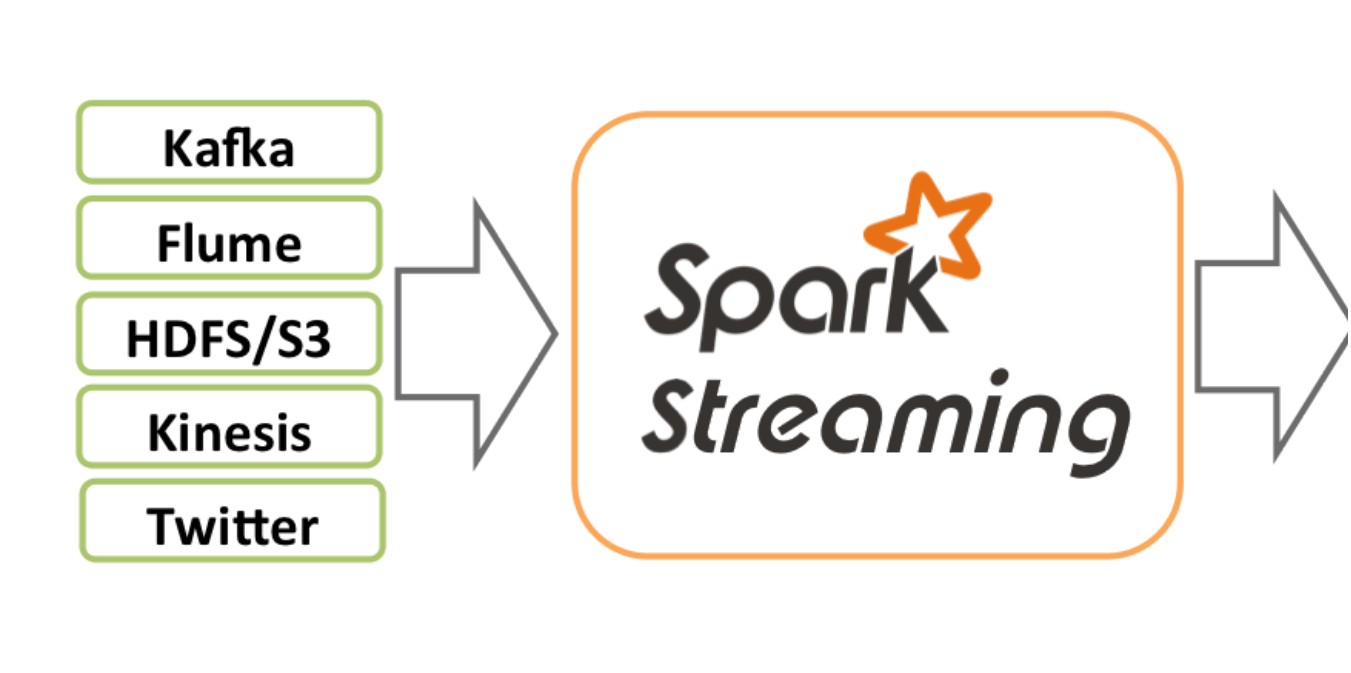
Start the consumer for consumption. Below is a screenshot of the consumption log:



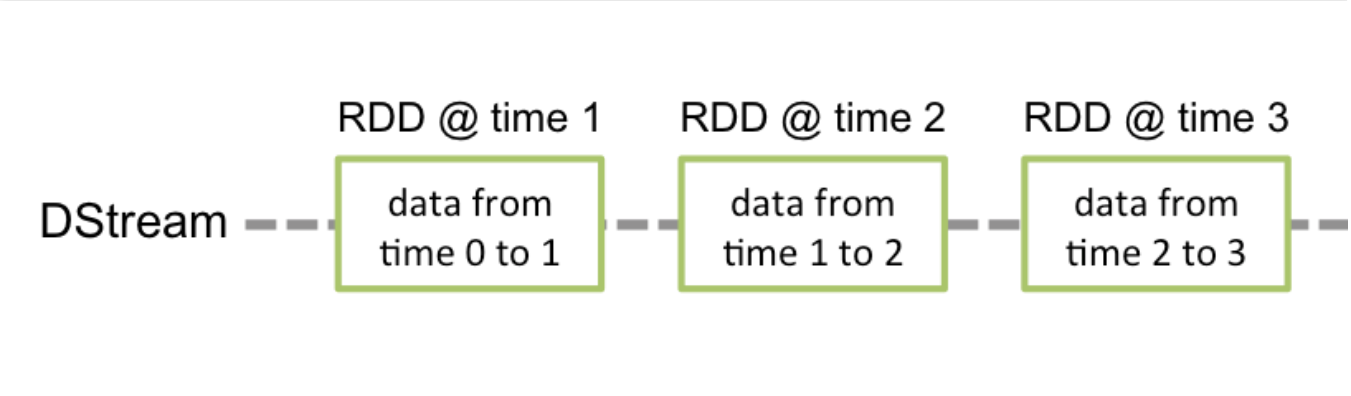
Connecting Spark Streaming to CKafka

Last updated：2024-01-09 14:56:36

As an extension of Spark Core, Spark Streaming is used for high-throughput and fault-tolerant processing of continuous data. Currently supported external input sources include Kafka, Flume, HDFS/S3, Kinesis, Twitter, and TCP socket.



Spark Streaming abstracts continuous data into a Discretized Stream (DStream), which consists of a series of continuous resilient distributed datasets (RDDs). Each RDD contains data generated at a certain time interval. Processing DStream with functions is actually processing these RDDs.



When Spark Streaming is used as data input for Kafka, the following stable and experimental Kafka versions are supported:

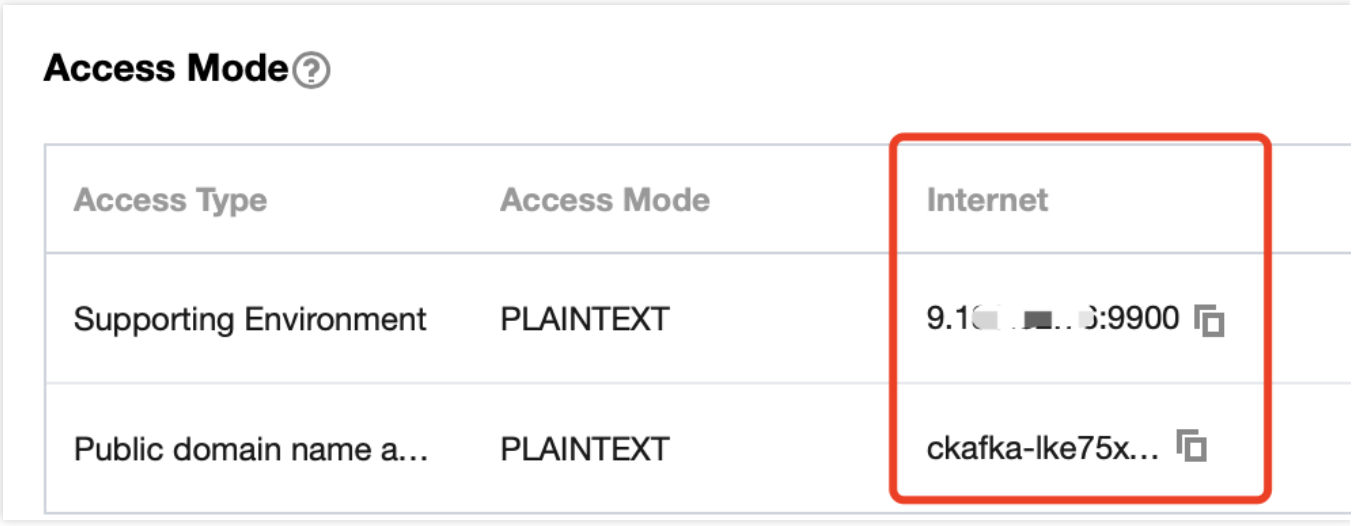
Kafka Version	spark-streaming-kafka-0.8	spark-streaming-kafka-0.10
Broker Version	0.8.2.1 or later	0.10.0 or later
API Maturity	Deprecated	Stable
Language Support	Scala, Java, and Python	Scala and Java
Receiver DStream	Yes	No
Direct DStream	Yes	Yes
SSL / TLS Support	No	Yes
Offset Commit API	No	Yes
Dynamic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

Currently, CKafka is compatible with version above 0.9. The Kafka dependency of v0.10.2.1 is used in this practice scenario.
In addition, Spark Streaming in EMR also supports direct connection to CKafka. For more information, see Connecting Spark Streaming to CKafka.

Directions

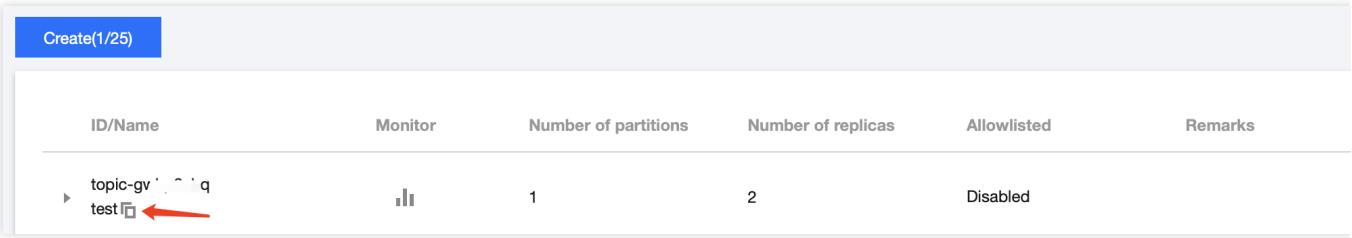
Step 1. Get the CKafka instance access address

1. Log in to the CKafka console.
2. Select **Instance List** on the left sidebar and click the **ID** of the target instance to enter its basic information page.
3. On the instance's basic information page, get the instance access address in the **Access Mode** module, which is the `bootstrap-server` required by production and consumption.



Step 2. Create a topic

1. On the instance's basic information page, select the **Topic Management** tab at the top.
2. On the topic management page, click **Create** to create a topic named `test`. This topic is used as an example below to describe how to produce and consume messages.



Step 3. Prepare the CVM environment

CentOS 6.8

Package	Version
sbt	0.13.16
Hadoop	2.7.3
Spark	2.1.0
Protobuf	2.5.0
SSH	Installed on CentOS by default
Java	1.8

For specific installation steps, see [Configuring environment](#Configuring environment).

Step 4. Connect to CKafka

Producing Messages to CKafka

Consuming Messages from CKafka

The Kafka dependency of v0.10.2.1 is used here.

1. Add dependencies to `build.sbt` :

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

2. Configure `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") // Private IP and port in t

     props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerial
     props.put("value.serializer", "org.apache.kafka.common.serialization.StringSeri

     val producer = new KafkaProducer[String, String](props)
     val TOPIC="test"  // Specify the topic to produce to
     for(i<- 1 to 50){
            val record = new ProducerRecord(TOPIC, "key", s"hello $i") // Produce a
            producer.send(record)
     }
     val record = new ProducerRecord(TOPIC, "key", "the end "+new java.util.Date)
     producer.send(record)
     producer.close() // Disconnect at the end
    }
```

For more information on how to use `ProducerRecord` , see ProducerRecord.

DirectStream

1. Add de

pe

ndencies to `build.sbt` :

```
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. Configure `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=>
            // Output the obtained message
            rdd.foreach{iter =>
                val i = iter.value
                println(s"${i}")
            }
            // Get the offset
            val offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges
            rdd.foreachPartition { iter =>
                val o: OffsetRange = offsetRanges(TaskContext.get.partitionId)
                println(s"${o.topic} ${o.partition} ${o.fromOffset} ${o.untilOffset
            }
        }

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

RDD

1. Configure `build.sbt` in the way as detailed here.
2. Configure `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
        // Pull messages in the corresponding offset range from the partition in or
        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()
    }
}
```

For more information on how to use `kafkaParams`, see kafkaParams.

Configuring environment[](id:Configuring environment)

Installing sbt

1. Download the sbt package from sbt's official website.

2. After decompression, create an `sbt_run.sh` script with the following content in the sbt directory and add executable permissions:

```
#!/bin/bash
SBT_OPTS="-Xms512M -Xmx1536M -Xss1M -XX:+CMSClassUnloadingEnabled -XX:MaxPermSize=2
java $SBT_OPTS -jar `dirname $0`/bin/sbt-launch.jar "$@"
```

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

3. Run the following command:

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

The display of sbt version indicates a successful installation.

Installing Protobuf

1. Download an appropriate version of Protobuf.
2. Decompress and enter the directory.

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

You should install gcc-g++ in advance, and the root permission may be required during installation.

3. Log in again and enter the following on the command line:

```
protoc --version
```

4. The display of Protobuf version indicates a successful installation.

Installing Hadoop

1. Download the required version at Hadoop's official website.
2. Add a Hadoop user.

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

3. Grant admin permissions.

```
visudo
```

4. Add the following in a new line under `root ALL=(ALL) ALL`:

`hadoop ALL=(ALL) ALL`

Save and exit.

5. Use Hadoop for operations.

```
su hadoop
```

6. Configure SSH password-free login.

```
cd ~/.ssh/                     # If there is no such directory, run `ssh localhost`
ssh-keygen -t rsa              # There will be prompts. Simply press Enter
cat id_rsa.pub >> authorized_keys  # Add authorization
chmod 600 ./authorized_keys    # Modify file permission
```

7. Install Java.

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

8. Configure `${JAVA_HOME}` .

```
vim /etc/profile
```

Add the following at the end:

```
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
```

Modify the corresponding path based on the installation information.

9. Decompress Hadoop and enter the directory.

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

The display of version information indicates a successful installation.

10. Configure the pseudo-distributed mode (so that you can build different forms of clusters as needed).

```
vim /etc/profile
```

Add the following at the end:

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

Modify the corresponding path based on the installation information.

11. Modify `/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. Modify `/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. Change `JAVA_HOME` in `/etc/hadoop/hadoop-env.sh` to the Java path.

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

14. Format the NameNode.

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

The display of `Exitting with status 0` indicates a success.

15. Start Hadoop.

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

`NameNode` , `DataNode` , and `SecondaryNameNode` processes will exist upon successful startup.

Installing Spark

Download the required version at Spark's official website.

As Hadoop has already been installed, select `Pre-build with user-provided Apache Hadoop` here.

Note:

This example also uses the `hadoop` user for operations.

1. Decompress and enter the directory.

2. Modify the configuration file.

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

Add the following in the first line:

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

Modify the path based on the Hadoop installation information.

3. Run the example.

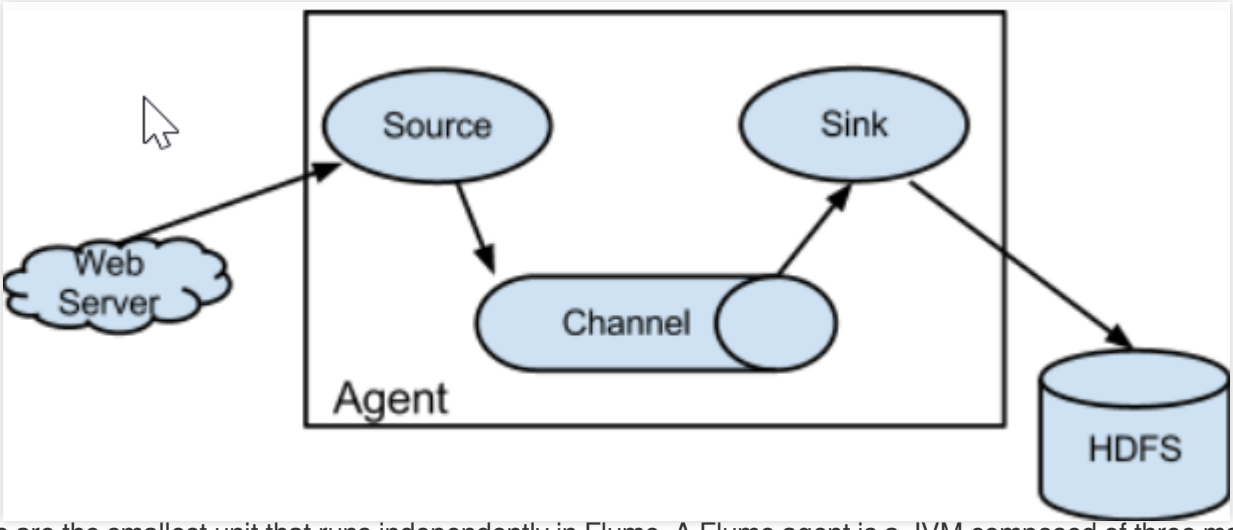
```
bin/run-example SparkPi
```

The display of an approximate value of `π` output by the program indicates a successful installation.

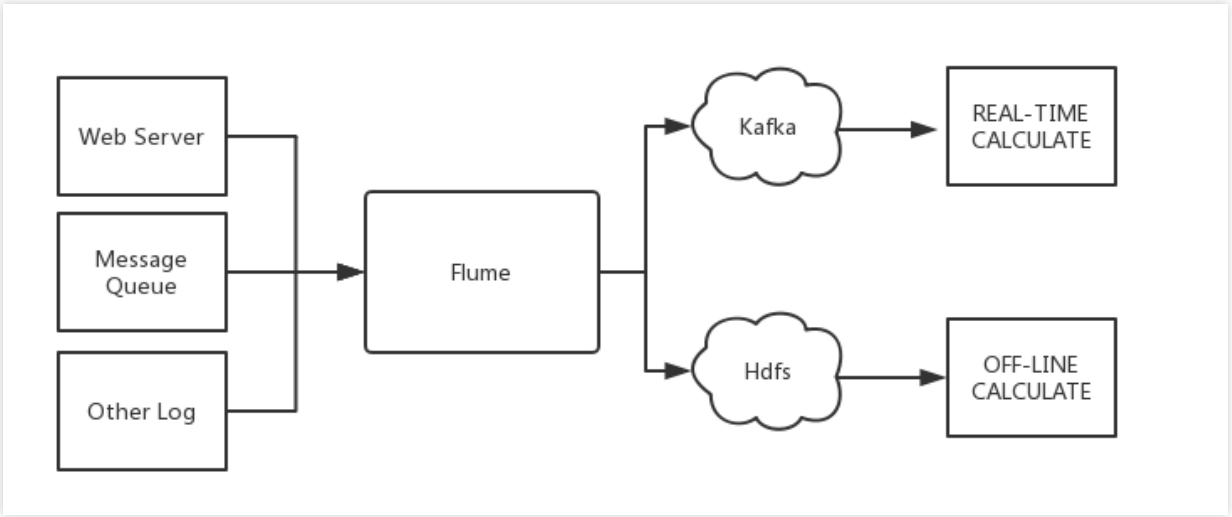
Connecting Flume to CKafka

Last updated：2024-01-09 14:56:36

Apache Flume is a distributed, reliable, and highly available log collection system that supports a wide variety of data sources such as HTTP, log files, JMS, and listening ports. It can efficiently collect, aggregate, move, and store massive amounts of log data to a specified storage system like Kafka, HDFS, and Solr search server.
Flume is structured as follows:



Agents are the smallest unit that runs independently in Flume. A Flume agent is a JVM composed of three main components: source, sink, and channel.



Flume and Kafka

When you store data in a downstream storage module or compute module such as HDFS or HBase, you need to consider a lot of complex factors such as the number of concurrent writes, system load, and network delay. As a

flexible distributed system, Flume provides various APIs and customizable pipelines.
In the production process, Kafka can act as a cache when the production and consumption are at different paces. It has a high throughput thanks to the partition structure and data appending feature. It is also very fault-tolerant because of the replication structure.
Therefore, Flume and Kafka can work together to meet most requirements in production environments.

Connecting Flume to Open-Source Kafka

Preparations

Download Apache Flume (v1.6.0 or later is compatible with Kafka).
Download Kafka (v0.9.x or later is required as v0.8 is no longer supported).
Confirm that Kafka's source and sink components are already in Flume.

Connection method

Kafka can be used as a source or sink to import or export messages.
Using Kafka as a Source
Using Kafka as a Sink
Configure Kafka as the message source, that is, pull data as a consumer from Kafka into a specified sink. The main configuration items are as follows:

Configuration Item	Description
channels	The configured channel
type	It must be `org.apache.flume.source.kafka.KafkaSource`
kafka.bootstrap.servers	Kafka broker server address
kafka.consumer.group.id	ID of Kafka's consumer group
kafka.topics	Data source topics in Kafka
batchSize	Size of each write into the channel
batchDurationMillis	The maximum write interval

Sample:

```
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
```

For more information, visit Apache Flume's official website.

Configure Kafka as the message receiver, that is, push data to the Kafka server as a producer for subsequent operations. The main configuration items are as follows:

Configuration Item	Description
channel	The configured channel
type	It must be `org.apache.flume.sink.kafka.KafkaSink`
kafka.bootstrap.servers	Kafka broker server
kafka.topics	Data target topics in Kafka
kafka.flumeBatchSize	Size of each written batch
kafka.producer.acks	Production policy of Kafka producer

Sample:

```
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
```

For more information, visit Apache Flume's official website.

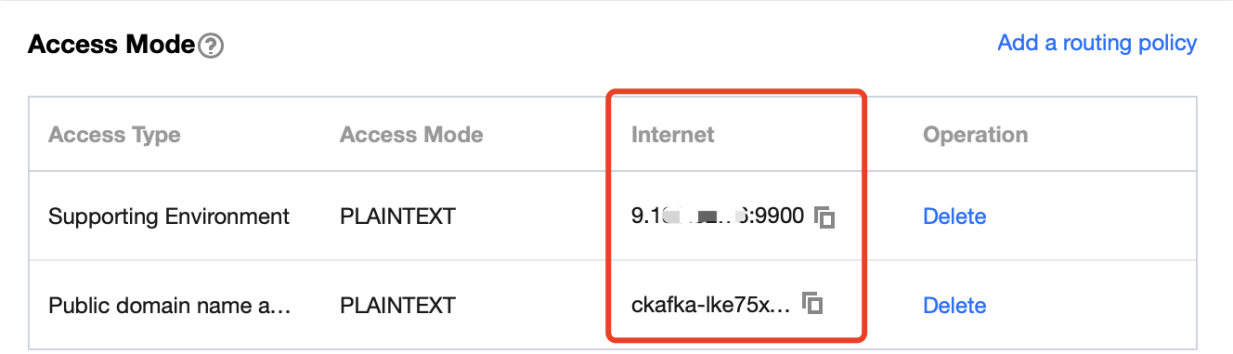
Connecting Flume to CKafka

Using CKafka as a Sink

Using CKafka as a Source

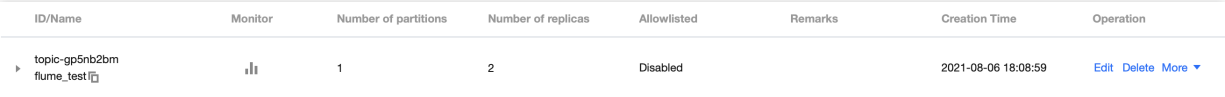
Step 1. Obtain the CKafka instance access address

1. Log in to the CKafka console.
2. Select **Instance List** on the left sidebar and click the **ID** of the target instance to enter the instance details page.
3. You can obtain the instance access address in the **Access Mode** module on the **Basic Info** tab page.



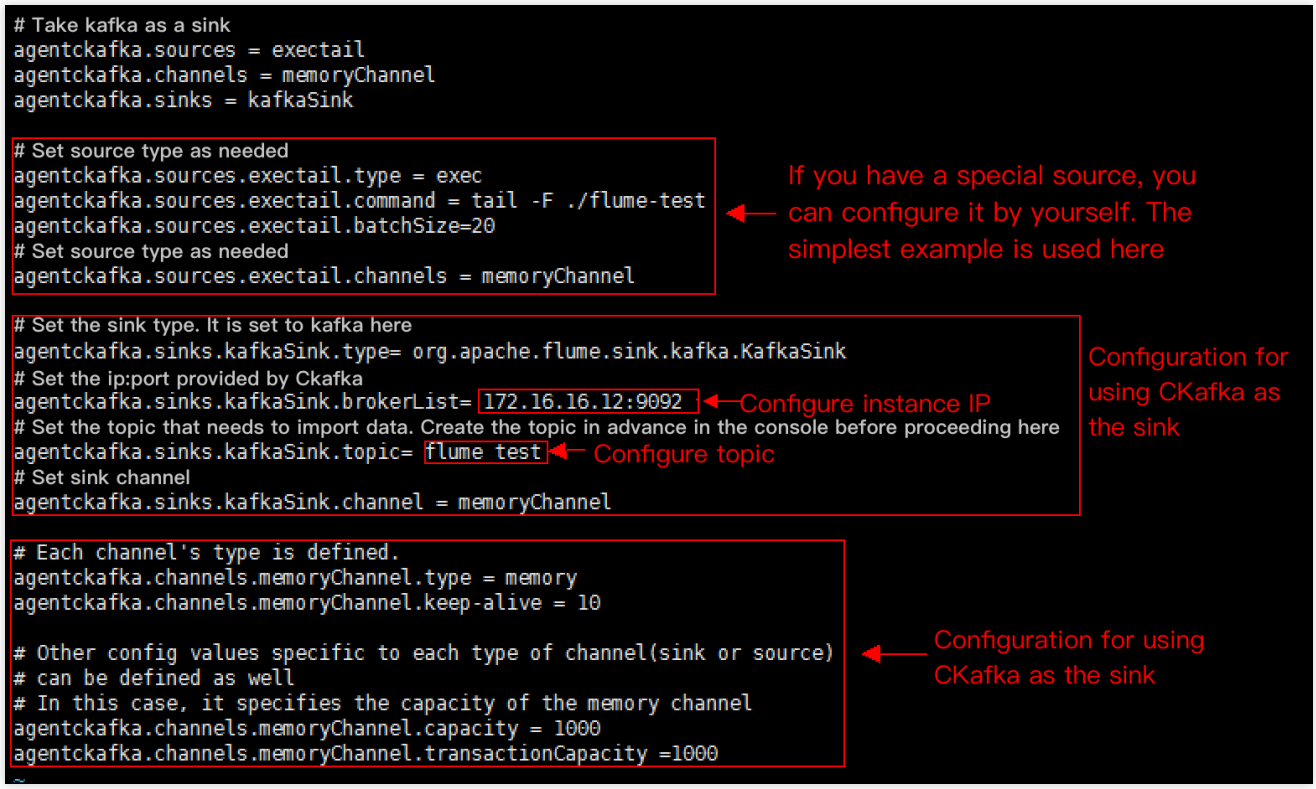
Step 2. Create a topic

1. On the instance details page, select the **Topic Management** tab at the top.
2. On the topic management page, click **Create** to create a topic named `flume_test` .



Step 3. Configure Flume

1. Download the Apache Flume toolkit and decompress it.
2. Write the configuration file `flume-kafka-sink.properties` . Below is a simple demo (configured in the `conf` folder in the extracted directory) for Java. If there is no special requirement, simply replace your own instance IP address and topic in the configuration file. In this demo, the source is `tail -F flume-test` , which is the newly added information in the file.



The sample code is as shown below:

```
# Demo for using Kafka as the sink
agentckafka.source = exectail
agentckafka.channels = memoryChannel
agentckafka.sinks = kafkaSink

# Set the source type based on different requirements. If you have a special source
agentckafka.sources.exectail.type = exec
agentckafka.sources.exetail.command = tail -F ./flume.test
agentckafka.sources.exectail.batchSize = 20
# Set the source channel
agentckafka.sources.exectail.channels = memoryChannel

# Set the sink type. It is set to Kafka here
agentckafka.sinks.kafkaSink.type = org.apache.flume.sink.kafka.KafkaSink
# Set the ip:port provided by CKafka
agentckafka.sinks.kafkaSink.brokerList = 172.16.16.12:9092 # Configure the instance
# Set the topic to which data is to be imported. Create the topic in the CKafka con
agentckafka.sinks.kafkaSink.topic = flume test #Configure the topic
# Set the sink channel
agentckafka.sinks.kafkaSink.channel = memoryChannel

# Use the default configuration for the channel
# Each channel's type is defined
agentckafka.channels.memoryChannel.type = memory
```

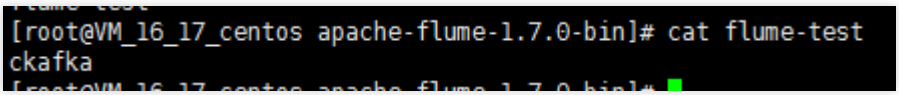
```
agentckafka.channels.memoryChannel.keep-alive = 10

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

3. Run the following command to start Flume:

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

4. Write messages to the `flume-test` file. At this time, the messages will be written by Flume to CKafka.



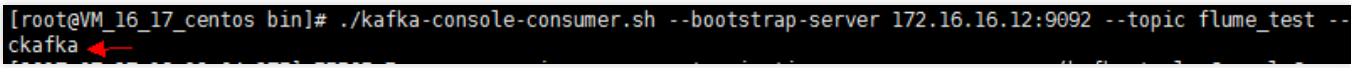
5. Start the CKafka client for consumption.

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

Note:

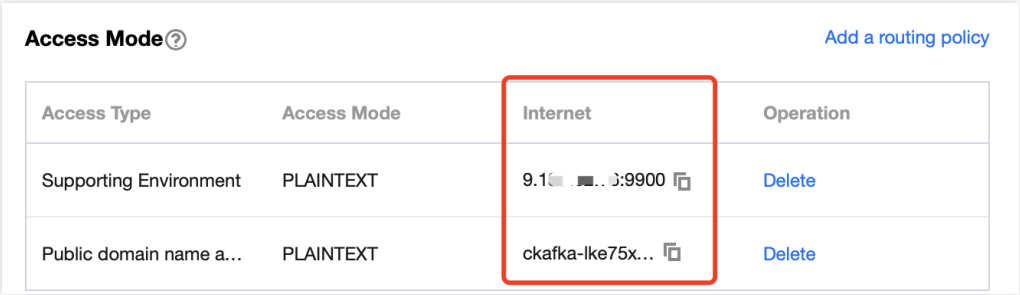
Enter the access address of the CKafka instance just created for the `bootstrap-server` field and the name of the topic just created for `topic` .

You can see that the messages have been consumed.



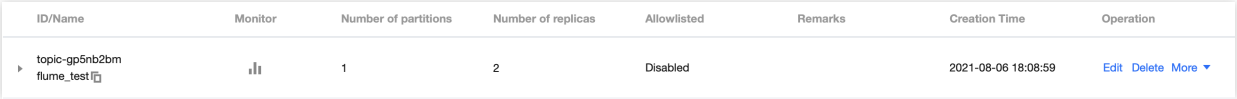
Step 1. Obtain the CKafka instance access address

1. Log in to the CKafka console.
2. Select **Instance List** on the left sidebar and click the **ID** of the target instance to enter the instance details page.
3. You can obtain the instance access address in the **Access Mode** module on the **Basic Info** tab page.



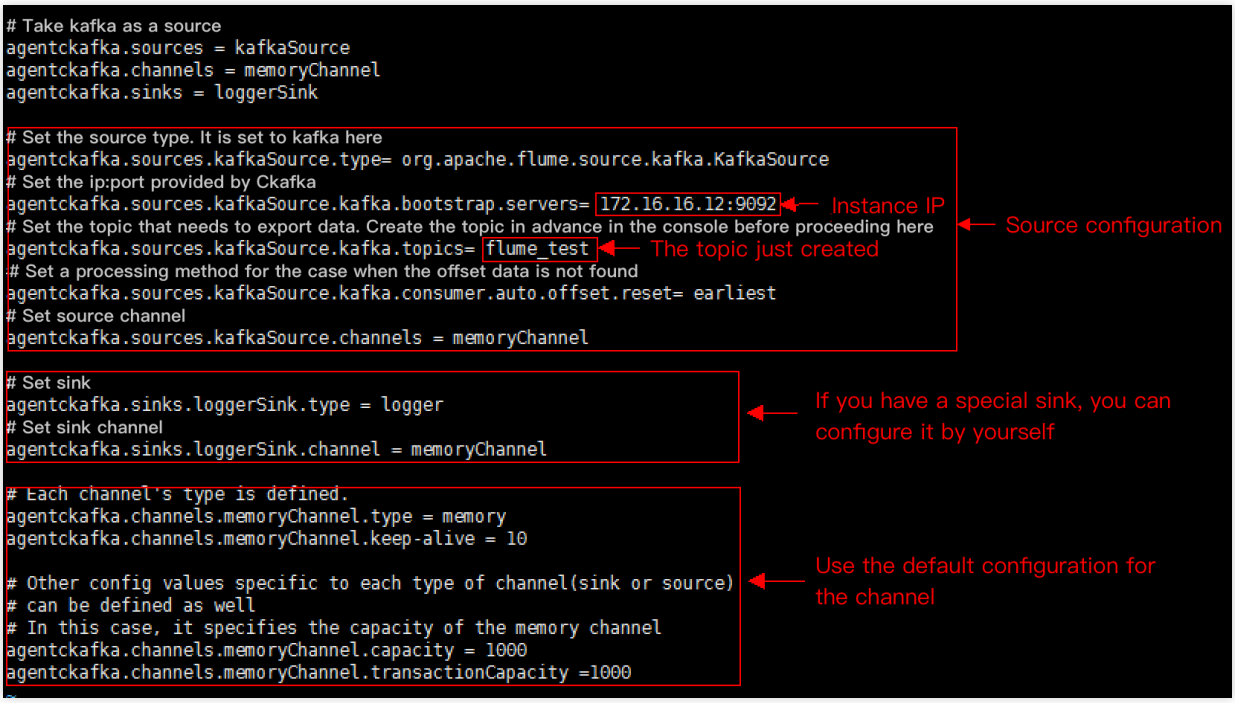
Step 2. Create a topic

1. On the instance details page, select the **Topic Management** tab at the top.
2. On the topic management page, click **Create** to create a topic named `flume_test` .



Step 3. Configure Flume

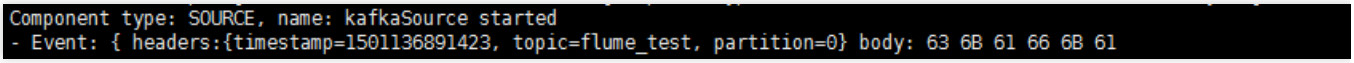
1. Download the Apache Flume toolkit and decompress it.
2. Write the configuration file `flume-kafka-source.properties` . Below is a simple demo (configured in the `conf` folder in the extracted directory). If there is no special requirement, simply replace your own instance IP address and topic in the configuration file. The sink is `logger` in this example.



3. Run the following command to start Flume:

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

4. View the logger output information. The default path is `logs/flume.log` .



Connecting Kafka Connect to CKafka

Last updated：2024-01-09 14:56:36

Kafka Connect currently supports two execution modes: standalone and distributed.

Starting Connect in Standalone Mode

Start Connect in standalone mode by running the following command:

```
bin/connect-standalone.sh config/connect-standalone.properties connector1.propertie
```

Accessing CKafka is basically the same as accessing the open-source Kafka, except that you need to change the value of `bootstrap.servers` to the IP address assigned when you apply for the instance.

Starting Connect in Distributed Mode

Start Connect in distributed mode by running the following command:

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

In this mode, Kafka Connect stores the offset, configuration, and task status information in Kafka topics, which are configured in the following fields in `connect-distributed` :

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

These topics need to be created manually to ensure that their attributes meet the requirements of Connect.

`config.storage.topic` should have only one partition and multiple replicas and be in compact mode.

`offset.storage.topic` should have multiple partitions and replicas and be in compact mode.

`status.storage.topic` should have multiple partitions and replicas and be in compact mode.

Set `bootstrap.servers` to the IP address assigned when you apply for the instance.

Configure `group.id` to identify the Connect cluster, which should be differentiated from the consumer group.

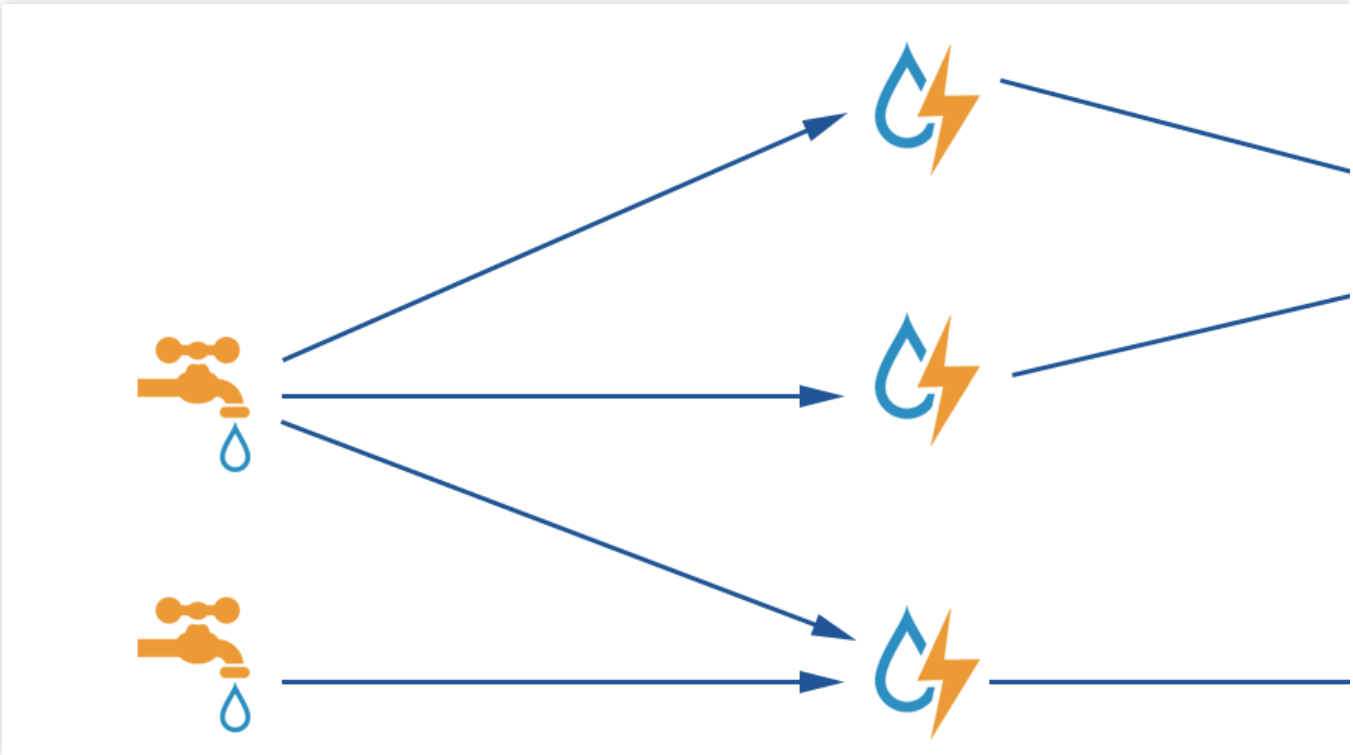
Connecting Storm to CKafka

Last updated：2024-01-09 14:56:36

Storm is a distributed real-time computing framework that can perform stream-based data processing and provide universal distributed RPC calling so as to reduce the delay of event processing down to sub-seconds. It is suitable for real-time data processing scenarios where low delay is required.

How Storm Works

There are two types of nodes in a Storm cluster: `master node` and `worker node`. The `Nimbus` process runs on the `master node` for resource allocation and status monitoring, and the `Supervisor` process runs on the `worker node` for listening on work tasks and starting the `executor`. The entire Storm cluster relies on `ZooKeeper` for common data storage, cluster status listening, task assignment, etc.
A data processing program submitted to Storm is called a `topology`. The minimum message unit it processes is `tuple` (an array of arbitrary objects). A `topology` consists of `spout` and `bolt`, where `spout` is the source of `tuple`, while `bolt` can subscribe to any `tuple` issued by `spout` or `bolt` for processing.



Storm with CKafka

Storm can use CKafka as a `spout` to consume data for processing or as a `bolt` to store the processed data for consumption by other components.

Testing environment

CentOS 6.8

Package	Version
Maven	3.5.0
Storm	2.1.0
SSH	5.3
Java	1.8

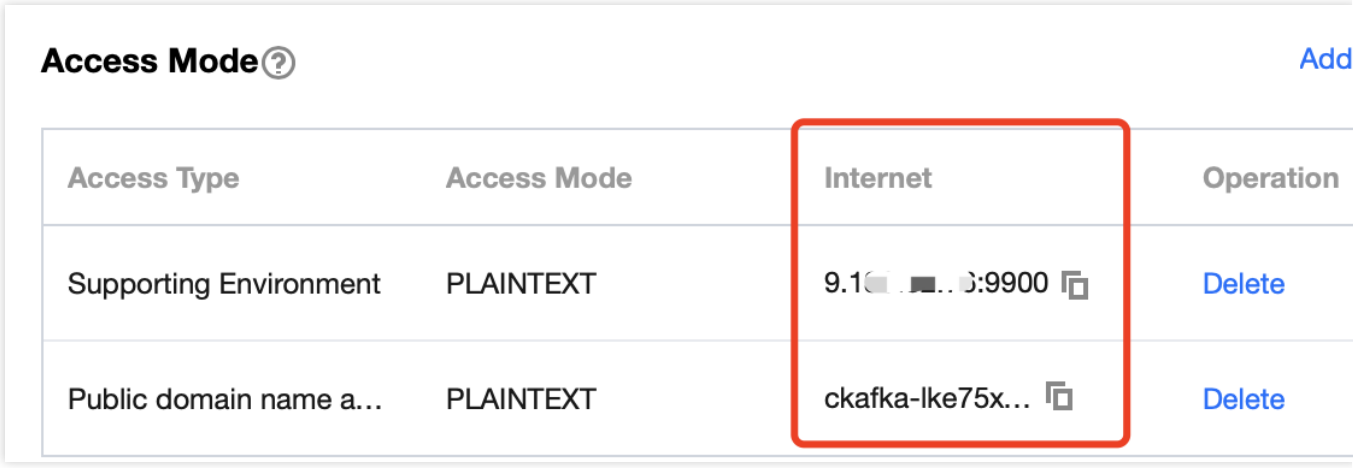
Prerequisites

Download and install JDK 8. For detailed directions, see Java SE Development Kit 8 Downloads.
Download and install Storm. For more information, see Apache Storm downloads.
You have created a CKafka instance.

Directions

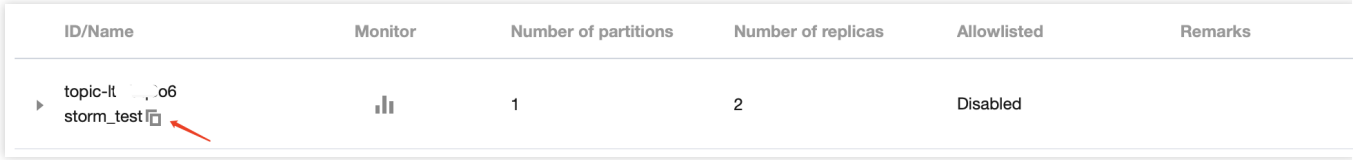
Step 1. Get the CKafka instance access address

1. Log in to the CKafka console.
2. Select **Instance List** on the left sidebar and click the **ID** of the target instance to enter its basic information page.
3. On the instance's basic information page, get the instance access address in the **Access Mode** module.



Step 2. Create a topic

1. On the instance's basic information page, select the **Topic Management** tab at the top.
2. On the topic management page, click **Create** to create a topic.



Step 3. Add Maven dependencies

Configure `pom.xml` as follows:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/200
  <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>
```

Step 4. Produce a message

Using spout/bolt

Topology code:

```
//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 {
    // `ip:port` of the CKafka instance applied for
    private final static String BOOTSTRAP_SERVERS = "xx.xx.xx.xx:xxxx";
    // Specify the topic to which to write messages
    private final static String TOPIC = "storm_test";
    public static void main(String[] args) throws Exception {
        // Set producer attributes
        // For functions, visit https://kafka.apache.org/0100/javadoc/index.html?or
        // For attributes, visit 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.Str
        properties.put("value.serializer", "org.apache.kafka.common.serialization.S

        // Create a bolt to be written to Kafka. `fields("key" "message")` is used
        KafkaBolt kafkaBolt = new KafkaBolt()
```

```
                .withProducerProperties(properties)
                .withTopicSelector(new DefaultTopicSelector(TOPIC))
                .withTupleToKafkaMapper(new FieldNameBasedTupleToKafkaMapper());
        TopologyBuilder builder = new TopologyBuilder();
        // A spout class that generates messages in sequence with the output field
        SerialSentenceSpout spout = new SerialSentenceSpout();
        AddMessageKeyBolt bolt = new AddMessageKeyBolt();
        builder.setSpout("kafka-spout", spout, 1);
        // Add the fields required to produce messages to CKafka for the tuple
        builder.setBolt("add-key", bolt, 1).shuffleGrouping("kafka-spout");
        // Write to CKafka
        builder.setBolt("sendToKafka", kafkaBolt, 8).shuffleGrouping("add-key");

        Config config = new Config();
        if (args != null && args.length > 0) {
            // Cluster mode, which is used to package a jar file and run it in Stor
            config.setNumWorkers(1);
            StormSubmitter.submitTopologyWithProgressBar(args[0], config, builder.c
        } else {
            // Local mode
            LocalCluster cluster = new LocalCluster();
            cluster.submitTopology("test", config, builder.createTopology());
            Utils.sleep(10000);
            cluster.killTopology("test");
            cluster.shutdown();
        }

    }
}
```

Create a spout class that generates messages in sequence:

```
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
        this.spoutOutputCollector = spoutOutputCollector;
    }

    @Override
    public void nextTuple() {
        Utils.sleep(1000);
        // Produce a `UUID` string and send it to the next component
        spoutOutputCollector.emit(new Values(UUID.randomUUID().toString()));
    }

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

Add `key` and `message` fields to the `tuple`. If `key` is null, the produced messages will be evenly allocated to each partition. If a key is specified, the messages will be hashed to specific partitions based on the key value:

```
//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) {
        // Take out the first field value
        String messae = tuple.getString(0);
//        System.out.println(messae);
        // Send to the next component
        basicOutputCollector.emit(new Values(null, messae));
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer) {
        // Create a schema to send to the next component
        outputFieldsDeclarer.declare(new Fields("key", "message"));
    }
```

```
}
```

Using trident

Use the trident class to generate a 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 {
    // `ip:port` of the CKafka instance applied for
    private final static String BOOTSTRAP_SERVERS = "xx.xx.xx.xx:xxxx";
    // Specify the topic to which to write messages
    private final static String TOPIC = "storm_test";
    public static void main(String[] args) throws Exception {
        // Set producer attributes
        // For functions, visit https://kafka.apache.org/0100/javadoc/index.html?or
        // For attributes, visit 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.Str
        properties.put("value.serializer", "org.apache.kafka.common.serialization.S
        // Set the trident
        TridentKafkaStateFactory stateFactory = new TridentKafkaStateFactory()
                .withProducerProperties(properties)
                .withKafkaTopicSelector(new DefaultTopicSelector(TOPIC))
                // Set to use `fields("key", "value")` as the written message, whic
                .withTridentTupleToKafkaMapper(new FieldNameBasedTupleToKafkaMapper
        TridentTopology builder = new TridentTopology();
        // A spout that generates messages in batches with the output field being `
```

```
        builder.newStream("kafka-spout", new TridentSerialSentenceSpout(5))
                .each(new Fields("sentence"), new AddMessageKey(), new Fields("key"
                .partitionPersist(stateFactory, new Fields("key", "value"), new Tri

        Config config = new Config();
        if (args != null && args.length > 0) {
            // Cluster mode, which is used to package a jar file and run it in Stor
            config.setNumWorkers(1);
            StormSubmitter.submitTopologyWithProgressBar(args[0], config, builder.b
        } else {
            // Local mode
            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 tridentColl
            // Take out the first field value
            String messae = tridentTuple.getString(0);
            //System.out.println(messae);
            // Send to the next component
            //tridentCollector.emit(new Values(Integer.toString(messae.hashCode()),
            tridentCollector.emit(new Values(null, messae));
        }
    }
}
```

Create a spout class that generates messages in batches:

```
//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");
    }
}
```

Step 5. Consume the message

Using 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 {
    // `ip:port` of the CKafka instance applied for
    private final static String BOOTSTRAP_SERVERS = "xx.xx.xx.xx:xxxx";
    // Specify the topic to which to write messages
    private final static String TOPIC = "storm_test";

    public static void main(String[] args) throws Exception {
        // Set a retry policy
        KafkaSpoutRetryService kafkaSpoutRetryService = new KafkaSpoutRetryExponent
                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.
                new Fields("topic", "partition", "offset", "key", "value"));
        // Set consumer parameters
        // For functions, visit http://storm.apache.org/releases/1.1.0/javadocs/org
        // For parameters, visit http://kafka.apache.org/0102/documentation.html
        KafkaSpoutConfig spoutConfig = KafkaSpoutConfig.builder(BOOTSTRAP_SERVERS,
                .setProp(new HashMap<String, Object>(){{
```

```
                    put(ConsumerConfig.GROUP_ID_CONFIG, "test-group1"); // Set the
                    put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, "50000"); // Set
                    put(ConsumerConfig.REQUEST_TIMEOUT_MS_CONFIG, "60000"); // Set
                }})
                .setOffsetCommitPeriodMs(10_000) // Set the automatic confirmation
                .setFirstPollOffsetStrategy(LATEST) // Set to pull the latest messa
                .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 outputFieldsDeclar

            }

            @Override
            public void prepare(Map map, TopologyContext topologyContext, OutputCol
                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.c
        }
        else {
            LocalCluster cluster = new LocalCluster();
            cluster.submitTopology("test", config, builder.createTopology());
            Utils.sleep(20000);
            cluster.killTopology("test");
            cluster.shutdown();
        }
    }
}
```

Using 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 {
    // `ip:port` of the CKafka instance applied for
    private final static String BOOTSTRAP_SERVERS = "xx.xx.xx.xx:xxxx";
    // Specify the topic to which to write messages
    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.
                new Fields("topic", "partition", "offset", "key", "value"));
        // Set consumer parameters
        // For functions, visit http://storm.apache.org/releases/1.1.0/javadocs/org
        // For parameters, visit http://kafka.apache.org/0102/documentation.html
        KafkaTridentSpoutConfig spoutConfig = KafkaTridentSpoutConfig.builder(BOOTS
                .setProp(new HashMap<String, Object>(){{
                    put(ConsumerConfig.GROUP_ID_CONFIG, "test-group1"); // Set the
                    put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true"); // Set a
                    put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, "50000"); // Set
                    put(ConsumerConfig.REQUEST_TIMEOUT_MS_CONFIG, "60000"); // Set
                }})
```

```
                .setFirstPollOffsetStrategy(LATEST) // Set to pull the latest messa
                .setRecordTranslator(trans)
                .build();

        TridentTopology builder = new TridentTopology();
//      Stream spoutStream = builder.newStream("spout", new KafkaTridentSpoutTransa
        Stream spoutStream = builder.newStream("spout", new KafkaTridentSpoutOpaque
        spoutStream.each(spoutStream.getOutputFields(), new BaseFunction(){
            @Override
            public void execute(TridentTuple tridentTuple, TridentCollector trident
                System.out.println(tridentTuple.getStringByField("value"));
                tridentCollector.emit(new Values(tridentTuple.getStringByField("val
            }
        }, 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.bui
        }
        else {
            StormTopology stormTopology = builder.build();
            LocalCluster cluster = new LocalCluster();
            cluster.submitTopology("test", conf, stormTopology);
            Utils.sleep(10000);
            cluster.killTopology("test");
            cluster.shutdown();stormTopology.clear();
        }
    }
}
```

Step 6. Submit Storm

After being compiled with `mvn package`, Storm can be submitted to the local cluster for debugging or submitted to the production cluster for running.

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

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

Connecting Logstash to CKafka

Last updated：2022-08-11 17:23:38

Logstash is an open-source log processing tool that can be used to collect data from multiple sources, filters it, and then stores it for other uses.

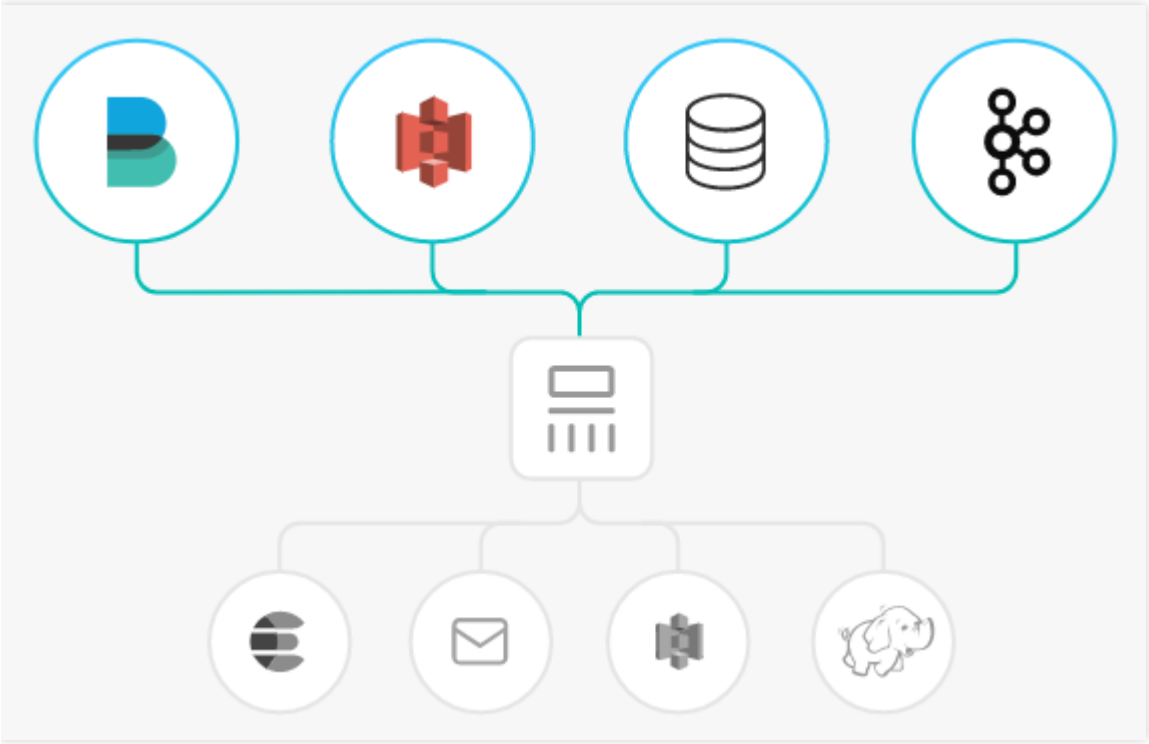
Logstash is highly flexible and has powerful syntax analysis capabilities. With a variety of plugins, it supports multiple types of inputs and outputs. In addition, as a horizontally scalable data pipeline, it has powerful log collection and retrieval features that work with Elasticsearch and Kibana.

How Logstash Works

The Logstash data processing pipeline can be divided into three stages: inputs → filters → outputs.

1. Inputs: Collect data from multiple sources like file, syslog, redis, and beats.
2. Filters: Modify and filter the collected data. Filters are intermediate processing components in the Logstash data pipeline. They can modify events based on specific conditions. Some commonly used filters are grok, mutate, drop, and clone.
3. Outputs: Transfer the processed data to other destinations. An event can be transferred to multiple outputs, and the event ends when the transfer is completed. Elasticsearch is the most commonly-used output.

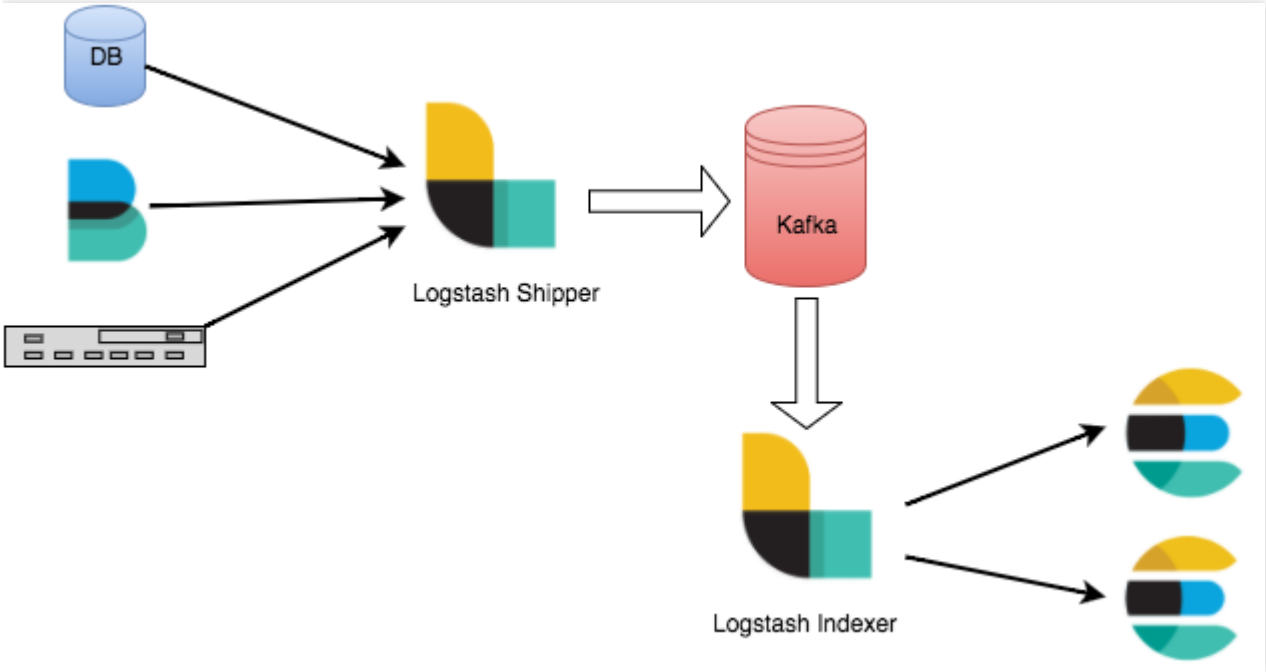
In addition, Logstash supports encoding and decoding data, so you can specify data formats on the input and output ends.



Strengths of Connecting Logstash to Kafka

- Data can be asynchronously processed to prevent traffic spikes.
- Components are decoupled, so when an exception occurs in Elasticsearch, the upstream work will not be affected.

Note：
Logstash consumes resources when processing data. If you deploy Logstash on a production server, the performance of the server may be affected.



Directions

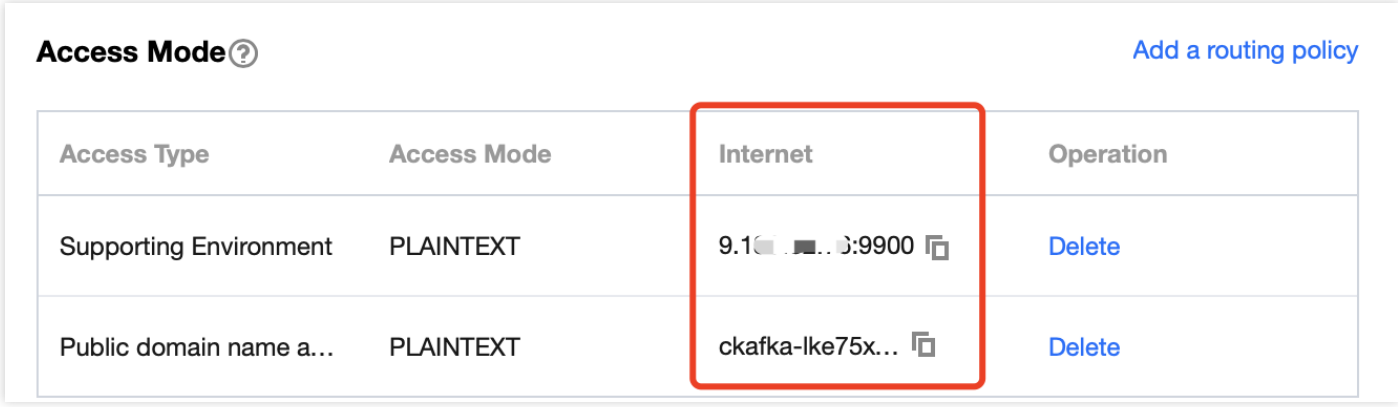
Preparations

- Download and install Logstash as instructed in Installing Logstash.
- Download and install JDK 8 as instructed in Java SE Development Kit 8u341.
- Create a CKafka instance as instructed in Creating Instance.

Step 1. Get the CKafka instance access address

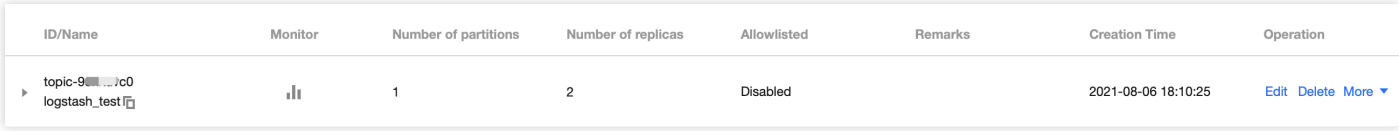
1. Log in to the CKafka console.
2. Select **Instance List** on the left sidebar and click the **ID** of the target instance to enter its basic information page.

3. On the instance's basic information page, get the instance access address in the **Access Mode** module.



Step 2. Create a topic

1. On the instance's basic information page, select the **Topic Management** tab at the top.
2. On the topic management page, click **Create** to create a topic named `logstash_test` .

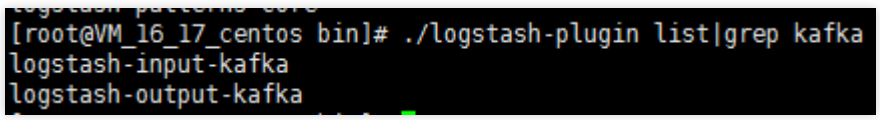


Step 3. Connect to CKafka

Note：
You can click the following tabs to view the detailed directions for using CKafka as `inputs` or `outputs` .

- Connecting as inputs
- Connecting as outputs

1. Run `bin/logstash-plugin list` to check whether `logstash-input-kafka` is included in the supported plugins.



2. Write the configuration file `input.conf` in the `.bin/` directory.

   In the following example, Kafka is used as the data source, and the standard output is taken as the data destination.

```
input {
kafka {
bootstrap_servers => "xx.xx.xx.xx:xxxx" // CKafka instance access address
group_id => "logstash_group" // CKafka group ID
topics => ["logstash_test"] // CKafka topic name
consumer_threads => 3 // Number of consumer threads, which is generally the sam
e as the number of CKafka partitions
auto_offset_reset => "earliest"
}
}
output {
stdout{codec=>rubydebug}
}
```

3. Run the following command to start Logstash and consume messages.

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

   The returned result is as follows:

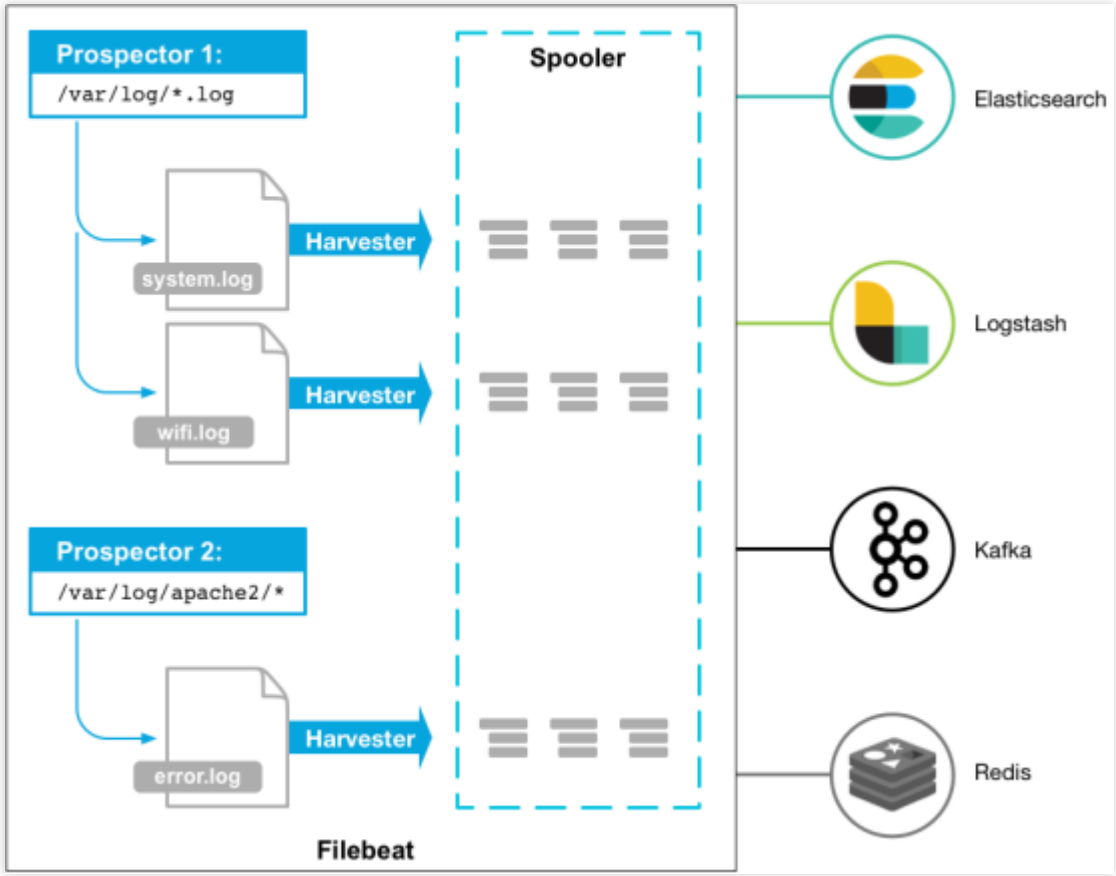
```
[root@VM_16_17_centos bin]# ./logstash -f input.conf
ERROR StatusLogger No log4j2 configuration file found. Using default configuration: logging only errors to the console.
Sending Logstash's logs to /data/ryan/logstash-5.5.2/logs which is now configured via log4j2.properties
[2017-09-06T18:07:41,926][INFO ][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"
}
```

   You can see that the data in the topic above has been consumed now.

Connecting Filebeat to CKafka

Last updated：2024-01-09 14:56:36

The Beats platform offers various single-purpose data shippers. Once installed, these shippers can be used as lightweight agents to send the collected data from hundreds or thousands of machines to the target systems.



Beats offers a wide variety of shippers. You can download the most appropriate one based on your needs. This document uses Filebeat, a lightweight log shipper, as an example to describe how to connect Filebeat to CKafka and handle common problems that may occur after the connection.

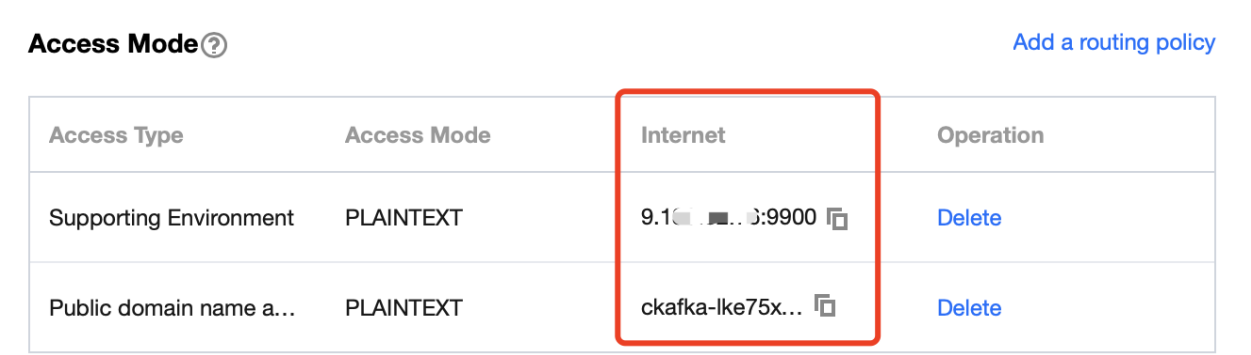
Prerequisites

You have downloaded and installed Filebeat. For more information, see Filebeat quick start: installation and configuration.
You have downloaded and installed JDK 8. For more information, see Java Downloads.
You have created a CKafka instance. For more information, see Creating Instance.

Directions

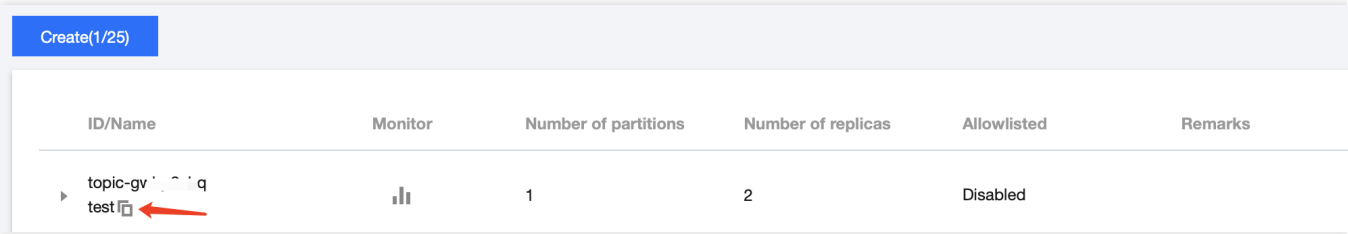
Step 1. Obtain the CKafka instance access address

1. Log in to the CKafka console.
2. Select **Instance List** on the left sidebar and click the **ID** of the target instance to enter the instance details page.
3. You can obtain the instance access address in the **Access Mode** module on the **Basic Info** tab page.



Step 2. Create a topic

1. On the instance details page, select the **Topic Management** tab at the top.
2. On the topic management page, click **Create** to create a topic named `test` .



Step 3. Prepare the configuration file

Enter the installation directory of Filebeat and create the configuration monitoring file `filebeat.yml` .

```
#======= For Filebeat 7.x or later versions, change `filebeat.prospectors` to `file
filebeat.prospectors:

- input_type: log

# This is the path to the monitoring file.
  paths:
```

```
    - /var/log/messages

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

#------------------ kafka -------------------------------------
output.kafka:
  version:0.10.2 // Set the value to the open-source version of the CKafka instance
  # Set to the access address of the CKafka instance
  hosts: ["xx.xx.xx.xx:xxxx"]
  # Set the name of the target topic
  topic: 'test'
  partition.round_robin:
    reachable_only: false

  required_acks: 1
  compression: none
  max_message_bytes: 1000000

  # The following parameters need to be configured for SASL. If SASL is not require
  username: "yourinstance#yourusername"  // You need to concatenate the instance ID
  password: "yourpassword"
```

Step 4. Use Filebeat to send a message

1. Run the following command to start the client:

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

2. Add data to the monitoring file (for example: `testlog`).

```
echo ckafka1 >> testlog
echo ckafka2 >> testlog
echo ckafka3 >> testlog
```

3. Start the consumer to consume the corresponding topic and obtain the following data.

```
{"@timestamp":"2017-09-29T10:01:27.936Z","beat":{"hostname":"10.193.9.26","name":"1
{"@timestamp":"2017-09-29T10:01:30.936Z","beat":{"hostname":"10.193.9.26","name":"1
{"@timestamp":"2017-09-29T10:01:33.937Z","beat":{"hostname":"10.193.9.26","name":"1
```

SASL/PLAINTEXT mode

If you want to configure SASL/PLAINTEXT, you need to set the username and password under the Kafka configuration.

```
  # The following parameters need to be configured for SASL. If SASL is not required
```

```
    username: "yourinstance#yourusername"  // You need to concatenate the instance ID
    password: "yourpassword"
```

FAQs

The Filebeat log file (default path: `/var/log/filebeat/filebeat`) contains a large number of INFO logs as follows:

```
2019-03-20T08:55:02.198+0800    INFO    kafka/log.go:53 producer/broker/544 startin
2019-03-20T08:55:02.198+0800    INFO    kafka/log.go:53 producer/broker/544 state c
2019-03-20T08:55:02.198+0800    INFO    kafka/log.go:53 producer/leader/wp-news-fil
2019-03-20T08:55:02.198+0800    INFO    kafka/log.go:53 producer/broker/478 state c
2019-03-20T08:55:02.199+0800    INFO    kafka/log.go:53 Closed connection to broker
2019-03-20T08:55:02.199+0800    INFO    kafka/log.go:53 producer/leader/wp-news-fil
2019-03-20T08:55:02.199+0800    INFO    kafka/log.go:53 producer/leader/wp-news-fil
2019-03-20T08:55:02.199+0800    INFO    kafka/log.go:53 producer/leader/wp-news-fil
2019-03-20T08:55:02.199+0800    INFO    kafka/log.go:53 producer/leader/wp-news-fil
2019-03-20T08:55:02.199+0800    INFO    kafka/log.go:53 producer/leader/wp-news-fil
2019-03-20T08:55:02.199+0800    INFO    kafka/log.go:53 producer/leader/wp-news-fil
2019-03-20T08:55:02.199+0800    INFO    kafka/log.go:53 producer/broker/478 shut do
```

This problem may be related to the Filebeat version. Products in the Elastic family are updated frequently, and major version incompatibility problems often occur.

For example, v6.5.x supports Kafka v0.9, v0.10, v1.1.0, and v2.0.0 by default, while v5.6.x supports Kafka v0.8.2.0 by default.

Check the version configuration in the configuration file:

```
output.kafka:
  version:0.10.2 // Set the value to the open-source version of the CKafka instance
```

Note

When data is sent to CKafka, `compression.codec` cannot be set.

Gzip compression is not supported by default. To use it, submit a ticket.

As Gzip compression causes high CPU consumption, if it is used, all messages will become `InValid`.

The program cannot run properly when the LZ4 compression is used. Possible causes include:

The message format is incorrect. The default message version of CKafka is v0.10.2. You need to use the message format v1.

The setting method for SDK varies by Kafka client. You can query the setting method in the open-source community (such as the description for C/C++ Client) to set the version of the message format.

Multi-AZ Deployment

Last updated：2024-01-09 14:56:36

Multi-AZ Deployment of CKafka

CKafka Pro Edition supports multi-AZ deployment. When you purchase a CKafka instance in a region that has three or more AZs, you can select up to four of them for multi-AZ deployment. Partition replicas of this instance will be forcibly distributed across the nodes in the three AZs, which enables your instance to provide services uninterruptedly when one AZ becomes unavailable.

Note:

Only the Pro Edition supports multi-AZ deployment.

How multi-AZ deployment of CKafka works

The multi-AZ deployment feature of CKafka involves three layers: network, data, and control.

Network layer

CKafka exposes a VIP to the client. After the client is connected to the VIP, it will get the metadata information (which is generally addresses mapped one to one at different ports on the same VIP) of the topic partitions .

This VIP can fail over to another AZ at any time. When an AZ becomes unavailable, the VIP will automatically shift to another AZ in the same region, thus achieving multi-AZ disaster recovery.

Data layer

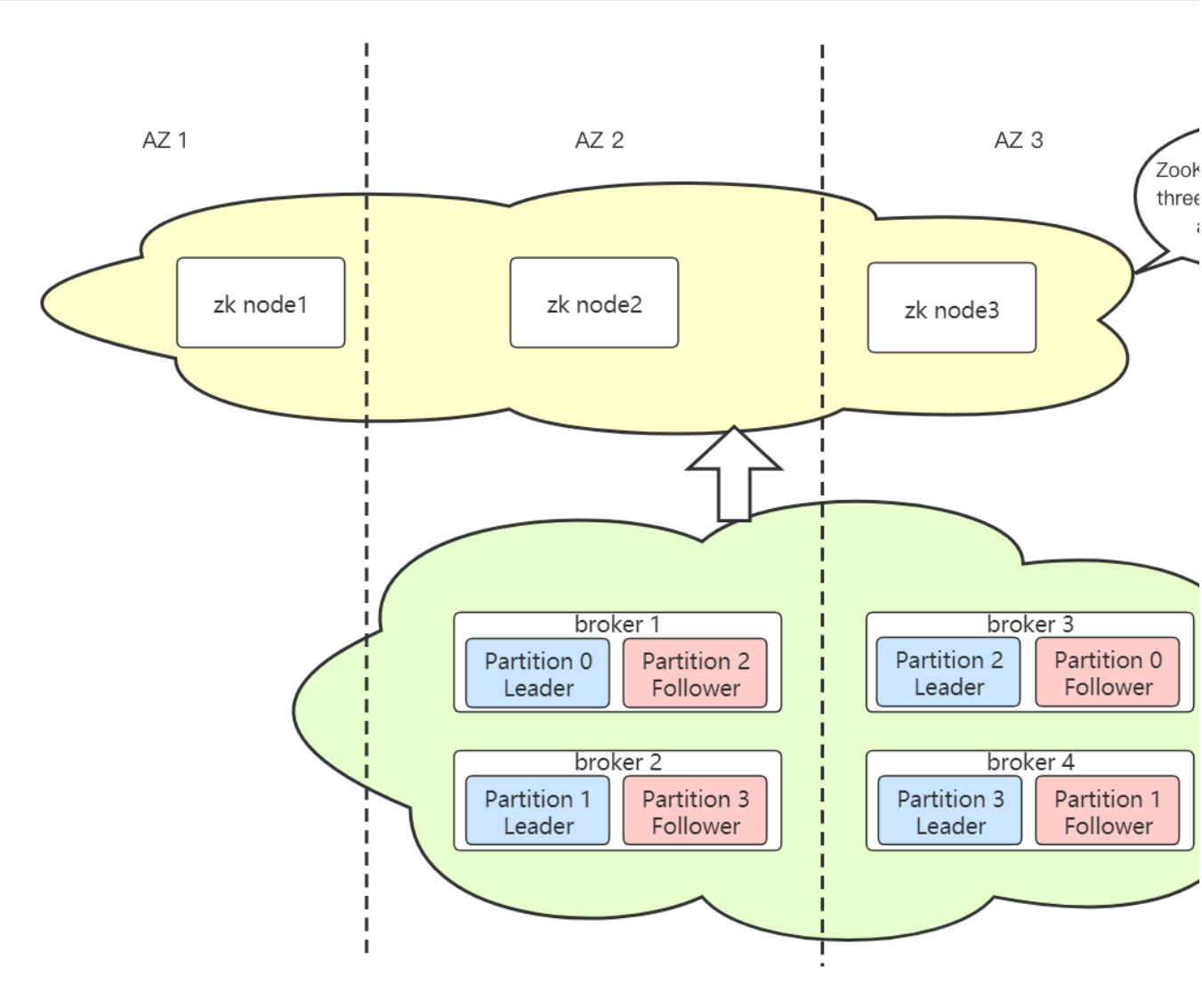
The data layer of CKafka is deployed in the same distributed way as Kafka; in other words, multiple data replicas are distributed on different broker nodes which are deployed in different AZs. When a partition is processed, there will be a leader-follower relationship between such nodes. When the leader goes offline due to an exception, the cluster controller will elect a new partition leader to handle the requests for the partition.

For the client, after an AZ becomes unavailable due to an exception, if the leader of a topic partition is located on a broker node in the AZ, the originally established link will time out or be closed. After an exception occurs on the leader node of the partition, the controller will elect a new leader node to provide services, and if the controller node is also abnormal, the remaining nodes will run for a new controller node. The leader switch can be completed within seconds (proportional to the number of nodes in the cluster and the size of the metadata). The client will periodically refresh the metadata of the topic partitions and link the new leader node for production and consumption.

Control layer

The control layer of CKafka applies the same technical scheme as Kafka. It relies on ZooKeeper to manage service discovery and cluster controller election for broker nodes. **The ZK nodes in the ZooKeeper cluster of a CKafka**

instance that supports multi-AZ deployment are deployed in three AZs (or data centers). In this way, even if the ZK nodes in any AZ fail and disconnect, the entire ZooKeeper cluster can still provide services.



Pros and Cons of Multi-AZ Deployment

Pros

Multi-AZ deployment can significantly improve the disaster recovery capability of the cluster. When a single AZ experiences force majeure events such as network instability and restart after power outage, the client can resume message production and consumption after waiting a short while to reconnect.

Cons

In multi-AZ deployment mode, as partition replicas are distributed in multiple AZs, message replication across AZs experiences an additional network latency compared with message replication in one single AZ, which directly affects the client write time for production (the client `ACK` parameter is greater than 1 or equal to -1 or all). Currently, the network latency across AZs in major regions such as Guangzhou, Shanghai, and Beijing typically ranges between 10 ms and 40 ms.

Multi-AZ Deployment Scenarios

Unavailability of one single AZ

After a single AZ becomes unavailable, as explained above, the client will disconnect, reconnect, and then provide services normally after reconnection.
As multi-AZ deployment is currently not supported for the management of API services, after a single AZ becomes unavailable, you may not be able to create topics, configure ACL policies, or view monitoring data in the console, but this will not affect the production and consumption of existing businesses.

Network isolation between two AZs

If network isolation occurs between two AZs (that is, they cannot communicate), a cluster "split-brain" event may occur; in other words, the nodes in both AZs provide services, but data written in one of the AZs will be treated as dirty data after the cluster is recovered.
Suppose network isolation occurs between the cluster controller node and a ZK node in the ZooKeeper cluster and other nodes, other nodes will run for a new controller (which can be successfully elected since most nodes in the ZooKeeper cluster communicate normally over the network), but the isolated controller still regards itself as the controller node. In this case, a "split-brain" event occurs in the cluster.
At this time, the client writes need to be considered on a case by case basis. For example, when the client's `ACK` is set to `-1` or `all` and the number of replicas is two, if the cluster contains three nodes, they will be distributed in a 2:1 ratio after the split-brain event occurs. In this case, an error will be reported when the original leader writes to the partitions in the AZ with one node, while writes in the other AZ will succeed. However, if the number of replicas is three and `ACK = -1` or `ACK = all` is configured, writes on neither AZ will succeed. Therefore, you need to take further actions based on the specific parameter configuration.
After the cluster network is back to normal, the client can resume production and consumption without performing any operations. As the server will normalize the data again, the data on one of the split nodes will be truncated directly, which, however, will not result in data loss in case of multi-replica multi-AZ data storage.

Directions

Selecting multiple AZs for instance

1. Log in to the CKafka console.
2. Click **Instance List** on the left sidebar and click **Create** to enter the purchase page.
3. On the instance purchase page, set the configuration information for purchase based on your actual needs.

Billing Mode: Monthly subscription

Specs Type: Select the Standard or Pro Edition based on your business needs.

Kafka Version: Select a Kafka version based on your business needs. For more information, see Suggestions for CKafka Version Selection.

Region: Select a region close to the resource for client deployment.

AZ: Select an AZ based on your actual needs.

Standard Edition: This edition does not support multi-AZ deployment.

Pro Edition: If the current region supports multi-AZ deployment, you can select up to three AZs for deployment. For more information on multi-AZ deployment, see Multi-AZ Deployment.

Product Specification: Select a model based on the peak bandwidth and disk capacity.

Message Retention Period: Select a value between 24 and 2,160 hours.

When the disk capacity is insufficient (that is, the disk utilization reaches 90%), old messages will be deleted in advance to ensure the service availability.

VPC: If you need to access other VPCs, you can modify the routing access rules as instructed in Adding Routing Policy.

Tag: It is optional. For more information, see Tag Overview.

Instance Name: When purchasing multiple instances, you can batch create instances by its numeric suffix (which is numbered in an ascending order) or its designated pattern string. For detailed directions, see Naming with Consecutive Numeric Suffixes or Designated Pattern String.

4. Click **Buy Now** to complete the instance creation process.

Log Access
Connecting CLS to CKafka

Last updated：2024-01-09 14:56:36

Overview

You can ship log topic data to CKafka for real-time stream computing and storage. If you haven't purchased a CKafka instance, you can consider using the Consumption over Kafka feature of CLS.

Prerequisites

You have activated the CKafka service.
Make sure that the current account has the permission to enable shipping to CKafka. If your account is a sub-account, it needs to be authorized by the root account first. For more information, see Examples of Custom Access Policies.

Directions

1. Create a CKafka instance in the same region as the log topic. For more information, see Creating Instance.
2. Configure the following parameters to create a topic in the same region as the log topic. For more information, see Creating Topic.
Preset ACL Policy: Disable this option.
Show advanced configuration:
CleanUp.policy: Select **delete**; otherwise, shipping will fail.
max.message.bytes: Set this value to 8 MB or above. Otherwise, when the size of a single message in CLS exceeds the specified limit, the message cannot be written to the CKafka topic, and shipping will fail.
3. Go to the CLS console and enter the shipping task management page or log topic management page as needed.
On the left sidebar, click **Shipping Task Management** and select a region, logset, and log topic.
On the left sidebar, click **Log Topic** and select a log topic to be shipped to CKafka to enter the log topic management page.
4. Click the **Ship to CKafka** tab to enter the configuration page.
5. Click **Edit** on the right to enable shipping to CKafka. Then, select the target CKafka instance and topic as well as the log field to be shipped.
6. Click **OK** to start shipping to CKafka. If the task status is **Enabled**, the feature is enabled successfully.
Note:

To cleanse the log data before shipping to CKafka, see Log Filtering and Distribution.

FAQs

What should I do if the log data cannot be shipped to CKafka?

If ACL authentication is enabled in CKafka, the log data cannot be shipped. In this case, you need to disable the ACL of the topic.

What should I do if the system prompts that I have no permissions to read/write the CKafka topic?

If you directly use an API to ship data to CKafka, you may not have the read/write permissions of the CKafka topic. If you ship data in the console, the system will guide you through the authorization process, but if you directly call an API for shipping, you need to authorize manually. For more information, see Viewing and Configuring Shipping Permissions.

Replacing Supportive Route (Old)

Last updated：2024-01-09 14:56:36

Background

To enjoy more stable and reliable services, we recommend you switch to the new supportive route.

Impact

Because it is necessary to update the bootstrap-server addresses of producers and consumers and restart the service, the switch will cause a momentary interruption in business production and consumption.

Directions

1. Create a supportive route.
2. Switch the CKafka bootstrap-server addresses of all producers and consumers to the newly created supportive route. The switch order does not matter as long as they are all switched.
3. Restart producers and consumers based on the business conditions. The restart order does not matter.
4. Observe whether the business is stable for at least 3 hours (recommended).
5. Delete the old supportive route.

Rollback

If an exception occurs during business verification, you need to roll back to the old supportive route in the following steps provided that it has not been deleted.
1. Change back to the old supportive route address on all producers and consumers.
2. Restart all producer and consumer services.
3. Verify the business.