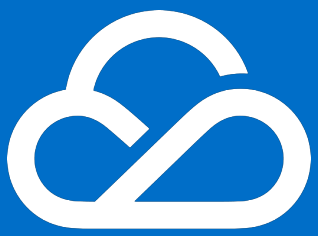
腾讯云无服务器云函数

构建云函数

产品文档



腾讯云

【版权声明】

【商标声明】

【服务声明】

本文档意在向客户介绍腾讯云全部或部分产品、服务的当时的整体概况，部分产品、服务的内容可能有所调整。您所购买的腾讯云产品、服务的种类、服务标准等应由您与腾讯云之间的商业合同约定，除非双方另有约定，否则，腾讯云对本文档内容不做任何明示或模式的承诺或保证。

文档目录

构建云函数

构建 SCF 函数的基本步骤

函数代码是无服务器云函数最重要的部分。用户以 SCF 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 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，目前仅支持 Python。可以任意选用代码编写工具如 SCF 控制台，本地编辑器和本地 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。

下面列出了平台支持的语言、可用工具和相关注意事项：

平台支持的语言	可用工具	注意事项
Python 2.7 & 3.6	SCF 控制台 您自己的编写环境如： 编辑器：可以选用vim + Pydiction, sublime text, NotePad 等多种编辑器编写Python代码。 IDE：需要 IDE 本身集成 python 环境，如 WIngIDE, eclipse + pydev, pycharm等。	如果代码不需要编译且没有引入任何外部库时，可直接使用 SCF 控制台编写代码。平台将自动把代码保存在单个 .py 文件中并自动部署。
Node.js 6.10	SCF 控制台 您自己的编写环境如： 编辑器：可以选用vim，sublime text, VS code, NotePad等多种编辑器编写 Nodejs 代码。	如果代码不需要编译且没有引入任何外部库时，可直接使用 SCF 控制台编写代码。平台将自动把代码保存在单个 .js 文件中并自动部署。

平台支持的语言	可用工具	注意事项
	IDE：如 WebStorm, Eclipse等。	

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 Python 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。

创建及及部署 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 SCF 云函数

您可以通过以下方法测试云函数：

- 在控制台点击【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，您可以功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。有关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，请参阅 函数日志 章节。

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。

为云函数编写代码

核心概念

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尊许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。

event 入参

SCF 平台将 event 对象作为第一个参数传递给执行方法。通过此 event
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。

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特定 log 语句将生成输出以便作调试及故障排除之用。

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，持久状态应存储在 COS、Memcached 或其他云存储服务中。

编写处理方法

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接收固定的入参：event 和 context，请不要删除任何固定入参。

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。

用户首先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）？还是被您的其他应用程序调用（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：

- 如果是由云服务触发函数，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。用户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。（比如 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）

- 如果云函数被其他应用程序调用，则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

；

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

（可选）返回值

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 ：

- 如果您同步调用函数，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。
- 如果您异步调用函数，则将丢弃该值。

例如，考虑以下 Python 示例代码。

```
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。

新建一个 SCF 函数，粘贴上述代码并将执行方法设置为

index.handler

，创建完成后点击【测试】按钮并运行，观察返回的 message 结果。有关创建函数的具体步骤请参阅步骤一：创建 Hello World 函数。

在代码中使用log语句

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

错误处理

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气场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。点击控制台【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
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

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 Python 库及腾讯云提供的COS、SCF 等 SDK 库，且只有一个 .py 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。

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](https://cloud.tencent.com/document/product/583/9712)。
2. 示例中将在本地使用 pip 安装依赖项，请确保您本地已经安装了 Python 和 pip。

Linux下创建部署程序包

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 PIL 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：

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

Windows 下创建部署程序包

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](#)。

配置及测试

配置环境变量

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

新增环境变量

在创建函数的过程中，或针对创建后的函数进行编辑时，可通过点击 【显示高级设置】
按钮，展开高级设置内容。

可通过点击【高级设置项】>【环境变量配置】>【新增环境变量】，增加环境变量输入框。

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。

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。

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为：

```
System.out.println("value: "+ System.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：

- 单个云函数的环境变量总条数为 128 条

- 每条记录，key 大小最多 64 字节，value 大小最多 128 字节，不允许 value 为空。
- 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。

配置 VPC 私有网络

在创建或编辑云函数时，您可以通过修改高级配置中的网络配置，来为云函数的运行环境增加至 VPC 网络的访问能力。

编辑网络配置

在创建函数的过程中，或针对创建后的函数进行编辑时，可通过点击 【显示高级设置】 按钮，展开高级设置内容。

可通过【高级设置】>【网络】配置项，选择需接入的 VPC 网络和所需要使用的子网。如果在当前地域无 VPC 网络，可通过【新建网络】 跳转至 VPC 网络控制台以创建新的 VPC 网络，也可以通过 【新建子网】 创建 VPC 网络下的新子网。

通过选择网络选项中的

无

，可重新切换云函数的网络环境至当前所属的独立网络环境

查看网络配置

在配置好云函数的网络项后，可通过查看云函数的函数配置，通过 所属网络 和 所在子网 了解到具体配置。

使用 VPC 网络

在配置完成并开始使用 VPC 网络后，此云函数的运行网络环境，将从当前独立的网络环境中切换至用户的 VPC 中。云函数启动时，将占用用户 VPC 子网中的 IP 地址作为云函数运行环境的 IP 地址。

> 注意
> 请确保子网中有足够的可用空闲的 IP 地址。如果由于无空闲 IP 导致的 IP 分配失败，将使得云函数启动运行失败。

云函数启动后，可通过代码访问 VPC 内的其他各产品，例如 弹性缓存 Redis、云数据库 CDB、或用户配置在 VPC 中的 CVM 等等各种访问入口位于 VPC 中的产品或服务，直接通过内网 IP 地址即可访问。如下为访问 弹性缓存 Redis 的示例代码，其中 Redis 实例在 VPC 内的 IP 地址为

10.0.0.86

。

```
# -*- coding: utf8 -*-
import redis

def main_handler(event,context):
  r = redis.StrictRedis(host='10.0.0.86', port=6379, db=0,password="crs-i4kg86dg:abcd1234")
  print(r.set('foo', 'bar'))
  print(r.get('foo'))
  return r.get('foo')
```

云函数切换至 VPC 网络环境后，将失去原有独立网络环境中的外网访问能力，如需继续访问外网，在 VPC 上通过 配置公网网关、配置 NAT 网关 等方式，打通 VPC 访问外网的能力。

使用场景

- 内网服务访问：访问内网的数据库、Redis、Kafka 等产品或服务，确保数据安全，连接安全。
- 访问控制：外网访问统一收敛至同一地址，通过 VPC 外网出口可控出口地址。

使用测试模版测试函数

云函数的测试功能，用于在页面上直接发起函数调用，模拟触发器发送的触发事件，并展示云函数的执行情况、返回内容、运行日志，可用于直接在页面上调试函数功能和代码。

测试操作

在创建函数后，您可以进入控制台，点击【测试】，在弹出的测试函数页面进行测试事件模版的选择与编辑，最后点击【测试运行】按钮。

测试事件模版

在产品迭代过程中，默认测试事件模版会不断新增。

测试事件模版用来模拟在相应触发器触发云函数运行时，传递给云函数的事件和内容，在函数中以 event 入参的形式体现。测试事件模版需要是 JSON 格式的数据结构。目前已包含的默认测试事件模版和说明如下：

- Hello World 事件模版：简单、自定义的事件模版，在通过云 API 触发函数时，可输入自定义事件内容。
- COS 上传、删除文件事件模版：模拟绑定 COS 对象存储触发器后，在 Bucket 中有文件上传或删除时触发云函数所产生和传递的事件。
- CMQ Topic 事件模版：模拟绑定 CMQ 消息队列主题订阅后，在消息队列中收到消息的情况下触发云函数所产生和传递的事件。
- API 网关事件模版：模拟 API 网关绑定云函数后，在有 API 请求到达 API 网关时触发云函数所产生和传递的事件。

测试事件模版自定义

在测试前，您可以根据自身的事件情况对测试模版进行修改。修改后的测试模版将用来作为触发函数运行的事件内容传递给函数。修改后的测试模版需要为 JSON 格式。

自定义测试事件模版新建和保存

在测试时，如果不想要每次进入测试界面时均需要修改模版，可以将修改后的测试模版保存为自定义模版。在

选定需要修改的模版后，可以点击 【新建模版】按钮，完成对模版的修改，并输入一个容易记忆的名字后保存。后续在使用保存的模版测试后，再次进入测试界面时，会仍然保存为上次测试使用的函数模版。

自定义测试事件模版删除

对于不再使用的自定义模版，可以通过选择模版后点击【删除】按钮进行删除。

使用限制

针对自定义测试事件模版，有如下使用限制：

- 自定义测试事件模版基于帐号范围，同一帐号下不同函数共用相同测试事件模版。
- 单个帐号最多可配置 5 个自定义测试模版。
- 每个自定义测试模版内容最大内容 64 KB。

创建 SCF 云函数

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、上海和广州三个地域。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128MB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 512MB 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也相对

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，避免因为函数内部因为 OOM 出错：

- 登录腾讯云控制台,点击【无服务器云函数】选项。
- 选择要更改内存大小的函数。
- 点击【函数配置】选项卡的【编辑】按钮，选择需要调整的内存量后保存。

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：

- 登录腾讯云控制台,点击【无服务器云函数】选项。
- 选择要更改超时时间的函数。
- 点击【函数配置】选项卡的【编辑】按钮，选择需要调整的超时时间后保存。

描述

用户可为函数设置及随时更改描述信息。

开发语言说明

Node.js

目前支持的 Node.js 开发语言包括如下版本：

- Node.js 6.10

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，函数的

return

将会被忽略。

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：可行参数，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。

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。

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。

Python

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。

Java

Java

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::mainHandle
```

，此处

example

标识为 Java package，

Hello

标识为类，

mainHandle

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。

POJO类型参数使用示例

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,
request.person.lastName, request.city.name);
  return new ResponseClass(greetingString);
  }
```

```
}
```

- RequestClass.java

```
package example;


public class RequestClass {
  PersonClass person;
  CityClass city;

  public PersonClass getPerson() {
  return person;
  }

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

  public CityClass getCity() {
  return city;
  }

  public void setCity(CityClass city) {
  this.city = city;
  }

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

  public RequestClass() {
```

```
  }
}
```

- PersonClass.java

```
package example;

public class PersonClass {
  String firstName;
  String lastName;

  public String getFirstName() {
  return firstName;
  }

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

  public String getLastName() {
  return lastName;
  }

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

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

  public PersonClass() {
```

```
  }
}
```

- CityClass.java

```
package example;

public class CityClass {
  String name;


  public String getName() {
  return name;
  }

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

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

  public CityClass() {
  }
}
```

- ResponseClass.java

```
package example;
```

```
public class ResponseClass {
  String greetings;

  public String getGreetings() {
  return greetings;
  }

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

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

  public ResponseClass() {
  }
}
```

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"}
}
```

点击运行后，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

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

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。