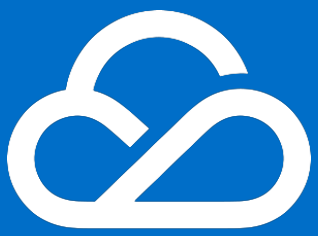
Cloud Message Queue

Getting Started

Product Introduction



Tencent
Cloud

Copyright Notice

Trademark Notice

Tencent Cloud

All trademarks associated with Tencent Cloud and its services are owned by Tencent Cloud Computing (Beijing) Company Limited and its affiliated companies. Trademarks of third parties referred to in this document are owned by their respective proprietors.

Service Statement

This document is intended to provide users with general information about Tencent Cloud's products and services only and does not form part of Tencent Cloud's terms and conditions. Tencent Cloud's products or services are subject to change. Specific products and services and the standards applicable to them are exclusively provided for in Tencent Cloud's applicable terms and conditions.

Contents

Getting Started

Getting Started

Cloud Message Queue (CMQ) is a distributed message queuing service that provides a reliable message-based asynchronous communication between distributed applications or components of an application. Each message is stored in highly available and highly reliable queues. Multiple processes can read/write from/to a queue at the same time without interfering with each other.

CMQ provides four SDKs. The following is illustrated in the case of Python.

1. Introduction to Python SDK

For ease of use, CMQ classifies users' actions, queue operations, and topic operations into the following categories:

- Account: To encapsulate account secretId and secretKey. Users can create/delete queues, topics and subscriptions, and view these objects;
- queue: To send/receive messages and view queue setting attributes;
- topic: To publish messages and view topic setting attributes and subscribers;
- cmq_client: Users can set some attributes for connection from client to server, such as whether to enable log writing, connection timeout and persistent connection.

Please note that all the categories are non-thread-safe. If you want to use it for multi-threading, you'd better instantiate your object for each thread.

Click to download SDK>>

2. Queue Model

The queue here is different from the Queue defined in the data structure. The queue in data structure must follow FIFO rule, but the distributed queue here is not strictly controlled by FIFO. (Later, we will develop dedicated FIFO products.) The latter is a container featuring high performance, high capacity and high reliability, where you can post generated messages or acquire messages for consumption. The queue is initialized with default setting attributes.

Now, let's see these attributes and their descriptions:

Attribute	Description
maxMsgHeapNum	Maximum number of messages in the queue. The number of messages that can be stored in the queue, which indicates the storage and retention capabilities of the queue.
pollingWaitSeconds	Waiting time for messages to be received when using long-polling. Value range is 0 to 30 seconds. This time is set to specify the default waiting time for the message to be received when consuming messages. For example, when the value is set to 10, it will wait 10 seconds and return if there is no message to be consumed; otherwise, it will return the acquired message immediately. Note: You can also set the custom waiting time when the message is received to replace the default attribute value of the queue.
visibilityTimeout	Message visibility timeout. When the message is acquired by a consumer, there will be an invisibility period of time, during which other consumers cannot receive this message. Value range is 1-43200 seconds (within 12 hours). Default is 30.
maxMsgSize	Maximum message length. Value range is 1024-65536 Byte (1-64 K). Default is 65536.
MsgRetentionSeconds	The message retention period, that is, the message storage time in the queue. Value range is 60-1296000 seconds (1 minute-15 days). Default is 345600 (4 days).
createTime	Queue creation time. A Unix timestamp will be returned (accurate to second).
lastModifyTime	The time when the queue attributes were

Attribute	Description
	modified for the last time. A Unix timestamp will be returned (accurate to second).
activeMsgNum	Total number of messages in the queue whose status is Active (i.e. not Consumed). This is an approximate value.
inactiveMsgNum	Total number of messages in the queue whose status is Inactive (i.e. being consumed). This is an approximate value.
rewindSeconds	The maximum rewind time for messages in the queue. Value range is 0-43200 seconds. 0 means message rewind is disabled.
rewindmsgNum	Number of messages that has been deleted by calling the DelMsg API but are still within the rewind time.
minMsgTime	Minimum time for messages to be in the "not consumed" status (in seconds).
delayMsgNum	Number of delayed messages.

View Queue Model Quick Start >>

3. Topic Model

The topic model is similar to the Publish/Subscribe model in the design pattern. Topic is equivalent to the one who publishes the message and the subscriber of the topic is equivalent to the observer. Topic will send the published messages to the subscribers:

Attribute	Description
msgCount	Current number of messages in the topic (number of retained messages).
maxMsgSize	Maximum message length. Value range is 1024-65536 Byte (1-64 K). Default is 65536.
msgRetentionSeconds	The maximum available time of the message in the topic (in seconds). Whether or not the

Attribute	Description
	message has been retrieved after being pushed to the users, it will be deleted after the period of time specified in this parameter. This parameter value is always one day (86,400 seconds) and cannot be modified.
Topic creation time	A Unix timestamp will be returned (accurate to second).
lastModifyTime	The time when the topic attributes were last modified. A Unix timestamp will be returned (accurate to second).
filterType	Specify the filtering rules when a user creates a subscription: If filterType =1, filterTag is used for tag filtering; If filterType =2, bindingKey is used for filtering.

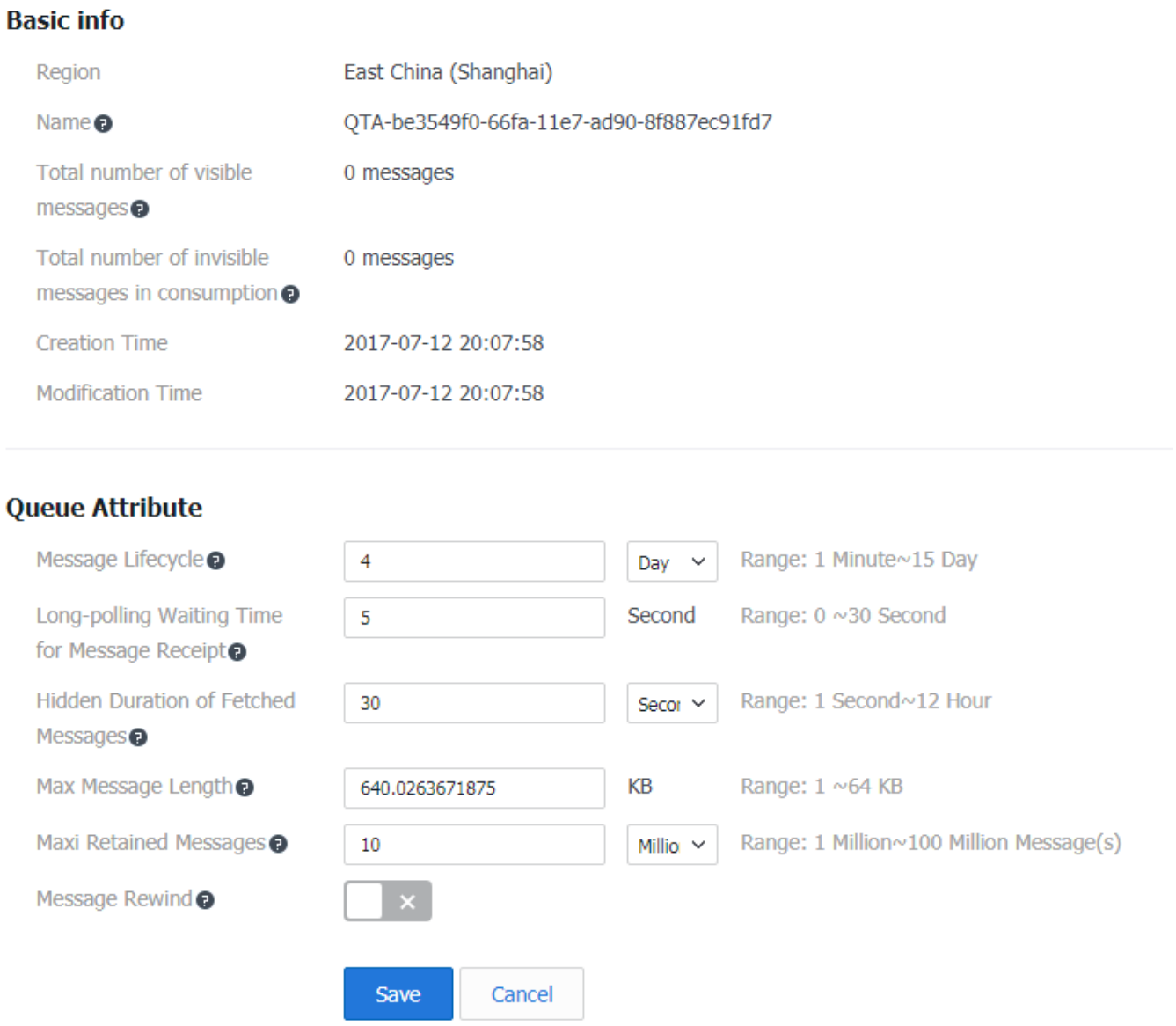
View Topic Model Quick Start >>

Queue Model

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

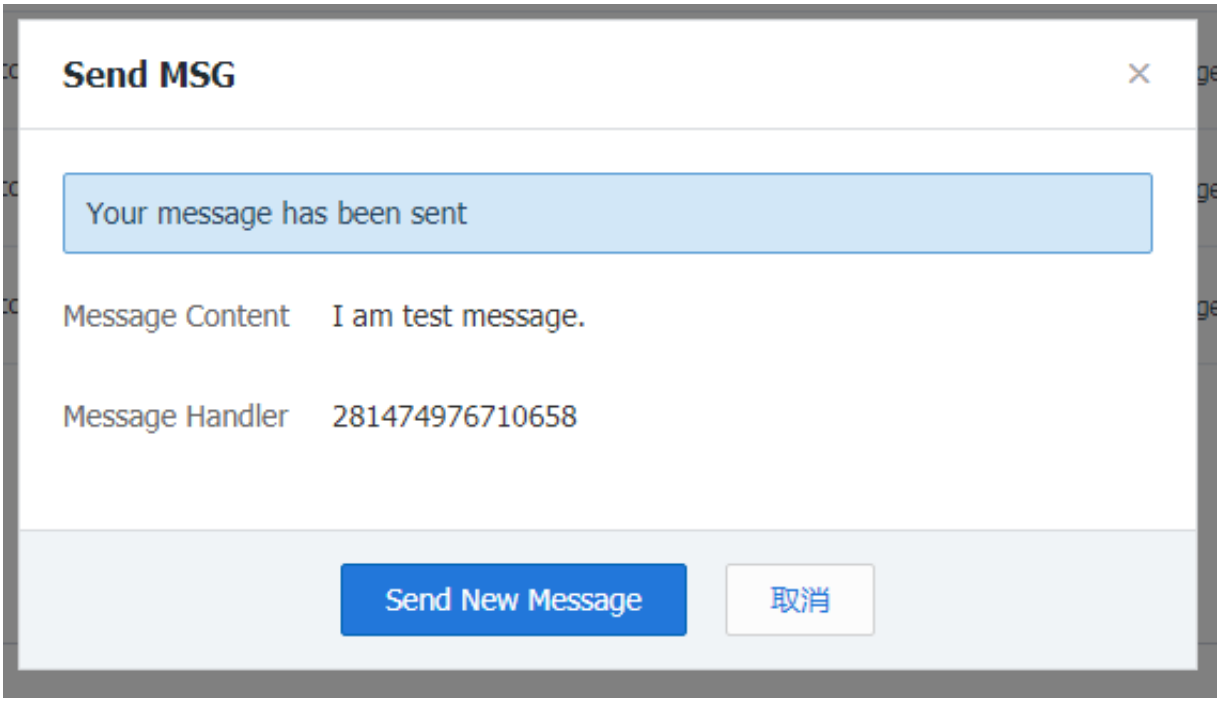
After obtaining the queue object, you can call Send Message API of the queue to send messages to the queue. You can perform the API by sending a message or sending messages in batch.

- 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
```

Topic Model

The prerequisite for publishing topic messages is that a subscriber must first subscribe a topic. If there is no subscriber, the message in the topic cannot be delivered. Thus, the publishing operation makes no sense.

1. Creating a Topic

```
endpoint='' //Domain name of CMQ
secretId ='' // User's 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 from the console. If QPS = 5,000, the maximum API calling frequency defaults to 5,000 counts/s. You can submit a ticket to apply for a higher quota if needed.

Topic Name	QTA-3e8926e1-66e1-11e7-84f0-8f887ec91fd7
Topic Name ID	topic-l66sgy83
Message Lifecycle	24 h
Max Message Length	64 KB
Message Retention	Enable
Retained Messages	0 messages
QPS	5,000
Message Filter Type	Tag
Creation Time	2017-07-14 10:55:09
Modification Time	2017-07-14 10:55:09

2. Publishing a Message

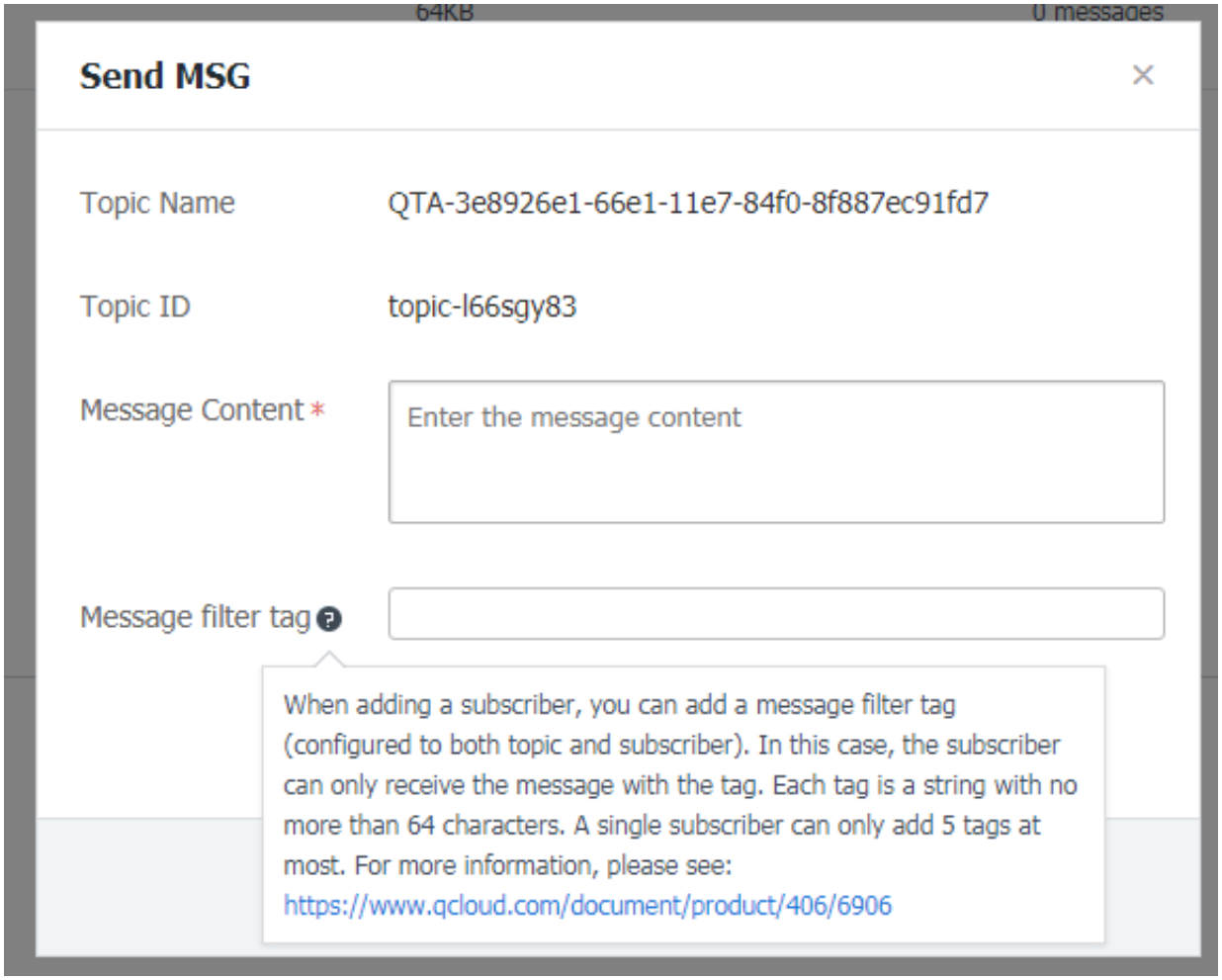
You can publish a message using SDK, as shown below.

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

You can also publish a message from the console, as shown below.

Topic currently supports message filtering, such as message tag, message type, to differentiate message categories under the Topic of a certain CMQ. CMQ allows consumers to filter messages based on the tags and thus only consume the messages that they're interested in. The function is

disabled by default. In this case, all messages are sent to all subscribers. But after a tag is added, subscribers can only receive messages with such tag. The message filter tag describes the tag used for message filtering for this subscription (only the messages with consistent tags will be pushed). Each tag is a string with no more than 16 characters. There can be at most 5 tags for a single Message.



Topic currently supports both tag filtering and routingKey filtering. Filtering rules are shown above.

Publishing messages in batch:

```
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. Message Processing

When a topic publishes a message, it will be automatically pushed to subscribers. If the push fails, two retry strategies are available:

- Backoff retry: Retry 3 times with a random interval between 10 and 20 seconds. After that, the message will be discarded for the subscriber, and will not try again;
- Exponential decay retry: Retry 176 times. The total retry time is 1 day with the interval: 2^0, 2^1, ..., 512, 512, ..., 512 seconds. Exponential decay retry is used by default.

Using a Queue to Process Messages

Subscribers can enter a Queue 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. Here is the queue name
subscription_meta.Endpoint = "queue name "
subscription_meta.Protocal = "queue"
my_sub.create(subscription_meta)
```

Other Means to Process Messages

Besides Queue, subscribers can also process messages by other means. For example, a Web code is provided to process messages pushed by Topic.

```
class MainHandler(tornado.web.RequestHandler):
def get(self):
self.write("Hello, world")
```

```
def post(self, *args, **kwargs):
self.write('hello, world')
def set_default_headers(self):
self.set_header('Content-type','application;charset=utf-8')
application = tornado.web.Application([(r"/",MainHandler),])
if __name__ == "__main__" :
application.listen(8889)
tornado.ioloop.IOLoop.instance().start()
```

Here, Web does not process the acquired messages, but only returns a "hellow world" string when receiving a message. CMQ will then use HTTP status codes to determine whether it is pushed successfully. If the returned status code is 2xx, the push succeeds.

Before using the Web, you need to create a subscription

```
subscription_name = "subsc-test"
my_sub = my_account.get_subscription(topic_name, subscription_name)
subscription_meta = SubscriptionMeta()
# Enter the subscription address. Here is your domain name or server IP
subscription_meta.Endpoint = "your endpoint "
subscription_meta.Protocal = "http"
my_sub.create(subscription_meta)
```

Then, Topic will push messages automatically to the corresponding endpoint.

4. Routing Match

Binding key and Routing key are used at the same time, compatible with rabbitmq topic matching mode. The Routing key when client sends messages must be a string without wildcards. The Binding key for subscription creation is used to bind the topic and the subscriber.

Service limits:

- The maximum number of binding keys is 5. This field indicates the routing path for sending messages. The length of a binding key should be <= 64 bytes and contain up to 15 ".", i.e. 16 phrases at most;
- Routing key contains one string. It indicates the routing path for sending messages. The length of a routing key should be <= 64 bytes and contain up to 15 ".", i.e. 16 phrases at most.

Wildcard description:

- * (Asterisk) can be a substitute for a word (a sequence of alphabetic string)

- # (Pound sign) can be used to match one or more characters

- Special case of rabbitmq: If routing_key is an empty string, * cannot be matched, but # can.

For example:

- Subscribers to "1.*.0" receive all messages for "1.any characters.0".
- Subscribers to "1.#.0" receive all messages for "1.2.3.4.4.2.2.0". (It can be any elements in between.)

Enabling Route Matching

```
endpoint='' //Domain name of CMQ
secretId ='' // User's ID and Key
secretKey = ''
account = Account(endpoint,secretId,secretKey)
topicName = 'TopicTest'
my_topic = account.get_topic(topicName)
topic_meta = TopicMeta()
topic_meta.filterType = =2 //Indicates the routing match used when the message is delivered to the subscriber.
//If filterType = 1, a tag is 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. Here is the queue name
subscription_meta.Endpoint = "queue name "
subscription_meta.Protocal = "queue"
subscription_meta.bindingKey=[1.*.0] //If the tag of the message is "1.any characters.0", all
subscribers can
//receive the message
my_sub.create(subscription_meta)
```

Publishing a Message

```
message = Message()
message.msgBody = "this is a test message"
routingKey = '1.test.0' //The message will be delivered to the subscription address in my_sub.
my_topic.publish_message(message)
```