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

最近更新时间：2021-01-13 11:20:36

用户在使用云函数（Serverless Cloud Function，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

SCF 平台在调用云函数时，首先会寻找执行方法作为入口，执行用户的代码。此时，用户需以**文件名.执行方法名**的形式进行设置。
例如，用户设置的执行方法为 `index.handler`，则 SCF 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。
在编写执行方法时，用户需遵循平台特定的编程模型。如下所示：

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

该模型中指定固定的 event 事件数据和 context 环境数据作为入参。在执行方法中，用户需对参数进行处理，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 **event 入参**和 **context 入参**两部分，但根据开发语言和环境的不同，入参个数可能有所不同，详情请参见 开发语言说明。

event 入参

作用

参数类型为 `dict`。将 event 入参传递给执行方法，实现代码与触发函数的事件（event）交互。例如，由于文件上传触发了函数运行，代码可从 event 参数中获取该文件所有信息，包括文件名、下载路径、文件类型、大小等。

使用说明

针对不同的函数情况，event 参数的值有以下区别：

- 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，您可以根据此格式编写代码并从 event 参数中获取信息。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定义一个 `dict` 类型的参数。调用方按照定义好的格式传入数据，函数代码按格式获取数据。
  例如，定义一个 `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`。

context 入参

作用

将 context 入参传递给执行方法，代码将通过 context 入参对象，了解到运行环境及当前请求的相关内容。

使用说明

请参考以下 context 入参，了解具体信息：

```
{
getRemainingTimeInMillis: [Function: getRemainingTimeInMillis],
memory_limit_in_mb: 128,
time_limit_in_ms: 3000,
request_id: '4ca7089c-3bb0-48cf-bcdb-26d130fed2ae',
environment: '{"SCF_NAMESPACE":"default"}',
environ: 'SCF_NAMESPACE=default;SCF_NAMESPACE=default',
function_version: '$LATEST',
function_name: 'test',
namespace: 'default',
tencentcloud_region: 'ap-chengdu',
tencentcloud_appid: '1253970226',
tencentcloud_uin: '3473058547'
}
```

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 结构内容将会随着 SCF 平台的开发迭代而增加更多内容。

了解 event 入参和 context 入参的基本用法后，在编写函数代码时您还需注意以下几点：

- 为保证针对各开发语言和环境的统一性，event 入参和 context 入参均使用 JSON 数据格式统一封装。
- 不同触发器在触发函数时，所传递的数据结构均有所不同。详情请参见 函数触发器说明。
- 在使用云 API 方法触发云函数时，您可以自行定义传递给云函数的入参。

- 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。

函数返回

SCF 平台会获取到云函数执行完成后的返回值，并根据下表中不同的触发方式进行处理。

触发方式	处理方式
同步触发	• 通过 API 网关、云 API 中 RequestResponse 触发函数的方式为同步触发。 • 使用同步方式触发的函数在执行期间，SCF 平台不会返回触发请求。 • 在函数执行完成后，SCF 平台会将函数返回值封装为 JSON 格式并返回给调用方。
异步触发	• 使用异步方式触发的云函数，SCF 平台接收触发事件后，会返回触发请求。 • 在函数执行完成后，函数的返回值会封装为 JSON 格式并存储在日志中。 • 用户可在函数执行完成后，通过记录请求返回中的 requestId 查询日志，即可获取该异步触发函数的返回值。

当函数中的代码返回具体值时，通常返回特定的数据结构。例如 ：

运行环境	返回数据结构类型
Python	简单数据结构或 `dict` 数据结构
Node.js	`JSON Object`
PHP	`Array` 结构
GO	简单的数据结构或带有 JSON 描述的 `struct`

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**。SCF 平台在获取到例如以上运行环境函数的返回值后，将会对返回的数据结构进行 JSON 化，并返回 JSON 内容到调用方。

> ⚠ **注意：**
>
> - 需确保函数的返回值均可被 JSON 化，若直接返回对象且不具备 JSON 化方法，将会导致 SCF 平台在 JSON 化操作时失败并报错。
> - 例如以上运行环境的返回值，无需在 return 前自行 JSON 化，否则会导致输出的字符串会进行再次 JSON 化。

异常处理

若函数在调试和运行过程中出现异常，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）。

处理方式

本文提供以下三种抛出异常方式，您可根据实际需求选择在代码中进行异常处理。

> **说明：**
>
> 您可以前往 SCF 控制台 按照以下步骤进行异常处理测试：
>
> 1. 新建函数并复制以下函数代码，不添加任何触发器。
> 2. 单击控制台【测试】，选择 “Hello World” 测试示例进行测试。

显式抛出异常

- **示例**

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

- **说明**

  此函数在运行过程中将引发异常，返回以下错误信息，SCF 平台会将此错误信息记录到函数日志中。

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

继承Exception类

- **示例**

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

- **说明**

  您可以在代码中自行定义错误的处理方式，保障应用程序的健壮性和可扩展型。

使用Try语句捕获错误

- **示例**

```
def create_user(event):
try:
```

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

- **说明**
  您可以在代码中自行定义错误的处理方式，保障应用程序的健壮性和可扩展型。

返回错误信息

当用户的代码逻辑中未进行异常处理及错误捕获时，SCF 平台会尽可能的捕获错误。例如，用户函数在运行过程中突然崩溃退出，当出现此类平台也无法捕获错误的情况时，系统将会返回一个通用的错误信息。
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
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"}

日志

SCF 平台会将函数调用的所有记录及函数代码中的全部输出存储在日志中，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，方便调试及故障排除时使用。

日志组成

函数运行日志包含：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。

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: '5761047512
720426853',¥n subscriptionName: 'test',¥n topicName: 'test',¥n topicOwner: 1251762227,¥n type: 't
opic' }¥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_lim
it_in_mb: 128,¥n time_limit_in_ms: 30000 }¥n"
}
```


日志写入

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。

您可以通过 print 语句或 logging 模块中的 Logger 函数生成日志条目。 如下所示：

使用logging语句

下例代码使用 logging 模块将信息写入日志中：

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

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。

> ⚠ **注意：**
>
> 日志级别标识日志的类型，例如 `INFO`、`ERROR` 和 `DEBUG`。

使用print语句

在代码中使用 print 语句：

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

> ⚠ **注意：**

当使用控制台【测试】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

您可以通过以下途径获取函数运行日志：

获取途径	获取方法
SCL CLI 命令行工具	执行 `scf logs` 命令，详情请参见 日志查看。
SCF 控制台	在函数代码页面，单击【测试】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。
触发器调用	该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。
SCF API	通过 GetFunctionLogs 接口获取函数日志。
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。

注意事项

由于 SCF 的特点，您须以**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 TencentDB、对象存储 COS、云数据库 Memcached 或其他云存储服务中。

开发流程

了解更多云函数开发流程，请参见 使用流程。

测试云函数

最近更新时间：2021-07-22 10:39:55

在创建并编写完云函数之后，您可以通过以下方式测试云函数，了解函数运行情况，并检查代码执行流程。

- SCF 控制台：云端测试

测试事件及模板

云函数通过事件触发的方式运行，不同的触发器在触发函数时，传递的事件数据结构均有所不同。云函数的测试方法，即为通过发送模拟的测试事件，触发函数运行。

云函数控制台提供了如下事件模板模拟对应事件：

- **Hello World 事件模板**：简单数据结构及内容，可用于触发 hello world 模板所创建的函数。
- **COS 对象存储文件事件模板**：模拟 COS 对象存储的文件上传、删除事件。
- **CMQ Topic 事件模板**：模拟 CMQ 消息队列主题模式收到消息事件。
- **API Gateway 事件模板**：模拟 API 网关收到 API 请求事件。
- **Ckafka 事件模板**：模拟 Ckafka topic 收到消息事件。

通过控制台模板管理位置的【更换】操作，更换当前使用的测试模板，也可以更换为提供的测试事件模板或自定义的测试模板。关于事件模板的消息结构，详情请参见 触发器事件消息结构汇总。

自定义模板配置及使用

在已提供的事件模板之外，我们还可以创建更多的自定义模板。通过控制台模板管理位置的【配置】操作，可以基于已有的模板，修改并保存为自定义模板，也可以直接输入自身设计的测试事件并保存为自定义模板。

注意事项

在使用测试事件模板时，您需注意以下几点：

- 测试事件模板的名称，目前仅支持英文、数字、-、_，且需要以英文字符开头。
- 已创建的自定义测试模板，如果不再使用，也可以通过配置界面删除。
- 针对同一个函数，目前自定义测试模板仅支持配置5个。如需配置新的测试模板，请先删除不再使用的旧测试模板。

环境变量

最近更新时间：2021-03-22 15:29:37

在创建或编辑云函数时，您可以通过修改配置中的环境变量，为云函数的运行环境增加、删除或修改环境变量。

在配置环境变量后，环境变量将在函数运行时配置到所在的操作系统环境中。函数代码可以使用读取系统环境变量的方式来获取到设置的具体值并在代码中使用。

新增环境变量

使用控制台新增环境变量

1. 登录云函数控制台，选择左侧导航栏中的【函数服务】。
2. 在创建函数的过程中，或针对已创建的函数进行编辑时，可在“环境变量”中，增加环境变量。
   环境变量通常以 `key-value` 对的形式出现，请在环境变量的输入框中，前一输入框输入所需的环境变量 key，后一输入框输入所需的环境变量 value。

本地新增环境变量

本地开发时，可以直接在 `template.yaml` 中的函数下配置环境变量 `Environment` ，然后执行 `scf deploy` 命令部署到云端。如下所示：

```
test:
Type: TencentCloud::Serverless::Function
Properties:
CodeUri: ./
Type: Event
Description: This is a template function
Environment:
Variables:
ENV_FIRST: env1
ENV_SECOND: env2
Handler: index.main_handler
MemorySize: 128
Runtime: Nodejs6.10
Timeout: 3
```

查看环境变量

在配置好云函数的环境变量后，可通过查看云函数的函数配置，查询到具体已配置的环境变量，环境变量以 `key=value` 的形式显示。

使用环境变量

已配置的环境变量，会在函数运行时配置到函数所在的运行环境中，可通过代码读取系统环境变量的方式来获取到具体值并在代码中使用。需要注意的是，**环境变量无法在本地进行读取**。
假设针对云函数，配置的环境变量的 key 为 `key`，以下为各运行环境读取并打印此环境变量值的示例代码。

- 在 Python 运行环境中，读取环境变量的方法为：

```
import os
value = os.environ.get('key')
print(value)
```

- 在 Node.js 运行环境中，读取环境变量的方法为：

```
var value = process.env.key
console.log(value)
```

- 在 Java 运行环境中，读取环境变量的方法分为临时授权字段和其他字段两种情况：
  - 临时授权字段包括：`TENCENTCLOUD_SESSIONTOKEN`、`TENCENTCLOUD_SECRETID`、`TENCENTCLOUD_SECRETKEY`，读取环境变量的方法为：

```
System.out.println("value: "+ System.getProperty("key"));
```

  - 其他字段，读取环境变量的方法为：

```
System.out.println("value: "+ System.getenv("key"));
```

- 在 Golang 运行环境中，读取环境变量的方法为：

```
import "os"
var value string
value = os.Getenv("key")
```

- 在 PHP 运行环境中，读取环境变量的方法为：

```
$value = getenv('key');
```

使用场景

- **可变值提取**：针对业务中有可能会变动的值，提取至环境变量中，可避免需要根据业务变更而修改代码。
- **加密信息外置**：认证、加密相关的 key，从代码中提取至环境变量，可避免相关 key 硬编码在代码中而引起的安全风险。
- **环境区分**：针对不同开发阶段所要进行的配置和数据库信息，可提取到环境变量中。针对开发和发布的不同阶段，仅需要修改环境变量的值，分别执行开发环境数据库和发布环境数据库即可。

使用限制

针对云函数的环境变量，有如下使用限制：

- key 必须以字母 **[a-zA-Z]** 开头，只能包含字母数字字符和下划线（**[a-zA-Z0-9_]**）。
- 预留的环境变量 key 无法配置。预留的 key 包括：
  - SCF_ 开头的 key，例如 SCF_RUNTIME。
  - QCLOUD_ 开头的 key，例如 QCLOUD_APPID。
  - TECENTCLOUD_ 开头的 key，例如 TENCENTCLOUD_SECRETID。

已内置环境变量

目前运行环境中已内置的环境变量的 Key 及 Value 见下表：

环境变量 Key	具体值或值来源
`TENCENTCLOUD_SESSIONTOKEN`	{临时 SESSION TOKEN}
`TENCENTCLOUD_SECRETID`	{临时 SECRET ID}
`TENCENTCLOUD_SECRETKEY`	{临时 SECRET KEY}
`_SCF_SERVER_PORT`	28902
`TENCENTCLOUD_RUNENV`	SCF
`USER_CODE_ROOT`	/var/user/
`TRIGGER_SRC`	timer（使用定时触发器时）
`PYTHONDONTWRITEBYTECODE`	x
`PYTHONPATH`	/var/user:/opt
`CLASSPATH`	/var/runtime/java8:/var/runtime/java8/lib/*:/opt
`NODE_PATH`	/var/user:/var/user/node_modules:/var/lang/node6/lib/node_modules:

_	/var/lang/python3/bin/python3
PWD	/var/user
LOGNAME	qcloud
LANG	en_US.UTF8
LC_ALL	en_US.UTF8
USER	qcloud
HOME	/home/qcloud
PATH	/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
SHELL	/bin/bash
SHLVL	3
LD_LIBRARY_PATH	/var/runtime/java8:/var/user:/opt
HOSTNAME	{host id}
SCF_RUNTIME	函数运行时
SCF_FUNCTIONNAME	函数名
SCF_FUNCTIONVERSION	函数版本
TENCENTCLOUD_REGION	区域
TENCENTCLOUD_APPID	账号APPID
TENCENTCLOUD_UIN	账号UIN
TENCENTCLOUD_TZ	时区，当前为UTC

依赖安装

最近更新时间：2021-08-13 10:53:23

内置依赖

云函数 SCF 各个运行时已内置部分常用依赖库，您可前往各运行时代码开发中查询：

- Node.js
- Python
- PHP
- Golang

安装依赖库

您可以将 SCF 代码所有的依赖库保存在代码包中，并上传至云端以供 SCF 使用。SCF 已支持的运行时及使用方法如下：

Node.js 运行时

Node.js 运行时支持以下三种依赖库安装方法：

- 依赖库同代码一起打包上传
- 在线依赖安装
- 使用依赖管理工具

通过依赖管理工具，例如 npm，在本地安装依赖后同函数代码一同打包上传。

> 注意
> 打包时函数入口文件需要在 `zip` 包的根目录下。如果打包整个文件夹并上传 `zip` 包，则会因解压后无法在根目录找到入口文件而导致函数创建失败。

本文以安装 `lodash` 库为例：

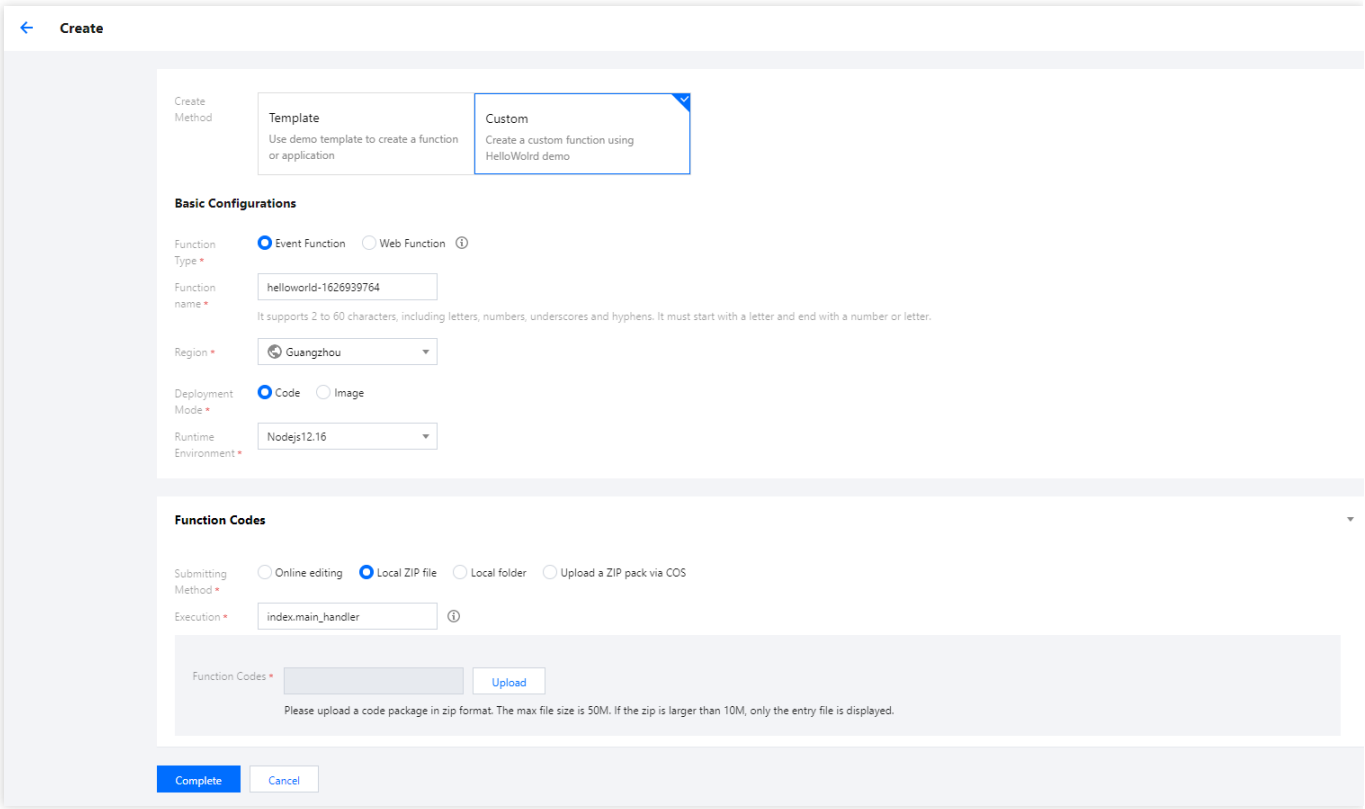
1. 在本地终端中执行 `mkdir test-package` 命令，创建一个目录用于存放函数代码和依赖库。
2. 执行以下命令，在该目录下安装 `lodash` 依赖库。

```
cd test-package
npm install lodash
```

3. 在该目录下创建函数入口文件 `index.js` 并在代码中引用 `lodash` 库。

```
'use strict';
const _ = require('lodash');
exports.main_handler = async (event, context) => {
console.log("Hello World")
console.log(event)
console.log(event["non-exist"])
console.log(context)
return event
};
```

4. 将函数代码及依赖库一同压缩为 zip 包，在 云函数控制台 中上传打包的 zip 包并创建一个新函数。操作步骤如下：
   i. 登录 云函数控制台，单击左侧导航栏的【函数服务】。
   ii. 在主界面上方选择期望创建函数的地域，并单击【新建】，进入函数创建流程。
   iii. 在“新建函数”页面，填写函数基本信息。如下图所示：



- **创建方式**：选择使用【自定义创建】来新建函数。
- **运行环境**：选择【Node.js12.16】。
- **提交方法**：选择【本地上传zip包】。

iv. 单击【完成】即可创建函数。

Python 运行时

Python 运行时支持以下两种依赖库安装方法：

- 依赖库同代码一起打包上传
- 使用依赖管理工具

通过依赖管理工具，例如 pip，在本地安装依赖后同函数代码一同打包上传。

> 注意
> - 打包时函数入口文件需要在 `zip` 包的根目录下。如果打包整个文件夹并上传 `zip` 包，则会因解压后无法在根目录找到入口文件而导致函数创建失败。
> - 由于运行环境不同，可自行将 `pip` 替换为 `pip3` 或 `pip2` 。
> - 函数运行系统为 CentOS 7，您需要在相同环境下进行安装。若环境不一致，则可能导致上传后运行时出现无法找到依赖的错误。您可参考 云函数容器镜像 进行依赖安装或使用在线 IDE 进行安装。
> - 若部分依赖涉及动态链接库，则需手动复制相关依赖包到依赖安装目录后再打包上传。详情请参见 使用 Docker 安装依赖 或使用在线 IDE 进行安装。

本文以安装 `numpy` 库为例：

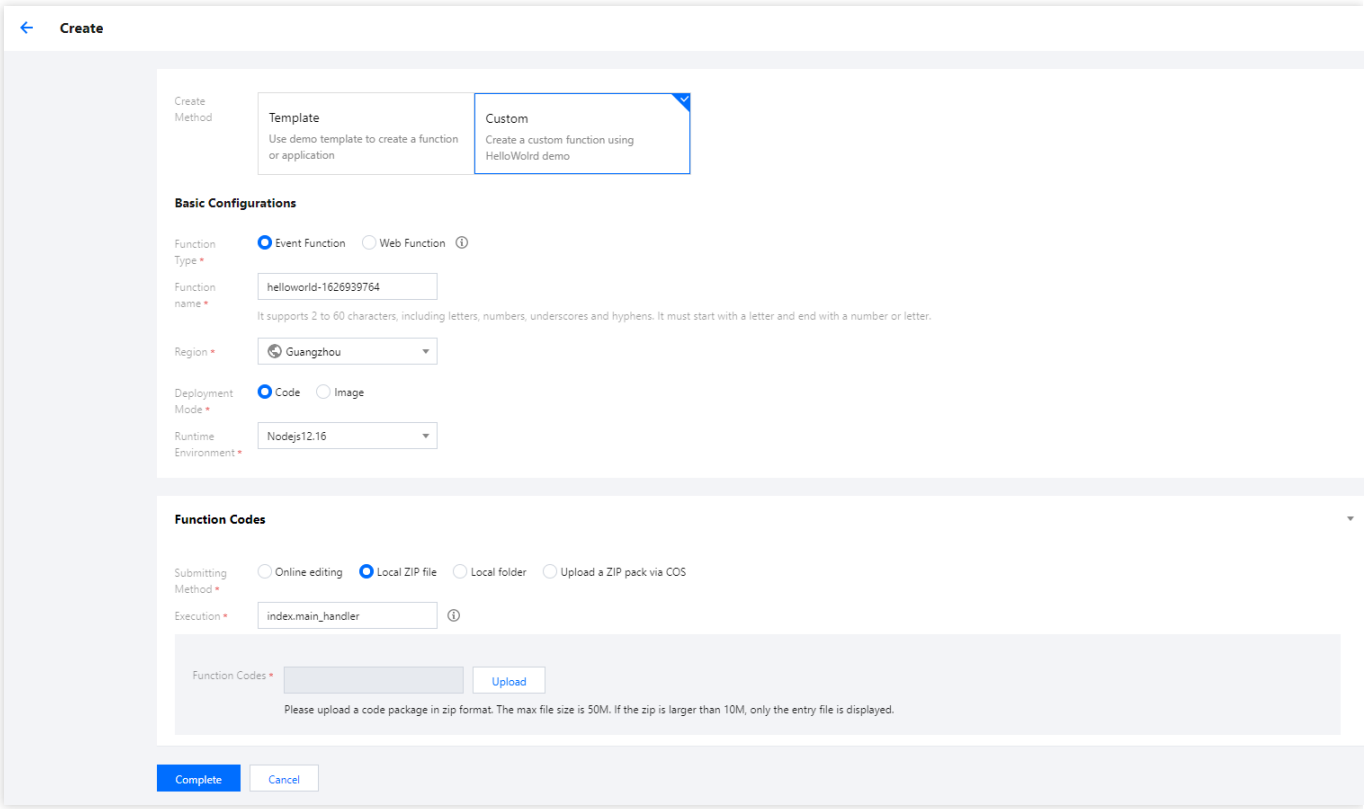
1. 在本地终端中执行 `mkdir test-package` 命令，创建一个目录用于存放函数代码和依赖库。
2. 执行以下命令，在该目录下安装 `numpy` 依赖库。

```
cd test-package
pip install numpy -t .
```

3. 在该目录下创建函数入口文件 `index.py` 并在代码中引用 `numpy` 库。

```
# -*- coding: utf8 -*-
import json
import numpy
def main_handler(event, context):
print("Received event: " + json.dumps(event, indent = 2))
print("Received context: " + str(context))
print("Hello world")
return("Hello World")
```

4. 将函数代码及依赖库一同压缩为 zip 包，在 云函数控制台 中上传打包的 zip 包并创建一个新函数。操作步骤如下：
   i. 登录 云函数控制台，单击左侧导航栏的【函数服务】。
   ii. 在主界面上方选择期望创建函数的地域，并单击【新建】，进入函数创建流程。
   iii. 在“新建函数”页面，填写函数基本信息。如下图所示：



   - **创建方式**：选择使用【自定义创建】来新建函数。
   - **运行环境**：选择【Python 3.6】。
   - **提交方法**：选择【本地上传zip包】。
   iv. 单击【完成】即可创建函数。

PHP 运行时

- 安装自定义库
- 安装自定义扩展

通过依赖管理工具，例如 composer，在本地安装依赖后同函数代码一同打包上传。

> 注意
> 打包时函数入口文件需要在 `zip` 包的根目录下。如果打包整个文件夹并上传 `zip` 包，则会因解压后无法在根目录找到入口文件而导致函数创建失败。

本文以 PHP7 安装 `requests` 库为例：

1. 在本地终端中执行 `mkdir test-package` 命令，创建一个目录用于存放函数代码和依赖库。
2. 在 `test-package` 下创建 `conposer.json` 并指定需要安装的依赖库及版本。

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

3. 执行以下命令，在该目录下安装 `requests` 依赖库。

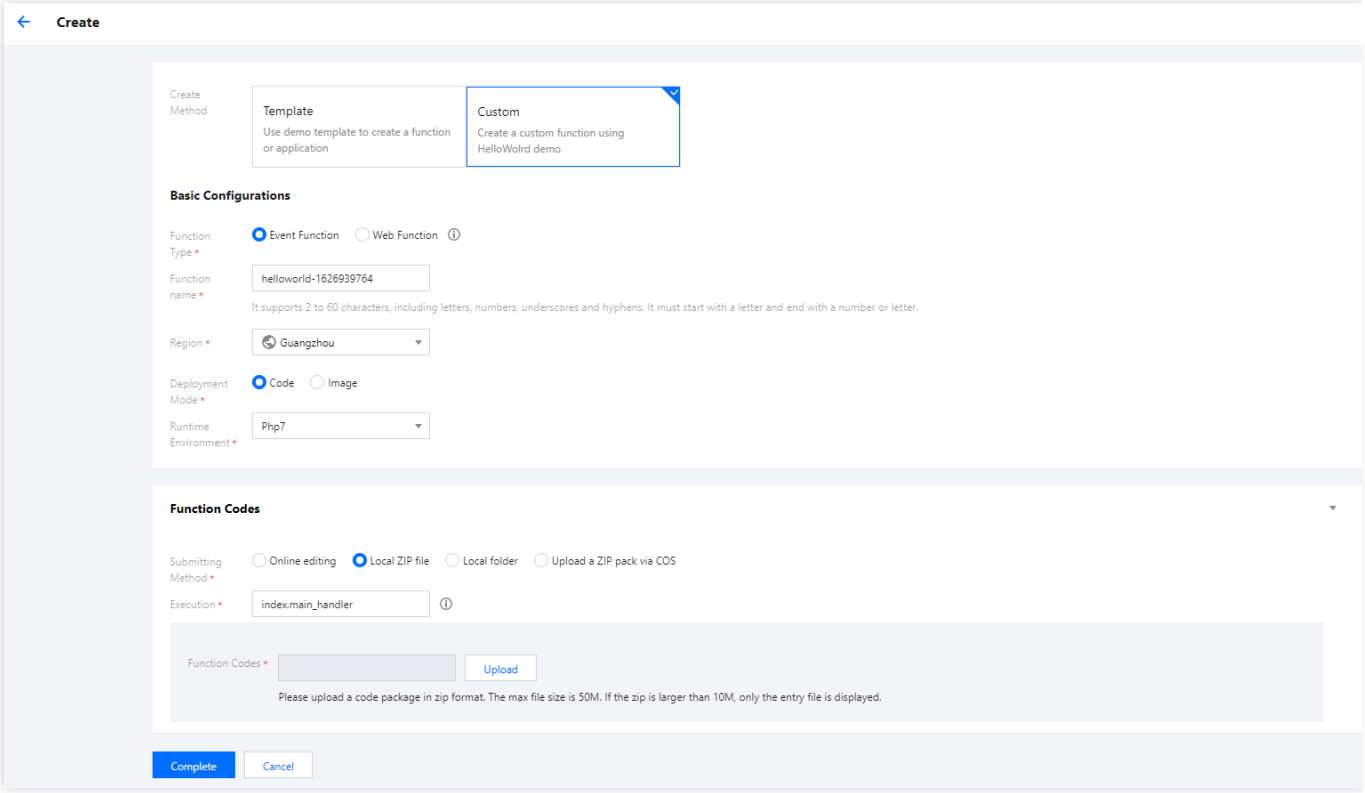
```
cd test-package
composer install
```

4. 在该目录下创建函数入口文件 `index.php` 并在代码中引用 `requests` 库。

```
<?php
require 'vendor/autoload.php';
function main_handler($event, $context) {
return "hello world";
}
?>
```

5. 将函数代码及依赖库一同压缩为 zip 包，在 云函数控制台 中上传打包的 zip 包并创建一个新函数。操作步骤如下：
   i. 登录 云函数控制台，单击左侧导航栏的【函数服务】。
   ii. 在主界面上方选择期望创建函数的地域，并单击【新建】，进入函数创建流程。

iii. 在“新建函数”页面，填写函数基本信息。如下图所示：



- **创建方式**：选择使用【自定义创建】来新建函数。
- **运行环境**：选择【Php7】。
- **提交方法**：选择【本地上传zip包】。

6. 单击【完成】即可创建函数。

Java 运行时

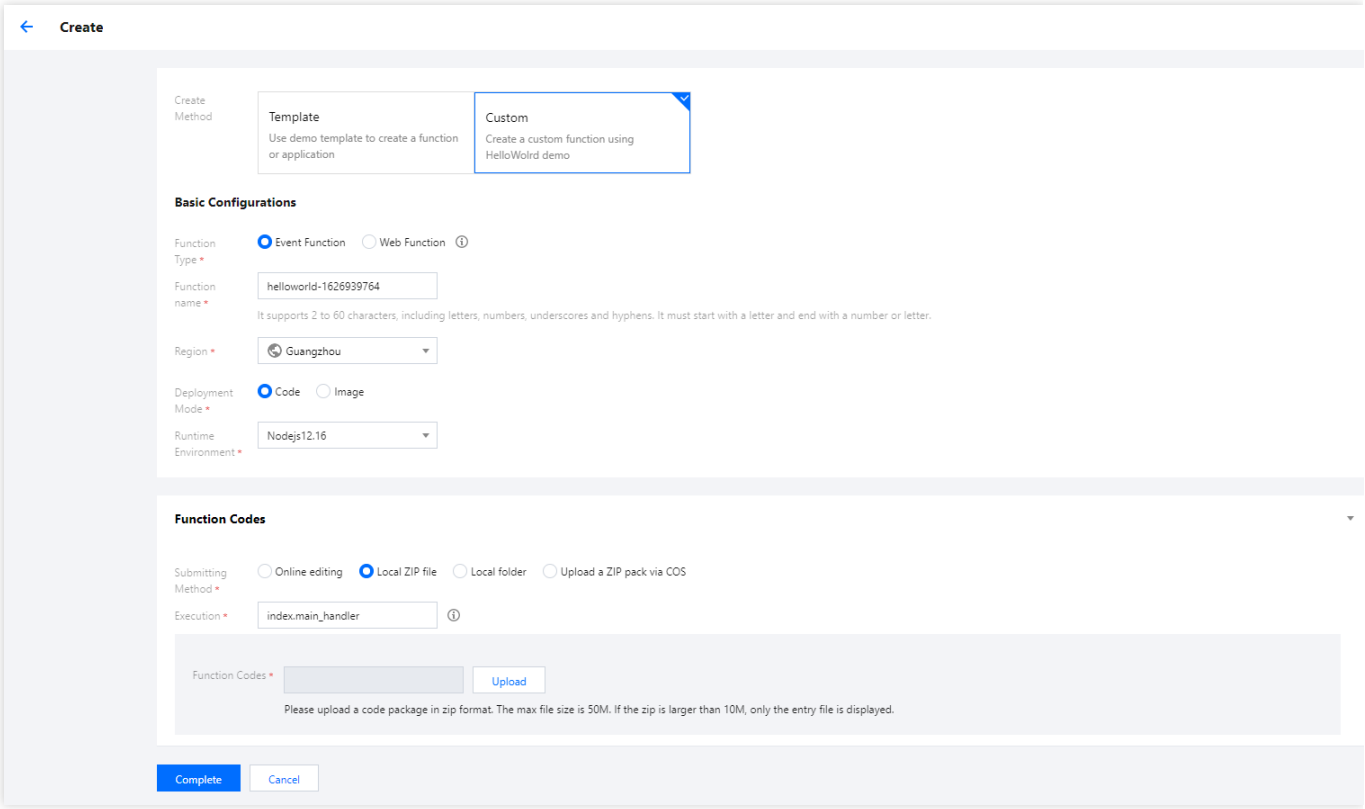
通过依赖管理工具，例如 maven，在本地安装依赖后同函数代码一同打包上传。

1. 在本地终端中执行 `mkdir test-package` 命令，创建一个目录用于存放函数代码和依赖库。
2. 在该目录下创建 `pom.xml`，并在 `pom.xml` 中配置依赖信息。
3. 在项目文件夹根目录下执行 `mvn package` 命令，编译输出如下：

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

4. 将函数代码及依赖库一同压缩为 jar 包，在 云函数控制台 中上传打包的 jar 包并创建一个新函数。操作步骤如下：
   i. 登录 云函数控制台，单击左侧导航栏的【函数服务】。
   ii. 在主界面上方选择期望创建函数的地域，并单击【新建】，进入函数创建流程。
   iii. 在“新建函数”页面，填写函数基本信息。如下图所示：



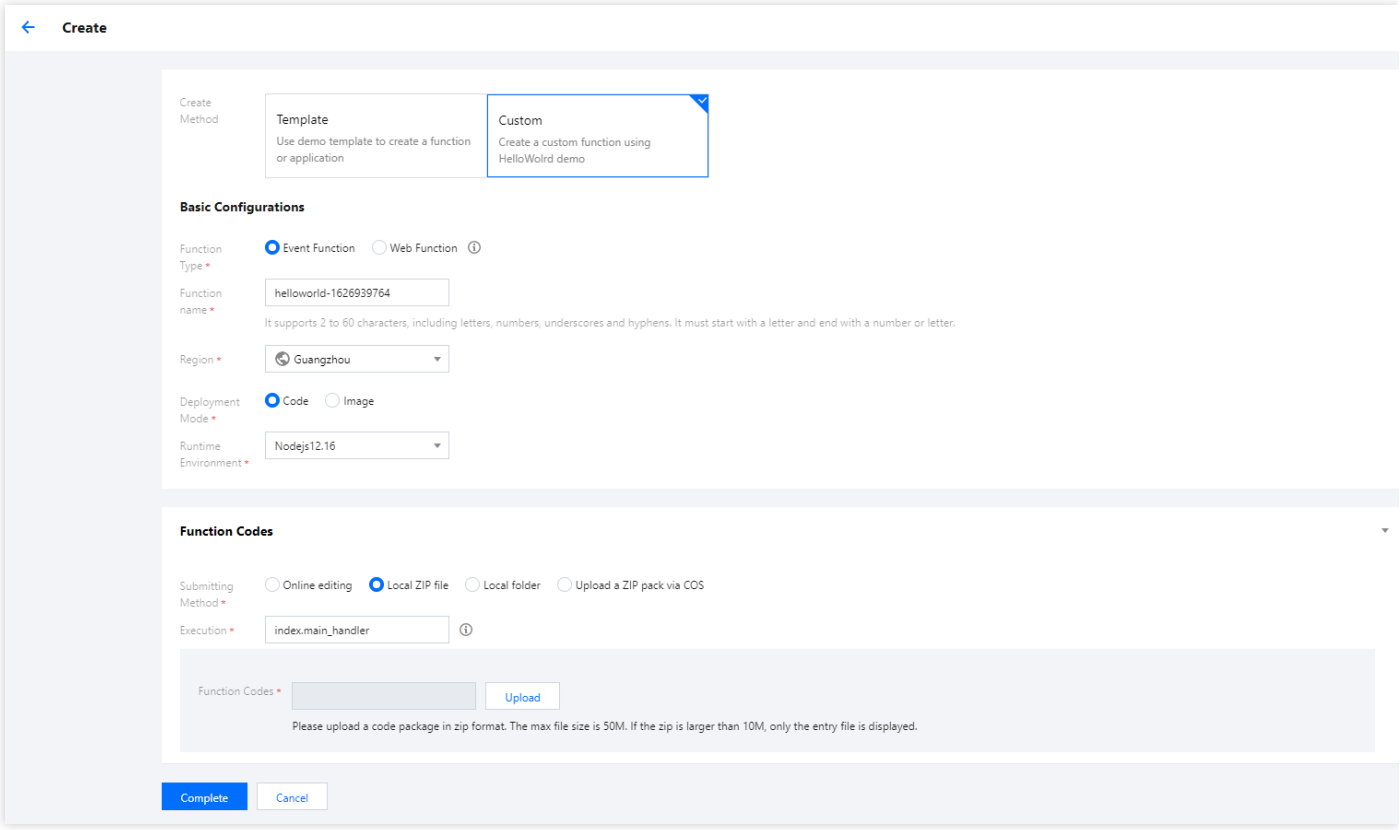
   - **创建方式**：选择使用【自定义创建】来新建函数。
   - **运行环境**：选择【Java8】。
   - **提交方法**：选择【本地上传zip包】。
5. 单击【完成】即可创建函数。

Go 运行时

使用方法：打包时上传最终的二进制文件。

Go 运行时的依赖库同代码一起编译后得到二进制文件，在 云函数控制台 中上传打包的二进制文件并创建一个新函数。操作步骤如下：

1. 登录 云函数控制台，单击左侧导航栏的【函数服务】。
2. 在主界面上方选择期望创建函数的地域，并单击【新建】，进入函数创建流程。
3. 在“新建函数”页面，填写函数基本信息。如下图所示：



- **创建方式**：选择使用【自定义创建】来新建函数。
- **运行环境**：选择【Go1】。
- **提交方法**：选择【本地上传zip包】。

4. 单击【完成】即可创建函数。

使用容器镜像

最近更新时间：2021-01-21 14:24:44

操作场景

目前云函数容器镜像已开放使用，您可以使用云函数容器镜像进行开发。本文为您介绍如何安装镜像和使用镜像进行开发。

前提条件

已在开发环境安装 Docker。

操作步骤

获取镜像

云函数镜像基于 Centos 7.7.1908版本，目前已公开在腾讯云容器服务 公有镜像。您可在公有镜像页面搜索 `scf-repo` 查看镜像信息。

1. 执行以下命令拉取镜像。示例如下：

```
# 拉取 SCF 源镜像
docker pull ccr.ccs.tencentyun.com/scf-repo/scf-runtimes-image:latest
```

> ⚠ **注意：**
>
> 如果命令提示权限错误无法正常执行，请在命令前加 `sudo` 再执行。

2. 您可以在 `/scf/lang/` 目录下查看当前镜像所包含的 Runtime。
   由于云函数源镜像包括所有的 Runtime，单个容器镜像较大。请参考以下表格内容，根据需求自行选择 Runtime 镜像。

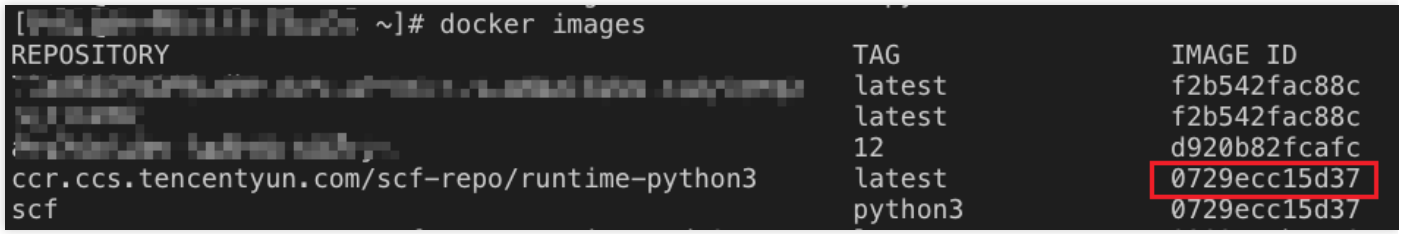
Runtime	地址
SCF	`ccr.ccs.tencentyun.com/scf-repo/scf-runtimes-image:latest`
Go 1.8	`ccr.ccs.tencentyun.com/scf-repo/runtime-go1:latest`

Runtime	地址
Python 2.7	`ccr.ccs.tencentyun.com/scf-repo/runtime-python2:latest`
Python 3.6	`ccr.ccs.tencentyun.com/scf-repo/runtime-python3:latest`
PHP 5.6	`ccr.ccs.tencentyun.com/scf-repo/runtime-php5:latest`
PHP 7.2	`ccr.ccs.tencentyun.com/scf-repo/runtime-php7:latest`
Java 8	`ccr.ccs.tencentyun.com/scf-repo/runtime-java8:latest`
Node 6.10	`ccr.ccs.tencentyun.com/scf-repo/runtime-node6:latest`
Node 8.9	`ccr.ccs.tencentyun.com/scf-repo/runtime-node8:latest`
Node 10.15	`ccr.ccs.tencentyun.com/scf-repo/runtime-node10:latest`
Node 12.16	`ccr.ccs.tencentyun.com/scf-repo/runtime-node12:latest`

3. 本文以 `scf:python3` 标签为例，执行以下命令为镜像重新贴上标签。示例如下：

```
docker pull ccr.ccs.tencentyun.com/scf-repo/runtime-python3:latest
# 通过该指令找到 IMAGE ID 并复制
docker images
# docker tag IMAGE_ID REPOSITORY:TAG
docker tag 0729ecc15d37 scf:python3
```

执行结果如下图所示：



> **说明：**
>
> 如果无需为镜像贴标签，则需将示例中的 `scf:python3` 替换为 `ccr.ccs.tencentyun.com/scf-repo/runtime-python3:latest` 。

使用镜像进行依赖安装

本文以 Node.js 12 环境安装 NodeJieba 为例，介绍如何使用镜像安装依赖。

获取 Node.js 12 镜像

执行以下命令拉取镜像。示例如下：

```
docker pull ccr.ccs.tencentyun.com/scf-repo/runtime-node12:latest
# 通过该指令找到 IMAGE ID 并复制
docker images
docker tag d64a665357b6 scf:node12
```

启动容器并挂载目录

执行以下命令，启动容器并将本地目录挂载到容器内的目录（若目录不存在，将会自动创建该目录），本文以将 `/path/to/your_project` 目录挂载至容器中的 `/tmp/your_project` 目录为例。示例如下：

```
docker run -it -v /path/to/your_project:/tmp/your_project scf:node12 /bin/bash
```

容器内安装依赖

1. 启动容器后，执行 `cd` 命令进入容器内目录，在该目录执行 `npm` 命令安装 NodeJieba。示例如下：

   ```
   cd /tmp/your_project
   npm install nodejieba --save
   ```

2. 依赖将安装在本地 `/path/to/your_project` 目录下。执行 `exit` 命令可退出容器。示例如下：

   ```
   # 退出容器
   exit
   ```

通过上述步骤即可通过容器镜像安装依赖，您可以将代码重新部署到云函数。针对 Node.js 语言，同时支持 在线依赖安装，上传时自动完成依赖的安装。

使用镜像进行开发

本文以 Python 3.6 为例，介绍如何使用容器进行开发测试。

获取 Python 3.6 镜像

执行以下命令拉取镜像。示例如下：

```
docker pull ccr.ccs.tencentyun.com/scf-repo/runtime-python3:latest
# 通过该指令找到 IMAGE ID 并复制
docker images
docker tag d64a665357b6 scf:python3
```

启动容器并挂载目录

1. 执行以下命令，启动容器并将本地项目目录挂载到容器内的目录（若目录不存在，将会自动创建该目录）。示例如下：

```
docker run --name test -it -v /path/to/your_project:/tmp/your_project scf:python3 /bin/bash
```

2. 执行 `docker exec` 命令进入容器进行开发。示例如下：

```
docker ps
# 获得CONTAINER ID
docker exec -it CONTAINER_ID /bin/bash
```

保存镜像

执行以下命令，可以将修改提交到本地的镜像中，以便后续继续使用。示例如下：

```
# 获得容器的 ID
docker ps
# 保存镜像至本地
# docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]
docker commit db47b8e66e64 scf:myimage
```

错误类型与重试策略

最近更新时间：2021-03-19 15:16:32

在函数调用的过程中，可能有多种原因导致函数调用失败。不同的**错误类型**以及**调用方式（同步调用、异步调用）**都会影响重试策略。您可以配置死信队列收集错误事件信息、分析失败原因。

错误类型

在函数调用的过程中，可能有多种原因导致函数调用失败。错误类型分为以下几类：

调用错误

调用错误发生在函数实际执行前。以下情形均会产生调用错误：

- **调用请求错误**。例如传入的 Event 数据结构过大、入参不符合要求、函数不存在等。
- **调用方错误**。主要出现在调用方权限不足的情形。
- **超限错误**。调用的并发数超出 最大并发数 限制。

运行错误

运行错误发生在函数实际运行中。运行错误有以下情形：

- **用户代码运行错误**。这类错误出现在用户代码执行过程中，例如函数代码抛出异常，或者返回结果格式问题等。
- **Runtime 错误**。函数运行过程中，Runtime 负责拉起用户代码并执行。Runtime 错误指的是 Runtime 发现并上报的错误，例如函数运行超时（超时的时间限制请参见 限制说明）、代码语法报错等。

系统错误

函数平台的错误，例如 internal error。

重试策略

不同**错误类型**以及**调用方式（同步调用、异步调用）**都会影响重试策略。

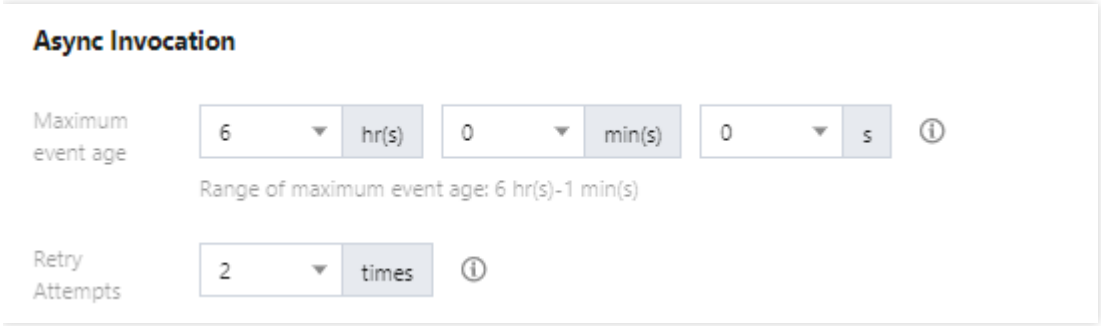
同步调用

同步调用包含 云 API 触发器 的同步调用、API 网关触发器 及 CKafka 触发器。
由于同步调用的过程中，错误信息会直接返回给用户，所以在同步调用中发生错误时，平台不会自动重试，重试策略（是否重试、重试几次）均由调用方决定。

> ⚠ **注意：**
>
> Ckafka 触发器会创建后台模块作为消费者，连接 CKafka 实例并消费消息。后台模块在获取到消息后，同步调用触发函数。由于 CKafka 触发器的后台模块是由云函数侧维护，即使是同步调用，其重试策略仍由云函数侧控制。
>
> - 对于运行错误（含用户代码错误和 Runtime 错误），CKafka 触发器会按照您配置的重试次数进行重试。默认重试10000次。
> - 对于超限错误和系统错误，CKafka 触发器会采用指数退避的方式持续重试，直至成功为止。

异步调用

异步调用包含 云 API 触发器 的异步调用、COS 触发器、定时触发器 及 CMQ Topic 触发器 等，具体触发器调用类型请参考相关触发器说明文档。
您可以根据业务诉求在函数配置中修改和自定义默认的【重试次数】，【最长等待时间】配置，该配置只适用于异步调用场景。



- **重试次数：**函数返回错误时云函数重试的次数，该参数只适用于**运行错误**的策略配置，默认配置为2次。
- **最长等待时间：**云函数在异步事件队列中保留事件的最长时间，该参数适用于所有异步调用的重试配置，默认配置为6小时，最大长度支持10w条。

异步调用发生各种错误类型的重试策略：

错误类型	重试策略
运行错误（用户代码运行错误和 Runtime 错误）	当发生该类错误时，函数平台将默认重试两次或使用配置的重试次数，固定间隔1分钟。在自动重试的同时，新的触发事件仍可正常处理。如果您配置了死信队列，三次失败后的事件将传入死信队列，否则事件将被函数平台丢弃。
系统错误	当发生该类错误时，函数平台会根据您配置的最长等待时间持续重试（默认持续重试6小时），重试间隔按照指数退避增加到5分钟。如果您配置了死信队列，重试超过最长等待时间仍失败的事件会被发送到死信队列，您可进一步处理，否则事件将被函数平台丢弃。

错误类型	重试策略
超限错误	当发生该类错误时，函数平台会根据您配置的最长等待时间持续重试（默认持续重试6小时），重试间隔为1分钟。如果您配置了死信队列，重试超过最长等待时间仍失败的事件会被发送到死信队列，您可进一步处理，否则事件将被函数平台丢弃。
调用请求错误和调用方错误	当发生该类错误时，除了**超限错误**，平台将不会对该类其他错误进行重试，原因为其他请求错误即便重试也不会成功。

死信队列

最近更新时间：2021-03-25 17:16:40

操作场景

死信队列 DLQ 是一个用户账号下的消息队列，可用于收集错误事件信息、分析失败原因。如果您为函数配置了死信队列，以下情形的事件会被发送到死信队列：

- 用户代码运行错误重试2次依然失败。
- 超限错误和系统错误重试超过24小时。
- 异步队列 消息堆积达到上限。

说明：

死信队列功能目前处于内测阶段，如需使用请 提交工单 申请开通消息队列 CMQ。

死信队列消息属性

- **RequestId**：（字符串）事件调用请求的唯一标识
- **ErrorCode**：（数字）错误码状态
- **ErrorMessage**：（字符串）错误信息

死信队列投递至消息队列 CMQ 时会把属性信息与事件信息封装在一个 JSON 请求体中，格式如下：

```
{
"RequestId": "b615b896-d197-47d7-8919-xxx",
"ErrorCode": -1,
"ErrorMessage": "Traceback (most recent call last):¥n File ¥"/var/user/index.py¥", line 5, in mai
n_handler¥n if 'key1' in event.keys():¥nNameError: global name 'event' is not defined",
"Body": {
"AppId": xxx,
"Uin": "xxx",
"SubAccountUin": "xxx",
"RequestSource": "TRIGGER_TIMER",
"FunctionName": "tabortest",
"Namespace": "default",
"Qualifier": "$DEFAULT",
"InvocationType": "RequestResponse",
"ClientContext": "{¥"Type¥":¥"Timer¥",¥"TriggerName¥":¥"tabortimer¥",¥"Time¥":¥"2020-10-10T01:22:
```

```
00Z¥",¥"Message¥":¥"¥"}",
"LogType": "",
"TimeStampForInvoker": "160229310xxx",
"RequestId": "b615b896-d197-47d7-8919-xxx",
"PushTime": "2020-10-10T09:22:00.061824591+08:00",
"RetryNum": 2,
"Ttl": 0
}
}
```

结构名	内容
AppId	APPID。
Uin	主账号 ID。
SubAccountUin	创建者的子账号 ID（此字段可能返回 null，表示取不到有效值）。
RequestSource	触发器请求来源。
FunctionName	函数名称。
Namespace	命名空间。
Qualifier	触发函数的版本/别名。
ClientContext	运行函数时的参数，以 JSON 格式传入，最大支持的参数长度请参见 配额限制。
TimeStampForInvoker	调用函数时的毫秒级时间戳。
RequestId	唯一请求 ID，每次请求都会返回。定位问题时需要提供该次请求的 RequestId。
PushTime	推送至消息队列的时间信息。
RetryNum	重试次数（0 - 2）。
Ttl	异步队列事件保留时长。

操作步骤

死信队列创建

说明：

云函数目前支持消息队列 CMQ 的主题模式和队列模式作为死信队列，您可自行选择配置。
函数别名的死信队列会跟随主版本的死信队列，主版本死信队列即在控制台创建别名时第一个选择并配置的死

信队列。

1. 登录 CMQ 消息队列控制台，创建您的死信队列。
   CMQ 的主题模式支持标签过滤与路由匹配两种筛选方案，为了确保您的订阅者可以收到所有错误信息，请在添加订阅者时标签栏筛选设为**空**，BindingKey 筛选填**“#”**。
2. 登录 云函数控制台，创建您的函数。
3. 配置死信队列。
   您可以在“新建函数”页面或“函数配置”页面进行死信队列的配置。

死信队列监控

使用死信队列时，权限错误、资源误配或消息的总大小超过目标队列或主题的限制大小均会导致死信投递失败。您可在云函数监控信息中查询“死信队列投递失败次数/次”。

1. 登录 云函数控制台，选择左侧导航栏中的【函数服务】。
2. 在主界面上方选择期望查看死信队列监控的函数所在地域，并单击列表页中期望查看监控的函数，进入函数详情页面。
3. 在函数详情页中，单击【监控信息】即可查看死信队列投递失败次数。

云函数状态码

最近更新时间：2021-07-15 11:18:04

对于函数运行后抛出的错误信息，您可以检索错误内容找到对应的问题产生原因和解决方案。

状态码及状态消息	说明	解决方法
200 Success	成功。	-
400 InvalidParameterValue	当函数执行传入参数错误时，会有该返回信息。	参数不符合规范，请参考 API 文档 修改后重试。
401 InvalidCredentials	认证失败。	权限认证失败，您的账号没有操作该函数的权限，请确认权限后重试。可参考 权限管理概述 对权限授予的说明。
404 InvalidSubnetID	当函数执行执调用时子网 id 错误时，会有该返回信息。	检查函数的 网络配置 信息是否正确以及子网 id 是否有效。
405 ContainerStateExited	容器退出。	请检查您的镜像或启动文件，是否可以本地正常启动。
406 RequestTooLarge	函数调用请求参数体太大时，会有该返回信息。	请求事件大小超限，同步请求事件最大为6MB，异步请求事件最大为128KB。
407 The size of response exceeds the upper limit (6MB)	函数返回值超出 6MB 限制。	函数返回值过大，超出 6MB 限制，请调整函数返回值大小后重试。
410 InsufficientBalance	账号余额不足。	由于您的腾讯云账户欠费导致服务停止，请充值后重试。
429 ResourceLimit	容器请求速度过高时，会有该返回信息。	容器资源请求速度过高，请稍后再试。
430 User code exception caught	当用户代码执行出现错误时，会有该返回信息。	可以根据控制台的错误日志，查看代码错误堆栈信息，检查代码是否能正常执行。

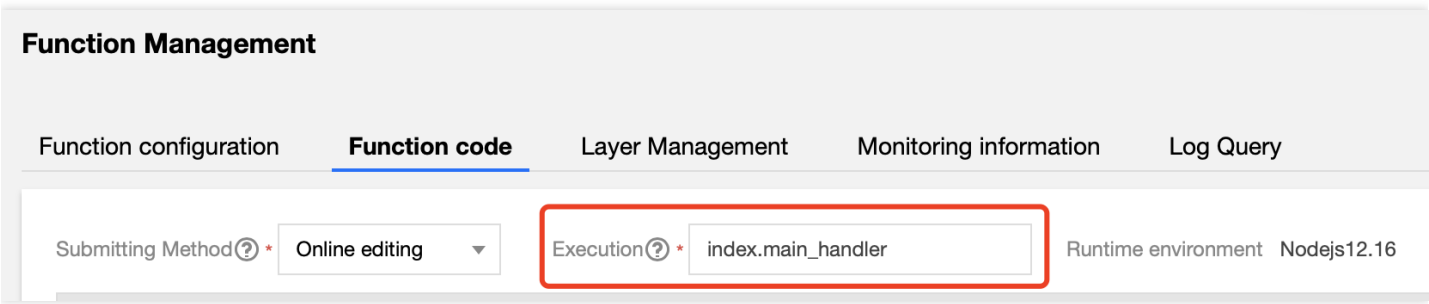
状态码及状态消息	说明	解决方法
432 ResourceLimitReached	当并发超出限制时，会有该返回信息。	容器资源使用超出最大限制（函数并发数 * 2），请调整代码或 提交工单 提升函数并发上限。
433 TimeLimitReached	当函数执行时间超出超时配置，会有该返回信息。	• 检查业务代码是否有大量耗时处理操作。 • 在函数配置页调整执行超时时间，如果当前已是最大时间设置，可 提交工单 申请提升超时限制。
434 MemoryLimitReached	当函数执行使用内存超过配置内存时，会有该返回信息。	• 检查代码逻辑，是否存在内存泄露。 • 在函数配置页面将内存配置调大，如果当前设置已经是最大内存设置，可 提交工单 申请提升内存限制。
435 FunctionNotFound	当用户函数不存在时，会有该返回信息。	查看用户函数是否被删除，或者查看传入参数函数信息是否正确。
436 InvalidParameterValue	参数不合法。	参数不符合规范，请参考 API 文档 修改后重试。
437 HandlerNotFound	当函数包加载错误时，会有该返回信息。	未找到函数执行入口文件，请确认代码包入口文件名和 handler 设置是否对应或代码压缩包是否正常。handler 详情请参见 执行方法。
438 FunctionStatusError	函数状态异常或函数关停。	• 函数状态非正常时发起调用，请等待函数状态正常后重试。 • 由于您的腾讯云账户欠费导致服务停止，请充值后重试。
439 User preocess exit when running	当函数执行时用户进程意外退出时，会有该返回信息。	可根据返回错误信息查询进程退出原因修复函数代码。
440 BorrowContainerDegrade	借容器错误熔断。	借容器熔断错误，可能由于扩容速度超限或并发超限导致，请 提交工单 提升扩容速度或提升 并发配额。
441 UnauthorizedOperation	当函数执行时，用户 CAM 鉴权不通过，会有该返回信息。	需确认函数调用角色的 CAM 鉴权信息是否传参正确。可参考 权限管理概述 对权限授予的说明。
442 QualifierNotFound	当函数指定版本调用时，未找到对应版本，会有该返回信息。	确认传入指定版本信息是否正确，确认控制台是否配置别名版本信息正确。

状态码及状态消息	说明	解决方法
443 UserCodeError	当用户代码执行出现错误时，会有该返回信息。	可以根据控制台的错误日志，查看代码错误堆栈信息，检查代码是否能正常执行。
444 PullImageFailed	拉取镜像失败。	请您确认所选择镜像的完整性和有效性后重试，如本地可正常下载。若仍无法解决，请 提交工单 。
445 ContainerInitError	容器启动失败。	容器启动失败，请检查您的启动文件是否已成功上传，并且保证调用路径正确。
446 PortBindingFailed	端口监听失败。	容器初始化超过30s最大时间，请检查您的监听端口是否为 `9000` 。
447 PullImageTimeOut	拉取镜像超时	可能是由于镜像较大或网络抖动原因引起的超时，建议在最小化镜像后重试。若仍无法解决，请 提交工单 。
450 InitContainerTimeout	当用户代码起容器超时情况下，会有该返回信息。	用户代码起容器超时（15s），请检查代码后重试。若仍无法解决，请 提交工单。
449 InsufficientResources	当大内存资源不足时，会有该返回信息。	大内存资源不足，请稍后再试。
500 InternalError	内部错误。	内部错误，请稍后重试。若仍无法解决，请联系 提交工单。

相关概念

执行方法

执行方法表明了调用云函数时需要从哪个文件中的哪个函数开始执行。如下图所示：



- 一段式格式为【文件名】，Golang 环境时使用。例如 `main` 。
- 两段式格式为【文件名.函数名】，Python、Node.js 及 PHP 环境时使用。例如 `index.main_handler` 。
  - 此执行方法**前一段指向代码包中不包含后缀的文件名，后一段指向文件中的入口函数名**。需要确保代码包中的文件名后缀与语言环境匹配，如 Python 环境为 `.py` 文件，Node.js 环境为 `.js` 文件。 更多执行方法相关说明，请参见 执行方法详情说明。
- 三段式格式为【package.class::method】，JAVA 环境时使用。例如 `example.Hello::mainHandler` 。
- 非固定段式格式，只针对 Custom Runtime 运行环境开放使用，根据自定义语言实现来设定执行方法。

云函数接入数据库
连接 PostgreSQL

最近更新时间：2020-08-04 16:05:09

操作场景

通过 Serverless Framework 组件，您可轻松完成 Serverless DB 的创建部署管理，并通过 SDK 在云函数中轻松完成数据库的连接访问，基于云上 Serverless 服务，实现“0”配置，极速部署，便捷开发，助力业务实现。

Serverless Framework 目前支持 **PostgreSQL** 与 **NoSQL** 两个类型数据库的部署连接，本文介绍如何使用云函数连接 PostgreSQL。

前提条件

- 已安装 Serverless Framework，且不低于以下版本。如未安装，请参考 安装 Serverless Framework 完成安装。

```
Framework Core: 1.67.3
Plugin: 3.6.6
SDK: 2.3.0
Components: 2.30.1
```

- 请确保当前使用账号已配置 **QcloudPostgreSQLFullAccess** 策略。配置方法请参见 授权管理。

操作步骤

本文以 Node.js 开发语言的函数为例，介绍如何通过 Serverless Framework 组件编写创建函数，并访问 PostgreSQL 数据库。配置流程如下：

1. **配置身份信息**：在本地配置腾讯云账户信息。
2. **配置私有网络**：通过 Serverless Framework VPC 组件创建 **VPC** 和 **子网**，支持云函数和数据库的网络打通和使用。
3. **配置 Serverless DB**：通过Serverless Framework PostgreSQL 组件创建 PostgreSQL 实例，为云函数项目提供数据库服务。

4. **编写业务代码**：通过 Serverless DB SDK 调用数据库，云函数支持直接调用 Serverless DB SDK，连接 PostgreSQL 数据库进行管理操作。
5. **部署与调试**：通过 Serverless Framework 部署项目至云端，并通过云函数控制台进行测试。
6. **移除项目**：可通过 Serverless Framework 移除项目。

配置身份信息

1. 在本地建立目录，用于存放代码及依赖模块。本文以 `test-postgreSQL` 为例。
2. 在 `test-postgreSQL` 下创建 `.env` 文件，并按照以下格式在文件中配置您的腾讯云 SecretId、SecretKey、地域和可用区信息。

```
# .env
TENCENT_SECRET_ID=xxx # 您账号的 SecretId
TENCENT_SECRET_KEY=xxx # 您账号的 SecretKey
# 地域可用区配置
REGION=ap-guangzhou # 资源部署区，该项目中指云函数与静态页面部署区
ZONE=ap-guangzhou-2 # 资源部署可用区 ， 该项目中指 DB 部署所在的可用区
```

> **说明：**
>
> - 如果没有腾讯云账号，请 注册新账号。
> - 如果已有腾讯云账号，请确保您的账号已经授权了 `AdministratorAccess` 权限。 同时，您可在 API 密钥管理 中获取 SecretId 和 SecretKey。

配置私有网络

1. 在 `test-postgreSQL` 下创建文件夹 `VPC` 。
2. 在 `VPC` 中新建 `serverless.yml` 文件，输入以下内容完成私有网络和子网的配置。

```
org: fullstack
app: fullstack-serverless-db
stage: dev
component: vpc # (required) name of the component. In that case, it's vpc.
name: serverlessVpc # (required) name of your vpc component instance.
inputs:
region: ${env:REGION}
zone: ${env:ZONE}
vpcName: serverless
subnetName: serverless
```

配置 Serverless DB

1. 在 `test-postgreSQL` 下创建文件夹 `DB` 。

2. 在 `DB` 中新建 `serverless.yml` 文件，输入以下内容完成 PostgreSQL 数据库创建配置。

```
org: fullstack
app: fullstack-serverless-db
stage: dev
component: postgresql
name: fullstackDB
inputs:
region: ${env:REGION}
zone: ${env:ZONE}
dBInstanceName: ${name}
vpcConfig:
vpcId: ${output:${stage}:${app}:serverlessVpc.vpcId}
subnetId: ${output:${stage}:${app}:serverlessVpc.subnetId}
extranetAccess: false
```

编写业务代码

1. 在 `test-postgreSQL` 下创建文件夹 `api`，用于存放业务逻辑代码和相关依赖项。
2. 在文件夹 `api` 下创建文件夹 `src`，并在命令行中进入 `src` 目录，执行以下命令，安装 PostgreSQL 依赖包。

```
npm install pg
```

3. 在 `src` 文件夹下，创建 `index.js` 文件，并输入如下示例代码。在函数中通过 PostgreSQL SDK 创建连接池，并调用数据库。

```
'use strict';
const { Pool } = require('pg');
exports.main_handler = async (event, context, callback) => {
let pgPool = new Pool({
connectionString: process.env.PG_CONNECT_STRING,
});
await pgPool.query(`CREATE TABLE IF NOT EXISTS users (
ID serial NOT NULL,
NAME TEXT NOT NULL,
EMAIL CHAR(50) NOT NULL,
SITE CHAR(50) NOT NULL
);`);
const client = await pgPool.connect();
const { rows } = await client.query({
text: 'select * from users',
});
await client.end();
console.log('pgsql query result:',rows)
}
```

4. 在 `api` 下创建 `serverless.yml` 文件，并输入以下内容，在环境变量中配置 Serverless DB 的 Connection String。

```
org: fullstack
app: fullstack-serverless-db
stage: dev
component: scf
name: fullstack-serverless-db
inputs:
name: ${app}
src:
src: ./src
exclude:
- .env
region: ${env:REGION}
runtime: Nodejs10.15
handler: index.main_handler
timeout: 30
vpcConfig:
vpcId: ${output:${stage}:${app}:serverlessVpc.vpcId}
subnetId: ${output:${stage}:${app}:serverlessVpc.subnetId}
environment:
variables:
PG_CONNECT_STRING: ${output:${stage}:${app}:fullstackDB.private.connectionString}
```

部署与调试

1. 通过命令行，在 `test-postgreSQL` 目录下，执行以下命令进行部署。

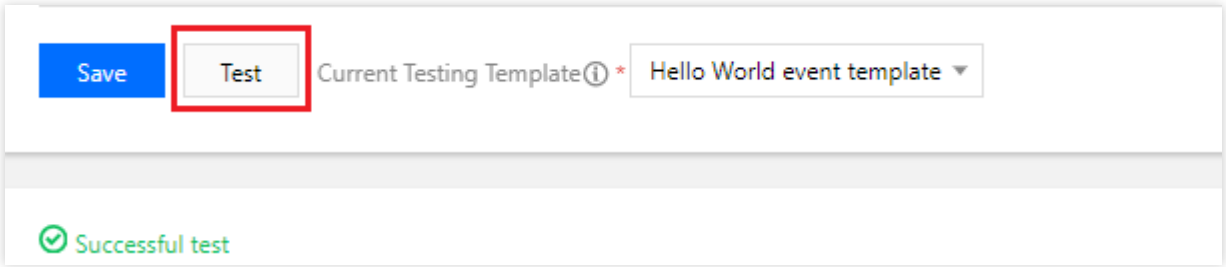
```
sls deploy --all
```

返回结果如下，即为部署成功。

```
serverless ⚡ framework
serverlessVpc:
region: ap-guangzhou
zone: ap-guangzhou-2
vpcId: vpc-0ncak84t
vpcName: serverless
subnetId: subnet-gi085his
subnetName: serverless
fullstackDB:
region: ap-guangzhou
zone: ap-guangzhou-2
vpcConfig:
subnetId: subnet-gi085his
vpcId: vpc-0ncak84t
```

```
dBInstanceName: fullstackDB
dBInstanceId: postgres-0y2x3fd3
private:
connectionString: postgresql://tencentdb_0y2x3fd3:GD0U1(q~g7%3D6ySI@10.0.0.10:5432/tencentdb_0
y2x3fd3
host: 10.0.0.10
port: 5432
user: tencentdb_0y2x3fd3
password: GD0U1(q~g7=6ySI
dbname: tencentdb_0y2x3fd3
fullstack-serverless-db:
FunctionName: fullstack-serverless-db
Description:
Namespace: default
Runtime: Nodejs10.15
Handler: index.main_handler
MemorySize: 128
25s › fullstack-serverless-db › Success
```

2. 部署成功后，您可登录 云函数控制台 查看函数并进行调试。测试步骤请参见 云端测试，测试成功如下图所示：



说明：

您还可通过 Serverless Dashboard，轻松实现已部署项目的实时监控。

移除项目

在 `test-postgreSQL` 目录下，执行以下命令可移除项目。

```
sls remove --all
```

返回结果如下，即为成功移除。

```
serverless ⚡ framework
38s › tencent-fullstack › Success
```

使用已有 SDK 连接数据库

最近更新时间：2020-12-15 15:03:05

操作场景

本文介绍如何使用已有 SDK 在云函数代码中连接 MySQL 数据库，并实现对数据库的插入、查询等操作。同时还支持连接 TDSQL 数据库，您可按需进行相应操作。

前提条件

已注册腾讯云账号并完成实名认证， 未进行实名认证的用户无法购买中国境内的资源。如未注册，请前往 注册页面。

操作步骤

创建私有网络 VPC

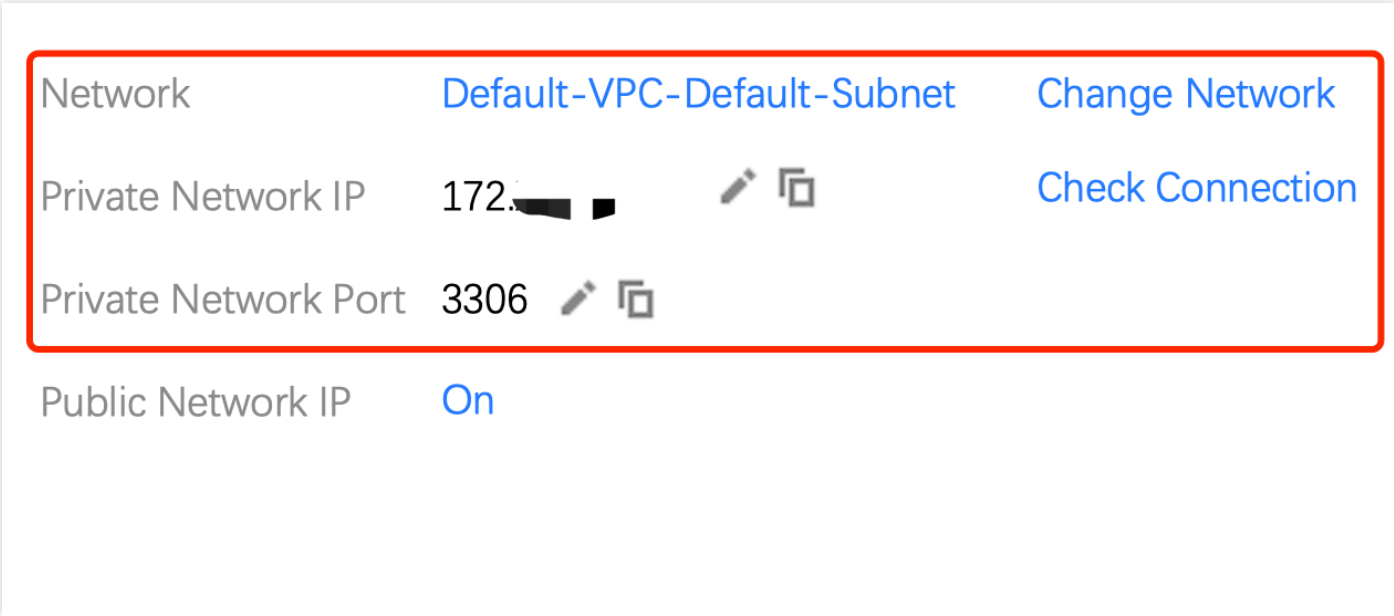
参考 快速搭建私有网络 创建 VPC 和子网。

创建数据库实例

1. 参考 购买方式 创建 MySQL。

> **说明：**
>
> 配置项“网络”请选择在 创建私有网络 VPC 步骤中已创建的 VPC。

2. 参考 初始化 MySQL 数据库 完成初始化操作，并获取数据库帐户名称及密码。
3. 在 “MySQL - 实例列表” 页面，选择实例 ID 进入数据库详情页面，获取该数据库的**内网地址**、**所属网络**、**内网端口**信息。如下图所示：

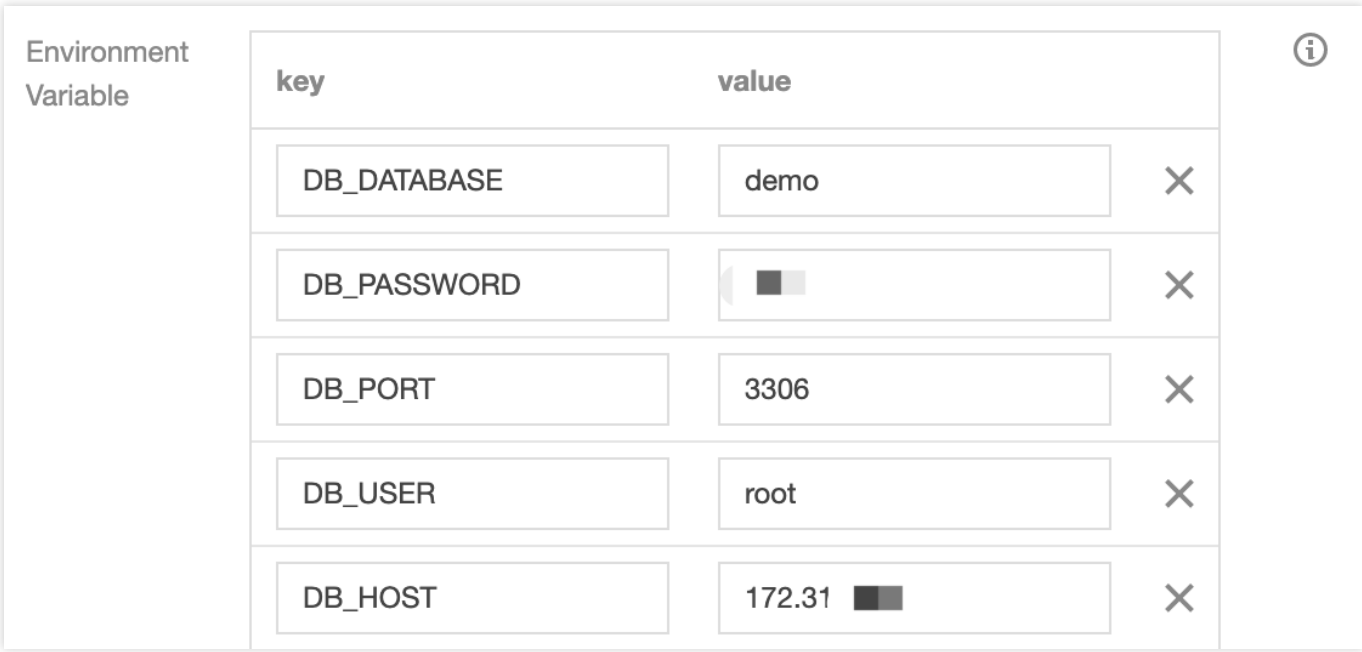


创建安全组（可选）

可参考 云数据库安全组 为您的数据库实例添加安全组。

配置环境变量和私有网络

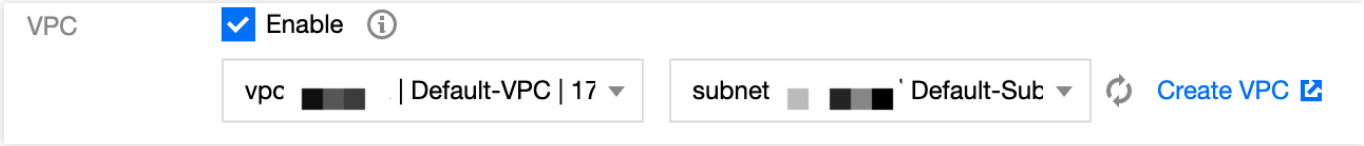
1. 登录 云函数控制台，单击左侧导航栏中的【函数服务】。
2. 单击需连接数据库的函数名，进入该函数的“函数配置”页面，参考以下信息进行配置。
   - 新增**环境变量**，并参考以下表格填写。如下图所示：



key	value

DB_PASSWORD	数据库密码
DB_USER	数据库用户名
DB_HOST	数据库地址
DB_PORT	数据库端口
DB_DATABASE	数据库名

- 开启私有网络，并选择和数据库相同的私有网络和子网。如下图所示：



函数代码示例

Python

Python 可使用云函数环境已经内置的 **pymysql** 依赖包进行数据库连接。示例代码如下：

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

import pymysql
from pymysql.err import OperationalError

mysql_conn = None

def __get_cursor():
try:
return mysql_conn.cursor()
except OperationalError:
mysql_conn.ping(reconnect=True)
return mysql_conn.cursor()

def main_handler(event, context):
global mysql_conn
if not mysql_conn:
mysql_conn = pymysql.connect(
host = getenv('DB_HOST', '<YOUR DB HOST>'),
user = getenv('DB_USER','<YOUR DB USER>'),
password = getenv('DB_PASSWORD','<YOUR DB PASSWORD>'),
```

```
db = getenv('DB_DATABASE','<YOUR DB DATABASE>'),
port = int(getenv('DB_PORT','<YOUR DB PORT>')),
charset = 'utf8mb4',
autocommit = True
)

with __get_cursor() as cursor:
cursor.execute('select * from employee')
myresult = cursor.fetchall()
print(myresult)
for x in myresult:
print(x)
```

Node.js

Node.js 支持使用连接池进行连接，连接池具备自动重连功能，可有效避免因云函数底层或者数据库释放连接造成的连接不可用情况。示例代码如下：

> **说明：**
>
> 使用连接池前需先安装 **mysql2** 依赖包，详情请参见 依赖安装。

```
'use strict';

const DB_HOST = process.env[`DB_HOST`]
const DB_PORT = process.env[`DB_PORT`]
const DB_DATABASE = process.env[`DB_DATABASE`]
const DB_USER = process.env[`DB_USER`]
const DB_PASSWORD = process.env[`DB_PASSWORD`]

const promisePool = require('mysql2').createPool({
host : DB_HOST,
user : DB_USER,
port : DB_PORT,
password : DB_PASSWORD,
database : DB_DATABASE,
connectionLimit : 1
}).promise();

exports.main_handler = async (event, context, callback) => {
let result = await promisePool.query('select * from employee');
console.log(result);
}
```

PHP

PHP 可使用 **pdo_mysql** 依赖包进行数据连接。示例代码如下：

```
<?php
function handler($event, $context) {
try{
$pdo = new PDO('mysql:host= getenv("DB_HOST");dbname= getenv("DB_DATABASE"),getenv("DB_USER"),get
env("DB_PASSWORD")');
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
}catch(PDOException $e){
echo '数据库连接失败: '.$e->getMessage();
exit;
}
}
```

JAVA

1. 请参考 依赖安装，安装以下依赖。

```
<dependencies>
<dependency>
<groupId>com.tencentcloudapi</groupId>
<artifactId>scf-java-events</artifactId>
<version>0.0.2</version>
</dependency>
<dependency>
<groupId>com.zaxxer</groupId>
<artifactId>HikariCP</artifactId>
<version>3.2.0</version>
</dependency>
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<version>8.0.11</version>
</dependency>
</dependencies>
```

2. 使用 Hikari 连接池进行连接，示例代码如下：

```
package example;

import com.qcloud.scf.runtime.Context;
import com.qcloud.services.scf.runtime.events.APIGatewayProxyRequestEvent;
import com.qcloud.services.scf.runtime.events.APIGatewayProxyResponseEvent;
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;
```

```
import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.HashMap;
import java.util.Map;

public class Http {
private DataSource dataSource;

public Http() {
HikariConfig config = new HikariConfig();
config.setJdbcUrl("jdbc:mysql://" + System.getenv("DB_HOST") + ":"+ System.getenv("DB_PORT") +
"/" + System.getenv("DB_DATABASE"));
config.setUsername(System.getenv("DB_USER"));
config.setPassword(System.getenv("DB_PASSWORD"));
config.setDriverClassName("com.mysql.jdbc.Driver");
config.setMaximumPoolSize(1);
dataSource = new HikariDataSource(config);
}

public String mainHandler(APIGatewayProxyRequestEvent requestEvent, Context context) {
System.out.println("start main handler");
System.out.println("requestEvent: " + requestEvent);
System.out.println("context: " + context);

try (Connection conn = dataSource.getConnection(); PreparedStatement ps = conn.prepareStatement(
"SELECT * FROM employee")) {
ResultSet rs = ps.executeQuery();
while (rs.next()) {
System.out.println(rs.getInt("id"));
System.out.println(rs.getString("first_name"));
System.out.println(rs.getString("last_name"));
System.out.println(rs.getString("address"));
System.out.println(rs.getString("city"));
}
} catch (SQLException e) {
e.printStackTrace();
}

APIGatewayProxyResponseEvent apiGatewayProxyResponseEvent = new APIGatewayProxyResponseEvent();
apiGatewayProxyResponseEvent.setBody("API GW Test Success");
apiGatewayProxyResponseEvent.setIsBase64Encoded(false);
apiGatewayProxyResponseEvent.setStatusCode(200);

Map<String, String> headers = new HashMap<>();
```

```
headers.put("Content-Type", "text");
headers.put("Access-Control-Allow-Origin", "*");
apiGatewayProxyResponseEvent.setHeaders(headers);

return apiGatewayProxyResponseEvent.toString();
}
}
```

连接 NoSQL DB

最近更新时间：2020-06-16 17:42:29

操作场景

通过 Serverless Framework 组件，您可轻松完成 Serverless DB 的创建部署管理，并通过 SDK 在云函数中轻松完成数据库的连接访问，基于云上 Serverless 服务，实现“0”配置，极速部署，便捷开发，助力业务实现。

Serverless Framework 目前支持 **PostgreSQL** 与 **NoSQL** 两个类型数据库的部署连接，本文介绍如何使用云函数连接 NoSQL DB。

前提条件

已安装 Serverless Framework，且不低于以下版本。如未安装，请参考 安装 Serverless Framework 完成安装。

```
Framework Core: 1.67.3
Plugin: 3.6.6
SDK: 2.3.0
Components: 2.30.1
```

注意事项

- 请确保您使用的账户下的 `SLS_QcsRole` 的运行角色已具备 `QcloudTCBFullAccess` 策略。如未具备，请前往 访问管理控制台 进行配置。
- 目前云开发 TCB 端仅支持每月最多创建销毁**4**次环境，请谨慎创建，若超过4次部署将会报错。

操作步骤

本文以 Node.js 开发语言的函数为例，介绍如何通过 Serverless Framework 组件编写创建函数，创建并访问 NoSQL DB。配置流程如下：

1. **身份信息配置**：在本地配置腾讯云账户信息。
2. **创建云开发环境配置文件**：通过 Serverless Framework 组件 创建云开发环境，在其中创建并使用 NoSQL DB。

3. **编写业务代码**：通过 Serverless DB SDK 调用数据库，云函数支持直接调用 Serverless DB SDK，创建 NoSQL DB 并进行管理操作。
4. **部署与调试**：通过 Serverless Framework 部署项目至云端，并通过云函数控制台进行测试。
5. **移除项目**：可通过 Serverless Framework 移除项目。

身份信息配置

1. 在本地建立目录，用于存放代码及依赖模块。本文以 `test-NoSQL` 文件夹为例。
2. Serverless Framework 支持以下2种方式配置身份信息，请按需选择：
   - 执行以下命令，并进行身份验证。

     ```
     serverless login
     ```

   - 在 `test-NoSQL` 下创建 `.env` 文件，按照以下内容配置对应的腾讯云 SecretId、SecretKey。

     ```
     # .env
     TENCENT_SECRET_ID=xxx // 您账号的 SecretId
     TENCENT_SECRET_KEY=xxx // 您账号的 SecretKey
     ```

     说明：
     - 如果没有腾讯云账号，请 注册新账号。
     - 如果已有腾讯云账号，请确保您的账号已经授权了 `AdministratorAccess` 权限。同时，您可在 API 密钥管理 中获取 SecretId 和 SecretKey。

创建云开发环境配置文件

1. 在 `test-NoSQL` 下创建文件夹 `DB`。
2. 在 `DB` 文件夹下新建 `serverless.yml` 文件，并输入以下内容，通过 Serverless Framework 组件完成云开发环境配置。

```
# serverless.yml
component: mongodb
name: mongoDBDemoMongo
org: anycodes
app: mongoDBAPP
stage: dev
inputs:
name: Mydemo
```

编写业务代码

1. 在 `test-NoSQL` 下创建文件夹 `function`，用于存放业务逻辑代码和相关依赖项。

2. 在文件夹 `function` 下创建文件夹 `src`，并在命令行中进入 `src` 目录，执行以下命令安装 tcb 依赖包。

```
npm install --save tcb-admin-node@latest
```

3. 在 `src` 文件夹下创建文件 `index.js`，并输入如下示例代码。在函数中通过 Serverless TCB SDK 调用云开发环境，并在其中完成 NoSQL 数据库的调用。

```
const tcb = require('tcb-admin-node')
const app = tcb.init({
secretId: process.env.SecretId,
secretKey: process.env.SecretKey,
env: process.env.MongoId,
serviceUrl: 'https://tcb-admin.tencentcloudapi.com/admin'
})
const db = app.database()
const _ = db.command
exports.main = async (event, context) => {
await db.createCollection('serverless')
const username = 'serverless'
const collection = db.collection('serverless')
if (username) {
await collection.add({username: username})
}
const userList = await collection.get()
return userList
}
```

4. 完成业务代码编写后，创建 `serverless.yml` 文件，并在环境变量中填写您的 **SecretId** 和 **SecretKey**。

> ⚠ **注意：**
>
> 若使用如下配置，则会创建免费云开发环境。如您已具备免费云开发环境，请将云开发环境 ID 填入 `MongoId` 中，否则会出现报错。

```
component: scf
name: mongoDBDemoSCF
org: anycodes
app: mongoDBAPP
stage: dev
inputs:
name: MongoDBDemo
src: ./src
runtime: Nodejs8.9
region: ap-guangzhou
handler: index.main
```

```
environment:
variables:
SecretId: 请填入您的SecretId
SecretKey: 请填入您的SecretKey
MongoId: ${output:${stage}:${app}:mongoDBDemoMongo.EnvId}
```

部署与调试

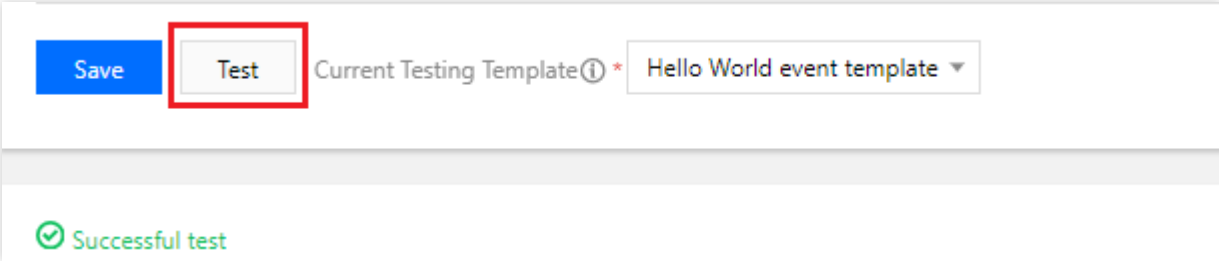
1. 使用命令行在 `test-NoSQL` 下，执行以下命令进行部署。

```
sls deploy --all
```

返回结果如下所示，即为部署成功。

```
serverless ⚡ framework
mongoDBDemoMongo:
Region: ap-guangzhou
Name: Mydemo
EnvID: Mydemo-dyxfxv
FreeQuota: basic
mongoDBDemoSCF:
FunctionName: MongoDBDemo
Description:
Namespace: default
Runtime: Nodejs8.9
Handler: index.main
MemorySize: 128
25s › tcbdemo › Success
```

2. 部署成功后，您可通过 云函数控制台，查看并进行函数调试。测试步骤请参见 云端测试，测试成功如下图所示：



说明：

您还可通过 Serverless Dashboard，松实现部署项目的实时监控。

移除项目

在 `test-NoSQL` 目录下，执行以下命令可移除项目。

```
sls remove --all
```

返回如下结果，即为成功移除。

```
serverless ⚡ framework
4s › test-NoSQL › Success
```

使用 Docker 安装依赖

最近更新时间：2020-08-26 17:57:49

操作场景

使用 Python，Node.js 等语言开发云函数 SCF 时，由于操作系统版本、系统库版本及语言版本不一致，在本地环境运行良好的程序部署到 SCF 后可能会出现错误。为解决依赖安装问题，本文档介绍使用 Docker 为函数安装依赖。详情请参考以下示例：

- Node.js 8.9 安装 nodejieba
- Python 3.6 安装 pandas

操作步骤

安装 Docker

在本地安装 Docker，详情请参见 Docker。

Node.js 8.9 安装 nodejieba

本节以下述代码为例：

```
'use strict';

const jieba = require('nodejieba');

exports.main_handler = async (event, context, callback) => {
return jieba.cut('你好世界');
};
```

此示例代码可在 Windows 和 macOS 上正确运行，但部署到 SCF 时会出现如下错误代码提示：

```
{"errorCode":1,"errorMessage":"user code exception caught","stackTrace":"/var/user/node_modules/nodejieba/build/Release/nodejieba.node: invalid ELF header"}
```

为解决此问题，可使用 Docker 来安装依赖。请参考以下命令：

```
$ docker run -it --network=host -v /path/to/your-project:/tmp/your-project node:8.9 /bin/bash -c 'cd /tmp/your-project && npm install nodejieba --save'
```

其中，`/path/to/your-project` 是项目路径，对应于 Docker 容器里的 `/tmp/your-project` 目录。因此需要在容器里的 `/tmp/your-project` 目录下安装 nodejieba，即在项目路径下安装 nodejieba。
依赖安装完成后，将代码重新部署到 SCF 上即可正常运行函数。

Python 3.6 安装 pandas

本节以下述代码为例：

```
import pandas as pd

def main_handler(event, context):
s = pd.Series([1, 3, 5, 6, 8])
print(s)
return len(s)
```

1. 参考以下命令，为 Python 3.6 安装 pandas。

   ```
   $ docker run -it --network=host -v /path/to/your-project:/tmp/your-project python:3.6.1 /bin/b
   ash -c 'cd /tmp/your-project && pip install pandas -t .'
   ```

2. 依赖安装完成后，将代码重新部署到 SCF 上并运行。函数可运行，但将产生警告提示“无法加载 lzma 模块，若使用 lzma 压缩则会导致运行时错误”。得到日志信息如下：

   ```
   /var/user/pandas/compat/__init__.py:84: UserWarning: Could not import the lzma module. Your in
   stalled Python is incomplete. Attempting to use lzma compression will result in a RuntimeErro
   r.
   warnings.warn(msg)
   0 1
   1 3
   2 5
   3 6
   4 8
   dtype: int64
   ```

3. 为解决此问题，需要进入容器内部执行以下命令：

   ```
   $ docker run -it --network=host -v /tmp/foo:/tmp/bar python:3.6.1 /bin/bash
   ```

4. 执行以下命令，安装 pandas。

```
$ cd /tmp/bar
$ pip install pandas -t .
echo <<EOF >> index.py
> import pandas as pd
>
> def main_handler(event, context):
> s = pd.Series([1, 3, 5, 6, 8])
> print(s)
> return len(s)
>
> main_handler({}, {})
> EOF
$ python -v index.py > run.log 2>&1
```

5. 执行以下命令，查看日志。示例如下：

```
$ grep lzma run.log
# /usr/local/lib/python3.6/__pycache__/lzma.cpython-36.pyc matches /usr/local/lib/python3.6/lz
ma.py
# code object from '/usr/local/lib/python3.6/__pycache__/lzma.cpython-36.pyc'
# extension module '_lzma' loaded from '/usr/local/lib/python3.6/lib-dynload/_lzma.cpython-36m
-x86_64-linux-gnu.so'
# extension module '_lzma' executed from '/usr/local/lib/python3.6/lib-dynload/_lzma.cpython-3
6m-x86_64-linux-gnu.so'
import '_lzma' # <_frozen_importlib_external.ExtensionFileLoader object at 0x7f446c40db70>
import 'lzma' # <_frozen_importlib_external.SourceFileLoader object at 0x7f446c40d160>
# cleanup[2] removing lzma
# cleanup[2] removing _lzma
# cleanup[3] wiping lzma
# cleanup[3] wiping _lzma
# destroy _lzma
# destroy lzma
```

分析日志信息可知函数运行时需要加载 lzma，需具备以下文件：

- `/usr/local/lib/python3.6/lzma.py`
- `/usr/local/lib/python3.6/lib-dynload/_lzma.cpython-36m-x86_64-linux-gnu.so`

6. 执行以下命令，查看 so 文件已具备的依赖：

```
$ ldd /usr/local/lib/python3.6/lib-dynload/_lzma.cpython-36m-x86_64-linux-gnu.so
linux-vdso.so.1 (0x00007fff75bb1000)
liblzma.so.5 => /lib/x86_64-linux-gnu/liblzma.so.5 (0x00007fc743370000)
```

```
libpython3.6m.so.1.0 => /usr/local/lib/libpython3.6m.so.1.0 (0x00007fc742e36000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fc742c19000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fc74286e000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fc74266a000)
libutil.so.1 => /lib/x86_64-linux-gnu/libutil.so.1 (0x00007fc742467000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fc742166000)
/lib64/ld-linux-x86-64.so.2 (0x00007fc74379c000)
```

分析命令执行结果可知除部分系统库外，还需以下文件：

- `/lib/x86_64-linux-gnu/liblzma.so.5`
- `/usr/local/lib/libpython3.6m.so.1.0`

7. 将上述 步骤5、步骤6 得到的4个文件，拷贝至项目路径下，并参考以下示例修改代码：

```
import os

os.environ['LD_LIBRARY_PATH'] = os.path.dirname(
os.path.realpath(__file__)) + ':' + os.environ['LD_LIBRARY_PATH']

import pandas as pd

def main_handler(event, context):
s = pd.Series([1, 3, 5, 6, 8])
print(s)
return len(s)
```

8. 将代码重新部署至 SCF，函数即可正常运行并且无告警提示。

常见错误码解决方法

最近更新时间：2021-05-08 10:43:53

常见错误码解决方法如下表所示：

错误码	处理方法
InvalidParameter.FunctionName	`FunctionName` 取值与规范不符，请参考 API 文档 修正后重试。
InvalidParameterValue.Action	所请求的 API 不存在，请参考 API 文档 修正后重试。
InvalidParameterValue.CosBucketRegion	`CosBucketRegion` 取值与规范不符，请参考 COS 地域和访问域名 修正后重试。
InvalidParameterValue.DeadLetterConfig	`DeadLetterConfig` 取值与规范不符，`Type` 取值不在 `CMQ-TOPIC` 、 `CMQ-QUEUE` 、 `topic` 、 `queue` 范围内或 `Name` 为空会触发该错误信息，请修正后重试。
InvalidParameterValue.Enable	`Enable` 取值与规范不符，`Enable` 取值不在 `OPEN` 和 `CLOSE` 范围内会触发该错误信息，请修正后重试。
InvalidParameterValue.Memory	`Memory` 取值与规范不符，函数运行时内存大小默认为128M，可选范64M、128M - 3072M，以128MB为阶梯，请修正后重试。
InvalidParameterValue.OrderBy	`OrderBy` 取值与规范不符，请参考对应 API 支持取值范围修正后重试。
InvalidParameterValue.RoutingConfig	`RoutingConfig` 取值与规范不符，请参考 API 文档 。