Function Compute

User Guide

MORE THAN JUST CLOUD | Alibaba Cloud

User Guide

Programming language

Execution environment

When a function is called, the function codes are run in a restricted environment:

Environment configuration	Description
Execution user	A common user (without the root permission)
Code directory	env["FC_FUNC_CODE_PATH"]
Writable directory	/tmp (Other directories are read-only.)
Operating system	Linux 4.4.24-2.al7.x86_64
Network	It is allowed to access the Internet and the intranets of Alibaba Cloud services (such as OSS and Table Store) in the same regions.

You can use this Web Shell tool to experience the function execution environment.

Environments in different programming languages contain different commonly-used libraries. For more information, see the programming documentation in each language:

- Python
- Node.js

Use a code directory

If you pack some configuration files or data files with codes and then upload them together, and need to access these files using codes, the FC_FUNC_CODE_PATH environment variable must be used to obtain the absolute file path. The following illustrates how to obtain the path in Python, Node.js and Java respectively:

Python:

```
import os

def handler(event, context):
cfg_file = os.environ['FC_FUNC_CODE_PATH'] + '/config.json'
print cfg_file
```

Node.js:

```
exports.handler = function(event, context, callback) {
cfgFile = process.env['FC_FUNC_CODE_PATH'] + '/config.json';
console.log(cfgFile);
callback(null, 'done');
}
```

Java:

```
import com.aliyun.fc.runtime.Context;
import com.aliyun.fc.runtime.StreamRequestHandler;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class HelloFC implements StreamRequestHandler {
@Override
public void handleRequest(InputStream inputStream, OutputStream outputStream, Context context) throws
IOException {
String cfgFile = System.getenv("FC_FUNC_CODE_PATH") + "/config.json";
outputStream.write(cfgFile.getBytes());
}
}
```

Use an intranet domain name

When a function needs to access other cloud services, you are recommended to use an intranet domain name. On one hand, an intranet domain name can produce better performance. On the other hand, no Internet traffic fee is charged in this way. The following illustrates how to use an intranet domain name to access OSS in Python and Node.js respectively:

Python:

```
import oss2

def my_handler(event, context):
creds = context.credentials
auth = oss2.StsAuth(creds.accessKeyId, creds.accessKeySecret, creds.securityToken)
bucket = oss2.Bucket(auth, 'oss-cn-shanghai-internal.aliyuncs.com', 'my-bucket')
```

```
bucket.put_object('my-object', 'hello world')
```

Node.js:

```
var OSSClient = require('ali-oss').Wrapper;

exports.handler = function(event, context, callback) {
console.log(event.toString());

var ossClient = new OSSClient({
accessKeyId: context.credentials.accessKeyId,
accessKeySecret: context.credentials.accessKeySecret,
stsToken: context.credentials.securityToken,
region: 'oss-cn-shanghai',
internal: true,
bucket: 'my-bucket',
});

ossClient.put('my-object', new Buffer('hello world'))
.then(function(res) {
callback(null, res);
}).catch(function(err) {
callback(err);
});
};
```

Java

Java

Function Compute currently supports the following Java running environment:

- OpenJDK 1.8.0 (runtime = java8)

When using Function Compute in Java, a class must be defined and a pre-defined Function Compute interface must be implemented. A simplest function is defined as follows:

```
package example;

import com.aliyun.fc.runtime.Context;
import com.aliyun.fc.runtime.StreamRequestHandler;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class HelloFC implements StreamRequestHandler {

@Override
public void handleRequest(
InputStream inputStream, OutputStream outputStream, Context context) throws IOException {

outputStream.write(new String("hello world").getBytes());
}
}
```

Package name/Class name

A package and a class can have random names, but their names must correspond to the "handler" field of the created function. In the previous example, the package name is "example" and the class name is "HelloFC". Therefore, the handler specified during function creation is example.HelloFC::handleRequest. The format of "handler" is {package}.{class}::{method}.

Implement interface

The pre-defined Function Compute interface must be implemented in your code. In the previous example, StreamRequestHandler is implemented, inputStream is the data imported when the function is called, and outputStream is used to return the function execution result. For more information about the function interfaces, see Function interfaces.

context parameter

The context parameter contains the operation information of the function (such as the request ID and temporary AccessKey). The parameter type is com.aliyun.fc.runtime.Context. The Use context section introduces the function structure and usage.

Return value

The function that implements the StreamRequestHandler interface returns execution results by using the outputStream parameter.

The dependency of the com.aliyun.fc.runtime package can be referenced in the following pom.xml:

```
<dependency>
<groupId>com.aliyun.fc.runtime</groupId>
<artifactId>fc-java-core</artifactId>
<version>1.0.0</version>
</dependency>
```

Before a function is created, you must package project with its dependency fc-java-core into a JAR file. You can learn how to package a JAR file in Use a custom module. After the JAR package is created, use the fcli or console to upload the package. The following uses fcli as an example:

```
rockuw-MBP:hello-java (master) $ ls -lrt
total 16
-rw-r--r-- 1 rockuw staff 7690 Aug 31 19:45 hellofc.jar

>>> mkf hello-java -t java8 -h example.HelloFC::handleRequest -d ./functions/hello-java
>>> invk hello-java
hello world
>>>
```

Advanced usage

- Use context
- Use logging
- Function interface
- Use a custom module
- Handle exception

Use context

context is an object generated during the Function Compute operation, and it contains the operation information. You can use the information in code. The type of context is object, and its definition is as follows. You can click here to learn about its detailed:

```
package com.aliyun.fc.runtime;

public interface Context {

public String getRequestId();

public Credentials getExecutionCredentials();

public FunctionParam getFunctionParam();

public FunctionComputeLogger getLogger();
}
```

In this definition, context contains four elements:

- RequestId: The unique ID of this execution request. It can be recorded for reference in case any exception occurs in future.
- FunctionParam: Some basic information about the function, such as function name, function entry, function memory, and time-out period.

- ExecutionCredentials: A group of temporary keys acquired when Function Compute plays a service role you provide. Its valid interval is 5 minutes. You can use it in code to access related service (such as OSS). This prevents you from permanently adding your own AccessKey information in function code.
- Logger: A logger encapsulated by Function Compute. For details, see Use logging below.

For example, the following code uses a temporary key to upload a file to OSS:

```
package example;

import com.aliyun.fc.runtime.Context;
import com.aliyun.fc.runtime.Credentials;
import com.aliyun.fc.runtime.StreamRequestHandler;
import com.aliyun.oss.OSSClient;

import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class HelloFC implements StreamRequestHandler {

@Override
public void handleRequest(
InputStream inputStream, OutputStream outputStream, Context context) throws IOException {

String endpoint = "oss-cn-shanghai.aliyuncs.com";
String bucketName = "my-bucket";

Credentials creds = context.getExecutionCredentials();
OSSClient client = new OSSClient(
endpoint, creds.getAccessKeyId(), creds.getAccessKeySecret(), creds.getSecurityToken());
client.putObject(bucketName, "my-object", new ByteArrayInputStream(new String("hello").getBytes()));
outputStream.write(new String("done").getBytes());
}
}
```

Use logging

The information about your function that has been printed by context.getLogger() is collected into the logstore that was assigned by you when creating a service:

```
package example;

import com.aliyun.fc.runtime.Context;
import com.aliyun.fc.runtime.StreamRequestHandler;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
```

```
public class HelloFC implements StreamRequestHandler {

@Override
public void handleRequest(
InputStream inputStream, OutputStream outputStream, Context context) throws IOException {

context.getLogger().info("hello world");
outputStream.write(new String("hello world").getBytes());
}
}
```

The logs output by the previous code are:

```
message:2017-07-05T05:13:35.920Z a72df088-f738-cee3-e0fe-323ad89118e5 [INFO] hello world
```

context.getLogger().warn and context.getLogger().error can be respectively used to pack logs at WARN and ERROR levels.

Function interfaces

When you use the Java programming, a class must be implemented, which must implement a pre-defined Function Compute interface. Currently, two pre-defined interfaces can be implemented:

- StreamRequestHandler

This interface uses the stream request handler to receive the information (events) input when calling a function and return execution results. You need to read the input information from inputStream and to write the function execution result into outputStream after the read operation is completed. The first example in this document uses this interface.

- PojoRequestHandler<I, O>

This interface uses the generic method to allow you to customize the input and output types, but note that the types must be POJO. The following gives an example on how to use this interface.

```
// HelloFC.java
package example;

import com.aliyun.fc.runtime.Context;
import com.aliyun.fc.runtime.PojoRequestHandler;

public class HelloFC implements PojoRequestHandler<SimpleRequest, SimpleResponse> {

@Override
public SimpleResponse handleRequest(SimpleRequest request, Context context) {
String message = "Hello, " + request.getFirstName() + " " + request.getLastName();
return new SimpleResponse(message);
}
}



// SimpleRequest.java
package example;

public class SimpleRequest {
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 SimpleRequest() {}
public SimpleRequest(String firstName, String lastName) {
this.firstName = firstName;
this.lastName = lastName;
}
}



// SimpleResponse.java
package example;

public class SimpleResponse {
String message;

public String getMessage() {
return message;
}

public void setMessage(String message) {
this.message = message;
}

public SimpleResponse() {}
public SimpleResponse(String message) {
this.message = message;
}
}
```

Prepare an input file for calling:

```
{
"firstName": "FC",
"lastName": "aliyun"
}
```

Use fcli to call the result:

```
>>> invk hello-java -f /tmp/a.json
{"message":"Hello, FC aliyun"}
>>>
```

Use a custom module

If you want to use custom modules, you must pack the modules with code when packaging a JAR file. Maven and IDEA are used here to demonstrate how to package the OSS Java SDK into a jar.

Use maven to package the jar file

Add OSS Java SDK dependency to pom.xml:

```
<dependencies>
<dependency>
<groupId>com.aliyun.fc.runtime</groupId>
<artifactId>fc-java-core</artifactId>
<version>1.0.0</version>
</dependency>

<dependency>
<groupId>com.aliyun.oss</groupId>
<artifactId>aliyun-sdk-oss</artifactId>
<version>2.6.1</version>
</dependency>
</dependencies>
```

Add maven-assembly-plugin to pom.xml

```
<build>
<plugins>
<plugin>
<artifactId>maven-assembly-plugin</artifactId>
<version>3.1.0</version>
<configuration>
<descriptorRefs>
<descriptorRef>jar-with-dependencies</descriptorRef>
</descriptorRefs>
<appendAssemblyId>false</appendAssemblyId> <!-- this is used for not append id to the jar name -->
```

```
</configuration>
<executions>
<execution>
<id>make-assembly</id> <!-- this is used for inheritance merges -->
<phase>package</phase> <!-- bind to the packaging phase -->
<goals>
<goal>single</goal>
</goals>
</execution>
</executions>
</plugin>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>
<configuration>
<source>1.8</source>
<target>1.8</target>
</configuration>
</plugin>
</plugins>
</build>
```

Package the java file

```
mvn package
```

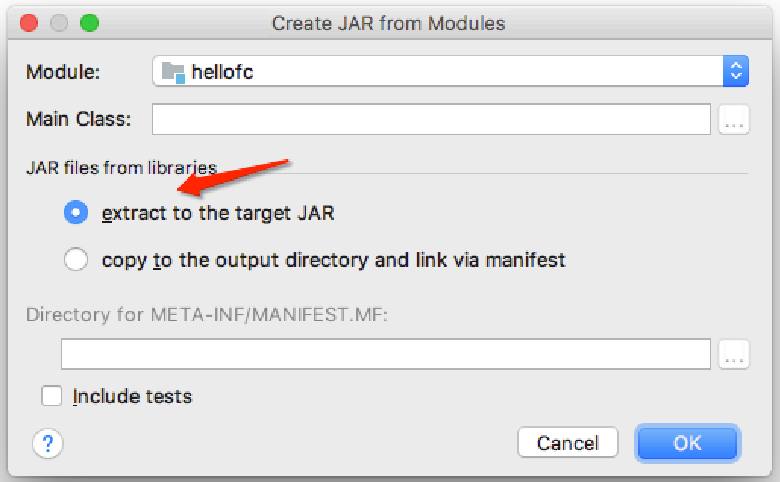
After completing the above steps, the dependent third-party jars are also packaged together into the jar. The generated jar will be stored in the target directory.

Use IDEA to package the jar file

Add OSS Java SDK dependency to pom.xml:

```
 <dependencies>
<dependency>
<groupId>com.aliyun.fc.runtime</groupId>
<artifactId>fc-java-core</artifactId>
<version>1.0.0</version>
</dependency>

<dependency>
<groupId>com.aliyun.oss</groupId>
<artifactId>aliyun-sdk-oss</artifactId>
<version>2.6.1</version>
</dependency>
</dependencies>
```

Configure JAR package export options:



Create JAR from Modules

Module: hellofc

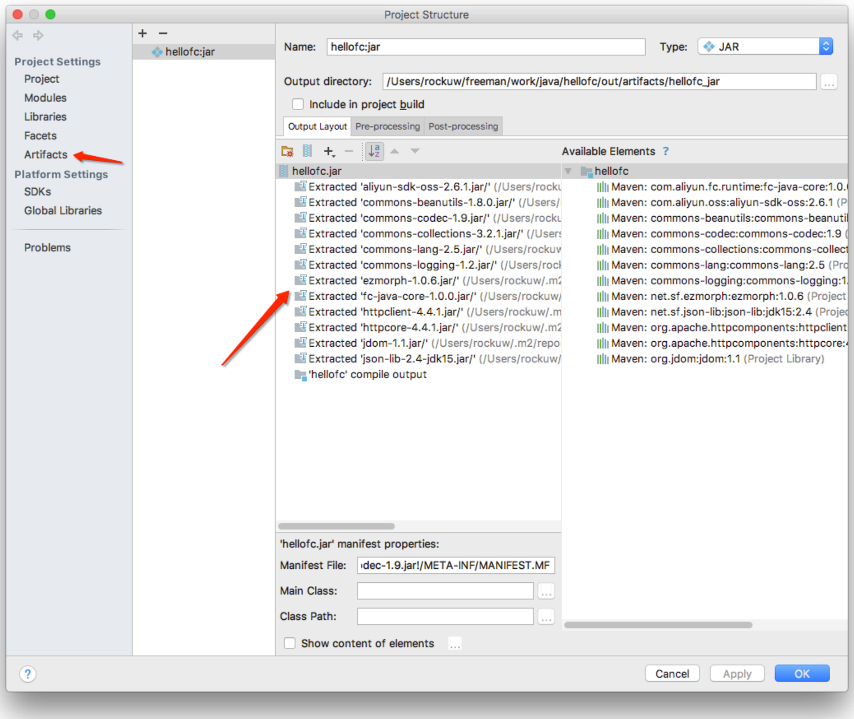
Main Class:

JAR files from libraries

- [✓ extract to the target JAR]
- copy to the output directory and link via manifest

Directory for META-INF/MANIFEST.MF:

☐ Include tests



Project Structure

Name: hellofc:jar Type: JAR

Output directory: /Users/rockuw/freeman/work/java/hellofc/out/artifacts/hellofc_jar

☐ Include in project build

```
hellofc.jar
  Extracted 'aliyun-sdk-oss-2.6.1.jar/' (/Users/rocku
  Extracted 'commons-beanutils-1.8.0.jar/' (/Users/
  Extracted 'commons-codec-1.9.jar/' (/Users/rocku
  Extracted 'commons-collections-3.2.1.jar/' (/Users
  Extracted 'commons-lang-2.5.jar/' (/Users/rockuw
  Extracted 'commons-logging-1.2.jar/' (/Users/rock
  Extracted 'ezmorph-1.0.6.jar/' (/Users/rockuw/.m2
  Extracted 'fc-java-core-1.0.0.jar/' (/Users/rockuw/
  Extracted 'httpclient-4.4.1.jar/' (/Users/rockuw/.m
  Extracted 'httpcore-4.4.1.jar/' (/Users/rockuw/.m2
  Extracted 'jdom-1.1.jar/' (/Users/rockuw/.m2/repo
  Extracted 'json-lib-2.4-jdk15.jar/' (/Users/rockuw/
  'hellofc' compile output
```

Available Elements ?

```
hellofc
  Maven: com.aliyun.fc.runtime:fc-java-core:1.0.
  Maven: com.aliyun.oss:aliyun-sdk-oss:2.6.1 (P
  Maven: commons-beanutils:commons-beanutil
  Maven: commons-codec:commons-codec:1.9 (
  Maven: commons-collections:commons-collect
  Maven: commons-lang:commons-lang:2.5 (Pro
  Maven: commons-logging:commons-logging:1.
  Maven: net.sf.ezmorph:ezmorph:1.0.6 (Project
  Maven: net.sf.json-lib:json-lib:jdk15:2.4 (Projec
  Maven: org.apache.httpcomponents:httpclient
  Maven: org.apache.httpcomponents:httpcore:4
  Maven: org.jdom:jdom:1.1 (Project Library)
```

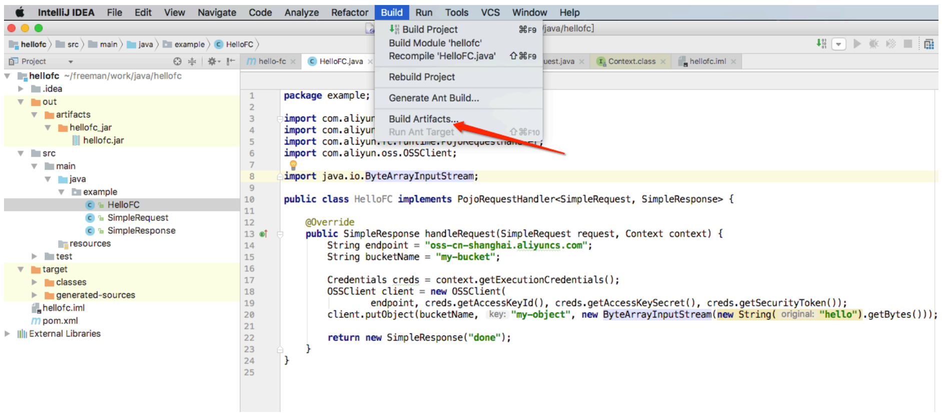
'hellofc.jar' manifest properties:

Manifest File: dec-1.9.jar!/META-INF/MANIFEST.MF

Main Class:

Class Path:

☐ Show content of elements



```
package example;

import com.aliyur
import com.aliyur
import com.aliyun.
import com.aliyun.oss.OSSClient;

import java.io.ByteArrayInputStream;

public class HelloFC implements PojoRequestHandler<SimpleRequest, SimpleResponse> {

    @Override
    public SimpleResponse handleRequest(SimpleRequest request, Context context) {
        String endpoint = "oss-cn-shanghai.aliyuncs.com";
        String bucketName = "my-bucket";

        Credentials creds = context.getExecutionCredentials();
        OSSClient client = new OSSClient(
                endpoint, creds.getAccessKeyId(), creds.getAccessKeySecret(), creds.getSecurityToken());
        client.putObject(bucketName, key: "my-object", new ByteArrayInputStream(new String( original: "hello").getBytes()));

        return new SimpleResponse("done");
    }
}
```

Verify the jar file

```
 rockuw-MBP:hello-java (master) $ ls -lrth
total 6520
-rw-r--r-- 1 rockuw staff 3.2M Aug 31 21:03 hellofc.jar
rockuw-MBP:hello-java (master) $ jar -tf hellofc.jar | head
Picked up _JAVA_OPTIONS: -Duser.language=en
META-INF/MANIFEST.MF
example/
example/HelloFC.class
example/SimpleRequest.class
example/SimpleResponse.class
META-INF/
META-INF//
org/
org//
org/apache/
```

Use maven to package the jar file and put dependency .jar files in a separate /lib directory

As project dependencies increase, the size of the jar becomes larger. The user-uploaded jar or zip code will be decompressed before execution. Therefore, in the two previous implementations, there is a problem that our packaged jar contains a large number of class files, which will undoubtedly increase the decompression time and thus increase the first startup time of the function.

A better practice is to put your dependency .jar files in a separate /lib directory.

Here is an example using maven-dependency-plugin :

```
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-dependency-plugin</artifactId>
<executions>
<execution>
<id>copy-dependencies</id>
<phase>prepare-package</phase>
```

```
<goals>
<goal>copy-dependencies</goal>
</goals>
<configuration>
<outputDirectory>${project.build.directory}/classes/lib</outputDirectory>
<includeScope>runtime</includeScope>
</configuration>
</execution>
</executions>
</plugin>
```

After mvn package, the packaged jar’ s directory structure is:

```
*/*.class
lib/*.jar
```

Handle exception

If an exception occurs when your function is executed, Function Compute captures the exception and returns its information. The following codes are used as an example:

```
package example;

import com.aliyun.fc.runtime.Context;
import com.aliyun.fc.runtime.StreamRequestHandler;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class HelloFC implements StreamRequestHandler {


@Override
public void handleRequest(InputStream inputStream, OutputStream outputStream, Context context) throws
IOException {
throw new IOException("oops");
}
}
```

The response returned upon calling is:

```
>>> invk hello-java -f /tmp/a.json
{
"errorMessage" : "oops",
"errorType" : "java.io.IOException",
"errorCause" : "oops",
"stackTrace" : [ "example.HelloFC.handleRequest(HelloFC.java:15)" ]
}
Error: Request id: 45dd8d90-6b78-cce3-087c-8bf4ebc6c9af. Error type: UnhandledInvocationError
```

When an exception occurs, the HTTP header in the function calling response includes X-Fc-Error-Type: UnhandledInvocationError.

Node.js

Node.js

Function Compute (FC) currently supports the following Node.js runtime environements:

- Nodejs6.14 (runtime = nodejs6)
- Nodejs8.9.0 (runtime = nodejs8)

When a function has runtime set to be nodejs*, a Node.js function must be exported as an entry point module for FC to call. A simple function can be written as follows:

```
exports.handler = function(event, context, callback) {
callback(null, 'hello world');
};
```

- Function name
- exports.handler the "handler" part must be the same with the second half of the "Handler" value given upon function creation. For example, if the "Handler" is set to be index.handler, FC automatically calls the handler function exported in index.js.
- event parameter
    - The event parameter is the data payload passed to your function upon invocation. The parameter type is Buffer.
- context parameter
    - The context parameter contains the operation information of the function (such as the request ID and temporary credentials). The parameter type is object. The Use context section describes object structure and usage.
- callback parameter
    - The callback parameter returns the result of a invocation. Its function signature is function(err, data). Similar to how callback is normally used in Node.js, its first parameter is error and second parameter is data. If err is not blank when the function is called, the function returns HandledInvocationError; otherwise, the function returns the data. If the data type is Buffer, the data is returned directly. If the data type is object, the data is converted into a JSON string and then returned.

If the data is of other types, it is converted into a string and then returned.

Advanced usage

- Event Usage
- Context Usage
- Logging
- Built-in Module Usage
- Custom Module Usage
- Calling an External Command
- Understanding Callback
- Exception Handling

Event usage

Parameter event is the data payload passed to your function upon invocation. FC does not do any transformation nor understand its content. The event passed to an invocation is of Buffer type. You can interpret the event parameter in your function. For example, if the event data is a JSON string, you can convert it into an object:

```
exports.handler = function(event, context, callback) {
var eventObj = JSON.parse(event.toString());
callback(null, eventObj['key']);
};
```

Context usage

Parameter context is an object generated during the FC invocation, and it contains information at run time that can be retrived from the context, which is of type object. The object has the following structure:

```
{
'requestId': 'b1c5100f-819d-c421-3a5e-7782a27d8a33',
'credentials': {
'accessKeyId': 'STS.access_key_id',
'accessKeySecret': 'access_key_secret',
'securityToken': 'security_token',
},
'function': {
'name': 'my-func',
'handler': 'index.handler',
'memory': 128,
'timeout': 10,
},
'service': {
'name': 'my-service',
'logProject': 'my-log-project',
'logStore': 'my-log-store',
},
'region': 'cn-shanghai',
'accountId': '123456'
}
```

In this definition, context contains six elements:

1. requestId: an unique ID of an invocation request. It can be refered in case an exception occurs during an invocation.
2. function: structure that contains function information such as function name, handler, memory, and timeout.
3. credentials: a group of temporary keys acquired when FC assumes the service role associated with the service that is valid for 5 minutes. The temporary credentials can be used to access other cloud services (such as OSS) inside your function. Having credentials in the context object prevents users from hardcoding their crendentials into function codes.
4. service: structure that contains service information that the called function belongs to, such as service name, Log Service logProject and logstore.
5. region: region that the called function belongs to, such as cn.shanghai.
6. accountId: ID of the Alibaba Cloud account that the called function belongs to.

For example, the following code uses temporary credentials to upload a file to OSS:

```
var OSSClient = require('ali-oss').Wrapper;

exports.handler = function(event, context, callback) {
console.log(event.toString());

var ossClient = new OSSClient({
accessKeyId: context.credentials.accessKeyId,
accessKeySecret: context.credentials.accessKeySecret,
stsToken: context.credentials.securityToken,
region: 'oss-cn-shanghai',
bucket: 'my-bucket',
});

ossClient.put('my-object', new Buffer('hello, fc')).then(function(res) {
callback(null, 'put object');
}).catch(function(err) {
callback(err);
});
};
```

Logging

The logs printed through console.log in the code is stored into the logstore that was given upon

Service creation/update.

```
exports.handler = function(event, context, callback) {
console.info(null, 'hello world');
callback(null, 'hello world');
};
```

The log printed by the previous code is:

```
message:2017-07-05T05:13:35.920Z a72df088-f738-cee3-e0fe-323ad89118e5 [INFO] hello world
```

console.warn and console.error can be used to log at WARN and ERROR levels, respectively.

Log level can be changed by using console.setLogLevel. Log levels rank from highest to lowest:

error, warn, info, verbose, debug ; The corresponding interfaces of these log levels are console.error, console.warn, console.info, console.log, and console.debug.

```
'use strict';
exports.handler = function(evt, ctx, cb) {
console.setLogLevel("error");
console.error("console error 1");
console.info("console info 1");
console.warn("console warn 1");
console.log("console log 1");

console.setLogLevel("warn");
console.error("console error 2");
console.info("console info 2");
console.warn("console warn 2");
console.log("console log 2");

console.setLogLevel("info");
cb(null, evt);
};
```

The logs printed by the previous code are:

```
message:2017-07-05T05:13:35.920Z a72df088-f738-cee3-e0fe-323ad89118e5 [ERROR] console error 1
message:2017-07-05T05:13:35.920Z a72df088-f738-cee3-e0fe-323ad89118e5 [ERROR] console error 2
message:2017-07-05T05:13:35.920Z a72df088-f738-cee3-e0fe-323ad89118e5 [WARN] console warn 2
```

Built-in modules usage

In addition to standard Node.js modules, FC Node.js runtimes also include some frequently used modules for easier imports. The following modules are currently included in FC Node.js runtime:

- co: control flow

- gm: image processing library
- ali-oss: Alibaba Cloud OSS SDK
- aliyun-sdk: Alibaba Cloud SDK
- @alicloud/fc: Function Compute SDK
- opencv: computer vision library

Module	Version	Info	Link
co	4.6.0	Control flow library	https://github.com/tj/co
gm	1.23.0	Image processing library	https://github.com/aheckmann/gm
ali-oss	4.10.1	OSS SDK	https://github.com/ali-sdk/ali-oss
ali-mns	2.6.5	MNS SDK	https://github.com/InCar/ali-mns
tablestore	4.0.11	OTS SDK	https://github.com/aliyun/aliyun-tablestore-nodejs-sdk
aliyun-sdk	1.10.12	Aliyun SDK	https://github.com/aliyun-UED/aliyun-sdk-js
@alicloud/fc2	2.0.3	Function Compute SDK	https://github.com/aliyun/fc-nodejs-sdk
opencv	6.0.0	Computer vision library	https://github.com/peterbraden/node-opencv
body	5.1.0	Http body parsing library	https://github.com/Raynos/body
raw-body	2.3.2	Http body parsing library	https://github.com/stream-utils/raw-body

For example, the following function uses gm to flip an image:

```
var gm = require('gm').subClass({imageMagick: true});

exports.handler = function(event, context, callback) {
gm(event)
.flip()
.toBuffer('PNG',function (err, buffer) {
if (err) return callback(err);
callback(null, buffer);
});
};
```

NOTE: The above function uses event directly as binary representation of an image, and returns flipped image in binary.

Custom modules usage

Custom modules can be packaged along with code. The following instruction details how to use fcli to add a module MySQL module to access a MySQL database:

Create a directory to save function code and modules:

```
mkdir /tmp/code
```

Create a new code file, such as /tmp/code/index.js, and use mysql module in code:

```
var mysql = require('mysql');

exports.handler = function(event, context, callback) {
var connection = mysql.createConnection({
host : 'localhost',
user : 'me',
password : 'secret',
database : 'my_db'
});

connection.connect();

connection.query('SELECT 1 + 1 AS solution', function (error, results, fields) {
if (error) return callback(error);
console.log('The solution is: ', results[0].solution);
callback(null, results[0].solution);
});

connection.end();
};
```

Install mysql module in the /tmp/code directory:

```
cd /tmp/code
npm install mysql
```

After installation, the /tmp/code directory must look like:

```
ls -l /tmp/code
-rw-r--r-- 1 rockuw wheel 511 Aug 15 17:58 index.js
drwxr-xr-x 13 rockuw wheel 442 Aug 15 18:01 node_modules
```

Use fcli to create and call the function:

```
 ./fcli shell
mkf my-func -h index.handler --runtime nodejs6 -d /tmp/code
invk my-func
```

Call an external command

In case your function needs to run some binary files that are not compiled by Node.js (such as executable files complied by shell scripts, C++, or Golang). You can still package and run those commands in functions. The following example describes how to run a shell script:

```
var exec = require('child_process');

exports.handler = function(event, context, callback) {
var scriptPath = process.env['FC_FUNC_CODE_PATH'] + '/script.sh';
exec.exec('bash '+scriptPath, {}, function(err, stdout, stderr) {
if (err) return callback(err);
console.log(stdout, stderr);
callback(null, stdout);
});
};
```

Note that the executable files complied by C, C++, or Golang must be compatible with the runtime environment of FC:

- Linux kernel version: Linux 4.4.24-2.al7.x86_64
- docker base image: docker pull node:6.10

Understand callback

Node.js uses an asynchronous programming model, therefore your function must call callback to return data or error.

1. Make sure that the callback is called.

If a function does not call callback, the system assumes that the function invocation is still in progress and thus waits for the function result until the operation times out. For example, when the following function is called, a timeout error is returned:

```
exports.handler = function(event, context, callback) {
console.log('hello world');
};
```

Calling result:

```
{"errorMessage":"Function timed out after 3 seconds"}
```

2. The function is completed after callback is called.

After callback is called, the function invocation ends. If callback is called multiple times, only the first result is effective. Make sure that all tasks are completed before the calling callback; otherwise, the remaining tasks are not executed. For example, the following function returns “hello world” but **does not** print “message”：

```
exports.handler = function(event, context, callback) {
callback(null, 'hello world');
callback(null, 'done');
setTimeout(function() {
console.log('message');
}, 1000);
};
```

Handle exception

Two types of errors can be returned by functions in Node.js runtime. The error types are returned in HTTP response Header field (X-Fc-Error-Type):

1. HandledInvocationError: error returned by the first callback parameter
2. UnhandledInvocationError: Other errors, such as a reception anomaly, a time-out error, or an OOM

Example 1, returning a HandledInvocationError:

```
exports.handler = function(event, context, callback) {
callback(new Error('oops'));
};
```

The response returned upon invocation is:

```
{
"errorMessage": "oops",
"errorType": "Error",
"stackTrace": [
"at exports.handler (/code/index.js:2:12)"
]
}
```

Example 2, returning an UnhandledInvocationError:

```
exports.handler = function(event, context, callback) {
throw new Error('oops');
};
```

The response returned upon invocation is:

```
{"errorMessage":"Process exited unexpectedly before completing request"}
```

Python

Python

Function Compute currently supports the following Python runtimes:

- Python 2.7 (runtime = python2.7)
- Python 3.6 (runtime = python3)

A simple Python function is defined as follows:

```
def my_handler(event, context):
return 'hello world'
```

Function name

my_handler must correspond to the “Handler” field of the created function. For example, if the Handler is set to main.my_handler upon the function creation, Function Compute automatically loads the my_handler function defined in main.py.

event parameter

The event parameter is the request payload when you invoke a function. event is str type in Python2.7 and bytes type in Python3.

context parameter

The context parameter contains function invocation metadata (such as the request ID and temporary AccessKey). The parameter type is FCContext. The Use context section introduces the function structure and usage.

Return value

The return value of a function is returned to you as the result of function invocation. It can be of any type. For primitive type, Function Compute converts the result to a string before returning it. For a complex type, Function Compute serializes the result into JSON string before returning it.

Advanced usage

- Use event
- Use context
- Use logging
- Use a built-in module
- Use a custom module
- Call an external command
- Handle exception

Use event

event is the request payload when you invoke a function. It can be a simple string, a JSON string, or an image (binary data). The event parameter in a function is a byte stream. Its type in Python2.7 is str and in Python3 is bytes. You can convert the event parameter in a function based on the actual situation. For example, if the imported data is a JSON string, you can convert it into a dict:

```
# -*- coding: utf-8 -*-
import json
def my_handler(event, context):
evt = json.loads(event)
return evt['key']
```

Use context

context is an object generated during the Function Compute operation, and it contains useful function metadata. You can use the metadata in codes. The type of context is FCContext, and it is defined as follows:

```
class Credentials:
def __init__(self, access_key_id, access_key_secret, security_token):
self.access_key_id = access_key_id
self.access_key_secret = access_key_secret
self.security_token = security_token

class ServiceMeta:
def __init__(self, service_name, log_project, log_store):
self.name = service_name
self.log_project = log_project
self.log_store = log_store

class FunctionMeta:
def __init__(self, name, handler, memory, timeout):
self.name = name
self.handler = handler
self.memory = memory
self.timeout = timeout

class FCContext:
def __init__(self, account_id, request_id, credentials, function_meta, service_meta, region):
self.requestId = request_id
self.credentials = credentials
self.function = function_meta
self.request_id = request_id
self.service = service_meta
self.region = region
self.account_id = account_id
```

In this definition, context contains six attributes:

- request_id: The unique ID of function invocation. It can be recorded for debugging purpose in case an exception occurs in the future.
- function: Function invocation metadata, such as the function name, function entry, function memory, and time-out period.
- credentials: Temporary credentials that assumed by Function Compute from the service role. A new temporary credential is generated every few minutes when the previous one expires. You can use it in codes to access related service (such as OSS). This prevents you from permanently adding your own AccessKey information in function codes.
- service: Service metadata. It includes the service name, as well as the logProject and logstore of the accessed Log Service.
- region: The region of the function is being invoked, such as cn.shanghai.
- account_id: The caller’ s Alibaba Cloud account ID.

For example, the following code uses a temporary credentials to upload a file to OSS:

```
import json
import oss2

def my_handler(event, context):
evt = json.loads(event)
creds = context.credentials
auth = oss2.StsAuth(creds.access_key_id, creds.access_key_secret, creds.security_token)
bucket = oss2.Bucket(auth, evt['endpoint'], evt['bucket'])
```

```
bucket.put_object(evt['objectName'], evt['message'])
return 'success'
```

Use logging

You can log information in your function through either printing to stdout or using logging framework.

Use print to output logs in stdout. This outputs the original information to logs.

```
 def my_handler(event, context):
print 'hello world'
return 'done'
```

Output:

```
message:hello world
```

Use logging module to output logs in pretty format. The log output includes timestamp, request ID, and log level.

```
 import logging

def my_handler(event, context):
logger = logging.getLogger()
logger.info('hello world')
return 'done'
```

Output:

```
 message:2017-07-05T05:13:35.920Z a72df088-f738-cee3-e0fe-323ad89118e5 [INFO] hello world
```

You are recommended to use the logging module to print logs, for the logs output by this method automatically contain information such as request ID, which can be used for trouble shooting.

Use a built-in module

Beside the standard Python modules, the Python runtime includes some frequently used modules for your direct reference. Currently, the following modules are supported:

- oss2: OSS SDK
- tablestore: Table Store SDK

- wand: image processing database
- opencv
- numpy
- scipy
- matplotlib
- scrapy

For example, the function that uses wand to flip an image is as follows:

```
from wand.image import Image

def my_handler(event, context):
with Image(blob=event) as img:
print img.size
with img.clone() as i:
i.rotate(180)
return i.make_blob()
```

Note: The preceding function directly uses event as the binary data of the image, and directly returns the generated image as the binary data.

TIPS: Click fc-python-demo to view the small demos used by other third-party databases.

Use a custom module

If you need to use a custom module, the module must be packed with its codes. The following describes how to use fcli to add a module PyMySQL that can access MySQL:

Create a directory to save codes and references:

```
mkdir /tmp/code
```

Create a new code file, such as /tmp/code/main.py, and use pymysql in codes:

```
 import pymysql.cursors

# Connect to the database
connection = pymysql.connect(host='localhost',
user='user',
password='passwd',
db='db',
charset='utf8mb4',
cursorclass=pymysql.cursors.DictCursor)

def my_handler(event, context):
```

```
with connection.cursor() as cursor:
# Read a single record
sql = "SELECT count(*) FROM `users`"
cursor.execute(sql)
result = cursor.fetchone()
print(result)
return result
```

Install references in the /tmp/code directory:

```
 cd /tmp/code
pip install -t . PyMySQL
```

After installation, the /tmp/code directory is displayed as follows:

```
 ls -l /tmp/code
drwxr-xr-x 9 rockuw staff 306 Jul 5 16:48 PyMySQL-0.7.11.dist-info
-rw-r--r-- 1 rockuw staff 74 Jul 5 16:02 main.py
drwxr-xr-x 26 rockuw staff 884 Jul 5 16:48 pymysql
```

Use fcli to create and call a function:

```
 ./fcli shell
mkf my-func -h main.my_handler --runtime python2.7 -d /tmp/code
invk my-func
```

Note that if the executable files or database files of the referenced module are compiled by C, C++, or go, see sbox.

Call an external command

Your function may need to use some tools that are unavailable in Python (such as executable files complied by shell scripts, C++, or go). You can pack them with codes and then use them by running external commands in functions. The following example describes how to run a shell script:

```
import os
import subprocess

def my_handler(event, context):
script_path = os.environ.get('FC_FUNC_CODE_PATH') + '/script.sh'
ret = subprocess.check_output(['bash', script_path])
return ret
```

Note that the executable files complied by C, C++, or go must be compatible with the running environment of Function Compute. The Python running environment of Function Compute is:

- Linux kernel version: Linux 4.4.24-2.al7.x86_64
- docker basic image: docker pull python:2.7; docker pull python:3.6

The sbox command of the fcli tool is recommended. The following describes how to install mysql-python (including .so files) when runtime is python2.7:

Run sbox -d code -t python2.7, wherein -d specifies the directory of the codes, and it is attached to the "/code" location in the sandbox environment', and -t specifies the language type. (Note: When this command is executed for the first time, the image must be pulled. This operation may take a longer time. Please be patient.)

When the sandbox environment starts, run pip install -t . mysql-python to install a reference database.

After the installation is completed, run exit to exit the sandbox environment. Now, the mysql-python database (including _mysql.so) has been installed in the code directory.

Note:

To run the sbox command, you must first have docker installed on your host. For more information about how to install docker, see the relevant documentation.

The image used by sbox is available on the official docker image database, but the access to this database by users in China may be slow. We recommend that you use the Alibaba Cloud image acceleration service.

You must have the root permission when using docker in Linux. Therefore, you must use sudo fcli shell to start the command line tool, or you can refer to the relevant documentation to change the settings for managing docker as a non-root user.

We recommend that you pack third-party databases, test functions, and investigate issues in the local sandbox environment. This can help to mitigate errors occur due to environment difference. In particular, when your functions depend on binary files, compile relevant dependencies in the local sandbox environment.

```
songluo@demo $ fcli shell
Welcome to the function compute world. Have fun!
>>> sbox -d code -t python2.7
```

```
Entering the container. Your code is in the /code direcotry.
root@c5adc6ffd861:/code# pip install -t . mysql-python
Collecting mysql-python
Downloading MySQL-python-1.2.5.zip (108kB)
100% |████████████████████████████████| 112kB 440kB/s
Building wheels for collected packages: mysql-python
Running setup.py bdist_wheel for mysql-python ... done
Stored in directory: /root/.cache/pip/wheels/38/a3/89/ec87e092cfb38450fc91a62562055231deb0049a029054dc62
Successfully built mysql-python
Installing collected packages: mysql-python
Successfully installed mysql-python-1.2.5
root@c5adc6ffd861:/code# exit
```

Handle exceptions

If an exception occurs when your function is executed, Function Compute captures the exception and returns its information. The following codes are used as an example:

```
def my_handler(event, context):
raise Exception('something is wrong')
```

The response returned upon calling is:

```
{
"errorMessage": "something is wrong",
"errorType": "Exception",
"stackTrace": [
[
"File \"/code/main.py\"",
"line 2",
"in my_handler",
"raise Exception('something is wrong')"
]
]
}
```

When an error occurs, the HTTP header in the function calling response includes X-Fc-Error-Type: UnhandledInvocationError.

Service management

Create a service

Service is the unit for managing Function Compute resources. All functions belonging to a service share some common settings, such as authorization and log configuration. You can use the console or command line tool to create services.

Service attributes

When creating a service, specify the following attributes:

- serviceName (required): Name of the service. It must be unique in one Alibaba Cloud account and meets the following restrictions:
  - The name must consist of English letters (a–z) (A–Z), numbers (0–9), underscores (_), and hyphens (-).
  - The first character must be an English letter (a–z) (A–Z) or underscore (_).
  - The name is case-sensitive.
  - The name must contain 1–128 characters.
- description (optional): Description of the service.
- role (optional): The role grants Function Compute permissions to access user’ s cloud resources or run functions which may access cloud resources. For example:
  - Use your Log Service resource to store function execution logs.
  - Run the functions that need to access other cloud resources.
  - For more information about the role, see Function Compute role management.
- logConfig (optional): Sets the Log Service project and LogStore which are used to store function execution logs.
  - If you have not configured this attribute, you cannot view the function execution logs. We recommend that you enable Log Service and set this attribute.

Note: The log configuration uses Alibaba Cloud Log Service. Log Service charges certain resource reservation fees, which means that even if you do not write any logs, you still pay fees. For more information, see Log Service charging description.

vpcConfig (optional): Sets the VPC config to enable VPC access for your functions.

internetAccess (optional): Sets it true to enable internet access for your functions.

Except for the service name, other attributes can be updated later.

Create and update service using the command line tool

In shell mode of the command line tool, you can run mks/ups to create or update service, and run info to view the service attributes.

In the following example, the service "my-service" is created, and the description and logConfig attributes are updated. Then, Function Compute is authorized to access the log resources.

Create the service.

```
mks my-service
```

Note: You can provide the service description and configuration together when creating the service. This service is created with name only to demonstrate the subsequent update operation.

Create a Log Service project and LogStore.

```
mkl -p fc-log-project -s fc-log-store
```

Update the service.

```
ups my-service -d "this is my service" -p fc-log-project -l fc-log-store
```

View the service.

```
info my-service
```

View a service

You can use the console or command line tool to list all services or view an individual service.

View a service using the command line tool

Enter the shell mode.

Run ls to list all services.

Run info to view the details of the specified service.

Animation example

```
Scorpion-MacBook:demo scorpion$
```

Delete a service

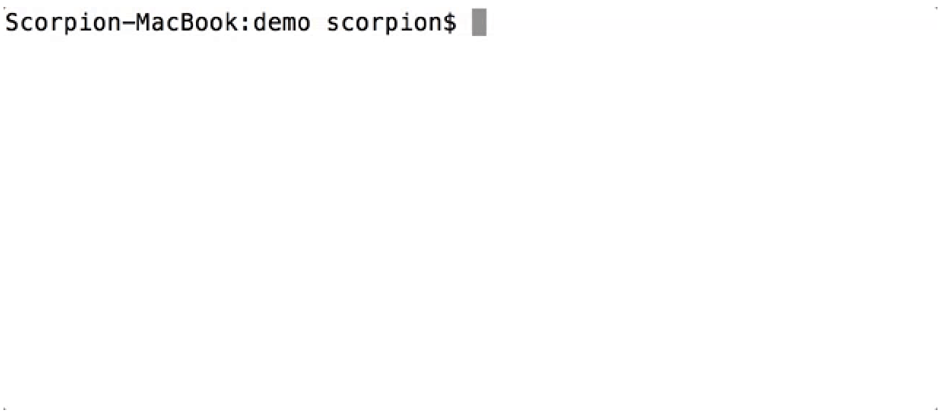
You can use the console or command line tool to delete a service. The service can be deleted only when it does not contain any function.

Delete a service using the command line tool

Run fcli shell to enter the interactive mode.

Run rm to delete the specified service.

Example



```
Scorpion-MacBook:demo scorpion$
```

Function management

Create a function

Function is the scheduling and operation unit. All functions belong to the same service share some common attributes, such as authorization and logging configuration.

Function attributes

When creating a function, specify the following attributes:

- functionName (required): Name of the function. It must be unique within the service with the following restrictions:
  - It must consist of English letters (a–z) (A–Z), numbers (0–9), underscores (_), and hyphens (-).
  - It must start with an English letter (a–z) (A–Z) or underscore (_).
  - It is case-sensitive.
  - It must contain 1–128 characters.
- runtime (required): Type of the function runtime.
- code (required): Code package. It must be a ZIP file, which can be stored in OSS or directly uploaded.
- handler (required): The entry point for Function Compute to run your functions.
- description (optional): Description of the function.
- timeout (optional): Maximum execution time, in second.
- MemorySize (optional): Memory resource required for function running, in MB. The memory size ranges from 128 to 3072 MB with 64 MB increments.

All attributes except for the function name can be modified later.

Supported function runtimes

Runtime	Description
nodejs6	Node.js 6.10.3
nodejs8	Node.js 8.9.0
python2.7	Python 2.7
python3	Python 3.6
java8	Java 8

Create and update a function using the CLI

In shell mode, you can run mkf/upf to create or update a function, and run info to view the function attributes. Follow these steps to create and update the world function in the hello service:

1. Create the hello service: mks hello.
2. Create a simple local function file index.js and store it in the code folder.

```
'use strict';
console.log('loading function');
module.exports.handler = function(event, context, callback) {
console.log('Receive event:', event.toString());
callback(null, event);
};
```

3. Create the world function.

```
mkf hello/world -t nodejs6 -h index.handler -d code
```

4. Update the description of the world function: upf hello/world --description "this is world function".
5. View the attributes of the world function: info hello/world.

Invoking a function

Invocation types

Function Compute supports synchronous and asynchronous invocation of a function.

- Synchronous invocation: The event is processed by the function, and then the result is returned.
- Asynchronous invocation: The event is queued in Message Queue, and then no result is returned. Function Compute will process event based on Message Queue At-Least-Once delivery guarantee.
- Different limits apply for synchronous and asynchronous invocation. For more information, see Limits.

You can manually invoke your functions using the console or command line tool. For more information, see Relevant examples. You can also invoke your function calling the REST API. For more information, see API specification. SDKs of different languages are provided to further simplify your operations. The following is an example of calling the Java SDK:

```
public class FcSample {
private static final String CODE_DIR = "/tmp/fc_code";
private static final String REGION = "cn-shanghai";
private static final String SERVICE_NAME = "test_service";
private static final String FUNCTION_NAME = "test_function";

public static void main(final String[] args) throws IOException {
String accessKey = System.getenv("ACCESS_KEY");
String accessSecretKey = System.getenv("SECRET_KEY");
String accountId = System.getenv("ACCOUNT_ID");
String role = System.getenv("ROLE");

// Initialize FC client
FunctionComputeClient fcClient = new FunctionComputeClient(REGION, accountId, accessKey, accessSecretKey);

// Create a service
CreateServiceRequest csReq = new CreateServiceRequest();
csReq.setServiceName(SERVICE_NAME);
csReq.setDescription("FC test service");
csReq.setRole(role);
CreateServiceResponse csResp = fcClient.createService(csReq);
System.out.println("Created service, request ID " + csResp.getRequestId());

// Create a function
CreateFunctionRequest cfReq = new CreateFunctionRequest(SERVICE_NAME);
cfReq.setFunctionName(FUNCTION_NAME);
cfReq.setDescription("Function for test");
cfReq.setMemorySize(128);
cfReq.setHandler("hello_world.handler");
cfReq.setRuntime("nodejs6");
Code code = new Code().setDir(CODE_DIR);
cfReq.setCode(code);
cfReq.setTimeout(10);
CreateFunctionResponse cfResp = fcClient.createFunction(cfReq);
System.out.println("Created function, request ID " + cfResp.getRequestId());

// Invoke the function with a string as function event parameter, Sync mode
InvokeFunctionRequest invkReq = new InvokeFunctionRequest(SERVICE_NAME, FUNCTION_NAME);
String payload = "Hello FunctionCompute!"
invkReq.setPayload(payload.getBytes())
InvokeFunctionResponse invkResp = fcClient.invokeFunction(invkReq);
System.out.println(new String(invkResp.getContent()));

// Invoke the function, Async mode
invkReq.setInvocationType(Const.INVOCATION_TYPE_ASYNC);
invkResp = fcClient.invokeFunction(invkReq);
if (HttpURLConnection.HTTP_ACCEPTED == invkResp.getStatus()) {
System.out.println("Async invocation has been queued for execution, request ID: " + invkResp.getRequestId());
} else {
System.out.println("Async invocation was not accepted");
}

// Delete the function
DeleteFunctionRequest dfReq = new DeleteFunctionRequest(SERVICE_NAME, FUNCTION_NAME);
DeleteFunctionResponse dfResp = fcClient.deleteFunction(dfReq);
System.out.println("Deleted function, request ID " + dfResp.getRequestId());

// Delete the service
DeleteServiceRequest dsReq = new DeleteServiceRequest(SERVICE_NAME);
DeleteServiceResponse dsResp = fcClient.deleteService(dsReq);
System.out.println("Deleted service, request ID " + dsResp.getRequestId());
}
}
```

Concurrent Execution

Concurrent Executions is the total concurrent function executions within a given time period. You can use the following formula to estimate the concurrent function execution number.

Request rate x Function execution time

The request rate is the function execution rate, with the unit of “Requests per second” or “Events per second”. The unit of Function execution time is “Second”. For example, given an OSS events processing function, assuming that the function execution time is 3s, and OSS generates 10 events per second. Therefore, your function has 30 concurrent executions.

Concurrent Execution Limits

In some cases, function execution may be out of control due to incorrect settings. For example, you have set an OSS trigger. When image files are uploaded to the foo bucket of OSS, the relevant function is called. This function adjusts the source image to three images with different resolutions and incorrectly writes the result back to the foo bucket. As a result, a new function is called, resulting in an infinite loop. To prevent financial loss caused by infinite function calls, Function Compute sets the maximum number of concurrent function executions (100 by default) for each account. You can check the throttles metrics of a function in CloudMonitor. To adjust the limit, open a ticket.

Throttling errors handling

Throttling errors are handled differently based on the different invocation type.

Synchronous invocation: It the function is invoked synchronously and is throttled, Function Compute retuens a 429 error and the invoking service is responsible for retries. For example, if you use API Gateway to invoke a function, please make sure that the response error of Function Compute is mapped to the error code of API Gateway. If you directly invoke the function through the SDK or CLI, you can decide whether to retry based on your requirement.

Asynchronous invocation: If your function is invoded asynchronously and is throttled, Function Compute automatically retries the throttled event for up to 5 hours, with exponential backoff. Asynchronous events are queued before they are used to invoke the function.

Function entry definition

This section describes the runtime envrionments currently supported by Function Compute and how to write basic functions. Function Compute supports the following languages:

- Nodejs
- python

- java

Nodejs

Function Compute currently supports the following Node.js runtime environments:

- Nodejs6 (runtime = nodejs6)
- Nodejs8 (runtime = nodejs8)

Function handler: index.handler

The format of handler is “[File name].[Function name]”. For example, if handler specified during function creation is index.handler, Function Compute automatically loads the handler function defined in index.js.

When Nodejs runtime is used, a Node.js function must be defined. A simple function is defined as follows:

```
exports.handler = function(event, context, callback) {
callback(null, 'hello world');
};
```

1. Function name
    - exports.handler must correspond to the “Handler” of the created function. For example, if the Handler is set to index.handler upon the function creation, Function Compute automatically loads the handler function defined in index.js.
2. event parameter
    - The event parameter is the data passed to function invocation call. Function Compute doesn’t do any transformation but just passes through the value to user function. The parameter type is Buffer.
3. context parameter
    - The context parameter contains the operation information of the function (such as the request ID and temporary Accesskeys). The parameter is of the object type. The Node.js guide section describes the context structure and usage.
4. callback parameter
    - The callback parameter returns the result of a called function. Its signature is function(err, data). Same as callback that is frequently used in Node.js, its first parameter is error and second parameter is data. If err is not blank when the function is called, the function returns HandledInvocationError; otherwise, the function returns the data information. If the data type is Buffer, the data is directly returned. If the data type is object, the data is converted into a JSON string and then returned. If the data is of another type, it is converted into a string and then returned.

Python

Function Compute currently supports the following Python running environments:

- Python 2.7 (runtime = python2.7)
- Python 3.6 (runtime = python3)

Function handler: main.my_handler

The format of handler is “[File name].[Function name]”. For example, if handler specified during function creation is main.my_handler, Function Compute automatically loads the my_handler function defined in main.py.

When Python runtime is used, a Python function must be defined. A simple function is defined as follows:

```
def my_handler(event, context):
return 'hello world'
```

1. Function name
   - my_handler must correspond to the “Handler” field of the created function. For example, if the Handler is set to main.my_handler upon the function creation, Function Compute automatically loads the my_handler function defined in main.py.
2. event parameter
   - The event parameter is the data passed to function invocation call. Function Compute doesn’t do any transformation but just passes through the value to user function. The parameter type in Python2.7 is str and in Python3 is bytes.
3. context parameter
   - The context parameter contains the operation information of the function (such as the request ID and temporary Accesskeys). The parameter is of the FCContext type. The Python guide section describes the context structure and usage.
4. Return value
   - The return value of a function is returned to you as the result of invoking the function. It can be of any type. For a simple type, Function Compute converts the result to a string before returning it. For a complicated type, Function Compute converts the result to a JSON string before returning it.

Java

Function Compute currently supports the following Java runtime environment:

- OpenJDK 1.8.0 (runtime = java8)

Function handler: example.HelloFC::handleRequest

The format of handler is {package}.{class}::{method}. For example, if the package name is example and the class name is HelloFC, the handler specified during function creation is example.HelloFC::handleRequest.

When Java runtime is used, a class must be defined and a pre-defined interface must be implemented. A simple function is defined as follows:

```
package example;

import com.aliyun.fc.runtime.Context;
import com.aliyun.fc.runtime.StreamRequestHandler;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class HelloFC implements StreamRequestHandler {

@Override
public void handleRequest(
InputStream inputStream, OutputStream outputStream, Context context) throws IOException {

outputStream.write(new String("hello world").getBytes());
}
}
```

1. Package name/Class name
   - A package and a class can have custom names, but their names must correspond to the “handler” field of the created function. In the previous example, the package name is “example” and the class name is “HelloFC”. Therefore, the handler specified during function creation is example.HelloFC::handleRequest. The format of “handler” is {package}.{class}::{method}. For now, the method name has to be handleRequest.
2. Implemented interface
   - Your code must implement the pre-defined interface. In the previous example, StreamRequestHandler is implemented, wherein inputStream is the data imported when the function is called, and outputStream is used to return the function execution result. For more information about the function interfaces, see Function interfaces.
3. context parameter
   - The context parameter contains the operation information of the function (such as the request ID and temporary Accesskeys). The parameter is of the com.aliyun.fc.runtime.Context type. The Java guide section describes the context structure and usage.
4. Return value
   - The function that implements the StreamRequestHandler interface returns execution results by using the outputStream parameter.

The dependency of the com.aliyun.fc.runtime package can be referenced in the following pom.xml:

```
<dependency>
```

```
<groupId>com.aliyun.fc.runtime</groupId>
<artifactId>fc-java-core</artifactId>
<version>1.0.0</version>
</dependency>
```

Before a function is created, the fc-java-core and other dependencies must be compiled into a JAR package. After the JAR package is created, use the fcli or console to create function with the package. The following uses fcli as an example:

```


rockuw-MBP:hello-java (master) $ ls -lrt
total 16
-rw-r--r-- 1 rockuw staff 7690 Aug 31 19:45 hellofc.jar

>>> mkf hello-java -t java8 -h example.HelloFC::handleRequest -d ./functions/hello-java
>>> invk hello-java
hello world
>>>
```

Trigger management

Create a trigger

In the event-driven computing model, the event sources produce events, the functions process events, and triggers manage different event sources in a centralized and unified manner. On the event source side, when an event matches the rules defined by a trigger occurs, the event source invokes the corresponding function. Currently, Function Compute supports OSS, API Gateway, Log Service and Time trigger. More event sources will be available in the future.

Note that as different triggers can be associated with the same source, such as the OSS bucket, the event source may restrict the number of triggers that can be associated. An OSS bucket can be associated with a maximum of 10 triggers.

Trigger attributes

When creating a trigger, specify the following attributes:

- triggerName (required): Name of the trigger. It must be unique in the current function and meets the following requirements:
    - It must consist of English letters (a–z) (A–Z), numbers (0–9), underscores (_), and

hyphens (-).
    - It must start with an English letter (a–z) (A–Z) or underscore (_).
    - It is case-sensitive.
    - It must contain 1–128 characters.
- triggerType (required): Type of the trigger.
- sourceArn (optional): Event source resource address, for example, OSS bucket (acs:oss:cn-shanghai:12345:mybucket). Note that this resource must be in the same region as Function Compute. The value format is acs:oss:$region:$account-id:$bucket-name.
- invocationRole (optional): Role used by the event source to call the function, for example, acs:ram::12345:role/myrole. The role grants event source the permission to invoke functions. The value format is acs:ram::$account-id:role/$role-name.
- triggerConfig (required): Configuration of the trigger. The configuration varies with the trigger type. For example, the OSS trigger configuration defines that when some events such as object upload or deletion occur, OSS calls the function to which the trigger belongs.

Only invocationRole and triggerConfig can be modified after the trigger is created.

Supported trigger type and configuration

OSS trigger configuration

Specify the following information:

- events: Events that trigger the execution of a function on OSS, which can be one or more of the following options:
    - oss:ObjectCreated:*
    - oss:ObjectCreated:PutObject
    - oss:ObjectCreated:PutSymlink
    - oss:ObjectCreated:PostObject
    - oss:ObjectCreated:CopyObject
    - oss:ObjectCreated:InitiateMultipartUpload
    - oss:ObjectCreated:UploadPart
    - oss:ObjectCreated:UploadPartCopy
    - oss:ObjectCreated:CompleteMultipartUpload
    - oss:ObjectCreated:AppendObject
    - oss:ObjectRemoved:DeleteObject
    - oss:ObjectRemoved:DeleteObjects
    - oss:ObjectRemoved:AbortMultipartUpload
- filter: OSS object filter. Only objects that meet the filter criteria can trigger function execution.
    - key: Currently, the filter only supports filtering based on the object key.
        - prefix: Prefix match.
        - suffix: Suffix match.

Create and update a trigger using the command line tool

In shell mode, you can run mkt/upt to create or update a trigger, and run info to view the trigger attributes. For example, assuming that the world function is created in the hello service. Follow these steps:

> Note: Due to the diversity of the trigger configuration, the trigger configuration is specified in a file.

sample_oss_trigger_config.yaml

```
triggerConfig:
events:
- oss:ObjectCreated:PostObject
- oss:ObjectCreated:PutObject
filter:
key:
prefix: source/
suffix: .png
```

1. Run fcli shell to enter the interactive mode.
2. Create a trigger.

   ```
   mkt hello/world/mytrigger -t oss -r acs:ram::12345:role/myrole -s acs:oss:cn-shanghai:12345:mybucket -c sample_oss_trigger_config.yaml
   ```

   > NOTE: The invocationRole and sourceArn must be valid. If you do not have any available role, you can use the command line to create and call a role.
   >
   > ```
   > mkir fc-invoke-function
   > mkrp fc-invoke-all -a '"fc:InvokeFunction"' -r '"*"'
   > attach -p /ram/policies/fc-invoke-all -r /ram/roles/fc-invoke-function
   > ```

3. Update the trigger.

   ```
   upt hello/world/mytrigger -r acs:ram::12345:role/myrole2 -c sample_oss_trigger_config2.yaml
   ```

Configure triggers and events

Different types of triggers have different configurations. This topic provides trigger configuration definitions and samples. Because the event that passed to a function from event sources differs by trigger type, this topic also describes the formats of event objects.

Currently, the following two types of triggers are supported:

- Triggers created in event source services: Table Store, API Gateway, Datahub, and IoT
- Triggers created in Function Compute: HTTP, OSS, Log Store, Timer and CDN Events

For triggers created in Function Compute, you can use the FCLI tool to enter Shell mode and run the mkt command to create a trigger. The parameters of this command are as follows:

```
>>> mkt --help
--etag string trigger etag for update
--help
-r, --invocation-role string invocation role
-s, --source-arn string event source arn
-c, --trigger-config string trigger config file
-t, --type string trigger type, support oss now (default "oss")
```

Note: The --trigger-config parameter configuration files for different triggers have different formats.

HTTP triggers

HTTP triggers differ from other triggers in that function signatures are request and response objects, rather than event objects. Therefore, HTTP triggers do not use the event format.

Configure a trigger

Trigger example: httpTrigger.yml

```
triggerConfig:
authType: anonymous
methods: ["GET", "POST"]
```

Trigger parameter descriptions:

- authType is the authorization mode. Optional values:
  - anonymous: No authorization required, allows access request from any user.
  - function: HTTP request headers must include a signature and time stamp.
- methods is the request method. Optional values:
  - GET: HTTP GET method
  - POST: HTTP POST method

- HEAD: HTTP HEAD method
- PUT: HTTP PUT method
- DELETE: HTTP DELETE method

OSS triggers

Configure a trigger

Trigger example: ossTrigger.yml

```
triggerConfig:
events:
- oss:ObjectCreated:PostObject
- oss:ObjectCreated:PutObject
filter:
key:
prefix: source/
suffix: .png
```

Trigger parameter descriptions:

- events is the function execution event triggered for OSS. Optional values:
    - oss:ObjectCreated:*
    - oss:ObjectCreated:PutObject
    - oss:ObjectCreated:PutSymlink
    - oss:ObjectCreated:PostObject
    - oss:ObjectCreated:CopyObject
    - oss:ObjectCreated:InitiateMultipartUpload
    - oss:ObjectCreated:UploadPart
    - oss:ObjectCreated:UploadPartCopy
    - oss:ObjectCreated:CompleteMultipartUpload
    - oss:ObjectCreated:AppendObject
    - oss:ObjectRemoved:DeleteObject
    - oss:ObjectRemoved:DeleteObjects
    - oss:ObjectRemoved:AbortMultipartUpload
- filter is the OSS object filter parameter. Only OSS objects that meet the filter conditions can trigger the function execution. The conditions include the following attributes:
    - key: The filter can filter objects by key, which includes the following attributes:
        - prefix: Must match the prefix
        - suffix: Must match the suffix

Event format

```
{
```

```
"events": [
{
"eventName": "ObjectCreated:PutObject",
"eventSource": "acs:oss",
"eventTime": "2017-04-21T12:46:37.000Z",
"eventVersion": "1.0",
"oss": {
"bucket": {
"arn": "acs:oss:cn-shanghai:1237050315505689:bucketname",
"name": "bucketname",
"ownerIdentity": "1237050315505689",
"virtualBucket": ""
},
"object": {
"deltaSize": 122539,
"eTag": "688A7BF4F233DC9C88A80BF985AB7329",
"key": "image/a.jpg",
"size": 122539
},
"ossSchemaVersion": "1.0",
"ruleId": "9adac8e253828f4f7c0466d941fa3db81161e853"
},
"region": "cn-shanghai",
"requestParameters": {
"sourceIPAddress": "140.205.128.221"
},
"responseElements": {
"requestId": "58F9FF2D3DF792092E12044C"
},
"userIdentity": {
"principalId": "262561392693583141"
}
}
]
}
```

Log Service triggers

Configure a trigger

Trigger example: slsTrigger.yml

```
triggerConfig:
sourceConfig:
logstore: "etl-log"
jobConfig:
maxRetryTime: 3
triggerInterval: 60
functionParameter:
a: "b"
c: "d"
logConfig:
project: "ali-fc-test"
```

```
logstore: "test-store"
enable: true
```

Trigger parameter descriptions:

- sourceConfig is the data source configuration parameter, which includes the following attributes:
    - {project}: The name of the Log Service project. For more information, see Manage a project.
    - logstore: The name of the Log Service Logstore that is used as the data source. The Log Service regularly checks the configured Logstore and sends the data to Function Compute for custom processing. After this parameter is created, it cannot be modified. For more information, see Manage a Logstore.
- jobConfig is the task configuration parameter, which includes the following attributes:
    - triggerInterval: The interval that Log Service triggers function execution. Value range: [3, 600]. This parameter defines the interval for Log Service to trigger the function invocation. For example, every 60 seconds, Log Service reads the locations of unprocessed data and uses them to invoke the function which then reads the data based on locations and does further processing. For shard with large traffic (1 MB/s or higher), we recommend that you reduce the interval so Log Service can trigger functions more frequently. For more information, see Manage a shard.
    - maxRetryTime: Value range: [0, 100]. This defines the number of times Log Service will retry if it fails to invoke function due to errors such as insufficient permissions, network failure, or invocation exceptions. If Log Service still fails after all the retries, it will wait for the next schedule and invoke function again.
- functionParameter: Log Service passes this value to function as part of the function event. A function can customize its logic based on this configuration. The configuration must be a string in JSON Object format. The default value is null ({}).
- logConfig specifies the log configuration, which includes the following attributes. Note that Log Service stores trigger related logs to the specified logstore.
    - {project}: The name of the Log Service project. For more information, see Manage a project.
    - logstore: The name of the Logstore in which logs are stored.
- enable indicates whether the trigger is enabled or not. Optional values: true | false.

Event format

```
{
"parameter":{
"a":"b",
"c":"d"
},
"source":{
"endpoint":"http://cn-shanghai-intranet.log.aliyuncs.com",
"projectName":"vangie-fc-test",
"logstoreName":"fc-test",
"shardId":0,
"beginCursor":"MTUyMzI2NzI5NDY1NjI4MzgzNg==",
"endCursor":"MTUyMzI2NzI5NDY1NjI4MzgzNw=="
},
"jobName":"05c79f637c6b46eaa85911cae032cf47551af7bb",
"taskId":"d22697c0-2a41-4d35-b27c-dccec8856768",
"cursorTime":1523323454
}
```

Timer triggers

Configure a trigger

Trigger example: timerTrigger.yml

```
triggerConfig:
payload: "aaaaa"
cronExpression: "0 1/1000 * * * *"
enable: true
```

Trigger parameter descriptions:

- payload is the trigger message. This parameter supports any text format. Each time a function is triggered, the payload is passed to the function as part of the event.
- cronExpression is a Cron expression. For more information, see Cron expressions.
- enable indicates whether the trigger is enabled or not. Optional values: true | false.

Event format

```
{
"triggerTime":"2018-04-10T01:31:00Z",
"triggerName":"t1",
"payload":"abcde"
}
```

CDN events trigger

Configure a trigger

Trigger example：cdn_events_trigger.yml

```
triggerConfig:
eventName:   “LogFileCreated”
eventVersion:   “1.0.0”
```

```
notes:   “cdn events trigger test”
filter:
domain: {  “www.taobao.com”  ,”  www.tmall.com”  }
```

Trigger parameter descriptions :

- eventName is CDN event which invoke the function execution
- eventVersionis CDN event version which invoke the function execution
- notesDescriptions
- filter Filters

eventName, eventVersion and filter key supported :

Event Name	Event Version	Filter Key	Description
CachedObjectsRefreshed	1.0.0	domain	See RefreshObjectCaches API for details.
CachedObjectsBlocked	1.0.0	domain	CDN resource blocked
CachedObjectsPushed	1.0.0	domain	See PushObjectCache API for details.
LogFileCreated	1.0.0	domain	See DescribeCdnDomainLogs API for details.

Filter need to contain at least one < key, values > pair, see schema below :

```
    filter:
key1: {value a , value b}
key2: {value c , value d}
```

Event format

CachedObjectsRefreshed, CachedObjectsPushed and CachedObjectsBlocked events schema :

```
{
"events": [
{
"eventName": "CachedObjectsRefreshed",
"eventVersion": "1.0.0",
"eventSource": "cdn",
"region": "cn-hangzhou",
"eventTime": "2018-03-16T14:19:55+08:00",
"traceId": "cf89e5a8-7d59-4bb5-a33e-4c3d08e25acf",
"resource": {
"domain": "example.com"
},
"eventParameter": {
"objectPath": [
"/2018/03/16/13/33b430c57e7.mp4",
"/2018/03/16/14/4ff6b9bd54d.mp4"
],
"createTime": 1521180769,
"domain": "example.com",
"completeTime": 1521180777,
"objectType": "File",
"taskId": 2089687230
},
"userIdentity": {
"aliUid": "1xxxxxxxxxx"
}
}
]
}
```

LogFileCreated event schema：

```
{
"events": [
{
"eventName": "LogFileCreated",
"eventSource": "cdn",
"region": "cn-hangzhou",
"eventVersion": "1.0.0",
"eventTime": "2018-06-14T15:31:49+08:00",
"userIdentity": {
"aliUid": "1xxxxxxxxxxxxx"
},
"resource": {
"domain": "example.com"
},
"eventParameter": {
"domain": "example.com",
"endTime": 1528959900,
"fileSize": 1788115,
"filePath": "http://cdnlog.cn-hangzhou.oss.aliyun-
inc.com/www.aliyun.com/2017_12_27/www.aliyun.com_2017_12_27_0800_0900.gz?OSSAccessKeyId=xxxx&Expires=
xxxx&Signature=xxxx",
"startTime": 1528959600
},
"traceId": "c6459282-6a4d-4413-894c-e4ea39686738"
}
]
}
```

More details about CDN events trigger.

Table Store triggers

Event format

Table Store triggers use CBOR format to encode incremental data and construct a Function Compute event. The incremental data event format is as follows:

```
{
"Version": "string",
"Records": [
{
"Type": "string",
"Info": {
"Timestamp": int64
},
"PrimaryKey": [
{
"ColumnName": "string",
"Value": formated_value
}
],
"Columns": [
{
"Type": "string",
"ColumnName": "string",
"Value": formated_value,
"Timestamp": int64
}
]
}
]
}
```

Parameter description:

- Version is the payload version number. Value: Sync-v1.
- Records indicates the set of incremental data rows in the table, which includes the following attributes:
    - Type: The data row type, including PutRow, UpdateRow, and DeleteRow.
    - Info: The basic information of the data row, which includes the following attributes:
        - Timestamp: The last time this row was modified, in UTC time.
        - PrimaryKey: The primary key array, which includes the following attributes:
            - ColumnName: The name of the primary key column.
            - Value: The content of the primary key column, supports integer, string, and blob.
        - Columns: The column attribute array, which includes the following attributes:
            - Type: The column attribute, includes Put, DeleteOneVersion, and DeleteAllVersions.
            - ColumnName: The name of the column.
            - Value: The column value, supports data types of integer, Boolean,

double, string, and blob.
- Timestamp: The last time this column was modified, in UTC time.

Event example:

```
{
"Version": "Sync-v1",
"Records": [
{
"Type": "PutRow",
"Info": {
"Timestamp": 1506416585740836
},
"PrimaryKey": [
{
"ColumnName": "pk_0",
"Value": 1506416585881590900
},
{
"ColumnName": "pk_1",
"Value": "2017-09-26 17:03:05.8815909 +0800 CST"
},
{
"ColumnName": "pk_2",
"Value": 1506416585741000
}
],
"Columns": [
{
"Type": "Put",
"ColumnName": "attr_0",
"Value": "hello_table_store",
"Timestamp": 1506416585741
},
{
"Type": "Put",
"ColumnName": "attr_1",
"Value": 1506416585881590900,
"Timestamp": 1506416585741
}
]
}
]
}
```

API Gateway triggers

Event format

Input format

When Function Compute is used as an API Gateway backend, API Gateway passes HTTP request to

Function Compute as an event which is in JSON format. Functions can obtain and process the request according to the following structure:

```
{
"path":"api request path",
"httpMethod":"request method name",
"headers":{all headers,including system headers},
"queryParameters":{query parameters},
"pathParameters":{path parameters},
"body":"string of request payload",
"isBase64Encoded":"true|false, indicate if the body is Base64-encode"
}
```

Parameter description:

- isBase64Encoded=true indicates that API Gateway uses Base64 to encode the body content transmitted to Function Compute. Then, Function Compute uses Base64 to decode the body content.
- isBase64Encoded=false indicates that API Gateway does not use Base64 to encode the body content transmitted to Function Compute.

Event example:

```
{
"body":"",
"headers":{
"X-Ca-Api-Gateway":"BDB46B3A-71A7-447B-B20B-28C594426407",
"X-Forwarded-For":"106.11.231.99"
},
"httpMethod":"GET",
"isBase64Encoded":false,
"path":"/fc",
"pathParameters":{

},
"queryParameters":{

}
}
```

Output format

To respond to an API Gateway request, a function should return the result in the following JSON format.

```
{
"isBase64Encoded":true|false,
"statusCode":httpStatusCode,
"headers":{response headers},
"body":"..."
}
```

Parameter description:

- When the body content is binary data, you must use Base64 to encode the body content in Function Compute and set isBase64Encoded=true. API Gateway uses Base64 to decode body content with the attribute isBase64Encoded=true and then transmits it to the client.
- If the body does not need to be encoded by using Base64 format, you can set isBase64Encoded to false.
- In the Node.js version of Function Compute, you must set callback based on the specific situation.
  - To return a request successful message: callback{null,{ “statusCode” :200,” body” :” …” }}
  - To return a request exception message: callback{new Error(‘internal server error’),null}
  - To return a client error: callback{null,{ “statusCode” :400,” body” :” param error” }}
- If the format of the result returned by Function Compute does not conform to these requirements, API Gateway returns 503 Service Unavailable to the client.

DataHub triggers

Event format

```
{
"eventSource": "acs:datahub",
"eventName": "acs:datahub:putRecord",
"eventSourceARN": "/projects/test_project_name/topics/test_topic_name",
"region": "cn-hangzhou",
"records": [
{
"eventId": "0:12345",
"systemTime": 1463000123000,
"data": "[\"col1's value\",\"col2's value\"]"
},
{
"eventId": "0:12346",
"systemTime": 1463000156000,
"data": "[\"col1's value\",\"col2's value\"]"
}
]
}
```

Parameter description:

- eventSource is the event source. Value: acs:datahub.
- eventName is the event name. Value: acs:datahub:putRecord.

- eventSourceARN is the event source ID, which includes the DataHub project and topic names. For example, /projects/test_project_name/topics/test_topic_name.
- region is the region of the event source’ s DataHub, such as cn-hangzhou. For more information, see Regions and zones.
- records is a list of records included in the event, which includes the following key values:
    - eventId: The record ID, with the format: shardId:SequenceNumber.
    - systemTime: The time stamp from when this event was stored in DataHub.
    - data: The data content of the event. For tuple-type topics, the data type of this field is list. Here, each list-type element corresponds to each field value of each topic for the string-type data. For blob-type topics, the data type of this field is string.

IoT triggers

Event format

The event content IoT Hub sends to Function Compute is non-encapsulated IoT message content. For example, you can use the following Java example to push messages to IoT topics:

```
PubRequest request = new PubRequest();
request.setProductKey("VHo5FRjudkZ");
request.setMessageContent(Base64.getEncoder().encodeToString("{\"hello\":\"world\"}".getBytes()));
request.setTopicFullName("/VHo5FRjudkZ/deviceName/update");
request.setQos(0);
PubResponse response = client.getAcsResponse(request);
System.out.println(response.getSuccess());
System.out.println(response.getErrorMessage());
```

The event received in Function Compute is:

```
{
"hello": "world"
}
```

References

- What is OSS?
- What is Log Service?
- What is Table Store?
- What is API Gateway?
- What is DataHub?
- What is Alibaba Cloud IoT?

Trigger

HTTP trigger

HTTP triggers are a common type of trigger used in Function Compute to call functions when an HTTP request is sent. With HTTP triggers, you can construct web alike services. HTTP triggers support HEAD, POST, PUT, GET, and DELETE method triggers.

Compared to API Gateway triggers, which are a bit difficult for new users, HTTP triggers offer the following advantages:

The easier learning and debugging processes help you to implement web services and APIs by using Function Compute.

You can use the HTTP testing tools you are familiar with to verify the functionality and performance of Function Compute.

Requests are processed in less stages. HTTP triggers support more efficient request and response formats and are not encoded or decoded into JSON strings. The performance is improved.

You can connect to other services that support WebHook callbacks, such as CDN back-to-source and MNS.

This topic discusses the following aspects of HTTP triggers:

- HTTP trigger configuration
- HTTP trigger interface format
- HTTP trigger examples
- Problem diagnosis

Limits

- Once an HTTP trigger is set for a function, the trigger type cannot be changed for the function.
- A function must have only one HTTP trigger.

Configure an HTTP trigger

Assemble the HTTP trigger URL

You can send an HTTP request to the following address to call the function:

```
<account_id>.<region>.fc.aliyuncs.com/<version>/proxy/<serviceName>/<functionName>/[action?queries]
```

The action, queries, request header, body, and other information in the URL is all contained in the request parameter of Function Compute handler.

HTTP trigger Config

HTTP trigger Config format:

```
{
"authType" : "anonymous",
"methods" : ["GET", "POST"]
}
```

The HTTP trigger Config has two main fields:

- authType: The authentication type. Optional values:
  - anonymous: The server does not require identity authentication and supports anonymous access. The system is of low security level.
    Example:

    ```
    curl "<account-id>.<region>.fc.aliyuncs.com/2016-08-15/proxy/serviceName/functionName/action?hello=world"
    ```

  - function: The server requires identity authentication and does not support anonymous access. The system is of high security level. Requests must pass signature authentication and HTTP request headers must contain Authorization and Date. The Date is in UTC format and used to calculate the signature. The server uses the Date value to calculate the signature and compares it with the transmitted Authorization value. The request passes the authentication only if the signatures match and the difference between the current time and the Date value is within 15 minutes.
- methods: The access methods supported by the HTTP trigger:
  - HEAD: HTTP HEAD method
  - GET: HTTP GET method
  - POST: HTTP POST method
  - PUT: HTTP PUT method

- DELETE: HTTP DELETE method

HTTP trigger functions

The interfaces for invoking functions with HTTP triggers are different from those for invoking the original functions in Function Compute. Currently, only Node.js runtime is supported. Python runtime and Java runtime are supported later.

HTTP trigger interface format

- **Nodejs runtime**:

```
function(request, response, context)
```

Request structure

- headers: Map type, stores the key-value pairs from the HTTP client.
- path: String type, the HTTP URL.
- queries: Map type, stores the key-value pairs from the HTTP URL’ s query section. The values can be strings or arrays.
- method: String type, the HTTP method.
- clientIP: String type, the IP address of the client.
- url: String type, the URL of the request.

Get HTTP body: Requests in HTTP triggers are compatible with HTTP requests. HTTP triggers use the bodies in HTTP requests directly. No additional field is provided for bodies.

```
  // Example
var getRawBody = require('raw-body')
getRawBody(request, function(err, data){
var body = data
})
```

Note: The following fields in the Headers key are ignored. Function Compute contains these fields and does not support user-defined settings. Keys that start with x-fc- are also ignored.

- accept-encoding
- connection
- keep-alive
- proxy-authorization
- te
- trailer
- transfer-encoding

Response methods

- response.setStatusCode(statusCode): Sets the status code.
  - param statusCode : (required, type integer)
- response.setHeader(headerKey, headerValue): Sets the header.
  - param headerKey : (required, type string)
  - param headerValue : (required, type string)
- response.deleteHeader(headerKey): Deletes the header.
  - param headerKey: (required, type string)
- response.send(body): Sends the body.
  - param body: (required, typeBuffer or a string or a stream.Readable)

Note: The following fields in the Headers key are ignored. Function Compute contains these fields and does not support user-defined settings. Keys that start with x-fc- are also ignored.

- connection
- content-length
- content-encoding
- date
- keep-alive
- proxy-authenticate
- server
- trailer
- transfer-encoding
- upgrade

Limits for requests

If the following limits are exceeded, the system throws a 400 status code and InvalidArgument error code.

Headers size: The total size of all keys and values in the headers cannot exceed 4 KB.

Path size: In each query parameter, the path size cannot exceed 4 KB.

Body size: The HTTP body size cannot exceed 6 MB.

Limits for responses

If the following limits are exceeded, the system throws a 502 status code and BadResponse error code.

Headers size: The total size of all keys and values in the headers cannot exceed 4 KB.

Body size: The HTTP body size cannot exceed 6 MB.

Example 1. Console operations

This example describes how to use the Function Compute console to set HTTP triggers. For more information about triggers and how to create them, see Use event source service and Create a trigger
.

Step 1. Select and set an HTTP trigger

You can select and set a trigger while or after creating a function.

Select and set a trigger while creating a function

Log on to the Function Compute console.

Select a region, such as China East 2 (Shanghai).

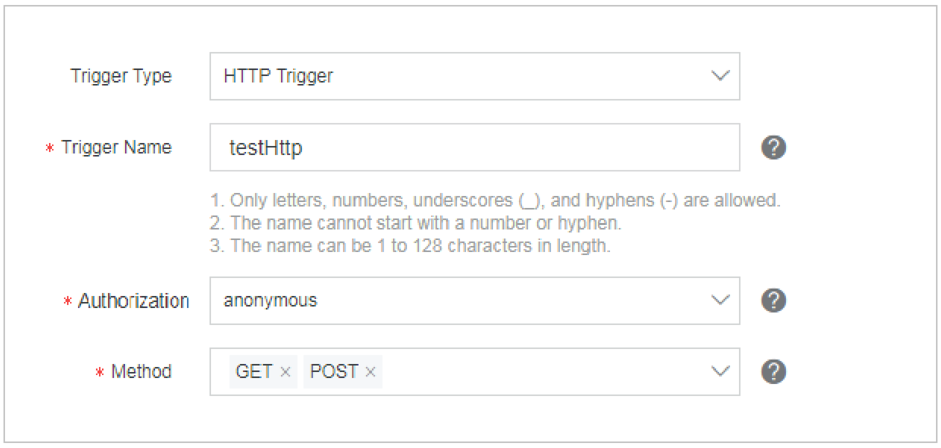
Select a service in the left-side navigation pane.

Click **Create Function** to go to the function creation page:

Click **Select All**, and select "nodejs8" from the drop-down menu.

Click **Select** in **Empty Function**.

Select **HTTP Trigger** and configure the parameters as follows. You may also select **No Trigger**. Then, click **Next**.



Trigger Type: HTTP Trigger

* Trigger Name: testHttp

1. Only letters, numbers, underscores (_), and hyphens (-) are allowed.
2. The name cannot start with a number or hyphen.
3. The name can be 1 to 128 characters in length.

* Authorization: anonymous

* Method: GET × POST ×

Configure the **Service Name**, **Function Name**, **Function Description**, and **Runtime** parameters.

In the **Code Configuration** section, select **In-line Edit** and paste the following sample code of the HTTP function for the Node.js runtime.

```
var getRawBody = require('raw-body')
module.exports.handler = function(request, response, context) {
// get requset header
var reqHeader = request.headers
var headerStr = ' '
for (var key in reqHeader) {
headerStr += key + ':' + reqHeader[key] + ' '
};

// get request info
var url = request.url
var path = request.path
var queries = request.queries
var queryStr = ''
for (var param in queries) {
queryStr += param + "=" + queries[param]+ ' '
};
var method = request.method
var clientIP = request.clientIP

// get request body
getRawBody(request, function(err, data){
var body = data
// you can deal with your own logic here

// set response
// var respBody = new Buffer('requestURI' + requestURI + ' path' + path + ' method' + method
+ ' clientIP' + clientIP)
var respBody = new Buffer('requestHeader:' + headerStr + '\n' + 'url: ' + url + '\n' + 'path: ' +
path + '\n' + 'queries: ' + queryStr + '\n' + 'method: ' + method + '\n' + 'clientIP: ' + clientIP
+'\n' + 'body: ' + body + '\n')
// var respBody = new Buffer( )
response.setStatusCode(200)
response.setHeader('content-type', 'application/json')

response.send(respBody)

})
};
```

vi. Click **Next**.
vii. (Optional) Configure permissions, and click **Next**.
viii. Check that all the information is error free, and click **Create**.

Select and set a trigger after creating a function

Log on to the Function Compute console.

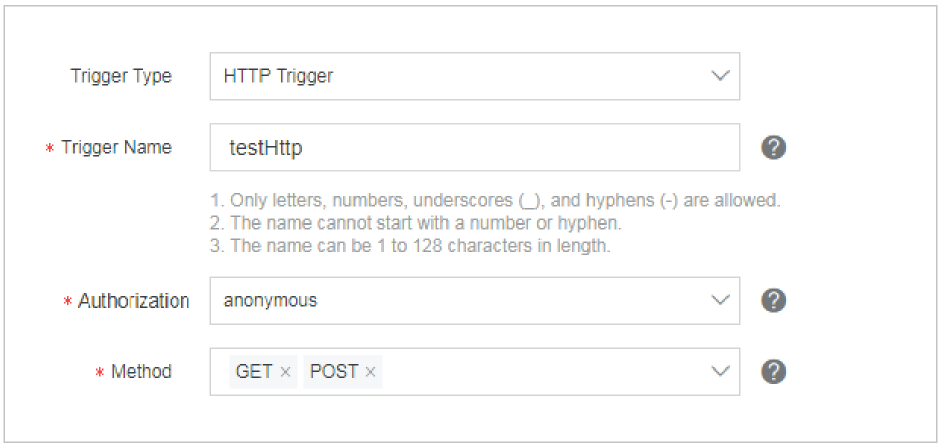
Select a region, such as China East 2 (Shanghai).

Select a service in the left-side navigation pane.

Select the function in the service.

Select the **Triggers** tab, and click **Create Trigger**.

On the **Create Trigger** page, select **HTTP Trigger** and configure the parameters as follows, and then click **OK**.

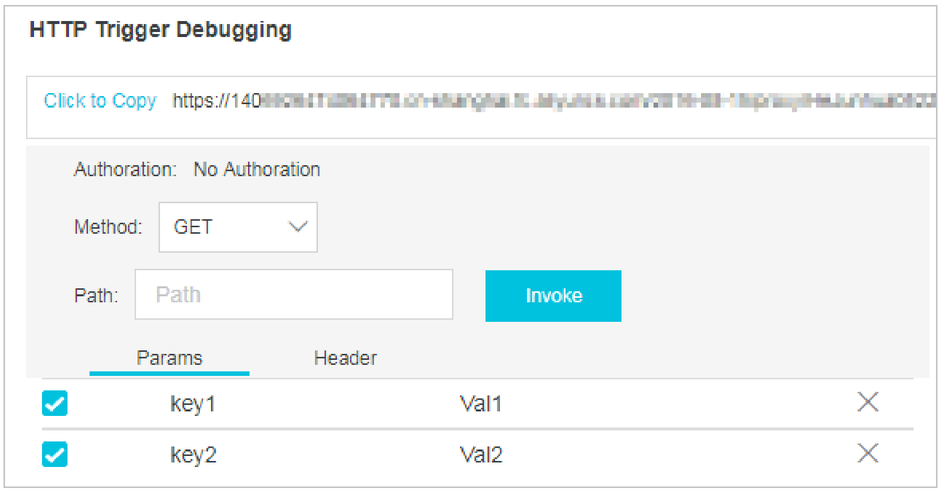


Trigger Type	HTTP Trigger
* Trigger Name	testHttp
	1. Only letters, numbers, underscores (_), and hyphens (-) are allowed. 2. The name cannot start with a number or hyphen. 3. The name can be 1 to 128 characters in length.
* Authorization	anonymous
* Method	GET × POST ×

Step 2. Debug the trigger

Select a function, select the **Code** tab, and then scroll down on the page.

Select the **Params** tab, and enter your key-value pairs. The key-value pairs are automatically added to the HTTP URL.


HTTP Trigger Debugging

Click to Copy https://140

Authoration: No Authoration

Method: GET

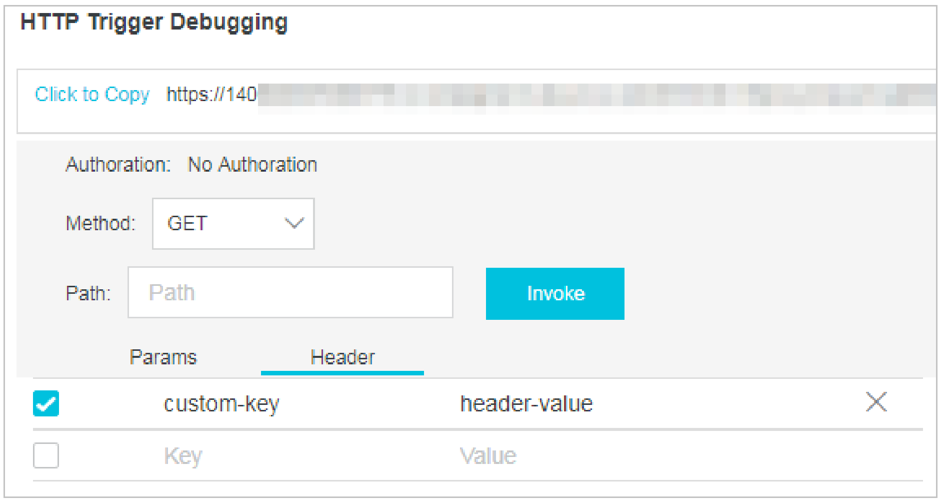
Path: Path

Invoke

Params Header

✓	key1	Val1
✓	key2	Val2

Select the **Header** tab to set the request header, and enter your key-value pairs. When authentication is required, the request header on this tab automatically contains the Date and Authorization keys.


HTTP Trigger Debugging

Click to Copy https://140

Authoration: No Authoration

Method: GET

Path: Path

Invoke

Params Header

✓	custom-key	header-value
☐	Key	Value

Click **Invoke**.

Example 2. SDK programming

This section uses the Python SDK as an example to describe how to create an HTTP trigger from an SDK.

Prerequisites

You have already deployed the Python environment.

You have already installed the Function Compute Python SDK and can run pip install aliyun-fc2 on your local device to obtain the Function Compute Python SDK.

Procedure

Compile a Function Compute function, and copy and paste the sample code to the main.js file in the code folder.

Create the file create_http_trigger.py, and copy and paste the following Python commands to the file.

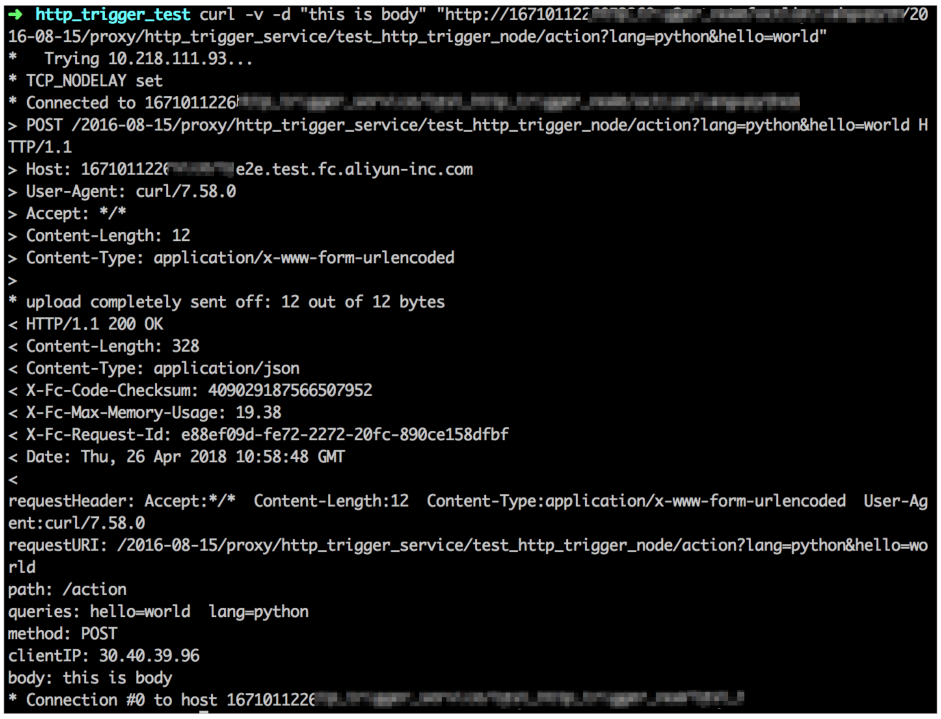
```
import fc2
import os
def main():
service_name = "http_trigger_service"
func_name = "test_http_trigger_node"
endpoint = os.getenv("FC_ENDPOINT")
url = "%s/2016-08-15/proxy/%s/%s" % (endpoint, service_name, func_name)
print url
fc_client = fc2.Client(
endpoint=endpoint,
accessKeyID=os.getenv("ACCESS_KEY_ID"),
accessKeySecret=os.getenv("ACCESS_KEY_SECRET"),
Timeout=5)
fc_client.create_service(service_name)
fc_client.create_function(service_name, func_name, "nodejs6", "main.handler", codeDir='./code')

trigger_config = {
"authType" : "anonymous",
"methods" : ["GET", "POST"],
}
fc_client.create_trigger(service_name, func_name, "trigger_on_echo", "http", trigger_config, "dummy_arn",
"")
main()
```

Run python create_http_trigger.py to create an HTTP trigger.

Send an HTTP request to call the function. Example:

```
curl -v  "<account-id>.<region>.fc.aliyuncs.com/2016-08-
15/proxy/http_trigger_service/test_http_trigger_node/action"
```

The result is as follows:



```
➜ http_trigger_test curl -v -d "this is body" "http://167101122[redacted]/20
16-08-15/proxy/http_trigger_service/test_http_trigger_node/action?lang=python&hello=world"
*   Trying 10.218.111.93...
* TCP_NODELAY set
* Connected to 1671011226[redacted]
> POST /2016-08-15/proxy/http_trigger_service/test_http_trigger_node/action?lang=python&hello=world H
TTP/1.1
> Host: 167101122[redacted]e2e.test.fc.aliyun-inc.com
> User-Agent: curl/7.58.0
> Accept: */*
> Content-Length: 12
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 12 out of 12 bytes
< HTTP/1.1 200 OK
< Content-Length: 328
< Content-Type: application/json
< X-Fc-Code-Checksum: 409029187566507952
< X-Fc-Max-Memory-Usage: 19.38
< X-Fc-Request-Id: e88ef09d-fe72-2272-20fc-890ce158dfbf
< Date: Thu, 26 Apr 2018 10:58:48 GMT
<
requestHeader: Accept:*/*  Content-Length:12  Content-Type:application/x-www-form-urlencoded  User-Ag
ent:curl/7.58.0
requestURI: /2016-08-15/proxy/http_trigger_service/test_http_trigger_node/action?lang=python&hello=wo
rld
path: /action
queries: hello=world  lang=python
method: POST
clientIP: 30.40.39.96
body: this is body
* Connection #0 to host 1671011226[redacted]
```

Troubleshooting

The main types of errors are request errors and function errors. A request error occurs when your request does is invalid and the response contains a 4xx status code. A function error occurs when the function is improperly programmed and a **5xx** status code is returned. Cases in which request and function errors may occur are described in the following table for your quick troubleshooting.

Error type	X-Fc-Error-Type	HTTP status code	Cause analysis	Still billed?
Request errors	FcCommonError	400	The request exceeds the limits for responses.	No
	FcCommonError	400	A request to invoke a function that requires identity authentication does not contain Date or Authorization.	No
	FcCommonError	403	The signature of a request to invoke a function that requires identity authentication is incorrect,	No

			that is, the Authorization is incorrect. A common cause is that the HTTP trigger requires identity authentication while the signature is invalid because the difference between the Date used to calculate the signature and the current time is more than 15 minutes.	
	FcCommonError	403	The method in the request is not configured in the HTTP trigger. For example, only the GET method is configured in the HTTP trigger while the POST method is requested.	No
	FcCommonError	404	An HTTP request is sent to invoke a function without any HTTP trigger.	No
User throttling	FcCommonError	429	You have been throttled. You can reduce the concurrent request volume or contact Function Compute to increase your concurrency.	No
Function errors	UnhandledInvocationError	502	The value returned by the function exceeds the	Yes

			limits for responses.	
	UnhandledInvocationError	502	The function code has a syntax error or exception.	Yes
	UnhandledInvocationError	502	An HTTP request is sent to a function that does not use an HTTP interface.	Yes
System error	FcCommonError	500	Function Compute system error. Please try again.	No
System throttling	FcCommonError	503	Function Compute has activated system throttling. Implement exponential backoff retry attempts.	No

If the problem persists, please contact us.

CDN events trigger

CDN events trigger

With 1,200+ nodes distributed across the globe, Alibaba Cloud CDN (Content Delivery Network) enables users to effectively reduce website response time to milliseconds, ensure smooth video streaming and handle large volumes of traffic. As a low-cost and scalable service, Alibaba Cloud CDN comes without any long-term contracts or minimum usage commitments. Based on the seamless integration of Alibaba Cloud Function Compute and Alibaba Cloud CDN, FC can receive and process different kinds of CDN events. The user can create CDN events trigger with filters to handle events from a specific domain, and process these events with specific functions. For example, the user can create CDN events trigger and some logging functions to handle CDN cache purge event from “www.taobao.com” domain.

CDN events trigger usage scenario:

1. When content is pre-warmed or purged, “CachedObjectsPushed ” or “CachedObjectsRefreshed ” event will fire and invoke user’ s function. So the user can get notified in time instead of keeping polling status from CDN.
2. When “CachedObjectsBlocked ” event fired, user’ s function can be invoked to delete the original content in time, instead of involving CDN team to handle it.
3. When CDN log file generated, “LogFileCreated ” event will fire and invoke user’ s function to copy or process log file immediately.

CDN events

CDN will convert the user specified event to JSON string, and invoke the function. Events supported by CDN events trigger as below:

Event Name	Event Version	Filter Key	Description
CachedObjectsRefreshed	1.0.0	domain	See RefreshObjectCaches API for details.
CachedObjectsBlocked	1.0.0	domain	CDN resource blocked
CachedObjectsPushed	1.0.0	domain	See PushObjectCache API for details.
LogFileCreated	1.0.0	domain	See DescribeCdnDomainLogs API for details.

CDN event schema：

CachedObjectsRefreshed，CachedObjectsPushed和CachedObjectsBlocked event schema：

```
{
"events": [
{
"eventName": "CachedObjectsRefreshed",//event name
"eventVersion": "1.0.0", // event version
"eventSource": "cdn", // event source name
"region": "cn-hangzhou", //default region:"cn-hangzhou"
"eventTime": "2018-03-16T14:19:55+08:00",//refresh start time
"traceId": "cf89e5a8-7d59-4bb5-a33e-4c3d08e25acf",//id of event source, used for debugging
"resource": {
"domain": "example.com"
},
"eventParameter": {
"objectPath": [
"/2018/03/16/13/33b430c57e7.mp4",//content URI
"/2018/03/16/14/4ff6b9bd54d.mp4"//content URI
],
"createTime": 1521180769,//refresh start time
"domain": "example.com",//domain
"completeTime": 1521180777,//refresh complete time
"objectType": "File", //refresh type, including File and Directory
"taskId": 2089687230 //refresh task ID
},
"userIdentity": {
"aliUid": "1xxxxxxxxxx" //account id
}
}
]
}
```

LogFileCreated event schema：

```
{
"events": [
{
"eventName": "LogFileCreated",//event name
"eventSource": "cdn",//event source name
"region": "cn-hangzhou",//default region:"cn-hangzhou"
"eventVersion": "1.0.0",//event versino
"eventTime": "2018-06-14T15:31:49+08:00",//event start time
"userIdentity": {
"aliUid": "1xxxxxxxxxxxx" //account id
},
"resource": {
"domain": "example.com"//domain
},
"eventParameter": {
"domain": "example.com",//domain
"endTime": 1528959900, //end time of log file
"fileSize": 1788115,//log file size
"filePath": "http://cdnlog.cn-hangzhou.oss.aliyun-
inc.com/www.aliyun.com/2017_12_27/www.aliyun.com_2017_12_27_0800_0900.gz?OSSAccessKeyId=xxxx&Expires=
xxxx&Signature=xxxx",//log file path
"startTime": 1528959600 //start time of log file
},
"traceId": "c6459282-6a4d-4413-894c-e4ea39686738" //id of event source, used for debugging
}
]
}
```

Configuration a CDN events trigger：

Trigger example：cdn_events_trigger.yml

```
triggerConfig:
eventName:   “LogFileCreated”
```

```
eventVersion:  “1.0.0”
notes:  “cdn events trigger test”
filter:
domain: { “www.taobao.com” ,” www.tmall.com” }
```

Trigger parameter descriptions：

- eventName is CDN event which invokes the function execution, can not be changed after creation.
- eventVersion is CDN event version which invokes the function execution, can not be changed after creation.
- notes Descriptions.
- filter Filters (Need to have at least one filter).

filter schema：

```

  filter:
key1: {value a，value b}
key2: {value c，value d}
```

Demos

There are 3 ways to set CDN events trigger for a given function, Function Compute Console, fcli and SDK:

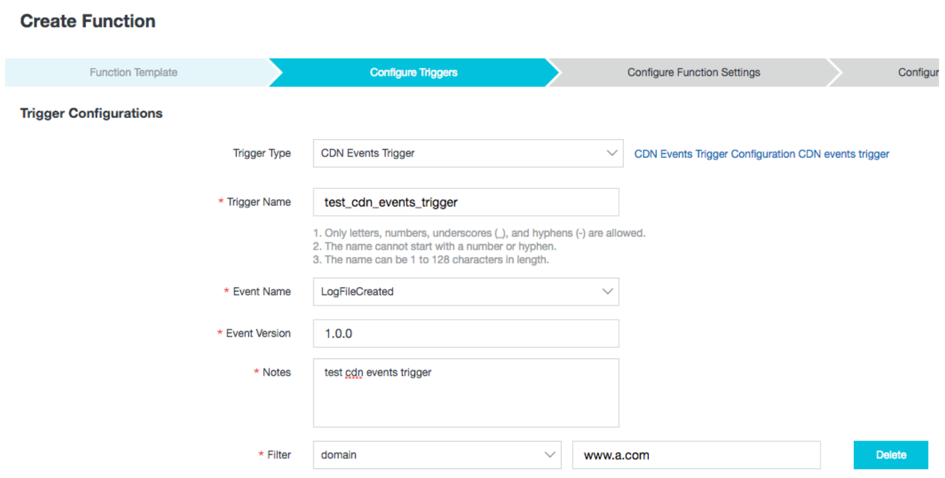
Demo 1: Console operations

This example describes how to use Function Compute Console to set CDN events trigger. For more information about how to create triggers, please see Use event source service and Create a trigger.

Log in on Function Compute Console，select region and service as you need. Please check Create a service if you need to create service.

Set a trigger while creating a function

1. Click **Create Function**, choose **Empty Function**, then click **Next**.
2. Select **CDN Events Trigger** and configure the parameters as below:


Create Function

Trigger Configurations

Trigger Type: CDN Events Trigger — CDN Events Trigger Configuration CDN events trigger

* Trigger Name: test_cdn_events_trigger

1. Only letters, numbers, underscores (_), and hyphens (-) are allowed.
2. The name cannot start with a number or hyphen.
3. The name can be 1 to 128 characters in length.

* Event Name: LogFileCreated

* Event Version: 1.0.0

* Notes: test cdn events trigger

* Filter: domain — www.a.com

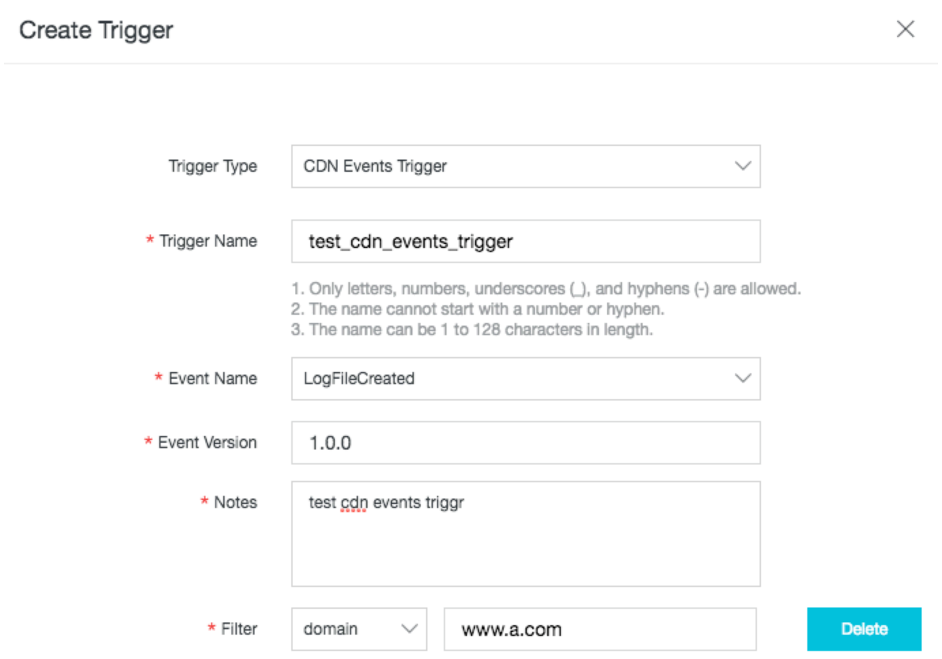
3. Configure Service Name, Function Name, Function Description and Runtime parameters, select **In-line Edit** and paste the following python runtime sample code , then click **Next**.

```
import json
import logging

LOG = logging.getLogger()

def handler(event, context):
logger = logging.getLogger()
eventObj = json.loads(event)["events"]
logger.info("EventCount: %d" % len(eventObj))
logger.info("eventName: %s" % eventObj[0]["eventName"])
logger.info("eventVersion: %s" % eventObj[0]["eventVersion"])
```

4. (Optional) Configure Permissions , click **Next**, check it all the information is error-free, then click **Create**.

Set a trigger after function creation

1. Select a function in the service, select the **Triggers** tab, then click **Create Trigger**.
2. Select **CDN Events Trigger** and configure the parameters as below, then click **OK**.


Create Trigger

Trigger Type: CDN Events Trigger

* Trigger Name: test_cdn_events_trigger

1. Only letters, numbers, underscores (_), and hyphens (-) are allowed.
2. The name cannot start with a number or hyphen.
3. The name can be 1 to 128 characters in length.

* Event Name: LogFileCreated

* Event Version: 1.0.0

* Notes: test cdn events triggr

* Filter: domain www.a.com Delete

Demo 2: fcli operations

First, create a yaml file as Trigger Config. A trigger with trigger config yaml file below will invoke some function when CDN receiving logFileCreated(version 1.0.0) event from www.taobao.com and www.tmall.com domains:

```
triggerConfig:
eventName: “LogFileCreated”
eventVersion: “1.0.0”
notes: “cdn events trigger test”
filter:
domain: { “www.taobao.com” ,” www.tmall.com” }
```

Command to create trigger under the function folder :

```
mkt serviceName/functionName -t timer -c TriggerConfig.yaml
```

For more about fcli, please see fcli

Demo 3: SDK programming

Using fc-python-sdk as an example to describe how to set a CDN events trigger by SDK. Function Compute provides fc-nodejs-sdk and fc-java-sdk as well.

Create trigger Python code

```
client = fc2.Client(
endpoint='<Your Endpoint>',
accessKeyID='<Your AccessKeyID>',
accessKeySecret='<Your AccessKeySecret>')
service_name = 'serviceName'
function_name = 'functionName'
trigger_name = 'triggerName'
trigger_type = 'cdn_events'
source_arn = 'acs:cdn:*:<Your Account ID>'
invocation_role = 'acs:ram::<Your Account ID>:role/<Your Invocation Role>'
trigger_config = {
'eventName': 'logFileCreated',
'eventVersion': '1.0.0',
'notes': 'notes',
'filter': {
'domain' : ['www.taobao.com'],
}
}

client.create_trigger(service_name, function_name, trigger_name, trigger_type, trigger_config, source_arn,
invocation_role)
```

Function Python code

```
import json
import logging

LOG = logging.getLogger()

def handler(event, context):
logger = logging.getLogger()
eventObj = json.loads(event)["events"]
logger.info("EventCount: %d" % len(eventObj))
logger.info("eventName: %s" % eventObj[0]["eventName"])
logger.info("eventVersion: %s" % eventObj[0]["eventVersion"])
```

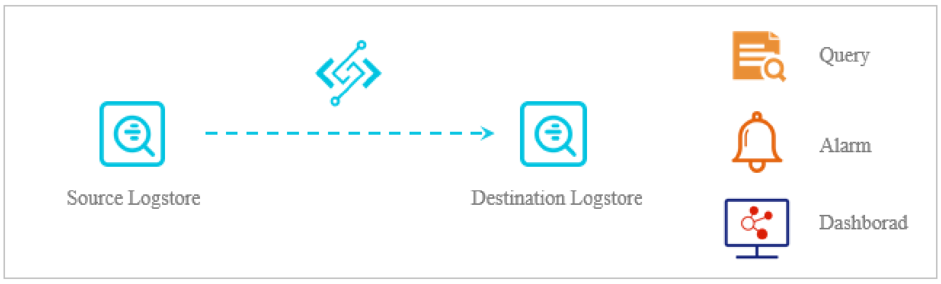
Please contact us for more information.

Log Service trigger

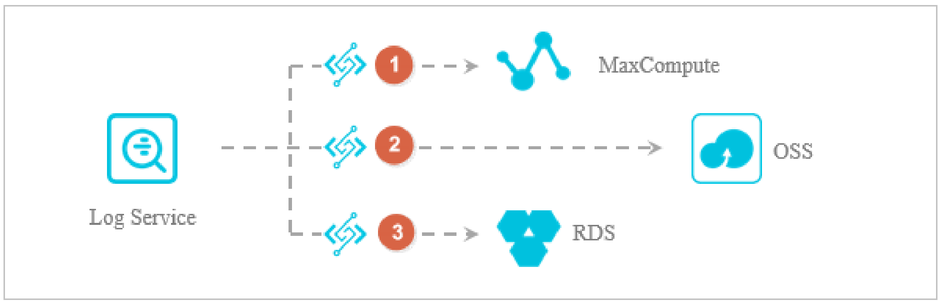
The Log Service trigger regularly subscribes to incremental data from the Logstore. The incremental data triggers Extract, Transform, and Load (ETL) of the log updates in Function Compute.

Scenarios

The Log Service trigger is suitable for Data scrubbing and ETL. Log Service allows you to quickly perform log collection, ETL, query, and analysis, as shown in the following figure:



The trigger supports shipping data to the destination, and building data pipelines between big data services in the cloud, as shown in the following figure:



Description:

1. Build and deliver Column-based storage.
2. Pre-process and deliver fields.
3. Customize and store response.

Configure Log Service trigger

Trigger example

The following slsTrigger.yml is a configuration template of the Log Service trigger.

```
triggerConfig:
sourceConfig:
project: "etl"
logstore: "etl-log"
jobConfig:
maxRetryTime: 3
triggerInterval: 60
functionParameter:
a: "b"
c: "d"
logConfig:
project: "ali-fc-test"
logstore: "test-store"
enable: true
```

Trigger parameters

- sourceConfig is the data source configuration parameter, and includes the following properties:
  - Logstore: The name of the data source (Logstore). The trigger regularly subscribes to data from this Logstore and sends the data to Function Compute for ETL. You cannot modify this parameter after you have created it.
- jobConfig is the task configuration parameter, and includes the following properties:
  - triggerInterval: The interval at which the function that is triggered by Log Service runs. Value range: [3, 600], unit: second. For example, triggerInterval: 60 indicates that this function reads the locations of data that occurs in the last 60 seconds in each shard at intervals of 60 seconds. The incremental data triggers executing this function. If your Logstore shards have a high traffic volume that is more than 1 MiB/s, we recommend that you use a shorter trigger interval. Therefore, the function can process data of a reasonable size.
  - maxRetryTime: The maximum number of retries allowed for a single trigger. Value range: [0, 100]. An error may occur when Log Service triggers function execution at the specified trigger interval, such as insufficient permissions, network failures, or function execution exceptions. The function execution may still fail after the maximum number of retries. In these conditions, Log Service triggers function execution again after the trigger interval. The impact of retries on the business varies according to the specific function code logic.
- functionParameter: The event parameter directly uses this parameter from the .yml file. Function usage depends on the custom logic of this function. Each function may require different function configurations. You must enter parameters for most default function templates according to the instructions. The default value is empty ({}).
- logConfig is the log configuration of the trigger, and includes the following properties:
  - project: The project name in Log Service.
  - logstore: The name of the Logstore that stores log files.
- enable is used to enable the trigger. Value range: true|false.

Event format

```
{
"parameter":{
"a":"b",
"c":"d"
},
"source":{
"endpoint":"http://cn-shanghai-intranet.log.aliyuncs.com",
"projectName":"vangie-fc-test",
"logstoreName":"fc-test",
"shardId":0,
"beginCursor":"MTUyMzI2NzI5NDY1NjI4MzgzNg==",
"endCursor":"MTUyMzI2NzI5NDY1NjI4MzgzNw=="
},
"jobName":"05c79f637c6b46eaa85911cae032cf47551af7bb",
"taskId":"d22697c0-2a41-4d35-b27c-dccec8856768",
"cursorTime":1523323454
}
```

- parameter: The function parameter that you configure for the trigger.
- source: The information of the log block that Function Compute reads from Log Service.
    - endpoint: The region of the Log Service project.
    - projectName: The project name.
    - logstoreName: The Logstore name.
    - shardId: The specified shard in the Logstore.
    - beginCursor: The location on the shard where the triggered function starts consuming data.
    - endCursor: The location on the shard where the triggered function stops consuming data.
- jobName: The name of the ETL job in Log Service. A Log Service trigger in Function Compute corresponds to an ETL job in Log Service.
- taskId: The identifier of the execution of the specified function. This execution corresponds to an ETL job.
- cursorTime: The unix_timestamp of the last log entry that reaches the backend of Log Service. The function reads this data when retrieving log entries.

Create Log Service trigger

Sample 1. By using Function Compute console

Make sure that Function Compute and Log Service are deployed in the same region. Otherwise, Function Service cannot locate Log Service when you configure the Log Service trigger. For more information, see Regions and zones.

Log on to the Alibaba Cloud console.

Navigate to the AliyunLogETLRole role management page, click the **Confirm Authorization Policy** button to grant Function Compute the AliyunLogETLRole access policy.

Log on to Log Service console, create one Logstore to process log files and data sources, and create another Logstore to store the log files that are generated by Function Compute. For more information, see *Log Service* topic Preparation.

Log on to the Function Compute console and create a Service. In the **Create Service** dialog box that appears:

Select a region, in this sample, we use the **China (Shanghai)** region.

Click **Create Service**.

In the Create Service dialog box that appears, enter the service name, in this sample, the service name is **log-com**.

Enable the **Advanced Settings** option.

In the **Log Configs**, set your available Log Project and Logstore.

In the **Role Config**, select **Create new role** from the Role Operation drop-down list, select AliyunLogFullAccess and AliyunLogReadOnlyAccess from the System Policies drop-down list.

Click **Authorize** and **OK** to confirm your action.

In the left-side navigation pane, select the new service you created.

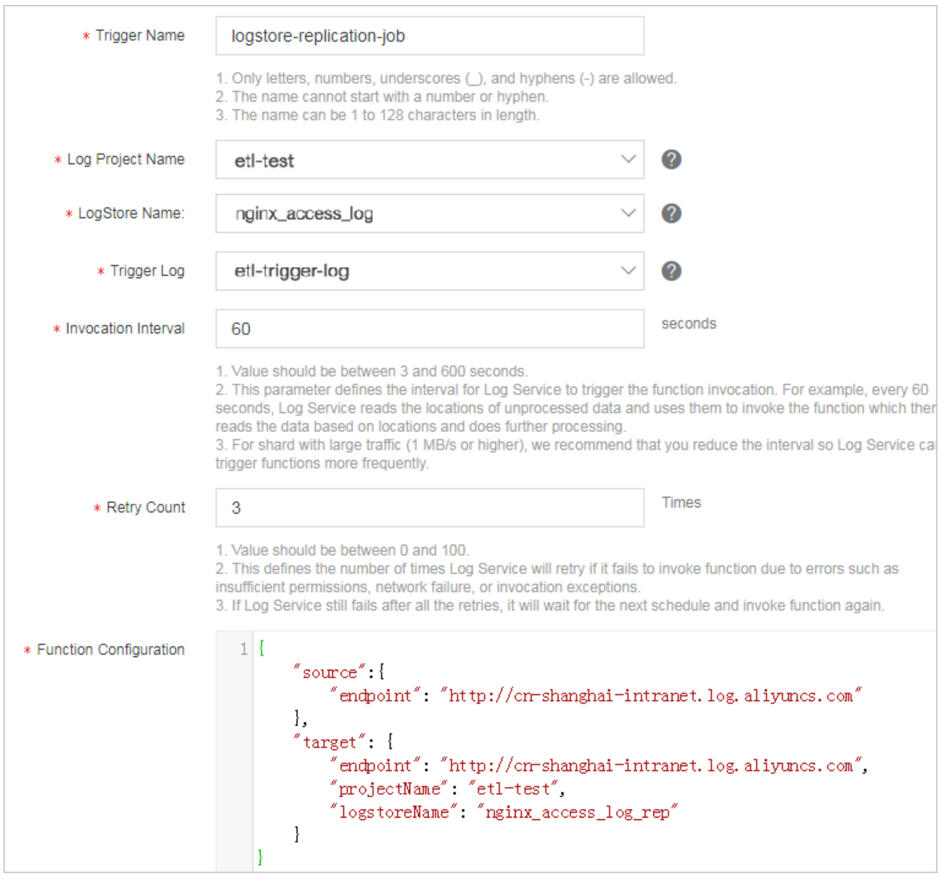
Click **Create Function** to go to the **Create Function** page.

i. Click **Select All**, and select **python2.7** from the drop-down list. This sample takes Python code execution for example.

Click **Select** in **Empty Function**.

Note: You can create the trigger during or after function creation. Then, you can configure the trigger. For more information, see Basic operations.

Select **Log Service (Log)** from the Trigger Type drop-down list, set the **Trigger Name**, **Log Project Name**, **Trigger Log**, **Invocation Interval**, **Retry Count**, and **Function Configuration** configuration. In this sample, we set the trigger as follows:



* Trigger Name: logstore-replication-job

1. Only letters, numbers, underscores (_), and hyphens (-) are allowed.
2. The name cannot start with a number or hyphen.
3. The name can be 1 to 128 characters in length.

* Log Project Name: etl-test

* LogStore Name: nginx_access_log

* Trigger Log: etl-trigger-log

* Invocation Interval: 60 seconds

1. Value should be between 3 and 600 seconds.
2. This parameter defines the interval for Log Service to trigger the function invocation. For example, every 60 seconds, Log Service reads the locations of unprocessed data and uses them to invoke the function which ther reads the data based on locations and does further processing.
3. For shard with large traffic (1 MB/s or higher), we recommend that you reduce the interval so Log Service ca trigger functions more frequently.

* Retry Count: 3 Times

1. Value should be between 0 and 100.
2. This defines the number of times Log Service will retry if it fails to invoke function due to errors such as insufficient permissions, network failure, or invocation exceptions.
3. If Log Service still fails after all the retries, it will wait for the next schedule and invoke function again.

* Function Configuration:

```
{
    "source":{
        "endpoint": "http://cn-shanghai-intranet.log.aliyuncs.com"
    },
    "target": {
        "endpoint": "http://cn-shanghai-intranet.log.aliyuncs.com",
        "projectName": "etl-test",
        "logstoreName": "nginx_access_log_rep"
    }
}
```

You can set the **Function Configuration** field according to the event parameter in your function code. In this sample, we set the **Function Configuration** as follows:

```
{
"source":{
"endpoint": "http://cn-shanghai-intranet.log.aliyuncs.com"
},
"target": {
"endpoint": "http://cn-shanghai-intranet.log.aliyuncs.com",
"projectName": "etl-test",
"logstoreName": "nginx_access_log_rep"
}
}
```

Configure the **Service Name**, **Function Name**, **Function Description**, **Runtime**, and **Runtime Environment** parameters.

v. Click **Next**.

vi. Make sure that all the settings are correct, and then click **Create**.

Sample 2. By using Pyhton SDK

This section takes fc-python-sdk for example:

```
import fc2

client = fc2.Client(
endpoint = '<Your Endpoint>',
accessKeyID = '<Your AccessKeyID>',
accessKeySecret = '<Your AccessKeySecret>')
service_name = '<service_name>'
function_name = '<function_name>'
trigger_name = '<trigger_name>'
# Create log trigger
log_trigger_config = {
'sourceConfig': {
'logstore': 'log_store_source'
},
'jobConfig': {
'triggerInterval': 60,
'maxRetryTime': 10
},
'functionParameter': {},
'logConfig': {
'project': 'log_project',
'logstore': 'log_store'
},
'enable': False
}
source_arn = 'acs:log:cn-shanghai:12345678:project/log_project'
invocation_role = 'acs:ram::12345678:role/aliyunlogetlrole'
client.create_trigger('service_name', 'function_name', 'trigger_name', 'oss',
log_trigger_config, source_arn, invocation_role)
```

References

Log Service trigger use case Demo overview.

Permission management

RAM

Function Compute supports RAM based resource access management. You can access Function Compute using RAM user or STS token. The RAM permissions of each operations are defined as follows:

Note: In RAM, theResource format is acs:fc:${region}:${account-id}:${resource}. The following table lists only the content of ${resource}. For example, you can specify the following authorization policies for CreateService:

```
{
"Version": "1",
"Statement": [
{
"Action": [
"fc:CreateService"
],
"Resource": "acs:fc:*:*:services/*",
"Effect": "Allow"
}
]
}
```

API	Action	Resource
CreateService	fc:CreateService	services/*
ListServices	fc:ListServices	services/*
GetService	fc:GetService	services/${serviceName}
UpdateService	fc:UpdateService	services/${serviceName}
DeleteService	fc:DeleteService	services/${serviceName}
CreateFunction	fc:CreateFunction	services/${serviceName}/functions/*
ListFunctions	fc:ListFunctions	services/${serviceName}/functions/*
GetFunction	fc:GetFunction	services/${serviceName}/functions/${functionName}
UpdateFunction	fc:UpdateFunction	services/${serviceName}/functions/${functionName}
DeleteFunction	fc:DeleteFunction	services/${serviceName}/functions/${functionName}
InvokeFunction	fc:InvokeFunction	services/${serviceName}/functions/${functionName}
CreateTrigger	fc:CreateTrigger	services/${serviceName}/functions/${functionName}/triggers/*
ListTriggers	fc:ListTriggers	services/${serviceName}/functions/${functionName}/triggers/*
GetTrigger	fc:GetTrigger	services/${serviceName}/functions/${functionName}/triggers/${triggerName}

UpdateTrigger	fc:UpdateTrigger	services/${serviceName}/functions/${functionName}/triggers/${triggerName}
DeleteTrigger	fc:DeleteTrigger	services/${serviceName}/functions/${functionName}/triggers/${triggerName}

User permissions

One must manage the following permissions before using Function Compute to build an application:

> One must authorize Function Compute to collect and write the function execution logs to the logproject/logstore you specified.
>
> If your function needs to access other Alibaba Cloud resources under your account, for example, data in OSS, you can create a RAM role and grant relevant permission to it. Function Compute will inherite the permissions from that role to run the functions for you.

Function Compute uses role-based permission management of Alibaba Cloud RAM.

Service role

Each service is associated with a RAM role (service role). When creating or updating a service, you can specify a role for the service. The permissions you grant to the role determine the operations that Function Compute can perform when running functions in the service.

Invocatino role

Each trigger is associated with a RAM role (invocation role). When creating a trigger, you must specify a role for the trigger. The permissions you grant to the role determine the functions that the event source service can invoke when an event occurs. For example, OSS must obtain your permission to call the associated function to process the event that occurs.

> Note: When creating a RAM role, you actually authorize Function Compute to play the role. In this case, you must have the ram:PassRole operation permission. If the role is created by an administrator, you do not need to grant any permissions except the ram:PassRole operation permission, because the administrator has the full permissions including ram:PassRole.

Monitor service

Metrics userguide

Function Compute metrics are reported in three dimensions:

- Region dimension
- Service dimension
- Function dimension

All metric values are aggregated to 1-minute granularity and are stored as timeseries data. The Duration metric is aggregated as average and all other metrics are aggregated as sum.

Region Dimension

Region level metrics are useful to measure and monitor the overall usage of FC resources in a region. The following table lists metrics and descriptions.

Region-dimension metric	Unit	Description
TotalInvocations	Count	Total number of invocations of the region, including all synchronous InvokeFunction API calls and requests handled by FC asynchronously. Note that asynchronous invocation metrics are only reported after FC finished handling the requests, not at the time asynchronous InvokeFunction API returns 202 (i.e. requests with HTTP status 202 are excluded).
BillableInvocations	Count	Number of invocations of the region that can be metered and billed, including the InvokeFunction requests that return the HTTP status 200 or the asynchronous invocation requests successfully processed by FC. Note that function errors such as

		syntax errors, handled/unhandled errors and function timeout are included in this metric.
Throttles	Count	Number of InvokeFunction requests of the region that are throttled, including the requests that call the InvokeFunction API for access and return the HTTP status 429 and the asynchronous invocation requests that fail to be executed due to throttling.
ClientErrors	Count	Number of InvokeFunction requests of the region that returned with the HTTP status 4xx (excluding 429) and the asynchronous invocation requests that fail to be executed due to client side errors.
ServerErrors	Count	Number of InvokeFunction requests of the region that returned the HTTP status 5XX and the asynchronous invocation requests that fail to be run due to server side (FC) errors.
BillableInvocationsRate	%	Percentage of BillableInvocations of the region from the total number of InvokeFunction requests of the region.
ThrottlesRate	%	Percentage of Throttles of the region from the total number of InvokeFunction requests of the region.
ClientErrorsRate	%	Percentage of ClientErrors of the region from the total number of InvokeFunction requests of the region.
ServerErrorsRate	%	Percentage of ServerErrors of the region from the total number of requests of the region.

Service Dimension

Service level metrics are useful to measure and monitor the overall usage of FC resources in a service. The following table lists metrics and descriptions.

Service-dimension metric	Unit	Description
TotalInvocations	Count	Total number of invocations of the service, including all synchronous InvokeFunction API calls and requests handled by FC asynchronously. Note that

		asynchronous invocation metrics are only reported after FC finished handling the requests, not at the time asynchronous InvokeFunction API returns 202 (i.e. requests with HTTP status 202 are excluded).
BillableInvocations	Count	Number of invocations of the service that can be metered and billed, including the InvokeFunction requests that return the HTTP status 200 or the asynchronous invocation requests successfully processed by FC. Note that function errors such as syntax errors, handled/unhandled errors and function timeout are included in this metric.
Throttles	Count	Number of InvokeFunction requests of the service that are throttled, including the requests that call the InvokeFunction API for access and return the HTTP status 429 and the asynchronous invocation requests that fail to be executed due to throttling.
ClientErrors	Count	Number of InvokeFunction requests of the service that returned with the HTTP status 4xx (excluding 429) and the asynchronous invocation requests that fail to be run due to client side errors.
ServerErrors	Count	Number of InvokeFunction requests of the service that returned the HTTP status 5XX and the asynchronous invocation requests that fail to be run due to server side (FC) errors.
BillableInvocationsRate	%	Percentage of BillableInvocations of the service from the total number of InvokeFunction requests of the service.
ThrottlesRate	%	Percentage of Throttles of the service from the total number of InvokeFunction requests of the service.
ClientErrorsRate	%	Percentage of ClientErrors of the service from the total number of InvokeFunction requests of the service.
ServerErrorsRate	%	Percentage of ServerErrors of the service from the total number of requests of the service.

Function Dimension

Function level metrics are useful to measure and monitor the usage of resources of a function. The following table lists the function-dimension metrics.

Function-dimension metric	**Unit**	**Description**
AvgDuration	Millisecond	The average time in milliseconds elapsed from the start of function execution to the stop during the aggregation period (1-minute). Unlike the metering rules that round up the request duration by 100ms, this metric reports actual function execution time.
MaxMemoryUsage	MB	The maximum memory in MB used by function executions during the aggregation period (1-minute).
TotalInvocations	Count	Total number of invocations of the function, including all synchronous InvokeFunction API calls and requests handled by FC asynchronously. Note that asynchronous invocation metrics are only reported after FC finished handling the requests, not at the time asynchronous InvokeFunction API returns 202 (i.e. requests with HTTP status 202 are excluded).
BillableInvocations	Count	Number of invocations of the function that can be metered and billed, including the InvokeFunction requests that return the HTTP status 200 or the asynchronous invocation requests successfully processed by FC. Note that function errors such as syntax errors, handled/unhandled errors and function timeout are included in this metric
FunctionErrors	Count	Number of function invocations encountered errors such as Handled/Unhandled/OOM/timeout that are caused by function execution.
Throttles	Count	Number of InvokeFunction requests of the function that are throttled, including the requests that call the InvokeFunction API for access and return the HTTP status 429 and the asynchronous invocation requests that fail to be

		executed due to throttling.
ClientErrors	Count	Number of InvokeFunction requests of the function that returned with the HTTP status 4xx (excluding 429) and the asynchronous invocation requests that fail to be run due to client side errors.
ServerErrors	Count	Number of InvokeFunction requests of the function that returned the HTTP status 5XX and the asynchronous invocation requests that fail to be run due to server side (FC) errors
BillableInvocationsRate	%	Percentage of BillableInvocations of the function from the total number of InvokeFunction requests of the function.
ThrottlesRate	%	Percentage of Throttles of the region from the function number of InvokeFunction requests of the function.
ClientErrorsRate	%	Percentage of ClientErrors of the function from the total number of InvokeFunction requests of the function.
ServerErrorsRate	%	Percentage of ServerErrors of the function from the total number of requests of the function.

Metrics data userguide

This section describes how to use the Alibaba Cloud OpenAPI or the CloudMonitor SDK to query Function Compute (FC) metrics from CloudMonitor Service (CMS).

CloudMonitor Interface introduction

CloudMonitor JavaSDK user manual

Project

FC metrics data is available for querying under CMS project: acs_fc.

Sample Java code using CMS SDK:

```
QueryMetricRequest request = new QueryMetricRequest();
request.setProject("acs_fc");
```

StartTime and EndTime

CloudMonitor metrics data time range is specified by (StartTime, EndTime] inputs where StartTime timestamp is exclusive and EndTime timestamp is inclusive.

CloudMonitor data retention is 31 days. The interval between StartTime and EndTime cannot exceed 31 days, and data earlier than 31 days cannot be queried.

For more detailed parameters information, see *CloudMonitor* Interface introduction.

Sample Java code using CMS SDK:

```
request.setStartTime("2017-04-26 08:00:00");
request.setEndTime("2017-04-26 09:00:00");
```

Dimensions

FC metrics are dimensioned into region, service, and function.

- Set Dimensions as the following to access regional dimension metrics data:

```
{"region": "${your_region}"}
```

- Set Dimensions as the following to access service-dimension metric data:

```
{"region": "${your_region}", "serviceName": "${your_serviceName}"}
```

- Set Dimensions as the following to access function-dimension metric data:

```
{"region": "${your_region}", "serviceName": "${your_serviceName}", "functionName": "${your_functionName}"}
```

Note: “Dimensions” parameter is a JSON string and has only one Key-Value pair for FC.

Sample Java code using CMS SDK:

```
request.setDimensions("{\"region\":\"your_region\"}");
```

Period

FC metrics are aggregated in 60s period.Sample Java code using CMS SDK:

```
request.setPeriod("60");
```

Metric

Sample Java code using CMS SDK:

```
request.setMetric("your_metric");
```

The following table lists all the supported metric keys.

Metric	Metric name
RegionTotalInvocations	Region total invocations
RegionBillableInvocations	Region-dimension billable invocations
RegionThrottles	Region-dimension throttles
RegionClientErrors	Region-dimension client errors
RegionServerErrors	Region-dimension server errors
RegionBillableInvocationsRate	Percentage of region-dimension billable invocations
RegionThrottlesRate	Percentage of region-dimension throttles
RegionClientErrorsRate	Percentage of region-dimension client errors
RegionServerErrorsRate	Percentage of region-dimension server errors
ServiceTotalInvocations	Service-dimension total invocations
ServiceBillableInvocations	Service-dimension billable invocations
ServiceThrottles	Service-dimension throttles
ServiceClientErrors	Service-dimension client errors
ServiceServerErrors	Service-dimension server errors
ServiceBillableInvocationsRate	Percentage of service-dimension billable invocations
ServiceThrottlesRate	Percentage of service-dimension throttles
ServiceClientErrorsRate	Percentage of service-dimension client errors
ServiceServerErrorsRate	Percentage of service-dimension server errors
FunctionTotalInvocations	Function-dimension total invocations

FunctionBillableInvocations	Function-dimension billable invocations
FunctionFunctionErrors	Function-dimension function errors
FunctionThrottles	Function-dimension throttles
FunctionFunctionErrorsRate	Percentage of function-dimension function errors
FunctionClientErrors	Function-dimension client errors
FunctionServerErrors	Function-dimension server errors
FunctionBillableInvocationsRate	Percentage of function-dimension billable invocations
FunctionThrottlesRate	Percentage of function-dimension throttles
FunctionClientErrorsRate	Percentage of function-dimension client errors
FunctionServerErrorsRate	Percentage of function-dimension server errors
FunctionAvgDuration	Function-dimension average duration
FunctionMaxMemoryUsage	Function-dimension maximum memory usage

Example

Adding CMS SDK using Maven pom.xml:

```
...
<dependencies>
<dependency>
<groupId>com.aliyun</groupId>
<artifactId>aliyun-java-sdk-core</artifactId>
<version>3.1.0</version>
</dependency>
<dependency>
<groupId>com.aliyun</groupId>
<artifactId>aliyun-java-sdk-cms</artifactId>
<version>5.0.1</version>
</dependency>
</dependencies>
...
```

Sample code:

```
import com.alibaba.fastjson.JSONObject;
import com.aliyuncs.DefaultAcsClient;
import com.aliyuncs.IAcsClient;
import com.aliyuncs.cms.model.v20170301.QueryMetricListRequest;
import com.aliyuncs.cms.model.v20170301.QueryMetricListResponse;
import com.aliyuncs.exceptions.ClientException;
import com.aliyuncs.exceptions.ServerException;
import com.aliyuncs.http.FormatType;
import com.aliyuncs.profile.DefaultProfile;
import com.aliyuncs.profile.IClientProfile;

public class MonitorService {
public static void main(String[] args) {
IClientProfile profile = DefaultProfile.getProfile("cn-hangzhou", "<your_access_key_id>",
"<your_access_key_secret>");
IAcsClient client = new DefaultAcsClient(profile);


QueryMetricListRequest request = new QueryMetricListRequest();
request.setProject("acs_fc");
request.setPeriod("60");
request.setStartTime("2017-04-26 16:20:00");
request.setEndTime("2017-04-26 16:30:00");
request.setAcceptFormat(FormatType.JSON);

try {
// Region dimension
JSONObject dim = new JSONObject();
request.setMetric("RegionTotalInvocations"); // Select the metric
dim.put("region", "<your_region>"); // 如: cn-shanghai
request.setDimensions(dim.toJSONString());
QueryMetricListResponse response = client.getAcsResponse(request);
System.out.println(response.getCode());
System.out.println(response.getMessage());
System.out.println(response.getRequestId());
System.out.println(response.getDatapoints());

// Service dimension
dim = new JSONObject();
request.setMetric("ServiceTotalInvocations"); // Select the metric
dim.put("region", "<your_region>");
dim.put("serviceName", "<your_service_name>");
request.setDimensions(dim.toJSONString());
response = client.getAcsResponse(request);
System.out.println(response.getCode());
System.out.println(response.getMessage());
System.out.println(response.getRequestId());
System.out.println(response.getDatapoints());

// Function dimension
dim = new JSONObject();
request.setMetric("FunctionTotalInvocations"); // Select the metric
dim.put("region", "<your_region>");
dim.put("serviceName", "<your_service_name>");
dim.put("functionName", "<your_function_name>");
request.setDimensions(dim.toJSONString());
response = client.getAcsResponse(request);
System.out.println(response.getCode());
System.out.println(response.getMessage());
System.out.println(response.getRequestId());
System.out.println(response.getDatapoints());
} catch (ServerException e) {
```

```
e.printStackTrace();
} catch (ClientException e) {
e.printStackTrace();
}
}
}
```

Limits

Service resource restrictions

Restriction	Default value
Maximum number of functions that can be created under a single service	50
Maximum number of triggers that can be created under a single function	10

Function runtime envrionment restrictions

Resource	Default value
Temporary disk space (space of “/tmp”)	1024 MB
Number of file descriptors	1024
Number of processes and threads (total)	1024
Maximum execution duration per request	600s
Request payload size for synchronous function call	6 MB
Response body for synchronous function call	6 MB
Request payload size for asynchronous function call	128 KB
Code package size (compressed .zip or .jar file)	100 MB
Uncompressed code size	500 MB

Resource restrictions per account in each region

Resource	Default value
Number of functions that can be concurrently executed	100

Total size of the packages that can be uploaded	100 GB

Subaccount userguide

RAM User Console Logon Prerequisites

The following prerequisites should be met when a RAM user is used to login to Function Compute console:

- Console Logon is enabled, and logon user name and password are set for the RAM user. For more information, see Logon to the console using a RAM user.

Permissions Required by the RAM User

A RAM policy is required to be attached to the RAM user so it has necessary permissions to access cloud services. You can add, delete, or modify below policy template to grants the RAM user required permissions.

```
{
"Version": "1",
"Statement": [
{
"Action": "fc:*",
"Resource": "*",
"Effect": "Allow"
},
{
"Action": [
"ram:PassRole"
],
"Effect": "Allow",
"Resource": "*"
},
{
"Effect": "Allow",
"Action": [
"log:ListProject",
"log:ListLogStore"
],
"Resource": "acs:log:*:*:project/*"
},
{
"Effect": "Allow",
"Action": [
"ram:ListRoles"
],
"Resource": [
"acs:ram:*:*:role/*"
]
},
{
"Action": [
"oss:ListBucket"
],
"Effect": "Allow",
"Resource": "*"
},
{
"Action": [
"oss:GetBucketEventNotification",
"oss:PutBucketEventNotification",
"oss:DeleteBucketEventNotification"
],
"Effect": "Allow",
"Resource": "*"
}
]
}
```

If you want to restrict the RAM user to have read-only permissions such as getting functions, invoking functions, below policy template can be attached to your RAM user:

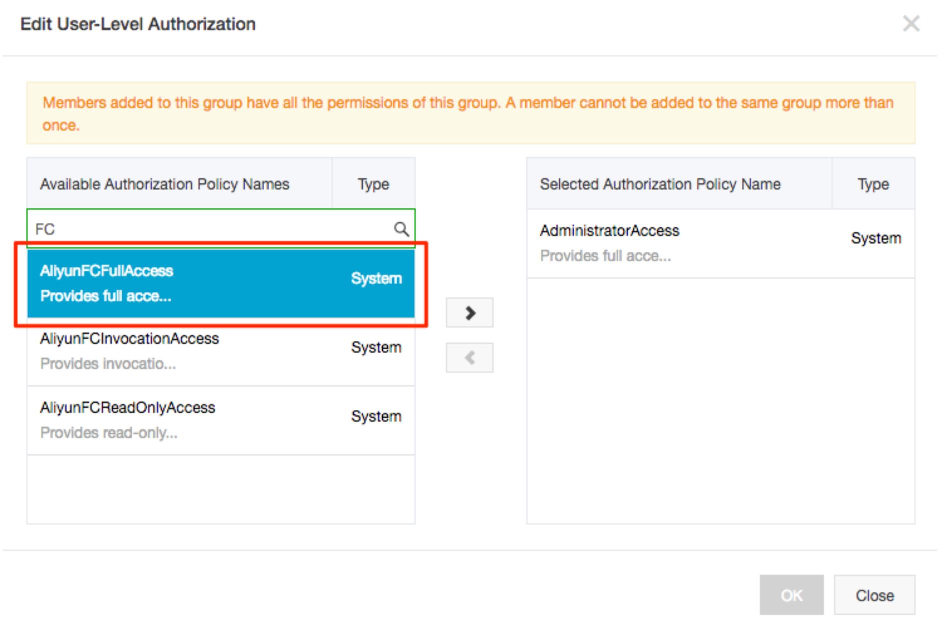
```
{
"Version": "1",
"Statement": [
{
"Effect": "Allow",
"Action": "fc:ListService",
"Resource": "acs:fc:cn-shanghai:*:services/*"
},
{
"Effect": "Allow",
"Action": [
"fc:GetFunction"
],
"Resource": [
"acs:fc:*:*:services/your-helloworld-fc/functions/*",
"acs:fc:*:*:services/your-helloworld-oss/functions/*"
]
},
{
"Effect": "Allow",
"Action": [
"fc:InvokeFunction"
],
"Resource": [
"acs:fc:*:*:services/your-helloworld-fc/functions/your-hello-world-fc"
]
},
{
"Action": [
"ram:PassRole"
],
"Effect": "Allow",
"Resource": "*"
},
{
"Effect": "Allow",
"Action": [
"log:ListProject",
"log:ListLogStore"
],
"Resource": "acs:log:*:*:project/*"
},
{
"Effect": "Allow",
"Action": [
"ram:ListRoles"
],
"Resource": [
"acs:ram:*:*:role/*"
]
},
{
"Action": [
"oss:ListBucket"
],
"Effect": "Allow",
"Resource": "*"
},
{
"Action": [
"oss:GetBucketEventNotification",
"oss:PutBucketEventNotification",
"oss:DeleteBucketEventNotification"
],
"Effect": "Allow",
"Resource": "*"
}
]
}
```

Authorize a RAM User to Perform Basic Functionalities

Basic functionalities include creating and deleting services, obtaining service information, creating and deleting functions, updating and obtaining function information, and executing functions. Operations related to logging and triggers are excluded. Below are the fundamental permissions that are needed for logging and triggering. Follow these steps to authorize the RAM user.

Grant the AliyunFCFullAccess permission to the RAM user.

In **Resource Access Management** console, select **Users** > **Authorize** for the RAM user


Edit User-Level Authorization

Members added to this group have all the permissions of this group. A member cannot be added to the same group more than once.

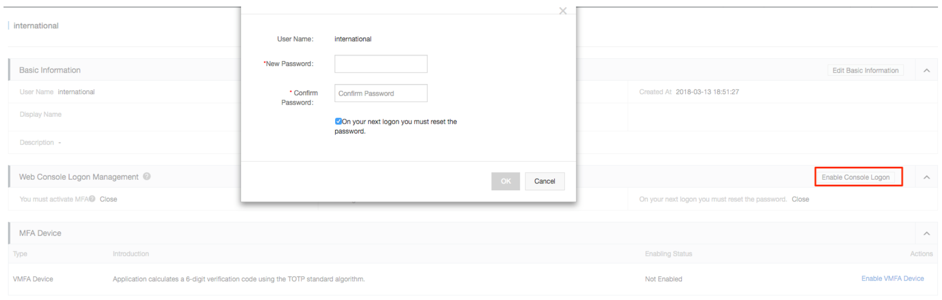
Available Authorization Policy Names	Type
FC	
AliyunFCFullAccess Provides full acce...	System
AliyunFCInvocationAccess Provides invocatio...	System
AliyunFCReadOnlyAccess Provides read-only...	System

Selected Authorization Policy Name	Type
AdministratorAccess Provides full acce...	System

Logon to console using the RAM user to perform basic functionalities.

After the preceding steps are completed, the RAM user can be used to create, delete, describe and update services/functions and invoke functions.

In **Resource Access Management** console, select **Users**, inside the User Details select **Web Console Logon Management** > *Enable Console Logon** (enter and confirm password).


international

Basic Information

User Name international

Display Name

Description -

Created At 2018-03-13 18:51:27

Web Console Logon Management

You must activate MFA Close

On your next logon you must reset the password. Close

MFA Device

Type	Introduction	Enabling Status	Actions
VMFA Device	Application calculates a 6-digit verification code using the TOTP standard algorithm.	Not Enabled	Enable VMFA Device

User Name: international

*New Password:

* Confirm Password: Confirm Password

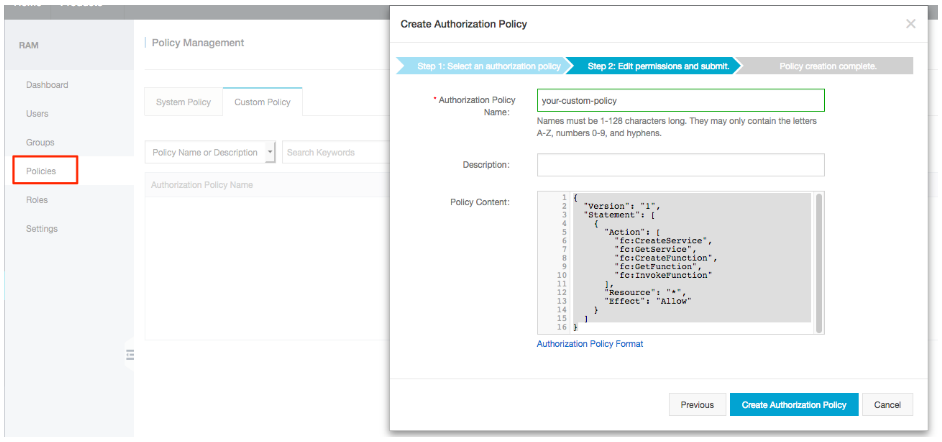
[✓] On your next logon you must reset the password.

Configure function logging. Skip this step if you do not want the RAM user to have log related functionalities.

Click **Create Function** and enter the basic information then execute the code.

How to restrict RAM user permissions, for example, only allowing the RAM user to create services, list services, create functions, and invoke functions?

Some basic RAM concepts of RAM such as the cloud resource policy are helpful before moving forward.


RAM
Dashboard
Users
Groups
Policies
Roles
Settings
Policy Management
System Policy
Custom Policy
Policy Name or Description
Search Keywords
Authorization Policy Name
Create Authorization Policy
Step 1: Select an authorization policy
Step 2: Edit permissions and submit.
Policy creation complete.
Authorization Policy Format
Previous
Create Authorization Policy
Cancel

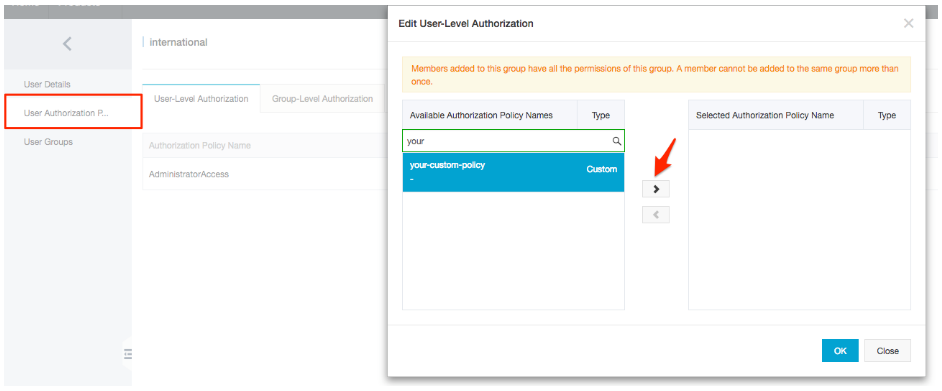
* Authorization Policy Name: your-custom-policy

Names must be 1-128 characters long. They may only contain the letters A-Z, numbers 0-9, and hyphens.

Description:

Policy Content:

```
{
  "Version": "1",
  "Statement": [
    {
      "Action": [
        "fc:CreateService",
        "fc:GetService",
        "fc:CreateFunction",
        "fc:GetFunction",
        "fc:InvokeFunction"
      ],
      "Resource": "*",
      "Effect": "Allow"
    }
  ]
}
```



Members added to this group have all the permissions of this group. A member cannot be added to the same group more than once.

Available Authorization Policy Names	Type
your	
your-custom-policy -	Custom

Selected Authorization Policy Name	Type

Create a custom policy to grant services, functions creation, listing and execution. Attach the custom policy to the service role. See the preceding figures for more details. Below sample policy template can be modifed and reused.

```
{
"Version": "1",
"Statement": [
{
"Action": [
"fc:CreateService",
"fc:GetService",
"fc:CreateFunction",
"fc:GetFunction",
"fc:InvokeFunction"
],
"Resource": "*",
"Effect": "Allow"
}
]
}
```

For more information about the action and resource, see Function Compute permission management.

Note: A role can be associated with up to five custom policies. We recommend that you organize multiple custom policies into one to avoid exceeding the policy limit.

```
  {
```

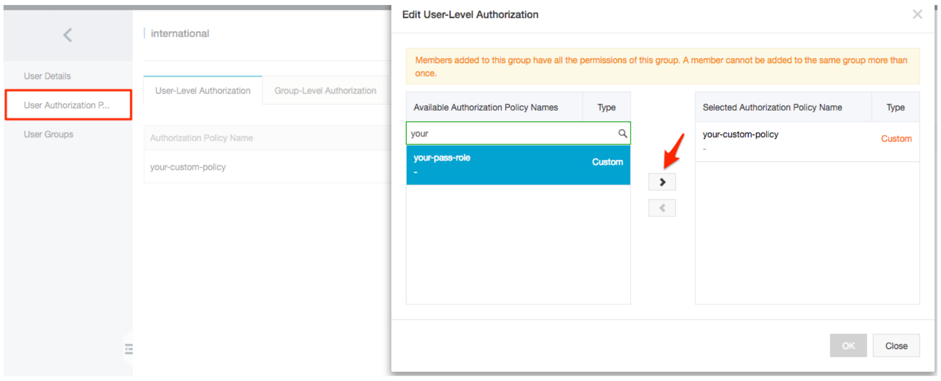
```
"Version": "1",
"Statement": [
{
"Effect": "Allow",
"Action": "fc:CreateService",
"Resource": "acs:fc:cn-shanghai:*:services/*"
},
{
"Effect": "Allow",
"Action": [
"fc:CreateFunction"
],
"Resource": [
"acs:fc:*:*:services/your-helloworld-fc/functions/*",
"acs:fc:*:*:services/your-helloworld-oss/functions/*"
]
},
{
"Effect": "Allow",
"Action": [
"fc:UpdateFunction"
],
"Resource": [
"acs:fc:*:*:services/your-helloworld-fc/functions/your-hello-world-fc"
]
}
]
}
```

See Permission definition for detailed permission granularity control.

Logging and Other Advanced Settings

Advanced settings are used to manage function logging and grant the permission of accessing other cloud services with service roles. PassRole permission is also required in addition to UpdateService permission. To create roles in advanced settings, you must have the create role permission. The following shows a policy for setting the PassRole permission.

```
{
"Statement": [
{
"Action": [
"ram:PassRole"
],
"Effect": "Allow",
"Resource": "*"
}
],
"Version": "1"
}
```



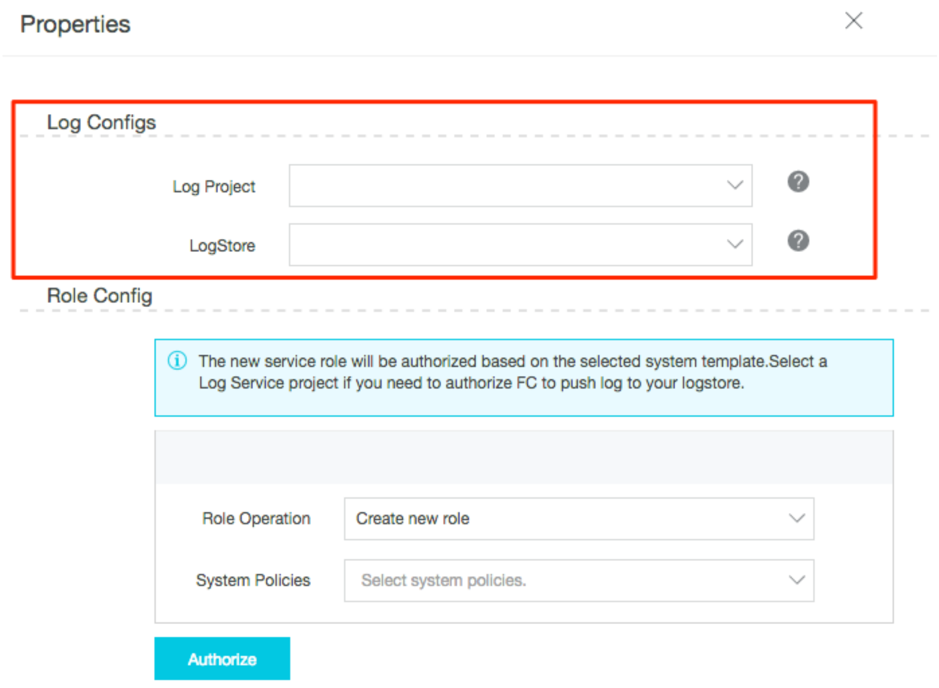
Edit User-Level Authorization

Members added to this group have all the permissions of this group. A member cannot be added to the same group more than once.

Available Authorization Policy Names	Type
your-pass-role	Custom

Selected Authorization Policy Name	Type
your-custom-policy	Custom

Log service project, logstore, and current role must be provided. It is recommended to add "list project", "list logstore" and "list role" to prevent manual errors. See Log Service permission management and RAM permission management for more details.



The new service role will be authorized based on the selected system template.Select a Log Service project if you need to authorize FC to push log to your logstore.

Role Operation: Create new role

System Policies: Select system policies.

The following is a template to help creating a policy to be attached to the RAM user.

```
{
"Version": "1",
"Statement": [
{
"Effect": "Allow",
"Action": [
"log:ListProject",
"log:ListLogStore"
],
"Resource": "acs:log:*:*:project/*"
},
{
"Effect": "Allow",
"Action": [
"ram:ListRoles"
],
"Resource": [
"acs:ram:*:*:role/*"
]
}
]
}
```

Set Permissions for RAM User for Trigger Functionalities

If you have attached PassRole and create trigger permissions, you can create a trigger. Only text boxes are provided if the RAM user does not have list bucket or list role permission. To use the drop-down selection list, you can use the following template to create a policy and attach it to the RAM user. See OSS permission management for more information.

```
{
"Statement": [
{
"Action": [
"oss:ListBucket"
],
"Effect": "Allow",
"Resource": "*"
}
],
"Version": "1"
}
```

Each role can be attach up to five custom policy templates. We recommend that you edit the statement to existing policy template to reduce the number of policies.

Note: If you successfully create a trigger but it is not displayed, you still need three permissions. The following is a template that can be used to create a policy and attach it with the RAM user.

```
{
"Statement": [
{
"Action": [
"oss:GetBucketEventNotification",
"oss:PutBucketEventNotification",
"oss:DeleteBucketEventNotification"
],
"Effect": "Allow",
"Resource": "*"
}
],
"Version": "1"
}
```

After completing previous steps, you have the required permissions to use console with the RAM user. Make sure you only select necessary permissions.

VPC Access

VPC Access

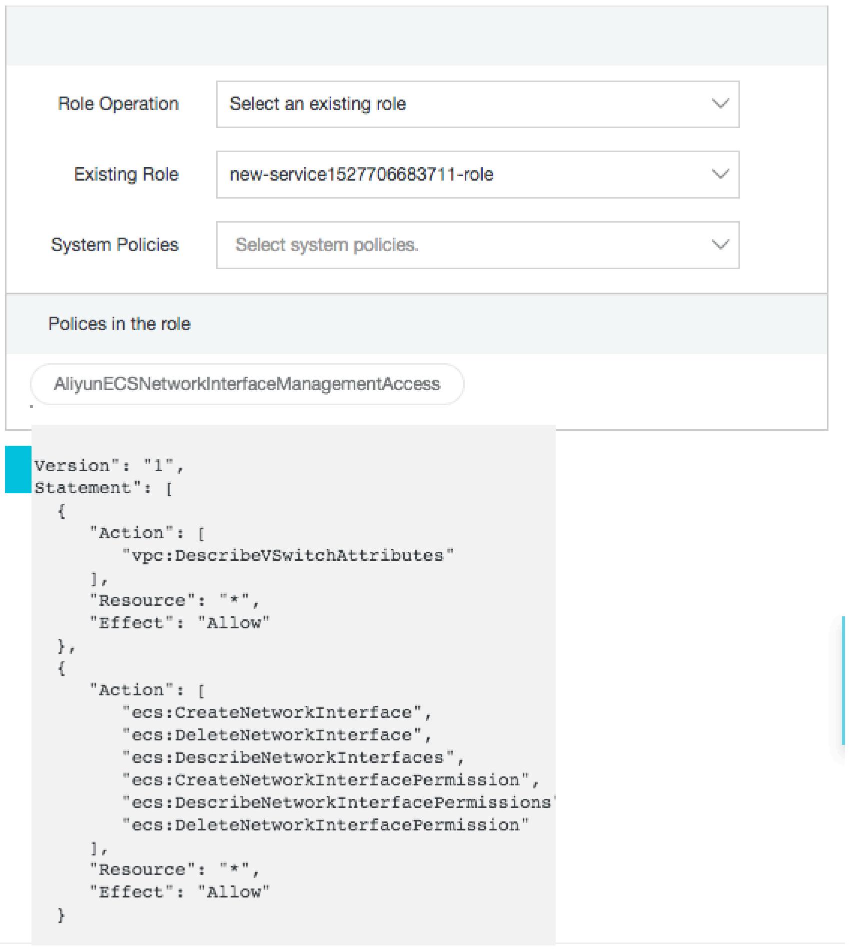
Virtual Private Cloud (VPC) is an isolated cloud network built for private usage. It allows you to logically isolate your cloud resources in a virtual network environment. All FC functions are running in the FC owned VPC network environment. By default, FC function cannot access your private VPC resources due to the nature of VPC network isolation.

Function Compute now seamless integrates with VPC. You can grant Function Compute permissions to manipulate elastic network interfaces (ENIs) and provide VPC-specific configuration information that includes VPC ID, vswitch IDs and security group ID. Function Compute will peer the function execution environment with the specific VPC by using the ENIs. Once VPC configuration is enabled, your function will run as if it is running inside the specific VPC.

VPC Configuraion

Permissions

You need to grant Function Compute ENI permissions in order to enable the VPC access. Function Compute obtains the permissions from the service role that you provide. You can grant Function Compute permissions by either creating a new service role with **AliyunECSNetworkInterfaceManagememtAccess** policy or attach this policy to your existing service role.



Role Operation: Select an existing role

Existing Role: new-service1527706683711-role

System Policies: Select system policies.

Polices in the role

AliyunECSNetworkInterfaceManagementAccess

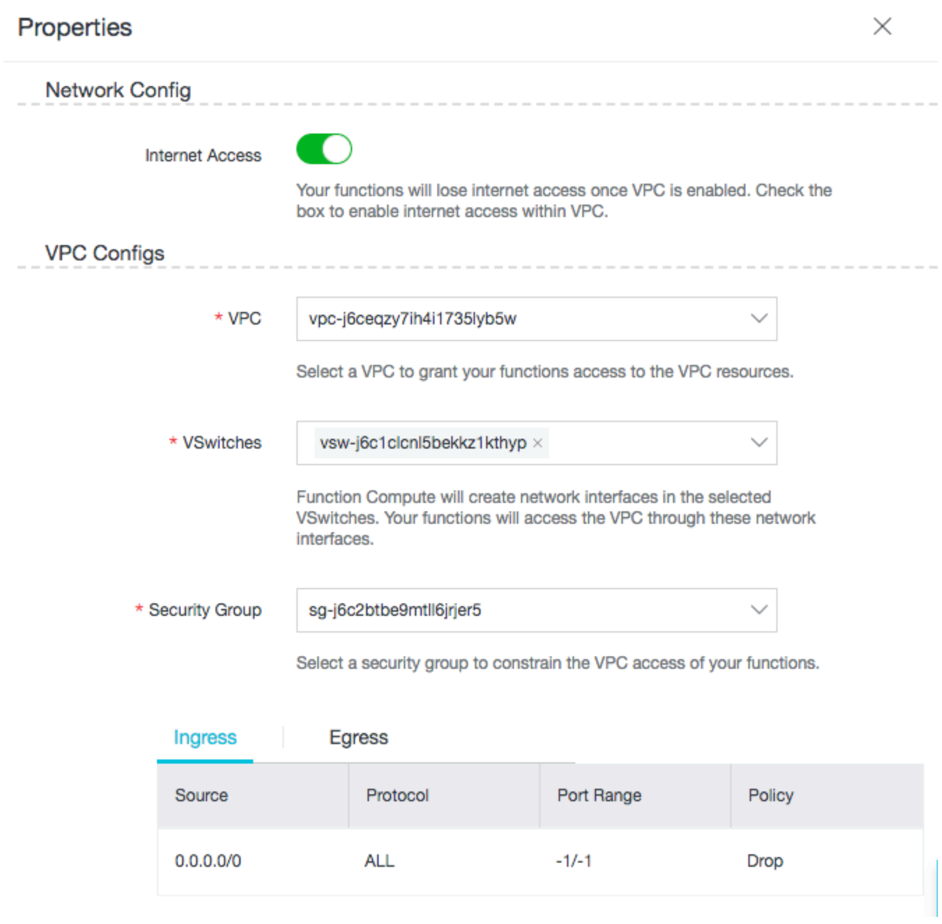
```
Version": "1",
Statement": [
  {
     "Action": [
        "vpc:DescribeVSwitchAttributes"
     ],
     "Resource": "*",
     "Effect": "Allow"
  },
  {
     "Action": [
        "ecs:CreateNetworkInterface",
        "ecs:DeleteNetworkInterface",
        "ecs:DescribeNetworkInterfaces",
        "ecs:CreateNetworkInterfacePermission",
        "ecs:DescribeNetworkInterfacePermissions'
        "ecs:DeleteNetworkInterfacePermission"
     ],
     "Resource": "*",
     "Effect": "Allow"
  }
```

VPC Information

You need to grant Funtion Compute VPC specific information, includes VPC ID, at least one VSwitch IDs and security group ID, in order to complete the set up. Function Compute will create ENI randomly in the provided VSwitches and uses that to access your specific VPC. We recommend that you provided at least one VSwitch from each availability zone so that your functions can still run in case that the availability zone is down or your VSwithc is running out of IP addresses.

Internet Access

Once VPC access is enabled, your function will run as if it is running inside your specific VPC where internet access may not be available. If your function needs both VPC access and internet access, your can set up a NAT to provide internet access for your specific VPC, or a more easy way is to enable internet access for your functions. Function Compute will setup a NAT and peers it with your function running environment. If you enable both VPC access and internet access for your functions, your functions can access your specific VPC through your ENIs and access internet through a FC NAT.



Properties

Network Config

Internet Access: on

Your functions will lose internet access once VPC is enabled. Check the box to enable internet access within VPC.

VPC Configs

* VPC: vpc-j6ceqzy7ih4i1735lyb5w

Select a VPC to grant your functions access to the VPC resources.

* VSwitches: vsw-j6c1clcnl5bekkz1kthyp

Function Compute will create network interfaces in the selected VSwitches. Your functions will access the VPC through these network interfaces.

* Security Group: sg-j6c2btbe9mtll6jrjer5

Select a security group to constrain the VPC access of your functions.

Source	Protocol	Port Range	Policy
0.0.0.0/0	ALL	-1/-1	Drop

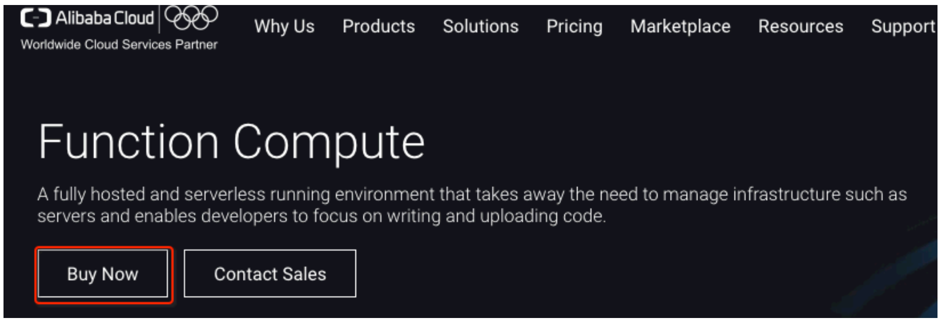
How to use the console

Alibaba Cloud Function Compute provides a complete console operation interface that simplifies most of your work through the interactive operations. You can see the following subtitles to use resources in the console, create your first service, function, and trigger, and view the Function Compute execution results, bills, and data monitoring results.

- Activate Function Compute
- View AccessKeys
- View service regions
- Create a service, function, and trigger
- Create a function trigger event
- View role authorization
- View execution results
- View data monitoring
- View bills

Activate Function Compute

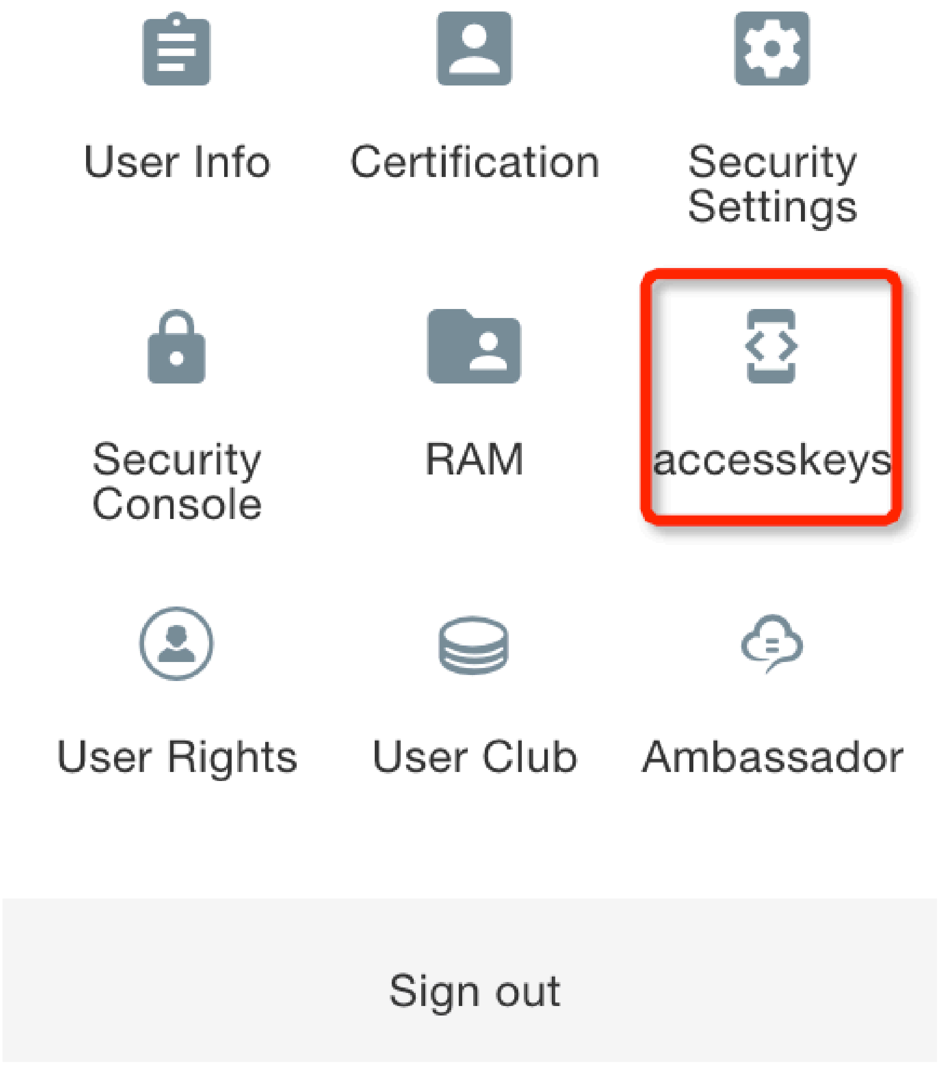
Go to the Alibaba Cloud Function Compute homepage and click **Activate Now**. If you have not logged on to the system yet, you are prompted to log on first. After successful logon before the activation page is displayed. Click **Activate Now** and **Management Console** to go to the console. If the console page is not displayed, click here.



A fully hosted and serverless running environment that takes away the need to manage infrastructure such as servers and enables developers to focus on writing and uploading code.

View AccessKeys

In the console, click your account in the upper-right corner and select **accesskeys** from the drop-down menu. Before the AccessKeys page is displayed, you are prompted to choose whether to continue with the AccessKey or use the AccessKey of the primary account or subaccount. Both AccessKeys can be used to normally access Function Compute. You can select one based on your business features.



If you do not have any AccessKey, click "Create Access Key".

Use your AccessKey if you already have one.

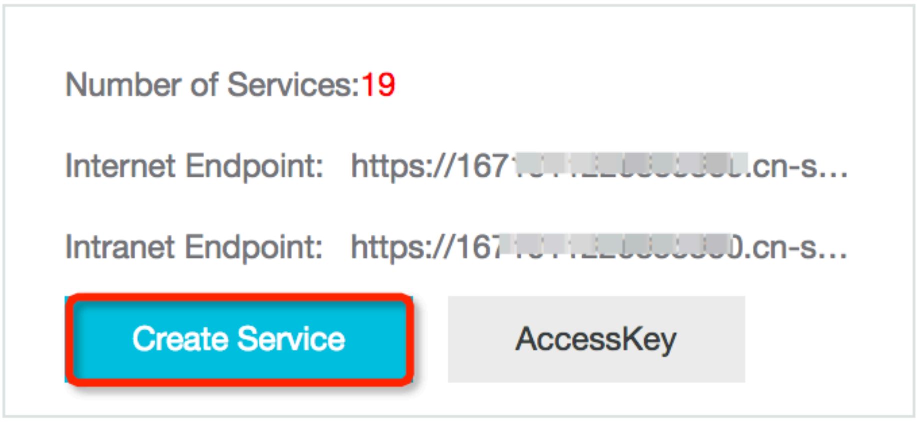
Note: You must enter the mobile phone number of the primary account and obtain the verification code to view the AccessKeySecret.

View service regions

For more information about the list of currently activated regions and corresponding region codes, see List of regions and codes.

Create a service

You can create a service in the Function Compute console. In the console, select **Function Compute** to go to the Function Compute homepage. Then, select a region and click **Create Service** in the upper-right corner.



Number of Services:19

Internet Endpoint: https://167[redacted].cn-s...

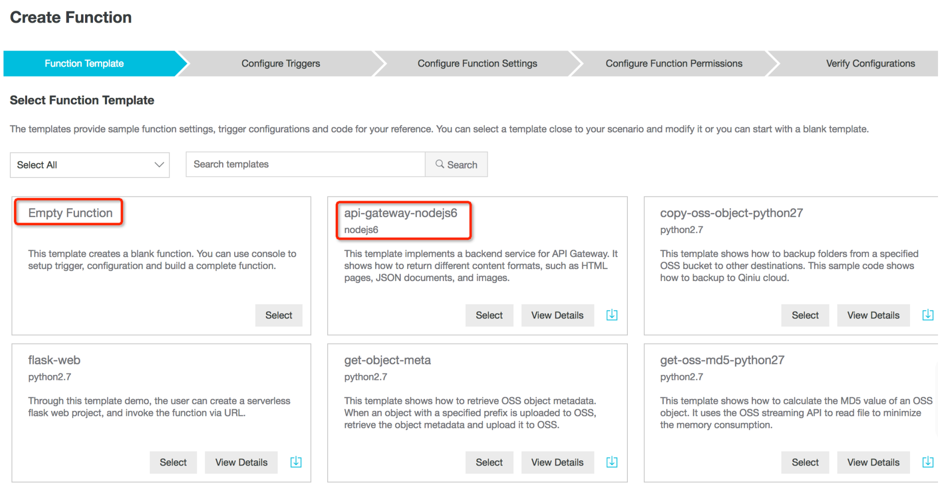
Intranet Endpoint: https://167[redacted]0.cn-s...

Note:

- After a service is successfully created, you can click **Advanced Settings** to configure log and role authorization.
- You can create unlimited number of services and a maximum of 50 functions for each service currently. For more information about the system restrictions, see Restrictions.

Create a function

You can create functions under a service. A function is the smallest code execution unit. Each function can have only one entry method but multiple other methods. A maximum of 10 triggers can be set for each function. A navigation page displays the function creation process, which includes three steps: selecting a template, configuring a trigger (optional), and configuring the function. Two templates are available currently. One is the blank function template that allows you to compile your own code. The other template contains a sample code, where you can compile your own business function using the sample code.



Create Function

Select Function Template

The templates provide sample function settings, trigger configurations and code for your reference. You can select a template close to your scenario and modify it or you can start with a blank template.

Empty Function

This template creates a blank function. You can use console to setup trigger, configuration and build a complete function.

api-gateway-nodejs6
nodejs6

This template implements a backend service for API Gateway. It shows how to return different content formats, such as HTML pages, JSON documents, and images.

copy-oss-object-python27
python2.7

This template shows how to backup folders from a specified OSS bucket to other destinations. This sample code shows how to backup to Qiniu cloud.

flask-web
python2.7

Through this template demo, the user can create a serverless flask web project, and invoke the function via URL.

get-object-meta
python2.7

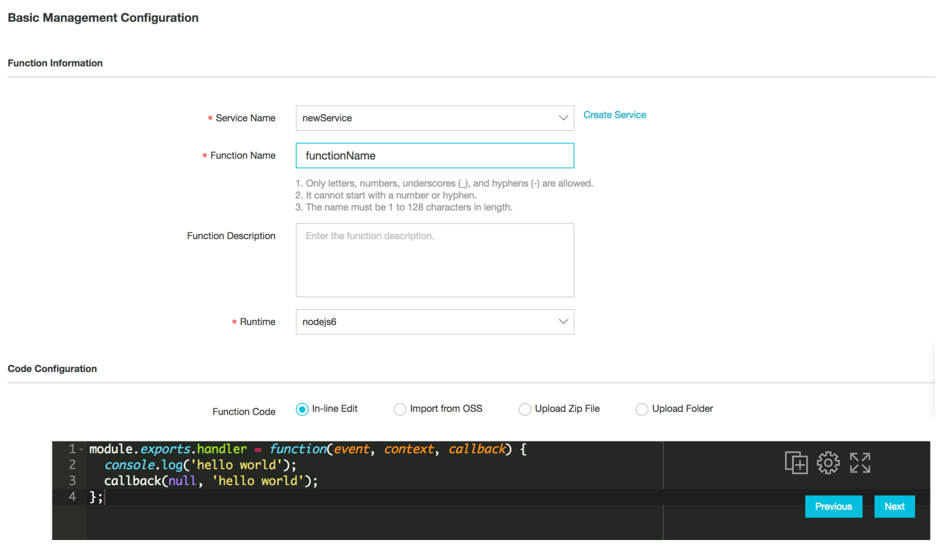
This template shows how to retrieve OSS object metadata. When an object with a specified prefix is uploaded to OSS, retrieve the object metadata and upload it to OSS.

get-oss-md5-python27
python2.7

This template shows how to calculate the MD5 value of an OSS object. It uses the OSS streaming API to read file to minimize the memory consumption.

Page for creating a function

Parameters with the red asterisk (*) are required, such as the function name, running environment, and code upload method.


Basic Management Configuration

Function Information

* Service Name	newService
* Function Name	functionName
Function Description	Enter the function description.
* Runtime	nodejs6

1. Only letters, numbers, underscores (_), and hyphens (-) are allowed.
2. It cannot start with a number or hyphen.
3. The name must be 1 to 128 characters in length.

Code Configuration

Function Code: [✓ In-line Edit] Import from OSS Upload Zip File Upload Folder

```
module.exports.handler = function(event, context, callback) {
  console.log('hello world');
  callback(null, 'hello world');
};
```

Precautions:

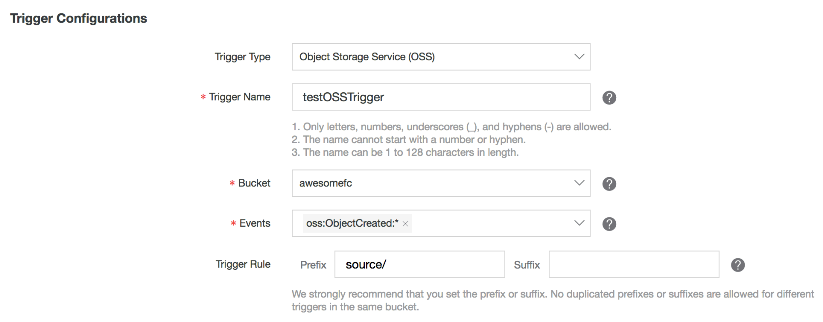
1. The function code entry varies with the language. For more information, click here.
2. You can set the entry function, memory size, and time-out for the function.
3. You can compile the code of a function online. Currently, you can compile codes online using Node.js or Python. If you use other languages, you can only upload codes through OSS or locally. You can upload a maximum of 5 MB codes locally. To upload codes greater than 5 MB, use the Command line tool or store the codes to OSS for execution.

Create a trigger

You can create a trigger on the navigation page when creating a function or after creating the function. The configuration varies with the trigger type. The following uses the OSS trigger as an example. Parameters with the asterisk (*) are required.

Note:

- When using the trigger for the first time, click **OK** on the displayed authorization page to authorize OSS to call Function Compute.
- If you have already configured a role, select it.
- The trigger events are stored in a Bucket. Therefore, you must first create a Bucket in the region of Function Compute in OSS.
- For trigger event execution, we recommend that you set the file prefix or suffix to prevent Function Compute from random execution.



Trigger Configurations

Trigger Type: Object Storage Service (OSS)

* Trigger Name: testOSSTrigger

1. Only letters, numbers, underscores (_), and hyphens (-) are allowed.
2. The name cannot start with a number or hyphen.
3. The name can be 1 to 128 characters in length.

* Bucket: awesomefc

* Events: oss:ObjectCreated:*

Trigger Rule: Prefix source/ Suffix

We strongly recommend that you set the prefix or suffix. No duplicated prefixes or suffixes are allowed for different triggers in the same bucket.

Create a function trigger event

A function trigger event is a tool used to simulate your actual request for function debugging. After you put a request string into an event source, the system uses this request string to run the function.

Note: The trigger event string is stored in the local browser cache. You must enter the string again if you clear the cached data or use a new browser.

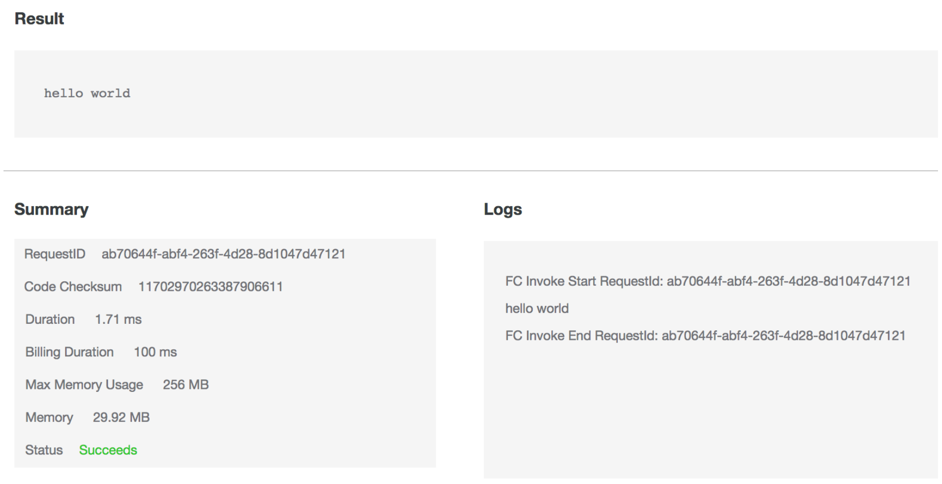
View role authorization

Function Compute supports two role authorization methods:

- You can configure role authorization in **Advanced Settings** of the service to **authorize Function Compute to access certain resources**. Generally, Function Compute can be authorized to access Log Service, OSS, and Table Store. Currently, only the Read-Only permission is granted. To grant more permissions, configure role authorization in RAM.
- You can also configure role authorization in **Advanced Settings** when creating the trigger to **authorize the trigger to trigger execution of Function Compute**. Generally, the trigger is authorized, and each trigger needs to be authorized only once.

View execution results

After a function is executed, the execution result and abstract of the execution process are displayed in the console. For example, the execution status, execution duration, actual billing time, configured memory size, actual used memory size, and other information are displayed. The execution log on the right displays the debugging information during the execution process.



Result

```
hello world
```

Summary

```
RequestID    ab70644f-abf4-263f-4d28-8d1047d47121
Code Checksum    11702970263387906611
Duration    1.71 ms
Billing Duration    100 ms
Max Memory Usage    256 MB
Memory    29.92 MB
Status    Succeeds
```

Logs

```
FC Invoke Start RequestId: ab70644f-abf4-263f-4d28-8d1047d47121
hello world
FC Invoke End RequestId: ab70644f-abf4-263f-4d28-8d1047d47121
```

Note:

If Log Service is disabled, the console displays only the latest 4 KB running logs. To view more logs, enable Log Service in **Advanced Settings** of the service.

View data monitoring

You can click **Real-time Service Monitoring** in the upper-right corner of the Function Compute console to view the function execution status in a period.

After logging on to CloudMonitor, you can view the execution status of each service in detail, such as the average latency and total number of calls, which are measured by service.

View bills

Select **Billing Center** > **Consumption Record** > **Consumption Details** > **Function Compute**. The billing items of Function Compute mainly include the resource usage, number of requests, and Internet outbound traffic. For more information about the billing information, see Product pricing. The bill is calculated by hour, which is one hour delayed. For example, the bill of data consumed from 01:00 to 02:00 is output at 04:00.