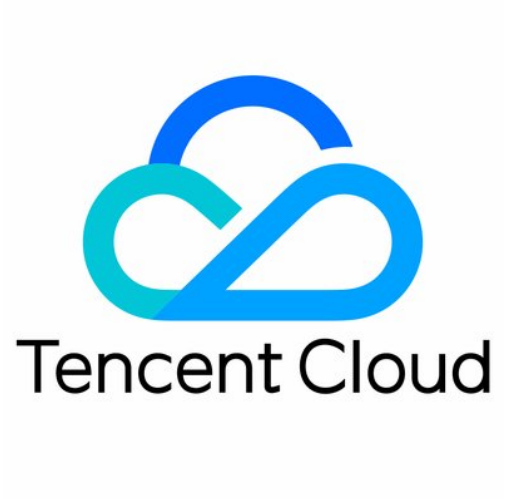
Serverless Cloud Function

Triggers

Product Documentation



Contents

Triggers
- Trigger Overview
- Trigger Event Message Structure Summary
- API Gateway Trigger
  - Overview
  - Websocket
    - How It Works
    - Usage
- COS Trigger
  - COS Trigger Description
  - Usage
- CLS Trigger
  - CLS Trigger Description
  - Usage
- Timer Trigger
  - Timer Trigger Description
  - Usage
- CKafka Trigger
  - CKafka Trigger Description
  - Usage
- Apache Kafka Trigger
  - Apache Kafka Trigger Description
  - Usage
- Trigger Configuration Description
- MPS Trigger
- CLB Trigger Description
- TencentCloud API Trigger

Triggers
Trigger Overview

Last updated：2024-12-02 19:58:17

SCF currently supports two triggering modes: **event-triggered** and **HTTP request-triggered**.

Event-Triggered

Event-Triggered is a typical serverless execution mode. Its core components are SCF functions and event sources, where an event source is a Tencent Cloud service or user-defined code that publishes an event, an SCF function is the handler of the event, and a function trigger is a set of correspondences between functions and event sources. For example, in the following scenarios:
Image/Video processing: the application crops the images uploaded by users into an appropriate size, stores the images in COS, creates thumbnails of each image, and displays them on the user page. In this scenario, you need to select COS as the event source and publish the event to the SCF function when the file is created. The event data provides all the information about the bucket and the file.
Data processing: the data collected during the day (such as clickstreams) is analyzed with a report generated at 00:00. In this scenario, you need to select a timer as the event source to publish an event to the SCF function at a specific time.
Custom application: the first image is invoked in your application to handle the SCF function as a module of the application. In this scenario, you need to call the `Invoke` API in the application to publish an event.
These event sources can be any of the following:
Internal event sources: these are preconfigured Tencent Cloud services that can be used with SCF. If you configure one of these event sources as a function trigger, the function will be invoked automatically when an event occurs. The relationship between the event source and the function (i.e., event source mapping) will be maintained on the event source side.
Custom applications: you can let custom applications publish events and invoke SCF functions.

Sample 1. COS publishes an event and invokes a function

You can configure the event source mapping for COS to determine what behaviors of COS will trigger an SCF function (such as object PUT or DELETE). The COS event source mapping is stored in COS and uses the bucket notification feature to direct COS to invoke the function when an event of a particular type occurs:
A COS trigger is created.
The user creates/deletes an object in the bucket.
COS detects an object creation/deletion event.

COS automatically invokes a function. It will be determined which function should be invoked based on the event source mapping stored in the COS configuration. The bucket and object information will be passed to the function as event data.

Sample 2. A timer publishes a time and invokes a function

The event source mapping of the timer is saved in the SCF function configuration to determine when the function should be automatically triggered:
A timer trigger is created.
The timer automatically invokes the function at the configured time.

Sample 3. A custom application invokes a function

If you need to invoke an SCF function in a custom application, you do not need to configure a function trigger or set up an event source mapping in this case; instead, you can use the `Invoke` API as the event source.
The custom application uses the `Invoke` API to invoke the function and pass in the event data.
The function receives the triggering request and is executed.
If sync invocation is used, the function will return the result to the application.
Note:
In this example, since the custom application and the function are produced by the same user, the user credentials (`APPID` , `SecretId` , and `SecretKey`) can be specified.

Notes

1. The current trigger-related restrictions for a single function can be viewed in Quota Limits.
2. There are specific restrictions on event source mappings due to the limitations of different Tencent Cloud services. For example, for a COS trigger, the same event (such as file upload) in the same COS bucket cannot trigger multiple different functions.

HTTP Request-Triggered

HTTP request-triggered is a special trigger method supported by SCF web functions. A native HTTP request can be directly passed through to the function environment through API Gateway to trigger the execution and processing of the function. It is suitable for the development of web service scenarios. For detailed usage, please see Function Overview.

Trigger Event Message Structure Summary

Last updated：2024-12-02 19:58:17

This document summarizes the message structures of all trigger events that are connected to SCF. For more information on trigger configuration and restrictions, see Trigger Management.

Note:

The event structure of the input parameter passed in by a trigger has been partially defined and can be used directly. You can get and use the Java library through the Java Cloud Event Definition or the Go library through the Go Cloud Event Definition.

Event Message Structure of Integration Request for API Gateway Trigger

When an API Gateway trigger receives a request, event data will be sent to the bound function in JSON format as shown below. For more information, see API Gateway Trigger.

```
{
  "requestContext": {
    "serviceId": "service-f94sy04v",
    "path": "/test/{path}",
    "httpMethod": "POST",
    "requestId": "c6af9ac6-****-****-9a41-93e8deadbeef",
    "identity": {
      "secretId": "abdcdxxxxxxxsdfs"
    },
    "sourceIp": "10.0.2.14",
    "stage": "release"
  },
  "headers": {
    "Accept-Language": "en-US,en,cn",
    "Accept": "text/html,application/xml,application/json",
    "Host": "service-3ei3tii4-251000691.ap-guangzhou.apigateway.myqloud.com",
    "User-Agent": "User Agent String"
  },
  "body": "{\\"test\\":\\"body\\"}",
  "pathParameters": {
    "path": "value"
  },
  "queryStringParameters": {
    "foo": "bar"
  },
```

```
  "headerParameters":{
    "Refer": "10.0.2.14"
  },
  "stageVariables": {
    "stage": "release"
  },
  "path": "/test/value",
  "queryString": {
    "foo" : "bar",
    "bob" : "alice"
  },
  "httpMethod": "POST"
}
```

Event Message Structure for Timer Trigger

When a function is invoked at a scheduled time, event data will be sent to the bound function in JSON format as shown below.

```
{
    "Type":"Timer",
    "TriggerName":"EveryDay",
    "Time":"2019-02-21T11:49:00Z",
    "Message":"user define msg body"
}
```

Event Message Structure for COS Trigger

When an object creation or deletion event occurs in the specified COS bucket, event data will be sent to the bound function in JSON format as shown below. For more information, see COS Trigger.

```
{
    "Records": [{
        "cos": {
            "cosSchemaVersion": "1.0",
            "cosObject": {
                "url": "http://testpic-1253970026.cos.ap-chengdu.myqcloud.com/testf
                "meta": {
                    "x-cos-request-id": "NWMxOWY4MGFfMjViMjU4NjRfMTUy********ZjM=",
                    "Content-Type": ""
                },
                "vid": "",
```

```
                "key": "/1253970026/testpic/testfile",
                "size": 1029
            },
            "cosBucket": {
                "region": "cd",
                "name": "testpic",
                "appid": "1253970026"
            },
            "cosNotificationId": "unkown"
        },
        "event": {
            "eventName": "cos:ObjectCreated:*",
            "eventVersion": "1.0",
            "eventTime": 1545205770,
            "eventSource": "qcs::cos",
            "requestParameters": {
                "requestSourceIP": "192.168.15.101",
                "requestHeaders": {
                    "Authorization": "q-sign-algorithm=******"
                }
            },
            "eventQueue": "qcs:0:lambda:cd:appid/1253970026:default.printevent.$LAT
            "reservedInfo": "",
            "reqid": 179398952
        }
    }]
}
```

Event Message Structure for CKafka Trigger

When a specified CKafka topic receives a message, the backend consumption module of SCF will consume the message and encapsulate it into an event in JSON format like the one below, which will trigger the bound function and pass the data content as input parameters to the function. For more information, see CKafka Trigger.

```
{
  "Records": [
    {
      "Ckafka": {
        "topic": "test-topic",
        "partition":1,
        "offset":36,
        "msgKey": "None",
        "msgBody": "Hello from Ckafka!"
      }
```

```
    },
    {
      "Ckafka": {
        "topic": "test-topic",
        "partition":1,
        "offset":37,
        "msgKey": "None",
        "msgBody": "Hello from Ckafka again!"
      }
    }
  ]
}
```

Event Message Structure for CLS Trigger

When the specified CLS trigger receives a message, the CLS backend consumption module will consume the message and encapsulate it to asynchronously invoke your function. In order to ensure the efficiency of data transfer in a single triggering action, the value of the data field is a Base64-encoded ZIP document. For more information, see CLS Trigger.

```
{
  "clslogs": {
    "data": "ewogICAgIm1lc3NhZ2VUeXBlIjogIkRBVEFfTUVTU0FHRSIsCiAgICAib3duZXIiOiAiMT
  }
}
```

After being decoded and decompressed, the log data will look like the following JSON body (using decoded CLS Logs message data as an example):

```
{
    "topic_id": "xxxx-xx-xx-xx-yyyyyyyy",
    "topic_name": "testname",
    "records": [{
        "timestamp": "1605578090000000",
        "content": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
    }, {
        "timestamp": "1605578090000003",
        "content": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
    }]
}
```

Event Message Structure for MPS Trigger

When a specified MPS trigger receives a message, the event structures and fields will be as shown below (with the `WorkflowTask` task as an example). For more information, see MPS Trigger.


```
{
    "EventType":"WorkflowTask",
    "WorkflowTaskEvent":{
        "TaskId":"245****654-WorkflowTask-f46dac7fe2436c47******d71946986t0",
        "Status":"FINISH",
        "ErrCode":0,
        "Message":"",
        "InputInfo":{
            "Type":"COS",
            "CosInputInfo":{
                "Bucket":"macgzptest-125****654",
                "Region":"ap-guangzhou",
                "Object":"/dianping2.mp4"
            }
        },
        "MetaData":{
            "AudioDuration":11.261677742004395,
            "AudioStreamSet":[
                {
                    "Bitrate":127771,
                    "Codec":"aac",
                    "SamplingRate":44100
                }
            ],
            "Bitrate":2681468,
            "Container":"mov,mp4,m4a,3gp,3g2,mj2",
            "Duration":11.261677742004395,
            "Height":720,
            "Rotate":90,
            "Size":3539987,
            "VideoDuration":10.510889053344727,
            "VideoStreamSet":[
                {
                    "Bitrate":2553697,
                    "Codec":"h264",
                    "Fps":29,
                    "Height":720,
                    "Width":1280
                }
            ],
            "Width":1280
```

```
        },
        "MediaProcessResultSet":[
            {
                "Type":"Transcode",
                "TranscodeTask":{
                    "Status":"SUCCESS",
                    "ErrCode":0,
                    "Message":"SUCCESS",
                    "Input":{
                        "Definition":10,
                        "WatermarkSet":[
                            {
                                "Definition":515247,
                                "TextContent":"",
                                "SvgContent":""
                            }
                        ],
                        "OutputStorage":{
                            "Type":"COS",
                            "CosOutputStorage":{
                                "Bucket":"gztest-125****654",
                                "Region":"ap-guangzhou"
                            }
                        },
                        "OutputObjectPath":"/dasda/dianping2_transcode_10",
                        "SegmentObjectName":"/dasda/dianping2_transcode_10_{number}
                        "ObjectNumberFormat":{
                            "InitialValue":0,
                            "Increment":1,
                            "MinLength":1,
                            "PlaceHolder":"0"
                        }
                    },
                    "Output":{
                        "OutputStorage":{
                            "Type":"COS",
                            "CosOutputStorage":{
                                "Bucket":"gztest-125****654",
                                "Region":"ap-guangzhou"
                            }
                        },
                        "Path":"/dasda/dianping2_transcode_10.mp4",
                        "Definition":10,
                        "Bitrate":293022,
                        "Height":320,
                        "Width":180,
                        "Size":401637,
```



```
                    "Duration":11.26200008392334,
                    "Container":"mov,mp4,m4a,3gp,3g2,mj2",
                    "Md5":"31dcf904c03d0cd78346a12c25c0acc9",
                    "VideoStreamSet":[
                        {
                            "Bitrate":244608,
                            "Codec":"h264",
                            "Fps":24,
                            "Height":320,
                            "Width":180
                        }
                    ],
                    "AudioStreamSet":[
                        {
                            "Bitrate":48414,
                            "Codec":"aac",
                            "SamplingRate":44100
                        }
                    ]
                }
            },
            "AnimatedGraphicTask":null,
            "SnapshotByTimeOffsetTask":null,
            "SampleSnapshotTask":null,
            "ImageSpriteTask":null
        },
        {
            "Type":"AnimatedGraphics",
            "TranscodeTask":null,
            "AnimatedGraphicTask":{
                "Status":"FAIL",
                "ErrCode":30010,
                "Message":"TencentVodPlatErr Or Unkown",
                "Input":{
                    "Definition":20000,
                    "StartTimeOffset":0,
                    "EndTimeOffset":600,
                    "OutputStorage":{
                        "Type":"COS",
                        "CosOutputStorage":{
                            "Bucket":"gztest-125****654",
                            "Region":"ap-guangzhou"
                        }
                    },
                    "OutputObjectPath":"/dasda/dianping2_animatedGraphic_20000"
                },
                "Output":null
```



```
            },
            "SnapshotByTimeOffsetTask":null,
            "SampleSnapshotTask":null,
            "ImageSpriteTask":null
        },
        {
            "Type":"SnapshotByTimeOffset",
            "TranscodeTask":null,
            "AnimatedGraphicTask":null,
            "SnapshotByTimeOffsetTask":{
                "Status":"SUCCESS",
                "ErrCode":0,
                "Message":"SUCCESS",
                "Input":{
                    "Definition":10,
                    "TimeOffsetSet":[

                    ],
                    "WatermarkSet":[
                        {
                            "Definition":515247,
                            "TextContent":"",
                            "SvgContent":""
                        }
                    ],
                    "OutputStorage":{
                        "Type":"COS",
                        "CosOutputStorage":{
                            "Bucket":"gztest-125****654",
                            "Region":"ap-guangzhou"
                        }
                    },
                    "OutputObjectPath":"/dasda/dianping2_snapshotByOffset_10_{n
                    "ObjectNumberFormat":{
                        "InitialValue":0,
                        "Increment":1,
                        "MinLength":1,
                        "PlaceHolder":"0"
                    }
                },
                "Output":{
                    "Storage":{
                        "Type":"COS",
                        "CosOutputStorage":{
                            "Bucket":"gztest-125****654",
                            "Region":"ap-guangzhou"
                        }
```



```
                    },
                    "Definition":0,
                    "PicInfoSet":[
                        {
                            "TimeOffset":0,
                            "Path":"/dasda/dianping2_snapshotByOffset_10_0.jpg"
                            "WaterMarkDefinition":[
                                515247
                            ]
                        }
                    ]
                }
            },
            "SampleSnapshotTask":null,
            "ImageSpriteTask":null
        },
        {
            "Type":"ImageSprites",
            "TranscodeTask":null,
            "AnimatedGraphicTask":null,
            "SnapshotByTimeOffsetTask":null,
            "SampleSnapshotTask":null,
            "ImageSpriteTask":{
                "Status":"SUCCESS",
                "ErrCode":0,
                "Message":"SUCCESS",
                "Input":{
                    "Definition":10,
                    "OutputStorage":{
                        "Type":"COS",
                        "CosOutputStorage":{
                            "Bucket":"gztest-125****654",
                            "Region":"ap-guangzhou"
                        }
                    },
                    "OutputObjectPath":"/dasda/dianping2_imageSprite_10_{number
                    "WebVttObjectName":"/dasda/dianping2_imageSprite_10",
                    "ObjectNumberFormat":{
                        "InitialValue":0,
                        "Increment":1,
                        "MinLength":1,
                        "PlaceHolder":"0"
                    }
                },
                "Output":{
                    "Storage":{
                        "Type":"COS",
```



```
                            "CosOutputStorage":{
                                "Bucket":"gztest-125****654",
                                "Region":"ap-guangzhou"
                            }
                        },
                        "Definition":10,
                        "Height":80,
                        "Width":142,
                        "TotalCount":2,
                        "ImagePathSet":[
                            "/dasda/imageSprite/dianping2_imageSprite_10_0.jpg"
                        ],
                        "WebVttPath":"/dasda/imageSprite/dianping2_imageSprite_10.v
                    }
                }
            }
        ]
    }
}
```

Event Message Structure for CLB Trigger

When a CLB trigger receives a request, event data will be sent to the bound function in JSON format as shown below. For more information, see CLB Trigger Description.

```
{
  "headers": {
    "Content-type": "application/json",
    "Host": "test.clb-scf.com",
    "User-Agent": "Chrome",

    "X-Stgw-Time": "1591692977.774",
    "X-Client-Proto": "http",
    "X-Forwarded-Proto": "http",
    "X-Client-Proto-Ver": "HTTP/1.1",
    "X-Real-IP": "9.43.175.219",
    "X-Forwarded-For": "9.43.175.xx"

    "X-Vip": "121.23.21.xx",
    "X-Vport": "xx",
    "X-Uri": "/scf_location",
    "X-Method": "POST"
    "X-Real-Port": "44347",
  },
```

```
  "payload": {
    "key1": "123",
    "key2": "abc"
  },
  "isBase64Encoded": "false"
}
```

Event Message Structure for EventBridge Trigger

With EventBridge, you can further expand the function event trigger sources and send events generated by EventBridge to SCF in the following form, where the content of the `"data"` field is determined by the event source. Here, TDMQ is used as an example:

```
{
 "specversion": "0",
 "id": "13a3f42d-7258-4ada-da6d-023a33******",
 "type": "connector:tdmq",
 "source": "tdmq.cloud.tencent",
 "subjuect": "qcs::tdmq:$region:$account:topicName/$topicSets.clusterId/$topicSets
 "time": "1615430559146",
 "region": "ap-guangzhou",
 "datacontenttype": "application/json;charset=utf-8",
 "data": {
          "topic":  "persistent://appid/namespace/topic-1",
        "tags": "testtopic",
          "TopicType": "0",
          "subscriptionName": "xxxxxx",
          "toTimestamp": "1603352765001",
        "partitions": "0",
          "msgId": "123345346",
          "msgBody": "Hello from TDMQ!"
 }
```

API Gateway Trigger Overview

Last updated：2024-12-02 19:58:17

You can implement backend web services by writing SCF functions and providing services through API Gateway which will pass the request content as parameters to the function and return the result from the function back to the requester as the response.

Note:

API Gateway triggers can trigger both event-triggered functions and HTTP-triggered functions. This document only describes the request method of **event-triggered function** triggering. For more information on HTTP-triggered function triggering, see Trigger Management.

Characteristics of API Gateway triggers:

Push model

After API Gateway receives an API request, if the API Gateway backend is connected with an SCF function, the function will be triggered. Meanwhile, API Gateway will send the relevant information of the API request to the triggered function as `event` input parameters, such as the specific service that receives the request, API rule, actual request path, method, and `path`, `header`, and `query` of the request.

Sync invocation

API Gateway invokes the function synchronously, and it will wait for the function to return before the timeout period configured in it elapses. For more information on invocation types, see Invocation Types.

API Gateway Trigger Configuration

API Gateway triggers can be configured in the **SCF console** or the **API Gateway console**.

SCF console

API Gateway console

In the **SCF console**, you can add API Gateway triggers, select existing or create API services, and define the request methods (currently, six methods are supported, namely, **ANY, GET, HEAD, POST, PUT, and DELETE**), environments (test, pre-release, and release environments), and authentication methods (API Gateway key pair).

When configuring API rules in the **API Gateway console**, you can select SCF as the backend and select functions in the same region as the API service. In the API Gateway console, you can configure and manage advanced API services such as traffic throttling plan, blocklist, and allowlist.

When you configure the connection with SCF in API Gateway, you also need to configure the timeout period. The request timeout period in API Gateway and the execution timeout period in SCF take effect respectively. The timeout rules are as follows:

API Gateway timeout period > SCF timeout period

The SCF timeout period takes effect first, the API request response is `200 HTTP code`, but the returned content is the error message of SCF timeout.

API Gateway timeout period < SCF timeout period

The API Gateway timeout period takes effect first, the API request response is `5xx HTTP code`, which indicates that the request timed out.

Limitation on API Gateway Trigger Binding

In API Gateway, one API rule can be bound to only one function, but one function can be bound to multiple API rules as the backend. You can create an API with different paths in the API Gateway console and point the backend to the same function. APIs with the same path, same request method, and different release environments are regarded as the same API and cannot be bound repeatedly.

API Gateway triggers currently can only be bound to functions in the same region; for example, a function created in the Guangzhou region can only be bound to and triggered by API rules created in the Guangzhou region. If you want to trigger a function through API Gateway configuration in a specific region, please create a function in that region.

Request and Response

Request method is the method to process request sent from API Gateway to SCF, and response method is the method to process the returned value sent from SCF to API Gateway. Both request and response methods can be planned and implemented by means of passthrough and integration.

Integration request and passthrough request

Integration request means that API Gateway converts the content of the HTTP request into request data structures which are passed to the function for handling as `event` input parameters of the function. The following details the request data structures.

For more information on passthrough requests, see Trigger Management.

Note:

When transferring images or files to SCF through API Gateway, you need to Base64-encode them. If the size of a Base64-encoded file is above 6 MB, we recommend you upload the file to COS through the client and pass the object address to SCF first. Then, SCF will pull the file from COS to complete the upload.

Event message structures of integration request for API Gateway trigger

When an API Gateway trigger receives a request, it sends the event data to the bound function in JSON format as shown below:

```
{
  "requestContext": {
    "serviceId": "service-f94sy04v",
    "path": "/test/{path}",
    "httpMethod": "POST",
    "requestId": "c6af9ac6-7b61-11e6-9a41-93e8deadbeef",
    "identity": {
      "secretId": "abdcdxxxxxxxsdfs"
    },
    "sourceIp": "10.0.2.14",
    "stage": "release"
  },
  "headers": {
    "accept-Language": "en-US,en,cn",
    "accept": "text/html,application/xml,application/json",
    "host": "service-3ei3tii4-251000691.ap-guangzhou.apigateway.myqloud.com",
    "user-Agent": "User Agent String"
  },
  "body": "{\\"test\\":\\"body\\"}",
  "pathParameters": {
    "path": "value"
  },
  "queryStringParameters": {
    "foo": "bar"
  },
  "headerParameters":{
    "Refer": "10.0.2.14"
  },
  "stageVariables": {
    "stage": "release"
  },
  "path": "/test/value",
  "queryString": {
    "foo" : "bar",
    "bob" : "alice"
  },
  "httpMethod": "POST"
}
```

The data structures are as detailed below:

Structure	Description
requestContext	Configuration information, request ID, authentication information, and source information of the API gateway where the request comes from. `serviceId` , `path` , and `httpMethod` are service ID, API path, and method of API Gateway.

	`stage` indicates the environment of the request source API. `requestId` identifies the unique ID of the current request. `identity` identifies the user's authentication method and information. `sourceIp` identifies the request source IP.
path	Records the complete `Path` information of the actual request.
httpMethod	Records the `HTTP` method of the actual request.
queryString	Records the complete `Query` content of the actual request.
body	Records the content of the actual request after being converted into a `String`.
headers	Records the complete `Header` content of the actual request.
pathParameters	Records the `Path` parameters configured in API Gateway and their actual values.
queryStringParameters	Records the `Query` parameters configured in API Gateway and their actual values.
headerParameters	Records the `Header` parameters configured in API Gateway and their actual values.

Note:

The content of `requestContext` may be increased during API Gateway iteration. At present, it is guaranteed that the content of the data structure will only be increased but not reduced, so that the existing structure will not be compromised.

Parameters in real requests may appear in multiple locations and can be selected based on your business needs.

Integration response and passthrough response

Integration response means that API Gateway parses the returned content of the function and constructs an HTTP response based on the parsed content. With the aid of integration response, you can control the status code, headers, and body content of the response by using code, and implement the response in a custom format, such as XML, HTML, JSON, and even JS. When using integration response, data structures need to be returned based on the rules of integration response for API Gateway trigger before they can be successfully parsed by API Gateway; otherwise, error message `{"errno":403,"error":"Invalid scf response format. please check your scf response format."}` will appear.

Passthrough response means that API Gateway directly passes the returned content of the function to the API requester. Generally, the data format of this type of responses is fixed at JSON format, the status code is defined according to the status of function execution, and status code 200 is returned if the function is successfully executed.

With passthrough response, you can get the JSON format and parse the structures at the call location to get the content in the structures.

Note:

If the API Gateway trigger is configured in the API Gateway console, the way to handle the response is passthrough response by default. To enable integration response, select **Enable integration response** at the backend configuration location in the API configuration and return the content in the data structures detailed below in the code. If the API Gateway trigger is configured in the SCF console, the integration response feature is enabled by default. Please pay attention to the format of the returned data.

Returned data structures of integration response for API Gateway trigger

If integration response is set for API Gateway, the data structure in the following JSON format should be returned to API Gateway.

```
{
    "isBase64Encoded": false,
    "statusCode": 200,
    "headers": {"Content-Type":"text/html"},
    "body": "<html><body><h1>Heading</h1><p>Paragraph.</p></body></html>"
}
```

The data structures are as detailed below:

Structure	Description
isBase64Encoded	This indicates whether the content in the `body` is Base64-encoded binary. It should be `true` or `false` in JSON format. The specification of `true` and `false` varies by language, so you should adjust based on your actual language.
statusCode	HTTP return code, which should be an integer value.
headers	HTTP return header, which should contain multiple `key-value` or `key:[value,value]` objects. Both key and value should be strings. The `Location` key header is not supported currently.
body	HTTP return body.

Taking Python 3.6 as an example, the sample code is as follows:

```
# -*- coding: utf8 -*-
import json
def main_handler(event, context):
    return {
        "isBase64Encoded": false,
        "statusCode": 200,
```

```
        "headers": {"Content-Type":"text/html"},
        "body": "<html><body><h1>Heading</h1><p>Paragraph.</p></body></html>"
    }
```

The returned result of a function triggered by API Gateway is as follows:

If you need to return multiple headers with the same key, you can use a string array to describe different values; for example:

```
{
    "isBase64Encoded": false,
    "statusCode": 200,
    "headers": {"Content-Type":"text/html","Key":["value1","value2","value3"]},
    "body": "<html><body><h1>Heading</h1><p>Paragraph.</p></body></html>"
}
```

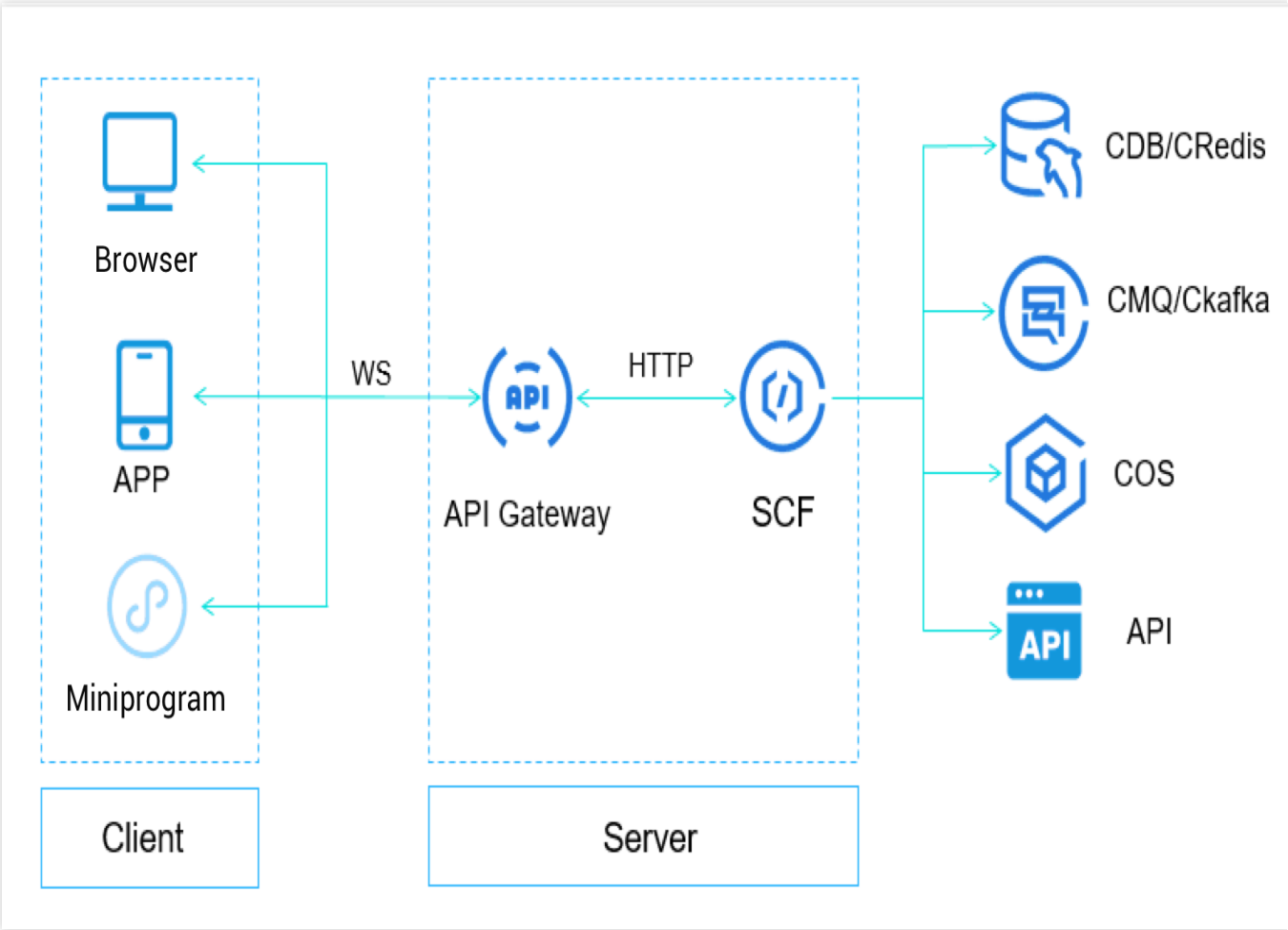
Websocket
How It Works

Last updated：2024-12-02 19:58:17

Note:
This document describes how to support WebSocket for **event-triggered functions**. Currently, **HTTP-triggered functions** already supports the native WebSocket protocol.

How to Implement

WebSocket is a new TCP-based network protocol. It implements full-duplex communication between browser and server, i.e., allowing server to actively send information to client. In contrast, a server using the traditional HTTP protocol only allows a client to get the data that needs to be pushed through polling or long polling.
Since SCF is stateless and trigger-based (i.e., it will be triggered only when an event arrives), in order to implement WebSocket, SCF is used in conjunction with API Gateway to sustain and maintain the connection with the client through API Gateway. You can assume that API Gateway and SCF work together to implement the server. When sent by the client, a message is first passed to API Gateway, which then triggers the SCF function. When the server function sends a message to the client, the function first posts the message to the reverse push link of API Gateway, which then pushes the message to the client. The specific implementation architecture is as follows:



The entire lifecycle of WebSocket mainly consists of the following events:

Connection establishment: the client requests to connect with the server and establishes a connection.

Data upstream: the client sends data to the server through the established connection.

Data downstream: the server sends data to the client through the established connection.

Client disconnection: the client requests to close the established connection.

Server disconnection: the server requests to close the established connection.

SCF and API Gateway handle the events throughout the lifecycle of WebSocket as follows:

Connection establishment: the client establishes a WebSocket connection with API Gateway which sends a connection establishment event to SCF.

Data upstream: the client sends data through WebSocket, and API Gateway forwards the data to SCF.

Data downstream: SCF sends a request to the push address specified by API Gateway, which then sends the data to the client through WebSocket.

Client disconnection: the client requests to disconnect, and API Gateway sends a disconnection event to SCF.

Server disconnection: SCF sends a disconnection request to the push address specified by API Gateway, which will close the WebSocket connection after receiving the request.

Therefore, the interaction between API Gateway and SCF needs to be sustained by three types of functions:

Registration function: this function is triggered when a WebSocket connection is requested and established between the client and API Gateway, notifying SCF of the `secConnectionID` of the WebSocket connection. The `secConnectionID` is usually recorded in the persistent storage in this function for reverse push of subsequent data.
Cleanup function: this function is triggered when the client initiates a WebSocket disconnection request, notifying SCF to prepare to disconnect the `secConnectionID`. The `secConnectionID` is usually cleaned from the persistent storage in this function.
Transfer function: this function is triggered when the client sends data through the WebSocket connection, notifying SCF of the `secConnectionID` of the connection and the data sent. Business data is usually processed in this function. For example, it determines whether to push data to other `secConnectionID` values in the persistent storage.

Note:

When you need to actively push data to a `secConnectionID` or disconnect a `secConnectionID`, the reverse push address of API Gateway has to be used.

Data Structures

Connection establishment

1. When the client initiates a WebSocket connection establishment request, API Gateway encapsulates the agreed upon JSON data structures in the request body and sends it to the registration function in the HTTP POST method. You can get the request body from the function's event. Below is a sample:

```
{
    "requestContext": {
    "serviceName": "testsvc",
    "path": "/test/{testvar}",
    "httpMethod": "GET",
    "requestId": "c6af9ac6-7b61-11e6-9a41-93e8deadbeef",
    "identity": {
      "secretId": "abdcdxxxxxxxsdfs"
    },
    "sourceIp": "10.0.2.14",
    "stage": "prod",
    "websocketEnable":true
    },
    "websocket":{
    "action":"connecting",
    "secConnectionID":"xawexasdfewezdfsdfeasdfffa==",
    "secWebSocketProtocol":"chat,binary",
    "secWebSocketExtensions":"extension1,extension2"
    }
```

```
}
```

The data structures are as detailed below:

Structure Name	Content
requestContext	Configuration information, request ID, authentication information, and source information of API Gateway where the request comes from, including: serviceName, path, httpMethod: they point to API Gateway's service, API path, and method. stage: it points to the environment where the request source API is located. requestId: it identifies the unique ID of the current request. identity: it identifies the user's authentication method and authentication information. sourceIp: it identifies the request source IP.
websocket	Details of connection establishment, including: action: action of this request. secConnectionID: a string that identifies the ID of the WebSocket connection. The original length is 128 bits, which is a Base64-encoded string with a total of 32 characters. secWebSocketProtocol: string; optional.It represents the list of sub-protocols. If this field is in the original request, it will be passed to the function; otherwise, it will not appear. secWebSocketExtensions: string; optional.It represents the list of extensions. If this field is in the original request, it will be passed to the function; otherwise, it will not appear.

Note:

The content of `requestContext` may be increased significantly during API Gateway iteration. At present, it is guaranteed that the content of the data structure will only be increased but not reduced, so that the existing structure will not be compromised.

2. When the registration function receives the connection establishment request, it needs to return the response message of whether to agree to establish the connection to API Gateway at the end of function handling. The response body must be in JSON format as shown in the sample below:

```
{
    "errNo":0,
    "errMsg":"ok",
    "websocket":{
      "action":"connecting",
      "secConnectionID":"xawexasdfewezdfsdfeasdfffa==",
      "secWebSocketProtocol":"chat,binary",
      "secWebSocketExtensions":"extension1,extension2"
     }
}
```

The data structures are as detailed below:

Structure Name	Content
errNo	Integer; required. Response error code. If `errNo` is 0, the handshake succeeded, and the connection was allowed to be established.
errMsg	String; required. Cause of error. If `errNo` is not 0, this field will take effect.
websocket	Details of connection establishment, including: action: action of this request. secConnectionID: a string that identifies the ID of the WebSocket connection. The original length is 128 bits, which is a Base64-encoded string with a total of 32 characters. secWebSocketProtocol: string; optional.It is the value of a single sub-protocol. If this field is in the original request, it will be passed through to the client by API Gateway. secWebSocketExtensions: string; optional.It is the value of a single extension. If this field is in the original request, it will be passed through to the client by API Gateway.

Note:

If the SCF request times out, it will be deemed by default that the connection establishment failed.

When API Gateway receives the response message from SCF, it will check the HTTP response code first. If the response code is 200, it will parse the response body; otherwise, it will deem that SCF failed and connection establishment was refused.

Data transfer

Upstream data transfer

Transfer request

When the client sends data through WebSocket, API Gateway will encapsulate the agreed JSON data structures in the request body and send it to the transfer function in the HTTP POST method. You can get the request body from the function's event. Below is a sample:

```
{
  "websocket":{
    "action":"data send",
    "secConnectionID":"xawexasdfewezdfsdfeasdfffa==",
    "dataType":"text",
    "data":"xxx"
  }
}
```

The data structures are as detailed below:

Parameter	Content
websocket	Details of data transfer.
action	Action of this request, such as "data send" in this document.
secConnectionID	A string that identifies the ID of the WebSocket connection. The original length is 128 bits, which is a Base64-encoded string with a total of 32 characters.
dataType	Type of the transferred data. "binary": binary. "text": text.
data	Transferred data. If `dataType` is `binary`, it will be a Base64-encoded binary stream; if `dataType` is `text`, it will be a string.

Transfer response

After the transfer function finishes executing, it will return an HTTP response to API Gateway which will act according to the response code:

If the response code is 200, the function was executed successfully.

If the response code is not 200, a system failure occurred, and API Gateway will actively send a FIN packet to the client.

Note:

API Gateway does not handle the content in the response body.

Downstream data callback

Callback request

When SCF needs to push data to the client or actively disconnect, it can initiate a request, encapsulate the data in the request body, and send it to the reverse push address of API Gateway in the POST method. The request body must be in JSON format, as shown in the sample below:

```
{
  "websocket":{
    "action":"data send", // Send data to the client
    "secConnectionID":"xawexasdfewezdfsdfeasdfffa==",
    "dataType":"text",
    "data":"xxx"
  }
```

```
}
```

```
{
  "websocket":{
    "action":"closing", // Send the disconnection request
    "secConnectionID":"xawexasdfewezdfsdfeasdfffa=="
  }
}
```

The data structures are as detailed below:

Field	Content
websocket	Details of data transfer.
action	Action of this request, which can be `data send` or `closing`: "data send": it is to send data to the client. "closing": it is to initiate a disconnection request to the client, where the `dataType` and `data` are optional.
secConnectionID	A string that identifies the ID of the WebSocket connection. The original length is 128 bits, which is a Base64-encoded string with a total of 32 characters.
dataType	Type of the transferred data, including two types: "binary": binary. "text": text.
data	Transferred data: If `dataType` is `binary`, it is a Base64-encoded binary stream. If `dataType` is `text`, it is a string.

Callback response

After the callback is over, the result of the callback can be determined based on the response code of API Gateway:

If the response code is 200, the call succeeded.

If the response code is not 200, a system failure occurred, and API Gateway will actively send a FIN packet to the client.

In addition, the response body in JSON format can be obtained in the response result as shown in the sample below:

```
{
   "errNo":0,
   "errMsg":"ok"
}
```

The data structures are as detailed below:

Field	Content

errNo	Integer; response error code. 0 means success.
errMsg	String; cause of error.

Connection cleanup

Active disconnection by client

Logout request

When the client actively initiates a WebSocket disconnection request, API Gateway will encapsulate the agreed upon JSON data structures in the request body and send it to the cleanup function in the HTTP POST method. You can get the request body from the function's event. Below is a sample:

```
{
  "websocket":{
    "action":"closing",
    "secConnectionID":"xawexasdfewezdfsdfeasdfffa=="
  }
}
```

The data structures are as detailed below:

Field	Content
websocket	Details of disconnection.
action	Action of this request, which is "closing" here.
secConnectionID	String. It identifies the ID of the WebSocket connection. The original length is 128 bits, which is a Base64-encoded string with a total of 32 characters.

Note:

In the cleanup function, you can get the `secConnectionID` from the event and delete the ID from the persistent storage (such as a database).

Logout response

After the cleanup function finishes executing, it will return an HTTP response to API Gateway, which will act according to the response code:

If the response code is 200, the function was executed successfully.

If the response code is not 200, a system failure occurred.

Note:

API Gateway does not handle the content in the response body.

Active disconnection by server

Please see **Downstream data callback**. SCF can initiate a request in the function, encapsulate the following data structures in the request body, and send it to the reverse push address of API Gateway in the POST method.

```
{
  "websocket":{
    "action":"closing", // Send the disconnection request
    "secConnectionID":"xawexasdfewezdfsdfeasdfffa=="
  }
}
```

Note:

When actively disconnecting the link with the client, you need to get the `secConnectionID` of the client's WebSocket, enter it in the data structure, and then delete the ID from the persistent storage (such as a database).

Usage

Last updated：2024-12-02 19:58:17

In the How It Works document, it is mentioned that three types of SCF functions are required to sustain the interaction with API Gateway:
Registration function: this function is triggered when a WebSocket connection is requested and established between the client and API Gateway, notifying SCF of the `secConnectionID` of the WebSocket connection. The `secConnectionID` is usually recorded in the persistent storage in this function for reverse push of subsequent data.
Cleanup function: this function is triggered when the client initiates a WebSocket disconnection request, notifying SCF to prepare to disconnect the `secConnectionID` . The `secConnectionID` is usually cleaned from the persistent storage in this function.
Transfer function: this function is triggered when the client sends data through the WebSocket connection, notifying SCF of the `secConnectionID` of the connection and the data sent. Business data is usually processed in this function. For example, it determines whether to push data to other `secConnectionID` values in the persistent storage.
Note:
When you need to actively push data to a `secConnectionID` or disconnect a `secConnectionID` , the reverse push address of API Gateway has to be used.
This document uses Python 2.7 as an example to describe how to write the `main_handler` for various functions.

Sample Function Code

Registration function

```
# -*- coding: utf8 -*-
import json
import requests
def main_handler(event, context):
    print('Start Register function')
    print("event is %s"%event)
    retmsg = {}
    global connectionID
    if 'requestContext' not in event.keys():
        return {"errNo":101, "errMsg":"not found request context"}
    if 'websocket' not in event.keys():
        return {"errNo":102, "errMsg":"not found websocket"}
    connectionID = event['websocket']['secConnectionID']
    retmsg['errNo'] = 0
```

```
    retmsg['errMsg'] = "ok"
    retmsg['websocket'] = {
            "action":"connecting",
            "secConnectionID":connectionID
        }
    if "secWebSocketProtocol" in event['websocket'].keys():
        retmsg['websocket']['secWebSocketProtocol'] = event['websocket']['secWebSoc
    if "secWebSocketExtensions" in event['websocket'].keys():
        ext = event['websocket']['secWebSocketExtensions']
        retext = []
        exts = ext.split(";")
        print(exts)
        for e in exts:
            e = e.strip(" ")
            if e == "permessage-deflate":
                #retext.append(e)
                pass
            if e == "client_max_window_bits":
                #retext.append(e+"=15")
                pass
        retmsg['websocket']['secWebSocketExtensions'] = ";".join(retext)
    print("connecting \\n connection id:%s"%event['websocket']['secConnectionID'])
    print(retmsg)
    return retmsg
```

Note:

In this function, you can add other business logic as needed. For example, you can save `secConnectionID` to TencentDB or create and associate a chat room.

Transfer function

```
# -*- coding: utf8 -*-
import json
import requests
g_connectionID = 'xxxx'  # Forward the message to a specific WebSocket connection
sendbackHost = "http://set-7og8wn64.cb-beijing.apigateway.tencentyun.com/api-xxxx"
# Actively push the message to the client
def send(connectionID,data):
    retmsg = {}
    retmsg['websocket'] = {}
    retmsg['websocket']['action'] = "data send"
    retmsg['websocket']['secConnectionID'] = connectionID
    retmsg['websocket']['dataType'] = 'text'
    retmsg['websocket']['data'] = json.dumps(data)
    print("send msg is %s"%retmsg)
    r = requests.post(sendbackHost, json=retmsg)
```

```
def main_handler(event, context):
    print('Start Transmission function')
    print("event is %s"%event)
    if 'websocket' not in event.keys():
        return {"errNo":102, "errMsg":"not found web socket"}
    for k in event['websocket'].keys():
        print(k+":"+event['websocket'][k])
    # Send the content to a specific client
    #connectionID = event['websocket']['secConnectionID']
    data = event['websocket']['data']
    send(g_connectionID,data)
    return event
```

Note:

In this function, you can add other business logic as needed. For example, you can forward the data obtained in this request to another `secConnectionID` stored in TencentDB.

In the API details in API Gateway, you can get the reverse push address.

Cleanup function

```
import json
import requests
g_connectionID = 'xxxx'  # Forward the message to a specific WebSocket connection
sendbackHost = "http://set-7og8wn64.cb-beijing.apigateway.tencentyun.com/api-xxxx"
# Actively send disconnection information
def close(connectionID):
    retmsg = {}
    retmsg['websocket'] = {}
    retmsg['websocket']['action'] = "closing"
    retmsg['websocket']['secConnectionID'] = connectionID
    r = requests.post(sendbackHost, json=retmsg)
    return retmsg
def main_handler(event, context):
    print('Start Delete function')
    print("event is %s"%event)
    if 'websocket' not in event.keys():
        return {"errNo":102, "errMsg":"not found web socket"}
    for k in event['websocket'].keys():
        print(k+":"+event['websocket'][k])
    #close(g_connectionID)
    return event
```

Note:

In this function, you can add other business logic as needed. For example, you can remove the `secConnectionID` disconnected in this request from TencentDB or force the client of a `secConnectionID`

to go offline.

COS Trigger

COS Trigger Description

Last updated：2024-12-02 19:58:17

You can write an SCF function to handle object creation and deletion events in a COS bucket. COS can publish events to the function and invoke it by using the event data as parameters. You can add a bucket notification configuration in the COS bucket, which can identify information such as the trigger event type and name of the function to be invoked.

Characteristics of COS triggers:

Push model

COS monitors the specified bucket action (event type) and invokes the associated function to push the event data to the function. In the push model, the bucket notification is used to store the event source mapping with COS.

Async invocation

A COS trigger always invokes a function asynchronously, and the result is not returned to the invoker. For more information on invocation types, see "Invocation Types" in How It Works.

COS Trigger Attributes

COS bucket (required): The configured COS bucket, which can only be a COS bucket in the same region.

Event type (required): It supports "file upload" and "file deletion" as well as finer-grained upload and deletion events. For specific event types, see the table below. The event type determines when the trigger triggers the function. For example, if "File upload" is selected, the function will be triggered when there is a file uploaded to the COS bucket.

Event Type	Description
cos:ObjectCreated:*	All upload events mentioned below can trigger the function.
cos:ObjectCreated:Put	The function will be triggered when a file is created through the `Put Object` API.
cos:ObjectCreated:Post	The function will be triggered when a file is created through the `Post Object` API.
cos:ObjectCreated:Copy	The function will be triggered when a file is created through the `Put Object - Copy` API.
cos:ObjectCreated:CompleteMultipartUpload	The function will be triggered when a file is created through the `CompleteMultipartUpload` API.
cos:ObjectCreated:Origin	The function will be triggered when an object is created through

	COS origin-pull.
cos:ObjectCreated:Replication	The function will be triggered when an object is created through cross-region replication.
cos:ObjectRemove:*	All deletion events mentioned below can trigger the function.
cos:ObjectRemove:Delete	The function will be triggered when an object in a bucket for which versioning is not enabled is deleted through the `Delete Object` API, or an object on a specified version is deleted with `versionid`.
cos:ObjectRemove:DeleteMarkerCreated	The function will be triggered when an object in a bucket for which versioning is enabled or suspended is deleted through the `Delete Object` API.
cos:ObjectRestore:Post	The function will be triggered when an archive restoration job is created.
cos:ObjectRestore:Completed	The function will be triggered when an archive restoration job is completed.

Prefix filtering (optional): Prefix filtering is usually used to filter file events in a specified directory. For example, if the prefix to be filtered is `test/`, only file events in the `test/` directory can trigger the function, while those in the `hello/` directory cannot.
Suffix filtering (optional): Suffix filtering is usually used to filter file events in a specified type or with a specified suffix. For example, if the suffix to be filtered is `.jpg`, only file events of the `.jpg` type can trigger the function, while those of the `.png/` type cannot.

COS Trigger Use Limits

In order to avoid errors in COS event production and delivery, for the combination of each event (such as file upload/deletion) and prefix/suffix filter in each bucket, COS limits that the same rule can be bound to only one function that can be triggered. Therefore, when you create a COS trigger, do not configure repeated rules for the same COS bucket. For example, if you configure a `Created: *` event trigger in the test bucket for function A (with no filter rule configured), then the upload events (including `Created:Put` and `Created:Post`) in the test bucket cannot be bound to other functions, but you can configure an `ObjectRemove` event trigger in the test bucket for function B.
When using prefix and suffix filter rule, in order to ensure the uniqueness of the triggering events in the same bucket, it should be noted that the same bucket cannot use overlapping prefixes, overlapping suffixes, or overlapping combinations of prefixes and suffixes to define the filter rule for the same event type. For example, if you configure a

`Created: *` trigger event with prefix filter of `Log` in the test bucket for function A, then you cannot configure a `Created: *` trigger event with prefix filter of `Log` in the test bucket.

In addition, COS triggers can only trigger functions in the same region; for example, for an SCF function created in the Guangzhou region, you can only select a COS bucket in the Guangzhou region (South China) when configuring a COS trigger. If you want to trigger a function through COS bucket events in a specific region, create a function in that region.

A COS trigger has limits in two dimensions: SCF and COS, as detailed below:

SCF dimension: One function can be bound to 10 COS triggers at most.

COS dimension: Only one function can be bound to the same event and prefix/suffix rules in a single COS bucket.

Event Message Structure for COS Trigger

When an object creation or deletion event occurs in the specified COS bucket, event data will be sent to the bound function in JSON format as shown below.

```
{
    "Records": [{
        "cos": {
            "cosSchemaVersion": "1.0",
            "cosObject": {
                "url": "http://testpic-1253970026.cos.ap-chengdu.myqcloud.com/testf
                "meta": {
                    "x-cos-request-id": "NWMxOWY4MGFfMjViMjU4NjRfMTUyMVxxxxxxxxx=",
                    "Content-Type": "",
                    "x-cos-meta-mykey": "myvalue"
                },
                "vid": "",
                "key": "/1253970026/testpic/testfile",
                "size": 1029
            },
            "cosBucket": {
                "region": "cd",
                "name": "testpic",
                "appid": "1253970026"
            },
            "cosNotificationId": "unkown"
        },
        "event": {
            "eventName": "cos:ObjectCreated:*",
            "eventVersion": "1.0",
            "eventTime": 1545205770,
            "eventSource": "qcs::cos",
            "requestParameters": {
```

```
                "requestSourceIP": "192.168.15.101",
                "requestHeaders": {
                    "Authorization": "q-sign-algorithm=sha1&q-ak=xxxxxxxxxxxxxx&q-s
                }
            },
            "eventQueue": "qcs:0:scf:cd:appid/1253970026:default.printevent.$LATEST
            "reservedInfo": "",
            "reqid": 179398952
        }
    }]
}
```

The data structures are as detailed below:

Structure	Description
Records	List structure. There may be multiple messages merged in the list.
event	This records the event information, including event version, event source, event name, time, queue information, request parameters, and request ID.
cos	This records the COS information corresponding to the event.
cosBucket	This records the bucket of the specific event, including bucket name, region, and user `APPID` (which can be obtained on the Account Info page).
cosObject	This records the object of the specific event, including object file path, size, custom metadata, and access URL.

Sample

The following is a sample COS trigger in Java for your reference:

```
https://github.com/tencentyun/scf-demo-
java/blob/master/src/main/java/example/Cos.java
```

Usage

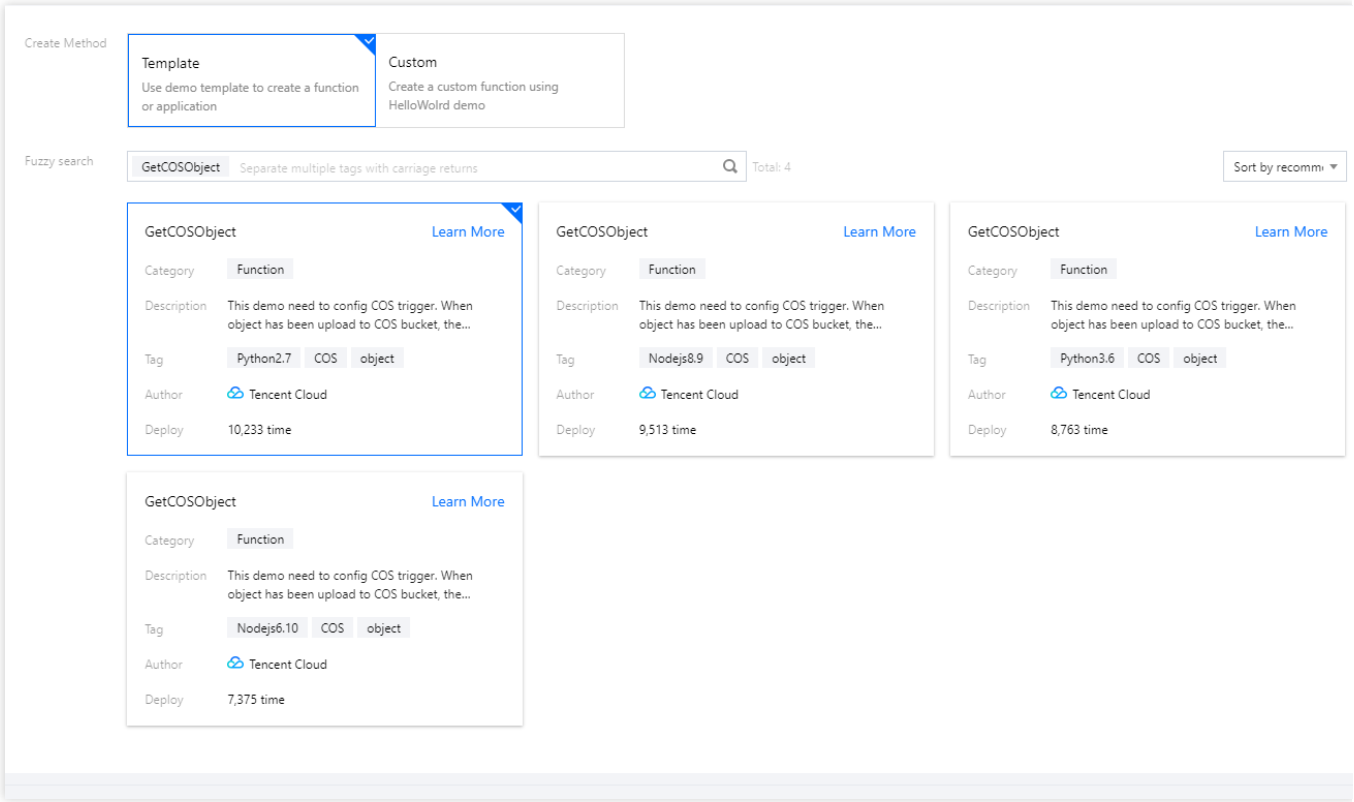
Last updated：2024-12-02 19:58:17

This document describes how to create a COS trigger and invoke a function.

Step 1. Create a function

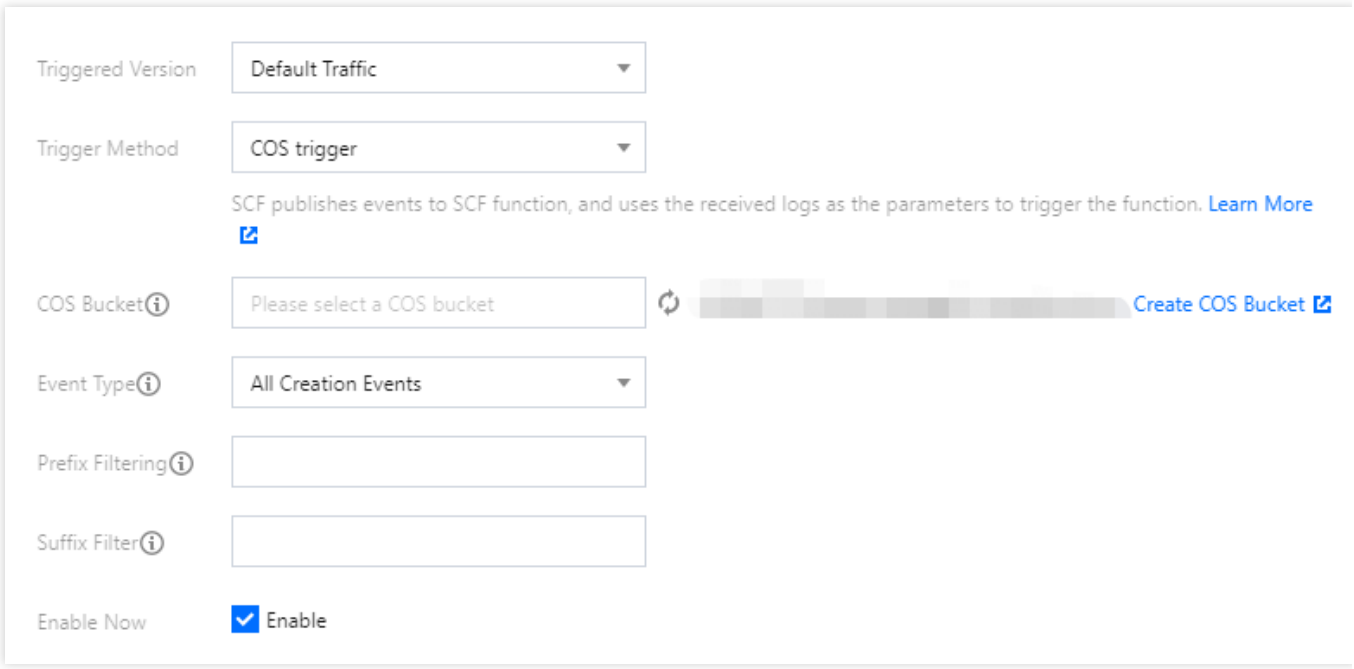
Log in to the SCF console and upload and deploy your function code on the **Create** page. For more information, please see Creating Functions in Console.

The following takes the COS sample template as an example to create a function project. In the default creation process with the template, a trigger is directly configured. In actual use cases, you can also configure a trigger after creating the function. Here, configuration after function creation is used as an example for description:



Step 2. Configure a trigger

After selecting **COS Trigger**, configure the bucket, triggering event type, and other information as prompted to create a trigger:



Step 3. Manage the trigger

After successful creation, you can see the information of the created trigger on the **Trigger Management** page.

CLS Trigger

CLS Trigger Description

Last updated：2024-12-02 19:58:17

You can write an SCF function to process the logs collected in the CLS service. By passing the collected logs as a parameter, the function can be invoked and the function code can process and analyze the data or dump it to other Tencent Cloud services.

Characteristics of CLS triggers:

Push model

CLS monitors the specified log topic, aggregates data within a certain period of time, and invokes the associated function to push the event data to the function.

Async invocation

a CLS trigger always invokes a function asynchronously, and the result is not returned to the invoker. For more information on invocation types, please see "Invocation Types" in How It Works.

CLS Trigger Attributes

Logset: configure the CLS logset you want to connect to. It can only be a logset in the same region as the function.

Log topic: configure the CLS log topic you want to connect to, which is the smallest unit for managing and configuring CLS triggers.

Maximum waiting time: configure the longest waiting time for a single event pull.

CLS Trigger Consumption and Message Delivery

After consuming messages, the CLS backend consumption module will encapsulate them into event structures according to the **max waiting time**, **delivery process**, and **message body size** and then initiate async function invocation. The applicable limits are as follows:

Maximum waiting time

at present, the SCF backend consumption module requires this value to be between 3s and 300s to avoid an excessive latency before consumption. For example, if the maximum waiting time is set to 60s, the consumption template will aggregate log data once every 60s and deliver it to the function.

Event size limit for async invocation

128 KB. For more information, please see Limits. If a message of the log topic is large (for example, if the message body size exceeds 128 KB after being compressed by the backend component within the maximum waiting time), then due to the limit of 128 KB for async invocation, the system will extract and deliver the message body of 128 KB

after being compressed by gzip in sequence from the event structure passed to the function instead of using the data aggregated within the maximum waiting time.

Delivery process

if a non-retriable error occurs during the delivery process, such as `AccessDeniedException` or `ResourceNotFoundException` , the CLS trigger will pause the transfer retry logic.

Note:

In the message delivery process, the time aggregation may vary by combination; that is, the number of messages in each event structure ranges from **1 to the maximum waiting time**. If the configured maximum waiting time is too long, there may be cases where the aggregation time in an event structure will never reach the maximum aggregation time.

After the event content is obtained by the function, each message can be guaranteed for processing by loop handling, and it is not assumed that the message time passed each time is constant.

Event Message Structure for CLS Trigger

When the specified CLS trigger receives a message, the CLS backend consumption module will consume the message and encapsulate it to asynchronously invoke your function. In order to ensure the efficiency of data transfer in a single triggering action, the value of the data field is a Base64-encoded Gzip document.

```
{
  "clslogs": {
    "data":
"ewogICAgIm1lc3NhZ2VUeXBlIjogIkRBVEFfTUVTU0FHRSIsCiAgICAib3duZXIiOiAiMTIzNDU2Nz
g5MDEyIiwKICAgICJsb2dHcm91cCI6I..."
  }
}
```

After being decoded and decompressed, the log data will look like the following JSON body (using decoded CLS Logs message data as an example):

```
{
    "topic_id": "xxxx-xx-xx-xx-yyyyyyyy",
    "topic_name": "testname",
    "records": [{
        "timestamp": "1605578090000000",
        "content": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
    }, {
        "timestamp": "1605578090000000",
        "content": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
    }]
}
```

The data structures are as detailed below:

Structure	Description
topic_id	Log topic ID
topic_name	Log topic name
timestamp	Log production time (timestamp at the microsecond level)
content	Log content

Usage

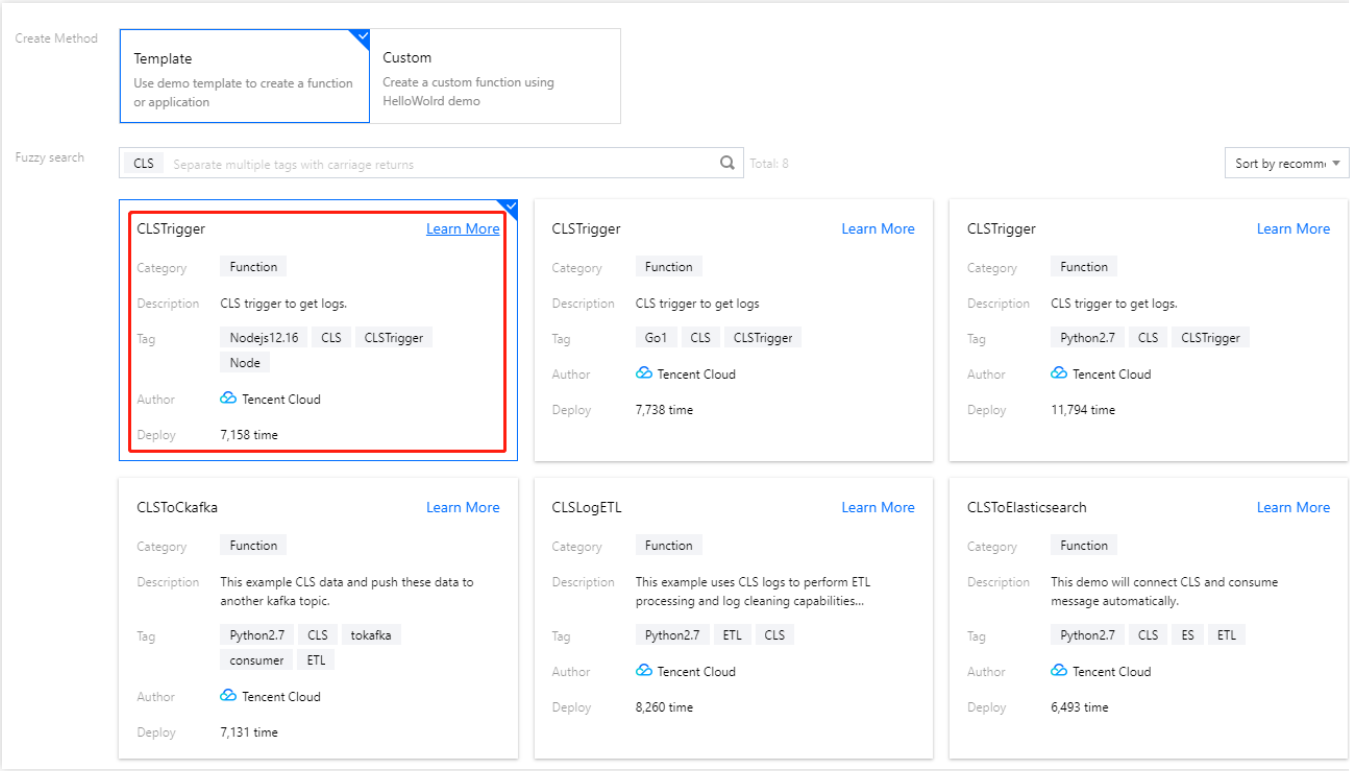
Last updated：2024-12-02 19:58:17

This document describes how to create a CLS trigger and invoke a function.

Step 1. Create a function

Log in to the SCF console and upload and deploy your function code on the **Create** page. For more information, please see Creating Functions in Console.

The following takes the CLS sample template as an example to create a function project. In the default creation process with the template, a trigger is directly configured. In actual use cases, you can also configure a trigger after creating the function. Here, trigger configuration after function creation is used as an example for description:



Step 2. Configure a trigger

After selecting **CLS Trigger**, configure the logset, log topic, and other information as prompted to create a trigger:

Step 3. Manage the trigger

After successful creation, you can see the information of the created trigger on the **Trigger Management** page, where you can enable/disable the trigger.

Timer Trigger
Timer Trigger Description

Last updated：2024-12-02 19:58:17

You can write an SCF function to handle a scheduled task (which can be triggered in seconds). The timer will automatically trigger the function at the specified time. Timer triggers have the following characteristics:
Push model: the timer directly calls the `Invoke` API of the function to trigger it at the specified time. The event source mapping is retained in the SCF function.
Async invocation: a timer trigger always invokes a function asynchronously, and the result is not returned to the invoker. For more information on invocation types, please see Invocation Types.

Timer Trigger Attributes

Timer name (required): it can contain up to 60 characters out of `a-z`, `A-Z`, `0-9`, `-`, and `_` and must begin with a letter and be unique under the same function.
Triggering cycle (required): this is the specified function triggering time. You can use the default value in the console or customize a standard cron expression to decide when to trigger the function. For more information on cron expressions, please see below.
Input parameter (optional): it can be a string of up to 4 KB, which can be obtained from the `event` parameter of the entry function.

Cron Expression

When creating a timer trigger, you can customize the triggering time by using a standard cron expression. Timer triggers can trigger functions in a matter of seconds. In order to be compatible with legacy timer triggers, cron expressions can be written in two ways:

Cron expression syntax 1 (recommended)

A cron expression has seven required fields, separated by spaces.

First	Second	Third	Fourth	Fifth	Sixth	Seventh
Second	Minute	Hour	Day	Month	Week	Year

Each field has a corresponding value range:

Field	Value	Wildcards
Second	An integer between 0 and 59	, - * /
Minute	An integer between 0 and 59	, - * /
Hours	An integer between 0 and 23	, - * /
Day	An integer between 1 and 31 (the number of days in the month needs to be considered)	, - * /
Month	An integer between 1 and 12 or JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC	, - * /
Week	An integer between 0 and 6 or SUN, MON, TUE, WED, THU, FRI, SAT; where 0 means Sunday, 1 means Monday, and so on	, - * /
Year	An integer between 1970 and 2099	, - * /

Cron expression syntax 2 (not recommended)

A cron expression has five required fields, separated by spaces.

First	Second	Third	Fourth	Fifth
Minute	Hour	Day	Month	Week

Each field has a corresponding value range:

Field	Value	Wildcards
Minute	An integer between 0 and 59	, - * /
Hours	An integer between 0 and 23	, - * /
Day	An integer between 1 and 31 (the number of days in the month needs to be considered)	, - * /
Month	An integer between 1 and 12 or JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC	, - * /
Week	An integer between 0 and 6 or SUN, MON, TUE, WED, THU, FRI, SAT; where 0 means Sunday, 1 means Monday, and so on	, - * /

Wildcards

Wildcard	Description

, (comma)	It represents the union of characters separated by commas; for example, 1, 2, 3 in the "Hour" field means 1:00, 2:00 and 3:00
- (hyphen)	It contains all values in the specified range; for example, in the "Day" field, 1-15 contains the 1st to the 15th day of the specified month
* (asterisk)	It means all values; for example, in the "Hour" field, * means every o'clock
/ (forward slash)	It specifies the increment; for example, in the "Minute" field, you can enter 1/10 to specify repeating every ten minutes from the first minute on (e.g., at the 11th minute, the 21st minute, the 31st minute, and so on)

Precautions

When both the "Day" and "Week" fields in a cron expression are specified, they are in an "or" relationship, i.e., the conditions of both are effective separately.

Sample

Below are some examples of cron expressions and their meanings:

Expression	Description
`*/5 * * * * * *`	Triggers once every 5 seconds
`0 15 10 1 * * *`	Triggers at 10:15 am on the 1st day of every month
`0 15 10 * * MON-FRI *`	Triggers every day at 10:15 am Monday through Friday
`0 0 10,14,16 * * * *`	Triggers every day at 10 am, 2 pm, and 4 pm
`0 */30 9-17 * * * *`	Triggers every half hour from 9 am to 5 pm every day
`0 0 12 * * WED *`	Triggers at 12:00 noon every Wednesday

Input Parameters of Timer Triggers

When a timer trigger triggers a function, the following data structures will be encapsulated in `event` and passed to the function. In addition, you can specify to pass the `message` for a timer trigger, which is empty by default.

```
{
    "Type":"Timer",
    "TriggerName":"EveryDay",
    "Time":"2019-02-21T11:49:00Z",
```

```
    "Message":"user define msg body"
}
```

Field	Description
Type	Type of the trigger, whose value is `Timer`
TriggerName	Timer name, which can contain up to 60 characters out of `a-z`, `A-Z`, `0-9`, `-`, and `_` and must begin with a letter and be unique under the same function
Time	Trigger creation time, in UTC+0
Message	String type

Usage

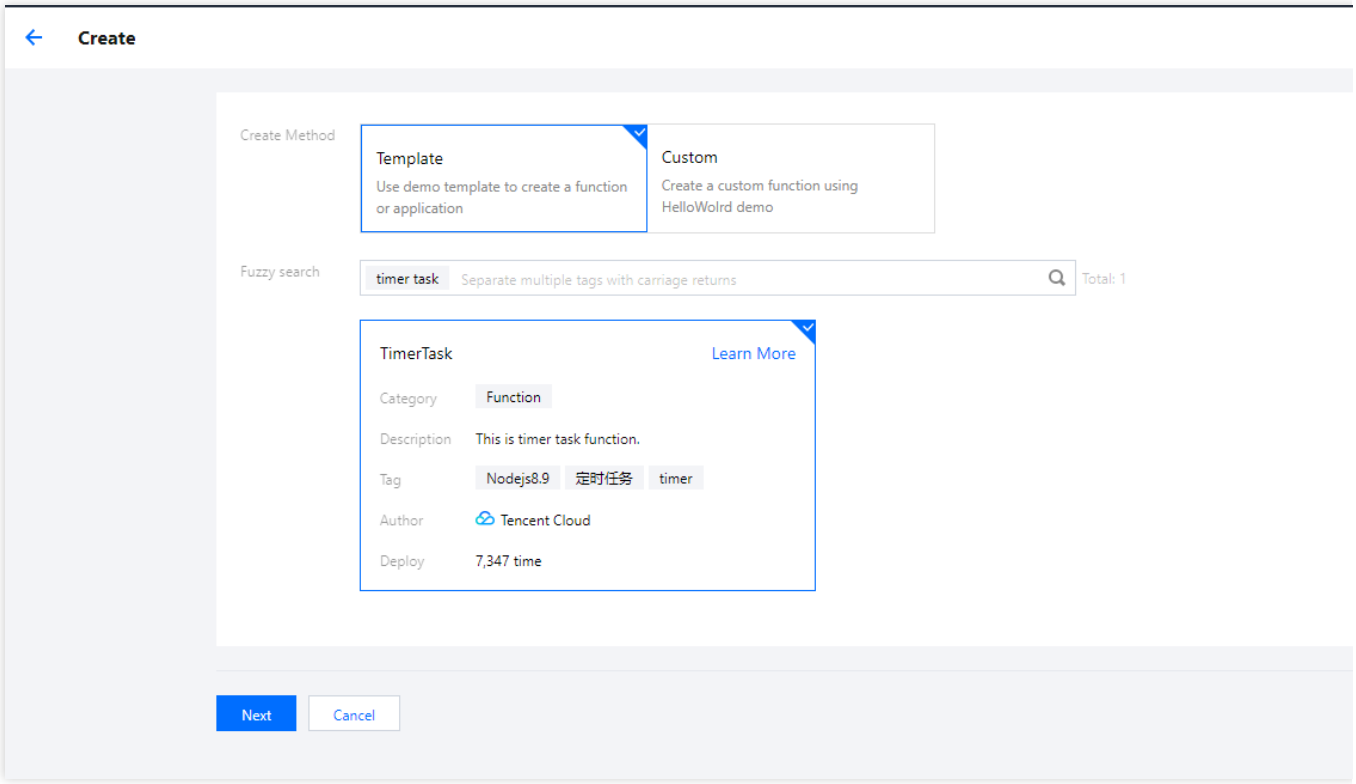
Last updated：2024-12-02 19:58:17

This document describes how to create a timer trigger and invoke a function.

Step 1. Create a function

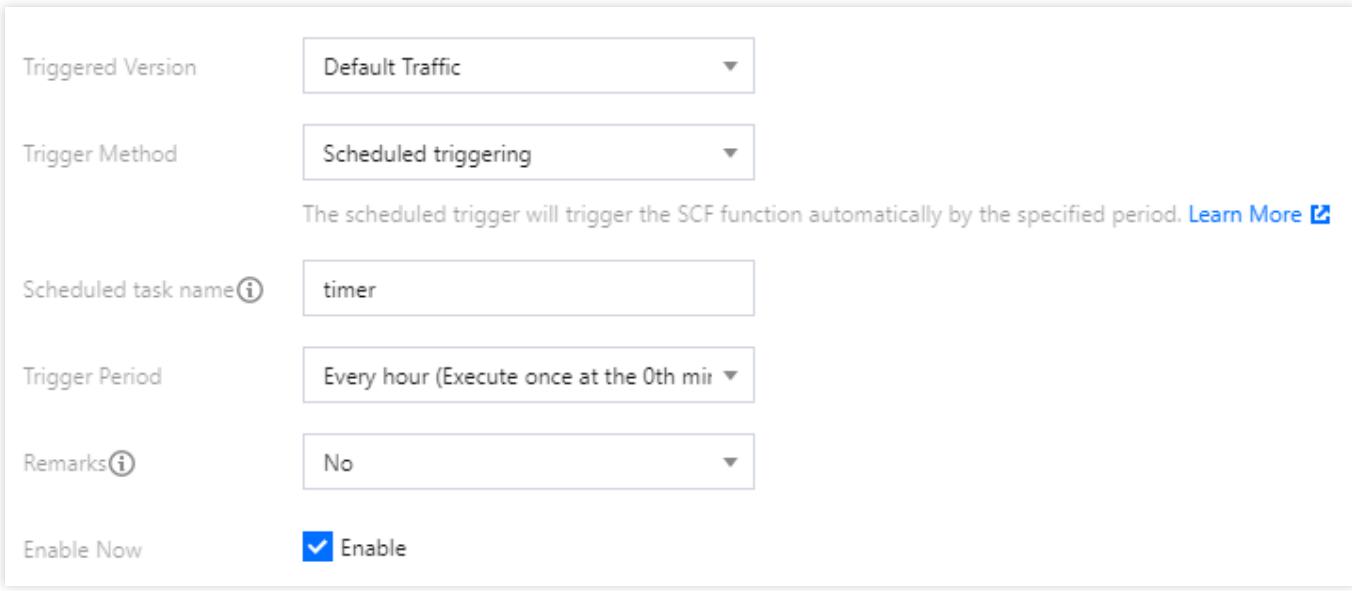
Log in to the SCF console and upload and deploy your function code on the **Create** page. For more information, please see Creating Functions in Console.

The following takes the scheduled task sample template as an example to create a function project. In the default creation process with the template, a trigger is directly configured. In actual use cases, you can also configure a trigger after creating the function. Here, configuration after function creation is used as an example for description:



Step 2. Configure a trigger

After selecting **Timer Trigger**, configure the task name, trigger period, and other information as prompted to create a trigger:



Step 3. Manage the trigger

After successful creation, you can see the information of the created trigger on the **Trigger Management** page, where you can enable/disable the trigger.

CKafka Trigger
CKafka Trigger Description

Last updated：2024-12-02 19:58:17

You can write an SCF function to process messages received in the specific CKafka instance. The SCF backend can consume the messages in CKafka as a consumer and pass them to the function.
Characteristics of CKafka triggers:
Pull model: the backend module of SCF acts as a consumer, connects to the CKafka instance, and consumes messages. When the backend module gets the message, it will encapsulate the message into data structures and invoke the specified function to pass the message data to the function.
Sync invocation: a CKafka trigger always invokes a function synchronously. For more information on invocation types, please see Invocation Types.
Note:
For execution errors (including user code errors and runtime errors), the CKafka trigger will retry according to the configured retry times, which is 10,000 by default.
For system errors, the CKafka trigger will continue to retry in an exponential backoff manner until it succeeds.

CKafka Trigger Attributes

CKafka instance: configure the CKafka instance you want to connect to. It can only be an instance in the same region as the function.
Topic: it can be an existing topic in the CKafka instance (only topics without ACL are supported).
Maximum messages: the maximum number of messages that can be pulled and batch delivered to the current function at a time, which can be up to 10,000 currently. According to the message size and writing speed, the number of messages delivered when the function is triggered each time may not always reach the maximum number; instead, it is a variable value between 1 and the maximum number.
Start Point: the start position from which the trigger consumes messages. Valid values: latest (default), oldest, specified time point.
Retry Attempts: the maximum number of retries when an error occurs during function execution (including user code errors and runtime errors).

CKafka Consumption and Message Delivery

CKafka does not push messages actively. The consumer needs to pull messages and consume them. Therefore, if a CKafka trigger is configured, the SCF backend will launch a CKafka consumption module as the consumer to create

an independent consumer group in CKafka for message consumption.
After consuming messages, the SCF backend consumption module will encapsulate them into event structures according to the **timeout period**, **accumulated messages**, and **maximum messages** and then initiate function invocation (sync invocation). Applicable limits are as follows:
Timeout period: the current timeout period of the consumption module on the backend of SCF is 60 seconds, which avoids waiting for too long before consuming. For example, if the CKafka topic has very few messages written in, and the consumption module fails to collect the configured maximum number of messages in 60 seconds, then the function invocation will still be initiated.
Event size limit for sync invocation: 6 MB. For more information, please see Limits. If the messages in the CKafka topic are large (for example, one single message is over 6 MB in size), then due to the 6 MB limit for sync invocation, there will be only one message in the event structure passed to the function instead of the user-configured maximum number of messages.
Maximum messages: this is the same as the user-defined CKafka trigger attribute, which can be up to 10,000 currently.
The consumption module on the backend of SCF will loop this process and ensure the order of message consumption, that is, the next batch of messages will be consumed only after the previous batch is completely consumed (sync invocation).
Note:
In this process, the number of encapsulated messages is different in each event structure, which ranges from **1 to the maximum number**. If the maximum number of messages is too high, there may be cases where the number of messages in an event structure will never reach the maximum number.
After the event content is obtained by the function, each message can be guaranteed for processing by loop handling, and it should not be assumed that the number of messages passed each time is constant.

Event Message Structure for CKafka Trigger

When the specified CKafka topic receives a message, the backend consumption module of SCF will consume the message and encapsulate it into an event in JSON format like the one below, which will trigger the bound function and pass the data content as input parameters to the function.

```
{
  "Records": [
    {
      "Ckafka": {
        "topic": "test-topic",
        "Partition":1,
        "offset":36,
        "msgKey": "None",
        "msgBody": "Hello from Ckafka!"
```

```
        }
      },
      {
        "Ckafka": {
          "topic": "test-topic",
          "Partition":1,
          "offset":37,
          "msgKey": "None",
          "msgBody": "Hello from Ckafka again!"
        }
      }
    ]
}
```

The data structures are as detailed below:

Structure	Description
Records	List structure. There may be multiple messages merged in the list
Ckafka	Identifies the event source as CKafka
topic	Message source topic
partition	Partition ID of message source
offset	Consumption offset number
msgKey	Message key
msgBody	Message content

FAQs

What should I do if a lot of CKafka messages heap up?

If a CKafka trigger is configured, the SCF backend will launch a CKafka consumption module as the consumer to create an independent consumer group in CKafka for message consumption. In addition, the number of consumption modules is equal to the number of partitions in the CKafka topic.

If a lot of CKafka messages heap up, you need to increase the consumption capability in the following ways:

Increase the number of partitions of the CKafka topic. The consumption capability of the function is proportional to the number of partitions. The CKafka consumption modules on the backend of the function will automatically match the number of CKafka topic partitions, that is, the consumption capability can be improved by adding partitions.

Optimize the execution duration of the function. The shorter the duration, the higher the consumption capability. If the duration becomes longer (for example, the database in the function needs to be written but the response of the database becomes slower), the consumption speed will decrease.

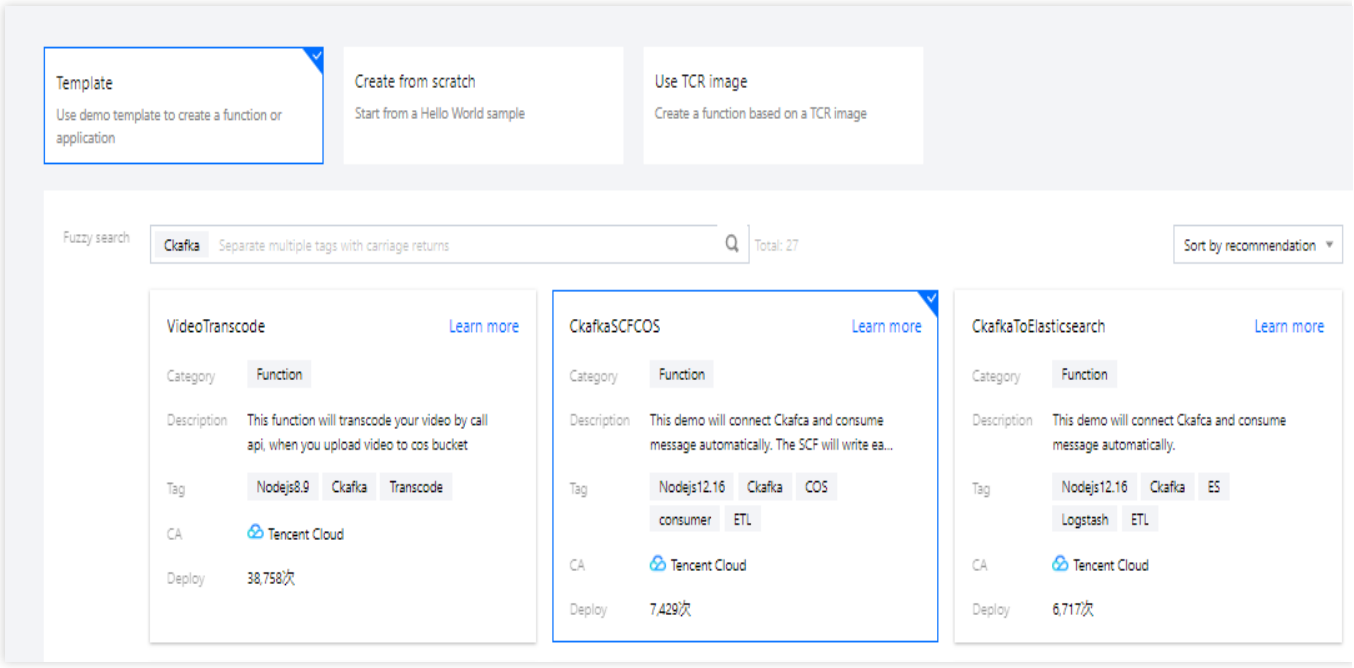
Usage

Last updated：2024-12-02 19:58:17

This document describes how to create a CKafka trigger and invoke a function.

Step 1. Create a function

1. Log in to the SCF console.
2. On the function creation page, select **Template**, search for `CKafka`, and select **CkafkaSCFCOS** as shown below:

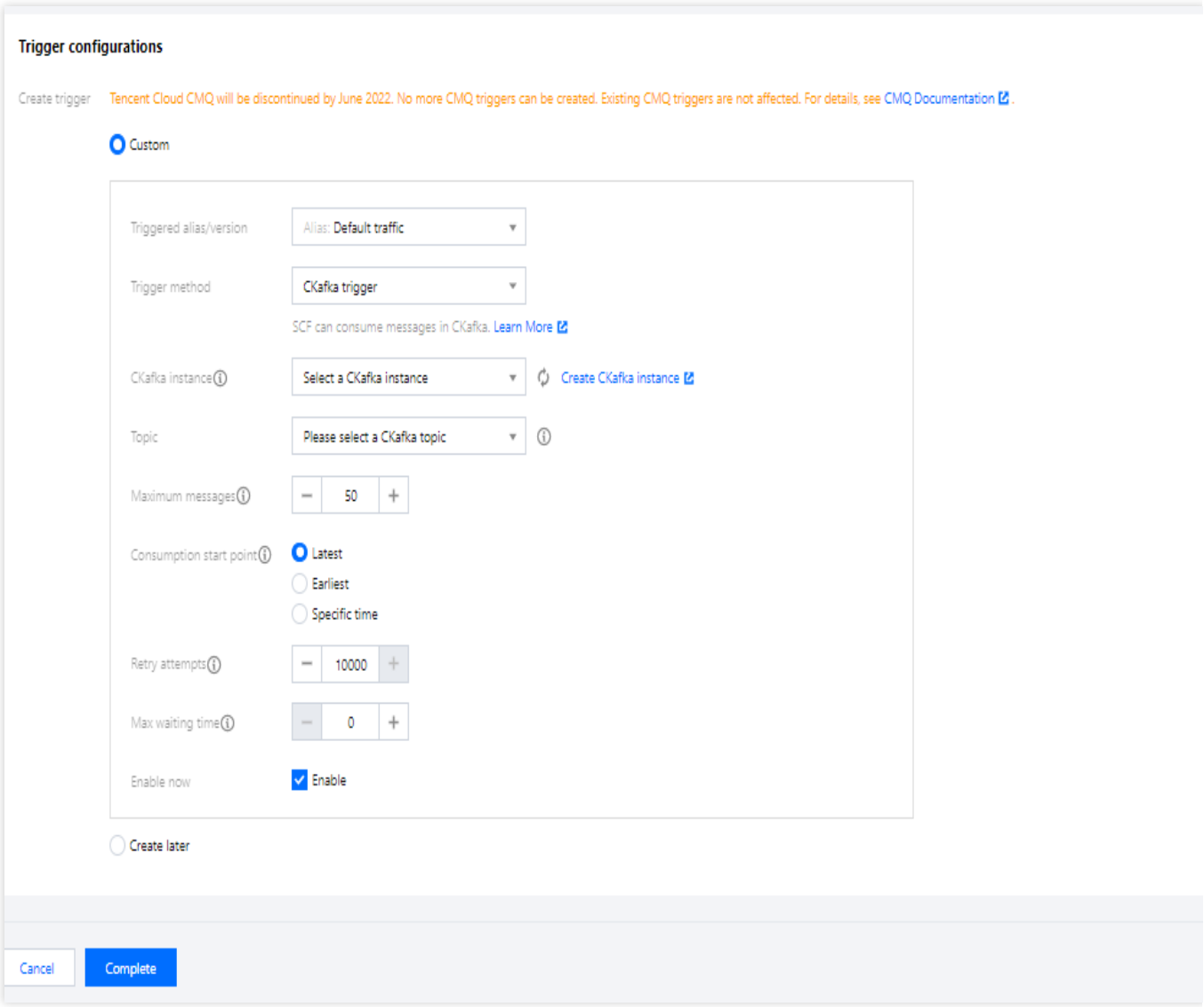


3. Click **Next**.

Step 2. Configure a trigger

On the **Function configuration** page, set the basic configurations of the function.

1. In **Trigger configurations**, select **Custom** to create a trigger as shown below:



Select **CKafka trigger** and configure the name, topic, and other information of the CKafka instance as the message source as prompted. You can also choose to create a CKafka instance.

Note:

Make sure that your function and CKafka instance are in the same VPC.

2. Click **Complete**.

Step 3. Manage the trigger

After the function is created, go to the function details page, and you can see the created trigger in **Trigger management**. You can also turn triggers on or off there.

Apache Kafka Trigger
Apache Kafka Trigger Description

Last updated：2024-12-25 10:34:04

Serverless Cloud Function (SCF) now supports Apache Kafka as the event trigger source, enabling batch consumption and processing of Kafka messages.

Apache Kafka Overview

Apache Kafka is an open-source event streaming platform designed for workloads such as data pipelines and stream processing. SCF enables integration of business functions with self-built Apache Kafka clusters. Supported Kafka clusters include cross-region CKafka clusters, Kafka clusters hosted on other cloud providers, or Kafka-like clusters managed through solutions such as Confluent Cloud, including Azure EventHub.
SCF supports event sources based on the Kafka protocol framework, enabling batch consumption. Batch processing behavior can be controlled using parameters such as the maximum number of messages per batch, maximum waiting time, and retry attempts.

Features of Self-Built Apache Kafka Triggers

Self-built Apache Kafka triggers have the following features:
Pull model: The SCF backend module acts as a consumer, connecting to the Kafka instance to consume messages. Once the backend module retrieves messages, it encapsulates them into a data structure and calls the specified function, passing the message data to the SCF.
Synchronous call: Self-built Apache Kafka triggers use the synchronous call type to call functions. For more information on call types, see Call Types.
Note:
For execution errors, including user code and runtime environment errors, self-built Apache Kafka triggers will retry based on the configured retry attempts, with a default of 10,000 retries.
For system errors, self-built Apache Kafka triggers use an exponential backoff strategy to continuously retry until the operation succeeds.

Attributes of Self-Built Apache Kafka Triggers

Trigger name: It should contain 2 to 60 characters, consisting of `a-z`, `A-Z`, `0-9`, `-`, and `_`. It should start with a letter and end with a letter or number. Multiple triggers with the same name are not allowed for one function.
Bootstrap Servers: It configures the connection addresses for the self-built Apache Kafka instances to be consumed. Multiple bootstrap servers are supported in the format of either `IP+port` or `Domain Name+port`.
Topic: Enter the topic of the existing Apache Kafka instance.
Consumer Group: Select the consumer group of the existing Apache Kafka instance. If the specified consumer group does not exist, one will be automatically created. It is recommended to use a dedicated consumer group, separate from existing businesses, to avoid interfering with ongoing message consumption.
Security protocol: The security protocol used by the Apache Kafka instance. Currently supported protocols include `PLAINTEXT`, `SASL_SSL`, and `SASL_PLAINTEXT`.
Identity verification mechanism: The authentication mechanism used by the Apache Kafka instance. Currently supported options include `None`, `PLAIN`, `SCRAM-SHA-256`, and `SCRAM-SHA-512`. If your instance does not require authentication, select `None`.
Username and Password: If an authentication mechanism is selected, you should provide the username and password authorized to access the instance.
Maximum messages: The maximum number of messages to be pulled and delivered to SCF in a single batch, with a current maximum configuration of 10,000. Due to factors such as message size and write speed, the actual number of messages delivered during each trigger may not always reach the maximum value, but will vary between 1 and the specified maximum batch size.
Consumption start point: The starting point for message consumption by the trigger. Currently, it supports consuming messages from the latest position.
Retry attempts: The maximum number of retries when the function encounters execution errors (including user code errors and runtime errors).
Max waiting time: The longest waiting time for one trigger. For example, if the user configures the maximum batch size as 1,000 messages and the maximum waiting time as 60 seconds, the function will be triggered if 1,000 messages are collected within 10 seconds. If only 50 messages are collected after 60 seconds, the function will still be triggered.
Note:
Currently, for existing self-built Apache Kafka triggers, only the following three configuration items can be edited: Maximum messages, Retry attempts, and Max waiting time.

Self-Built Apache Kafka Consumption and Message Delivery

Since self-built Apache Kafka messages do not have push capabilities, the consumer should pull the messages for consumption. Therefore, after the self-built Apache Kafka trigger is configured, the SCF backend will activate the self-

built Apache Kafka consumer module to act as the consumer. It will also create an independent consumer group within the self-built Apache Kafka for message consumption.
After the SCF backend consumer module consumes the messages, it will combine information such as the **Timeout**, **Accumulated message size**, and **Maximum messages** to form an event structure and initiate a function call (synchronous call). The related limitations are as follows:
Timeout: The current timeout for the SCF backend consumer module is 60 seconds, to avoid delays before consumption. For example, if there are few messages written to the topic and the consumer module does not accumulate enough messages to reach the maximum batch size within 60 seconds, it will still initiate the function call.
Event Size Limit for Synchronous Calls: 6 MB. For details, see Quota Limits. If the messages in the topic are large, for example, if a message already reaches 6 MB, due to the 6 MB limit for synchronous calls, the event structure passed to the SCF will contain only one message instead of the maximum number of messages configured by the user.
Maximum Batch Size: This attribute is the same as the one in the self-built Apache Kafka trigger and is set by the user. The current maximum supported configuration is 10,000.
The SCF backend consumer module will loop through this process and ensure the order of message consumption. This means that the next batch of messages will not be consumed until the previous batch has been fully processed (synchronous call).
Note:
During this process, the number of messages in each batch may vary, meaning the number of messages in each event structure will be between **1 and the configured maximum batch size**. If the configured maximum batch size is set too high, it is possible that the number of messages in the event structure will never reach the maximum batch size.
After receiving the event content in SCF, you can choose to process the messages in a loop to ensure that each message is handled. You should not assume that the number of messages passed in each event is constant.
SCF will use the standard Kafka protocol to retrieve the number of partitions for the specified topic. The backend consumer module will automatically create the same number of consumers. If the partition count cannot be obtained, 20 consumers will be created by default.

FAQs

How to Handle a Large Accumulation of Messages in a Self-Built Apache Kafka Instance?

After you configure the self-built Apache Kafka trigger, the SCF backend will activate the consumer module as the consumer, creating an independent consumer group in the self-built Apache Kafka for message consumption. The number of consumer modules will equal the number of partitions in the topic. If there is a large accumulation of messages, the consumption capacity needs to be increased. The following methods can be used to enhance consumption capacity:

Optimize the execution time of the SCF. The shorter the execution time of the SCF, the higher the consumption capacity. If the execution time increases (for example, if the SCF needs to write to a database and the database response slows down), the consumption speed will decrease.

Usage

Last updated：2024-12-25 10:34:25

This document provides a guide on creating a self-built Apache Kafka trigger and calling the function, using the example of cross-region consumption of a CKafka cluster.

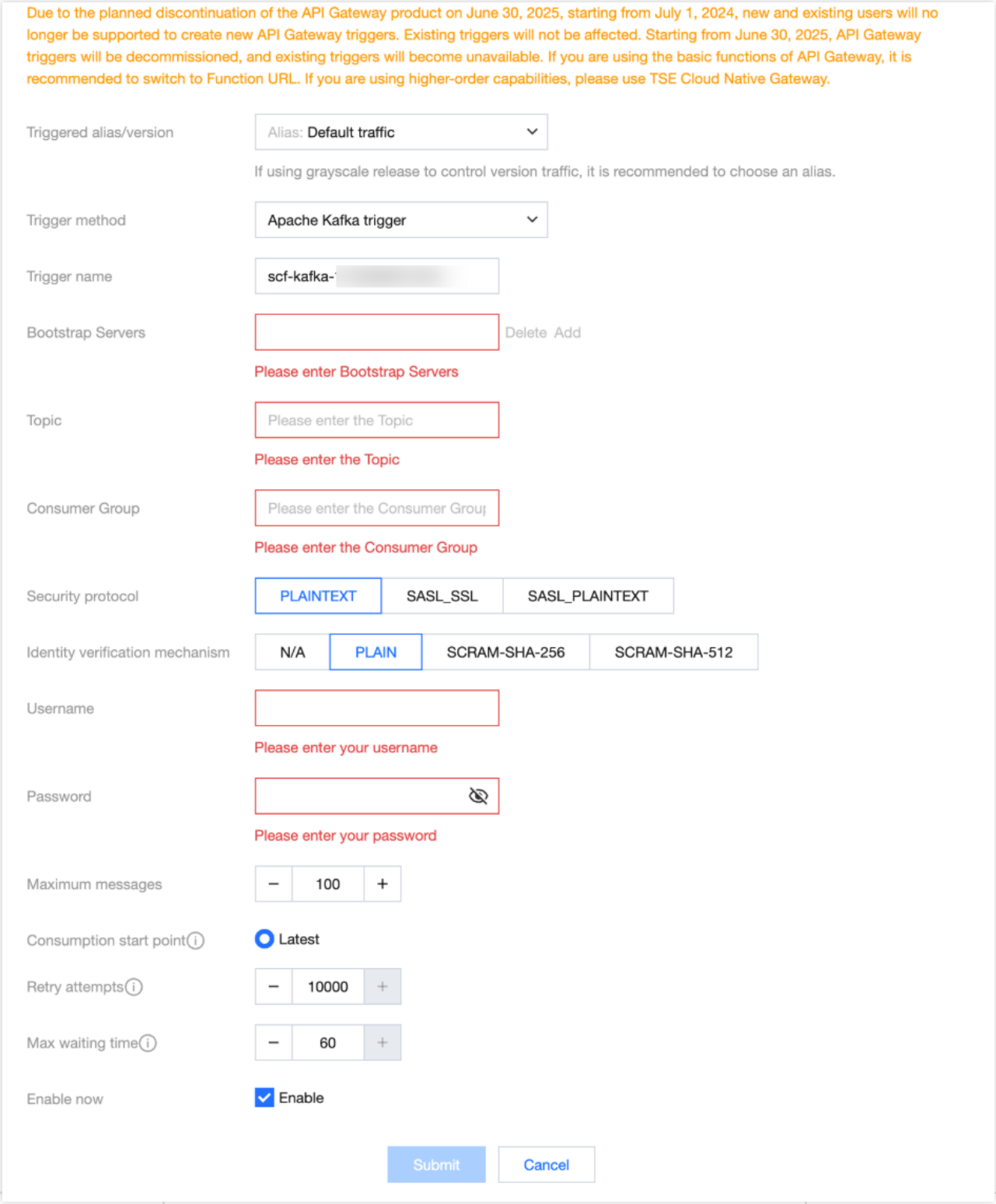
Prerequisites

The function has been created.
The Kafka cluster and topic have been created.

Directions

Step 1: Creating an Apache Kafka Trigger

1. Log in to the Serverless Console and click **Function Service** in the left sidebar.
2. At the top of the main interface, select the region and namespace where the function is located. Then, click the function name in the list to access the function details page.
3. In the left sidebar, choose **Trigger Management** and click **Create trigger**.
4. In the Create Trigger panel, select **Apache Kafka Trigger** and fill in the relevant trigger information, as shown below:



Configuration Item	Operation	Example in this Document
Trigger version/alias	The default value is Default, but it can also be switched to an alias or other versions of the published function.	Default
Trigger method	Apache Kafka trigger.	Apache Kafka trigger

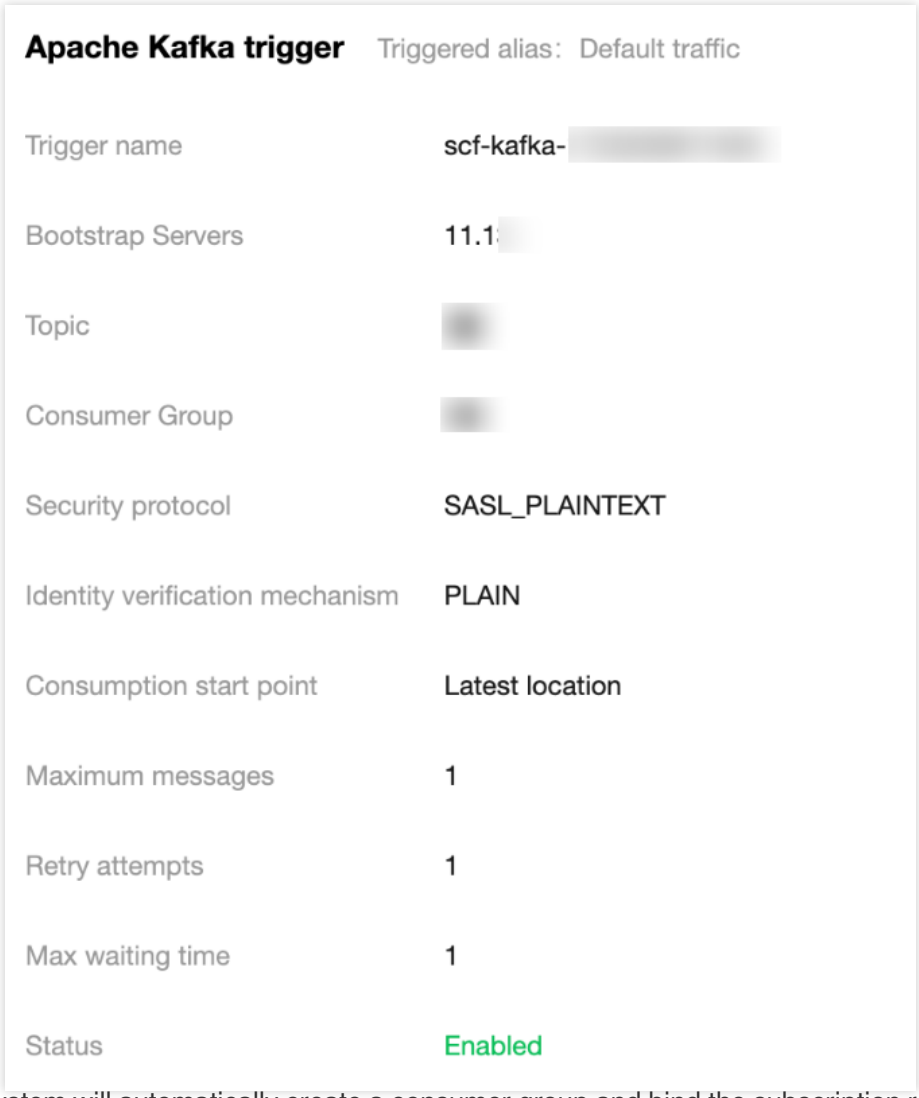
Trigger name	Enter a custom trigger name.	scf-kafka-1728981649432
Bootstrap Servers	Enter the host and port addresses of the Kafka instance to be accessed. Multiple entries are allowed.	11.135.x.x:7661
Topic	Select the topic of the existing Kafka instance.	test1015
Consumer Group	The desired consumer group name. If the consumer group already exists under the topic, it will continue consuming from that group; otherwise, a consumer group will be created.	A new consumer group, test1015, is used here.
Security protocol	The security protocol applied by the Kafka instance. The available values are as follows: PLAINTEXT SASL_SSL SASL_PLAINTEXT	SASL_PLAINTEXT
Identity verification mechanism	The authentication mechanism used by the Kafka instance. The available values are as follows: None PLAIN SCRAM-SHA-256 SCRAM-SHA-512	PLAIN
Username	When authentication in the mechanism requires verification through username and password information, you need to configure the Apache Kafka username for authentication.	admin
Password	When the authentication mechanism requires username and password verification, the Apache Kafka username should be configured for authentication.	******
Maximum messages	The maximum number of messages to be pulled and delivered to SCF in a single batch, with a current maximum configuration of 10,000. Due to factors such as message size and write speed, the actual number of messages delivered during each trigger may not always reach the maximum value, but will vary between 1 and the specified maximum batch size.	1
Consumption start point	Select the consumption offset for messages. Currently, it supports consuming from the latest position.	Latest position
Retry attempts	The maximum number of retries when the function encounters execution errors (including user code errors and execution errors). The maximum supported configuration is 10,000.	1

Max waiting time	The longest waiting time for one trigger. For example, if the user configures the maximum batch size as 1,000 messages and the maximum waiting time as 60 seconds, the function will be triggered if 1,000 messages are collected within 10 seconds. If only 50 messages are collected after 60 seconds, the function will still be triggered.	1
Trigger status	Indicates whether the trigger is enabled immediately after creation. By default, the trigger is enabled, meaning it will be activated as soon as it is created.	Enable the trigger

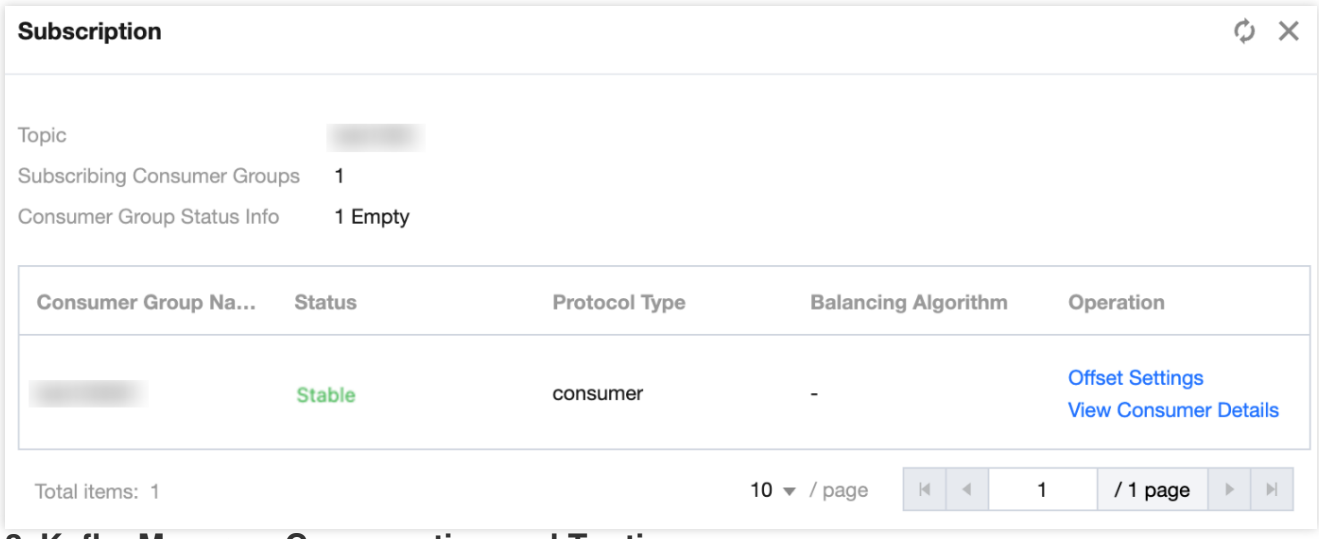
5. Click **Submit**.

Step 2: Checking the Trigger Status

After the trigger is successfully created, go to the Trigger Management page to check the trigger status, as shown below:

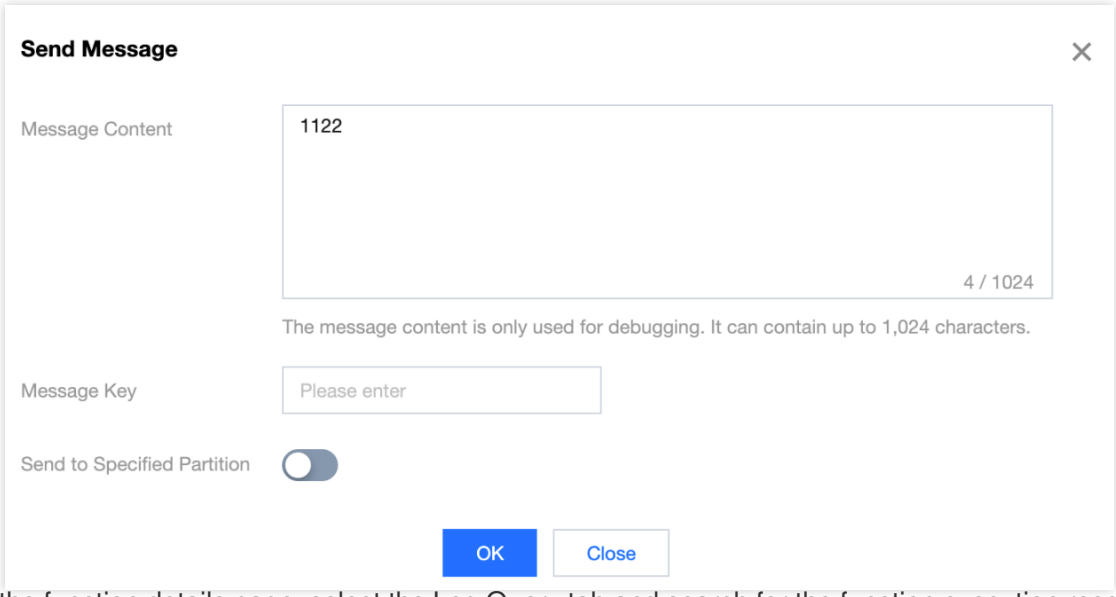


The system will automatically create a consumer group and bind the subscription relationship. As shown below:

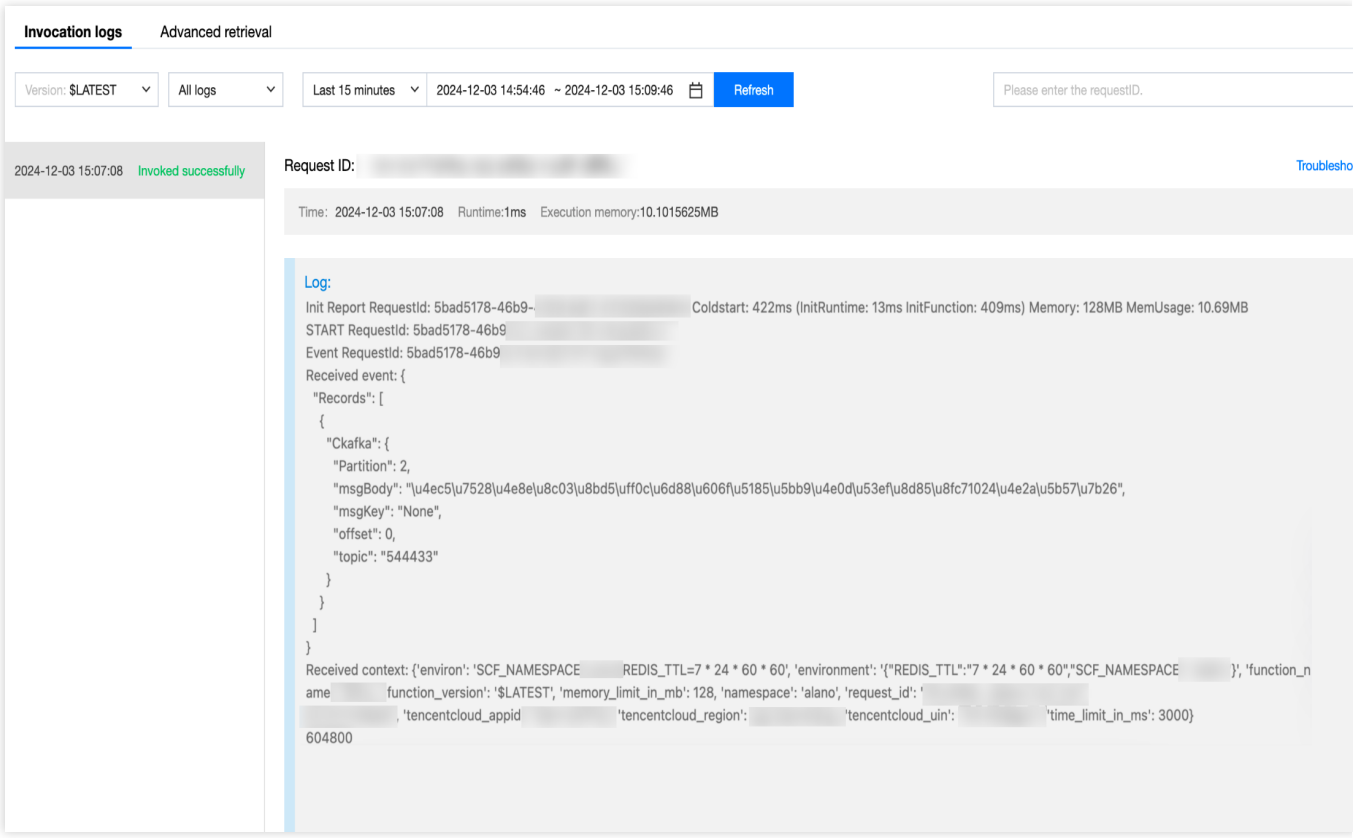


Step 3: Kafka Message Consumption and Testing

1. Deliver messages to Kafka, as shown below:



2. On the function details page, select the Log Query tab and search for the function execution records. You will then be able to see that the function has been successfully consumed and triggered.



Trigger Configuration Description

Last updated：2024-12-02 19:58:17

When you call the trigger API CreateTrigger, the corresponding `TriggerDesc` parameter will be the trigger description, which can be used as instructed in this document.

Timer Trigger

Please directly enter a cron expression for this parameter. For more information, please see Timer Trigger Description.

Sample TriggerDesc

Triggered once every five minutes:

```
0 */5 * * * * *
```

API Gateway Trigger

Name	Type	Required	Description
api	ApigwApi	No	API configuration of the created API gateway
service	ApigwService	No	Service configuration of the created API gateway
release	ApigwRelease	No	Release environment for the created API gateway

ApigwApi

Name	Type	Required	Description
authRequired	String	No	Whether authentication is required. Valid values: TRUE, FALSE. Default value: FALSE
requestConfig	ApigwApiRequestConfig	No	Configuration of request backend API
isIntegratedResponse	String	No	Whether to use integrated response. Valid values: TRUE, FALSE. Default value: FALSE

IsBase64Encoded	String	No	Whether to enable Base64-encoding. Valid values: TRUE, FALSE. Default value: FALSE

ApigwApiRequestConfig

Name	Type	Required	Description
method	String	No	Method configuration of request backend API. Valid values: `ANY`, `GET`, `HEAD`, `POST`, `PUT`, `DELETE`

ApigwService

Name	Type	Required	Description
serviceId	String	No	Apigw Service ID (if this parameter is not passed in, a new service will be created)

ApigwRelease

Name	Type	Required	Description
environmentName	String	Yes	Release environment. Valid values: `release`, `test`, `prepub`. If this parameter is left empty, `release` will be used by default

Sample TriggerDesc

```
{
  "api":{
    "authRequired":"FALSE",
    "requestConfig":{
      "method":"ANY"
    },
    "isIntegratedResponse":"FALSE"
  },
  "service":{
    "serviceName":"SCF_API_SERVICE"
  },
  "release":{
    "environmentName":"release"
  }
}
```

CKafka Trigger

Name	Type	Required	Description
maxMsgNum	String	Yes	A function invocation will be triggered once every time `maxMsgNum` CKafka messages are aggregated within 5 seconds
offset	String	Yes	`offset` is the position where consumption of CKafka messages starts. Currently, three values are supported: `latest` , `earliest` , and `millisecond-level timestamp`
retry	String	Yes	Maximum number of retries when the function reports an error

Sample TriggerDesc

```
{"maxMsgNum":100,"offset":"latest","retry":10000}
```

```
{"maxMsgNum":999,"offset":"1595927203000","retry":10}
```

API request description

To create a CKafka trigger by using an API request, the `TriggerName` field needs to be defined as the `instanceId` and `topicName` of the target CKafka instance in the following format: `[instanceId]-[topicName]` . Below is a sample request:

```
TriggerName: "ckafka-8tfxzia3-test"
```

COS Trigger

Name	Type	Required	Description
event	String	Yes	COS event type
filter	CosFilter	Yes	COS filename filter

CosFilter

Name	Type	Required	Description

Prefix	String	No	Prefix rule of file filter
Suffix	String	No	Suffix rule of file filter, which must begin with `.`

COS event conflict rules

Core concept: an event can trigger a function invocation at most once. If an event is bound to another product, it cannot be bound to a function.
Set at most one prefix filter and one suffix filter.
If the `cos:ObjectCreated:*` event is set but no prefix/suffix is set, subsequent binding to any event that starts with `cos:ObjectCreated` will fail.
The filter will be valid only if both the prefix and suffix are matched, and if there are conflicts with both the prefix and suffix, subsequent binding will fail.

Sample TriggerDesc

```
{"event":"cos:ObjectCreated:*","filter":{"Prefix":"","Suffix":""}}
```

Note:
When `TriggerDesc` is used as a trigger description, the JSON string must be continuous with no spaces contained.

API request description

To create a COS trigger by using an API request, the `TriggerName` field needs to be defined as the XML API access domain name of the target COS bucket. Below is an example:

```
TriggerName: "xxx.cos.ap-guangzhou.myqcloud.com"
```

Note:
The access domain name can be viewed in **Bucket List** > **Basic Configuration** > **Basic Information** in the COS console.

CMQ Trigger

Name	Type	Required	Description
filterType	String	No	Message filter type. 1: tag; 2: route match
filterKey	String	No	When `filterType` is `1`, it indicates the message filter tag; when `filterType` is `2`, it indicates the `Binding Key`

Sample TriggerDesc

```
{"filterType":1,"filterKey":["test"]}
```

```
{"filterType":2,"filterKey":["#test"]}
```

API request description

To create a CMQ trigger by using an API request, the `TriggerName` field needs to be defined as `CMQ Topic`. Below is an example:

```
TriggerName: "Tabortest"
```

MPS Trigger

Last updated：2024-12-02 19:58:17

Media Processing Service (MPS) is an on-cloud transcoding and audio/video processing service for massive amounts of multimedia data. You can write functions to process the callback information in MPS and dump, publish, and process relevant events and subsequent content in video tasks by receiving relevant callbacks.

Characteristics of MPS triggers:

Push model: an MPS trigger listens on the callback information of video processing and pushes the event data to the SCF function through one single triggering action.

Async invocation: an MPS trigger always invokes a function asynchronously, and the result is not returned to the invoker. For more information on invocation types, please see Invocation Types.

MPS Trigger Attributes

Event type: an MPS trigger pushes events in the account-level event type. Currently, two event types are supported: workflow task (`WorkflowTask`) and video editing task (`EditMediaTask`).

Event processing: an MPS trigger uses events generated at the service level as the event source, regardless of attributes such as region and resources. Each account can only create one MPS trigger in all regions. If you need multiple functions to handle a task, please see SDK for Node.js.

Event Structure for MPS Trigger

When a specified MPS trigger receives a message, the event structure and fields will be as shown below (with the `WorkflowTask` task as an example):

```
{
    "EventType":"WorkflowTask",
    "WorkflowTaskEvent":{
        "TaskId":"245****654-WorkflowTask-f46dac7fe2436c47******d71946986t0",
        "Status":"FINISH",
        "ErrCode":0,
        "Message":"",
        "InputInfo":{
            "Type":"COS",
            "CosInputInfo":{
                "Bucket":"macgzptest-125****654",
                "Region":"ap-guangzhou",
                "Object":"/dianping2.mp4"
```

```
            }
        },
        "MetaData":{
            "AudioDuration":11.261677742004395,
            "AudioStreamSet":[
                {
                    "Bitrate":127771,
                    "Codec":"aac",
                    "SamplingRate":44100
                }
            ],
            "Bitrate":2681468,
            "Container":"mov,mp4,m4a,3gp,3g2,mj2",
            "Duration":11.261677742004395,
            "Height":720,
            "Rotate":90,
            "Size":3539987,
            "VideoDuration":10.510889053344727,
            "VideoStreamSet":[
                {
                    "Bitrate":2553697,
                    "Codec":"h264",
                    "Fps":29,
                    "Height":720,
                    "Width":1280
                }
            ],
            "Width":1280
        },
        "MediaProcessResultSet":[
            {
                "Type":"Transcode",
                "TranscodeTask":{
                    "Status":"SUCCESS",
                    "ErrCode":0,
                    "Message":"SUCCESS",
                    "Input":{
                        "Definition":10,
                        "WatermarkSet":[
                            {
                                "Definition":515247,
                                "TextContent":"",
                                "SvgContent":""
                            }
                        ],
                        "OutputStorage":{
                            "Type":"COS",
```



```
                            "CosOutputStorage":{
                                "Bucket":"gztest-125****654",
                                "Region":"ap-guangzhou"
                            }
                        },
                        "OutputObjectPath":"/dasda/dianping2_transcode_10",

"SegmentObjectName":"/dasda/dianping2_transcode_10_{number}",
                        "ObjectNumberFormat":{
                            "InitialValue":0,
                            "Increment":1,
                            "MinLength":1,
                            "PlaceHolder":"0"
                        }
                    },
                    "Output":{
                        "OutputStorage":{
                            "Type":"COS",
                            "CosOutputStorage":{
                                "Bucket":"gztest-125****654",
                                "Region":"ap-guangzhou"
                            }
                        },
                        "Path":"/dasda/dianping2_transcode_10.mp4",
                        "Definition":10,
                        "Bitrate":293022,
                        "Height":320,
                        "Width":180,
                        "Size":401637,
                        "Duration":11.26200008392334,
                        "Container":"mov,mp4,m4a,3gp,3g2,mj2",
                        "Md5":"31dcf904c03d0cd78346a12c25c0acc9",
                        "VideoStreamSet":[
                            {
                                "Bitrate":244608,
                                "Codec":"h264",
                                "Fps":24,
                                "Height":320,
                                "Width":180
                            }
                        ],
                        "AudioStreamSet":[
                            {
                                "Bitrate":48414,
                                "Codec":"aac",
                                "SamplingRate":44100
                            }
```



```
                        ]
                    }
                },
                "AnimatedGraphicTask":null,
                "SnapshotByTimeOffsetTask":null,
                "SampleSnapshotTask":null,
                "ImageSpriteTask":null
            },
            {
                "Type":"AnimatedGraphics",
                "TranscodeTask":null,
                "AnimatedGraphicTask":{
                    "Status":"FAIL",
                    "ErrCode":30010,
                    "Message":"TencentVodPlatErr Or Unkown",
                    "Input":{
                        "Definition":20000,
                        "StartTimeOffset":0,
                        "EndTimeOffset":600,
                        "OutputStorage":{
                            "Type":"COS",
                            "CosOutputStorage":{
                                "Bucket":"gztest-125****654",
                                "Region":"ap-guangzhou"
                            }
                        },

"OutputObjectPath":"/dasda/dianping2_animatedGraphic_20000"
                    },
                    "Output":null
                },
                "SnapshotByTimeOffsetTask":null,
                "SampleSnapshotTask":null,
                "ImageSpriteTask":null
            },
            {
                "Type":"SnapshotByTimeOffset",
                "TranscodeTask":null,
                "AnimatedGraphicTask":null,
                "SnapshotByTimeOffsetTask":{
                    "Status":"SUCCESS",
                    "ErrCode":0,
                    "Message":"SUCCESS",
                    "Input":{
                        "Definition":10,
                        "TimeOffsetSet":[
```



```
                        ],
                        "WatermarkSet":[
                            {
                                "Definition":515247,
                                "TextContent":"",
                                "SvgContent":""
                            }
                        ],
                        "OutputStorage":{
                            "Type":"COS",
                            "CosOutputStorage":{
                                "Bucket":"gztest-125****654",
                                "Region":"ap-guangzhou"
                            }
                        },

"OutputObjectPath":"/dasda/dianping2_snapshotByOffset_10_{number}",
                        "ObjectNumberFormat":{
                            "InitialValue":0,
                            "Increment":1,
                            "MinLength":1,
                            "PlaceHolder":"0"
                        }
                    },
                    "Output":{
                        "Storage":{
                            "Type":"COS",
                            "CosOutputStorage":{
                                "Bucket":"gztest-125****654",
                                "Region":"ap-guangzhou"
                            }
                        },
                        "Definition":0,
                        "PicInfoSet":[
                            {
                                "TimeOffset":0,

"Path":"/dasda/dianping2_snapshotByOffset_10_0.jpg",
                                "WaterMarkDefinition":[
                                    515247
                                ]
                            }
                        ]
                    }
                },
                "SampleSnapshotTask":null,
                "ImageSpriteTask":null
```



```
            },
            {
                "Type":"ImageSprites",
                "TranscodeTask":null,
                "AnimatedGraphicTask":null,
                "SnapshotByTimeOffsetTask":null,
                "SampleSnapshotTask":null,
                "ImageSpriteTask":{
                    "Status":"SUCCESS",
                    "ErrCode":0,
                    "Message":"SUCCESS",
                    "Input":{
                        "Definition":10,
                        "OutputStorage":{
                            "Type":"COS",
                            "CosOutputStorage":{
                                "Bucket":"gztest-125****654",
                                "Region":"ap-guangzhou"
                            }
                        },

"OutputObjectPath":"/dasda/dianping2_imageSprite_10_{number}",
                        "WebVttObjectName":"/dasda/dianping2_imageSprite_10",
                        "ObjectNumberFormat":{
                            "InitialValue":0,
                            "Increment":1,
                            "MinLength":1,
                            "PlaceHolder":"0"
                        }
                    },
                    "Output":{
                        "Storage":{
                            "Type":"COS",
                            "CosOutputStorage":{
                                "Bucket":"gztest-125****654",
                                "Region":"ap-guangzhou"
                            }
                        },
                        "Definition":10,
                        "Height":80,
                        "Width":142,
                        "TotalCount":2,
                        "ImagePathSet":[
                            "/dasda/imageSprite/dianping2_imageSprite_10_0.jpg"
                        ],

"WebVttPath":"/dasda/imageSprite/dianping2_imageSprite_10.vtt"
```

```
                        }
                    }
                }
            ]
        }
}
```

WorkflowTask event

The detailed fields of a `WorkflowTask` event message body are as follows:

```
{
    "EventType":"WorkflowTask",
    "WorkflowTaskEvent":{
        // WorkflowTaskEvent field
     }
}
```

The data structure and fields of `WorkflowTask` are as detailed below:

Name	Type	Description
TaskId	String	ID of video processing task.
Status	String	Task flow status. Valid values: PROCESSING: processing. FINISH: finished.
ErrCode	Integer	Disused. Please use `ErrCode` of each specific task.
Message	String	Disused. Please use `Message` of each specific task.
InputInfo	MediaInputInfo	Information of the target file of video processing. Note: this field may return null, indicating that no valid values can be obtained.
MetaData	MediaMetaData	Source video metadata. Note: this field may return null, indicating that no valid values can be obtained.
MediaProcessResultSet	Array of MediaProcessTaskResult	Execution status and result of video processing task.
AiContentReviewResultSet	Array of AiContentReviewResult	Execution status and result of video content moderation task.

AiAnalysisResultSet	Array of AiAnalysisResult	Execution status and result of video content analysis task.
AiRecognitionResultSet	Array of AiRecognitionResult	Execution status and result of video content recognition task.

EditMediaTask event

The detailed fields of an `EditMediaTask` event message body are as follows:

```
{
    "EventType":"EditMediaTask",
    "EditMediaTaskEvent":{
        // EditMediaTask field
     }
}
```

The data structure and fields of `EditMediaTask` are as detailed below:

Name	Type	Description
TaskId	String	Task ID.
Status	String	Task status. Valid values: PROCESSING: processing. FINISH: finished.
ErrCode	Integer	Error code. 0: success; other values: failure.
Message	String	Error message.
Input	EditMediaTaskInput	Input of video editing task.
Output	EditMediaTaskOutput	Output of video editing task. Note: this field may return null, indicating that no valid values can be obtained.

CLB Trigger Description

Last updated：2024-12-02 19:58:17

You can implement backend web services by writing SCF functions and providing services through CLB, which will pass the request content as parameters to the function and return the result from the function back to the requester as the response.

Characteristics of CLB triggers:

Push model

After a CLB trigger receives a request from CLB, if CLB is configured to connect with a function on the backend, the function will be triggered for execution, and CLB will send the information of the request as `event` input parameters to the triggered function, including the specific method how the request is received as well as the `path`, `header`, and `query` of the request.

Sync invocation

A CLB trigger invokes functions synchronously. For more information on invocation types, please see Invocation Types.

Note:

CLB accounts are divided into standard accounts and traditional accounts. Traditional accounts cannot be bound to SCF. We recommend you upgrade them to standard accounts.

CLB Trigger Configuration

CLB triggers can be configured in either the **SCF** or **CLB** console.

SCF console

CLB console

In the **Serverless console**, you can add CLB triggers in trigger method, select existing CLB instances, create routing rules, and configure URL request paths.

When configuring routing rules in the **CLB console**, you can choose **Cloud Function** as the backend and select functions in the same region as the CLB instance. In the CLB console, you can also configure and manage advanced CLB capabilities, such as WAF protection, SNI multi-domain certificate, and ENI.

CLB Trigger Binding Limits

One CLB path rule can be bound to only one function, but one function can be bound to multiple CLB rules as the backend. Rules with the same path, listener, and host are regarded as the same rule and cannot be bound repeatedly.

Currently, CLB triggers can only be bound to functions in the same region; for example, a function created in the Guangzhou region can only be bound to and triggered by CLB rules created in the Guangzhou region.

Request and Response

Request method refers to the method to process request sent from CLB to SCF, and response method refers to the method to process the returned value sent from SCF to CLB. Both request and response methods are automatically processed by the CLB trigger. When it triggers the function, data structures must be returned in the request method.

Note:

`X-Vip` , `X-Vport` , `X-Uri` , `X-Method` , and `X-Real-Port` fields must be customized in the CLB console before they can be transferred. For custom configurations, see Layer-7 Custom Configuration.

Event message structure of integration request for CLB trigger

When a CLB trigger receives a request, event data will be sent to the bound function in JSON format as shown below.

Note:

In the CLB trigger scenario, all requests and responses need to be transferred in JSON. For images, files, and other data, as directly passing in JSON content will cause invisible characters to be lost, Base64 encoding is required as detailed below:

If the `Content-type` is `text/*` , `application/json` , `application/javascript` , or `application/xml` , CLB will not transcode the body content.

For all other types, CLB will Base64-encode them first and then forward them.

```
{
  "headers": {
    "Content-type": "application/json",
    "Host": "test.clb-scf.com",
    "User-Agent": "Chrome",

    "X-Stgw-Time": "1591692977.774",
    "X-Client-Proto": "http",
    "X-Forwarded-Proto": "http",
    "X-Client-Proto-Ver": "HTTP/1.1",
    "X-Real-IP": "9.43.175.219",
    "X-Forwarded-For": "9.43.175.xx"

    "X-Vip": "121.23.21.xx",
    "X-Vport": "xx",
    "X-Uri": "/scf_location",
    "X-Method": "POST"
    "X-Real-Port": "44347",
  },
```

```
  "payload": {
    "key1": "123",
    "key2": "abc"
  },
}
```

The data structures are as detailed below:

Structure	Description
X-Stgw-Time	Request start timestamp
X-Forwarded-Proto	`scheme` structure of the request
X-Client-Proto-Ver	Protocol type
X-Real-IP	Client IP address
X-Forward-For	Passed proxy IP address
X-Real-Port	Records the `Path` parameters configured in CLB and their actual values (optional custom configuration of CLB)
X-Vip	CLB VIP address (optional custom configuration of CLB)
X-Vport	CLB Vport (optional custom configuration of CLB)
X-Url	CLB request path (optional custom configuration of CLB)
X-Method	CLB request method (optional custom configuration of CLB)

Note:

The content may be increased significantly during CLB iteration. At present, it is guaranteed that the content of the data structure will only be increased but not reduced, so that the existing structure will not be compromised. Parameters in real requests may appear in multiple locations and can be selected based on your business needs.

Integration response

Integration response means that CLB parses the returned content of the function and constructs an HTTP response based on the parsed content. With the aid of integration response, you can control the status code, headers, and body content of the response by using code and implement response to content in custom formats, such as XML, HTML, JSON, and even JS. When using integration response, data structures need to be returned in the returned data structures of integration response for CLB trigger before they can be successfully parsed; otherwise, the error message `{"errno":403,"error":"Analyse scf response failed."}` will appear.

Returned data structures of integration response for CLB trigger

If integration response is set for CLB, data needs to be returned in the following structures:

```
{
    "isBase64Encoded": false,
    "statusCode": 200,
    "headers": {"Content-Type":"text/html"},
    "body": "<html><body><h1>Heading</h1><p>Paragraph.</p></body></html>"
}
```

The data structures are as detailed below:

Structure	Description
isBase64Encoded	This indicates whether the content in the `body` is Base64-encoded binary. It should be `true` or `false` in JSON format.
statusCode	HTTP return code, which should be an integer value.
headers	HTTP return header, which should contain multiple `key-value` or `key:[value,value]` objects. Both key and value should be strings.
body	HTTP return body.

If you need to return multiple headers with the same key, you can use a string array to describe different values; for example:

```
{
    "isBase64Encoded": false,
    "statusCode": 200,
    "headers": {"Content-Type":"text/html","Key":["value1","value2","value3"]},
    "body": "<html><body><h1>Heading</h1><p>Paragraph.</p></body></html>"
}
```

TencentCloud API Trigger

Last updated：2024-12-02 19:58:17

Overview

You can write SCF functions to handle your own business logic and trigger the functions through the management APIs exposed by SCF. The management APIs are collectively referred to as TencentCloud API in Tencent Cloud. By using the `Invoke` API of SCF, you can trigger and invoke functions as needed.
The detailed TencentCloud API call method can be found in Invoke. TencentCloud API triggers have the following characteristics:
Invocation method: the sync or async triggering method can be defined according to the `InvocationType` parameter.
Custom event: the event or data content that triggers the function can be defined according to the `ClientContext` parameter, and the content has to be encoded in JSON format.

TencentCloud API Call

To trigger a function through TencentCloud API, you need to:
1. Authenticate the API;
2. Enter the common parameters;
3. Parse the returned result.
In addition, if you do not want to construct or parse the request content on your own, you can directly use the TencentCloud API SDK for Python, PHP, Java, Go, .NET, or Node.js to trigger functions. For more information on how to use the SDK, please see SDK.