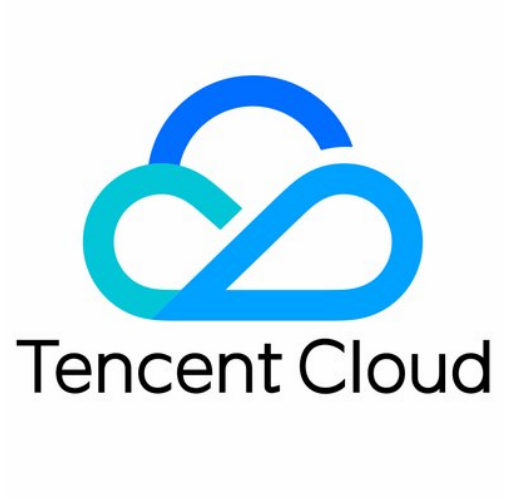
Serverless Cloud Function

Code Development

Product Documentation



Contents

Code Development
Python
Runtime Environment

Last updated：2025-01-03 15:00:05

Python Versions

Currently, the following versions of Python are supported:
Python 3.10
Python 3.9
Python 3.7
Python 3.6
Python 2.7
You can choose a desired runtime environment when creating a function. Click here to check for the official guide on selection of Python 2 or Python 3.

Environment Variables

The table below lists the Python related environment variables built in the current runtime environment:

Environment Variable Key	Specific Value or Value Source
`PYTHONDONTWRITEBYTECODE`	x
`PYTHONPATH`	/var/user:/opt

For detailed description on the environment variables, see Environment Variables.

Included Libraries and Usage

Note:
For Python 3.7 and later, the platform no longer has additional built-in dependent libraries. For more information on the dependencies required by code execution, see Dependency Installation.

COS SDK

SCF's running environment for Python 3.6 and Python 2.7 has integrated the COS SDK. You can introduce and use the COS SDK in your code as follows:

```
from qcloud_cos_v5 import CosConfig
from qcloud_cos_v5 import CosS3Client
```

For detailed instructions on usage of the COS SDK, see COS SDK for Python.

Built-in Libraries

The table below lists the supported SCF libraries in the Python 3.6 cloud runtime environment.

Note:

To use a library not listed here, you need to locally install, package, and upload it. For detailed directions, see Installing Dependent Libraries.

Library Name	Version
absl-py	0.2.2
asn1crypto	0.24.0
astor	0.7.1
bleach	1.5.0
certifi	2019.3.9
cffi	1.12.2
chardet	3.0.4
cos-python-sdk-v5	1.6.6
cryptography	2.6.1
dicttoxml	1.7.4
gast	0.2.0
grpcio	1.13.0
html5lib	0.9999999
idna	2.8
iniparse	0.4
Markdown	2.6.11

mysqlclient	1.3.13
numpy	1.15.0
Pillow	6.0.0
pip	9.0.1
protobuf	3.6.0
psycopg2-binary	2.8.2
pycparser	2.19
pycurl	7.43.0
PyMySQL	0.9.3
pytz	2019.1
qcloud-image	1.0.0
qcloudsms-py	0.1.3
requests	2.21.0
serverless-db-sdk	0.0.1
setuptools	28.8.0
six	1.12.0
tencentcloud-sdk-python	3.0.65
tencentserverless	0.1.4
tensorboard	1.9.0
tensorflow	1.9.0
tensorflow-serving-api	1.9.0
termcolor	1.1.0
urllib3	1.24.2
Werkzeug	0.14.1
wheel	0.31.1

The table below lists the supported SCF libraries in the Python 2.7 cloud runtime environment.

Library Name	Version
absl-py	0.2.2
asn1crypto	0.24.0
astor	0.7.1
backports.ssl-match-hostname	3.4.0.2
backports.weakref	1.0.post1
bleach	1.5.0
cassdk	1.0.2
certifi	2017.11.5
cffi	1.12.2
chardet	3.0.4
cos-python-sdk-v5	1.6.6
cryptography	2.6.1
dicttoxml	1.7.4
enum34	1.1.6
funcsigs	1.0.2
futures	3.2.0
gast	0.2.0
grpcio	1.13.0
html5lib	0.9999999
idna	2.6
iniparse	0.4
ipaddress	1.0.22
Markdown	2.6.11

mock	2.0.0
mysqlclient	1.3.13
nose	1.3.7
numpy	1.14.5
ordereddict	1.1
pbr	4.1.0
Pillow	6.0.0
pip	18
protobuf	3.6.0
psycopg2-binary	2.8.2
pyaml	2019.4.1
pycparser	2.19
pycurl	7.43.0.1
pygpgme	0.3
PyMySQL	0.9.3
pytz	2019.1
PyYAML	5.1
qcloud-image	1.0.0
qcloudsms-py	0.1.3
requests	2.18.4
serverless-db-sdk	0.0.1
setuptools	39.1.0
six	1.11.0
tencentcloud-sdk-python	3.0.65
tencentserverless	0.1.4

tensorboard	1.9.0
tensorflow	1.9.0
tensorflow-serving-api	1.9.0
termcolor	1.1.0
urlgrabber	3.10.2
urllib3	1.22
Werkzeug	0.14.1
wheel	0.31.1

Deployment Methods

Last updated：2024-12-02 18:12:22

Function Form

The Python function form is generally as follows:

```
import json

def main_handler(event, context):
    print("Received event: " + json.dumps(event, indent = 2))
    print("Received context: " + str(context))
    return("Hello World")
```

Execution Method

An execution method should be specified when each SCF function is created. The execution method of the Python programming language is similar to `index.main_handler`, where `index` refers to the `index.py` entry file, and `main_handler` refers to the `main_handler` entry function. When submitting the zip code package by uploading a local zip package or uploading it through COS, the root directory of the package should contain the specified entry file and the file should contain the specified entry function. The filename and function name should match those entered in the execution method to ensure successful execution.

Input Parameters

The input parameters in the Python environment include event and context, both of which are of the Python dict type.
event: this parameter is used to pass the trigger event data.
context: this parameter is used to pass runtime information to your handler.
The event parameter varies with trigger or event source. For more information on its data structure, see Trigger Overview.

Response

Your handler can use `return` to return a value. The return value will be handled differently depending on the function invocation type.

In the Python environment, a serializable object such as `dict` object can be directly returned:

```
def main_handler(event, context):
    resp = {
        "isBase64Encoded": false,
        "statusCode": 200,
        "headers": {"Content-Type":"text/html","Key":["value1","value2","value3"]},
        "body": "<html><body><h1>Heading</h1><p>Paragraph.</p></body></html>"
    }
    return(resp)
```

Two methods are available to return values:

Sync invocation: the return value of a sync invocation will be serialized in JSON and returned to the caller for subsequent processing. The function testing in the console is sync invocation, which can capture the function's return value and display it after the invocation is completed.

Async invocation: the return value of an async invocation will be discarded, since the invocation method responds right after the function is triggered, without waiting for function execution to complete.

Note:

The return value of both sync and async invocations will be recorded in the function logs.

Exception Handling

You can throw an exception using `raise Exception` inside the function.

If the exception is captured and handled before returning without being thrown outside, the function is considered to have been executed. In this case, information specified in the entry function's `return` will be returned.

The following sample codes show that the function is successfully executed and will return `Hello World` .

```
# -*- coding: utf8 -*-
def main_handler(event, context):
  try:
      print("try exception")
      raise Exception("err msg")
  except Exception as e:
      print(e)
  return("Hello World")
```

If an exception is not captured before returning, it will be thrown outside the entry function and captured by SCF. In this case, the function is considered to have failed, and an error message will be returned.

The following sample codes show that the function execution fails.

```
# -*- coding: utf8 -*-
def main_handler(event, context):
```

```
    print("try exception")
    raise Exception("err msg")
    return("Hello World")
```

The function returns information similar to the following:

```
{
  "errorCode":-1,
  "errorMessage":"user code exception caught",
  "requestId":"a325b967-ef5b-4aa3-a329-c6bb0df72948",
  "stackTrace":"Traceback (most recent call last):\\n  File \\"/var/user/index.py\\
  "statusCode":430
}
```

The `errorCode` field indicates a code error, and `errorMessage` provides error details. The `stackTrace` field indicates an error stack, and `statusCode` provides error details. For more information about `statusCode`, see Function Status Code.

Development Methods

Last updated：2024-12-02 18:12:22

Deployment Methods

Tencent Cloud SCF provides the following function deployment methods. For more information about how to create and update a function, see Create and Update a Function.
Uploading and deploying a zip package, as instructed in Installing and Deploying Dependencies.
Editing and deploying functions via the console, as instructed in Deploying Functions.
Using the command line, as instructed in Deployment Through Serverless Framework CLI.

Installing and Deploying Dependencies

Currently, the SCF standard Python Runtime only supports writing to the `/tmp` directory, and other directories are read-only. Therefore, you need to install, package and upload the local dependent library for use. The Python dependency package can be uploaded with function codes to the cloud, or uploaded to the layer that will be bound to the required function.

Locally installing dependency packages

Dependency manager

In Python, dependencies can be managed with the pip package manager. Replace `pip` with `pip3` or `pip2` according to the environment configurations.

Directions

1. Configure dependency information in `requirements.txt` .
2. Run the `pip install -r requirements.txt -t .` command under the code directory to install the dependency package. You can use the `-t` parameter to specify the installation directory, or directly run `-t .` under the project's code directory to install the dependency package in the current directory.

Note:

Use the `pip freeze > requirements.txt` command to generate a `requirements.txt` file that contains all dependencies of the current environment.
Because the function is running on CentOS 7, install the dependency package in the same environment to avoid errors. For detailed directions, see Using Container Image.
If some dependencies require dynamic link library, please manually copy these dependencies to the installation directory, and then package them for uploading. For more information, see Installing Dependency with Docker.

Sample

1. Use the `index.py` code file shown below to install the `requests` dependency locally.

```
# -*- coding: utf8 -*-
import requests

def main_handler(event, context):
     addr = "www.qq.com"
     resp = requests.get(addr)
     print(resp)
     return resp
```

2. Run the `pip3 install requests -t .` command to install the `requests` dependency under the current directory of the project. The code file is as follows:

```
$ pip3 install requests -t .
Collecting requests
      Using cached requests-2.25.1-py2.py3-none-any.whl (61 kB)
Collecting certifi>=2017.4.17
      Using cached certifi-2020.12.5-py2.py3-none-any.whl (147 kB)
Collecting chardet<5,>=3.0.2
      Using cached chardet-4.0.0-py2.py3-none-any.whl (178 kB)
Collecting idna<3,>=2.5
      Using cached idna-2.10-py2.py3-none-any.whl (58 kB)
Collecting urllib3<1.27,>=1.21.1
      Using cached urllib3-1.26.4-py2.py3-none-any.whl (153 kB)
Installing collected packages: urllib3, idna, chardet, certifi, requests
Successfully installed certifi-2020.12.5 chardet-4.0.0 idna-2.10 requests-
2.25.1 urllib3-1.26.4

$ ls -l
total 8
drwxr-xr-x   3 xxx  111    96  4 29 16:45 bin
drwxr-xr-x   7 xxx  111   224  4 29 16:45 certifi
drwxr-xr-x   8 xxx  111   256  4 29 16:45 certifi-2020.12.5.dist-info
drwxr-xr-x  44 xxx  111  1408  4 29 16:45 chardet
drwxr-xr-x   9 xxx  111   288  4 29 16:45 chardet-4.0.0.dist-info
drwxr-xr-x  11 xxx  111   352  4 29 16:45 idna
drwxr-xr-x   8 xxx  111   256  4 29 16:45 idna-2.10.dist-info
-rw-r--r--@ 1 xxx 111 177 4 29 16:33 index.py
drwxr-xr-x  21 xxx  111   672  4 29 16:45 requests
drwxr-xr-x   9 xxx  111   288  4 29 16:45 requests-2.25.1.dist-info
drwxr-xr-x  17 xxx  111   544  4 29 16:45 urllib3
drwxr-xr-x  10 xxx  111   320  4 29 16:45 urllib3-1.26.4.dist-info
```

Packaging and uploading

You can upload dependencies together with the project, and use them through the `import` statement in function codes. You can also package and deploy dependencies to a layer, and bind the layer to a function being created to reuse them.

The zip package for deploying functions or layers can be generated automatically by a local folder via the console or manually. All the packaging should be under the project directory to place codes and dependencies in the root directory of the zip package. For more information, see Packaging requirements.

Special dependency packages

Some Python dependencies such as the `pycryptodome` dependency need to be compiled for installation. Because the compilation varies with the operating system, the dependent library, dynamic library, and other programs compiled on Windows or Mac may be unable to run in the SCF environment. The following solutions are recommended.

Use the dependent library that is ready for FaaS open source implementations.

Search dependencies or submit requirements in the SCF public layer. This layer collects and stores special dependency packages, and provides the deployment support.

Use the container solution and SCF container image to install and extract special dependencies locally, and then package and upload them to the code runtime environment.

Log Description

Last updated：2024-12-02 18:12:22

Log Development

You can use the following statements in the program to output a log:

print

logging module

For example, you can query the output content in the function log by running the following code:

```
import logging
logger = logging.getLogger()
logger.setLevel(logging.INFO)

def main_handler(event, context):
    logger.info('got event{}'.format(event))
    print("got event{}".format(event))
    return 'Hello World!'
```

Log Query

Currently, all function logs are delivered to CLS. You can configure the function log delivery. For more information, see Log Delivery Configuration.

You can query function execution logs on the log query page of SCF or CLS. For more information on the log query method, see Log Search Guide.

Note:

Function logs are delivered to the `LogSet` log set and `LogTopic` log topic in CLS, both of which can be queried through the function configuration.

Custom Log Fields

Currently, the string content output by simple `print` or `logger` in the function code will be recorded in the `SCF_Message` field when it is delivered to CLS. For descriptions of CLS fields, see Log Delivery Configuration.

At present, SCF supports adding custom fields to the content output to CLS. By doing so, you can output business fields and related data content to logs and use the search capability of CLS to query and track them in the execution

process.

Note:

If you need to query the key value of a custom field such as `SCF_CustomKey: SCF` , add a key-value index to the log topic for SCF log delivery as instructed in Configuring Indexes.

To avoid function log query failures caused by misuse of the index configuration, the default destination topic for SCF log delivery (prefixed with `SCF_LogTopic_`) does not support modifying the index configuration. You need to set the destination topic to custom delivery first and then update the log topic's index configuration.

After the index configuration is modified for a log topic, it will take effect only for newly written data.

Output method

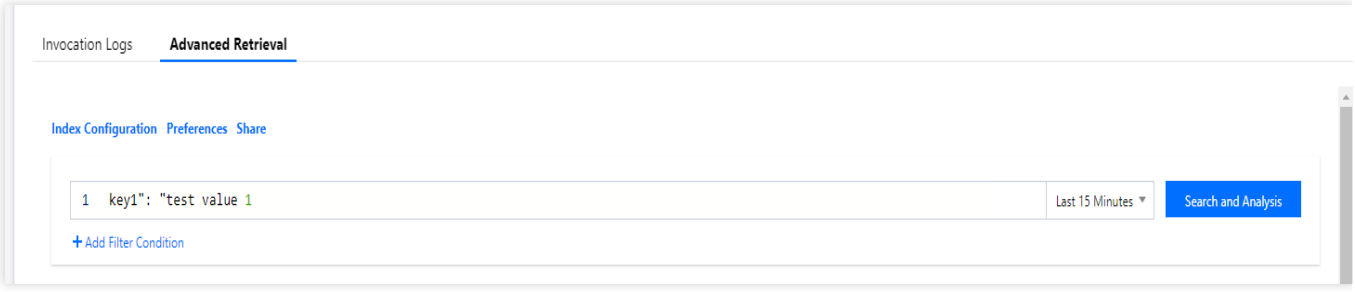
If a single-line log output by a function is in JSON format, the JSON content will be parsed into the format of `field:value` when it is delivered to CLS. Only the first level of the JSON content can be parsed in this way, while other nested structures will be recorded as values.

You can run the following code to test:

```
# -*- coding: utf8 -*-
import json

def main_handler(event, context):
    print(json.dumps({"key1": "test value 1","key2": "test value 2"}))
    return("Hello World!")
```

Search method

After using the above code to perform a test, you can run the following statement to search in **Function Management > Log Query > Advanced Search**:



Search result

After the test is written to CLS, you can find the `key1` field in the log query as shown below:

Examples

Last updated：2024-12-02 18:12:22

These examples provide code snippets based on Python 3.6 for your reference.

You can click scf-python-code-snippet to obtain the relevant code snippets and directly use them for deployment.

Obtaining Environment Variables

This example shows you how to obtain all or a single environment variable.

```
# -*- coding: utf8 -*-
import os

def main_handler(event, context):
      print(os.environ)
      print(os.environ.get("SCF_RUNTIME"))
      return("Hello World")
```

Formatting Local Time

This example shows you how to output date and time in the specified format.

The SCF environment uses the UTC format by default. To output in Beijing time, you can add the `TZ=Asia/Shanghai` environment variable to the function.

```
# -*- coding: utf8 -*-
import time

def main_handler(event, context):
      print(time.strftime('%Y-%m-%d %H:%M:%S',time.localtime(time.time())))
      return("Hello World")
```

Accessing TencentDB for MySQL Instance

This example shows you how to use the PyMySQL library for the database connection. You need to run the `pip3 install PyMySQL -t .` command under the project directory to install this dependent library.

Please note the following:

Configure the function in the same VPC where the TencentDB for MySQL instance is located to ensure the network accessibility.
Replace the database's IP, username, password, name and other information shown in the codes with your actual values.

```
# -*- coding: utf8 -*-
import pymysql

def main_handler(event, context):

        # Start connecting to the database
        db = pymysql.connect(host="host ip",port=3306,user="user",password="password

        # Use the cursor() method to create a cursor object
        cursor = db.cursor()

        # Use the execute() method to execute the SQL query statement
        cursor.execute("SELECT VERSION()")

        # Use the fetchone() method to obtain a single data entry.
        data = cursor.fetchone()

        print ("Database version : %s " % data)

        # Stop connecting to database
        db.close()
```

Initiating Network Connections in a Function

This example shows you how to use the requests library to initiate network connections in a function and obtain page information. You can run the `pip3 install requests -t .` command under the project directory to install this dependent library.

```
# -*- coding: utf8 -*-
import requests


def main_handler(event, context):
      addr = "https://cloud.tencent.com"
      resp = requests.get(addr)
      print(resp)
      print(resp.text)
```

```
        return resp.status_code
```

Obtaining Webpages Using SCF and API Gateway

This example shows you how to access the URL through API Gateway while obtaining the HTML page by configuring API Gateway trigger and enabling integration response.

```
# -*- coding: utf8 -*-
import time

def main_handler(event, context):
      resp = {
          "isBase64Encoded": False,
          "statusCode": 200,
          "headers": {"Content-Type":"text/html"},
          "body": "<html><body><h1>Hello</h1><p>Hello World.</p></body></html>"
      }
      return resp
```

Obtaining Images Using SCF and API Gateway

This example shows you how to access the URL through API Gateway while obtaining the binary image file by configuring API Gateway trigger and enabling integration response.

```
# -*- coding: utf8 -*-
import base64

def main_handler(event, context):
       with open("tencent_cloud_logo.png","rb") as f:
           data = f.read()
       base64_data = base64.b64encode(data)
       base64_str = base64_data.decode('utf-8')
       resp = {
           "isBase64Encoded": True,
           "statusCode": 200,
           "headers": {"Content-Type":"image/png"},
           "body": base64_str
       }
       return resp
```

Node.js
Notes on Node.js

Last updated：2024-12-02 18:12:22

Node.js Version Selection

Currently, the following versions of Node.js programming language are supported:

Node.js 16.13

Node.js 14.18

Node.js 12.16

Node.js 10.15

Node.js 8.9 (deactivating soon)

Node.js 6.10 (deactivating soon)

You can choose a desired runtime environment when creating a function.

Environment Variables

The environment variables built in the current Node.js runtime environment are as shown in the table below:

Node.js Version	Environment Variable Key	Specific Value or Value Source
Node.js 16.13	`NODE_PATH`	/var/user:/var/user/node_modules:/var/lang/node16/lib/node_modules:/opt:/opt/node
Node.js 14.18	`NODE_PATH`	/var/user:/var/user/node_modules:/var/lang/node14/lib/node_modules:/opt:/opt/node
Node.js 12.16	`NODE_PATH`	/var/user:/var/user/node_modules:/var/lang/node12/lib/node_modules:/opt:/opt/node
Node.js 10.15	`NODE_PATH`	/var/user:/var/user/node_modules:/var/lang/node10/lib/node_modules:/opt:/opt/node
Node.js 8.9	`NODE_PATH`	/var/user:/var/user/node_modules:/var/lang/node8/lib/node_modules:/opt:/opt/node_
Node.js 6.10	`NODE_PATH`	/var/user:/var/user/node_modules:/var/lang/node6/lib/node_modules:/opt:/opt/node_

For more information on environment variables, see Environment Variables.

Included Library and Usage

Note:

For Node.js 14.18 and later, the platform no longer has additional built-in dependency libraries. For more information on the dependencies required by code execution, see Dependency Installation and Online Dependency Installation.

COS SDK

SCF's runtime environment for Node.js 12.16 or earlier already contains the COS SDK for Node.js, and the specific version is `cos-nodejs-sdk-v5` .

The COS SDK can be referenced and used within the code as follows:

```
var COS = require('cos-nodejs-sdk-v5');
```

For more information on how to use the COS SDK, see COS SDK for Node.js.

Built-in library in environment

The following libraries are supported in Node.js runtime:

Node.js 12.16

Node.js 10.15

Node.js 8.9

Node.js 6.10

Library Name	Version
cos-nodejs-sdk-v5	2.5.20
base64-js	1.3.1
buffer	5.5.0
crypto-browserify	3.12.0
ieee754	1.1.13
imagemagick	0.1.3
isarray	2.0.5
jmespath	0.15.0
lodash	4.17.15

microtime	3.0.0
npm	6.13.4
punycode	2.1.1
puppeteer	2.1.1
qcloudapi-sdk	0.2.1
querystring	0.2.0
request	2.88.2
sax	1.2.4
scf-nodejs-serverlessdb-sdk	1.1.0
tencentcloud-sdk-nodejs	3.0.147
url	0.11.0
uuid	7.0.3
xml2js	0.4.23
xmlbuilder	15.1.0

Library Name	Version
cos-nodejs-sdk-v5	2.5.14
base64-js	1.3.1
buffer	5.4.3
crypto-browserify	3.12.0
ieee754	1.1.13
imagemagick	0.1.3
isarray	2.0.5
jmespath	0.15.0
lodash	4.17.15

microtime	3.0.0
npm	6.4.1
punycode	2.1.1
puppeteer	2.0.0
qcloudapi-sdk	0.2.1
querystring	0.2.0
request	2.88.0
sax	1.2.4
scf-nodejs-serverlessdb-sdk	1.0.1
tencentcloud-sdk-nodejs	3.0.104
url	0.11.0
uuid	3.3.3
xml2js	0.4.22
xmlbuilder	13.0.2

Library Name	Version
cos-nodejs-sdk-v5	2.5.8
base64-js	1.2.1
buffer	5.0.7
crypto-browserify	3.11.1
ieee754	1.1.8
imagemagick	0.1.3
isarray	2.0.2
jmespath	0.15.0
lodash	4.17.4

npm	5.6.0
punycode	2.1.0
puppeteer	1.14.0
qcloudapi-sdk	0.1.5
querystring	0.2.0
request	2.87.0
sax	1.2.4
tencentcloud-sdk-nodejs	3.0.56
url	0.11.0
uuid	3.1.0
xml2js	0.4.17
xmlbuilder	9.0.1

Library Name	Version
base64-js	1.2.1
buffer	5.0.7
cos-nodejs-sdk-v5	2.0.7
crypto-browserify	3.11.1
ieee754	1.1.8
imagemagick	0.1.3
isarray	2.0.2
jmespath	0.15.0
lodash	4.17.4
npm	3.10.10
punycode	2.1.0

qcloudapi-sdk	0.1.5
querystring	0.2.0
request	2.87.0
sax	1.2.4
tencentcloud-sdk-nodejs	3.0.10
url	0.11.0
uuid	3.1.0
xml2js	0.4.17
xmlbuilder	9.0.1

Development Methods

Last updated：2024-12-02 18:12:22

Function Form

A Node.js function generally has the following two forms:

Example 1:

```
exports.main_handler = async (event, context) => {
  console.log(event);
  console.log(context);
  return event
};
```

Example 2:

```
exports.main_handler = (event,context,callback)=>{
  console.log(event);
  console.log(context);
  callback(null,"hello world");
}
```

Execution Method

When you create an SCF function, you need to specify an execution method. If the Node.js programming language is used, the execution method is similar to `index.main_handler`, where `index` indicates that the executed entry file is `index.js`, and `main_handler` indicates that the executed entry function is `main_handler`. When submitting the zip code package by uploading the zip file locally or through COS, please make sure that the root directory of the package contains the specified entry file, the file contains the entry function specified by the definition, and the names of the file and function match those entered in the trigger; otherwise, execution will fail as the entry file or entry function cannot be found.

Input Parameters

The input parameters in the Node.js environment include `event`, `context`, and `callback`, where `callback` is optional.

event: This parameter is used to pass the trigger event data.

context: This parameter is used to pass runtime information to your handler.
callback (optional): `callback` is a function that can be used in **non-async handlers** to return a response. The response object must be compatible with `JSON.stringify`. The `callback` function has two parameters: `Error` and response. When invoking this function, SCF will wait for the function to complete before returning a response or error.

Return and Exception

Async handler

Async handlers must use the `async` keyword, use `return` to return a response, and use `throw` to return an error message.
In SCF, if your Node.js function contains an async task, a promise must be returned to ensure that the task is executed on the current invocation. When you fulfill or reject the promise, SCF will return a response or error message.
Note:
The promise method does not support returning with the `callback` method. You should use `return`.
Sample async handler:

```
exports.main_handler = async(event,context,callback) => {
  const promise = new Promise((resolve,reject) => {
    setTimeout(function() {
       resolve('success')
       // reject('failure')
    }, 2000)
  })
  return promise
};
```

Non-async handler

For non-async handlers, the function will be continuously executed until the function execution completes or times out, and SCF will return a response or error message.
Note:
Some externally referenced libraries may cause the event loop to never get empty, which leads to timeout due to failed return of the function. In order to avoid the impact of external libraries, you can control the function return timing by turning off event loop wait. You can modify the default callback behavior by **setting** `context.callbackWaitsForEmptyEventLoop` **to** `false` to avoid waiting for the event loop to get empty. By setting `context.callbackWaitsForEmptyEventLoop = false;` before the `callback` callback is executed, the SCF backend can freeze the process immediately after the `callback` callback is invoked and return immediately after the sync process is completed without waiting for the event in the event loop.

Sample non-async handler:

```
exports.main_handler = (event, context,callback) => {
    context.callbackWaitsForEmptyEventLoop = false
    callback(null,'success')
    setTimeout(() => {
        console.log('finish')
     },5000);
};
```

Async Feature Support

In the runtime of **Node.js 10.15 and above**, sync execution return and async event processing can be performed separately:

After the sync execution process of an entry function is completed and the result is returned, function invocation will immediately return its result, and the return information in the code will be send to the function invoker.

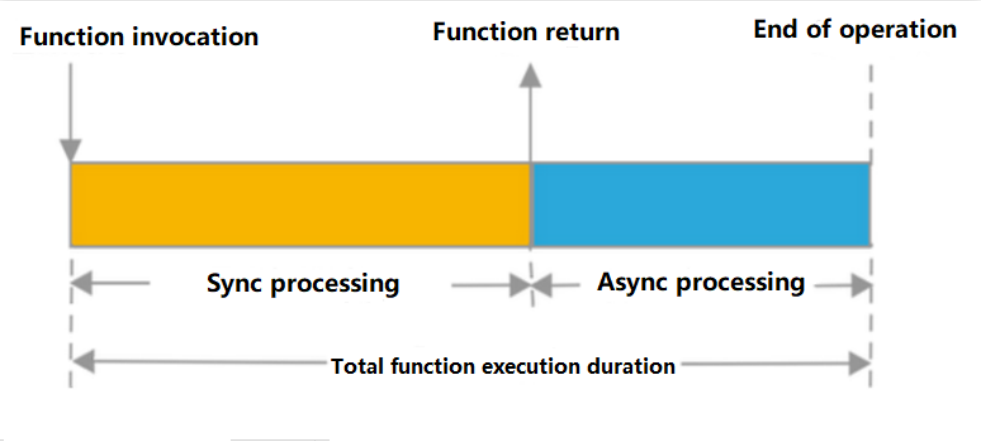
After the sync process is completed and the result returned, the async logic in the code will continue to be executed and processed. The actual function execution process ends and exits only when the async event is completely executed.

Note:

SCF logs are collected and processed after the entire execution process ends. Therefore, before the sync execution process is completed and the result is returned, logs and operation information such as time used and memory utilization cannot be provided in the SCF return information. You can query the detailed information in logs by using `Request Id` after the actual function execution process is completed.

The function execution duration is calculated based on the async event execution duration. If the async event queue cannot get empty or its execution cannot be completed, function timeout will occur. In this case, the invoker may have received the correct response result of the function, but the execution status of the function will still be marked as failure due to timeout, and the timeout period will be calculated as the execution duration.

The sync and async execution attributes, return time, and execution duration in Node.js are as shown below:



Async attribute sample

Use the following sample code to create a function, where the `setTimeout` method is used to set a function that will be executed in 2 seconds:

```
'use strict';
exports.main_handler = (event, context, callback) => {
    console.log("Hello World")
    console.log(event)
    setTimeout(timeoutfunc, 2000, 'data');
    callback(null, event);
};

function timeoutfunc(arg) {
    console.log(`arg => ${arg}`);
}
```

After saving the code, you can invoke this function through the testing feature in the console or the `Invoke` API.
You can see that the function can return the result in a response period below 1 second.
You can see the following statistics in the function execution log:

```
START RequestId: 1d71ddf8-5022-4461-84b7-e3a152403ffc
Event RequestId: 1d71ddf8-5022-4461-84b7-e3a152403ffc
2020-03-18T09:16:13.440Z        1d71ddf8-5022-4461-84b7-e3a152403ffc    Hello World
2020-03-18T09:16:13.440Z        1d71ddf8-5022-4461-84b7-e3a152403ffc    { key1: 'te
2020-03-18T09:16:15.443Z        1d71ddf8-5022-4461-84b7-e3a152403ffc    arg => data
END RequestId: 1d71ddf8-5022-4461-84b7-e3a152403ffc
Report RequestId: 1d71ddf8-5022-4461-84b7-e3a152403ffc Duration:2005ms Memory:128MB
```

A 2,005-ms execution period is logged. You can also find in the log that the `arg => data` is output 2 seconds later, which shows that the relevant async operations are executed in the current invocation after the execution of the sync process is completed, while function invocation ends after execution of the async task is completed.

Deployment Methods

Last updated：2024-12-02 18:12:22

Deployment Methods

Tencent Cloud SCF provides the following function deployment methods. For more information about how to create and update a function, see Create and Update a Function.
Uploading and deploying a zip package, as instructed in Installing and Deploying Dependencies.
Editing and deploying functions via the console, as instructed in Deployment Through Console.
Using the command line, as instructed in Deployment Through Serverless Cloud Framework.

Installing and Deploying Dependencies

Currently, the SCF standard Node.js Runtime only supports writing to the `/tmp` directory, and other directories are read-only. Therefore, you need to install, package and upload the local dependent library for use. The Node.js dependency package can be uploaded with function codes to the cloud, or uploaded to the layer that will be bound to the required function.

Online dependency installation

Node.js provides an online dependency installation feature as detailed in Online Dependency Installation.

Local dependency installation

Dependency manager

In Node.js, dependencies can be managed with the npm package manager.

Directions

Run the `npm install xxx` command in the code directory to install dependencies.

Note:

Because the function is running on CentOS 7, install the dependency package in the same environment to avoid errors. For detailed directions, see Using Container Image.
If some dependencies require dynamic link library, please manually copy these dependencies to the installation directory, and then package them for uploading. For more information, see Installing Dependency with Docker.

Sample

1. Use the `index.js` code file shown below to install the `requests` dependency locally.

```
'use strict';
var request = require('request');
exports.main_handler = async (event, context) => {
 request('https://cloud.tencent.com/', function (error, response, body) {
     if (!error && response.statusCode == 200) {
         console.log(body) // Processing logic for successful request
     }
 })
 return "success"
};
```

2. Run the `npm install request` command to install the requests dependency under the current directory of the project.

Packaging and uploading

You can upload dependencies together with the project, and use them through the `require` statement in function codes. You can also package and deploy dependencies to a layer, and bind the layer to a function being created to reuse them.

The zip package for deploying functions or layers can be generated automatically by a local folder via the console or manually. All the packaging should be under the project directory to place codes and dependencies in the root directory of the zip package. For more information, see Packaging requirements.

Log Description

Last updated：2024-12-02 18:12:22

Logging

You can use the following statements in the program to output the log:

console.log()

console._stdout.write() (supported for Node.js 8.9 or later)

process.stdout.write() (supported for Node.js 8.9 or later)

For example, you can query the output in the function log by running the following code.

```
'use strict';
exports.main_handler = async (event, context) => {
    console.log("Hello World")
    console._stdout.write("Hello World")
    process.stdout.write("Hello World")
    return event
};
```

Notes

The Node.js 12.16 and 10.15 runtime environments use `console.log` to print logs. The platform will encapsulate the log content in the format of "timestamp RerquestId log content" and write it to CLS.

Example:

Log print statement: `console.log("hello world")`

Log output: `2021-12-27T03:53:59.192Z a7358cce-489a-4674-8e4e-68665fa2b81d Hello World`

The Node.js 8.9 runtime environment uses `console.log` , `console._stdout.write()` , or `process.stdout.write()` to print logs. The platform will encapsulate the log content in the format of "timestamp RerquestId log content" and write it to CLS.

Example:

Log print statement: `console.log("hello world")`

Log output: `2021-12-27T03:53:59.192Z a7358cce-489a-4674-8e4e-68665fa2b81d Hello World`

The Node.js 6.10 runtime environment uses `console.log` to print logs. The platform will encapsulate the log content in the format of "timestamp RerquestId log content" and write it to CLS.

Example:

Log print statement: `console.log("hello world")`

Log output: `2021-12-27T03:53:59.192Z a7358cce-489a-4674-8e4e-68665fa2b81d Hello World`

Log Query

Currently, the function logs can be uploaded to Tencent Cloud SCF. You can complete the configuration as instructed in Log Delivery Configuration.
You can search function execution logs on the log query page of the SCF or CLS console. For more information on how to query logs, see Log Search Guide.
Note:
Function logs uploaded to logset and log topic can be both queried with a function configured.

Custom Log Fields

Currently, the content output by simple log print statements in the function code will be recorded in the `SCF_Message` field when it is delivered to CLS. For descriptions of CLS fields, see Log Delivery Configuration.
Currently, SCF supports adding custom fields to logs that are uploaded to CLS. The custom fields allow you to output business fields and relevant data to logs, and track them using the log search feature of CLS.
Note:
If you need to query the key value of a custom field such as `SCF_CustomKey: SCF` , add a key-value index to the log topic for SCF log delivery as instructed in Configuring Indexes.
To avoid function log query failures caused by misuse of the index configuration, the default destination topic for SCF log delivery (prefixed with `SCF_LogTopic_`) does not support modifying the index configuration. You need to set the destination topic to custom delivery first and then update the log topic's index configuration.
After the index configuration is modified for a log topic, it will take effect only for newly written data.

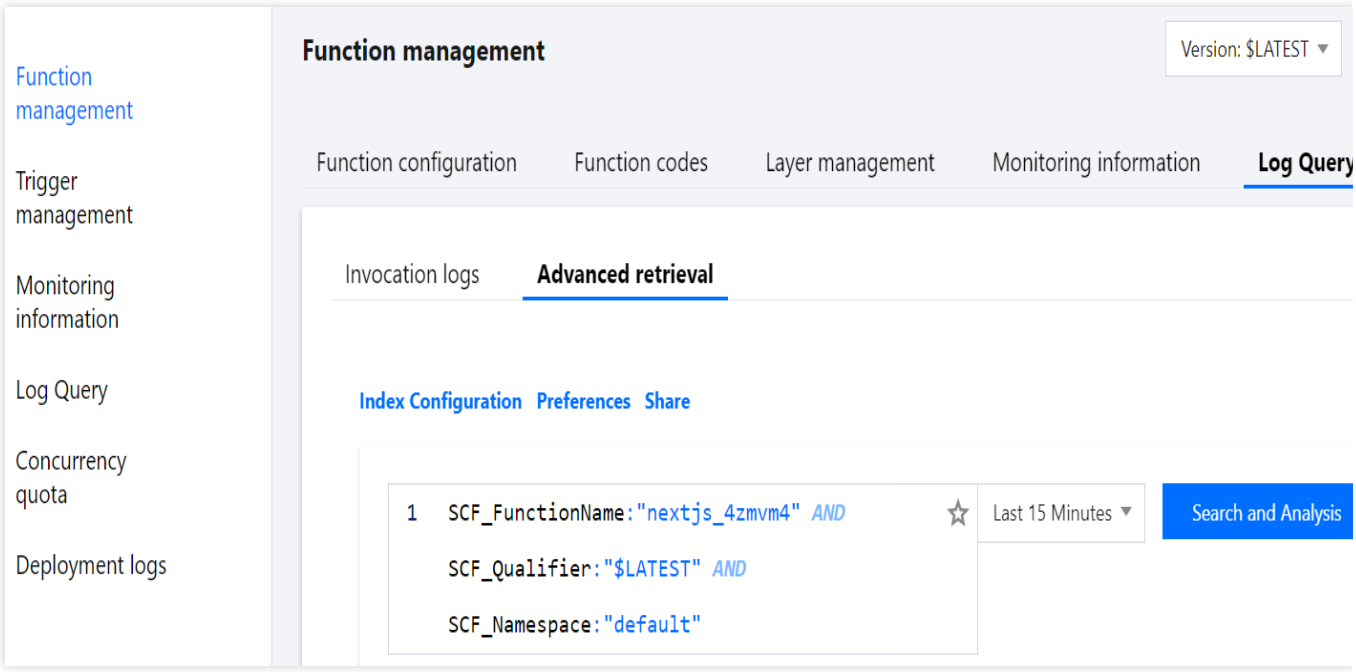
Output

If a function outputs a single-line log in JSON, the log will be parsed and uploaded to CLS in the `field:value` format. Only the first layer will be parsed, and the remaining nested structure will be recorded as values.
Run the following codes and check results:

```
'use strict';
exports.main_handler = async (event, context) => {
    console._stdout.write(JSON.stringify({"key1": "test value 1","key2": "test valu
    return "hello world"
};
```

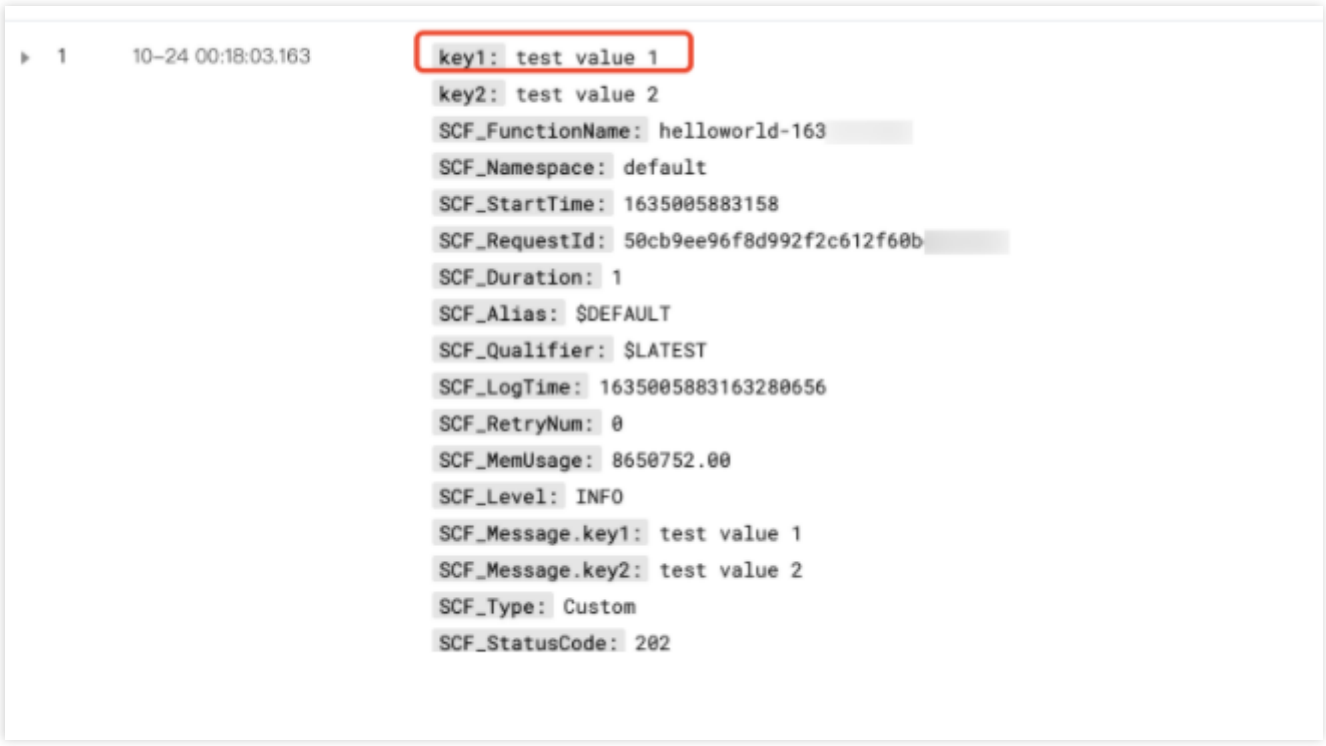
Search

After using the above code to perform a test, you can run the following statement to search in **Function Management** > **Log Query** > **Advanced Search**:



Search for logs

After the test is written to CLS, you can find the `key1` field in the log query as shown below:



Examples

Last updated：2024-12-02 18:12:22

These examples provide code snippets based on Node.js 12.16 for your reference.
You can click scf-nodejs-code-snippet to obtain the relevant code snippets and directly use them for deployment.

Obtaining Environment Variables

This example shows you how to obtain all or a single environment variable.

```
'use strict';
exports.main_handler = async (event, context) => {
        console.log(process.env)
    console.log(process.env.SCF_RUNTIME)
    return "Hello world"
};
```

Formatting Local Time

This example uses the `moment` library to obtain the local time, and provides a time formatted output method to output date and time in the specified format.
The SCF environment uses the UTC format by default. To output in Beijing time, you can add the `TZ=Asia/Shanghai` environment variable to the function.

```
'use strict';
const moment = require('moment')
exports.main_handler = async (event, context) => {
    let currentTime = moment(Date.now()).format('YYYY-MM-DD HH:mm:ss')
    console.log(currentTime)
    return "hello world"
};
```

Initiating Network Connection in Function

This example shows you how to use the requests library to initiate network connections in a function and obtain page information. You can run the `npm install request` command under the project directory to install this dependent library.

```
'use strict';
var request = require('request');
exports.main_handler = async (event, context) => {
    request('https://cloud.tencent.com/', function (error, response, body) {
        if (!error && response.statusCode == 200) {
            console.log(body) // Processing logic for successful request
        }
    })
    return "success"
};
```

Online Dependency Installation

Last updated：2024-12-02 18:12:22

Overview

Tencent Cloud SCF supports online dependency installation during function deployment.

Features

Note:

Online dependency installation is supported only for Node.js.

If **Online install dependency** is enabled in the function configuration, each time the code is uploaded, the SCF backend will check the `package.json` file in the root directory of the code package and try using `npm install` to install the dependencies based on the dependencies in `package.json`.

For example, if the `package.json` file in the project lists the following dependency:

```
{
      "dependencies": {
      "lodash": "4.17.15"
    }
}
```

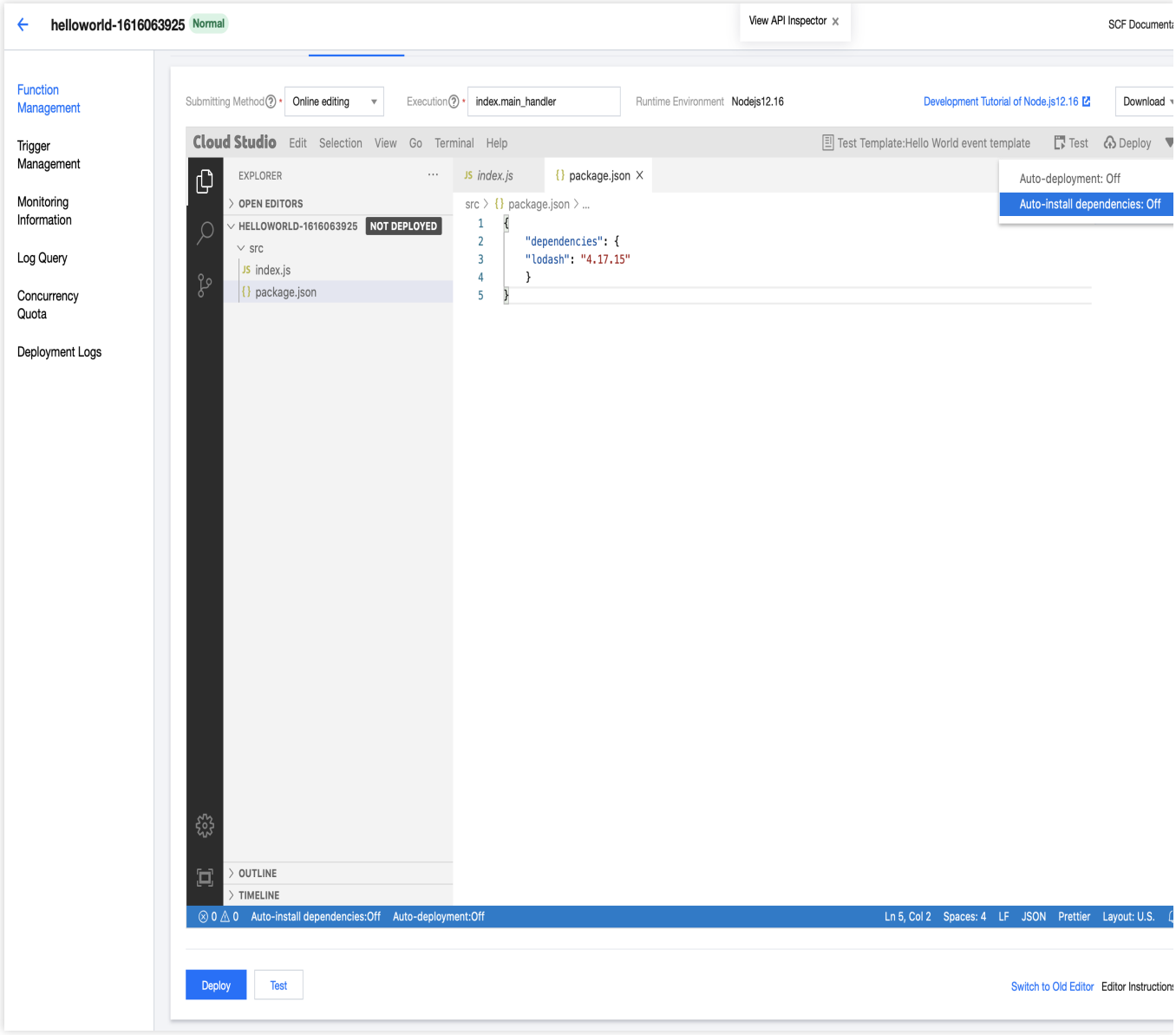
Then this dependency will be imported into the function during the deployment:

```
const _ = require('lodash');
exports.handle = (event, context, callback) => {
    _.chunk(['a', 'b', 'c', 'd'], 2);
    // => [['a', 'b'], ['c', 'd']]
};
```

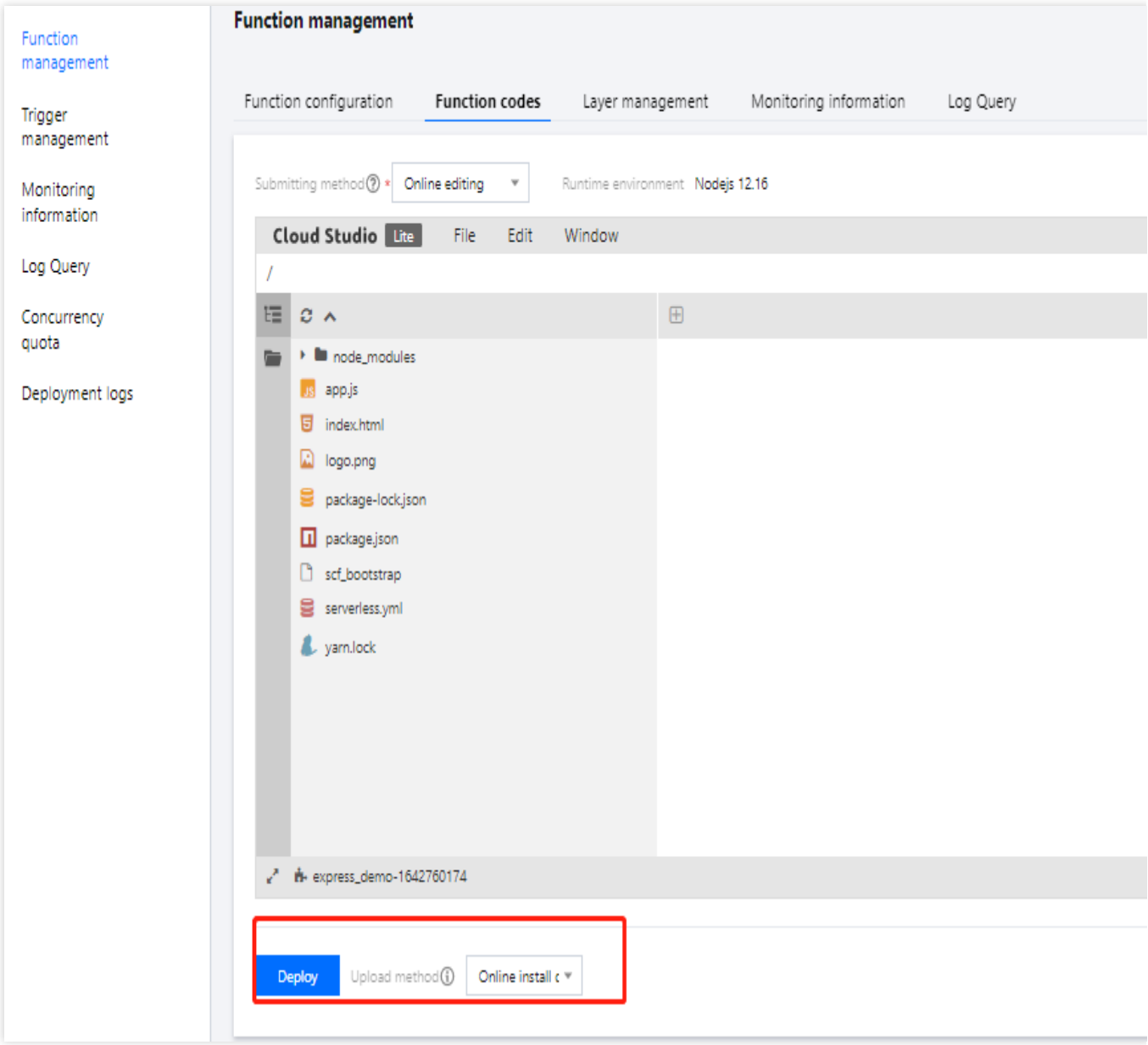
Directions

1. Log in to the SCF console and select **Guangzhou** as the region.
2. Select **Functions** on the left sidebar and locate the target function.
3. Select the **Function codes** tab and modify the function code as needed.
4. In the top-right corner of the IDE code editing window, click

and select **Automatic dependency installation: disable** in the drop-down list to enable automatic dependency installation as shown below:



After enabling online installation, refresh the page. In the bottom right corner of the code editing page, click **Switch to Old Editor**. In **Upload method**, select **Online install dependency**.



5. Click **Deploy**, and SCF will automatically install dependencies according to `package.json` .

Golang
Environment Description

Last updated：2024-12-02 18:12:22

Go Version Selection

Currently, the following versions of Go programming language are supported:
Go 1.8 and above
You can choose the `Go1` runtime environment when creating a function.

Notes

Go is used in SCF in a different way from scripting languages such as Python and Node.js, with the following restrictions:
Code upload is not supported: When Go is used, only developed, compiled, and packaged binary files can be uploaded. The SCF environment does not provide Go compiling capability.
Online editing is not supported: Because code cannot be uploaded, online editing of code is not supported. The code page of a Go runtime function only lists the ways to upload the code through the page or submit the code file through COS.

Development Methods

Last updated：2024-12-02 18:12:22

Function Form

The Go function form is generally as follows:

```
package main

import (
    "context"
    "fmt"
    "github.com/tencentyun/scf-go-lib/cloudfunction"
)

type DefineEvent struct {
    // test event define
    Key1 string `json:"key1"`
    Key2 string `json:"key2"`
}

func hello(ctx context.Context, event DefineEvent) (string, error) {
    fmt.Println("key1:", event.Key1)
    fmt.Println("key2:", event.Key2)
    return fmt.Sprintf("Hello %s!", event.Key1), nil
}

func main() {
    // Make the handler available for Remote Procedure Call by Cloud Function
    cloudfunction.Start(hello)
}
```

Pay attention to the following during code development:

You need to use `package main` to include the `main` function.

Import the `github.com/tencentyun/scf-go-lib/cloudfunction` library by running `go get github.com/tencentyun/scf-go-lib/cloudfunction` before packaging and compilation.

0–2 parameters can be used as the input parameters for the entry function. If parameters are included, `context` needs to be in front, followed by `event`, and the combination of input parameters can be `()`, `(event)`, `(context)`, or `(context, event)`. For more information, see Input parameters.

0–2 parameters can be used as the returned values for the entry function. If parameters are included, the returned content `ret` needs to be in front, followed by `error`, and the combination of returned values can be `()`,

`(ret)` , `(error)` , or `(ret, error)` . For more information, see Returned values.
The `event` input parameter and `ret` returned value need to be compatible with the `encoding/json` standard library for Marshal and Unmarshal operations.
Start the entry function in the `main` function by using the `Start` function inside the package.

Execution Method

When you create an SCF function, you need to specify an execution method. If the Go programming language is used, the execution method is similar to `main` , where `main` indicates that the executable entry file is the compiled `main` binary file.
When submitting the ZIP code package by uploading the ZIP file locally or through COS, make sure that the root directory of the ZIP package contains the specified binary files; otherwise, function creation or execution will fail as the execution file cannot be found.

`package` and `main` functions

When developing an SCF function with Go, you need to make sure that the `main` function is in the `main` package. In the `main` function, start the entry function that actually handles the business by using the `Start` function in the `cloudfunction` package.
You can use the `Start` function in the package in the `main` function through `import "github.com/tencentyun/scf-go-lib/cloudfunction"` .

Entry function

An entry function is the function started by `cloudfunction.Start` , which usually handles the actual business. The input parameters and returned values of the entry function need to be written according to certain specifications.

Input parameters

The entry function can have 0–2 input parameters, such as:

```
func hello()
func hello(ctx context.Context)
func hello(event DefineEvent)
func hello(ctx context.Context, event DefineEvent)
```

If two input parameters are present, you need to make sure that the `context` parameter is before the custom parameter.
The custom parameter can be in Go's own basic data structures (such as `string` or `int`) or custom data structures (such as `DefineEvent` in the sample). If a custom data structure is used, you need to make sure that

the data structure is compatible with the `encoding/json` standard library for Marshal and Unmarshal operations; otherwise, an error will occur when the input parameters are passed in.
The JSON structure corresponding to the custom data structure usually corresponds to the input parameters when the function is executed. When the function is invoked, the JSON data structure of the input parameters will be converted to a custom data structure variable and passed to the entry function.

Note:

The event structures of input parameters passed in by certain triggers have been defined and can be used directly. You can get and use the Go libraries through the Cloud Event Definition by importing "github.com/tencentyun/scf-go-lib/events"` into the code. If you have any questions during use, you can submit an issue or ticket for assistance.

Response

The entry function can have 0–2 returned values, such as:

```
func hello()()
func hello()(error)
func hello()(string, error)
```

If two returned values are defined, you need to make sure that the custom returned value is before the error value. The custom returned value can be in Go's own basic data structures (such as `string` or `int`) or custom data structures. If a custom data structure is used, you need to make sure that it is compatible with the `encoding/json` standard library for Marshal and Unmarshal operations; otherwise, an error will occur due to exceptional conversion when the returned values are returned to the external API.
The JSON structure corresponding to the custom data structure is usually converted to the corresponding JSON data structure in the platform as execution response passed to the invoker when the function invocation is completed.

Deployment Methods

Last updated：2024-12-02 18:12:22

Deployment Methods

A function in the Go environment can only be uploaded as a zip package. You can choose to upload the local zip package or use COS to import the package. The package should include the compiled executable binary file.

Compiling and Packaging

Cross-platform Go compilation can be achieved by specifying OS and ARCH on any platform, so it can be done on Linux, Windows, or macOS.
Compile and package on Linux or macOS as follows:

```
GOOS=linux GOARCH=amd64 go build -o main main.go
zip main.zip main
```

Compile and package on Windows as follows:
1.1 Press **Windows + R** to open the "Run" window, enter **cmd**, and press **Enter**.
1.2 Run the following command to compile:

```
set GOOS=linux
set GOARCH=amd64
go build -o main main.go
```

1.3 Use a packaging tool to package the output binary file, which should be placed in the root directory of the zip package.

Log Description

Last updated：2024-12-02 18:12:22

Logging

You can use the following statements in the program to output the log:

`fmt.Println`

Or, use a method like `fmt.Sprintf`

For example, you can query the output in the function log by running the following code.

```
package main

import (
    "context"
    "fmt"
    "github.com/tencentyun/scf-go-lib/cloudfunction"
)

type DefineEvent struct {
    // test event define
    Key1 string `json:"key1"`
    Key2 string `json:"key2"`
}

func hello(ctx context.Context, event DefineEvent) (string, error) {
    fmt.Println("key1:", event.Key1)
    fmt.Println("key2:", event.Key2)
    return fmt.Sprintf("Hello %s!", event.Key1), nil
}

func main() {
    // Make the handler available for Remote Procedure Call by Cloud Function
    cloudfunction.Start(hello)
}
```

Log Query

Currently, the function logs can be uploaded to Tencent Cloud SCF. You can complete the configuration as instructed in Log Delivery Configuration.

You can search function execution logs on the log query page of the SCF or CLS console. For more information on how to query logs, see Log Search Guide.

Note:

Function logs uploaded to logset and log topic can be both queried with a function configured.

Custom Log Fields

Currently, the content output by simple log print statements in the function code will be recorded in the `SCF_Message` field when it is delivered to CLS. For descriptions of CLS fields, see Log Delivery Configuration.

Currently, SCF supports adding custom fields to logs that are uploaded to CLS. The custom fields allow you to output business fields and relevant data to logs, and track them using the log search feature of CLS.

Note:

If you need to query the key value of a custom field such as `SCF_CustomKey: SCF` , add a key-value index to the log topic for SCF log delivery as instructed in Configuring Indexes.

To avoid function log query failures caused by misuse of the index configuration, the default destination topic for SCF log delivery (prefixed with `SCF_LogTopic_`) does not support modifying the index configuration. You need to set the destination topic to custom delivery first and then update the log topic's index configuration.

After the index configuration is modified for a log topic, it will take effect only for newly written data.

Output

If a function outputs a single-line log in JSON, the log will be parsed and uploaded to CLS in the `field:value` format. Only the first layer will be parsed, and the remaining nested structure will be recorded as values.

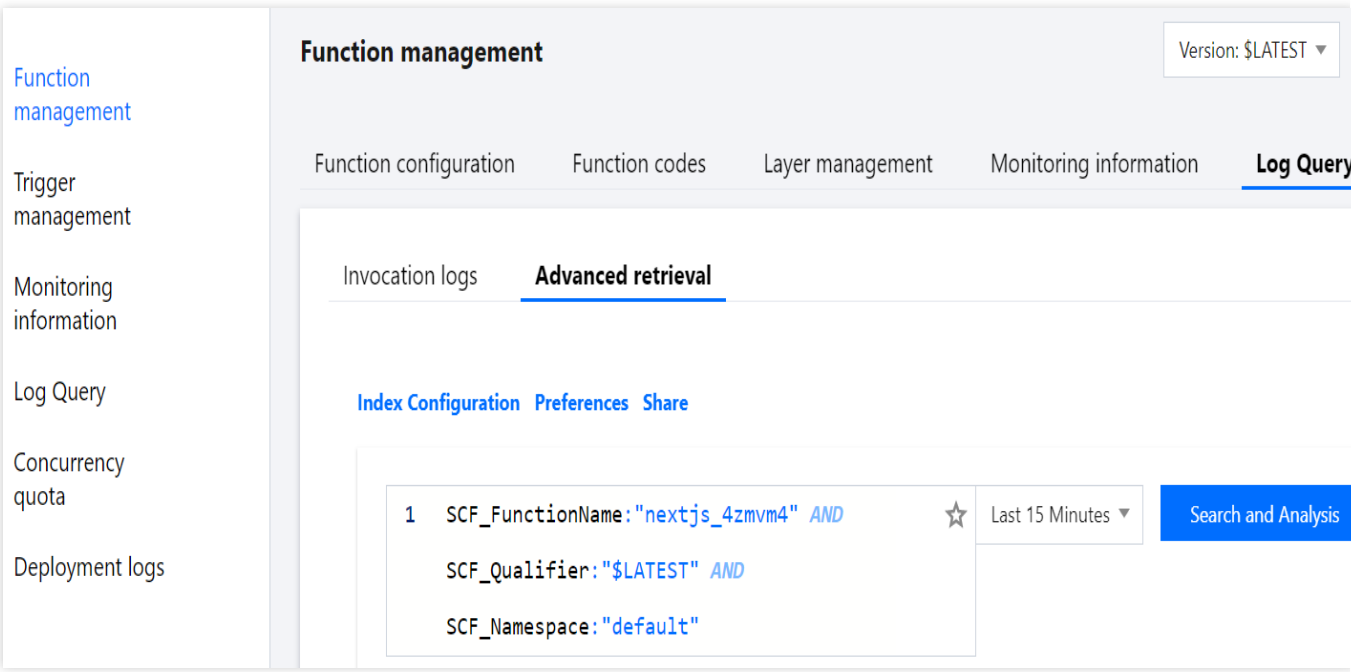
Run the following codes and check results:

```
package main

import (
    "context"
    "fmt"
    "github.com/tencentyun/scf-go-lib/cloudfunction"
)

type DefineEvent struct {
    Key1 string `json:"key1"`
    Key2 string `json:"key2"`
}

func hello(ctx context.Context, event DefineEvent) (string, error) {
    m := map[string]string{"key1": "test value 1", "key2": "test value 2"}
        data, _ := json.Marshal(m)
        fmt.Println(string(data))
```

```
    return fmt.Sprintf("hello world"), nil
}

func main() {
    cloudfunction.Start(hello)
}
```

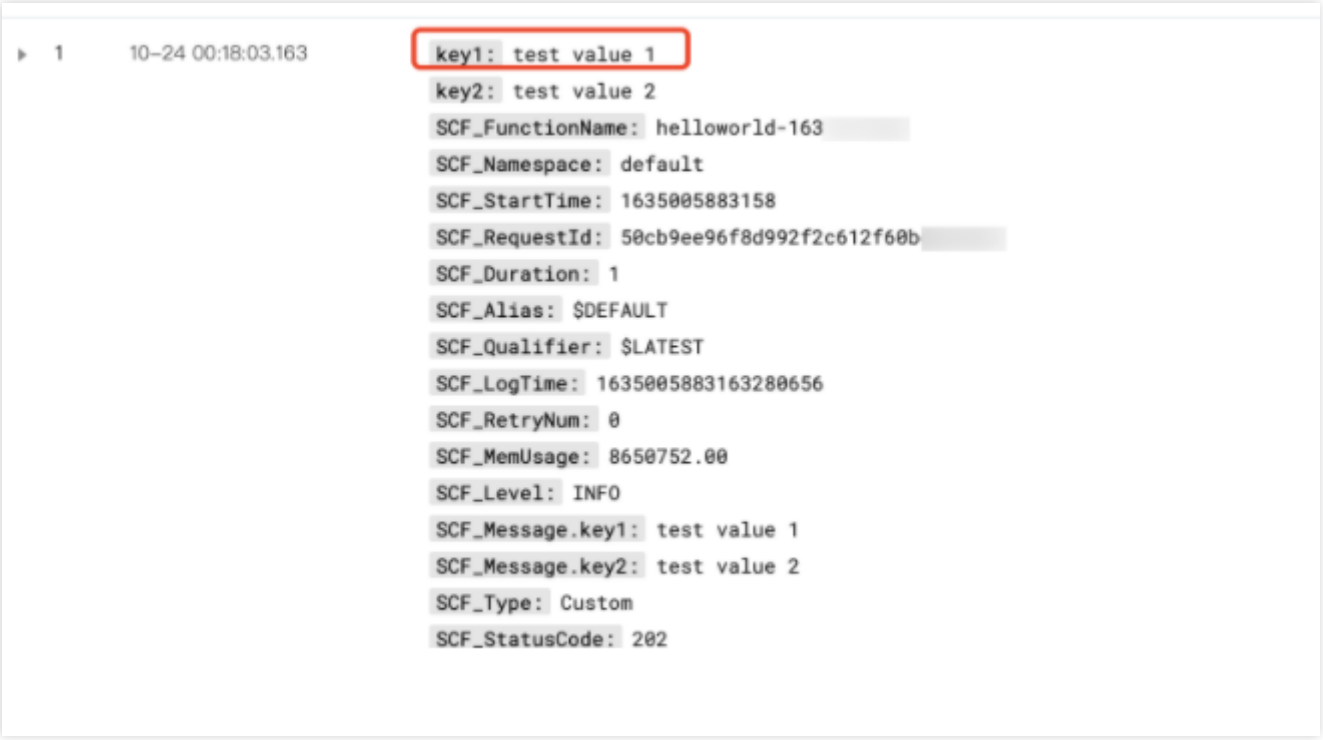
Search

After using the above code to perform a test, you can run the following statement to search in **Function Management > Log Query > Advanced Search**:



Search for logs

After the test is written to CLS, you can find the `key1` field in the log query as shown below:



PHP
Runtime Environment

Last updated：2025-02-12 17:34:45

PHP Version Selection

Currently, SCF supports the following versions of PHP programming language:
PHP 8.0
PHP 7.4
PHP 7.2
PHP 5.6
You can choose a desired runtime environment when creating a function, such as PHP 8.0, 7.4, 7.2, or 5.6.

Environment Variables

The PHP environment variables built in the current PHP 8.0 and 7.4 runtime environments are as shown in the table below:

Environment Variable Key	Specific Value or Value Source
`PHP_INI_SCAN_DIR`	/opt/php_extension:/var/user/php_extension

The PHP environment variables built in the current PHP 7.2 and 5.6 runtime environments are as shown in the table below:

Environment Variable Key	Specific Value or Value Source
`PHP_INI_SCAN_DIR`	/var/user/php_extension:/opt/php_extension

For more information on environment variables, see Environment Variables.

List of Built-in Extensions

Note:
For PHP 7.4 and later, the platform no longer has additional built-in dependency libraries. For more information on the dependencies required by code execution, see Dependency Installation.

If the built-in extensions cannot meet your business requirements, you can install custom extensions as instructed in Dependency Installation.

You can print and view the installed extensions at any time by using the `print_r(get_loaded_extensions());` code.

The currently installed PHP extensions are listed below:

PHP Versions	PHP Extensions
PHP 8.0、PHP 7.4	Core, runkit7, date, libxml, openssl, pcre, sqlite3, zlib, bcmath, bz2, calendar, ctype, curl, dom, hash, fileinfo, filter, ftp, gd, SPL, iconv, json, mbstring, session, standard, mysqlnd, PDO, pdo_mysql, pdo_sqlite, Phar, posix, Reflection, mysqli, SimpleXML, soap, sockets, exif, tidy, tokenizer, xml, xmlreader, xmlwriter, zip, eio, memcached, imagick, mongodb, protobuf, redis, swoole, Zend OPcache, runtime
PHP 7.2	Core, runkit7, date, libxml, openssl, pcre, sqlite3, zlib, bcmath, bz2, calendar, ctype, curl, dom, hash, fileinfo, filter, ftp, gd, SPL, iconv, json, mbstring, session, standard, mysqlnd, PDO, pdo_mysql, pdo_sqlite, Phar, posix, Reflection, mysqli, SimpleXML, soap, sockets, exif, tidy, tokenizer, xml, xmlreader, xmlwriter, zip, eio, memcached, imagick, mongodb, protobuf, redis, swoole, Zend OPcache, runtime
PHP 5.6	Core, runkit, date, ereg, libxml, openssl, pcre, sqlite3, zlib, bcmath, bz2, calendar, ctype, curl, dom, hash, fileinfo, filter, ftp, gd, SPL, iconv, json, mbstring, session, standard, mysqlnd, PDO, pdo_mysql, pdo_sqlite, Phar, posix, Reflection, mysqli, SimpleXML, soap, sockets, exif, tidy, tokenizer, xml, xmlreader, xmlwriter, zip, eio, memcached, imagick, mongodb, protobuf, redis, Zend OPcache, runtime

Development Methods

Last updated：2024-12-02 18:12:22

Function Format

The PHP function format is generally as follows:

```
<?php
function main_handler($event, $context) {
    print_r ($event);
    print_r ($context);
    return "hello world";
}
?>
```

Execution Method

You need to specify the execution method when creating a SCF function. If the PHP programming language is used, the execution method is similar to `index.main_handler`, where `index` indicates that the executed entry file is `index.php`, and `main_handler` indicates that the executed entry function is `main_handler`.
When submitting the ZIP code package by uploading the ZIP file locally or through COS, make sure that the root directory of the ZIP package contains the specified entry file, which contains the entry function specified by the definition, file name, function name, and execution method; otherwise, execution will fail as the entry file or entry function cannot be found.

Input Parameters

The input parameters in the PHP environment include `$event` and `$context`.
$event: This parameter is used to pass the trigger event data.
$context: This parameter is used to pass runtime information to your handler.
The event parameter varies with trigger or event source. For more information on its data structure, see Trigger Overview.
Note:
The input parameters `event` and `context` in PHP 8.0, 7.4, 7.2, and 5.6 are in `object` format.

Response

Your handler can use `return` to return a value. The return value will be handled differently depending on the type of invocation when the function is invoked.
In the PHP environment, a serializable object such as `dict` object can be directly returned:
Sync invocation: The return value of a sync invocation will be serialized in JSON and returned to the caller for subsequent processing. The function testing in the console is sync invocation, which can capture the function's return value and display it after the invocation is completed.
Async invocation: The return value of an async invocation will be discarded, since the invocation method responds right after the function is triggered, without waiting for function execution to complete.
Note:
The return value of both sync and async invocations will be recorded in the function logs. The return value will be written to the function invocation log `SCF_Message` in the format of `Response RequestId:xxx RetMsg:xxx` .
The value of `SCF_Message` is limited to 8 KB in length, and excessive parts will be truncated.

Exception Handling

You can exit the function by calling `die()` . At this point, the function will be marked as execution failed, and the output from the exit using `die()` will also be recorded in the log.

Deployment Methods

Last updated：2024-12-02 18:12:22

Deployment method

Tencent Cloud SCF provides the following function deployment methods. For more information about how to create and update a function, see Create and Update a Function.
Uploading and deploying a ZIP package, as instructed in Installing and Deploying Dependencies
Editing and deploying functions via the console, as instructed in Deployment Through Console.
Using the command line, as instructed in Deployment Through Serverless Framework CLI.

Installing and Deploying Dependencies

Currently, the SCF standard PHP only supports writing to the `/tmp` directory, and other directories are read-only. Therefore, you need to install, package and upload the local dependent library for use. The PHP dependency package can be uploaded with function codes to the cloud.

Locally installing dependency packages

Dependency manager

In PHP, dependencies can be managed with the composer package manager.

Directions

1. Create a local folder `/code` to store the codes and dependent files, and create the dependency package configuration file `composer.json` in the code root directory and configure the dependency information. Here takes the installation of `requests` as an example, and the `composer.json` file is as follows:

```
{
    "require": {
        "rmccue/requests": ">=1.0"
    }
}
```

2. Run the following command in the '/code' folder to install according to the dependent package and version specified in the configuration file.

```
composer install
```

Note:

Because the function is running on CentOS 7, install the dependency package in the same environment to avoid errors. For detailed directions, see Using Container Image.

Packaging and uploading

You can upload dependencies together with the project, and use them through the `require` statement in function codes.

The zip package for deploying functions can be generated automatically by a local folder via the console or manually. All the packaging should be under the project directory to place codes and dependencies in the root directory of the zip package. For more information, see Packaging requirements.

Log Description

Last updated：2024-12-02 18:12:22

Log Development

You can use the following statements in the program to output a log:

echo or echo()

print or print()

print_r()

var_dump()

For example, you can query the output content in the function log by running the following code:

```
<?php
function main_handler($event, $context) {
    print_r ($event);
    print_r ($context);
    echo "hello world\\n";
    print "hello world\\n";
    var_dump ($event);
    return "hello world";
}
?>
```

Log Query

Currently, all function logs are delivered to CLS. You can configure the function log delivery. For more information, see Log Delivery Configuration.

You can query function execution logs on the log query page of SCF or CLS. For more information on the log query method, see Log Search Guide.

Note:

Function logs are delivered to the `LogSet` log set and `LogTopic` log topic in CLS, both of which can be queried through the function configuration.

Custom Log Fields

Currently, the string content output by simple `print` or `logger` in the function code will be recorded in the `SCF_Message` field when it is delivered to CLS. For descriptions of CLS fields, see Log Delivery Configuration.
At present, SCF supports adding custom fields to the content output to CLS. By doing so, you can output business fields and related data content to logs and use the search capability of CLS to query and track them in the execution process.

Note:

If you need to query the key value of a custom field such as `SCF_CustomKey: SCF` , add a key-value index to the log topic for SCF log delivery as instructed in Configuring Indexes.
To avoid function log query failures caused by misuse of the index configuration, the default destination topic for SCF log delivery (prefixed with `SCF_LogTopic_`) does not support modifying the index configuration. You need to set the destination topic to custom delivery first and then update the log topic's index configuration.
After the index configuration is modified for a log topic, it will take effect only for newly written data.

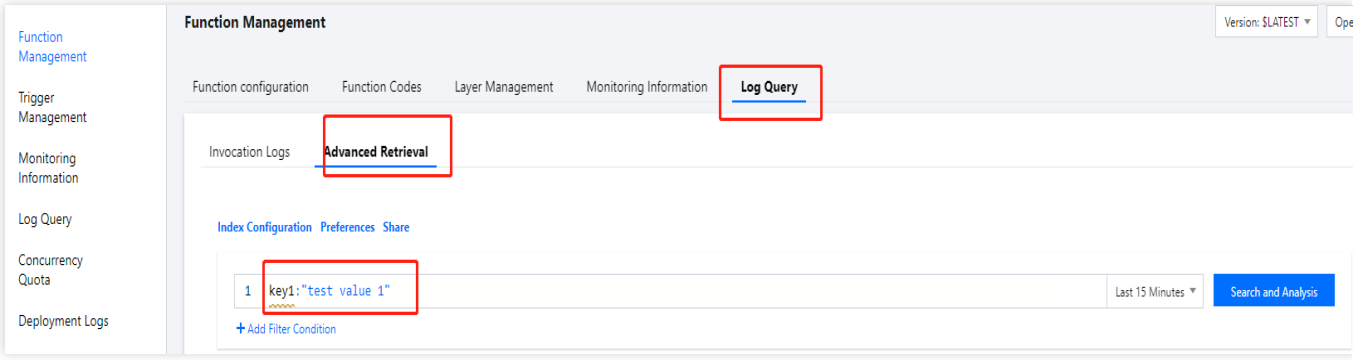
Output method

If a single-line log output by a function is in JSON format, the JSON content will be parsed into the format of **field:value** when it is delivered to CLS. Only the first level of the JSON content can be parsed in this way, while other nested structures will be recorded as values.
You can run the following code to test:

```
<?php
function main_handler($event, $context) {
    $custom_key = array('key1' => 'test value 1', 'key2' => 'test value 2');
    echo json_encode($custom_key);
    return "hello world";
}
?>
```

Search method

After using the above code to perform a test, you can run the following statement to search in **Function Management > Log Query > Advanced Search**:



Search result

After the test is written to CLS, you can find the `key1` field in the log query as shown below:

Examples

Last updated：2024-12-02 18:12:22

These examples provide code snippets based on PHP 7.2 for your reference.

You can click scf-php-code-snippet to obtain the relevant code snippets and directly use them for deployment.

Obtaining Environment Variables

This example shows you how to obtain all or a single environment variable.

```
<?php
function main_handler($event, $context) {
    print_r($_ENV);
    echo getenv('SCF_RUNTIME');
    return "hello world";
}
?>
```

Formatting Local Time

This example shows you how to output date and time in the specified format.

The SCF environment uses the UTC format by default. To output in Beijing time, you can add the `TZ=Asia/Shanghai` environment variable to the function, and use `date_default_timezone_set(getenv('TZ'));` in the function code to set the needed time zone, as shown below:

```
<?php
function main_handler($event, $context) {
    date_default_timezone_set(getenv('TZ'));
    echo date("Y-m-d H:i:s",time());
    return "hello world";
}
?>
```

Initiating Network Connections in a Function

```
<?php
function main_handler($event, $context) {
    $url = 'https://cloud.tencent.com';
    echo file_get_contents($url);
    return "hello world";
}
?>
```

Java
Environment Description

Last updated：2024-12-02 18:12:22

Java Version Selection

Currently, SCF supports the following versions of Java programming language:
Java 11 (Kona JDK)
Java 8 (Open JDK)
You can choose Java 8 or Java 11 as the runtime environment when creating a function.
SCF Java 11 is provided based on Tencent Kona. Tencent Kona is based on OpenJDK and maintained, optimized, and safeguarded by Tencent's professional technical team. The Tencent Cloud team has further developed and optimized Kona's support and features in cloud scenarios, making it more suitable for Java cloud businesses and delivering the best Java cloud production environment and solution.

Environment Variables

The environment variables built in the Java 8 and Java 11 runtime environments are as shown in the table below:

Java 11

Environment Variable Key	Specific Value or Value Source
`CLASSPATH`	/var/runtime/java11:/var/runtime/java11/lib/*

Java 8

Environment Variable Key	Specific Value or Value Source
`CLASSPATH`	/var/runtime/java8:/var/runtime/java8/lib/*:/opt

For more information on environment variables, see Environment Variables.

Notes

As the Java language can be executed in JVM only after compilation, it is used in SCF in a different way from scripting languages such as Python and Node.js, with the following restrictions:
Code upload is not supported: When Java is used, only developed, compiled, and packaged ZIP or JAR packages can be uploaded. The SCF environment does not provide Java compiling capability.
Online editing is not supported: Because code cannot be uploaded, online code editing is not supported. The code page of a Java runtime function only lists the ways to upload the code through the page or submit the code through

COS.

Notes on Java

Last updated：2024-12-02 18:12:22

Code Form

The code form of a SCF function developed in Java is generally as follows:

```
package example;

public class Hello {
    public String mainHandler(KeyValueClass kv) {
        System.out.println("Hello world!");
        System.out.println(String.format("key1 = %s", kv.getKey1()));
        System.out.println(String.format("key2 = %s", kv.getKey2()));
        return String.format("Hello World");
    }
}
```

Create the parameter `KeyValueClass` class:

```
package example;
public class KeyValueClass {
    String key1;
    String key2;

    public String getKey1() {
        return this.key1;
    }
    public void setKey1(String key1) {
        this.key1 = key1;
    }
    public String getKey2() {
        return this.key2;
    }
    public void setKey2(String key2) {
        this.key2 = key2;
    }

    public KeyValueClass() {
    }
}
```

Execution Method

As Java has the concept of package, its execution method is different from other languages and requires package information. The corresponding execution method in the code example is `example.Hello::mainHandler`, where `example` is identified as the Java package, `Hello` the class, and `mainHandler` the class method.

Input Parameters and Returns

In the sample code, the input parameters used by `mainHandler` are of POJO type, and the response is of string type. Currently, types supported for event input parameters and function responses include Java base types and POJO type, the function runtime is of `com.qcloud.scf.runtime.Context` type, and its associated library files can be downloaded here.

Types supported for event input and response parameters

Event Input Parameter	Response Parameter Type
Java base types	These include eight basic types and wrapper classes (`byte`, `int`, `short`, `long`, `float`, `double`, `char`, and `boolen`) and `String` type.
POJO (Plain Old Java Object) type	You should use variable POJOs and public getters and setters to provide implementations of the corresponding types in the code.

Context input parameters

To use Context, you need to use `com.qcloud.scf.runtime.Context;` in the code to reference the package and bring the jar package when it is packaged.
If this object is not used, you can ignore it in the function input parameters, which can be written as `public String mainHandler(String name)`.

Note:

The event structures of input parameters passed in by certain triggers have been defined and can be used directly. You can get and use the Java libraries through the Cloud Event Definition. If you have any questions during use, you can submit an issue or ticket for assistance.

Deployment Methods

Last updated：2024-12-02 18:12:22

This document describes how to create a zip deployment package with Gradle and create a jar deployment package with Maven. Then, you can upload the created package (if less than 50 MB) directly in the SCF console, or upload it to a COS bucket and then specify the bucket and object information in the SCF console.

Creating zip Deployment Package with Gradle

This document describes how to create a Java-type SCF function deployment package with Gradle. As long as the created zip package conforms to the following rules, it can be recognized and invoked by the SCF runtime environment.
The compiled package, class files and resource files are located in the root directory of the zip package.
The jar package required by the dependencies is located in the /lib directory.

Preparing environment

Make sure that you have Java and Gradle installed.
For Java 11, install TencentKona 11 or JDK 11.
For Java 8, install JDK 8. You can download and install the JDK appropriate for your system by using OpenJDK (Linux) or through Java.

Installing Gradle

For Gradle installation details, see Gradle Installation. The manual installation steps are as follows:
1. Download Gradle's binary package or full package with documentation and source code.
2. Unzip the package to a desired directory, such as `C:\\Gradle` (Windows) or `/opt/gradle/gradle-4.1` (Linux).
3. Add the path of the `bin` directory in the unzipped directory to the system environment variable `PATH` in the following way as appropriate:
Linux: Add through `export PATH=$PATH:/opt/gradle/gradle-4.1/bin` .
Windows: Right click Computer and select **Properties > Advanced system settings > Advanced > Environment Variables**, select the `Path` variable, click Edit, and add `;C:\\Gradle\\bin;` at the end of the variable value.
4. Run the following command on the command line to check whether Gradle is installed correctly.

```
gradle -v
```

If the following is output, Gradle is properly installed. If you have any questions, see Gradle's official documentation.

```
------------------------------------------------------------
```

```
Gradle 4.1
------------------------------------------------------------
Build time:   2017-08-07 14:38:48 UTC
Revision:     941559e020f6c357ebb08d5c67acdb858a3defc2
Groovy:       2.4.11
Ant:          Apache Ant(TM) version 1.9.6 compiled on June 29 2015
JVM:          1.8.0_144 (Oracle Corporation 25.144-b01)
OS:           Windows 7 6.1 amd64
```

Preparing code

Preparing code file

1. Create a project folder in the selected location, such as `scf_example` .

2. In the root directory of the project folder, create a directory `src/main/java/` as the directory where the package is stored.

3. Create an `example` package directory in the created directory and then create a `Hello.java` file in the package directory. The final directory structure is formed as follows:

```
scf_example/src/main/java/example/Hello.java
```

4. Enter the following code content in the `Hello.java` file:

```
package example;
public class Hello {
 public String mainHandler(String name, Context context) {
     System.out.println("Hello world!");
     return String.format("Hello %s.", name);
 }
}
```

Preparing compilation file

Create a `build.gradle` file in the root directory of the project folder and enter the following content:

```
apply plugin: 'java'

task buildZip(type: Zip) {
    from compileJava
    from processResources
    into('lib') {
        from configurations.runtime
    }
}

build.dependsOn buildZip
```

Using Maven Central library to handle package dependencies

If you need to reference the external package of Maven Central, you can add dependencies as needed. The content of the `build.gradle` file is as follows:

```
apply plugin: 'java'

repositories {
    mavenCentral()
}

dependencies {
    compile (
        'com.tencentcloudapi:scf-java-events:0.0.2'
    )
}

task buildZip(type: Zip) {
    from compileJava
    from processResources
    into('lib') {
        from configurations.runtime
    }
}

build.dependsOn buildZip
```

After `repositories` is used to indicate that the dependent library source is mavenCentral, Gradle will pull the dependencies from Maven Central during compilation, i.e., the `com.tencentcloudapi:scf-java-events:0.0.2` package specified in `dependencies` .

Using local jar package to handle package dependencies

If you have already downloaded the Jar package to your local system, you can use the local library to handle package dependencies. In this case, create a `jars` directory in the root directory of the project folder and place the downloaded dependent Jar package into it. The content of the `build.gradle` file is as follows:

```
apply plugin: 'java'

dependencies {
    compile fileTree(dir: 'jars', include: '*.jar')
}

task buildZip(type: Zip) {
    from compileJava
```

```
    from processResources
    into('lib') {
        from configurations.runtime
    }
}

build.dependsOn buildZip
```

After `dependencies` is used to indicate that the search directory is the `*.jar` file in the jars directory, the dependencies will be automatically searched for during compilation.

Compiling and packaging

Execute the command `gradle build` in the root directory of the project folder, and the compilation output should be like the example below:

```
Starting a Gradle Daemon (subsequent builds will be faster)

BUILD SUCCESSFUL in 5s
3 actionable tasks: 3 executed
```

If a compilation failure is displayed, adjust the code based on the outputted compilation error message.
The compiled zip package is located in the `/build/distributions` directory of the project folder and named `scf_example.zip` after the project folder.

Creating jar Deployment Package Using Maven

This document describes how to create a Java-type function deployment jar package with Maven.

Preparing environment

Make sure that you have Java and Gradle installed.
For Java 11, install TencentKona 11 or JDK 11.
For Java 8, install JDK 8. You can download and install the JDK appropriate for your system by using OpenJDK (Linux) or through Java.

Installing Maven

For Maven installation details, see Installing Apache Maven. The manual installation steps are as follows:
1. Download Maven's zip or tar.gz package.
2. Unzip the package to a desired directory, such as `C:\\Maven` (Windows) or `/opt/mvn/apache-maven-3.5.0` (Linux).

3. Add the path of the `bin` directory in the unzipped directory to the system environment variable `PATH` in the following way as appropriate:
Linux: Add through `export PATH=$PATH:/opt/mvn/apache-maven-3.5.0/bin` .
Windows: Right click Computer and select **Properties > Advanced system settings > Advanced > Environment Variables**, select the `Path` variable, click Edit, and add `;C:\\Maven\\bin;` at the end of the variable value.
4. Run the following command on the command line to check whether Maven is installed correctly.

```
mvn -v
```

If the following is output, Maven is properly installed. If you have any questions, see Maven's official documentation.

```
Apache Maven 3.5.0 (ff8f5e7444045639af65f6095c62210b5713f426; 2017-04-
04T03:39:06+08:00)
Maven home: C:\\Program Files\\Java\\apache-maven-3.5.0\\bin\\..
Java version: 1.8.0_144, vendor: Oracle Corporation
Java home: C:\\Program Files\\Java\\jdk1.8.0_144\\jre
Default locale: zh_CN, platform encoding: GBK
OS name: "windows 7", version: "6.1", arch: "amd64", family: "windows"
```

Preparing code

Preparing code file

1. Create a project folder in the selected location, such as `scf_example` .
2. In the root directory of the project folder, create a directory `src/main/java/` as the directory where the package is stored.
3. Create an `example` package directory in the created directory and then create a `Hello.java` file in the package directory. The final directory structure is formed as follows:

```
scf_example/src/main/java/example/Hello.java
```

4. Enter the following code content in the `Hello.java` file:

```
package example;
public class Hello {
 public String mainHandler(String name, Context context) {
     System.out.println("Hello world!");
     return String.format("Hello %s.", name);
 }
}
```

Preparing compilation file

Create a `pom.xml` file in the root directory of the project folder and enter the following content:

Java 11

Java 8

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

  <groupId>examples</groupId>
  <artifactId>java-example</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>java-example</name>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>2.3</version>
        <configuration>
          <source>11</source>
          <target>11</target>
        </configuration>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

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

  <groupId>examples</groupId>
  <artifactId>java-example</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>java-example</name>
```

```
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>2.3</version>
        <configuration>
          <createDependencyReducedPom>false</createDependencyReducedPom>
        </configuration>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Using Maven Central library to handle package dependencies

If you need to reference the external package of Maven Central, you can add dependencies as needed. The content of the `pom.xml` file is as follows. To add dependencies, pay attention to the `<dependencies>` section.

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

  <groupId>examples</groupId>
  <artifactId>java-example</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>java-example</name>

  <dependencies>
    <dependency>
      <groupId>com.tencentcloudapi</groupId>
      <artifactId>scf-java-events</artifactId>
      <version>0.0.2</version>
    </dependency>
  </dependencies>
```

```
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>2.3</version>
        <configuration>
          <createDependencyReducedPom>false</createDependencyReducedPom>
        </configuration>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Compiling and packaging

Execute the command `mvn package` in the root directory of the project folder, and the compilation output should be like the example below:

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building java-example 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
...
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.785 s
[INFO] Finished at: 2017-08-25T10:53:54+08:00
[INFO] Final Memory: 17M/214M
[INFO] ------------------------------------------------------------------------
```

If a compilation failure is displayed, adjust the code based on the outputted compilation error message.

The compiled zip package is located in the `target` directory of the project folder and named `java-example-1.0-SNAPSHOT.jar` after the artifactId and version fields in pom.xml.

Using Maven to Create JAR Packs

Last updated：2024-12-02 18:12:22

Logging

You can use the `System.out.println()` and `java.util.logging.Logger()` statements in the program to output a log:
For example, you can query the output in the function log by running the following code.

```
System.out.println("Hello world!");
```

```
Logger logger = Logger.getLogger("AnyLoggerName");
logger.setLevel(Level.INFO);
logger.info("logging message here!");
```

Log Query

Currently, the function logs can be uploaded to Tencent Cloud SCF. You can complete the configuration as instructed in Log Delivery Configuration.
You can search function execution logs on the log query page of the SCF or CLS console. For more information on how to query logs, see Log Search Guide.
Note:
Function logs uploaded to logset and log topic can be both queried with a function configured.

Custom Log Fields

Currently, the content output by simple log print statements in the function code will be recorded in the `SCF_Message` field when it is delivered to CLS. For descriptions of CLS fields, see Log Delivery Configuration.
Currently, SCF supports adding custom fields to logs that are uploaded to CLS. The custom fields allow you to output business fields and relevant data to logs, and track them using the log search feature of CLS.
Note:
If you need to query the key value of a custom field such as `SCF_CustomKey: SCF` , add a key-value index to the log topic for SCF log delivery as instructed in Configuring Indexes.
To avoid function log query failures caused by misuse of the index configuration, the default destination topic for SCF log delivery (prefixed with `SCF_LogTopic_`) does not support modifying the index configuration. You need to set

the destination topic to custom delivery first and then update the log topic's index configuration.
After the index configuration is modified for a log topic, it will take effect only for newly written data.

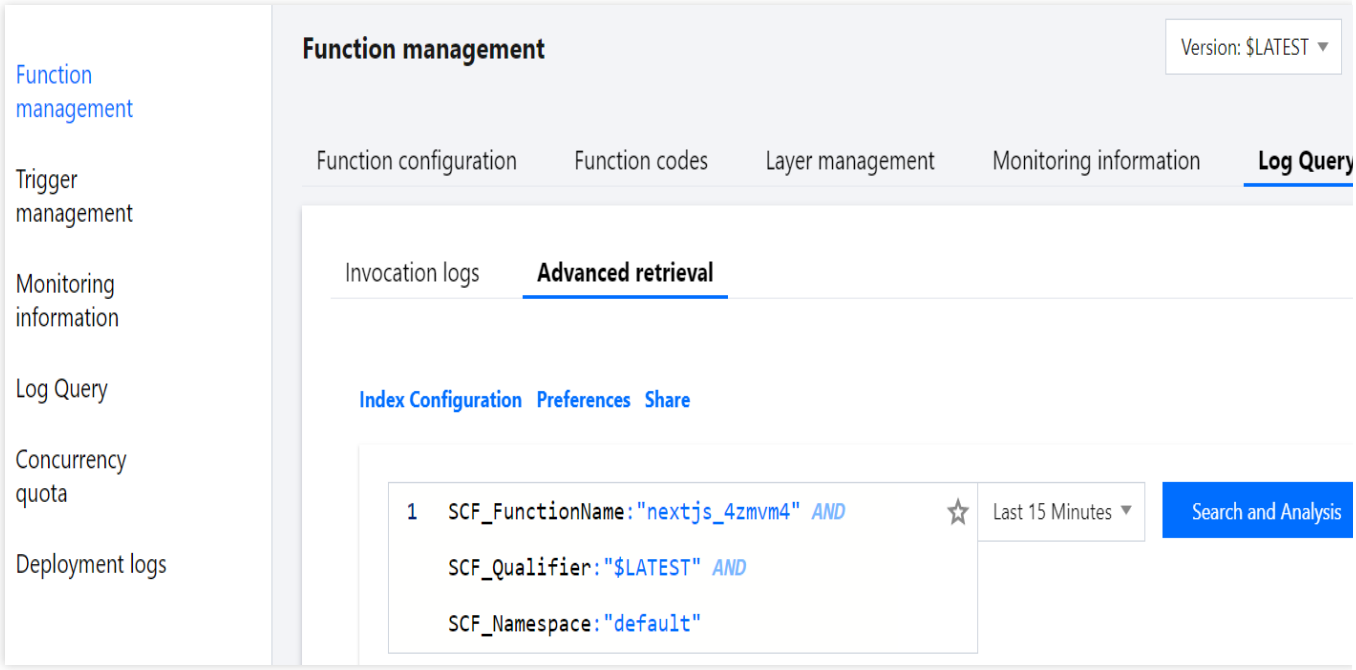
Output

If a function outputs a single-line log in JSON, the log will be parsed and uploaded to CLS in the `field:value` format. Only the first layer will be parsed, and the remaining nested structure will be recorded as values.
Run the following codes and check results:

```
package example;

public class Hello {
    public String mainHandler(KeyValueClass kv) {
        System.out.println("{\\"key1\\": \\"test value 1\\",\\"key2\\": \\"test val
      return String.format("hello world");
    }
}
```

Search

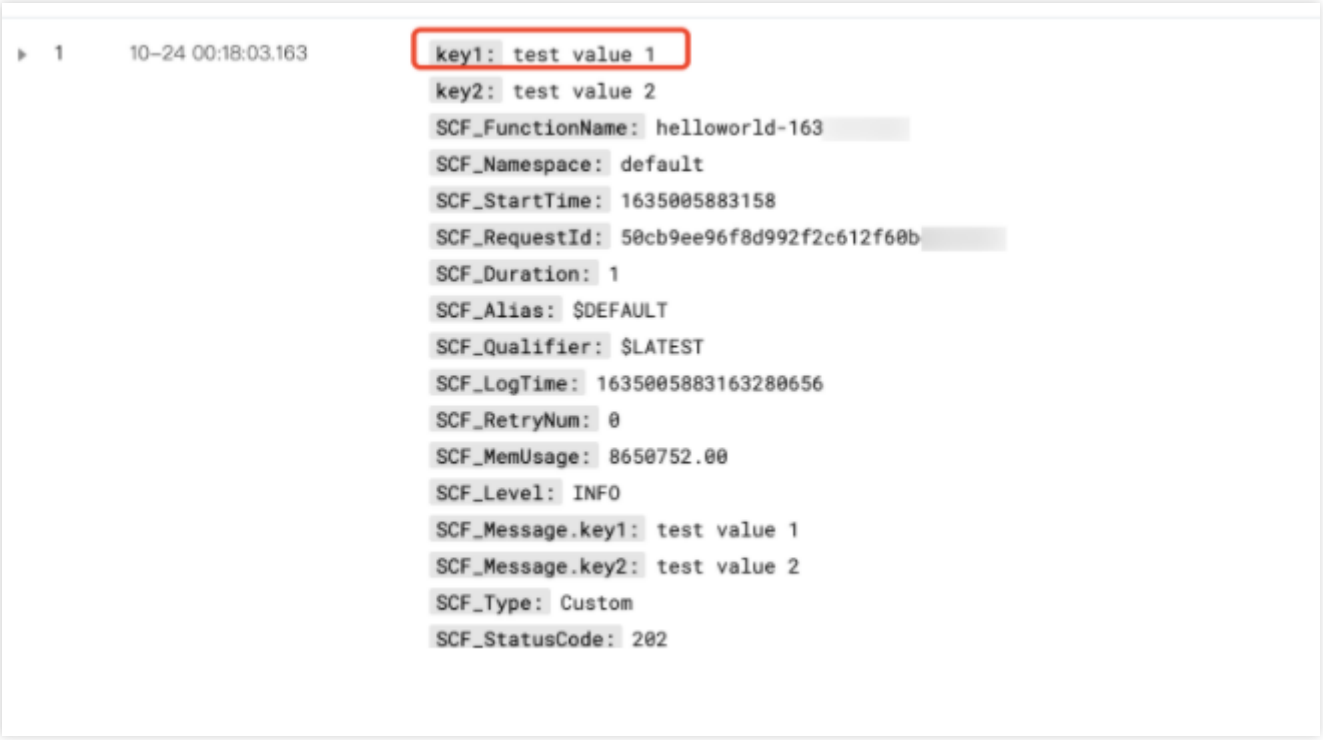
After using the above code to perform a test, you can run the following statement to search in **Function Service > Log Query > Advanced Search**:



For detailed directions, see Log Search Guide.

Search for logs

After the test is written to CLS, you can find the `key1` field in the log query as shown below:



Common Examples

Last updated：2024-12-02 18:12:22

Scenario

With POJO type parameters, you can handle more complex data structures than simple event input parameters. This document uses an example to describe how to use POJO parameters in SCF and which input parameter formats are supported.

Prerequisites

You have logged in to the SCF console.

Directions

Event input parameters and POJOs

The event input parameters used in this document are as follows:

```
{
  "person": {"firstName":"bob","lastName":"zou"},
  "city": {"name":"shenzhen"}
}
```

With the input parameters above, the following content is outputted:

```
{
  "greetings": "Hello bob zou.You are from shenzhen"
}
```

Based on the input parameters, the following four classes are built:

RequestClass: Used to accept the event as the class that accepts events.

PersonClass: Used to handle the `person` section in the event JSON.

CityClass: Used to handle the `city` section in the event JSON.

ResponseClass: Used to assemble the response content.

Prepare the code

Prepare the code by following the steps below for the four classes constructed based on the input parameters and entry function:

Step 1. Prepare the project directory

Create a project root directory, such as `scf_example` .

Step 2. Prepare the code directory

1. Create a folder `src\\main\\java` as the code directory in the project root directory.
2. According to the name of the package to be used, create a package folder in the code directory.
For example, create `example` to form the directory structure `scf_example\\src\\main\\java\\example` .

Step 3. Prepare the code

Create files `Pojo.java` , `RequestClass.java` , `PersonClass.java` , `CityClass.java` , and `ResponseClass.java` in the `example` folder with the following contents respectively:

Pojo.java

RequestClass.java

PersonClass.java

CityClass.java

ResponseClass.java

```
package example;

public class Pojo{
    public ResponseClass handle(RequestClass request){
        String greetingString = String.format("Hello %s %s.You are from %s", reques
        return new ResponseClass(greetingString);
    }
}
```

```
package example;

public class RequestClass {
    PersonClass person;
    CityClass city;

    public PersonClass getPerson() {
        return person;
    }

    public void setPerson(PersonClass person) {
```

```
        this.person = person;
    }

    public CityClass getCity() {
        return city;
    }

    public void setCity(CityClass city) {
        this.city = city;
    }

    public RequestClass(PersonClass person, CityClass city) {
        this.person = person;
        this.city = city;
    }

    public RequestClass() {
    }

}
```

```
package example;

public class PersonClass {
    String firstName;
    String lastName;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public PersonClass(String firstName, String lastName) {
        this.firstName = firstName;
```

```
        this.lastName = lastName;
    }

    public PersonClass() {
    }

}
```

```
package example;

public class CityClass {
    String name;


    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public CityClass(String name) {
        this.name = name;
    }

    public CityClass() {
    }

}
```

```
package example;

public class ResponseClass {
    String greetings;

    public String getGreetings() {
        return greetings;
    }

    public void setGreetings(String greetings) {
        this.greetings = greetings;
    }
```

```
    public ResponseClass(String greetings) {
        this.greetings = greetings;
    }

    public ResponseClass() {
    }

}
```

Code compilation

Note

In the example, Maven is chosen to compile and package the code. You can choose another packaging method according to your own conditions.

1. Create the pom.xml function in the project root directory and enter the following content:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>examples</groupId>
  <artifactId>java-example</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>java-example</name>

  <build>
 <plugins>
   <plugin>
     <groupId>org.apache.maven.plugins</groupId>
     <artifactId>maven-shade-plugin</artifactId>
     <version>2.3</version>
     <configuration>
       <createDependencyReducedPom>false</createDependencyReducedPom>
     </configuration>
     <executions>
       <execution>
         <phase>package</phase>
         <goals>
           <goal>shade</goal>
         </goals>
       </execution>
     </executions>
```

```
    </plugin>
  </plugins>
   </build>
</project>
```

2. Run the `mvn package` command on the command line and make sure that there is a successful compilation message. If the output result is as follows, the packaging is successful:

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.800 s
[INFO] Finished at: 2017-08-25T15:42:41+08:00
[INFO] Final Memory: 18M/309M
[INFO] ------------------------------------------------------------------------
```

If the compilation fails, please modify the code as prompted.

3. The generated package after compilation is located at `target\\java-example-1.0-SNAPSHOT.jar` .

Function creation and testing

1. Create a function as instructed in Create a Function.

2. Upload the compiled package as a submission package.

You can choose to upload the packaging by zipping it or uploading it to a COS bucket and then submitting it through COS bucket upload.

3. Set the execution method of the function to `example.Pojo::handle` .

4. Enter the input parameters that are expected to be processed in the **Test event** template on the **Functions** page:

```
{  "person": {"firstName":"bob","lastName":"zou"},  "city": {"name":"shenzhen"}}
```

After clicking **Execute**, you can see the returned result:

```
{  "greetings": "Hello bob zou.You are from shenzhen"}
```

You can also modify the values of the structures in the test input parameters, and you can see the modification effect after execution.

Sample

You can go to scf-demo-java to pull the sample code for testing.

Using Gradle to Create ZIP Packs

Last updated：2024-12-02 18:12:22

This document describes how to create a Java-type SCF function deployment package with Gradle. As long as the created zip package conforms to the following rules, it can be recognized and invoked by the SCF runtime environment.
The compiled package, class files, and resource files are located in the root directory of the zip package.
The jar package required by the dependencies is located in the `/lib` directory.

Environment Preparations

Make sure that you have installed Java and Gradle. For the Java version, please use JDK 8. You can download and install the JDK appropriate for your system through OpenJDK (Linux) or at `www.java.com` .

Installing Gradle

The specific installation instructions can be found at `https://gradle.org/install/` . The following describes how to install it manually:
1. Download Gradle's binary package or full package with documentation and source code.
2. Unzip the package to a desired directory, such as `C:\\Gradle` (Windows) or `/opt/gradle/gradle-4.1` (Linux).
3. Add the path of the `bin` directory in the unzipped directory to the system environment variable `PATH` in the following way as appropriate:
Linux: add through `export PATH=$PATH:/opt/gradle/gradle-4.1/bin` .
Windows: right click Computer and select `Properties > Advanced system settings > Advanced > Environment Variables` , select the `Path` variable, click Edit, and add `;C:\\Gradle\\bin;` at the end of the variable value.
4. Run the following command on the command line to check whether Gradle is installed correctly.

```
gradle -v
```

If the following is output, Gradle is properly installed. If you have any questions, please see Gradle's official documentation.

```
------------------------------------------------------------
Gradle 4.1
------------------------------------------------------------
Build time:   2017-08-07 14:38:48 UTC
Revision:     941559e020f6c357ebb08d5c67acdb858a3defc2
```

```
Groovy:       2.4.11
Ant:          Apache Ant(TM) version 1.9.6 compiled on June 29 2015
JVM:          1.8.0_144 (Oracle Corporation 25.144-b01)
OS:           Windows 7 6.1 amd64
```

Code Preparations

Preparing code file

1. Create a project folder in the selected location, such as `scf_example` .

2. In the root directory of the project folder, create a directory `src/main/java/` as the directory where the package is stored.

3. Create an `example` package directory in the created directory and then create a `Hello.java` file in the package directory. The final directory structure is formed as follows:

```
scf_example/src/main/java/example/Hello.java
```

4. Enter the following code content in the `Hello.java` file:

```
package example;
public class Hello {
 public String mainHandler(String name, Context context) {
     System.out.println("Hello world!");
     return String.format("Hello %s.", name);
 }
}
```

Preparing the compilation file

Create a `build.gradle` file in the root directory of the project folder and enter the following content:

```
apply plugin: 'java'

task buildZip(type: Zip) {
    from compileJava
    from processResources
    into('lib') {
        from configurations.runtime
    }
}

build.dependsOn buildZip
```

Using Maven Central library to handle package dependencies

If you need to reference the external package of Maven Central, you can add dependencies as needed. The content of the `build.gradle` file is as follows:

```
apply plugin: 'java'

repositories {
    mavenCentral()
}

dependencies {
    compile (
        'com.tencentcloudapi:scf-java-events:0.0.2'
    )
}

task buildZip(type: Zip) {
    from compileJava
    from processResources
    into('lib') {
        from configurations.runtime
    }
}

build.dependsOn buildZip
```

After `repositories` is used to indicate that the dependent library source is mavenCentral, Gradle will pull the dependencies from Maven Central during compilation, i.e., the `com.tencentcloudapi:scf-java-events:0.0.2` package specified in `dependencies` .

Using local Jar package to handle package dependencies

If you have already downloaded the Jar package to your local system, you can use the local library to handle package dependencies. In this case, create a `jars` directory in the root directory of the project folder and place the downloaded dependent Jar package into it. The content of the `build.gradle` file is as follows:

```
apply plugin: 'java'

dependencies {
    compile fileTree(dir: 'jars', include: '*.jar')
}

task buildZip(type: Zip) {
    from compileJava
    from processResources
    into('lib') {
```

```
        from configurations.runtime
    }
}

build.dependsOn buildZip
```

After `dependencies` is used to indicate that the search directory is the `*.jar` file in the jars directory, the dependencies will be automatically searched for during compilation.

Compilation and Packaging

Run the `gradle build` command in the root directory of the project folder, and the compilation output should be like the example below:

```
Starting a Gradle Daemon (subsequent builds will be faster)

BUILD SUCCESSFUL in 5s
3 actionable tasks: 3 executed
```

If a compilation failure is displayed, adjust the code based on the output compilation error message.
The compiled zip package is located in the `/build/distributions` directory of the project folder and named `scf_example.zip` after the project folder.

Function Usage

After the zip package is generated after compilation and packaging, when creating or modifying a function, you can upload the package (if less than 10 MB) through the page or upload it (if bigger) to a COS bucket and then update it into the function through COS upload.

Custom Runtime Overview

Last updated：2024-12-02 18:12:22

In addition to the standard runtime environments for supported programming languages and versions, to satisfy personalized needs of function implementation in more custom programming languages and versions, SCF also provides the Custom Runtime service, which allows you to customize runtime environments. It can implement custom function runtime by opening up its capabilities, enable you to use any programming language on any version to write functions as needed, and implement global operations in function invocation, such as extension load, security plugin, and monitoring agent. SCF and Custom Runtime respond to and process events over HTTP.

Custom Runtime Deployment File Description

Bootstrap file: fixed executable bootstrap file of Custom Runtime. You need to create an executable file with the same name and implement it with a custom programming language and version. It can be directly processed or start another executable file to initialize and invocate the function runtime.
Function file: function program file developed and implemented with a custom programming language and version.
Library files or executable files dependent on by the runtime: relevant dependent library files or executable files required by the runtime in the custom programming language and version.
The function is published in the form of deployment package, which consists of the following files:
Bootstrap file (required)
Function file (required)
Library files or executable files dependent on by runtime (optional)
As the deployment package size is limited, if a library file or executable file dependent on by the runtime is large, we recommend you publish the function by binding the deployment package to applicable layers, which involves the following files:
Deployment package:
Bootstrap file (required)
Function file (required)
Layer
Library files or executable files dependent on by runtime (optional)

Note:

Before publishing the executable files among the above deployment files to SCF, you should set the file executable permission for them, package the deployment files into a ZIP package, and upload the package directly or through COS.

Custom Runtime Operating Mechanism

Custom Runtime divides the function runtime into initialization stage and invocation stage. Initialization is executed only once during the instance cold start, while invocation is the execution process called for every event response. The start time and execution time vary by programming language and version. The **initialization timeout period** and **execution timeout period** configuration items are added to SCF specially for Custom Runtime to manage the runtime lifecycle.

Loading function bootstrap

SCF first searches for the executable bootstrap file in the deployment package and performs the following operations based on the search result:
If the bootstrap file is found and executable, SCF will load and execute it to enter the function initialization stage.
If the bootstrap file cannot be found or is not executable, a message indicating that the bootstrap file does not exist and the start failed will be returned.

Initializing function

The bootstrap file is executed to start function initialization. You can customize the bootstrap to implement custom operations as needed and directly process it or call another executable file to complete initialization. We recommend you perform the following basic operations during initialization:
Set the paths and environment variables of the runtime's dependent libraries.
Load the dependent library files and extensions of the custom programming language and version. If there are dependent files that need to be pulled in real time, you can download them to the `/tmp` directory.
Parse the function file and execute the global operations or initialization processes (such as initializing SDK client (HTTP client) and creating database connection pool) required before function invocation, so they can be reused during invocation.
Start plugins such as security and monitoring.
After initialization is completed, you need to proactively call the runtime API to access the initialization readiness API `/runtime/init/ready` , which informs SCF that Custom Runtime has been initialized and is ready; otherwise, SCF will keep waiting until the configured initialization timeout period elapses and then end Custom Runtime and return an initialization timeout error. If the notifications are repeated, the first access time will be used as the readiness time.

Logs and exceptions

SCF logs all standard outputs during initialization.
If function initialization is executed and completed normally within the timeout period, the logs generated during initialization will be combined and returned together with the logs of the first invocation. If initialization failed due to an error or exception within the timeout period, an initialization timeout error will be returned as the execution result, and

program errors written into the standard outputs and exception logs will be reported to SCF and displayed in logs and log query in the console.

Function invocation

In the function invocation stage, you need to customize event acquisition, function invocation, and result return and loop this process.

Get events through long polling. You can use the custom programming language and version to access the event acquisition API `/runtime/invocation/next` of the runtime API with HTTP client. The response body contains the event data. If this API is repeatedly accessed during an invocation, the same event data will be returned.

Note:

Do not set a timeout period for the GET method of the HTTP client.

Construct function invocation parameters based on the environment variables, required information in the response header, and event information.

Push parameter data such as event information and call the function processing program.

Access the runtime response result API `/runtime/invocation/response` to push the processing result of the function. The first invocation success will be considered as the final event status, which will be locked by SCF, and the result cannot be changed after push.

If an error occurs during function invocation, you can call the runtime invocation error API `/runtime/invocation/error` to push the error message. The current invocation will end, and the first invocation will be considered as the final event status, which will be locked by SCF, and the result cannot be changed in subsequent pushes.

Release the resources that are no longer needed after the current invocation.

Logs and exceptions

SCF logs all standard outputs during invocation.

After SCF distributes an event, if Custom Runtime does not get it after the function execution timeout period elapses, SCF will end the instance and return an event acquisition wait timeout error.

After SCF distributes an event, if Custom Runtime gets it but does not return the execution result after the function execution timeout period elapses, SCF will end the instance and return an execution timeout error.

Custom Runtime API

You need to implement Custom Runtime with your custom programming language and version, and Custom Runtime and SCF need to communicate with each other over a standard protocol during processes such as event distribution and result returning. Therefore, SCF provides runtime APIs to meet the interaction needs in the lifecycle of Custom Runtime.

SCF has the following built-in environment variables:

SCF_RUNTIME_API: runtime API address.

SCF_RUNTIME_API_PORT: runtime API port.

For more information, please see Environment Variables.

Custom Runtime can access runtime APIs through `SCF_RUNTIME_API:SCF_RUNTIME_API_PORT` .

Path	Method	Description
/runtime/init/ready	post	Calls the API to mark the ready status after runtime initialization.
/runtime/invocation/next	get	Gets invocation event. The response header contains the following information request_id: request ID, which identifies the request triggering function invocation. memory_limit_in_mb: maximum function memory in MB time_limit_in_ms: function timeout period in milliseconds For the structures of the event data contained in the response body, please see Trigger Event Message Structure Summary.
/runtime/invocation/response	post	Function processing result.After calling the function processing program, the runtime will push the response from the function to the invocation response API.
/runtime/invocation/error	post	A function return error will be pushed to the invocation error API to mark the current invocation failure.

Creating Sample Bash Function

Last updated：2024-12-02 18:12:22

Scenario

This document describes how to create, package, and release a Custom Runtime cloud function to respond to the triggered event. You will learn the development process and operating mechanism of Custom Runtime.

Directions

Before creating a Custom Runtime cloud function, you need to create a runtime boot file bootstrap and function file.

Creating a bootstrap file

Bootstrap is a runtime entry bootloader. When Custom Runtime loads a function, it retrieves the file named “bootstrap” and executes the file to start Custom Runtime, which allows developers to develop runtime functions using any programming language and version. Bootstrap must:
Have the execute permission.
Be able to run in the SCF system environment (CentOS 7.6).
You can refer to the following sample code to create a bootstrap file in a command line terminal. Bash is used as an example in this document.

```
#! /bin/bash
set -euo pipefail

# Initialization - Load the function file.
source ./"$(echo $_HANDLER | cut -d. -f1).sh"

# After the initialization is completed, access the runtime API to report the
readiness.
curl -d " " -X POST -s
"http://$SCF_RUNTIME_API:$SCF_RUNTIME_API_PORT/runtime/init/ready"

### Start the loop that listens to events, handles events, and pushes event
handling results.
while true
do
  HEADERS="$(mktemp)"
  # Gets event data through long polling
```

```
  EVENT_DATA=$(curl -sS -LD "$HEADERS" -X GET -s
"http://$SCF_RUNTIME_API:$SCF_RUNTIME_API_PORT/runtime/invocation/next")
  # Invokes a function to handle the event
  RESPONSE=$($(echo "$_HANDLER" | cut -d. -f2) "$EVENT_DATA")
  # Pushes the function handling result
  curl -X POST -s
"http://$SCF_RUNTIME_API:$SCF_RUNTIME_API_PORT/runtime/invocation/response"  -d
"$RESPONSE"
done
```

Sample description

In the preceding sample, Custom Runtime has two phases, the initialization and invocation phases. Initialization is executed only once during the cold start of a function instance. After the initialization, the invocation loop starts, which listens to events, invokes functions for handling, and pushes the handling results.

Initialization phase

For more information, see Initializing function.

After the initialization, you need to proactively invoke the runtime API to report the readiness to SCF. The sample code is as follows:

```
# After the initialization is completed, access the runtime API to report the
readiness.
curl -d " " -X POST -s
"http://$SCF_RUNTIME_API:$SCF_RUNTIME_API_PORT/runtime/init/ready"
```

Because Custom Runtime is implemented with a custom programming language and version, a standard protocol is needed for the communication between Custom Runtime and SCF. In the current sample, SCF provides runtime APIs and built-in environment variables to Custom Runtime over HTTP. For more information, see Environment Variables.

`SCF_RUNTIME_API` : Runtime API address

`SCF_RUNTIME_API_PORT` : Runtime API port

Initialization logs and exceptions

For more information, see Logs and exceptions.

Invocation phase

For more information, see Function invocation.

1.1 After initialization, the invocation loop starts. A function is invoked to listen to events. The sample code is as follows:

```
# Gets event data through long polling
  EVENT_DATA=$(curl -sS -LD "$HEADERS" -X GET -s
"http://$SCF_RUNTIME_API:$SCF_RUNTIME_API_PORT/runtime/invocation/next")
```

During the long-polling of events, do not set timeout of the GET method. Access the runtime event acquisition API (`/runtime/invocation/next`) to wait for event delivery. If you access this API repeatedly within an invocation,

the same event data will be returned. The response body is `event_data`. The response header includes:

`Request_Id`: Request ID, identifying the request that triggers function invocation.

`Memory_Limit_In_Mb`: Function memory limit, in MB.

`Time_Limit_In_Ms`: Function timeout limit, in milliseconds.

1.2 Based on the environment variables, response header information, and event information, construct parameters of the function and start function invocation for event handling. The sample code is as follows:

```
# Invokes a function to handle the event
  RESPONSE=$($(echo "$_HANDLER" | cut -d. -f2) "$EVENT_DATA")
```

1.3 Access the runtime invocation response API to push the function handling result. The first invocation success will be considered as the final event status, which will be locked by SCF. The pushed result cannot be changed. The sample code is as follows:

```
# Pushes the function handling result
  curl -X POST -s
"http://$SCF_RUNTIME_API:$SCF_RUNTIME_API_PORT/runtime/invocation/response"  -d
"$RESPONSE"
```

If an error occurs during function invocation, access the runtime invocation error API to push the error message, which ends the current invocation. The first invocation result will be considered as the final event status, which will be locked by SCF. The pushed result cannot be changed. The sample code is as follows:

```
# Push the function handling error.
  curl -X POST -s
"http://$SCF_RUNTIME_API:$SCF_RUNTIME_API_PORT/runtime/invocation/error"  -d
"parse event error"
```

Invocation logs and exception

For more information, see Logs and exceptions.

Creating a function file

Note:

The function file contains the implementation of function logic. The execution method and parameters can be implemented by Custom Runtime.

Create `index.sh` in the command line terminal.

```
function main_handler () {
  EVENT_DATA=$1
  echo "$EVENT_DATA" 1>&2;
  RESPONSE="Echoing request: '$EVENT_DATA'"
  echo $RESPONSE
}
```

Releasing a Function

1. After the bootstrap file and function file are successfully created, the directory structure is as follows:

```
├ bootstrap
└ index.sh
```

2. Run the following command to grant execute permission on the bootstrap file:

Note:

Windows does not support the `chmod 755` command. Therefore, you need to run the command in Linux or macOS.

```
$ chmod 755 index.sh bootstrap
```

3. Create and publish functions by using Serverless Cloud Framework. Or, run the following command to generate a ZIP package and then create and publish functions through the SDK or SCF console.

```
$ zip demo.zip index.sh bootstrap
adding: index.sh (deflated 23%)
adding: bootstrap (deflated 46%)
```

Using Serverless Cloud Framework to create and publish a function

Creating a function

1. Install Serverless Cloud Framework.
2. Configure the `Serverless.yml` file in the bootstrap directory to create the dotnet function.

```
#Component information
component: scf # Component name. `scf` is used as an example.
name: ap-guangzhou_default_helloworld # Instance name.
#Component parameters
inputs:
  name: helloworld #Function name.
  src: ./
  description: helloworld blank template function.
  handler: index.main_handler
  runtime: CustomRuntime
  namespace: default
  region: ap-guangzhou
  memorySize: 128
  timeout: 3
  events:
    - apigw:
        parameters:
```

```
            endpoints:
              - path: /
                method: GET
```

Note:

For more information on the configurations of SCF components, see Configuration Documentation.

3. Run the `scf deploy` command to create a cloud function. A successful creation returns the following message:

```
serverless-cloud-framework
Action: "deploy" - Stage: "dev" - App: "ap-guangzhou_default_helloworld" -
Instance: "ap-guangzhou_default_helloworld"
functionName: helloworld
description:  helloworld blank template function.
namespace:    default
runtime:      CustomRuntime
handler:      index.main_handler
memorySize:   128
lastVersion:  $LATEST
traffic:      1
triggers:
  apigw:
    - http://service-xxxxxx-123456789.gz.apigw.tencentcs.com/release/
Full details: https://serverless.cloud.tencent.com/apps/ap-
guangzhou_default_helloworld/ap-guangzhou_default_helloworld/dev
36s › ap-guangzhou_default_helloworld › Success
```

Note:

For more information, see SCF Component.

Invoking a function

As `events` is set to `apigw` in the `serverless.yml` file, an API gateway is created together with the function. The cloud function can be accessed over this API gateway. If a message similar to the following is returned, the access is successful.

```
Echoing request:
'{
        "headerParameters":{},
        "headers":{
"accept":"text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,imag
e/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9",
        "accept-encoding":"gzip, deflate",
        "accept-language":"zh-CN,zh-TW;q=0.9,zh;q=0.8,en-US;q=0.7,en;q=0.6",
        "cache-control":"max-age=259200",
        "connection":"keep-alive",
        "host":"service-eiu4aljg-1259787414.gz.apigw.tencentcs.com",
```

```
        "upgrade-insecure-requests":"1",
        "user-agent":"Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/85.0.4183.121 Safari/537.36",
        "x-anonymous-consumer":"true",
        "x-api-requestid":"b8b69e08336bb7f3e06276c8c9******",
        "x-api-scheme":"http",
        "x-b3-traceid":"b8b69e08336bb7f3e06276c8c9******",
        "x-qualifier":"$LATEST"},
        "httpMethod":"GET",
        "path":"/",
        "pathParameters":{},
        "queryString":{},
        "queryStringParameters":{},
        "requestContext":{"httpMethod":"GET","identity":{},"path":"/",
        "serviceId":"service-xxxxx",
        "sourceIp":"10.10.10.1",
        "stage":"release"
        }
}'
```

Using SDK to create and release a function

Creating a function

Run the following commands to use the Python SDK of SCF to create a function named `CustomRuntime-Bash` .

```
from tencentcloud.common import credential
from tencentcloud.common.profile.client_profile import ClientProfile
from tencentcloud.common.profile.http_profile import HttpProfile
from tencentcloud.common.exception.tencent_cloud_sdk_exception import
TencentCloudSDKException
from tencentcloud.scf.v20180416 import scf_client, models
from base64 import b64encode
try:
    cred = credential.Credential("SecretId", "secretKey")
    httpProfile = HttpProfile()
    httpProfile.endpoint = "scf.tencentcloudapi.com"

    clientProfile = ClientProfile()
    clientProfile.httpProfile = httpProfile
    client = scf_client.ScfClient(cred, "na-toronto", clientProfile)

    req = models.CreateFunctionRequest()
    f = open('demo.zip', 'r')
    code = f.read()
    f.close()
```

```
    params = '{\\"FunctionName\\":\\"CustomRuntime-Bash\\",\\"Code\\":
{\\"ZipFile\\":\\"'+b64encode(code)+'\\"},\\"Timeout\\":3,\\"Runtime\\":\\"Cust
omRuntime\\",\\"InitTimeout\\":3}'
    req.from_json_string(params)

    resp = client.CreateFunction(req)
    print(resp.to_json_string())

except TencentCloudSDKException as err:
    print(err)
```

Special parameters of Custom Runtime

Parameter Type	Description
`"Runtime":"CustomRuntime"`	Runtime type of Custom Runtime.
`"InitTimeout":3`	Initialization timeout period. Custom Runtime adds a configuration of initialization timeout period. The initialization period starts from the boot-time of bootstrap and ends when the runtime API is reported ready. When the initialization period exceeds the timeout period, the execution ends and an initialization timeout error is returned.
`"Timeout":3`	Invocation timeout period. This parameter configures the timeout period of function invocation. The invocation period starts from the event delivery time and ends upon the time when the function pushes the handling result to the runtime API. When the invocation period exceeds the timeout period, the execution ends and an invocation timeout error is returned.

Invoking a function

Run the following commands to use the Python SDK of SCF to invoke the CustomRuntime-Bash function.

```
from tencentcloud.common import credential
from tencentcloud.common.profile.client_profile import ClientProfile
from tencentcloud.common.profile.http_profile import HttpProfile
from tencentcloud.common.exception.tencent_cloud_sdk_exception import
TencentCloudSDKException
from tencentcloud.scf.v20180416 import scf_client, models
try:
    cred = credential.Credential("SecretId", "secretKey")
    httpProfile = HttpProfile()
    httpProfile.endpoint = "scf.tencentcloudapi.com"
```

```
    clientProfile = ClientProfile()
    clientProfile.httpProfile = httpProfile
    client = scf_client.ScfClient(cred, "na-toronto", clientProfile)

    req = models.InvokeRequest()
    params = '{\\"FunctionName\\":\\"CustomRuntime-
Bash\\",\\"ClientContext\\":\\"{   \\\\\\"key1\\\\\\": \\\\\\"test value
1\\\\\\",   \\\\\\"key2\\\\\\": \\\\\\"test value 2\\\\\\" }\\"}'
    req.from_json_string(params)

    resp = client.Invoke(req)
    print(resp.to_json_string())

except TencentCloudSDKException as err:
    print(err)
```

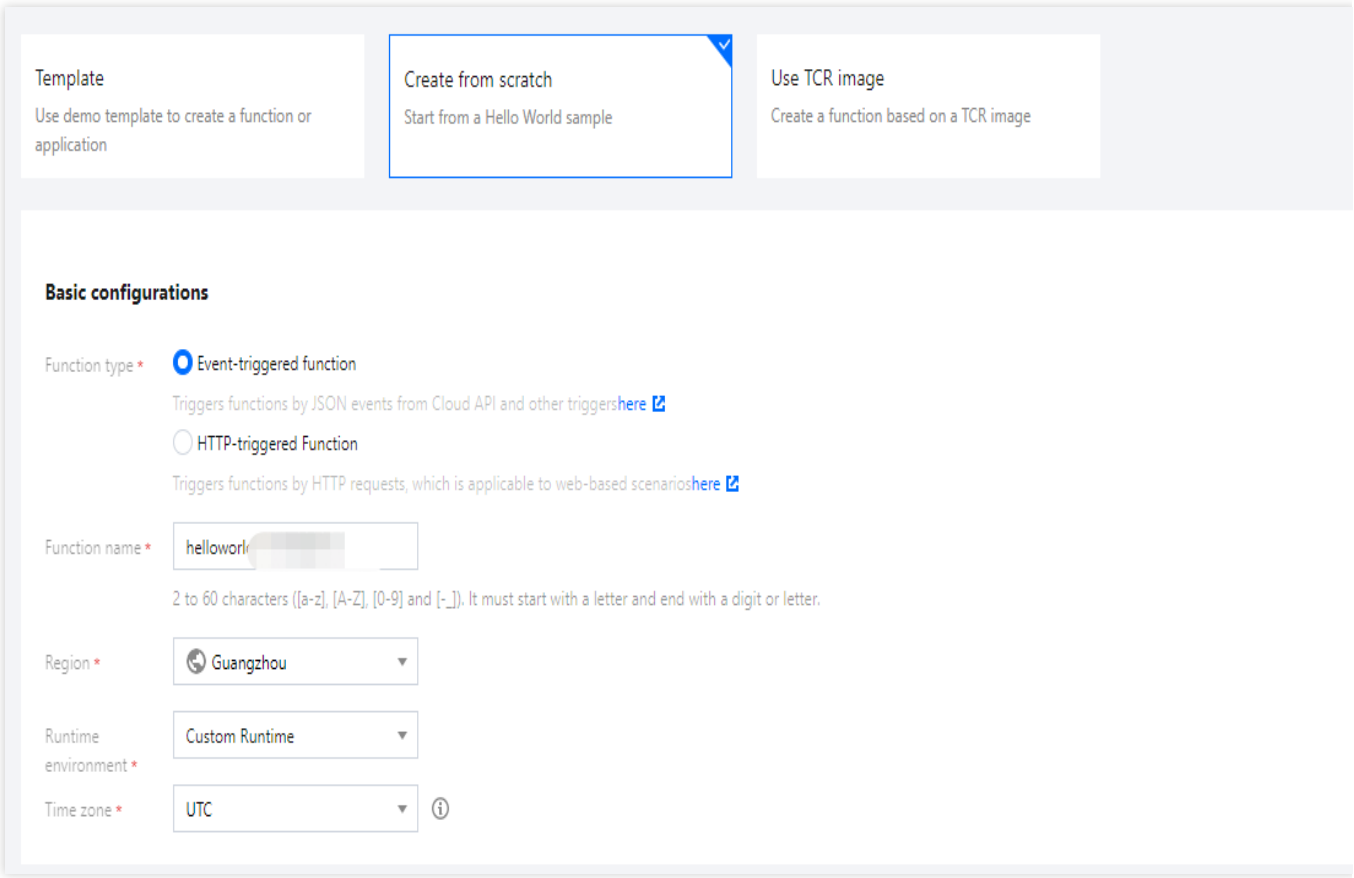
If a message similar to the following is returned, the invocation is successful.

```
{"Result":
    {"MemUsage": 7417***,
    "Log": "", "RetMsg":
    "Echoing request: '{
        \\"key1\\": \\"test value 1\\",
        \\"key2\\": \\"test value 2\\"
        }'",
    "BillDuration": 101,
    "FunctionRequestId": "3c32a636-****-****-****-d43214e161de",
    "Duration": 101,
    "ErrMsg": "",
    "InvokeResult": 0
    },
    "RequestId": "3c32a636-****-****-****-d43214e161de"
}
```

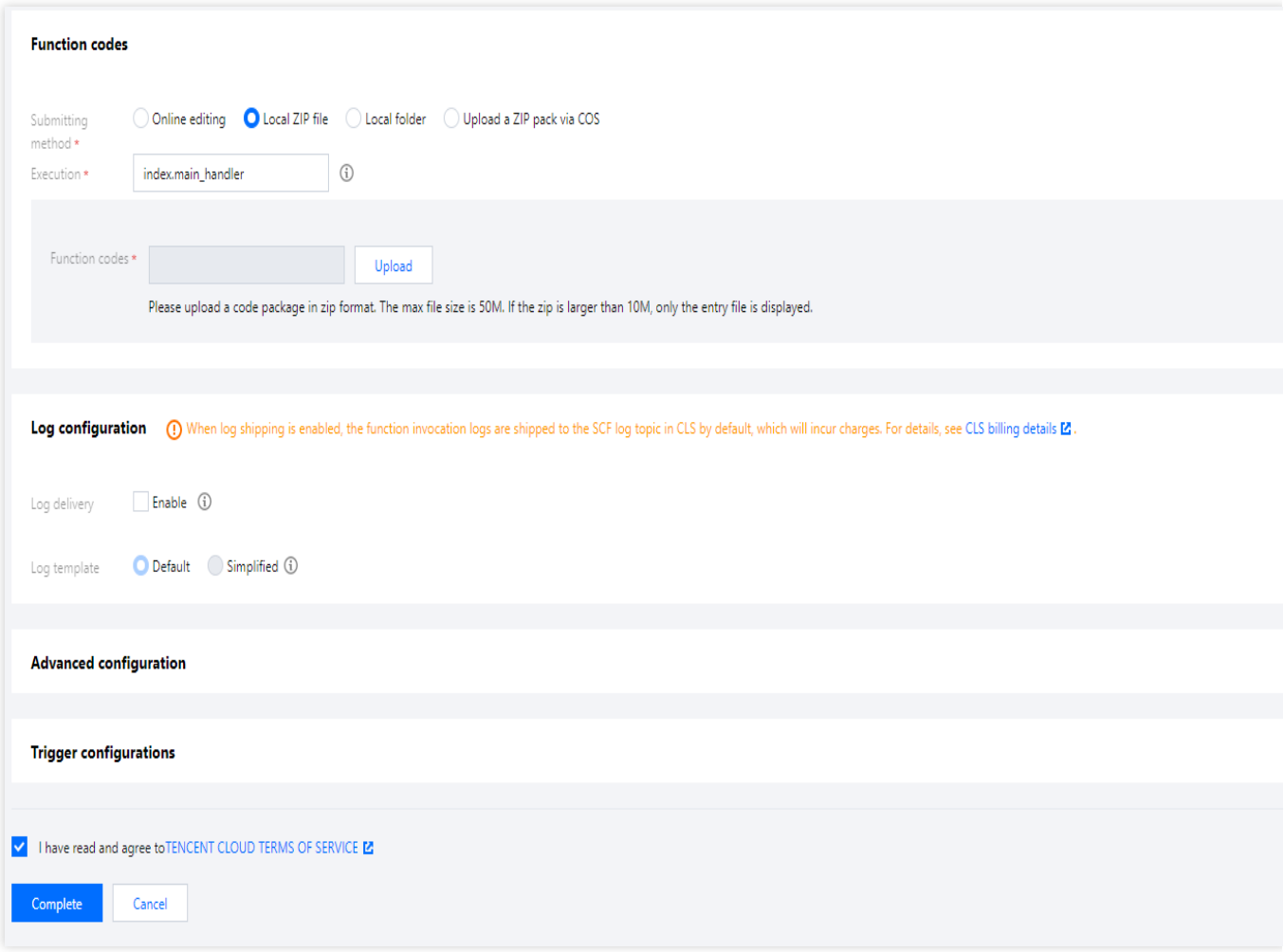
Creating and publishing a function in the console

Creating a function

1. Log in to the SCF console and click **Functions** on the left sidebar.
2. Choose a region at the top of the **Functions** page and click **Create** to start creating a function.
3. On the **Create function** page, click **Create from scratch** and select **Custom Runtime** in **Running environment**.



4. In **Function codes**, set **Submitting method** and **Function codes**.



Submitting method: Select **Local ZIP file**.

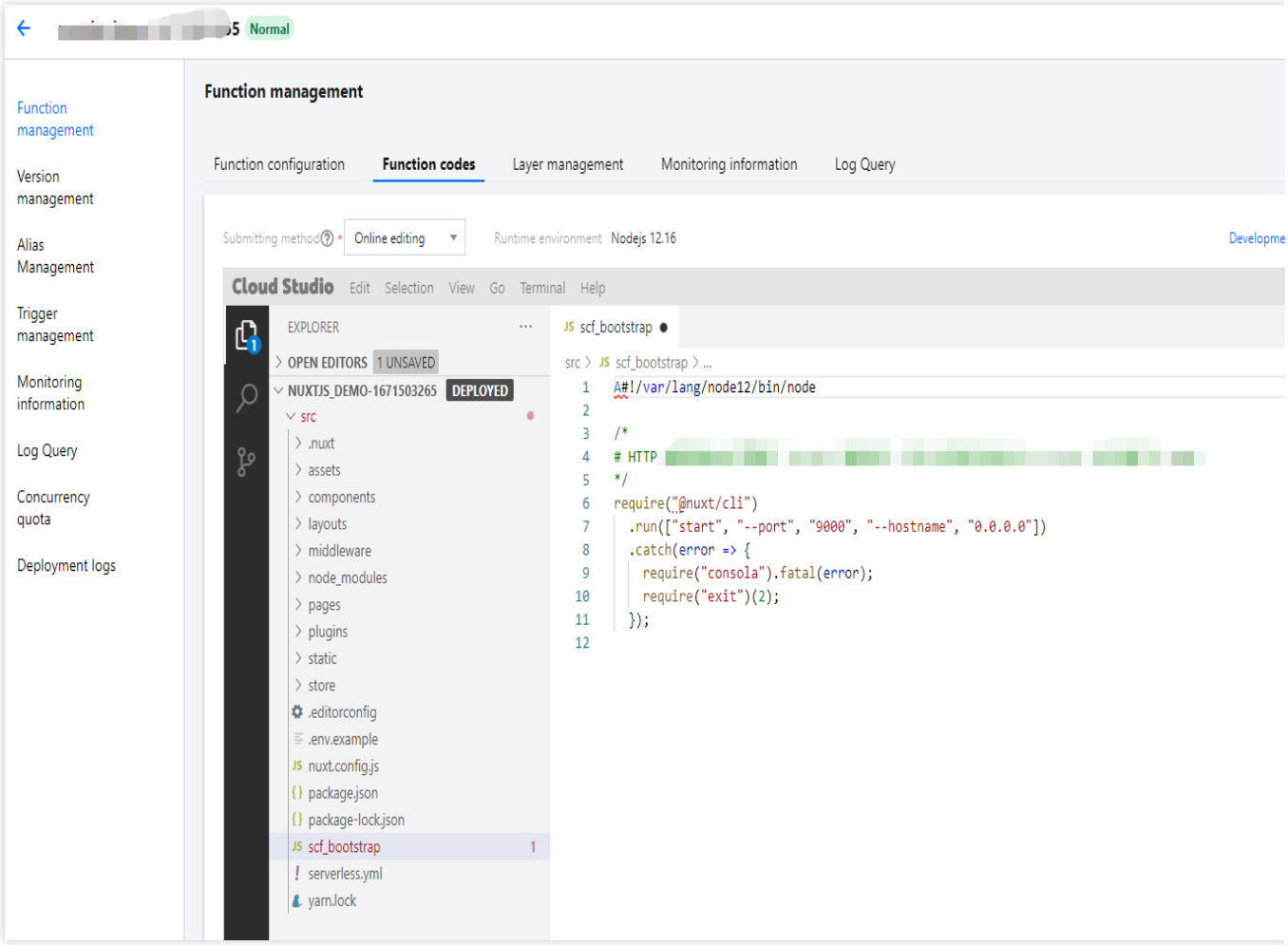
Function codes: Select the `demo.zip` .

Advanced settings: Expand **Advanced settings** and set **Initialization timeout period** as well as other related parameters.

5. Click **Complete**.

Invoking a function

1. Log in to the SCF console and click **Functions** on the left sidebar.
2. Choose a region at the top of the **Functions** page, and click the function to be invoked to go to the function detail page.
3. Select **Function management** on the left and select the **Function codes** tab.



4. Click **Test** below the editor, and the invocation execution result and log will be displayed in the console.

Deploying Image as Function
WebServer Image Function

Last updated：2024-12-02 18:12:22

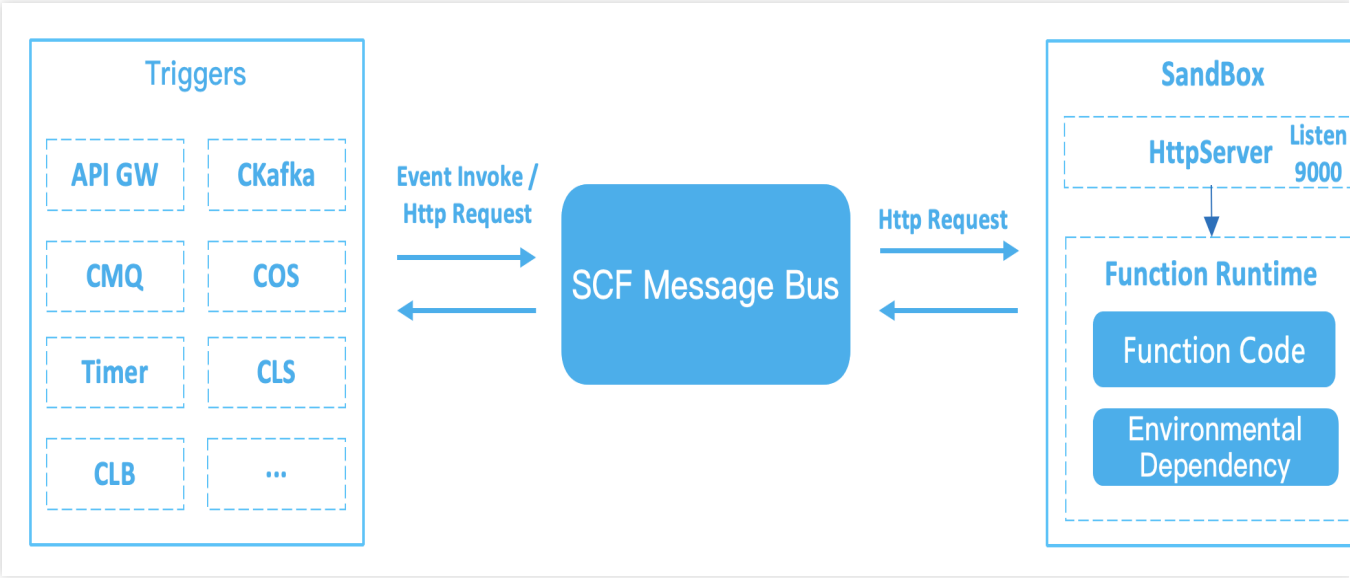
SCF allows you to deploy container images as functions. This document describes the background, principles, development, log printing, cold start optimization, billing, and use limits of image deployment functions.

Background

SCF is a FaaS service based on the cloud native architecture from the very beginning of design. After adding support for deploying container images as functions at the runtime layer, its entire service form has evolved towards a containerized ecosystem. On the one hand, it solves the environment dependency problem in function runtime and gives you more freedom to customize. On the other hand, this service form enables you to cross the technical thresholds, such as Kubernetes cluster management, security maintenance, and troubleshooting, and sinks auto scaling, availability, and other needs to the computing platform, further unleashing the capabilities of cloud computing.

How It Works

Before developing specific function logic, you need to determine the function type. SCF provides event-triggered functions and HTTP-triggered functions.
During the initialization of the function instance, SCF will obtain the temporary user name and password of the image repository as the access credential to pull the image. After the image is pulled successfully, start the HTTP Server you defined according to the specified startup command `Command`, the parameter `Args` and the port (fixed to 9000). Finally, HTTP Server will receive all entry requests of SCF, including the event-triggered function invocations and HTTP-triggered function invocations.
How a function works is as shown below:



Developing Function Deployed Based on Image

Building HTTP server

For a function deployed based on an image, you need to build an HTTP server and configure it as follows:

It should listen on `0.0.0.0:9000` or `*:9000`.

It should be started within 30 seconds.

If the above step is not completed, health check may time out, and the following error may be reported:

```
The request timed out in 30000ms.Please confirm your http server have enabled
listening on port 9000.
```

Function input parameters

event: POST request body (HTTP body)

The request body contains the event data. For its structure, see Trigger Event Message Structure Summary.

context: Request header (HTTP header)

Common parameters: Parameters used to identify the user and API signature, which must be carried in each request.

Use `X-Scf-Request-Id` to get the current request ID.

Note:

Both event-triggered and HTTP-triggered functions contain common headers.

The common request headers are generated by SCF, which mainly contain permissions and basic function information.

The input parameters are as detailed below:

Header Field	Description

X-Scf-Request-Id	Current request ID
X-Scf-Memory	Maximum memory that can be used during function instance execution
X-Scf-Timeout	Timeout period for function execution
X-Scf-Version	Function version
X-Scf-Name	Function name
X-Scf-Namespace	Function namespace
X-Scf-Region	Function region
X-Scf-Appid	`Appid` of function owner
X-Scf-Uin	`Uin` of function owner
X-Scf-Session-Token	Temporary `SESSION TOKEN`
X-Scf-Secret-Id	Temporary `SECRET ID`
X-Scf-Secret-Key	Temporary `SECRET KEY`
X-Scf-Trigger-Src	Timer (if a scheduled trigger is used)

Built-in environment variables

Environment variables built in the container in custom image-based deployment are different from those in code package-based deployment. You can import them as needed.

Environment Variable Key	Specific Value or Value Source
TENCENTCLOUD_RUNENV	SCF
USER_CODE_ROOT	/var/user/
USER	qcloud
SCF_FUNCTIONNAME	Function name
SCF_FUNCTIONVERSION	Function version
TENCENTCLOUD_REGION	Region
TENCENTCLOUD_APPID	Account `APPID`
TENCENTCLOUD_UIN	Account `UIN`

Function invocation

For event-triggered functions, you need to listen on the fixed path `/event-invoke` to receive function invocation requests.
For HTTP-triggered functions, you don't need to listen on a specified path; instead, API Gateway uses layer-7 reverse proxy to pass through the request path.

Function log printing

SCF collects standard output logs such as `stdout` and `stderr` generated in the container in a non-intrusive manner and reports them to the log module. After invoking a function, you can view the aggregated logs in the console.

Cold Start Optimization

As file layers such as basic environment and system dependency are added to the image, compared with code package-based function deployment where files are completely built-in, image-based function deployment requires extra file download and image decompression time. To further reduce the cold start time, we recommend you use the following practices:

Downsizing image

Create an image repository and a function in the same region, so that the image can be pulled over VPC when the function triggers image pull, which makes the pull faster and more stable.
The image should be as small as possible, that is, it should contain only the necessary basic environment and execution dependencies without any unnecessary files.

Enabling image acceleration

After image acceleration is enabled, the function platform will prefetch the image nearby through the internal acceleration mechanism. When calling a function instance, it will directly load and decompress the image from the cache, eliminating the time for downloading the image file. This expedites the launch by five times on average.

Directions

1. Log in to the SCF console and select **Functions** on the left sidebar.
2. On the **Functions** list page, select the name of the target image function.
3. Select **Function codes** on the **Function management** page.
4. On the **Function codes** page, click **Edit**.
5. Select ***Enable** in **Image acceleration**.
6. Click **Save**.

Notes

This feature is currently in beta test free of charge. Acceleration can be enabled for up to five functions in one region under each account.

Using provisioned concurrency

Use provisioned concurrency when deploying an image to start a function instance in advance and thus reduce the cold start time and optimize the user experience. For more information, see Provisioned Concurrency.

Billing

The billable items of image-based functions are the same as those of code package-based functions. For more information on billing, see Pay-As-You-Go (Postpaid).

Use Limits

Image size

Currently, only images below 1 GiB in size are supported. We recommend you select an appropriate function instance execution memory based on the image size. To increase the limit, submit a ticket for application.

Image Size (X)	Execution Memory (Y)
X < 256 MB	256 MB < Y < 512 MB
256 MB < X < 512 MB	512 MB < Y < 1 GiB
512 MB < X < 1 GiB	Y > 1 GiB

Image repository access

Currently, only the TCR Enterprise Edition and Personal Edition are supported. For more information, see Tencent Container Registry.
For more information on the TCR Enterprise Edition, see Basic Image Repository Operations.
For more information on the TCR Personal Edition, see Personal Getting Started (Old).
Currently, only images in a private image repository in the same region can be read.

Permission to read/write file in container

`/tmp` is readable and writable by default. We recommend you select `/tmp` when outputting a file.
Avoid using other users' files with restricted access or execution.
The storage space of the writable layer in the container is 512 MB.

CPU architecture limits for image build client

Currently, SCF functions run on the x86 architecture, so images built on the ARM platform (such as Apple Mac with M1 chip) are not supported.

Image build client limits

The client should meet one of the following requirements:

Docker Image Manifest v2, Schema 2 (Docker should be on v1.10 or later)

Open Container Initiative (OCI) Specifications (v1.0.0 or later)

Usage Method

Last updated：2024-12-02 18:12:22

This document describes how to use image to deploy function via the console.

Prerequisites

SCF supports the image repositories of TCR Enterprise Edition and Personal Edition. You can select the image repository as needed.
Purchase a TCR Enterprise Edition instance. For more information, see Quick Start.
Use a TCR Personal Edition image repository. For more information, see Getting Started.

Creating functions via the console

Image pushing

Run the following code to push the built image to your image repository.

```
# Switch to the file download directory
cd /opt

# Download Demo
git clone https://github.com/awesome-scf/scf-custom-container-code-snippet.git

# Log in to the image repository. Replace $YOUR_REGISTRY_URL with your image
repository, and replace $USERNAME and $PASSWORD with your login credentials
respectively.
docker login $YOUR_REGISTRY_URL --username $USERNAME --password $PASSWORD

# Build images. Replace $YOUR_IMAGE_NAME with your image address.
docker build -t $YOUR_IMAGE_NAME .

# Push images
docker push  $YOUR_IMAGE_NAME
```

Creating a function

1. Log in to the SCF console and click **Function Service** in the left sidebar.
2. At the top of the page, select the region and the namespace where to create a function, and click **Create** to enter the function creation process.

3. Select **Use TCR image** and specify the basic function information.

Parameter	Operation
Function type	Select **Event-triggered function** or **HTTP-triggered Function**.
Function name	Define the function name.
Region	Select the region where to deploy the function. The function must be in the same region as the image repository.
time zone	SCF uses the UTC time by default, which you can modify by configuring the `TZ` environment variable. After you select a time zone, the `TZ` environment variable corresponding to the time zone will be added automatically.
Images	Select the Personal Edition or Enterprise Edition image repository you created.
Image tag	Select the image tag. If this parameter is left empty, the latest version of the image will be used by default.
Entrypoint	Enter the bootstrap command of the container. Parameter input specification: enter an executable command, such as a python command. This parameter is optional. If it is left empty, the Entrypoint value in the Dockerfile will be used by default.
CMD	Enter the bootstrap parameter of the container. Parameter input specification: use "space" as the parameter separator, for example, `-u app.py` . This parameter is optional. If it is left empty, the CMD value in the Dockerfile will be used by default.
Image acceleration	Enable image acceleration as needed. After image acceleration is enabled, SCF can pull images more efficiently. It takes over 30 seconds to enable image acceleration, so please be patient.

4. Click **Complete**.