云函数

代码开发

产品文档

腾讯云

文档目录

代码开发
Python
环境说明

最近更新时间：2024-04-22 18:03:28

Python 版本选择

目前支持的 Python 开发语言包括如下版本：
Python 2.7
Python 3.6
您可以在函数创建时，选择您所期望使用的运行环境，`Python 2.7` 或 `Python 3.6` 。您可以在 这里 查看 Python 官方对 Python 2 或 Python 3 语言选择的建议。

相关环境变量

目前运行环境中内置的 Python 相关环境变量见下表：

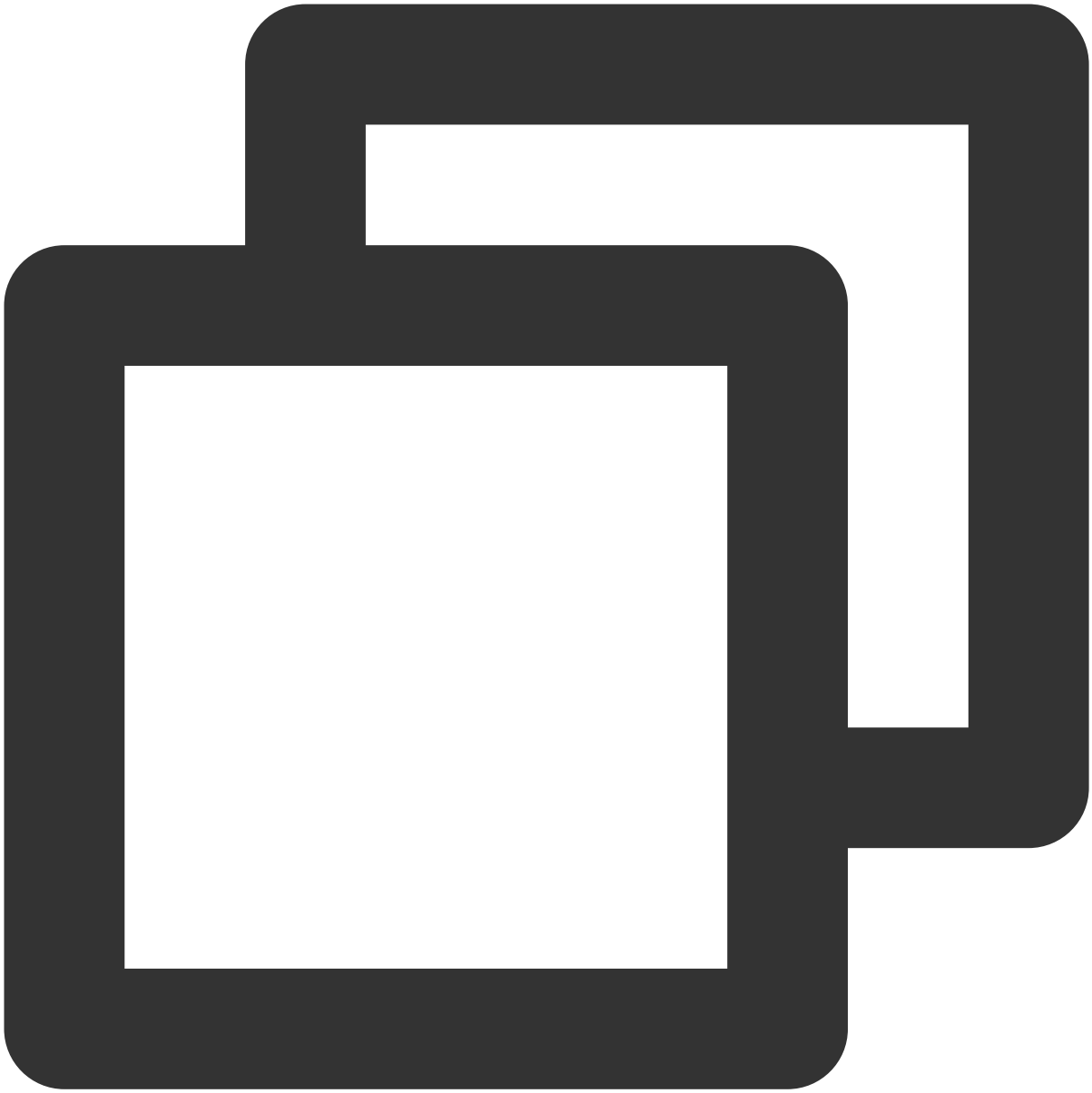
环境变量 Key	具体值或值来源
`PYTHONDONTWRITEBYTECODE`	x
`PYTHONPATH`	/var/user:/opt
`_`	/var/lang/python3/bin/python3

更多详细环境变量说明请参见 环境变量说明。

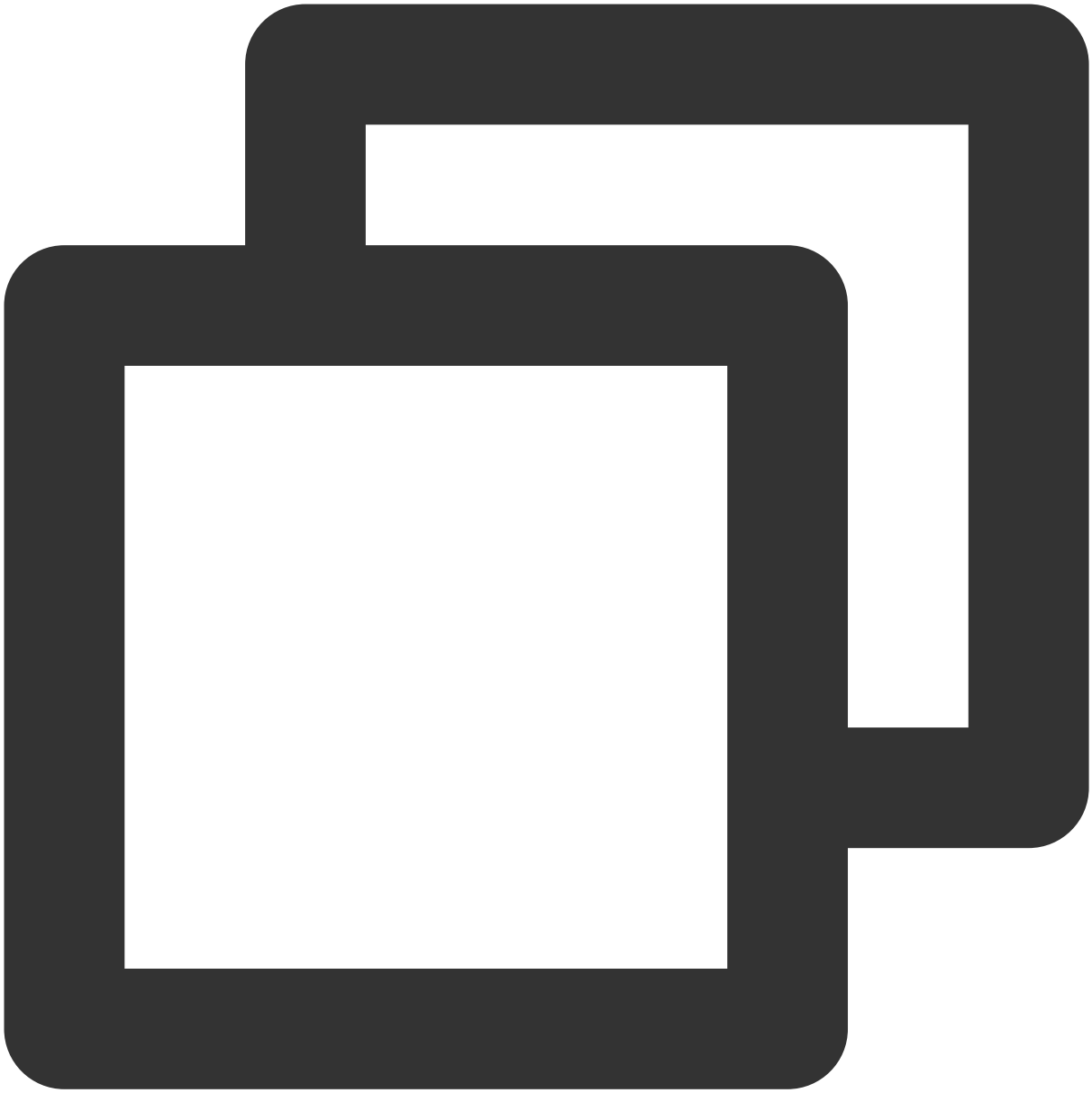
已包含的库及使用方法

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_v5` （推荐）和 `cos_sdk_v4` 。
可在代码内通过如下方式引入 COS SDK 并使用：
对于 `cos_sdk_v5` 版本：

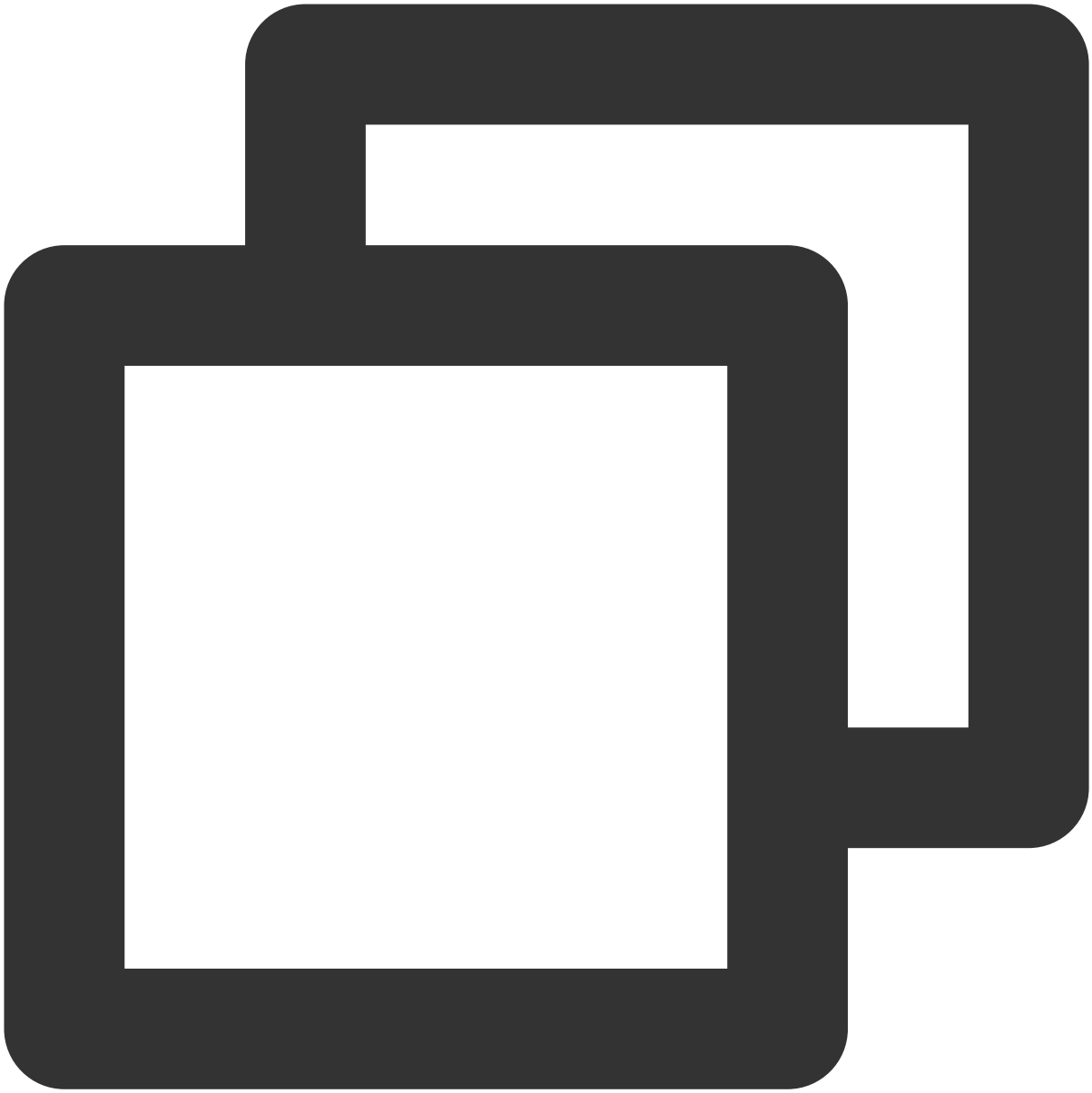


```
import qcloud_cos_v5
```

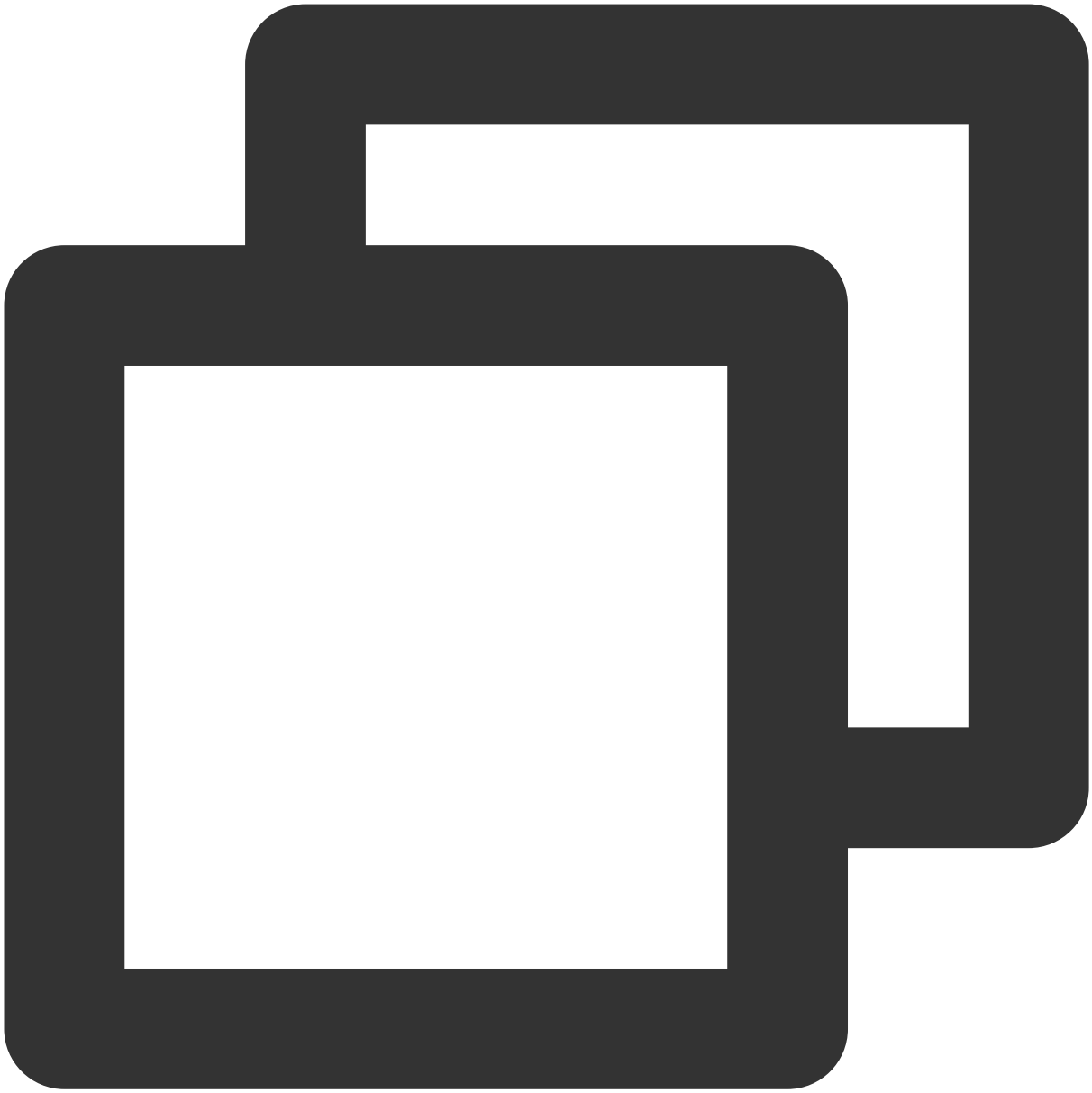


```
from qcloud_cos_v5 import CosConfig
from qcloud_cos_v5 import CosS3Client
```

对于 `cos_sdk_v4` 版本：



```
import qcloud_cos
```



```
from qcloud_cos_v4 import CosClient
from qcloud_cos_v4 import DownloadFileRequest
from qcloud_cos_v4 import UploadFileRequest
```

更详细的 COS SDK 使用说明见 COS Python SDK 说明。

内置的库列表

Python 3 云端运行时已支持的库如下表：

说明：

若您需要使用表中尚未支持的库，请在本地安装并打包上传后使用。详情请参见 安装依赖库。

库名称	版本
absl-py	0.2.2
asn1crypto	0.24.0
astor	0.7.1
bleach	1.5.0
certifi	2019.3.9
cffi	1.12.2
chardet	3.0.4
cos-python-sdk-v5	1.6.6
cryptography	2.6.1
dicttoxml	1.7.4
gast	0.2.0
grpcio	1.13.0
html5lib	0.9999999
idna	2.8
iniparse	0.4
Markdown	2.6.11
mysqlclient	1.3.13
numpy	1.15.0
Pillow	6.0.0
pip	9.0.1
protobuf	3.6.0
psycopg2-binary	2.8.2
pycparser	2.19

pycurl	7.43.0
PyMySQL	0.9.3
pytz	2019.1
qcloud-image	1.0.0
qcloudsms-py	0.1.3
requests	2.21.0
serverless-db-sdk	0.0.1
setuptools	28.8.0
six	1.12.0
tencentcloud-sdk-python	3.0.65
tencentserverless	0.1.4
tensorboard	1.9.0
tensorflow	1.9.0
tensorflow-serving-api	1.9.0
termcolor	1.1.0
urllib3	1.24.2
Werkzeug	0.14.1
wheel	0.31.1

Python 2 云端运行时已支持的库如下表：

库名称	版本
absl-py	0.2.2
asn1crypto	0.24.0
astor	0.7.1
backports.ssl-match-hostname	3.4.0.2
backports.weakref	1.0.post1

bleach	1.5.0
cassdk	1.0.2
certifi	2017.11.5
cffi	1.12.2
chardet	3.0.4
cos-python-sdk-v5	1.6.6
cryptography	2.6.1
dicttoxml	1.7.4
enum34	1.1.6
funcsigs	1.0.2
futures	3.2.0
gast	0.2.0
grpcio	1.13.0
html5lib	0.9999999
idna	2.6
iniparse	0.4
ipaddress	1.0.22
Markdown	2.6.11
mock	2.0.0
mysqlclient	1.3.13
nose	1.3.7
numpy	1.14.5
ordereddict	1.1
pbr	4.1.0
Pillow	6.0.0

pip	18
protobuf	3.6.0
psycopg2-binary	2.8.2
pyaml	2019.4.1
pycparser	2.19
pycurl	7.43.0.1
pygpgme	0.3
PyMySQL	0.9.3
pytz	2019.1
PyYAML	5.1
qcloud-image	1.0.0
qcloudsms-py	0.1.3
requests	2.18.4
serverless-db-sdk	0.0.1
setuptools	39.1.0
six	1.11.0
tencentcloud-sdk-python	3.0.65
tencentserverless	0.1.4
tensorboard	1.9.0
tensorflow	1.9.0
tensorflow-serving-api	1.9.0
termcolor	1.1.0
urlgrabber	3.10.2
urllib3	1.22
Werkzeug	0.14.1

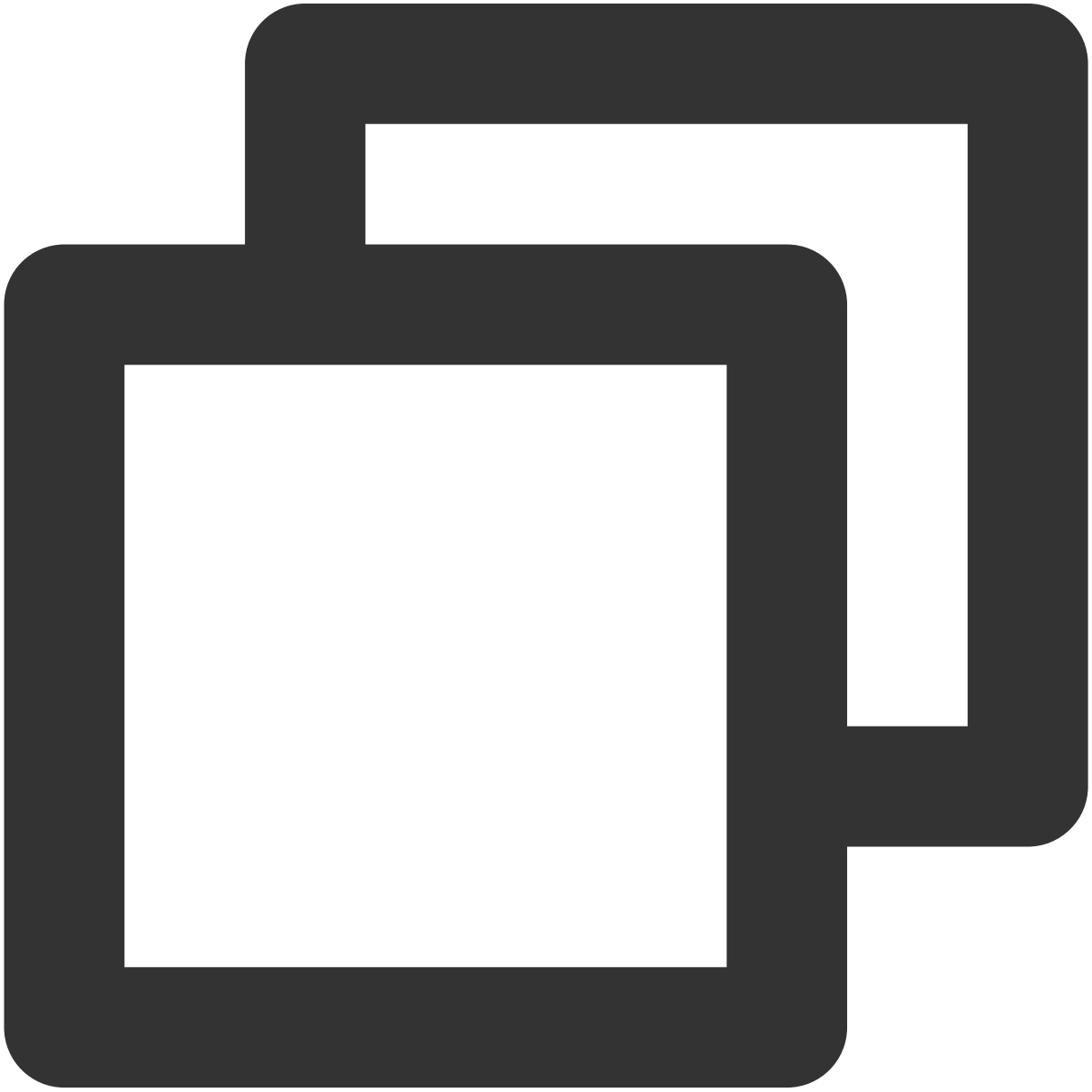
wheel	0.31.1

开发方法

最近更新时间：2024-04-22 18:03:28

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

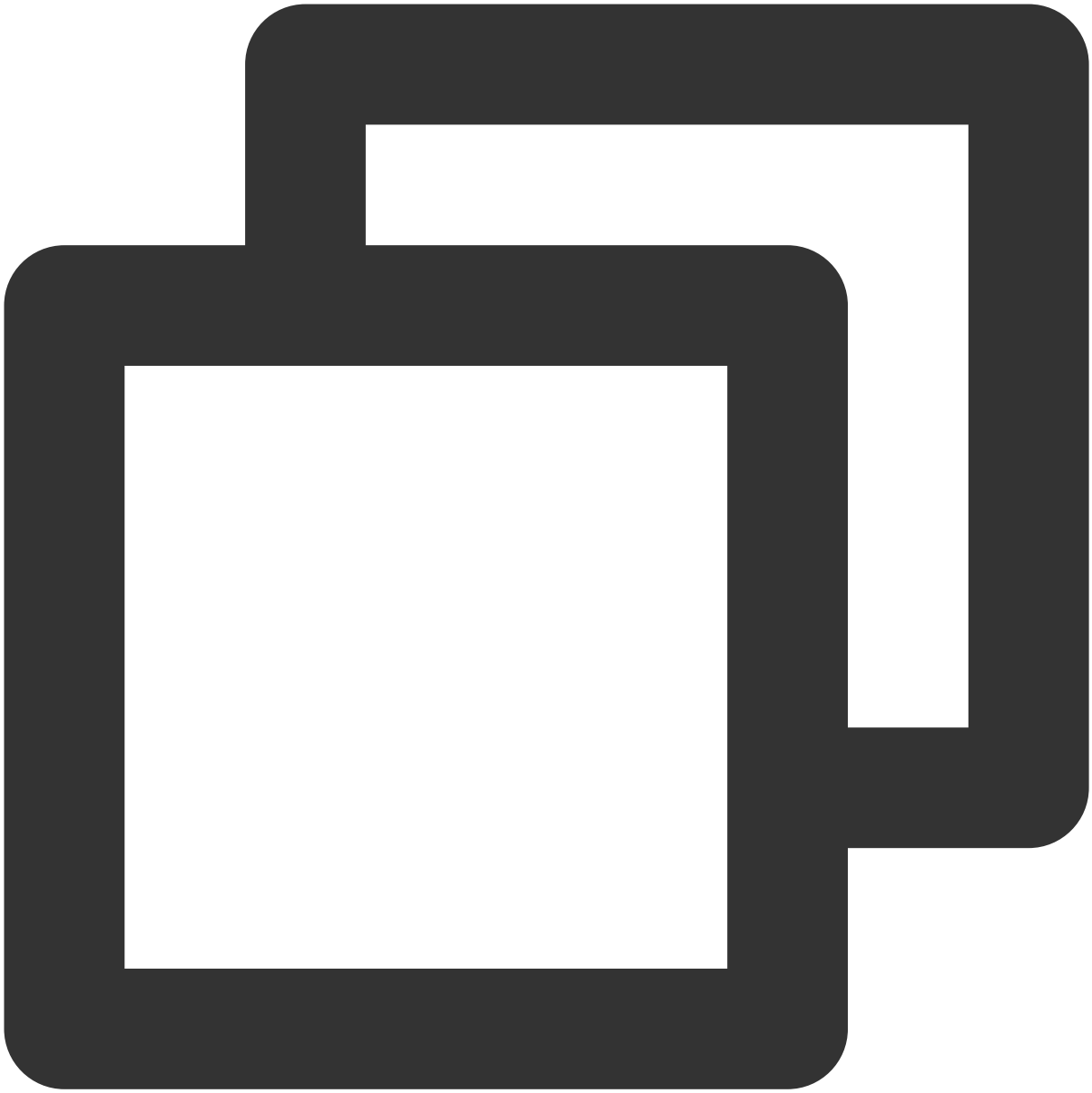
在创建 SCF 云函数时，均需要指定执行方法。使用 Python 开发语言时，执行方法类似 `index.main_handler`，此处 `index` 表示执行的入口文件为 `index.py`，`main_handler` 表示执行的入口函数为 `main_handler` 函数。在使用本地 zip 文件上传、COS 上传等方法提交代码 zip 包时，请确认 zip 包的根目录下包含有指定的入口文件，文件内有定义指定的入口函数，文件名和函数名和执行方法处填写的能够对应，避免因为无法查找到入口文件和入口函数导致的执行失败。

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。
event 的具体内容，根据不同触发器或调用来源而变化。详细数据结构说明请参见 触发器相关说明。

返回

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。
在 Python 环境下，您可以直接返回一个可序列化的对象。例如，返回一个 `dict` 对象：



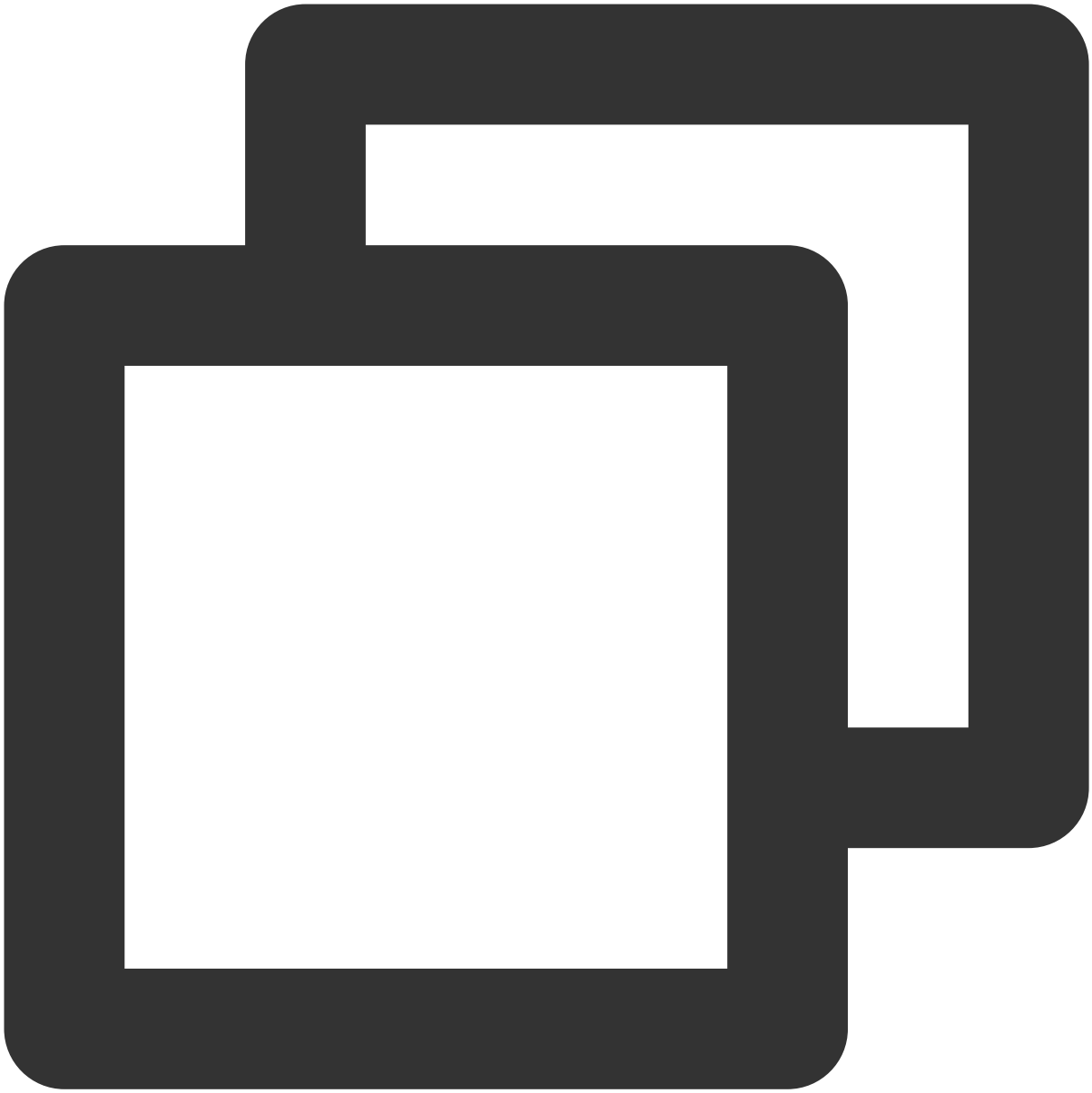
```
def main_handler(event, context):
    resp = {
        "isBase64Encoded": false,
        "statusCode": 200,
        "headers": {"Content-Type":"text/html","Key":["value1","value2","value3"]},
        "body": "<html><body><h1>Heading</h1><p>Paragraph.</p></body></html>"
    }
    return(resp)
```

返回值不同处理方式：

同步调用：使用同步调用时，返回值序列化后以 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。
说明：
无论同步调用还是异步调用，返回值均会在函数的日志中记录。

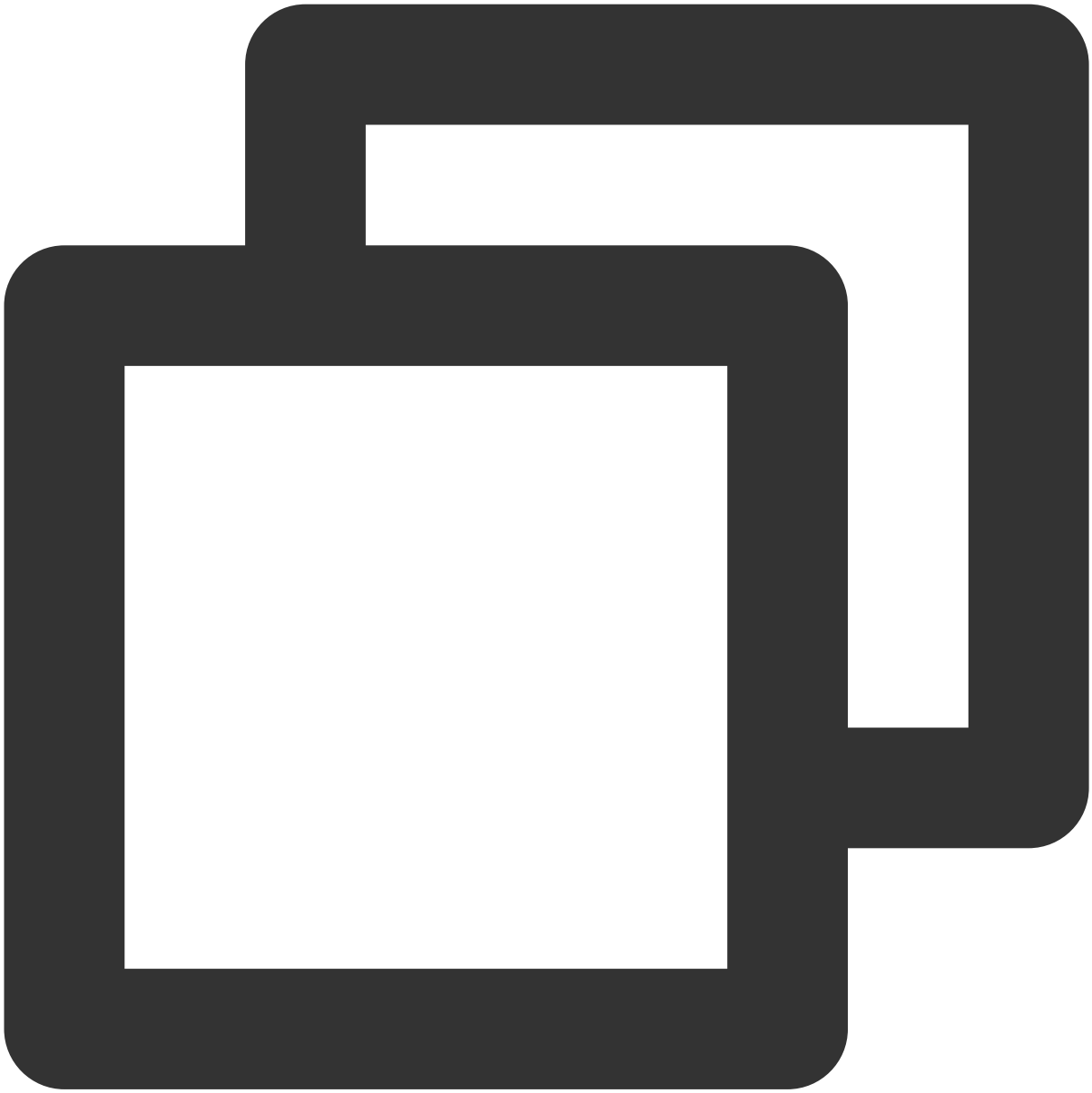
异常处理

您可以在函数内通过使用 `raise Exception` 的方式抛出异常。
如果您在返回前捕捉并处理异常，未继续向外抛出时，函数将认为是成功执行并正常返回入口函数中 `return` 指定的信息。
在如下示例代码中，函数执行成功，则返回值为 `Hello World` 。



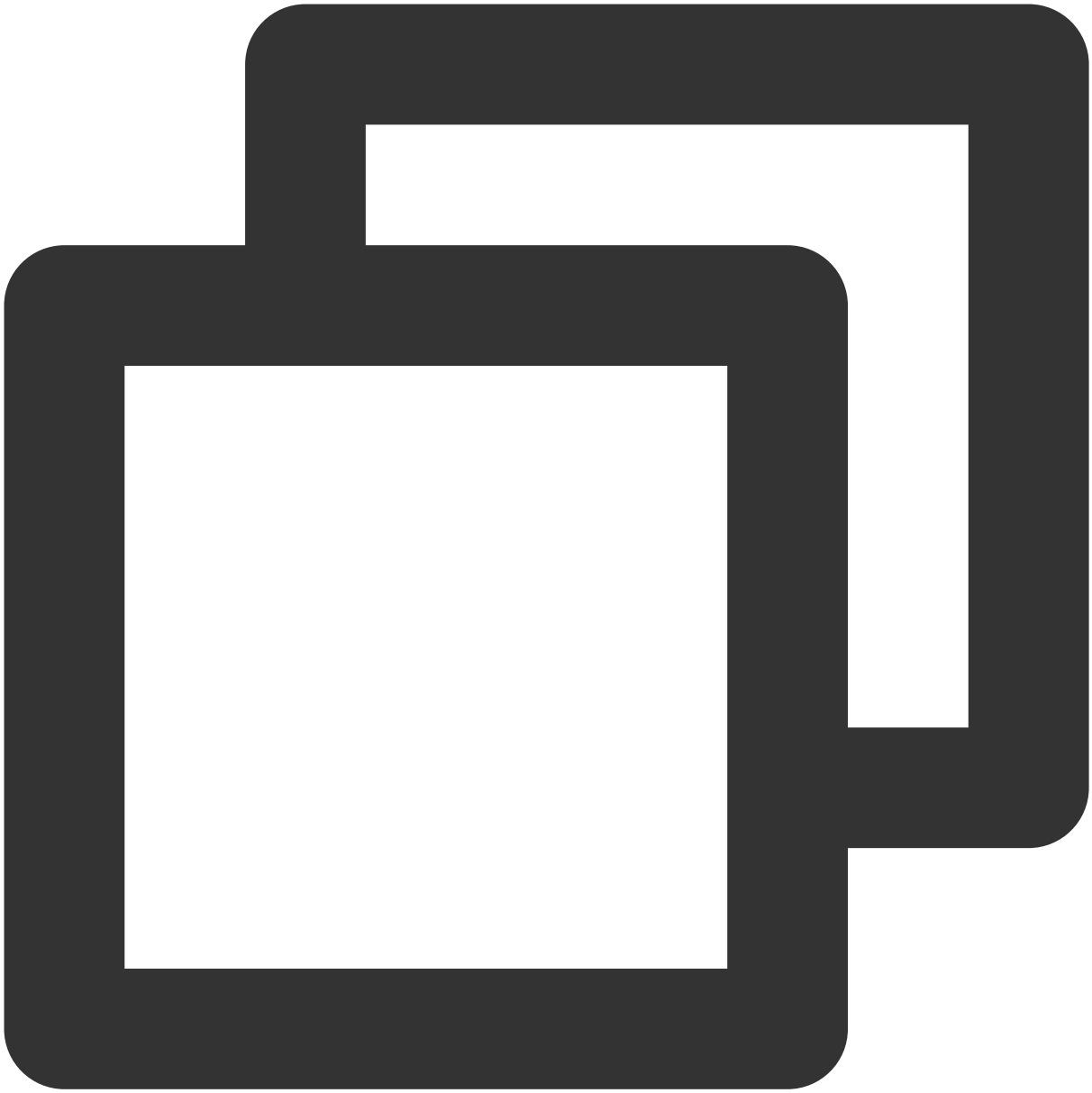
```
# -*- coding: utf8 -*-
def main_handler(event, context):
  try:
      print("try exception")
      raise Exception("err msg")
  except Exception as e:
      print(e)
  return("Hello World")
```

如果在返回前未捕捉异常，异常将会抛出到入口函数之外并由函数平台捕捉。此时函数将认为是执行失败，函数返回信息替换为执行失败信息。
在如下示例代码中，函数执行失败。



```
# -*- coding: utf8 -*-
def main_handler(event, context):
  print("try exception")
  raise Exception("err msg")
  return("Hello World")
```

函数返回信息如下：



```
{
  "errorCode":-1,
  "errorMessage":"user code exception caught",
  "requestId":"a325b967-ef5b-4aa3-a329-c6bb0df72948",
  "stackTrace":"Traceback (most recent call last):\\n  File \\"/var/user/index.py\\
  "statusCode":430
}
```

其中 `errorCode` 字段标识错误为代码错误， `errorMessage` 字段标识错误具体说明， `stackTrace` 字段输出错误堆栈信息， `statusCode` 字段标识具体错误。 `statusCode` 具体说明请参见 云函数状态码。

部署方法

最近更新时间：2024-04-22 18:03:28

部署方法

腾讯云云函数提供以下几种方式部署函数，您可以按需选择使用。创建、更新函数操作详情可参见 创建及更新函数。
通过 zip 打包上传部署，详情可参见 依赖安装和部署。
通过控制台编辑和部署，详情可参见 通过控制台部署函数。
使用命令行部署，详情可参见 通过 Serverless Framework CLI 命令行部署函数。

依赖安装和部署

当前的函数标准 Python Runtime 中仅提供了 `/tmp` 目录可写，其他目录只读，因此在用到依赖库时，需要使用本地安装、打包、上传的方式。Python 依赖包通常可以与函数代码一同上传，或上传至层中，然后绑定使用。

本地安装依赖包

依赖管理工具

Python 可以通过 pip 包管理器进行依赖管理。由于环境配置不同，可自行将 `pip` 替换为 `pip3` 或 `pip2` 。

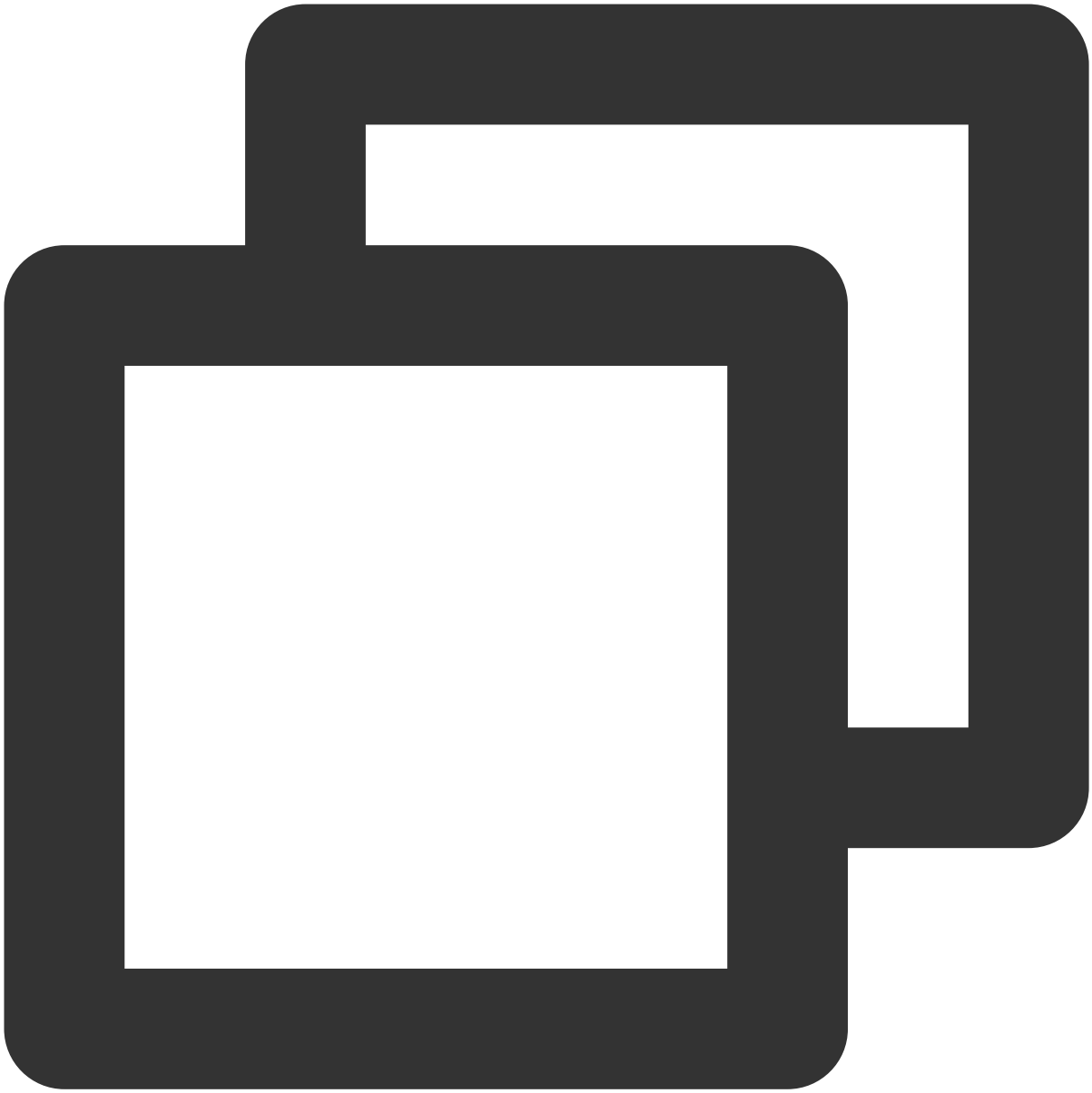
使用方法

1. 在 `requirements.txt` 中配置依赖信息。
2. 通过在代码目录下执行 `pip install -r requirements.txt -t .` 命令安装依赖包。通过使用 `-t` 参数，可以指定依赖包的安装目录，在项目代码目录下执行时，可以使用 `-t .` 安装在当前目录下。

注意：
您可以使用 `pip freeze > requirements.txt` 生成当前环境下所有依赖的 `requirements.txt` 文件。
函数运行的系统是 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找不到依赖的错误。您可参考 云函数容器镜像 进行依赖安装。
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。详情请参阅 使用 Docker 安装依赖。

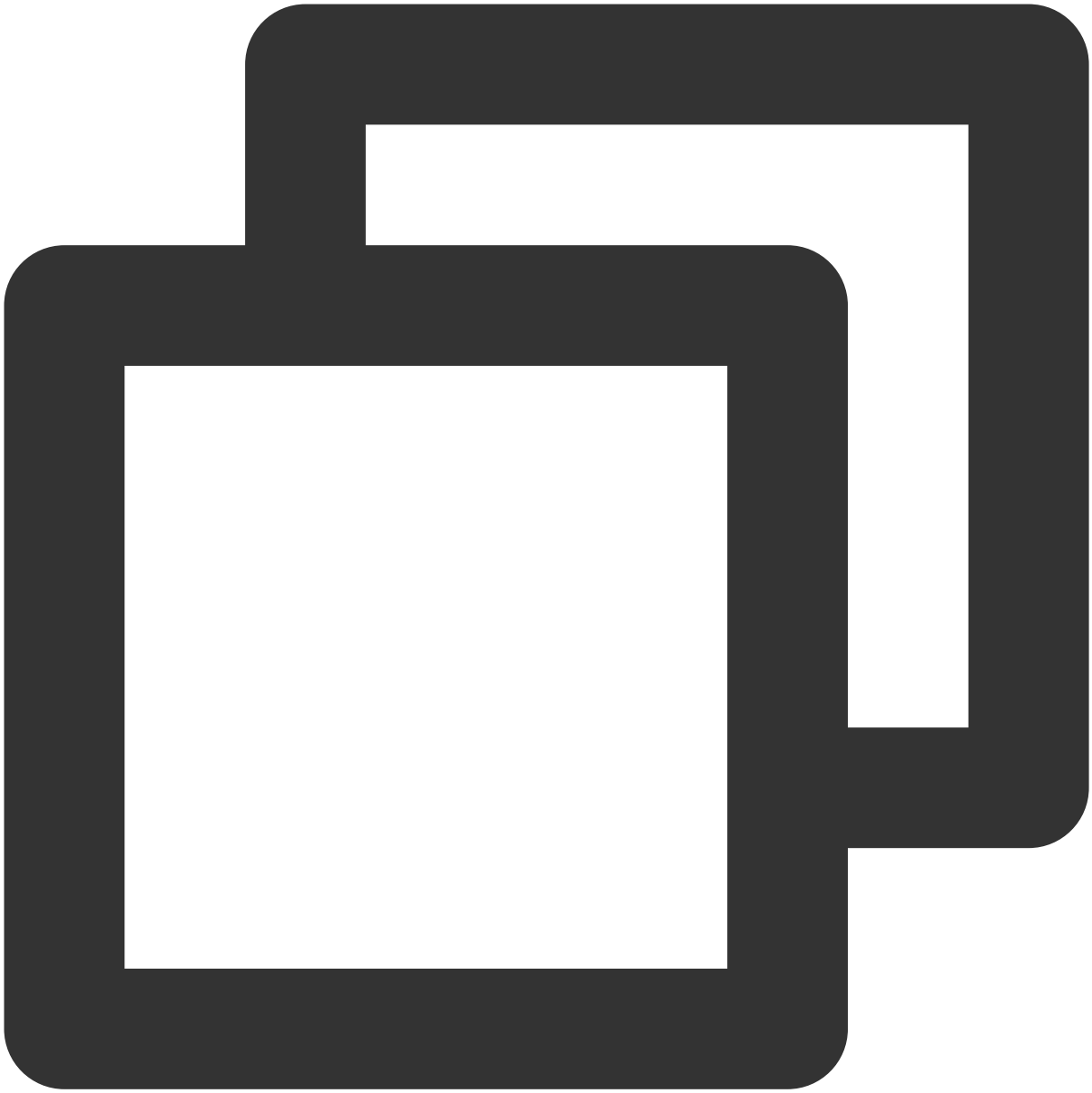
示例

1. 在本地安装 `requests` 依赖，其中代码文件 `index.py` 如下所示：



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

def main_handler(event, context):
    addr = "www.qq.com"
    resp = requests.get(addr)
    print(resp)
    return resp
```

2. 使用 `pip3 install requests -t .` 在项目当前目录安装 `requests` 依赖。代码文件如下所示：



```
$ pip3 install requests -t .
Collecting requests
      Using cached requests-2.25.1-py2.py3-none-any.whl (61 kB)
Collecting certifi>=2017.4.17
      Using cached certifi-2020.12.5-py2.py3-none-any.whl (147 kB)
Collecting chardet<5,>=3.0.2
      Using cached chardet-4.0.0-py2.py3-none-any.whl (178 kB)
Collecting idna<3,>=2.5
      Using cached idna-2.10-py2.py3-none-any.whl (58 kB)
Collecting urllib3<1.27,>=1.21.1
      Using cached urllib3-1.26.4-py2.py3-none-any.whl (153 kB)
```

```
Installing collected packages: urllib3, idna, chardet, certifi, requests
Successfully installed certifi-2020.12.5 chardet-4.0.0 idna-2.10 requests-2.25.1 ur

$ ls -l
total 8
drwxr-xr-x   3 xxx  111    96  4 29 16:45 bin
drwxr-xr-x   7 xxx  111   224  4 29 16:45 certifi
drwxr-xr-x   8 xxx  111   256  4 29 16:45 certifi-2020.12.5.dist-info
drwxr-xr-x  44 xxx  111  1408  4 29 16:45 chardet
drwxr-xr-x   9 xxx  111   288  4 29 16:45 chardet-4.0.0.dist-info
drwxr-xr-x  11 xxx  111   352  4 29 16:45 idna
drwxr-xr-x   8 xxx  111   256  4 29 16:45 idna-2.10.dist-info
-rw-r--r--@  1 xxx  111   177  4 29 16:33 index.py
drwxr-xr-x  21 xxx  111   672  4 29 16:45 requests
drwxr-xr-x   9 xxx  111   288  4 29 16:45 requests-2.25.1.dist-info
drwxr-xr-x  17 xxx  111   544  4 29 16:45 urllib3
drwxr-xr-x  10 xxx  111   320  4 29 16:45 urllib3-1.26.4.dist-info
```

打包上传

依赖可以和项目一同上传，并在函数代码中通过 `import` 方式引入和使用。同时，依赖也可以打包部署为层，并通过在函数创建部署时，与函数绑定，提供复用能力。

您可以通过控制台选择本地文件夹的方式自动化打包，也可以通过手工打包的方式形成可以用于部署函数或层的 zip 包。在打包部署时，需要注意的是均在项目目录下进行打包操作，即确保代码、依赖均在 zip 文件内的根目录中。详情可参见 打包要求。

特殊依赖包

部分 Python 的依赖包，在安装时需要进行相关的编译操作，例如 `pycryptodome` 依赖。由于编译程序会根据不同 OS 进行操作系统相关的编译操作，在 Windows、Mac 等环境下编译生成的依赖库及动态库等程序，可能无法在云函数的环境中运行。您可通过以下方案尝试解决：

寻找针对 FaaS 的开源实现，使用开源实现已经准备好的依赖库。

在 SCF 公共 Layer 项目中寻找依赖或提交需求，本项目用于常用特殊依赖包的整理和存储，并以 Layer 形式提供部署支持。

通过使用容器方案及 SCF 容器镜像，在本地完成特殊依赖包安装和内容抽取后，打包上传至代码运行环境中。

日志说明

最近更新时间：2024-04-22 18:03:28

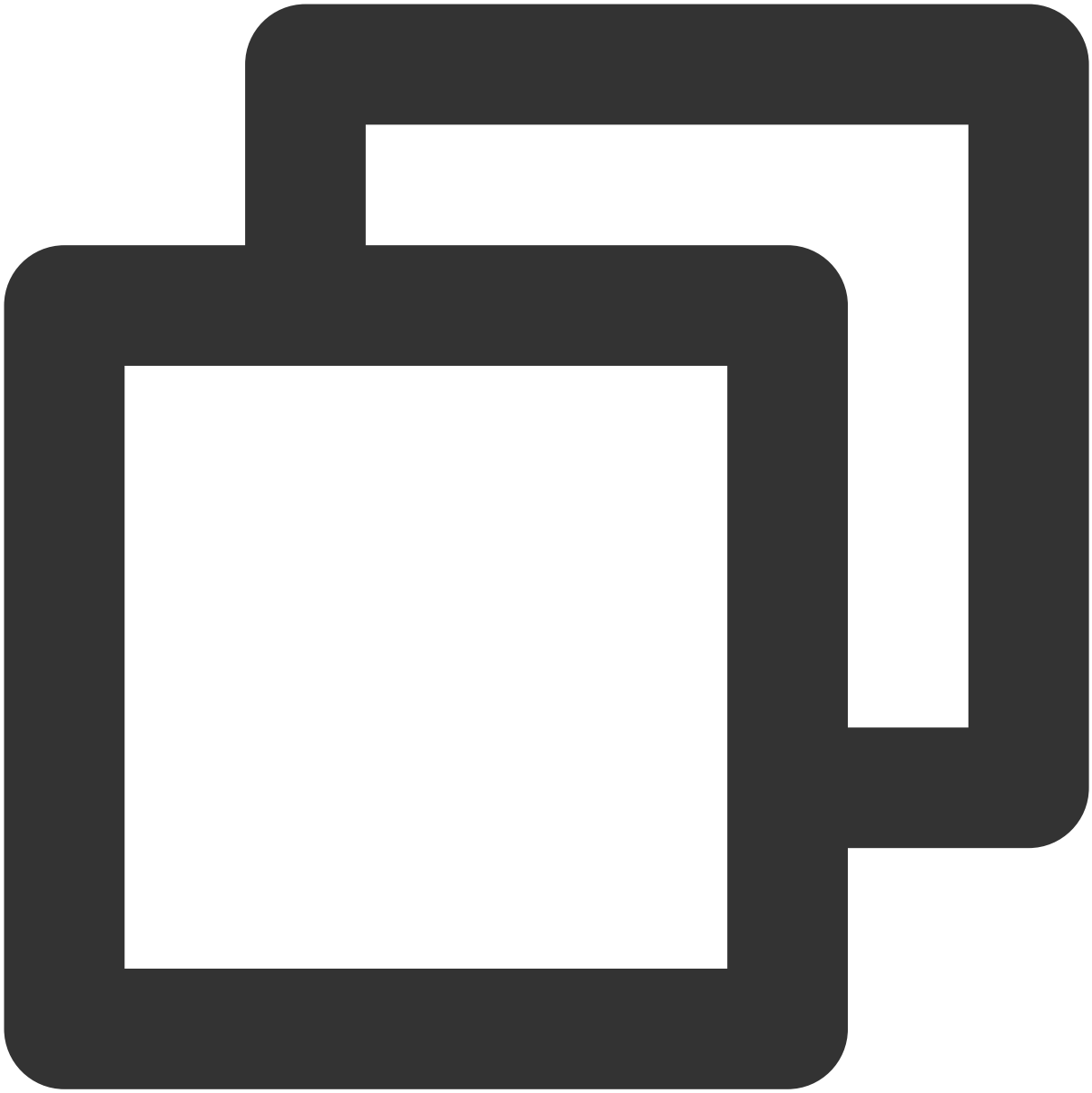
日志开发

您可以在程序中使用以下语句来完成日志输出：

print

logging 模块

例如，执行以下代码，您可在函数日志中查询输出内容。



```
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!'
```

日志查询

当前函数日志均会投递至腾讯云日志服务 CLS 中，您可对函数日志进行投递配置，详情可参见 日志投递配置。
您可通过云函数的日志查询界面或通过日志服务的查询界面，查询函数执行日志。日志查询方法详情可参见 日志检索教程。

说明：

函数日志投递到日志服务日志集 LogSet 和日志主题 LogTopic，均可以通过函数配置查询。

自定义日志字段

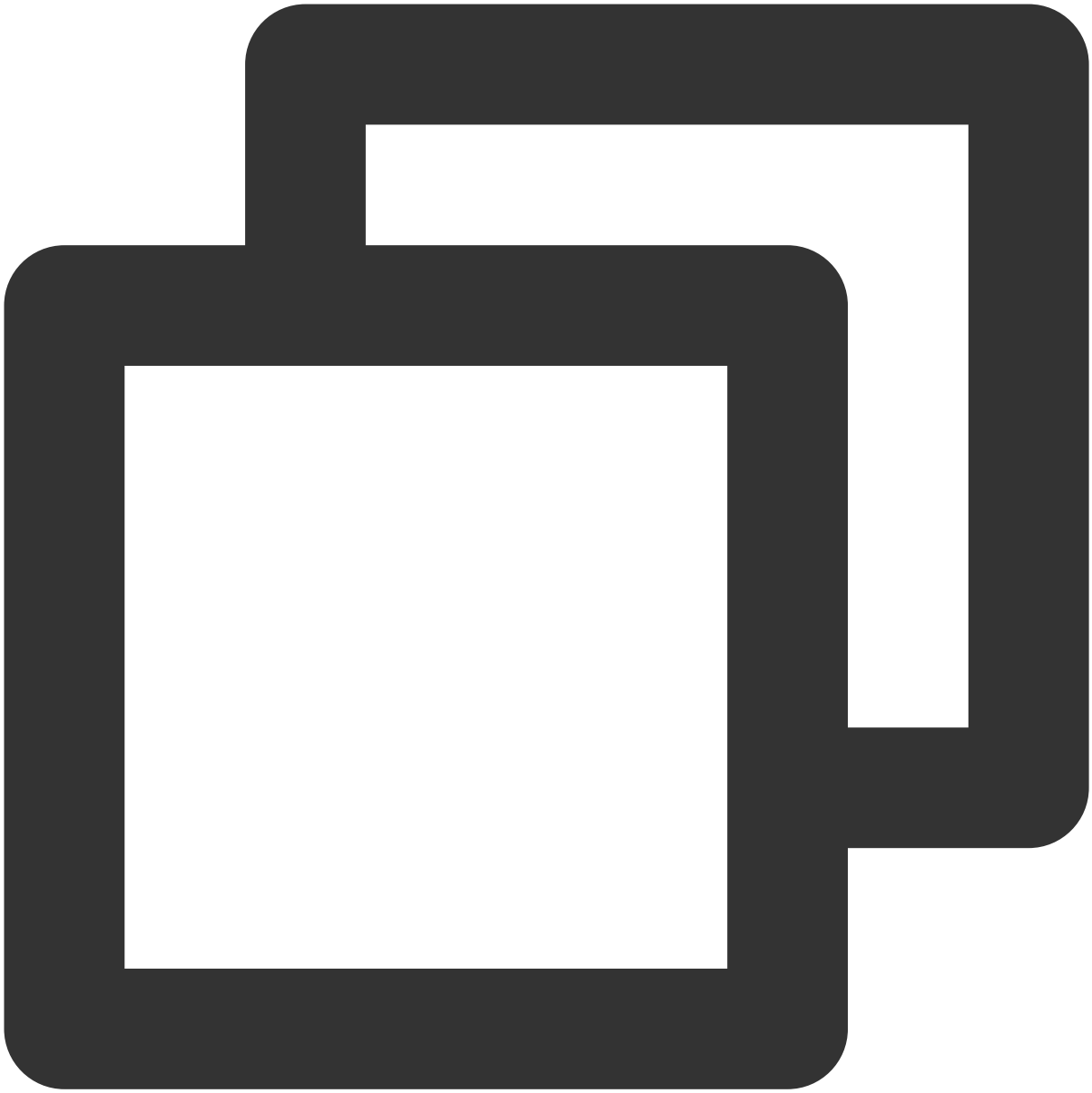
当前在函数代码中通过简单的 `print`，或通过 `logger` 输出的字符串内容，将会在投递到日志服务时，记录在 `SCF_Message` 字段中。日志服务的字段说明可参见 索引说明。
目前云函数已经支持在输出到日志服务的内容中增加自定义字段，通过增加自定义字段，您可以将业务字段及相关数据内容输出到日志中，并通过使用日志服务的检索能力，对执行过程中的业务数据及相关内容进行查询跟踪。

注意：

如需对自定义字段进行键值查询，如 `SCF_CustomKey：SCF`，请参考 日志服务索引配置 为函数日志投递的日志主题添加键值索引。
为避免误操作索引配置导致函数日志查询失败，函数配置的默认投递日志主题（以 `SCF_LogTopic_` 为前缀命名）不支持修改索引配置。请将函数日志投递主题设置为 自定义投递 后再更新日志主题索引配置。
日志主题修改索引配置后，仅对新写入的数据有效。

输出方法

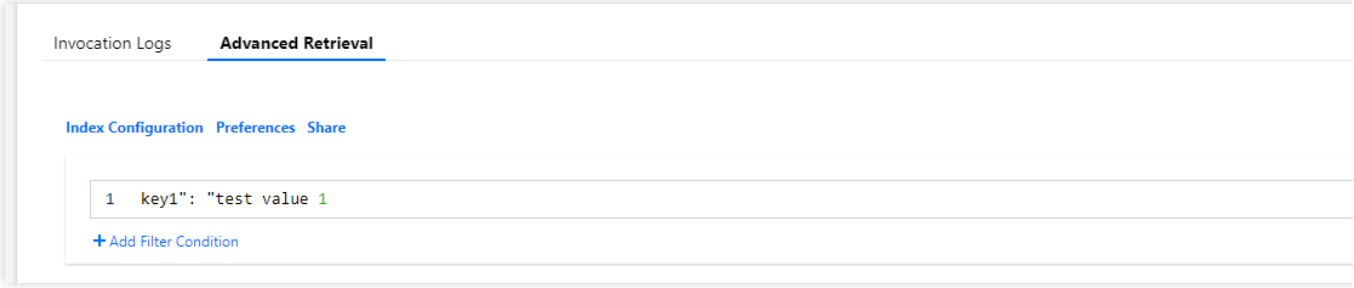
当函数输出的单行日志为 JSON 格式时，JSON 内容将被解析并在投递至日志服务时按 `字段:值` 的方式进行投递。JSON 内容的解析仅能解析第一层，更多的嵌套结构将作为值进行记录。
您可执行以下代码进行测试：



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

def main_handler(event, context):
    print(json.dumps({"key1": "test value 1","key2": "test value 2"}))
    return("Hello World!")
```

检索方法

在使用上述代码进行测试运行后，您可在函数-日志查询-高级检索中通过如下语句进行检索：



检索结果

在测试写入日志服务后，您可以在日志查询中检索到 `key1` 字段。如下图所示：

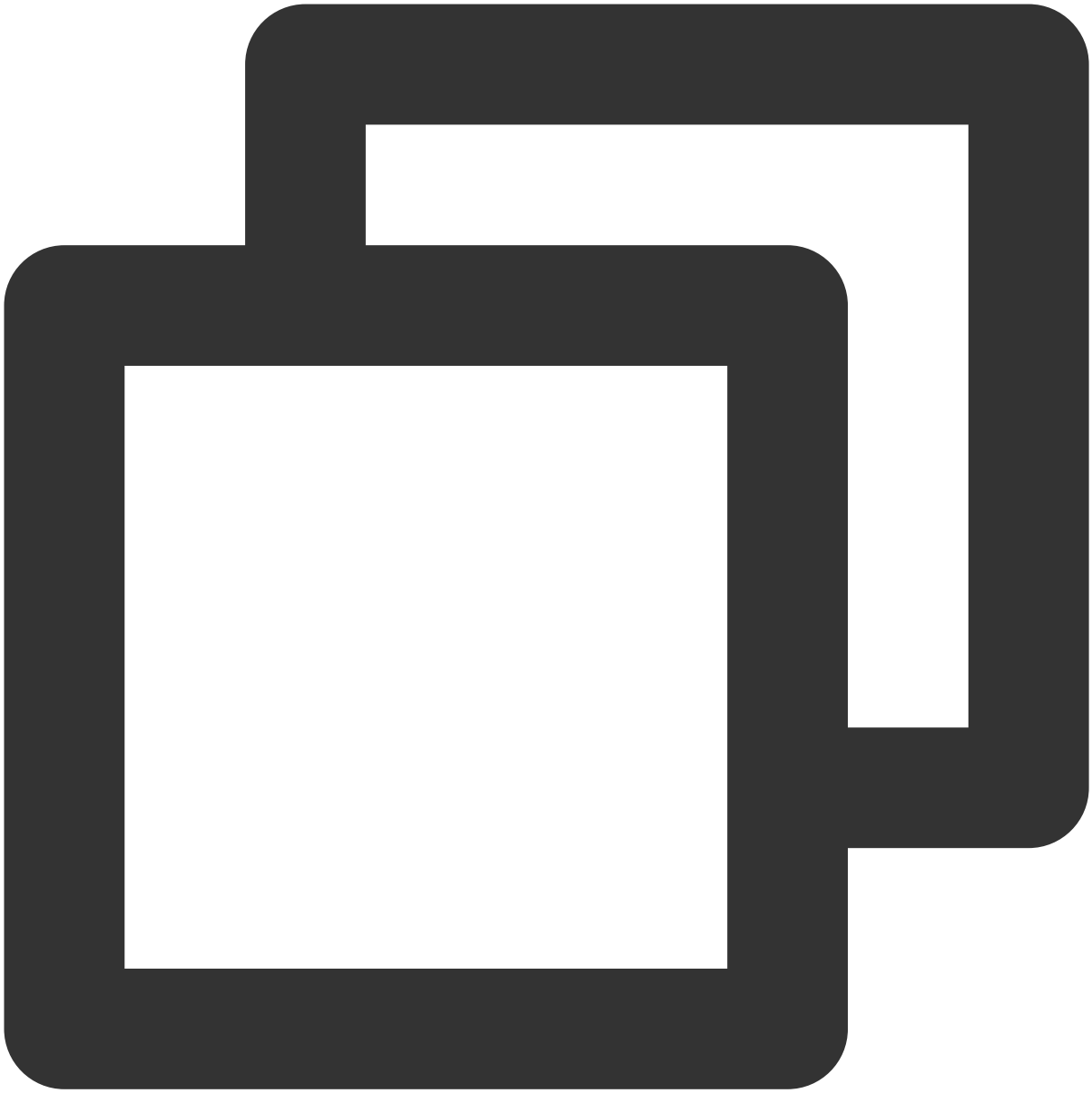
常见示例

最近更新时间：2024-04-22 18:03:28

常见示例中包含了 Python 环境下可以试用的相关代码片段，您可以根据需要选择尝试。示例均基于 Python 3.6 环境提供。
您可从 github 项目 scf-python-code-snippet 中获取相关代码片段并直接部署。

环境变量读取

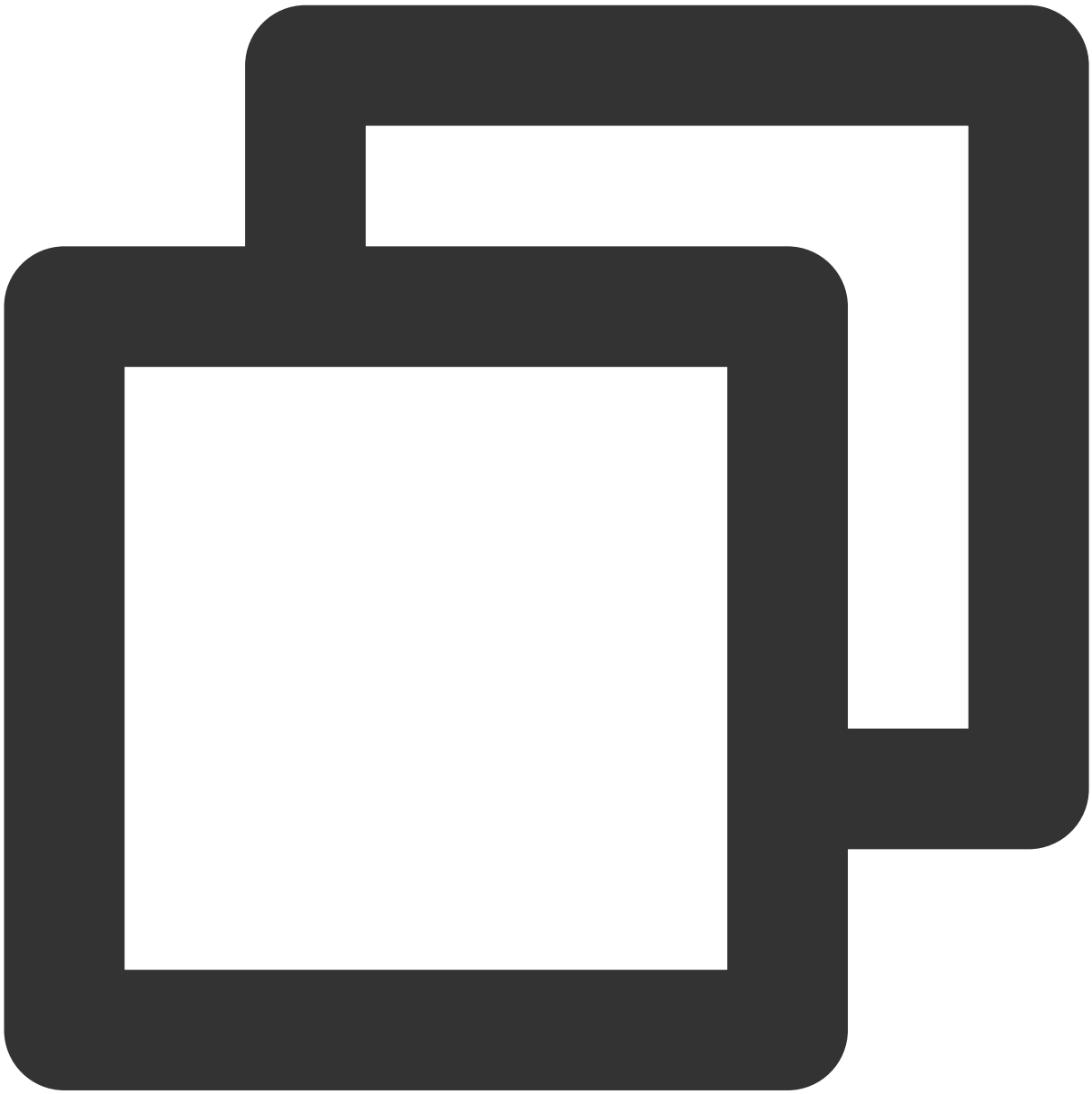
本示例提供了获取全部环境变量列表，或单一环境变量值的方法。



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

def main_handler(event, context):
      print(os.environ)
      print(os.environ.get("SCF_RUNTIME"))
      return("Hello World")
```

本地时间格式化输出

本示例提供了时间格式化输出方法，按指定格式进行日期和时间输出。
SCF 环境默认是 UTC 时间，如果期望按北京时间输出，可以为函数添加 `TZ=Asia/Shanghai` 环境变量。

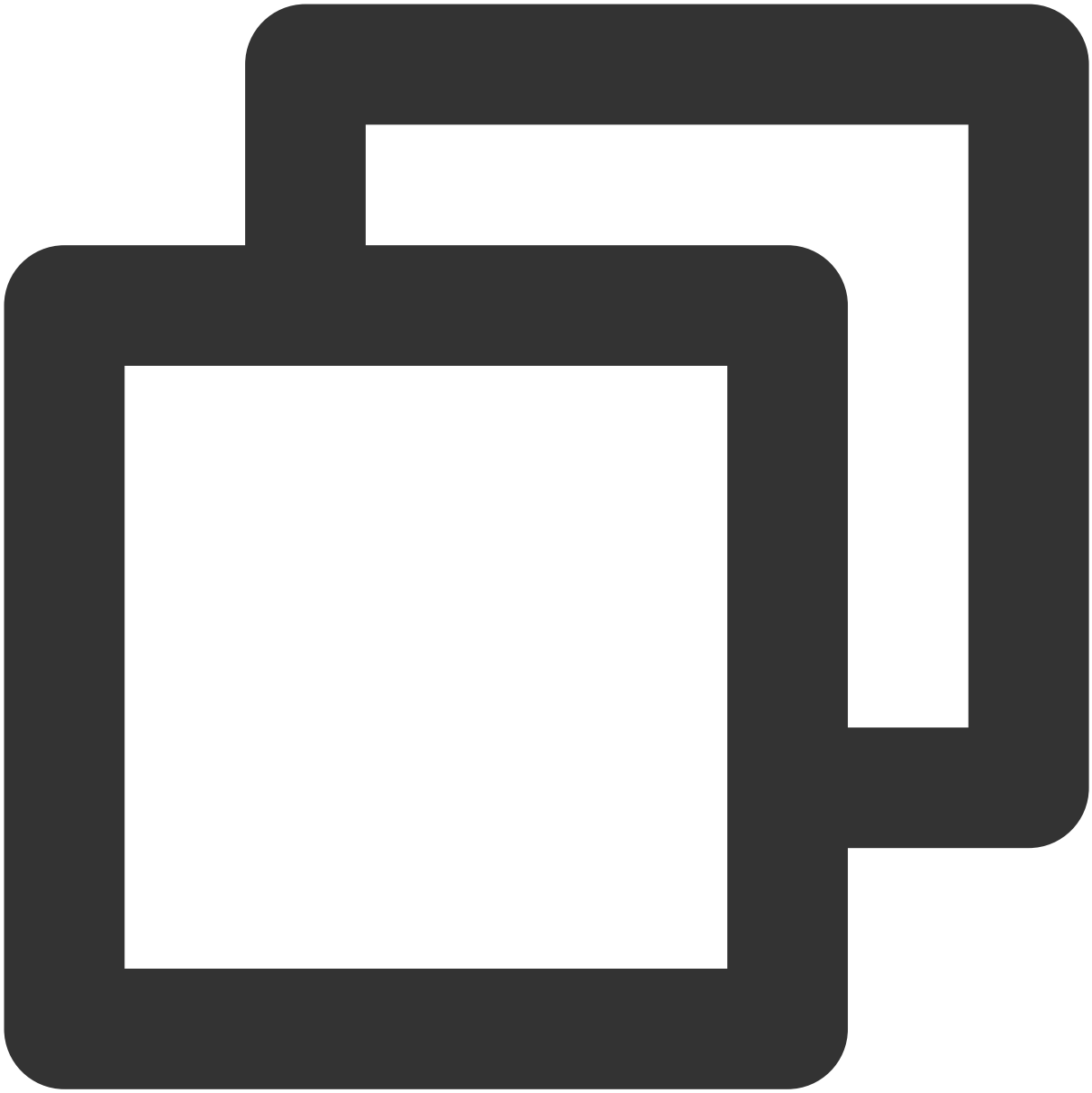


```
# -*- coding: utf8 -*-
import time

def main_handler(event, context):
      print(time.strftime('%Y-%m-%d %H:%M:%S',time.localtime(time.time())))
```

```
        return("Hello World")
```

函数访问 MySQL 数据库

本示例使用了 PyMySQL 库来进行数据库连接，在项目目录下需要执行 `pip3 install PyMySQL -t .` 命令完成依赖库安装。
在使用本示例时：
需要注意函数网络配置，将函数网络配置到 MySQL 数据库所在的 VPC 中，确保网络可达。
根据数据库具体情况，修改代码中的数据库连接 IP、用户名、密码、数据库名等信息。

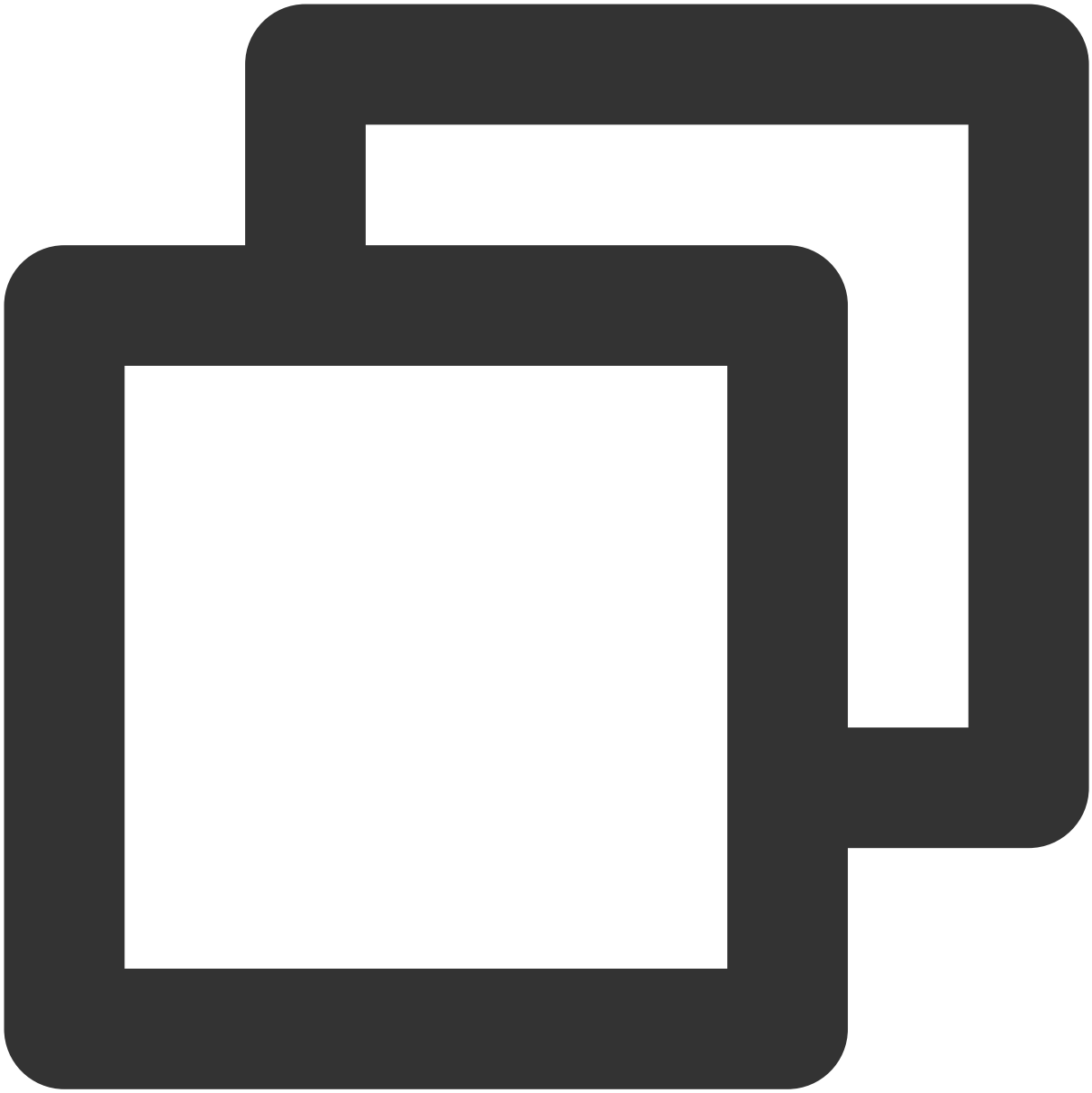


```
# -*- coding: utf8 -*-
import pymysql

def main_handler(event, context):

        # 打开数据库连接
        db = pymysql.connect(host="host ip",port=3306,user="user",password="password

        # 使用 cursor() 方法创建一个游标对象 cursor
        cursor = db.cursor()
```

```
        # 使用 execute()  方法执行 SQL 查询
        cursor.execute("SELECT VERSION()")

        # 使用 fetchone() 方法获取单条数据.
        data = cursor.fetchone()

        print ("Database version : %s " % data)

        # 关闭数据库连接
        db.close()
```

函数内发起网络连接

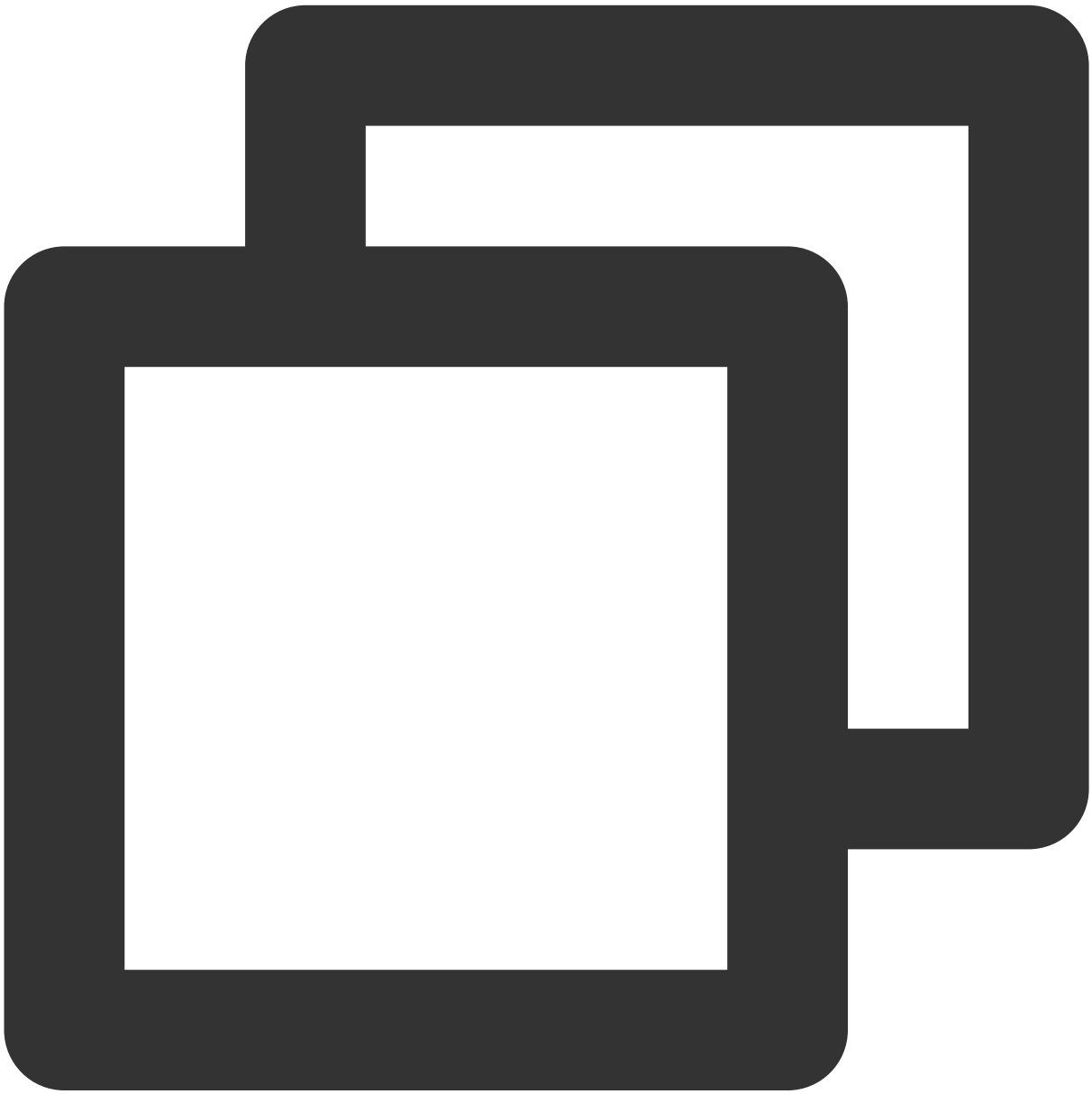
本示例使用了 requests 库在函数内发起网络连接，获取页面信息。可以通过在项目目录下执行 `pip3 install requests -t .` 命令完成依赖库安装。



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


def main_handler(event, context):
      addr = "https://cloud.tencent.com"
      resp = requests.get(addr)
      print(resp)
      print(resp.text)
      return resp.status_code
```

函数 + API 网关返回网页

通过配置 API 网关触发器并启用集成响应，可以实现 API 网关 URL 访问时获取 html 页面。

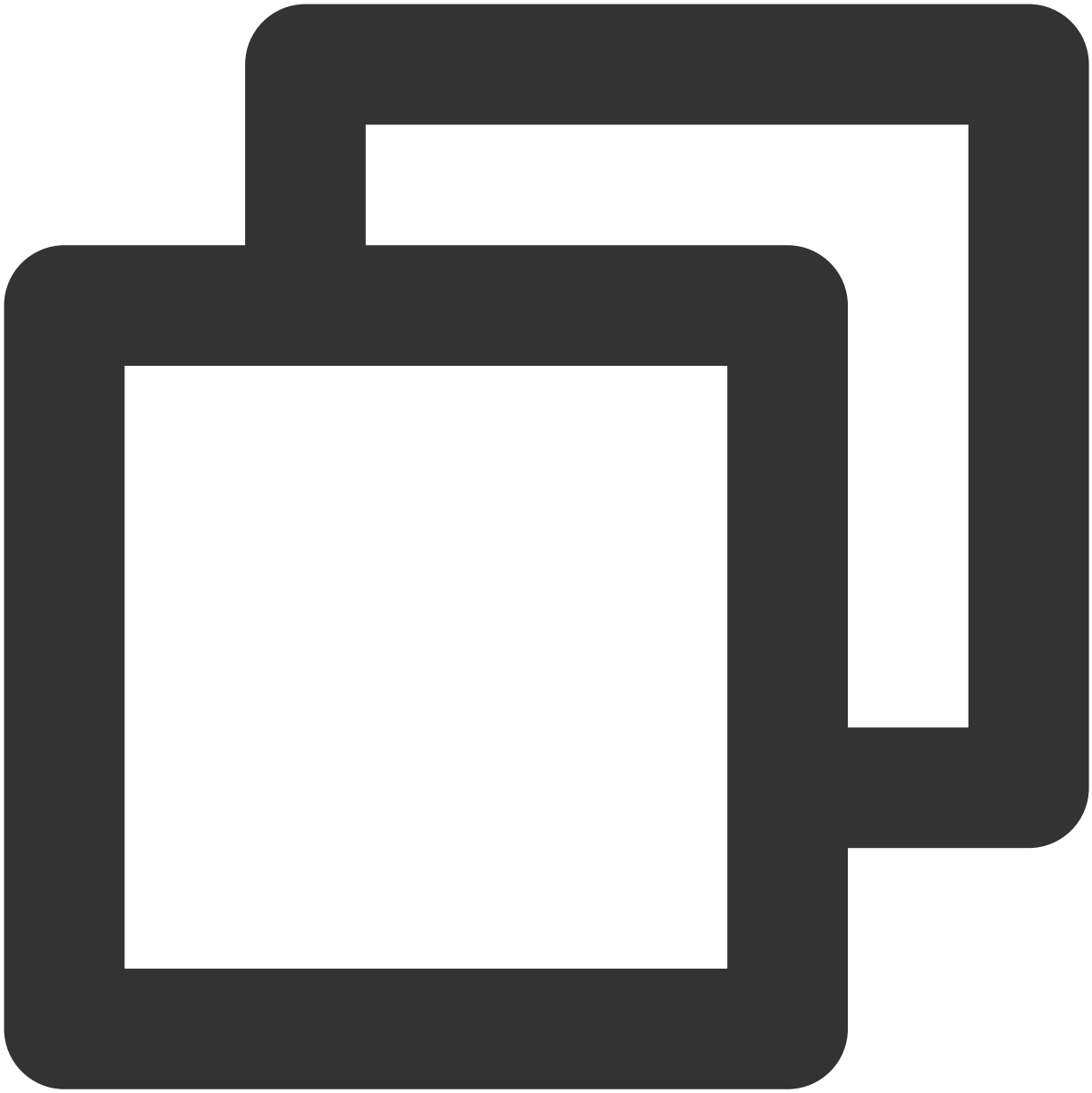


```
# -*- coding: utf8 -*-
import time

def main_handler(event, context):
```

```
        resp = {
            "isBase64Encoded": False,
            "statusCode": 200,
            "headers": {"Content-Type":"text/html"},
            "body": "<html><body><h1>Hello</h1><p>Hello World.</p></body></html>"
        }
        return resp
```

函数 + API 网关返回图片

通过配置 API 网关触发器并启用集成响应，可以实现 API 网关 URL 访问时获取到二进制图片文件。



```
# -*- coding: utf8 -*-
import base64

def main_handler(event, context):
        with open("tencent_cloud_logo.png","rb") as f:
            data = f.read()
        base64_data = base64.b64encode(data)
        base64_str = base64_data.decode('utf-8')
        resp = {
            "isBase64Encoded": True,
            "statusCode": 200,
```

```
            "headers": {"Content-Type":"image/png"},
            "body": base64_str
        }
        return resp
```

Node.js 环境说明

最近更新时间：2024-04-22 18:03:28

Node.js 版本选择

目前支持的 Node.js 开发语言包括如下版本：

Node.js 16.13

Node.js 14.18

Node.js 12.16

Node.js 10.15

Node.js 8.9（即将下线）

Node.js 6.10（即将下线）

您可以在函数创建时，选择您所期望使用的运行环境。

相关环境变量

目前 Node.js 运行环境中内置的相关环境变量见下表：

Node.js 版本	环境变量 Key	具体值或值来源
Node.js 16.13	`NODE_PATH`	/var/user:/var/user/node_modules:/var/lang/node16/lib/node_modules:/opt:/opt/node
Node.js 14.18	`NODE_PATH`	/var/user:/var/user/node_modules:/var/lang/node14/lib/node_modules:/opt:/opt/node
Node.js 12.16	`NODE_PATH`	/var/user:/var/user/node_modules:/var/lang/node12/lib/node_modules:/opt:/opt/node
Node.js 10.15	`NODE_PATH`	/var/user:/var/user/node_modules:/var/lang/node10/lib/node_modules:/opt:/opt/node
Node.js 8.9	`NODE_PATH`	/var/user:/var/user/node_modules:/var/lang/node8/lib/node_modules:/opt:/opt/node_
Node.js 6.10	`NODE_PATH`	/var/user:/var/user/node_modules:/var/lang/node6/lib/node_modules:/opt:/opt/node_

更多详细环境变量说明请参见 环境变量说明。

已包含的库及使用方法

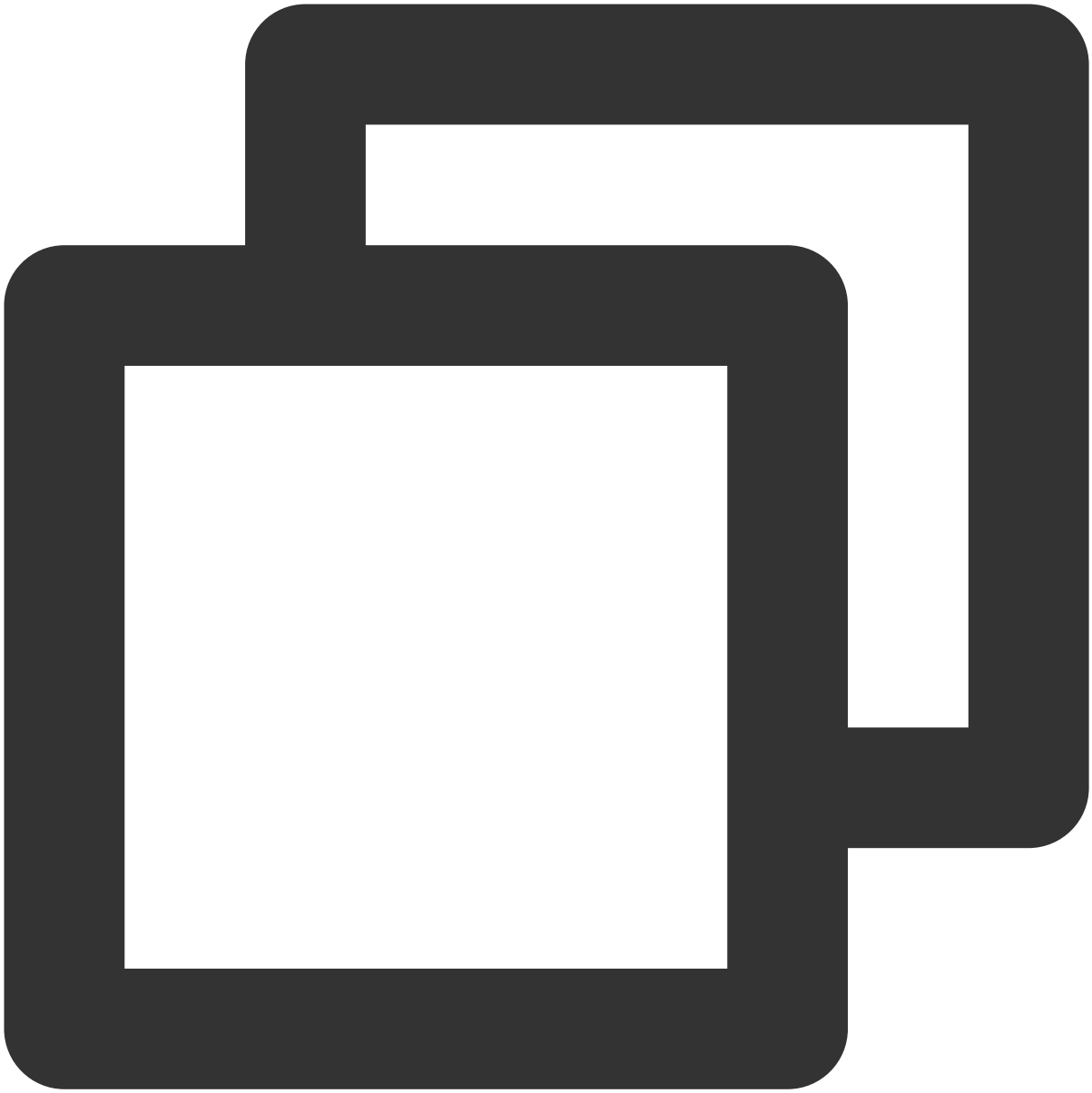
注意：

Node.js 14.18 及之后版本，平台不再额外内置依赖库。代码运行所需依赖，请参考 依赖安装 及 在线依赖安装。

COS SDK

云函数 Node.js 12.16 及更早版本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 使用说明请参见 COS Node.js SDK。

环境内的内置库

Node.js 各版本运行时内已支持的库如下表：

Node.js 12.16

Node.js 10.15

Node.js 8.9

Node.js 6.10

库名称	版本
cos-nodejs-sdk-v5	2.5.20
base64-js	1.3.1
buffer	5.5.0
crypto-browserify	3.12.0
ieee754	1.1.13
imagemagick	0.1.3
isarray	2.0.5
jmespath	0.15.0
lodash	4.17.15
microtime	3.0.0
npm	6.13.4
punycode	2.1.1
puppeteer	2.1.1
qcloudapi-sdk	0.2.1
querystring	0.2.0
request	2.88.2
sax	1.2.4
scf-nodejs-serverlessdb-sdk	1.1.0
tencentcloud-sdk-nodejs	3.0.147
url	0.11.0
uuid	7.0.3
xml2js	0.4.23
xmlbuilder	15.1.0

库名称	版本
cos-nodejs-sdk-v5	2.5.14
base64-js	1.3.1
buffer	5.4.3
crypto-browserify	3.12.0
ieee754	1.1.13
imagemagick	0.1.3
isarray	2.0.5
jmespath	0.15.0
lodash	4.17.15
microtime	3.0.0
npm	6.4.1
punycode	2.1.1
puppeteer	2.0.0
qcloudapi-sdk	0.2.1
querystring	0.2.0
request	2.88.0
sax	1.2.4
scf-nodejs-serverlessdb-sdk	1.0.1
tencentcloud-sdk-nodejs	3.0.104
url	0.11.0
uuid	3.3.3
xml2js	0.4.22
xmlbuilder	13.0.2

库名称	版本
cos-nodejs-sdk-v5	2.5.8
base64-js	1.2.1
buffer	5.0.7
crypto-browserify	3.11.1
ieee754	1.1.8
imagemagick	0.1.3
isarray	2.0.2
jmespath	0.15.0
lodash	4.17.4
npm	5.6.0
punycode	2.1.0
puppeteer	1.14.0
qcloudapi-sdk	0.1.5
querystring	0.2.0
request	2.87.0
sax	1.2.4
tencentcloud-sdk-nodejs	3.0.56
url	0.11.0
uuid	3.1.0
xml2js	0.4.17
xmlbuilder	9.0.1

库名称	版本

base64-js	1.2.1
buffer	5.0.7
cos-nodejs-sdk-v5	2.0.7
crypto-browserify	3.11.1
ieee754	1.1.8
imagemagick	0.1.3
isarray	2.0.2
jmespath	0.15.0
lodash	4.17.4
npm	3.10.10
punycode	2.1.0
qcloudapi-sdk	0.1.5
querystring	0.2.0
request	2.87.0
sax	1.2.4
tencentcloud-sdk-nodejs	3.0.10
url	0.11.0
uuid	3.1.0
xml2js	0.4.17
xmlbuilder	9.0.1

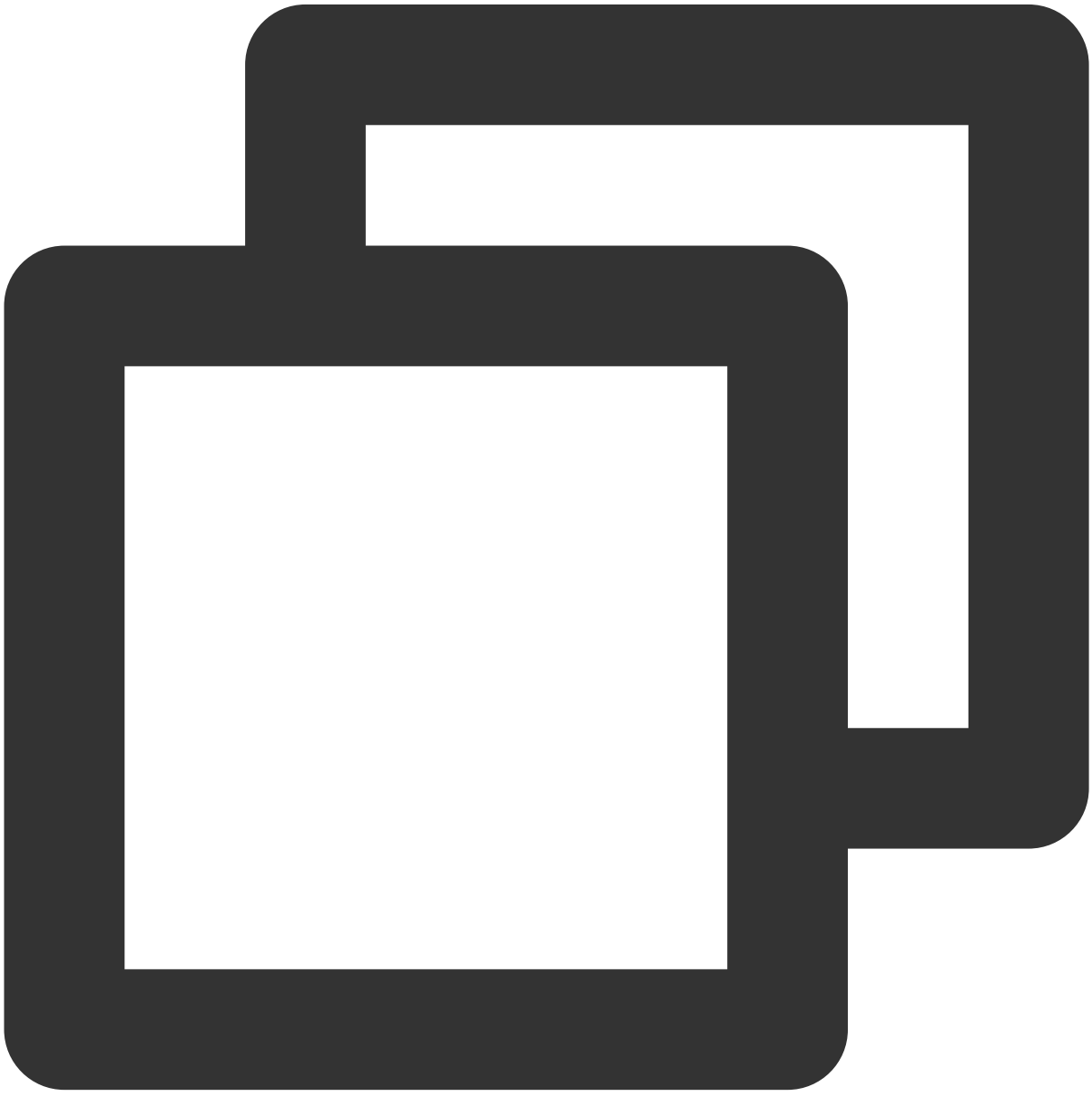
开发方法

最近更新时间：2024-04-22 18:03:28

函数形态

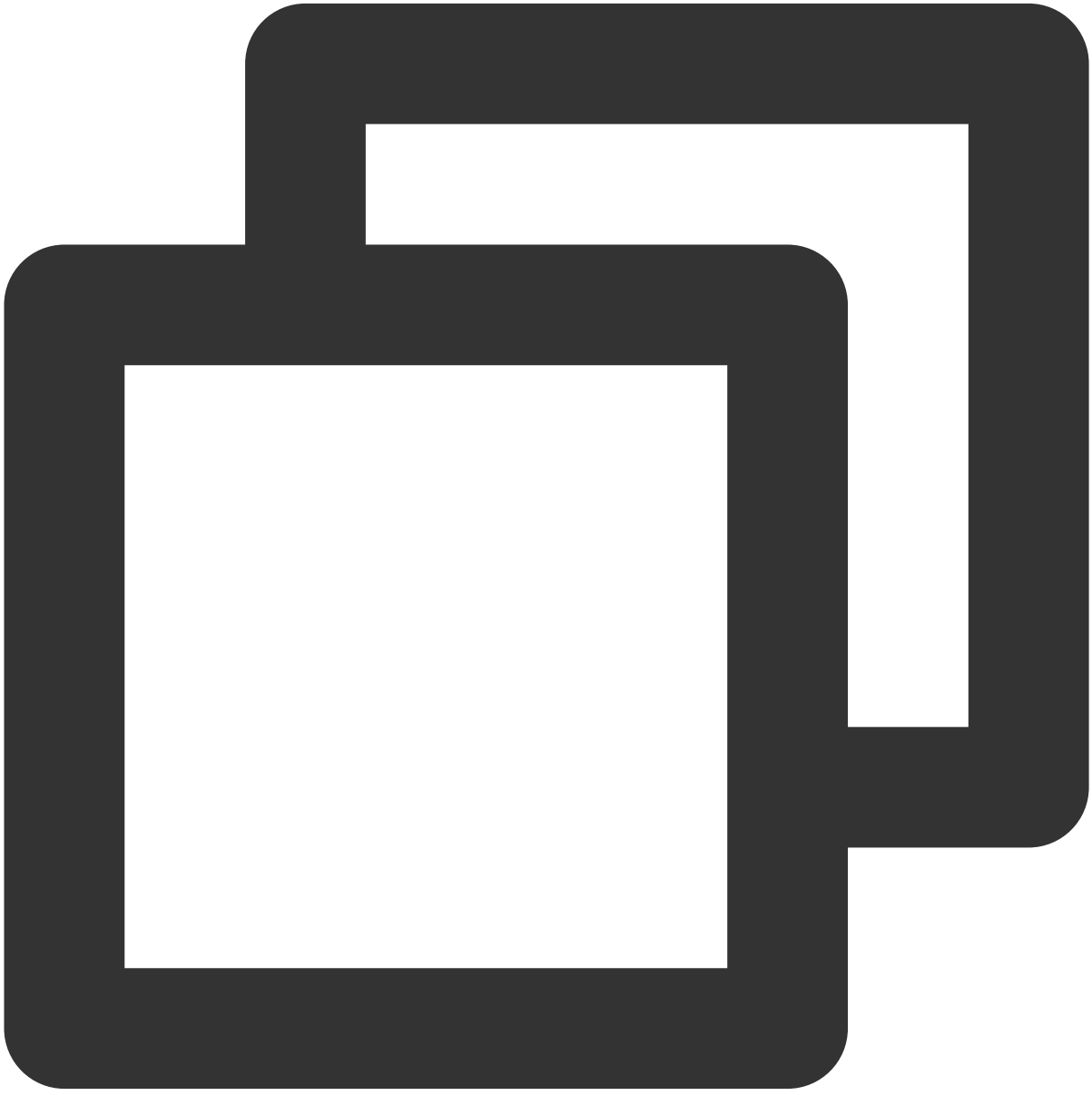
Node.js 函数形态一般有以下两种：

示例 1：



```
exports.main_handler = async (event, context) => {
  console.log(event);
  console.log(context);
  return event
};
```

示例 2：



```
exports.main_handler = (event,context,callback)=>{
  console.log(event);
  console.log(context);
```

```
    callback(null,"hello world");
}
```

执行方法

在创建云函数 SCF 时，均需要指定执行方法。在使用 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。

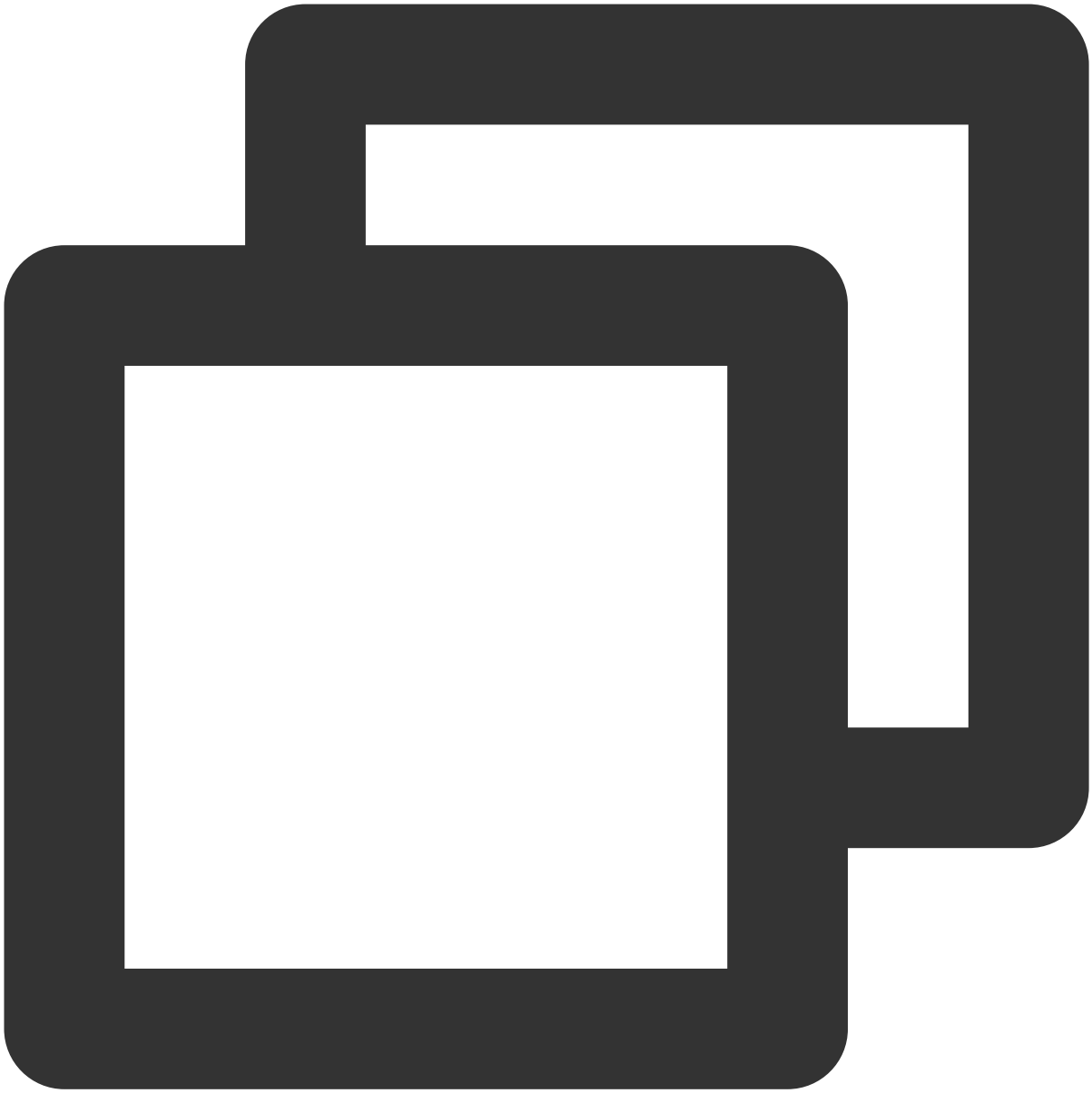
入参

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（可选）：`callback` 是一个函数，可以在**非异步处理程序中**使用它来返回响应。响应对象必须与 `JSON.stringify` 兼容。callback 函数有 `Error` 和响应两个参数，调用该函数时，SCF 等待函数执行完成后将响应或错误返回。

返回和异常

异步处理程序

异步处理程序必须使用 `async` 关键字，使用 `return` 来返回响应，使用 `throw` 返回错误信息。
在 SCF 中，如果您的 Node.js 函数中包含异步任务，须返回一个 promise 以确保该异步任务在当次调用执行。当您完成或拒绝该 promise 时，SCF 会返回响应或错误信息。
注意：
promise 方法不支持用 callback 方法返回，请使用 return。
异步处理程序示例：



```
exports.main_handler = async(event,context,callback) => {
  const promise = new Promise((resolve,reject) => {
    setTimeout(function() {
       resolve('成功')
       // reject('失败')
    }, 2000)
  })
  return promise
};
```

非异步处理程序

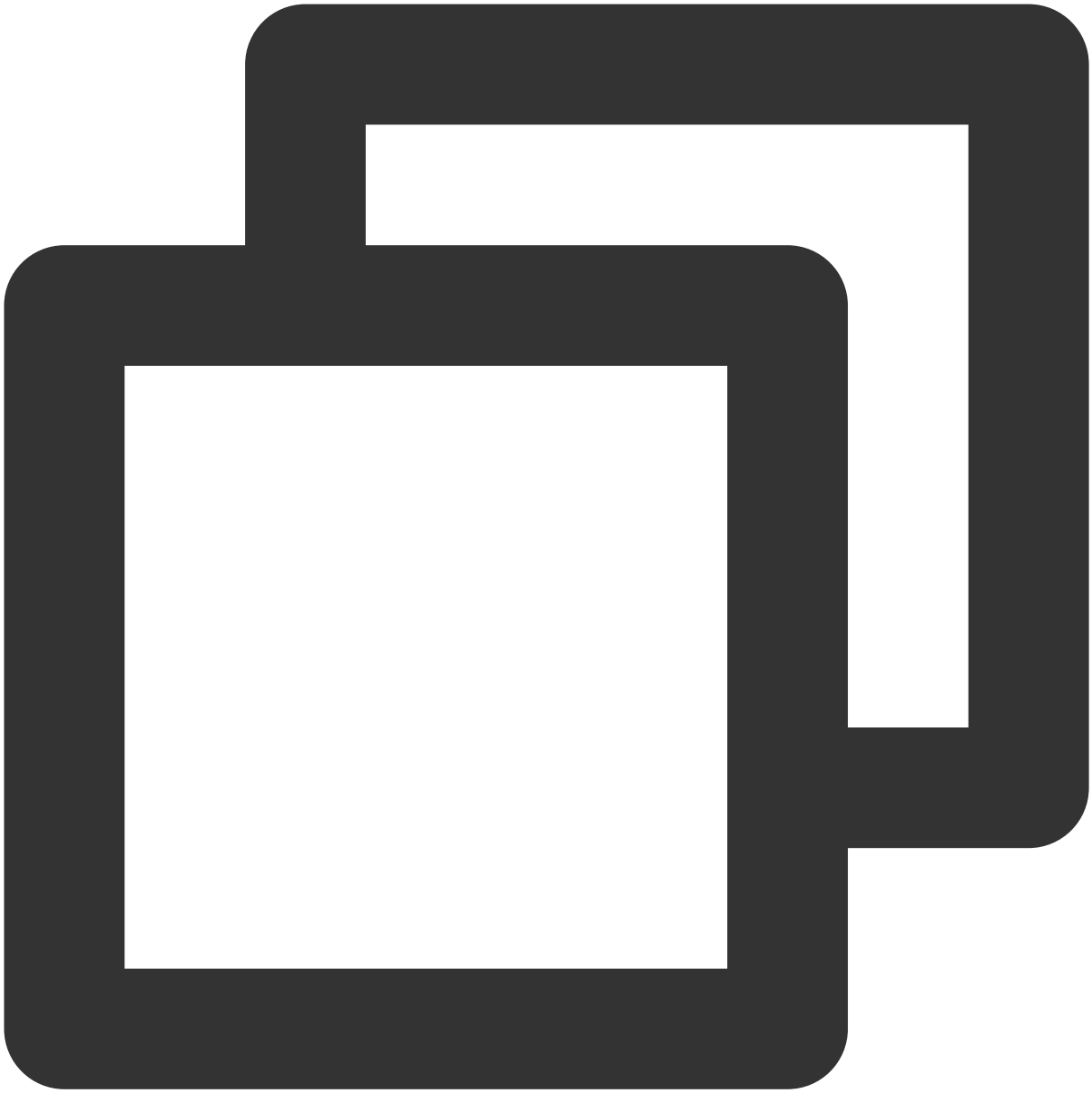
非异步处理程序，函数会一直执行，直到函数执行完成或函数超时后，SCF 将响应或错误信息返回。

注意：

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，从而导致函数无法返回直到超时。为了避免外部库的影响，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**设置** `context.callbackWaitsForEmptyEventLoop` **为 false**，可以修改默认的回调行为，避免等待事件循环为空。

您可以在 callback 回调执行前设置 `context.callbackWaitsForEmptyEventLoop = false;`，使云函数后台在 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，而在同步过程完成后立刻返回。

非异步处理程序示例：



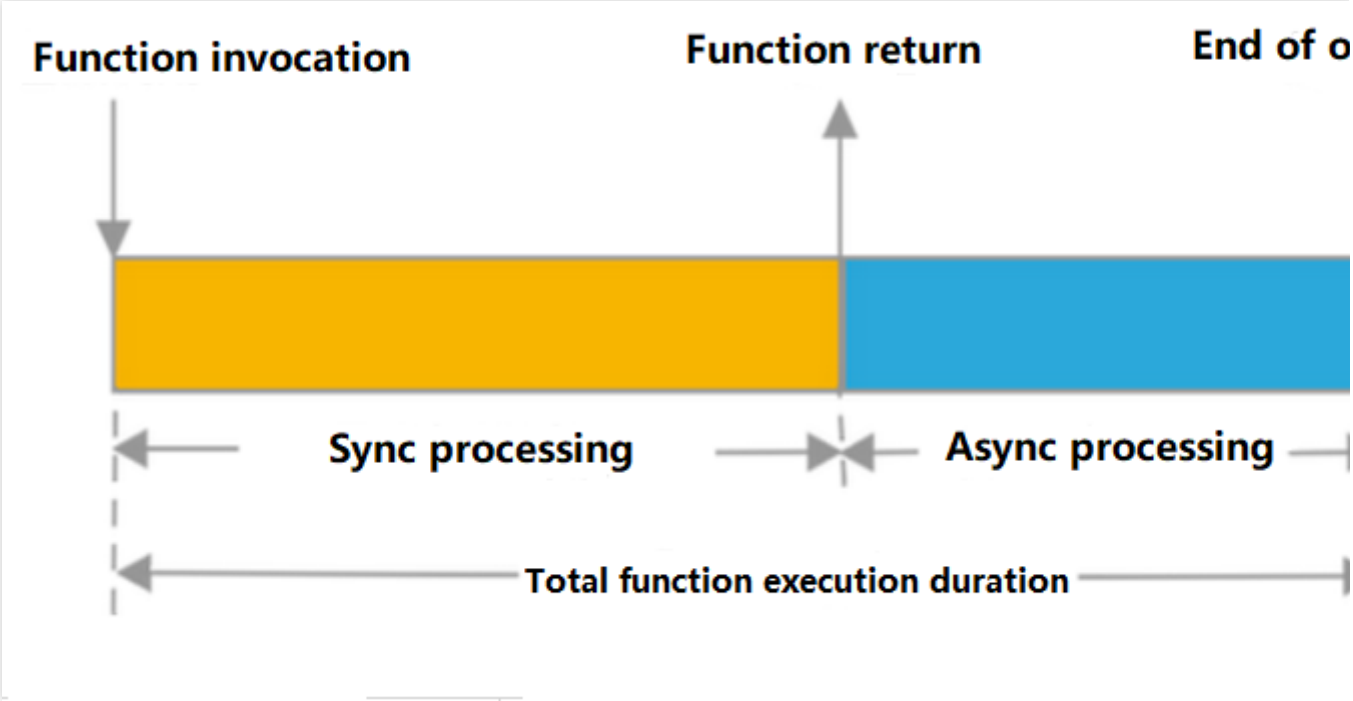
```
exports.main_handler = (event, context,callback) => {
    context.callbackWaitsForEmptyEventLoop = false
    callback(null,'success')
    setTimeout(() => {
        console.log('finish')
     },5000);
};
```

异步特性支持

在 **Node.js 10.15 及以上** 的 runtime 中，我们支持了将函数的同步执行返回和异步事件处理分开进行的能力。
入口函数的同步执行过程完成及返回后，云函数的调用将立刻返回，并将代码的返回信息返回给函数调用方。
同步流程处理并返回后，代码中的异步逻辑可以继续执行和处理，直到异步事件执行完成后，云函数的实际执行过程才完成和退出。

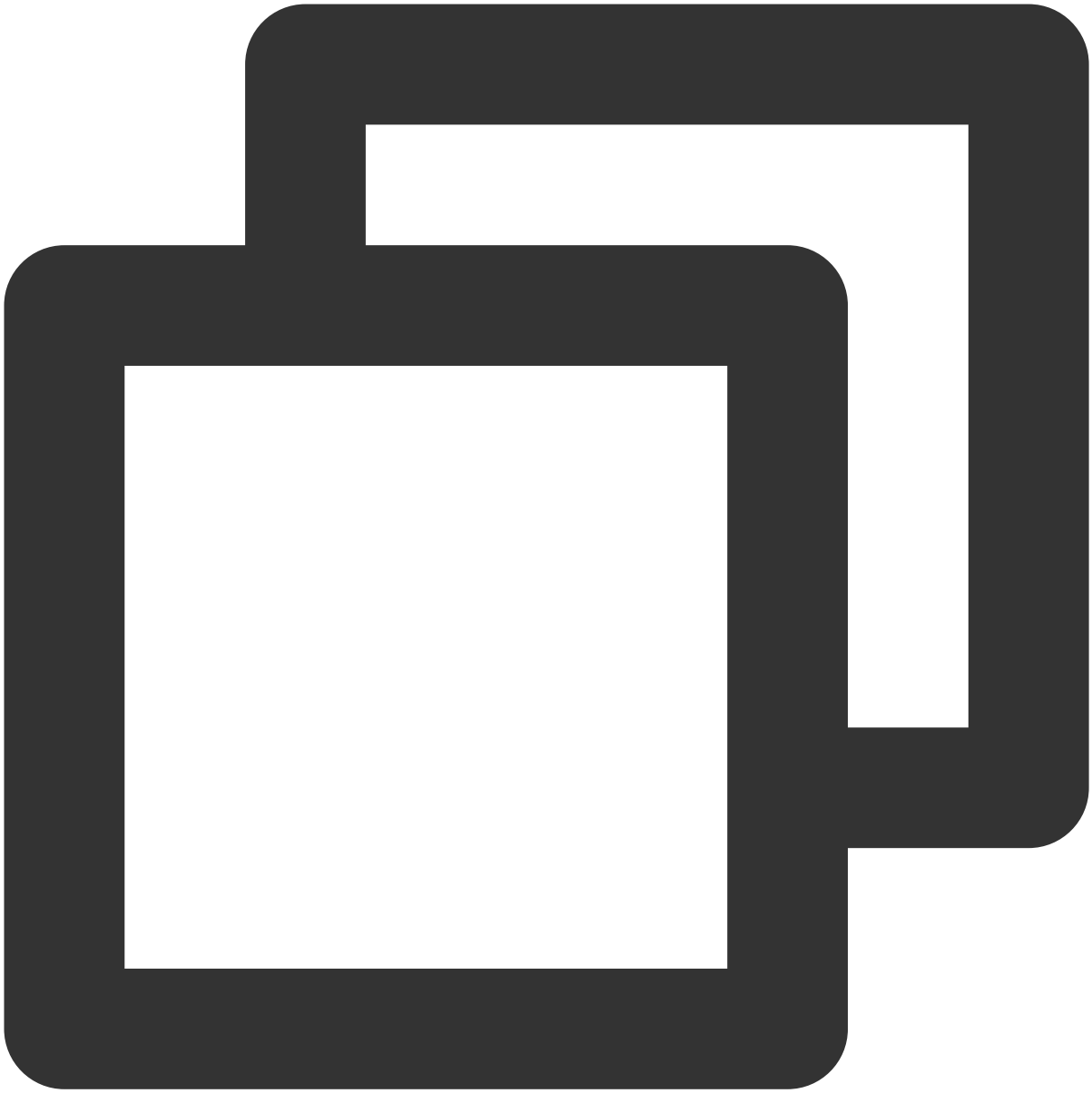
注意：

在这个过程中，由于云函数的日志是在整个执行过程完成后才进行收集和处理，因此在同步执行过程完成并返回时，云函数的返回信息中暂时无法提供日志、运行信息包括耗时、内存消耗等内容。具体信息可以在函数实际执行过程完成后，通过 Request Id 在日志中查询。
云函数的运行时长，将按照异步事件执行完成后进行计算。如果异步事件队列一直无法清空或执行完成，将会导致函数超时。这种情况下，调用方可能已经获得了函数的正确响应结果，但是云函数的运行状态将标注为由于超时而失败，同时运行时长按超时时间统计。
Node.js 的同步和异步运行特性、返回时间及运行时长示例如下图所示：



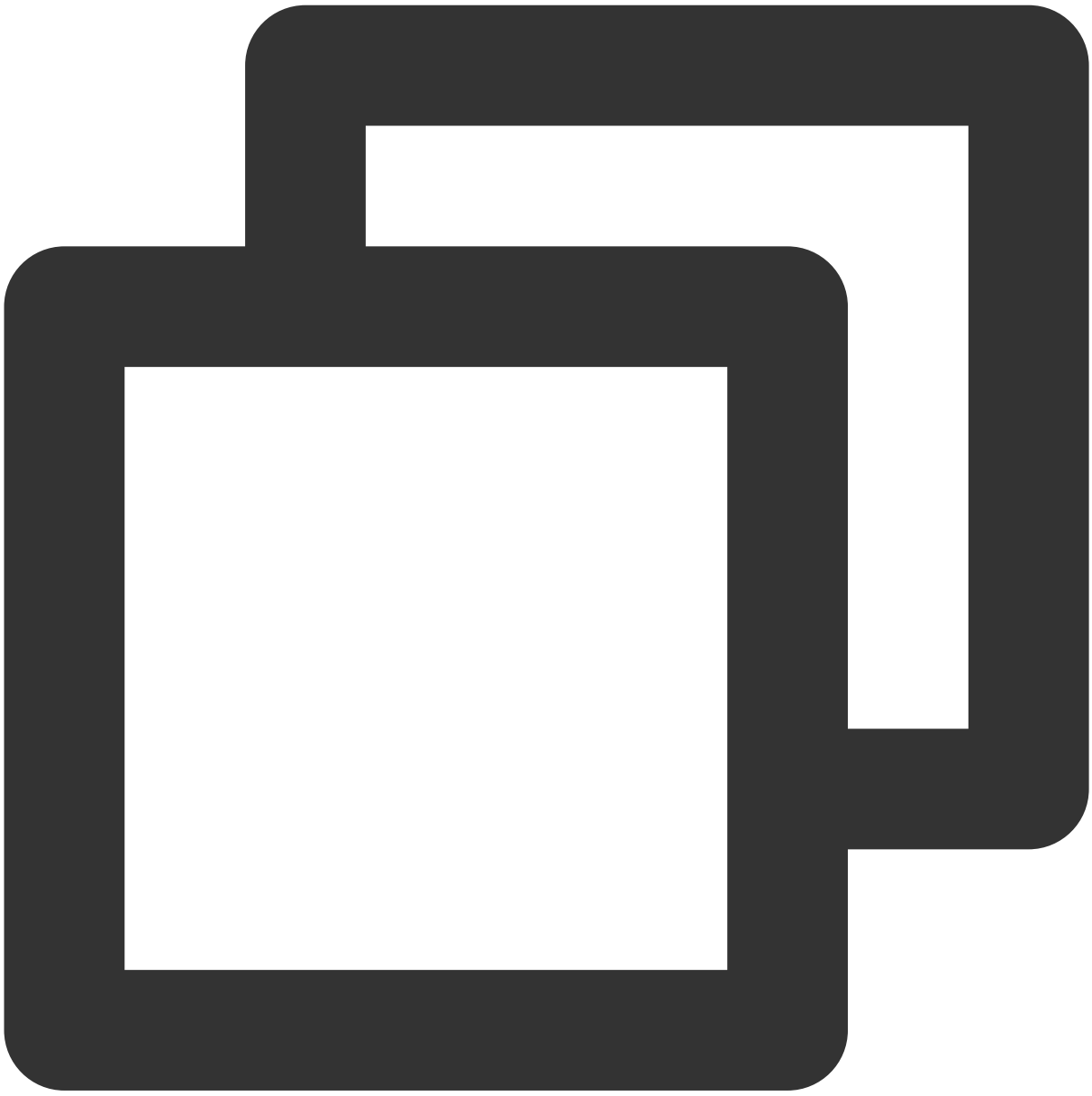
异步特性示例

使用如下示例代码创建函数，其中使用 setTimeout 方法设置了一个2秒后执行的函数：



```
'use strict';
exports.main_handler = (event, context, callback) => {
    console.log("Hello World")
    console.log(event)
    setTimeout(timeoutfunc, 2000, 'data');
    callback(null, event);
};

function timeoutfunc(arg) {
    console.log(`arg => ${arg}`);
}
```

在保存代码后，通过控制台测试或者通过云 API 的 Invoke 接口调用此函数，可以看到此函数在很短时间内即返回，响应时间小于1秒。
而查看函数的执行日志时，可以看到相关统计信息类似如下：



```
START RequestId: 1d71ddf8-5022-4461-84b7-e3a152403ffc
Event RequestId: 1d71ddf8-5022-4461-84b7-e3a152403ffc
2020-03-18T09:16:13.440Z        1d71ddf8-5022-4461-84b7-e3a152403ffc    Hello World
2020-03-18T09:16:13.440Z        1d71ddf8-5022-4461-84b7-e3a152403ffc    { key1: 'te
2020-03-18T09:16:15.443Z        1d71ddf8-5022-4461-84b7-e3a152403ffc    arg => data
END RequestId: 1d71ddf8-5022-4461-84b7-e3a152403ffc
```

```
Report RequestId: 1d71ddf8-5022-4461-84b7-e3a152403ffc Duration:2005ms Memory:128MB
```

在日志中统计了2005ms的执行时间，同时日志中可以看到在2秒后输出了 `arg => data` 的内容，即相关异步操作在当前调用中执行且是在同步过程执行完成后执行的，而函数调用在异步任务执行完成后才结束。

部署方法

最近更新时间：2024-04-22 18:03:28

部署方法

腾讯云云函数提供以下几种方式部署函数，您可以按需选择使用。创建、更新函数操作详情可参见 创建及更新函数。
通过 zip 打包上传部署，详情可参见 依赖安装和部署。
通过控制台编辑和部署，详情可参见 通过控制台部署函数。
使用命令行部署，详情可参见 通过 Serverless Cloud Framework 部署函数。

依赖安装和部署

当前的函数标准 Node.js Runtime 中仅提供了 `/tmp` 目录可写，其他目录只读，因此在用到依赖库时，需要使用本地安装、打包、上传的方式。Node.js 依赖包通常可以与函数代码一同上传，或上传至层中，然后绑定使用。

在线安装依赖

Node.js 提供在线依赖安装功能，可参考 在线依赖安装 文档使用。

本地安装依赖

依赖管理工具

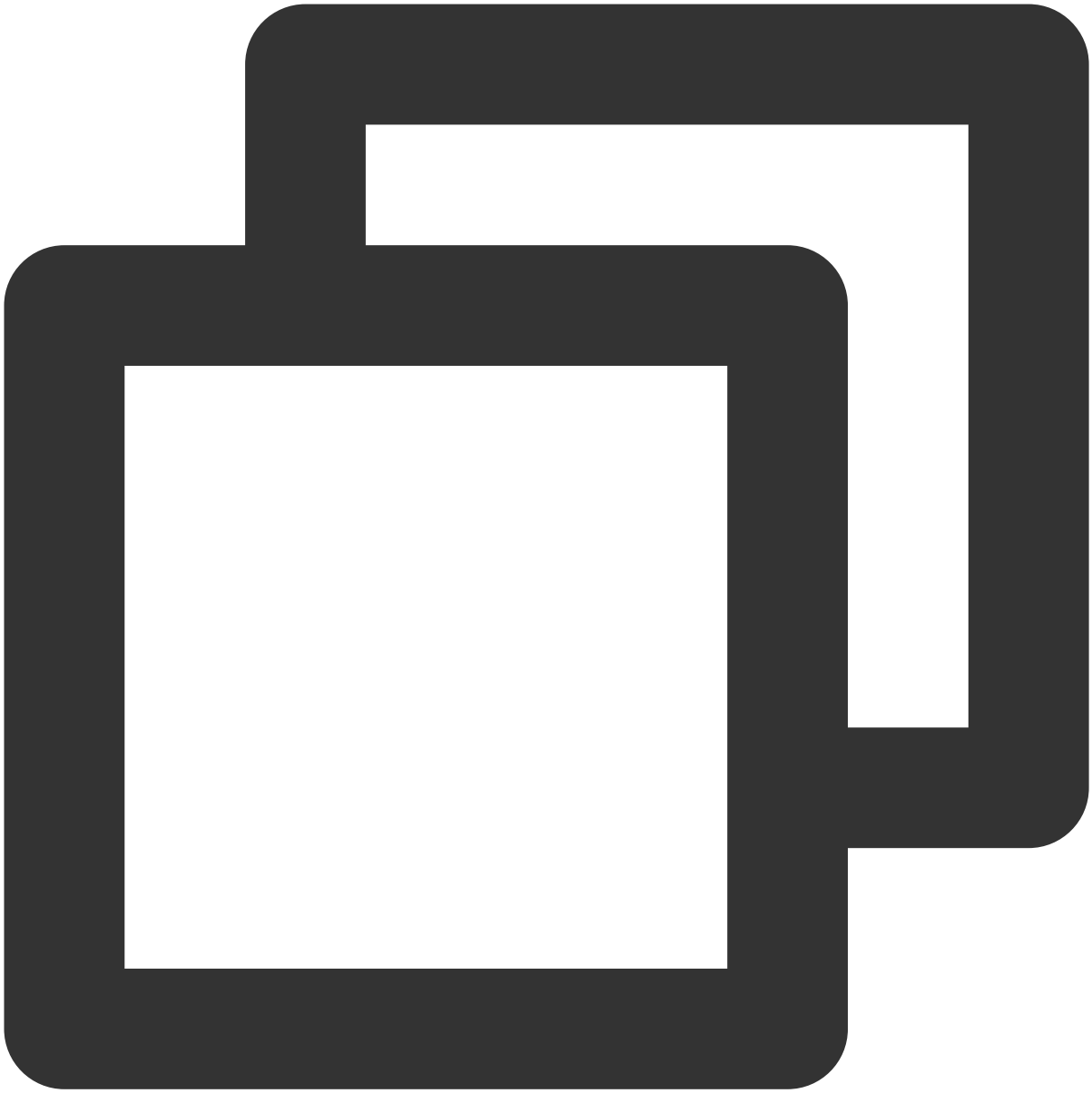
Node.js 可以通过 npm 包管理器进行依赖管理。

使用方法

在代码目录下执行 `npm install xxx` 命令安装依赖包。
注意：
函数运行的系统是 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找不到依赖的错误。您可参考 云函数容器镜像 进行依赖安装。
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。详情请参阅 使用 Docker 安装依赖。

示例

1. 在本地安装 `requests` 依赖，其中代码文件 `index.js` 如下所示：



```
'use strict';
var request = require('request');
exports.main_handler = async (event, context) => {
 request('https://cloud.tencent.com/', function (error, response, body) {
     if (!error && response.statusCode == 200) {
         console.log(body) // 请求成功的处理逻辑
     }
 })
 return "success"
};
```

2. 使用 `npm install request` 命令在项目当前目录安装 requests 依赖。

打包上传

依赖可以和项目一同上传，并在函数代码中通过 `require` 方式引入和使用。同时，依赖也可以打包部署为层，并通过在函数创建部署时，与函数绑定，提供复用能力。

您可以通过控制台选择本地文件夹的方式自动化打包，也可以通过手工打包的方式形成可以用于部署函数或层的 zip 包。在打包部署时，需要注意的是均在项目目录下进行打包操作，即确保代码、依赖均在 zip 文件内的根目录中，详情可参见 打包要求。

日志说明

最近更新时间：2024-04-22 18:03:28

日志开发

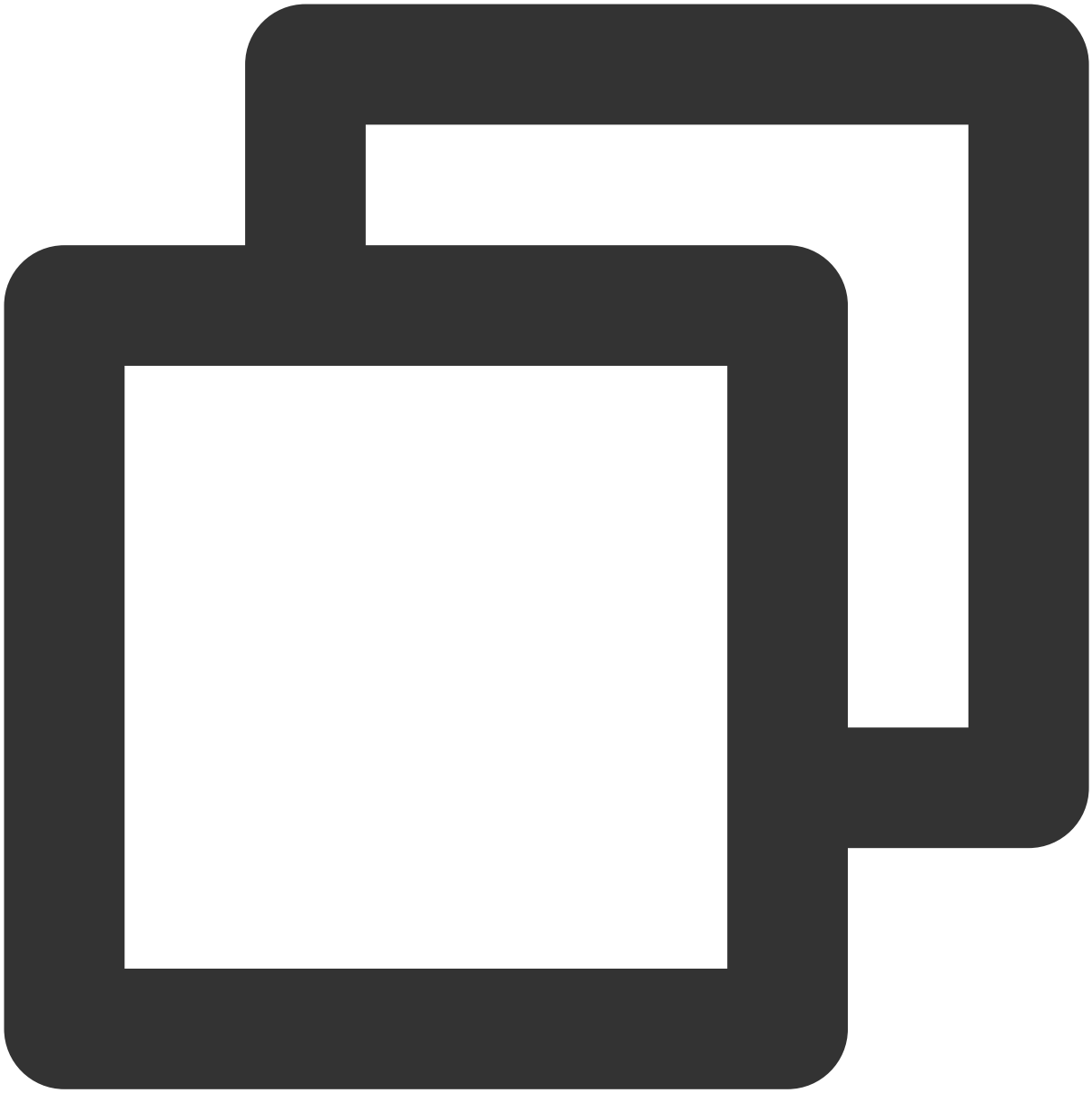
您可以在程序中使用如下语句来完成日志输出：

console.log()

console._stdout.write()（Node.js 8.9 及以上版本支持）

process.stdout.write()（Node.js 8.9 及以上版本支持）

例如，执行以下代码，可以在函数日志中查询输出内容。



```
'use strict';
exports.main_handler = async (event, context) => {
    console.log("Hello World")
    console._stdout.write("Hello World")
    process.stdout.write("Hello World")
    return event
};
```

使用须知

Node.js 12.16、Node.js 10.15 运行环境下，使用 `console.log` 打印日志，平台会按照“时间戳 RerquestId 日志内容”的格式对日志内容封装写入日志服务 CLS。
示例：
日志打印语句：`console.log("hello world")`
日志输出效果：`2021-12-27T03:53:59.192Z a7358cce-489a-4674-8e4e-68665fa2b81d Hello World`
Node.js 8.9 运行环境下，使用 `console.log`、`console._stdout.write()`、`process.stdout.write()` 打印日志，平台会按照“时间戳 RerquestId 日志内容”的格式对日志内容封装写入日志服务 CLS。
示例：
日志打印语句：`console.log("hello world")`
日志输出效果：`2021-12-27T03:53:59.192Z a7358cce-489a-4674-8e4e-68665fa2b81d Hello World`
Node.js 6.10 运行环境下，使用 `console.log` 打印日志，平台会按照“时间戳 RerquestId 日志内容”的格式对日志内容封装写入日志服务 CLS。
示例：
日志打印语句：`console.log("hello world")`
日志输出效果：`2021-12-27T03:53:59.192Z a7358cce-489a-4674-8e4e-68665fa2b81d Hello World`

日志查询

当前函数日志均会投递至腾讯云日志服务 CLS 中，您可对函数日志进行投递配置，详情可参见 日志投递配置。
您可通过云函数的日志查询界面或通过日志服务的查询界面，查询函数执行日志。日志查询方法详情可参见 日志检索教程。
说明：
函数日志投递到日志服务日志集 LogSet 和日志主题 LogTopic，均可以通过函数配置查询。

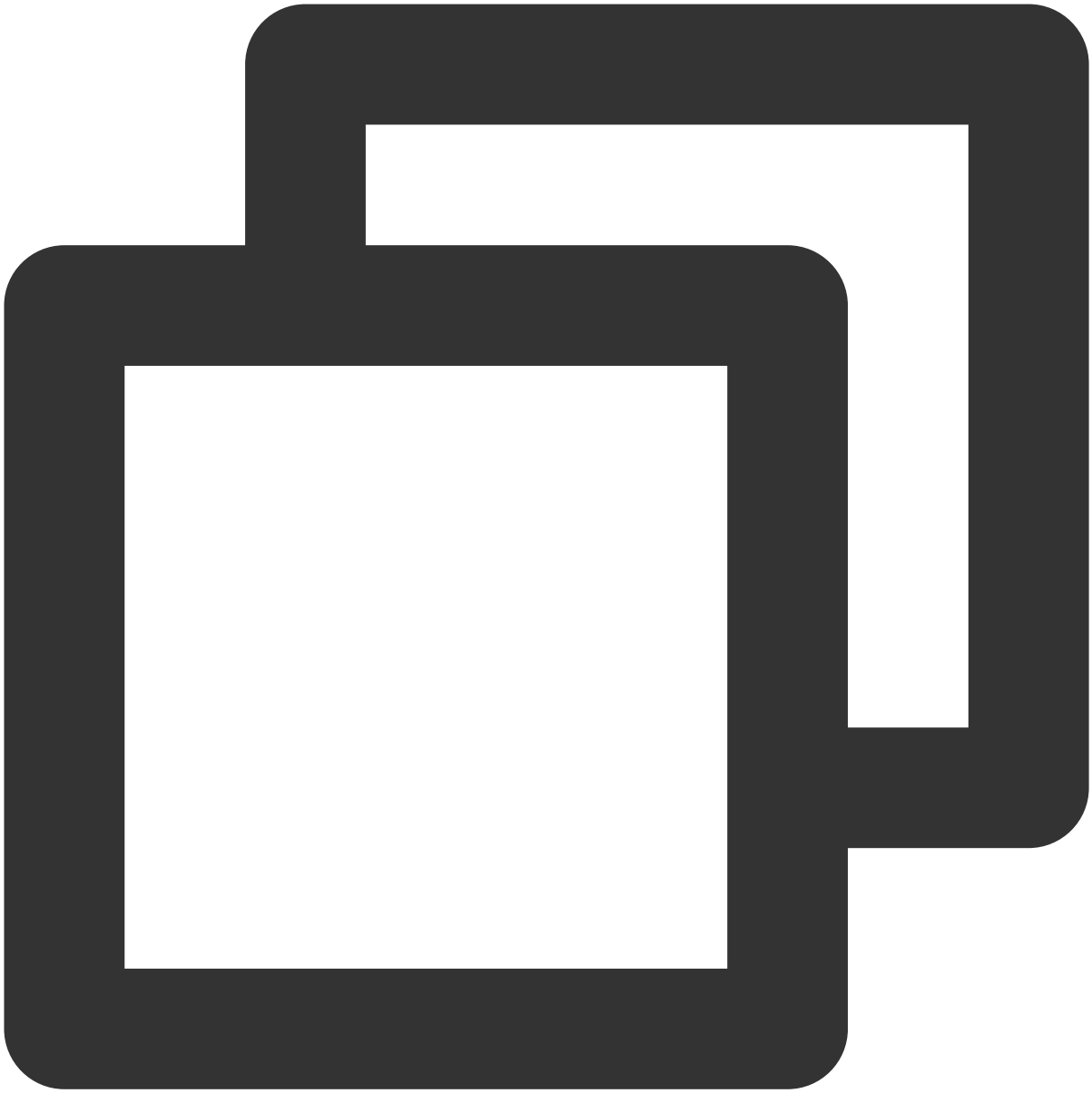
自定义日志字段

当前在函数代码中使用简单日志打印语句，将会在投递到日志服务时，记录在 `SCF_Message` 字段中。日志服务的字段说明可参见 索引说明。
目前云函数已经支持在输出到日志服务的内容中增加自定义字段，通过增加自定义字段，您可以将业务字段及相关数据内容输出到日志中，并通过使用日志服务的检索能力，对执行过程中的业务数据及相关内容进行查询跟踪。
注意：

如需对自定义字段进行键值查询，如 `SCF_CustomKey：SCF` ，请参考 日志服务索引配置 为函数日志投递的日志主题添加键值索引。
为避免误操作索引配置导致函数日志查询失败，函数配置的默认投递日志主题（以 `SCF_LogTopic_` 为前缀命名）不支持修改索引配置。请将函数日志投递主题设置为 自定义投递 后再更新日志主题索引配置。
日志主题修改索引配置后，仅对新写入的数据有效。

输出方法

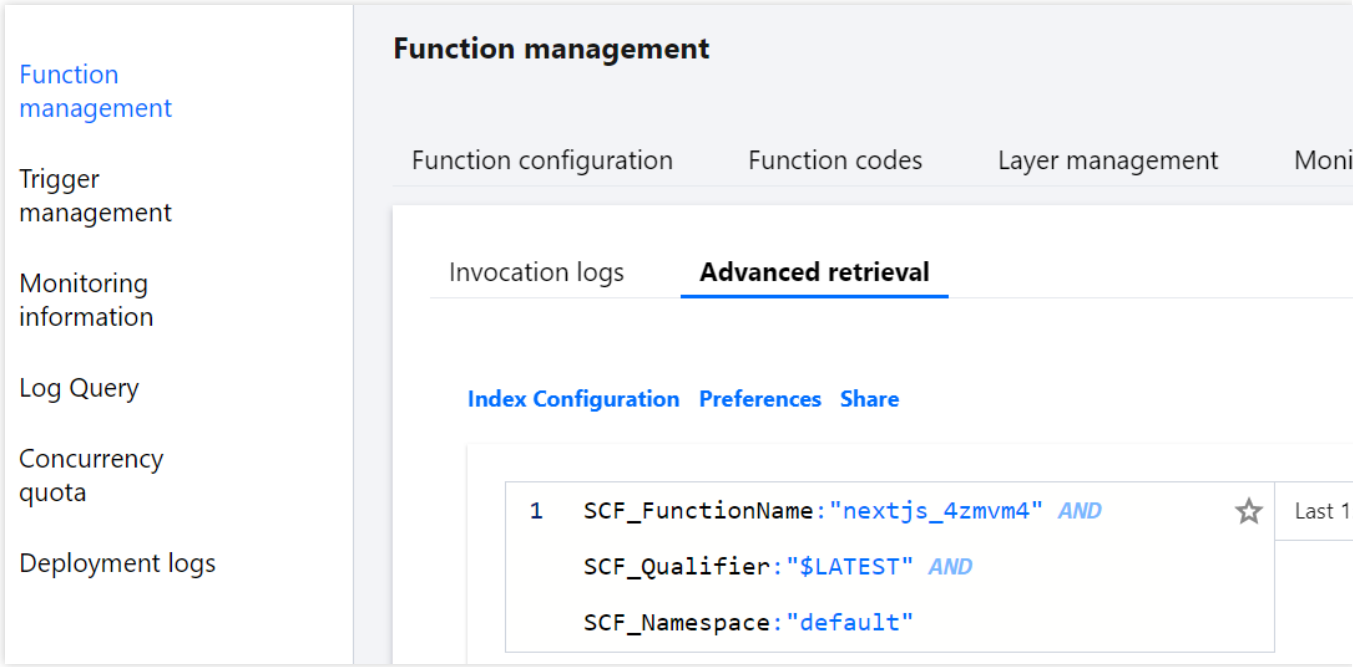
当函数输出的单行日志为 JSON 格式时，JSON 内容将被解析并在投递至日志服务时按 `字段:值` 的方式进行投递。JSON 内容的解析仅能解析第一层，更多的嵌套结构将作为值进行记录。
您可执行以下代码进行测试：



```
'use strict';
exports.main_handler = async (event, context) => {
    console._stdout.write(JSON.stringify({"key1": "test value 1","key2": "test valu
    return "hello world"
};
```

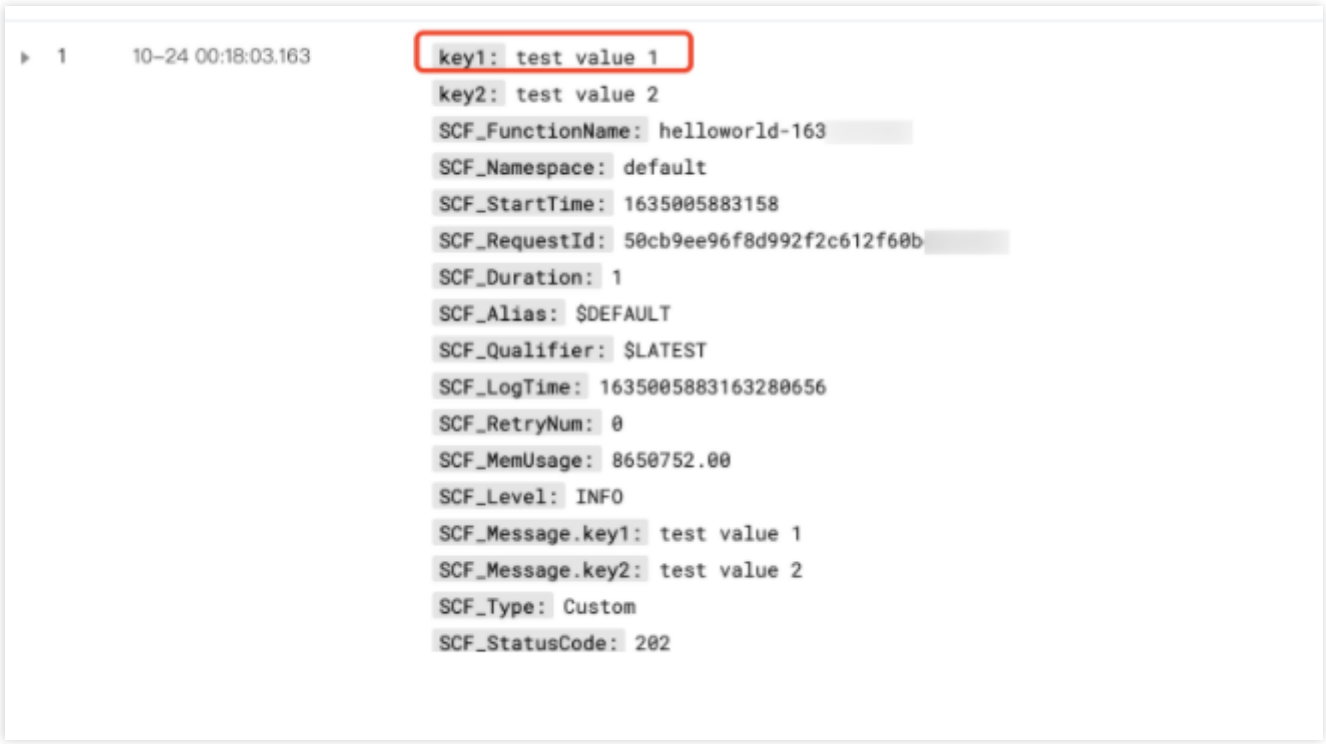
检索方法

在使用上述代码进行测试运行后，您可在函数-日志查询-高级检索中通过如下语句进行检索：



检索结果

在测试写入日志服务后，您可以在日志查询中检索到 `key1` 字段。如下图所示：



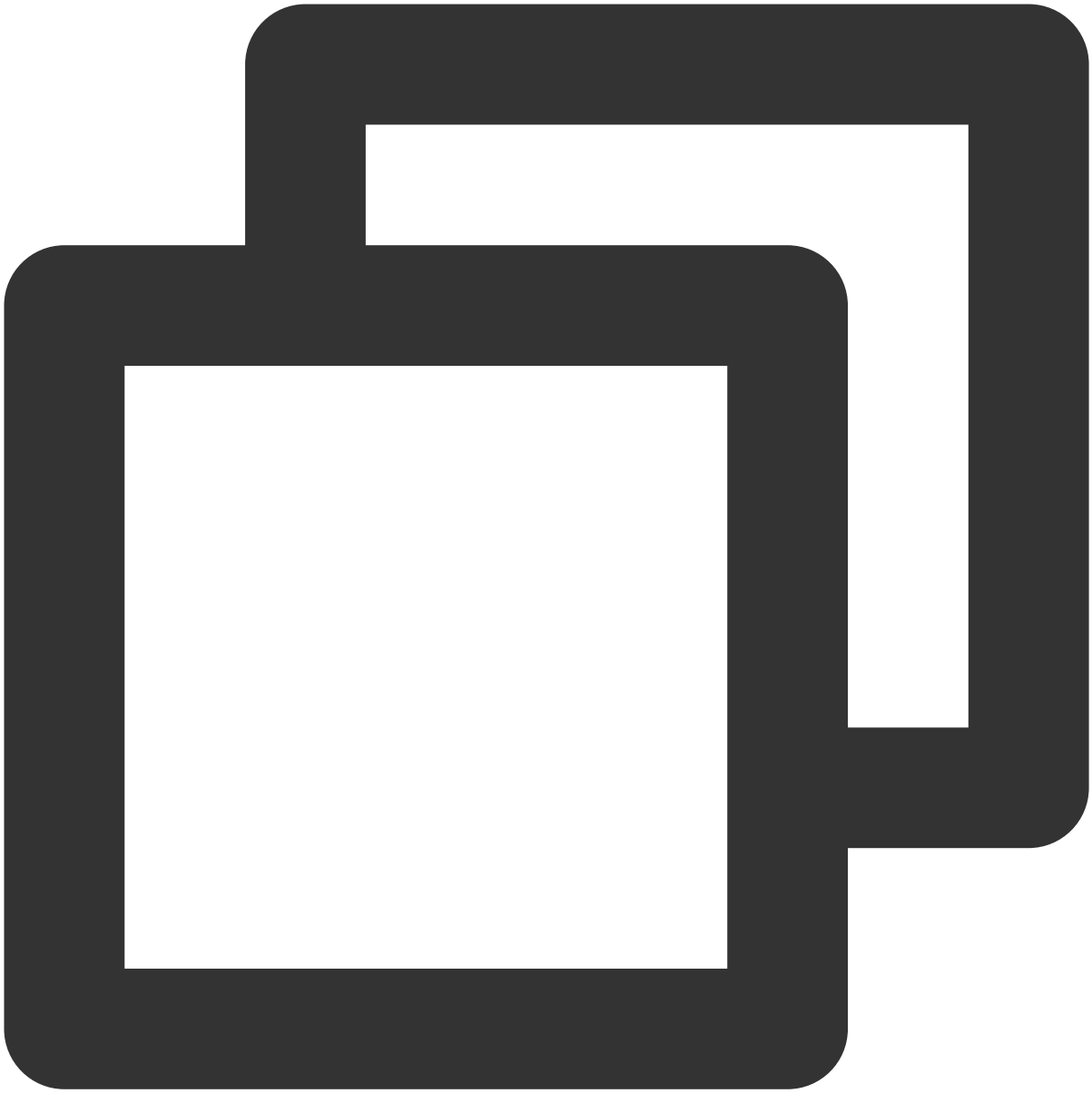
常见示例

最近更新时间：2024-04-22 18:03:28

常见示例中包含了 Node.js 环境下可以试用的相关代码片段，您可以根据需要选择尝试。示例均基于 Node.js 12.16 环境提供。
您可从 github 项目 scf-nodejs-code-snippet 中获取相关代码片段并直接部署。

环境变量读取

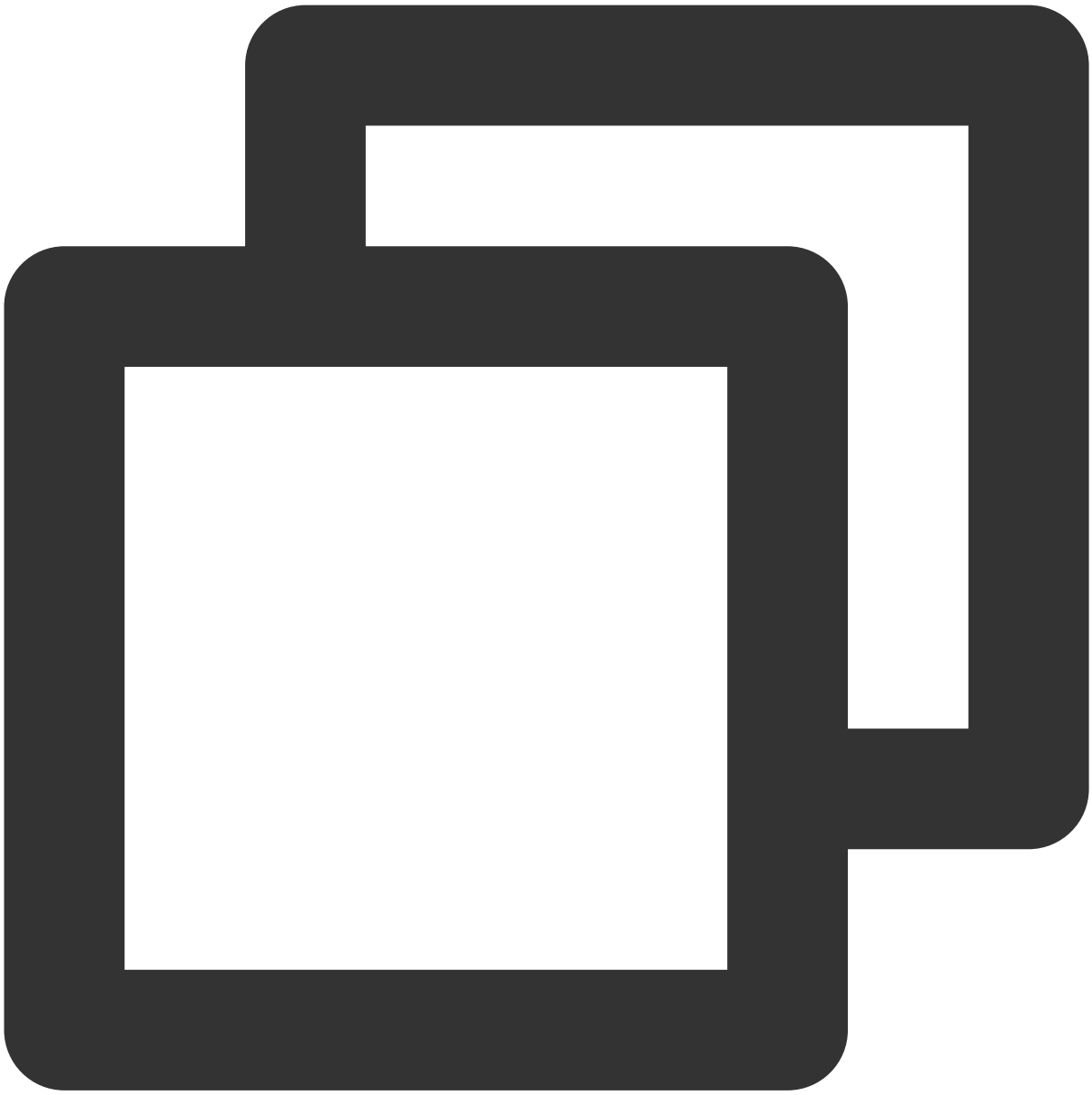
本示例提供了获取全部环境变量列表，或单一环境变量值的方法。



```
'use strict';
exports.main_handler = async (event, context) => {
        console.log(process.env)
    console.log(process.env.SCF_RUNTIME)
    return "Hello world"
};
```

本地时间格式化输出

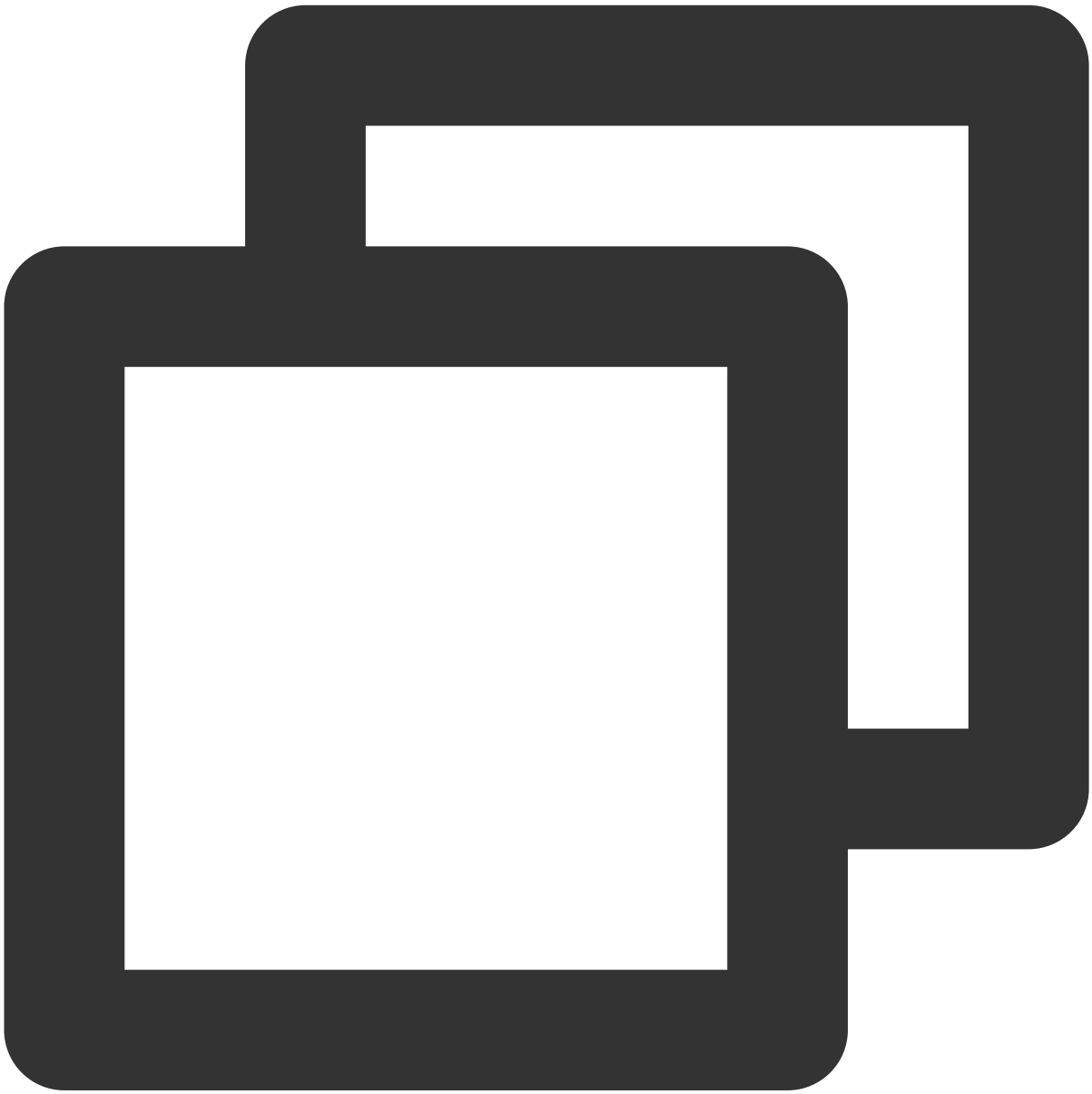
本示例使用了 moment 库获取本地时间，并提供了时间格式化输出方法，按指定格式进行日期和时间输出。
SCF 环境默认是 UTC 时间，如果期望按北京时间输出，可以为函数添加 `TZ=Asia/Shanghai` 环境变量。



```
'use strict';
const moment = require('moment')
exports.main_handler = async (event, context) => {
    let currentTime = moment(Date.now()).format('YYYY-MM-DD HH:mm:ss')
    console.log(currentTime)
    return "hello world"
};
```

函数内发起网络连接

本示例使用了 requests 库在函数内发起网络连接，获取页面信息。可以通过在项目目录下执行 `npm install request` 命令完成依赖库安装。



```
'use strict';
var request = require('request');
exports.main_handler = async (event, context) => {
    request('https://cloud.tencent.com/', function (error, response, body) {
        if (!error && response.statusCode == 200) {
```

```
            console.log(body) // 请求成功的处理逻辑
        }
    })
    return "success"
};
```

在线依赖安装

最近更新时间：2024-04-22 18:03:28

操作场景

腾讯云云函数提供函数部署时在线安装依赖功能。

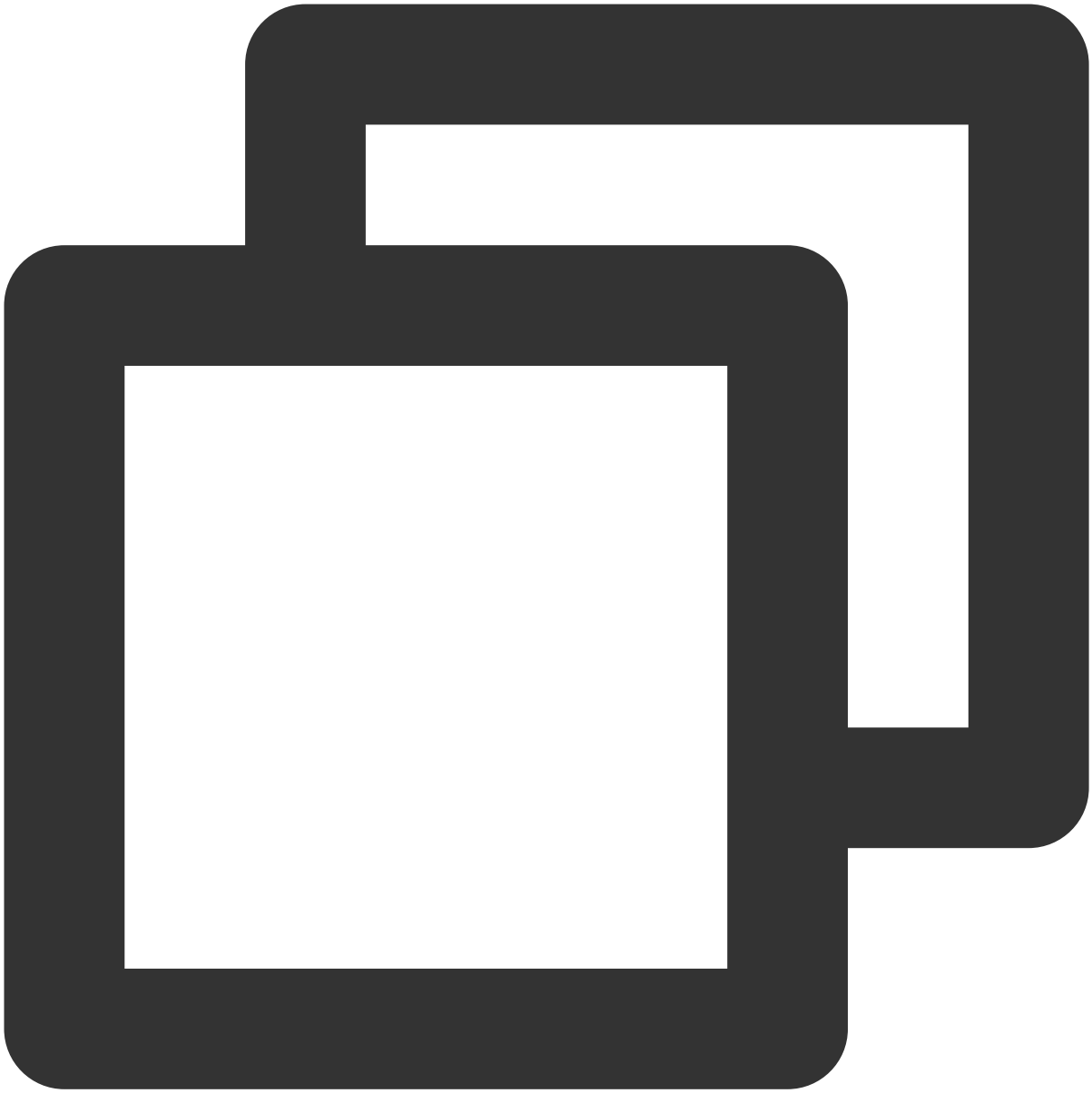
功能特性

说明：

目前仅针对 Node.js 提供在线安装依赖功能。

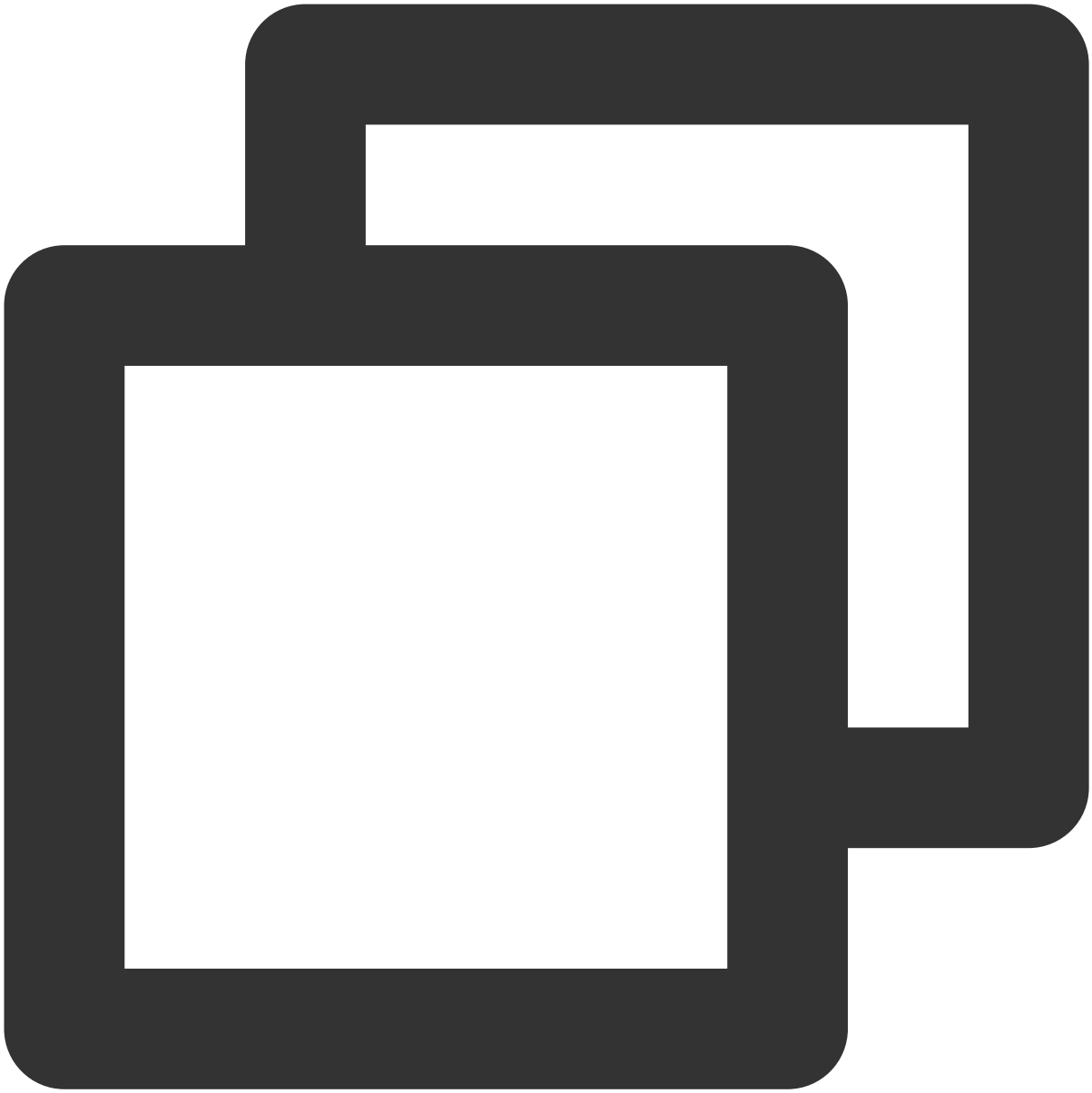
如果在函数配置中启用了“在线安装依赖”，在每次上传代码后，云函数后台将检查代码包根目录的 `package.json` 文件，并根据 `package.json` 中的依赖，尝试使用 npm install 安装依赖包。

例如，项目中的 `package.json` 文件中列出了如下依赖包：



```
{
       "dependencies": {
       "lodash": "4.17.15"
    }
}
```

此依赖包会在部署时导入函数中：

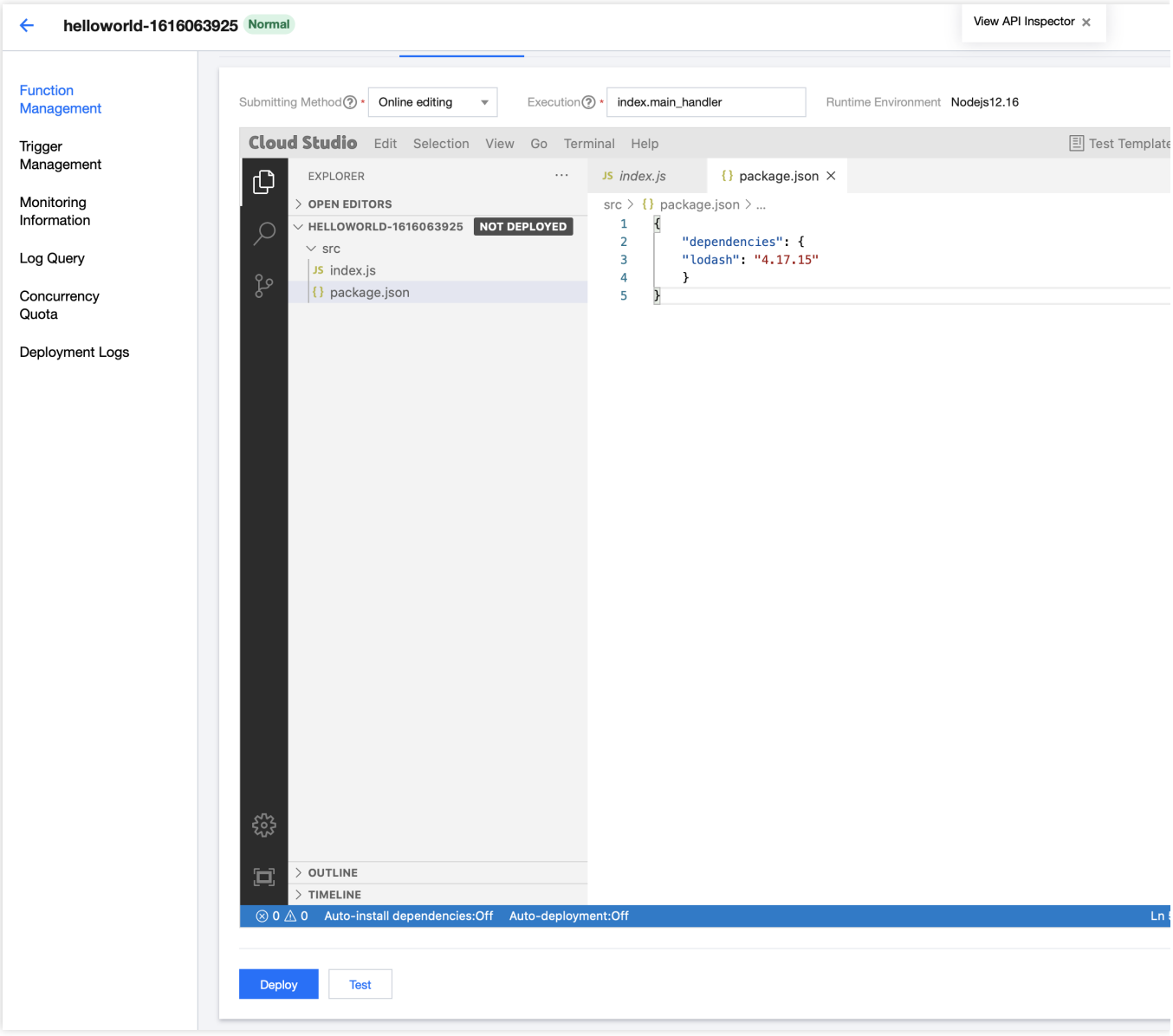


```
const _ = require('lodash');
exports.handle = (event, context, callback) => {
    _.chunk(['a', 'b', 'c', 'd'], 2);
    // => [['a', 'b'], ['c', 'd']]
};
```

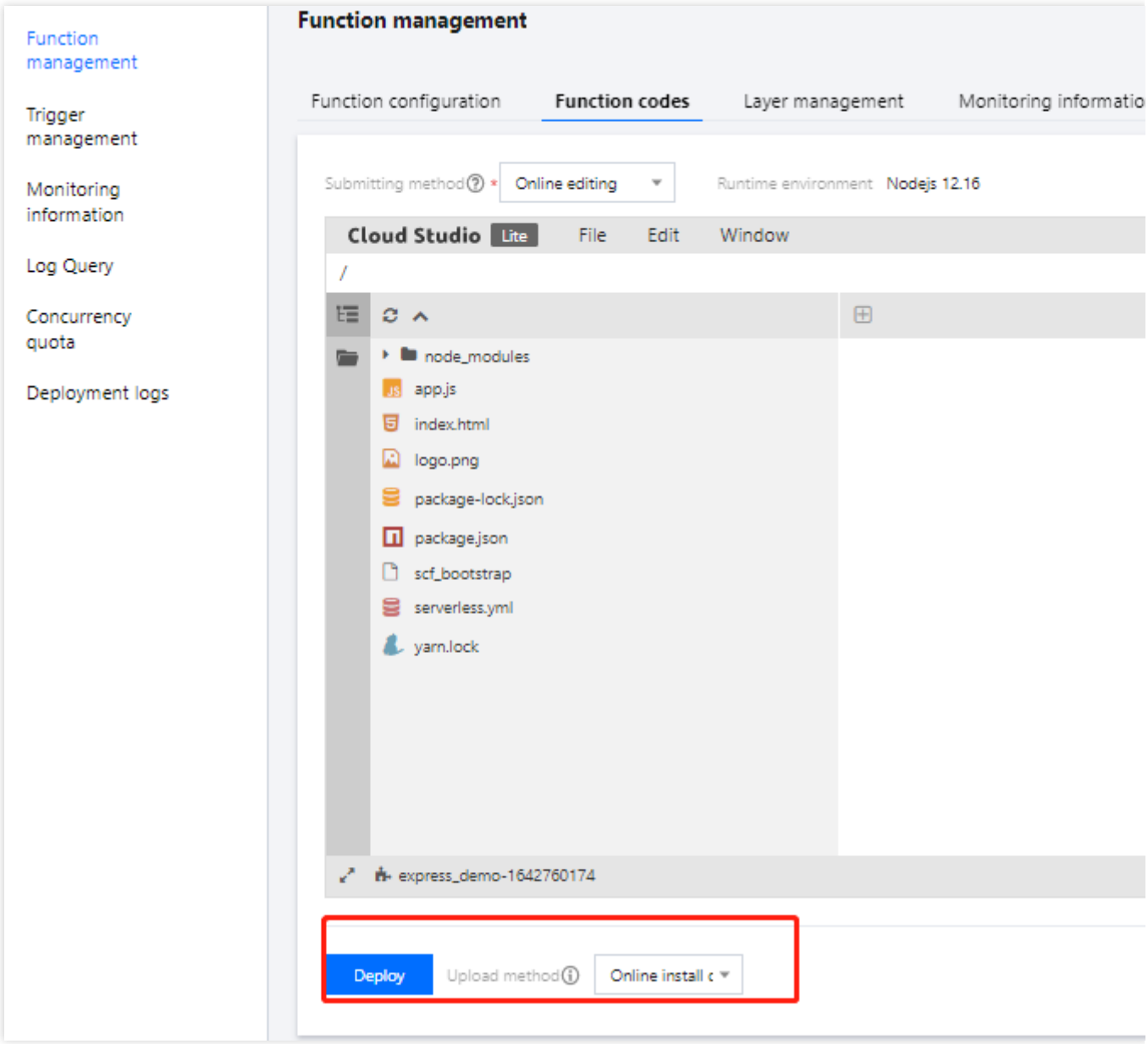
操作步骤

1. 登录 Serverless 控制台，选择**广州**地域。
2. 选择左侧导航栏**函数服务**，在“函数服务”列表页面选择函数名。
3. 选择**函数代码**页签，根据您的实际需求修改函数代码。
4. 在 IDE 代码编辑窗口右上角中单击

，在下拉列表中选择**自动安装依赖:关闭**以开启自动安装依赖，如下图所示：



开启自动安装依赖后，刷新界面，在代码编辑区域右下角单击**切换到旧版编辑器**，选择上传方式为**在线安装依赖**。



5. 单击**部署**，云函数后台会根据 `package.json` 自动安装依赖。

Golang
环境说明

最近更新时间：2024-04-22 18:03:28

Golang 版本选择

目前支持的 Golang 开发语言包括如下版本：
Golang 1.8 及以上版本
您可以在函数创建时，选择 `Go1` 运行环境。

注意事项

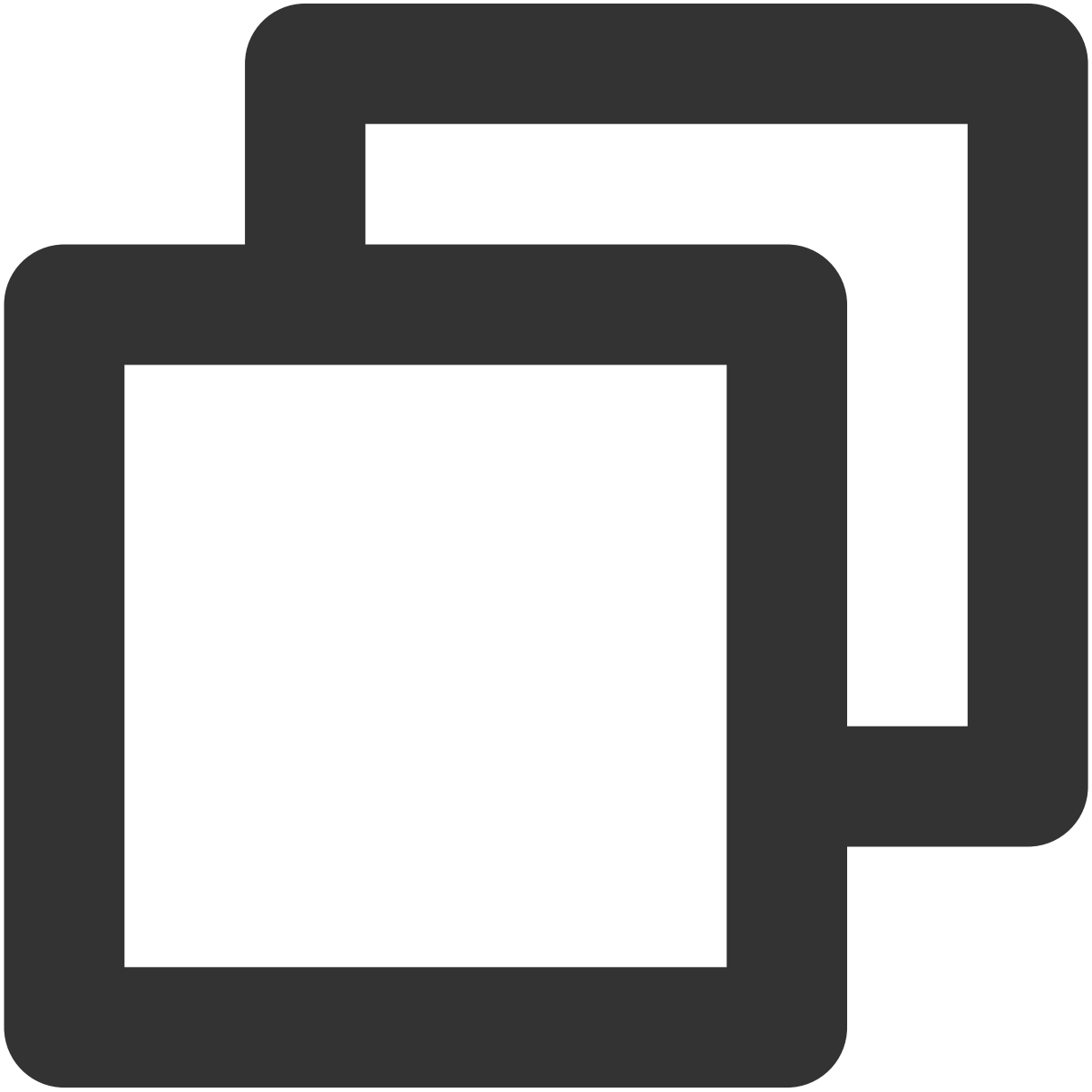
在 SCF 中，Go 的使用和 Python、Node.js 这类脚本型语言不同，有如下限制：
不支持上传代码：使用 Go 语言，仅支持上传已经开发完成，编译打包后的二进制文件。SCF 环境不提供 Go 的编译能力。
不支持在线编辑：不能上传代码，所以不支持在线编辑代码。Go 运行时的函数，在代码页面仅能看到通过页面上传或 COS 提交代码文件的方法。

开发方法

最近更新时间：2024-04-22 18:03:28

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)
}
```

代码开发时，请注意以下几点：

需要使用 package main 包含 main 函数。

引用 `github.com/tencentyun/scf-go-lib/cloudfunction` 库，在编译打包之前，执行 `go get github.com/tencentyun/scf-go-lib/cloudfunction` 。

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），具体说明请参见 入参。

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 错误信息在后，返回值组合有（），（ret），（error），（ret，error），具体说明请参见 返回值。

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。

在 main 函数中使用包内的 Start 函数启动入口函数。

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。

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 包的根目录下包含指定的二进制文件，避免因为无法查找到执行文件导致函数创建或执行失败。

package 与 main 函数

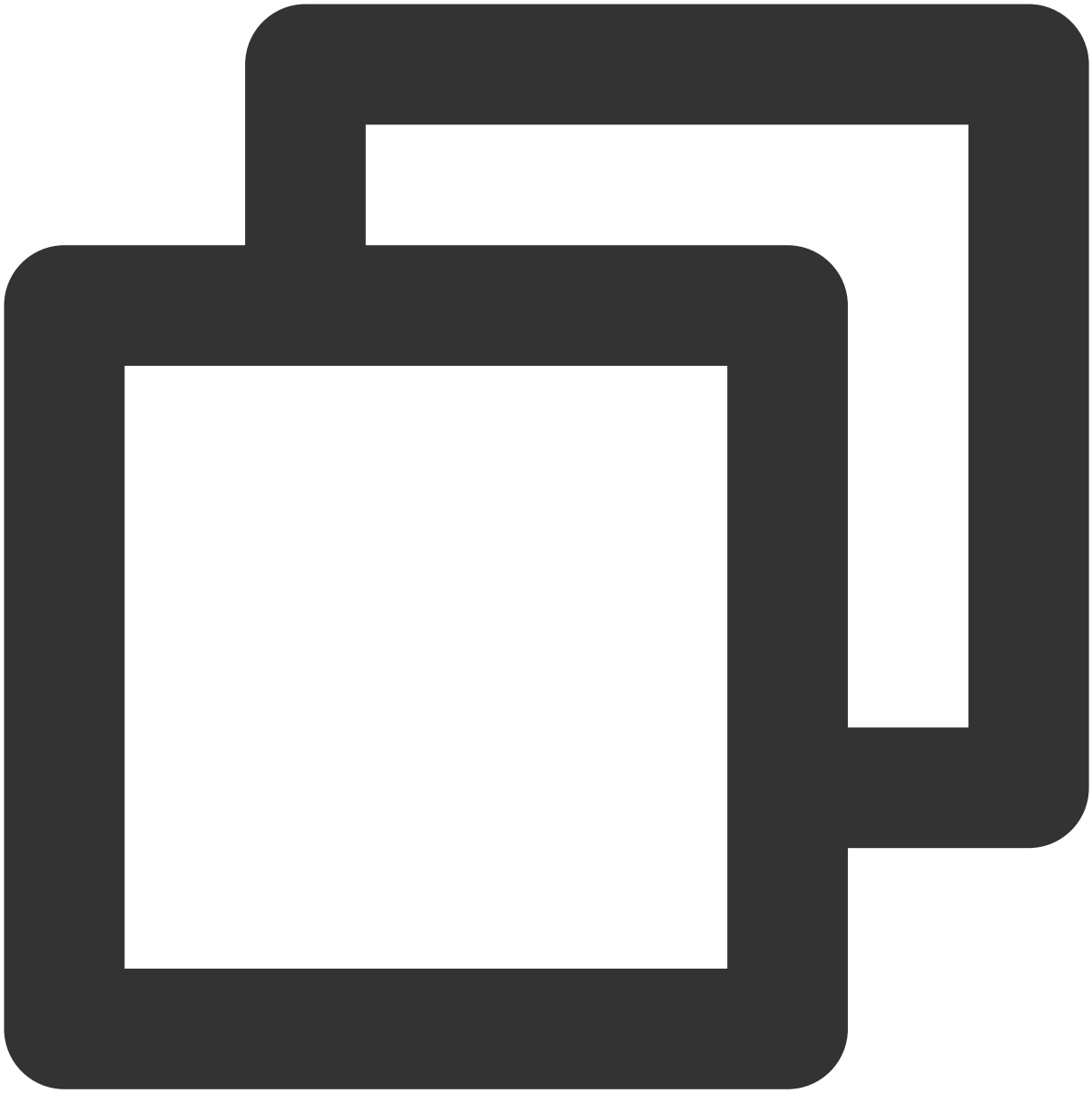
在使用 Golang 开发云函数时，需要确保 main 函数位于 main package 中。在 main 函数中，通过使用 cloudfunction 包中的 Start 函数，启动实际处理业务的入口函数。
通过 `import "github.com/tencentyun/scf-go-lib/cloudfunction"` ，可以在 main 函数中使用包内的 Start 函数。

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，如示例中的 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。

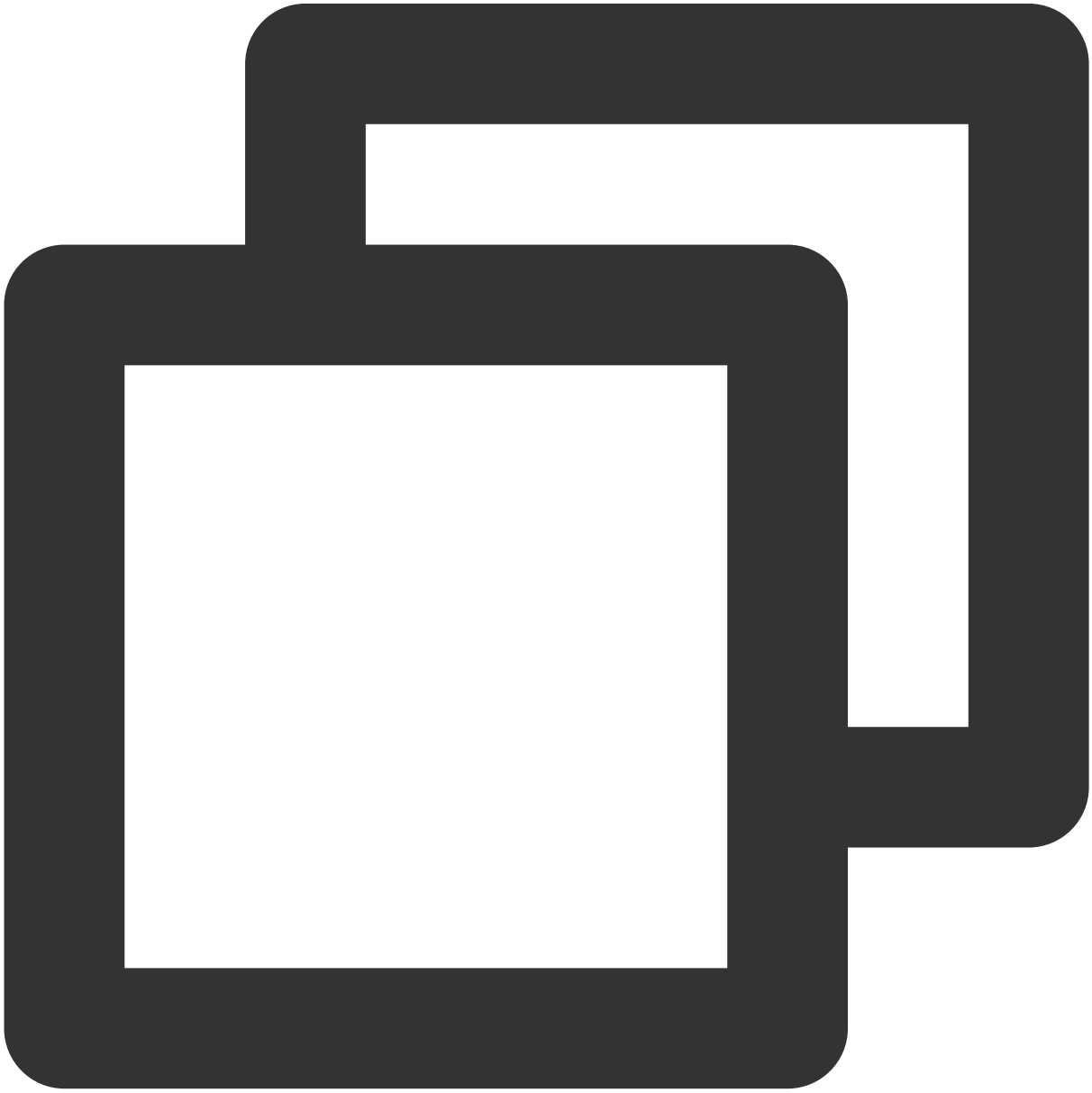
自定义数据结构对应的 JSON 结构，通常与函数执行时的入参对应。在函数调用时，入参的 JSON 数据结构将会转换为自定义数据结构变量并传递和入口函数。

注意：

部分触发器传递的入参事件结构目前已有一部分已定义，可直接使用。您可通过 cloud event 定义 获取 golang 的库并使用。通过在代码中引用 `import "github.com/tencentyun/scf-go-lib/events"` 来直接使用。如果使用过程中发现问题，可以通过 提交 issue 或 提交工单 解决。

返回

入口函数可以带有0 - 2个返回值，例如：



```
func hello()()
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函数。

部署方法

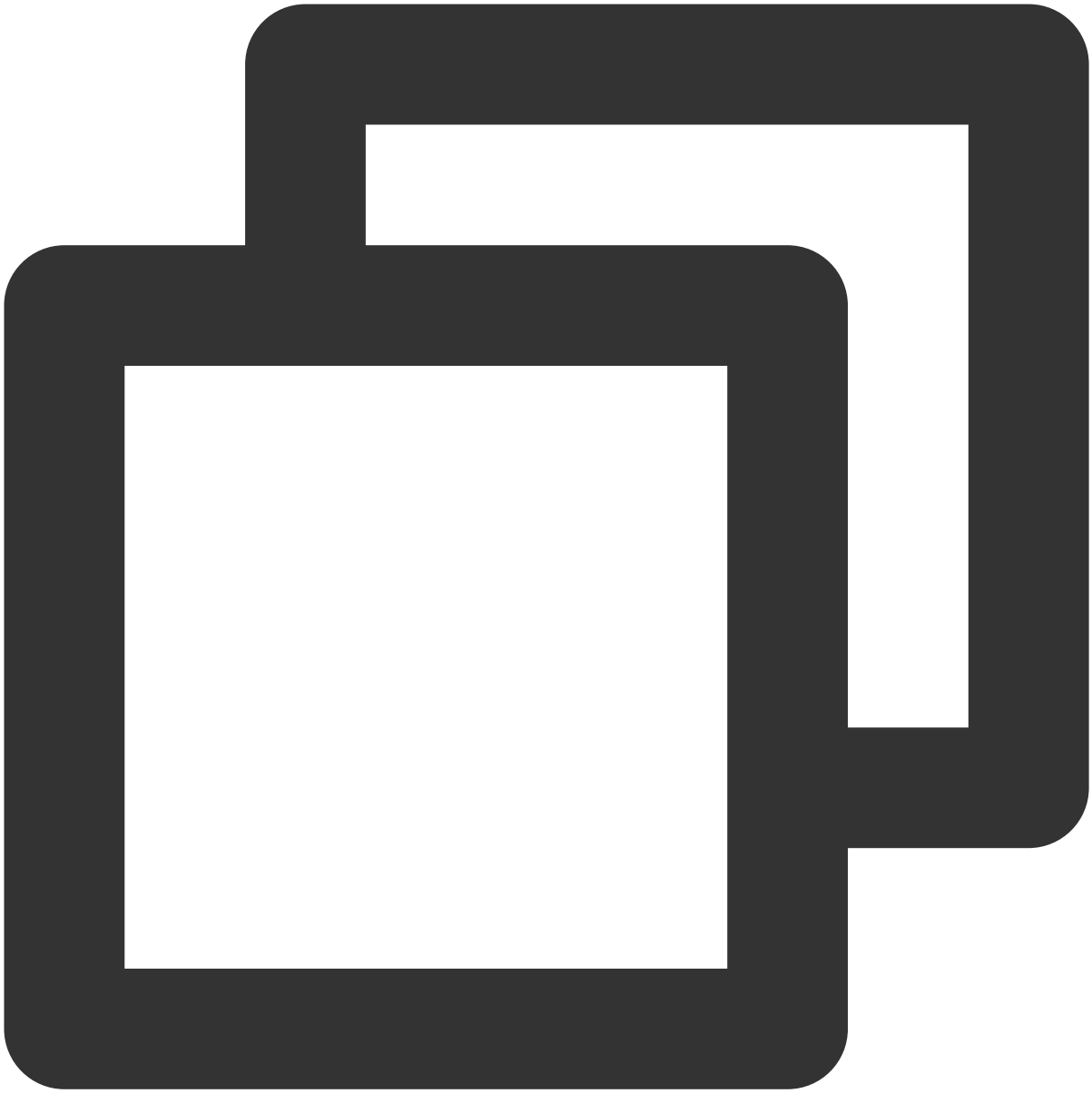
最近更新时间：2024-04-22 18:03:28

部署方法

Golang 环境的云函数，仅支持 zip 包上传，您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。

编译打包

Golang 编译可以在任意平台上通过指定 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 的终端通过如下方法完成编译及打包：

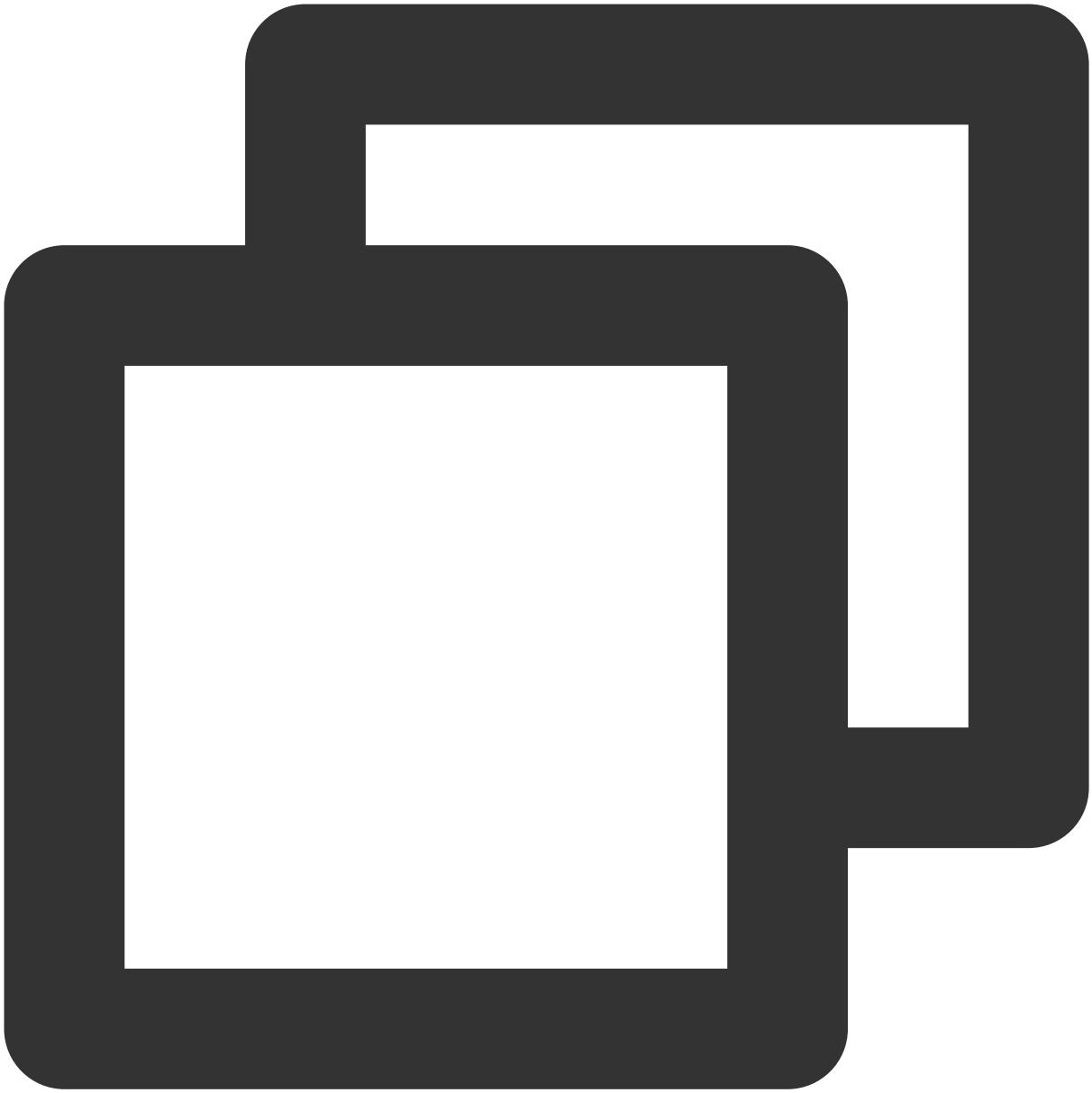


```
GOOS=linux GOARCH=amd64 go build -o main main.go
zip main.zip main
```

在 Windows 下，请按照以下步骤进行编译打包：

1.1 按 **Windows + R** 打开运行窗口，输入 **cmd** 后按 **Enter**。

1.2 执行以下命令，进行编译。



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

1.3 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。

日志说明

最近更新时间：2024-04-22 18:03:28

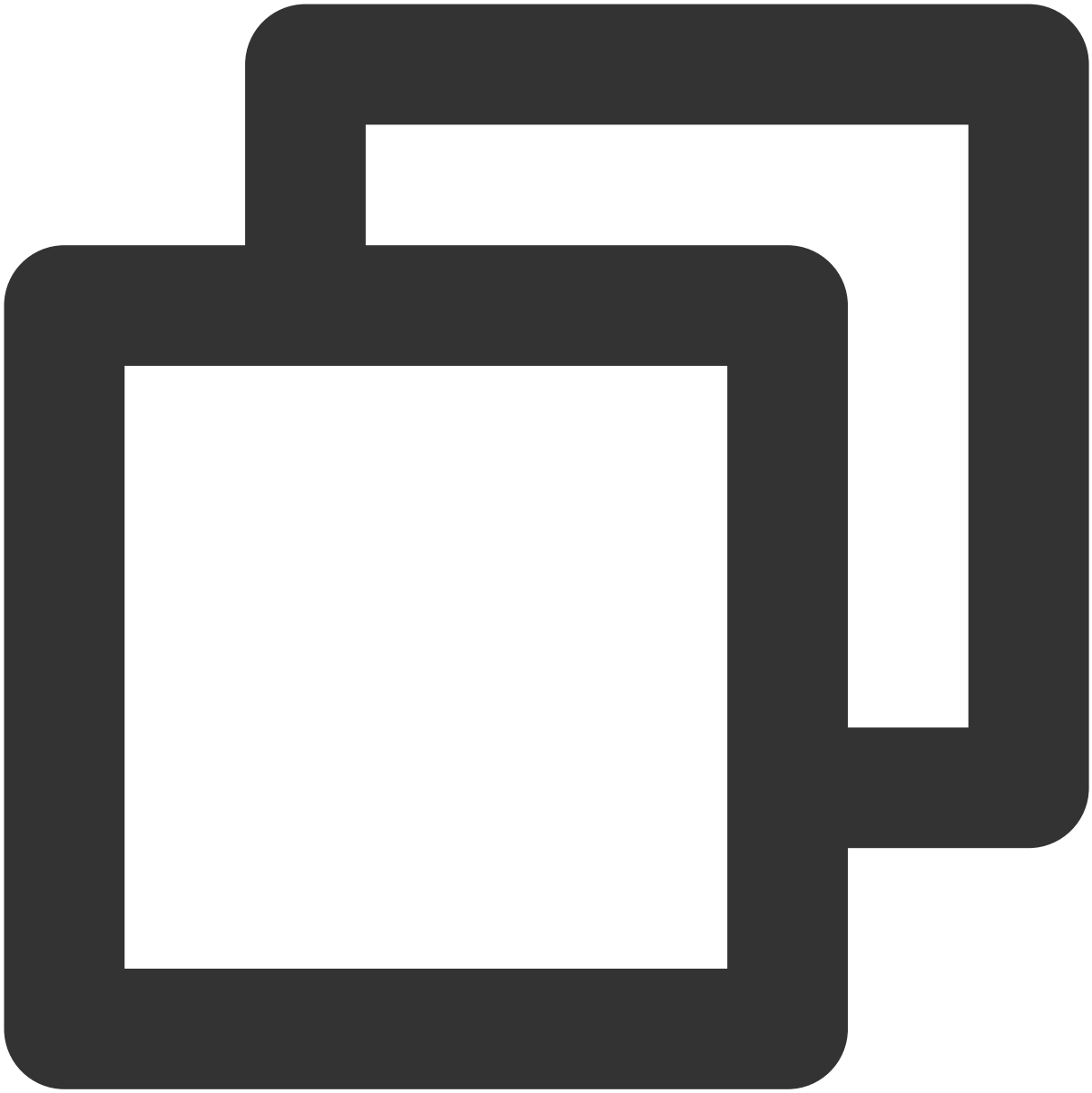
日志开发

您可以在程序中使用如下语句来完成日志输出：

`fmt.Println`

或使用 `fmt.Sprintf` 类似方法

例如，执行以下代码，可以在函数日志中查询输出内容。



```
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)
}
```

日志查询

当前函数日志均会投递至腾讯云日志服务 CLS 中，您可对函数日志进行投递配置，详情可参见 日志投递配置。
您可通过云函数的日志查询界面或通过日志服务的查询界面，查询函数执行日志。日志查询方法详情可参见 日志检索教程。

说明：

函数日志投递到日志服务日志集 LogSet 和日志主题 LogTopic，均可以通过函数配置查询。

自定义日志字段

当前在函数代码中使用简单日志打印语句，将会在投递到日志服务时，记录在 `SCF_Message` 字段中。日志服务的字段说明可见 索引说明。
目前云函数已经支持在输出到日志服务的内容中增加自定义字段，通过增加自定义字段，您可以将业务字段及相关数据内容输出到日志中，并通过使用日志服务的检索能力，对执行过程中的业务数据及相关内容进行查询跟踪。

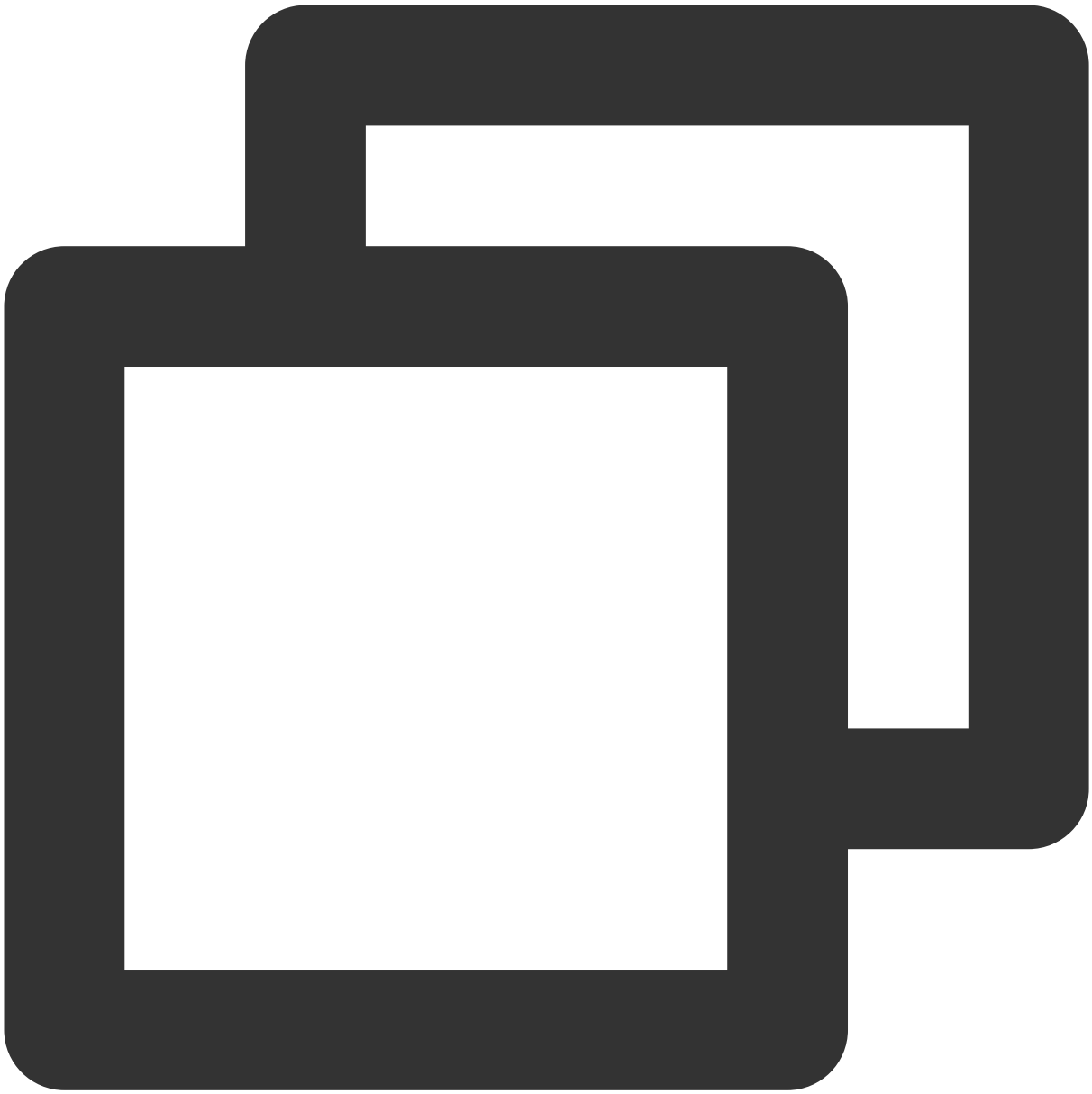
注意：

如需对自定义字段进行键值查询，如 `SCF_CustomKey：SCF` ，请参考 日志服务索引配置 为函数日志投递的日志主题添加键值索引。
为避免误操作索引配置导致函数日志查询失败，函数配置的默认投递日志主题（以 `SCF_LogTopic_` 为前缀命名）不支持修改索引配置。请将函数日志投递主题设置为 自定义投递 后再更新日志主题索引配置。
日志主题修改索引配置后，仅对新写入的数据有效。

输出方法

当函数输出的单行日志为 JSON 格式时，JSON 内容将被解析并在投递至日志服务时按 `字段:值` 的方式进行投递。JSON 内容的解析仅能解析第一层，更多的嵌套结构将作为值进行记录。

您可执行以下代码进行测试：

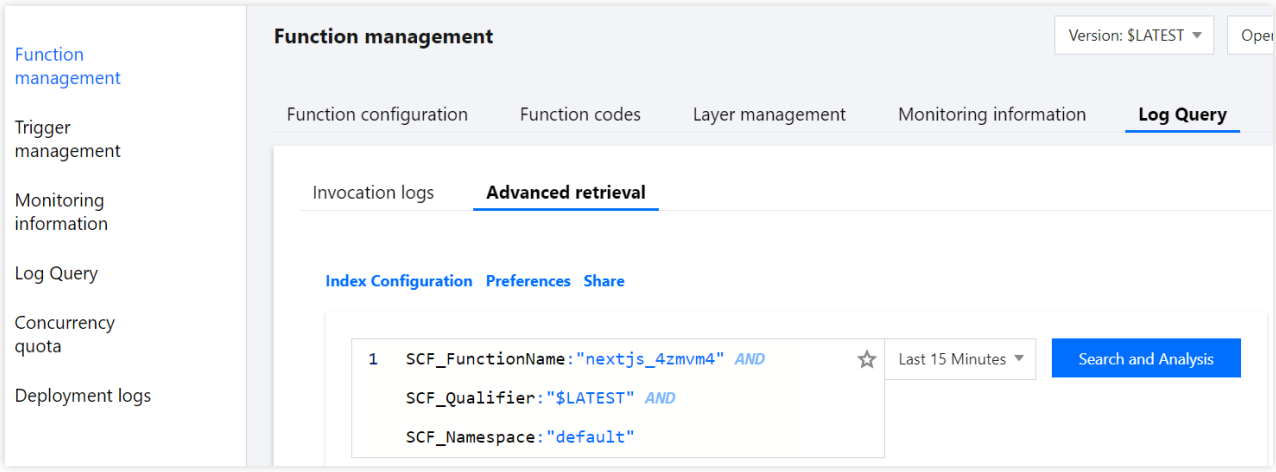


```
package main

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

type DefineEvent struct {
```

```
    Key1 string `json:"key1"`
    Key2 string `json:"key2"`
}

func hello(ctx context.Context, event DefineEvent) (string, error) {
    m := map[string]string{"key1": "test value 1", "key2": "test value 2"}
        data, _ := json.Marshal(m)
        fmt.Println(string(data))
    return fmt.Sprintf("hello world"), nil
}

func main() {
    cloudfunction.Start(hello)
}
```

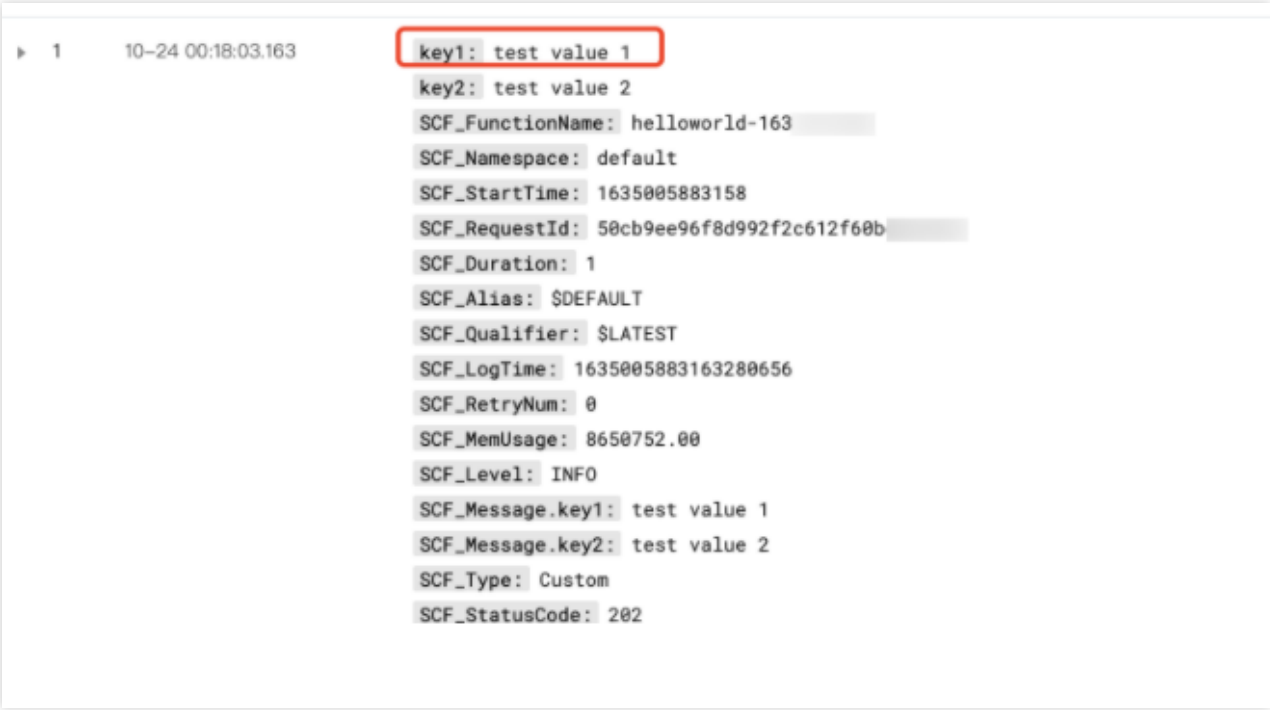
检索方法

在使用上述代码进行测试运行后，您可在函数-日志查询-高级检索中通过如下语句进行检索：



检索结果

在测试写入日志服务后，您可以在日志查询中检索到 `key1` 字段。如下图所示：


1
10-24 00:18:03.163
key1: test value 1
key2: test value 2
SCF_FunctionName: helloworld-163
SCF_Namespace: default
SCF_StartTime: 1635005883158
SCF_RequestId: 50cb9ee96f8d992f2c612f60b
SCF_Duration: 1
SCF_Alias: $DEFAULT
SCF_Qualifier: $LATEST
SCF_LogTime: 1635005883163280656
SCF_RetryNum: 0
SCF_MemUsage: 8650752.00
SCF_Level: INFO
SCF_Message.key1: test value 1
SCF_Message.key2: test value 2
SCF_Type: Custom
SCF_StatusCode: 202

PHP
环境说明

最近更新时间：2024-04-22 18:03:28

PHP 版本选择

云函数 SCF 目前支持的 PHP 开发语言包括如下版本：

PHP 8.0

PHP 7.4

PHP 7.2

PHP 5.6

您可以在函数创建时，选择您所期望使用的运行环境，PHP 8.0、PHP 7.4、PHP 7.2 或 PHP 5.6。

相关环境变量

目前 PHP 8.0、PHP 7.4 运行环境中内置的 PHP 相关环境变量见下表：

环境变量 Key	具体值或值来源
`PHP_INI_SCAN_DIR`	/opt/php_extension:/var/user/php_extension

目前 PHP 7.2、PHP 5.6 运行环境中内置的 PHP 相关环境变量见下表：

环境变量 Key	具体值或值来源
`PHP_INI_SCAN_DIR`	/var/user/php_extension:/opt/php_extension

更多详细环境变量说明请参见 环境变量说明。

内置的扩展列表

注意：

PHP 7.4 及之后版本，平台不再额外内置依赖库。代码运行所需依赖，请参考 依赖安装 进行安装。

如内置扩展不足以满足业务要求，可参考 依赖安装 进行自定义扩展安装。

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

如下列出目前已安装的 PHP 扩展：

PHP 8.0、PHP 7.4

Core	calendar	gd	mysqlnd	SimpleXML
runkit7	ctype	SPL	PDO	soap
date	curl	iconv	pdo_mysql	exif
libxml	dom	intl	pdo_sqlite	tokenizer
openssl	hash	json	Phar	xml
pcre	fileinfo	mbstring	posix	xmlreader
sqlite3	filter	session	Reflection	xmlwriter
zlib	ftp	standard	mysqli	runtime
bcmath				

PHP 7.2

Core	curl	mbstring	mysqli	zip
runkit7	dom	session	SimpleXML	eio
date	hash	standard	soap	memcached
libxml	fileinfo	mysqlnd	sockets	imagick
openssl	filter	PDO	exif	mongodb
pcre	ftp	pdo_mysql	tidy	protobuf
sqlite3	gd	pdo_sqlite	tokenizer	redis
zlib	SPL	Phar	xml	swoole
bcmath	iconv	posix	xmlreader	Zend OPcache

bz2 calendar ctype	json	Reflection	xmlwriter	runtime

PHP 5.6

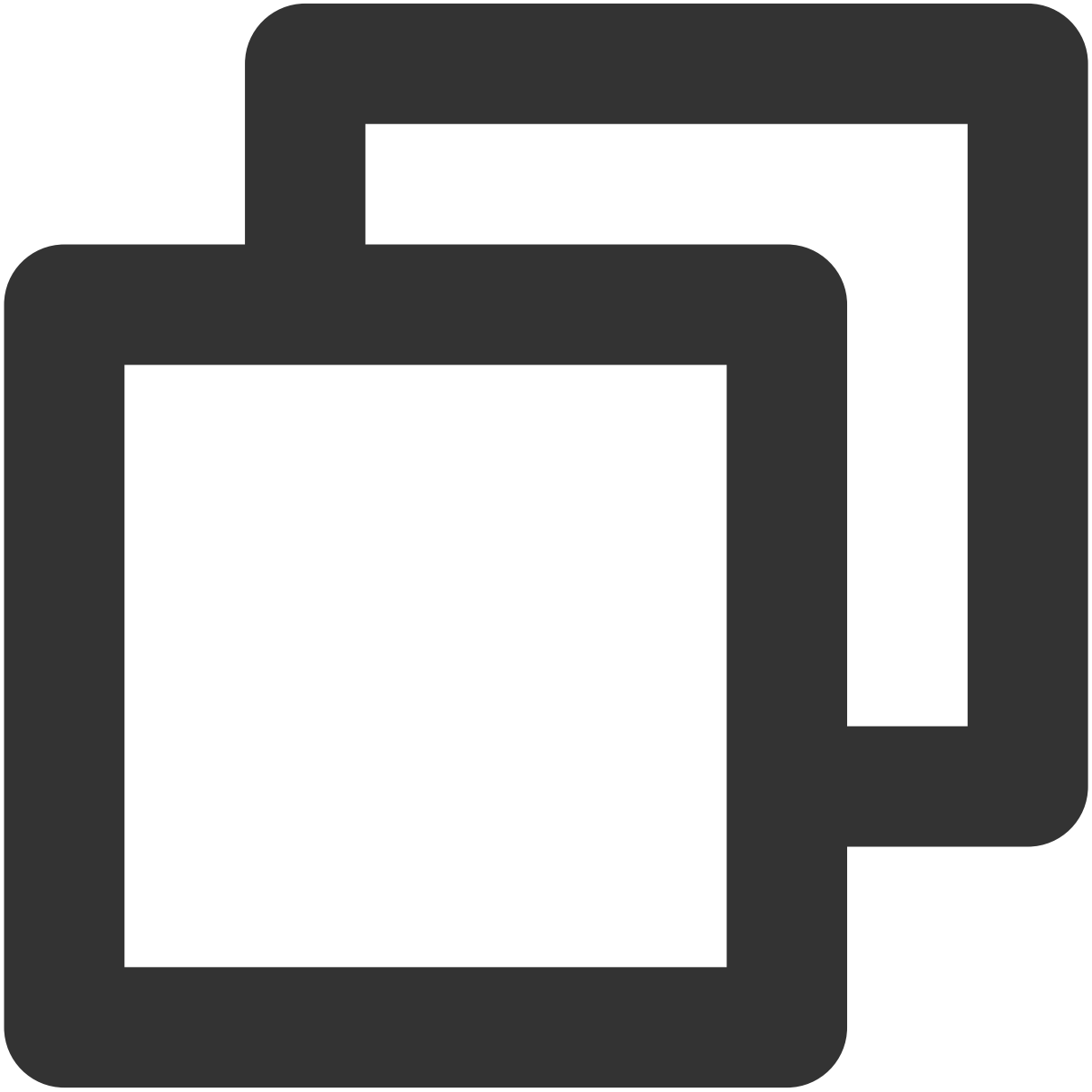
Core runkit date ereg libxml openssl pcre sqlite3 zlib bcmath bz2 calendar	ctype curl dom hash fileinfo filter ftp gd SPL iconv	json mbstring session standard mysqlnd PDO pdo_mysql pdo_sqlite Phar posix	Reflection mysqli SimpleXML soap sockets exif tidy tokenizer xml xmlreader	xmlwriter zip eio memcached imagick mongodb protobuf redis Zend OPcache runtime

开发方法

最近更新时间：2024-04-22 18:03:28

函数形态

PHP 函数形态一般如下所示：



```
<?php
```

```
function main_handler($event, $context) {
    print_r ($event);
    print_r ($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 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 包的根目录下包含指定的入口文件，文件内有定义指定的入口函数、文件名、函数名和执行方法，避免因为无法查找到入口文件和入口函数导致执行失败。

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。
event 的具体内容，根据不同触发器或调用来源而变化。详细数据结构说明请参见 触发器相关说明。
注意：
PHP 8.0、PHP 7.4、PHP 7.2、PHP 5.6 入参 event 和 context 格式为 `object`。

返回

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。
在 PHP 环境下，您可以直接返回一个可序列化的对象。例如，返回一个 `dict` 对象：
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来进行后续处理。例如，通过控制台进行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。
说明：
无论同步调用还是异步调用，返回值均会在函数的日志中记录。返回值将写入函数调用日志的 `SCF_Message` 中，格式为 `Response RequestId:xxx RetMsg:xxx`。
`SCF_Message` 的值长度限制为8KB，超出时将截取前8KB。

异常处理

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。

部署方法

最近更新时间：2024-04-22 18:03:28

部署方法

腾讯云云函数提供以下几种方式部署函数，您可以按需选择使用。创建、更新函数操作详情可参见 创建及更新函数。
通过 ZIP 打包上传部署，详情可参见 依赖安装和部署。
通过控制台编辑和部署，详情可参见 通过控制台部署函数。
使用命令行部署，详情可参见 通过 Serverless Framework CLI 命令行部署函数。

依赖安装和部署

当前的函数标准 PHP 中仅提供 `/tmp` 目录可写，其他目录只读，因此在使用依赖库时，需要使用本地安装、打包、上传的方式。PHP 依赖包可以与函数代码一同上传使用。

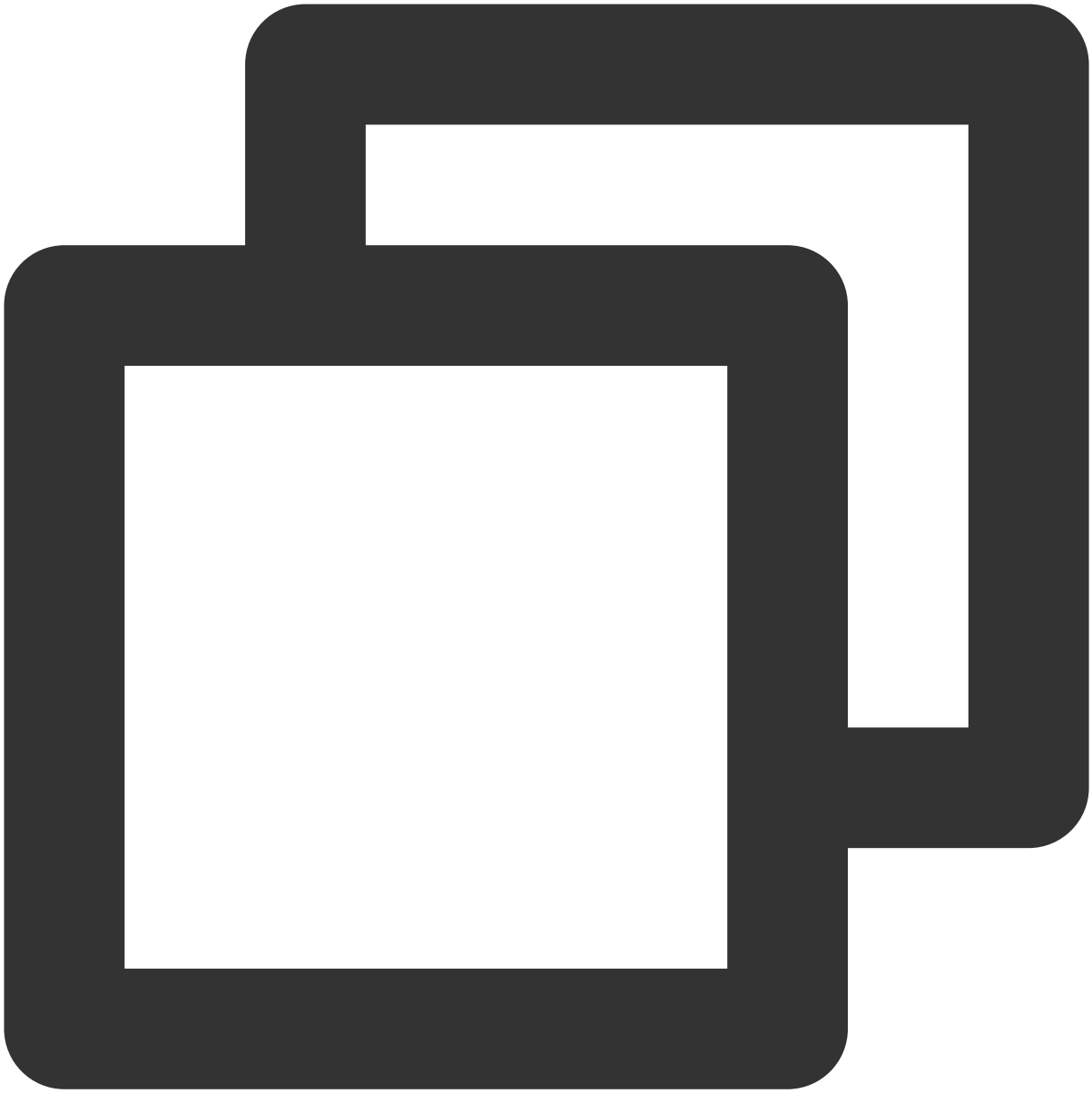
本地安装依赖包

依赖管理工具

PHP 可以通过 composer 包管理器进行依赖管理。

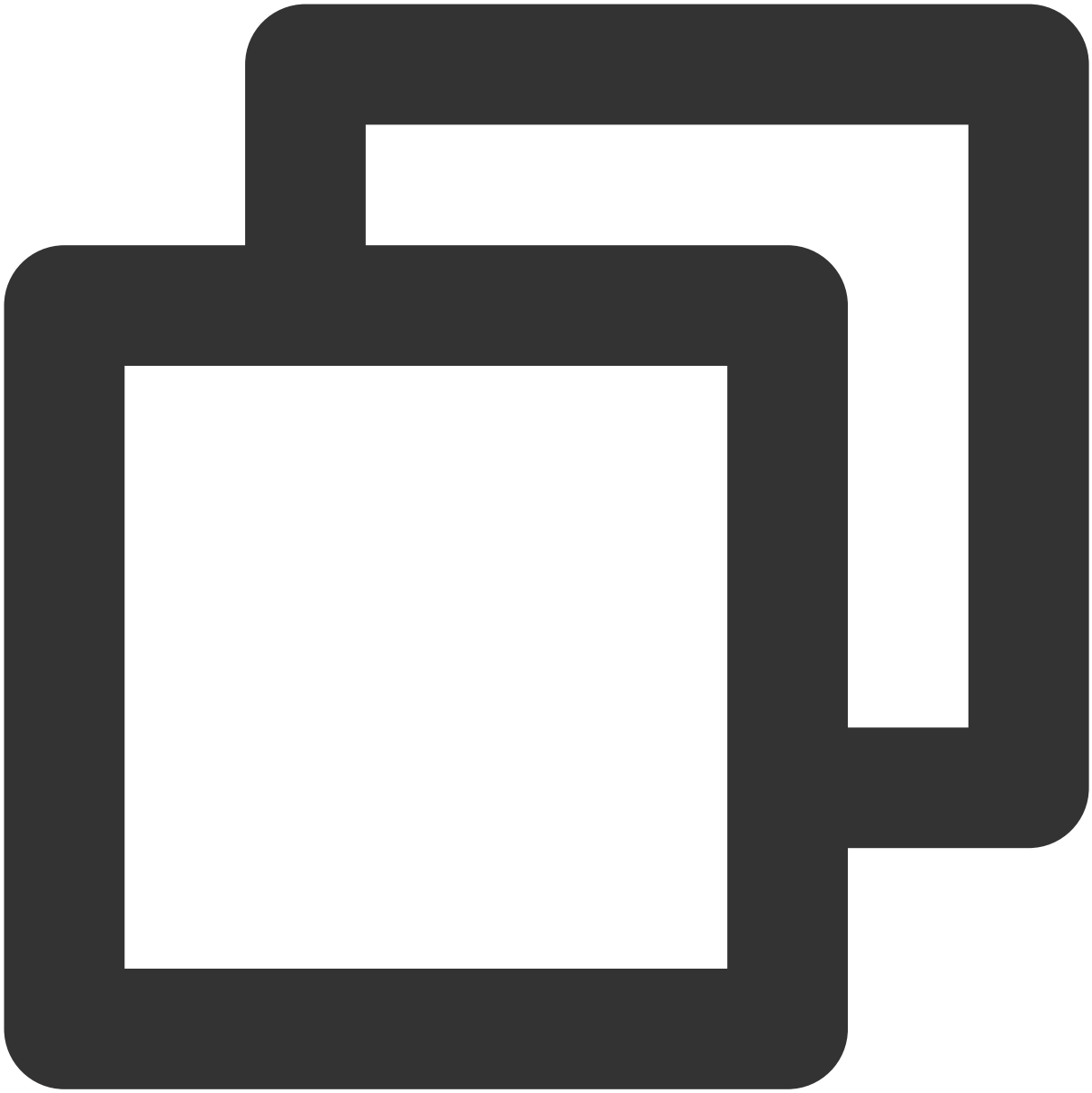
操作步骤

1. 本地新建文件夹 `/code` 用于存放函数代码及依赖文件。在代码根目录下新建依赖包配置文件 `composer.json` 并配置依赖信息。以安装 `requests` 为例， `composer.json` 文件如下：



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

2. 在 `/code` 文件夹下执行以下命令，即可按照配置文件中指定的依赖包及版本进行安装。



```
composer install
```

注意：

函数运行的系统为 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找不到依赖的错误。详情可参见 云函数容器镜像 进行依赖安装。

打包上传

依赖可以和项目一同上传，并在函数代码中通过 `require` 方式引入和使用。

您可以通过控制台选择本地文件夹的方式自动化打包，也可以通过手工打包的方式形成可以用于部署函数的 ZIP 包。在打包部署时，需要在项目目录下进行打包操作，即确保代码、依赖均在 ZIP 文件内的根目录中。详情可参见打包要求。

日志说明

最近更新时间：2024-04-22 18:03:28

日志开发

您可以在程序中使用以下语句来完成日志输出：

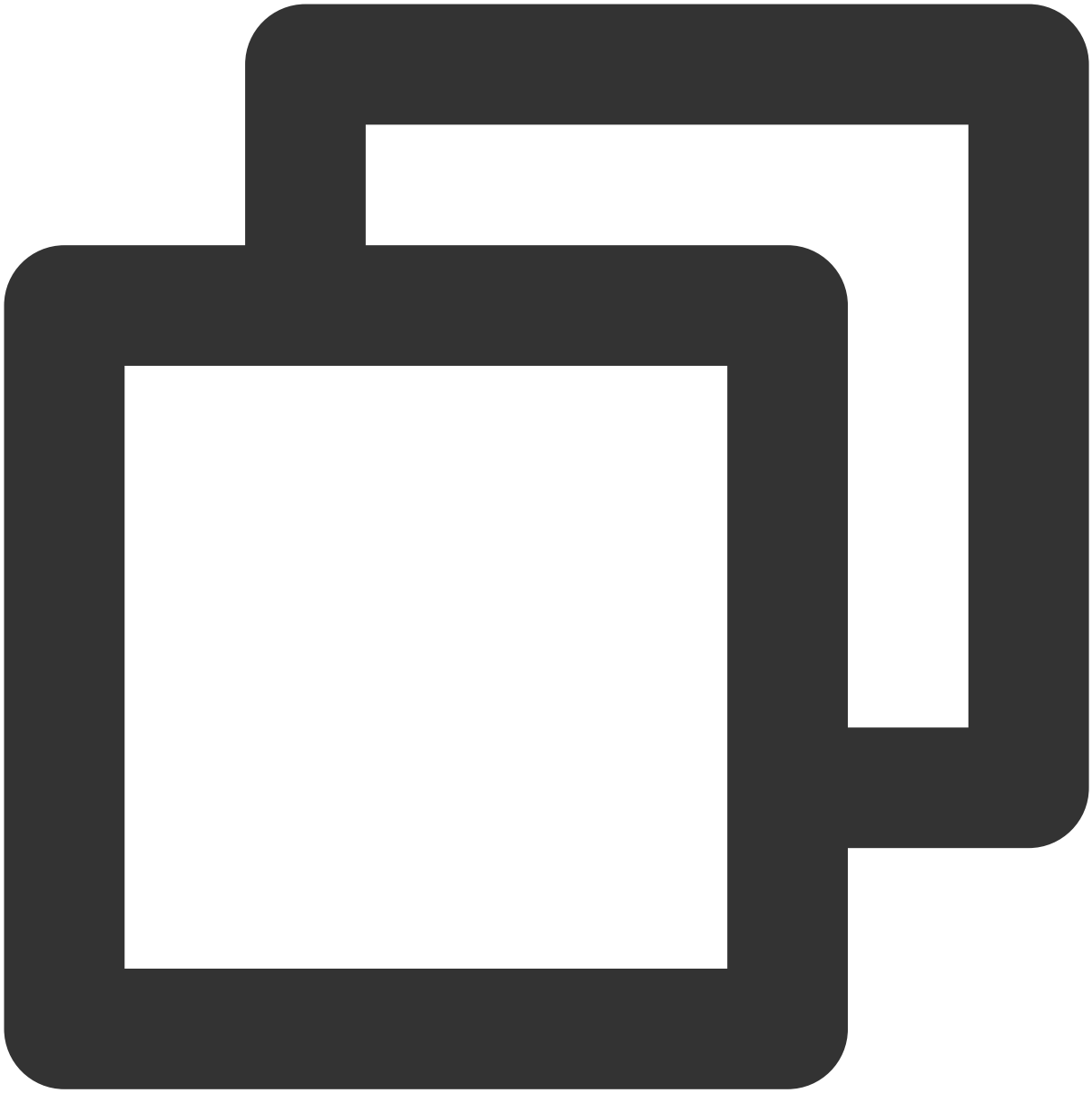
echo 或 echo()

print 或 print()

print_r()

var_dump()

例如，执行以下代码，可以在函数日志中查询输出内容。



```
<?php
function main_handler($event, $context) {
    print_r ($event);
    print_r ($context);
    echo "hello world\\n";
    print "hello world\\n";
    var_dump ($event);
    return "hello world";
}
?>
```

日志查询

当前函数日志均会投递至腾讯云日志服务 CLS 中，您可对函数日志进行投递配置，详情可参见 日志投递配置。
您可通过云函数的日志查询界面或通过日志服务的查询界面，查询函数执行日志。日志查询方法详情可参见 日志检索教程。

说明：

函数日志投递到日志服务日志集 LogSet 和日志主题 LogTopic，均可以通过函数配置查询。

自定义日志字段

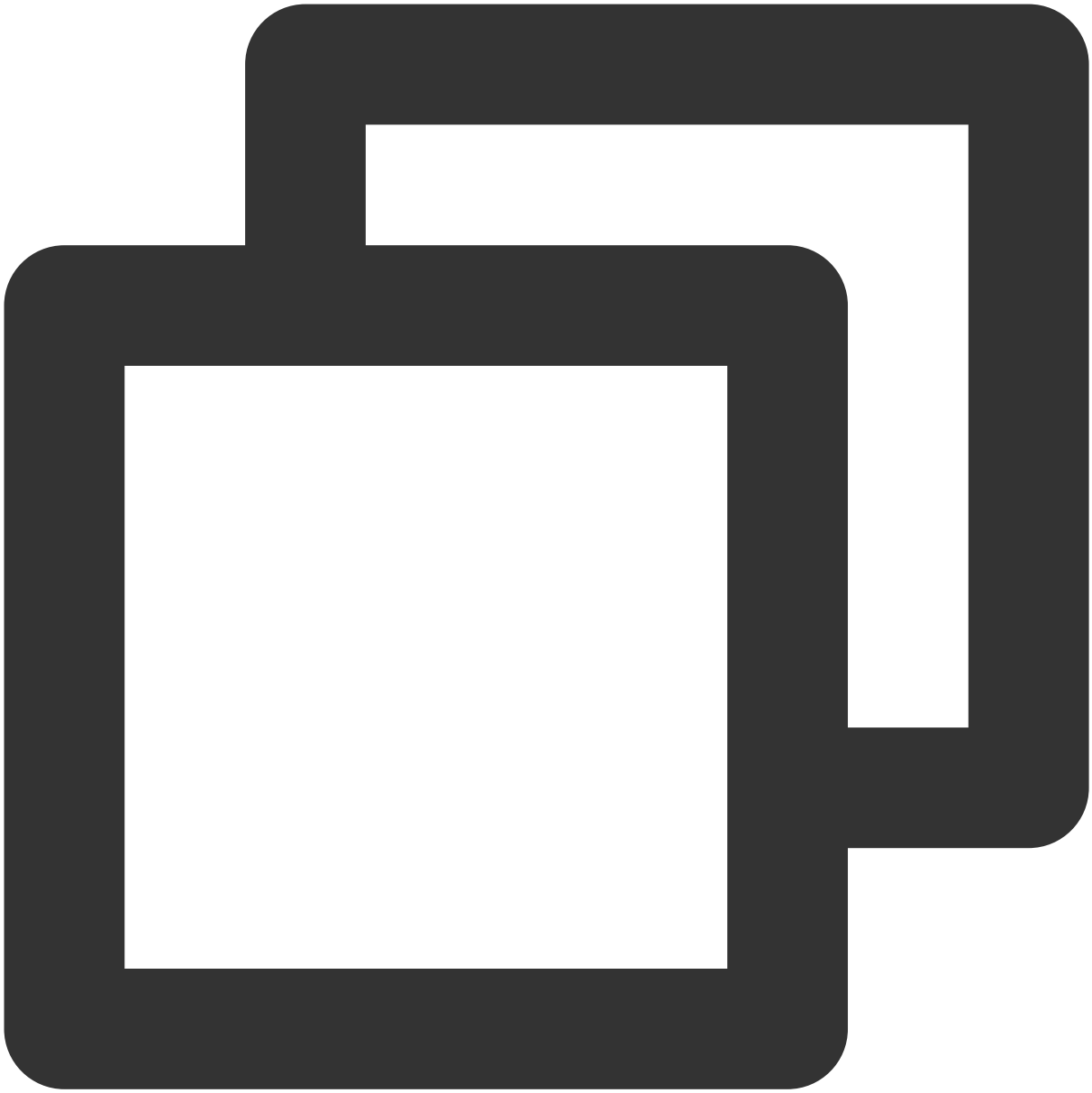
当前在函数代码中通过简单的 `print`，或通过 `logger` 输出的字符串内容，将会在投递到日志服务时，记录在 `SCF_Message` 字段中。日志服务的字段说明可参见 索引说明。
目前云函数已经支持在输出到日志服务的内容中增加自定义字段，通过增加自定义字段，您可以将业务字段及相关数据内容输出到日志中，并通过使用日志服务的检索能力，对执行过程中的业务数据及相关内容进行查询跟踪。

注意：

如需对自定义字段进行键值查询，如 `SCF_CustomKey：SCF`，请参见 日志服务索引配置 为函数日志投递的日志主题添加键值索引。
为避免误操作索引配置导致函数日志查询失败，函数配置的默认投递日志主题（以 `SCF_LogTopic_` 为前缀命名）不支持修改索引配置。请将函数日志投递主题设置为 自定义投递 后再更新日志主题索引配置。
日志主题修改索引配置后，仅对新写入的数据有效。

输出方法

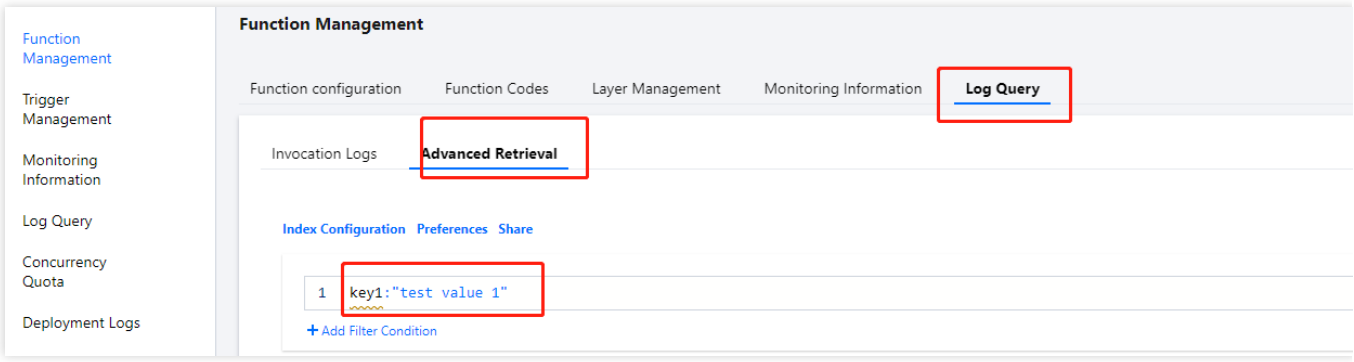
当函数输出的单行日志为 JSON 格式时，JSON 内容将被解析并在投递至日志服务时按“**字段:值**”的方式进行投递。
JSON 内容的解析仅能解析第一层，更多的嵌套结构将作为值进行记录。
您可执行以下代码进行测试：



```
<?php
function main_handler($event, $context) {
    $custom_key = array('key1' => 'test value 1', 'key2' => 'test value 2');
    echo json_encode($custom_key);
    return "hello world";
}
?>
```

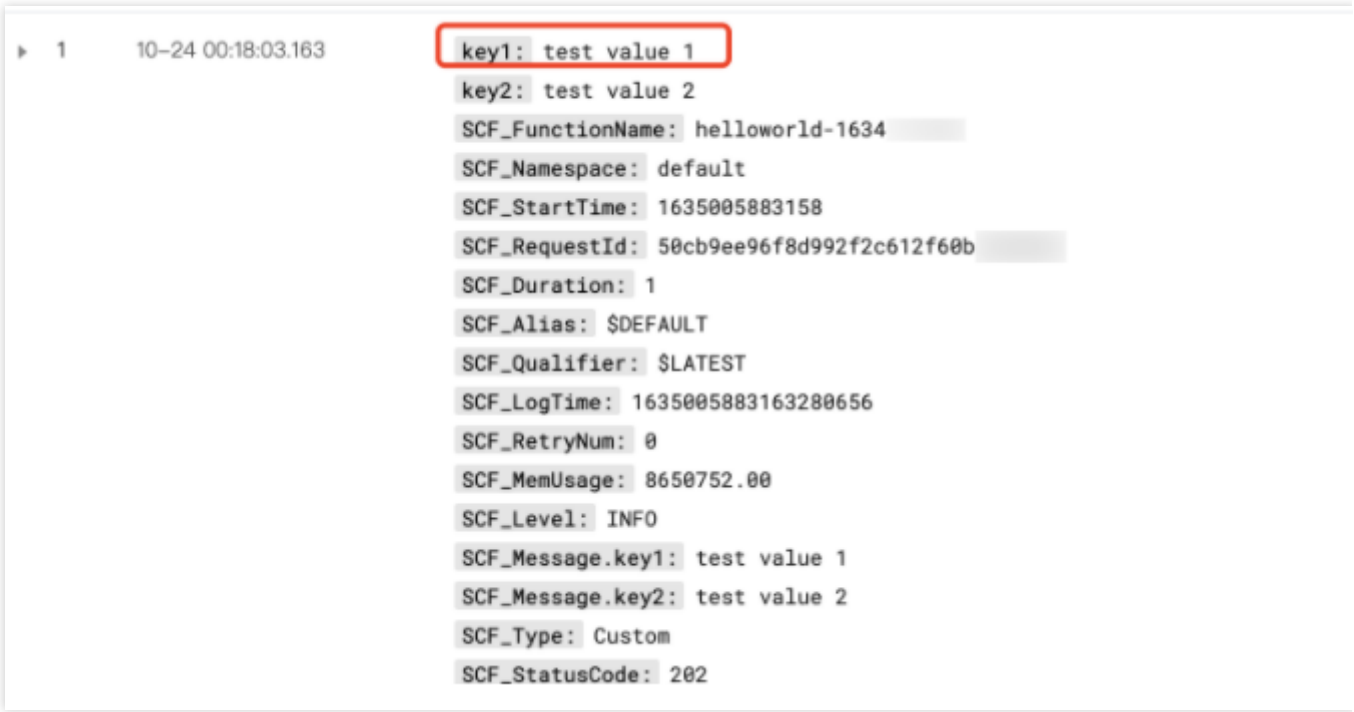
检索方法

在使用上述代码进行测试运行后，您可在函数-日志查询-高级检索中通过如下语句进行检索：



检索结果

在测试写入日志服务后，您可以在日志查询中检索到 `key1` 字段。如下图所示：



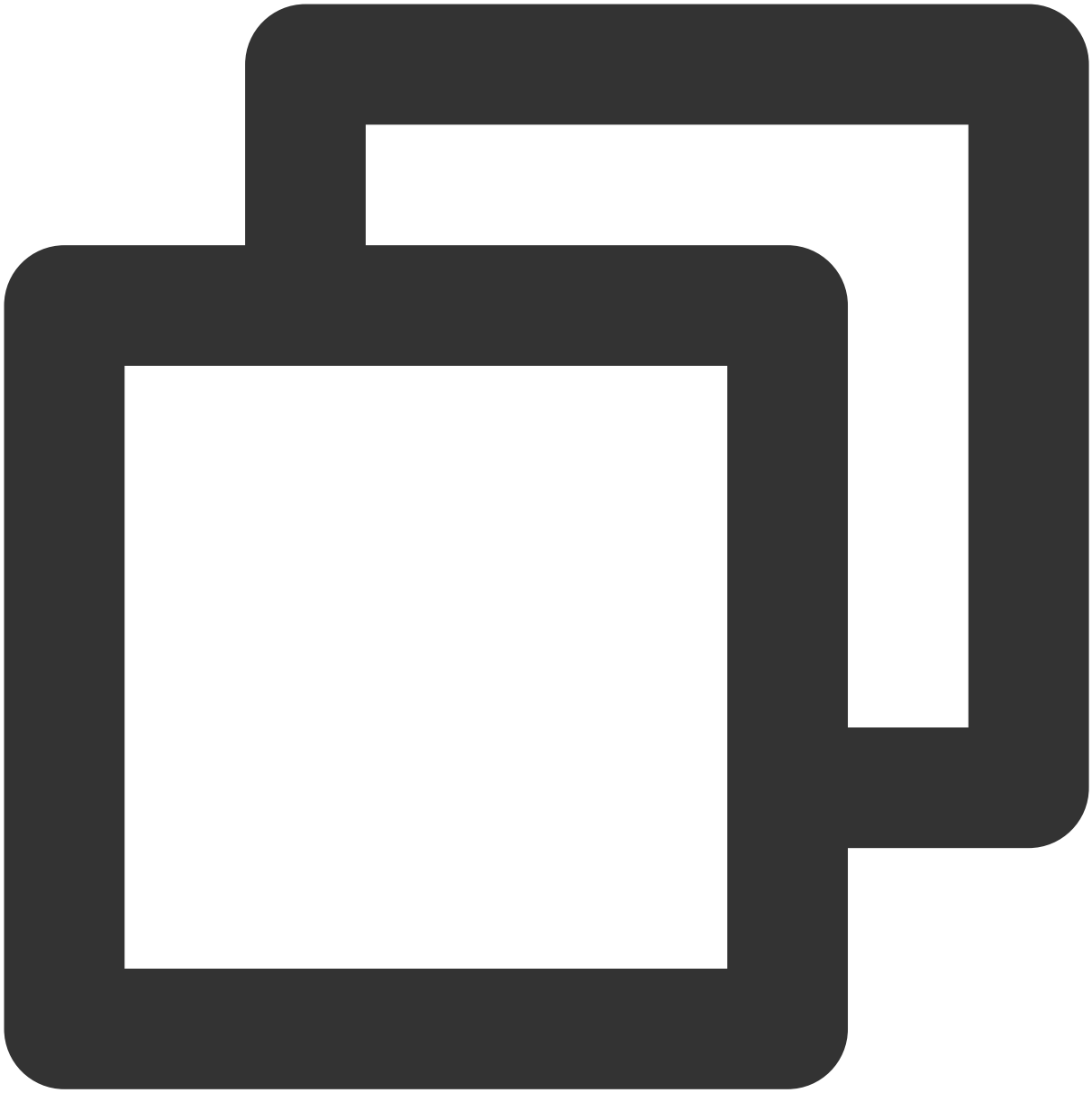
常见示例

最近更新时间：2024-04-22 18:03:28

常见示例中包含了 PHP 环境下可以试用的相关代码片段，您可以根据需要选择尝试。示例均基于 PHP 7.2 环境提供。
您可以从 GitHub 项目 scf-php-code-snippet 中获取相关代码片段并直接部署。

环境变量读取

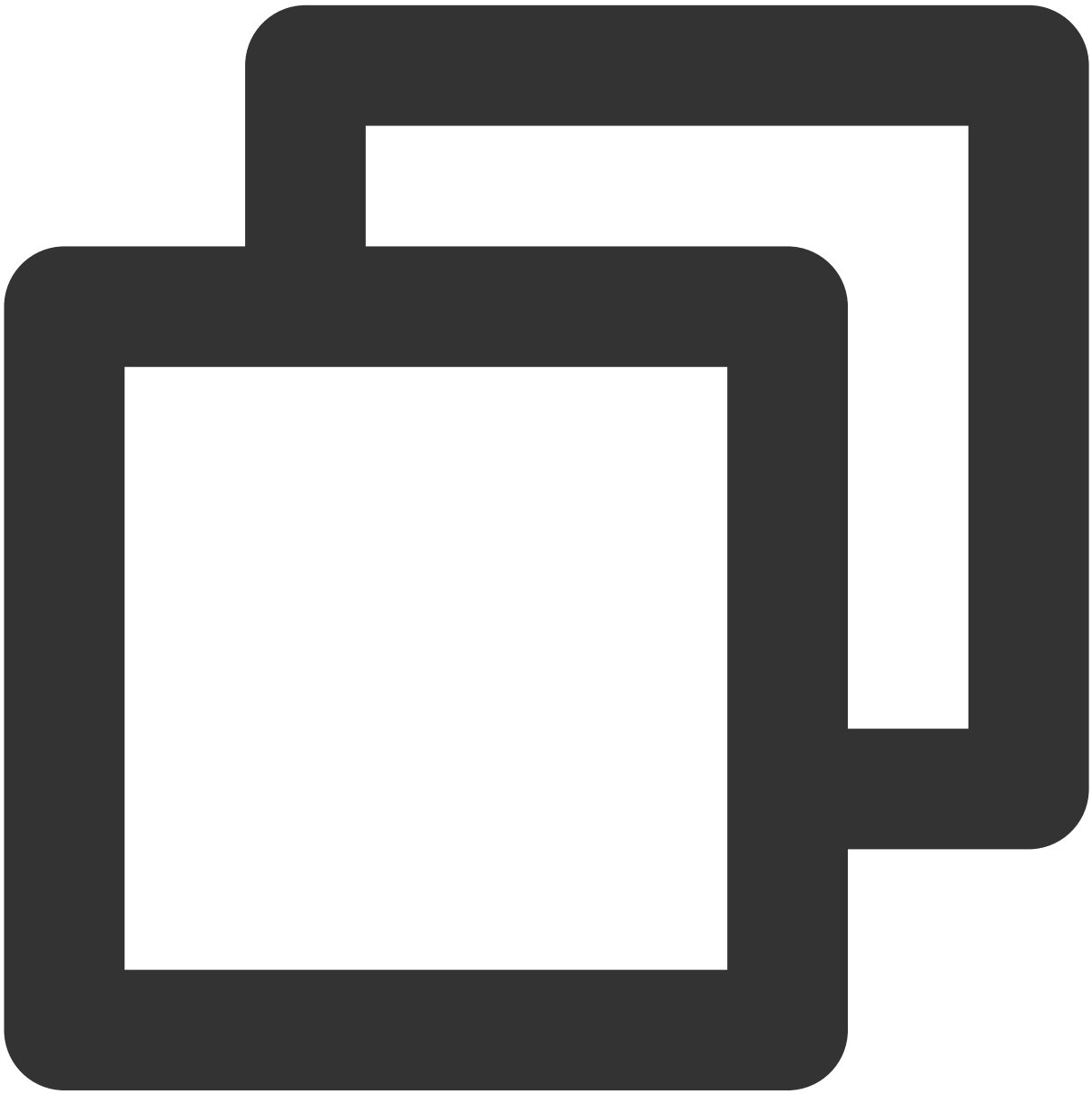
本示例提供了获取全部环境变量列表，或单一环境变量值的方法。示例如下：



```
<?php
function main_handler($event, $context) {
    print_r($_ENV);
    echo getenv('SCF_RUNTIME');
    return "hello world";
}
?>
```

本地时间格式化输出

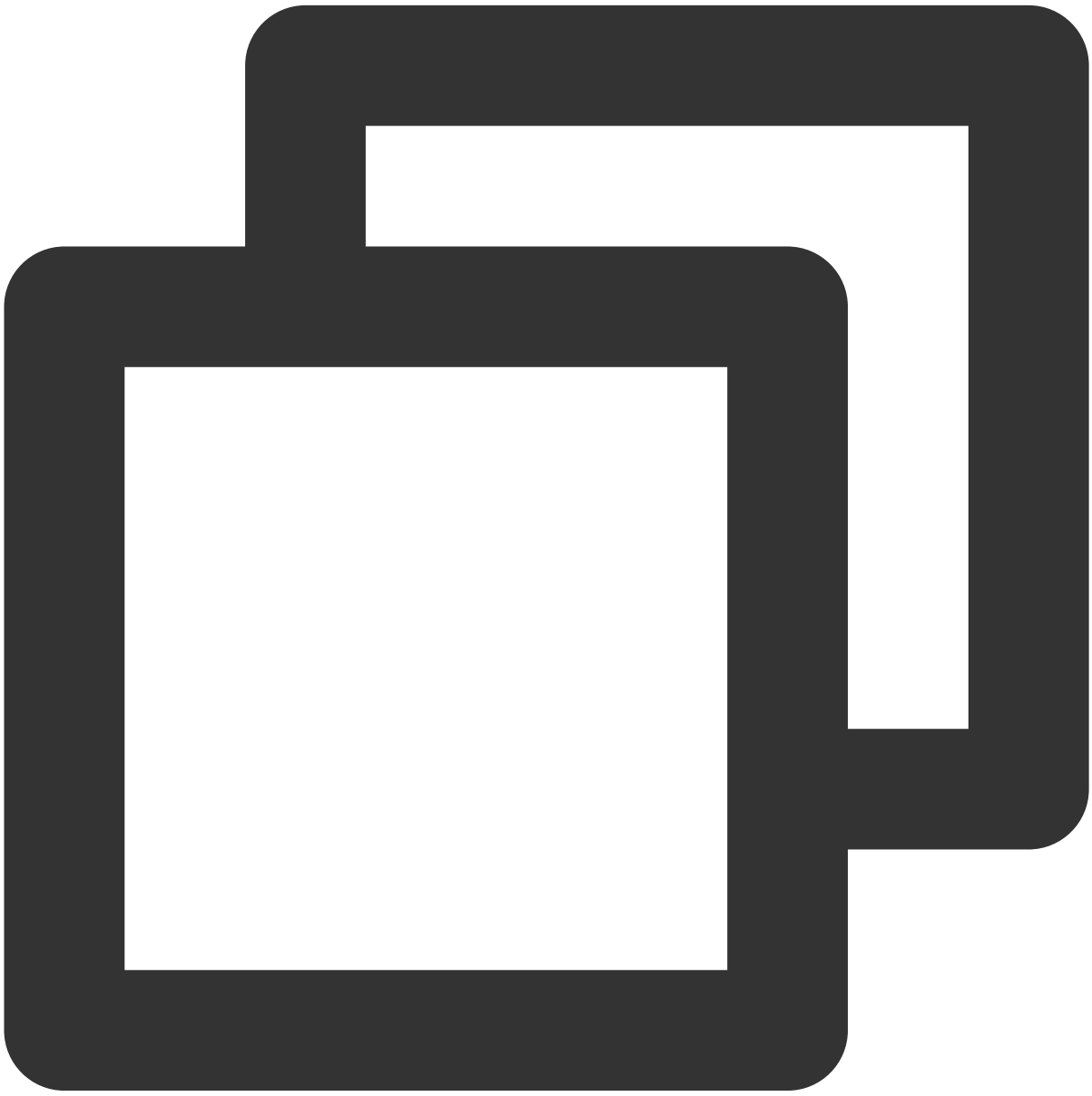
本示例提供了时间格式化输出方法，按指定格式进行日期和时间输出。
SCF 环境默认是 UTC 时间，如果期望按北京时间输出，可以为函数添加 `TZ=Asia/Shanghai` 环境变量，并在函数代码中通过 `date_default_timezone_set(getenv('TZ'));` 设置需要使用的时区。示例如下：



```
<?php
function main_handler($event, $context) {
    date_default_timezone_set(getenv('TZ'));
    echo date("Y-m-d H:i:s",time());
```

```
    return "hello world";
}
?>
```

函数内发起网络连接



```
<?php
function main_handler($event, $context) {
```

```
    $url = 'https://cloud.tencent.com';
    echo file_get_contents($url);
    return "hello world";
}
?>
```

Java
环境说明

最近更新时间：2024-04-22 18:03:28

Java 版本选择

云函数 SCF 目前支持的 Java 开发语言包括如下版本：
Java 11（Kona JDK）
Java 8（Open JDK）
您可以在函数创建时，选择您所期望使用的运行环境，Java 11 或 Java 8。
SCF Java 11 基于腾讯 Kona 提供，腾讯 Kona（Tencent Kona）基于 OpenJDK，由腾讯专业技术团队提供技术维护、优化及安全保障。腾讯云团队针对 Kona 在云场景的支撑及特性进行了开发及优化，使其更加适合云场景下的 Java 业务，为您提供最优的 Java 云生产环境及解决方案。

相关环境变量

Java 11 和 Java 8 运行环境中内置的相关环境变量见下表：

Java 11

环境变量 Key	具体值或值来源
`CLASSPATH`	/var/runtime/java11:/var/runtime/java11/lib/*

Java 8

环境变量 Key	具体值或值来源
`CLASSPATH`	/var/runtime/java8:/var/runtime/java8/lib/*:/opt

更多详细环境变量说明请参见 环境变量说明。

注意事项

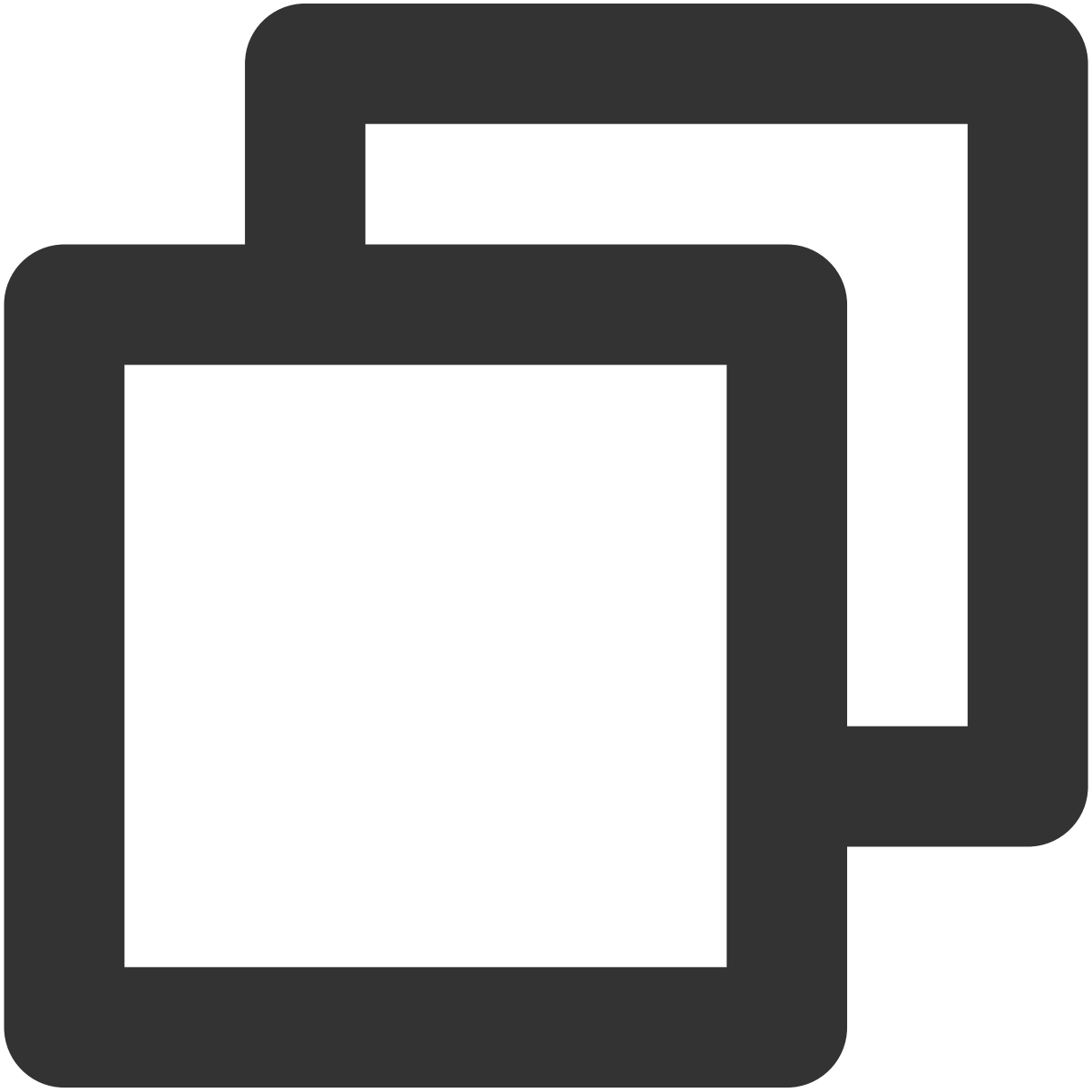
Java 语言由于需要编译后才可以在 JVM 虚拟机中运行。因此在 SCF 中的使用方式，和 Python、Node.js 这类脚本型语言不同，有如下限制：
不支持上传代码：使用 Java 语言仅支持上传已经开发完成编译打包后的ZIP包或JAR包。SCF 环境不提供 Java 的编译能力。
不支持在线编辑：由于不支持上传代码，所以不支持在线编辑代码。Java 运行时的函数，在代码页面仅能看到通过页面上传或 COS 提交代码的方法。

开发方法

最近更新时间：2024-04-22 18:03:28

代码形态

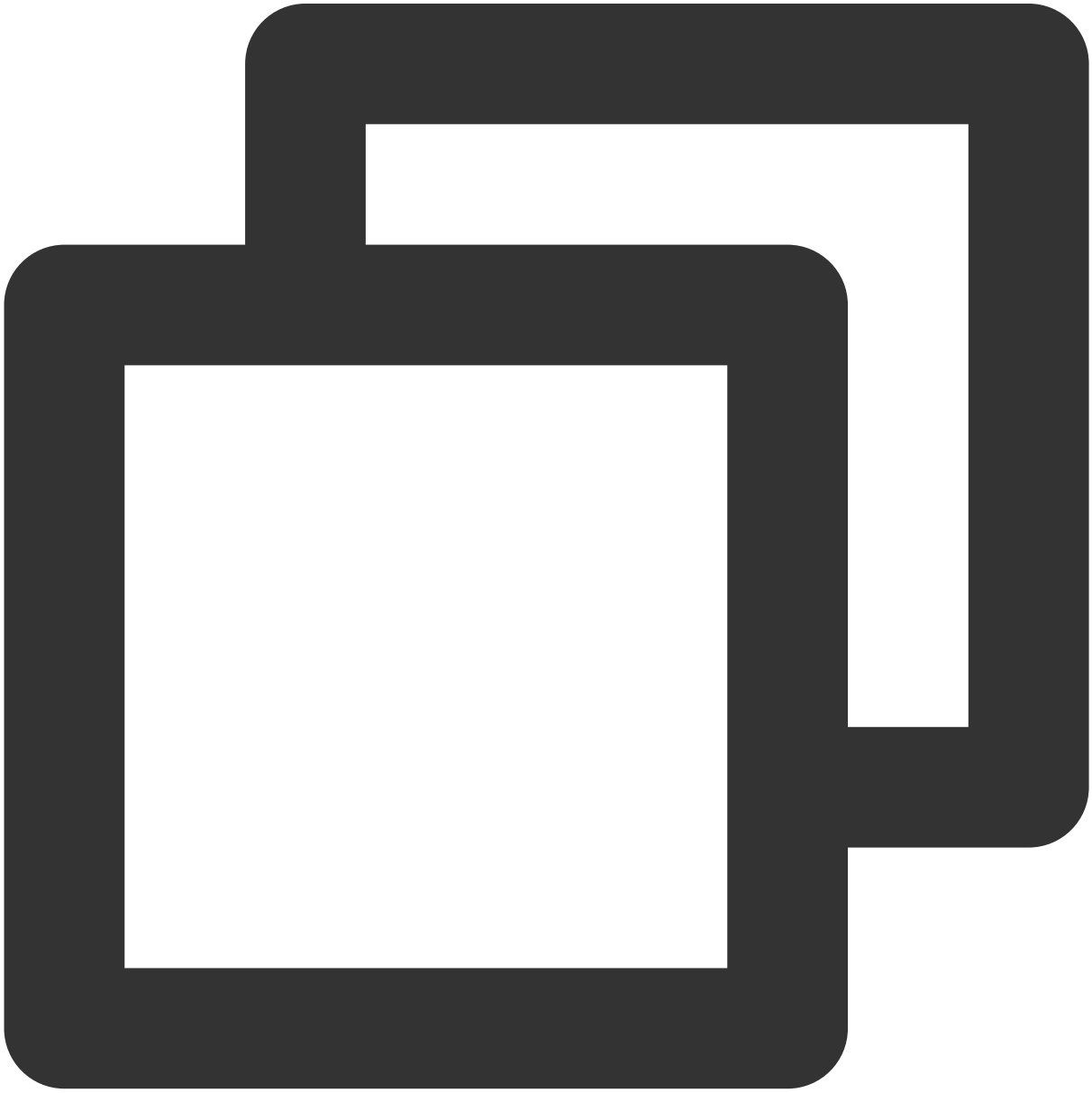
Java 开发的 SCF 云函数的代码形态一般如下所示：



```
package example;
```

```
public class Hello {
    public String mainHandler(KeyValueClass kv) {
        System.out.println("Hello world!");
        System.out.println(String.format("key1 = %s", kv.getKey1()));
        System.out.println(String.format("key2 = %s", kv.getKey2()));
        return String.format("Hello World");
    }
}
```

建立参数 KeyValueClass 类：



```
package example;
public class KeyValueClass {
    String key1;
    String key2;

    public String getKey1() {
        return this.key1;
    }
    public void setKey1(String key1) {
        this.key1 = key1;
    }
    public String getKey2() {
        return this.key2;
    }
    public void setKey2(String key2) {
        this.key2 = key2;
    }

    public KeyValueClass() {
    }
}
```

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。代码示例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。

入参和返回

代码示例中，mainHandler 所使用的入参使用了 POJO 类型，返回使用了 String 类型。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，其相关库文件可单击 此处 下载。

事件入参及返回参数类型

事件入参	返回参数类型
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)` 。

注意：

部分触发器传递的入参事件结构目前已有一部分已定义，可直接使用。您可通过 cloud event 定义 获取 Java 的库并使用。如果使用过程中发现问题，可以通过 提交 issue 、提交工单 来寻求帮助。

部署方法

最近更新时间：2024-04-22 18:03:28

本文介绍如何 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 50MB），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，完成代码包提交。

使用 Gradle 创建 zip 部署包

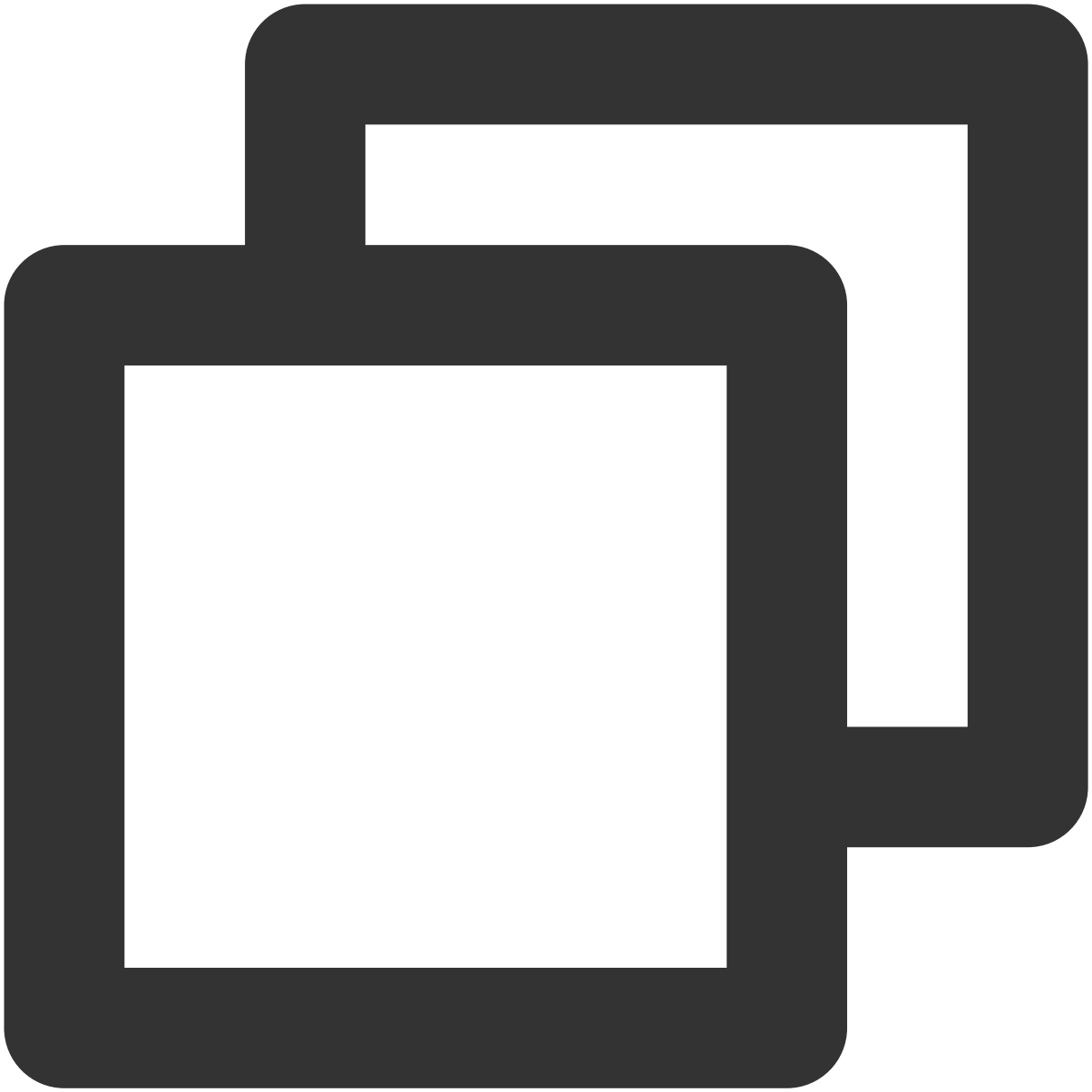
本节内容提供了通过使用 Gradle 工具来创建 Java 类型云函数部署包的方式。创建好的 zip 包符合以下规则，即可以由云函数执行环境所识别和调用。
编译生成的包、类文件、资源文件位于 zip 包的根目录。
依赖所需的 jar 包，位于 /lib 目录内。

环境准备

确保您已经安装 Java 和 Gradle。
Java 11 请安装 TencentKona11或 JDK 11。
Java 8 请安装 JDK 8，您可以使用 OpenJDK（Linux）或通过 Java 下载安装合适您系统的 JDK。

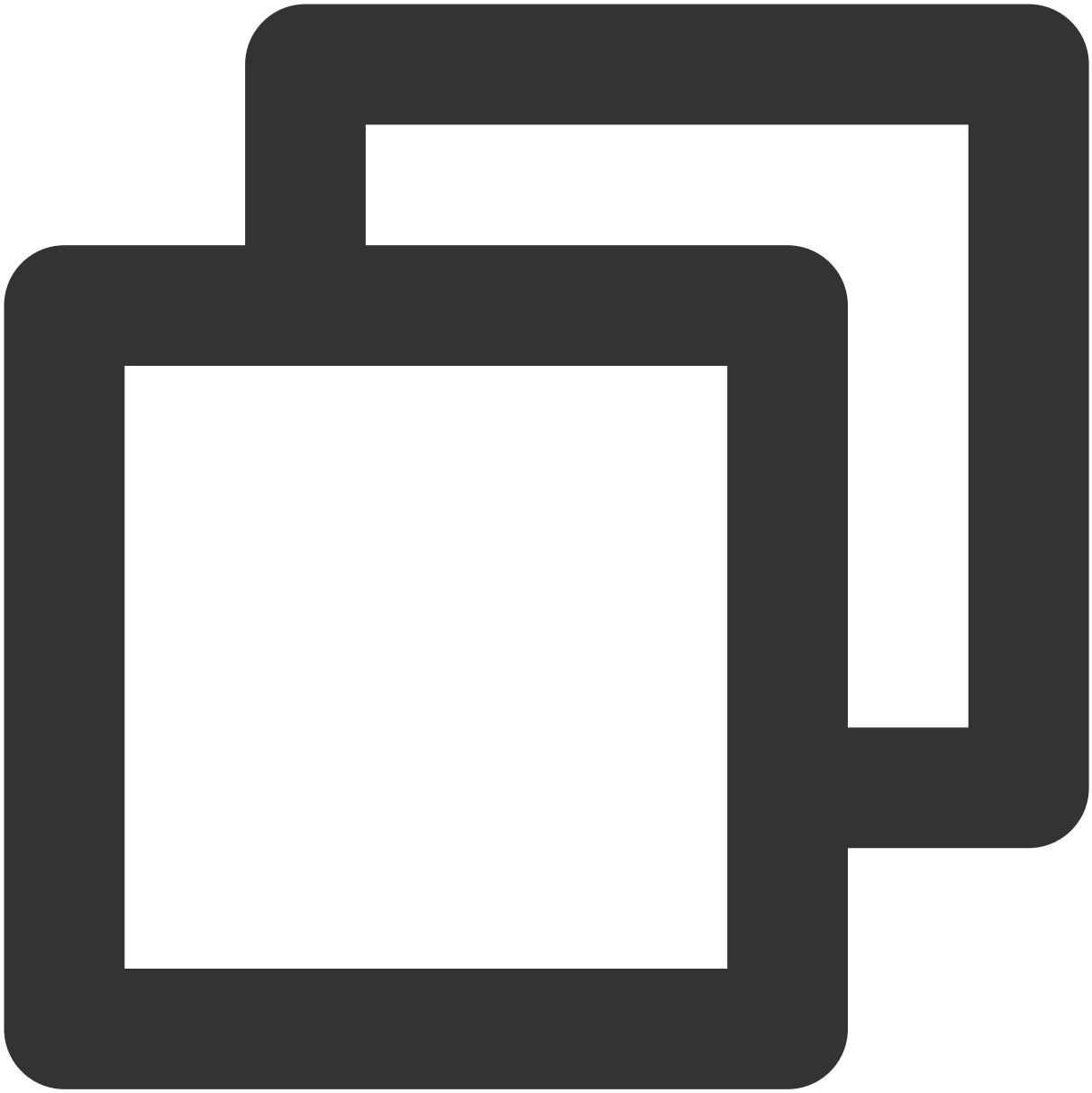
Gradle 安装

Gradle 安装详情见 Gradle Installation。手工安装步骤如下：
1. 单击下载 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，Windows 下的 `C:\\Gradle` 目录或 Linux 下的 `/opt/gradle/gradle-4.1` 目录。
3. 将解压目录下 bin 目录的路径添加到系统 PATH 环境变量中，请对应操作系统执行以下步骤：
Linux：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. 在命令行执行以下命令，查看 Gradle 是否正确安装。



```
gradle -v
```

输出结果如下，则证明 Gradle 已正确安装。如有问题，请查询 Gradle 的 官方文档。

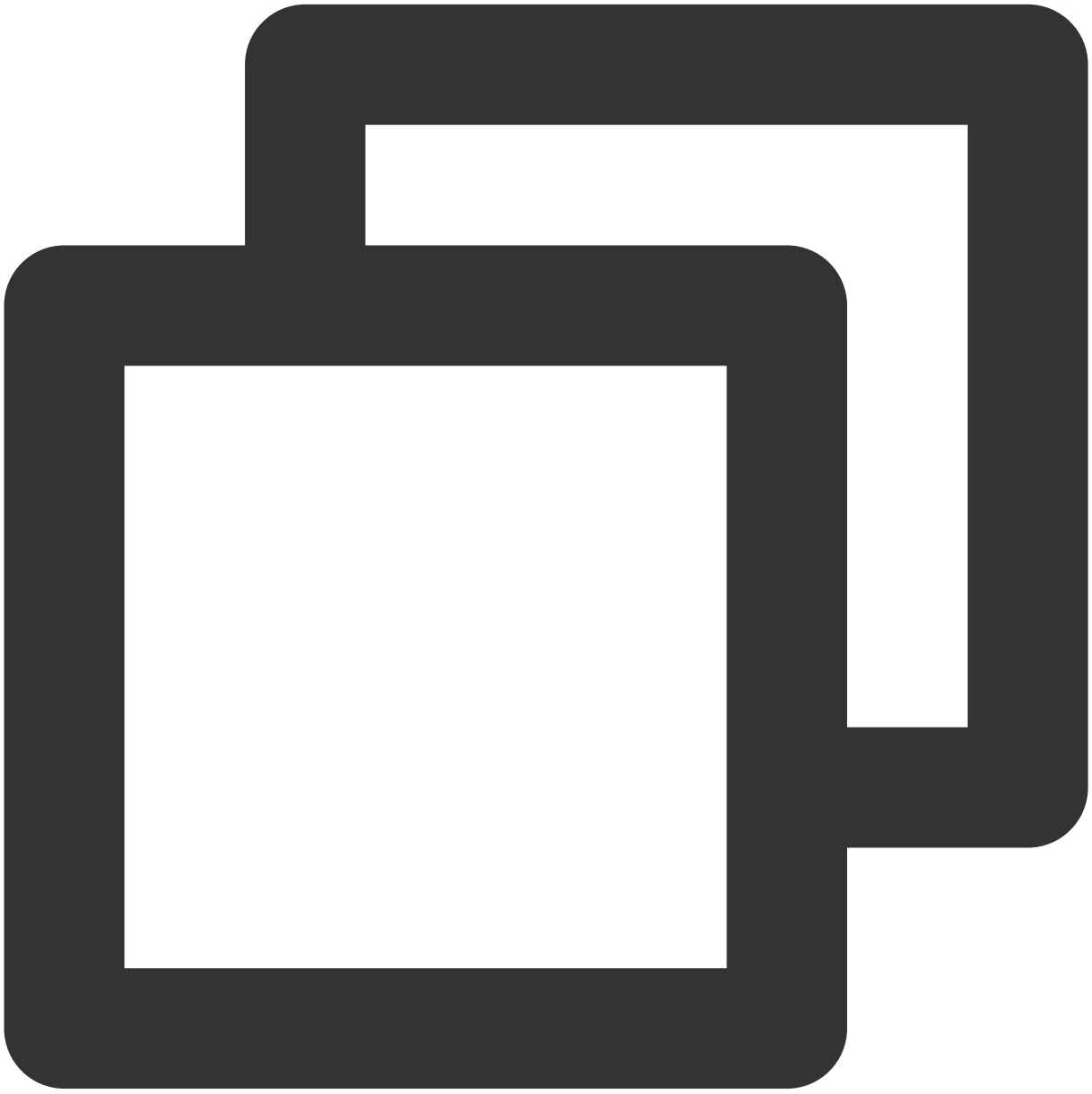


```
------------------------------------------------------------
Gradle 4.1
------------------------------------------------------------
Build time:   2017-08-07 14:38:48 UTC
Revision:     941559e020f6c357ebb08d5c67acdb858a3defc2
Groovy:       2.4.11
Ant:          Apache Ant(TM) version 1.9.6 compiled on June 29 2015
JVM:          1.8.0_144 (Oracle Corporation 25.144-b01)
OS:           Windows 7 6.1 amd64
```

代码准备

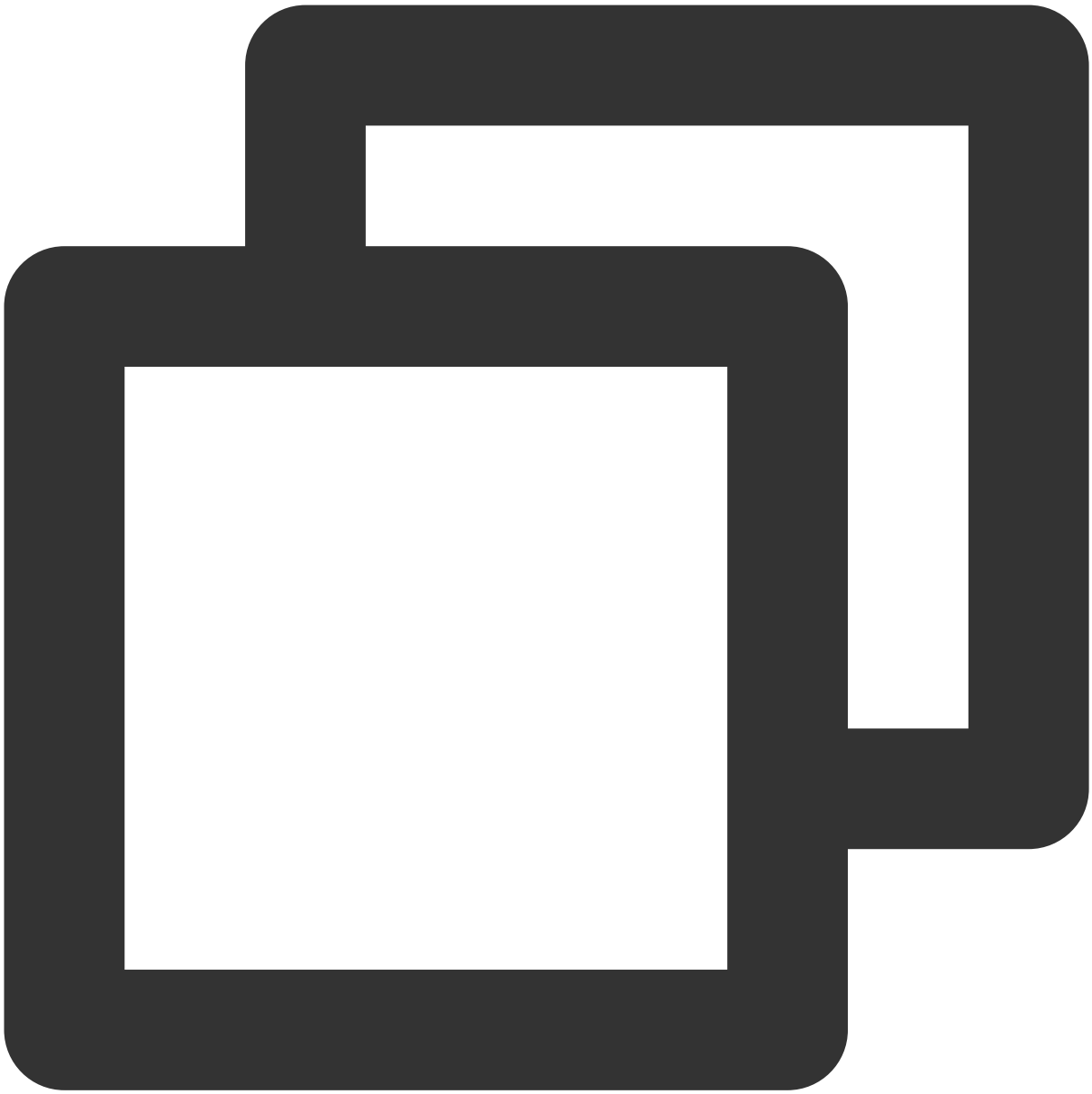
准备代码文件

1. 在选定的位置创建项目文件夹，例如 `scf_example` 。
2. 在项目文件夹根目录，依次创建目录 `src/main/java/` 作为包的存放目录。
3. 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
```

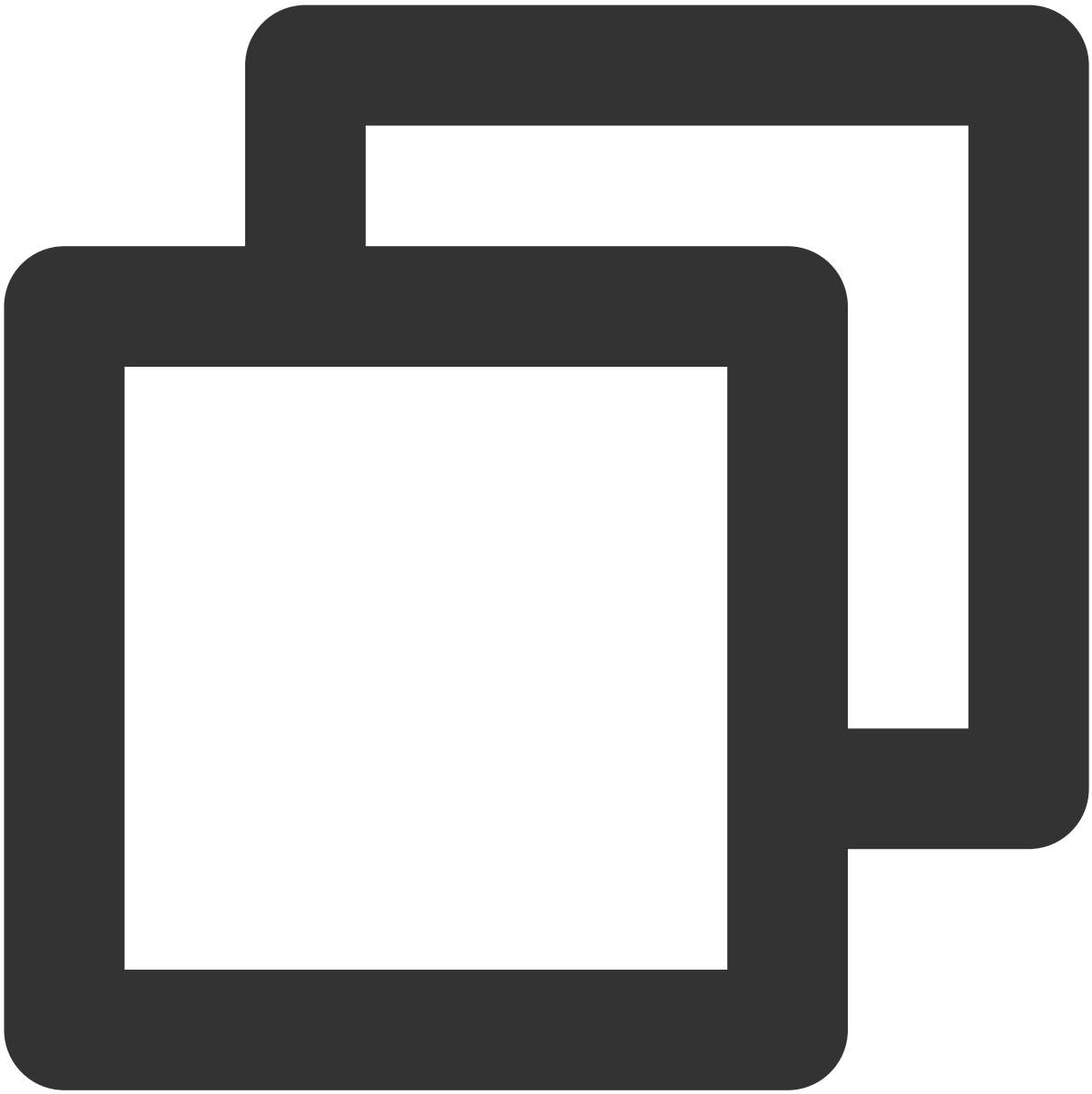
4. 在 `Hello.java` 文件内输入如下代码内容：



```
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：

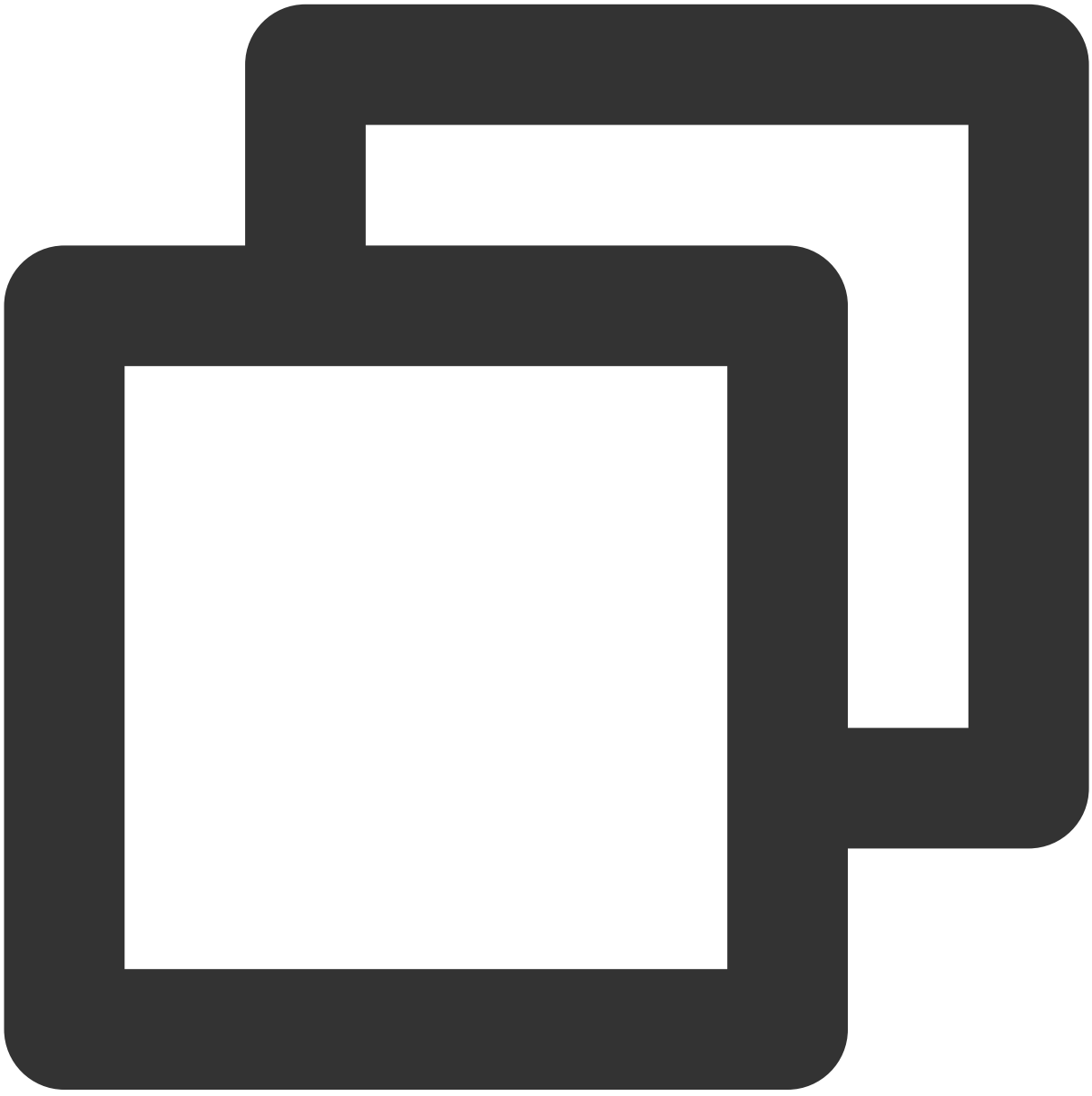


```
apply plugin: 'java'

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，可根据需要添加依赖，`build.gradle` 文件内容如下：



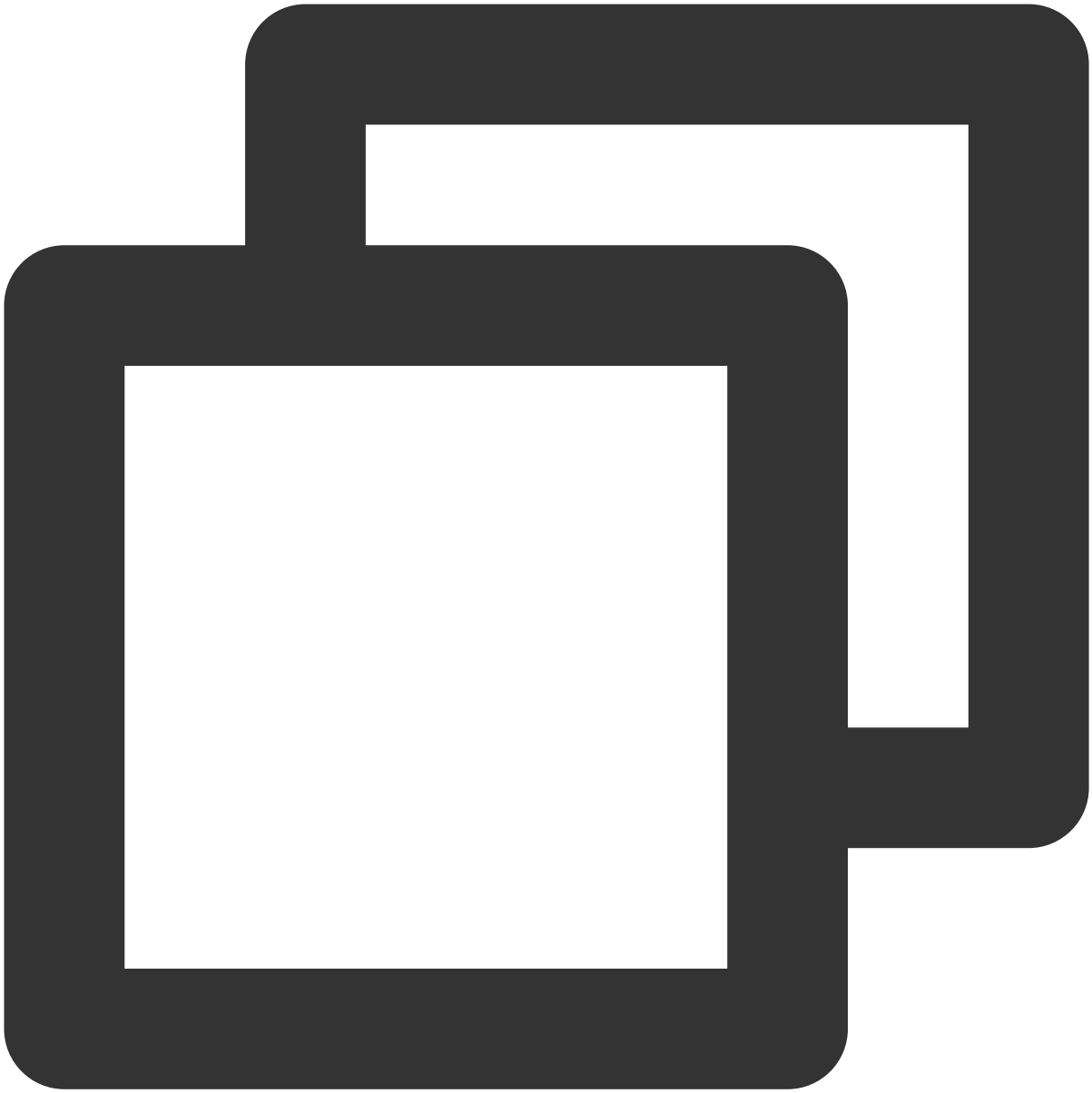
```
apply plugin: 'java'

repositories {
    mavenCentral()
```

```
}

dependencies {
    compile (
        'com.tencentcloudapi:scf-java-events:0.0.2'
    )
}

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.tencentcloudapi:scf-java-events:0.0.2` 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：



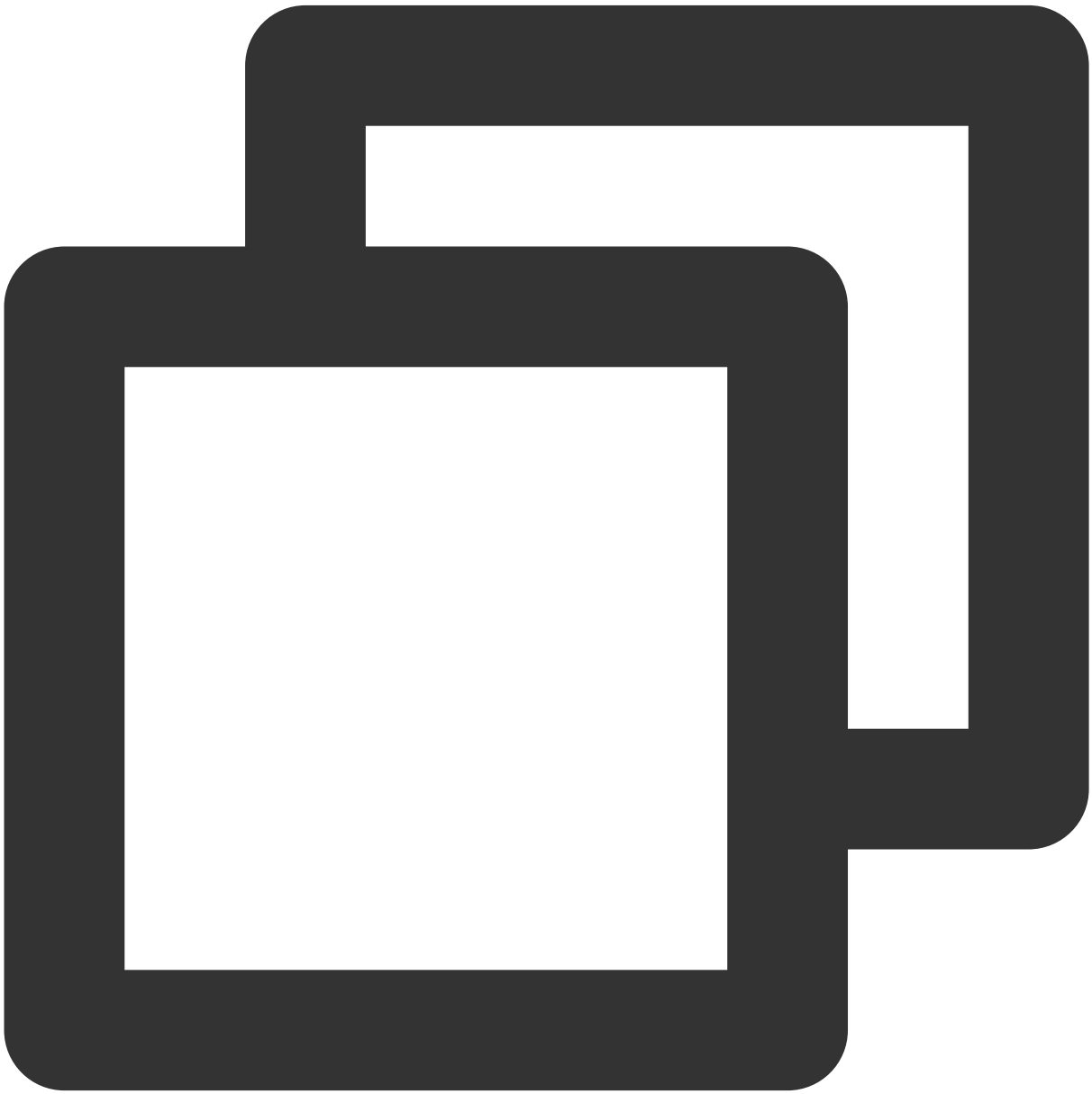
```
apply plugin: 'java'

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` 。

使用 Maven 创建 jar 部署包

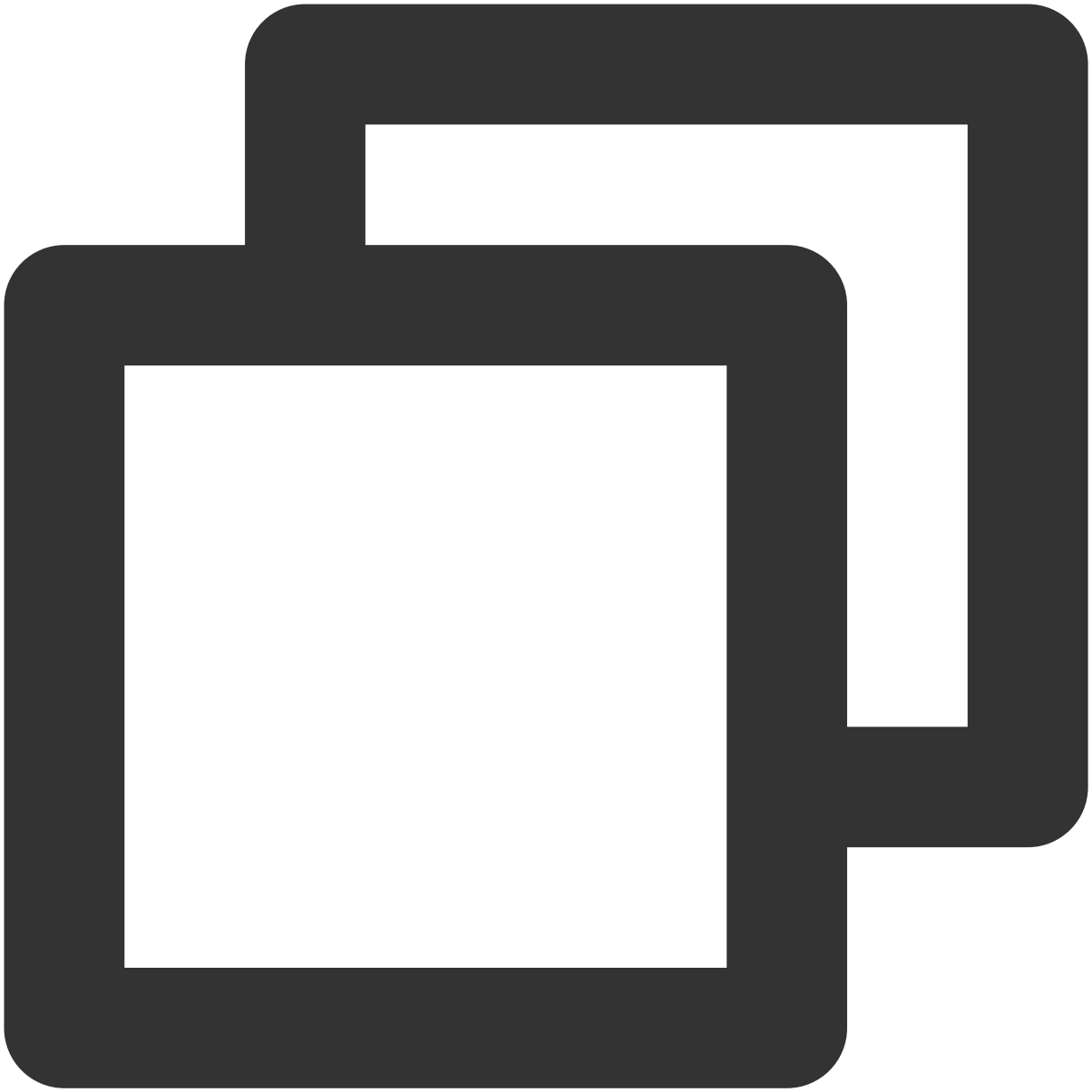
本节内容提供了通过使用 Maven 工具来创建 Java 类型云函数部署 jar 包的方式。

环境准备

确保您已经安装 Java 和 Gradle。
Java 11 请安装 TencentKona11或 JDK 11。
Java 8 请安装 JDK 8，您可以使用 OpenJDK（Linux）或通过 Java 下载安装合适您系统的 JDK。

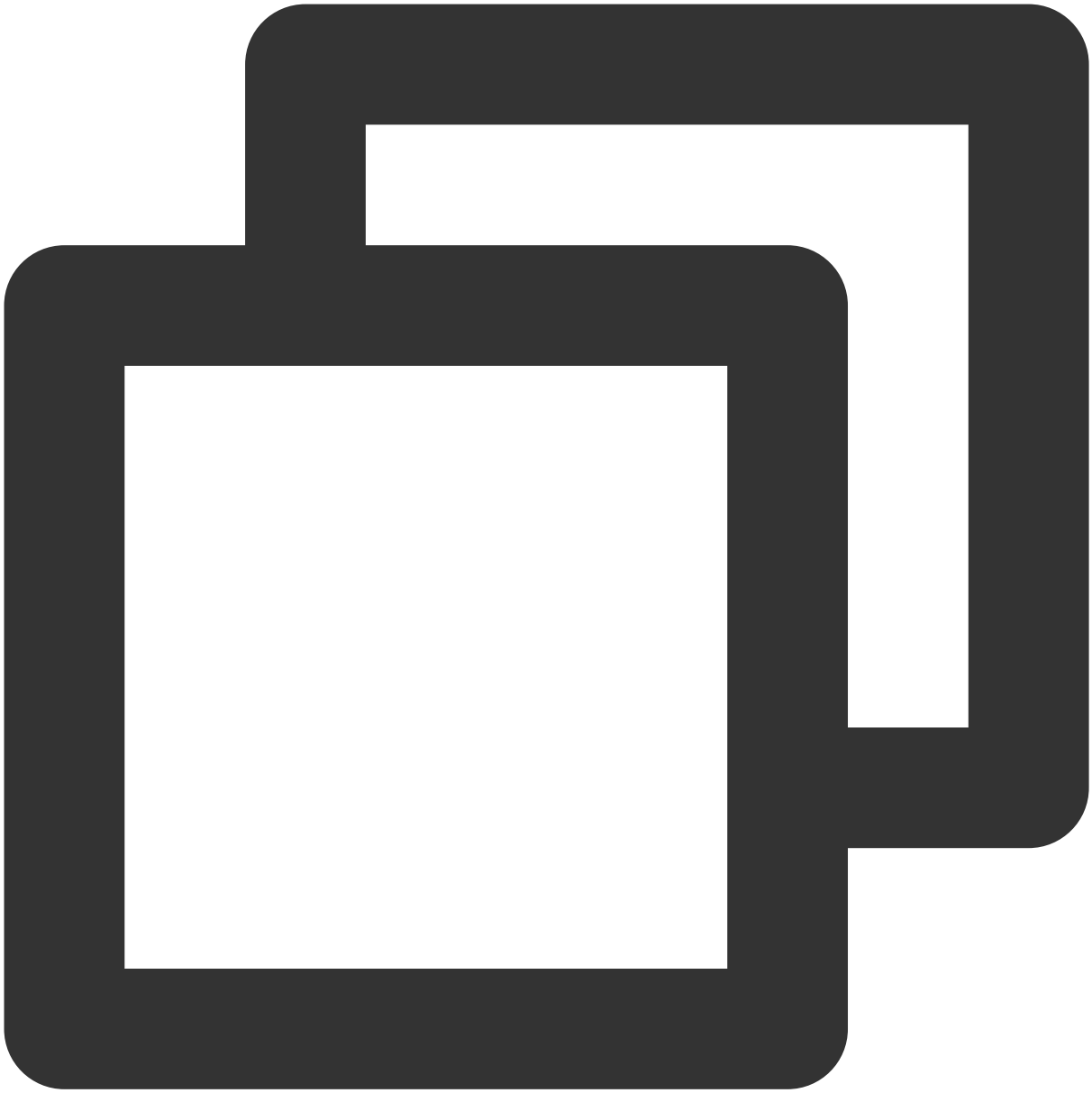
Maven 安装

Maven 安装详情见 Installing Apache Maven。手工安装步骤如下：
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，Windows 下的 `C:\\Maven` 目录或 Linux 下的 `/opt/mvn/apache-maven-3.5.0` 目录。
3. 将解压目录下 bin 目录的路径添加到系统 PATH 环境变量中，请对应操作系统执行以下步骤：
Linux：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. 在命令行下执行以下命令，查看 Maven 是否正确安装。



```
mvn -v
```

输出结果如下，则证明 Maven 已正确安装。如有问题，请查询 Maven 的 官方文档。

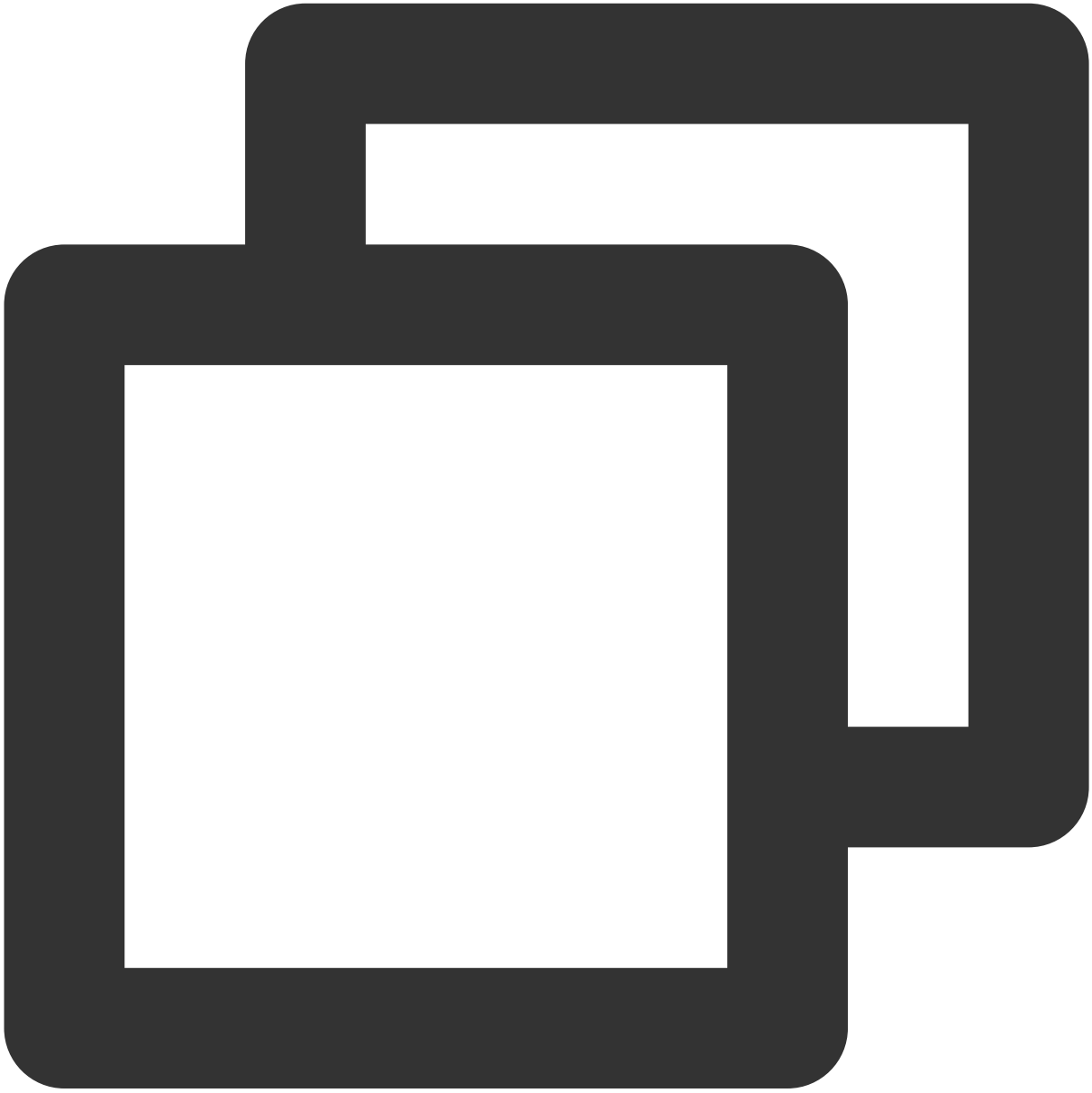


```
Apache Maven 3.5.0 (ff8f5e7444045639af65f6095c62210b5713f426; 2017-04-04T03:39:06+0
Maven home: C:\\Program Files\\Java\\apache-maven-3.5.0\\bin\\..
Java version: 1.8.0_144, vendor: Oracle Corporation
Java home: C:\\Program Files\\Java\\jdk1.8.0_144\\jre
Default locale: zh_CN, platform encoding: GBK
OS name: "windows 7", version: "6.1", arch: "amd64", family: "windows"
```

代码准备

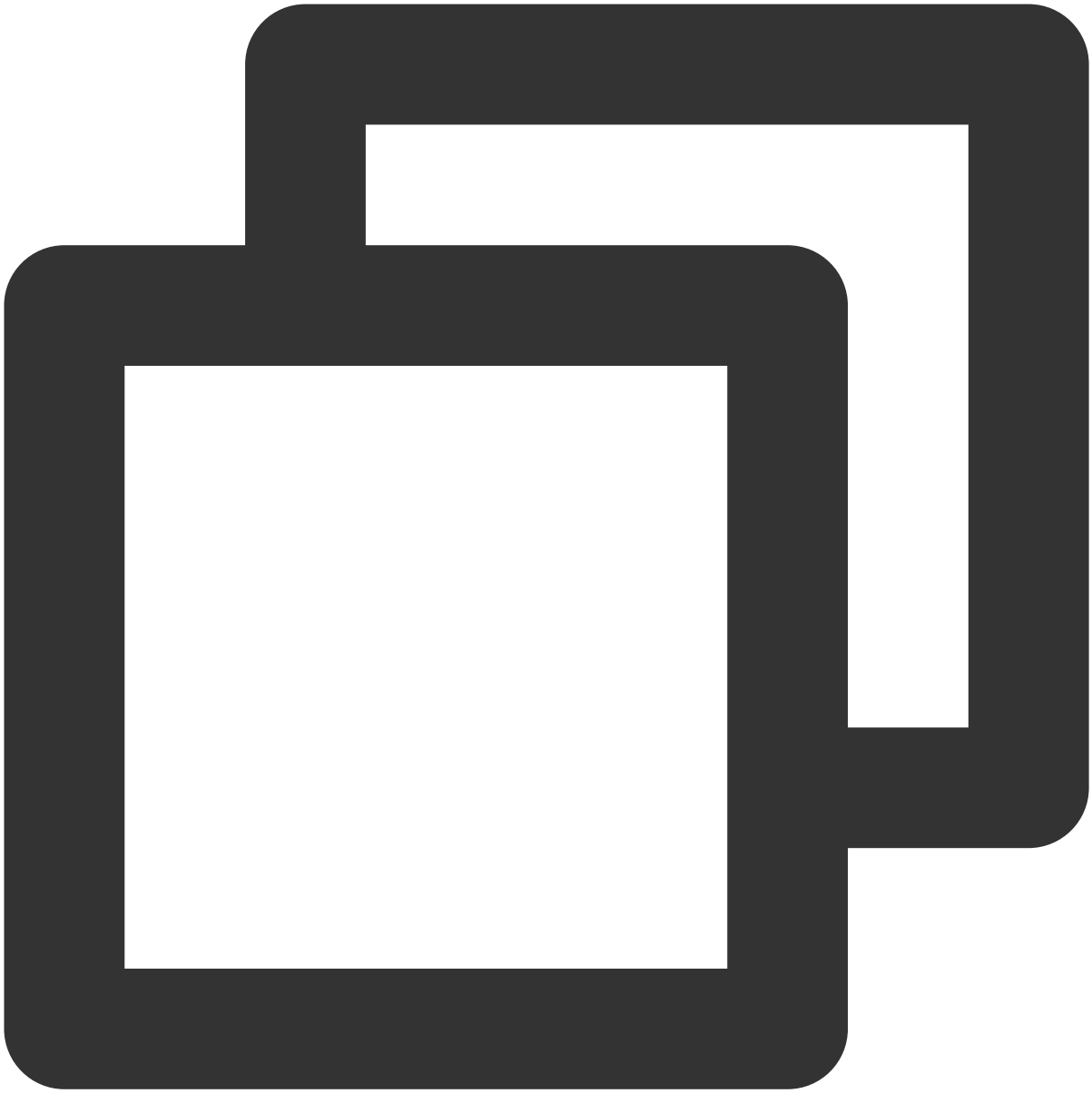
准备代码文件

1. 在选定的位置创建项目文件夹，例如 `scf_example` 。
2. 在项目文件夹根目录，依次创建目录 `src/main/java/` 作为包的存放目录。
3. 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
```

4. 在 `Hello.java` 文件内输入如下代码内容：



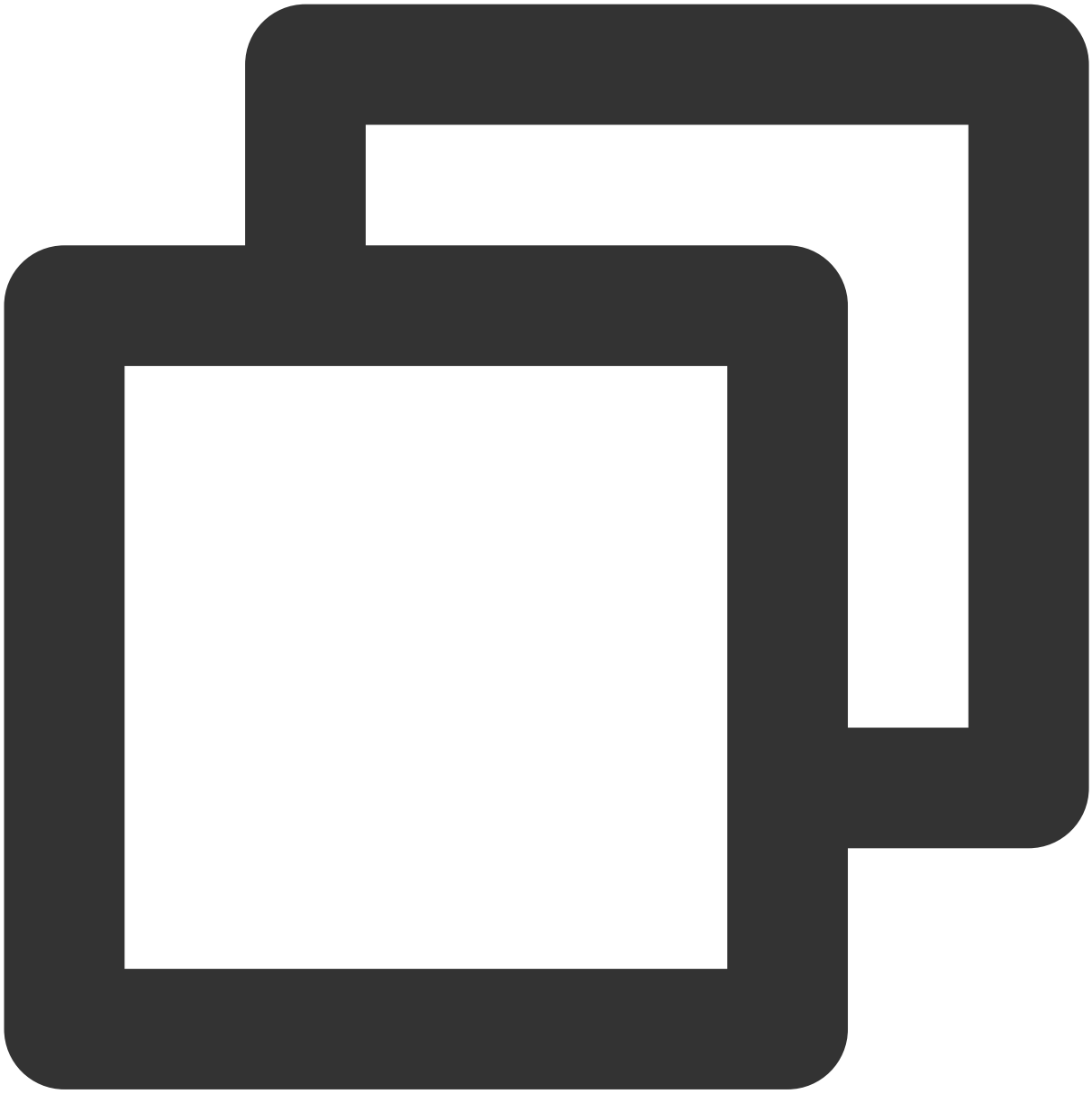
```
package example;
public class Hello {
 public String mainHandler(String name, Context context) {
     System.out.println("Hello world!");
     return String.format("Hello %s.", name);
 }
}
```

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：

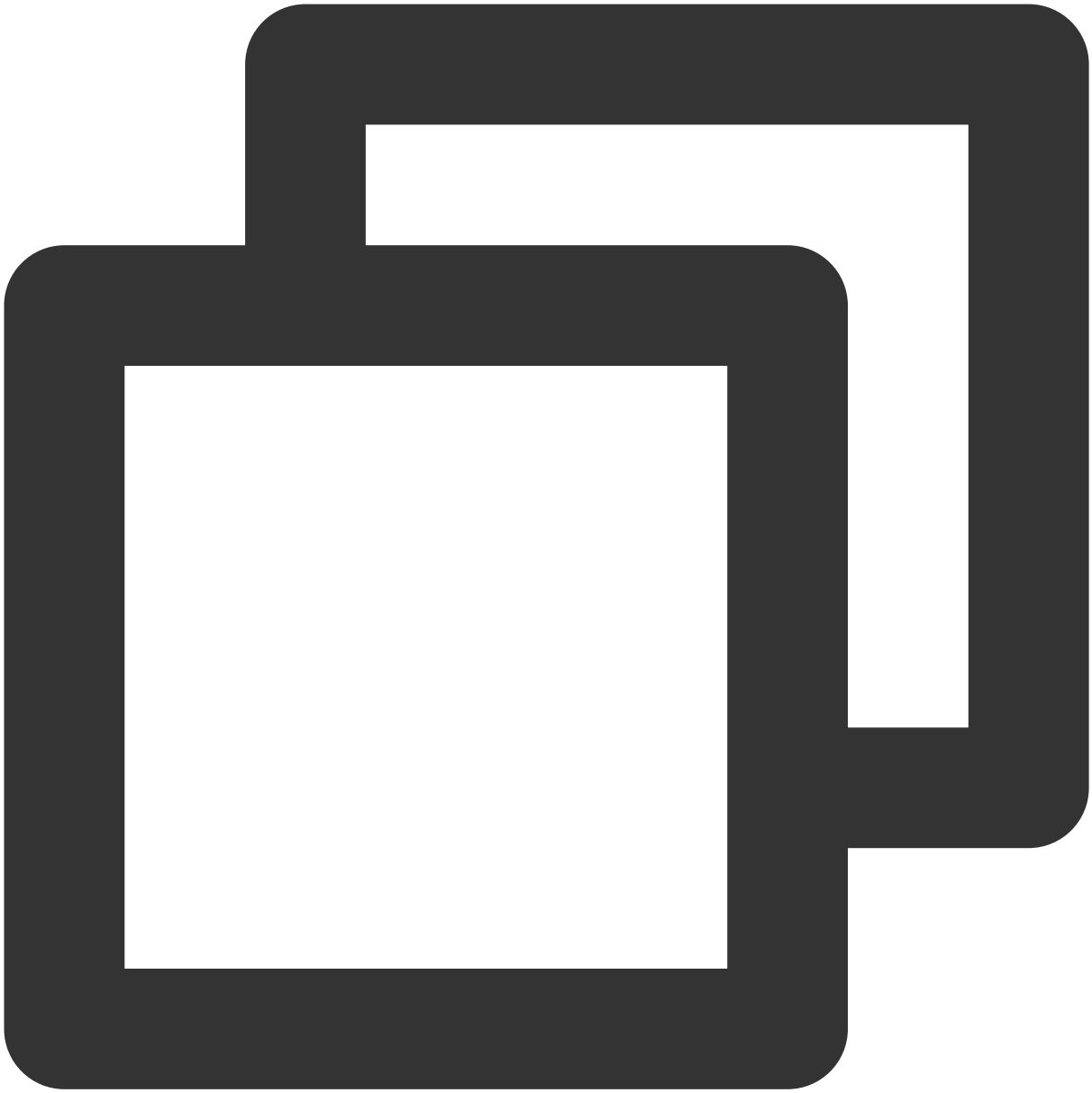
Java 11

Java 8



```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/200
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/m
  <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>
            <source>11</source>
            <target>11</target>
          </configuration>
          <executions>
            <execution>
              <phase>package</phase>
              <goals>
                <goal>shade</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
</project>
```

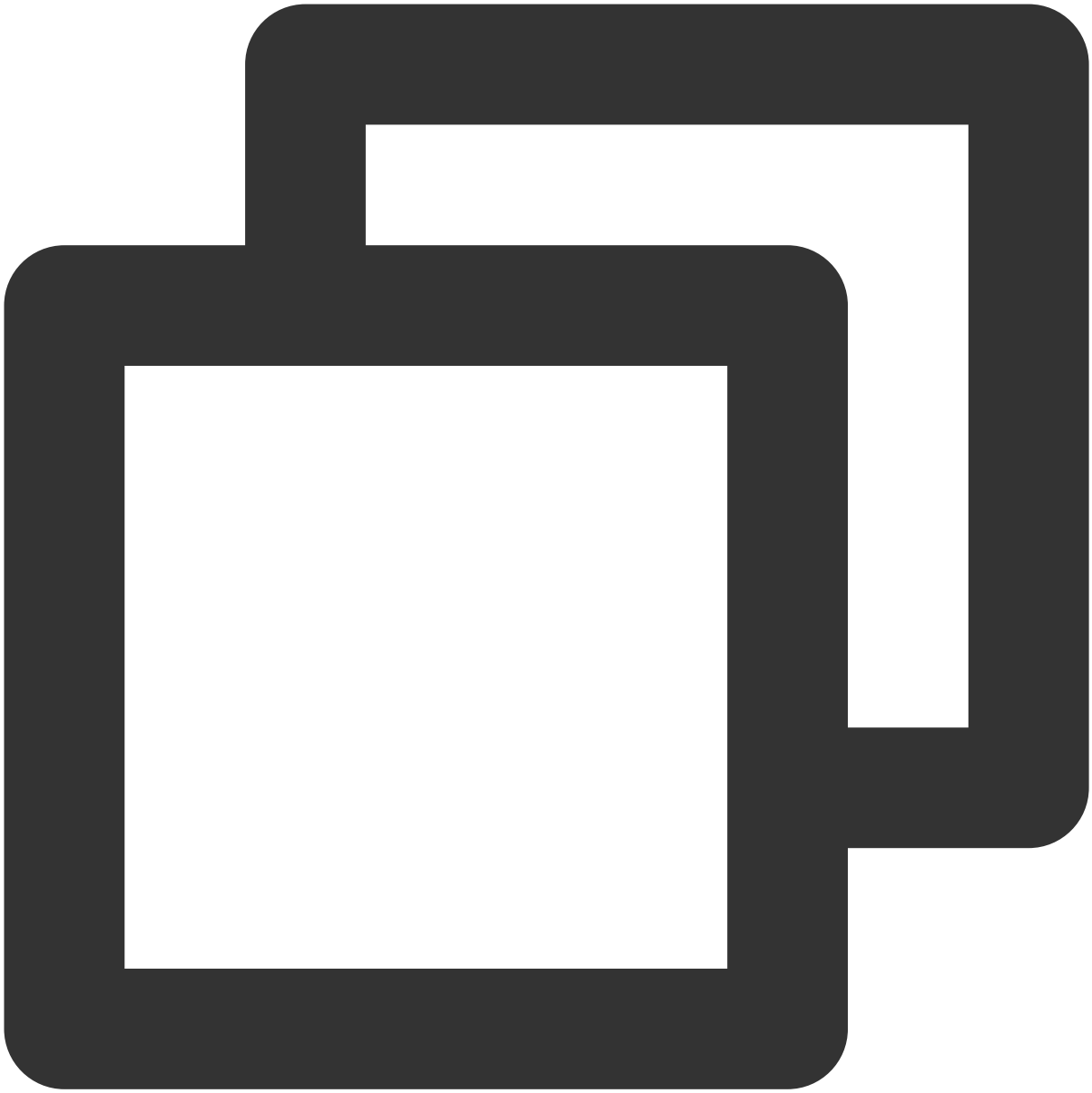


```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/200
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/m
  <modelVersion>4.0.0</modelVersion>

  <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` 文件内容如下，添加依赖请关注 `<dependencies>` 部分。

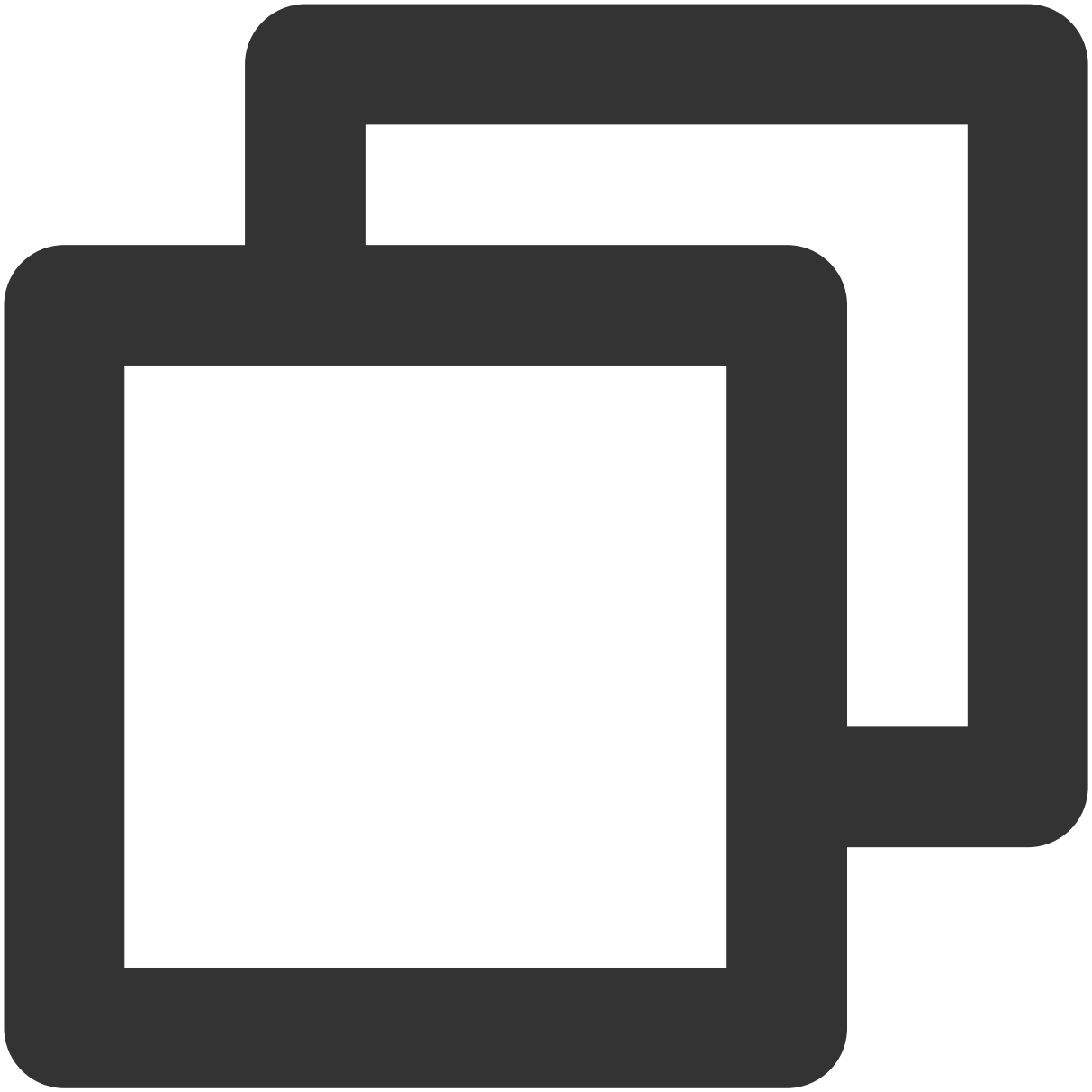


```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/200
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/m
  <modelVersion>4.0.0</modelVersion>

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

  <dependencies>
```

```
    <dependency>
      <groupId>com.tencentcloudapi</groupId>
      <artifactId>scf-java-events</artifactId>
      <version>0.0.2</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。

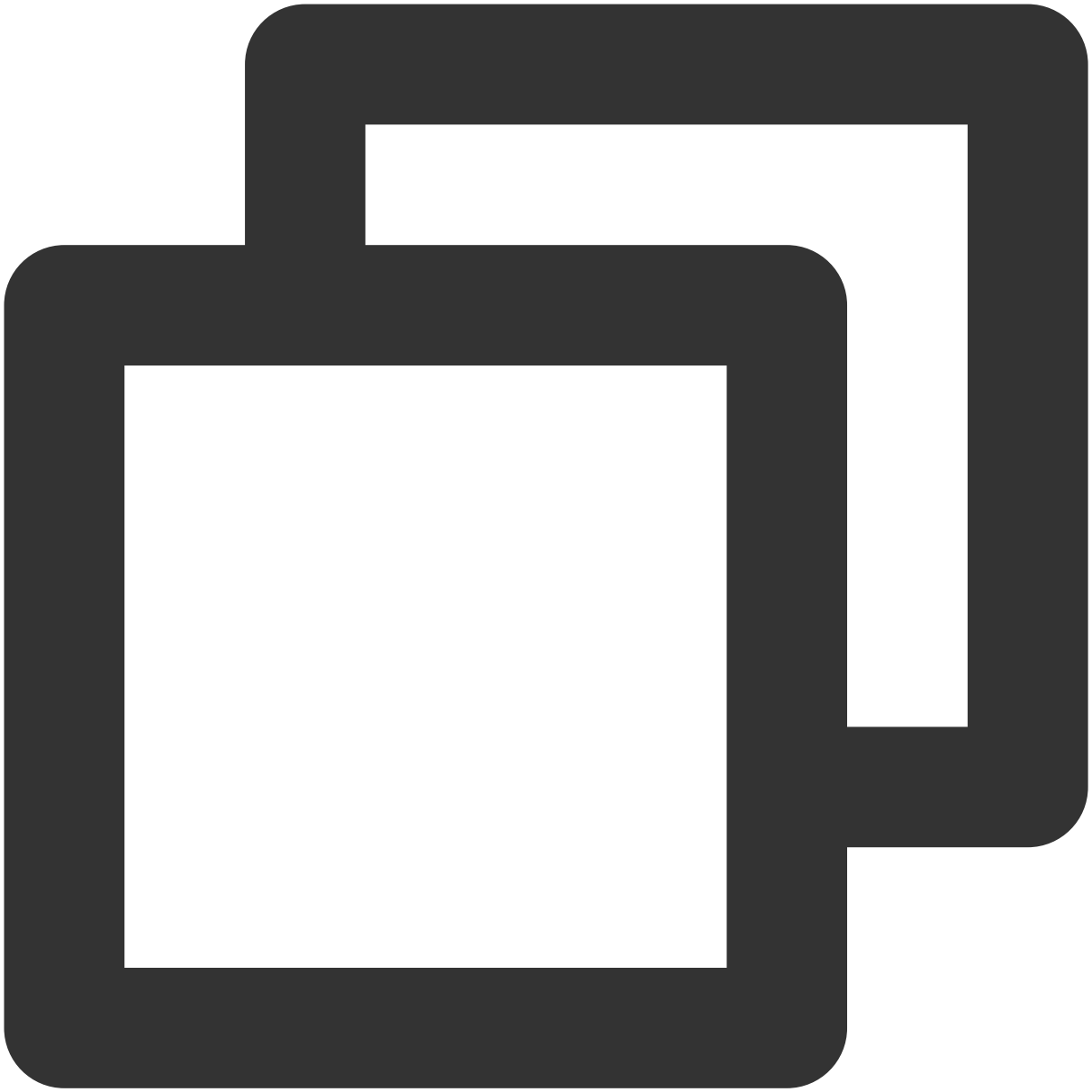
编译后的 jar 包位于项目文件夹内的 `target` 目录内，并根据 pom.xml 内的 artifactId、version 字段命名为 `java-example-1.0-SNAPSHOT.jar` 。

日志说明

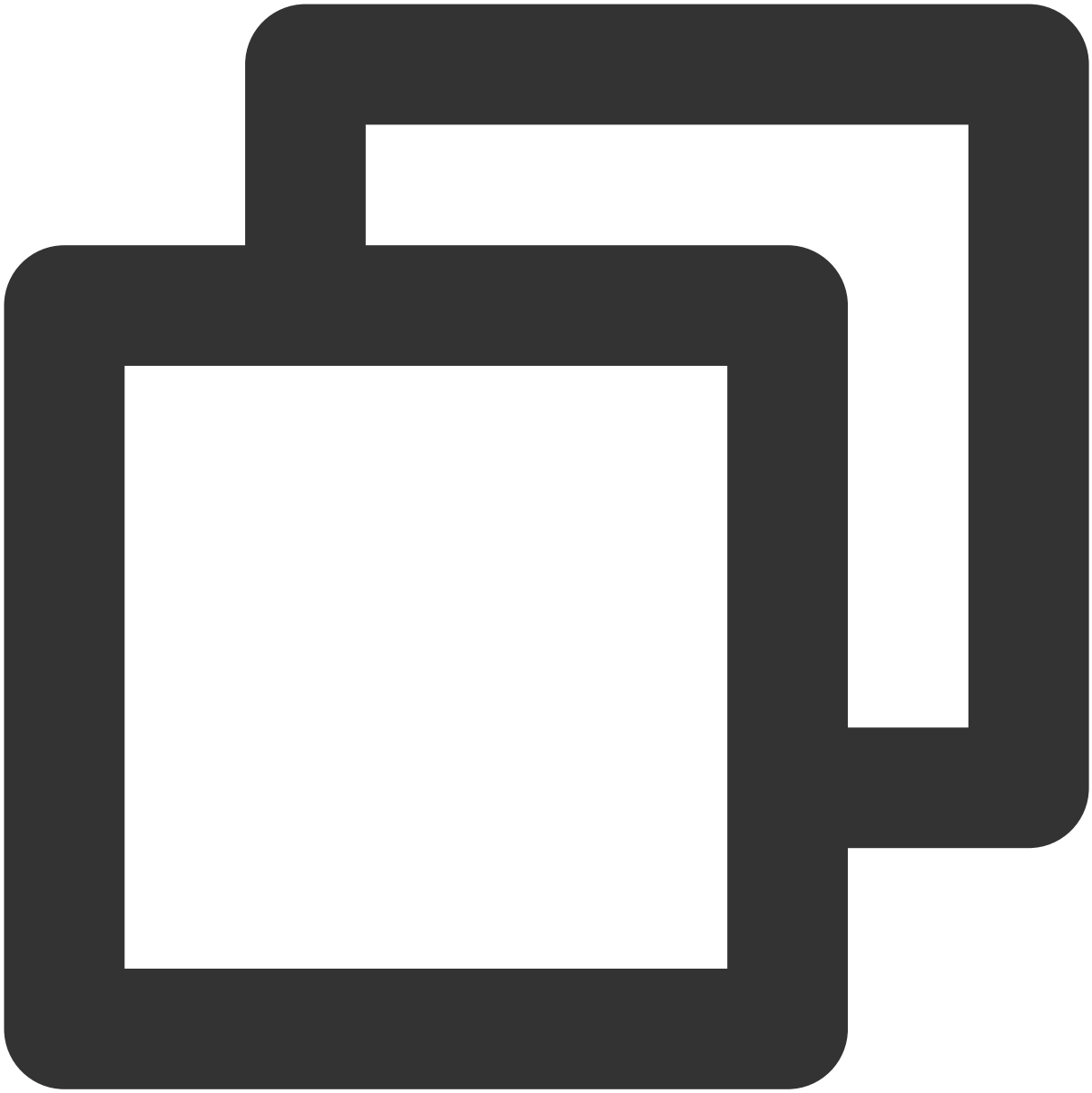
最近更新时间：2024-04-22 18:03:28

日志开发

您可以在程序中使用 `System.out.println()` 和 `java.util.logging.Logger()` 语句来完成日志输出。
例如，执行以下代码，可以在函数日志中查询输出内容。



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



```
Logger logger = Logger.getLogger("AnyLoggerName");
logger.setLevel(Level.INFO);
logger.info("logging message here!");
```

日志查询

当前函数日志均会投递至腾讯云日志服务 CLS 中，您可对函数日志进行投递配置，详情可参见 日志投递配置。
您可通过云函数的日志查询界面或通过日志服务的查询界面，查询函数执行日志。日志查询方法详情可参见 日志检

索教程。

说明：

函数日志投递到日志服务日志集 LogSet 和日志主题 LogTopic，均可以通过函数配置查询。

自定义日志字段

当前在函数代码中使用简单日志打印语句，将会在投递到日志服务时，记录在 `SCF_Message` 字段中。日志服务的字段说明可参见 索引说明。

目前云函数已经支持在输出到日志服务的内容中增加自定义字段，通过增加自定义字段，您可以将业务字段及相关数据内容输出到日志中，并通过使用日志服务的检索能力，对执行过程中的业务数据及相关内容进行查询跟踪。

注意：

如需对自定义字段进行键值查询，如 `SCF_CustomKey：SCF` ，请参考 日志服务索引配置 为函数日志投递的日志主题添加键值索引。

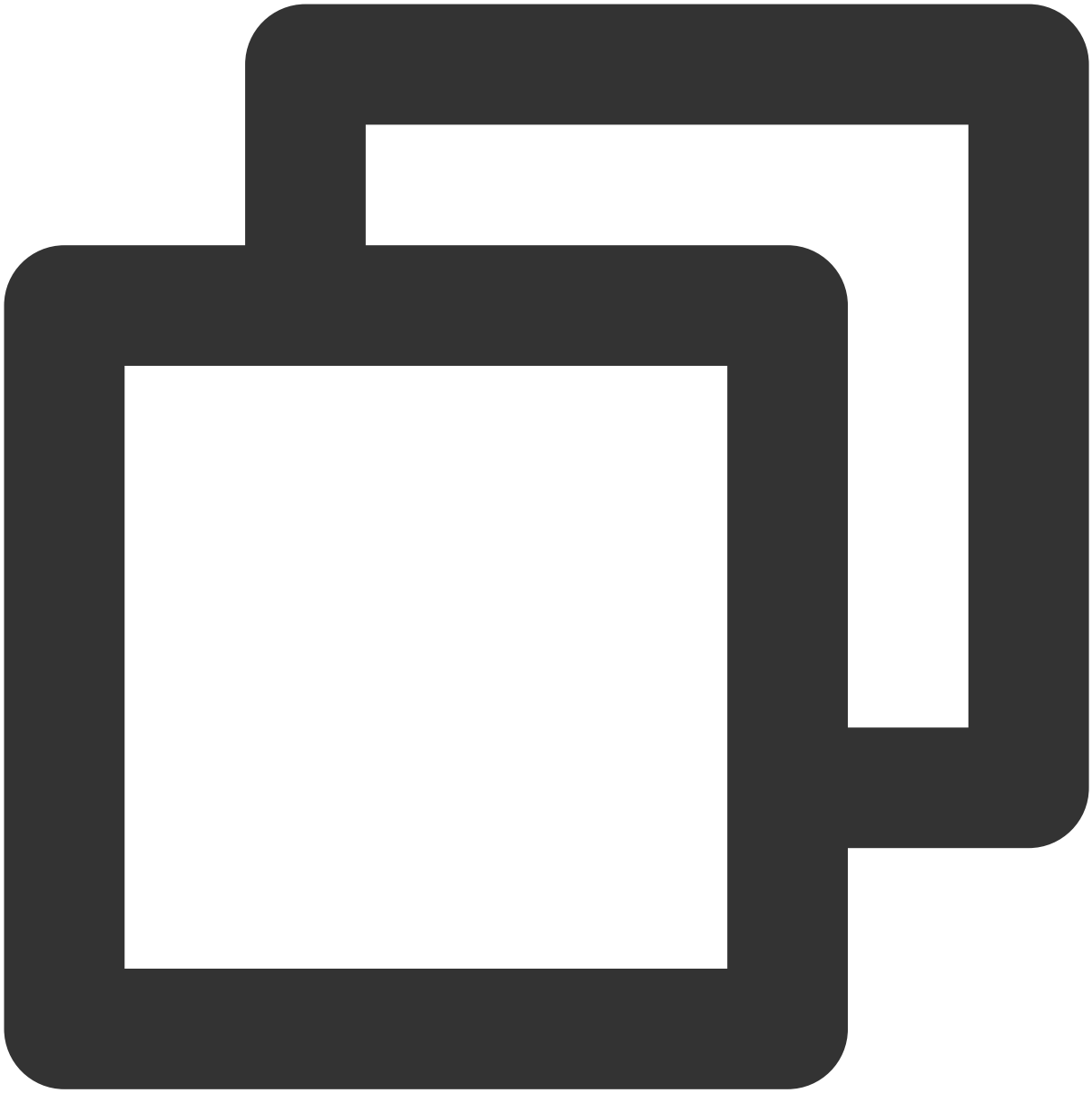
为避免误操作索引配置导致函数日志查询失败，函数配置的默认投递日志主题（以 `SCF_LogTopic_` 为前缀命名）不支持修改索引配置。请将函数日志投递主题设置为 自定义投递 后再更新日志主题索引配置。

日志主题修改索引配置后，仅对新写入的数据有效。

输出方法

当函数输出的单行日志为 JSON 格式时，JSON 内容将被解析并在投递至日志服务时按 `字段:值` 的方式进行投递。JSON 内容的解析仅能解析第一层，更多的嵌套结构将作为值进行记录。

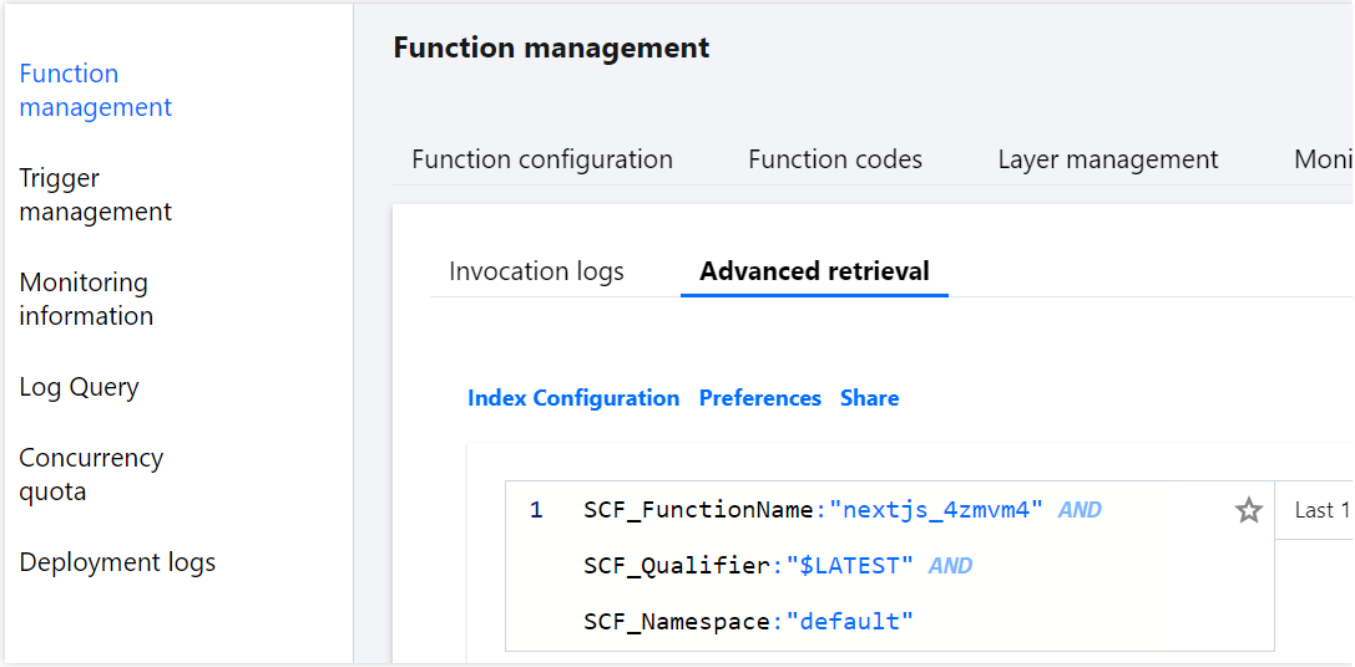
您可执行以下代码进行测试：



```
package example;

public class Hello {
    public String mainHandler(KeyValueClass kv) {
        System.out.println("{\\"key1\\": \\"test value 1\\",\\"key2\\": \\"test val
      return String.format("hello world");
    }
}
```

检索方法

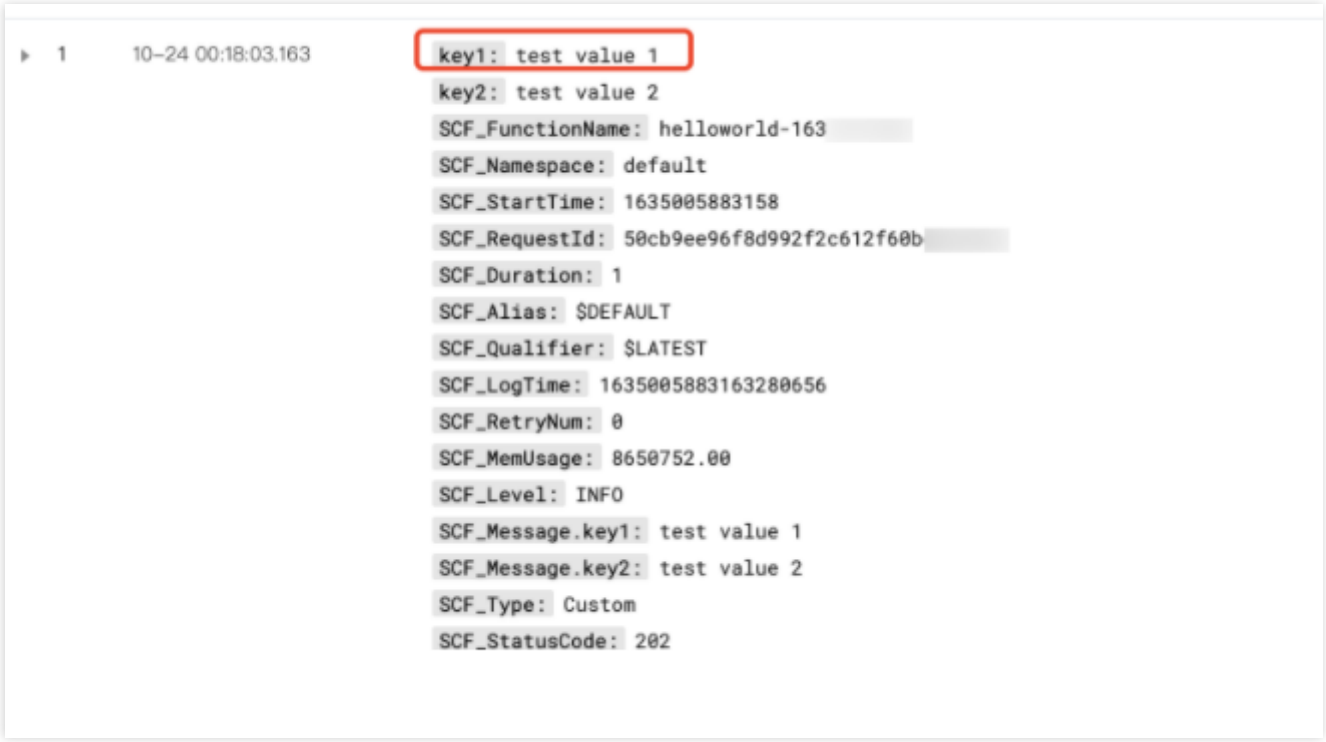
在使用上述代码进行测试运行后，您可在**函数服务 > 日志查询 > 高级检索**中通过如下语句进行检索：



操作详情可参见 控制台检索。

检索结果

在测试写入日志服务后，您可以在日志查询中检索到 `key1` 字段。如下图所示：



常见示例

最近更新时间：2024-04-22 18:03:28

操作场景

通过使用 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。在本文档中，将使用一组示例说明在 SCF 中如何使用 POJO 参数，并能支持何种格式的入参。

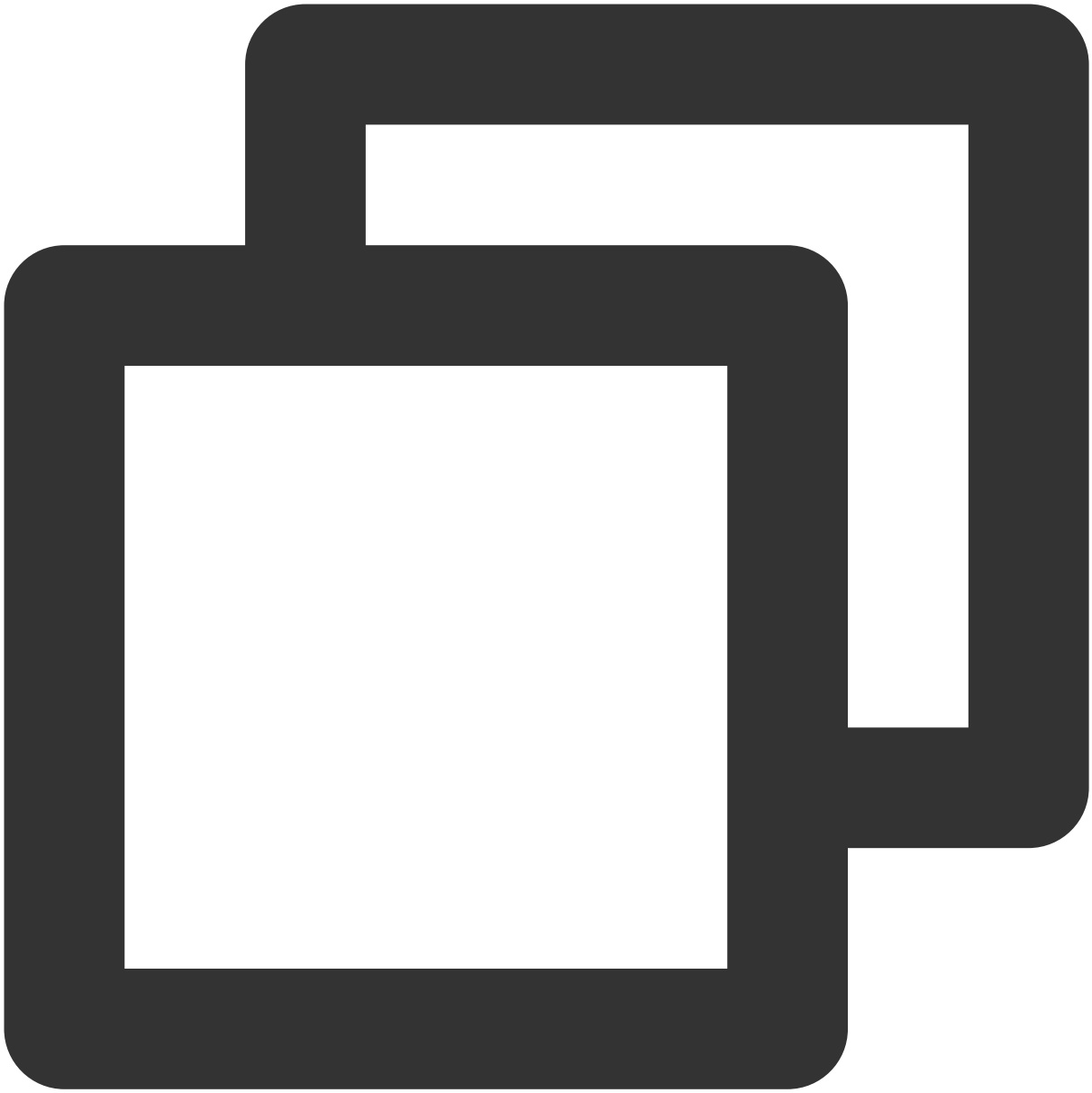
前提条件

已登录 Serverless 控制台。

操作步骤

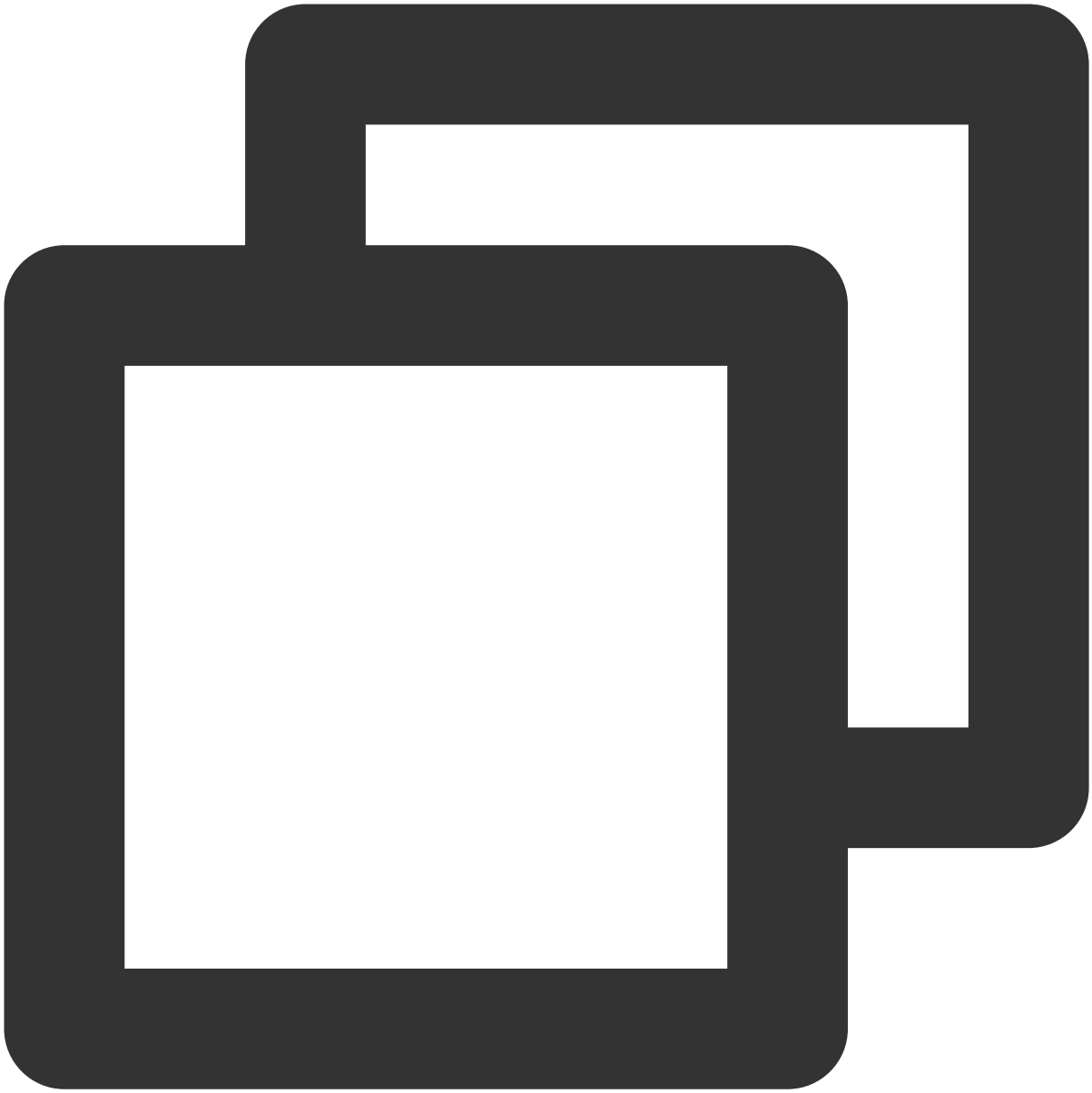
事件入参和 POJO

本文使用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，输出内容如下：



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

根据入参，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

根据入参已构建的四个类及入口函数，请按照以下步骤进行代码准备。

步骤1：项目目录准备

创建项目根目录，例如 `scf_example` 。

步骤2：代码目录准备

1. 在项目根目录下创建文件夹 `src\\main\\java` 作为代码目录。
2. 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。

步骤3：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：

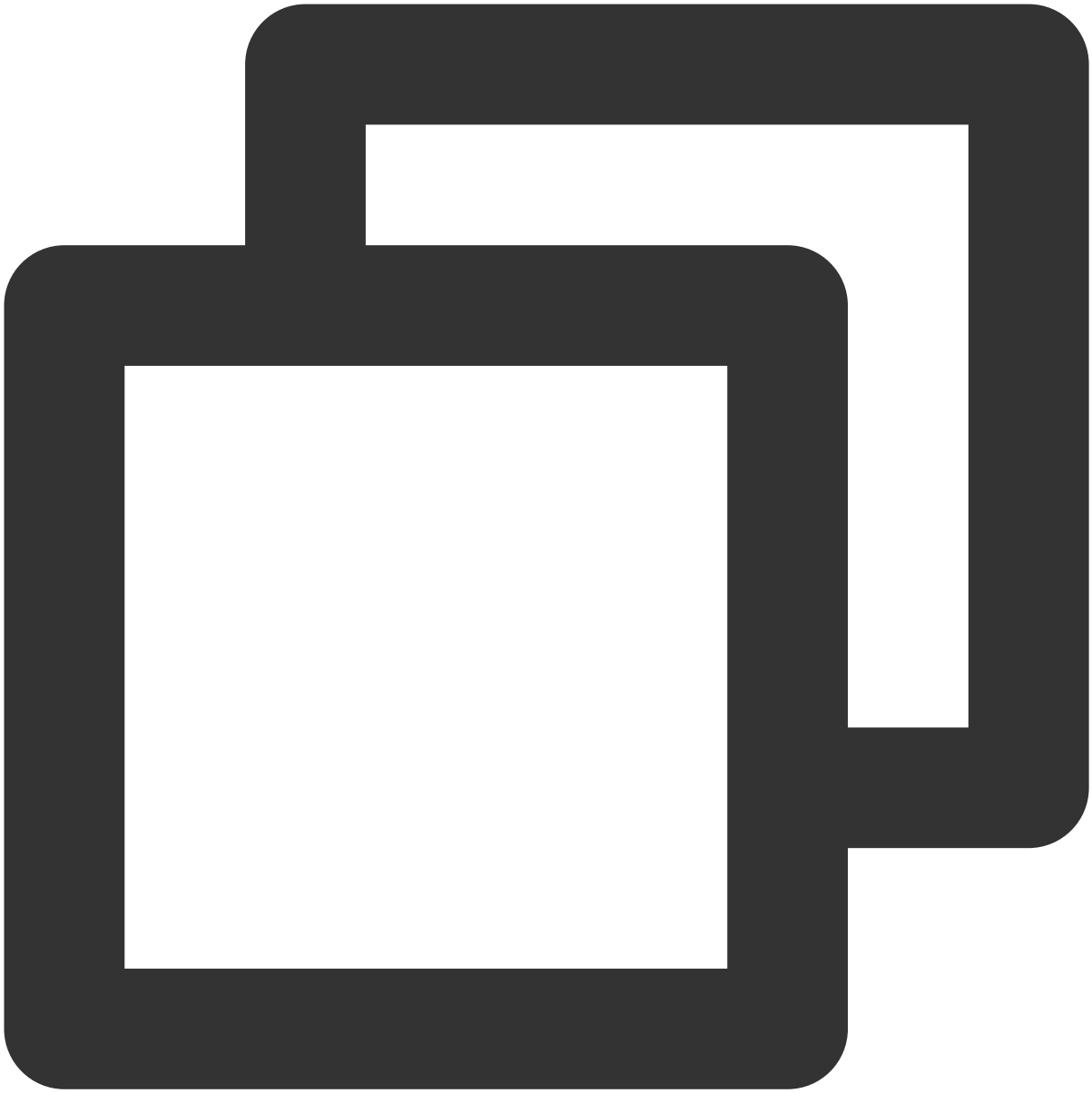
Pojo.java

RequestClass.java

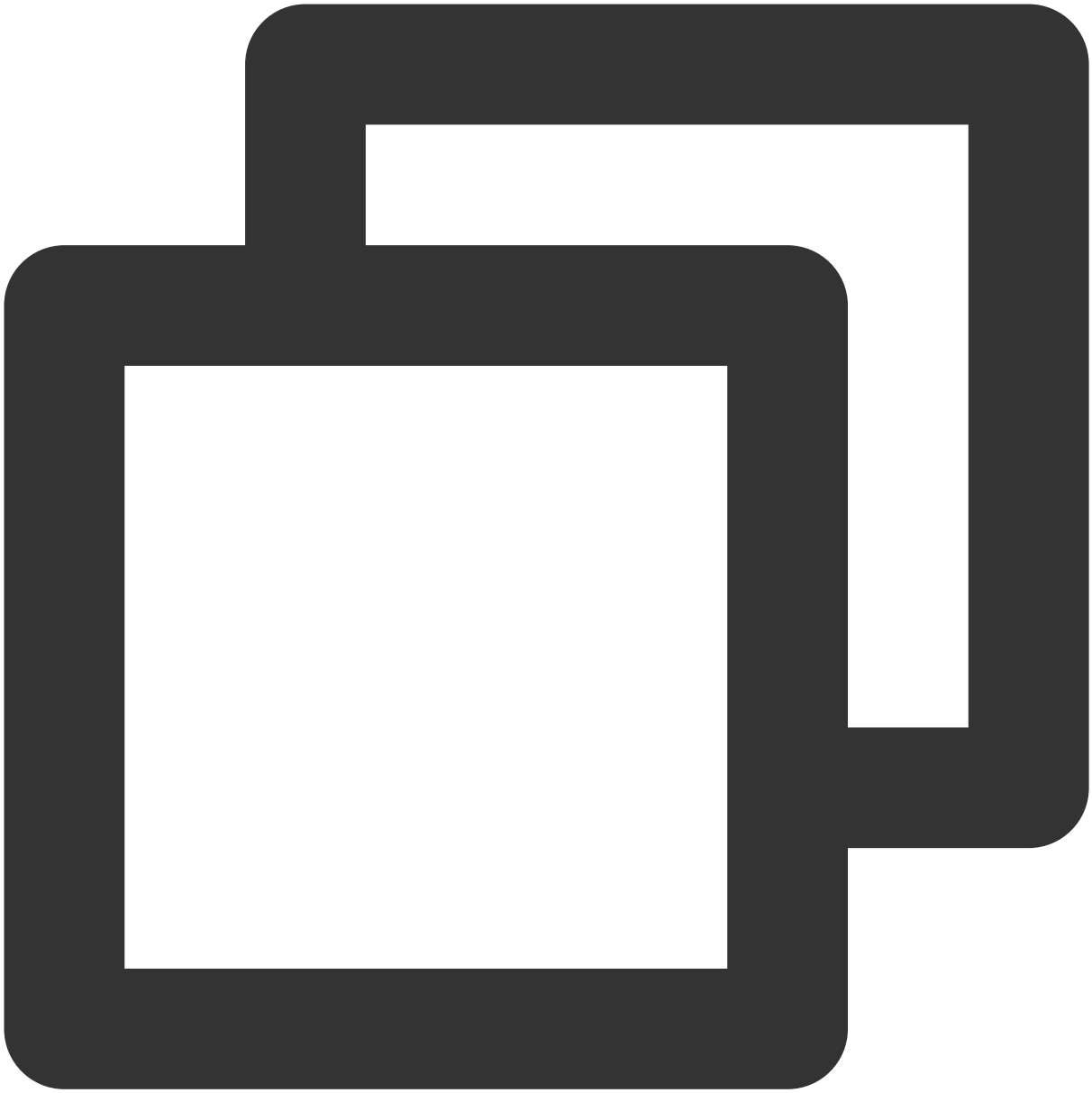
PersonClass.java

CityClass.java

ResponseClass.java

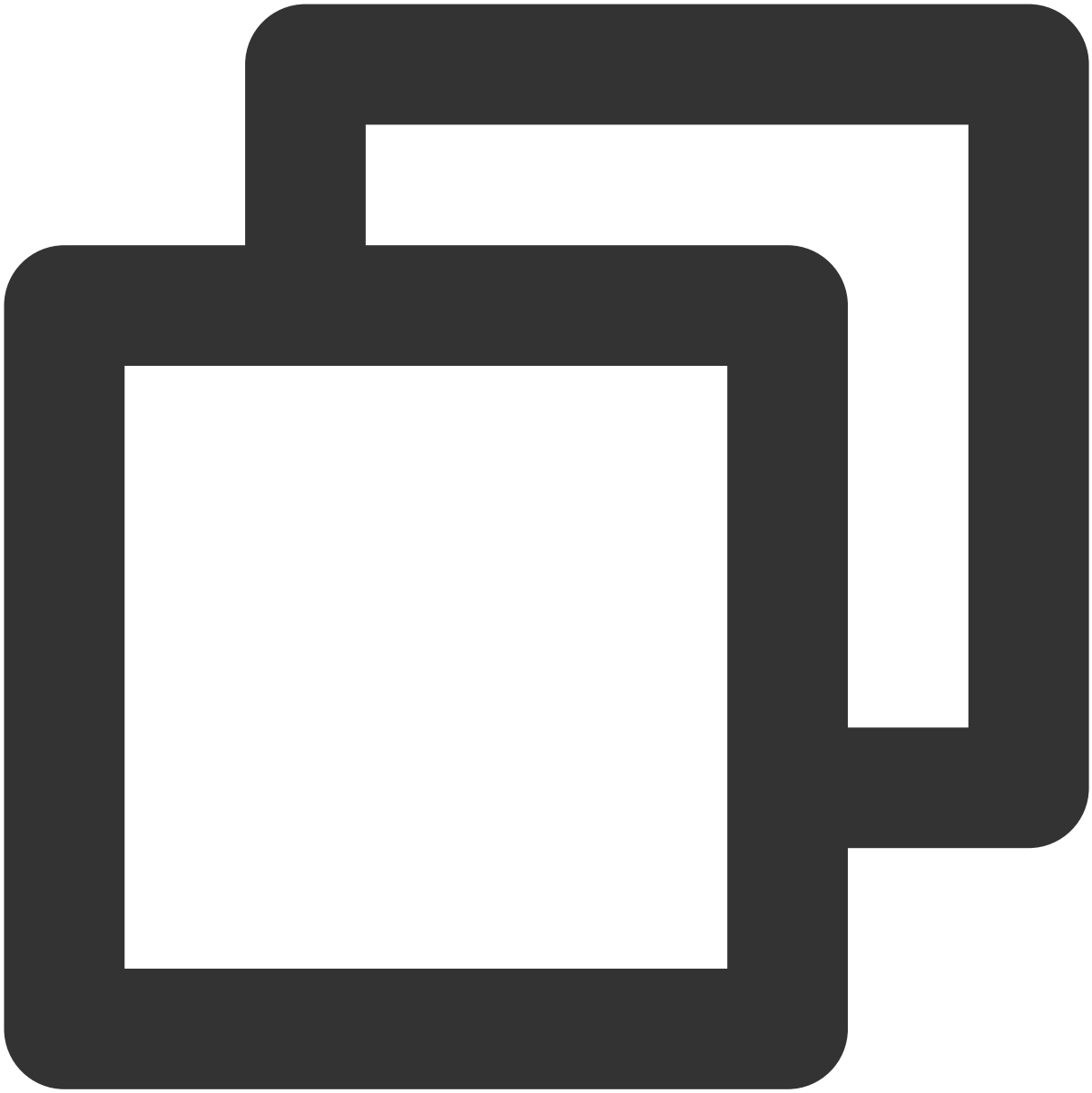


```
package example;

public class Pojo{
    public ResponseClass handle(RequestClass request){
        String greetingString = String.format("Hello %s %s.You are from %s", reques
        return new ResponseClass(greetingString);
    }
}
```



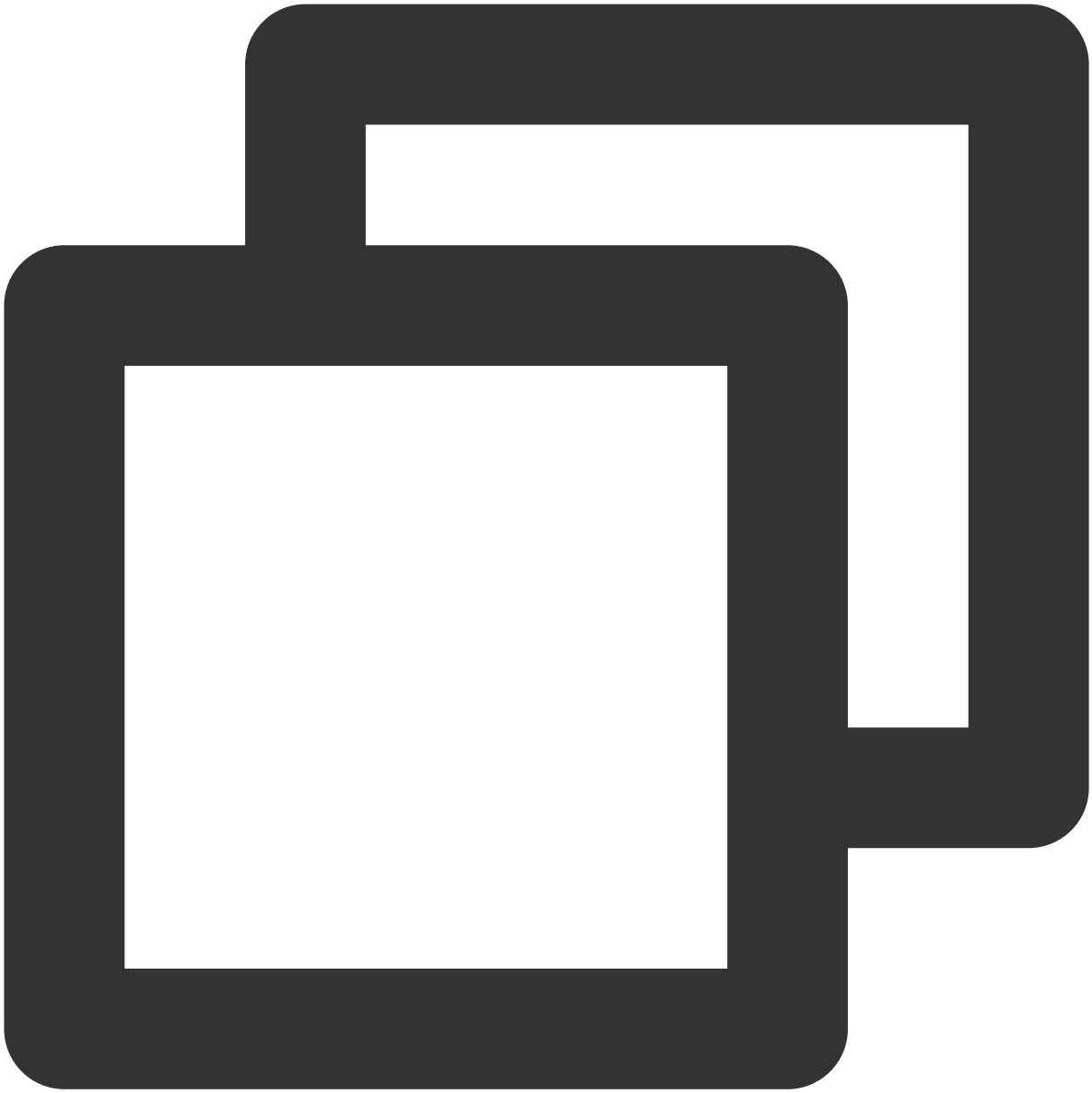
```
package example;

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() {
    }

}
```



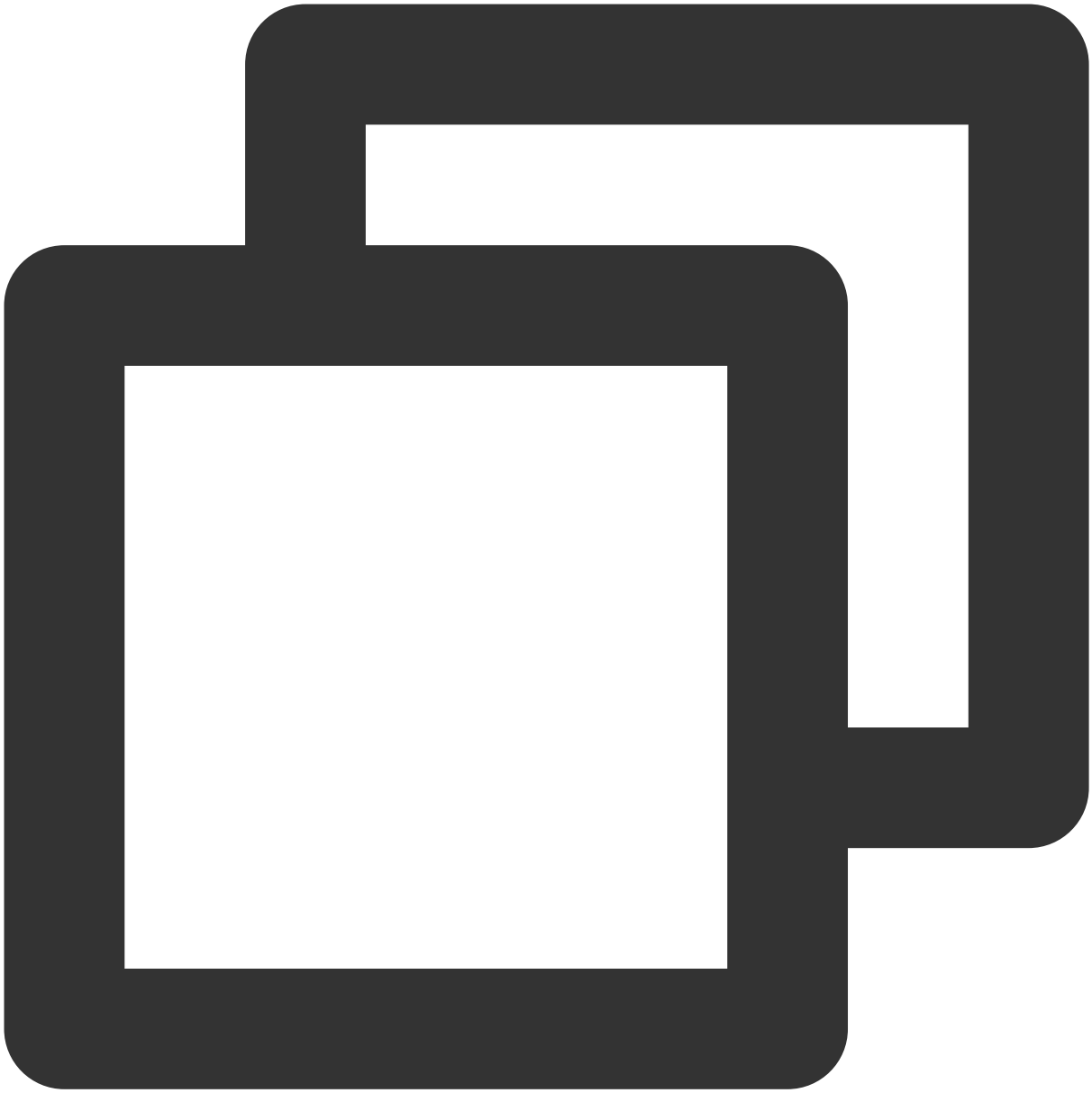
```
package example;

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() {
    }

}
```



```
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() {
    }

}
```



```
package example;

public class ResponseClass {
    String greetings;

    public String getGreetings() {
        return greetings;
    }

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

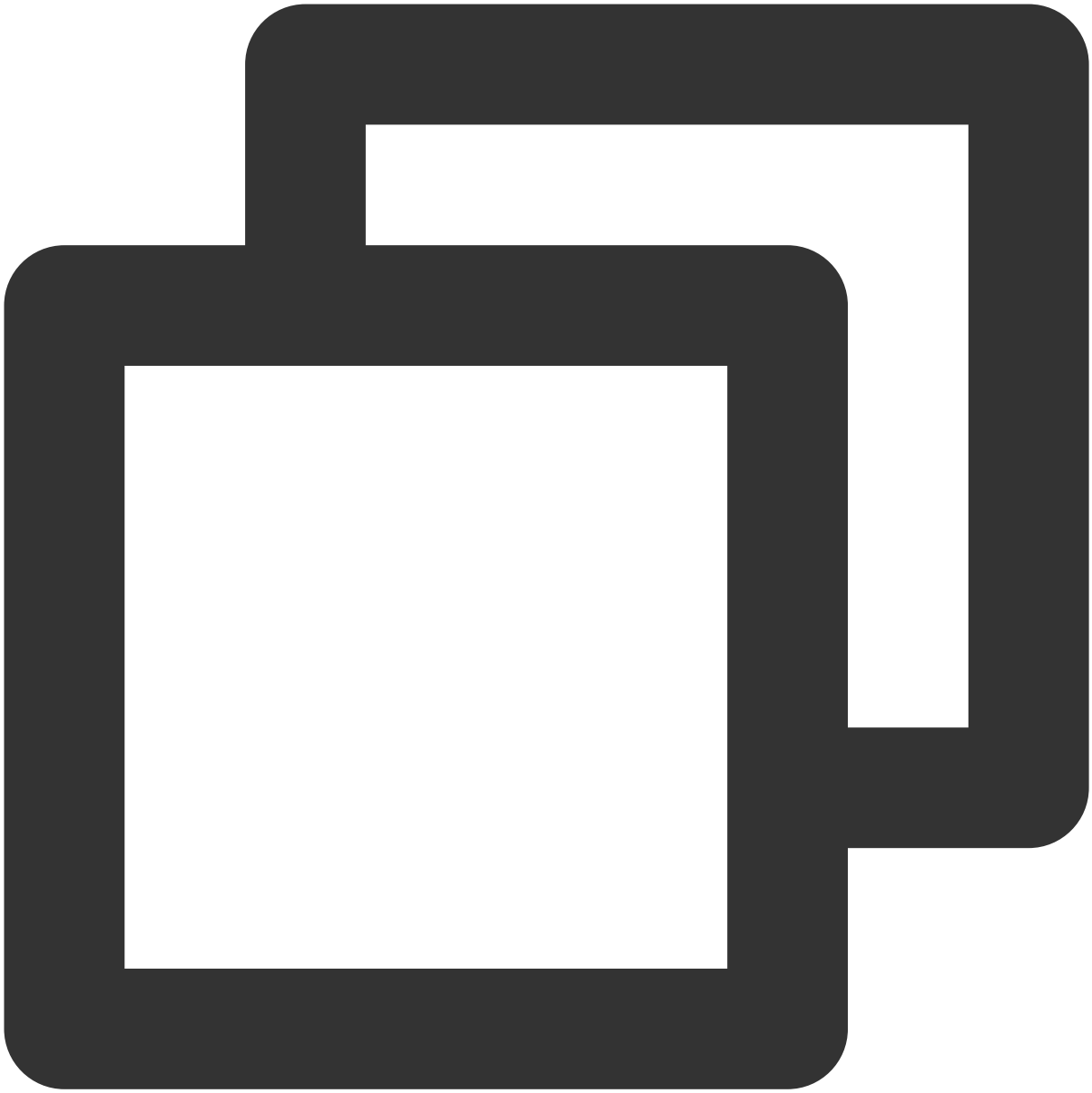
```
    }

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

    public ResponseClass() {
    }

}
```

代码编译

说明

本示例使用 Maven 进行编译打包，您可以根据自身情况，选择使用打包方法。

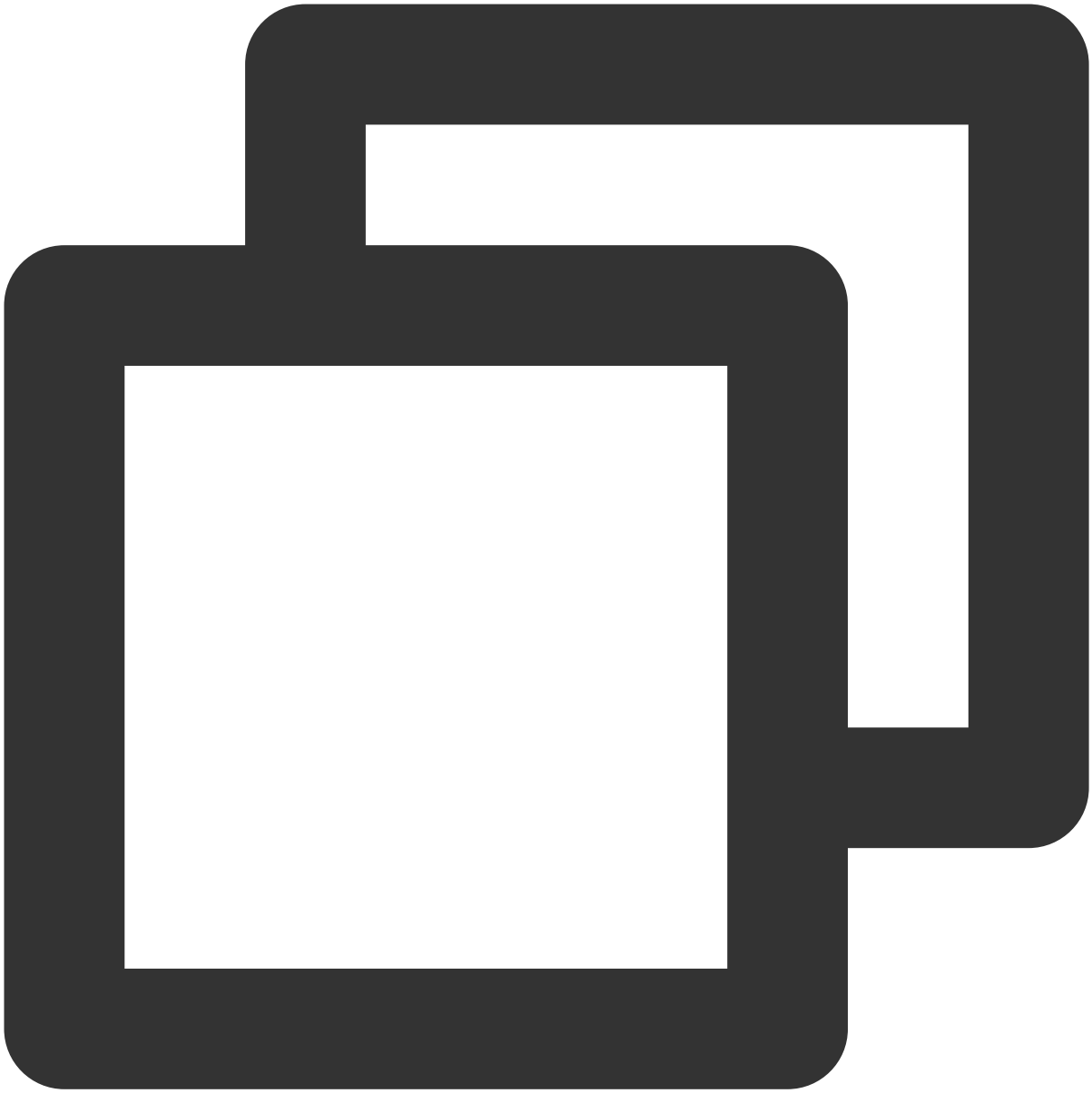
1. 在项目根目录创建 pom.xml 函数，并输入如下内容：



```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/200
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/mave
  <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>
```

2. 在命令行内执行命令 `mvn package` ，确保有编译成功字样，输出结果如下则说明成功打包。



```
[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。

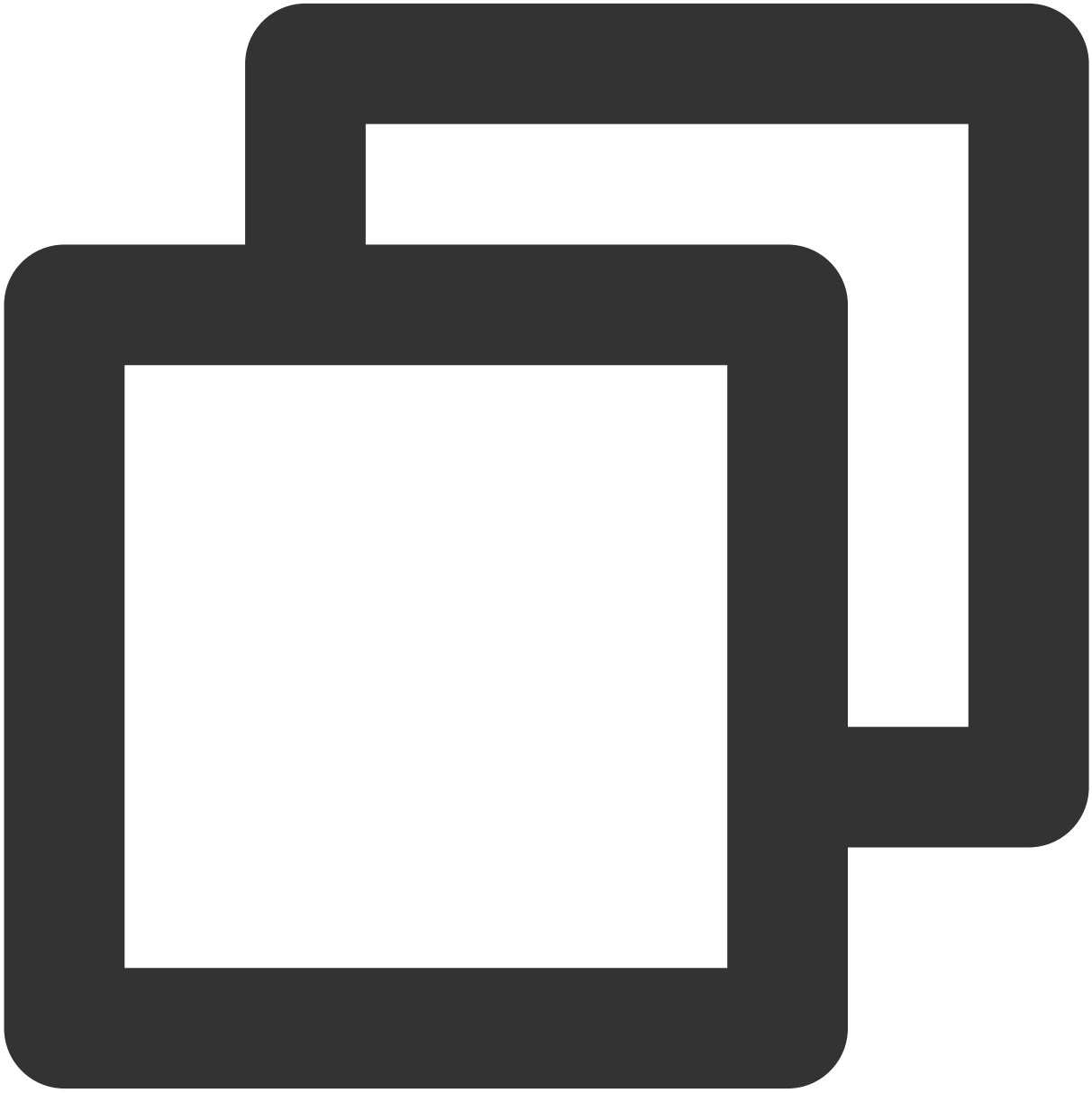
3. 编译后的生成包位于 `target\\java-example-1.0-SNAPSHOT.jar` 。

SCF 云函数创建及测试

1. 创建云函数，详情请参见 创建及更新函数。
2. 使用编译后的包作为提交包上传。

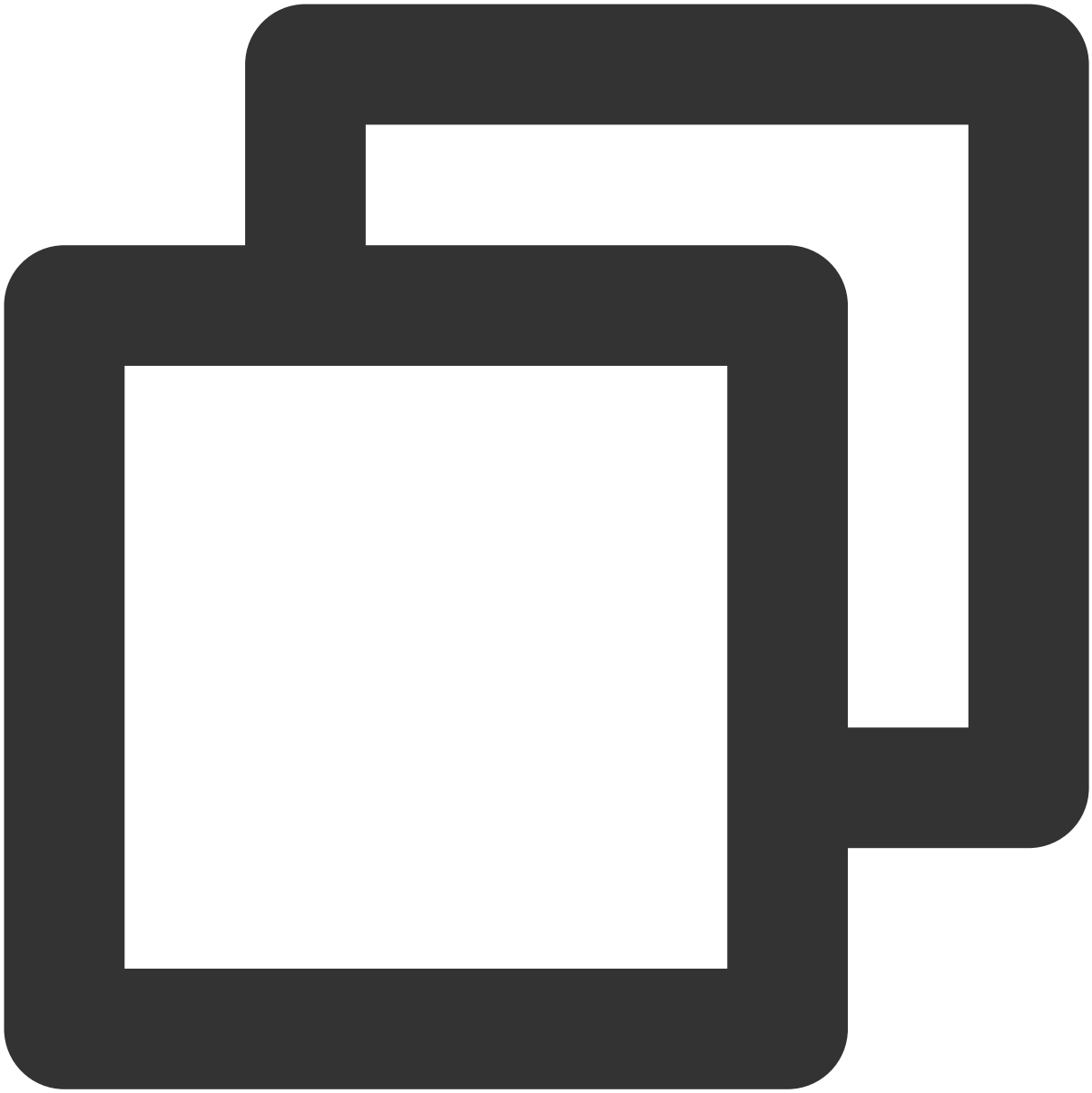
您可以自行选择使用 zip 上传，或先上传至 COS Bucket 后再通过选择 COS Bucket 上传来提交。

3. 设置云函数的执行方法为 `example.Pojo::handle` 。
4. 在**函数代码**页的测试事件模板内输入期望能处理的入参：



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

单击运行后，可查看返回内容为：



```
{   "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。

相关示例

您可前往 scf-demo-java，拉取示例代码并进行测试。

使用 Gradle 创建 zip 部署包

最近更新时间：2024-04-22 18:03:28

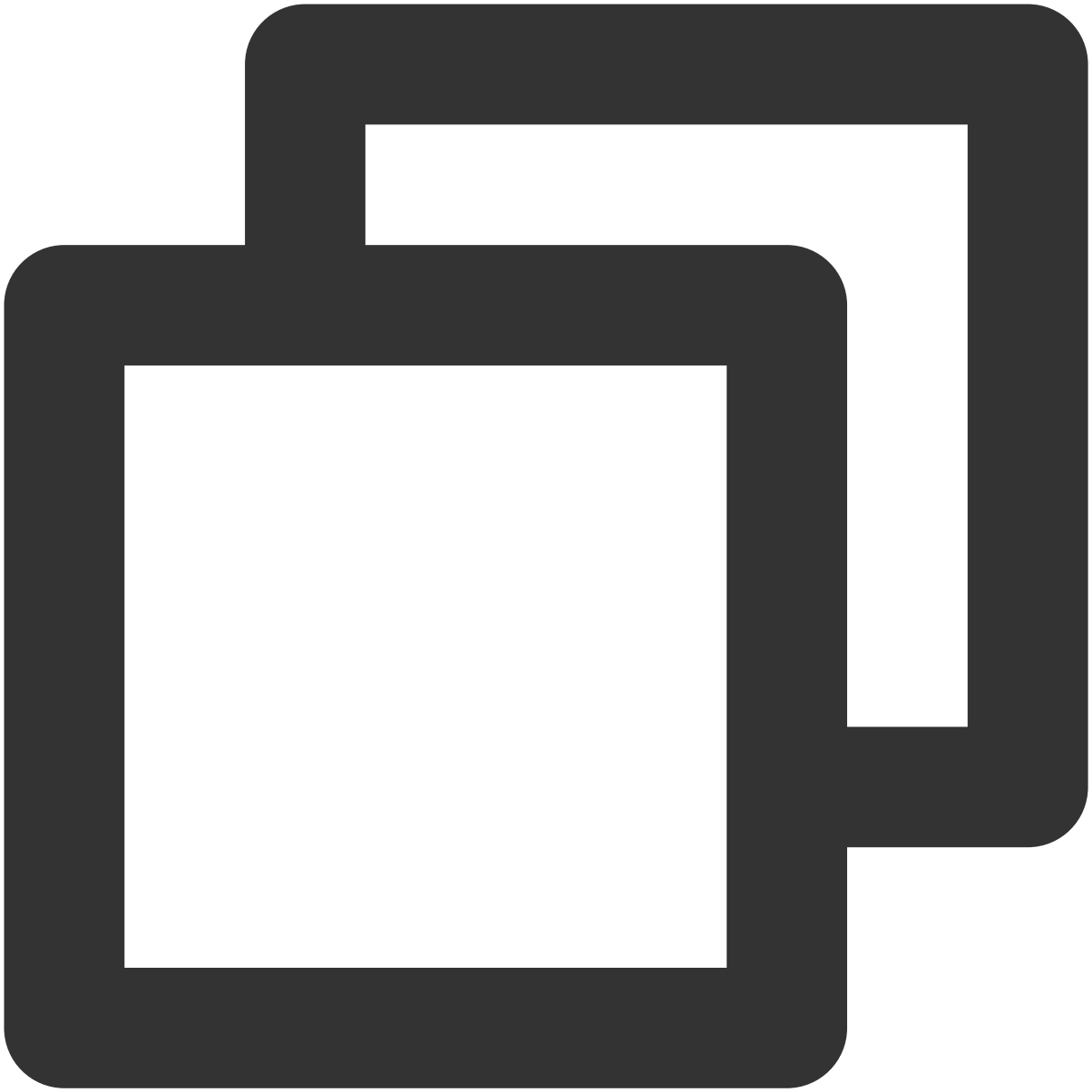
本节内容提供了通过使用 Gradle 工具来创建 Java 类型云函数部署包的方式。创建好的 zip 包符合以下规则，即可以由云函数执行环境所识别和调用。
编译生成的包、类文件、资源文件位于 zip 包的根目录。
依赖所需的 jar 包，位于 /lib 目录内。

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。

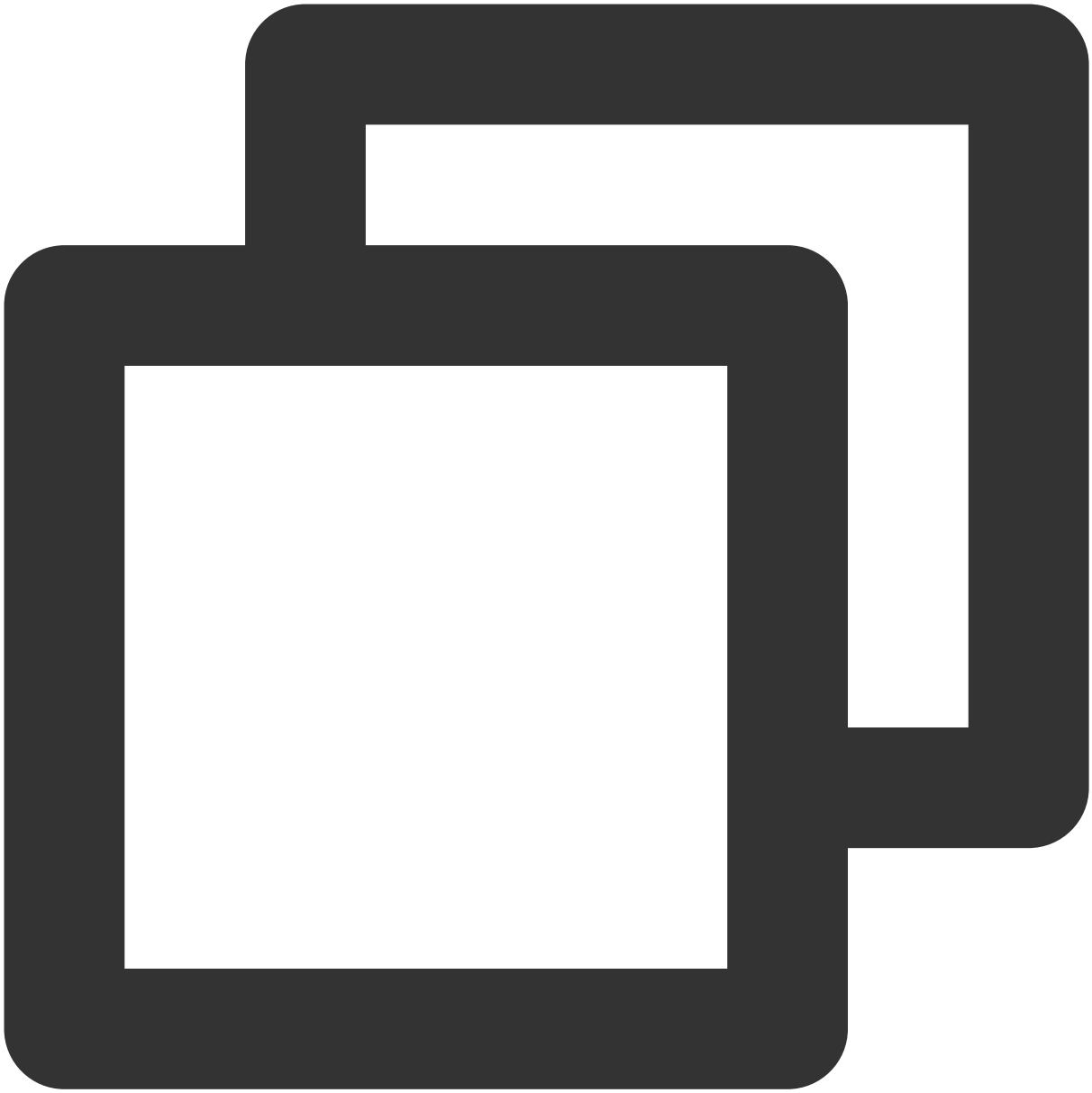
Gradle 安装

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，Windows 下的 `C:\\Gradle` 目录或 Linux 下的 `/opt/gradle/gradle-4.1` 目录。
3. 将解压目录下 bin 目录的路径添加到系统 PATH 环境变量中，请对应操作系统执行以下步骤：
Linux：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. 在命令行执行以下命令，查看 Gradle 是否正确安装。



```
gradle -v
```

输出结果如下，则证明 Gradle 已正确安装。如有问题，请查询 Gradle 的 官方文档。

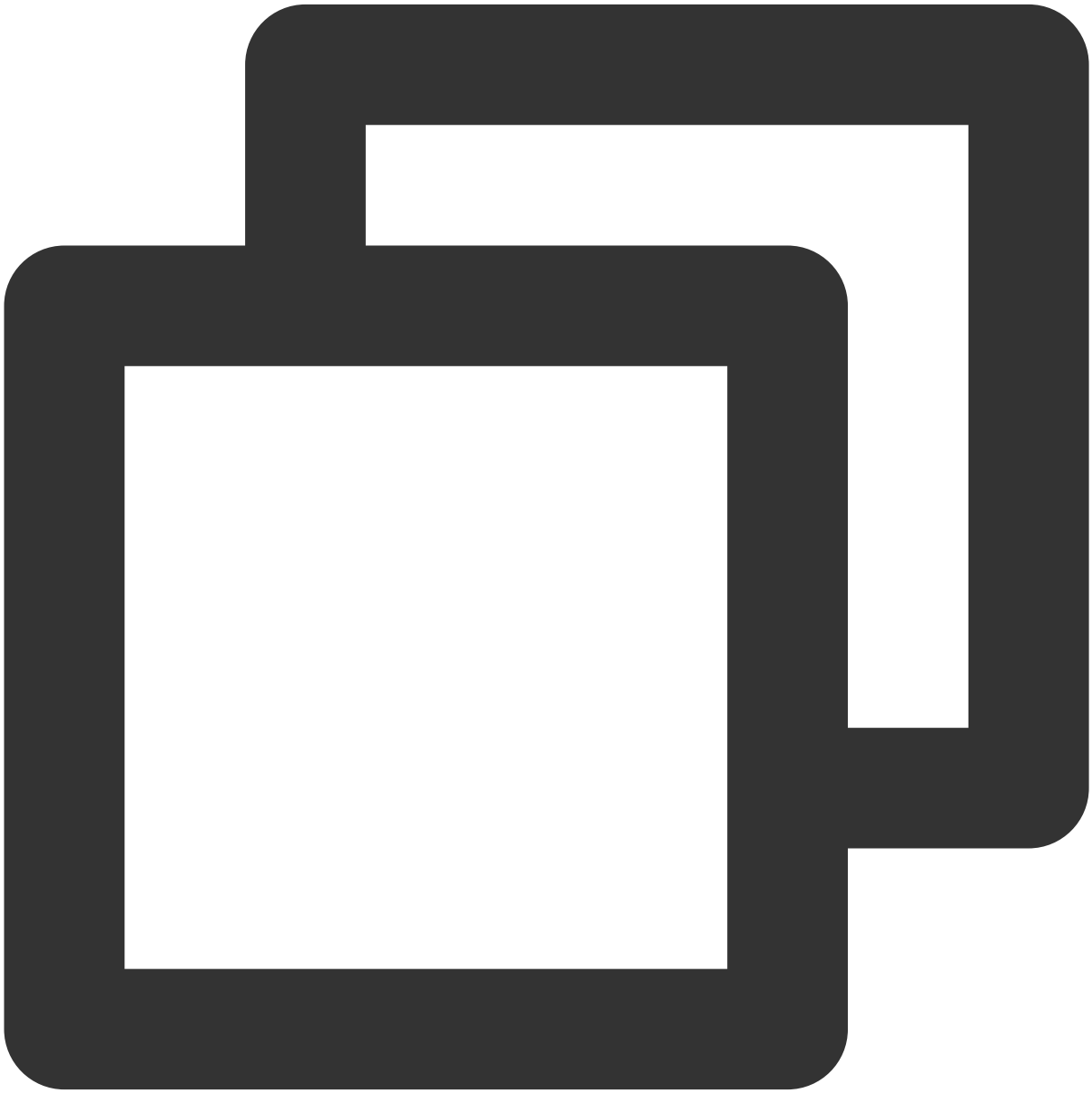


```
------------------------------------------------------------
Gradle 4.1
------------------------------------------------------------
Build time:   2017-08-07 14:38:48 UTC
Revision:     941559e020f6c357ebb08d5c67acdb858a3defc2
Groovy:       2.4.11
Ant:          Apache Ant(TM) version 1.9.6 compiled on June 29 2015
JVM:          1.8.0_144 (Oracle Corporation 25.144-b01)
OS:           Windows 7 6.1 amd64
```

代码准备

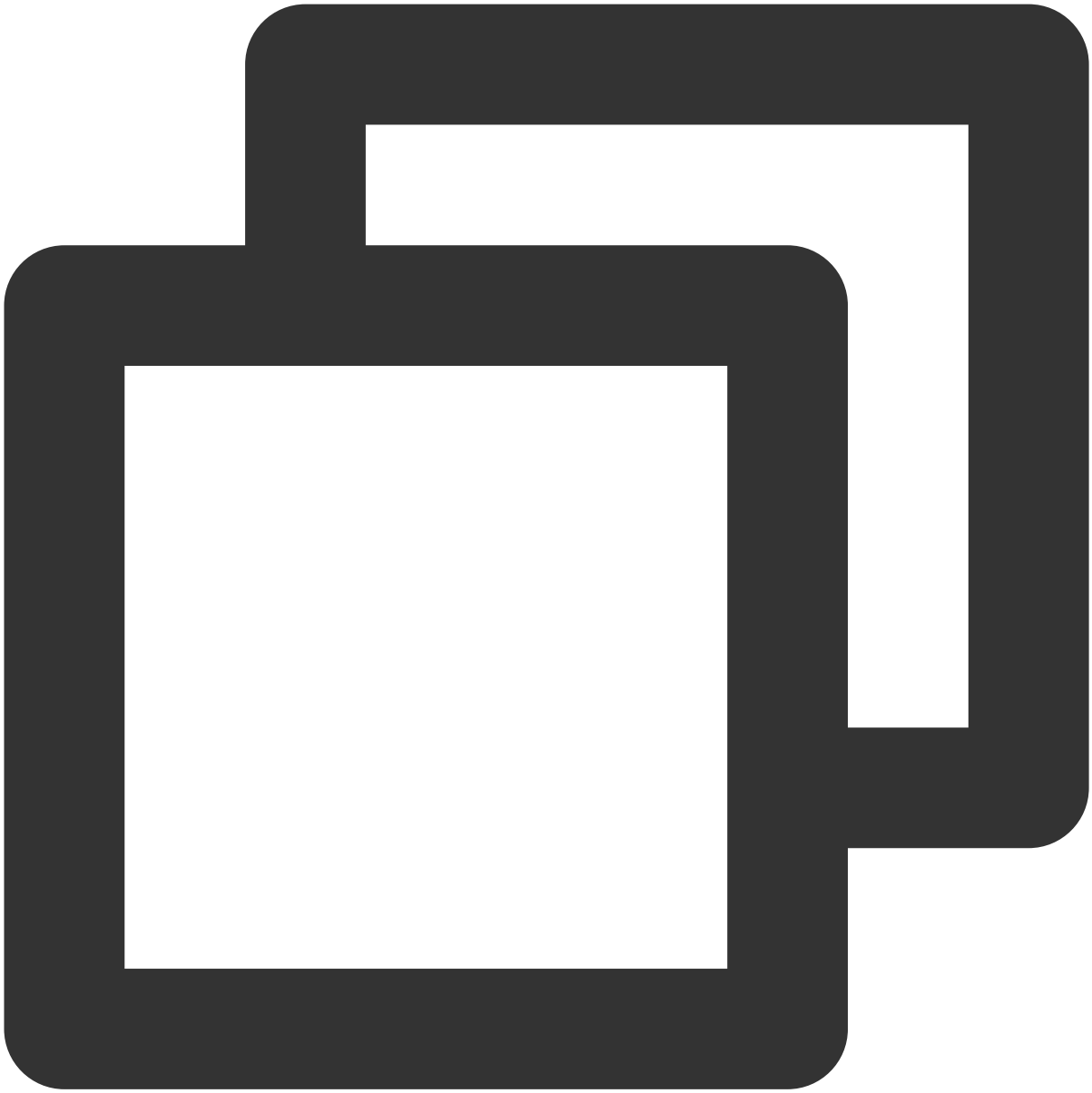
准备代码文件

1. 在选定的位置创建项目文件夹，例如 `scf_example` 。
2. 在项目文件夹根目录，依次创建目录 `src/main/java/` 作为包的存放目录。
3. 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
```

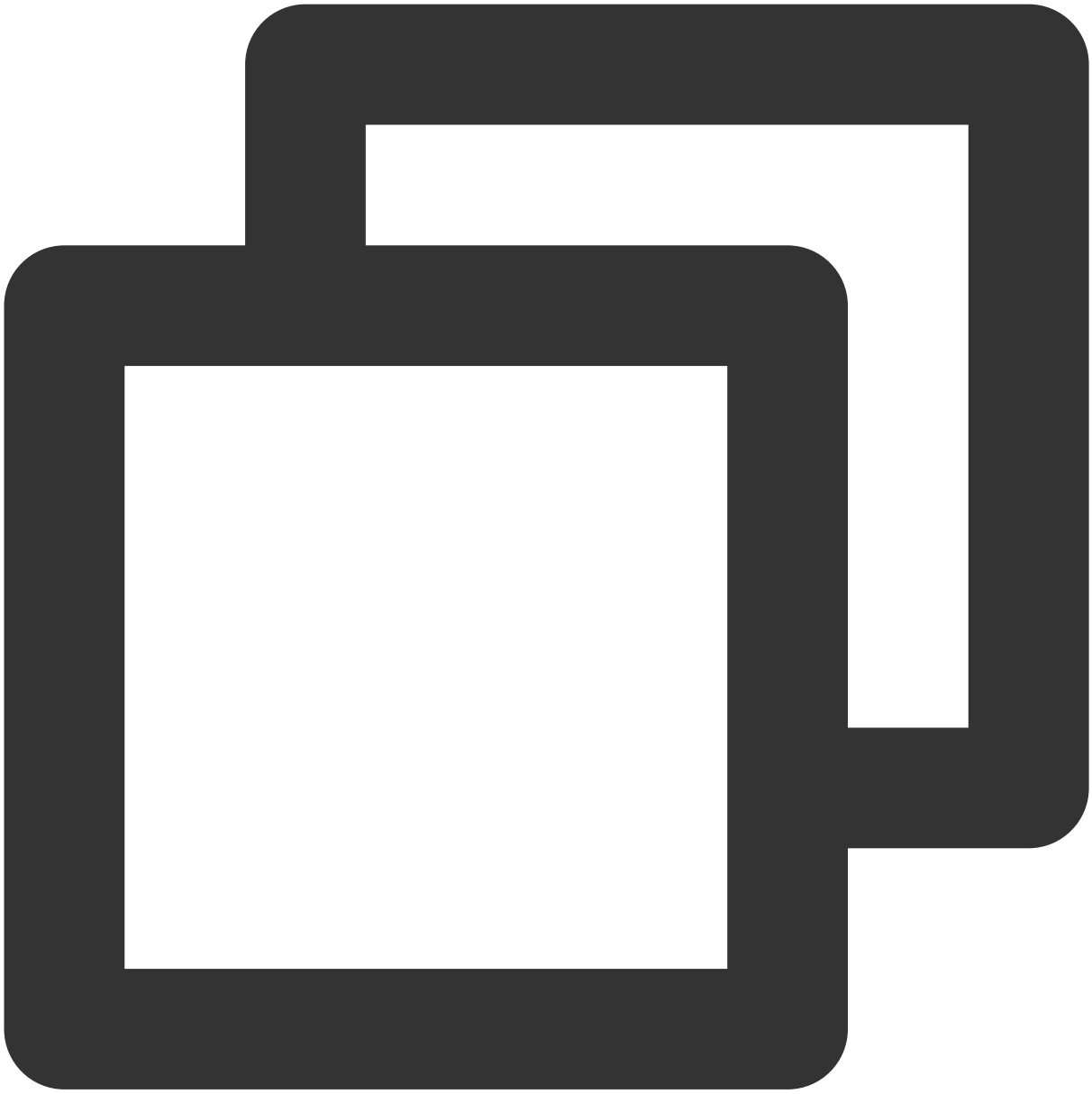
4. 在 `Hello.java` 文件内输入如下代码内容：



```
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：

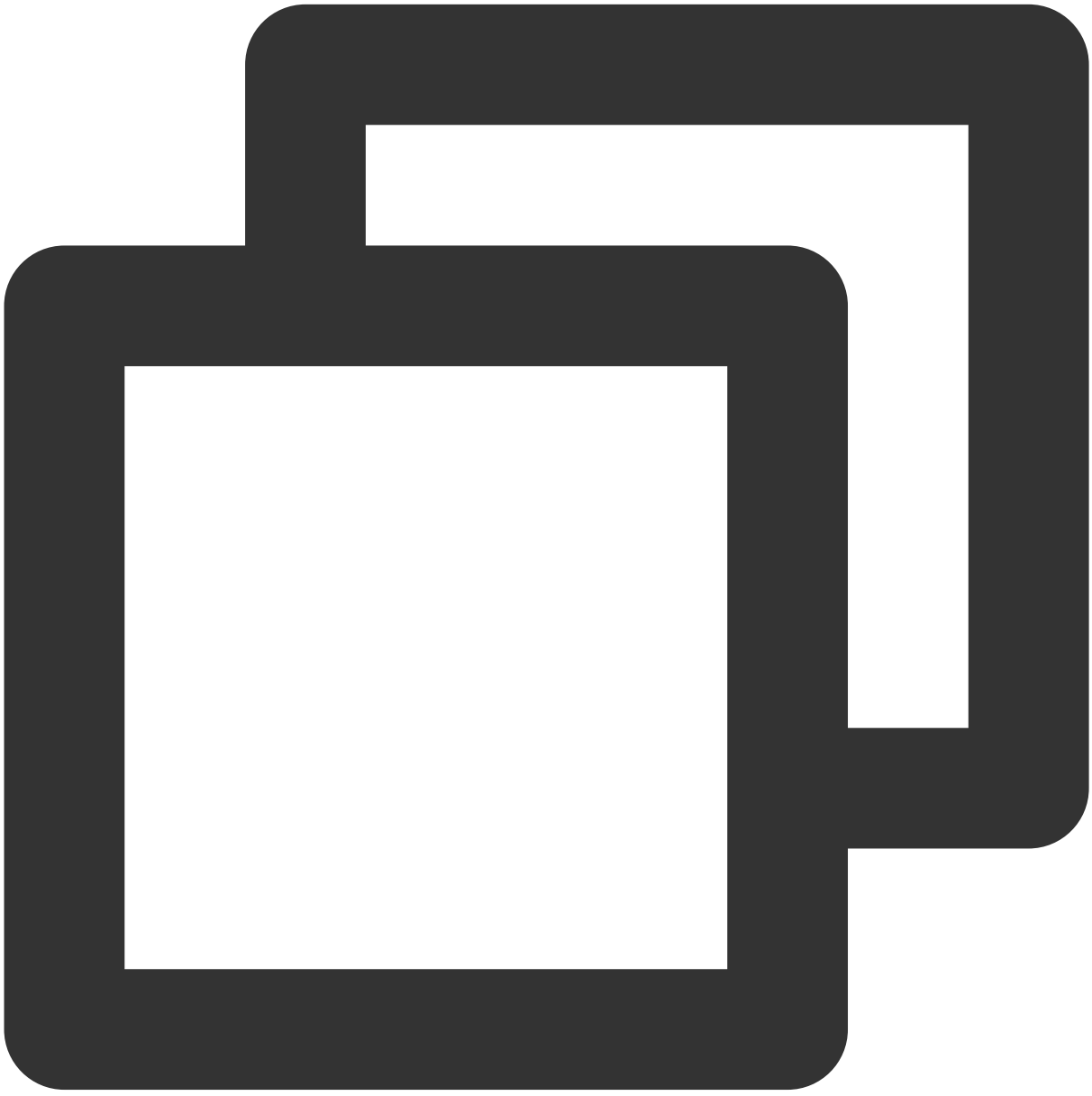


```
apply plugin: 'java'

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，可根据需要添加依赖，`build.gradle` 文件内容如下：



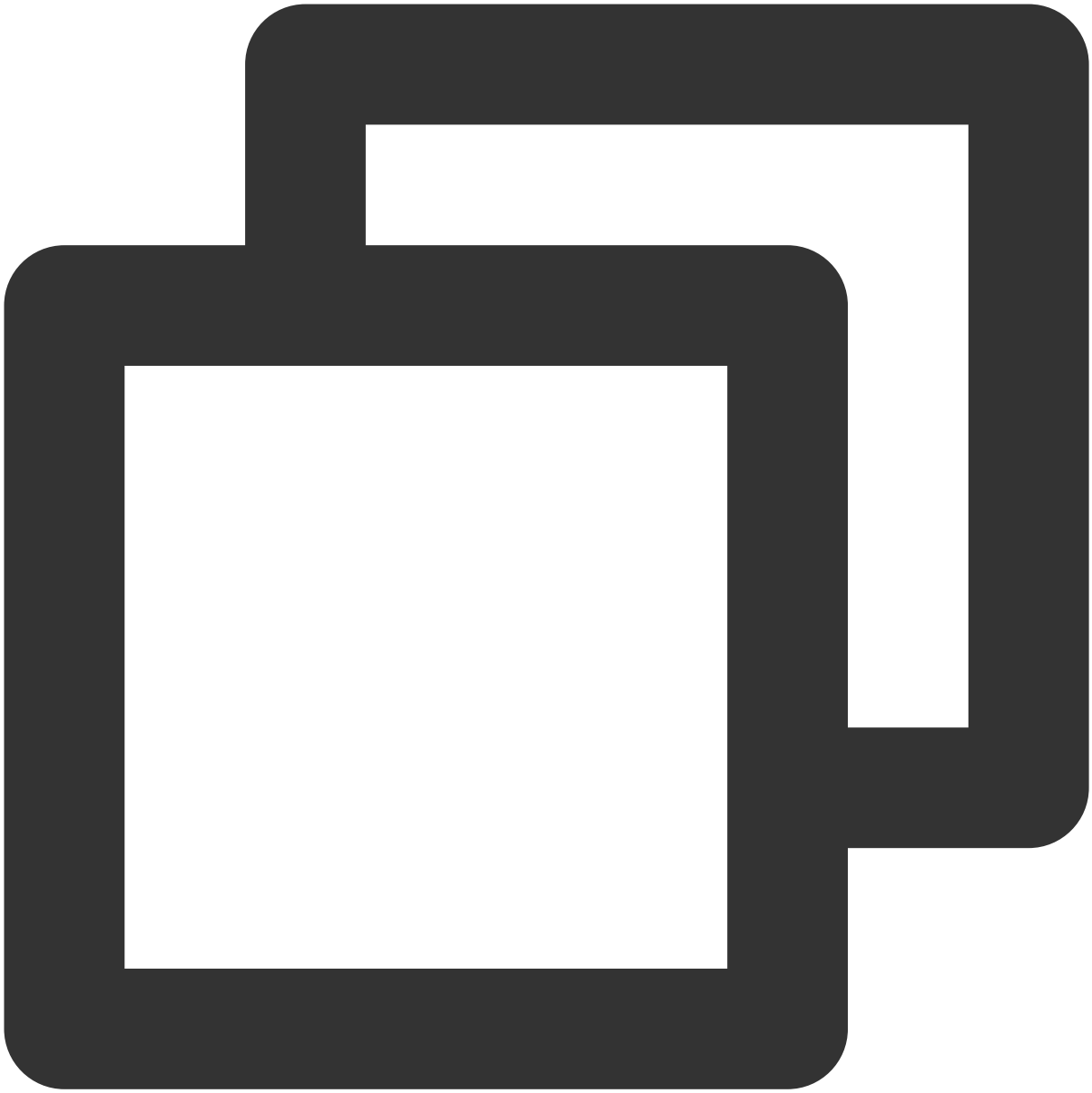
```
apply plugin: 'java'

repositories {
```

```
    mavenCentral()
}

dependencies {
    compile (
        'com.tencentcloudapi:scf-java-events:0.0.2'
    )
}

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.tencentcloudapi:scf-java-events:0.0.2` 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：



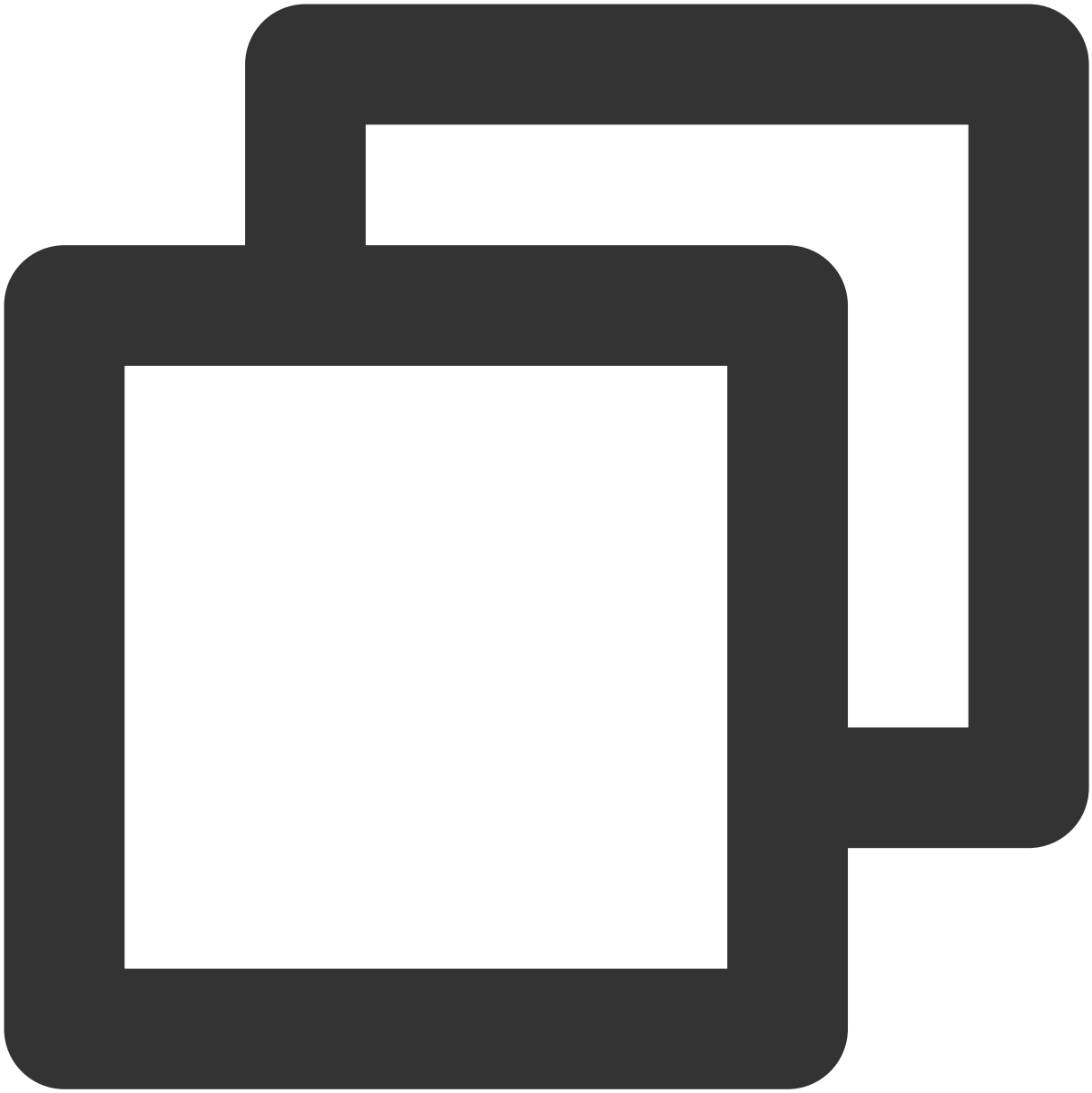
```
apply plugin: 'java'

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。

Custom Runtime

Custom Runtime 说明

最近更新时间：2024-04-22 18:03:28

在云函数（Serverless Cloud Function，SCF）已支持的开发语言及版本的标准运行环境外，为了满足更多个性化开发语言及版本的函数实现，SCF 提供了 Custom Runtime 服务，即可定制化运行环境。通过开放实现自定义函数运行时，支持根据需求使用任意开发语言的任意版本来编写函数，并实现函数调用中的全局操作，如扩展程序的加载，安全插件，监控 agent 等。SCF 与 Custom Runtime 通过 HTTP 协议通信完成事件的响应处理。

Custom Runtime 部署文件说明

bootstrap：Custom Runtime 固定的可执行引导程序文件，由开发者创建同名可执行程序文件，通过自定义语言及版本实现，直接处理或者通过启动其他可执行文件来处理，完成函数运行时的初始化和调用。
函数文件：开发者通过自定义语言及版本开发实现的函数程序文件。
运行时依赖的库文件或可执行文件：根据自定义语言及版本运行时需要，添加相关依赖的库文件或可执行文件。
通过部署包方式发布，文件构成如下：
bootstrap（必须）
函数文件（必须）
运行时依赖的库文件或可执行文件（可选）
由于部署包体积限制，运行时依赖的库文件或者可执行文件体积较大时，建议通过部署包绑定层的组合方式发布，文件构成如下：
部署包：
bootstrap（必须）
函数文件（必须）
层：
运行时依赖的库文件或可执行文件（可选）
注意：
以上部署文件中的可执行文件发布至 SCF 前请务必设置好文件的可执行权限，并将部署文件打包后通过 zip 包方式直接上传或通过 COS 上传。例如，当您在 Linux 系统中使用 root 账户编译文件，您可以使用 `chmod -R 777 your_path` 指令更改文件的权限。

Custom Runtime 运行机制

Custom Runtime 将函数的运行时周期分为初始化阶段和调用阶段。初始化阶段只在实例冷启动过程中一次性执行，调用阶段是实例启动之后每次响应事件调用的执行过程。

不同开发语言及版本的启动时间和执行时间会有差异，SCF 针对 Custom Runtime 增加了**初始化超时时间**配置，与**执行超时时间**配置一起管理 Custom Runtime 的运行时生命周期。

函数引导加载

SCF 首先检索部署包中的可执行引导文件 bootstrap，根据检索结果并进行如下操作：
检索到 bootstrap 文件且可执行，加载执行 bootstrap 程序，进入函数初始化阶段。
未检索到 bootstrap 文件或文件不可执行，返回 bootstrap 文件不存在，启动失败。

函数初始化

开始于执行 bootstrap 引导程序文件，开发者可以根据需要在 bootstrap 中自定义实现个性化操作，直接处理或调用其他可执行程序文件来完成初始化操作。以下为建议在初始化阶段完成的基础操作：
设定运行时依赖库的路径及环境变量等。
加载自定义语言及版本依赖的库文件及扩展程序等，如仍有依赖文件需要实时拉取，可下载至 `/tmp` 目录。
解析函数文件，并执行函数调用前所需的全局操作或初始化程序（如开发工具包客户端 HTTP CLIENT 等初始化、数据库连接池创建等），便于调用阶段复用。
启动安全、监控等插件。
初始化阶段完成后，需要主动调用运行时 API，访问初始化就绪接口 `/runtime/init/ready` 通知 SCF，Custom Runtime 运行时已完成初始化，进入就绪状态，否则 SCF 会持续等待，直到达到配置的初始化超时时间后，结束 Custom Runtime 并返回初始化超时错误。若重复通知，则会以首次访问时间作为就绪状态时间结点。

日志及异常

SCF 将会收集初始化阶段所有标准输出上报至日志。
函数初始化在超时限制内正常执行完成，初始化阶段的日志会与首次调用日志合并返回。如果在初始化超时限制内由于错误异常未能完成执行，则执行结果返回初始化超时错误，写入标准输出的程序错误及异常日志会上报给 SCF，在控制台日志及日志查询中展示。

函数调用

函数调用阶段需要开发者自定义实现事件获取、调用函数及结果的返回，并循环处理这个过程。
长轮询获取事件，开发者通过自定义语言及版本实现 HTTP CLIENT 访问运行时 API 的事件获取接口 `/runtime/invocation/next`，响应正文包含事件数据，在一次调用内重复访问此接口均返回相同事件数据。
注意：
HTTP CLIENT 请勿设置 get 方法的超时。
根据环境变量、响应头中所需信息及事件信息构建函数调用的参数。
推送事件信息等参数数据，调用函数处理程序。
访问运行时 API 响应结果接口 `/runtime/invocation/response` 推送函数处理结果，首次调用成功为事件终态，SCF 将进行状态锁定，推送后结果不可变更。
如果函数调用阶段出现错误，通过访问运行时 API 调用错误接口 `/runtime/invocation/error` 推送错误信息，同时本次调用结束，首次调用视为事件终态，SCF 将进行状态锁定，继续推送结果不可变更。

清理释放本次调用结束后不再需要的资源。

日志及异常

SCF 将会收集调用阶段所有标准输出上报至日志。
SCF 下发事件后，Custom Runtime 较长时间未获取事件，超过函数执行超时限制，SCF 将结束实例并返回等待获取事件超时错误。
SCF 下发事件后，Custom Runtime 获取到事件但未在函数执行超时限制内返回执行结果，SCF 将结束实例并返回执行超时错误。

Custom Runtime 运行时 API

Custom Runtime 由开发者使用自定义语言及版本实现，与 SCF 之间的事件下发、处理结果反馈等需要通过标准协议来进行通信。因此 SCF 提供了运行时 API，来满足与 Custom Runtime 生命期中的交互需要。
SCF 内置有以下环境变量：
SCF_RUNTIME_API：运行时 API 地址。
SCF_RUNTIME_API_PORT：运行时 API 端口。
更多信息请参阅 环境变量。
Custom Runtime 可以通过 `SCF_RUNTIME_API:SCF_RUNTIME_API_PORT` 来访问运行时 API。

路径	方法	说明
/runtime/init/ready	post	运行时初始化完成后，调用接口标识进入就绪状态。
/runtime/invocation/next	get	获取调用事件。 响应头包含以下信息： request_id：请求 ID，用于标识触发了函数调用的请求。 memory_limit_in_mb：函数内存限制，单位为MB。 time_limit_in_ms：函数超时时间，单位为毫秒。 响应体包含事件数据，结构参见 触发器事件消息结构汇总。
/runtime/invocation/response	post	函数处理结果。在运行时调用函数处理程序后，将来自函数的响应推送到调用响应接口。
/runtime/invocation/error	post	函数返回错误推送到调用错误接口，标识本次调用失败。

Custom Runtime 创建 Bash 示例函数

最近更新时间：2024-04-22 18:03:28

操作场景

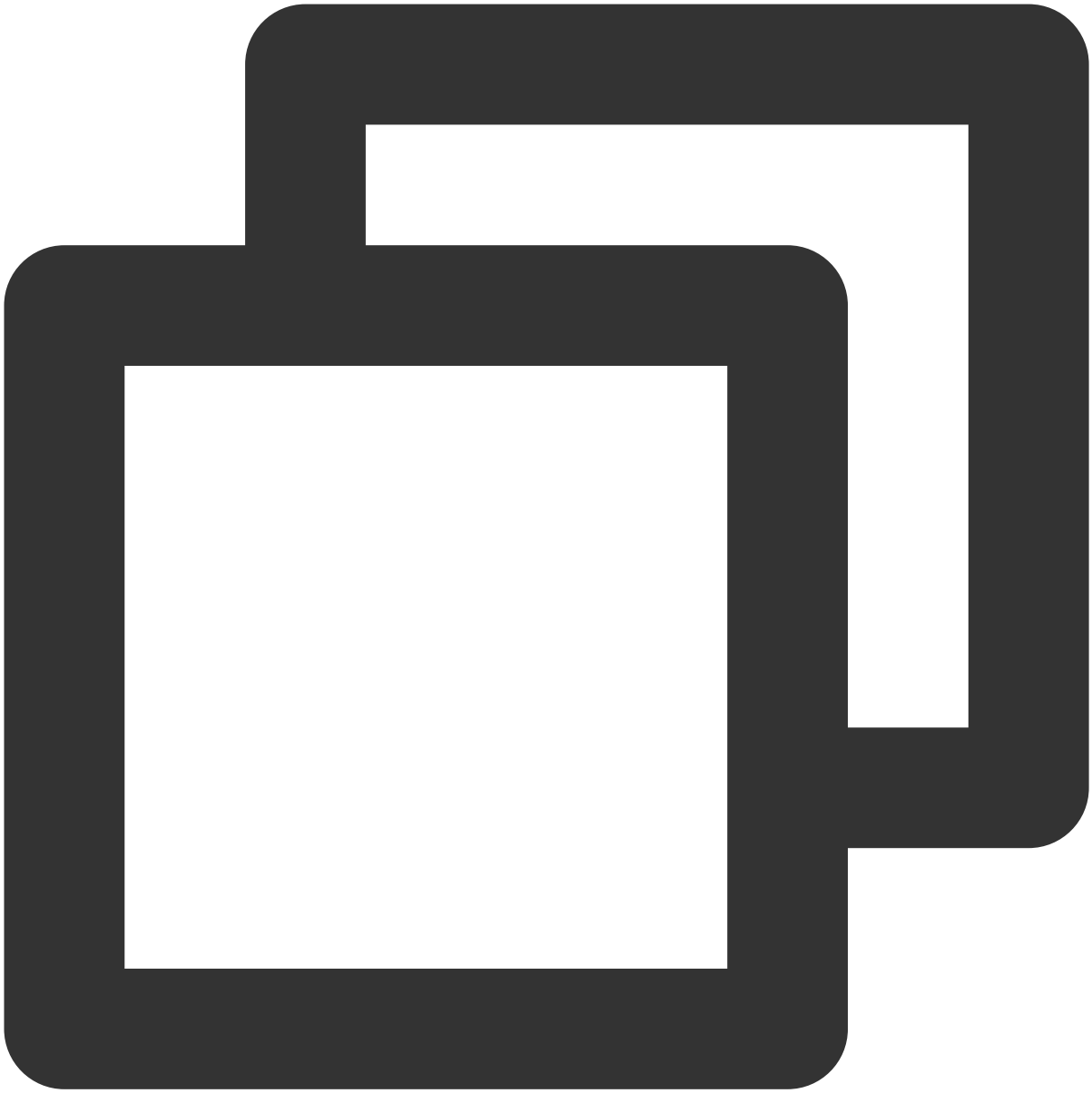
本文档介绍如何新建 Custom Runtime 云函数，并将其打包发布，响应触发事件。您可通过本文了解 Custom Runtime 的开发流程及运行机制。

操作步骤

创建 Custom Runtime 云函数前，需要创建运行时引导文件 bootstrap 和 函数处理文件。

创建 bootstrap 文件

bootstrap 是运行时入口引导程序文件，Custom Runtime 加载函数时固定检索 bootstrap 同名文件，并执行该程序来启动 Custom Runtime 运行时。Custom Runtime 支持任意语言及版本开发运行函数，主要基于 bootstrap 引导程序由开发者自定义实现。其中，bootstrap 需具备以下条件：
需具有可执行权限。
能够在 SCF 系统环境（CentOS 7.6）中运行。
您可参考以下示例代码，在命令行终端创建 bootstrap 文件。本示例通过 bash 实现。



```
#! /bin/bash
set -euo pipefail

# 初始化 - 加载函数文件
source ./"$(echo $_HANDLER | cut -d. -f1).sh"

# 初始化完成，访问运行时API上报就绪状态
curl -d " " -X POST -s "http://$SCF_RUNTIME_API:$SCF_RUNTIME_API_PORT/runtime/init/

### 循环监听处理事件调用
while true
```

```
do
  HEADERS="$(mktemp)"
  # 长轮询获取事件
  EVENT_DATA=$(curl -sS -LD "$HEADERS" -X GET -s "http://$SCF_RUNTIME_API:$SCF_RUNT
  # 调用函数处理事件
  RESPONSE=$($(echo "$_HANDLER" | cut -d. -f2) "$EVENT_DATA")
  # 推送函数处理结果
  curl -X POST -s "http://$SCF_RUNTIME_API:$SCF_RUNTIME_API_PORT/runtime/invocation
done
```

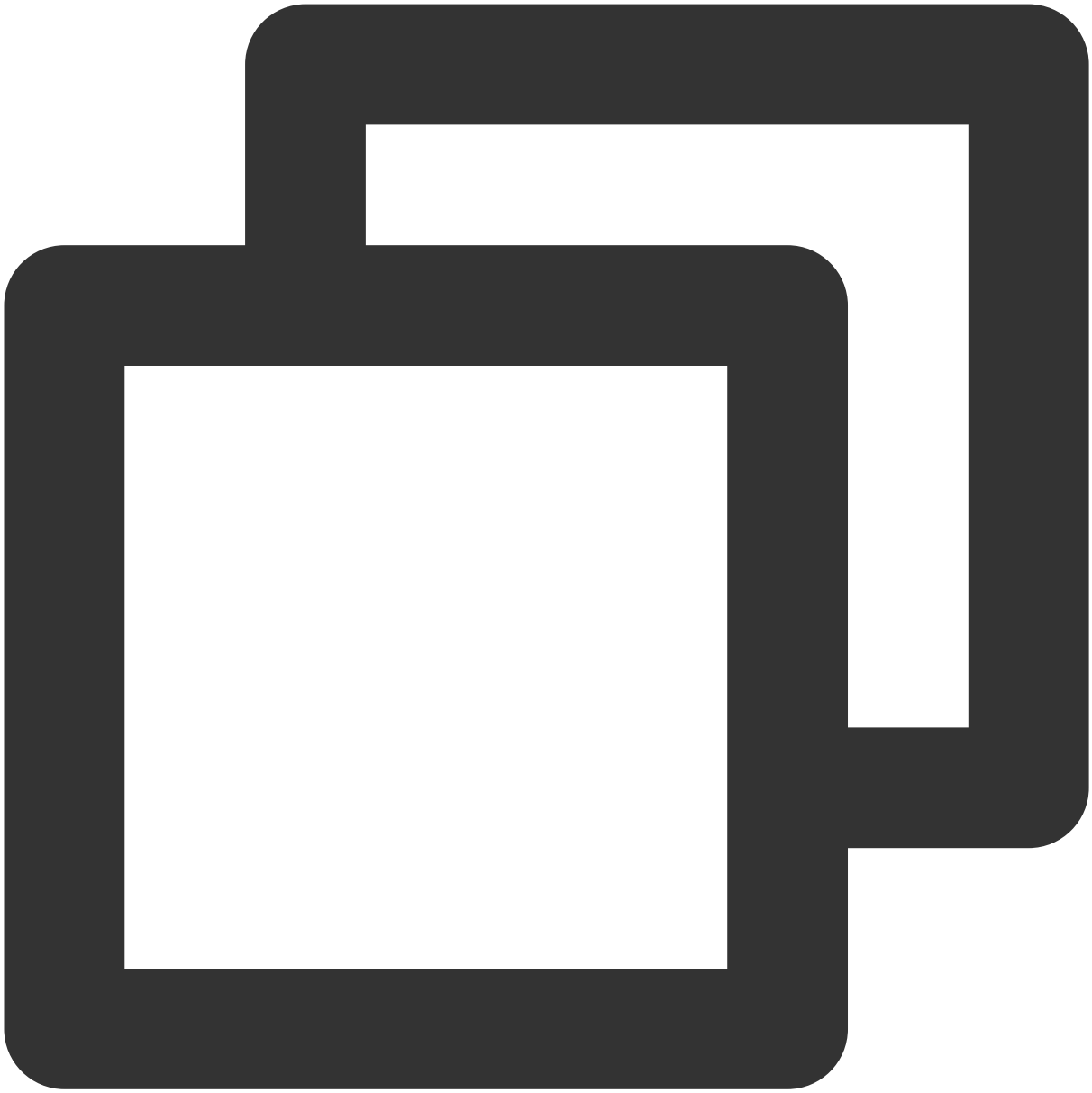
示例文件解析

在示例中，Custom Runtime 运行时分为初始化阶段和调用阶段。初始化阶段仅在函数的执行实例冷启动过程中一次性执行。初始化完成后进入循环的调用阶段，监听事件并调用函数处理。

初始化阶段

关于初始化阶段详细信息，请查阅 函数初始化。

初始化阶段完成后，需要主动调用运行时 API 初始化就绪接口通知 SCF。示例代码如下：



```
# 初始化完成，访问运行时API上报就绪状态
curl -d " " -X POST -s "http://$SCF_RUNTIME_API:$SCF_RUNTIME_API_PORT/runtime/init/
```

其中，由于 Custom Runtime 由开发者使用自定义语言及版本实现，需要通过标准协议与 SCF 进行通信，本实例中 SCF 通过 HTTP 协议提供运行时 API 及内置环境变量，更多关于环境变量信息请参见 环境变量。

`SCF_RUNTIME_API` ：运行时 API 地址。

`SCF_RUNTIME_API_PORT` ：运行时 API 端口。

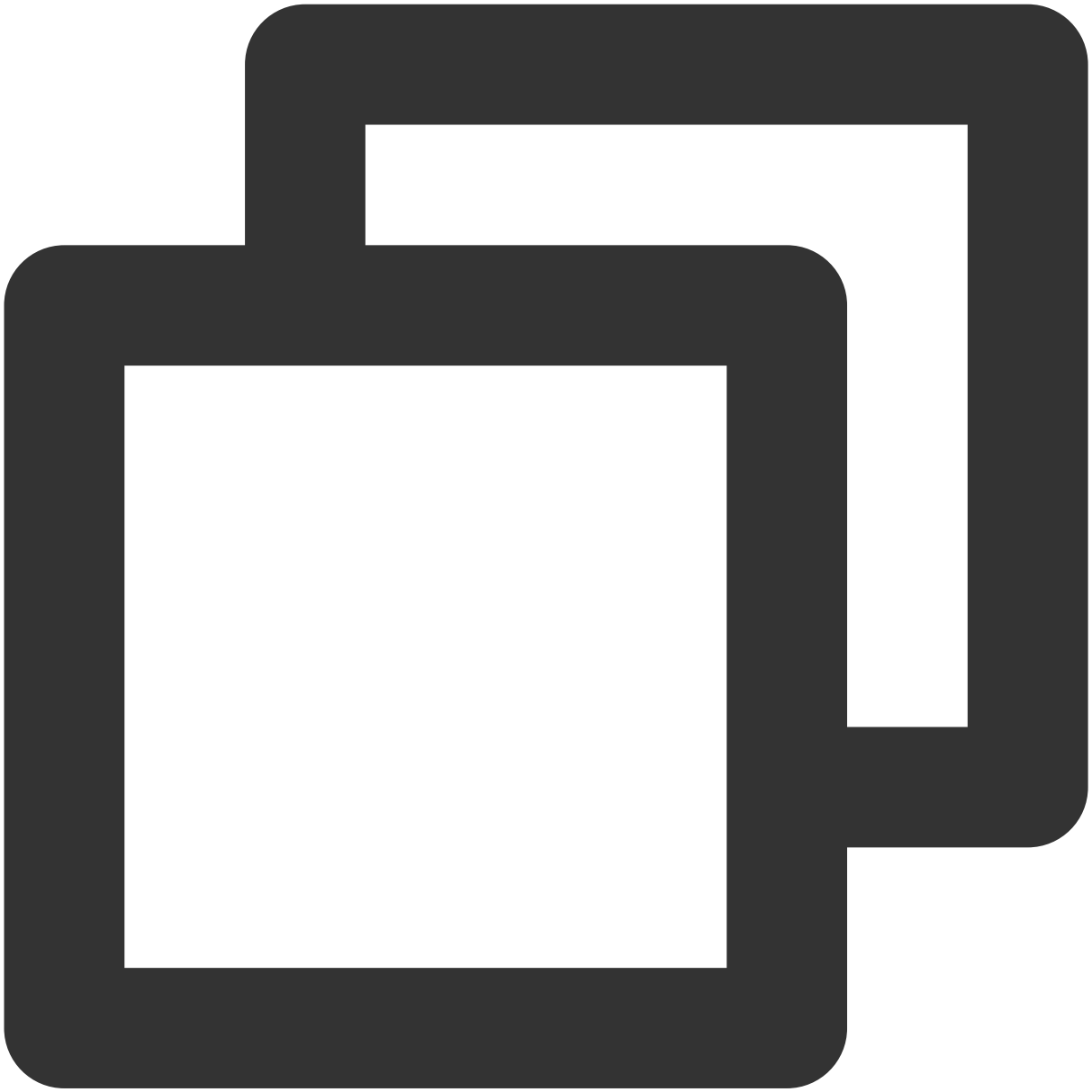
初始化日志及异常

初始化阶段日志及异常信息，请参见 日志及异常。

调用阶段

关于调用阶段详细信息，请参见 函数调用。

1.1 完成初始化后，进入循环的调用阶段，监听事件并调用函数处理。示例代码如下：



```
# 长轮询获取事件
  EVENT_DATA=$(curl -sS -LD "$HEADERS" -X GET -s "http://$SCF_RUNTIME_API:$SCF_RUNT
```

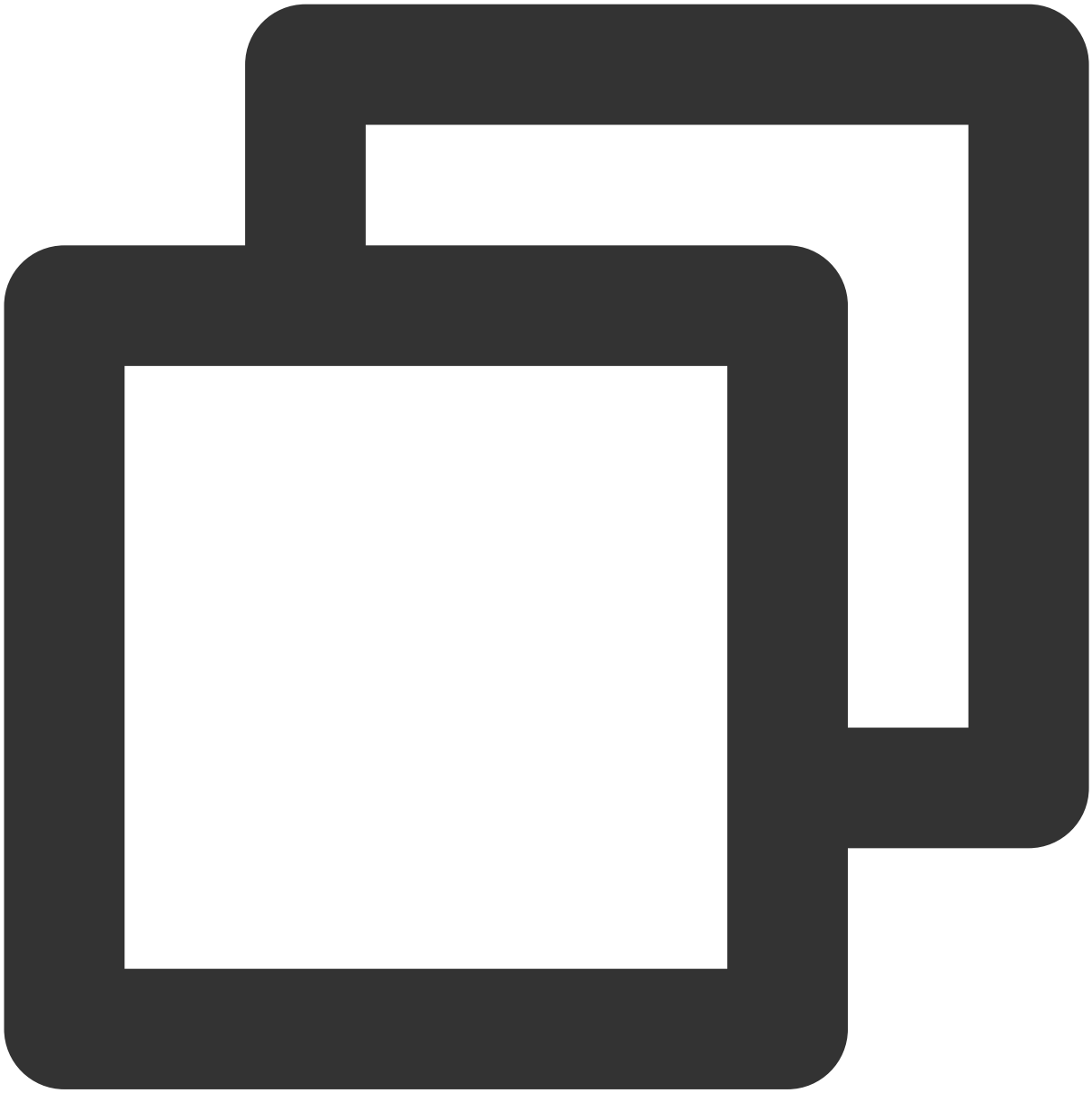
长轮询获取事件请勿设置 get 方法超时，在访问运行时 API 的事件获取接口，阻塞等待事件下发，在一次调用内重复访问此接口均返回相同事件数据。响应体为事件数据 event_data，响应头包含以下信息：

Request_Id：请求 ID，用于标识触发了函数调用的请求。

Memory_Limit_In_Mb：函数内存限制，单位为 MB。

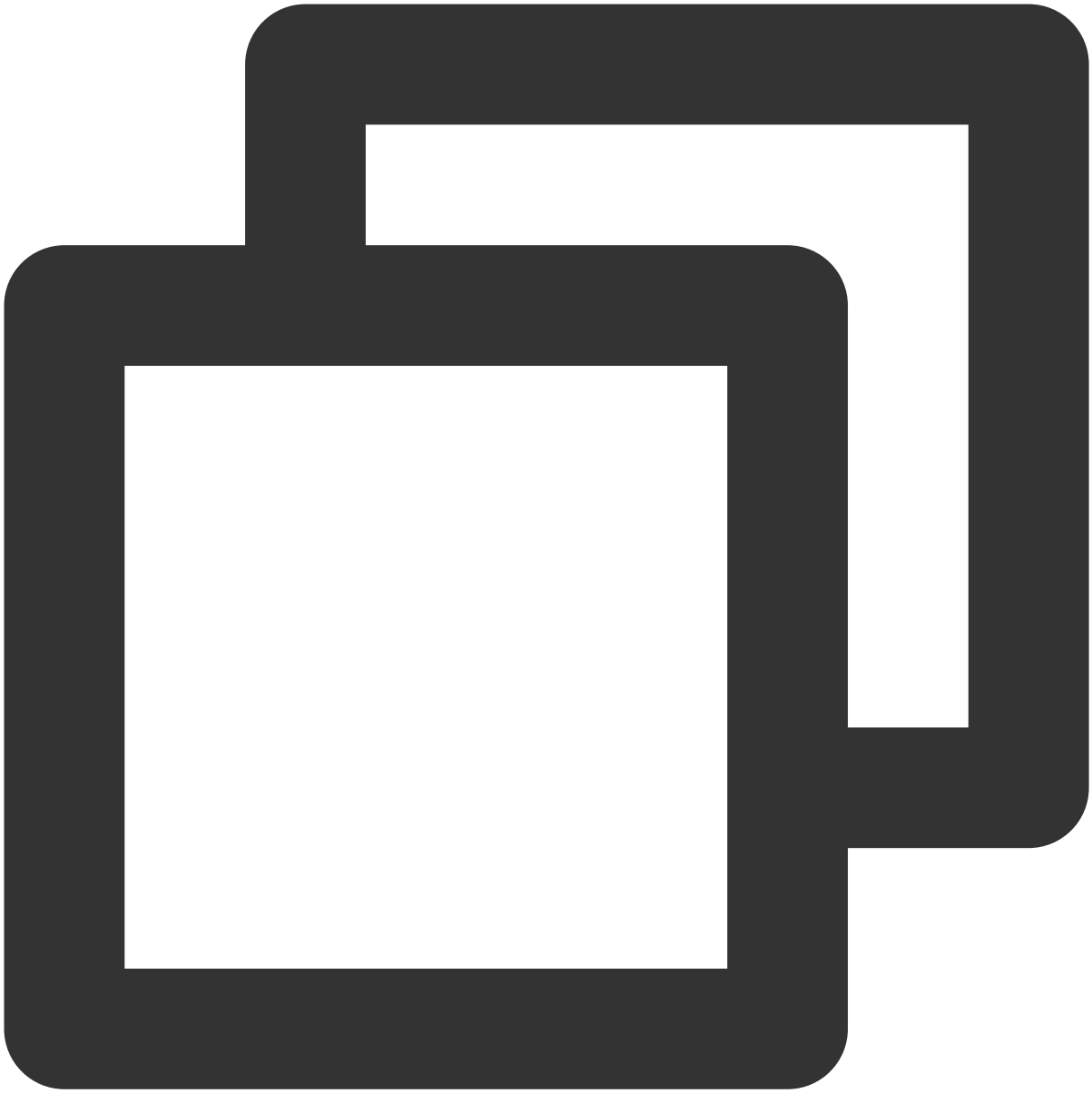
Time_Limit_In_Ms：函数超时时间，单位为毫秒。

1.2 根据环境变量、响应头中所需信息及事件信息构建函数调用的参数，调用函数处理程序。示例代码如下：



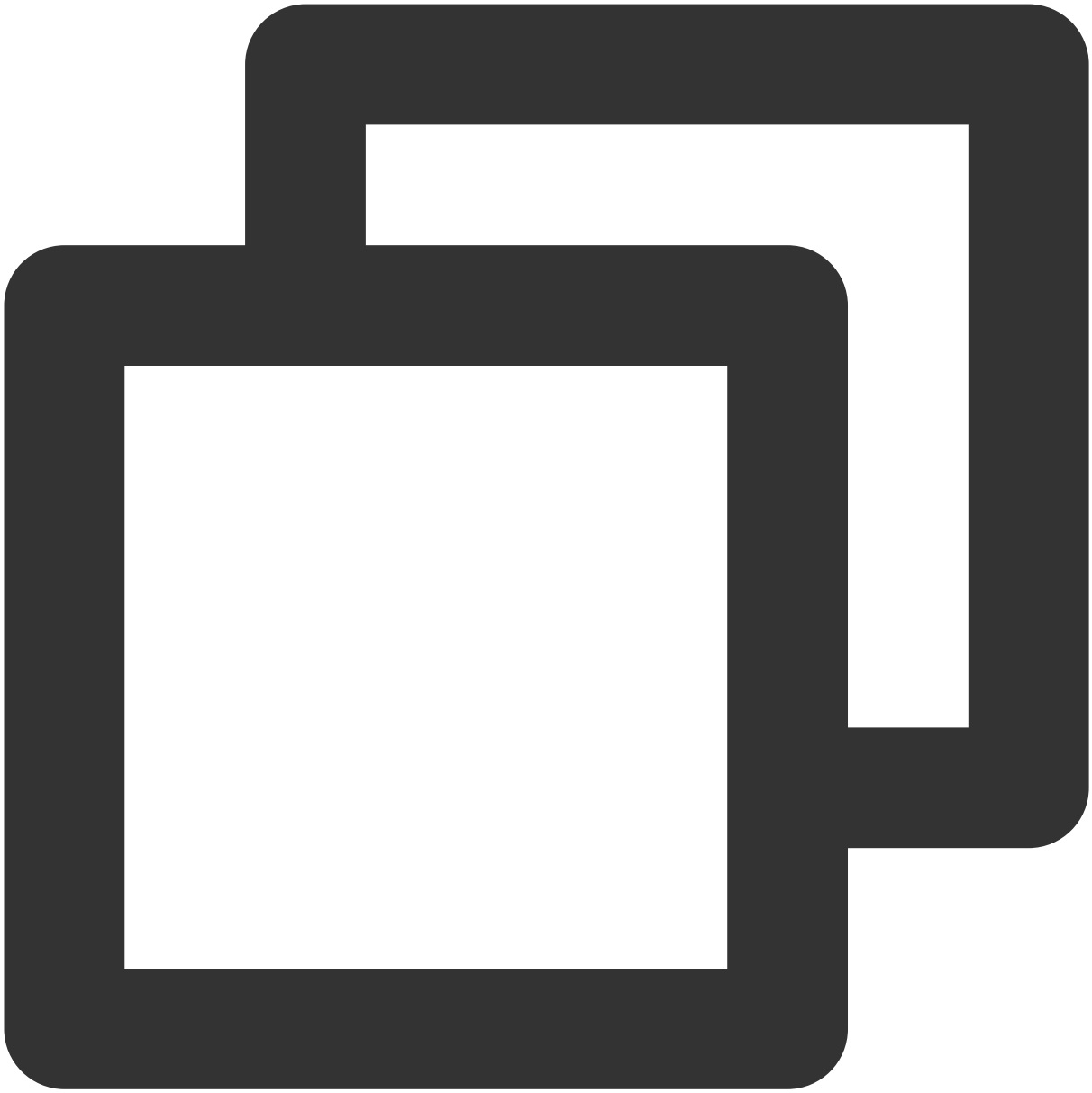
```
# 调用函数处理事件
  RESPONSE=$($(echo "$_HANDLER" | cut -d. -f2) "$EVENT_DATA")
```

1.3 访问运行时 API 响应结果接口，推送函数处理结果。若首次调用成功为事件终态，则 SCF 将进行状态锁定，推送后结果不可变更。示例代码如下：



```
# 推送函数处理结果
  curl -X POST -s "http://$SCF_RUNTIME_API:$SCF_RUNTIME_API_PORT/runtime/invocation
```

如果函数调用阶段出现错误，通过访问运行时 API 调用错误接口推送错误信息。同时本次调用结束，首次调用视为事件终态，SCF 将进行状态锁定，继续推送结果不可变更。示例代码如下：



```
# 推送函数处理错误
  curl -X POST -s "http://$SCF_RUNTIME_API:$SCF_RUNTIME_API_PORT/runtime/invocation
```

调用日志及异常

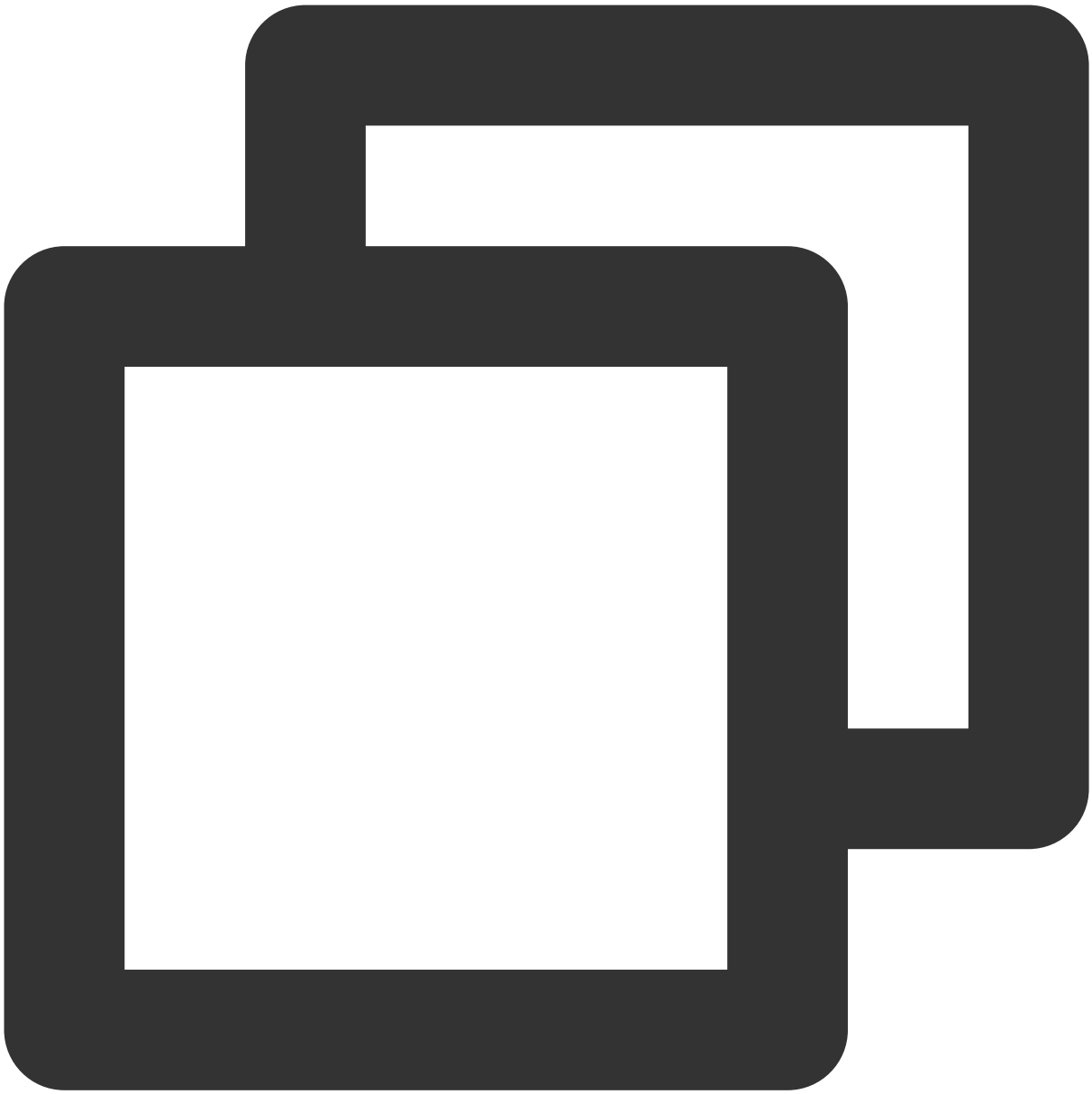
调用阶段日志及异常信息，请参见 日志及异常。

创建函数处理文件

说明

函数处理文件包含函数逻辑的具体实现，执行方式及参数可以通过运行时自定义实现。

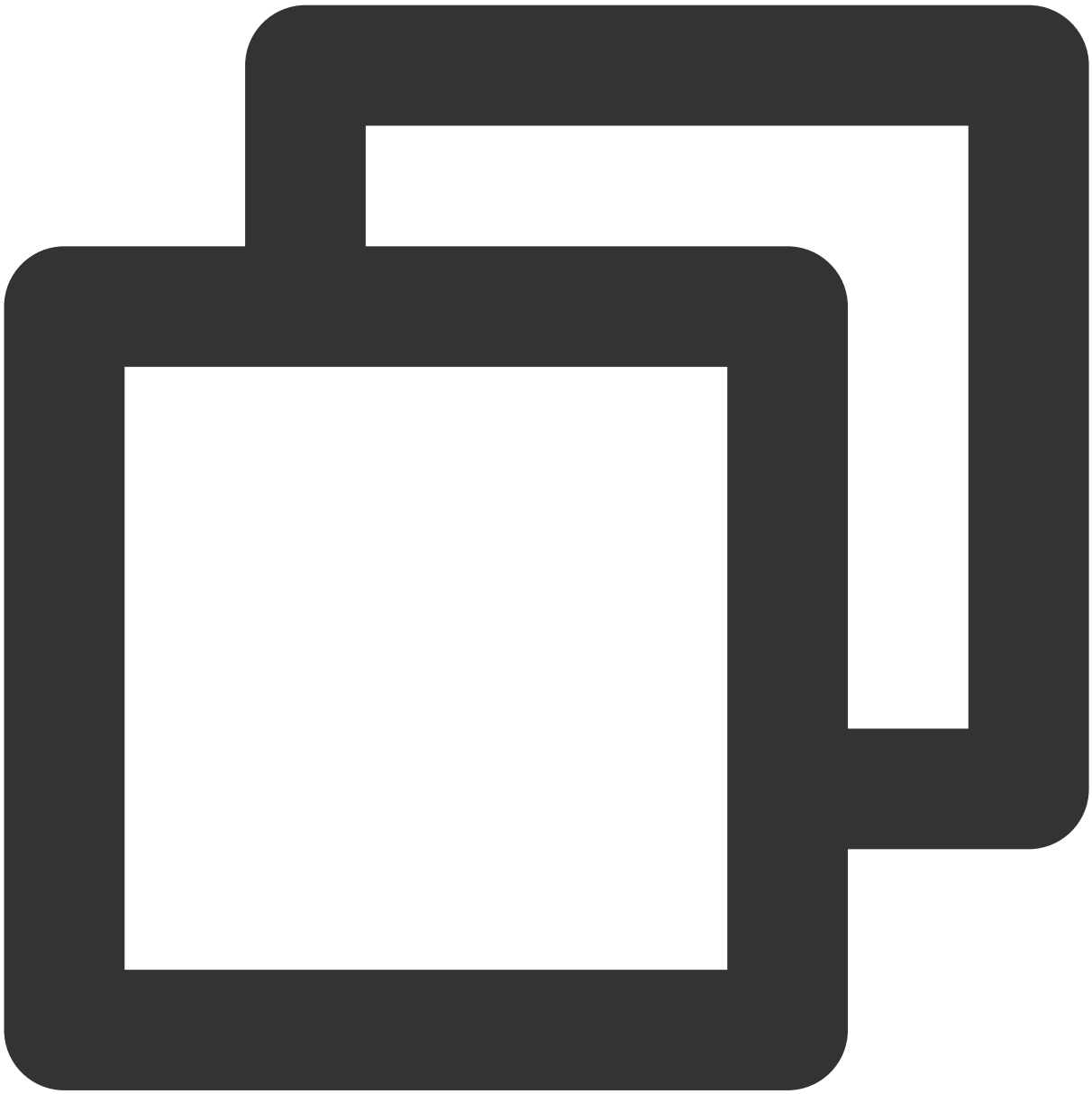
在命令行终端创建 index.sh。



```
function main_handler () {
  EVENT_DATA=$1
  echo "$EVENT_DATA" 1>&2;
  RESPONSE="Echoing request: '$EVENT_DATA'"
  echo $RESPONSE
}
```

发布函数

1. 成功创建 bootstrap 和 函数文件 后，目录结构如下所示：

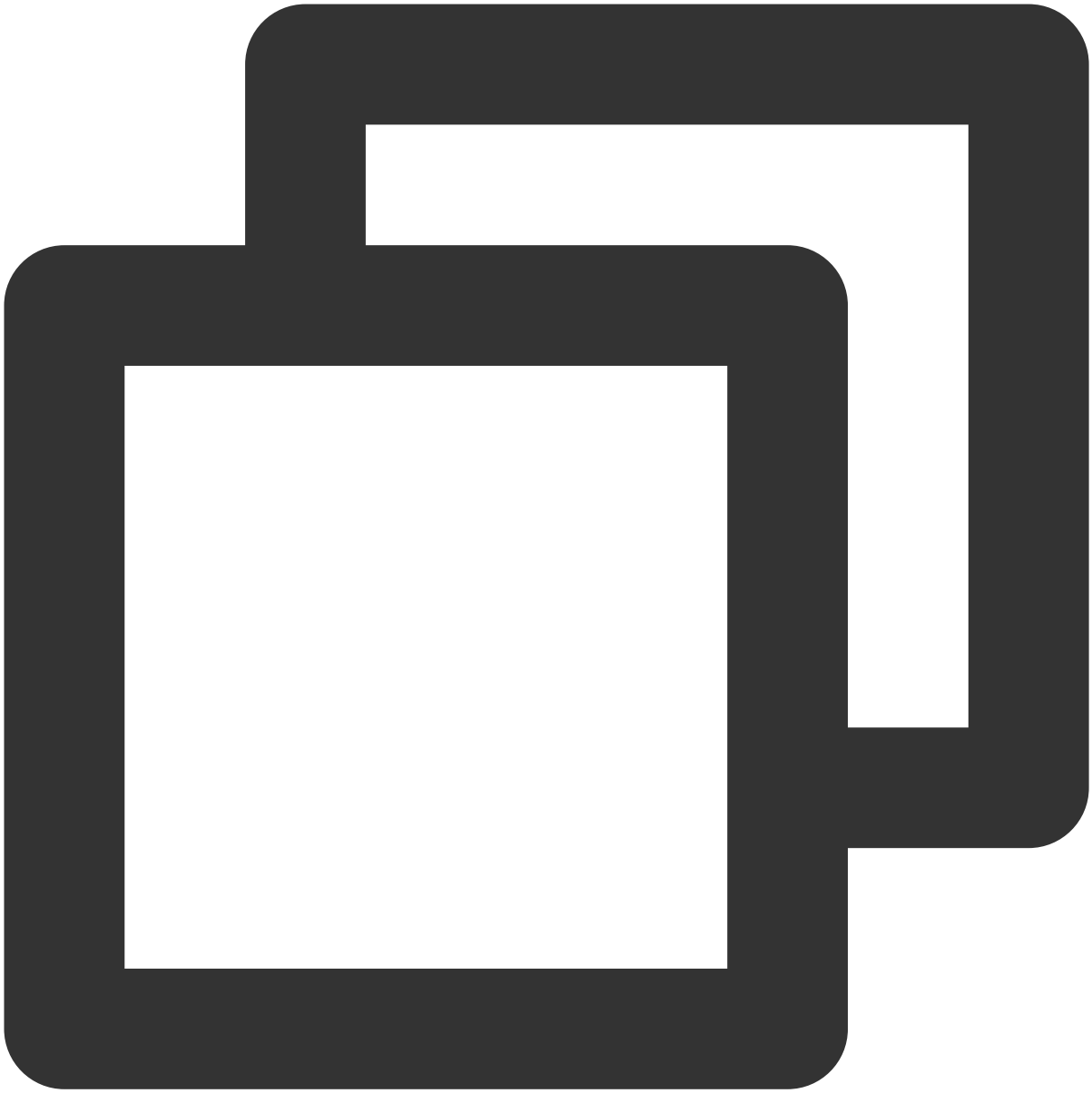


```
├ bootstrap
└ index.sh
```

2. 执行以下命令，设置文件可执行权限：

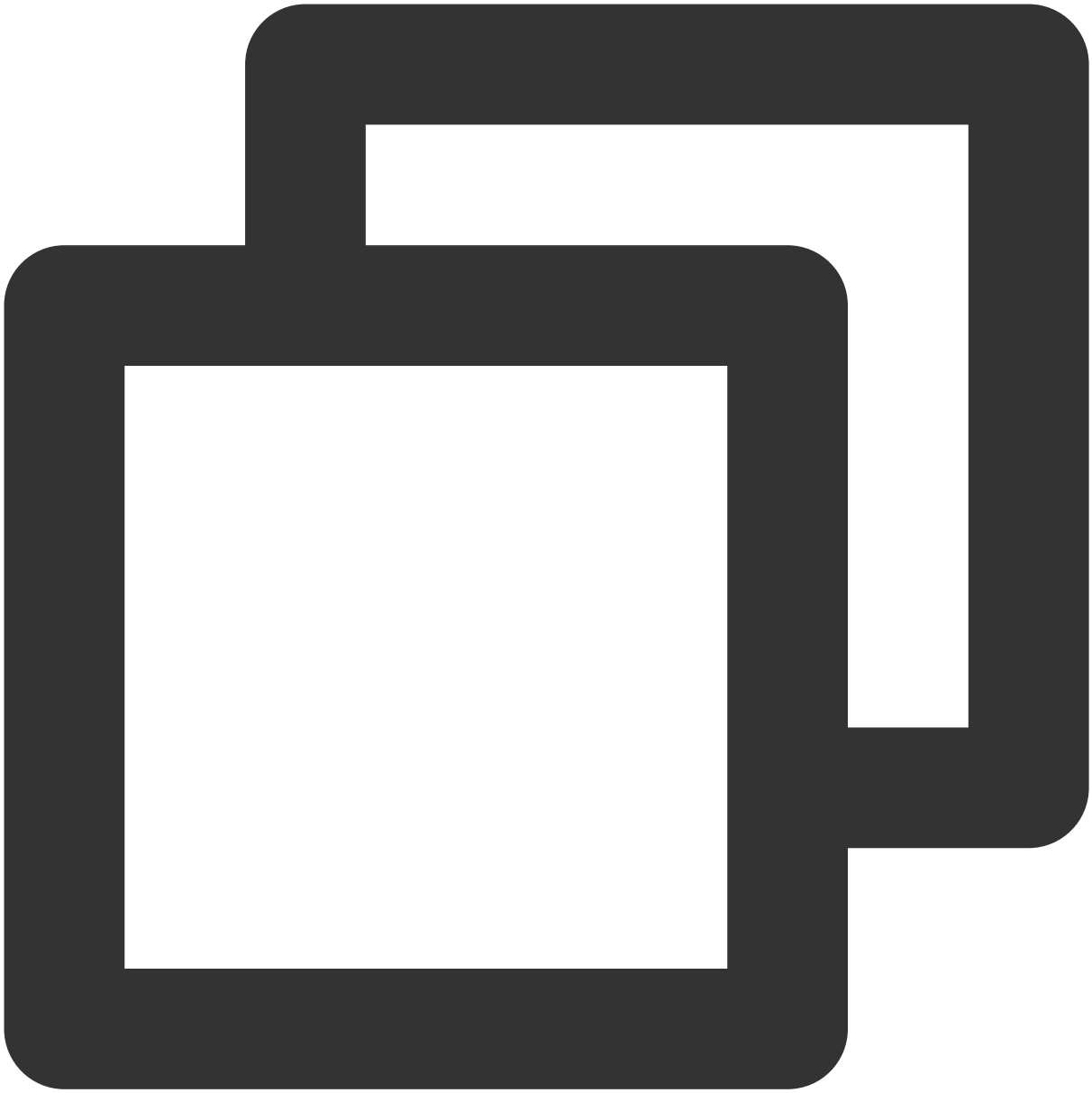
说明

Windows 系统下不支持 `chmod 755` 命令，需要在 Linux 或 Mac OS 系统下执行。



```
$ chmod 755 index.sh bootstrap
```

3. 使用 Serverless Cloud Framework 创建和发布函数。或执行以下命令，打包生成 zip 包，通过 SDK 或 Serverless 控制台 来创建和发布函数。

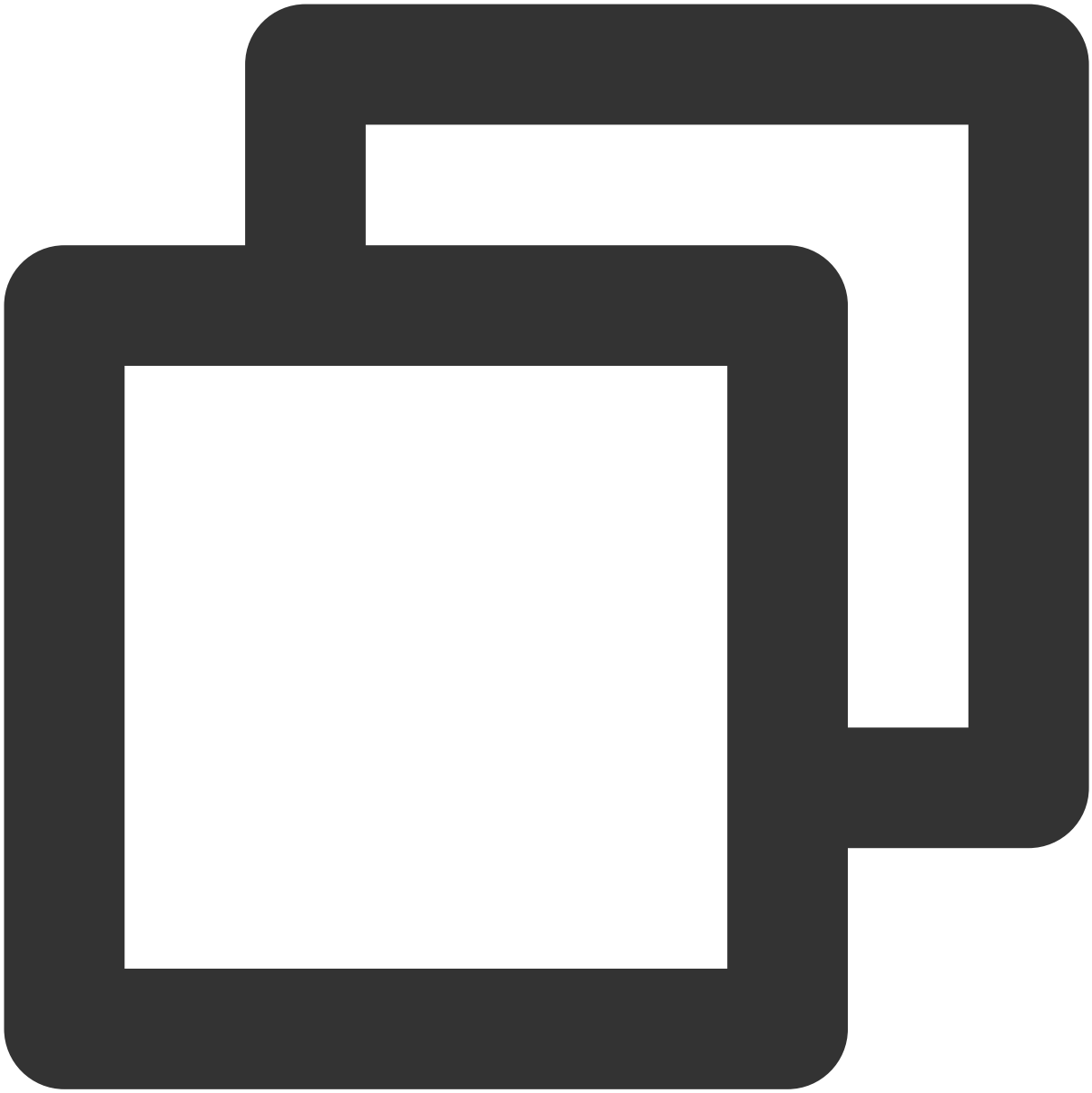


```
$ zip demo.zip index.sh bootstrap
adding: index.sh (deflated 23%)
adding: bootstrap (deflated 46%)
```

使用 Serverless Cloud Framework 创建及发布函数

创建函数

1. 安装 Serverless Cloud Framework。
2. 在 bootstrap 目录下配置 Serverless.yml 文件，创建 dotnet 函数：



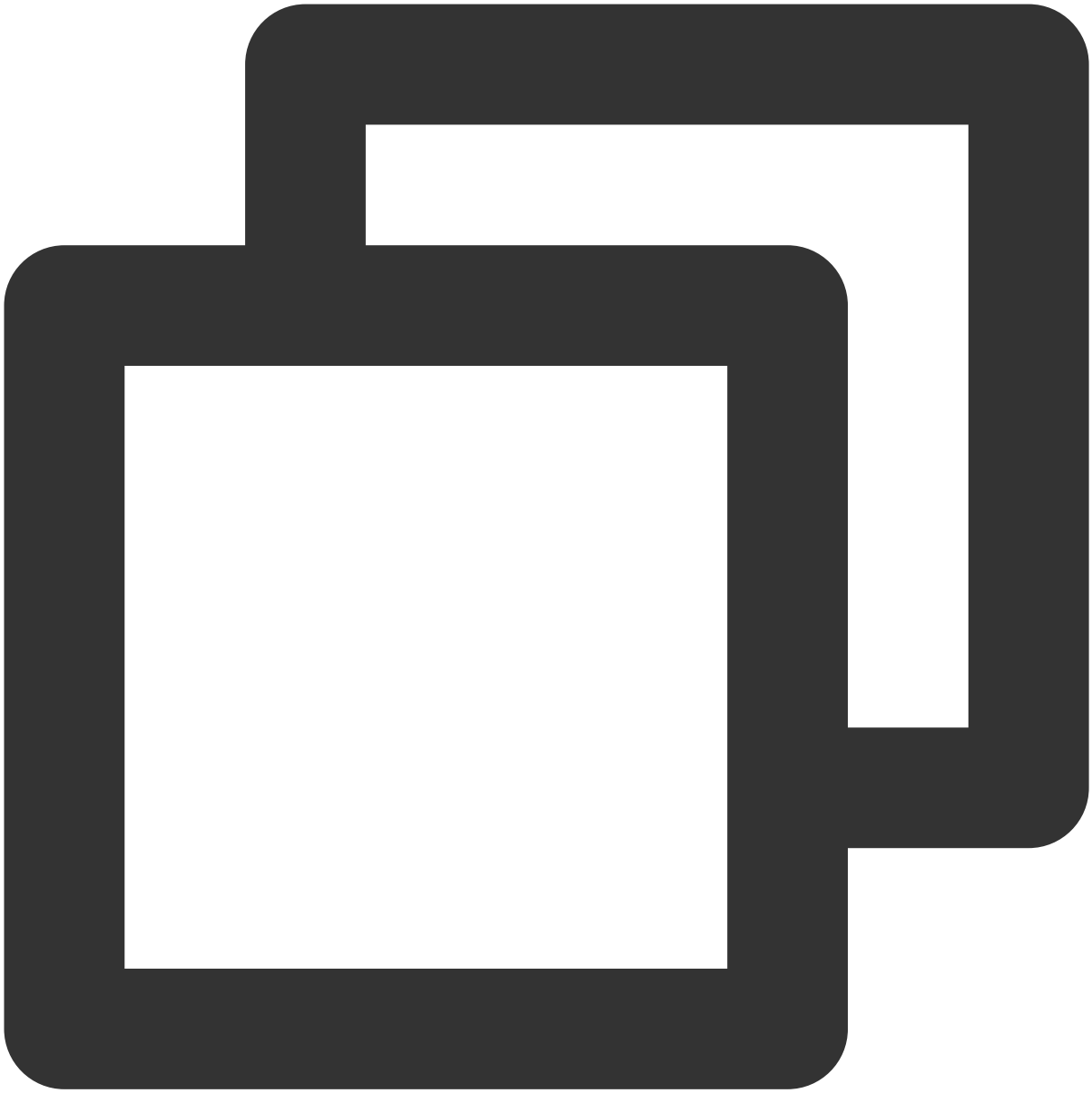
```
#组件信息
component: scf # 组件名称，本例中为scf组件
name: ap-guangzhou_default_helloworld # 实例名称
#组件参数
inputs:
  name: helloworld #函数名称
  src: ./
  description: helloworld blank template function.
  handler: index.main_handler
  runtime: CustomRuntime
  namespace: default
```

```
    region: ap-guangzhou
    memorySize: 128
    timeout: 3
    events:
      - apigw:
          parameters:
            endpoints:
              - path: /
                method: GET
```

说明

SCF 组件的详细配置，请参见 全量配置文档。

3. 执行 `scf deploy` 命令创建云函数，创建成功则返回结果如下：



```
serverless-cloud-framework
Action: "deploy" - Stage: "dev" - App: "ap-guangzhou_default_helloworld" - Instance
functionName: helloworld
description:  helloworld blank template function.
namespace:    default
runtime:      CustomRuntime
handler:      index.main_handler
memorySize:   128
lastVersion:  $LATEST
traffic:      1
triggers:
```

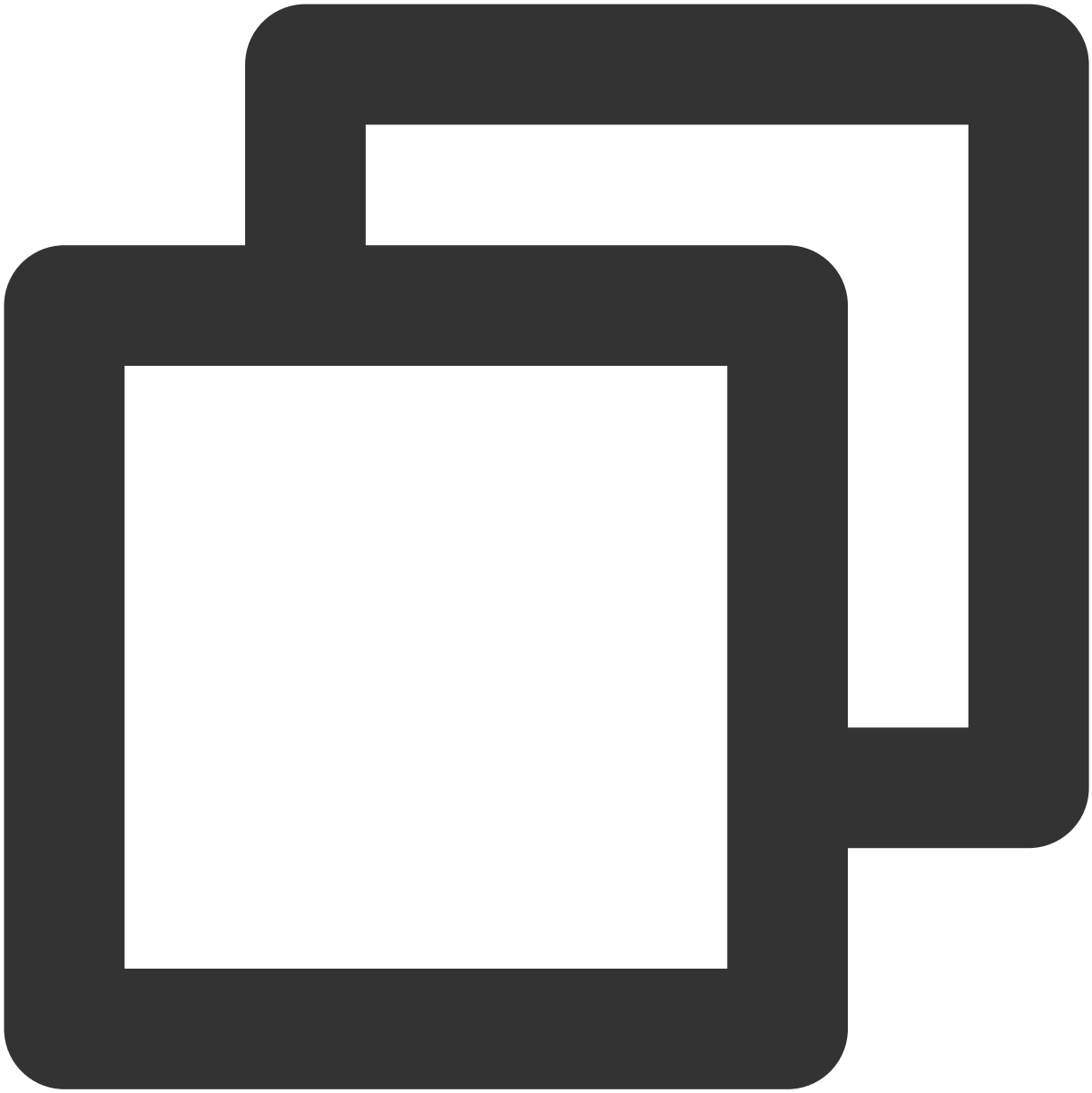
```
  apigw:
    - http://service-xxxxxx-123456789.gz.apigw.tencentcs.com/release/
Full details: https://serverless.cloud.tencent.com/apps/ap-guangzhou_default_hellow
36s › ap-guangzhou_default_helloworld › Success
```

说明

更多 SCF 组件使用，请参见 SCF 组件。

调用函数

由于 serverless.yml 中添加了 `events` 为 `apigw` 的配置，因此创建函数的同时也创建了 api 网关，可通过 api 网关访问云函数。返回类似如下信息，即表示访问成功。



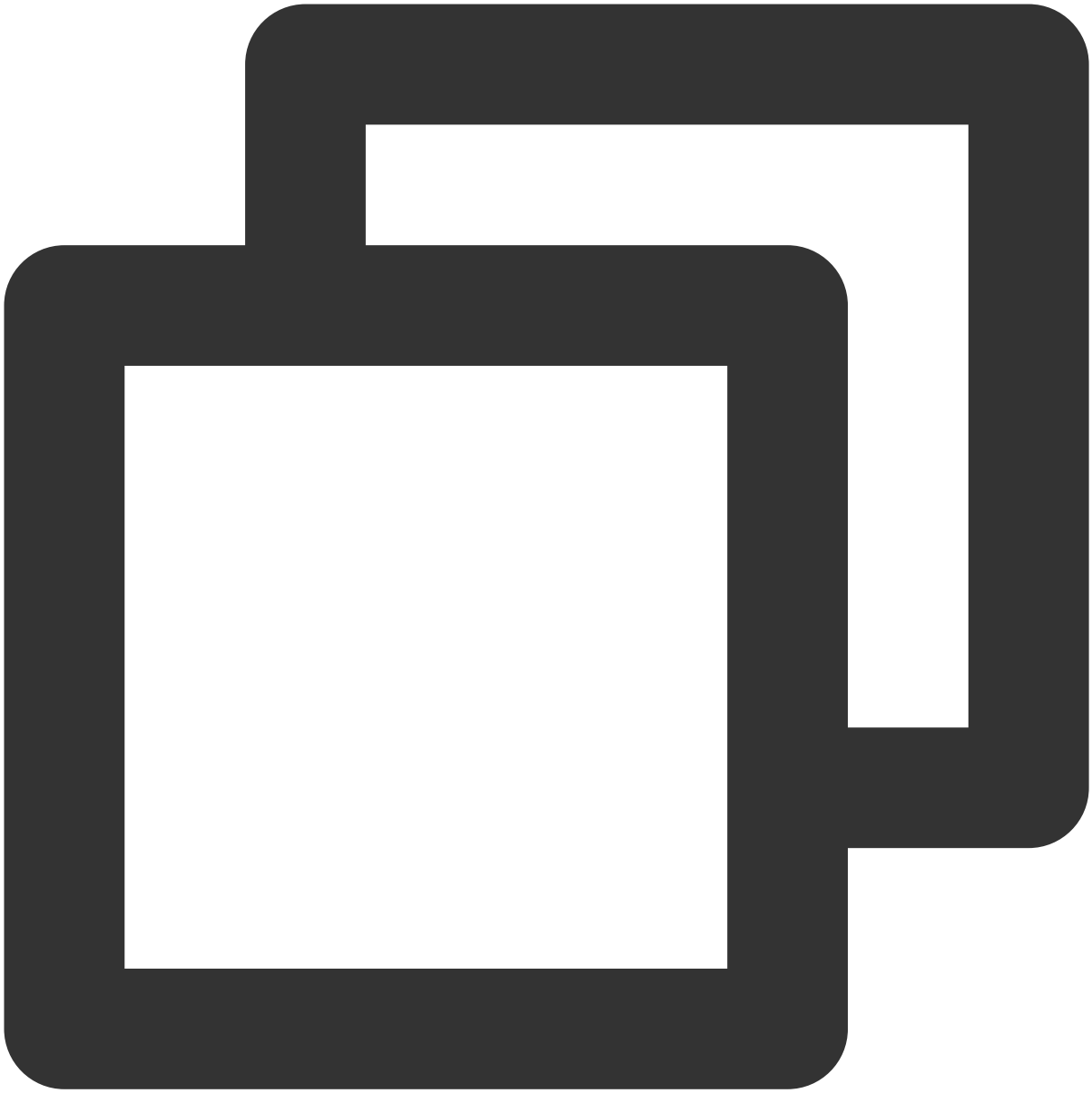
```
Echoing request:
'{
        "headerParameters":{},
        "headers":{
"accept":"text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/we
        "accept-encoding":"gzip, deflate",
        "accept-language":"zh-CN,zh-TW;q=0.9,zh;q=0.8,en-US;q=0.7,en;q=0.6",
        "cache-control":"max-age=259200",
        "connection":"keep-alive",
        "host":"service-eiu4aljg-1259787414.gz.apigw.tencentcs.com",
        "upgrade-insecure-requests":"1",
```

```
        "user-agent":"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
        "x-anonymous-consumer":"true",
        "x-api-requestid":"b8b69e08336bb7f3e06276c8c9******",
        "x-api-scheme":"http",
        "x-b3-traceid":"b8b69e08336bb7f3e06276c8c9******",
        "x-qualifier":"$LATEST"},
        "httpMethod":"GET",
        "path":"/",
        "pathParameters":{},
        "queryString":{},
        "queryStringParameters":{},
        "requestContext":{"httpMethod":"GET","identity":{},"path":"/",
        "serviceId":"service-xxxxx",
        "sourceIp":"10.10.10.1",
        "stage":"release"
        }
}'
```

使用 SDK 创建及发布函数

创建函数

执行以下命令，通过 SCF 的 Python SDK 创建名为 CustomRuntime-Bash 的函数。



```
from tencentcloud.common import credential
from tencentcloud.common.profile.client_profile import ClientProfile
from tencentcloud.common.profile.http_profile import HttpProfile
from tencentcloud.common.exception.tencent_cloud_sdk_exception import TencentCloudS
from tencentcloud.scf.v20180416 import scf_client, models
from base64 import b64encode
try:
    cred = credential.Credential("SecretId", "secretKey")
    httpProfile = HttpProfile()
    httpProfile.endpoint = "scf.tencentcloudapi.com"
```

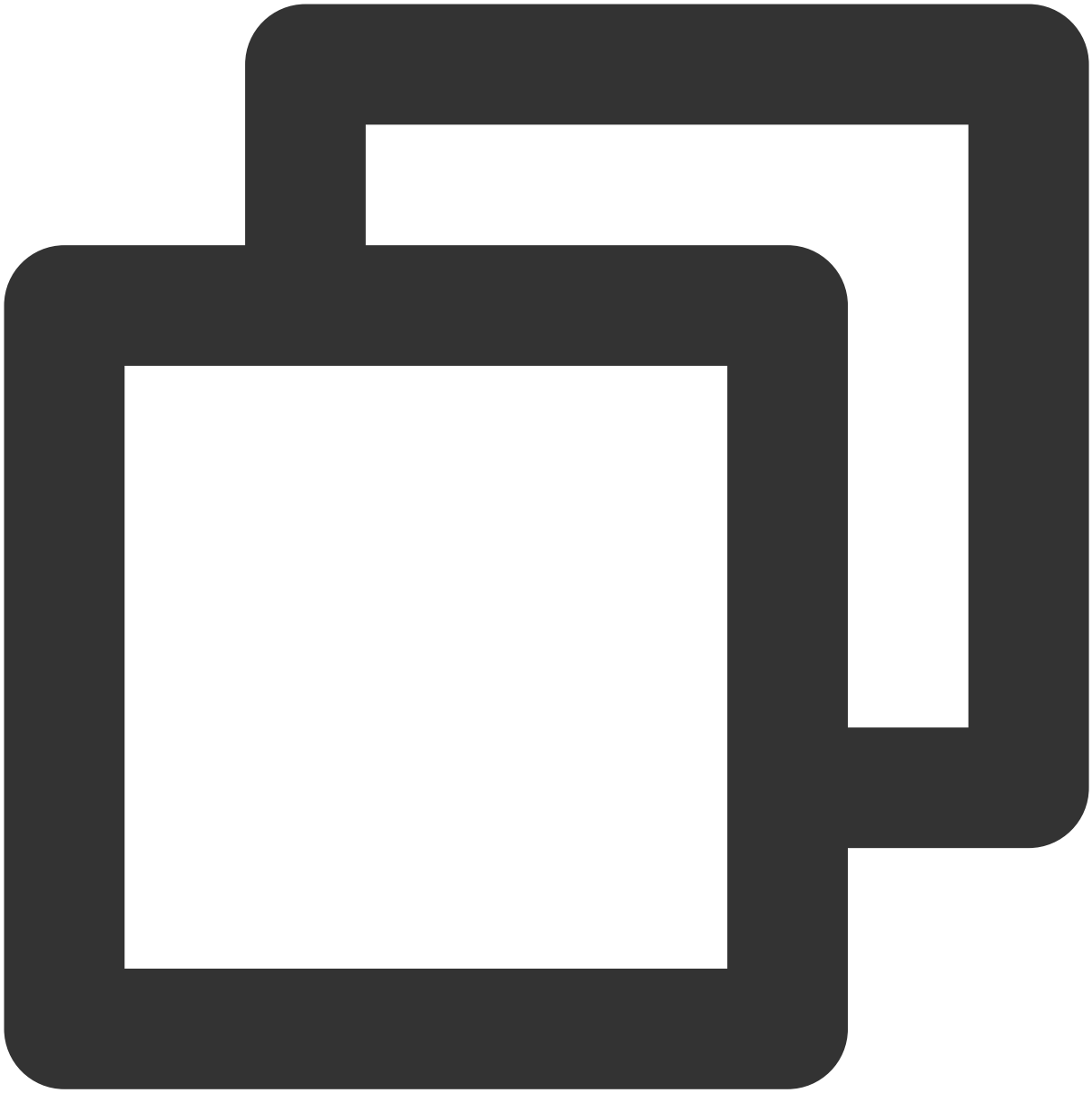
```
    clientProfile = ClientProfile()
    clientProfile.httpProfile = httpProfile
    client = scf_client.ScfClient(cred, "na-toronto", clientProfile)

    req = models.CreateFunctionRequest()
    f = open('demo.zip', 'r')
    code = f.read()
    f.close()

    params = '{\\"FunctionName\\":\\"CustomRuntime-Bash\\",\\"Code\\":{\\"ZipFile\\
    req.from_json_string(params)

    resp = client.CreateFunction(req)
    print(resp.to_json_string())

except TencentCloudSDKException as err:
    print(err)
```

Custom Runtime 特殊参数说明

参数类型	说明
`"Runtime":"CustomRuntime"`	Custom Runtime 对应的 runtime 类型。
`"InitTimeout":3`	初始化超时时间。Custom Runtime 针对初始化阶段新增超时控制配置，时间区间以 bootstrap 启动为始，以上报运行时 API 就绪状态为止。超出后将终止执行并返回初始化超时错误。
`"Timeout":3`	调用超时时间。事件调用的超时控制配置，时间区间以事件下发为始，以函数处理完成推送结果至运行时 API 为止。超出后将终止执行并返回调用超时错误。

调用函数

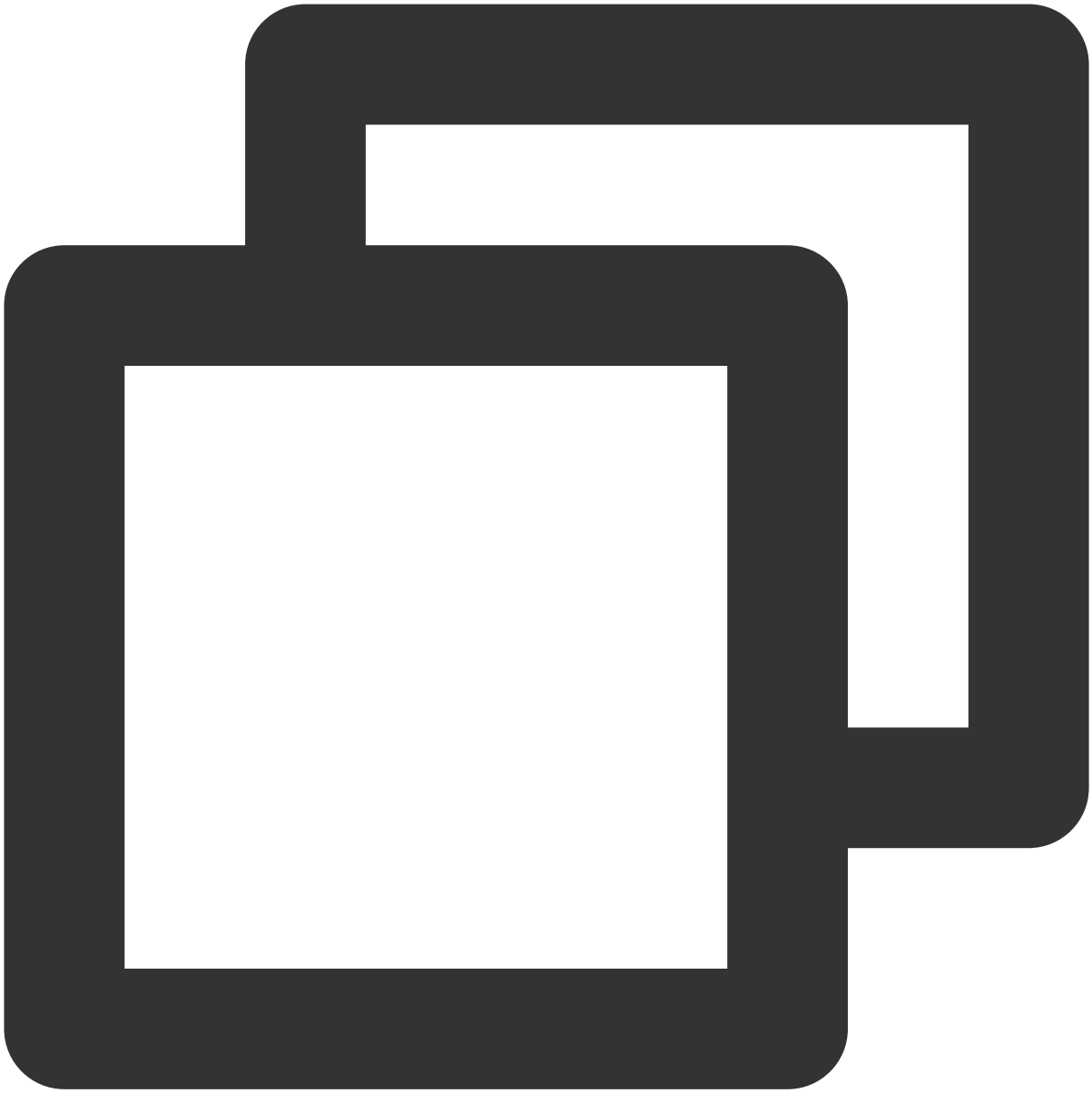
执行以下命令，通过 SCF 的 Python SDK 调用已创建的 CustomRuntime-Bash 函数。



```
from tencentcloud.common import credential
from tencentcloud.common.profile.client_profile import ClientProfile
from tencentcloud.common.profile.http_profile import HttpProfile
from tencentcloud.common.exception.tencent_cloud_sdk_exception import TencentCloudS
from tencentcloud.scf.v20180416 import scf_client, models
try:
    cred = credential.Credential("SecretId", "secretKey")
    httpProfile = HttpProfile()
    httpProfile.endpoint = "scf.tencentcloudapi.com"

    clientProfile = ClientProfile()
```

```
    clientProfile.httpProfile = httpProfile
    client = scf_client.ScfClient(cred, "na-toronto", clientProfile)

    req = models.InvokeRequest()
    params = '{\\"FunctionName\\":\\"CustomRuntime-Bash\\",\\"ClientContext\\":\\"{
    req.from_json_string(params)

    resp = client.Invoke(req)
    print(resp.to_json_string())

except TencentCloudSDKException as err:
    print(err)
```

返回类似如下信息，即表示调用成功。



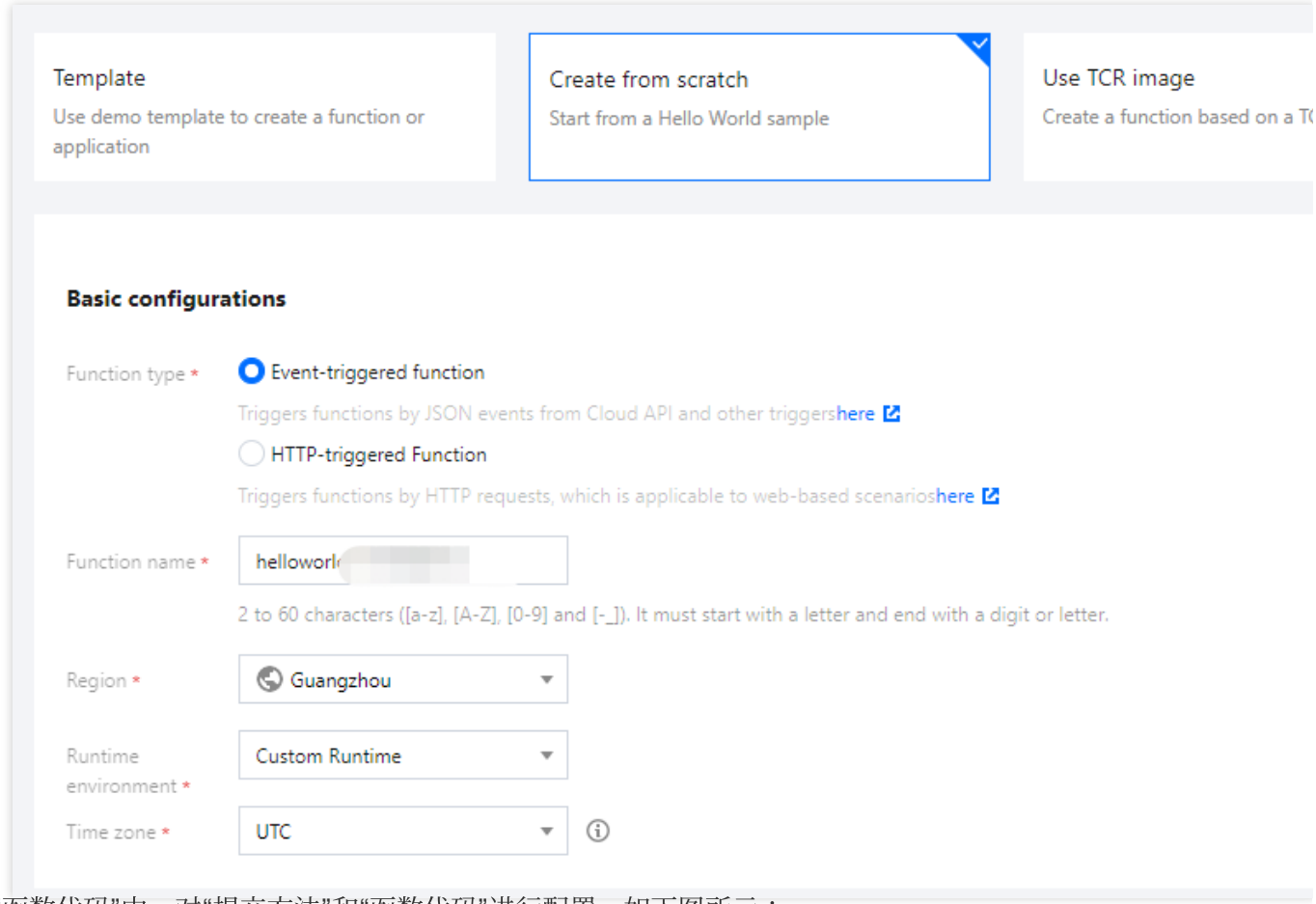

```
{"Result":
    {"MemUsage": 7417***,
    "Log": "", "RetMsg":
    "Echoing request: '{
        \\"key1\\": \\"test value 1\\",
        \\"key2\\": \\"test value 2\\"
        }'",
    "BillDuration": 101,
    "FunctionRequestId": "3c32a636-****-****-****-d43214e161de",
    "Duration": 101,
    "ErrMsg": "",
```

```
    "InvokeResult": 0
    },
    "RequestId": "3c32a636-****-****-****-d43214e161de"
}
```

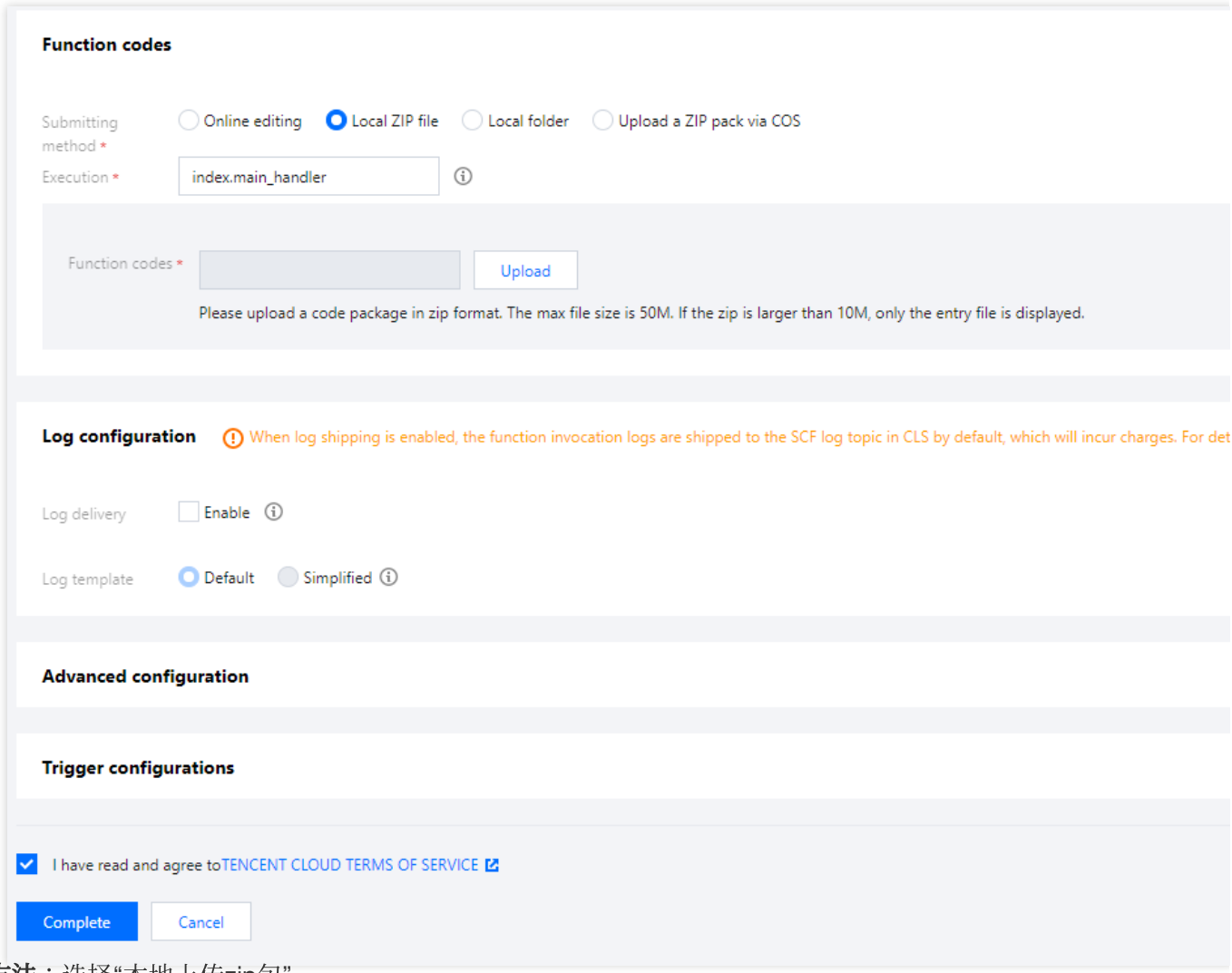
使用控制台创建及发布函数

创建函数

1. 登录 Serverless 控制台，单击左侧导航栏的**函数服务**。
2. 在“函数服务”页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**页面中选择**从头开始**，并在**运行环境**中选择 **Custom Runtime**。如下图所示：



4. 在“函数代码”中，对“提交方法”和“函数代码”进行配置。如下图所示：



提交方法：选择“本地上传zip包”。

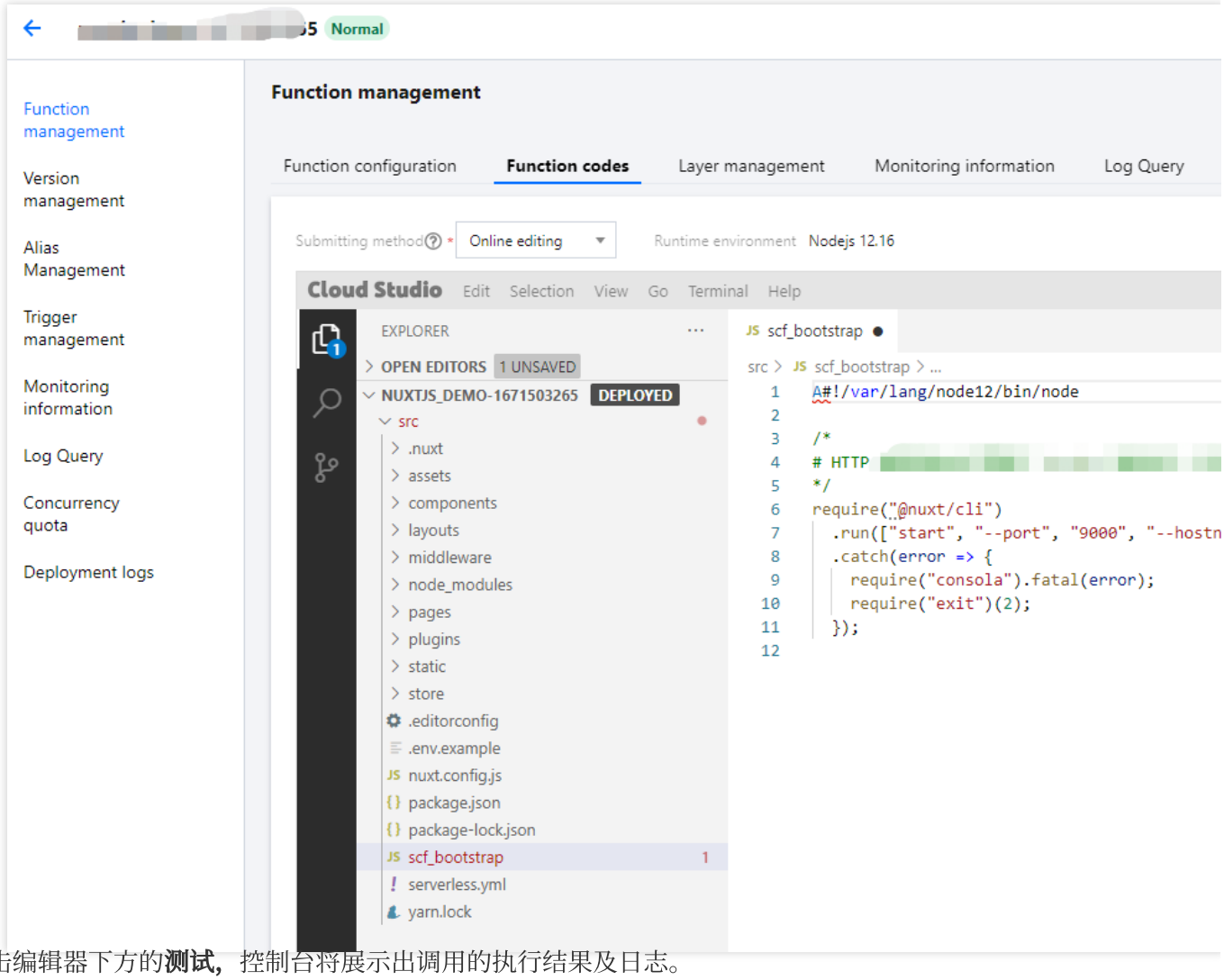
函数代码：选择打包好的 demo.zip。

高级设置：展开配置项，配置“初始化超时时间”及其他相关参数。

5. 单击**完成**即可完成函数创建。

调用函数

1. 登录 Serverless 控制台，单击左侧导航栏的**函数服务**。
2. 在“函数服务”页面上方选择期望调用函数的地域，并单击列表页中期望调用的函数，进入函数详情页面。
3. 选择左侧**函数管理**，并在“函数管理”页面选择**函数代码**页签。如下图所示：



4. 单击编辑器下方的**测试**，控制台将展示出调用的执行结果及日志。

使用镜像部署函数
WebServer 镜像函数

最近更新时间：2024-04-22 18:03:28

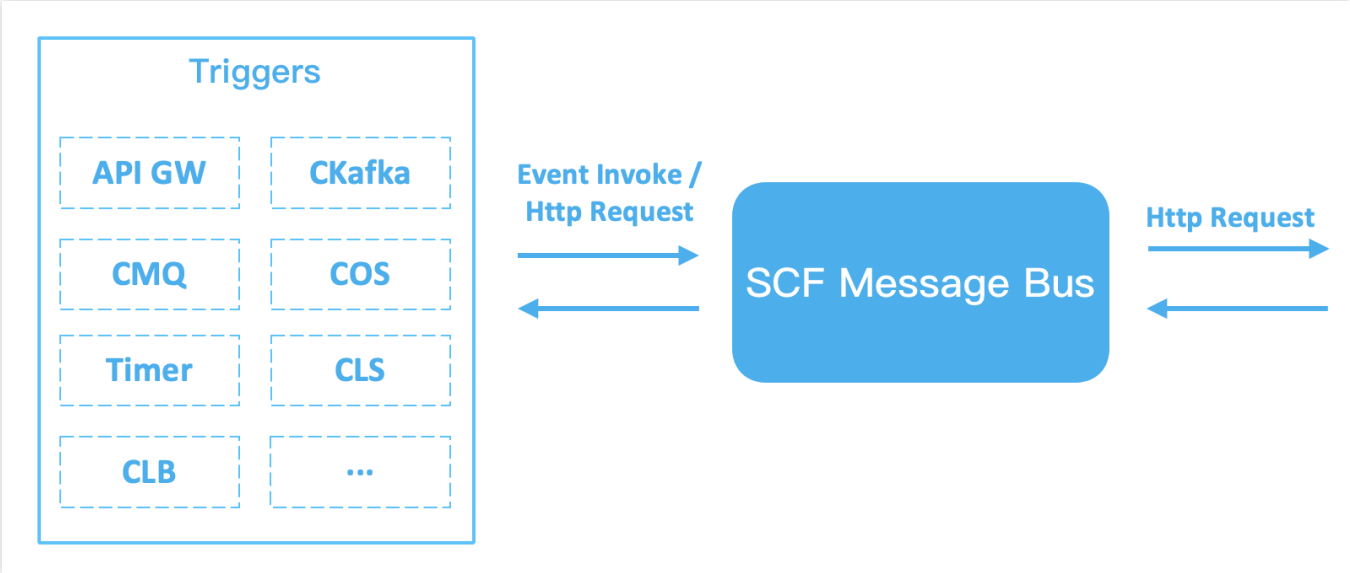
云函数 SCF 支持开发者将容器镜像部署为函数的功能。本文介绍镜像部署函数的背景信息、工作原理、函数开发、函数日志打印、冷启动优化、计费说明及使用限制。

背景信息

云函数 SCF 从设计开始即是基于云原生架构的 FaaS 产品。在 Runtime 层支持容器镜像部署为函数后，产品形态整体向容器化生态迈进。一方面，解决函数运行时的环境依赖问题，给予用户更大的自由发挥空间。另一方面，产品形态层面的呈现使得用户无需受困于 Kubernetes 集群管理、安全维护、故障诊断等技术门槛，将弹性伸缩、可用性等需求下沉至计算平台，进一步释放云计算能力。

工作原理

您在开发函数具体的逻辑之前，首先需要确认函数类型，云函数提供事件函数和 Web 函数两种类型。
云函数在函数实例初始化阶段，获得镜像仓库的临时用户名和密码作为访问凭证来拉取镜像。镜像拉取成功后，根据指定您所定义的启动命令 `Command` 、参数 `Args` 及端口（固定为9000）启动您定义的 HTTP Server。最后，HTTP Server 将接收云函数的所有入口请求，包括来自您的事件函数调用及 Web 函数调用。
函数工作原理如下图所示：



基于镜像部署的函数开发

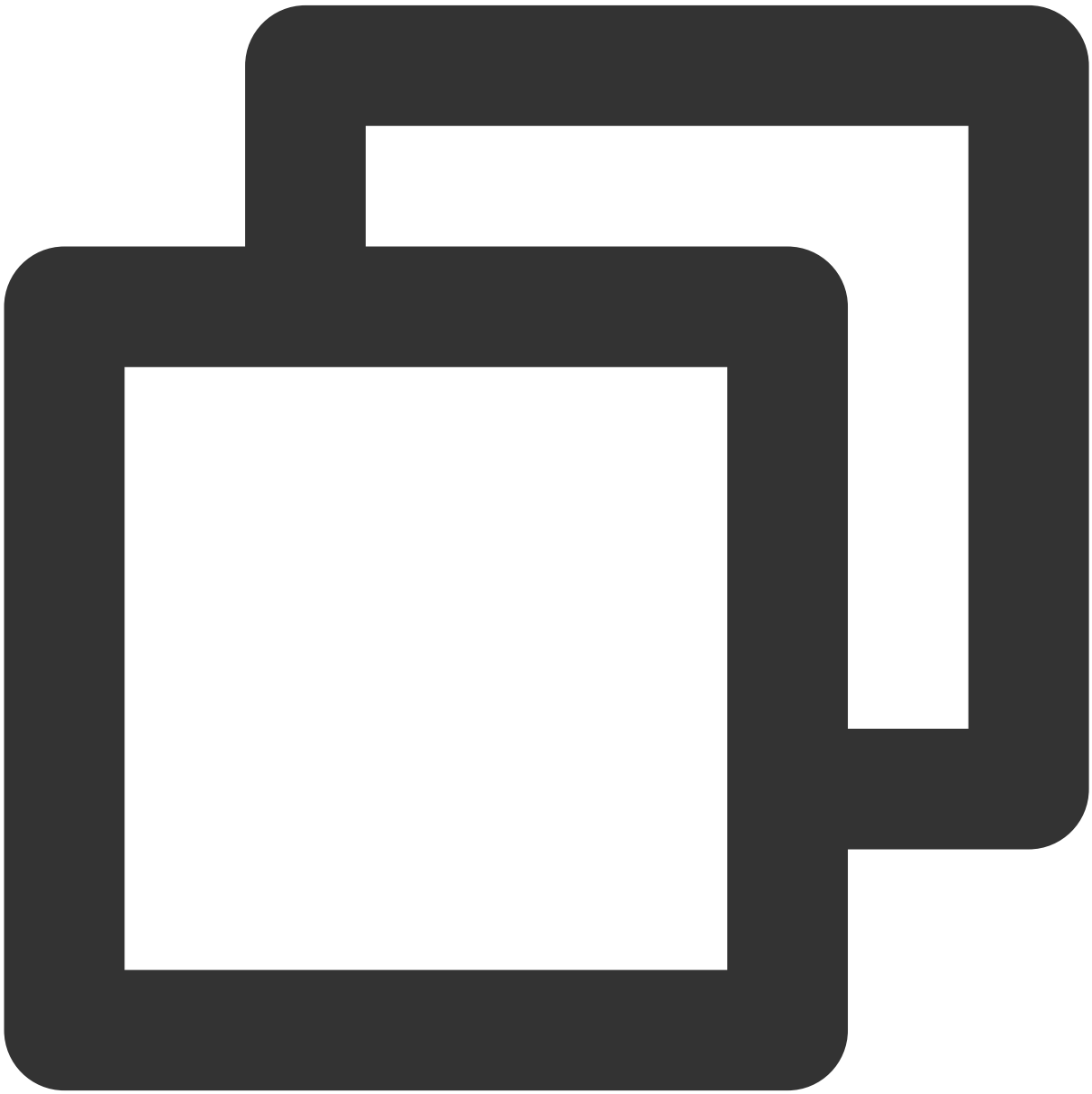
HTTP Server 搭建

基于镜像部署的函数，需要搭建 HTTP Server，配置要求如下：

监听 `0.0.0.0:9000` 或 `*:9000` 。

HTTP Server 需在30秒内启动完毕。

如未完成，则可能会导致健康检查超时，出现以下错误：



```
The request timed out in 30000ms.Please confirm your http server have enabled liste
```

函数入参

event：POST 请求体（HTTP Body）

请求体包含事件数据，结构请参见 触发器事件消息结构汇总。

context：请求头（HTTP Header）

公共参数，用于标识用户和接口签名的参数，每次请求中均需携带。

通过 X-Scf-Request-Id 获取当前请求 ID。

说明：

事件函数和 Web 函数均包含 Common Headers。

公共请求头由云函数生成，主要包含权限、函数基本信息等。

详细列表如下表所示：

Header 字段	描述
X-Scf-Request-Id	当前请求ID
X-Scf-Memory	函数实例运行时可使用的最大内存
X-Scf-Timeout	函数执行的超时时间
X-Scf-Version	函数版本
X-Scf-Name	函数名称
X-Scf-Namespace	函数所在命名空间
X-Scf-Region	函数所在地域
X-Scf-Appid	函数所有者的Appid
X-Scf-Uin	函数所有者的Uin
X-Scf-Session-Token	临时 SESSION TOKEN
X-Scf-Secret-Id	临时 SECRET ID
X-Scf-Secret-Key	临时 SECRET KEY
X-Scf-Trigger-Src	Timer（使用定时触发器时）

内置环境变量

自定义镜像场景相较于基于代码包部署，在容器内置的环境变量做了变更，您可以根据实际需要进行引用。

环境变量 Key	具体值或值来源
TENCENTCLOUD_RUNENV	SCF
USER_CODE_ROOT	/var/user/
USER	qcloud
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

函数调用

事件函数，用户需要监听固定路径 `/event-invoke` 来接受函数调用请求。
Web 函数，用户无需监听指定路径，API 网关将会以7层反向代理的方式透传请求路径。

函数日志打印

云函数 SCF 将以无侵入的方式，收集在容器内所产生的 stdout、stderr 等标准输出日志，并上报至日志模块。您在调用函数后，可以通过控制台查看日志聚合展示效果。

冷启动优化

镜像由于增加了基础环境、系统依赖等文件层，相较于基于代码包部署的完全内置，存在额外的文件下载和镜像解压的时间。为了进一步降低冷启动时间，推荐您使用以下策略。

优化镜像体积

在同一地域创建镜像仓库与函数。在函数触发镜像拉取时通过 VPC 网络进行拉取，以此获得更快更稳定的镜像拉取效率。
镜像制作秉承最小化原则，即仅包含必要基础环境、运行依赖，去除不必要的文件等。

开启镜像加速

通过开启镜像加速开关，函数平台会通过内部加速机制，预先将镜像就近缓存，在调用函数实例时，直接从缓存进行加载和解压，省去了镜像文件下载的时间。可获得平均 5 倍的启动速度提升。

操作步骤

1. 登录云函数控制台，选择左侧导航栏中的 函数服务。
2. 在“函数服务”列表页面，选择需进行配置的镜像函数名。
3. 在“函数管理”页面中，选择**函数代码**。
4. 在“函数代码”页面中，单击**编辑**，进入编辑模式。
5. 在“镜像加速”中，勾选“启用”，开启镜像加速模式。
6. 单击**保存**完成配置。

注意事项

该功能目前属于公测免费期，每个账户在单个地域下限制为最多开启5个函数加速。

使用预置并发

镜像部署搭配预置并发功能，预先启动函数实例，达到最优降低冷启动的体验，详情请参见 预置并发。

计费说明

使用镜像部署函数的计费项与使用代码包部署的计费项完全一致。计费详情请参见 计费方式。

使用限制

镜像大小

目前仅支持镜像（解压前）小于1Gi，如有特殊需要，可 提交工单 申请。建议您根据镜像的大小来选择适合的函数实例执行内存。

镜像大小（X）	执行内存（Y）
X < 256MB	256MB < Y < 512MB
256MB < X <512MB	512MB < Y < 1Gi
512MB < X < 1Gi	Y>1Gi

镜像仓库访问

仅支持腾讯云容器镜像服务企业版和个人版，详情可参见 容器镜像服务。
容器镜像服务企业版镜像仓库详情可参见 镜像仓库基本操作。
容器镜像服务个人版镜像仓库详情可参见 开通镜像仓库。
仅支持同地域（Region）下私有镜像仓库的镜像读取。

容器内文件读写权限

默认 `/tmp` 可读可写，建议输出文件时选择 `/tmp` 。
避免使用其他用户的存在限制访问或执行的文件。
容器内文件可写层存储空间限制为512M。

构建镜像的客户端 CPU 架构限制

云函数当前是基于 X86 架构运行的，所以暂不支持运行在 ARM 平台上构建的镜像。ARM 的平台典型如 Apple Mac 搭载 M1 芯片的 PC 端。

构建镜像的客户端限制

满足其中之一即可：
Docker image manifest V2，schema 2 (使用 Docker version 1.10或者更新版本)
Open Container Initiative (OCI) Specifications (v1.0.0及以上)

使用方法

最近更新时间：2024-04-22 18:03:28

本文介绍如何通过控制台使用镜像来部署函数。

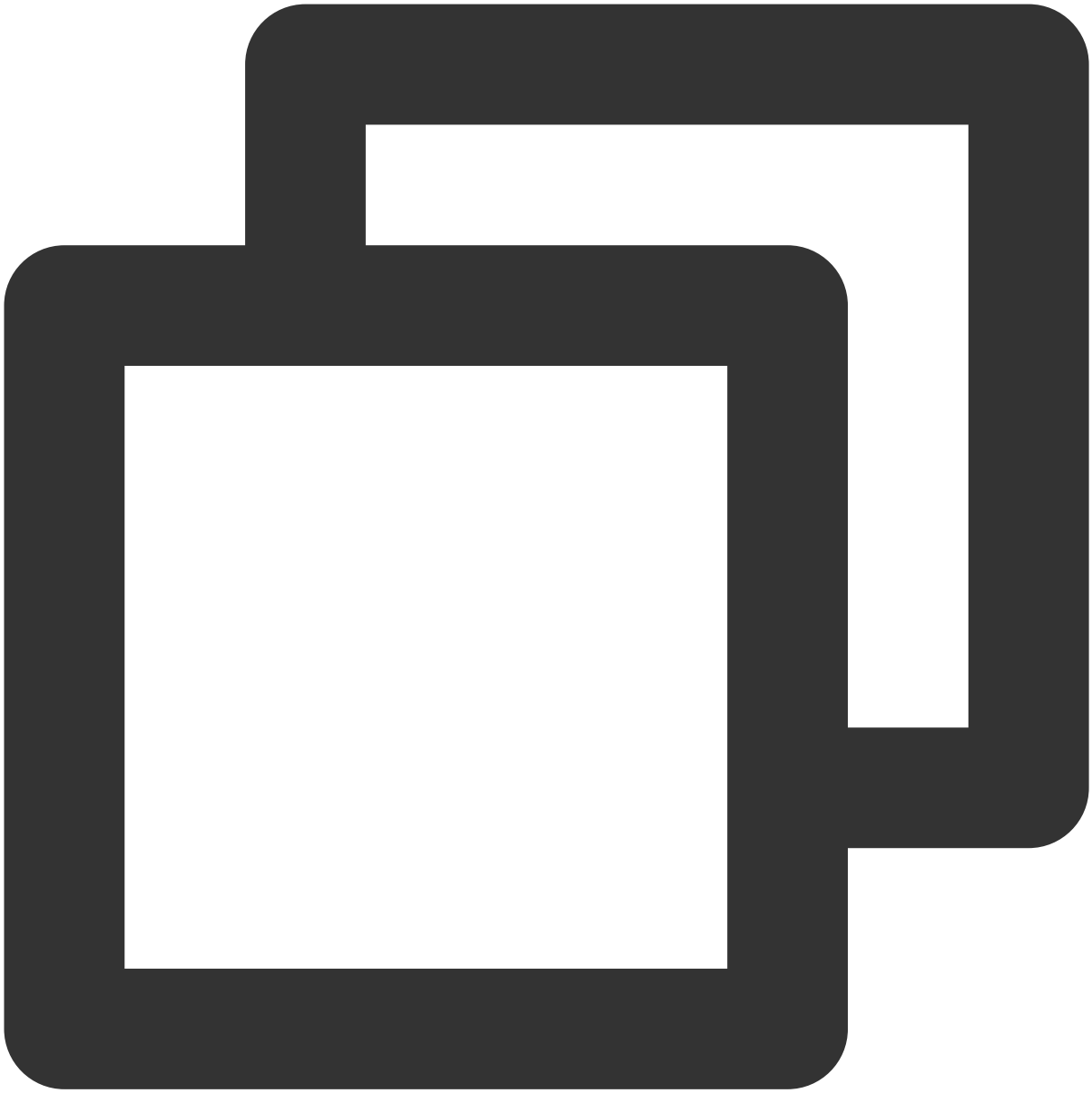
前提条件

云函数 SCF 支持容器镜像服务企业版和个人版的镜像仓库，您可以根据自身的实际需求进行镜像仓库选型。
购买容器镜像服务企业版实例，详情可参见 快速入门。
使用容器镜像服务个人版镜像仓库，详情可参见 快速入门。

使用控制台创建函数

镜像推送

执行以下代码，将构建完成的镜像推送到您的镜像仓库。



```
# 切换到文件下载目录
cd /opt

# 下载 Demo
git clone https://github.com/awesome-scf/scf-custom-container-code-snippet.git

# 登录镜像仓库，$YOUR_REGISTRY_URL请替换为您所使用的镜像仓库，$USERNAME、$PASSWORD分别替换为
docker login $YOUR_REGISTRY_URL --username $USERNAME --password $PASSWORD

# 镜像构建，$YOUR_IMAGE_NAME 请替换为您所使用的镜像地址
docker build -t $YOUR_IMAGE_NAME .
```

```
# 镜像推送
docker push  $YOUR_IMAGE_NAME
```

创建函数

1. 登录 Serverless 控制台，单击左侧导航栏的**函数服务**。
2. 在主界面上方选择地域和命名空间，并单击**新建**，进入函数创建流程。
3. 选择**使用容器镜像**来新建函数，并填写函数基础配置。

参数	操作
函数类型	选择事件函数或者 Web 函数。
函数名称	定义函数名称。
地域	选择函数部署的地域，请务必于镜像仓库处于同一地域。
时区	云函数内默认使用 UTC 时间，您可以通过配置环境变量 TZ 修改。在您选择时区后，将自动添加对应时区的 TZ 环境变量。
镜像	选择您所创建的个人版或者企业版的镜像仓库。
镜像版本（Tag）	选择镜像的版本如果为空，将默认使用镜像的 latest 版本。
Entrypoint	填写容器的启动命令。参数书写规范，填写可运行的指令，例如 python。该参数为可选参数，如果不填写，则默认使用 Dockerfile 中的 Entrypoint。
CMD	填写容器的启动参数。参数书写规范，以“空格”作为参数的分割标识，例如 -u app.py。该参数为可选参数，如果不填写，则默认使用 Dockerfile 中的 CMD。
镜像加速	选择是否开启镜像加速。开启加速后，云函数将较大程度减少拉取镜像的耗时。开启过程需要 30 秒以上时间，请耐心等待。

4. 单击**完成**，即可创建函数。