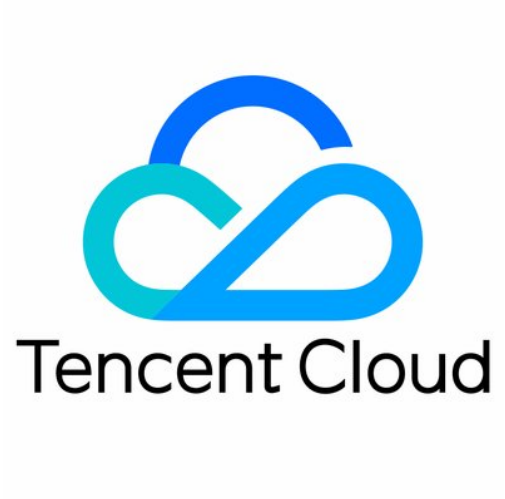
Cloud Message Queue

Getting Started

Product Documentation



Contents

Getting Started
Quick Start

Last updated：2021-06-01 14:11:52

Tencent Cloud Message Queue (CMQ) is a distributed message queue service that provides a reliable message-based async communication mechanism among different applications deployed in a distributed manner (or different components of the same application). Messages are stored in highly reliable and available CMQ queues, and multiple processes can be read/written at the same time without affecting one another.
CMQ provides four SDKs. This document uses the SDK for Python as an example.

SDK for Python Overview

To facilitate your use, CMQ operations for objects such as users, queues, and topics are divided into the following classes:

- Account: encapsulates account `SecretId` and `SecretKey` so that you can create, delete, and view queues, topics, and subscriptions.
- queue: receives/sends messages and views/sets queue attributes.
- topic: publishes messages, views and sets topic attributes, and views subscribers.
- cmq_client: sets the attributes of the connection between client and server, such as setting whether log write is enabled, connection timeout period, and whether persistent connection is enabled

> All classes are not thread-safe. If you need to use them in multiple threads, it is recommended that each thread instantiate its own objects.

Download SDK >>

Queue Model

A queue in CMQ is different from that defined in a data structure. Queues in data structures are manipulated in strict compliance with the FIFO method, while CMQ distributed queues do not strictly follow FIFO (a dedicated FIFO product will be released in the future). A CMQ queue is considered as a container featuring high performance, capacity, and reliability, to which produced messages can be delivered, and from which messages can be fetched out for consumption. It has its own attribute settings during initialization as detailed below:

Attribute	Description
maxMsgHeapNum	Maximum number of retained messages, i.e., number of messages that can be stored in a queue. It represents the queue storage and retention capacity.
pollingWaitSeconds	Long-polling waiting time for message receipt, which ranges from 0 to 30 seconds. It indicates the default time it takes to receive a message during message consumption. For example, when this attribute is set to 10, if there are no messages during message consumption, the result will be returned after 10 seconds of waiting by default; otherwise, the result will be immediately returned. You can customize the waiting time when receiving messages, which will take precedence over this attribute.
visibilityTimeout	Message visibility timeout period. After a message is obtained by a consumer, there will be an invisibility period, during which other consumers cannot get this message. This value ranges from 1 to 43,200 seconds (12 hours), and the default value is 30.
maxMsgSize	Maximum message size, which ranges from 1,024 to 1,048,576 bytes (i.e., 1–1,024 KB). The default value is 65,536.
msgRetentionSeconds	Message lifecycle, i.e., message retention time period in queue, which ranges from 60 to 1,296,000 seconds (1 minute to 15 days). The default value is 345,600 (4 days).
createTime	Queue creation time. A Unix timestamp accurate down to the second will be returned.
lastModifyTime	Time when the queue attribute is last modified. A Unix timestamp accurate down to the second will be returned.
activeMsgNum	Total number of messages in `Active` state (not being consumed) in queue, which is an approximate value.
inactiveMsgNum	Total number of messages in `Inactive` state (being consumed) in queue, which is an approximate value.
rewindSeconds	Maximum time range during which a message can be rewound in the queue, which ranges from 0 to 43,200 seconds. 0 indicates that message rewind is disabled.
rewindmsgNum	Number of retained messages which have been deleted by the `DelMsg` API but are still within their rewind time range.
minMsgTime	Minimum unconsumed time of message in seconds.
delayMsgNum	Number of delayed messages.

Getting Started with Queue Model >>

Topic Model

The topic model is similar to the publish/subscribe design pattern. A topic is the unit for sending messages, and subscribers under a topic are equivalent to observers. A topic will actively push published messages to subscribers.

Attribute	Description
msgCount	Number of messages currently retained in topic (number of retained messages)
maxMsgSize	Maximum message size, which ranges from 1,024 to 1,048,576 bytes (i.e., 1–1,024 KB). The default value is 65,536.
msgRetentionSeconds	Maximum lifecycle of message in topic. After the period specified by this parameter has elapsed since a message is sent to the topic, the message will be deleted no matter whether it has been successfully pushed to the user. This parameter is measured in seconds and defaulted to one day (86,400 seconds), which cannot be modified.
createTime	Topic creation time. A Unix timestamp accurate down to the second will be returned.
lastModifyTime	Time when the topic attribute is last modified. A Unix timestamp accurate down to the second will be returned.
filterType	Filtering policy selected when a subscription is created: If `filterType` is 1, `filterTag` will be used for filtering. If `filterType` is 2, `bindingKey` will be used for filtering.

Getting Started with Topic Model >>

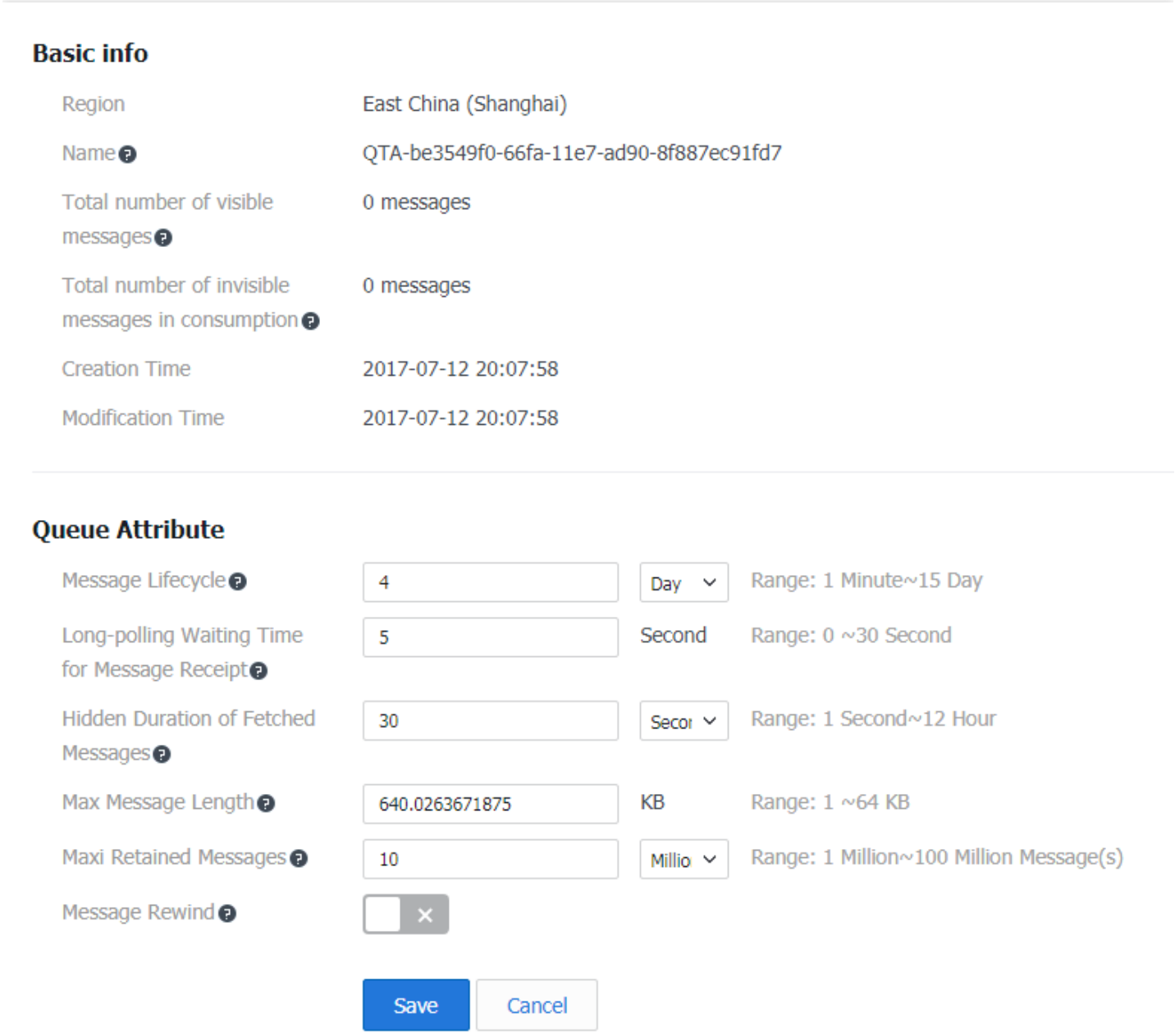
Getting Started with Queue Model

Last updated：2021-06-01 14:26:45

1. Creating a Queue

```
endpoint='' //Domain name of CMQ
secretId ='' // User's ID and Key
secretKey = ''
account = Account(endpoint,secretId,secretKey)
queueName = 'QueueForTest'
queue=account.get_queue(queueName)
queue_meta = QueueMeta()
queue_meta.queueName = queueName
queue_meta.visibilityTimeout = 10
queue_meta.maxMsgSize = 65536
queue_meta.pollingWaitSeconds = 10
try:
queue.create(queue_meta)
except CMQExceptionBase,e:
print e
```

After the queue is created, you can view the created queue information from the console.



2. Generating a Message

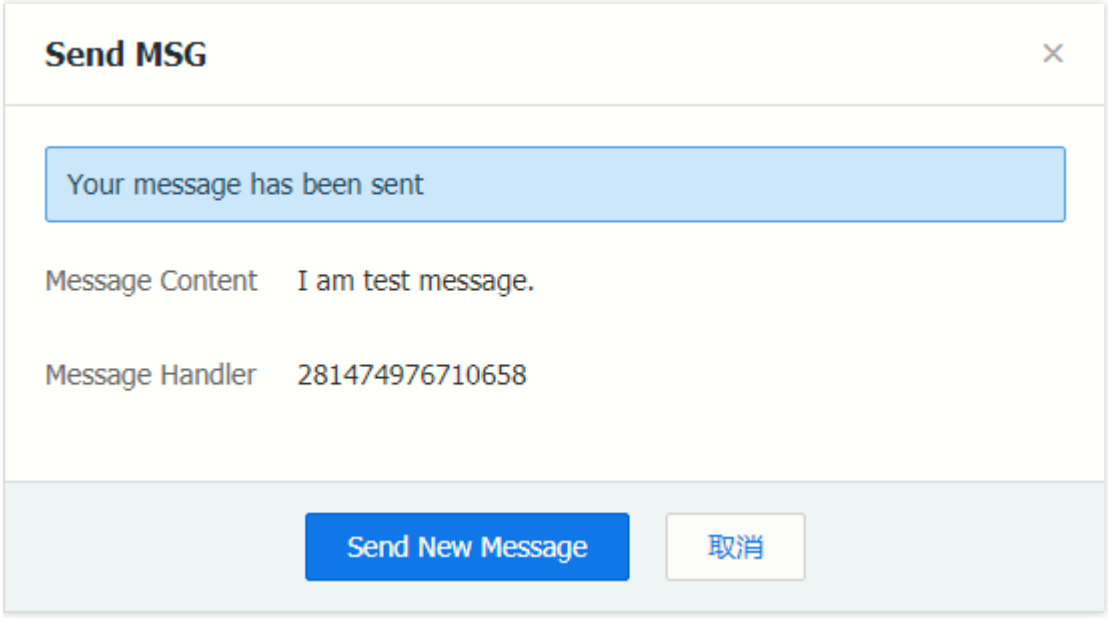
After obtaining the queue object, you can call Send Message API of the queue to send messages to the queue. You can perform the API by sending a message or sending messages in batch.

- **Generating a message:**

```
msg_body = "I am test message."
```

```
msg = Message(msg_body)
re_msg = my_queue.send_message(msg)
```

You can view the message attributes directly from the console.



- **Generating messages in batch:**

```
msg_count=3
messages=[]
for i in range(msg_count):
msg_body = "I am test message %s." % i
msg = Message(msg_body)
messages.append(msg)
re_msg_list = my_queue.batch_send_message(messages)
```

3. Consuming a Message

The default parameter pollingWaitSeconds indicates the desired waiting time when consuming messages. If left empty, it will use the attribute value in the queue.

- **Consuming a message:**

```
wait_seconds=3
recv_msg = my_queue.receive_message(wait_seconds)
```

- **Consuming messages in batch:**

```
wait_seconds = 3
num_of_msg = 3
recv_msg_list = my_queue.batch_receive_message(num_of_msg, wait_seconds)
```

Note

- **Set the appropriate pollingWaitSeconds**
  You can either customize the value of pollingWaitSeconds or use the default value of the queue. If the value is set to 0, it will not wait for messages. But if so, no messages may be returned (even if there is a message in the queue). That's because a large number of consumers may queue up for the queuing service when consuming messages at the same time. If you set the value to 0, you may receive the exception of no message since your request has timed out before your turn. Therefore, you are not recommended to set the waiting time to 0.

- **If number of messages in the queue < number of messages consumed in batch, your consumption will not be blocked.**
  When consuming messages in batch, you need to fill in the number of messages to be received this time. If the number of messages in the queue is less than the number of messages to be consumed, your operation will not be blocked.

- **In the queue attributes, by setting invisibility time > message retention period, you can consume each message once.**
  When the invisibility time > message retention period, the message consumed will become invisible and removed from the queue after the timeout of the retention period. In this way, the message is only consumed once and will not be consumed again.
  However, there may be duplicate generation and failed consumption in the process of generation and consumption. It is impossible to ensure that the queue is only consumed once by modifying the queue attributes. The service end need to involve in duplicate removal and fault tolerance for message consumption. Please see Duplicate Message Removal

4. Message Rewind

Let's see the use of the message rewind in the following scenario:

Assuming that there are A/B services in normal generation and consumption scenarios, A generates messages and delivers them to the queue and B consumes messages from the queue. In this case, A and B work independently without interfering with each other. A only generates messages for delivery while B acquires and deletes messages from the queue and then consumes messages locally.

For example, although B service has consumed messages, an exception has occurred with the consumption in a period of time. At this time, the deleted messages cannot be re-consumed, thus affecting the service. In this case, B service will be suspended and can only be resumed after developers or O&M personnel repair the bug. But O&M personnel cannot provide real-time monitoring for B service. It may take a while before the exception is detected.

To prevent this situation, A service needs to interfere in the processing of B service, back up generated messages and delete such backup data until B service is running properly, so as to ensure the normal operation of existing networks.

In this case, you can use **message rewind** function. The developer will repair B service and rewind the message to the latest point in time with normal consumption. Then, B service will acquire messages from such point in time. Thus, A service don't need to interfere in the exception of B service. Please note B need to perform idempotent operations for the consumption.

Learn more about Message Rewind >>

Enabling Message Rewind

```
endpoint='' //Domain name of CMQ
secretId ='' // User's ID and Key
secretKey = ''
account = Account(endpoint,secretId,secretKey)
queueName = 'QueueTest'
my_queue = account.get_queue(queueName)
queue_meta = QueueMeta()
queue_meta.rewindSeconds = 43200 //Time allowed for message rewind (in seconds)
my_queue.create(queue_meta)
```

Using Message Rewind

```
my_queue.rewindQueue(1488718862) //Point in time for this message rewind (Unix timestamp)
```

5. Delayed Messages

Delayed messages: When generating messages, you can specify a flight time, that is, the time spend in delivering messages to the queue. The message can only be consumed by consumers after such time.

Some services may fail, and then they need to re-consume messages after a certain period of time. In this case, you can use delayed messages.

For example:

```
message_body='i am test'
msg = Message(message_body)
my_queue.send_message(msg)
//Message consumption is found failed. You can re-deliver the message and set the message's flight time.
my_queue.send_message(msg,600) //The flight time is set to 10 minutes.
//You can view the number of delayed messages in the queue via message attributes
queue_meta = my_queue.get_attributes()
print queue_meta.delayMsgNum
```

Getting Started with Topic Model

Last updated：2021-06-01 14:27:23

A topic can publish messages only if it is subscribed to by at least one subscriber. If there are no subscribers, messages in the topic will not be delivered, and message publishing will be meaningless.

1. Create a Topic

```
endpoint='' // CMQ domain name
secretId ='' // User ID and key
secretKey = ''
account = Account(endpoint,secretId,secretKey)
topicName = 'TopicTest8B'
my_topic = account.get_topic(topicName)
topic_meta = TopicMeta()
my_topic.create(topic_meta)
```

You can view the created topic in the console. Here, QPS is 5000, indicating that the default highest frequency for calling the same API is 5,000 calls per second. To increase the upper limit, please submit a ticket for application.

2. Publish a Message

You can publish a message through the SDK or in the console.

Through SDK

```
message = Message()
message.msgBody = "this is a test message"
my_topic.publish_message(message)
```

In Console

Topics support message filtering. A tag, i.e., message tag or message type, is used to identify a message category under a topic in CMQ. A consumer can filter messages by tag so that it can consume only message types of interest to it. This feature is disabled by default. If it is disabled, all messages will be sent to all subscribers. If a tag is added, subscribers can receive only messages with the set tag. A message filter tag describes the tag used for message

filtering in the subscription (only messages with the same tag can be pushed). One tag can contain up to 16 characters, and up to 5 tags can be added to one message.
Topics currently support filtering by tag and `routingKey`.

Publish messages in batches:

```
vmsg = []
for i in range(6):
message = Message()
message.msgBody = "this is a test message"
vmsg.append(message)
my_topic.batch_publish_message(vmsg)
```

3. Process the Message

After a message is published by a topic, it will automatically be pushed to the subscription. If the push fails, there are two retry policies:

- **Backoff retry**: an attempt will be retried three times at random intervals between 10 and 20 seconds. After three retries, the message will be discarded for the subscriber and will not be retried again.
- **Exponential decay retry**: an attempt will be retried 176 times at exponentially increasing intervals: 2^0 seconds, 2^1 seconds, ..., 512 seconds, 512 seconds, ..., 512 seconds. The total retry duration is 1 day. This is the default retry policy.

Using queue to process messages

A subscriber can enter a queue so as to use it to receive published messages.

```
subscription_name = "subsc-test"
my_sub = my_account.get_subscription(topic_name, subscription_name)
subscription_meta = SubscriptionMeta()
# Enter the subscription name, which is a queue name here
subscription_meta.Endpoint = "queue name "
subscription_meta.Protocal = "queue"
my_sub.create(subscription_meta)
```

Using other means to process messages

Subscribers can process messages on their own without using queues. For more information, please see Delivering Messages.

4. Use Routing Matching

The binding key and routing key are used together and are compatible with the topic match mode of RabbitMQ. The routing key carried when a message is sent is added by the client and must be a string without a wildcard, and the binding key carried when a subscription relationship is created is the binding relationship between the topic and the subscriber.

Use limits:

- There can be up to 5 binding keys, and each of them can contain up to 64 bytes to represent the route for message sending, which can have up to 15 `.` , i.e., up to 16 phrases.
- All routing keys are contained in a string, and each of them can contain up to 64 bytes to represent the route for message sending, which can have up to 15 `.` , i.e., up to 16 phrases.

Wildcard description:

- * (asterisk): represents a word (a letter string).

. (hashtag) matches one or multiple characters.

- Special rule of RabbitMQ: when `routing_key` is an empty string, it cannot match * but can match #.

Example:

- If the subscriber is **1.*.0**, and the message is **1.any character.0**, then it can be received by the subscriber.
- If the subscriber is **1.#.0**, and the message is **1.2.3.4.4.2.2.0**, then it can be received by the subscriber (the elements in the middle of the message can be arbitrary).

Using routing matching feature

```
endpoint='' // CMQ domain name
secretId ='' // User ID and key
secretKey = ''
account = Account(endpoint,secretId,secretKey)
topicName = 'TopicTest'
my_topic = account.get_topic(topicName)
topic_meta = TopicMeta()
topic_meta.filterType = =2 // It indicates that routing matching will be used whe
n messages are delivered to subscriptions
// If `filterType` is 1, tags are used for filtering
```

```
my_topic.create(topic_meta)
subscription_name = "subsc-test"
my_sub = my_account.get_subscription(topic_name, subscription_name)
subscription_meta = SubscriptionMeta()
// Enter the subscription name, which is a queue name here
subscription_meta.Endpoint = "queue name "
subscription_meta.Protocal = "queue"
subscription_meta.bindingKey=[1.*.0] // If the message tag is `[1.any characters.
0]`, all subscribers will
// receive messages carrying this tag
my_sub.create(subscription_meta)
```

Publishing message

```
message = Message()
message.msgBody = "this is a test message"
routingKey = '1.test.0' // This message will be delivered to the address subscrib
ed to by `my_sub`
my_topic.publish_message(message)
```