无服务器云函数

开发指南

产品文档

腾讯云

文档目录

开发指南
- 基本概念
  - 开发语言
    - Python
    - Node.js
    - Golang
    - PHP
      - Java
        - Java 说明
        - POJO 类型参数使用示例
        - 使用 Gradle 创建 zip 部署包
        - 使用 Maven 创建 jar 部署包
- 开发流程
  - 编写代码
    - 处理方法
    - 日志介绍
    - 错误类型
- 创建部署程序包
- 创建云函数

开发指南
基本概念

最近更新时间：2018-08-28 15:18:38

用户在使用 SCF 平台支持的语言编写代码时，需要采用一个通用的范式，包含以下核心概念：

执行方法

执行方法决定了 SCF 平台从何处开始执行您的代码，以 文件名.方法名 的形式被用户指定。SCF 在调用云函数时，将通过寻找执行方法来开始执行您的代码。例如，用户指定的执行方法为 index.handler ，则平台会首先寻找代码程序包中的 index 文件，并找到该文件中的 handler 方法开始执行。

用户在编写执行方法时需遵循平台特定的编程模型，该模型中指定固定的入参：事件数据 event 和环境数据 context。执行方法应该对参数进行处理，并且可任意调用代码中的任何其他方法。

函数入参

函数入参，是指函数在被触发调用时，所传递给函数的内容。通常情况下，函数入参包括两块：event 入参与 context 入参。根据开发语言和环境的不同，入参个数可能有所不同，不同开发语言在入参上的具体差异，可见 开发语言说明。

> **注意：**
> 为保证针对各开发语言和环境的统一性，event 入参和 context 入参都是使用 JSON 数据格式统一封装。

event 入参

SCF 平台将 event 入参传递给执行方法，通过此 event 入参对象，代码将与触发函数的事件（event）交互。
例如：由于文件上传触发了函数运行，代码可从 event 参数中获取该文件的所有信息，包括文件名、下载路径、文件类型、大小等。
不同触发器在触发函数时，所传递的数据结构均有所不同，具体数据结构可见 函数触发器说明。同时在通过云 API 方法触发云函数时，用户可自行定义传递给云函数的入参。

context 入参

SCF 平台将 context 入参传递给执行方法，通过此 context 入参对象，代码将能了解到运行环境及当前请求的相关内容。当前 context 内容如下：

```
{
"time_limit_in_ms": 3000,
"request_id": "627466b4-8049-11e8-8758-5254005d5fdb",
"memory_limit_in_mb": 512
}
```

其中包括了当前调用的执行超时时间，内存限制，以及当次请求 ID。

> **注意：**
> context 结构内容将可能随着 SCF 平台的开发迭代而增加更多内容。

函数返回

云函数执行完成后的返回值，会由 SCF 平台获取到，并根据不同的触发方式进行处理。

- 同步触发：通过同步方式触发的云函数，在函数执行期间，请求不会返回。在函数执行完成后，会将函数返回值封装为 JSON 格式以后返回给调用方。通过 API 网关、云 API 中 RequestResponse 方式触发的方式为同步触发。
- 异步触发：通过异步方式触发的云函数，在触发事件由 SCF 平台接收后，触发请求就会返回。在函数执行完成后，函数的返回值会封装为 JSON 格式后存储在日志中。如果用户需要在异步触发后获取到函数返回值，可通过记录请求返回中的 requestId，并在函数执行完成后，通过 requestId 查询日志，获取到此次执行的函数返回值。

> **注意：**
> 为保证针对各开发语言和环境的统一性，函数返回会使用 JSON 数据格式统一封装。

日志

SCF 平台会将函数调用的所有记录及函数代码中的输出全部存储在日志中，请使用编程语言中的打印输出语句或日志语句生成输出日志，以便作调试及故障排除之用。

在函数运行日志中，包括了函数名、启动时间、执行时间、计费时间、内存实际用量、返回码、返回值、代码日志、执行状态信息。

函数的运行日志通常数据结构如下所示：

```
{
"functionName": "testnode",
"retMsg": "\"ok\"",
"requestId": "b33b9d0b-9c51-11e7-b38f-525400c7c826",
"startTime": "2017-09-18 17:13:57",
"retCode": 0,
"invokeFinished": 1,
"duration": 7.437,
"billDuration": 100,
"memUsage": 131072,
"log": "2017-09-18T09:13:57.155Z\tundefined\tHello World\n2017-09-18T09:13:57.156Z\tundefined\t{ Records: [ { CMQ: [Object] } ] }\n2017-09-18T09:13:57.158Z\tundefined\t{ msgBody: '3',\n msgId: '3096224743817223',\n msgTag: '',\n publishTime: '2017-09-18T17:13:57Z',\n requestId: '5761047512720426853',\n subscriptionName: 'test',\n topicName: 'test',\n topicOwner: 1251762227,\n type: 'topic' }\n2017-09-18T09:13:57.159Z\tundefined\t{ callbackWaitsForEmptyEventLoop: [Getter/Setter],\n done: [Function: done],\n succeed: [Function: succeed],\n fail: [Function: fail],\n memory_limit_in_mb: 128,\n time_limit_in_ms: 30000 }\n"
}
```


注意事项

由于无服务器云函数的特点，**必须** 以无状态的风格编写您的函数代码。本地文件存储等函数生命周期内的状态特征在函数调用结束后将随之销毁。因此，持久状态建议存储在关系型数据库 CDB、对象存储 COS、云数据库 Memcached 或其他云存储服务中。

开发语言
Python

最近更新时间：2018-08-28 15:20:25

目前支持的 Python 开发语言包括如下版本：

- Python 2.7
- Python 3.6

函数形态

Python 函数形态一般如下所示：

```
import json

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

执行方法

在创建 SCF 云函数时，均需要指定执行方法。使用 Python 开发语言时，执行方法类似 `index.main_handler` ，此处 `index` 表示执行的入口文件为 `index.py` ， `main_handler` 表示执行的入口函数为 `main_handler` 函数。在使用 本地 zip 文件上传、COS 上传等方法提交代码 zip 包时，请确认 zip 包的根目录下包含有指定的入口文件，文件内有定义指定的入口函数，文件名和函数名和执行方法处填写的能够对应，避免因为无法查找到入口文件和入口函数导致的执行失败。

入参

Python 环境下的入参包括 event 和 context，两者均为 Python dict 类型。

- event：使用此参数传递触发事件数据。
- context：使用此参数向您的处理程序传递运行时信息。

返回和异常

您的处理程序可以使用 `return` 来返回值，根据调用函数时的调用类型不同，返回值会有不同的处理方式。

- 同步调用：使用同步调用时，返回值会序列化后以 JSON 的格式返回给调用方，调用方可以获取返回值已进行后续处理。例如通过控制台进行的函数调试的调用方法就是同步调用，能够在调用完成后捕捉到函数返回值并显示。
- 异步调用：异步调用时，由于调用方法仅触发函数就返回，不会等待函数完成执行，因此函数返回值会被丢弃。

同时，无论同步调用还是异步调用，返回值均会在函数日志中 `ret_msg` 位置显示。

您可以在函数内使用 `raise Exception` 的方式抛出异常。抛出的异常会在函数运行环境中被捕捉到并在日志中以 `Traceback` 的形式展示。

日志

您可以在程序中使用 `print` 或使用 `logging` 模块来完成日志输出。例如如下函数：

```
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!'
```

输出内容您可以在函数日志中的 `log` 位置查看。

已包含的库及使用方法

COS SDK

云函数的运行环境内已包含 COS 的 Python SDK，具体版本为 `cos_sdk_v4`。

可在代码内通过如下方式引入 COS SDK 并使用：

```
import qcloud_cos
```

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

更详细的 COS SDK 使用说明见COS Python SDK 说明。

Python 2 或 3 ?

您可以在函数创建时，通过选择运行环境中的 `Python 2.7` 或 `Python 3.6` 选择您所期望使用的运行环境。

您可以在这里查看 Python 官方对 Python 2 或 Python 3 语言选择的建议。

Node.js

最近更新时间：2018-08-28 15:21:04

目前支持的 Node.js 开发语言包括如下版本：

- Node.js 6.10
- Node.js 8.9

函数形态

Node.js 函数形态一般如下所示：

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

执行方法

在创建 SCF 云函数时，均需要指定执行方法。使用 Node.js 开发语言时，执行方法类似 `index.main_handler` ，此处 `index` 表示执行的入口文件为 `index.js` ， `main_handler` 表示执行的入口函数为 `main_handler` 函数。在使用 本地 zip 文件上传、COS 上传等方法提交代码 zip 包时，请确认 zip 包的根目录下包含有指定的入口文件，文件内有定义指定的入口函数，文件名和函数名和执行方法处填写的能够对应，避免因为无法查找到入口文件和入口函数导致的执行失败。

入参

Node.js 环境下的入参包括 event 、context 和 callback，其中 callback为可选参数。

- event：使用此参数传递触发事件数据。
- context：使用此参数向您的处理程序传递运行时信息。
- callback：使用此参数用于将您所希望的信息返回给调用方。如果没有此参数值，返回值为null。

返回和异常

您的处理程序需要使用 `callback` 入参来返回信息。 `callback` 的语法为：

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

其中：

- error：可选参数，在函数执行内部失败时使用此参数返回错误内容。成功情况下可设置为null。
- result：可选参数，使用此参数返回函数成功的执行结果信息。参数需兼容 JSON.stringify 以便序列化为 JSON 格式。

如果在代码中未调用 callback，云函数后台将会隐式调用，并且返回 null。

根据调用函数时的调用类型不同，返回值会有不同的处理方式。同步调用的返回值将会序列化为 JSON 格式后返回给调用方，异步调用的返回值将会被抛弃。同时，无论同步调用还是异步调用，返回值均会在函数日志中 `ret_msg` 位置显示。

Node.js 事件循环

由于 Node.js 大量采用了异步事件循环的方式处理回调，在云函数中运行 Node.js 代码时，同样支持异步事件。在入口函数中调用 callback 后，云函数后台会等待事件队列为空后才返回。

因此，如下代码：

```
'use strict';

exports.callback_handler = function(event, context, callback) {
console.log("event = " + event);
console.log("before callback");
setTimeout(
function(){
console.log(new Date);
console.log("timeout before callback");
},
500
);
callback(null, "success callback");
console.log("after callback");
};
```

实际日志输出结果为：

```
2018-06-14T08:07:16.545Z f3cb1ef4-6fa9-11e8-aa8a-525400c7c826 event = [object Object]
2018-06-14T08:07:16.546Z f3cb1ef4-6fa9-11e8-aa8a-525400c7c826 before callback
2018-06-14T08:07:16.546Z f3cb1ef4-6fa9-11e8-aa8a-525400c7c826 after callback
2018-06-14T08:07:17.047Z f3cb1ef4-6fa9-11e8-aa8a-525400c7c826 2018-06-14T08:07:17.047Z
2018-06-14T08:07:17.048Z f3cb1ef4-6fa9-11e8-aa8a-525400c7c826 timeout before callback
```

可以看到，setTimeout 设置的异步任务在 callback 执行后执行，函数在异步任务完成后才实际返回。

关闭事件循环等待

由于部分外部引入的库的原因，可能会导致事件循环持续不为空。这种情况将会导致函数无法返回，直到超时。为了避免外部库的影响，我们可以通过关闭事件循环等待，来自行控制函数的返回时机。通过如下两种方式，我们可以修改默认的回调行为，避免等待事件循环为空。

- 设置 context.callbackWaitsForEmptyEventLoop 为 false
- 使用 context.done 回调

通过在 callback 回调执行前设置 `context.callbackWaitsForEmptyEventLoop = false;` ，可以使得云函数后台在 callback 回调被调用后立刻冻结进程，不再等待事件循环内的事件，而在同步命令完成后立刻返回。

同样也可以通过使用 context.done 回调替换掉 callback 回调。context.done 回调的入参与 callback 回调入参相同。context.done 回调在被执行后同样会冻结事件循环监听的进程，在同步命令执行完成后立刻返回。

日志

您可以在程序中使用如下语句来完成日志输出：

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

输出内容您可以在函数日志中的 `log` 位置查看。

已包含的库及使用方法

COS SDK

云函数的运行环境内已包含 COS 的 Node.js SDK，具体版本为 `cos-nodejs-sdk-v5` 。

可在代码内通过如下方式引入 COS SDK 并使用：

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

更详细的 COS SDK 使用说明见COS Node.js SDK。

Golang

最近更新时间：2018-08-28 15:21:32

目前支持的 Golang 开发语言包括如下版本：

- Golang 1.8

函数形态

Golang 函数形态一般如下所示：

```
package main

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

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

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

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

代码开发注意点

1. 需要使用 package main 包含 main 函数。
2. 引用 `github.com/tencentyun/scf-go-lib/cloudfunction` 库。

3. 入口函数入参可选 0~2 参数，如包含参数，需 context 在前，event 在后，入参组合有（ ），（ event ），（ context ），（ context，event ），具体说明可见如下的入参说明。
4. 入口函数返回值可选 0~2 参数，如包含参数，需 返回内容在前，error 错误信息在否，返回值组合有（ ），（ ret ），（ error ），（ ret，error ），具体说明可见如下的返回值说明。
5. 入参 event ，和返回值 ret，均需要能够兼容 `encoding/json` 标准库，可以进行 Marshal、Unmarshal。
6. 在 main 函数中使用包内的 Start 函数启动入口函数。

开发规范说明

package 与 main 函数

在使用 Golang 开发云函数时，需要确保 main 函数位于 main package 中。在 main 函数中，通过使用 cloudfunction 包中的 Start 函数，启动实际处理业务的入口函数。

通过 `import "github.com/tencentyun/scf-go-lib/cloudfunction"` ，可以在 main 函数中使用包内的 Start 函数。

入口函数

入口函数为通过 cloudfunction.Start 来启动的函数，通常通过入口函数来处理实际业务。入口函数的入参和返回值都需要根据一定的规范编写。

入参

入口函数可以带有 0 ~ 2 个入参，例如：

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

在带有 2 个入参时，需要确定 context 参数在前，自定义参数在后。

自定义参数可以为 Golang 自带基础数据结构，例如 string，int，也可以为自定义的数据结构，如例子中的 DefineEvent。在使用自定义的数据结构时，需要确定数据结构可以兼容 `encoding/json` 标准库，可以进行 Marshal、Unmarshal 操作，否则在送入入参时会因为异常而出错。

自定义数据结构对应的 JSON 结构，通常与函数执行时的入参对应。在函数调用时，入参的 JSON 数据结构将会转换为自定义数据结构变量并传递和入口函数。

返回值

入口函数可以带有 0 ~ 2 个返回值，例如：

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

在定义 2 个返回值时，需要确定自定义返回值在前，error 返回值在后。

自定义返回值可以为 Golang 自带基础数据结构，例如 string，int，也可以为自定义的数据结构。在使用自定义的数据结构时，需要确定数据结构可以兼容 `encoding/json` 标准库，可以进行 Marshal、Unmarshal 操作，否则在返回至外部接口时会因为异常转换而出错。

自定义数据结构对应的 JSON 结构，通常会在函数调用完成返回时，在平台内转换为对应的 JSON 数据结构，作为运行响应传递给调用方数。

日志

您可以在程序中使用 `fmt.Println` 或使用 `fmt.Sprintf` 类似方法完成日志输出。例如例子中的函数，将可以在日志中输出入参中 Key1，Key2 的值。

输出内容您可以在函数日志中的 `log` 位置查看。

编译打包

Golang 环境的云函数，仅支持 zip 包上传，可以选择使用本地上传 zip 包或通过 COS 对象存储引用 zip 包。zip 包内包含的应该是编译后的可执行二进制文件。

Golang 编译可以在任意平台上通过制定 OS 及 ARCH 完成跨平台的编译，因此在 Linux，Windows 或 MacOS 下都可以进行编译。

在 Linux 或 MacOS 下通过如下方法完成编译及打包：

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

在 Windows 下可使用如下命令编译，然后使用打包工具对输出的二进制文件进行打包，二进制文件需要在 zip 包根目录。

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

创建函数

在创建函数时，运行环境选择 “Go1” 即可创建 Golang 环境的云函数。

执行方法

在创建 SCF 云函数时，均需要指定执行方法。使用 Golang 开发语言时，执行方法类似 `main` ，此处 `main` 表示执行的入口文件为编译后的 `main` 二进制文件。

代码包

Golang 开发语言仅支持通过使用 本地 zip 文件上传、COS 上传等方法提交 zip 包。在上传 zip 包时，请确认 zip 包的根目录下包含有指定的入口文件，文件名能够与执行方法处填写对应，避免因为无法查找到入口文件导致的执行失败。

测试函数

通过控制台右上角的测试按钮，可以打开测试界面，同步触发云函数并查看运行结果。针对代码例子，由于入参是 `DefineEvent` 数据结构，对应的 JSON 结构类似为 `{"key1":"value1"," key2":"value2"}` ，因此在使用调试界面进行触发运行时，可以输入的测试模板为类似 `{"key1":"value1"," key2":"value2"}` 的数据结构。如需使用其他数据结构测试函数，或函数入参为自定义的其他数据结构，则在函数入参数据结构与测试模板 JSON 数据结构对应时，才可运行成功。

PHP

最近更新时间：2018-08-28 15:22:08

目前支持的 PHP 开发语言包括如下版本：

- PHP 5.6
- PHP 7.2

函数形态

PHP 函数形态一般如下所示：

```
<?php

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

?>
```

执行方法

在创建 SCF 云函数时，均需要指定执行方法。使用 PHP 开发语言时，执行方法类似 `index.main_handler` ，此处 `index` 表示执行的入口文件为 `index.php` ， `main_handler` 表示执行的入口函数为 `main_handler` 函数。在使用 本地 zip 文件上传、COS 上传等方法提交代码 zip 包时，请确认 zip 包的根目录下包含有指定的入口文件，文件内有定义指定的入口函数，文件名和函数名和执行方法处填写的能够对应，避免因为无法查找到入口文件和入口函数导致的执行失败。

入参

PHP 环境下的入参包括 $event 、$context。

- $event：使用此参数传递触发事件数据。
- $context：使用此参数向您的处理程序传递运行时信息。

返回和异常

您的处理程序可以使用 `return` 来返回值，根据调用函数时的调用类型不同，返回值会有不同的处理方式。

- 同步调用：使用同步调用时，返回值会序列化后以 JSON 的格式返回给调用方，调用方可以获取返回值已进行后续处理。例如通过控制台进行的函数调试的调用方法就是同步调用，能够在调用完成后捕捉到函数返回值并显示。
- 异步调用：异步调用时，由于调用方法仅触发函数就返回，不会等待函数完成执行，因此函数返回值会被丢弃。

同时，无论同步调用还是异步调用，返回值均会在函数日志中 `ret_msg` 位置显示。

在函数中，可以通过调用 die() 退出函数。此时函数会被标记为执行失败，同时日志中也会记录使用 die() 退出时的输出。

日志

您可以在程序中使用如下语句来完成日志输出：

- echo 或 echo()
- print 或 print()
- print_r()
- var_dump()

输出内容您可以在函数日志中的 `log` 位置查看。

已安装扩展

如下列出目前已安装的 PHP 扩展供参考，如有其他扩展需求请提交工单联系我们

- date
- libxml
- openssl
- pcre
- sqlite3
- zlib
- bcmath
- bz2
- calendar
- ctype

- curl
- dom
- hash
- fileinfo
- filter
- ftp
- SPL
- iconv
- json
- mbstring
- session
- standard
- mysqlnd
- PDO
- pdo_mysql
- pdo_sqlite
- Phar
- posix
- Reflection
- mysqli
- SimpleXML
- soap
- sockets
- exif
- tidy
- tokenizer
- xml
- xmlreader
- xmlwriter
- zip
- eio
- protobuf
- redis
- Zend OPcache

您也可以随时在函数中通过 `print_r(get_loaded_extensions());` 代码打印查看已安装的扩展。

Java

Java 说明

最近更新时间：2018-08-28 15:24:36

SCF 云函数在 Java 运行时环境中提供了 Java8 的运行环境。

Java 语言由于需要编译后才可以在 JVM 虚拟中运行，因此在 SCF 中的使用方式，和 Python、Node.js 这类脚本型语言不太一样，有如下限制：

- 不支持上传代码：使用 Java 语言，仅支持上传已经开发完成，编译打包后的 zip/jar 包。SCF 云函数环境不提供 Java 的编译能力。
- 不支持在线编辑：不能上传代码，所以不支持在线编辑代码。Java 运行时的函数，在代码页面仅能看到再次通过页面上传或 COS 提交代码的方法。

代码形态

Java 开发的 SCF 云函数的代码形态一般如下所示：

```
package example;

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

执行方法

由于 Java 包含有包的概念，因此执行方法和其他语言有所不同，需要带有包信息。代码例子中对应的执行方法为 `example.Hello::mainHandler` ，此处 `example` 标识为 Java package， `Hello` 标识为类， `mainHandler` 标识为类方法。

部署包上传

可以通过 使用 Gradle 创建 zip 部署包 和 使用 Maven 创建 jar 部署包 这两种方式来创建zip或jar包。创建完成后，可通过控制台页面直接上传包（小于 10 M），或通过把部署包上传至 COS Bucket 后，在 SCF 控制台上通过指定部署包的 Bucket 和 Object 信息，完成部署包提交。

入参和返回

代码例子中，mainHandler 所使用的入参包含了两个类型，String 和 Context，返回使用了 String 类型。其中入参的前一类型标识事件入参，后一类型标识函数运行时信息。事件入参和函数返回目前支持的类型包括 Java 基础类型和 POJO 类型；函数运行时目前为 `com.qcloud.scf.runtime.Context` 类型，其相关库文件可从 此处 下载。

- 事件入参及返回参数类型支持
  - Java 基础类型，包括 byte，int，short，long，float，double，char，boolen 这八种基本类型和包装类，也包含 String 类型。
  - POJO 类型，Plain Old Java Object，您应使用可变 POJO 及公有 getter 和 setter，在代码中提供相应类型的实现。
- Context 入参
  - 使用 Context 需要在代码中使用 `com.qcloud.scf.runtime.Context;` 引入包，并在打包时带入 jar 包。
  - 如不使用此对象，可在函数入参中忽略，可写为 `public String mainHandler(String name)`

日志

您可以在程序中使用如下语句来完成日志输出：

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

输出内容您可以在函数日志中的 `log` 位置查看。

测试

通过控制台界面的测试按钮，可以打开测试界面，实时触发云函数并查看运行结果。针对代码例子，由于入参是 `String name` 字符串类型，因此在使用调试界面进行触发运行时，需要输入的为字符串内容，例如 `"Tencent Cloud"` 。

如果修改了示例代码，期望接收较复杂格式的 JSON 入参，可使用 POJO 类型参数，在代码中定义对应的数据结构。SCF 平台在传递对应 JSON 参数到入口函数时，会转换为对象实例，可由代码直接使用。

POJO 类型参数使用示例

最近更新时间：2018-08-28 15:25:31

使用 POJO 类型参数，您可以处理除简单事件入参外更复杂的数据结构。在本节中，将使用一组示例，来说明在 SCF 云函数中如何使用 POJO 参数，并能支持何种格式的入参。

事件入参和 POJO

假设我们的事件入参如下所示：

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

在有如上入参的情况下，输出接下来的内容：

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

根据入参，我们构建了如下四个类：

- RequestClass：用于接受事件，作为接受事件的类
- PersonClass：用于处理事件 JSON 内 `person` 段
- CityClass：用于处理事件 JSON 内 `city` 段
- ResponseClass：用于组装响应内容

代码准备

根据入参已经设计的四个类和入口函数，按接下来的步骤进行准备。

项目目录准备

创建项目根目录，例如 `scf_example` 。

代码目录准备

在项目根目录下创建文件夹 `src\main\java` 作为代码目录。

根据准备使用的包名，在代码目录下创建包文件夹，例如 `example` ，形成 `scf_example\src\main\java\example` 的目录结构。

代码准备

在 example 文件夹内创建 `Pojo.java` ， `RequestClass.java` ， `PersonClass.java` ， `CityClass.java` ， `ResponseClass.java` 文件，文件内容分别如下

- 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() {
```

```
}
}
```

代码编译

示例在这里选择了使用 Maven 进行编译打包，您可以根据自身情况，选择使用打包方法。

在项目根目录创建 pom.xml 函数，并输入如下内容：

```
<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>
```

在命令行内执行命令 `mvn package` 并确保有编译成功字样，如下所示：

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

如果编译失败，请根据提示进行相应修改。

编译后的生成包位于 `target\java-example-1.0-SNAPSHOT.jar` 。

SCF 云函数创建及测试

根据指引，创建云函数，并使用编译后的包作为提交包上传。您可以自行选择使用 zip 上传，或先上传至 COS Bucket后再通过选择 COS Bucket上传来提交。

云函数的执行方法配置为 `example.Pojo::handle` 。

通过测试按钮展开测试界面，在测试模版内输入我们一开始期望能处理的入参：

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

单击运行后，应该能看到返回内容为：

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

您也可以自行修改测试入参内结构的 value 内容，运行后可以看到修改效果。

使用 Gradle 创建 zip 部署包

最近更新时间：2018-08-28 15:26:33

使用 Gradle 创建 zip 部署包

本节内容提供了通过使用 Gradle 工具来创建 Java 类型 SCF 云函数部署包的方式。创建好的 zip 包符合以下规则，即可以由云函数执行环境所识别和调用。

- 编译生成的包、类文件、资源文件位于 zip 包的根目录
- 依赖所需的 jar 包，位于 /lib 目录内

环境准备

确保您已经安装 Java 和 Gradle。Java 请安装 JDK8，您可以使用 OpenJDK（Linux）或通过 www.java.com 下载安装合适您系统的 JDK。

Gradle 安装

具体安装方法可见 https://gradle.org/install/，以下说明手工安装过程：

1. 下载 Gradle 的 二进制包 或 带文档和源码的完整包。
2. 解压包到自己所期望的目录，例如 `C:\Gradle`（Windows）或 `/opt/gradle/gradle-4.1`（Linux）。
3. 将解压目录下 bin 目录的路径添加到系统 PATH 环境变量中，Linxu 通过 `export PATH=$PATH:/opt/gradle/gradle-4.1/bin` 完成添加，Windows 通过 `计算机-右键-属性-高级系统设置-高级-环境变量` 进入到环境变量设置页面，选择 `Path` 变量编辑，在变量值最后添加 `;C:\Gradle\bin;`。
4. 通过在命令行下执行 `gradle -v`，确认有如下类似输出，证明 Gradle 已正确安装。如有问题，请查询 Gradle 的官方文档。

```
------------------------------------------------------------
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**

代码准备

准备代码文件

在选定的位置创建项目文件夹，例如 scf_example 。在项目文件夹根目录，依次创建如下目录 src/main/java/ 作为包的存放目录。在创建好的目录下再创建 example 包目录，并在包目录内创建 Hello.java 文件。最终形成如下目录结构：
scf_example/src/main/java/example/Hello.java

在 Hello.java 文件内输入代码内容：

```
package example;

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

准备编译文件

在项目文件夹根目录下创建 build.gradle 文件并输入如下内容：

```
apply plugin: 'java'

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

build.dependsOn buildZip
```

使用 Maven Central 库处理包依赖

如果需要引用 Maven Central 的外部包，可以根据需要添加依赖， build.gradle 文件内容写为如下：

```
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
```

通过 repositories 指明依赖库来源为 mavenCentral 后，在编译过程中，Gradle 会自行从 Maven Central 拉取依赖项，也就是 dependencies 中指明的 `com.qcloud:qcloud-scf-java-events:1.0.0` 包。

使用本地 Jar 包库处理包依赖

如果已经下载 Jar 包到本地，可以使用本地库处理包依赖。在这种情况下，请在项目文件夹根目录下创建 `jars` 目录，并将下载好的依赖 Jar 包放置到此目录下。 `build.gradle` 文件内容写为如下：

```
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
```

通过 dependencies 指明搜索目录为 jars 目录下的 *.jar 文件，依赖会在编译时自动进行搜索。

编译打包

在项目文件夹根目录下执行命令 `gradle build` ，应有编译输出类似如下：

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

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

如果显示编译失败，请根据输出的编译错误信息调整代码。
编译后的 zip 包位于项目文件夹内的 `/build/distributions` 目录内，并以项目文件夹名命名为 `scf_example.zip` 。

函数使用

编译打包后生成的 zip 包，可在创建或修改函数时，根据包大小，选择使用页面上传（小于10M），或将包上传 COS Bucket 后再通过 COS 上传的方式更新到函数内。

使用 Maven 创建 jar 部署包

最近更新时间：2018-10-09 18:13:44

使用 Maven 创建 jar 部署包

本节内容提供了通过使用 Maven 工具来创建 Java 类型 SCF 云函数部署 jar 包的方式。

环境准备

确保您已经安装 Java 和 Maven。Java 请安装 JDK8，您可以使用 OpenJDK（Linux）或通过 www.java.com 下载安装合适您系统的 JDK。

Maven 安装

具体安装方法可见 https://maven.apache.org/install.html，以下说明手工安装过程：

1. 下载 Maven 的 zip包 或 tar.gz包 。
2. 解压包到自己所期望的目录，例如 `C:\Maven`（Windows）或 `/opt/mvn/apache-maven-3.5.0`（Linux）。
3. 将解压目录下 bin 目录的路径添加到系统 PATH 环境变量中，Linxu 通过 `export PATH=$PATH:/opt/mvn/apache-maven-3.5.0/bin` 完成添加，Windows 通过 `计算机-右键-属性-高级系统设置-高级-环境变量` 进入到环境变量设置页面，选择 `Path` 变量编辑，在变量值最后添加 `;C:\Maven\bin;`。
4. 通过在命令行下执行 `mvn -v`，确认有如下类似输出，证明 Maven 已正确安装。如有问题，请查询 Maven 的官方文档。

```
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"
```

代码准备

准备代码文件

在选定的位置创建项目文件夹，例如 `scf_example` 。在项目文件夹根目录，依次创建如下目录 `src/main/java/` 作为包的存放目录。在创建好的目录下再创建 `example` 包目录，并在包目录内创建 `Hello.java` 文件。最终形成如下目录结构：
`scf_example/src/main/java/example/Hello.java`

在 `Hello.java` 文件内输入代码内容：

```
package example;

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

准备编译文件

在项目文件夹根目录下创建 `pom.xml` 文件并输入如下内容：

```
<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>
```

使用 Maven Central 库处理包依赖

如果需要引用 Maven Central 的外部包，可以根据需要添加依赖， pom.xml 文件内容写为如下：

```
<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>
```

编译打包

在项目文件夹根目录下执行命令 `mvn package` ，应有编译输出类似如下：

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

如果显示编译失败，请根据输出的编译错误信息调整代码。

编译后的 jar 包位于项目文件夹内的 `target` 目录内，并根据 pom.xml 内的 artifactId、version 字段命名为 `java-example-1.0-SNAPSHOT.jar` 。

函数使用

编译打包后生成的 jar 包，可在创建或修改函数时，根据包大小，选择使用页面上传（小于10M），或将包上传 COS Bucket 后再通过 COS 上传的方式更新到函数内。

开发流程

最近更新时间：2018-08-28 15:29:26

函数代码是无服务器云函数最重要的部分。用户以 *SCF 云函数* 的形式将应用程序代码上传至腾讯云无服务器函数平台，由 *SCF 云函数* 代表用户运行代码，并执行所有相关的服务器管理工作。

基于云函数的应用程序的生命周期通常包括：编写代码、创建云函数、部署云函数至 SCF 平台、测试、监控和故障排除等。本部分介绍与函数代码相关的一切，监控和故障排除的具体内容请参考监控云函数和云函数日志部分。

为云函数编写代码

用户需要使用 SCF 平台支持的语言编写云函数代码。编写代码时，可以任意选用代码编写工具如 SCF 控制台，本地编辑器或本地 IDE 等。需要注意的是，如果您的代码中引入了平台暂未引入的其他依赖库，则 *必须* 上传这些依赖库，平台提供的依赖库请参考 执行环境 章节，上传代码请参考 创建部署程序包 章节。

函数的开发语言目前已支持 Python、Node.js、PHP、JAVA，各语言的函数编写方法与特性，可参考 开发语言说明 章节。

同时，SCF 平台提供了一套函数编写的基本范式。例如，如何确定函数最先调用的方法、如何从参数中获取信息、如何输出日志、如何与当前运行环境交互等。具体的函数范式请参考 编写处理方法 章节。

创建部署程序包

用户需要提供代码或部署程序包（deployment package）：

- 当您的代码中使用的都是 Python 标准库和腾讯云提供的库（如各类云产品的 SDK）时，只需在控制台提供代码，SCF 平台会自动打包此代码文件并上传至 SCF 平台。

- 如果您需要引入外部库，请按照 创建部署程序包 中的特定方式组织您的代码和依赖项，打包并上传至 SCF 平台。

- 通过上传zip包创建函数时，还需注意打包方式和“执行方法”的填写：
  执行方法格式为 a.b ，其中：a是py文件的名称，b是代码中的方法名。如果用户上传的zip包在解压后，根目录下找不到名为 a.py 的文件，则会提示“函数服务创建失败，请重试”，或者“函数代码无法显示，代码 zip 包中找不到执行方法指定的文件名”。
  例如：文件结构如下
  --RootFolder
  ----SecondFolder

------a.py
------thirdfolder
--------sth.json
压缩代码zip包时，如果压缩的是SecondFolder，则会出现上述错误；需要选择 a.py 和 thirdfoler 进行压缩。

创建及部署 SCF 云函数

用户可以通过 SCF 控制台、 API、 SDK 或 qcloud cli 工具等创建云函数。首先您需要提供云函数的配置信息，包括计算资源、运行环境等，详细信息请参考 创建 SCF 云函数。

测试及触发 SCF 云函数

您可以通过以下方法测试云函数：

- 在控制台单击【测试】按钮测试云函数。
- 使用 API、SDK 或 qcloud cli 工具的 InvokeFunction 方法测试云函数。

测试时需要提供调用数据，您可以通过传入特定云产品的调用数据（如 COS 等）来测试函数是否按您期望地响应这些云产品产生的事件。

同时，针对配置触发器的方法，以及不同云产品产生的事件数据的详细信息可以参考 管理云函数触发器 章节。

监控和故障排除

在云函数进入生产环境时，腾讯云 SCF 将自动为您监控函数的运行状况。后续将把云函数指标上报至云监控平台，以便用户自行查阅函数的运行状态。

为了帮助您调试和排除故障，腾讯云 SCF 平台将记录此函数的所有调用及处理结果，并将用户代码中生成的输出以日志的形式存储下来。有关更多信息，请查看函数日志详情。

基于无服务器云函数的应用程序示例

请确保您在使用云函数前，阅读并练习以下章节中的示例：

- 新手入门：如果您第一次使用腾讯云无服务器云函数，请首先阅读并尝试新手入门章节的所有操作。
- 代码实操：如果您需要引入外部库，则必须在本地环境中创建您的代码程序包，并上传至 SCF 平台。请根据您选用的编程语言和需要处理的事件阅读并练习使用示例中的操作步骤。

编写代码
处理方法

最近更新时间：2018-08-28 15:41:47

编写 SCF 函数代码时，首先也是最重要的步骤是编写一个处理方法（method），SCF 平台会在调用函数时首先执行该方法。创建处理方法时，遵循一个通用的语法结构：

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

所有函数的处理方法都可以接收两个入参：event 和 context。

从 event 参数中获取输入事件的详细信息

SCF 使用 event 参数将事件数据传递到函数，此参数是一个 `Python dict` 类型。

首先您需要明确函数的作用是什么，是响应一个云服务的事件触发请求（例如 COS 上传文件触发函数）？还是被其他应用程序调用（比如实现一个通用模块）？抑或不需要任何输入？

针对不同的情况，event 的值有所不同：

- 由云服务触发函数时，云服务会将事件以一种平台预定义的、不可更改的格式作为 event 参数传给 SCF 函数。您可以根据这个格式编写代码来从 event 参数中获得需要的信息。
  例如，COS 触发函数时会将 Bucket 及文件的具体信息以 JSON 格式 传递给 event 参数。

- 云函数被其他应用程序调用时，您可以在调用方和函数代码之间自由定义一个 dict 类型的参数，调用方按约定好的格式传入数据，函数代码按格式获得数据。
  例如，约定一个 dict 类型的数据结构：`{"key":"XXX"}`，当调用方传入数据 `{"key":"abctest"}` 时，函数代码可以通过 `event[key]` 来获得值 `abctest`。

- 当云函数不需要任何输入时，您可以在代码中忽略 event 和 context 参数。

返回值（可选）

返回值是用户在代码中使用 `return` 语句的返回结果，根据函数的调用类型不同，返回值将有不同的处理方法。关于函数调用类型的更多内容，请参考 核心概念 ：

-同步调用函数时，SCF 将会把代码中 return 语句的值返回给函数的调用方。
例如，腾讯云控制台的【测试】按钮将同步调用函数，因此当您使用控制台调用函数时，控制台将显示返回的值。
如果代码中未返回任何内容，则将返回空值。

- 您异步调用函数时，则将丢弃该值。

示例：新建一个 SCF 函数，复制粘贴以下代码并将执行方法设置为 index.handler ，创建完成后单击【测试】按钮并运行，观察返回的 message 结果。

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

代码从 event 参数接收输入事件并返回包含数据的消息。
有关创建函数的具体步骤请参阅 步骤一：创建 Hello World 函数。

日志介绍

最近更新时间：2018-08-28 15:43:40

log 语句为函数提供必要的执行过程中的信息，是开发者对代码进行排障的必要手段。SCF 平台会将用户在代码中使用 log 语句生成的日志全部写入日志系统中，如果您使用控制台调用函数，控制台将显示相同的日志。

用户可以通过以下语句生成日志条目：

- print
- logging 模块中的 Logger 函数

使用 logging 语句写入日志

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

上述代码使用 logging 模块将信息写入日志中，您可以前往控制台日志模块或通过 获取函数运行日志 API 来查看代码中的日志信息。日志级别标识日志的类型，例如 `INFO`、`ERROR` 和 `DEBUG`。

使用 print 语句写入日志

您也可以在代码中使用 print 语句，如以下示例所示：

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

使用控制台【测试】按钮同步调用此函数时，控制台将显示 print 语句和 return 的值。

获取日志

您可以通过以下方法获取函数运行日志

- 如果您是通过控制台【测试】按钮同步调用函数

- 执行完成后会直接在控制台展示本次调用的日志
- 如果函数被触发器调用
  - 函数的日志选项卡中会展示函数每一次被调用产生的日志
  - 也可以通过 GetFunctionLogs 接口获取函数日志
- 如果函数被 Invoke API 同步调用
  - 可在返回值的 logMsg 字段中获取本次调用的日志

错误类型

最近更新时间：2018-08-28 15:44:16

函数在调试和运行过程中如果出现异常，腾讯云 SCF 平台会尽最大可能捕获异常并将异常信息写入日志中。函数运行产生的异常包括捕获的异常（Handled error）和未捕获的异常（Unhandled Error）。例如，用户可以在代码中显式地抛出异常：

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

此函数在运行过程中将引发异常，返回下面的错误信息：

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

SCF 平台会将此错误信息记录到函数日志中。

如果您需要测试此代码，请新建函数并复制上面的函数代码，不添加任何触发器。单击控制台【测试】按钮，选择“Hello　World”测试示例进行测试。

您可以在代码中自行定义如何处理可能出现的错误，保障应用程序的健壮性和可扩展型。例如：继承Exception类

```
class UserNameAlreadyExistsException(Exception): pass

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

或者使用 Try 语句捕获错误：

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

当用户的代码逻辑中未进行错误捕获时，SCF 会尽可能捕获错误。但遇到平台也无法捕捉的错误时，例如用户函数在运行过程中突然崩溃退出，系统将会返回一个通用的错误消息。

下表展示了代码运行中常见的一些错误

错误场景	返回消息

错误场景	返回消息
使用 raise 抛出异常	{File "/var/user/index.py", line 2, in always_failed_handler raise Exception('xxx') Exception: xxx}
处理方法不存在	{'module' object has no attribute 'xxx'}
依赖模块并不存在	{global name 'xxx' is not defined}
超时	{"time out”}

创建部署程序包

最近更新时间：2018-08-28 15:44:48

部署程序包是 SCF 平台运行的所有代码和依赖项的 zip 集合文件，在创建函数时需要指定部署程序包。用户可以在本地环境创建部署程序包并上传至 SCF 平台，或直接在 SCF 控制台上编写代码由控制台为您创建并上传部署程序包。请根据以下条件确定您是否可使用该控制台创建部署程序包：

- 简单场景：如果自定义代码只需要使用标准库及腾讯云提供的 COS、SCF 等 SDK 库，且只有一个代码文件时，则可以使用 SCF 控制台中的内联编辑器。控制台会将代码及相关的配置信息自动压缩至一个能够运行的部署程序包中。
- 高级场景：如果编写的代码需要用到其他资源（如使用图形库进行图像处理，使用 Web 框架进行 Web 编程，使用数据库连接库用于执行数据库命令等），则需要先在本地环境创建函数部署程序包，然后再使用控制台上传部署程序包。

打包要求

ZIP 格式

直接上传至 SCF 平台，或通过上传 COS 再导入 SCF 方式提交的代码包，要求为 ZIP 格式 。用于压缩或解压的工具，在 Windows 平台下可使用例如 7-Zip 工具，在 Linux 平台下可使用 zip 命令行工具。

打包方式

打包时，需要针对文件进行打包，而不是针对代码整体目录进行打包；打包完成后，入口函数文件需要位于包内的根目录。

在 Windows 下打包时，可以进入函数代码目录，全选所有文件以后，单击鼠标右键，选择“压缩为 zip 包”，生成部署程序包。通过 7-Zip 等工具打开 zip 包浏览时，包内应该直接包含入口程序与其他库。

在 Linux 下打包时，可以进入函数代码目录，通过调用 zip 命令时，将源文件指定为代码目录下的所有文件，实现生成部署程序包，例如 `zip /hoem/scf_code.zip * -r` 。

部署程序包示例

下面展示在本地环境创建部署程序包的示例过程。

> **注意：**

1. 通常情况下在本地安装的依赖库在 SCF 平台上也能很好运行，但少部分情况下安装的 binary 文件可能产生兼容性问题，如果发生了此问题请您尝试联系我们。
2. 示例中针对 Python 开发语言，将在本地使用 pip 工具安装库及依赖项，请确保您本地已经安装了 Python 和 pip。

Python 部署程序包

Linux 下创建 Python 部署程序包

1) 创建一个目录：

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

2) 将创建的此函数所有 Python 源文件（.py 文件）保存在此目录，有关如何创建函数，请参考 新手入门 - 创建 DownloadImage 函数 部分。

3) 使用 pip 安装所有依赖项至此目录：

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

例如，以下命令会将 Pillow 库安装在 my-first-scf 目录下：

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

4) 在 my-first-scf 目录下，压缩所有内容。特别注意，需要压缩目录内的内容而不是目录本身：

```
cd /data/my-first-scf && zip my_first_scf.zip * -r
```

注意：

1. 针对有编译过程的库，为保持和 SCF 运行环境的统一，建议打包过程在 CentOS 7 下进行。
2. 如果在安装过程中或编译过程中，有其他软件、编译环境、开发库的需求，请根据安装提示解决编译和安装问题。

Windows 下创建 Python 部署程序包

建议您将已经在 Linux 环境下运行成功的依赖包和代码打包成 zip 包作为函数的执行代码，具体操作请参考 代码实操 - 获取COS上的图像并创建缩略图。

针对 Windows 系统，同样可以使用 `pip install <module-name> -t <code-store-path>` 命令安装 Python 库，但是针对需要编译或带有静态、动态库的包，由于 Windows 下编译生成的库无法在 SCF 的运行环境（CentOS

7）中被调用运行，因此 Windows 下的库安装仅适合纯 Python 实现的库。

创建云函数

最近更新时间：2018-08-28 15:45:14

用户在打包应用服务代码及相关依赖并将其上传到腾讯云云函数后，就创建了一个 SCF 云函数。云函数包含了用户上传的代码及依赖，以及一些与运行关联的配置信息。本文档将介绍云函数的具体配置及相关意义，帮助用户了解如何构建适合自身业务的云函数。

函数名称

腾讯云通过函数名称来唯一标识用户单个地域下的 SCF 云函数，函数名在函数创建后不可修改。函数名称应遵循以下规则：

- 长度在 60 个字符以内
- 只得包括 a-z，A-Z，0-9，-，_
- 必须以字母为开头

地域

用户指定 SCF 云函数运行在哪个地域中，当前支持北京、上海、广州、成都地域。云函数将自动在地域内的多个可用区进行高可用的部署。在函数创建后，地域属性不可更改。

计算资源

腾讯云支持用户自行定义 SCF 云函数分配的*内存量*，可从 128MB - 1536MB 的区间中以 128 MB 为增量。腾讯云将自动根据用户指定的内存为 SCF 函数自动分配成比例的 CPU 处理能力。例如：如果为 SCF 函数分配 1024 MB 内存，则函数获得的 CPU 将是分配 512 MB 内存时的 *两倍*。因此，在常规场景下，提高函数的内存量通常能使代码实际运算的时间相对减少。

用户可以随时更改云函数所需的内存量，强烈建议您在测试时发现云函数运行消耗的内存已经接近或达到设置的内存量时提高此参数，避免函数内部因为 OOM 出错。

超时时间

云函数根据运行时间和请求次数收费，为了防止云函数无限期运行（如代码中出现死循环时），每个云函数都有一个用户可自定义的超时时间，当前最大超时时间为 300 秒，默认设置为 3 秒。在达到超时时间时，若云函数仍在运

行，则 SCF 平台将自动终止该函数。

用户可以随时更改云函数的超时时间，强烈建议您在测试时发现函数运行超时或实际运行时间已经接近超时时间时提高此参数，避免函数执行时间过长被超时时间限制而中断。

描述

用户可为函数设置及随时更改描述信息。

函数代码上传方式

目前函数代码支持【在线编辑】、【本地上传 zip 包】、【通过 COS 上传 zip 包】三种方式。

在线编辑

可通过控制台的文件编辑窗口，直接修改入口文件的代码。通过【本地上传 zip 包】、【通过 COS 上传 zip 包】这两种方式上传的代码包，在上传成功后同样可以编辑入口文件，而不影响其他的库或引用文件。

本地上传 zip 包

通过控制台的上传接口，可以直接选择已经打为 zip 格式的函数代码包并上传。通过此方式上传的 zip 格式代码包，要求大小不得超过 5 M。更大体积的 zip 包，可通过【通过 COS 上传 zip 包】的方式提交。

通过 COS 上传 zip 包

通过 COS 上传 zip 包，是先将 zip 包上传至同地域的某个 COS Bucket 中，然后在创建或修改云函数时，通过指明 Bucket 及文件位置，实现 zip 包提交至云函数平台。使用 COS 上传 zip 包需要填写两项信息：COS Bucket 及 zip 文件路径。

- COS Bucket：仅可以选择与函数在相同地域的 COS Bucket。通过下拉列表，选择 zip 包所在的 Bucket。
- COS 对象文件：zip 包在 COS Bucket 内的路径及文件名，以根目录 `/` 为起始。例如，根目录下的 test.zip 文件，应写为 `/test.zip` ；根目录下创建了 functioncode 文件夹后，在文件夹里放置的 test.zip 文件，应写为 `/functioncode/test.zip` 。

执行方法

执行方法表明了调用的函数应该是哪个文件的哪个函数。执行方法通常写为 `index.main_handler` ，指向的是 index 文件内的 main_handler 函数方法。执行方法可以根据实际文件名和函数名修改。

执行方法的前一段标明为文件名，例如 index ，在 Python 环境下，指的是代码包根目录下的 index.py 文件，在 Node.js 环境下，指的是 index.js 文件，在 PHP 环境下，指的是 index.php 文件。请确保文件名后缀与运行环境

可以对应。

执行方法的后一段标明为函数名，例如 main_handler ，指向文件内的 main_handler 函数。云函数在触发时，将首先调用此函数，并将 event 及 context 参数传递给函数。具体函数写法及入参相关信息，可见各开发语言的说明。