Serverless Cloud Function

Create Cloud Function

Product Introduction

Contents

Create Cloud Function Key Concepts

Last updated：2018-08-28 15:18:38

When you write code using a language supported by the SCF platform, you need a general pattern which includes the following key concepts:

Execution Method

The execution method determines where the SCF platform begins to execute your code. It is specified in the format of `file name.method name` by the user. When calling a cloud function, SCF executes your code by searching for the execution method. For example, if you specify `index.handler` as the execution method, the platform first locates the index file in the code package and then finds the handler method in the file to start execution.

When writing the execution method, you need to follow the platform-specific programming model, which specifies fixed input parameters: event data "event" and context data "context". The execution method should process the parameters and can call any other methods in the code.

Input Parameter event

SCF platform passes the event object to the execution method as the first parameter. With this event object, the code interacts with the event that triggers the function. For example, as the file upload triggers the function, the code can obtain all information of the file from the event parameter, including file name, download path, file type, size, etc.

Log

SCF platform stores all the function call records and the output in function codes in the log. Please use the specific log statement in programming languages to generate the output for debugging and troubleshooting.

Notes

Based on the features of SCF, you *must* write function code in a stateless style. The status features within a function lifecycle such as local file storage will be terminated with the function at the end of a function call. Therefore, persistent statuses should be stored in COS, Memcached or other cloud storage services.

Programming Model
Python

Last updated：2018-08-28 15:20:25

The following Python programming language versions are supported:

- Python 2.7
- Python 3.6

Function Form

The form of Python function is generally as follows:

```
import json

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

Execution Method

When creating an SCF, you need to specify the execution method. When using Python, the execution method is similar to `index.main_handler`, where `index` indicates that the entry file to be executed is `index.py` and `main_handler` indicates that the entry function to be executed is `main_handler` function. When submitting a code zip package using local zip file upload and COS upload, make sure that the root directory of the zip package contains the specified entry file and the file contains the defined entry function. File name and function name must be same with those entered in execution method to avoid execution failures caused by the inability to find the entry file and entry function.

Input Parameter

The input parameters in the Python environment include event and context, both of which are of Python dict type.

- event: Passes triggering event data.

- context: Passes runtime information to your processing program.

Return and Exception

Your processing program can return values using `return`. The processing method of returned values varies depending on the call type of the function.

- Synchronous call: In case of a synchronous call, the returned values are serialized and returned to the caller in JSON format. The caller can obtain the returned values for subsequent processing. For example, the call method for function debugging via the console is a synchronous call, which can capture and display the values returned by the function after the call is completed.
- Asynchronous call: For asynchronous call, because the call method returns a value just after the function is triggered and does not wait for the function to complete its execution, the returned value of the function will be discarded.

Regardless of synchronous or asynchronous calls, the returned values are displayed in `ret_msg` of the function log.

You can throw exceptions within a function using `raise Exception`. The thrown exceptions will be captured in the function running environment and displayed in the log in the form of `Traceback`.

Log

You can use `print` or the `logging` module in your program to accomplish log output, as shown in the following function:

```
import logging
logger = logging.getLogger()
logger.setLevel(logging.INFO)
def main_handler(event, context):
logger.info('got event{}'.format(event))
print("got event{}".format(event))
return 'Hello World!'
```

The output can be found in the `log` of the function log.

Included Libraries and Usage

COS SDK

Python SDK of COS (`cos_sdk_v4` version) is included in the SCF running environment.

The COS SDK can be introduced and used in the code as follows:

```
import qcloud_cos

from qcloud_cos import CosClient
from qcloud_cos import DownloadFileRequest
from qcloud_cos import UploadFileRequest
```

For more information on COS SDK, please see COS Python SDK.

Python 2 or 3?

When creating a function, you can select the desired running environment by choosing between `Python 2.7` or `Python 3.6`.

Python's official suggestions on selection of Python 2 or Python 3 can be found here.

Node.js

Last updated：2018-08-28 15:21:04

The following Node.js programming language versions are supported:

- Node.js 6.10
- Node.js 8.9

Function Form

The form of Node.js function is generally as follows:

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

Execution Method

When creating an SCF, you need to specify the execution method. When using Node.js, the execution method is similar to `index.main_handler`, where `index` indicates that the entry file to be executed is `index.js` and `main_handler` indicates that the entry function to be executed is `main_handler` function. When submitting a code zip package using local zip file upload and COS upload, make sure that the root directory of the zip package contains the specified entry file and the file contains the defined entry function. File name and function name must be same with those entered in execution method to avoid execution failures caused by the inability to find the entry file and entry function.

Input Parameter

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

- event: Passes triggering event data.
- context: Passes runtime information to your processing program.

- callback: Returns the information to the caller based on your needs. If this parameter is not specified, null is returned.

Return and Exception

The input parameter `callback` is required for your processing program to return information. The syntax for `callback` is:

```
callback(Error error, Object result);
```

Where:

- error: Optional. Returns the error message when the function execution fails internally. It can be set to null for successful executions.
- result: Feasible parameter. Returns the successful execution result of the function. The parameter needs to be compatible with JSON.stringify for JSON format serialization.

The processing methods of returned values vary depending on the call type of the function. The returned values of synchronous call will be returned to the caller in serialized JSON format, and the returned values of asynchronous call will be discarded. Regardless of synchronous or asynchronous calls, the returned values are displayed in `ret_msg` of the function log.

Log

You can use the following statement in the program to complete the log output:

- console.log()
- console.error()
- console.warn()
- console.info()

The output can be found in the `log` of the function log.

Included Libraries and Usage

COS SDK

Node.js SDK of COS (`cos-nodejs-sdk-v5` version) is included in the SCF running environment.

The COS SDK can be introduced and used in the code as follows:

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

For more information on COS SDK, please see COS Node.js SDK.

PHP

Last updated：2018-08-28 15:22:08

The following PHP programming language versions are supported:

- PHP 5.6
- PHP 7.2

Function Form

The form of PHP function is generally as follows:

```
<?php

function main_handler($event, $context) {
echo("hello world");
print_r($event);
return "hello world";
}

?>
```

Execution Method

When creating an SCF, you need to specify the execution method. When using PHP, the execution method is similar to `index.main_handler`, where `index` indicates that the entry file to be executed is `index.php` and `main_handler` indicates that the entry function to be executed is `main_handler` function. When submitting a code zip package using local zip file upload and COS upload, make sure that the root directory of the zip package contains the specified entry file and the file contains the defined entry function. File name and function name must be same with those entered in execution method to avoid execution failures caused by the inability to find the entry file and entry function.

Input Parameter

The input parameters in the PHP environment include $event and $context.

- $event: Passes triggering event data.

- $context: Passes runtime information to your processing program.

Return and Exception

Your processing program can return values using `return` . The processing method of returned values varies depending on the call type of the function.

- Synchronous call: In case of a synchronous call, the returned values are serialized and returned to the caller in JSON format. The caller can obtain the returned values for subsequent processing. For example, the call method for function debugging via the console is a synchronous call, which can capture and display the values returned by the function after the call is completed.
- Asynchronous call: For asynchronous call, because the call method returns a value just after the function is triggered and does not wait for the function to complete its execution, the returned value of the function will be discarded.

Regardless of synchronous or asynchronous calls, the returned values are displayed in `ret_msg` of the function log.

In the function, you can exit the function by calling die(). Then the function will be marked as failed and the log will also record the output when the function exits using die().

Log

You can use the following statement in the program to complete the log output:

- echo or echo()
- print or print()
- print_r()
- var_dump()

The output can be found in the `log` of the function log.

Java

Java

Last updated：2018-08-28 15:24:36

SCF provides Java8 runtime environment for Java runtime environment.

Because the Java language needs to be compiled before it can be run in JVM, its use method in SCF is different with such scripting languages as Python and Node.js. Here are the restrictions:

- Code upload is not supported: For the Java language, only developed, compiled and packaged zip/jar packages can be uploaded. SCF environment does not provide Java compiling capability.
- Online editing is not supported: Online code editing is not supported as code cannot be uploaded. For Java runtime function, you can only see the methods of "uploading again via page" or "COS code submitting" on the code page.

Code Form

The code form of SCF developed with Java is generally as follows:

```
package example;

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

Execution Method

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

Deployment Package Upload

You can create a zip or jar package via two methods: Create zip Deployment Package Using Gradle and Create jar Deployment Package Using Maven. After the package is created, you can directly upload it (less than 10 MB) via the console, or upload the deployment package to COS bucket and specify the Bucket and Object of the deployment package on SCF console to complete the deployment package submission.

Input Parameters and Returned Values

In the code example, the input parameters used by mainHandler contains two classes (String and Context), while returned values are String classes. The first input class identifies the input event, and the second one identifies the runtime information of the function. Event input and function return support Java base classes and POJO classes. The function runtime is of class `com.qcloud.scf.runtime.Context` and its associated library file can be downloaded here.

- Supported event input parameters and return parameter classes
  - Java base classes, including eight basic classes (byte, int, short, long, float, double, char, and boolen), wrapper classes and String class.
  - POJO (Plain Old Java Object) classes. You should use variable POJOs and public getters and setters to implement corresponding classes in your code.

- Context input parameter
  - To use Context, you need to use the import package `com.qcloud.scf.runtime.Context;` in the code and bring it into the jar package for packaging.
  - If you do not use this object, you can ignore it in the function input parameter, which can be written as `public String mainHandler(String name)` .

Log

You can use the following statement in the program to complete the log output:

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

The output can be found in the `log` of the function log.

POJO parameters

Last updated：2018-08-28 15:25:31

With POJO parameters, in addition to simple event input parameters, you can process more complex data structures. In this section, a set of examples will be used to illustrate how to use POJO parameters in SCF and which input parameters formats are supported.

Event Input Parameters and POJO

Suppose our event input parameters are as follows:

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

For the above input parameters, the output is as follows:

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

Based on the input parameters, we have constructed the following four classes:

- RequestClass: Used to accept events as an event accepting class
- PersonClass: Used to process the `person` field in the event JSON
- CityClass: Used to process the `city` field in the event JSON
- ResponseClass: Used to organize response content

Code Preparation

According to the four classes and entry functions designed for input parameters, follow the following steps to prepare.

Project directory preparation

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

Code directory preparation

Create the folder `src\main\java` under the project root directory as the code directory.

Based on the package name to be used, create a package folder in the code directory, such as `example`, to form the directory structure of `scf_example\src\main\java\example`.

Code Preparation

Create files `Pojo.java`, `RequestClass.java`, `PersonClass.java`, `CityClass.java` and `ResponseClass.java` in the folder "example". The content of the files are as follows:

- Pojo.java

```
package example;

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

- RequestClass.java

```
package example;


public class RequestClass {
PersonClass person;
CityClass city;

public PersonClass getPerson() {
return person;
}

public void setPerson(PersonClass person) {
this.person = person;
}

public CityClass getCity() {
return city;
}

public void setCity(CityClass city) {
```

```
this.city = city;
}

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

public RequestClass() {
}
}
```

- PersonClass.java

```
package example;

public class PersonClass {
String firstName;
String lastName;

public String getFirstName() {
return firstName;
}

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

public String getLastName() {
return lastName;
}

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

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

public PersonClass() {
}
}
```

- CityClass.java

```
package example;

public class CityClass {
String name;


public String getName() {
return name;
}

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

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

public CityClass() {
}
}
```

- ResponseClass.java

```
package example;


public class ResponseClass {
String greetings;

public String getGreetings() {
return greetings;
}

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

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

```
public ResponseClass() {
}
}
```

Code Compilation

In the example, we use Maven for compilation and packaging. You can choose the packaging method based on your needs.

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

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

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

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

```
</build>
</project>
```

Execute the command `mvn package` in the command line, and make sure that it has been successfully compiled, as shown below:

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

If the compilation fails, modify it according to the prompt.

The compiled package is located at `target\java-example-1.0-SNAPSHOT.jar` .

Creating and Testing SCF

According to the guidelines, create a SCF and upload the compiled package as a submission package. You can choose to use zip upload, or upload it to COS Bucket and then select "COS Bucket upload" for submission.

The execution method of SCF is configured as `example.Pojo::handle` .

Click the **test** button to go to the test interface, and then enter the input parameters that we initially want to process in the test template:

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

Click **Run** and you can see the returned content:

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

You can also modify the value in the structure of the test input parameter. After running the SCF, you can see the modification effect.

Create a zip Deployment Package using Gradle

Last updated：2018-08-28 15:26:33

Create a zip Deployment Package using Gradle

This section describes how to create a Java SCF deployment package using the Gradle tool. The created zip package conforming to the following rules can be identified and called by the SCF execution environment.

- The compiled package, class files and resource files are located under the root directory of the zip package.
- The jar package required for dependency is located under the /lib directory.

Environment Preparation

Make sure Java and Gradle have been installed. Install JDK8. You can use OpenJDK (Linux) or download and install the JDK appropriate for your system via www.java.com.

Gradle installation

For the specific installation method, please see https://gradle.org/install/. Here we describe the manual installation process:

1. Download Binary Package or Complete Package with Documentation and Source Code of Gradle.
2. Decompress the package to your desired directory, for example `C:\Gradle` (Windows) or `/opt/gradle/gradle-4.1` (Linux).
3. Add the path of the bin directory under the decompression directory to the system PATH environment variable. For Linux, add it using `export PATH=$PATH:/opt/gradle/gradle-4.1/bin` . For Windows, `right-click **Computer**, and select **Attribute** -> **Advanced System Settings** -> **Advanced** -> **Environment Variables**` to enter the environment variable settings page, and then select the `Path` variable and add `;C:\Gradle\bin;` at the end of the variable value.

4. Run `gradle -v` in the command line. If the following content shows, it indicates that Gradle has been installed successfully. For any questions, please see Gradle User Manual.

```
------------------------------------------------------------
Gradle 4.1
------------------------------------------------------------

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

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

Code Preparation

Prepare code file

Create a project folder in the selected location, for example `scf_example` . Under the root directory of project folder, create the directory `src/main/java/` for storing the package. Create the `example` package directory under the created directory, and create the `Hello.java` file under the package directory. The final directory structure is as follows:
`scf_example/src/main/java/example/Hello.java`

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

```
package example;

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

Prepare compilation file

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

```
apply plugin: 'java'

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

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

build.dependsOn buildZip
```

Process package dependency with Maven Central library

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

```
apply plugin: 'java'

repositories {
mavenCentral()
}

dependencies {
compile (
'com.qcloud:qcloud-java-sdk:2.0.1'
)
}

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

build.dependsOn buildZip
```

With mavenCentral specified in repositories as the dependent library source, Gradle will pull dependency from Maven Central in the compilation process, namely `com.qcloud:qcloud-scf-java-events:1.0.0` package specified in dependencies.

Process package dependency using local Jar package library

If you have downloaded the Jar package locally, you can use the local library to process package dependency. In this case, create `jars` directory under the root directory of project folder, and place the downloaded dependency Jar package under this directory. Write the `build.gradle` file as follows:

```
apply plugin: 'java'

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

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

build.dependsOn buildZip
```

Specify *.jar file under the jars directory as the searching directory via dependencies, and dependency will perform auto search in the compilation process.

Compiling and Packaging

Run the command `gradle build` under the root directory of the project folder. The compiling output should be similar to the following:

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

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

If compiling fails, adjust the code according to the output compiling error message.
The compiled zip package is located under the `/build/distributions` directory of the project folder and is named as `scf_example.zip` with the project folder name.

Function Use

For the generated zip package after compiling and packaging, you can choose the upload method based on the package size when creating or modifying the function. If the package is less than 10 MB, you can use page upload, otherwise you can upload the package to COS Bucket and then update it into the function via COS upload.

Create jar Deployment Package Using Maven

Last updated：2018-10-09 18:13:44

Create jar Deployment Package Using Maven

This section describes how to create a jar deployment package using the Maven tool for Java SCF.

Environment Preparation

Make sure Java and Maven have been installed. Install JDK8. You can use OpenJDK (Linux) or download and install the JDK appropriate for your system via www.java.com.

Maven installation

For the specific installation method, please see https://maven.apache.org/install.html. Here we describe the manual installation process:

1. Download zip Package or tar.gz Package of Maven.
2. Decompress the package to your desired directory, for example, `C:\Maven` (Windows) or `/opt/mvn/apache-maven-3.5.0` (Linux).
3. Add the path of the bin directory under the decompression directory to the system PATH environment variable. For Linux, add it using `export PATH=$PATH:/opt/mvn/apache-maven-3.5.0/bin` . For Windows, right-click **Computer**, and select **Attribute** -> **Advanced System Settings** -> **Advanced** -> **Environment Variables** to enter the environment variable settings page, and then select the `Path` variable and add `;C:\Maven\bin;` at the end of the variable value.
4. Run `mvn -v` in the command line. If the following content shows, it indicates that Maven has been installed successfully. For any questions, please see Installing Apache Maven.

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

Code Preparation

Prepare code file

Create a project folder in the selected location, for example `scf_example` . Under the root directory of project folder, create the directory `src/main/java/` for storing the package. Create the `example` package directory under the created directory, and create the `Hello.java` file under the package directory. The final directory structure is as follows:
`scf_example/src/main/java/example/Hello.java`

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

```
package example;

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

Prepare compilation file

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

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

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

<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-shade-plugin</artifactId>
<version>2.3</version>
<configuration>
```

```
<createDependencyReducedPom>false</createDependencyReducedPom>
</configuration>
<executions>
<execution>
<phase>package</phase>
<goals>
<goal>shade</goal>
</goals>
</execution>
</executions>
</plugin>
</plugins>
</build>
</project>
```

Process package dependency with Maven Central library

If you need to reference external package of Maven Central, you can add dependency as needed. The content of the `pom.xml` file is written as follows:

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

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

<dependencies>
<dependency>
<groupId>com.qcloud</groupId>
<artifactId>qcloud-java-sdk</artifactId>
<version>2.0.1</version>
</dependency>
</dependencies>


<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
```

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

Compiling and Packaging

Run the command `mvn package` under the root directory of the project folder. The compiling output should be similar to the following:

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

If compiling fails, adjust the code according to the output compiling error message.

The compiled jar package is located under the `target` directory of the project folder and is named as `java-example-1.0-SNAPSHOT.jar` based on the fields artifactId and version of pom.xml.

Function Use

For the generated jar package after compiling and packaging, you can choose the upload method based on the package size when creating or modifying the function. If the package is less than 10 MB, you can use page upload, otherwise you can upload the package to COS Bucket and then update it into the function via COS upload.

How to Create SCF

Last updated：2018-08-28 15:29:26

The function code is the most important part of SCF. Users upload the application code in the form of *SCF* to Tencent Cloud SCF platform, to allow *SCF* to run the code on behalf of users and perform all relevant server management work.

The lifecycle of an SCF-based application generally includes: writing codes, creating an SCF, deploying the SCF to SCF platform, testing, monitoring and troubleshooting, etc. This section describes everything about the function code. For more information about monitoring and troubleshooting, please see Monitoring SCFs and Their Logs.

Writing Codes for SCFs

You can only use the languages supported on SCF platform to write cloud function codes. You can use any coding tool, such as SCF console, local editor, and local IDE. Note: If other dependent libraries not introduced to the platform are introduced to the code, you **must** upload these dependent libraries. The dependent libraries provided by the platform can be found in Execution Environment section. For more information on how to upload the code, please see Create Deployment Packages section.

Meanwhile, SCF platform provides a set of basic patterns for writing function. For example, how to determine a preferred method to call function, how to obtain information from parameters, how to output logs, how to interact with the current running environment, etc. For more information about function patterns, please see Writing a Processing Method section.

Creating a Deployment Package

You need to provide codes or deployment packages:

- If standard Python library and the library provided by Tencent Cloud (such as Python SDK of various cloud products) are used in your code, you just need to provide the code in the console, so that SCF console would automatically package this code file and upload it to SCF console.

- If you need to introduce external libraries, you can organize your codes and dependencies according to the specific method in Create Deployment Packages, package and upload them to SCF platform.

Creating and Deploying SCFs

You can create SCFs using SCF console, API, SDK or Tencent Cloud CLI tool. You first need to provide the configuration information of SCFs. including computing resources, running environment, etc. For more information, please see Create SCFs.

Testing SCFs

You can test SCFs using the following methods:

- Click **Test** in the console to test SCFs.
- Test SCFs using API, SDK or InvokeFunction method of Tencent Cloud CLI tool.

The calling data is required during test period. You can pass the calling data (such as COS, etc.) of specific cloud products to test whether the function responses to the events generated by these cloud products as expected. For more information about the event data generated by different cloud products, please see Manage SCF Triggers section.

Monitoring and Troubleshooting

After an SCF is introduced into the production environment, Tencent Cloud SCF automatically monitors its running status. Then, SCF metrics are uploaded to Cloud Monitor platform, so that users can check its running status.

To help you debug and troubleshoot, Tencent Cloud SCF platform records all calling and processing results of this function, and stores the output generated in the code in log format. For more information, please see Function Log section.

Example of SCF-based Application

Make sure to read and practice the examples in the following sections before using the cloud function:

- Getting Started: If it is your first time to use Tencent Cloud SCF, please read Getting Started section and try to perform all the operations in this section.
- Practical Operation of Code: If you need to introduce external libraries, you must create a code package in the local environment and upload it to SCF platform. Read and practice using each step in the example based on the selected programming language and the event to be processed.

Write Code for SCF
Writing Processing Method

Last updated：2018-08-28 15:41:47

When you write SCF code, the first and foremost step is to write a method which is executed first when the SCF platform calls a function. To create a method, a common syntax structure should be followed:

```
def method_name(event,context):
...
return some_value
```

The method of all functions receives fixed input parameters: event and context. Do not delete any fixed input parameter.

Getting Details of the Input Event from the event Parameter

SCF uses event parameter to pass event data to the function. This parameter is a `Python dict` parameter.

The user first needs to clarify what the function is for. For responding to the event trigger request from a cloud service (for example, uploading a file via COS to trigger the function)? Or for being called by other applications (for example, implementing a common module)? Or it does not require any input?

In different cases, the value of event is different:

- If the function is triggered by a cloud service, the cloud service will pass the event in an unchangeable format predefined by the platform as the event parameter to the SCF. The user can write code in this format to get necessary information from the event parameter. (For example, when COS triggers the function, the details of Bucket and the files will be passed in json format to the event parameter)

- If the cloud function is called by other applications, you can freely define a parameter of dict type between the caller and the function code. The caller passes the data in the agreed format, which is then obtained by the function code. For example, the agreed data structure of dict type is `{"key":"XXX"}` . When the caller passes the data `{"key":"abctest"}` , the function code can get the value `abctest` through `event[key]` .

- If the cloud function does not require any input, you can ignore event and context parameters in the code.

(Optional) Returned Value

Returned value is the result returned after a user uses the `return` statement. The method of returned value is different depending on the call type of the function. For more information on the call types of functions, please see Key Concepts:

- In case of a synchronous function call, SCF will return the value of the `return` statement in the code to the caller. For example, you can call the function synchronously by clicking **Test** on the Tencent Cloud console. So when you use the console to call the function, the console will display the returned value. If nothing is returned in the code, a null value is returned.
- In case of an asynchronous function call, the value is discarded.

For example, consider the following Python sample code.

```
def handler(event, context):
message = 'Hello {} {}!'.format(event['first_name'],
event['last_name'])
return {
'message' : message
}
```

The code receives the input event from the `event` parameter and returns a message containing data.

Create a SCF, paste the code above and set the execution method to `index.handler`. After creation, click the **Test** button and run it to see the returned message. For more information on how to create a function, please see Step 1: Create Hello World function.

Use Log Statement When Writing Code

Last updated：2018-08-28 15:43:40

Log statement provides necessary execution information for functions, which is indispensable to the troubleshooting of codes. SCF platform writes all the logs generated when the user uses log statement in code to the log system. If you use the console to call a function, the console displays the same log.

Users can use the following statement to generate log entry:

- print
- Logger function in logging module

Using logging Statement to Write to Log

```
import logging
logger = logging.getLogger()
def my_logging_handler(event):
logger.info('got event{}'.format(event))
logger.error('something went wrong')
return 'Hello World!'
```

In the above code, the logging module is used to write information to the log. You can view the log information in the code via the log module on the console or the API "Obtain Function Operation Log". Log level identifies the type of log, such as `INFO`, `ERROR` and `DEBUG`.

Using print Statement to Write to Log

You can also use print statement in code, as shown below:

```
def print_handler(event):
print('this will show up in logging')
return 'Hello World!'
```

When you call the function synchronously using the **Test** button on the console, the values of print statement and return statement will display on the console.

Obtaining Log

You can obtain the function operation log using the following methods

- If the function is called synchronously via the **Test** button on the console
  - The log for this call is displayed on the console after the execution
- If the function is called by the trigger
  - The log for each call is displayed in the Log tab of the function
  - The function log can also be obtained through the API GetFunctionLogs
- If the function is called synchronously through the API Invoke
  - The log for this call can be obtained in the logMsg field of returned value

Error Processing

Last updated：2018-08-28 15:44:16

If an exception occurs during the debugging or running of a function, Tencent Cloud SCF platform will catch the exception as far as possible and write the exception information to log. The exception generated during the running of a function includes handled error and unhandled error. For example, users can explicitly throw an exception in code:

```
def always_failed_handler(event,context):
raise Exception('I failed!')
```

This function throws an exception in running process and returns the following error message:

```
File "/var/user/index.py", line 2, in always_failed_handler
raise Exception('I failed!')
Exception: I failed!
```

SCF platform will write this error message to the function log.

If you need to test this code, create a function and copy the function code without adding any trigger. Click the **Test** button on the console and select the "Hello World" example to test.

You can define how to deal with potential errors in your code to guarantee your application's robustness and scalability. For example: Inherit Exception class

```
class UserNameAlreadyExistsException(Exception): pass

def create_user(event):
raise UserNameAlreadyExistsException('The username already exists,please change a name!')
```

Or use the Try statement to catch the error:

```
def create_user(event):
try:
createUser(event[username],event[pwd])
except UserNameAlreadyExistsException,e:
//catch error and do something
```

When the code logic of the user does not catch the error, SCF will catch the error as far as possible. But for any error that cannot be caught by the platform, for example, the user function crashes and exits suddenly in the running process, the system will return a common error message.

The following list provides some common errors during code running

Error Scenario	Returned Message
Throw an exception with raise	{File "/var/user/index.py", line 2, in always_failed_handler raise Exception('xxx') Exception: xxx}
The method does not exist	{'module' object has no attribute 'xxx'}
The Dependency module does not exist	{global name 'xxx' is not defined}
Timeout	{"time out"}

Create Deployment Package

Last updated：2018-08-28 15:44:48

A deployment package is a zip file into which all codes and dependencies that are running in the SCF platform are compressed. You need to specify a deployment package when creating a function. You can create a deployment package in local environment and upload it to the SCF platform, or write code directly in the SCF console, so that the console can create and upload a package for you. Determine whether you can use the console to create a deployment package according to the following conditions:

- Simple scenario: If standard Python library and SDK library (such as COS and SCF) provided by Tencent Cloud are required to write custom code and when there is only one .py file, you can use the inline editor of the SCF console. The console can automatically compress the code and relevant configuration information into an executable deployment package.
- Advanced scenario: If other resources (such as graphic database for graphic processing, Web framework for Web programming) are required to write code, you need to create a function deployment package in local environment, and upload the package through the console.

The following example shows how to create a deployment package in local environment.

> Notes:
> 1. Generally, the dependent library installed locally can run well in the SCF platform. However, the inst alled binary file may be incompatible in a few scenarios. If such a problem is found, please [contact u s](https://cloud.tencent.com/document/product/583/9712).
> 2. In the example for Python, libraries and dependencies are installed using pip in local environment, so make sure that you have installed Python and pip locally.

Creating a Python Deployment Package in Linux

1) Create a directory:

```
mkdir /data/my-first-scf
```

2) Store all Python source files (.py files) required to create this function into this directory. For more information on how to create a function, please see the section Getting Started - Create DownloadImage Function.

3) Install all dependencies in this directory using pip:

```
pip install <module-name> -t /data/my-first-scf
```

For example, you can install Pillow library in my-first-scf directory by executing the following command:

```
pip install Pillow -t /data/my-first-scf
```

4) Under my-first-scf directory, compress all the files. Note: What you need to compress is the content in the directory instead of the directory:

```
zip my_first_scf.zip /data/my-first-scf/*
```

Notes:
1. For libraries needing compilation, it is recommended to compress the files in CentOS 7 on which SCFs run.
2. If you have other requirements on software, compilation environment, or development library during the installation or compilation process, follow the installation prompts.

Creating a Python Deployment Package in Windows

We recommend that you compress the dependencies and codes that run successfully in Linux environment into a zip file as the function execution code. For more information, please see Practical Operation of Code - Obtain Images on COS and Create Thumbnails.

For Windows, you can also use the `pip install <module-name> -t <code-store-path>` command to install the Python library. But for packages that need to be compiled or carry static and dynamic libraries, only libraries completely implemented in Python can be installed in Windows, because libraries compiled in Windows cannot be invoked to run in SCF's running environment (CentOS 7).

Create SCF

Last updated：2018-08-28 15:45:14

You can create a cloud function by packaging application service codes and relevant dependencies and uploading them to Tencent Cloud SCF. The cloud function contains the codes and dependencies you uploaded, as well as some configuration information associated with the execution of function. This document describes the configurations of cloud functions and their meanings, to help you understand how to create an cloud function that fits your business needs.

Function Name

Tencent Cloud uses function name to exclusively identify the SCF in each of user's regions. The function name cannot be modified after creation. The function name should follow the rules below:

- The length is limited to 60 characters.
- It can only contain `a-z, A-Z, 0-9, -, _` .
- It must begin with a letter.

Region

You can specify the region in which you need to run the SCF. Beijing, Shanghai and Guangzhou are supported. The SCF is automatically deployed with high availability in multiple availability zones of a region. The region attribute cannot be changed after the SCF is created.

Computing Resources

Tencent Cloud allows you to customize the *amount of memory* allocated to SCF. Take the increment of 128 MB to allocate the amount of memory between `128 MB - 1536 MB` . Tencent Cloud automatically assigns the proportional CPU processing capacity for SCF based on the amount of memory you specified. For example, if 1024 MB memory is allocated to SCF, the CPU capacity obtained by the SCF is *twice* the allocated 512 MB memory. Therefore, in regular scenarios, the time to actually run the code is generally shortened by increasing the amount of memory of SCF.

You can change the amount of memory needed by SCF at any time. We strongly recommend that you increase the size of this parameter if you find the memory consumed in running SCF is nearly or reaches the configured memory amount, to prevent error in SCF due to OOM:

- Log in to the Tencent Cloud console, and click **SCF**.
- Select the function whose memory amount needs to be modified.
- Click **Edit** in the **Function Configuration** tab, select the memory amount to be adjusted and then click **Save**.

Timeout

Cloud functions are charged based on running time and the number of requests. To prevent cloud functions from running for an indefinite period (for example, infinite loop occurs in the code), each cloud function has a user-defined timeout value. The maximum value is 300 seconds. Default is 3 seconds. When timeout is reached, if a cloud function is still running, SCF platform will automatically terminate it.

You can change the timeout of the cloud function at any time. We strongly recommend that you increase the size of this parameter if you find the function times out or the actual running time is close to the configured timeout during test period, so as to prevent interruption of function which is limited by timeout during long running time:

- Log in to the Tencent Cloud console, and click **SCF**.
- Select the function whose timeout needs to be changed.
- Click **Edit** in the **Function Configuration** tab, select the timeout value to be adjusted and then click **Save**.

Description

You can configure the description information for the function, and change it at any time.