Serverless Cloud Function Code Development

Contact Us

We are committed to providing personalized pre-sales consultation and technical after-sale support. Don't hesitate to contact us at 4009100100 or 95716 for any inquiries or concerns.

Contents

Code Development
Python
Environment Description

Last updated: 2023-09-28 16:15:17

Python Versions

The currently supported Python programming languages include the following versions:

- Python 3.7
- Python 3.6
- Python 2.7

During the function creation, you can select the desired runtime environment, Python 3.7, Python 3.6, or Python 2.7. You can view the official Python recommendations for choosing between Python 2 or Python 3 here.

Environment Variables

The Python-related environment variables built into the current runtime environment are listed in the table below:

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

For more information on environment variables, see Environment Variables.

Included Library and Usage

> **Note**
> For Python 3.7 and later versions, the platform no longer includes additional built-in dependency libraries. For dependencies required to run the code, please refer to Dependency Installation.

COS SDK

The Python 3.6 and Python 2.7 runtime environments for Cloud Functions already include the Python SDK for COS, specifically versions `cos_sdk_v5` (recommended) and `cos_sdk_v4`.

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

- For the `cos_sdk_v5` version:

```
import qcloud_cos_v5
```

```
from qcloud_cos import CosConfig
from qcloud_cos import CosS3Client
```

- For the `cos_sdk_v4` version:

```
import qcloud_cos
```

```
from qcloud_cos_v4 import CosClient
from qcloud_cos_v4 import DownloadFileRequest
from qcloud_cos_v4 import UploadFileRequest
```

For more detailed instructions on how to use the COS SDK, see COS SDK for Python.

Built-in Libraries

The libraries supported by the Python 3.6 cloud runtime are as follows:

> **Description**
> 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 libraries supported by the Python 2.7 cloud runtime are as follows:

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

Development Methods

Last updated: 2023-09-27 16:20:05

Function Form

The general form of a Python function is 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

In the Python environment, the input parameters include 'event' and 'context', both of which are Python dict types.

- 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 you capture and handle exceptions before returning, and do not continue to throw them, the function will be considered to have executed successfully and will return the information specified in `return` in the entry function. In the following example code, if the function executes successfully, the return value is `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 exceptions are not captured before returning, they will be thrown outside the entry function and captured by the function platform. At this point, the function is considered to

have failed, and the function return information is replaced with failure information. In the following example code, the function 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\", line
4, in main_handler\n    raise Exception(\"err msg\")\nException: err msg",
  "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.

Deployment Methods

Last updated: 2023-09-28 16:15:30

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.

- Deploy by uploading a zip package. For more details, refer to Dependency Installation and Deployment.
- Editing and deploying functions via the console, as instructed in Deployment Through Console.
- Deploy using the command line. For more information, refer to Deploying Functions via Serverless Cloud Framework.

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.

Usage

1. Configure dependency information in the `requirements.txt` file.
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.

Samples

1. Install the `requests` dependency locally. The code file `index.py` is as follows:

```
# -*- 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 `pycryptodome`, require related compilation operations during installation. As the compilation program performs OS-related operations based on different operating systems, the dependency libraries and dynamic libraries compiled in Windows, Mac, and other environments may not run in the cloud function environment. You can try the following solutions:

- 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: 2023-09-28 16:15:53

Logging

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

- print
- logging module

For instance, by executing the following code, you can query the output content in the function log.

```
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

All current function logs will be shipped to Tencent Cloud Log Service (CLS). You can configure the log shipping for function logs. For more details, see Log Shipping Configuration. You can query the function execution logs through the log query interface of the cloud function or the query interface of the log service. For detailed log query methods, see Log Retrieval Tutorial.

> **Description**
> 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

The output of a simple `print` or `logger` method in the function codes will be recorded in the `SCF_Message` field when being uploaded to CLS. For more information about CLS fields, see Configuring index.
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.

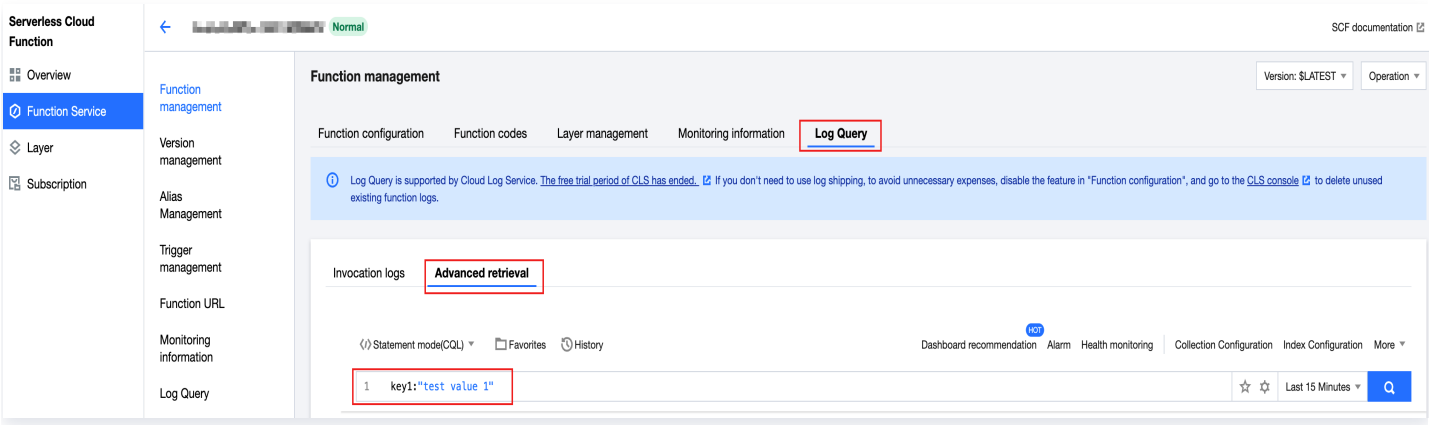
Output

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.
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!")
```

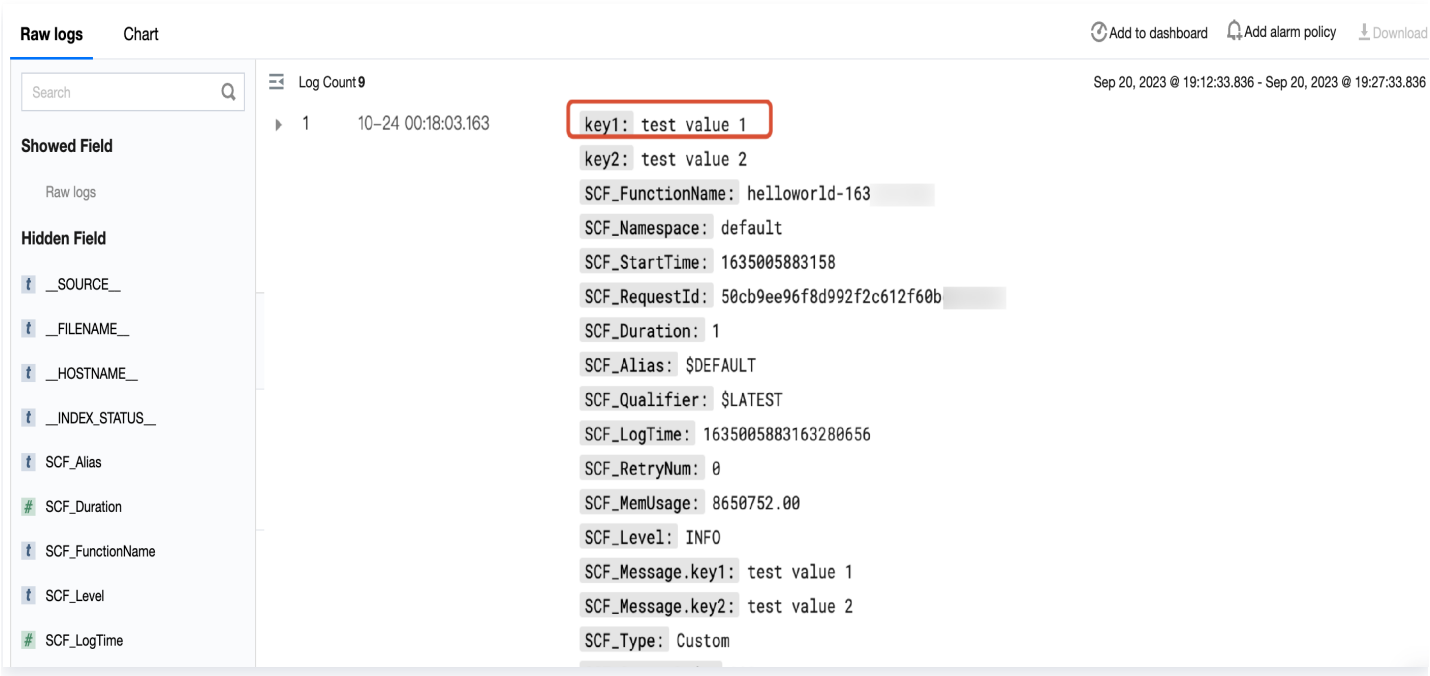
Search

After running the test with the above code, you can search in Function-Log Query-Advanced Search using the following statement:



Search Results

After testing the log service, you can find the `key1` field in the log query, as shown in the image below:



Examples

Last updated: 2023-09-28 16:16:00

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):

    # Establishing a database connection
    db = pymysql.connect(host="host
ip",port=3306,user="user",password="password",database="db name")

    # Create a cursor object using the cursor() method
    cursor = db.cursor()

    # Execute SQL queries using the execute() method
    cursor.execute("SELECT VERSION()")

    # Retrieve a single record using the fetchone() method.
    data = cursor.fetchone()

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

    # Terminate the database connection
    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 Environment Description

Last updated: 2023-09-28 16:16:13

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_modules
Node.js 14.18	NODE_PATH	/var/user:/var/user/node_modules:/var/lang/node14/lib/node_modules:/opt:/opt/node_modules
Node.js 12.16	NODE_PATH	/var/user:/var/user/node_modules:/var/lang/node12/lib/node_modules:/opt:/opt/node_modules
Node.js 10.15	NODE_PATH	/var/user:/var/user/node_modules:/var/lang/node10/lib/node_modules:/opt:/opt/node_modules
Node.js 8.9	NODE_PATH	/var/user:/var/user/node_modules:/var/lang/node8/lib/node_modules:/opt:/opt/node_modules
Node.js 6.10	NODE_PATH	/var/user:/var/user/node_modules:/var/lang/node6/lib/node_modules:/opt:/opt/node_modules

For more information on environment variables, see Environment Variables.

Included Library and Usage

> **Note**
> For Node.js version 14.18 and later, the platform no longer includes additional built-in libraries. For dependencies required to run the code, please refer to Dependency Installation and Online Dependency Installation.

COS SDK

The runtime environment for Cloud Function Node.js version 12.16 and earlier includes the Node.js SDK for COS, specifically version `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 libraries supported in the runtime of various Node.js versions are as follows:

Node.js 12.16

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

Node.js 10.15

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

Node.js 8.9

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

Node.js 6.10

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: 2023-09-28 16:16:27

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-asynchronous handlers**

to return a response. The response object must be compatible with `JSON.stringify`. The callback function has two parameters, `Error` and response. When this function is called, SCF waits for the function to complete before returning the 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**
> Due to the influence of certain externally imported libraries, the event loop may persistently remain non-empty, causing the function to be unable to return until it times out. To mitigate the impact of external libraries, you can control the timing of the function's return by disabling the event loop wait. By setting **context.callbackWaitsForEmptyEventLoop to false**, you can modify the default

callback behavior to avoid waiting for the event loop to be empty.
You can set `context.callbackWaitsForEmptyEventLoop = false;` before the callback is executed, allowing the cloud function background to immediately freeze the process after the callback is called, no longer waiting for events in the event loop, and returning immediately after the synchronous process is completed.

Sample non-async handler:

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

Async Feature Support

Version Support

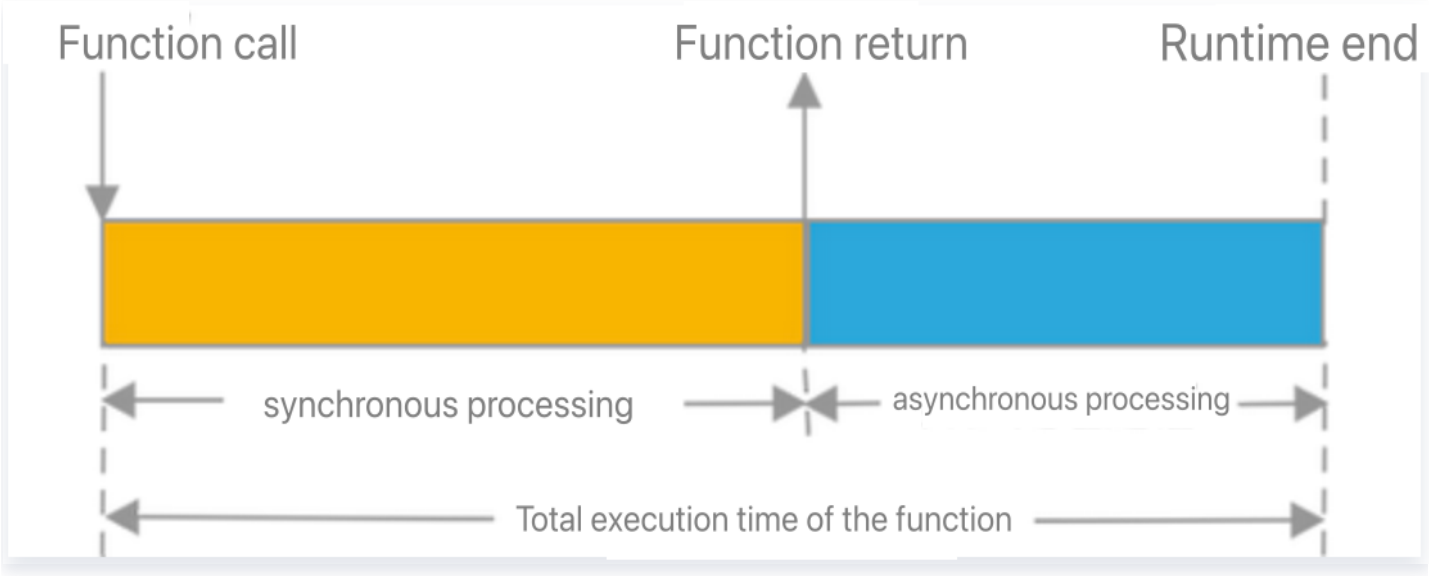
- The capability to separate synchronous execution returns and asynchronous event handling in functions is supported in the following versions:
  - Node.js 16.13 (Not Supported)
  - Node.js 14.18 (Not Supported)
  - Node.js 12.16 (Supported)
  - Node.js 10.15 (Supported)
  - Node.js 8.9 (Not Supported, Soon to be Deprecated)
  - Node.js 6.10 (Not Supported, Soon to be Deprecated)
- 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 synchronous and asynchronous running characteristics, return time, and running duration of Node.js are illustrated in the following diagram:



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, test the function through the console or call it through the `Invoke` API. You will see that the function returns in a very short time, with a response time of less than 1 second.
When you view the function execution logs, you can see related statistical information similar to the following:

```
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: 'test
value 1', key2: 'test value 2' }
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 MemUsage:13.425781MB
```

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: 2023-09-28 16:16:34

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.

- Deploy by uploading a zip package. For more details, refer to Dependency Installation and Deployment.
- Editing and deploying functions via the console, as instructed in Deployment Through Console.
- Deploy using the command line. For more details, refer to Deploying Functions via 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.

Usage

Execute the `npm install xxx command` in the code directory to install the dependency package.

> **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.

Samples

1. Install the requests dependency locally. The code file `index.js` is shown below:

```
'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) // Logic for successful request handling
    }
 })
 return "success"
};
```

2. Use the command `npm install request` to install the requests dependency in the current project directory.

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: 2023-09-27 21:29:20

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
};
```

Overview

- In the Node.js 12.16 and Node.js 10.15 runtime environments, using `console.log` to print logs, the platform will encapsulate the log content in the format of "timestamp RerquestId log content" and write it into the log service CLS.
  Example:
  Log print statement: `console.log("hello world")`
  Log output effect:
  `2021-12-27T03:53:59.192Z a7358cce-489a-4674-8e4e-68665fa2b81d Hello World`
- In the Node.js 8.9 runtime environment, using `console.log` , `console._stdout.write()` , `process.stdout.write()` to print logs, the platform will encapsulate the log content in the format of "timestamp RerquestId log content" and write it into the log service CLS.
  Example:
  Log print statement: `console.log("hello world")`
  Log output effect:
  `2021-12-27T03:53:59.192Z a7358cce-489a-4674-8e4e-68665fa2b81d Hello World`
- In the Node.js 6.10 runtime environment, using `console.log` to print logs, the platform will encapsulate the log content in the format of "timestamp RerquestId log content" and write it into the log service CLS.
  Example:

Log print statement: `console.log("hello world")`
Log output effect:
`2021-12-27T03:53:59.192Z a7358cce-489a-4674-8e4e-68665fa2b81d Hello World`

Log Query

All current function logs will be shipped to Tencent Cloud Log Service (CLS). You can configure the log shipping for function logs. For more details, see Log Shipping Configuration. You can query the function execution logs through the log query interface of the cloud function or the query interface of the log service. For detailed methods of log query, see Log Retrieval Tutorial.

> **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 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.

> **Caution**
> - 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

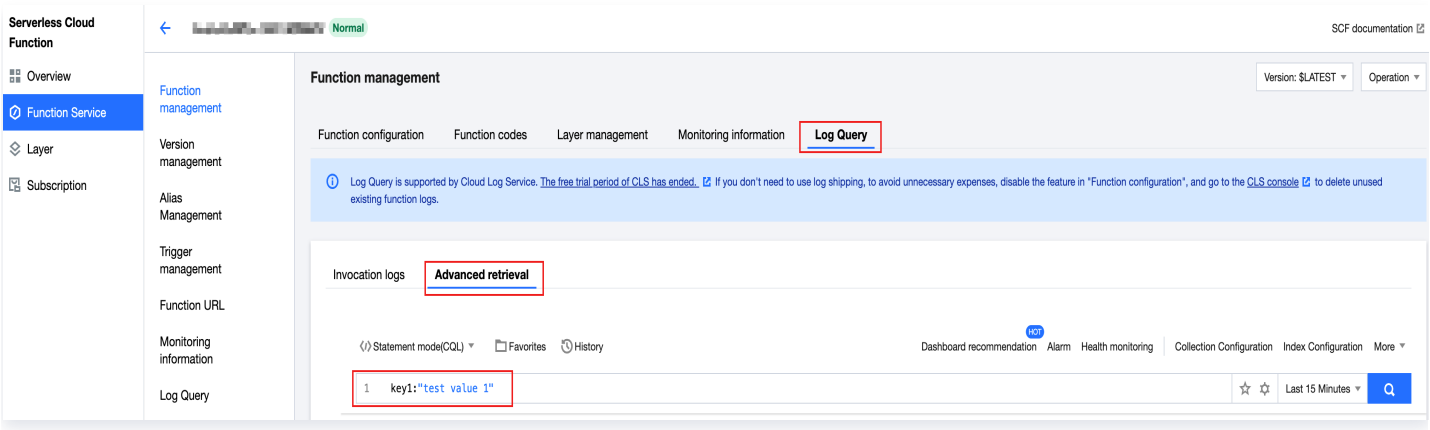
If a function outputs a single-line log in JSON, the log will be parsed and uploaded to CLS in the `field:value` format. Only the first layer will be parsed, and the remaining nested structure

will be recorded as values.
You can run the following code to test:

```
'use strict';
exports.main_handler = async (event, context) => {
    console._stdout.write(JSON.stringify({"key1": "test value 1","key2": "test value 2"})+'\n');
    return "hello world"
};
```

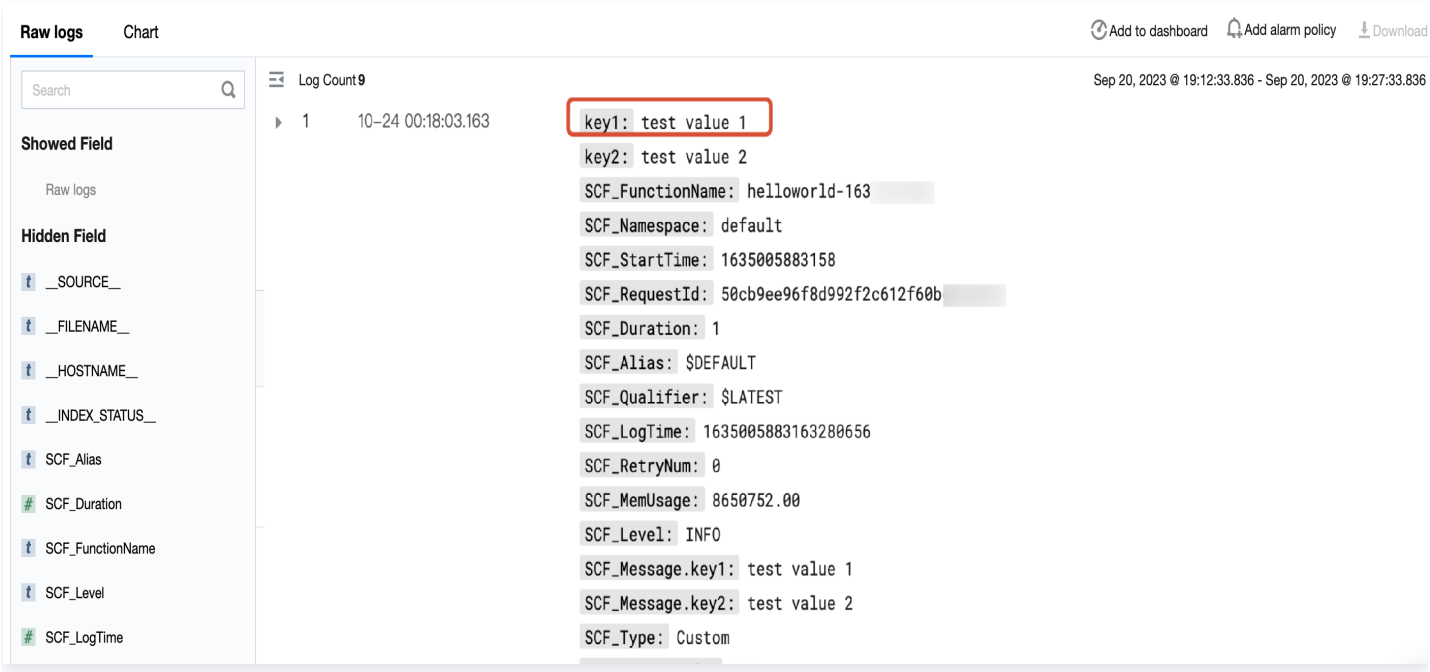
Search

After running the test with the above code, you can search in Function–Log Query–Advanced Search using the following statement:



Search Results

After testing the log writing service, you can find the `key1` field in the log search, as shown in the image below:



Examples

Last updated: 2023-09-27 21:29:30

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 Connections in a Function

This example uses the `requests` library to initiate network connections within the function and retrieve page information. You can install the dependency library by executing the `npm install request` command in the project directory.

```
'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) // Logic for successful request handling
        }
    })
    return "success"
};
```

Online Dependency Installation

Last updated: 2023-09-27 21:29:44

Scenario

SCF supports online dependency installation during function deployment.

Features

> **Note**
> Online dependency installation is supported only for Node.js.

If "Online Dependency Installation" is enabled in the function configuration, after each code upload, the SCF backend will check the `package.json` file in the root directory of the code package and attempt to install the dependencies listed in the `package.json` file using npm install.
For example, if the `package.json` file in the project lists the following dependency:

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

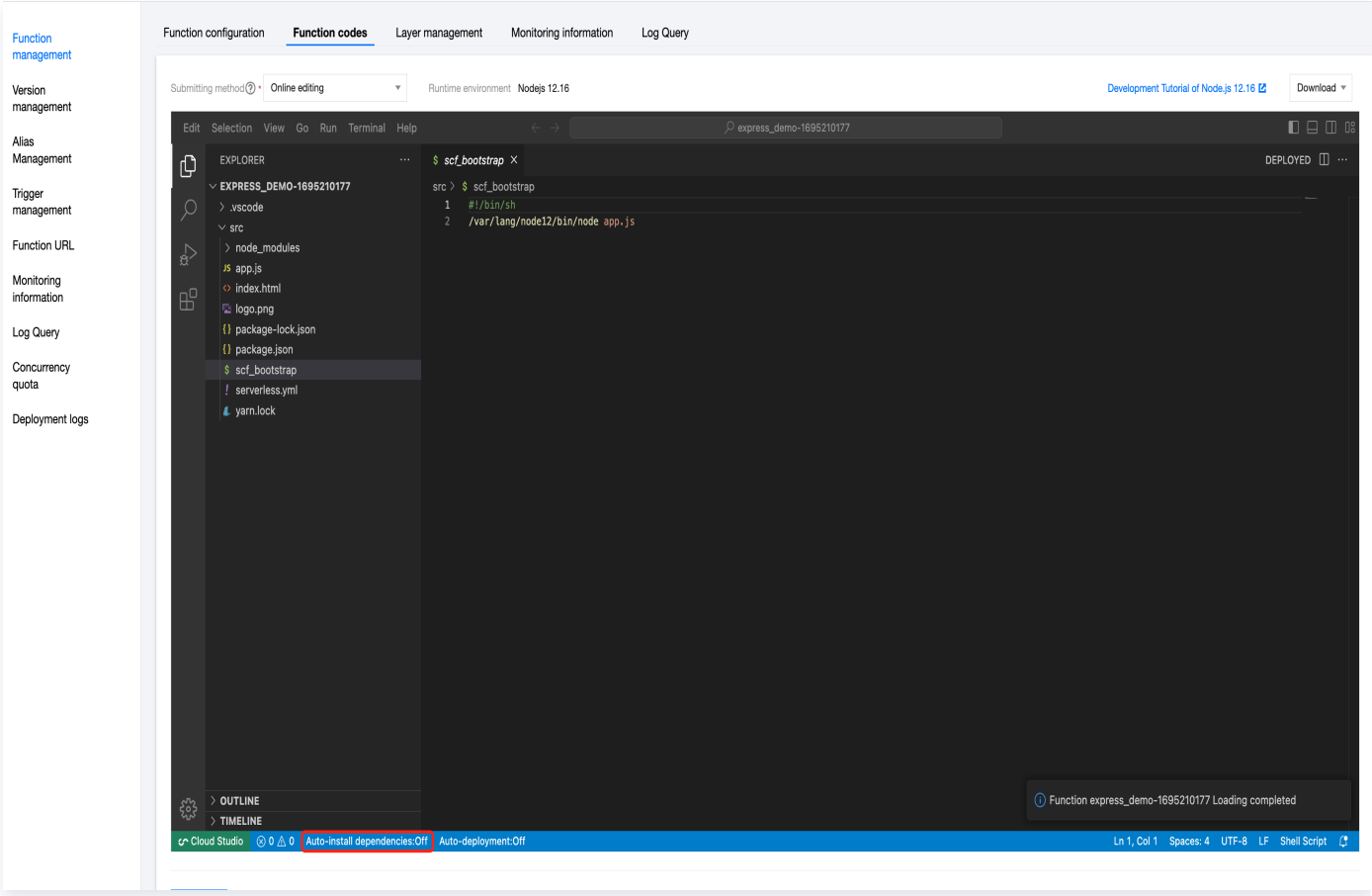
Then this dependency will be imported into the function during the deployment:

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

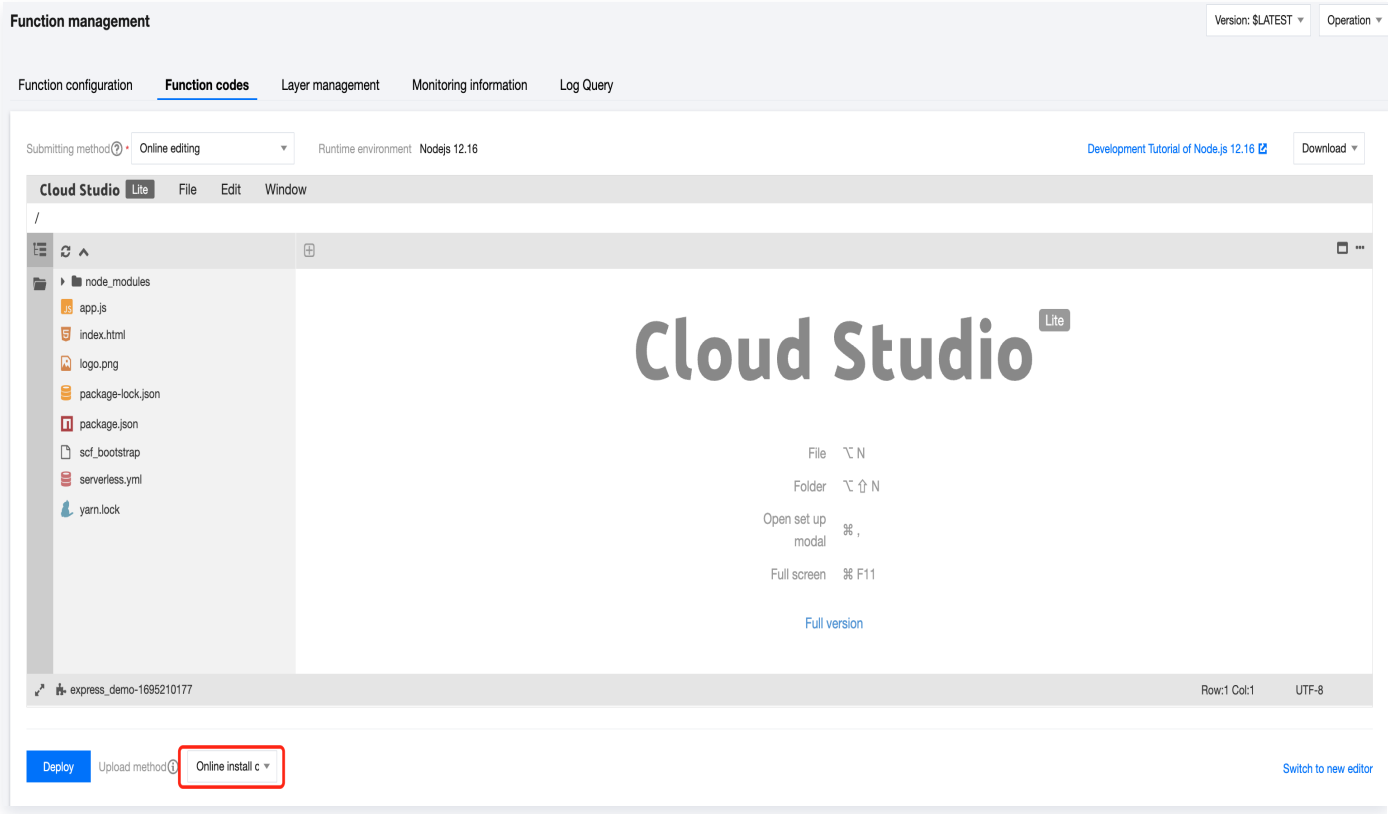
Instructions

1. Log in to the Serverless Console and select the **Guangzhou** region.
2. Select **Function Service** from the left sidebar, and choose the function name from the "Function Service" list page.
3. Select the **Function codes** tab and modify the function code as needed.

4. Click in the top right corner of the IDE code editing window, select **Auto-install dependencies: Off** from the dropdown list to enable automatic dependency installation, as shown below:



5. After enabling automatic dependency installation, refresh the page, click **Switch to old editor** in the bottom right corner of the code editing area, and select **Online dependency installation** as the upload method.



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

Golang Environment Description

Last updated: 2023-09-27 21:30:21

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.

Precautions

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: 2023-09-28 16:59:18

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.
- The entry function can take 0 – 2 parameters. If parameters are included, `context` should come before `event`. The parameter combinations are (), (event), (context), (context, event). For more details, please refer to Parameters.
- The entry function can return 0 – 2 parameters. If parameters are included, the return content should come before the error message. The return value combinations are (), (ret),

(error), (ret, error). For more details, please refer to Return Values.

- Both the input parameter `event` and the return value `ret` need to be compatible with the `encoding/json` standard library, allowing 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.
By using `import "github.com/tencentyun/scf-go-lib/cloudfunction"`, you can utilize the `Start` function within the `main` function.

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.
Custom parameters can be basic data structures provided by Golang, such as string, int, or they can be custom data structures, like the DefineEvent in the example. When using custom

data structures, it is necessary to ensure that the data structure is compatible with the `encoding/json` standard library and can perform Marshal and Unmarshal operations. Otherwise, errors may occur due to exceptions when input parameters are submitted.
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**
> Some of the input parameter event structures passed by certain triggers have already been defined and can be used directly. You can obtain the golang library through cloud event definition and use it. You can use it directly by referencing `import "github.com/tencentyun/scf-go-lib/events"` in your code. If you encounter any issues during use, you can resolve them by submitting an issue or contacting online customer service.

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 return value can be a basic data structure provided by Golang, such as string, int, or it can be a custom data structure. When using a custom data structure, you need to ensure that the data structure is compatible with the `encoding/json` standard library and can perform Marshal and Unmarshal operations. Otherwise, errors may occur due to abnormal conversion when returning to the external interface.
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: 2023-09-27 21:30:53

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:
  - Press **Windows + R** to open the Run window, type **cmd** and press **Enter** to open the command prompt.
  - Execute the following command in the command prompt to set the compilation parameters:

```
set GOOS=linux
set GOARCH=amd64
```

    Here, GOOS is set to Linux and GOARCH is set to amd64, indicating that the compiled binary file is suitable for 64-bit architecture on Linux systems.
  - Execute the following command in the command prompt to compile:

```
go build -o main main.go
```

  - 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: 2023-09-28 16:15:20

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

All current function logs will be shipped to Tencent Cloud Log Service (CLS). You can configure the log shipping for function logs. For more details, see Log Shipping Configuration. You can query the function execution logs through the log query interface of the cloud

function or the query interface of the log service. For detailed methods of log query, see Log Retrieval Tutorial.

> **Description**
> 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 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

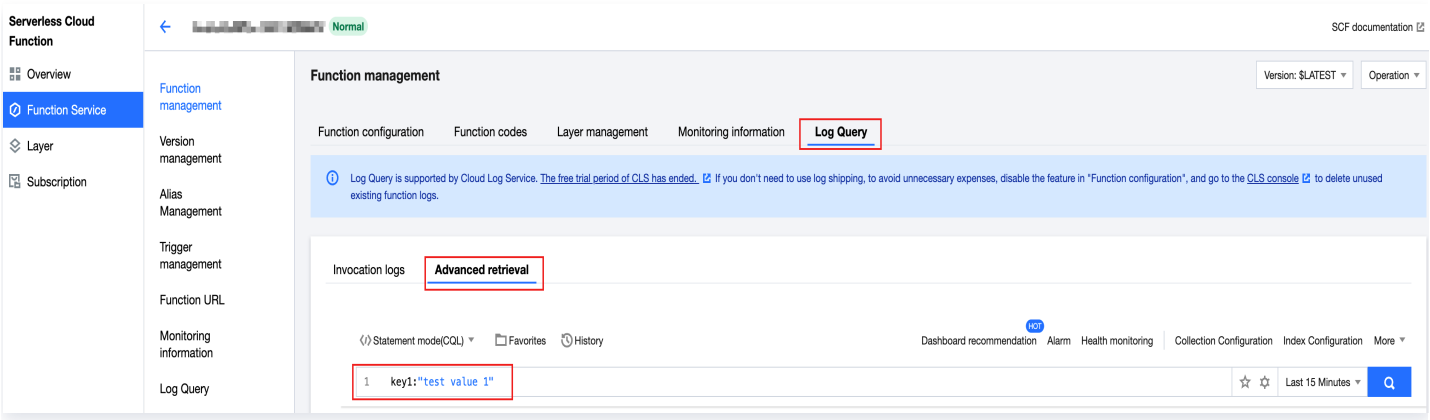
If a function outputs a single-line log in JSON, the log will be parsed and uploaded to CLS in the `field:value` format. Only the first layer will be parsed, and the remaining nested structure will be recorded as values.
You can run the following code to test:

```
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)
}
```

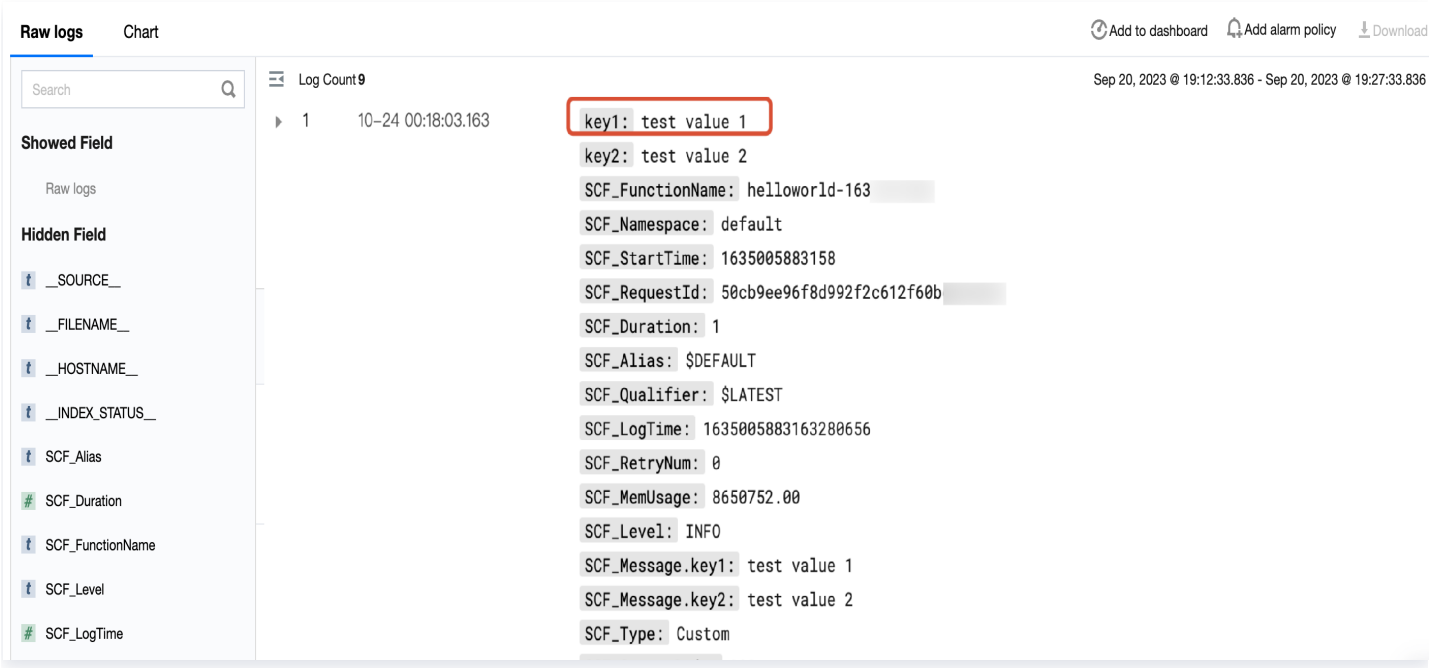
Search

After running the test with the above code, you can search in Function–Log Query–Advanced Search with the following statement:



Search Results

After testing the log writing service, you can find the `key1` field in the log search, as shown below:



PHP
Environment Description

Last updated: 2023-09-27 21:31:23

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 do not meet your business requirements, you can install

> custom extensions as per Dependency Installation. You can view the installed extensions in the function by running `print_r(get_loaded_extensions());`.

The currently installed PHP extensions are listed below:

PHP 8.0、PHP 7.4

Core	calendar	gd	mysqlnd	SimpleXML
runkit7	ctype	SPL	PDO	soap
date	curl	iconv	pdo_mysql	exif
libxml	dom	intl	pdo_sqlite	tokenizer
openssl	hash	json	Phar	xml
pcre	fileinfo	mbstring	posix	xmlreader
sqlite3	filter	session	Reflection	xmlwriter
zlib	ftp	standard	mysqli	runtime
bcmath				

PHP 7.2

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

PHP 5.6

Core	ctype	json	Reflection	xmlwriter
runkit	curl	mbstring	mysqli	zip
date	dom	session	SimpleXML	eio
ereg	hash	standard	soap	memcached
libxml	fileinfo	mysqlnd	sockets	imagick
openssl	filter	PDO	exif	mongodb
pcre	ftp	pdo_mysql	tidy	protobuf
sqlite3	gd	pdo_sqlite	tokenizer	redis
zlib	SPL	Phar	xml	Zend OPcache
bcmath	iconv	posix	xmlreader	runtime

bz2 calendar				

Development Methods

Last updated: 2023-09-27 21:31:41

Function Form

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**
> In PHP 8.0, PHP 7.4, PHP 7.2, and PHP 5.6, the input parameters `event` and `context` are in the `object` format.

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 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.

> **Description**
> - 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 length of `SCF_Message` is limited to 8KB. If exceeded, only the first 8KB will be retained.

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: 2023-09-27 21:32:02

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.

- Deploy by uploading a zip package. For more details, refer to Dependency Installation and Deployment.
- Editing and deploying functions via the console, as instructed in Deployment Through Console.
- Deploy using the command line. For more details, refer to Deploying Functions via Serverless Cloud Framework.

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.

Instructions

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. In the `/code` folder, execute the following command to install the dependency packages

and versions 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: 2023-09-27 21:32:11

Logging

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 in the function log by running the following code.

```php
<?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

All current function logs will be shipped to Tencent Cloud Log Service (CLS). You can configure the log shipping for the function. For more details, see Log Shipping Configuration. You can query the function execution logs through the log query interface of the cloud function or the query interface of the log service. For detailed log query methods, see Log Retrieval Tutorial.

> **Explanation**
> 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

The output of a simple `print` or `logger` method in the function codes will be recorded in the `SCF_Message` field when being uploaded to CLS. For more information about CLS fields, see

Configuring index.
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.

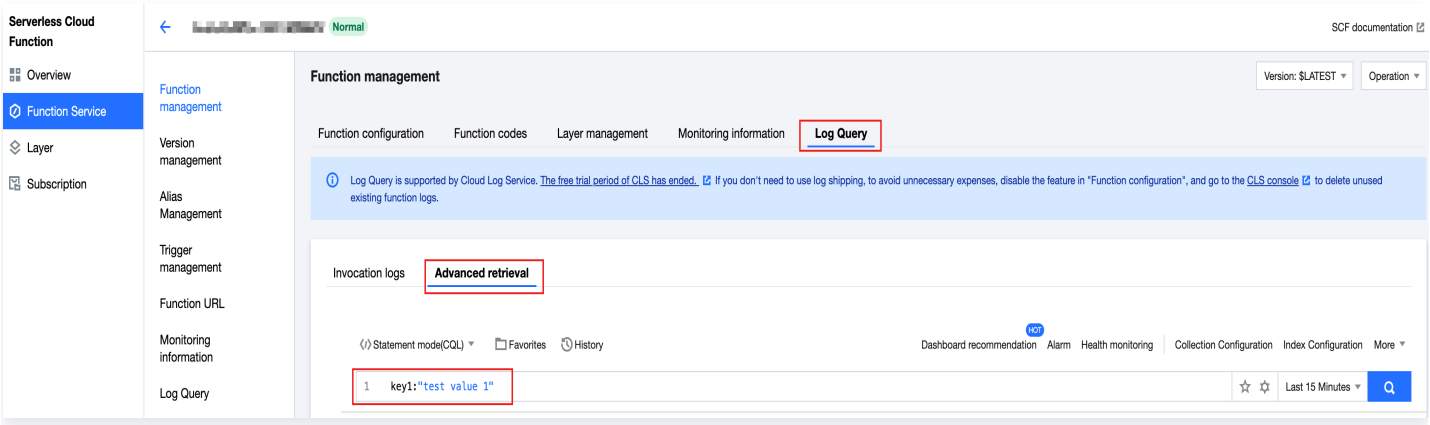
Output

When the function outputs a single line of log in JSON format, the JSON content will be parsed and delivered to the log service in the form of "**field:value**". The parsing of JSON content can only parse the first layer, and more 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";
}
?>
```

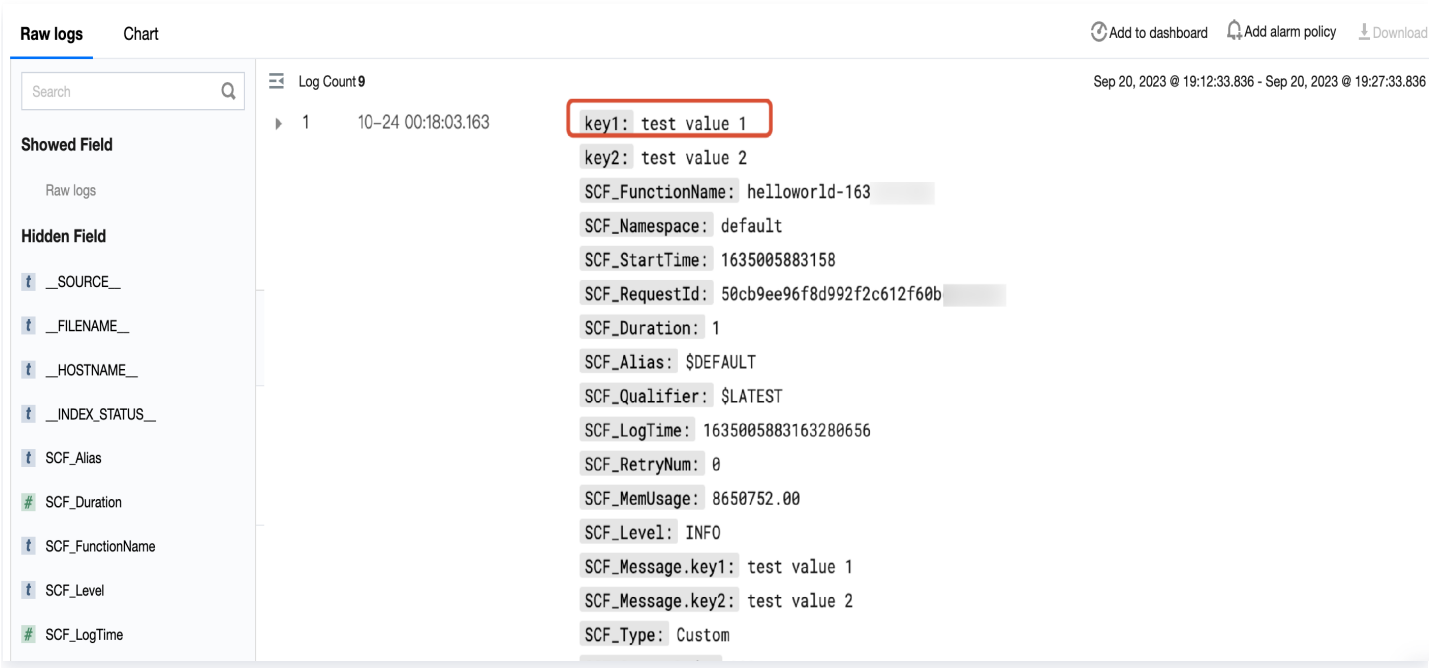
Search

After running the test with the above code, you can search in Function-Log Query-Advanced Search using the following statement:



Search Results

After testing the log service, you can find the `key1` field in the log query, as shown below:



Examples

Last updated: 2023-09-27 21:32:26

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 demonstrates how to obtain a list of all environment variables or the value of 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: 2023-09-27 21:32:42

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.

Precautions

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.

Development Methods

Last updated: 2023-09-27 21:33:26

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 code example, the `mainHandler` uses POJO type for input parameters and String type for return. The event input parameters and function return currently support Java basic types and POJO types. The function runtime is currently of `com.qcloud.scf.runtime.Context` type, and its related library files can be downloaded by clicking here.

Types supported for event input and response parameters

Event Input Parameter	Response Parameter Type
Java base types	This includes eight basic types and their corresponding wrapper classes: byte, int, short, long, float, double, char, boolean, as well as the 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**
> Some of the event structures passed by certain triggers have already been defined and can be used directly. You can obtain and use the Java library through cloud event definition. If you encounter any issues during use, you can seek help by submitting an issue or consulting online customer service.

Deployment Methods

Last updated: 2023-09-27 21:33:54

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 a 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.

Environment Preparation

Make sure that you have Java and Gradle installed.

- For Java 11, please install TencentKona11 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
```

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);
 }
}
```

Prepare 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 specifying mavenCentral as the source of the dependency library through repositories, Gradle will automatically pull the dependencies from Maven Central during the compilation process, that is, 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
```

By specifying the search directory as the `*.jar` files in the jars directory through dependencies, the dependencies will be automatically searched 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 the compilation fails, adjust the code according to the output compilation error information. The compiled zip package is located in the `/build/distributions` directory within the project folder and is named `scf_example.zip` after the project folder.

Creating JAR Deployment Package with Maven

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

Environment Preparation

Make sure that you have Java and Gradle installed.

- For Java 11, please install TencentKona11 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.

Using 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, please 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"
```

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);
 }
}
```

Prepare the compilation file

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

Java 11

```
<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>
       <source>11</source>
       <target>11</target>
      </configuration>
      <executions>
       <execution>
        <phase>package</phase>
        <goals>
         <goal>shade</goal>
        </goals>
       </execution>
      </executions>
     </plugin>
    </plugins>
  </build>
</project>
```

Java 8

```
<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>
```

Using Maven Central library to handle package dependencies

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

```
<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>

 <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 the compilation fails, adjust the code according to the output compilation error information. The compiled jar package is located in the `target` directory within the project folder and is named `java-example-1.0-SNAPSHOT.jar` according to the artifactId and version fields in pom.xml.

Log Description

Last updated: 2023-09-27 21:34:10

Logging

You can use `System.out.println()` and `java.util.logging.Logger()` statements in your program to accomplish log output.
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

All current function logs will be shipped to Tencent Cloud Log Service (CLS). You can configure the log shipping for function logs. For more details, see Log Shipping Configuration.
You can query the function execution logs through the log query interface of the cloud function or the query interface of the log service. For detailed methods of log query, see Log Retrieval Tutorial.

> **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 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.

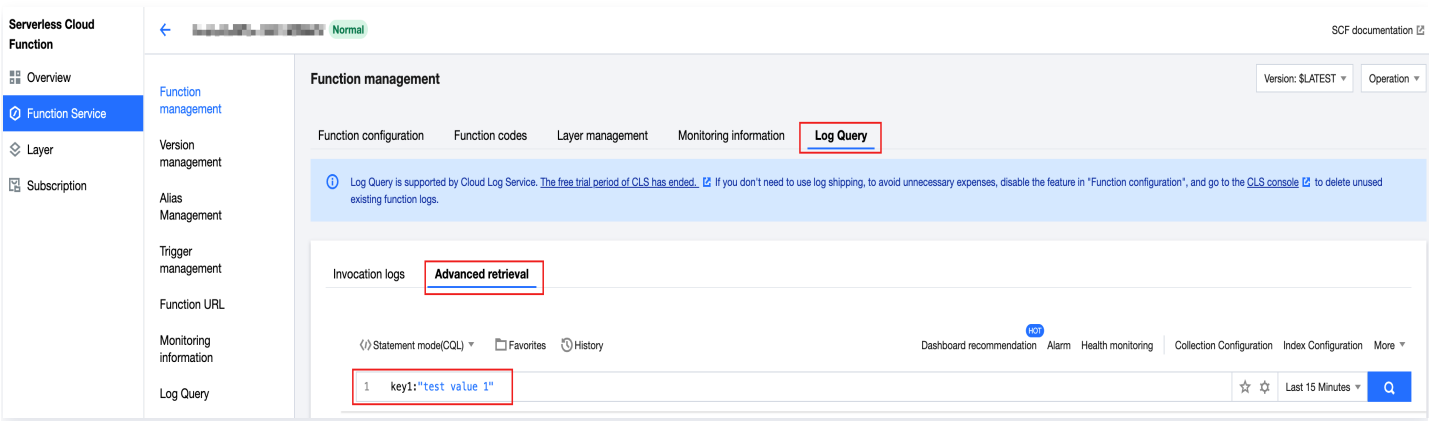
Output

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.
You can run the following code to test:

```
package example;

public class Hello {
    public String mainHandler(KeyValueClass kv) {
        System.out.println("{\"key1\": \"test value 1\",\"key2\": \"test value 2\"}");
      return String.format("hello world");
    }
}
```

Search

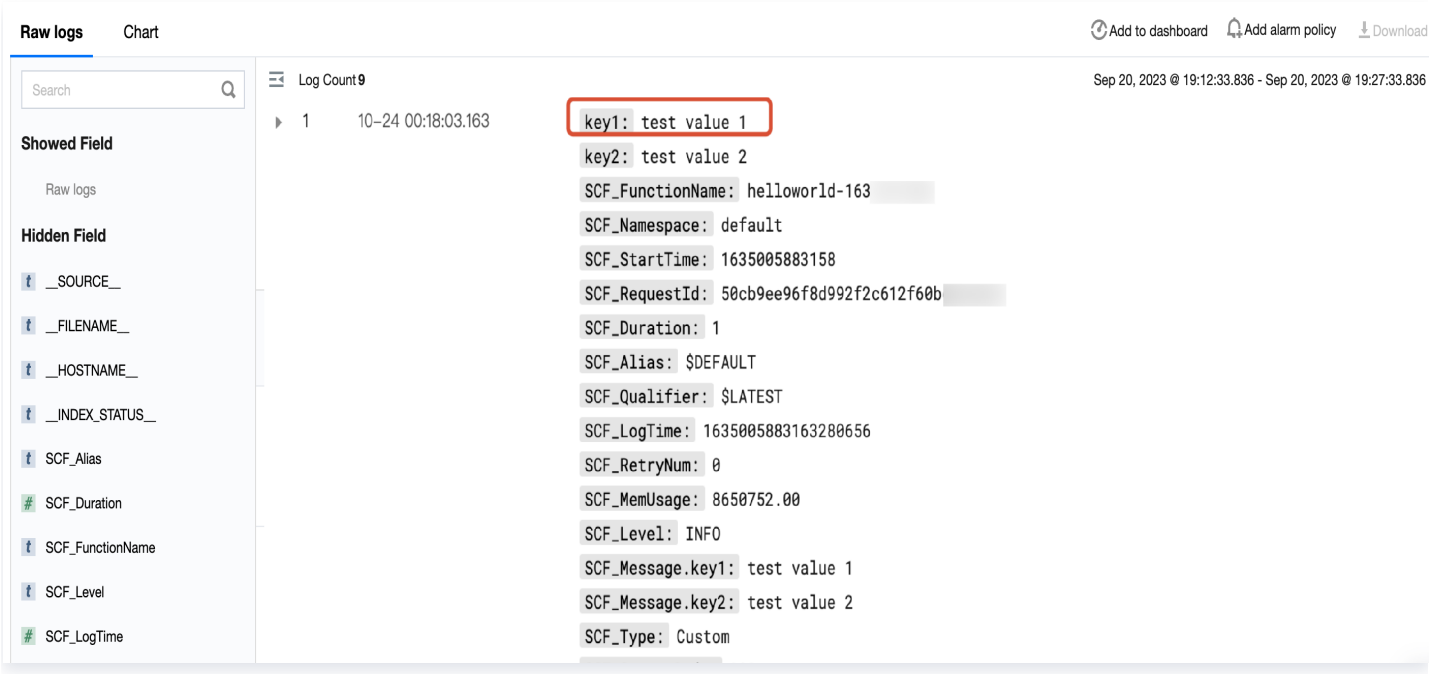
After running the test with the above code, you can search using the following statement in **Function Service > Log Query > Advanced Search**:



For operation details, please refer to Console Search.

Search Results

After testing the log writing service, you can find the `key1` field in the log search, as shown in the image below:



Examples

Last updated: 2023-09-27 21:34:20

Scenario

By utilizing POJO type parameters, you can manage more complex data structures beyond simple event inputs. This document provides a set of examples illustrating how to use POJO parameters in SCF and what format of inputs it supports.

Preparations

You have logged in to the Serverless console.

Instructions

Event Input and POJO

The event inputs used in this document are as follows:

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

Given the above inputs, the output is as follows:

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

Based on the inputs, the following four classes have been constructed:

- RequestClass: Utilized for event reception, serving as the class for event acceptance.
- PersonClass: Employed for handling the `person` segment within the event JSON.
- CityClass: Engaged in managing the `city` section within the event JSON.
- ResponseClass: Utilized for assembling the response content.

Code Preparations

Given the four classes and the entry function constructed based on the input parameters, please proceed with the code preparations following the steps below.

Step 1: Preparation of Project Directory

Establish the root directory of the project, for instance, `scf_example`.

Step 2: Preparation of Code Directory

1. Under the root directory of the project, create a folder named `src\main\java` to serve as the code directory.
2. Based on the package name you intend to use, create a package folder under the code directory. For instance, `example`, resulting in a directory structure like `scf_example\src\main\java\example`.

Step 3: Code Preparation

Within the `example` folder, create the following files: `Pojo.java`, `RequestClass.java`, `PersonClass.java`, `CityClass.java`, and `ResponseClass.java`. The contents of these files are as follows:

Pojo.java

```
package example;

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

RequestClass.java

```
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() {
    }

}
```

PersonClass.java

```
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() {
    }

}
```

CityClass.java

```
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() {
    }

}
```

ResponseClass.java

```
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**
> This example uses Maven for compilation and packaging. You can choose the packaging method according to your own circumstances.

1. Create a 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. Execute the command `mvn package` in the command line. Ensure that there is a successful compilation message. The following output indicates successful packaging.

```
[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] ------------------------------------------------------------------------
```

In case of compilation failure, please make the necessary modifications as per the prompts.

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

Creation and Testing of SCF Function

1. Create a cloud function. For more details, refer to Creating and Updating Functions.
2. Use the compiled package for upload submission. You can choose to upload via ZIP, or first upload to the COS Bucket and then submit by selecting the COS Bucket.
3. Set the execution method of the cloud function to `example.Pojo::handle`.
4. Enter the expected input parameters in the test event template on the **Function Code** page:

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

After clicking Run, you can view the returned content as follows:

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

You can also modify the value content within the test input parameters structure. After running, you can see the modification effect.

Relevant Examples

You can go to scf-demo-java, pull the sample code and perform tests.

Custom Runtime Overview

Last updated: 2023-09-27 21:36:46

Beyond the standard runtime environments supported by Serverless Cloud Function (SCF), SCF offers a Custom Runtime service to accommodate the implementation of functions in more personalized development languages and versions. By enabling the creation of custom function runtimes, it supports the use of any version of any development language to write functions, and to carry out global operations during function calls, such as loading extension programs, security plugins, monitoring agents, etc. SCF and Custom Runtime communicate via the HTTP protocol to handle event responses.

Custom Runtime Deployment Documentation

Bootstrap: A fixed executable bootstrap program file in Custom Runtime, created by developers as an executable program file with the same name. It is implemented through custom languages and versions, directly processing or initiating other executable files to handle the initialization and invocation of function runtime.
Function Files: Function program files developed and implemented by developers through custom languages and versions.
Runtime-dependent Library Files or Executable Files: Add relevant dependent library files or executable files as needed by the custom language and version runtime.
Published through a deployment package, the file structure is as follows:

- Bootstrap (Required)
- Function Files (Required)
- Runtime-dependent Library Files or Executable Files (Optional)

Due to the size limitations of the deployment package, when the volume of runtime-dependent library files or executable files is large, it is recommended to publish through a combination of deployment package binding layers. The file structure is as follows:

- Deployment Package:
  - Bootstrap (Required)
  - Function Files (Required)
- Layer:
  - Runtime-dependent Library Files or Executable Files (Optional)

> **Note**

> Before deploying the executable files in the above deployment files to SCF, please ensure to set the executable permissions of the files, and then package the deployment files and upload them directly via a zip package or through COS. For instance, when you compile files using the root account in a Linux system, you can use the `chmod -R 777 your_path` command to change the file permissions.

Custom Runtime Operating Mechanism

Custom Runtime divides the lifecycle of a function into an initialization phase and an invocation phase. The initialization phase is executed only once during the cold start process of an instance, while the invocation phase is the execution process for each event call after the instance is started.
Different programming languages and versions will have different startup and execution times. SCF has added an **Initialization Timeout** configuration for Custom Runtime, which, along with the **Execution Timeout** configuration, manages the lifecycle of Custom Runtime.

Function Bootstrapping

SCF first retrieves the executable bootstrap file in the deployment package and performs the following operations based on the retrieval results:

- If the bootstrap file is retrievable and executable, the bootstrap program is loaded and executed, entering the function initialization phase.
- If the bootstrap file is not retrievable or not executable, it returns that the bootstrap file does not exist, resulting in a startup failure.

Function Initialization

It begins with the execution of the bootstrap program file. Developers can customize personalized operations in the bootstrap as needed, directly processing or calling other executable program files to complete the initialization operation. The following are basic operations recommended to be completed during the initialization phase:

- Set the paths and environment variables of the runtime's dependency libraries.
- Load library files and extension programs dependent on custom languages and versions. If there are still dependent files that need to be pulled in real-time, they can be downloaded 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.
- Upon completion of the initialization phase, it is necessary to actively call the runtime API and access the initialization ready interface `/runtime/init/ready` to notify SCF that the

Custom Runtime has completed initialization and entered the ready state. Otherwise, SCF will continue to wait until the configured initialization timeout is reached, at which point it will terminate the Custom Runtime and return an initialization timeout error. If notified repeatedly, the time of the first access will be used as the ready state time node.

Logs and Exceptions

- SCF will collect all standard outputs during the initialization phase and report them to the logs.
- If the function initialization is completed normally within the timeout limit, the logs from the initialization phase will be merged with the logs from the first invocation. If the execution is not completed due to an error within the initialization timeout limit, the execution result will return an initialization timeout error. The program errors and exception logs written to the standard output will be reported to SCF and displayed in the console logs and log queries.

Function invocation

During the function invocation phase, developers need to custom implement the acquisition of events, function invocation, and return of results, and continuously manage this process.

- Implement long polling for event acquisition. Developers can use a custom language and version to implement an HTTP CLIENT to access the event acquisition interface of the runtime API `/runtime/invocation/next`. The response body contains event data, and repeated access to this interface within a single invocation will return the same event data.

> **Note**
> Do not set a timeout for the GET method of the HTTP CLIENT.

- Construct the parameters for the function invocation based on the required information in the environment variables, response headers, and event information.
- Push event information and other parameter data, and invoke the function handler.
- Access the runtime API response result interface `/runtime/invocation/response` to push the function processing results. The first successful call is the final state of the event, and SCF will lock the status. Once pushed, the result cannot be changed.
- If an error occurs during the function invocation, push the error information by accessing the runtime API call error interface `/runtime/invocation/error`. This will end the current call. The first call is considered the final state of the event, and SCF will lock the status. The result cannot be changed after it has been pushed.
- Clean up and release resources that are no longer needed after this call is completed.

Logs and Exceptions

- SCF will collect all standard outputs during the invocation phase and report them to the logs.
- If the Custom Runtime does not fetch the event for a long time after SCF issues it, and it exceeds the function execution timeout limit, SCF will terminate the instance and return a timeout error for waiting to fetch the event.
- If the Custom Runtime fetches the event after SCF issues it but does not return the execution result within the function execution timeout limit, SCF will terminate the instance and return an execution timeout error.

Custom Runtime API

Custom Runtime is implemented by developers using custom languages and versions. Communication with SCF, such as event distribution and processing result feedback, needs to be conducted through a standard protocol. Therefore, SCF provides a runtime API to meet the interaction needs during the lifecycle of Custom Runtime.
SCF has the following built-in environment variables:

- SCF_RUNTIME_API: The address of the runtime API.
- SCF_RUNTIME_API_PORT: The port of the runtime API.

  For more information, see Environment Variables.

Custom Runtime can access the runtime API through `SCF_RUNTIME_API:SCF_RUNTIME_API_PORT`.

Path	HTTP Method	Description
/runtime/init/ready	post	Once the runtime initialization is complete, call the interface to indicate readiness.
/runtime/invocation/next	get	Retrieve the invocation event. The response headers contain the following information: • `Request_Id`: request ID, identifying the request that triggers function invocation. • `Memory_limit_in_mb`: Memory limit for the function, in MB. • `Time_Limit_In_Ms`: function timeout limit, in milliseconds. The response body contains event data. For the structure, see Summary of Trigger Event Message Structure.
/runtime/invocation/	post	Function processing result. After invoking the

response		function handler at runtime, the response from the function is pushed to the invocation response interface.
/runtime/invocation/error	post	If the function returns an error, it is pushed to the invocation error interface, indicating that this invocation has failed.

Creating Sample Bash Function

Last updated: 2023-09-28 16:23:53

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.

Instructions

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

Creating a bootstrap file

The bootstrap is a runtime entry boot program file. When Custom Runtime loads a function, it always retrieves a file named bootstrap and executes this program to start the Custom Runtime runtime. Custom Runtime supports the development and operation of functions in any language and version, mainly based on the bootstrap boot program customized by the developer. The bootstrap must meet the following conditions:

- 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 - Loading function files
source ./"$(echo $_HANDLER | cut -d. -f1).sh"

# Initialization completed, ready status reported to the runtime API.
curl -d " " -X POST -s
"http://$SCF_RUNTIME_API:$SCF_RUNTIME_API_PORT/runtime/init/ready"

### Continuously listening and handling event calls
while true
do
  HEADERS="$(mktemp)"
  # Long-Polling to Fetch Events
```

```
  EVENT_DATA=$(curl -sS -LD "$HEADERS" -X GET -s
"http://$SCF_RUNTIME_API:$SCF_RUNTIME_API_PORT/runtime/invocation/next")
  # Invoking function to handle events
  RESPONSE=$($(echo "$_HANDLER" | cut -d. -f2) "$EVENT_DATA")
  # Pushing function processing results
  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 detailed information about the initialization phase, please refer to Function Initialization.
  After the initialization phase is completed, you need to actively call the runtime API initialization ready interface to notify SCF. The sample code is as follows:

  ```
  # Initialization completed, ready status reported to the runtime API.
  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, please see Environment Variables.
  - `SCF_RUNTIME_API`: The address of the runtime API.
  - `SCF_RUNTIME_API_PORT`: The port of the runtime API.
- **Initialization Logs and Exceptions**
  For logs and exception information during the initialization phase, please refer to Logs and Exceptions.
- **Invocation Phase**
  For detailed information about the invocation phase, please refer to Function Invocation.
  1.1 After initialization, the invocation loop starts. A function is invoked to listen to events. The sample code is as follows:

```
# Long-Polling to Fetch Events
 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:

```
# Invoking function to handle events
 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:

```
# Pushing function processing results
 curl -X POST -s
"http://$SCF_RUNTIME_API:$SCF_RUNTIME_API_PORT/runtime/invocation/respon
se"  -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:

```
# Pushing Function Handling Errors
 curl -X POST -s
"http://$SCF_RUNTIME_API:$SCF_RUNTIME_API_PORT/runtime/invocation/error"
-d "parse event error"
```

- **Invocation Logs and Exceptions**
  For logs and exception information during the invocation phase, please refer to 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. Upon successful creation of bootstrap and function files, 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 Mac OS.

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

3. Utilize the Serverless Cloud Framework to create and release functions. Alternatively, execute the following command to package into a zip file, then use the SDK or Serverless Console to create and release functions.

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

Utilizing Serverless Cloud Framework for Function Creation and Release

Creating function version alias

1. Install the Serverless Cloud Framework.
2. Within the bootstrap directory, configure the Serverless.yml file to create a dotnet function:

```
#Component information
component: scf # Component name, the scf component is used in this instance.
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 detailed configuration of the SCF component, please refer to the comprehensive configuration documentation.

3. Execute the `scf deploy` command to create a cloud function. If the creation is successful, the following results will be returned:

```
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 usage of the SCF component, please refer to the SCF Component.

Invoking a function

Since the `events` configuration in serverless.yml has been set to `apigw`, an API gateway is created along with the function. This allows the cloud function to be accessed through the API gateway. If the following information is returned, it indicates successful access.

```
Echoing request:
'{
      "headerParameters":{},
      "headers":{
"accept":"text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/w
ebp,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 function version alias

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\":\"CustomRuntime\",\"I
nitTimeout\":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	Note
`"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)
```

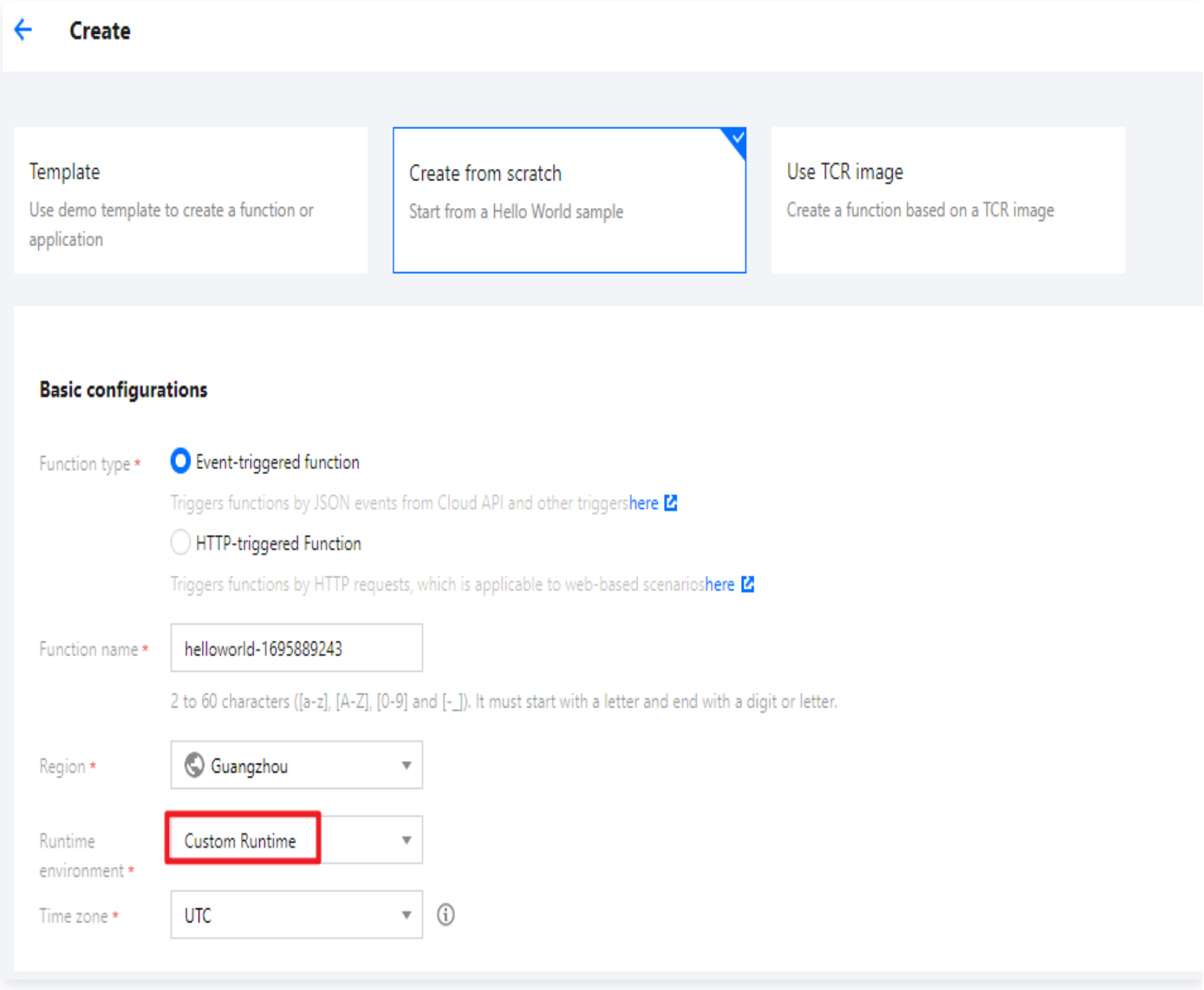
If a message similar to the following is returned, the invocation is successful.

```
{"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"
}
```

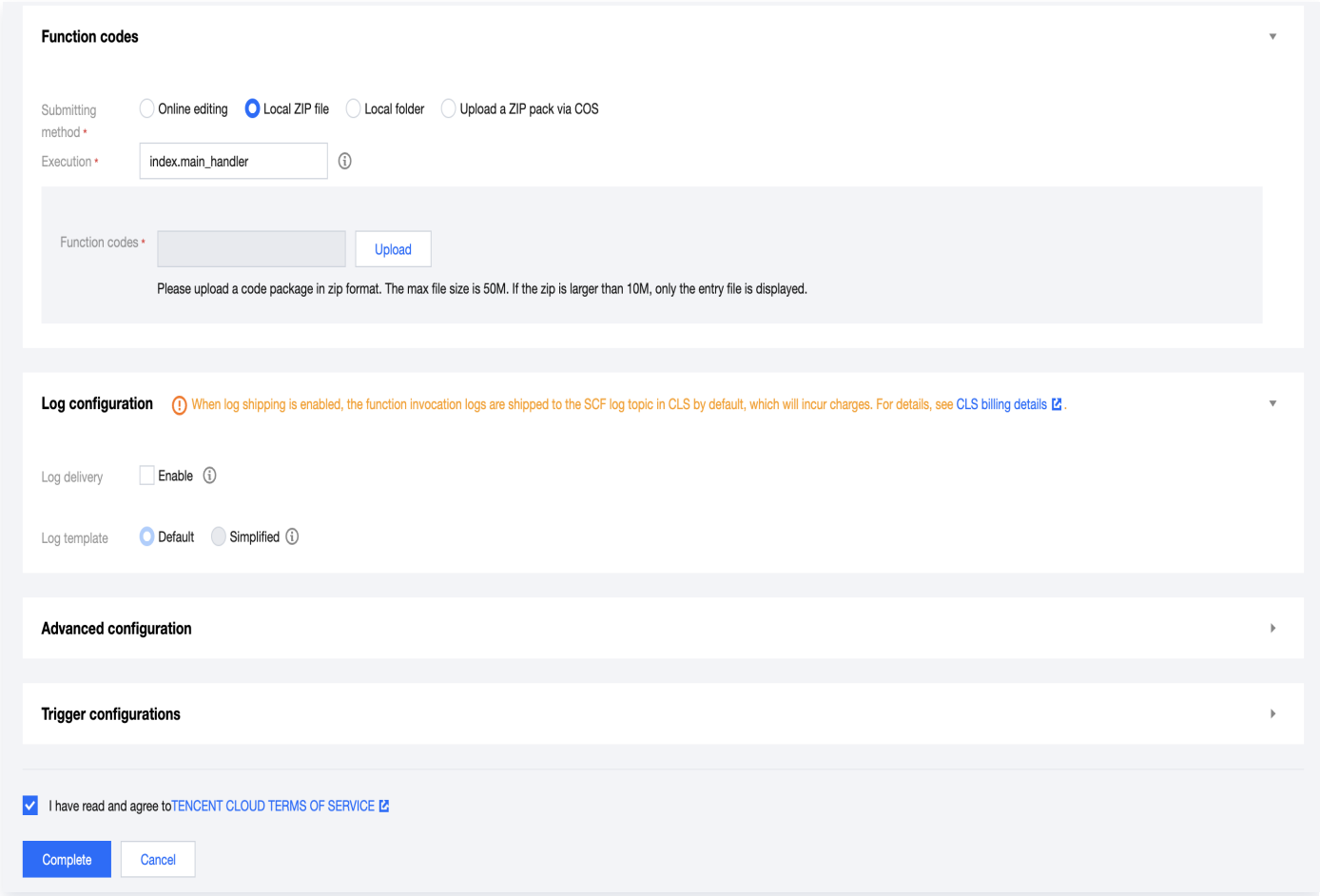
Using Console to create and release a function

Creating function version alias

1. Log in to the Serverless Console and click **Function Service** on the left sidebar.
2. At the top of the "Function Service" page, select the desired region for function creation and click **Create** to initiate the function creation process.
3. On the **Create** page, select **Create from scratch** and choose **Custom Runtime** in the **Runtime environment** section, as shown below:



4. In the "Function codes" section, configure the "Submitting method" and "Function codes", as illustrated below:

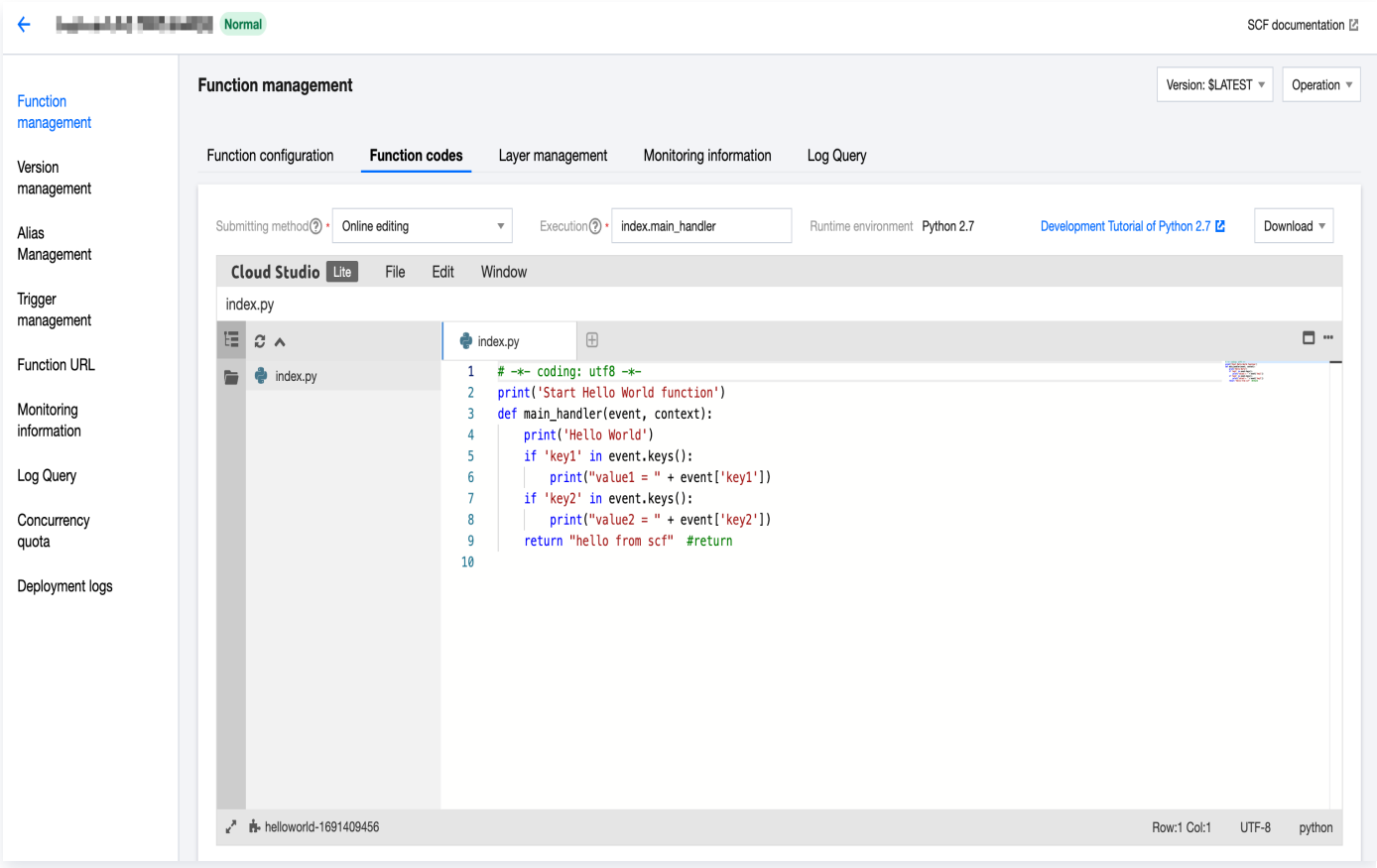


- **Submitting method**: Select "Upload a local zip package".
- **Execution**: Select the pre-packaged demo.zip.
- Advanced configuration: Expand the configuration items to set the "Initialization Timeout" and other related parameters.

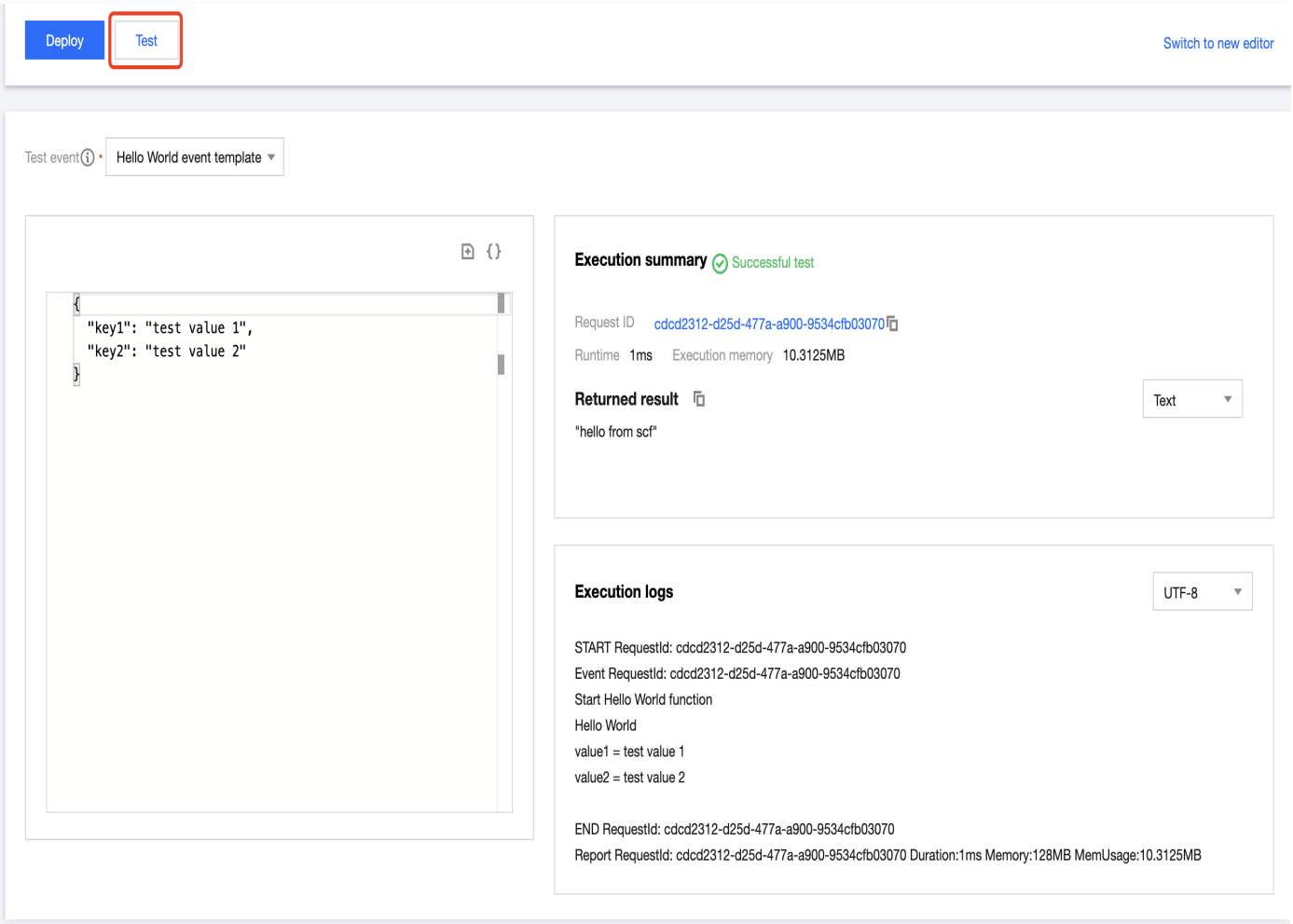
5. Click **Complete** to complete the function creation.

Invoking a function

1. Log in to the Serverless Console and click **Function Service** on the left sidebar.
2. At the top of the "Function Service" page, select the region where you want to invoke the function, and click on the desired function in the list to enter its details page.
3. Select **Function management** on the left, and choose the **Function codes** tab on the "Function management" page, as shown below:



4. Click **Test** below the editor, and the console will display the execution results and logs of the invocation, as shown below:



Deploying Image As Function
WebServer Image Function

Last updated: 2023-09-27 17:37:21

SCF supports developers in deploying container images as functions. Currently, two types of image functions are supported.

- **WebServer Image Function**: The image must contain an Http Server and listen on port 9000. Once the instance is launched, the event body is passed into the instance in the form of an Http request.
- **Job Image Function**: The image does not need to contain an Http Server and does not expose any ports. Once the instance is launched, it will execute specific commands based on the user-specified `CMD` and `EntryPoint` (as simple as the `date` command, or it can run specific code logic), and the instance will be automatically released after execution. The event body is injected into the instance in the form of environment variables.

This document provides an overview of the WebServer image function, including its background information, working principle, function development, function log printing, cold start optimization, billing instructions, and usage restrictions.

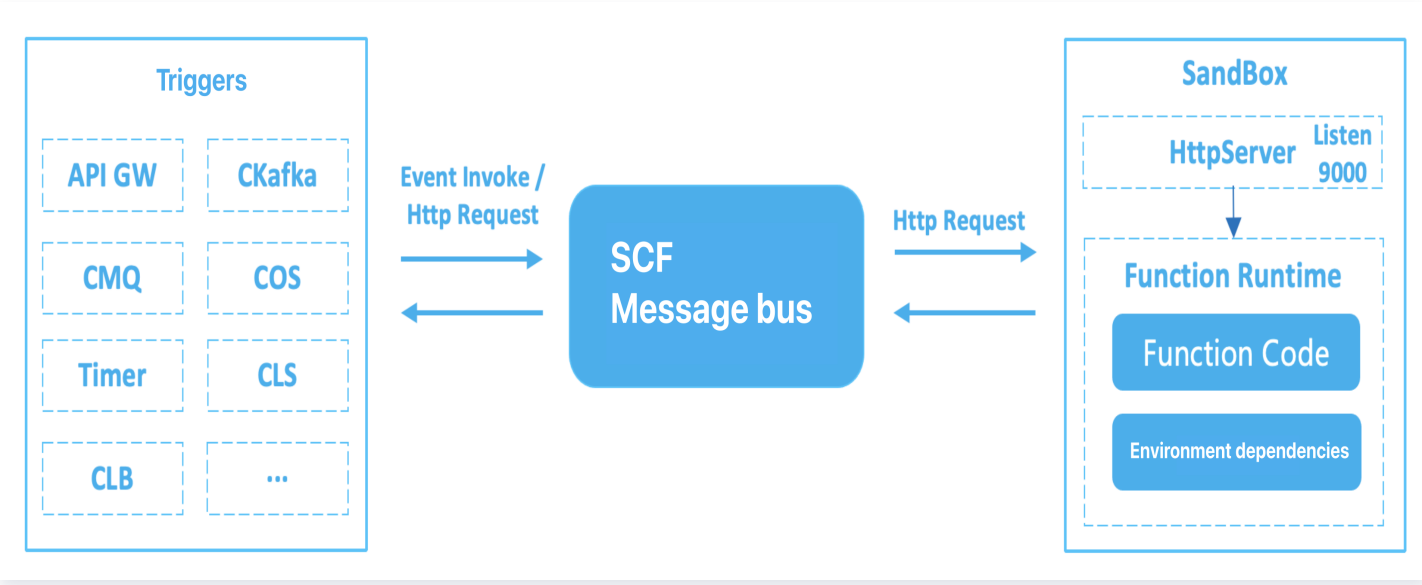
Background Information

From its inception, SCF was designed as a FaaS product based on a cloud-native architecture. With the support for deploying container images as functions at the Runtime layer, the product has moved towards a containerized ecosystem. On one hand, it solves the environmental dependency issues of function runtime, providing users with greater freedom. On the other hand, the product's form allows users to avoid technical barriers such as Kubernetes cluster management, security maintenance, and fault diagnosis, sinking the needs for elasticity and availability to the computing platform, further unleashing cloud computing capabilities.

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 start 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.

The function operates as shown below:



Developing Functions Deployed Based on Web Server Images

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 event data. For the structure, please refer to Summary of Trigger Event Message Structure.
- **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.

Description

- 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 detailed list is as follows:

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 (to use a timer trigger)

Built-in environment variables

In comparison to deployment based on code packages, custom image scenarios have made changes to the built-in environment variables within the container. You can reference these according to your actual needs.

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

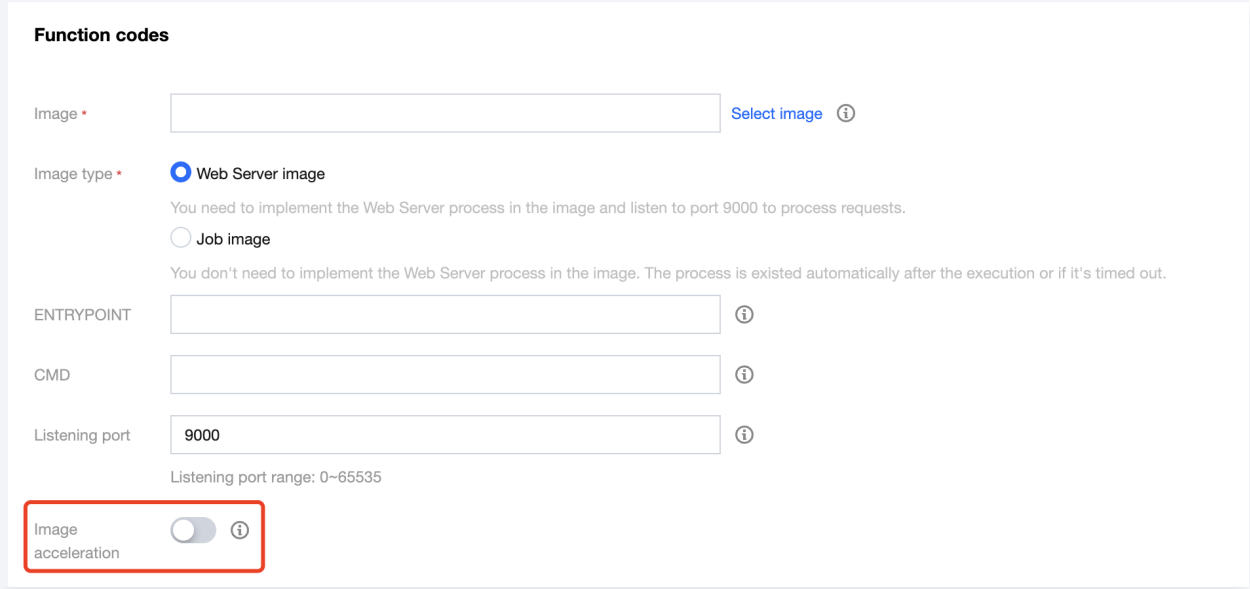
- Create an image repository and a function in the same region, so that the image can be pulled over VPC when the function triggers image pull, which makes the pull faster and more stable.
- The image should be as small as possible, that is, it should contain only the necessary basic environment and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

Enable 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.

Procedure

1. Log in to the Serverless Console and select **Function Service** from the left navigation bar.
2. In the **Function Service** list, select the image function that needs to be configured.
3. On the **Function Management** page, select **Function Code**, click **Edit** on the right side of the image configuration to enter edit mode.
4. Click on the to the right of Image Acceleration to enable the image acceleration mode, as shown below:



5. Click **Save**.

Precautions

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 description

The billable items of image-based functions are the same as those of code package-based functions. For more information on billing, please see Billing Mode.

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 < 256MB	256MB < Y < 512MB
256MB < X <512MB	512MB < Y < 1Gi
512MB < X < 1Gi	Y>1Gi

Image repository access

- Currently, only the TCR Enterprise Edition and Personal Edition are supported. For more information, please see Tencent Container Registry.
  - For more information on the TCR Enterprise Edition, please see Basic Image Repository Operations.
  - For more information on the TCR Individual Edition, please see Enabling Individual Image Repository.
- Currently, only images in a private image repository in the same region can be read.

Permission to read/write file in container

- By default, the `/tmp` directory is readable and writable. It is recommended to select `/tmp` when outputting files.
- 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)

Job Image Function

Last updated: 2023-09-27 21:39:27

SCF supports developers in deploying container images as functions. Currently, two types of image functions are supported.

- **WebServer Image Function**: The image must contain an HTTP Server and listen on port 9000. Once the instance is launched, the event body is passed into the instance in the form of an HTTP request.
- **Job Image Function**: The image does not need to contain an HTTP Server and does not expose any ports. Once the instance is launched, it will execute specific commands (as simple as the `date` command, or it can run specific code logic) based on the user-specified `CMD` and `EntryPoint`, and automatically release the instance after execution. The event body is injected into the instance in the form of environment variables.

This document provides an overview of the background information, working principles, function development, function log printing, cold start optimization, billing instructions, and usage limitations of the Job Image Function.

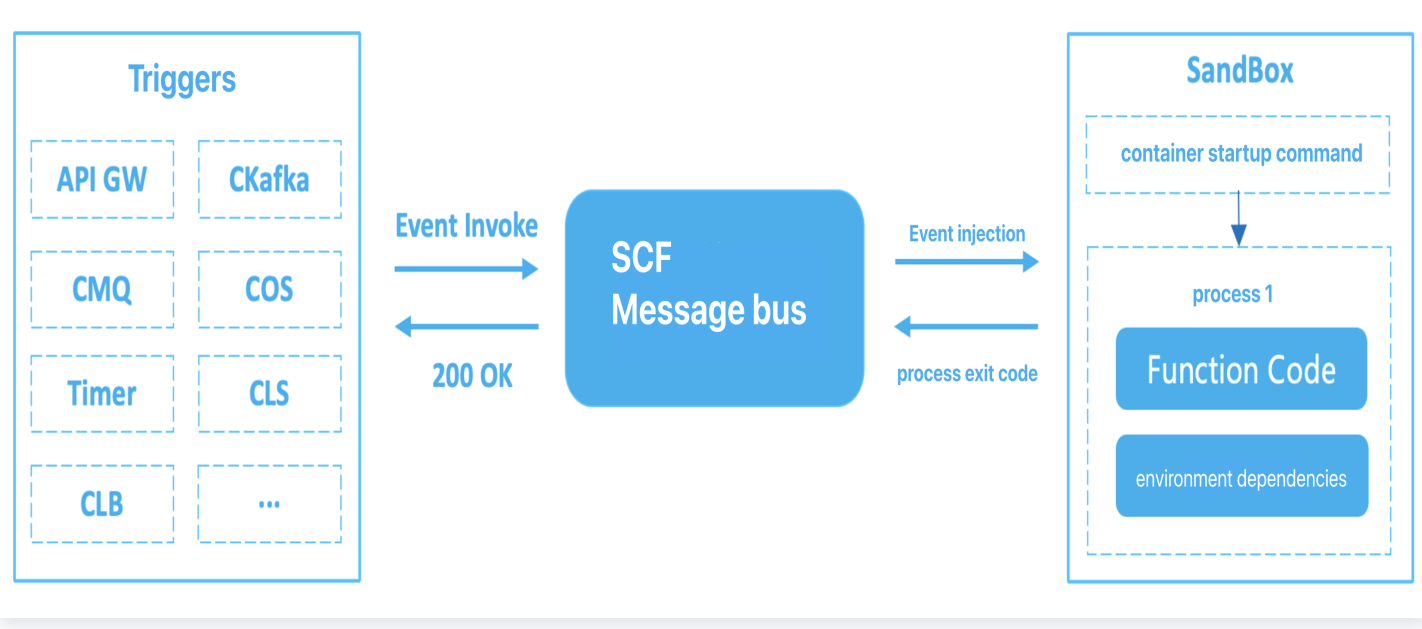
Background Information

With the launch of the WebServer image, runtime environment dependencies have been well resolved, and this feature also provides users with greater freedom. However, on the other hand, we found that the requirement for WebServer image functions to include an HTTP Server in the image has brought significant migration and transformation costs for cloud-native users using image functions. To this end, we have introduced a new type of Job image function, **which allows users to quickly migrate to SCF without any modifications to existing images**. This allows users to fully enjoy the convenience of Kubernetes cluster management, security maintenance, resource fault diagnosis, and more. It also offloads requirements for elasticity and availability to the computing platform, further unleashing the power of cloud computing.

How It Works

Currently, only event functions support Job Images.
During the initialization phase of a function instance, SCF obtains temporary usernames and passwords from the image repository as access credentials to pull the image. Currently, it responds to call requests asynchronously. That is, once a call request is received, the image is pulled successfully, and the instance is launched, it immediately returns the instance scheduling result (200 OK). It then asynchronously executes a specific process based on the startup command `Command` and parameters `Args` you defined. Finally, the instance is automatically destroyed after the lifecycle of the process ends.

Note: The event body will be injected into the instance as an environment command. The entire call process is illustrated in the diagram below:



Developing Functions Deployed Based on Job Images

Function input parameters

SCF_CUSTOM_CONTAINER_EVENT: This is a JSON type event body. Upon receiving a request, it is directly injected into the instance as an **environment variable**.

```
SCF_CUSTOM_CONTAINER_EVENT={"key1":"test value 1","key2":"test value 2"}
```

For the event structure, please refer to Summary of Trigger Event Message Structures.

Description

- Job image functions do not expose any ports after being launched, hence it is not possible to directly transmit HTTP requests to the instance.
- The total size limit for all environment variables in a Job image is 4KB. Therefore, please ensure that your event body size is within 4KB.

Other 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 Method

You can directly invoke it from the console or by using an API, and it also supports invocation through triggers.

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.

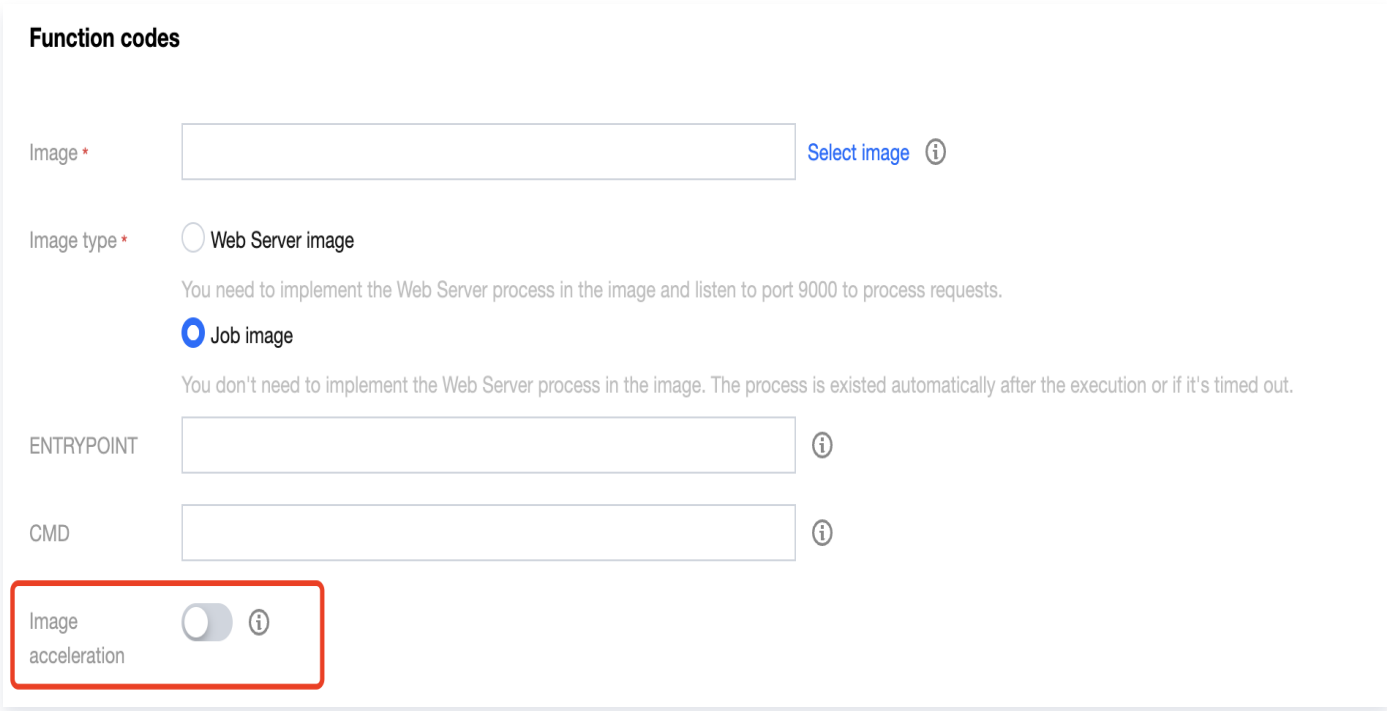
Enable Image Acceleration

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.

Procedure

1. Log in to the Serverless Console and select **Function Service** from the left navigation bar.
2. In the **Function Service** list, select the image function that needs to be configured.
3. On the **Function Management** page, select **Function Code**, click **Edit** on the right side of the image configuration to enter the editing mode.
4. Click on the to the right of Image Acceleration to enable the image acceleration mode, as shown in the figure below:



5. Click **Save**.

Precautions

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 description

The billable items of image-based functions are the same as those of code package-based functions. For more information on billing, please see Billing Mode.

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 < 256MB	256MB < Y < 512MB
256MB < X <512MB	512MB < Y < 1Gi
512MB < X < 1Gi	Y>1Gi

Image repository access

- Currently, only the TCR Enterprise Edition and Personal Edition are supported. For more information, please see Tencent Container Registry.
  - For more information on the TCR Enterprise Edition, please see Basic Image Repository Operations.
  - For more information on the TCR Individual Edition, please see Enabling Individual Image Repository.
- Currently, only images in a private image repository in the same region can be read.

Permission to read/write file in container

- By default, the `/tmp` directory is readable and writable. It is recommended to select `/tmp` when outputting files.
- 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

Last updated: 2023-09-27 21:40:03

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, please see Quick Start.
- Use a TCR Personal Edition image repository. For more information, please 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 the 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 the image repository you are using, and replace $USERNAME and $PASSWORD with your login credentials respectively.
docker login $YOUR_REGISTRY_URL --username $USERNAME --password $PASSWORD

# Building the image, please replace $YOUR_IMAGE_NAME with the image address you are using.
docker build -t $YOUR_IMAGE_NAME .

# Pushing the Image
docker push  $YOUR_IMAGE_NAME
```

Creating a Function

1. Log in to the Serverless console and click on **Function Service** in the left navigation bar.
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.
Image	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**.