ALIBABA CLOUD

Alibaba Cloud

Message Queue for Apache Kafka

Best practices

Document Version: 20210330

Alibaba Cloud

Legal disclaimer

Alibaba Cloud reminds you to carefully read and fully understand the terms and conditions of this legal disclaimer before you read or use this document. If you have read or used this document, it shall be deemed as your total acceptance of this legal disclaimer.

1. You shall download and obtain this document from the Alibaba Cloud website or other Alibaba Cloud-authorized channels, and use this document for your own legal business activities only. The content of this document is considered confidential information of Alibaba Cloud. You shall strictly abide by the confidentiality obligations. No part of this document shall be disclosed or provided to any third party for use without the prior written consent of Alibaba Cloud.

2. No part of this document shall be excerpted, translated, reproduced, transmitted, or disseminated by any organization, company or individual in any form or by any means without the prior written consent of Alibaba Cloud.

3. The content of this document may be changed because of product version upgrade, adjustment, or other reasons. Alibaba Cloud reserves the right to modify the content of this document without notice and an updated version of this document will be released through Alibaba Cloud-authorized channels from time to time. You should pay attention to the version changes of this document as they occur and download and obtain the most up-to-date version of this document from Alibaba Cloud-authorized channels.

4. This document serves only as a reference guide for your use of Alibaba Cloud products and services. Alibaba Cloud provides this document based on the "status quo", "being defective", and "existing functions" of its products and services. Alibaba Cloud makes every effort to provide relevant operational guidance based on existing technologies. However, Alibaba Cloud hereby makes a clear statement that it in no way guarantees the accuracy, integrity, applicability, and reliability of the content of this document, either explicitly or implicitly. Alibaba Cloud shall not take legal responsibility for any errors or lost profits incurred by any organization, company, or individual arising from download, use, or trust in this document. Alibaba Cloud shall not, under any circumstances, take responsibility for any indirect, consequential, punitive, contingent, special, or punitive damages, including lost profits arising from the use or trust in this document (even if Alibaba Cloud has been notified of the possibility of such a loss).

5. By law, all the contents in Alibaba Cloud documents, including but not limited to pictures, architecture design, page layout, and text description, are intellectual property of Alibaba Cloud and/or its affiliates. This intellectual property includes, but is not limited to, trademark rights, patent rights, copyrights, and trade secrets. No part of this document shall be used, modified, reproduced, publicly transmitted, changed, disseminated, distributed, or published without the prior written consent of Alibaba Cloud and/or its affiliates. The names owned by Alibaba Cloud shall not be used, published, or reproduced for marketing, advertising, promotion, or other purposes without the prior written consent of Alibaba Cloud. The names owned by Alibaba Cloud include, but are not limited to, "Alibaba Cloud", "Aliyun", "HiChina", and other brands of Alibaba Cloud and/or its affiliates, which appear separately or in combination, as well as the auxiliary signs and patterns of the preceding brands, or anything similar to the company names, trade names, trademarks, product or service names, domain names, patterns, logos, marks, signs, or special descriptions that third parties identify as Alibaba Cloud and/or its affiliates.

6. Please directly contact Alibaba Cloud for any errors of this document.

Document conventions

Style	Description	Example
Danger	A danger notice indicates a situation that will cause major system changes, faults, physical injuries, and other adverse results.	**Danger:** Resetting will result in the loss of user configuration data.
Warning	A warning notice indicates a situation that may cause major system changes, faults, physical injuries, and other adverse results.	**Warning:** Restarting will cause business interruption. About 10 minutes are required to restart an instance.
Notice	A caution notice indicates warning information, supplementary instructions, and other content that the user must understand.	**Notice:** If the weight is set to 0, the server no longer receives new requests.
Note	A note indicates supplemental instructions, best practices, tips, and other content.	**Note:** You can use Ctrl + A to select all files.
>	Closing angle brackets are used to indicate a multi-level menu cascade.	Click **Settings**> **Network**> **Set network type**.
Bold	Bold formatting is used for buttons , menus, page names, and other UI elements.	Click **OK**.
`Courier font`	Courier font is used for commands	Run the `cd /d C:/window` command to enter the Windows system folder.
Italic	Italic formatting is used for parameters and variables.	`bae log list --instanceid` *Instance_ID*
[] or [a|b]	This format is used for an optional value, where only one item can be selected.	`ipconfig [-all|-t]`
{} or {a|b}	This format is used for a required value, where only one item can be selected.	`switch {active|stand}`

Table of Contents

1.Best practices for producers

This article describes the best practices of Message Queue for Apache Kafka producers to help you reduce errors when you send messages. The best practices in this article are written based on a Java client. A Java client shares the basic concepts and ideas with other programming languages, but its implementation details may be different.

Message sending

Sample code for sending a message:

```
Future<RecordMetadata> metadataFuture = producer.send(new ProducerRecord<String, String>(
    topic,  // The topic of the message.
    null,  // The partition number. We recommend that you set this parameter to null, and then the producer automatically allocates a partition number.
    System.currentTimeMillis(),  // The timestamp.
    String.valueOf(value.hashCode()),  // The key of the message.
    value  // The value of the message.
));
```

For more information about the complete sample code, see Overview.

Key and Value fields

Message Queue for Apache Kafka version 0.10.2.2 has the following two message fields:

- Key: the identifier of a message.
- Value: the content of a message.

To facilitate tracing, set a unique key for each message. When you need to track the sending and consumption of a message, you can use a unique key to query the sending and consumption logs of the message.

If you want to send a large number of messages, we recommend that you implement the sticky partitioning strategy instead of setting a key. For more information about the sticky partitioning strategy, see the Sticky partitioning strategy section of this article.

> **Notice** Message Queue for Apache Kafka version 0.11.0 or later supports headers. If you need to use headers, upgrade your broker to version 2.2.0.

Retry

In a distributed environment, a message may fail to be sent due to network issues. This may occur after a message is sent but ACK failure occurs, or a message fails to be sent.

Message Queue for Apache Kafka uses a virtual IP address (VIP) network architecture where connections are closed after they are idle for more than 30 seconds. Therefore, inactive producers or consumers may receive the "Connection reset by peer" error message. In this case, we recommend that you resend the message.

You can set the following retry parameters based on your business needs:

- `retries`: the number of retries. We recommend that you set this parameter to `3`.
- `retry.backoff.ms`: the interval between retries. We recommend that you set this parameter to

`1000` .

Asynchronous transmission

Messages are sent in asynchronous mode. To obtain the sending result, you can call the metadataFuture.get(timeout, TimeUnit.MILLISECONDS) method.

Thread safety

Producers are thread-safe and they can send messages to all of the topics. In most cases, one application corresponds to one producer.

ACKs

ACKs have the following settings:

- `acks=0` : No response is returned from the broker. In this mode, the performance is high, but the risk of data loss is also high.
- `acks=1` : A response is returned when data is written to the leader. In this mode, the performance and the risk of data loss are moderate. Data loss may occur if the leader fails.
- `acks=all` : A response is returned when data is written to the leader and synchronized to the followers. In this mode, the performance is low, but the risk of data loss is also low. Data loss occurs if the leader and the followers fail at the same time.

We recommend that you set `acks=1` for regular services and set `acks=all` for key services.

Capability improvement for message sending

A Message Queue for Apache Kafka topic has multiple partitions. Before the Message Queue for Apache Kafka producer sends messages to the broker, the producer needs to select a partition of a topic to send messages to. To send multiple messages to the same partition, the producer packages relevant messages into a batch and sends the messages to the broker in batches. When the producer processes messages in batches, it incurs additional overheads. Small batches can result in a large number of requests that are generated by the producer. The requests queue on the producer and the broker and also lead to high CPU utilization. This prolongs the duration of message sending and increases the consumption latency. When the producer sends messages to the broker, a suitable size of each batch can reduce requests from the producer to the broker. This can also increase the throughput and lower the latency for message sending.

The Message Queue for Apache Kafka producer manages batches based on two parameters:

- `batch.size` : the volume of cached messages that are sent to each partition. This parameter specifies the total number of bytes in all of the messages in a batch, rather than the number of messages. When the volume of cached messages reaches the specified upper limit, a network request is triggered. Then, the producer sends the messages to the broker in a batch.
- `linger.ms` : the maximum storage duration for each message in the cache. If a message is stored longer than the specified time limit in the cache, the producer immediately sends the message to the broker without considering the setting of the `batch.size` parameter.

Therefore, the `batch.size` and `linger.ms` parameters work together to determine when the Message Queue for Apache Kafka producer sends messages in batches to the broker. You can set these two parameters based on your business needs.

Sticky partitioning strategy

Only messages to be sent to the same partition can be packaged into the same batch. The partitioning strategy of the Message Queue for Apache Kafka producer determines how to generate a batch. You can use the Partitioner class to select a suitable partition for the Message Queue for Apache Kafka producer based on your business needs. For messages that have a key, the default partitioning strategy of the Message Queue for Apache Kafka producer is to hash the key of each message, and then select a partition based on the hash result. Messages with the same key are sent to the same partition.

For messages that do not have a key, the default partitioning strategy of the Message Queue for Apache Kafka producer in versions earlier than 2.4 is to recycle all of the partitions of a topic, and then send messages to each partition by polling. However, this default partitioning strategy may cause higher latency because a large number of small batches may be generated. In view of the low efficiency of this default partitioning strategy for key-free messages, the sticky partitioning strategy is introduced in Message Queue for Apache Kafka version 2.4.

The sticky partitioning strategy can reduce the small batches, which are generated because the key-free messages scatter in different partitions. When a batch is full of messages, the producer randomly selects another partition and sends subsequent messages to this partition. In this strategy, messages are sent to the same partition in a short time, but messages can be evenly distributed in each partition when the producer works longer. This strategy can avoid partition skew of messages, and improve the overall performance with lower latency.

If you are using the Message Queue for Apache Kafka producer in version 2.4 or later, the producer uses the sticky partitioning strategy by default. If you are using the producer in a version earlier than 2.4, you can set the `partitioner.class` parameter to specify a partitioning strategy based on the principles of the sticky partitioning strategy.

To implement the sticky partitioning strategy, you can use the following Java sample code. The implementation logic of this code is to change partitions based on a specific interval.

```
public class MyStickyPartitioner implements Partitioner {
  // Record the time of the last partition change.
  private long lastPartitionChangeTimeMillis = 0L;
  // Record the current partition.
  private int currentPartition = -1;
  // The interval between partition changes. Set the interval based on your business needs.
  private long partitionChangeTimeGap = 100L;
  public void configure(Map<String, ? > configs) {}
  /**
   * Compute the partition for the given record.
   *
   * @param topic The topic name
   * @param key The key to partition on (or null if no key)
   * @param keyBytes serialized key to partition on (or null if no key)
   * @param value The value to partition on or null
   * @param valueBytes serialized value to partition on or null
   * @param cluster The current cluster metadata
   */
  public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster clust
er) {
    // Query the information about all partitions.
    List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
    int numPartitions = partitions.size();
    if (keyBytes == null) {
      List<PartitionInfo> availablePartitions = cluster.availablePartitionsForTopic(topic);
      int availablePartitionSize = availablePartitions.size();
      // Determine the available partitions.
      if (availablePartitionSize > 0) {
        handlePartitionChange(availablePartitionSize);
        return availablePartitions.get(currentPartition).partition();
      } else {
        handlePartitionChange(numPartitions);
        return currentPartition;
      }
    } else {
      // For messages that have a key, select a partition based on the hash value of the key.
      return Utils.toPositive(Utils.murmur2(keyBytes)) % numPartitions;
    }
  }
  private void handlePartitionChange(int partitionNum) {
    long currentTimeMillis = System.currentTimeMillis();
    // If the interval between partition changes is longer than the specified time, select another partition. If n
ot, select the same partition.
    if (currentTimeMillis - lastPartitionChangeTimeMillis >= partitionChangeTimeGap
      || currentPartition < 0 || currentPartition >= partitionNum) {
      lastPartitionChangeTimeMillis = currentTimeMillis;
      currentPartition = Utils.toPositive(ThreadLocalRandom.current().nextInt()) % partitionNum;
    }
  }
  public void close() {}
}
```

OOM

Based on the design of batches in Message Queue for Apache Kafka , Message Queue for Apache Kafka caches messages and then sends them in batches. However, if excessive messages are cached, an out of memory (OOM) error may occur.

- When the total size of all cached messages exceeds the cache size that is specified by the `buffer.memory` parameter, the producer sends these messages to the broker. In this case, the settings of the `batch.size` and `linger.ms` parameters are ignored.
- The default cache size that is specified by the `buffer.memory` parameter is 32 MB, which is sufficient for a single producer.

  > **Notice** If you enable multiple producers on the same Java virtual machine (JVM), an OOM error may occur because each producer may occupy 32 MB of the cache space.

- In most cases, you do not need to enable multiple producers during production. To avoid OOM errors in special scenarios, you must set the `buffer.memory` parameter.

Partitionally ordered messages

In each partition, messages are stored in the order that they are sent, and therefore are ordered.

By default, to improve the availability, Message Queue for Apache Kafka does not ensure the absolute order of messages in a single partition. A small number of messages become out of order during upgrade or downtime due to failovers. Messages in a failed partition will be moved to other partitions.

For Professional Edition instances that are billed in subscription mode, if your business requires messages to be strictly ordered in a partition, select local storage when you create a topic.

2.Best practices for subscribers

This topic describes the best practices of Message Queue for Apache Kafka subscribers to help you reduce the possibility of message consumption errors.

Basic process of message consumption

Message Queue for Apache Kafka subscribers use the following message consumption process:

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

Load balancing

Each consumer group can contain multiple consumer instances. Specifically, you can enable multiple Message Queue for Apache Kafka consumers and set the `group.id` parameter to the same value for the consumers. Consumer instances in the same consumer group consume the subscribed topics in load balancing mode.

For example, consumer group A has subscribed to topic A and enabled consumer instances C1, C2, and C3. In this case, each message sent to topic A will only be sent to one of C1, C2, and C3. By default, Message Queue for Apache Kafka evenly transfers messages to different consumer instances to balance the consumption loads.

To achieve load balancing in consumption, Message Queue for Apache Kafka evenly distributes the partitions of subscribed topics to the consumer instances. Therefore, the number of consumer instances cannot be greater than the number of partitions. Otherwise, some instances may not be assigned with any partitions and will be in the dry-run state. In addition, load balancing is triggered not only during first launch, but also when a consumer instance is restarted, increased, or decreased.

Each Message Queue for Apache Kafka topic contains 16 partitions by default, which is sufficient for most scenarios. In addition, the number of partitions will be adjusted for cloud services based on the capacity.

Multiple subscriptions

Message Queue for Apache Kafka supports the following modes:

- A consumer group subscribes to multiple topics.

  A consumer group can subscribe to multiple topics. Messages from multiple topics are evenly consumed by consumers in the consumer group. For example, consumer group A has subscribed to topic A, topic B, and topic C, so the messages from the three topics are evenly consumed by consumers in consumer group A.

  The following sample code for subscription to multiple topics by a consumer group is provided:

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

- Multiple consumer groups subscribe to a topic.

Multiple consumer groups can subscribe to the same topic, and each consumer group separately consumes all messages under the topic. For example, consumer groups A and B have both subscribed to topic A. Each message sent to topic A will be transferred to the consumer instances in both consumer groups A and B. The two processes are independent of each other without mutual effects.

One consumer group for a single application

We recommend that you configure one consumer group for one application. Specifically, different applications correspond to different pieces of code. If you need to write different pieces of code in the same application, you must prepare multiple different kafka.properties, such as kafka1.properties and kafka2.properties.

Consumer offset

Each topic contains multiple partitions, and each partition counts the total number of current messages, which is the maximum offset MaxOffset.

In Message Queue for Apache Kafka , a consumer sequentially consumes messages in the partition and records the number of consumed messages, which is ConsumerOffset.

Number of unconsumed messages (accumulated messages) = MaxOffset - ConsumerOffset

Consumer offset committing

Message Queue for Apache Kafka provides the following consumer offset committing parameters for consumers:

- enable.auto.commit: The default value is true.
- auto.commit.interval.ms: The default value is 1000, indicating 1 second.

After you set the two parameters, the system checks the last consumer offset committing time before each data polling. If the interval between this time and the current time exceeds the interval specified by the auto.commit.interval.ms parameter, the consumer commits a consumer offset.

Therefore, if the enable.auto.commit parameter is set to true, you must ensure that all the data polled last time has been consumed before each poll. Otherwise, unconsumed messages may be skipped.

To control offset committing, you must set the enable.auto.commit parameter to false and call the commit(offsets) function.

Consumer offset resetting

The consumer offset is reset in the following scenarios:

- No offset has been committed to the broker, for example, when the consumer is brought online for the first time.
- A message is pulled from an invalid offset. For example, the maximum offset in a partition is 10, but the consumer starts consumption from offset 11.

On the Java client, you can configure the following resetting policies by using the auto.offset.reset parameter.

- latest: Reset the consumer offset to the maximum offset.
- earliest: Reset the consumer offset to the minimum offset.
- none: Do not reset the consumer offset.

> **Note**
>
> - We recommend that you set this parameter to latest instead of earliest to prevent heavily repetitive consumption when consumption starts from the beginning due to an invalid offset.
> - If you manage the offset, you can set the parameter to none.

Large message pulling

During consumption, the consumer actively pulls messages from the broker. When the consumer pulls large messages, you need to control the pulling speed by modifying the following parameters:

- max.poll.records: If the size of a message exceeds 1 MB, we recommend that you set this parameter to 1.
- fetch.max.bytes: Set this parameter to a value that is slightly larger than the size of a single message.
- max.partition.fetch.bytes: Set this parameter to a value that is slightly larger than the size of a single message.

Large messages are pulled one by one.

Message duplication and consumption idempotence

In Message Queue for Apache Kafka , the semantics for consumption is consuming each message "at least once". Specifically, a message is delivered at least once to ensure that the message will not be lost. However, this does not ensure that messages are not duplicated. When a network error occurs or the client restarts, a small number of messages may be duplicated. In this case, if the application consumer is sensitive to message duplication (for example, order transactions), the messages must be idempotent.

The following common practices are for database applications:

- When you send a message, pass in a key as a unique sequence ID.
- When you consume a message, check whether the key has been consumed. If yes, skip the message. If no, consume the message once.

Certainly, if the application is not sensitive to duplication of a few messages, the idempotence check is not required.

Consumption failure

Message Queue for Apache Kafka messages are consumed one by one in a partition. If the consumer fails to execute the consumption logic after it receives a message, for example, a message fails to be processed due to dirty data on the application server, you can use the following methods to handle this issue:

- Make the system keep trying to execute the consumption logic upon failure. This method may block the consumption thread at the current message, resulting in message accumulation.
- Message Queue for Apache Kafka is not designed to process failed messages. Therefore, you can print failed messages or store them to a service. For example, you can create a topic that is dedicated to store failed messages. Then, you can check the failed messages regularly, analyze the causes, and take appropriate measures.

Consumption latency

In Message Queue for Apache Kafka , the consumer automatically pulls messages from the broker to consume. Therefore, if the consumer can consume the data promptly, the latency is low. If the latency is high, first check whether any messages are accumulated, and then increase the consumption speed.

Consumption blocking and accumulation

The most common issue on a consumer is consumption accumulation. The most common causes for accumulation are as follows:

- Consumption is slower than production. In this case, you need to increase the consumption speed. For more information, see Consumption speed increase.
- The consumer is blocked.

After receiving a message, the consumer executes the consumption logic and usually makes some remote calls. If the consumer waits for the call result at this time, the consumer may keep waiting, causing the consumption process to suspend.

The consumer needs to try to prevent consumption thread blocking. If the consumer waits for the call result, we recommend that you set a timeout period for waiting, so that the consumption is considered failed if no result is returned within the set timeout period.

Consumption speed increase

You can increase the consumption speed in either of the following ways:

- Add consumer instances.

  You can add consumer instances in a process and ensure that each instance corresponds to one thread. Alternatively, you can deploy multiple consumer instance processes. When the number of instances exceeds the number of partitions, the speed cannot be increased and some consumer instances become idle.

- Add consumption threads.

  Adding a consumer instance is essentially the same as adding a consumption thread to increase the speed. Therefore, to improve the performance, it is more important to add a consumption thread. You can perform the following basic steps:

  i. Define a thread pool.
  ii. Poll data.
  iii. Submit data to the thread pool for concurrent processing.
  iv. Poll data again after the concurrent processing result is returned.

Message filtering

Message Queue for Apache Kafka does not provide any semantics for filtering messages. You can use either of the following methods to filter messages:

- If a few types of messages need to be filtered, you can use multiple topics to filter them.
- If many types of messages need to be filtered, we recommend that you filter the messages by services on the client.

You can select either of the methods as required or integrate both methods.

Message broadcasting

Message Queue for Apache Kafka does not provide semantics for broadcasting a message. You can simulate message broadcasting by creating different consumer groups.

Subscription relationship

To facilitate troubleshooting, we recommend that consumer instances in the same consumer group subscribe to the same topics.