03 开发指南

文档版本 01

发布日期 2025-02-17

华为云计算技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为云计算技术有限公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为云计算技术有限公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

目 录

1 概述

1.1 函数开发简介

运行时概述

在函数工作流服务中创建函数时，需选定所需的运行时（Runtime）。函数工作流服务通过支持多种编程语言的运行时，为相应语言提供执行环境，用户可以使用函数工作流服务提供的运行时或自行构建定制运行时。

函数支持的运行时语言

FunctionGraph函数Runtime支持多种运行时语言：Python 、Node.js、Java、Go、C#、PHP、Cangjie及自定义运行时，说明如表1-1所示。

说明

建议使用相关语言的最新版本。

表 1-1 运行时说明

运行时语言	支持版本	SDK下载
Node.js	6.10、8.10、10.16、12.13、14.18、16.17、18.15	-
Python	2.7、3.6、3.9、3.10	-
Java	8、11	Java SDK下载（软件包检验文件：fss-java-sdk_sha256） **说明** Java SDK集成了云服务OBS SDK。
Go	1.x	Go1.x SDK（软件包检验文件：Go SDK_sha256）

运行时语言	支持版本	SDK下载
C#	.NET Core 2.1、.NET Core 3.1	CsharpSDK（软件包检验文件：fssCsharp_sha256）
PHP	7.3	-
定制运行时	-	-
Cangjie	1.0	-

Node.js Runtime 集成的三方件

表 1-2 Node.js Runtime 集成的三方件

名称	功能	版本号
q	异步方法封装	1.5.1
co	异步流程控制	4.6.0
lodash	常用工具方法库	4.17.10
esdk-obs-nodejs	OBS SDK	2.1.5
express	极简web开发框架	4.16.4
fgs-express	在FunctionGraph和API Gateway之上使用现有的Node.js应用程序框架运行无服务器应用程序和REST API 。提供的示例允许您使用Express框架轻松构建无服务器Web应用程序/服务和RESTful API 。	1.0.1
request	简化http调用，支持HTTPS并默认遵循重定向	2.88.0

Python Runtime 集成的非标准库

表 1-3 Python Runtime 集成的非标准库

模块	功能	版本号
dateutil	日期/时间处理	2.6.0
requests	http库	2.7.0
httplib2	httpclient	0.10.3
numpy	数学计算	1.13.1

模块	功能	版本号
redis	redis客户端	2.10.5
obsclient	OBS客户端	-
smnsdk	访问SMN服务	1.0.1

函数样例工程包下载

本手册使用样例工程包下载地址如表1-4所示，可以下载到本地，创建函数时上传使用。

表 1-4 样例工程包下载

函数	工程包下载	软件包校验文件
Node.js函数	fss_examples_nodejs.zip	fss_examples_nodejs.sha256
Python函数	fss_examples_python2.7.zip	fss_examples_python2.7_sha256
Java函数	fss_example_java8.jar	fss_example_java8_sha256
Go函数	fss_examples_go1.8.zip	fss_examples_go1.8_sha256
C#函数	fss_example_csharp2.0、fss_example_csharp2.1	fss_example_csharp2.0_sha256 fss_example_csharp2.1_sha256
PHP函数	fss_examples_php7.3.zip	fss_examples_php7.3_sha256

1.2 函数支持的事件源

本节列出了FunctionGraph函数支持的云服务，可以将这些服务配置为FunctionGraph函数的事件源。在预配置事件源映射后，这些事件源检测事件时将自动调用FunctionGraph函数。

消息通知服务 SMN

可以编写FunctionGraph函数来处理SMN的通知，在将消息发布到SMN主题时，服务可以通过将消息负载作为参数传递（SMN示例事件）来调用FunctionGraph函数，FunctionGraph函数代码可以处理事件，比如将消息发布到其他SMN主题或将消息发送到其他云服务。SMN消息触发的使用过程请参考使用SMN触发器。

API Gateway

可以通过HTTPS调用FunctionGraph函数，使用API Gateway自定义REST API和终端节点来实现。可以将各个API操作（如GET和PUT）映射到特定的FunctionGraph函数，当向该API终端节点发送HTTPS请求时（**APIG示例事件**），API Gateway会调用相应的FunctionGraph函数。HTTPS调用触发函数的使用过程请参考**使用APIG触发器**。

数据接入服务 DIS

可以将FunctionGraph函数配置为自动轮询流并处理任何新记录，例如网站点击流、财务交易记录、社交媒体源、IT日志和位置跟踪事件等（**DIS示例事件**）。FunctionGraph会定期轮询DIS数据流中的新记录。DIS数据流触发函数的过程请参考**使用DIS触发器**。

定时触发器 TIMER

可以使用TIMER的计划事件功能定期调用您的代码，可以指定固定频率（分钟、小时、天数）或指定 cron 表达式定期调用函数（**TIMER示例事件**）。定时触发器的使用请参考**使用定时触发器**。

日志触发器 LTS

可以编写FunctionGraph函数来处理云日志服务订阅的日志，当云日志服务采集到订阅的日志后，可以通过将采集到的日志作为参数传递（**LTS示例事件**）来调用FunctionGraph函数，FunctionGraph函数代码可以对其进行自定义处理、分析或将其加载到其他系统。LTS日志触发的使用过程请参考**使用LTS触发器**。

云审计服务触发器 CTS

可以编写FunctionGraph函数，根据CTS云审计服务类型和操作订阅所需要的事件通知，当CTS云审计服务获取已订阅的操作记录后，通过CTS触发器将采集到的操作记录作为参数传递（**CTS示例事件**）来调用FunctionGraph函数。经由函数对日志中的关键信息进行分析和处理，对系统、网络等业务模块进行自动修复，或通过短信、邮件等形式产生告警，通知业务人员进行处理。CTS触发器的使用请参考**使用CTS触发器**。

文档数据库服务 DDS

使用DDS触发器，每次更新数据库中的表时，都可以触发Functiongraph函数以执行额外的工作。DDS触发器的使用请参考**使用DDS触发器**。

分布式消息服务 Kafka 版

使用Kafka触发器，当向Kafka实例的Topic生产消息时，FunctionGraph会消费消息，触发函数以执行额外的工作，关于Kafka触发器的使用请参见**使用Kafka触发器**。

云监控服务 CES

FunctionGraph函数实现了与云监控服务对接，函数上报云监控服务的监控指标，用户可以通过云监控服务来查看函数产生的监控指标和告警信息。查看监控指标请参考**查看监控图表**。

分布式消息服务 RabbitMQ 版

使用RabbitMQ触发器，FunctionGraph会定期轮询RabbitMQ实例指定Exchange绑定的队列下的新消息，FunctionGraph将轮询得到的消息作为参数传递来调用函数。

事件网格服务 EventGrid

可以编写FunctionGraph函数来处理EventGrid的通知，在将消息发布到EG事件源时，服务可以通过将消息负载作为参数传递（**EG示例事件**）来调用FunctionGraph函数，FunctionGraph函数代码可以处理事件，比如将消息发送到其他云服务。EG消息触发的使用过程请参考**使用EG触发器**。

设备接入服务 IoTDA

使用IoTDA触发器，对于设备上报到平台的数据，FunctionGraph可跟踪设备的设备属性、消息上报，状态变更，分析、整理和计量数据流（**IoTDA示例事件**）。关于IoTDA触发器的使用请参见**使用IoTDA触发器**。

说明

IoTDA触发器当前仅支持华南-广州、华北-北京四、中国-香港、亚太-曼谷。

示例事件

- SMN示例事件

```
{
   "record": [
      {
         "event_version": "1.0",
         "smn": {
            "topic_urn": "urn:smn:{region}:0162c0f220284698b77a3d264376343a:{function_name}",
            "timestamp": "2018-01-09T07:11:40Z",
            "message_attributes": null,
            "message": "this is smn message content",
            "type": "notification",
            "message_id": "a51671f77d4a479cacb09e2cd591a983",
            "subject": "this is smn message subject"
         },
         "event_subscription_urn": "urn:fss:
{region}:0162c0f220284698b77a3d264376343a:function:default:read-smn-message:latest",
         "event_source": "smn"
      }
],
   "functionname": "test",
   "requestId": "7c307f6a-cf68-4e65-8be0-4c77405a1b2c",
   "timestamp": "Wed Nov 15 2017 12:00:00 GMT+0800 (CST)"
}
```

表 1-5 参数说明

参数	类型	示例值	描述
event_version	String	1.0	事件协议的版本。
topic_urn	String	参考示例	SMN事件唯一编号

参数	类型	示例值	描述
type	String	notification	事件的类型
RequestID	String	7c307f6a-cf68-4e65-8be0-4c77405a1b2c	请求ID。每个请求的ID取值唯一。
message_id	String	a51671f77d4a479cacb09e2cd591a983	消息ID。每条消息的ID取值唯一。
Message	String	this is smn message content	消息内容
event_source	String	smn	事件源
event_subscription_urn	String	参考示例	添加的订阅唯一编号
timestamp	String	Wed Nov 15 2017 12:00:00 GMT+0800 (CST)	事件发生的时间

- APIG示例事件

```
{
    "body": "{\"test\":\"body\"}",
    "requestContext": {
        "apiId": "bc1dcffd-aa35-474d-897c-d53425a4c08e",
        "requestId": "11cdcdcf33949dc6d722640a13091c77",
        "stage": "RELEASE"
    },
    "queryStringParameters": {
        "responseType": "html"
    },
    "httpMethod": "GET",
"pathParameters": {
"path":"value"
},
    "headers": {
        "accept-language": "zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2",
        "accept-encoding": "gzip, deflate, br",
        "x-forwarded-port": "443",
        "x-forwarded-for": "103.218.216.98",
        "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
        "upgrade-insecure-requests": "1",
        "host": "50eedf92-c9ad-4ac0-827e-d7c11415d4f1.apigw.region.cloud.com",
        "x-forwarded-proto": "https",
        "pragma": "no-cache",
        "cache-control": "no-cache",
        "x-real-ip": "103.218.216.98",
        "user-agent": "Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:57.0) Gecko/20100101 Firefox/57.0"
    },
    "path": "/apig-event-template",
    "isBase64Encoded": true
}
```

约束与限制：

- 通过APIG服务调用函数服务时，isBase64Encoded的值默认为true，表示APIG传递给FunctionGraph的请求体body已经进行Base64编码，需要先对body内容Base64解码后再处理。
- 函数必须按以下结构返回字符串。

```
{
    "isBase64Encoded": true|false,
    "statusCode": httpStatusCode,
    "headers": {"headerName":"headerValue",...},
    "body": "..."
}
```

表 1-6 参数说明

参数	类型	示例值	描述
body	String	"{\"test\":\"body\"}"	记录实际请求转换为String字符串后的内容。
requestContext	Map	参考示例	请求来源的API网关的配置信息、请求标识、认证信息、来源信息。
httpMethod	String	GET	记录实际请求的HTTP方法
queryStringParameters	Map	参考示例	记录在API网关中配置过的Query参数以及实际取值。
pathParameters	Map	参考示例	记录在API网关中配置过的Path参数以及实际取值。
headers	Map	参考示例	记录实际请求的完整Header内容
path	String	/apig-event-template	记录实际请求的完整的Path信息
isBase64Encoded	Boolean	True	默认为true

- DIS示例事件

```
{
    "ShardID": "shardId-0000000000",
    "Message": {
        "next_partition_cursor":
"eyJnZXRJdGVyYXRvclBhcmFtIjp7InN0cmVhbS1uYW1lIjoiZGlzLXN3dGVzdCIsInBhcnRpdGlvbi1pZCI6InNoYXJkSWQtMDAwMDAwMDAwMCIsImN1cnNvci10eXBlIjoiVFJJTV9IT1JJWk9OIiwic3RhcnRpbmctc2VxdWVuY2UtbnVtYmVyIjoiNCJ9LCJnZW5lcmF0ZVRpbWVzdGFtcCI6MTUwOTYwNjM5MjE5MX0",
        "records": [
            {
                "partition_key": "shardId_0000000000",
                "data": "d2VsY29tZQ==",
```



```
            "sequence_number": "0"
        },
        {
            "partition_key": "shardId_0000000000",
            "data": "dXNpbmc=",
            "sequence_number": "1"
        },
        {
            "partition_key": "shardId_0000000000",
            "data": "RnVuY3Rpb25TdGFnZQ==",
            "sequence_number": "2"
        },
        {
            "partition_key": "shardId_0000000000",
            "data": "c2VydmljZQ==",
            "sequence_number": "3"
        }
    ],
    "millis_behind_latest": ""
  },
  "Tag": "latest",
  "StreamName": "dis-swtest"
}
```

表 1-7 参数说明

参数	类型	示例值	描述
ShardID	String	shardId-0000000000	数据下载分区的ID
next_partition_cursor	String	参考示例	下一个分区的游标
Records	Map	参考示例	存储在DIS通道中的数据单元
partition_key	String	参考示例	分区键
data	String	参考示例	数据块，由数据生产者添加到数据通道
sequence_number	Int	参考示例	每个记录的唯一标识符，由DIS服务自动分配
Tag	String	latest	通道的标签
StreamName	String	dis-swtest	通道名称

- TIMER示例事件

```
{
  "version": "v1.0",
  "time": "2018-06-01T08:30:00+08:00",
  "trigger_type": "TIMER",
  "trigger_name": "Timer_001",
  "user_event": "User Event"
}
```


表 1-8 参数说明

参数	类型	示例值	描述
version	String	V1.0	事件协议的版本
time	String	2018-06-01T08:30:00+08:00	事件产生的时间
trigger_type	String	TIMER	触发器的类型
trigger_name	String	Timer_001	触发器的名字
user_event	String	User Event	在创建触发器时配置的附加信息

- LTS示例事件

```
{
   "lts": {
      "data":
"ICB7CiAgICAibG9ncyI6W3sKICAgICAgICAgIm1lc3NhZ2UiOiIyMDE4LTA4LTA4LzA4OjA4OjA4IFtXUk5dIFt0ZXN0LmdvOjA4XVRoaXMgaXMgYSB0ZXN0IG1lc3NhZ2UuIiwKICAgICAgICAgInRpbWUiOjE1MzAwMDk2NTMwNTksCiAgICAgICAgICJob3N0X25hbWUiOiJlY3MtdGVzdCIsCiAgICAgICAgICJpcCI6IjE5Mi4xNjguMS4xIiwKICAgICAgICAgInBhdGgiOiJ2YXIvbG9nL3Rlc3QubG9nIiwKICAgICAgICAgImxvZ191aWQiOiI2NjNkNjkzMC03OTJkLTExZTgtOGIwOC0yODZlZDQ4OGNlNzAiLAogICAgICAgICAgImxvZ19ubyI6MQogICAgICAgICAgIH1dLAogICAgICAgICAgIm93bmVyIjogIjYyODBlMTcwYmQ5MzRmNjBhNGQ4NTFjZjVjYTA1MTI5IiwKICAgICAgICAgICJsb2dfZ3JvdXBfaWQiOiAiOTdhOWQyODQtNDQ0OC0xMWU4LThmYTQtMjg2ZWQ0ODhjZTcwIiwKICAgICAgICAgICJsb2dfdG9waWNfaWQiOiAiMWE5Njc1YTctNzg0ZC0xMWU4LTlmNzAtMjg2ZWQ0ODhjZTcwIgogfQ=="
   }
}
```

表 1-9 Event 中涉及的参数解释

参数	类型	示例值	描述
data	Sting	参考示例	Base64编码后的数据

- CTS示例事件

```
{
   "cts": {
      "time": "2018/06/26 08:54:07 GMT+08:00",
      "user": {
         "name": "userName",
         "id": "5b726c4fbfd84821ba866bafaaf56aax",
         "domain": {
            "name": "domainName",
            "id": "b2b3853af40448fcb9e40dxj89505ba"
         }
      },
      "request": {},
      "response": {},
      "code": 204,
      "service_type": "vpc",
      "resource_type": "VPC",
      "resource_name": "workflow-2be1",
      "resource_id": "urn:fgs:{region}:2d1d891d93054bbaa69b9e866c0971ac:graph:workflow-2be1",
      "trace_name": "deleteGraph",
      "trace_type": "ConsoleAction",
      "record_time": "2018/06/26 08:54:07 GMT+08:00",
      "trace_id": "69be64a7-0233-11e8-82e4-e5d37911193e",
      "trace_status": "normal"
```



```
    }
}
```

表 1-10 参数说明

参数	类型	示例值	描述
User	Map	参考示例	本次请求的发起用户信息
Request	Map	参考示例	事件请求内容
Response	Map	参考示例	事件响应内容
Code	Int	204	事件响应码，例如200、400
service_type	String	vpc	发送方的简写，比如vpc，ecs等等
resource_type	String	VPC	发送方资源类型，比如vm，vpn等等
resource_name	String	workflow-2be1	资源名称，例如ecs服务中某个虚拟机的名称
trace_name	String	deleteGraph	事件名称,比如:startServer, shutDown等
trace_type	String	ConsoleAction	事件发生源头类型,例如ApiCall
record_time	string	2018/06/26 08:54:07 GMT +08:00	cts服务接受到这条trace的时间
trace_id	String	69be64a7-0233-11e8-82e4-e5d37911193e	事件的唯一标识符
trace_status	String	normal	事件的状态

- DDS示例事件

```
{
    "records": [
        {
            "event_source": "dds",
            "event_name": "insert",
            "region": "{region}",
            "event_version": "1.0",
            "dds": {
                "size_bytes": "100",
                "token": "{\"_data\":
\"825D8C2F4D0000001529295A100474039A3412A64BA89041DC952357FB4446645F696400645D8C2F
8E5BECCB6CF5370D6A0004\"}",
                "full_document": "{\"_id\": {\"$oid\": \"5d8c2f8e5beccb6cf5370d6a\"},\"name\": \"dds
```



```
\",\"age\": {\"$numberDouble\": \"52.0\"}}",
                "ns": "{\"db\": \"functiongraph\",\"coll\": \"person\"}"
            },
            "event_source_id": "e6065860-f7b8-4cca-80bd-24ef2a3bb748"
        }
    ]
}
```

表 1-11 参数说明

参数	类型	示例值	描述
region	String	cn-north-1	DDS实例所在的地域
event_version	String	1.0	事件协议的版本
event_source	String	dds	事件的来源
event_name	String	insert	事件的名字
size_bytes	Int	100	消息的字节数
token	String	参考示例	Base64编码后的数据
full_document	String	参考示例	完整的文件信息
ns	String	参考示例	列名
event_source_id	e6065860-f7b8-4cca-80bd-24ef2a3bb748	参考示例	事件源唯一标识符

- Kafka示例事件

```
{
    "event_version": "v1.0",
    "event_time": 1576737962,
    "trigger_type": "KAFKA",
    "region": "{region}",
    "instance_id": "81335d56-b9fe-4679-ba95-7030949cc76b",
    "records": [
        {
            "messages": [
                "kafka message1",
                "kafka message2",
                "kafka message3",
                "kafka message4",
                "kafka message5"
            ],
            "topic_id": "topic-test"
        }
    ]
}
```

表 1-12 参数说明

参数	类型	示例值	描述
event_version	String	v1.0	事件协议的版本

参数	类型	示例值	描述
event_time	String	2018-01-09T07:50:50.028Z	事件发生的时间
trigger_type	String	KAFKA	事件类型
region	String	cn-north-1	Kafka实例所在的地域
instance_id	String	81335d56-b9fe-4679-ba95-7030949cc76b	创建的Kafka实例的唯一标识符。
messages	String	参考示例	消息内容
topic_id	String	topic-test	消息的唯一标识符

- GeminiDB示例事件

```
{
   "records": [
      {
         "event_name": "\"insert\"",
         "event_version": "1.0",
         "event_source": "gemini_mongo",
         "region": "{region}",
         "gemini_mongo": {
            "full_document": "{\"_id\": {\"$oid\":\"5f61de944778db5fcded3f87\"},\"zhangsan\":
\"zhangsan\"}",
            "ns": "{\"db\": \"zhangsan\",\"coll\": \"zhangsan\"}",
            "size_bytes": "100",
            "token": "{\"_data\":
\"825F61DE940000000129295A1004A2D9AE61206C43A5AF47CAF7C5C00C5946645F696400645F61DE
944778DB5FCDED3F870004\"}"
         },
         "event_source_id": "51153d19-2b7d-402c-9a79-757163258a36"
      }
   ],
   "vernier": "{\"_data\":
\"825F61DE940000000129295A1004A2D9AE61206C43A5AF47CAF7C5C00C5946645F696400645F61DE
944778DB5FCDED3F870004\"}"
}
```

表 1-13 参数说明

参数	类型	示例值	描述
region	String	cn-north-1	GeminiDB实例所在的地域
event_source	String	gemini_mongo	事件的来源
event_version	String	1.0	事件协议的版本
full_document	String	参考示例	完整的文件信息
size_bytes	Int	100	消息的字节数

参数	类型	示例值	描述
token	String	参考示例	Base64编码后的数据
vernier	String	参考示例	游标

- RabbitMQ示例事件

```
{
  "event_version": "v1.0",
  "event_time": 1576737962,
  "trigger_type": "RABBITMQ",
  "region": "{region}",
  "records": [
    {
      "messages": [
        "rabbitmq message1",
        "rabbitmq message2",
        "rabbitmq message3",
        "rabbitmq message4",
        "rabbitmq message5"
      ],
      "instance_id": "81335d56-b9fe-4679-ba95-7030949cc76b",
      "exchange": "exchange-test"
    }
  ]
}
```

表 1-14 参数说明

参数	类型	示例值	描述
event_version	String	v1.0	事件协议的版本
Region	String	cn-north-1	RabbitMQ实例所在的地域
instance_id	String	81335d56-b9fe-4679-ba95-7030949cc76b	创建的RabbitMQ实例的唯一标识符。

- EG示例事件

  RocketMQ自定义事件源

```
{
  "datacontenttype": "application/json",
  "data": {
    "context": "yyyyy"
  },
  "subject": "ROCKETMQ:region:domainId/projectId:ROCKETMQ:eventSourceName",
  "specversion": "1.0",
  "id": "016d5bd3-6231-4e9e-86ef-e451a070d598",
  "source": "eventSourceName",
  "time": "2023-04-07T11:51:10Z",
  "type": "ROCKETMQ:CloudTrace:RocketmqCall"
}
```

  OBS应用事件源

```
{
  "channel_id":"b65779ed-d9d0-4a6c-b312-c767226964cf",
  "description":"",
```



```
  "name":"subscription-xeak",
  "sources":[
    {
      "id":null,
      "name":"HC.OBS.DWR",
      "detail":{
        "bucket":"eventbucket",
        "objectKeyEncode":true
      },
      "filter":{
        "source":[
          {
            "op":"StringIn",
            "values":[
              "HC.OBS.DWR"
            ]
          }
        ],
        "type":[
          {
            "op":"StringIn",
            "values":[
              "OBS:DWR:ObjectCreated:PUT",
              "OBS:DWR:ObjectCreated:POST"
            ]
          }
        ],
        "subject":{
          "and":[
            {
              "op":"StringStartsWith",
              "values":[
                "/ddd"
              ]
            }
          ]
        },
        "data":{
          "obs":{
            "bucket":{
              "name":[
                {
                  "op":"StringIn",
                  "values":[
                    "output-your"
                  ]
                }
              ]
            }
          }
        }
      },
      "provider_type":"OFFICIAL"
    }
  ],
  "targets":[
    {
      "id":null,
      "name":"HC.FunctionGraph",
      "detail":{
        "urn":"urn:fss:cn-north-7:c53626012ba84727b938ca8bf03108ef:function:A-nodejs-
lqz:pylog:latest",
        "agency_name":"EG_AGENCY"
      },
      "dead_letter_queue":null,
      "provider_type":"OFFICIAL",
      "transform":{
        "type":"ORIGINAL",
        "value":""
```



```
        }
      }
   ]
}
```

云服务事件源：

```
{
   "specversion":"1.0",
   "id":"eaf3b6a6-d525-11ed-a4ca-1baaeb906770",
   "source":"HC.OBS",
   "type":"OBS:CloudTrace:Others",
   "datacontenttype":"application/json",
   "subject":"OBS:cn-north-5:1d60cc02b9814b9e8cab1ff36886cacb/
a5b94f2084a14e3eb8273dd224b89d9a:OBJECT",
   "time":"2023-04-07T09:21:53.271Z",
   "data":{
      "code":"200",
      "source_ip":"10.62.9.176",
      "trace_type":"Others",
      "event_type":"data",
      "project_id":"a5b94f2084a14e3eb8273dd224b89d9a",
      "total_time":"138",
      "content_length":"6887848",
      "trace_id":"eaf3b6a6-d525-11ed-a4ca-1baaeb906770",
      "trace_name":"GET.OBJECT",
      "resource_type":"OBJECT",
      "trace_rating":"normal",
      "service_type":"OBS",
      "tracker_name":"obs-eg",
      "time":"1680859313271",
      "resource_name":"fangxin-sdk:SDK/nodejs-sdk.zip",
      "record_time":"1680859313271",
      "request_id":"000001875B05B4AB8411EF94DDE202C0",
      "user":{
         "domain":{
            "id":"1d60cc02b9814b9e8cab1ff36886cacb",
            "name":"hwstaff_pub_fcspaasw3"
         },
         "name":"devuser",
         "id":"0d8880584b0090271f7cc00857a7c7b9"
      }
   }
}
```

说明

更多云服务事件源，请参见**云服务事件源**。

表 1-15 参数说明

参数	类型	示例值	描述
datacontenttype	String	application/json	数据类型
data	Map	见示例	数据
subject	String	见示例	目标值
specversion	String	1.0	版本
id	String	见示例	唯一键值
source	String	eventSourceName	来源名称
time	String	见示例	发布订阅时间

参数	类型	示例值	描述
type	String	ROCKETMQ:CloudTrace:RocketmqCall	订阅类型

- IoTDA示例事件


```
{
   "resource" : "device",
   "event" : "create",
   "event_time" : "20240919T011335Z",
   "event_time_ms" : "2024-09-19T01:13:35.854Z",
   "request_id" : "75127474-1a26-4578-8847-3128d6101954",
   "notify_data" : {
      "body" : {
         "app_id" : "3d40caf3ddfc4e83815b54b50f13aad7",
         "app_name" : "DefaultApp_6439vdv2",
         "device_id" : "66eb7a0ffa8d9c36870c6892_ttytytytytytyt",
         "node_id" : "ttytytytytytyt",
         "gateway_id" : "66eb7a0ffa8d9c36870c6892_ttytytytytytyt",
         "node_type" : "GATEWAY",
         "auth_info" : {
            "auth_type" : "SECRET",
            "secure_access" : false,
            "timeout" : 0
         },
         "product_id" : "66eb7a0ffa8d9c36870c6892",
         "product_name" : "test",
         "status" : "INACTIVE",
         "create_time" : "20240919T011335Z"
      }
   }
}
```


表 1-16 参数说明

参数	类型	示例值	描述
resource	string	device	数据来源，包括：设备、设置属性、设备消息、设备消息状态、设备状态、产品、设备异步命令状态、运行日志、批量任务。
event	string	create	触发事件。
event_time	string	20240919T011335Z	字符串格式的事件触发时间。
event_time_ms	string	2024-09-19T01:13:35.854Z	datetime格式的事件触发时间。

参数	类型	示例值	描述
request_id	string	75127474-1a26-4578-8847-3128d6101954	请求id。
notify_data	object 参见表1-17	-	推送消息。

表 1-17 NotifyData

参数	类型	示例值	描述
body	object参见表1-18	-	推送消息内容。

表 1-18 NotifyDataBody

参数	类型	示例值	描述
app_id	string	3d40caf3ddfc4e83815b54b50f13aad7	资源空间ID。
app_name	string	DefaultApp_6439vdv2	资源空间名称。
device_id	string	66eb7a0ffa8d9c36870c6892_ttytytytytytyt	设备ID，用于唯一标识一个设备。在注册设备时直接指定，或者由物联网平台分配获得。由物联网平台分配时，生成规则为"product_id" + "_" + "node_id"拼接而成。 最大长度：256
node_id	string	ttytytytytytyt	设备标识码，通常使用IMEI、MAC地址或Serial No作为nodeId。 最大长度：64

参数	类型	示例值	描述
gateway_id	string	66eb7a0ffa8d9c36870c6892_ttytytytytytyt	网关ID，用于标识设备所属的父设备，即父设备的设备ID。当设备是直连设备时，gateway_id与设备的device_id一致。当设备是非直连设备时，gateway_id为设备所关联的父设备的device_id。
node_type	string	GATEWAY	设备节点类型。
product_id	string	66eb7a0ffa8d9c36870c6892	设备关联的产品ID，用于唯一标识一个产品模型。
product_name	string	test	设备关联的产品名称。
status	string	INACTIVE	设备的状态。 ONLINE：设备在线。 OFFLINE：设备离线。 ABNORMAL：设备异常。 INACTIVE：设备未激活。 FREEZED：设备冻结。
create_time	string	20240919T011335Z	在物联网平台注册设备的时间。格式：yyyyMMdd'T'HHmmss'Z'，如20151212T121212Z。
auth_info	Object参见表1-19	-	设备的接入认证信息。

表 1-19 AuthInfo

参数	类型	示例值	描述
auth_type	string	SECRET	鉴权类型。支持密钥认证接入(SECRET)和证书认证接入(CERTIFICATES)两种方式。使用密钥认证接入方式(SECRET)填写secret字段，使用证书认证接入方式(CERTIFICATES)填写fingerprint字段，不填写auth_type默认为密钥认证接入方式(SECRET)。
secure_access	Boolean	false	指设备是否通过安全协议方式接入，默认值为true。 true：通过安全协议方式接入。 false：通过非安全协议方式接入。
timeout	Integer	0	设备验证码的有效时间，单位：秒，默认值：0。若设备在有效时间内未接入物联网平台并激活，则平台会删除该设备的注册信息。若设置为“0”，则表示设备验证码不会失效（建议填写为“0”）。注意：只有注册设备接口或者修改设备接口修改timeout时返回该参数。 最小值：0 最大值：2147483647 缺省值：0

具体的设备相关消息请以IoTDA官网为准，例如设备添加：**设备添加通知**。

1.3 函数工程打包规范

打包规范说明

函数除了支持在线编辑代码，还支持上传ZIP、JAR、引入OBS文件等方式上传代码，上传操作过程请参见**配置函数代码**，函数工程的打包规范说明如**表1-20**所示。

表 1-20 函数工程打包规范

编程语言	JAR包	ZIP包	OBS文件
Node.js	不支持该方式	• 假如函数工程文件保存在“~/Code/”文件夹下，在打包的时候务必进入Code文件夹下选中所有工程文件进行打包，确保入口函数是程序执行的入口，确保解压后，入口函数所在的文件位于根目录。 • 如果函数工程引入了第三方依赖，可以将第三方依赖打成ZIP包，在函数代码界面设置外部依赖包；也可以将第三方依赖和函数工程文件一起打包。	将工程打成ZIP包，上传到OBS存储桶。
PHP	不支持该方式	• 假如函数工程文件保存在“~/Code/”文件夹下，在打包的时候务必进入Code文件夹下选中所有工程文件进行打包，确保入口函数是程序执行的入口，确保解压后，入口函数所在的文件位于根目录。 • 如果函数工程引入了第三方依赖，可以将第三方依赖打成ZIP包，在函数代码界面设置外部依赖包；也可以将第三方依赖和函数工程文件一起打包。	将工程打成ZIP包，上传到OBS存储桶。

编程语言	JAR包	ZIP包	OBS文件
Python 2.7	不支持该方式	• 假如函数工程文件保存在“~/Code/”文件夹下，在打包的时候务必进入Code文件夹下选中所有工程文件进行打包，确保入口函数是程序执行的入口，确保解压后，入口函数所在的文件位于根目录。 • 如果函数工程引入了第三方依赖，可以将第三方依赖打成ZIP包，在函数代码界面设置外部依赖包；也可以将第三方依赖和函数工程文件一起打包。	将工程打成ZIP包，上传到OBS存储桶。
Python 3.6	不支持该方式	• 假如函数工程文件保存在“~/Code/”文件夹下，在打包的时候务必进入Code文件夹下选中所有工程文件进行打包，确保入口函数是程序执行的入口，确保解压后，入口函数所在的文件位于根目录。 • 如果函数工程引入了第三方依赖，可以将第三方依赖打成ZIP包，在函数代码界面设置外部依赖包；也可以将第三方依赖和函数工程文件一起打包。	将工程打成ZIP包，上传到OBS存储桶。
Java 8	如果函数没有引用第三方件，可以直接将函数工程编译成Jar包。	如果函数引用第三方件，将函数工程编译成Jar包后，将所有依赖三方件和函数JAR包打成ZIP包。	将工程打成ZIP包，上传到OBS存储桶。

编程语言	JAR包	ZIP包	OBS文件
Go 1.x	不支持该方式	必须在编译之后打ZIP包，编译后的二进制文件必须与执行函数入口保持一致，例如二进制名称为Handler，则执行入口为Handler。	将工程打成ZIP包，上传到OBS存储桶。
C#	不支持该方式	必须在编译之后打ZIP包，必须包含“工程名.deps.json”，“工程名.dll”，“工程名.runtimeconfig.json”，“工程名.pdb”和“HC.Serverless.Function.Common.dll”文件。	将工程打成ZIP包，直接上传到OBS存储桶。
定制运行时	不支持该方式	打ZIP包，必须包含“bootstrap”可执行引导文件。	将工程打成ZIP包，直接上传到OBS存储桶。
Cangjie	不支持该方式	必须在编译之后打ZIP包，编译后的二进制文件必须与执行函数入口保持一致，例如二进制名称为libuser_func_test_success.so，则执行入口为libuser_func_test_success.so。	将工程打成ZIP包，上传到OBS存储桶。

ZIP 工程包示例

- Nods.js工程ZIP包目录示例

```
Example.zip                 示例工程包
|--- lib                    业务文件目录
|--- node_modules           npm三方件目录
|--- index.js               入口js文件（必选）
|--- package.json           npm项目管理文件
```

- PHP工程ZIP包目录示例

```
Example.zip                 示例工程包
|--- ext                    扩展库目录
|--- pear                   PHP扩展与应用仓库
|--- index.php              入口PHP文件
```

- Python工程ZIP包目录示例

```
Example.zip                 示例工程包
|--- com                    业务文件目录
|--- PLI                    第三方依赖PLI目录
|--- index.py               入口py文件（必选）
|--- watermark.py           实现打水印功能的py文件
|--- watermark.png          水印图片
```


- Java工程ZIP包目录示例

```
Example.zip                          示例工程包
|--- obstest.jar                     业务功能JAR包
|--- esdk-obs-java-3.20.2.jar        第三方依赖JAR包
|--- jackson-core-2.10.0.jar         第三方依赖JAR包
|--- jackson-databind-2.10.0.jar     第三方依赖JAR包
|--- log4j-api-2.12.0.jar            第三方依赖JAR包
|--- log4j-core-2.12.0.jar           第三方依赖JAR包
|--- okhttp-3.14.2.jar               第三方依赖JAR包
|--- okio-1.17.2.jar                 第三方依赖JAR包
```

- Go工程ZIP包目录示例

```
Example.zip                          示例工程包
|--- testplugin.so                   业务功能包
```

- C#工程ZIP包目录示例

```
Example.zip                                  示例工程包
|--- fssExampleCsharp2.0.deps.json           工程编译产生文件
|--- fssExampleCsharp2.0.dll                 工程编译产生文件
|--- fssExampleCsharp2.0.pdb                 工程编译产生文件
|--- fssExampleCsharp2.0.runtimeconfig.json  工程编译产生文件
|--- Handler                                 帮助文件，可直接使用
|--- HC.Serverless.Function.Common.dll       函数工作流提供的dll
```

- Cangjie工程ZIP包目录示例

```
fss_example_cangjie.zip                示例工程包
|--- libuser_func_test_success.so       业务功能包
```

- 定制运行时

```
Example.zip                          示例工程包
|--- bootstrap                       可执行引导文件
```

1.4 在函数中引入动态链接库

在函数中引入动态链接库的方式如下：

- 函数运行环境中已经默认将代码根目录和根目录下的lib目录加入到LD_LIBRARY_PATH中，只需要将动态链接库放到此处即可。
- 在代码中直接修改LD_LIBRARY_PATH环境变量。
- 如果依赖的.so文件放在其他目录，可以在配置页面设置LD_LIBRARY_PATH环境变量指明对应的目录，具体请参考**配置环境变量**。

  如下所示，其中，**/opt/function/code**、**/opt/function/code/lib**表示函数代码的工程目录。

图 1-1 配置环境变量

- 如果使用了挂载文件系统中的库，可以在配置页面设置LD_LIBRARY_PATH环境变量指明挂载文件系统中对应的目录。

2 函数初始化入口 Initializer

概述

Initializer是函数的初始化逻辑入口，不同于请求处理逻辑入口的handler，在有函数初始化的需求场景中，设置了Initializer后，FunctionGraph首先调用initializer完成函数的初始化，之后再调用handler处理请求；如果没有函数初始化的需求则可以跳过initializer，直接调用handler处理请求。

适用场景

用户函数执行调度包括以下几个阶段：

1. FunctionGraph预先为函数分配执行函数的容器资源。
2. 下载函数代码。
3. 通过runtime运行时加载代码。
4. 用户函数内部进行初始化逻辑。
5. 函数处理请求并将结果返回。

其中1、2和3是系统层面的冷启动开销，通过对调度以及各个环节的优化，函数工作流服务能做到负载快速增长时稳定的延时。4是函数内部初始化逻辑，属于应用层面的冷启动开销，例如深度学习场景下加载规格较大的模型、数据库场景下连接池构建、函数依赖库加载等等。

为了减小应用层冷启动对延时的影响，FunctionGraph推出了initializer接口，系统能识别用户函数的初始化逻辑，从而在调度上做相应的优化。

引入 initializer 接口的价值

- 分离初始化逻辑和请求处理逻辑，程序逻辑更清晰，让用户更易写出结构良好，性能更优的代码。
- 用户函数代码更新时，系统能够保证用户函数的平滑升级，规避应用层初始化冷启动带来的性能损耗。新的函数实例启动后能够自动执行用户的初始化逻辑，在初始化完成后再处理请求。
- 在应用负载上升，需要增加更多函数实例时，系统能够识别函数应用层初始化的开销，更准确的计算资源伸缩的时机和所需的资源量，让请求延时更加平稳。
- 即使在用户有持续的请求且不更新函数的情况下，系统仍然有可能将已有容器回收或更新，这时没有平台方的冷启动，但是会有业务方冷启动，Initializer可以最大限度减少这种情况。

initializer 接口规范

各个runtime的initializer接口有以下共性：

- 无自定义参数

  Initializer不支持用户自定义参数，只能获取FunctionGraph提供的context参数中的变量进行相关逻辑处理。

- 无返回值

  开发者无法从invoke的响应中获取initializer预期的返回值。

- 超时时间

  开发者可单独设置initializer的超时时间，与handler的超时相互独立，但最长不超过 300 秒。

- 执行时间

  运行函数逻辑的进程称之为函数实例，运行在容器内。FunctionGraph会根据用户负载伸缩函数实例。每当有新函数实例创建时，系统会首先调用initializer。系统保证一定initializer执行成功后才会执行handler逻辑。

- 最多成功执行一次

  FunctionGraph保证每个函数实例启动后只会成功执行一次initializer 。如果执行失败，那么该函数实例执行失败，选取下一个实例重新执行，最多重试3次。一旦执行成功，在该实例的生命周期内不会再执行initializer，收到Invoke请求之后只执行请求处理函数。

- initializer入口命名

  除Java外，其他runtime的initializer入口命名规范与原有的执行函数命名保持一致，格式为 [文件名].[initializer名]，其中initializer名可自定义。Java需要定义一个类并实现函数计算预定义的初始化接口。

- 计量计费

  Initializer的执行时间也会被计量，用户需要为此付费，计费方式同执行函数。

3 Node.js

3.1 开发事件函数

Node.js 函数接口定义

Node.js 6.10函数接口定义

```
export.handler = function(event, context, callback)
```

- 入口函数名（handler）：入口函数名称， 需和函数执行入口处用户自定义的入口函数名称一致。
- 执行事件（event）：函数执行界面由用户输入的执行事件参数， 格式为JSON对象。
- 上下文环境（context）：Runtime提供的函数执行上下文，其接口定义在**SDK接口**说明。
- 回调函数（callback）：callback方法完整声明为callback(err, message)，用户通过此方法可以返回err和message至前台结果显示页面。具体的err或message内容需要用户自己定义，如字符串。

Node.js 8.10、Node.js 10.16、Node.js 12.13、Node.js 14.18、Node.js 16.17、Node.js 18.15函数接口定义

Node.js 8.10、Node.js 10.16、Node.js 12.13、Node.js 14.18、Node.js 16.17、Node.js 18.15 Runtime除了兼容Node.js 6.10 Runtime函数的接口定义规范，还支持使用async的异步形式作为函数入口。通过return进行返回。

```
exports.handler = async (event, context, callback[可选]) => { return data;}
```

Node.js函数的函数执行入口参数格式为：**[文件名].[函数名]**，请参考**函数执行入口**通过FunctionGraph控制台进行配置。例如创建函数时设置函数执行入口为index.handler，那么FunctionGraph会去加载index.js中定义的handler函数。

说明

建议使用Node.js 12.13版本。

Node.js 的 initializer 入口介绍

FunctionGraph目前支持以下Node.js运行环境：

- Node.js 6.10 (runtime = Node.js 6)
- Node.js 8.10 (runtime = Node.js 8)
- Nodejs 10.16(runtime = Node.js 10)
- Nodejs 12.13(runtime = Node.js 12)
- Node.js 14.18(runtime = Node.js 14)
- Node.js 16.17(runtime = Node.js 16)
- Node.js 18.15(runtime = Node.js 18)

Initializer入口格式为：**[文件名].[initializer名]**

示例：实现initializer接口时指定的Initializer入口为“index.initializer”，那么函数工作流服务会去加载index.js中定义的initializer函数。

在函数工作流服务中使用Node.js编写initializer逻辑，需要定义一个Node.js函数作为initializer入口，以下为initializer的简单示例。

```
exports.initializer = function(context, callback) {
    callback(null, '');
    };
```

- 函数名

  exports.initializer需要与实现initializer接口时的Initializer字段相对应。

  示例：创建函数时指定的Initializer入口为index.initializer，那么FunctionGraph会去加载index.js中定义的initializer函数。

- context参数

  context参数中包含一些函数的运行时信息。例如：request id、临时AccessKey、function meta等。

- callback参数

  callback参数用于返回调用函数的结果，其签名是function(err, data)，与Nodejs中惯用的callback一样，它的第一个参数是error，第二个参数data。如果调用时error不为空，则函数将返回HandledInitializationError，由于屏蔽了初始化函数的返回值，所以data中的数据是无效的，可以参考上文的示例设置为空。

SDK 接口

Context类中提供了许多上下文方法供用户使用，其声明和功能如表3-1所示。

getToken()、getAccessKey()和getSecretKey()方法返回的内容包含敏感信息，请谨慎使用，避免造成用户敏感信息的泄露。

表 3-1 Context 类上下文方法说明

方法名	方法说明
getRequestID()	获取请求ID。
getRemainingTimeInMilliSeconds ()	获取函数剩余运行时间。

方法名	方法说明
getAccessKey()	获取用户委托的AccessKey（有效期24小时），使用该方法需要给函数配置委托。 **当前函数工作流已停止维护Runtime SDK 中getAccessKey接口，您将无法使用getAccessKey获取临时AK。**
getSecretKey()	获取用户委托的SecretKey（有效期24小时），使用该方法需要给函数配置委托。 **当前函数工作流已停止维护Runtime SDK 中getSecretKey接口，您将无法使用getSecretKey获取临时SK。**
getSecurityAccessKey()	获取用户委托的SecurityAccessKey（有效期24小时），使用该方法需要给函数配置委托。
getSecuritySecretKey()	获取用户委托的SecuritySecretKey（有效期24小时），使用该方法需要给函数配置委托。
getSecurityToken()	获取用户委托的SecurityToken（有效期24小时），使用该方法需要给函数配置委托。
getUserData(string key)	通过key获取用户通过环境变量传入的值。
getFunctionName()	获取函数名称。
getRunningTimeInSeconds ()	获取函数超时时间。
getVersion()	获取函数的版本。
getMemorySize()	分配的内存。
getCPUNumber()	获取函数占用的CPU资源。
getPackage()	获取函数组。
getToken()	获取用户委托的token（有效期24小时），使用该方法需要给函数配置委托。
getLogger()	获取context提供的logger方法，返回一个日志输出类，通过使用其info方法按“时间-请求ID-输出内容”的格式输出日志。 如调用info方法输出日志： logg = context.getLogger() logg.info("hello")
getAlias	获取函数的别名

开发 Node.js 函数

以下为本地开发后上传代码文件，也可以直接在页面创建在线编辑。

约束与限制：

- 本例函数工程文件保存在“~/Code/”文件夹下，在打包的时候务必进入Code文件夹下选中所有工程文件进行打包，这样做的目的：定义了FunctionGraph函数的index.js是程序执行入口，确保fss_examples_nodejs.zip解压后，index.js文件位于根目录。
- callback返回的第一个参数不为null，则认为函数执行失败，会返回定义在第二个参数的HTTP错误信息。
- 当使用APIG触发器时，函数返回必须使用示例中output的格式， 函数Body参数仅支持返回如下几种类型的值。**相关约束条件请参考Base64解码和返回结构体的说明**。
  - null：函数返回的HTTP响应Body为空。
  - []byte: 函数返回的HTTP响应Body内容为该字节数组内容。
  - string：函数返回的HTTP响应Body内容为该字符串内容。

步骤1 创建函数工程

1. 创建函数代码（同步形式入口函数）

   打开文本编辑器，编写函数，代码如下，文件命名为index.js，保存文件。如下为同步方式入口函数。

```
exports.handler = function (event, context, callback) {
    const error = null;
    const output = {
        'statusCode': 200,
        'headers':
        {
            'Content-Type': 'application/json'
        },
        'isBase64Encoded': false,
        'body': JSON.stringify(event),
    }
    callback(error, output);
}
```

2. 创建函数代码(异步形式的入口函数, 运行时 8.10 及以上支持）

```
exports.handler = async (event, context) => {
    const output =
    {
        'statusCode': 200,
        'headers':
        {
            'Content-Type': 'application/json'
        },
        'isBase64Encoded': false,
        'body': JSON.stringify(event),
    }
    return output;
}
```

   如果您的Node.js函数中包含异步任务，须使用Promise以确保该异步任务在当次调用执行，可以直接return声明的Promise，也可以await执行该Promise。 暂时不支持在函数响应请求后继续执行异步任务的能力。

```
exports.handler =  async(event, context ) => {
    const output =
    {
        'statusCode': 200,
        'headers':
        {
            'Content-Type': 'application/json'
        },
```



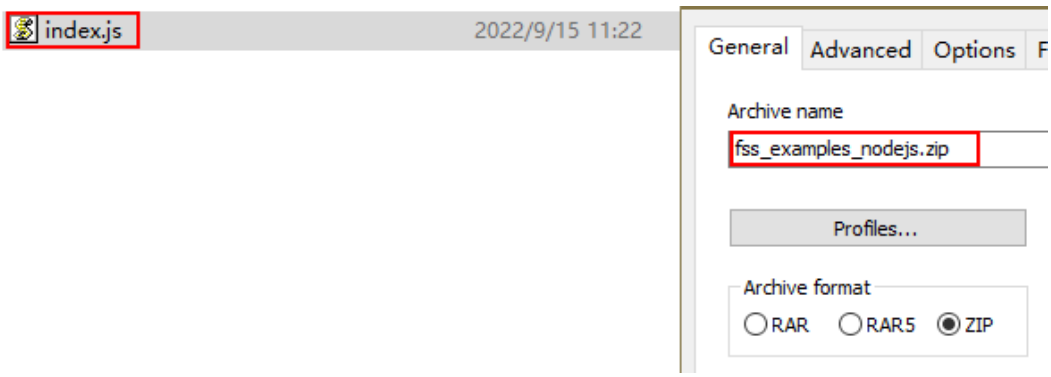
```
        'isBase64Encoded': false,
        'body': JSON.stringify(event),
    }

    const promise = new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(output)
        }, 2000)
    })
    return promise
    // anthor way
    // res = await promise;
    // return res
}
```

如果期望函数先响应，随后继续执行任务。 可以通过SDK/API调用FunctionGraph的**异步执行函数**接口实现。如果使用APIG触发器，可以单击生成的APIG触发器名称，跳转到APIG服务页面，选择Asynchronous方式调用。

步骤2 工程打包

下列示例中，使用异步形式入口作为演示。函数工程创建以后，可以得到以下目录，选中工程所有文件，打包命名为“fss_examples_nodejs.zip”。

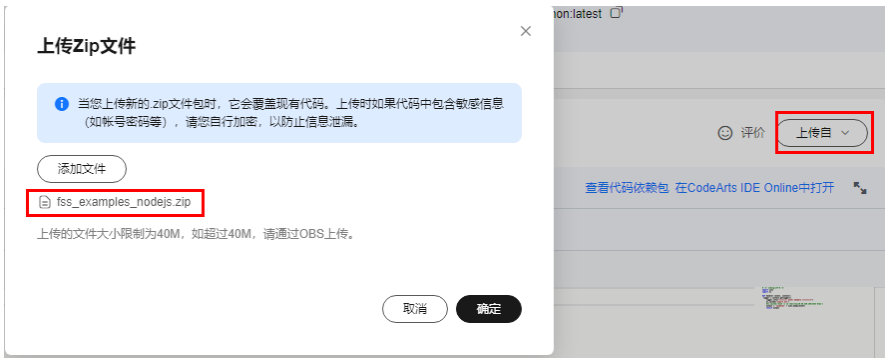
图 3-1 打包



步骤3 创建FunctionGraph函数，上传程序包

登录FunctionGraph控制台，创建Node.js函数，上传fss_examples_nodejs.zip文件。如**图3-2**所示。

图 3-2 上传程序包



函数设置中，**函数执行入口**中的index与步骤**创建函数工程**中创建的函数文件名保持一致，通过该名称找到FunctionGraph函数所在文件。

函数执行入口中的handler为函数名，与步骤创建函数工程中创建的index.js文件中的函数名保持一致。

在函数工作流控制台左侧导航栏选择“函数 > 函数列表”，单击需要设置的“函数名称”进入函数详情页，选择“设置 > 常规设置”，配置“函数执行入口”参数，如图3-3所示。其中参数值为“index.handler”格式，“**index**”和“**handler**”支持自定义命名。

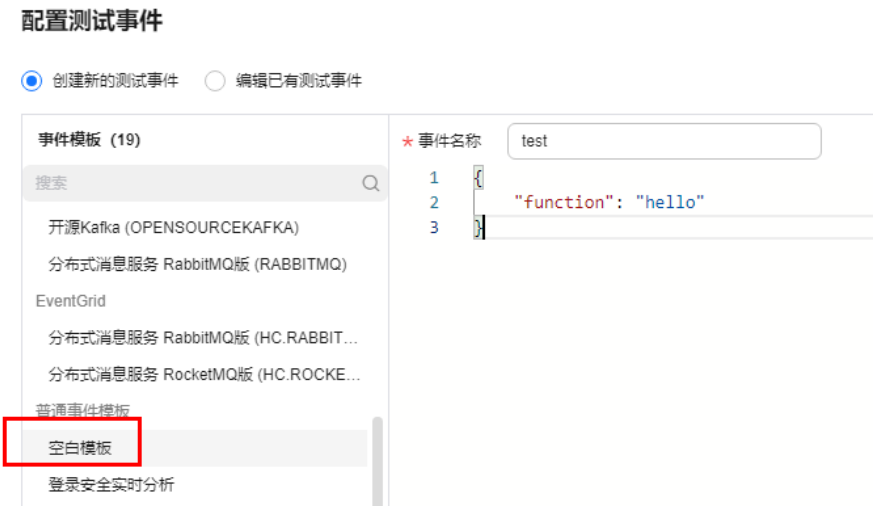
图 3-3 函数执行入口参数

步骤4 测试函数

1. 创建测试事件。

   在函数详情页，单击“配置测试事件”，弹出“配置测试事件”页，输入测试信息如图3-4所示，单击“创建”。

   图 3-4 配置测试事件

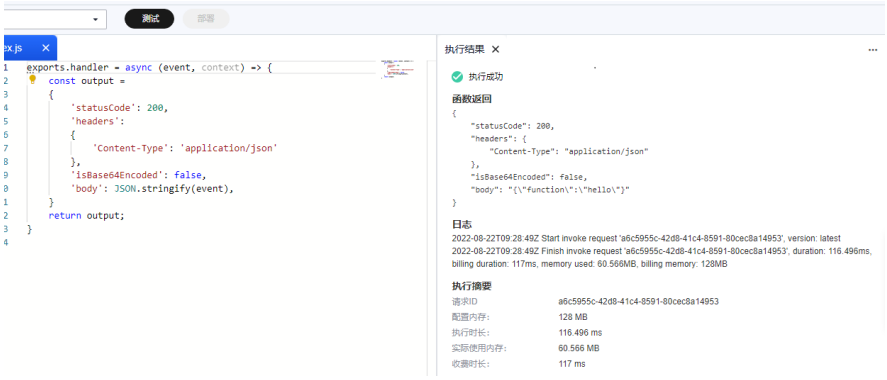


2. 在函数详情页，选择已配置测试事件，单击“测试”。

步骤5 函数执行

函数执行结果分为三部分，分别为函数返回（由callback返回）、执行摘要、日志输出（由console.log或getLogger()方法获取的日志方法输出），如图3-5所示。

图 3-5 测试结果



----结束

执行结果

执行结果由3部分组成：函数返回、执行摘要和日志。

表 3-2 执行结果说明

参数项	执行成功	执行失败

函数返回	返回函数中定义的返回信息。	返回包含错误信息和错误类型的JSON文件。格式如下：
		{
		"errorMessage": "",
		"errorType":"",
		}
		errorMessage：Runtime返回的错误信息
		errorType：错误类型

执行摘要	显示请求ID、配置内存、执行时长、实际使用内存和收费时长。	显示请求ID、配置内存、执行时长、实际使用内存和收费时长。
日志	打印函数日志，最多显示4KB的日志。	打印报错信息，最多显示4KB的日志。

3.2 开发 HTTP 函数

本章节通过HTTP函数部署koa框架，更多HTTP详情，请参见**创建HTTP函数**。

约束与限制

- HTTP函数只能绑定APIG/APIC触发器，根据函数和APIG/APIC之间的转发协议，函数的返回合法的http响应报文中必须包含body(String)、statusCode(int)、headers(Map)和isBase64Encoded(boolean)，HTTP函数会默认对返回结果做Base64编码，isBase64Encoded默认为true，其它框架同理。**相关约束条件请参考Base64解码和返回结构体的说明**。

- HTTP函数默认开放端口为8000。
- Context类中提供了许多上下文方法供用户使用，其声明和功能请参见表3-1。

前提条件

1. 准备一个bootstrap启动文件，作为HTTP函数的启动文件。举例如下：
   ```
   /opt/function/runtime/nodejs14.18/rtsp/nodejs/bin/node $RUNTIME_CODE_ROOT/index.js
   ```
   - /opt/function/runtime/nodejs14.18/rtsp/nodejs/bin/node：表示nodejs编译环境所在路径。
   - $RUNTIME_CODE_ROOT：系统变量，表示容器中项目代码存放路径/opt/function/code。
   - index.js：项目入口文件，可自定义名称。

   目前支持的Nodejs语言和对应的路径请参见表3-3。

   表 3-3 Nodejs 语言对应路径

语言	路径
Node.js6	/opt/function/runtime/nodejs6.10/rtsp/nodejs/bin/node
Node.js8	/opt/function/runtime/nodejs8.10/rtsp/nodejs/bin/node
Node.js10	/opt/function/runtime/nodejs10.16/rtsp/nodejs/bin/node
Node.js12	/opt/function/runtime/nodejs12.13/rtsp/nodejs/bin/node
Node.js14	/opt/function/runtime/nodejs14.18/rtsp/nodejs/bin/node

2. 在Linux机器上安装node环境并准备nodejs项目文件。

   Koa Web应用

   a. 创建项目文件夹。
   ```
   mkdir koa-example && cd koa-example
   ```
   b. 初始化nodejs项目，下载koa框架，文件夹中会新增node_modules文件夹和package.json、package-lock.json文件。
   ```
   npm init -y
   npm i koa
   ```
   c. 创建index.js文件，在index.js文件中引入koa框架，koa框架的使用参考koa指南。

   代码示例：
   ```
   const Koa = require("koa");
   const app = new Koa();
   const main = (ctx) = >{
       if (ctx.request.path == ("/koa")) {
           ctx.response.type = " application/json";
           ctx.response.body = "Hello World, user!";
           ctx.response.status = 200;
       } else {
           ctx.response.type = " application/json";
           ctx.response.body = 'Hello World!';
           ctx.response.status = 200;
       }
   };
   app.use(main);
   app.listen(8000, '127.0.0.1');
   console.log('Node.js web server at port 8000 is running..')
   ```


d. 创建bootstrap文件。

```
/opt/function/runtime/nodejs14.18/rtsp/nodejs/bin/node $RUNTIME_CODE_ROOT/index.js
```

3. 把项目文件和bootstrap文件打包成zip包。以koa框架为例：

```
[root@                koa-example]# ls
bootstrap  index.js  koa.zip  node_modules  package.json  package-lock.json
```

3.3 nodejs 模板

```
exports.handler = async (event, context) => {
    const output =
    {
        'statusCode': 200,
        'headers':
        {
            'Content-Type': 'application/json'
        },
        'isBase64Encoded': false,
        'body': JSON.stringify(event),
    }
    return output;
}
```

3.4 制作依赖包

制作函数依赖包推荐在Huawei Cloud EulerOS 2.0环境中进行。使用其他系统打包可能会因为底层依赖库的原因，运行出问题，比如找不到动态链接库。

约束与限制

如果安装的依赖模块需要添加依赖库，请将依赖库归档到zip依赖包文件中，例如，添加.dll、.so、.a等依赖库。

为 Nodejs 函数制作依赖包

需要先保证环境中已经安装了对应版本的Nodejs。

为Nodejs 8.10安装MySQL依赖包，可以执行如下命令。

```
npm install mysql --save
```

可以看到当前目录下会生成一个node_modules文件夹。

- Linux系统

  Linux系统下可以使用以下命令生成zip包。

  ```
  zip -rq mysql-node8.10.zip node_modules
  ```

  即可生成最终需要的依赖包。

- windows系统

  用压缩软件将node_modules目录压缩成zip文件即可。

如果需要安装多个依赖包，也可以先新建一个package.json文件，例如在package.json中填入如下内容后，执行如下命令。

```
{
    "name": "test",
```



```
    "version": "1.0.0",
    "dependencies": {
        "redis": "~2.8.0",
        "mysql": "~2.17.1"
    }
}
npm install --save
```

说明

不要使用**CNPM**命令制作nodejs依赖包。

然后将node_modules打包成zip即可生成一个既包含MySQL也包含redis的依赖包。

Nodejs其他版本制作依赖包过程与上述相同。

4 Python

4.1 开发事件函数

Python 函数接口定义

FunctionGraph运行时支持Python 2.7版本、Python 3.6、Python3.9、Python3.10版本。

函数有明确的接口定义，如下所示。

```
def handler (event, context)
```

- 入口函数名（handler）：入口函数名称，需和函数执行入口处用户自定义的入口函数名称一致。
- 执行事件（event）： 函数执行界面由用户输入的执行事件参数， 格式为JSON对象。
- 上下文环境（Context）：Runtime提供的函数执行上下文，其接口定义在**SDK接口**说明。

Python函数的函数执行入口参数格式为：**[文件名].[函数名]**，请参考**函数执行入口**通过FunctionGraph控制台进行配置。

说明

建议使用Python 3.6版本。

Python 的 initializer 入口介绍

FunctionGraph目前支持以下Python运行环境。

- Python 2.7 (runtime = python2.7)
- Python 3.6 (runtime = python3)
- Python 3.9（runtime = python3）
- Python 3.10（runtime = python3）

Initializer入口格式为：**[文件名].[initializer名]**

示例：实现initializer接口时指定的Initializer入口为main.my_initializer，那么FunctionGraph会去加载main.py中定义的my_initializer函数。

在FunctionGraph中使用Python编写initializer，需要定义一个Python函数作为initializer入口，以下为initializer的简单示例（以Python 2.7版本为例）。

```
def my_initializer(context):
    print 'hello world!'
```

- 函数名

  my_initializer需要与实现initializer接口时的Initializer字段相对应，实现initializer接口时指定的Initializer入口为main.my_initializer ，那么函数工作流服务会去加载main.py中定义的my_initializer函数。

- context参数

  context参数中包含一些函数的运行时信息，例如：request id、临时AccessKey、function meta等。

SDK 接口

Context类中提供了许多上下文方法供用户使用，其声明和功能如表4-1所示。

getToken()、getAccessKey()和getSecretKey()方法返回的内容包含敏感信息，请谨慎使用，避免造成用户敏感信息的泄露。

表 4-1 Context 类上下文方法说明

方法名	方法说明
getRequestID()	获取请求ID。
getRemainingTimeInMilliSeconds ()	获取函数剩余运行时间。
getAccessKey()	获取用户委托的AccessKey（有效期24小时），使用该方法需要给函数配置委托。 **当前函数工作流已停止维护Runtime SDK 中getAccessKey接口，您将无法使用getAccessKey获取临时AK。**
getSecretKey()	获取用户委托的SecretKey（有效期24小时），使用该方法需要给函数配置委托。 **当前函数工作流已停止维护Runtime SDK 中getSecretKey接口，您将无法使用getSecretKey获取临时SK。**
getSecurityAccessKey()	获取用户委托的SecurityAccessKey（有效期24小时），使用该方法需要给函数配置委托。
getSecuritySecretKey()	获取用户委托的SecuritySecretKey（有效期24小时），使用该方法需要给函数配置委托。
getSecurityToken()	获取用户委托的SecurityToken（有效期24小时），使用该方法需要给函数配置委托。
getUserData(string key)	通过key获取用户通过环境变量传入的值。
getFunctionName()	获取函数名称。

方法名	方法说明
getRunningTimeInSeconds ()	获取函数超时时间。
getVersion()	获取函数的版本。
getMemorySize()	分配的内存。
getCPUNumber()	获取函数占用的CPU资源。
getPackage()	获取函数组。
getToken()	获取用户委托的token（有效期24小时），使用该方法需要给函数配置委托。
getLogger()	获取context提供的logger方法，返回一个日志输出类，通过使用其info方法按“时间-请求ID-输出内容”的格式输出日志。 如调用info方法输出日志： log = context.getLogger() log.info("test")
getAlias	获取函数的别名。

开发 Python 函数

开发Python函数步骤如下。

约束与限制：

- 函数仅支持返回如下几种类型的值：
  - None：函数返回的HTTP响应Body为空。
  - String：函数返回的HTTP响应Body内容为该字符串内容。
  - 其他：当函数返回值的类型不为None和String时，函数会将返回值作为对象进行json编码，并将编码后的内容作为HTTP响应的Body，同时设置响应的“Content-Type”头为“application/json”。
- **当函数的事件源是APIG时，相关约束条件请参考Base64解码和返回结构体的说明。**
- 本例函数工程文件保存在“~/code/”文件夹下，在打包的时候务必进入code文件夹下选中所有工程文件进行打包，这样做的目的：由于定义了FunctionGraph函数的index.py是程序执行入口，确保fss_examples_python2.7.zip解压后，index.py文件位于根目录。
- 用Python语言写代码时，自己创建的包名不能与Python标准库同名，否则会提示module加载失败。例如“json”、“lib”、“os”等。

说明

以下示例使用的Python 2.7版本。

步骤1 创建函数工程。

1. 编写打印helloworld的代码。

   打开文本编辑器，编写helloworld函数，代码如下，文件命名为helloworld.py，保存文件。

   ```
   def printhello():
       print 'hello world!'
   ```

2. 定义FunctionGraph函数。

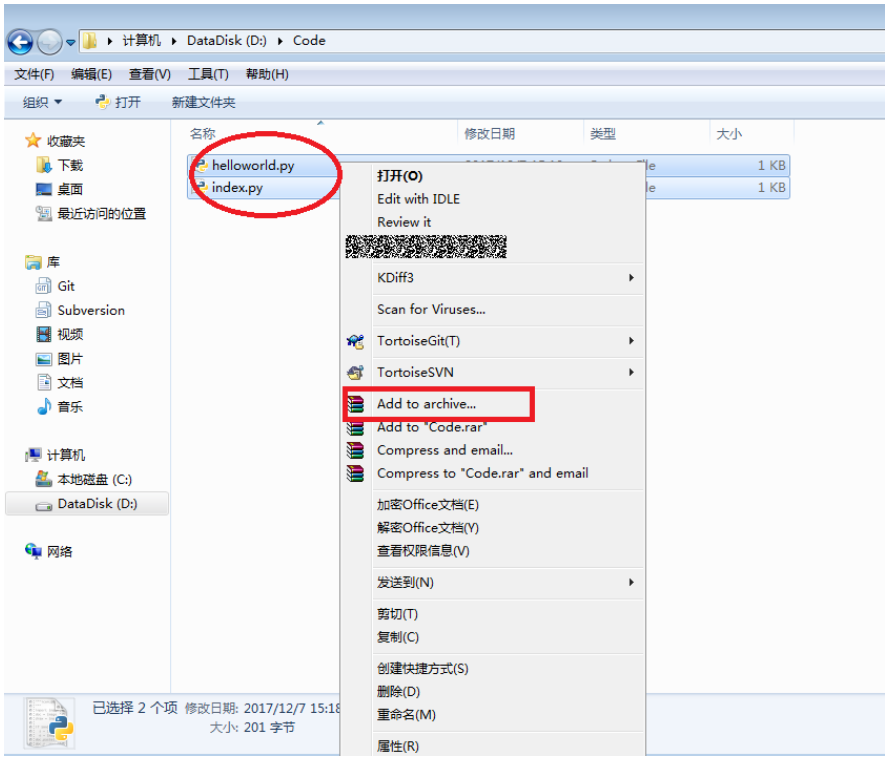
   打开文本编辑，定义函数，代码如下，文件命名为index.py，保存文件（与helloworld.py保存在同一文件夹下）。

   ```
   import json
   import helloworld

   def handler (event, context):
       output =json.dumps(event)
       helloworld.printhello()
       return output
   ```

步骤2 工程打包。

函数工程创建以后，可以得到以下目录，选中工程所有文件，打包命名为“fss_examples_python2.7.zip”，如图4-1所示。

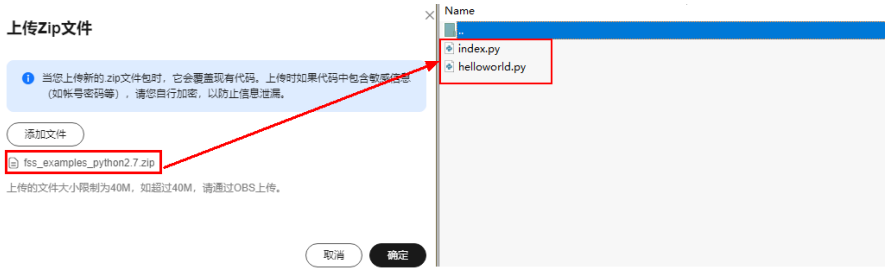
图 4-1 打包



步骤3 创建FunctionGraph函数，上传程序包。

登录FunctionGraph控制台，创建Python函数，上传fss_examples_python2.7.zip文件。如图4-2所示。

图 4-2 上传程序包



函数执行入口中的index与步骤**定义FunctionGraph函数**的文件名保持一致，通过该名称找到FunctionGraph函数所在文件。

函数执行入口中的handler为函数名，与步骤**定义FunctionGraph函数**中创建的index.py文件中的handler名称保持一致。

函数执行过程为：用户上传fss_examples_python2.7.zip保存在OBS中，触发函数后，解压缩zip文件，通过index匹配到FunctionGraph函数所在文件，通过handler匹配到index.py文件中定义的FunctionGraph函数，找到程序执行入口，执行函数。

在函数工作流控制台左侧导航栏选择“函数 > 函数列表”，单击需要设置的“函数名称”进入函数详情页，选择“设置 > 常规设置”，配置“函数执行入口”参数，如**图4-3**所示。其中参数值为“index.handler”格式，“**index**”和“**handler**”支持自定义命名。

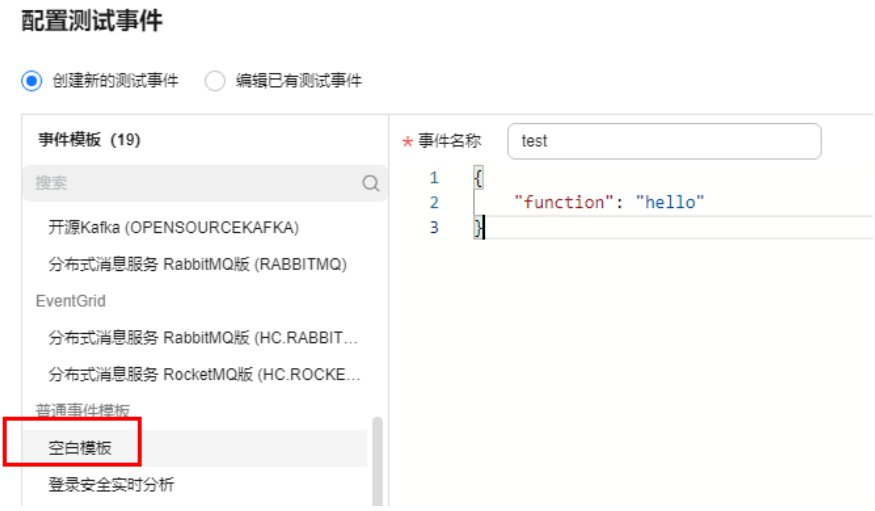
图 4-3 函数执行入口参数

步骤4 测试函数。

1. 创建测试事件。

   在函数详情页，单击“配置测试事件”，弹出“配置测试事件”页，输入测试信息如**图4-4**所示，单击“创建”。

图 4-4 配置测试事件



2. 在函数详情页，选择已配置测试事件，单击“测试”。

步骤5 函数执行。

函数执行结果分为三部分，分别为函数返回（由callback返回）、执行摘要、日志输出（由print()方法获取的日志方法输出）， 如**图10 测试结果**所示。

图 4-5 测试结果

----结束

执行结果

执行结果由3部分组成：函数返回、执行摘要和日志。

表 4-2 执行结果说明

参数项	执行成功	执行失败

函数返回	返回函数中定义的返回信息。	返回包含错误信息、错误类型和堆栈异常报错信息的JSON文件。格式如下：
		{
		"errorMessage": "",
		"errorType": "",
		"stackTrace": []
		}
		errorMessage：Runtime返回的错误信息
		errorType：错误类型
		stackTrace：Runtime返回的堆栈异常报错信息

执行摘要	显示请求ID、配置内存、执行时长、实际使用内存和收费时长。	显示请求ID、配置内存、执行时长、实际使用内存和收费时长。
日志	打印函数日志，最多显示4KB的日志。	打印报错信息，最多显示4KB的日志。

4.2 python 模板

```
# -*- coding:utf-8 -*-
import json
def handler (event, context):
    return {
        "statusCode": 200,
        "isBase64Encoded": False,
        "body": json.dumps(event),
        "headers": {
            "Content-Type": "application/json"
        }
    }
```

4.3 制作依赖包

制作函数依赖包推荐在Huawei Cloud EulerOS 2.0环境中进行。使用其他系统打包可能会因为底层依赖库的原因，运行出问题，比如找不到动态链接库。

约束与限制

如果安装的依赖模块需要添加依赖库，请将依赖库归档到zip依赖包文件中，例如，添加.dll、.so、.a等依赖库。

为 Python 函数制作依赖包

打包环境中的Python版本要和对应函数的运行时版本相同，如Python2.7建议使用2.7.12及以上版本，Python3.6建议使用3.6.3以上版本。

为Python 2.7安装PyMySQL依赖包，并指定此依赖包的安装路径为本地的/tmp/pymysql下，可以执行如下命令。


```
pip install PyMySQL --root /tmp/pymysql
```

执行成功后，执行以下命令。

```
cd /tmp/pymysql/
```

进入子目录直到site-packages路径下（一般路径为usr/lib64/python2.7/site-packages/），接下来执行以下命令。

```
zip -rq pymysql.zip *
```

所生成的包即为最终需要的依赖包。

说明

如果需要安装存放在的本地wheel安装包，直接输入：

```
pip install piexif-1.1.0b0-py2.py3-none-any.whl --root /tmp/piexif
//安装包名称以piexif-1.1.0b0-py2.py3-none-any.whl为例，请以实际安装包名称为准
```


5 Java

5.1 开发事件函数

5.1.1 Java 函数开发指南（使用 Eclipse 工具）

Java 函数接口定义

Java函数接口定义：*作用域 返回参数 函数名（函数参数，Context参数）*

- 作用域：提供给FunctionGraph调用的用户函数必须定义为public。
- 返回参数：用户定义，FunctionGraph负责转换为字符串，作为HTTP Response返回。对于返回参数对象类型，HTTP Response该类型的JSON字符串。
- 函数名：用户定义函数名称。
- 函数参数：用户定义参数，当前函数仅支持一个用户参数。对于复杂参数，建议定义为对象类型，以JSON字符串提供数据。FunctionGraph调用函数时，解析JSON为对象。
- Context：runtime提供函数执行上下文，其接口定义在**SDK接口**说明。

Java函数的函数执行入口参数格式为：**[包名].[类名].[执行函数名]**，请参考**函数执行入口**进行配置或修改。

约束与限制

getToken()、getAccessKey()和getSecretKey()方法返回的内容包含敏感信息，请谨慎使用，避免造成敏感信泄露。

Java 的 initializer 入口介绍

函数工作流服务目前支持以下Java运行环境。

- Java 8 (runtime = Java8)
- Java 11（runtime = Java11）

Initializer格式为：**[包名].[类名].[执行函数名]**

示例：创建函数时指定的initializer为com.huawei.Demo.my_initializer，那么FunctionGraph会去加载com.huawei包，Demo类中定义的my_initializer函数。

在函数工作流服务中使用Java实现initializer接口，需要定义一个java函数作为initializer入口，以下为initializer的简单示例。

```
public void my_initializer(Context context)
{
RuntimeLogger log = context.getLogger();
log.log(String.format("ak:%s", context.getAccessKey()));
}
```

- 函数名

  my_initializer需要与实现initializer接口时的initializer字段相对应。

- context参数

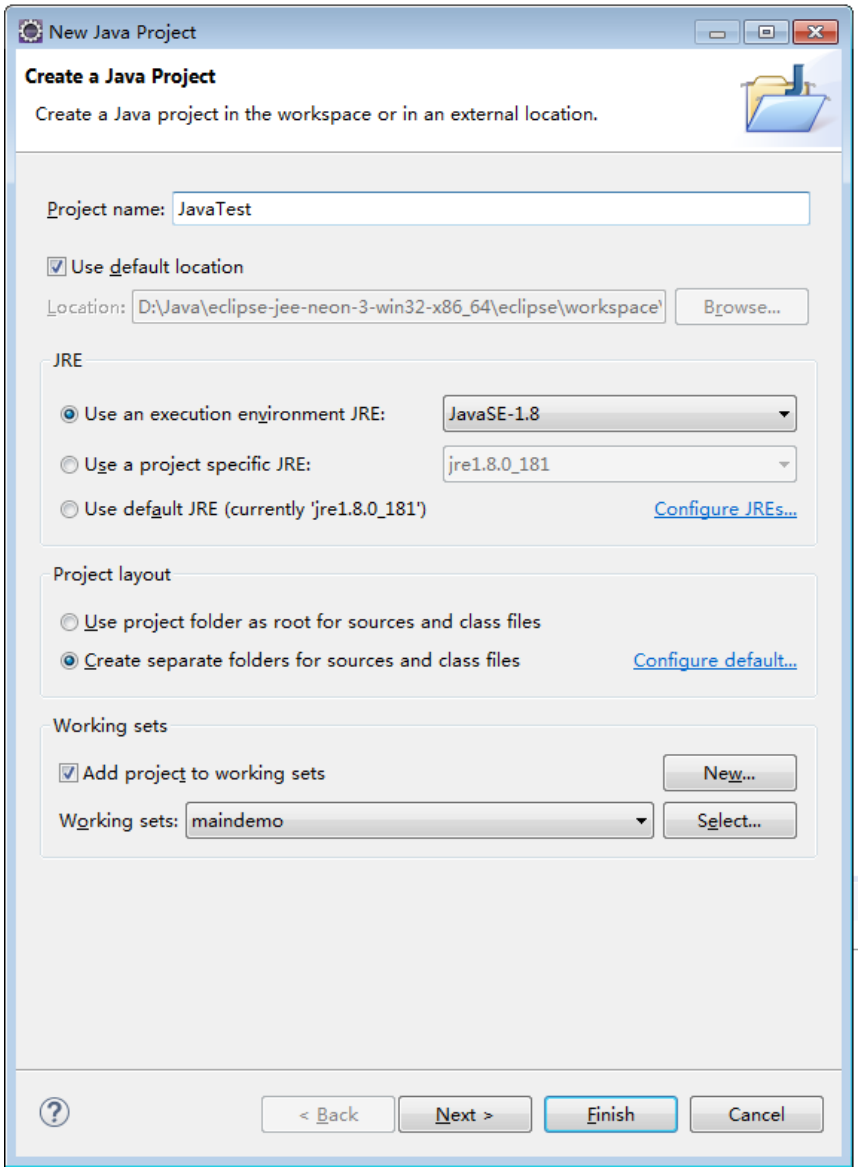
  context参数中包含一些函数的运行时信息，例如：request id、临时AccessKey、function meta等。

步骤一：使用 Eclipse 创建 Java 工程

1. 配置Eclipse

   如图5-1所示，创建Java工程，工程命名为JavaTest。

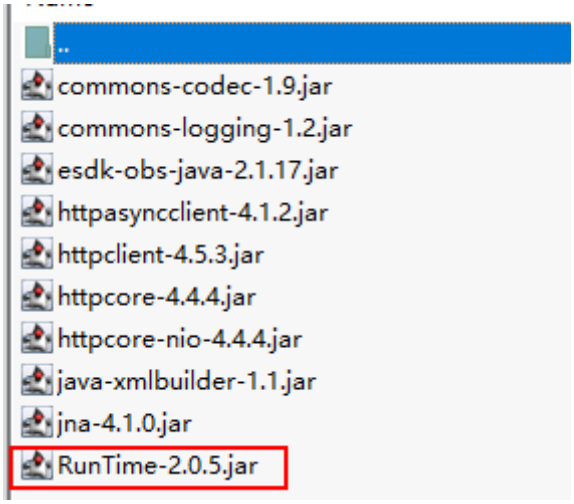
图 5-1 创建工程



2. 添加工程依赖

   如**图5-2**所示，根据**Java SDK下载**提供的SDK地址，下载JavaRuntime SDK到本地开发环境解压。

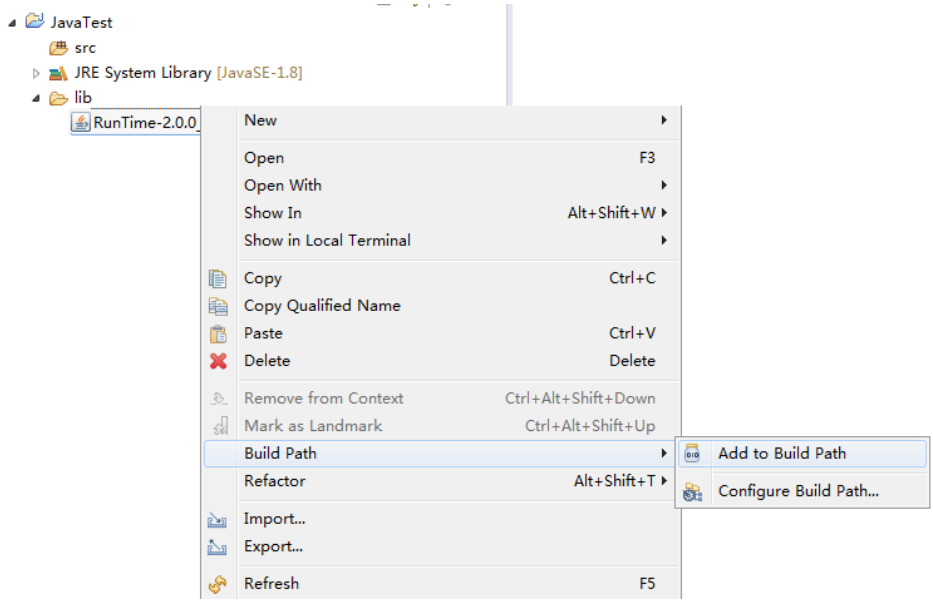
图 5-2 下载 SDK 解压



3. 配置依赖

   如图5-3所示，在工程目录下创建lib目录，将zip中的Runtime-2.0.5.jar拷贝到该目录中，并把该jar添加为工程依赖。

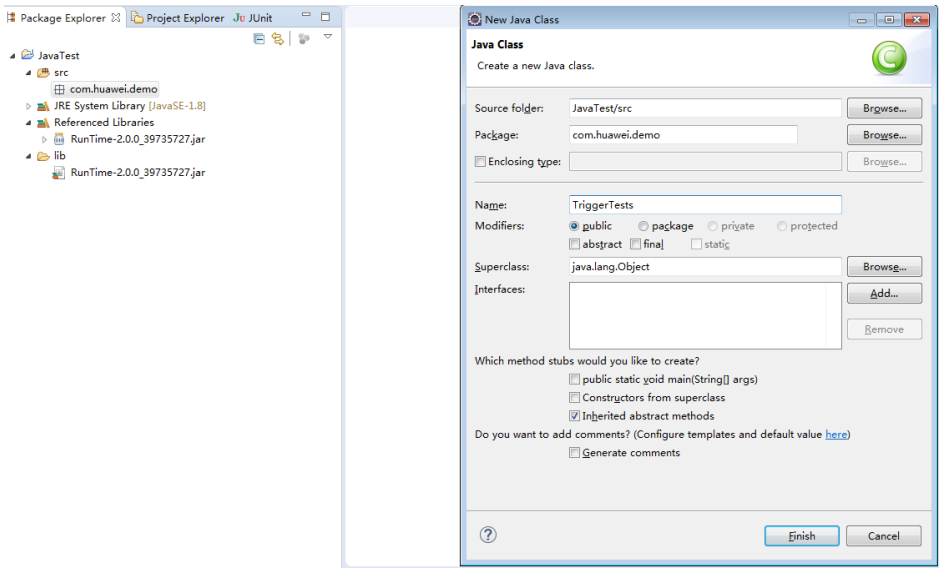
图 5-3 配置依赖



4. 配置函数资源

   如图5-4所示，创建包com.huawei.demo，并在包下创建TriggerTests类。

图 5-4 创建 com.huawei.demo



5. 配置函数代码

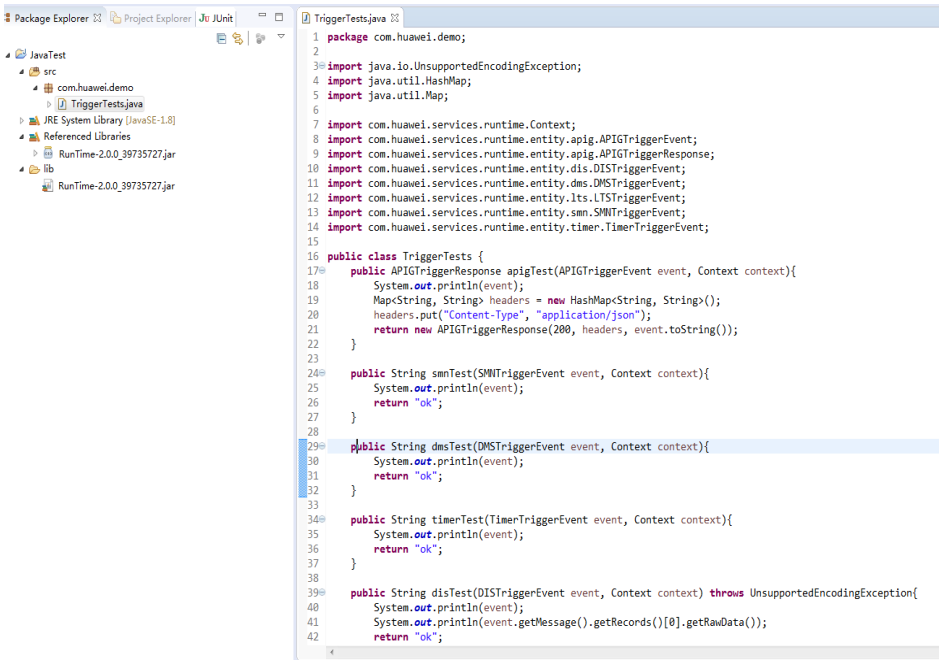
如图5-5所示，在TriggerTests.java中定义函数执行入口，示例代码如下。

```
package com.huawei.demo;

import java.io.UnsupportedEncodingException;
import java.util.HashMap;
import java.util.Map;

import com.huawei.services.runtime.Context;
import com.huawei.services.runtime.entity.apig.APIGTriggerEvent;
import com.huawei.services.runtime.entity.apig.APIGTriggerResponse;
import com.huawei.services.runtime.entity.dis.DISTriggerEvent;
import com.huawei.services.runtime.entity.dms.DMSTriggerEvent;
import com.huawei.services.runtime.entity.lts.LTSTriggerEvent;
import com.huawei.services.runtime.entity.smn.SMNTriggerEvent;
import com.huawei.services.runtime.entity.timer.TimerTriggerEvent;

public class TriggerTests {
    public APIGTriggerResponse apigTest(APIGTriggerEvent event, Context context){
        System.out.println(event);
        Map<String, String> headers = new HashMap<String, String>();
        headers.put("Content-Type", "application/json");
        return new APIGTriggerResponse(200, headers, event.toString());
    }

    public String smnTest(SMNTriggerEvent event, Context context){
        System.out.println(event);
        return "ok";
    }

    public String dmsTest(DMSTriggerEvent event, Context context){
        System.out.println(event);
        return "ok";
    }

    public String timerTest(TimerTriggerEvent event, Context context){
        System.out.println(event);
        return "ok";
    }

    public String disTest(DISTriggerEvent event, Context context) throws
UnsupportedEncodingException{
        System.out.println(event);
        System.out.println(event.getMessage().getRecords()[0].getRawData());
```



```
        return "ok";
    }

    public String ltsTest(LTSTriggerEvent event, Context context) throws
UnsupportedEncodingException {
        System.out.println(event);
        System.out.println("raw data: " + event.getLts().getRawData());
        return "ok";
    }
}
```

图 5-5 定义函数运行入口

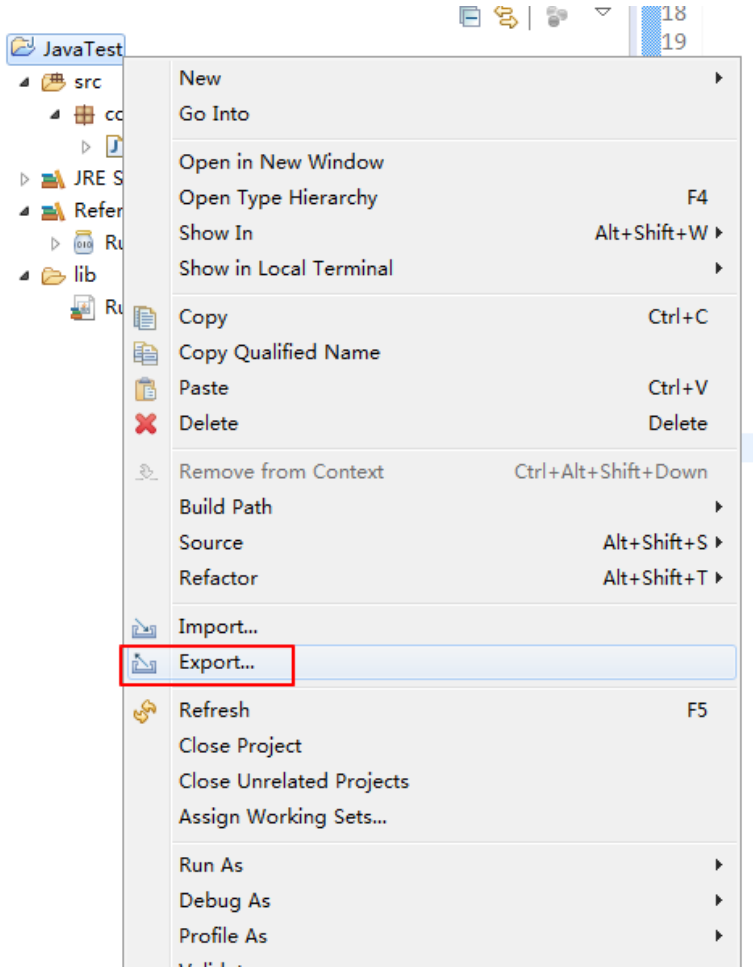


示例代码中添加了多个入口函数，分别使用了不同的触发器事件类型，可通过函数工作流控制台修改函数执行入口测试不同的入口函数。**当函数的事件源是APIG时，相关约束条件请参考Base64解码和返回结构体的说明**。

步骤二：打包 Java 工程

1. 如图5-6所示，右击工程，选择“Export”。

图 5-6 打包



2. 如图5-7、图5-8所示，选择导出为jar文件，设置导出目录。

图 5-7 选择

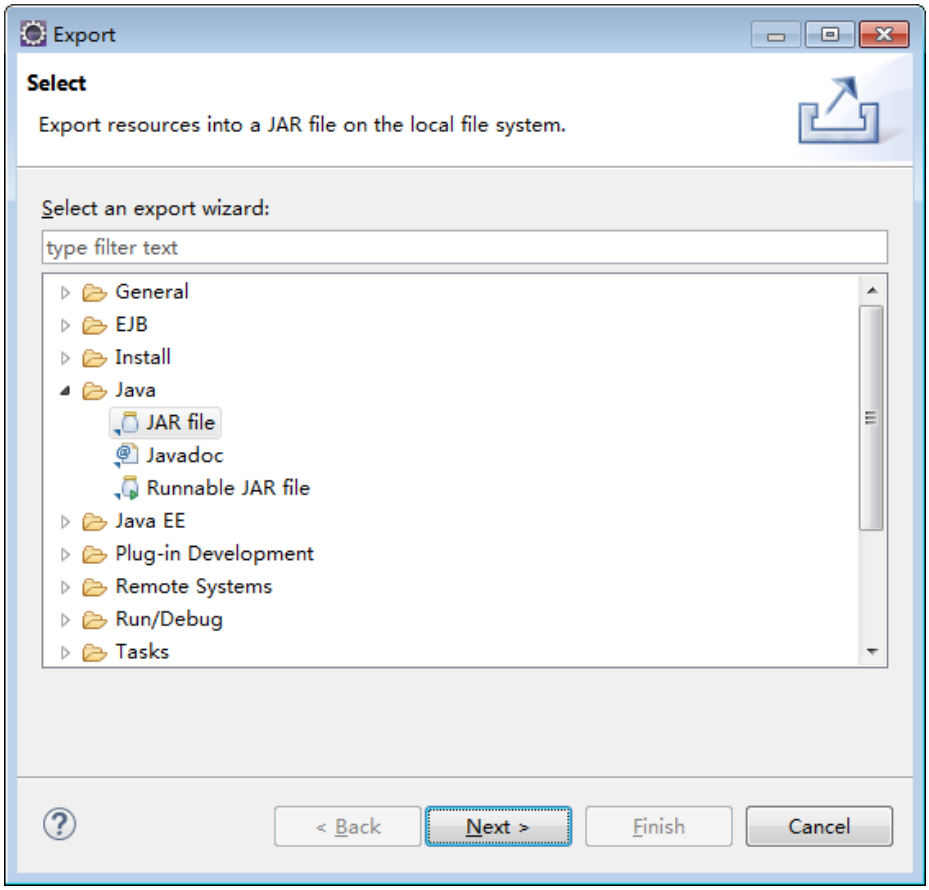
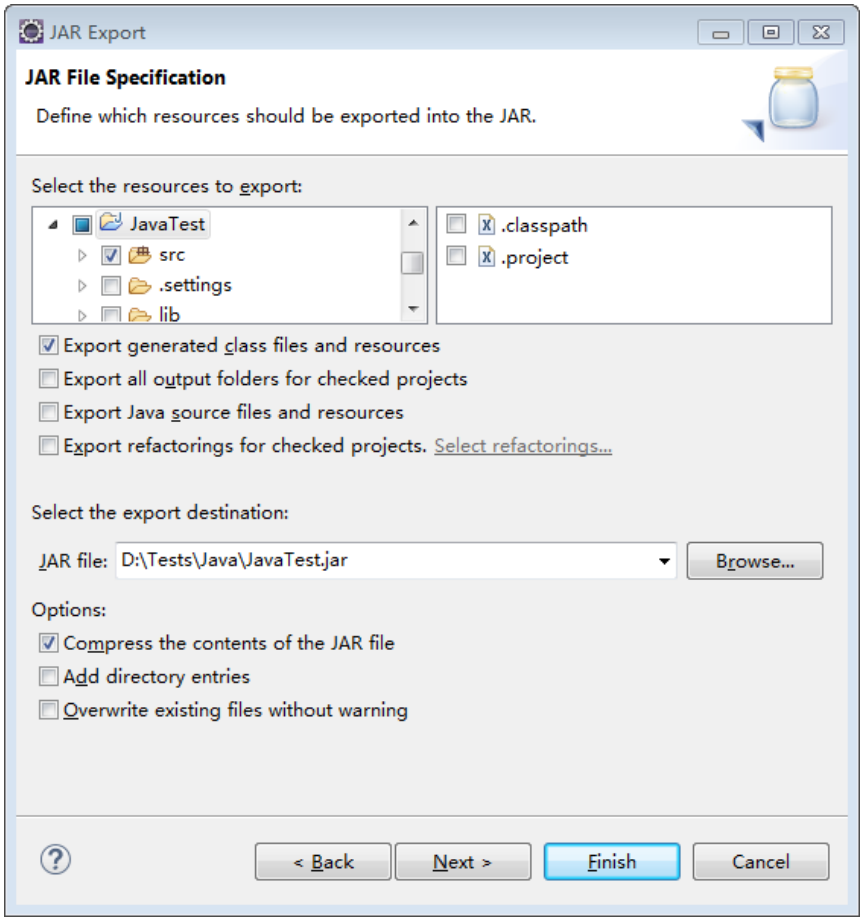


图 5-8 导出



步骤三：创建 Java 函数并测试

1. 登录**函数工作流控制台**，左侧导航栏选择“函数 > 函数列表”，单击右上角“创建函数”进入创建函数页面，选择“创建空白函数”。
2. 如**图5-9**所示，配置函数基本信息，配置完成后单击右下角“创建函数”完成创建。

图 5-9 创建 Java 函数

3. 进入函数详情页，在“代码”页签下，单击右侧“上传代码 > JAR文件 ”，如图5-10所示，添加步骤二：打包Java工程的JAR文件上传。

图 5-10 上传 JAR 文件

4. 上传成功后，单击代码编辑区的“测试”，选择“创建新的测试事件”，在云事件模板列表中选择“API 网关服务”，单击“创建”。
5. 单击“测试”，查看函数执行结果。

   函数执行结果分为三部分，分别为函数返回（由callback返回）、执行摘要、日志输出（由console.log或getLogger()方法获取的日志方法输出），参考表5-1查看结果说明。

表 5-1 执行结果说明

参数项	执行成功	执行失败

函数返回	返回函数中定义的返回信息。	返回包含错误信息和堆栈异常报错信息的JSON文件。格式如下：
		{
		"errorMessage": "",
		"stackTrace": []
		}
		errorMessage：Runtime返回的错误信息
		stackTrace：Runtime返回的堆栈异常报错信息

执行摘要	显示请求ID、配置内存、执行时长、实际使用内存和收费时长。	显示请求ID、配置内存、执行时长、实际使用内存和收费时长。
日志	打印函数日志，最多显示4KB的日志。	打印报错信息，最多显示4KB的日志。

6. 进入“设置 > 触发器”页签，单击“创建触发器”，参考创建APIG触发器创建一个APIG触发器，并访问调用URL，调用结果如图5-11所示。

图 5-11 访问调用 URL

说明

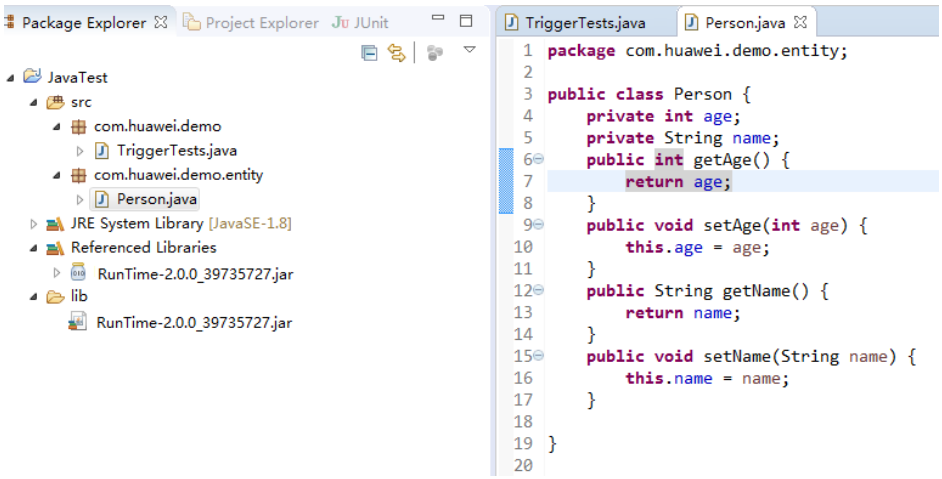
如需测试示例代码中的其他事件源触发器，例如SMN事件源，可在FunctionGraph控制台，参考**修改函数执行入口**将函数执行入口修改为com.huawei.demo.TriggerTests.smnTest，并参考以上步骤，创建新的消息通知服务SMN测试事件和SMN触发器进行函数测试。

使用自定义参数类型示例

以下介绍在Java中使用自定义参数类型，可参考该示例自定义创建Java函数。

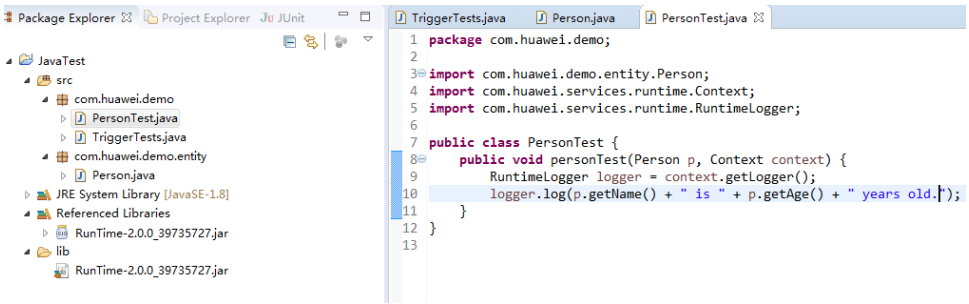
1. 如**图5-12**所示，在工程中新建Person类。

图 5-12 新建 Person 类



2. 如**图5-13**所示，新建PersonTest.java，并在其中写入函数执行入口。

图 5-13 新建 PersonTest.java



3. 参见**步骤二：打包Java工程**打包工程，并在函数详情界面上传新的JAR包，再修改函数执行入口为“com.huawei.demo.PersonTest.personTest”并保存。
4. 上传成功后，在“代码”页签下配置新的测试事件，选择“空白模板”，自定义输入测试事件的内容，单击“创建”后执行测试。

SDK 接口

FunctionGraph函数JavaSDK提供了Event事件接口、Context接口和日志记录接口，SDK下载地址见**Java SDK下载**（校验文件：**fss-java-sdk-2.0.5.sha256**）。

- Event事件接口

  Java SDK引入了触发器事件结构体定义，当前支持CTS、DMS、DIS、SMN、LTS、TIMER、APIG、Kafka。在需要使用触发器的场景下，编写相应代码更简便。

  a. **APIG触发器相关方法说明**

  - APIGTriggerEvent相关方法说明

表 5-2 APIGTriggerEvent 相关方法说明

方法名	方法说明
isBase64Encoded()	Event中的body是否是base64编码。
getHttpMethod()	获取Http请求方法。
getPath()	获取Http请求路径。
getBody()	获取Http请求body。
getPathParameters()	获取所有路径参数。
getRequestContext()	获取相关的APIG配置。（返回**APIGRequestContext对象**）
getHeaders()	获取Http请求头。
getQueryStringParameters()	获取查询参数。 **当前查询参数不支持取值为数组，如果查询参数的取值需要为数组，请自定义对应的触发器事件结构体。**
getRawBody()	获取base64编码前的内容。
getUserData()	获取APIG自定义认证中设置的userdata。

表 5-3 APIGRequestContext 相关方法说明

方法名	方法说明
getApiId()	获取API的ID。
getRequestId()	获取此次API请求的requestId。
getStage()	获取发布环境名称。

方法名	方法说明
getSourceIp()	获取APIG自定义认证信息中的源IP。

- APIGTriggerResponse相关方法说明

表 5-4 APIGTriggerResponse 构造方法说明

方法名	方法说明
无参构造APIGTriggerResponse()	其中headers设置为null，statusCode设置为200，body设置为" "，isBase64Encoded设置为false。
三个参数构造APIGTriggerResponse(statusCode, headers, body)	isBase64Encoded设置为false，其他均以输入为准。
四个参数构造APIGTriggerResponse(statusCode, headers, isBase64Encoded, body)	按照对应顺序设置值即可。

表 5-5 APIGTriggerResponse 相关方法说明

方法名	方法说明
setBody(String body)	设置消息体。
setHeaders(Map<String,String> headers)	设置最终返回的Http响应头。
setStatusCode(int statusCode)	设置Http状态码。
setBase64Encoded(boolean isBase64Encoded)	设置body是否经过base64编码。
setBase64EncodedBody(String body)	将输入进行base64编码，并设置到Body中。
addHeader(String key, String value)	增加一组Http header。
removeHeader(String key)	从现有的header中移除指定header。
addHeaders(Map<String,String> headers)	增加多个header。

> **说明**
>
> APIGTriggerResponse有headers属性，可以通过setHeaders方法和带有headers参数的构造函数对齐进行初始化。

b. **DIS触发器相关方法说明**

表 5-6 DISTriggerEvent 相关方法说明

方法名	方法说明
getShardID()	获取分区ID。
getMessage()	获取DIS消息体。（**DISMessage结构**）
getTag()	获取函数版本。
getStreamName()	获取通道名称。

表 5-7 DISMessage 相关方法说明

方法名	方法说明
getNextPatitionCursor()	获取下一个游标。
getRecords()	获取消息记录。（**DISRecord结构**）
getMillisBehindLatest()	保留方法（目前返回0）。

表 5-8 DISRecord 相关方法说明

方法名	方法说明
getPartitionKey()	获取数据分区。
getData()	获取数据。
getRawData()	data经过base64解码后的字符串，UTF-8编码。
getSequenceNumber()	获取序列号（每个记录的唯一标识）。

c. **DMS触发器相关方法说明**

表 5-9 DMSTriggerEvent 相关方法说明

方法名	方法说明
getQueueId()	获取队列ID。

方法名	方法说明
getRegion()	获取区域名称。
getEventType()	获取事件类型。（返回“MessageCreated”）
getConsumerGroupId()	获取消费组ID。
getMessages()	获取DMS消息。（**DMSMessage结构**）

表 5-10 DMSMessage 相关方法说明

方法名	方法说明
getBody()	获取DMS消息体。
getAttributes()	获取DMS消息属性集合。

d. **SMN触发器相关方法说明**

表 5-11 SMNTriggerEvent 相关方法说明

方法名	方法说明
getRecord()	获取消息记录集合。（**SMNRecord结构**）

表 5-12 SMNRecord 相关方法说明

方法名	方法说明
getEventVersion()	获取事件版本。（当前为1.0）
getEventSubscriptionUrn()	获取订阅URN。
getEventSource()	获取事件源。
getSmn()	获取SMN事件消息体。（**SMNBody结构**）

表 5-13 SMNBody 相关方法说明

方法名	方法说明
getTopicUrn()	获取SMN主题URN。
getTimeStamp()	获取消息时间戳。

方法名	方法说明
getMessageAtrributes()	获取消息属性集合。
getMessage()	获取消息体。
getType()	获取消息类型。
getMessageId()	获取消息ID。
getSubject()	获取消息主题。

e. **定时触发器相关方法说明**

表 5-14 TimerTriggerEvent 相关方法说明

方法名	方法说明
getVersion()	获取版本名称。（当前为“v1.0”）
getTime()	获取当前时间。
getTriggerType()	获取触发器类型。（“Timer”）
getTriggerName()	获取触发器名称。
getUserEvent()	获取触发器附加信息。

f. **LTS触发器相关方法说明**

表 5-15 LTSTriggerEvent 相关方法说明

方法名	方法说明
getLts()	获取LTS消息。（**LTSBody结构**）

表 5-16 LTSBody 相关方法说明

方法名	方法说明
getData()	获取LTS原始消息。
getRawData()	获取经过base64解码后的消息，UTF-8编码。

g. **CTS触发器相关方法说明**

表 5-17 CTSTriggerEvent 说明

方法名	方法说明
getCTS()	获取CTS消息体。（CTS结构）

表 5-18 CTS 结构相关方法说明

方法名	方法说明
getTime()	获取事件产生时间。
getUser()	获取触发该事件的用户信息（User结构）。
getRequest()	获取事件请求内容。
getResponse()	获取事件响应内容。
getCode()	获取响应码。
getServiceType()	获取事件触发的服务名称。
getResourceType()	获取事件触发的资源类型。
getResourceName()	获取事件触发的资源名称。
getResourceId()	获取事件触发资源的唯一标识。
getTraceName()	获取事件名称。
getTraceType()	获取事件触发的方式。（如ConsoleAction：代表前台操作）
getRecordTime()	获取CTS服务接收事件时间。
getTraceId()	获取当前事件的唯一标识。
getTraceStatus()	获取事件状态。

表 5-19 User 方法说明

方法名	方法说明
getName()	获取用户名。（同一账号可以创建多个子用户）
getId()	获取用户ID。
getDomain()	获取账号信息。

表 5-20 Domain 方法说明

方法名	方法说明
getName()	获取账号名称。
getId()	获取账号ID。

h. **Kafka触发器相关方法说明**

表 5-21 Kafka 触发器相关方法说明

方法名	方法说明
getEventVersion	获取事件版本。
getRegion	获取地区。
getEventTime	获取产生时间。
getTriggerType	获取触发器类型。
getInstanceId	获取实例ID。
getRecords	获取记录体。

说明

1. 例如使用APIG触发器时，只需要把入口函数（假如函数名为handler）的第一个参数按照如下方式设置：handler(APIGTriggerEvent event, Context context)。
2. 关于所有TriggerEvent，上面提到的TriggerEvent方法均有与之对应的set方法，建议在本地调试时使用；DIS和LTS均有对应的getRawData()方法，但无与之相应的setRawData()方法。

- Context接口

  Context接口提供函数获取函数执行上下文，例如，用户委托的AccessKey/SecretKey、当前请求ID、函数执行分配的内存空间、CPU数等。

  Context接口说明如表5-22所示。

表 5-22 Context 类上下文方法说明

方法名	方法说明
getRequestID( )	获取请求ID。
getRemainingTimeInMilliSeconds ( )	获取函数剩余运行时间。
getAccessKey( )	获取用户委托的AccessKey（有效期24小时），使用该方法需要给函数配置委托。 **当前函数工作流已停止维护Runtime SDK 中getAccessKey接口，您将无法使用getAccessKey获取临时AK。**
getSecretKey( )	获取用户委托的SecretKey（有效期24小时），使用该方法需要给函数配置委托。 **当前函数工作流已停止维护Runtime SDK 中getSecretKey接口，您将无法使用getSecretKey获取临时SK。**
getSecurityAccessKey( )	获取用户委托的SecurityAccessKey（有效期24小时），使用该方法需要给函数配置委托。

方法名	方法说明
getSecuritySecretKey( )	获取用户委托的SecuritySecretKey（有效期24小时），使用该方法需要给函数配置委托。
getSecurityToken( )	获取用户委托的SecurityToken（有效期24小时），使用该方法需要给函数配置委托。
getUserData(string key)	通过key获取用户通过环境变量传入的值。
getFunctionName( )	获取函数名称。
getRunningTimeInSeconds ( )	获取函数超时时间。
getVersion( )	获取函数的版本。
getMemorySize( )	分配的内存。
getCPUNumber( )	获取函数占用的CPU资源。
getPackage( )	获取函数组。
getToken( )	获取用户委托的token（有效期24小时），使用该方法需要给函数配置委托。
getLogger( )	获取context提供的logger方法（默认会输出时间、请求ID等信息）。
getAlias	获取函数的别名

- 日志接口

  Java SDK日志接口日志说明如表5-23所示。

  表 5-23 日志接口说明表

方法名	方法说明
RuntimeLogger()	记录用户输入日志。包含方法如下：log(String string)。

5.1.2 Java 函数开发指南（使用 IDEA 工具普通 Java 项目）

Java 函数接口定义

Java函数接口定义：*作用域 返回参数 函数名（函数参数，Context参数）*

- 作用域：提供给FunctionGraph调用的用户函数必须定义为public。
- 返回参数：用户定义，FunctionGraph负责转换为字符串，作为HTTP Response返回。对于返回参数对象类型，HTTP Response该类型的JSON字符串。
- 函数名：用户定义函数名称。

- 函数参数：用户定义参数，当前函数仅支持一个用户参数。对于复杂参数，建议定义为对象类型，以JSON字符串提供数据。FunctionGraph调用函数时，解析JSON为对象。
- Context：runtime提供函数执行上下文，其接口定义在**SDK接口**说明。

Java函数的函数执行入口参数格式为：**[包名].[类名].[执行函数名]**，请参考**函数执行入口**进行配置。

约束与限制

getToken()、getAccessKey()和getSecretKey()方法返回的内容包含敏感信息，请谨慎使用，避免造成敏感信泄露。

Java 的 initializer 入口介绍

函数工作流服务目前支持以下Java运行环境。

- Java 8 (runtime = Java8)
- Java 11（ runtime = Java11 ）

Initializer格式为：**[包名].[类名].[执行函数名]**

示例：创建函数时指定的initializer为com.huawei.Demo.my_initializer，那么FunctionGraph会去加载com.huawei包，Demo类中定义的my_initializer函数。

在函数工作流服务中使用Java实现initializer接口，需要定义一个java函数作为initializer入口，以下为initializer的简单示例。

```
public void my_initializer(Context context)
{
RuntimeLogger log = context.getLogger();
log.log(String.format("ak:%s", context.getAccessKey()));
}
```

- 函数名

  my_initializer需要与实现initializer接口时的initializer字段相对应。
- context参数

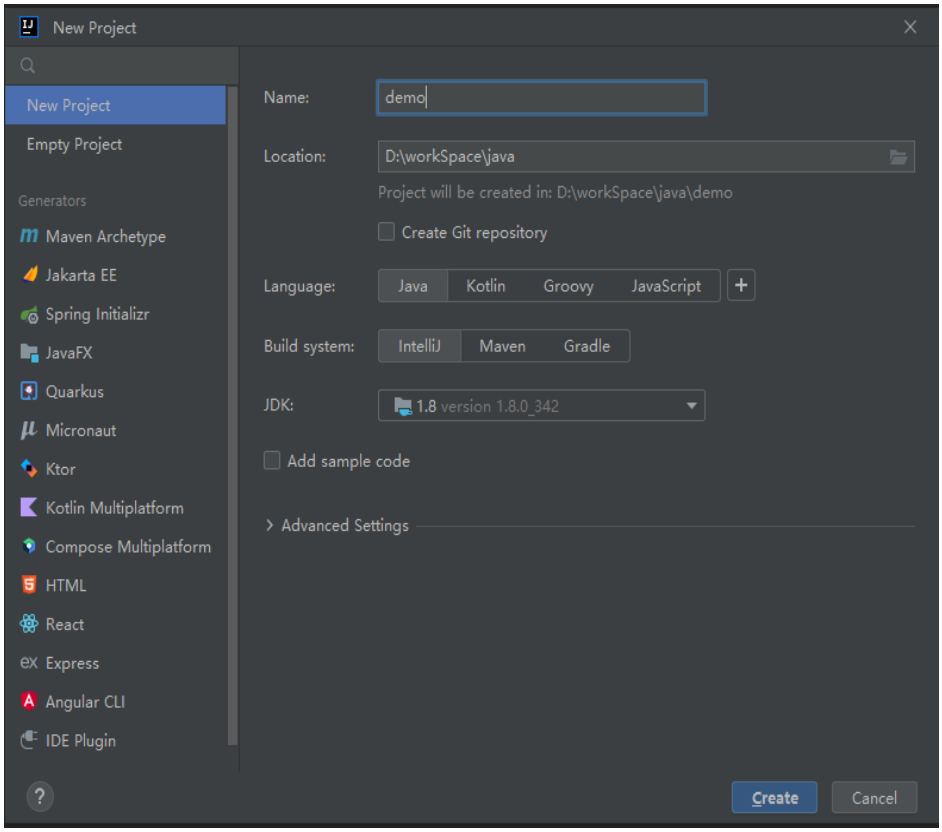
  context参数中包含一些函数的运行时信息，例如：request id、临时AccessKey、function meta等。

步骤一：使用 IDEA 创建 Java 工程

1. 配置IDEA

   如图**创建工程**所示，创建java工程JavaTest。

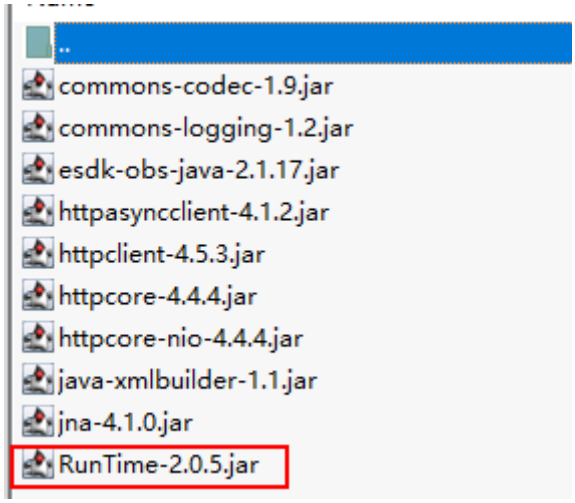
图 5-14 创建工程



2. 添加工程依赖

   根据**Java SDK下载**提供的SDK地址，下载JavaRuntime SDK到本地开发环境解压，如**图5-15**所示。

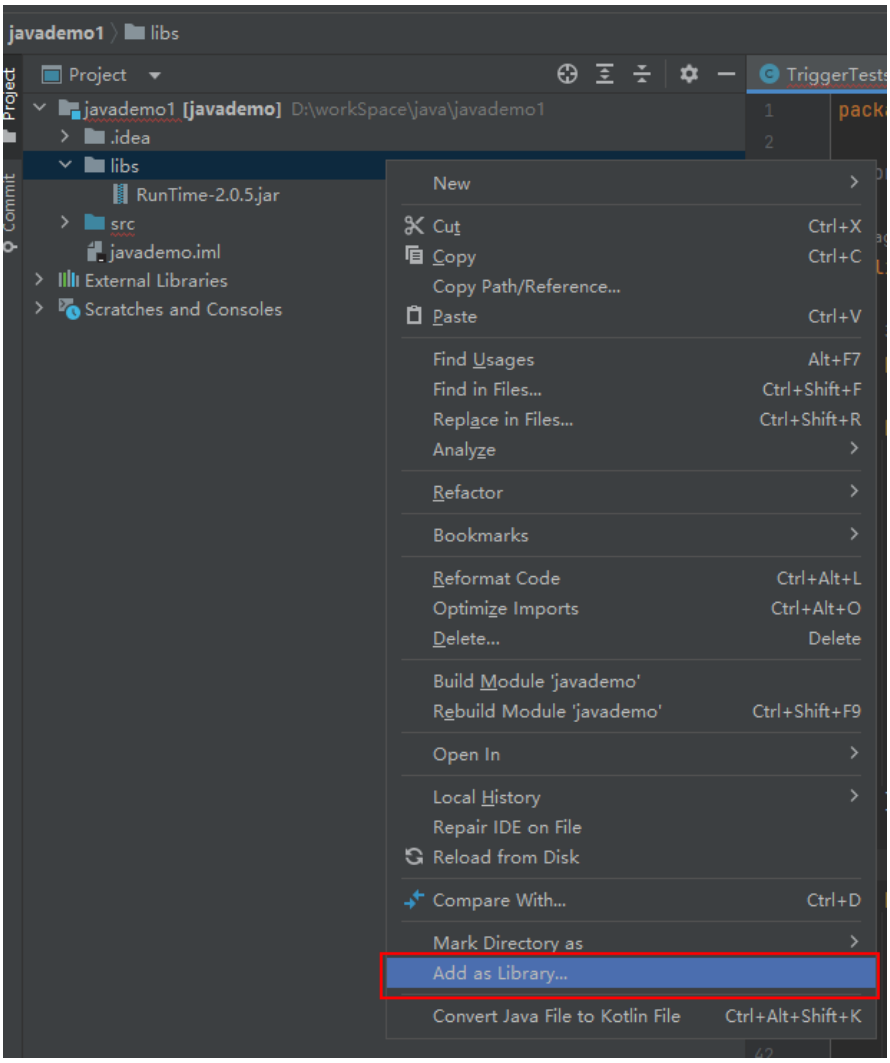
图 5-15 下载 SDK 解压



3. 配置依赖

   在工程目录下创建lib目录，将zip中的Runtime2.0.5.jar和代码所需要的三方依赖包拷贝到该目录，并把该jar添加为工程依赖，如**图5-16**所示。

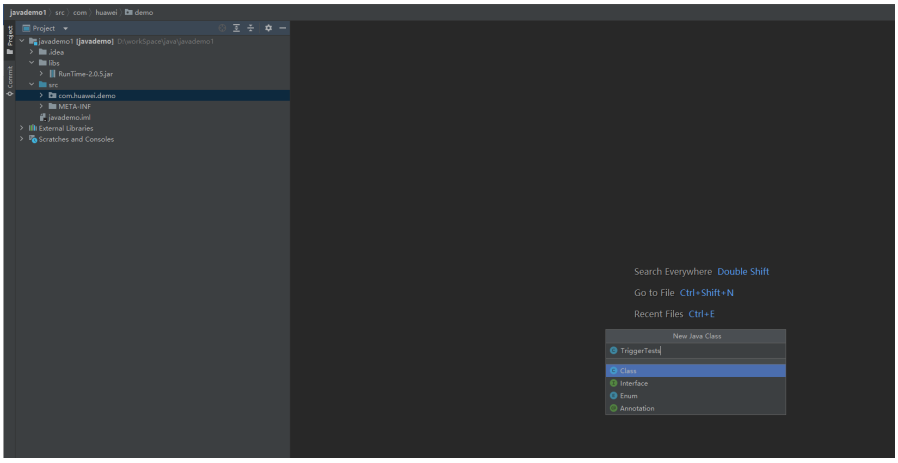
图 5-16 配置依赖



4. 配置函数资源

   创建包com.huawei.demo，并在包下创建TriggerTests类，如图5-17所示。

图 5-17 创建 TriggerTests 类

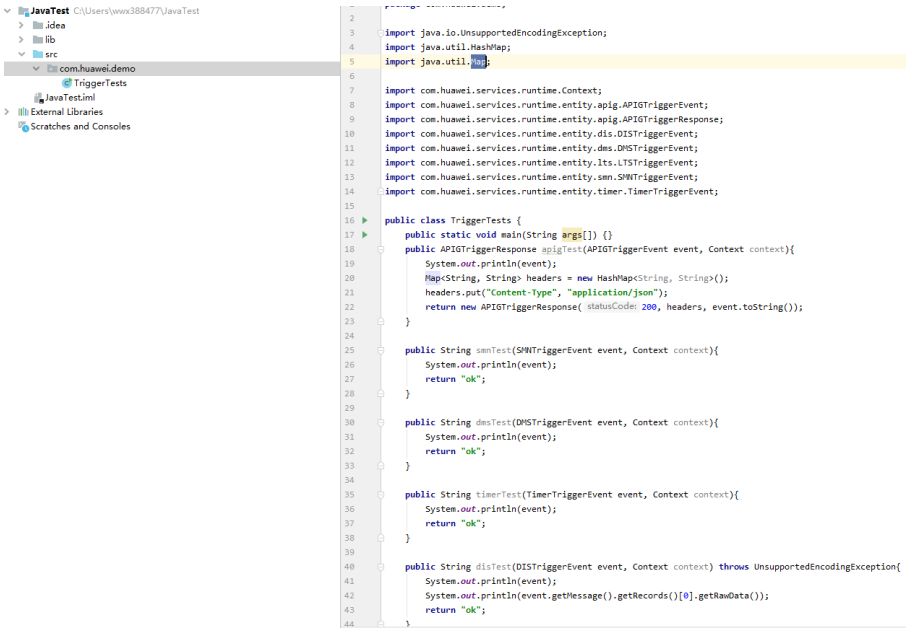


5. 配置函数代码

如图5-18所示，在TriggerTests.java中定义函数运行入口，示例代码如下。（**普通java项目需要通过Artifacts来进行编译，因此需要定义一个main函数。**）

```
package com.huawei.demo;

import java.io.UnsupportedEncodingException;
import java.util.HashMap;
import java.util.Map;

import com.huawei.services.runtime.Context;
import com.huawei.services.runtime.entity.apig.APIGTriggerEvent;
import com.huawei.services.runtime.entity.apig.APIGTriggerResponse;
import com.huawei.services.runtime.entity.dis.DISTriggerEvent;
import com.huawei.services.runtime.entity.dms.DMSTriggerEvent;
import com.huawei.services.runtime.entity.lts.LTSTriggerEvent;
import com.huawei.services.runtime.entity.smn.SMNTriggerEvent;
import com.huawei.services.runtime.entity.timer.TimerTriggerEvent;

public class TriggerTests {
    public static void main(String args[]) {}
    public APIGTriggerResponse apigTest(APIGTriggerEvent event, Context context){
        System.out.println(event);
        Map<String, String> headers = new HashMap<String, String>();
        headers.put("Content-Type", "application/json");
        return new APIGTriggerResponse(200, headers, event.toString());
    }

    public String smnTest(SMNTriggerEvent event, Context context){
        System.out.println(event);
        return "ok";
    }

    public String dmsTest(DMSTriggerEvent event, Context context){
        System.out.println(event);
        return "ok";
    }

    public String timerTest(TimerTriggerEvent event, Context context){
        System.out.println(event);
        return "ok";
    }

    public String disTest(DISTriggerEvent event, Context context) throws
UnsupportedEncodingException{
        System.out.println(event);
        System.out.println(event.getMessage().getRecords()[0].getRawData());
        return "ok";
    }

    public String ltsTest(LTSTriggerEvent event, Context context) throws
UnsupportedEncodingException {
        System.out.println(event);
        event.getLts().getData();
        System.out.println("raw data: " + event.getLts().getRawData());
        return "ok";
    }
}
```


图 5-18 定义函数运行入口

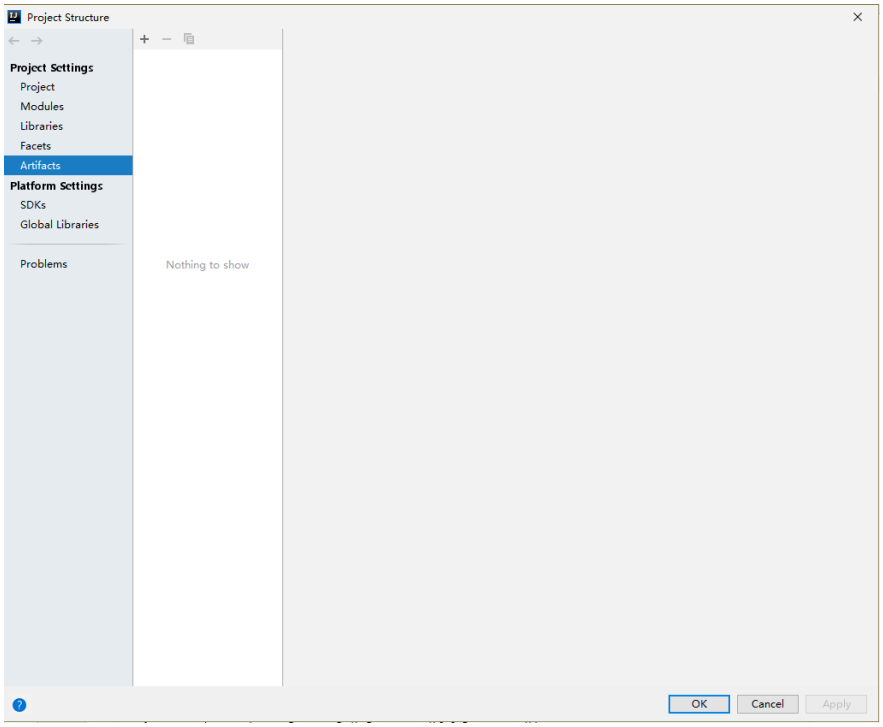


示例代码中添加了多个入口函数，分别使用了不同的触发器事件类型，可通过函数工作流控制台**修改函数执行入口**测试不同的入口函数。**当函数的事件源是APIG时，相关约束条件请参考Base64解码和返回结构体的说明**。

步骤二：打包 Java 工程

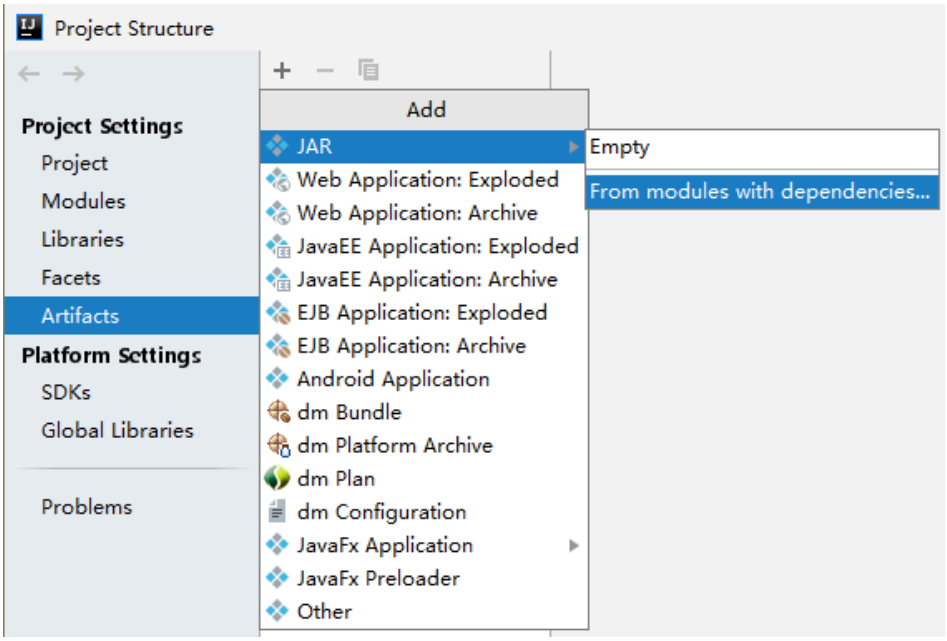
1. 单击“File > Project Structure”打开Project Structure窗口，如**图5-19**所示。

图 5-19 Project Structure



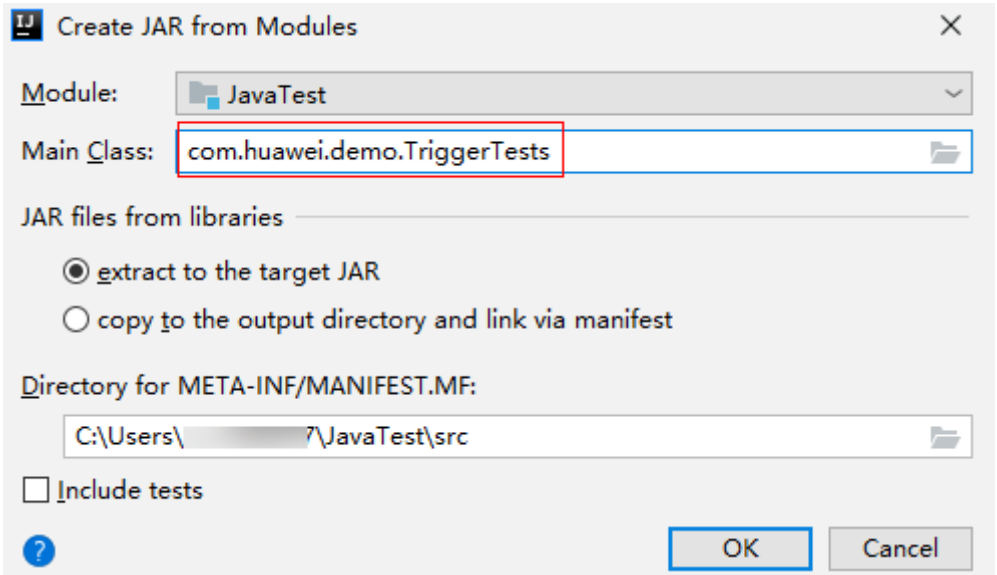
2. 选择上图中的“Artifacts”，单击“+”，进入添加“Artifacts”窗口，如图5-20所示。

图 5-20 添加 Artifacts



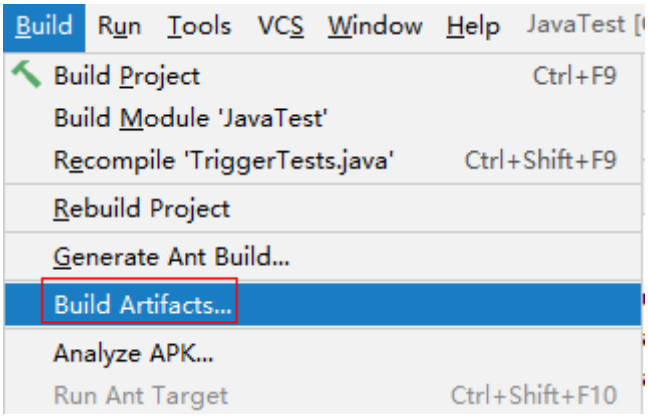
3. 添加“Main Class”，如图5-21所示。

图 5-21 添加 Main Class



4. 单击“Build > Build Artifacts”来编译JAR包，如图5-22所示。

图 5-22 Build Artifacts



步骤三：创建 Java 函数并测试

1. 登录**函数工作流控制台**，左侧导航栏选择“函数 > 函数列表”，单击右上角“创建函数”进入创建函数页面，选择“创建空白函数”。
2. 如**图5-23**所示，配置函数基本信息，配置完成后单击右下角“创建函数”完成创建。

图 5-23 创建 Java 函数

3. 进入函数详情页，在“代码”页签下，单击右侧“上传代码 > JAR文件 ”，如**图5-24**所示，添加**步骤二：打包Java工程**的JAR文件上传。

图 5-24 上传 JAR 文件

4. 上传成功后，请参见**修改函数执行入口**修改函数执行入口，单击“保存”。
5. 回到“代码”页签，单击代码编辑区的“测试”，选择“创建新的测试事件”，在云事件模板列表中选择需要测试的事件模板，单击“创建”。
6. 单击“测试”，查看函数执行结果。

   函数执行结果分为三部分，分别为函数返回（由callback返回）、执行摘要、日志输出（由console.log或getLogger()方法获取的日志方法输出），参考**表5-24**查看结果说明。

表 5-24 执行结果说明

参数项	执行成功	执行失败

函数返回	返回函数中定义的返回信息。	返回包含错误信息和堆栈异常报错信息的JSON文件。格式如下：
		{
		"errorMessage": "",
		"stackTrace": []
		}
		errorMessage：Runtime返回的错误信息
		stackTrace：Runtime返回的堆栈异常报错信息

执行摘要	显示请求ID、配置内存、执行时长、实际使用内存和收费时长。	显示请求ID、配置内存、执行时长、实际使用内存和收费时长。
日志	打印函数日志，最多显示4KB的日志。	打印报错信息，最多显示4KB的日志。

说明

如需测试示例代码中的其他事件源触发器，例如SMN事件源，可在FunctionGraph控制台，参考**修改函数执行入口**将函数执行入口修改为com.huawei.demo.TriggerTests.smnTest，并参考以上步骤，创建新的消息通知服务SMN测试事件进行函数测试。

SDK 接口

FunctionGraph函数JavaSDK提供了Event事件接口、Context接口和日志记录接口，SDK下载地址见**Java SDK下载**（校验文件：**fss-java-sdk-2.0.5.sha256**）。

- Event事件接口

  Java SDK引入了触发器事件结构体定义，当前支持CTS、DMS、DIS、SMN、LTS、TIMER、APIG、Kafka。在需要使用触发器的场景下，编写相应代码更简便。

  a. **APIG触发器相关方法说明**

  - APIGTriggerEvent相关方法说明

表 5-25 APIGTriggerEvent 相关方法说明

方法名	方法说明
isBase64Encoded()	Event中的body是否是base64编码。
getHttpMethod()	获取Http请求方法。
getPath()	获取Http请求路径。
getBody()	获取Http请求body。
getPathParameters()	获取所有路径参数。
getRequestContext()	获取相关的APIG配置。（返回**APIGRequestContext对象**）
getHeaders()	获取Http请求头。
getQueryStringParameters()	获取查询参数。 **当前查询参数不支持取值为数组，如果查询参数的取值需要为数组，请自定义对应的触发器事件结构体。**
getRawBody()	获取base64编码前的内容。
getUserData()	获取APIG自定义认证中设置的userdata。

表 5-26 APIGRequestContext 相关方法说明

方法名	方法说明
getApiId()	获取API的ID。
getRequestId()	获取此次API请求的requestId。
getStage()	获取发布环境名称。

方法名	方法说明
getSourceIp()	获取APIG自定义认证信息中的源IP。

- APIGTriggerResponse相关方法说明

表 5-27 APIGTriggerResponse 构造方法说明

方法名	方法说明
无参构造APIGTriggerResponse()	其中headers设置为null，statusCode设置为200，body设置为" "，isBase64Encoded设置为false。
三个参数构造APIGTriggerResponse(statusCode, headers, body)	isBase64Encoded设置为false，其他均以输入为准。
四个参数构造APIGTriggerResponse(statusCode, headers, isBase64Encoded, body)	按照对应顺序设置值即可。

表 5-28 APIGTriggerResponse 相关方法说明

方法名	方法说明
setBody(String body)	设置消息体。
setHeaders(Map<String,String> headers)	设置最终返回的Http响应头。
setStatusCode(int statusCode)	设置Http状态码。
setBase64Encoded(boolean isBase64Encoded)	设置body是否经过base64编码。
setBase64EncodedBody(String body)	将输入进行base64编码，并设置到Body中。
addHeader(String key, String value)	增加一组Http header。
removeHeader(String key)	从现有的header中移除指定header。
addHeaders(Map<String,String> headers)	增加多个header。

> **说明**
>
> APIGTriggerResponse有headers属性，可以通过setHeaders方法和带有headers参数的构造函数对齐进行初始化。

b. **DIS触发器相关方法说明**

表 5-29 DISTriggerEvent 相关方法说明

方法名	方法说明
getShardID()	获取分区ID。
getMessage()	获取DIS消息体。（**DISMessage结构**）
getTag()	获取函数版本。
getStreamName()	获取通道名称。

表 5-30 DISMessage 相关方法说明

方法名	方法说明
getNextPatitionCursor()	获取下一个游标。
getRecords()	获取消息记录。（**DISRecord结构**）
getMillisBehindLatest()	保留方法（目前返回0）。

表 5-31 DISRecord 相关方法说明

方法名	方法说明
getPartitionKey()	获取数据分区。
getData()	获取数据。
getRawData()	data经过base64解码后的字符串，UTF-8编码。
getSequenceNumber()	获取序列号（每个记录的唯一标识）。

c. **DMS触发器相关方法说明**

表 5-32 DMSTriggerEvent 相关方法说明

方法名	方法说明
getQueueId()	获取队列ID。

方法名	方法说明
getRegion()	获取区域名称。
getEventType()	获取事件类型。（返回“MessageCreated”）
getConsumerGroupId()	获取消费组ID。
getMessages()	获取DMS消息。（**DMSMessage结构**）

表 5-33 DMSMessage 相关方法说明

方法名	方法说明
getBody()	获取DMS消息体。
getAttributes()	获取DMS消息属性集合。

d. **SMN触发器相关方法说明**

表 5-34 SMNTriggerEvent 相关方法说明

方法名	方法说明
getRecord()	获取消息记录集合。（**SMNRecord结构**）

表 5-35 SMNRecord 相关方法说明

方法名	方法说明
getEventVersion()	获取事件版本。（当前为1.0）
getEventSubscriptionUrn()	获取订阅URN。
getEventSource()	获取事件源。
getSmn()	获取SMN事件消息体。（**SMNBody结构**）

表 5-36 SMNBody 相关方法说明

方法名	方法说明
getTopicUrn()	获取SMN主题URN。
getTimeStamp()	获取消息时间戳。

方法名	方法说明
getMessageAtrributes()	获取消息属性集合。
getMessage()	获取消息体。
getType()	获取消息类型。
getMessageId()	获取消息ID。
getSubject()	获取消息主题。

e. **定时触发器相关方法说明**

表 5-37 TimerTriggerEvent 相关方法说明

方法名	方法说明
getVersion()	获取版本名称。（当前为“v1.0”）
getTime()	获取当前时间。
getTriggerType()	获取触发器类型。（“Timer”）
getTriggerName()	获取触发器名称。
getUserEvent()	获取触发器附加信息。

f. **LTS触发器相关方法说明**

表 5-38 LTSTriggerEvent 相关方法说明

方法名	方法说明
getLts()	获取LTS消息。（**LTSBody结构**）

表 5-39 LTSBody 相关方法说明

方法名	方法说明
getData()	获取LTS原始消息。
getRawData()	获取经过base64解码后的消息，UTF-8编码。

g. **CTS触发器相关方法说明**

表 5-40 CTSTriggerEvent 说明

方法名	方法说明
getCTS()	获取CTS消息体。（CTS结构）

表 5-41 CTS 结构相关方法说明

方法名	方法说明
getTime()	获取事件产生时间。
getUser()	获取触发该事件的用户信息（User结构）。
getRequest()	获取事件请求内容。
getResponse()	获取事件响应内容。
getCode()	获取响应码。
getServiceType()	获取事件触发的服务名称。
getResourceType()	获取事件触发的资源类型。
getResourceName()	获取事件触发的资源名称。
getResourceId()	获取事件触发资源的唯一标识。
getTraceName()	获取事件名称。
getTraceType()	获取事件触发的方式。（如ConsoleAction：代表前台操作）
getRecordTime()	获取CTS服务接收事件时间。
getTraceId()	获取当前事件的唯一标识。
getTraceStatus()	获取事件状态。

表 5-42 User 方法说明

方法名	方法说明
getName()	获取用户名。（同一账号可以创建多个子用户）
getId()	获取用户ID。
getDomain()	获取账号信息。

表 5-43 Domain 方法说明

方法名	方法说明
getName()	获取账号名称。
getId()	获取账号ID。

h. **Kafka触发器相关方法说明**

表 5-44 Kafka 触发器相关方法说明

方法名	方法说明
getEventVersion	获取事件版本。
getRegion	获取地区。
getEventTime	获取产生时间。
getTriggerType	获取触发器类型。
getInstanceId	获取实例ID。
getRecords	获取记录体。

说明

1. 例如使用APIG触发器时，只需要把入口函数（假如函数名为handler）的第一个参数按照如下方式设置：handler(APIGTriggerEvent event, Context context)。
2. 关于所有TriggerEvent，上面提到的TriggerEvent方法均有与之对应的set方法，建议在本地调试时使用；DIS和LTS均有对应的getRawData()方法，但无与之相应的setRawData()方法。

- Context接口

  Context接口提供函数获取函数执行上下文，例如，用户委托的AccessKey/SecretKey、当前请求ID、函数执行分配的内存空间、CPU数等。

  Context接口说明如表5-45所示。

表 5-45 Context 类上下文方法说明

方法名	方法说明
getRequestID( )	获取请求ID。
getRemainingTimeInMilliSeconds ( )	获取函数剩余运行时间。
getAccessKey( )	获取用户委托的AccessKey（有效期24小时），使用该方法需要给函数配置委托。 **当前函数工作流已停止维护Runtime SDK 中getAccessKey接口，您将无法使用getAccessKey获取临时AK。**
getSecretKey( )	获取用户委托的SecretKey（有效期24小时），使用该方法需要给函数配置委托。 **当前函数工作流已停止维护Runtime SDK 中getSecretKey接口，您将无法使用getSecretKey获取临时SK。**
getSecurityAccessKey( )	获取用户委托的SecurityAccessKey（有效期24小时），使用该方法需要给函数配置委托。

方法名	方法说明
getSecuritySecretKey( )	获取用户委托的SecuritySecretKey（有效期24小时），使用该方法需要给函数配置委托。
getSecurityToken( )	获取用户委托的SecurityToken（有效期24小时），使用该方法需要给函数配置委托。
getUserData(string key)	通过key获取用户通过环境变量传入的值。
getFunctionName( )	获取函数名称。
getRunningTimeInSeconds ( )	获取函数超时时间。
getVersion( )	获取函数的版本。
getMemorySize( )	分配的内存。
getCPUNumber( )	获取函数占用的CPU资源。
getPackage( )	获取函数组。
getToken( )	获取用户委托的token（有效期24小时），使用该方法需要给函数配置委托。
getLogger( )	获取context提供的logger方法（默认会输出时间、请求ID等信息）。
getAlias	获取函数的别名

- 日志接口

  Java SDK日志接口日志说明如表5-46所示。

  表 5-46 日志接口说明表

方法名	方法说明
RuntimeLogger()	记录用户输入日志。包含方法如下：log(String string)。

5.1.3 Java 函数开发指南（使用 IDEA 工具 maven 项目）

Java 函数接口定义

Java函数接口定义：*作用域 返回参数 函数名（函数参数，Context参数）*

- 作用域：提供给FunctionGraph调用的用户函数必须定义为public。
- 返回参数：用户定义，FunctionGraph负责转换为字符串，作为HTTP Response返回。对于返回参数对象类型，HTTP Response该类型的JSON字符串。
- 函数名：用户定义函数名称。

- 函数参数：用户定义参数，当前函数仅支持一个用户参数。对于复杂参数，建议定义为对象类型，以JSON字符串提供数据。FunctionGraph调用函数时，解析JSON为对象。
- Context：runtime提供函数执行上下文，其接口定义在**SDK接口**说明。

Java函数的函数执行入口参数格式为：**[包名].[类名].[执行函数名]**，请参考**函数执行入口**进行配置。

约束与限制

getToken()、getAccessKey()和getSecretKey()方法返回的内容包含敏感信息，请谨慎使用，避免造成敏感信泄露。

Java 的 initializer 入口介绍

函数工作流服务目前支持以下Java运行环境。

- Java 8 (runtime = Java8)
- Java 11（ runtime = Java11 ）

Initializer格式为：**[包名].[类名].[执行函数名]**

示例：创建函数时指定的initializer为com.huawei.Demo.my_initializer，那么FunctionGraph会去加载com.huawei包，Demo类中定义的my_initializer函数。

在函数工作流服务中使用Java实现initializer接口，需要定义一个java函数作为initializer入口，以下为initializer的简单示例。

```
public void my_initializer(Context context)
{
RuntimeLogger log = context.getLogger();
log.log(String.format("ak:%s", context.getAccessKey()));
}
```

- 函数名

  my_initializer需要与实现initializer接口时的initializer字段相对应。
- context参数

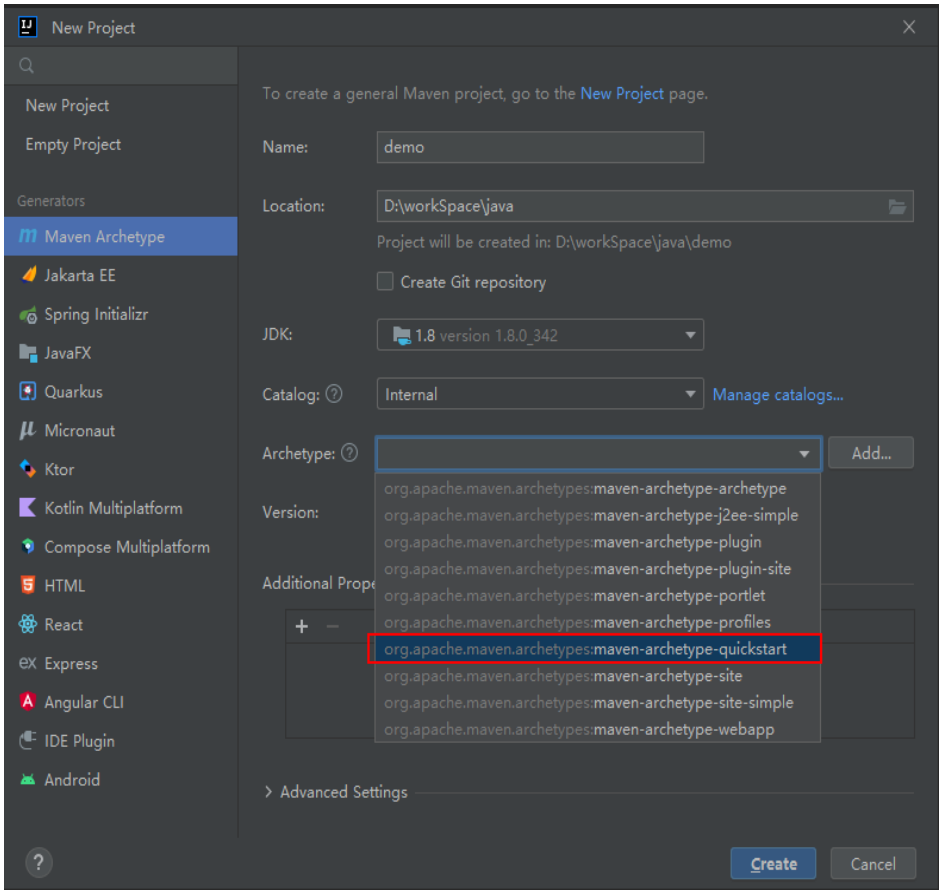
  context参数中包含一些函数的运行时信息，例如：request id、临时AccessKey、function meta等。

步骤一：使用 IDEA 创建 maven 工程

1. 创建函数工程

   配置IDEA，创建maven工程，如**图5-25**所示。

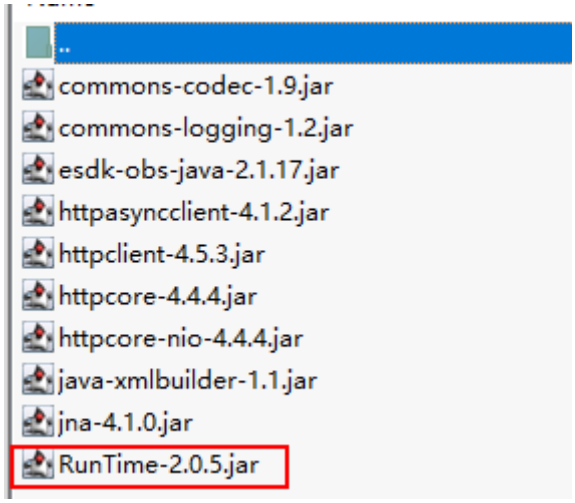
图 5-25 创建工程



2. 添加工程依赖

根据**Java SDK下载**提供的SDK地址，下载JavaRuntime SDK到本地开发环境解压，如**图5-26**所示。

图 5-26 下载 SDK 解压



a. 将zip中的Runtime-2.0.5.jar拷贝到本地目录，例如d:\\runtime中，在命令行窗口执行如下命令安装到本地的maven仓库中。

```
mvn install:install-file -Dfile=d:\runtime\RunTime-2.0.5.jar -DgroupId=RunTime -DartifactId=RunTime -Dversion=2.0.5 -Dpackaging=jar
```


b. 在pom.xml中配置dependency，如下所示。

```
<dependency>
<groupId>Runtime</groupId>
<artifactId>Runtime</artifactId>
<version>2.0.5</version>
</dependency>
```

c. 配置其他的依赖包，以obs依赖包为例，如下所示。

```
<dependency>
<groupId>com.huaweicloud</groupId>
<artifactId>esdk-obs-java</artifactId>
<version>3.21.4</version>
</dependency>
```

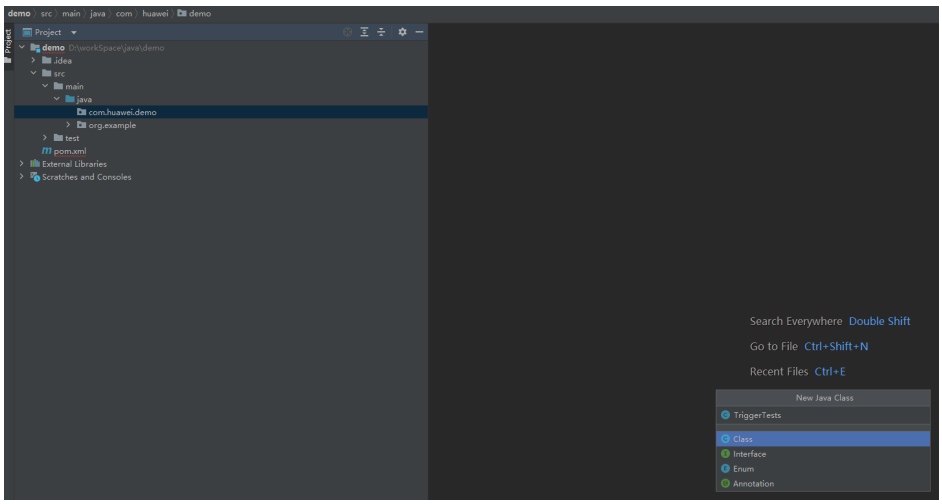
d. 在pom.xml中添加插件用来将代码和依赖包打包到一起。**请把mainClass替换为真实的类。**

```
<build>
 <plugins>
  <plugin>
   <artifactId>maven-assembly-plugin</artifactId>
   <configuration>
    <descriptorRefs>
     <descriptorRef>jar-with-dependencies</descriptorRef>
    </descriptorRefs>
    <archive>
     <manifest>
      <mainClass>com.huawei.demo.TriggerTests</mainClass>
     </manifest>
    </archive>
    <finalName>${project.name}</finalName>
   </configuration>
   <executions>
    <execution>
     <id>make-assembly</id>
     <phase>package</phase>
     <goals><goal>single</goal>
     </goals>
    </execution>
   </executions>
  </plugin>
 </plugins>
</build>
```

3. 配置函数资源

创建包com.huawei.demo，并在包下创建TriggerTests类，如图5-27所示。

图 5-27 创建 TriggerTests 类



4. 配置函数代码

   在TriggerTests.java中定义函数运行入口，示例代码如下：

```
package com.huawei.demo;

import java.io.UnsupportedEncodingException;
import java.util.HashMap;
import java.util.Map;

import com.huawei.services.runtime.Context;
import com.huawei.services.runtime.entity.apig.APIGTriggerEvent;
import com.huawei.services.runtime.entity.apig.APIGTriggerResponse;
import com.huawei.services.runtime.entity.dis.DISTriggerEvent;
import com.huawei.services.runtime.entity.dms.DMSTriggerEvent;
import com.huawei.services.runtime.entity.lts.LTSTriggerEvent;
import com.huawei.services.runtime.entity.smn.SMNTriggerEvent;
import com.huawei.services.runtime.entity.timer.TimerTriggerEvent;

public class TriggerTests {
    public APIGTriggerResponse apigTest(APIGTriggerEvent event, Context context){
        System.out.println(event);
        Map<String, String> headers = new HashMap<String, String>();
        headers.put("Content-Type", "application/json");
        return new APIGTriggerResponse(200, headers, event.toString());
    }

    public String smnTest(SMNTriggerEvent event, Context context){
        System.out.println(event);
        return "ok";
    }

    public String dmsTest(DMSTriggerEvent event, Context context){
        System.out.println(event);
        return "ok";
    }

    public String timerTest(TimerTriggerEvent event, Context context){
        System.out.println(event);
        return "ok";
    }

    public String disTest(DISTriggerEvent event, Context context) throws
UnsupportedEncodingException{
        System.out.println(event);
        System.out.println(event.getMessage().getRecords()[0].getRawData());
        return "ok";
    }

    public String ltsTest(LTSTriggerEvent event, Context context) throws
UnsupportedEncodingException {
        System.out.println(event);
        event.getLts().getData();
        System.out.println("raw data: " + event.getLts().getRawData());
        return "ok";
    }
}
```

   示例代码中添加了多个入口函数，分别使用了不同的触发器事件类型，可通过函数工作流控制台**修改函数执行入口**测试不同的入口函数。**当函数的事件源是APIG时，相关约束条件请参考Base64解码和返回结构体的说明**。

5. 工程编译打包

   在命令行窗口执行如下命令进行编译打包。编译完成后在target目录会生成一个demo-jar-with-dependencies.jar。

```
mvn package assembly:single
```


步骤二：创建 Java 函数并测试

1. 登录函数工作流控制台，左侧导航栏选择“函数 > 函数列表”，单击右上角“创建函数”进入创建函数页面，选择“创建空白函数”。
2. 如图5-28所示，配置函数基本信息，配置完成后单击右下角“创建函数”完成创建。

图 5-28 创建 Java 函数

3. 进入函数详情页，在“代码”页签下，单击右侧“上传代码 > JAR文件 ”，如图5-29所示，添加步骤一：使用IDEA创建maven工程的JAR文件上传。

图 5-29 创建函数

4. 上传成功后，请参见修改函数执行入口修改函数执行入口，单击“保存”。
5. 回到“代码”页签，单击代码编辑区的“测试”，选择“创建新的测试事件”，在云事件模板列表中选择需要测试的事件模板，单击“创建”。
6. 单击“测试”，查看函数执行结果。

   函数执行结果分为三部分，分别为函数返回（由callback返回）、执行摘要、日志输出（由console.log或getLogger()方法获取的日志方法输出），参考表5-47查看结果说明。

表 5-47 执行结果说明

参数项	执行成功	执行失败

函数返回	返回函数中定义的返回信息。	返回包含错误信息和堆栈异常报错信息的JSON文件。格式如下：
		{
		"errorMessage": "",
		"stackTrace": []
		}
		errorMessage：Runtime返回的错误信息
		stackTrace：Runtime返回的堆栈异常报错信息

执行摘要	显示请求ID、配置内存、执行时长、实际使用内存和收费时长。	显示请求ID、配置内存、执行时长、实际使用内存和收费时长。
日志	打印函数日志，最多显示4KB的日志。	打印报错信息，最多显示4KB的日志。

说明

如需测试示例代码中的其他事件源触发器，例如SMN事件源，可在FunctionGraph控制台，参考**修改函数执行入口**将函数执行入口修改为com.huawei.demo.TriggerTests.smnTest，并参考以上步骤，创建新的消息通知服务SMN测试事件进行函数测试。

SDK 接口

FunctionGraph函数JavaSDK提供了Event事件接口、Context接口和日志记录接口，SDK下载地址见**Java SDK下载**（校验文件：**fss-java-sdk-2.0.5.sha256**）。

- Event事件接口

  Java SDK引入了触发器事件结构体定义，当前支持CTS、DMS、DIS、SMN、LTS、TIMER、APIG、Kafka。在需要使用触发器的场景下，编写相应代码更简便。

  a. **APIG触发器相关方法说明**

  - APIGTriggerEvent相关方法说明

    表 5-48 APIGTriggerEvent 相关方法说明

方法名	方法说明
isBase64Encoded()	Event中的body是否是base64编码。
getHttpMethod()	获取Http请求方法。
getPath()	获取Http请求路径。
getBody()	获取Http请求body。

方法名	方法说明
getPathParameters()	获取所有路径参数。
getRequestContext()	获取相关的APIG配置。（返回**APIGRequestContext对象**）
getHeaders()	获取Http请求头。
getQueryStringParameters()	获取查询参数。 **当前查询参数不支持取值为数组，如果查询参数的取值需要为数组，请自定义对应的触发器事件结构体。**
getRawBody()	获取base64编码前的内容。
getUserData()	获取APIG自定义认证中设置的userdata。

表 5-49 APIGRequestContext 相关方法说明

方法名	方法说明
getApiId()	获取API的ID。
getRequestId()	获取此次API请求的requestId。
getStage()	获取发布环境名称。
getSourceIp()	获取APIG自定义认证信息中的源IP。

- APIGTriggerResponse相关方法说明

表 5-50 APIGTriggerResponse 构造方法说明

方法名	方法说明
无参构造APIGTriggerResponse()	其中headers设置为null，statusCode设置为200，body设置为" "，isBase64Encoded设置为false。
三个参数构造APIGTriggerResponse(statusCode, headers, body)	isBase64Encoded设置为false，其他均以输入为准。
四个参数构造APIGTriggerResponse(statusCode, headers, isBase64Encoded, body)	按照对应顺序设置值即可。

表 5-51 APIGTriggerResponse 相关方法说明

方法名	方法说明
setBody(String body)	设置消息体。
setHeaders(Map<String,String> headers)	设置最终返回的Http响应头。
setStatusCode(int statusCode)	设置Http状态码。
setBase64Encoded(boolean isBase64Encoded)	设置body是否经过base64编码。
setBase64EncodedBody(String body)	将输入进行base64编码，并设置到Body中。
addHeader(String key, String value)	增加一组Http header。
removeHeader(String key)	从现有的header中移除指定header。
addHeaders(Map<String,String> headers)	增加多个header。

说明

APIGTriggerResponse有headers属性，可以通过setHeaders方法和带有headers参数的构造函数对齐进行初始化。

b. **DIS触发器相关方法说明**

表 5-52 DISTriggerEvent 相关方法说明

方法名	方法说明
getShardID()	获取分区ID。
getMessage()	获取DIS消息体。（**DISMessage结构**）
getTag()	获取函数版本。
getStreamName()	获取通道名称。

表 5-53 DISMessage 相关方法说明

方法名	方法说明
getNextPatitionCursor()	获取下一个游标。
getRecords()	获取消息记录。（**DISRecord结构**）

方法名	方法说明
getMillisBehindLatest()	保留方法（目前返回0）。

表 5-54 DISRecord 相关方法说明

方法名	方法说明
getPartitionKey()	获取数据分区。
getData()	获取数据。
getRawData()	data经过base64解码后的字符串，UTF-8编码。
getSequenceNumber()	获取序列号（每个记录的唯一标识）。

c. **DMS触发器相关方法说明**

表 5-55 DMSTriggerEvent 相关方法说明

方法名	方法说明
getQueueId()	获取队列ID。
getRegion()	获取区域名称。
getEventType()	获取事件类型。（返回“MessageCreated”）
getConsumerGroupId()	获取消费组ID。
getMessages()	获取DMS消息。（**DMSMessage结构**）

表 5-56 DMSMessage 相关方法说明

方法名	方法说明
getBody()	获取DMS消息体。
getAttributes()	获取DMS消息属性集合。

d. **SMN触发器相关方法说明**

表 5-57 SMNTriggerEvent 相关方法说明

方法名	方法说明
getRecord()	获取消息记录集合。（**SMNRecord结构**）

表 5-58 SMNRecord 相关方法说明

方法名	方法说明
getEventVersion()	获取事件版本。（当前为1.0）
getEventSubscriptionUrn()	获取订阅URN。
getEventSource()	获取事件源。
getSmn()	获取SMN事件消息体。（**SMNBody结构**）

表 5-59 SMNBody 相关方法说明

方法名	方法说明
getTopicUrn()	获取SMN主题URN。
getTimeStamp()	获取消息时间戳。
getMessageAtrributes()	获取消息属性集合。
getMessage()	获取消息体。
getType()	获取消息类型。
getMessageId()	获取消息ID。
getSubject()	获取消息主题。

e. **定时触发器相关方法说明**

表 5-60 TimerTriggerEvent 相关方法说明

方法名	方法说明
getVersion()	获取版本名称。（当前为“v1.0”）
getTime()	获取当前时间。
getTriggerType()	获取触发器类型。（“Timer”）
getTriggerName()	获取触发器名称。
getUserEvent()	获取触发器附加信息。

f. **LTS触发器相关方法说明**

表 5-61 LTSTriggerEvent 相关方法说明

方法名	方法说明
getLts()	获取LTS消息。（ **LTSBody结构** ）

表 5-62 LTSBody 相关方法说明

方法名	方法说明
getData()	获取LTS原始消息。
getRawData()	获取经过base64解码后的消息，UTF-8编码。

g. **CTS触发器相关方法说明**

表 5-63 CTSTriggerEvent 说明

方法名	方法说明
getCTS()	获取CTS消息体。（ CTS结构 ）

表 5-64 CTS 结构相关方法说明

方法名	方法说明
getTime()	获取事件产生时间。
getUser()	获取触发该事件的用户信息（ User结构 ）。
getRequest()	获取事件请求内容。
getResponse()	获取事件响应内容。
getCode()	获取响应码。
getServiceType()	获取事件触发的服务名称。
getResourceType()	获取事件触发的资源类型。
getResourceName()	获取事件触发的资源名称。
getResourceId()	获取事件触发资源的唯一标识。
getTraceName()	获取事件名称。
getTraceType()	获取事件触发的方式。（ 如ConsoleAction：代表前台操作 ）

方法名	方法说明
getRecordTime()	获取CTS服务接收事件时间。
getTraceId()	获取当前事件的唯一标识。
getTraceStatus()	获取事件状态。

表 5-65 User 方法说明

方法名	方法说明
getName()	获取用户名。（同一账号可以创建多个子用户）
getId()	获取用户ID。
getDomain()	获取账号信息。

表 5-66 Domain 方法说明

方法名	方法说明
getName()	获取账号名称。
getId()	获取账号ID。

h. **Kafka触发器相关方法说明**

表 5-67 Kafka 触发器相关方法说明

方法名	方法说明
getEventVersion	获取事件版本。
getRegion	获取地区。
getEventTime	获取产生时间。
getTriggerType	获取触发器类型。
getInstanceId	获取实例ID。
getRecords	获取记录体。

说明

1. 例如使用APIG触发器时，只需要把入口函数（假如函数名为handler）的第一个参数按照如下方式设置：handler(APIGTriggerEvent event, Context context)。
2. 关于所有TriggerEvent，上面提到的TriggerEvent方法均有与之对应的set方法，建议在本地调试时使用；DIS和LTS均有对应的getRawData()方法，但无与之相应的setRawData()方法。

- Context接口

  Context接口提供函数获取函数执行上下文，例如，用户委托的AccessKey/SecretKey、当前请求ID、函数执行分配的内存空间、CPU数等。

  Context接口说明如表5-68所示。

  表 5-68 Context 类上下文方法说明

方法名	方法说明
getRequestID( )	获取请求ID。
getRemainingTimeInMilliSeconds ( )	获取函数剩余运行时间。
getAccessKey( )	获取用户委托的AccessKey（有效期24小时），使用该方法需要给函数配置委托。 **当前函数工作流已停止维护Runtime SDK 中getAccessKey接口，您将无法使用getAccessKey获取临时AK。**
getSecretKey( )	获取用户委托的SecretKey（有效期24小时），使用该方法需要给函数配置委托。 **当前函数工作流已停止维护Runtime SDK 中getSecretKey接口，您将无法使用getSecretKey获取临时SK。**
getSecurityAccessKey( )	获取用户委托的SecurityAccessKey（有效期24小时），使用该方法需要给函数配置委托。
getSecuritySecretKey( )	获取用户委托的SecuritySecretKey（有效期24小时），使用该方法需要给函数配置委托。
getSecurityToken( )	获取用户委托的SecurityToken（有效期24小时），使用该方法需要给函数配置委托。
getUserData(string key)	通过key获取用户通过环境变量传入的值。
getFunctionName( )	获取函数名称。
getRunningTimeInSeconds ( )	获取函数超时时间。
getVersion( )	获取函数的版本。
getMemorySize( )	分配的内存。
getCPUNumber( )	获取函数占用的CPU资源。
getPackage( )	获取函数组。
getToken( )	获取用户委托的token（有效期24小时），使用该方法需要给函数配置委托。
getLogger( )	获取context提供的logger方法（默认会输出时间、请求ID等信息）。

方法名	方法说明
getAlias	获取函数的别名

- 日志接口
  Java SDK日志接口日志说明如表5-69所示。

  表 5-69 日志接口说明表

方法名	方法说明
RuntimeLogger()	记录用户输入日志。包含方法如下：log(String string)。

5.2 java 模板

```
package com.huawei.demo;
import com.huawei.services.runtime.Context;
import com.huawei.services.runtime.entity.apig.APIGTriggerEvent;
import com.huawei.services.runtime.entity.apig.APIGTriggerResponse;
import com.huawei.services.runtime.entity.dis.DISTriggerEvent;
import com.huawei.services.runtime.entity.dms.DMSTriggerEvent;
import com.huawei.services.runtime.entity.lts.LTSTriggerEvent;
import com.huawei.services.runtime.entity.smn.SMNTriggerEvent;
import com.huawei.services.runtime.entity.timer.TimerTriggerEvent;
import java.io.UnsupportedEncodingException;
import java.util.HashMap;
import java.util.Map;
public class TriggerTests {
  public APIGTriggerResponse apigTest(APIGTriggerEvent event, Context context) {
    System.out.println(event);
    Map<String, String> headers = new HashMap<>();
    headers.put("Content-Type", "application/json");
    return new APIGTriggerResponse(200, headers, event.toString());
  }
  public String smnTest(SMNTriggerEvent event, Context context) {
    System.out.println(event);
    return "ok";
  }
  public String dmsTest(DMSTriggerEvent event, Context context) {
    System.out.println(event);
    return "ok";
  }
  public String timerTest(TimerTriggerEvent event, Context context) {
    System.out.println(event);
    return "ok";
  }
  public String disTest(DISTriggerEvent event, Context context) throws UnsupportedEncodingException {
    System.out.println(event);
    System.out.println(event.getMessage().getRecords()[0].getRawData());
    return "ok";
  }
  public String ltsTest(LTSTriggerEvent event, Context context) throws UnsupportedEncodingException {
    System.out.println(event);
    System.out.println("raw data: " + event.getLts().getRawData());
    return "ok";
  }
}
```


5.3 制作依赖包

制作函数依赖包推荐在Huawei Cloud EulerOS 2.0环境中进行。使用其他系统打包可能会因为底层依赖库的原因，运行出问题，比如找不到动态链接库。

如果安装的依赖模块需要添加依赖库，请将依赖库归档到zip依赖包文件中，例如，添加.dll、.so、.a等依赖库。

使用Java编译型语言开发函数时，依赖包需要在本地编译。开发Java函数中如何添加依赖包请参见**Java函数开发指南（使用IDEA工具普通Java项目）**。

6 Go

6.1 开发事件函数

Go 函数接口定义

FunctionGraph运行时支持Go 1.x版本，函数有明确的接口定义，如下所示：

```
func Handler (payload []byte, ctx context.RuntimeContext)
```

- 入口函数名（Handler）：入口函数名称。
- 执行事件体（payload）： 函数执行界面由用户输入的执行事件参数， 格式为JSON对象。
- 上下文环境（ctx）：Runtime提供的函数执行上下文，其接口定义在**SDK接口**说明。

Go函数的函数执行入口参数格式为：与代码包中的可执行文件名保持一致。编译后的动态库文件名称必须与函数执行入口的插件名称保持一致，例如：动态库名称为testplugin.so，则函数执行入口命名为testplugin.Handler。请参考**函数执行入口**进行配置或修改。

约束与限制

GetToken()、GetAccessKey()和GetSecretKey()方法返回的内容包含敏感信息，请谨慎使用，避免造成用户敏感信息的泄露。

SDK 接口

FunctionGraph函数GoSDK提供了Event事件接口、Context接口和日志记录接口。**Go SDK下载**（**Go SDK下载.sha256**）。

- Event事件接口

  Go SDK加入了触发器事件结构体定义，目前支持CTS、KAFKA、DIS、DDS、SMN、LTS、TIMER、APIG、八种。在需要使用触发器的场景时，编写相关代码更简单。

  a. **APIG触发器相关字段说明**

     i. APIGTriggerEvent相关字段说明

表 6-1 APIGTriggerEvent 相关字段说明

字段名	字段描述
IsBase64Encoded	Event中的body是否是base64编码
HttpMethod	Http请求方法
Path	Http请求路径
Body	Http请求body
PathParameters	所有路径参数
RequestContext	相关的APIG配置（**APIGRequestContext对象**）
Headers	Http请求头
QueryStringParameters	查询参数
UserData	APIG自定义认证中设置的userdata

表 6-2 APIGRequestContext 相关字段说明

字段名	字段描述
ApiId	API的ID
RequestId	此次API请求的requestId
Stage	发布环境名称

ii. APIGTriggerResponse相关字段说明

表 6-3 APIGTriggerResponse 相关字段说明

字段名	字段描述
Body	消息体
Headers	最终返回的Http响应头
StatusCode	Http状态码，int类型
IsBase64Encoded	body是否经过base64编码，bool类型

> **说明**
>
> APIGTriggerEvent提供GetRawBody()方法获取base64解码后的body体，相应的APIGTriggerResponse提供SetBase64EncodedBody()方法来设置base64编码的body体。

b. **DIS触发器相关字段说明**

表 6-4 DISTriggerEvent 相关字段说明

字段名	字段描述
ShardID	分区ID
Message	DIS消息体（**DISMessage结构**）
Tag	函数版本
StreamName	通道名称

表 6-5 DISMessage 相关字段说明

字段名	字段描述
NextPartitionCursor	下一个游标
Records	消息记录（**DISRecord结构**）
MillisBehindLatest	保留字段

表 6-6 DISRecord 相关字段说明

字段名	字段描述
PartitionKey	数据分区
Data	数据
SequenceNumber	序列号（每个记录的唯一标识）

c. **KAFKA触发器相关字段说明**

表 6-7 KAFKATriggerEvent 相关字段说明

字段名	字段描述
InstanceId	实例ID
Records	消息记录（**表6-8**）
TriggerType	触发器类型，返回KAFKA
Region	region

字段名	字段描述
EventTime	事件发生时间，秒数
EventVersion	事件版本

表 6-8 KAFKARecord 相关字段说明

字段名	字段描述
Messages	DMS消息体
TopicId	DMS的主题ID

d. **SMN触发器相关字段说明**

表 6-9 SMNTriggerEvent 相关字段说明

字段名	字段描述
Record	消息记录集合（**SMNRecord结构**）

表 6-10 SMNRecord 相关字段说明

字段名	字段描述
EventVersion	事件版本（当前为1.0）
EventSubscriptionUrn	订阅URN
EventSource	事件源
Smn	SMN事件消息体（**SMNBody结构**）

表 6-11 SMNBody 相关字段说明

字段名	字段描述
TopicUrn	SMN主题URN
TimeStamp	消息时间戳
MessageAtrributes	消息属性集合
Message	消息体
Type	消息类型
MessageId	消息ID

字段名	字段描述
Subject	消息主题

e. **定时触发器相关字段说明**

表 6-12 TimerTriggerEvent 相关字段说明

字段名	字段描述
Version	版本名称（当前为“v1.0”）
Time	当前时间
TriggerType	触发器类型（“Timer”）
TriggerName	触发器名称
UserEvent	触发器附加信息

f. **LTS触发器相关字段说明**

表 6-13 LTSTriggerEvent 相关字段说明

字段名	字段描述
Lts	LTS消息（**LTSBody结构**）

表 6-14 LTSBody 相关字段说明

字段名	字段描述
Data	LTS原始消息

说明

LTSBody提供GetRawData()函数返回base64解码后的消息。

g. **CTS触发器相关字段说明**

表 6-15 CTSTriggerEvent 字段说明

字段名	字段说明
CTS	CTS消息体（**表6-16**）

表 6-16 CTS 结构相关字段说明

字段名	字段描述
Time	事件产生时间
User	触发该事件的用户信息（表6-17）
Request	事件请求内容
Response	事件响应内容
Code	响应码
ServiceType	事件触发的服务名称
ResourceType	事件触发的资源类型
ResourceName	事件触发的资源名称
ResourceId	事件触发资源的唯一标识
TraceName	事件名称
TraceType	事件触发的方式（如ConsoleAction：代表前台操作）
RecordTime	CTS服务接收事件时间
TraceId	当前事件的唯一标识
TraceStatus	事件状态

表 6-17 User 字段说明

字段名	字段描述
Name	用户名（同一账号可以创建多个子用户）
Id	用户ID
Domain	账号信息（表6-18）

表 6-18 Domain 字段说明

字段名	字段描述
Name	账号名称
Id	账号ID

说明

1. 例如使用APIG触发器时，只需要把入口函数（假如函数名为handler）的第一个参数按照如下方式设置：handler(APIGTriggerEvent event, Context context)。**相关约束条件请参考Base64解码和返回结构体的说明**。
2. 关于所有TriggerEvent，上面提到的TriggerEvent方法均有与之对应的set方法，建议在本地调试时使用；DIS和LTS均有对应的getRawData()方法，但无与之相应的setRawData()方法。

- Context接口

  Context接口提供函数获取函数执行上下文，例如，用户委托的AccessKey/SecretKey、当前请求ID、函数执行分配的内存空间、CPU数等。

  Context接口说明如表6-19所示。

表 6-19 Context 类上下文方法说明

方法名	方法说明
getRequestID( )	获取请求ID。
getRemainingTimeInMilligetRunningTimeInSecondsSeconds( )	获取函数剩余运行时间。
getAccessKey( )	获取用户委托的AccessKey（有效期24小时），使用该方法需要给函数配置委托。 **当前函数工作流已停止维护Runtime SDK中getAccessKey接口，您将无法使用getAccessKey获取临时AK。**
getSecretKey( )	获取用户委托的SecretKey（有效期24小时），使用该方法需要给函数配置委托。 **当前函数工作流已停止维护Runtime SDK中getSecretKey接口，您将无法使用getSecretKey获取临时SK。**
getSecurityAccessKey()	获取用户委托的SecurityAccessKey（有效期24小时），使用该方法需要给函数配置委托。
getSecuritySecretKey()	获取用户委托的SecuritySecretKey（有效期24小时），使用该方法需要给函数配置委托。
getSecurityToken()	获取用户委托的SecurityToken（有效期24小时），使用该方法需要给函数配置委托。
getUserData(string key)	通过key获取用户通过环境变量传入的值。
getFunctionName( )	获取函数名称。
getRunningTimeInSeconds ( )	获取函数超时时间。
getVersion( )	获取函数的版本。
getMemorySize( )	分配的内存。

方法名	方法说明
getCPUNumber( )	获取函数占用的CPU资源。
getPackage( )	获取函数组。
getToken( )	获取用户委托的token（有效期24小时），使用该方法需要给函数配置委托。
getLogger( )	获取context提供的logger方法（默认会输出时间、请求ID等信息）。
getAlias	获取函数的别名

- 日志接口Go SDK日志接口日志说明如表6-20所示。

表 6-20 日志接口说明表

方法名	方法说明
RuntimeLogger()	• 记录用户输入日志对象，包含方法如下：Logf(format string, args ...interface{}) • 该方法会将内容输出到标准输出，格式："时间-请求ID-输出内容"，示例如下：2017-10-25T09:10:03.328Z 473d369d-101a-445e-a7a8-315cca788f86 test log output。

开发 Go 函数

登录已经安装了Go 1.x SDK的linux服务器。（当前支持Ubuntu 14.04，Ubuntu 16.04，SuSE 11.3，SuSE 12.0，SuSE 12.1）

- 如果Go的版本支持go mod（go版本要求：1.11.1 及以上版本，>=1.11.1），可以按照如下步骤进行编译和打包：

步骤1 创建一个临时目录例如“/home/fssgo”，将FunctionGraph的Go RUNTIME SDK解压到新创建的目录，并开启go module开关，操作如下：

$ mkdir -p /home/fssgo

$ unzip functiongraph-go-runtime-sdk-1.0.1.zip -d /home/fssgo

$ export GO111MODULE="on"

步骤2 在目录“/home/fssgo”下生成go.mod文件，操作如下，以模块名为test为例：

$ go mod init test

步骤3 在目录“/home/fssgo”下编辑go.mod文件，添加加粗部分内容：

```
module test
```



```
go 1.14

require (
    huaweicloud.com/go-runtime v0.0.0-00010101000000-000000000000
)

replace (
    huaweicloud.com/go-runtime => ./go-runtime
)
```

步骤4 在目录“/home/fssgo”下创建函数文件，并实现如下接口：

func Handler(payload []byte, ctx context.RuntimeContext) (interface{}, error)

其中payload为客户端请求的body数据，ctx为FunctionGraph提供的运行时上下文对象，具体提供的方法可以参考**表6-19**，以test.go为例：

```
package main

import (
    "fmt"
    "huaweicloud.com/go-runtime/go-api/context"
    "huaweicloud.com/go-runtime/pkg/runtime"
    "huaweicloud.com/go-runtime/events/apig"
    "huaweicloud.com/go-runtime/events/cts"
    "huaweicloud.com/go-runtime/events/dds"
    "huaweicloud.com/go-runtime/events/dis"
    "huaweicloud.com/go-runtime/events/kafka"
    "huaweicloud.com/go-runtime/events/lts"
    "huaweicloud.com/go-runtime/events/smn"
    "huaweicloud.com/go-runtime/events/timer"
    "encoding/json"
)

func ApigTest(payload []byte, ctx context.RuntimeContext) (interface{}, error) {
    var apigEvent apig.APIGTriggerEvent
    err := json.Unmarshal(payload, &apigEvent)
    if err != nil {
        fmt.Println("Unmarshal failed")
        return "invalid data", err
    }
    ctx.GetLogger().Logf("payload:%s", apigEvent.String())
    apigResp := apig.APIGTriggerResponse{
        Body: apigEvent.String(),
        Headers: map[string]string {
            "content-type": "application/json",
        },
        StatusCode: 200,
    }
    return apigResp, nil
}

func CtsTest(payload []byte, ctx context.RuntimeContext) (interface{}, error) {
    var ctsEvent cts.CTSTriggerEvent
    err := json.Unmarshal(payload, &ctsEvent)
    if err != nil {
        fmt.Println("Unmarshal failed")
        return "invalid data", err
    }
    ctx.GetLogger().Logf("payload:%s", ctsEvent.String())
    return "ok", nil
}

func DdsTest(payload []byte, ctx context.RuntimeContext) (interface{}, error) {
    var ddsEvent dds.DDSTriggerEvent
    err := json.Unmarshal(payload, &ddsEvent)
    if err != nil {
        fmt.Println("Unmarshal failed")
        return "invalid data", err
```



```
    }
    ctx.GetLogger().Logf("payload:%s", ddsEvent.String())
    return "ok", nil
}

func DisTest(payload []byte, ctx context.RuntimeContext) (interface{}, error) {
    var disEvent dis.DISTriggerEvent
    err := json.Unmarshal(payload, &disEvent)
    if err != nil {
        fmt.Println("Unmarshal failed")
        return "invalid data", err
    }
    ctx.GetLogger().Logf("payload:%s", disEvent.String())
    return "ok", nil
}

func KafkaTest(payload []byte, ctx context.RuntimeContext) (interface{}, error) {
    var kafkaEvent kafka.KAFKATriggerEvent
    err := json.Unmarshal(payload, &kafkaEvent)
    if err != nil {
        fmt.Println("Unmarshal failed")
        return "invalid data", err
    }
    ctx.GetLogger().Logf("payload:%s", kafkaEvent.String())
    return "ok", nil
}

func LtsTest(payload []byte, ctx context.RuntimeContext) (interface{}, error) {
    var ltsEvent lts.LTSTriggerEvent
    err := json.Unmarshal(payload, &ltsEvent)
    if err != nil {
        fmt.Println("Unmarshal failed")
        return "invalid data", err
    }
    ctx.GetLogger().Logf("payload:%s", ltsEvent.String())
    return "ok", nil
}

func SmnTest(payload []byte, ctx context.RuntimeContext) (interface{}, error) {
    var smnEvent smn.SMNTriggerEvent
    err := json.Unmarshal(payload, &smnEvent)
    if err != nil {
        fmt.Println("Unmarshal failed")
        return "invalid data", err
    }
    ctx.GetLogger().Logf("payload:%s", smnEvent.String())
    return "ok", nil
}

func TimerTest(payload []byte, ctx context.RuntimeContext) (interface{}, error) {
    var timerEvent timer.TimerTriggerEvent
    err := json.Unmarshal(payload, &timerEvent)
    if err != nil {
        fmt.Println("Unmarshal failed")
        return "invalid data", err
    }
    return timerEvent.String(), nil
}

func main() {
    runtime.Register(ApigTest)
}
```

约束与限制：

1. 如果函数返回的error参数不是nil，则会认为函数执行失败。
2. 如果函数返回的error为nil，FunctionGraph仅支持返回如下几种类型的值。
   nil：返回的HTTP响应Body为空。

[]byte：返回的HTTP响应Body内容为该字节数组内容。

string：返回的HTTP响应Body内容为该字符串内容。

其它：FunctionGraph会将返回值作为对象进行json编码，并将编码后的内容作为HTTP响应的Body，同时设置响应的"Content-Type"头为"application/json"。

3. **上面的例子是APIG触发器的事件类型，如果是其他触发器类型需要修改main函数的内容，例如cts触发器修改为runtime.Register(CtsTest)，目前只支持注册一个入口。**
4. **当函数的事件源是APIG时，相关约束条件请参考Base64解码和返回结构体的说明。**

步骤5 编译和打包

函数代码编译完成后，按照如下方式编译和打包。

1. 编译

   $ cd /home/fssgo

   $ go build -o handler test.go

   说明

   handler可以自定义，后面作为函数入口

2. 打包：

   $ zip fss_examples_go1.x.zip handler

----结束

- 如果Go的版本不支持go mod（go版本低于1.11.1），可以按照如下步骤进行编译和打包：

步骤1 创建一个临时目录例如“/home/fssgo/src/huaweicloud.com”，将FunctionGraph的sdk **Go RUNTIME SDK**解压到新创建的目录，操作如下：

$ mkdir -p /home/fssgo/src/huaweicloud.com

$ unzip functiongraph-go-runtime-sdk-1.0.1.zip -d /home/fssgo/src/huaweicloud.com

步骤2 在目录“/home/fssgo/src”下创建函数文件，并实现如下接口：

func Handler(payload []byte, ctx context.RuntimeContext) (interface{}, error)

其中payload为客户端请求的body数据，ctx为FunctionGraph提供的运行时上下文对象，具体提供的方法可以参考SDK接口，以test.go为例：

```
package main

import (
    "fmt"
    "huaweicloud.com/go-runtime/go-api/context"
    "huaweicloud.com/go-runtime/pkg/runtime"
    "huaweicloud.com/go-runtime/events/apig"
    "huaweicloud.com/go-runtime/events/cts"
    "huaweicloud.com/go-runtime/events/dds"
    "huaweicloud.com/go-runtime/events/dis"
    "huaweicloud.com/go-runtime/events/kafka"
    "huaweicloud.com/go-runtime/events/lts"
    "huaweicloud.com/go-runtime/events/smn"
    "huaweicloud.com/go-runtime/events/timer"
    "encoding/json"
```



```
)

func ApigTest(payload []byte, ctx context.RuntimeContext) (interface{}, error) {
    var apigEvent apig.APIGTriggerEvent
    err := json.Unmarshal(payload, &apigEvent)
    if err != nil {
        fmt.Println("Unmarshal failed")
        return "invalid data", err
    }
    ctx.GetLogger().Logf("payload:%s", apigEvent.String())
    apigResp := apig.APIGTriggerResponse{
        Body: apigEvent.String(),
        Headers: map[string]string {
            "content-type": "application/json",
        },
        StatusCode: 200,
    }
    return apigResp, nil
}

func CtsTest(payload []byte, ctx context.RuntimeContext) (interface{}, error) {
    var ctsEvent cts.CTSTriggerEvent
    err := json.Unmarshal(payload, &ctsEvent)
    if err != nil {
        fmt.Println("Unmarshal failed")
        return "invalid data", err
    }
    ctx.GetLogger().Logf("payload:%s", ctsEvent.String())
    return "ok", nil
}

func DdsTest(payload []byte, ctx context.RuntimeContext) (interface{}, error) {
    var ddsEvent dds.DDSTriggerEvent
    err := json.Unmarshal(payload, &ddsEvent)
    if err != nil {
        fmt.Println("Unmarshal failed")
        return "invalid data", err
    }
    ctx.GetLogger().Logf("payload:%s", ddsEvent.String())
    return "ok", nil
}

func DisTest(payload []byte, ctx context.RuntimeContext) (interface{}, error) {
    var disEvent dis.DISTriggerEvent
    err := json.Unmarshal(payload, &disEvent)
    if err != nil {
        fmt.Println("Unmarshal failed")
        return "invalid data", err
    }
    ctx.GetLogger().Logf("payload:%s", disEvent.String())
    return "ok", nil
}

func KafkaTest(payload []byte, ctx context.RuntimeContext) (interface{}, error) {
    var kafkaEvent kafka.KAFKATriggerEvent
    err := json.Unmarshal(payload, &kafkaEvent)
    if err != nil {
        fmt.Println("Unmarshal failed")
        return "invalid data", err
    }
    ctx.GetLogger().Logf("payload:%s", kafkaEvent.String())
    return "ok", nil
}

func LtsTest(payload []byte, ctx context.RuntimeContext) (interface{}, error) {
    var ltsEvent lts.LTSTriggerEvent
    err := json.Unmarshal(payload, &ltsEvent)
    if err != nil {
        fmt.Println("Unmarshal failed")
```



```
        return "invalid data", err
    }
    ctx.GetLogger().Logf("payload:%s", ltsEvent.String())
    return "ok", nil
}

func SmnTest(payload []byte, ctx context.RuntimeContext) (interface{}, error) {
    var smnEvent smn.SMNTriggerEvent
    err := json.Unmarshal(payload, &smnEvent)
    if err != nil {
        fmt.Println("Unmarshal failed")
        return "invalid data", err
    }
    ctx.GetLogger().Logf("payload:%s", smnEvent.String())
    return "ok", nil
}

func TimerTest(payload []byte, ctx context.RuntimeContext) (interface{}, error) {
    var timerEvent timer.TimerTriggerEvent
    err := json.Unmarshal(payload, &timerEvent)
    if err != nil {
        fmt.Println("Unmarshal failed")
        return "invalid data", err
    }
    return timerEvent.String(), nil
}

func main() {
    runtime.Register(ApigTest)
}
```

约束与限制：

1. 如果函数返回的error参数不是nil，则会认为函数执行失败。
2. 如果函数返回的error为nil，FunctionGraph仅支持返回如下几种类型的值。

   nil：返回的HTTP响应Body为空。

   []byte：返回的HTTP响应Body内容为该字节数组内容。

   string：返回的HTTP响应Body内容为该字符串内容。

   其它：FunctionGraph会将返回值作为对象进行json编码，并将编码后的内容作为HTTP响应的Body，同时设置响应的"Content-Type"头为"application/json"。
3. **上面的例子是APIG触发器的事件类型，如果是其他触发器类型需要修改main函数的内容，例如cts触发器修改为runtime.Register(CtsTest)，目前只支持注册一个入口。**
4. **当函数的事件源是APIG时，相关约束条件请参考Base64解码和返回结构体的说明。**

步骤3 编译和打包

函数代码编译完成后，按照如下方式编译和打包。

1. 设置GOROOT和GOPATH环境变量：

   $ export GOROOT=/usr/local/go（假设Go安装到了/usr/local/go目录）

   $ export PATH=$GOROOT/bin:$PATH

   $ export GOPATH=/home/fssgo
2. 编译：

   $ cd /home/fssgo

   $ go build -o handler test.go

说明

handler可以自定义，后面作为函数入口

3. 打包：

   $ zip fss_examples_go1.x.zip handler

- 创建函数

  登录FunctionGraph控制台，创建Go1.x函数，上传代码包**fss_examples_go1.x.zip**。

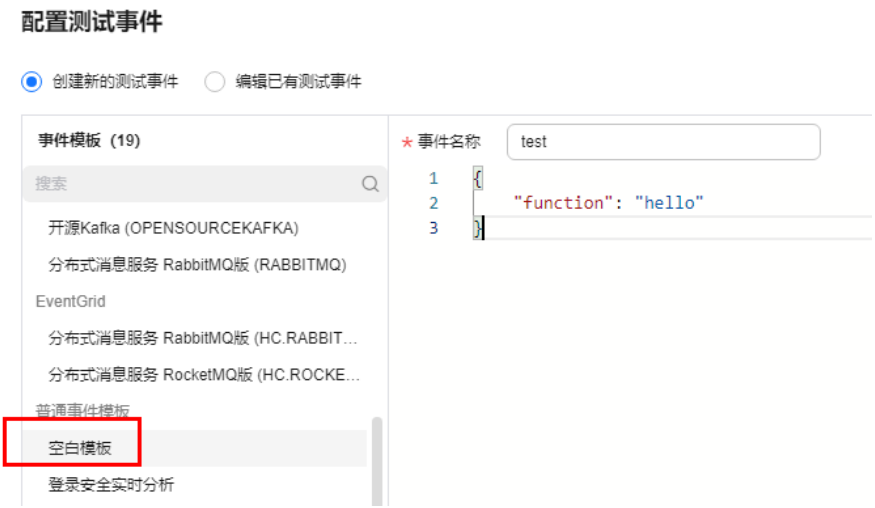
  对于Go runtime，必须在编译之后打zip包，编译后的文件名称必须与函数执行入口的名称保持一致，例如：二进制文件名为handler，则“函数执行入口”命名为handler，Handler与**步骤1**中定义的函数保持一致。

- 测试函数

  a. 创建测试事件。

  在函数详情页，单击“配置测试事件”，弹出“配置测试事件”页，输入测试信息如**图6-1**所示，单击“创建”。

  图 6-1 配置测试事件

  b. 在函数详情页，选择已配置测试事件，单击“测试”。

- 函数执行

  函数执行结果分为三部分，分别为函数返回（由callback返回）、执行摘要、日志输出（由fmt.Println()方法获取的日志方法输出）。

----结束

执行结果

执行结果由3部分组成：函数返回、执行摘要和日志。

表 6-21 执行结果说明

参数项	执行成功	执行失败

函数返回	返回函数中定义的返回信息。	返回包含错误信息和错误类型的JSON文件。格式如下：
		{
		"errorMessage": "",
		"errorType":"",
		}
		errorMessage：Runtime返回的错误信息
		errorType：错误类型

执行摘要	显示请求ID、配置内存、执行时长、实际使用内存和收费时长。	显示请求ID、配置内存、执行时长、实际使用内存和收费时长。
日志	打印函数日志，最多显示4KB的日志。	打印报错信息，最多显示4KB的日志。

7 C#

7.1 开发事件函数

7.1.1 C#函数开发

C#函数接口定义

FunctionGraph运行时目前支持C#(.NET Core 2.1)、C#(.NET Core 3.1)、C#(.NET Core 6.0，当前仅支持华北-乌兰察布二零二、华北-乌兰察布二零一、拉美-墨西哥城二)版本。

C#函数接口定义：*作用域 返回参数 函数名（函数参数，Context参数）*

- 作用域：提供给FunctionGraph调用的用户函数必须定义为public。
- 返回参数：用户定义，FunctionGraph负责转换为字符串，作为HTTP Response返回。
- 函数名：用户自定义函数名称，需要和函数执行入口处用户自定义的入口函数名称一致。
- 执行事件体：函数执行界面由用户输入的执行事件参数。
- 上下文环境（context）：Runtime提供的函数执行上下文，相关属性定义在对象说明中。

  HC.Serverless.Function.Common –部署在FunctionGraph服务中的项目工程需要引入该库，其中包含IFunctionContext对象，详情见context类说明。

  创建csharp函数时，需要定义某个类中的方法作为函数执行入口，该方法可以通过定义IFunctionContext类型的参数来访问当前执行函数的信息。例如：

```
public Stream handlerName(Stream input,IFunctionContext context)
{
    // TODO
}
```

C#函数的函数执行入口参数格式为：**[程序集名]::[命名空间].[类名]::[执行函数名]**，例如CsharpDemo::CsharpDemo.Program::MyFunc，请参考**函数执行入口**通过FunctionGraph控制台进行配置或修改。

说明

建议使用.NET Core 3.1版本。

函数 Handler 定义

ASSEMBLY::NAMESPACE.CLASSNAME::METHODNAME

- .ASSEMBLY为应用程序的.NET程序集文件的名称，假设文件夹名称为HelloCsharp。
- NAMESPACE、CLASSNAME即入口执行函数所在的namespace和class名称。
- METHODNAME即入口执行函数名称。例如：
  创建函数时Handler：HelloCsharp::Example.Hello::Handler。

SDK 接口

- Context接口
  Context类中提供了许多属性供用户使用，如表7-1所示。

表 7-1 Context 对象说明

属性名	属性说明
String RequestId	请求ID。
String ProjectId	Project Id
String PackageName	函数所在分组名称
String FunctionName	函数名称
String FunctionVersion	函数版本
Int MemoryLimitInMb	分配的内存。
Int CpuNumber	获取函数占用的CPU资源。
String Accesskey	获取用户委托的AccessKey（有效期24小时），使用该方法需要给函数配置委托。 **当前函数工作流已停止维护Runtime SDK 中String AccessKey接口，您将无法使用String AccessKey获取临时AK。**
String Secretkey	获取用户委托的SecretKey（有效期24小时），使用该方法需要给函数配置委托。 **当前函数工作流已停止维护Runtime SDK 中String SecretKey接口，您将无法使用String SecretKey获取临时SK。**
String SecurityAccessKey	获取用户委托的SecurityAccessKey（有效期24小时），使用该方法需要给函数配置委托。
String SecuritySecretKey	获取用户委托的SecuritySecretKey（有效期24小时），使用该方法需要给函数配置委托。

属性名	属性说明
String SecurityToken	获取用户委托的SecurityToken（有效期24小时），使用该方法需要给函数配置委托。
String Token	获取用户委托的Token（有效期24小时），使用该方法需要给函数配置委托。
Int RemainingTimeInMilliSeconds	函数剩余运行时间
String GetUserData(string key,string defvalue=" ")	通过key获取用户通过环境变量传入的值。

- 日志接口

  FunctionGraph中C# SDK中接口日志说明如所示。

  表 7-2 日志接口说明

方法名	方法说明

Log(string message)	利用context创建logger对象：
	var logger = context.Logger;
	logger.Log("hello CSharp runtime test(v1.0.2)");

Logf(string format, args ...interface{})	利用context创建logger对象：
	var logger = context.Logger;
	var version = "v1.0.2"
	logger.Logf("hello CSharp runtime test({0})", version);

开发 C#函数

约束与限制：

- 如果是使用FunctionGraph服务提供的样例程序包**fss_example_csharp2.0**，请跳过**步骤1**和**步骤2**，直接执行**步骤3**，并修改函数执行入口为：MyCsharpPro::src.Program::myFunc。
- **当函数的事件源是APIG时，相关约束条件请参考Base64解码和返回结构体的说明**。

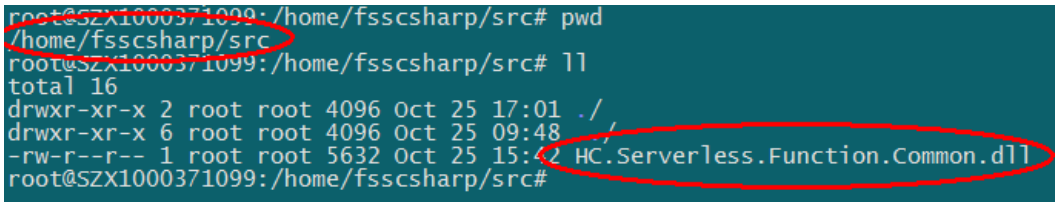
此处以Linux环境，C# (.NET Core 2.0)为例，开发C#函数步骤如下：

步骤1 创建C#编译工程

登录已经安装了.NET SDK和运行环境的linux服务器，创建目录“/home/fsscsharp/src”，下载**dll文件**，将FunctionGraph函数dll文件解压到该目录。如**图7-1**所示。

本文以fssCsharp2.0-1.0.1版本的dll函数为例，不同版本的dll无差异。

图 7-1 函数解压



使用“dotnet --info”命令查看dotnet环境是否已安装，回显代码如下所示：

```
root@SZX1000371099:/home/fsscsharp/src# dotnet --info
.NET Command Line Tools (2.1.202)

Product Information:
 Version:            2.1.202
 Commit SHA-1 hash:  281caedada

Runtime Environment:
 OS Name:     ubuntu
 OS Version:  14.04
 OS Platform: Linux
 RID:         ubuntu.14.04-x64
 Base Path:   /home/lusinking/dotnetdev/sdk/2.1.202/

Microsoft .NET Core Shared Framework Host

  Version  : 2.0.9
  Build    : 1632fa1589b0eee3277a8841ce1770e554ece037
```

创建并初始化console application工程，命令如下：

“dotnet new console -n project_name”

示例命令：

```
dotnet new console -n MyCsharpPro
```

在目录“/home/fsscsharp/src/MyCsharpPro”下的Program.cs代码文件中创建入口执行函数。其中input为客户端请求的body数据，context为FunctionGraph提供的运行时上下文对象，具体提供的属性可以参考属性接口。代码如下：

```
using HC.Serverless.Function.Common;
using System;
using System.IO;
using System.Text;

namespace src
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
        public Stream myFunc(Stream input,IFunctionContext context)
        {
            string payload = "";
            if (input != null && input.Length > 0)
            {
                byte[] buffer = new byte[input.Length];
                input.Read(buffer, 0, (int)(input.Length));
                payload = Encoding.UTF8.GetString(buffer);
            }
            var ms = new MemoryStream();
            using (var sw = new StreamWriter(ms))
```



```
        {
            sw.WriteLine("CSharp runtime test(v1.0.2)");
            sw.WriteLine("====================================");
            sw.WriteLine("Request Id:       {0}", context.RequestId);
            sw.WriteLine("Function Name:    {0}", context.FunctionName);
            sw.WriteLine("Function Version: {0}", context.FunctionVersion);
            sw.WriteLine("Project:          {0}", context.ProjectId);
            sw.WriteLine("Package:          {0}", context.PackageName);
            sw.WriteLine("Security Access Key:       {0}", context.SecurityAccessKey);
            sw.WriteLine("Security Secret Key:       {0}", context.SecuritySecretKey);
            sw.WriteLine("Security Token:           {0}", context.SecurityToken);
            sw.WriteLine("Token:            {0}", context.Token);
            sw.WriteLine("User data(ud-a):  {0}", context.GetUserData("ud-a"));
            sw.WriteLine("User data(ud-notexist): {0}", context.GetUserData("ud-notexist", ""));
            sw.WriteLine("User data(ud-notexist-default): {0}", context.GetUserData("ud-notexist", "default
value"));
            sw.WriteLine("====================================");

            var logger = context.Logger;
            logger.Logf("Hello CSharp runtime test(v1.0.2)");
            sw.WriteLine(payload);
        }
        return new MemoryStream(ms.ToArray());
    }
  }
}
```

步骤2 编译C#工程

手动在项目配置文件“MyCsharpPro.csproj”中添加FunctionGraph服务提供的dll引用（HinPath中填入dll的相对路径）。如下所示：

```
<Project Sdk="Microsoft.NET.Sdk">

 <PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>netcoreapp2.0</TargetFramework>
 </PropertyGroup>
 <ItemGroup>
  <Reference Include="HC.Serverless.Function.Common, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null">
    <HintPath>../HC.Serverless.Function.Common.dll</HintPath>
  </Reference>
 </ItemGroup>
</Project>
```

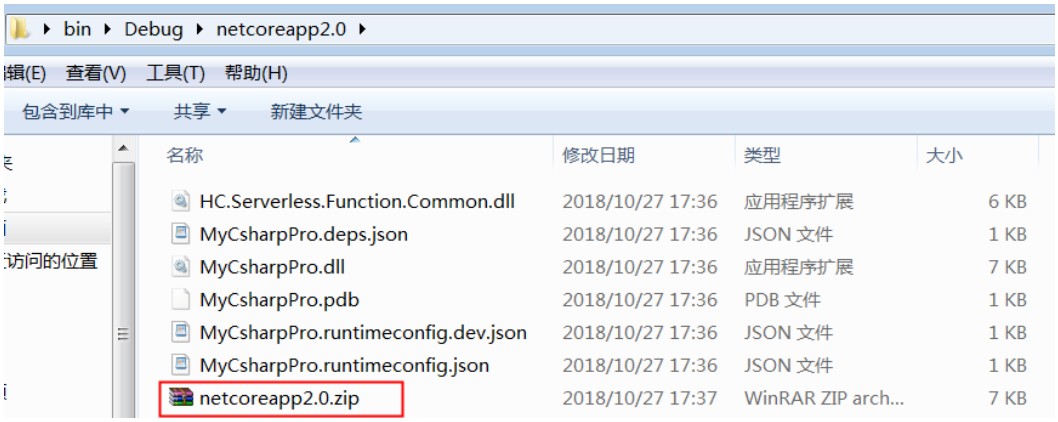
用“dotnet build”命令编译工程，回显信息如下：

```
root@SZX1000371099:/home/fsscsharp/src/MyCsharpPro# vi MyCsharpPro.csproj
root@SZX1000371099:/home/fsscsharp/src/MyCsharpPro# dotnet build
Microsoft (R) Build Engine version 15.7.179.6572 for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

 Restore completed in 54.28 ms for /home/fsscsharp/src/MyCsharpPro/MyCsharpPro.csproj.
 MyCsharpPro -> /home/fsscsharp/src/MyCsharpPro/bin/Debug/netcoreapp2.0/MyCsharpPro.dll

Build succeeded.
   0 Warning(s)
   0 Error(s)

Time Elapsed 00:00:01.47
```

步骤3 部署C#工程到FunctionGraph服务

用ssh工具将编译后的文件拷贝并打包，如图7-2所示。

图 7-2 打包文件



创建函数并上传上一步的zip包。

执行函数，函数执行结果分为三部分，分别为函数返回（由callback返回）、执行摘要、日志输出（由Console.WriteLine()方法输出）。

----结束

执行结果

执行结果由3部分组成：函数返回、执行摘要和日志。

表 7-3 执行结果说明

参数项	执行成功	执行失败

函数返回	返回函数中定义的返回信息。	返回包含错误信息和错误类型的JSON文件。格式如下：
		{
		"errorMessage": "",
		"errorType": ""
		}
		errorMessage：Runtime返回的错误信息
		errorType：错误类型

执行摘要	显示请求ID、配置内存、执行时长、实际使用内存和收费时长。	显示请求ID、配置内存、执行时长、实际使用内存和收费时长。
日志	打印函数日志，最多显示4KB的日志。	打印报错信息，最多显示4KB的日志。

7.1.2 函数支持 json 序列化和反序列化

7.1.2.1 使用 NET Core CLI

C#新增json序列化和反序列化接口，并提供HC.Serverless.Function.Common.JsonSerializer.dll 。

提供的接口如下：

T Deserialize<T>(Stream ins)：反序列化值传递到Function处理程序的对象中。

Stream Serialize<T>(T value)：序列化值传递到返回的响应负载中。

本例以.NET Core2.1创建“test”工程为例说明，.NET Core3.1、C#(.NET Core 6.0，当前仅支持华北-乌兰察布二零二、华北-乌兰察布二零一、拉美-墨西哥城二)方法类似。执行环境已装有.NET SDK2.1。

新建项目

1. 创建目录“/tmp/csharp/projects /tmp/csharp/release” ，执行命令如下：
```
mkdir -p  /tmp/csharp/projects;mkdir -p  /tmp/csharp/release
```
2. 进入“/tmp/csharp/projects/”目录，执行命令如下：
```
cd /tmp/csharp/projects/
```
3. 新建工程文件 “test.csproj”，文件内容如下：
```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.1</TargetFramework>
    <RootNamespace>test</RootNamespace>
    <AssemblyName>test</AssemblyName>
  </PropertyGroup>
  <ItemGroup>
    <Reference Include="HC.Serverless.Function.Common">
      <HintPath>HC.Serverless.Function.Common.dll</HintPath>
    </Reference>
    <Reference Include="HC.Serverless.Function.Common.JsonSerializer">
      <HintPath> HC.Serverless.Function.Common.JsonSerializer.dll</HintPath>
    </Reference>
  </ItemGroup>
</Project>
```

生成代码库

1. 下载**dll文件**并上传。

   将HC.Serverless.Function.Common.dll、HC.Serverless.Function.Common.JsonSerializer.dll、Newtonsoft.Json.dll文件上传至目录“/tmp/csharp/projects/”。
2. 在“/tmp/csharp/projects/”路径下，新建“Class1.cs”文件，代码内容如下：
```
using HC.Serverless.Function.Common;
using System;
using System.IO;

namespace test
{
    public class Class1
    {
        public Stream ContextHandlerSerializer(Stream input, IFunctionContext context)
        {
            var logger = context.Logger;
            logger.Logf("CSharp runtime test(v1.0.2)");
            JsonSerializer test = new JsonSerializer();
            TestJson Testjson = test.Deserialize<TestJson>(input);
            if (Testjson != null)
```



```
        {
            logger.Logf("json Deserialize KetTest={0}", Testjson.KetTest);

        }
        else
        {
            return null;
        }

       return test.Serialize<TestJson>(Testjson);
    }

    public class TestJson
    {
        public string KetTest { get; set; }//定义序列化的类中的属性为KetTest

    }
  }
}
```

3. 执行以下命令，生成代码库:

```
/home/tools/dotnetcore-sdk/dotnet-sdk-2.1.302-linux-x64/dotnet build /tmp/csharp/projects/test.csproj -c Release -o /tmp/csharp/release
```

说明

dotnet的路径：/home/tools/dotnetcore-sdk/dotnet-sdk-2.1.302-linux-x64/dotnet。

4. 执行以下命令，进入“/tmp/csharp/release”路径。

```
cd /tmp/csharp/release
```

5. 在路径“/tmp/csharp/release”下查看编译生成的dll文件，如下所示：

```
-rw-r--r-- 1 root root 468480 Jan 21 16:40 Newtonsoft.Json.dll
-rw-r--r-- 1 root root   5120 Jan 21 16:40 HC.Serverless.Function.Common.JsonSerializer.dll
-rw-r--r-- 1 root root   5120 Jan 21 16:40 HC.Serverless.Function.Common.dll
-rw-r--r-- 1 root root    232 Jan 21 17:10 test.pdb
-rw-r--r-- 1 root root   3584 Jan 21 17:10 test.dll
-rw-r--r-- 1 root root   1659 Jan 21 17:10 test.deps.json
```

6. 在“/tmp/csharp/release”路径下，新建文件“test.runtimeconfig.json”文件，文件内容如下：

```
{
  "runtimeOptions": {
    "framework": {
      "name": "Microsoft.NETCore.App",
      "version": "2.1.0"
    }
  }
}
```


说明

- *.runtimeconfig.json文件的名称为程序集的名称。
- 文件内容中的version：项目属性中的目标框架的版本号，2.0则为2.0.0 ，2.1则为2.1.0。
- 当目标框架为.NET Core2.0时，要注意生成*.deps.json文件中是否已引入Newtonsoft.Json。如果没有引入，则需要自己手动引入，如下所示：
  1. 需要在“targets”中引入以下内容：
  ```
  "Newtonsoft.Json/9.0.0.0": {
        "runtime": {
          "Newtonsoft.Json.dll": {
            "assemblyVersion": "9.0.0.0",
            "fileVersion": "9.0.1.19813"
          }
        }
      }
  ```
  2. 在“libraries”引入以下内容：
  ```
  "Newtonsoft.Json/9.0.0.0": {
      "type": "reference",
      "serviceable": false,
      "sha512": ""
    }
  ```

7. 在“/tmp/csharp/release”路径下，执行如下命令，打包test.zip代码库压缩包。
```
zip -r test.zip ./*
```

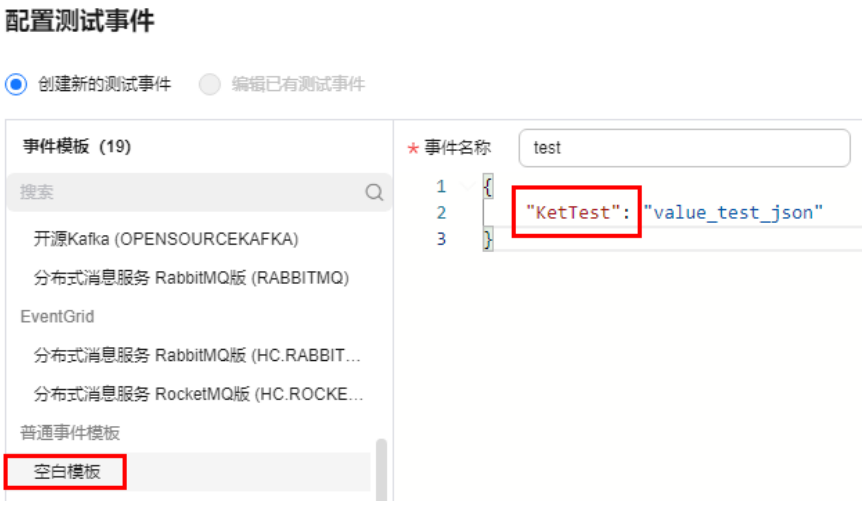
测试示例

1. 在华为云FunctionGraph控制台新建一个C#（.NET 2.1）函数，上传打包好的“test.zip”压缩包，如图7-3所示。

图 7-3 上传代码包

2. 配置一个测试事件。如图7-4所示。其中的key必须设置为“KetTest”，value可以自定义。（**测试串必须为json格式。**）

图 7-4 配置测试事件



说明

KetTest：定义序列化的类中的属性为KetTest。

3. 单击“测试”，查看测试执行结果。

7.1.2.2 使用 Visual Studio

新增json序列化和反序列化接口，并提供HC.Serverless.Function.Common.JsonSerializer.dll 。

提供的接口如下：

T Deserialize<T>(Stream ins)：反序列化值传递到Function处理程序的对象中。

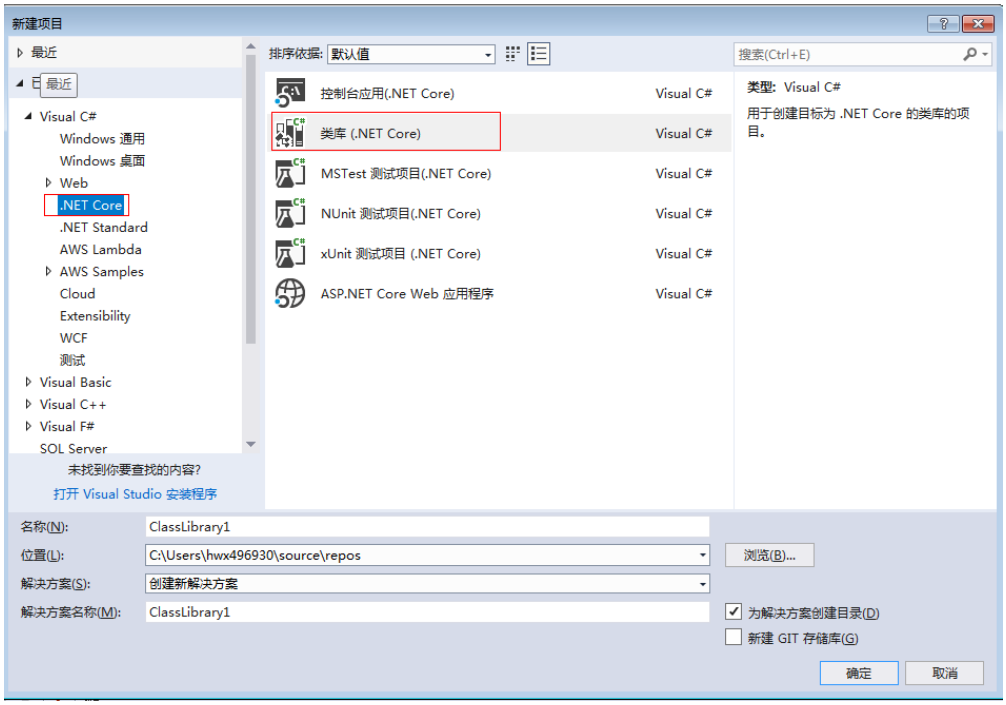
Stream Serialize<T>(T value)：序列化值传递到返回的响应负载中。

本例以Visual Studio 2017新建一个.NET Core2.0的“test”工程，.NET Core2.1、.NET Core3.1、C#(.NET Core 6.0，当前仅支持华北-乌兰察布二零二、华北-乌兰察布二零一、拉美-墨西哥城二)类似。

新建项目

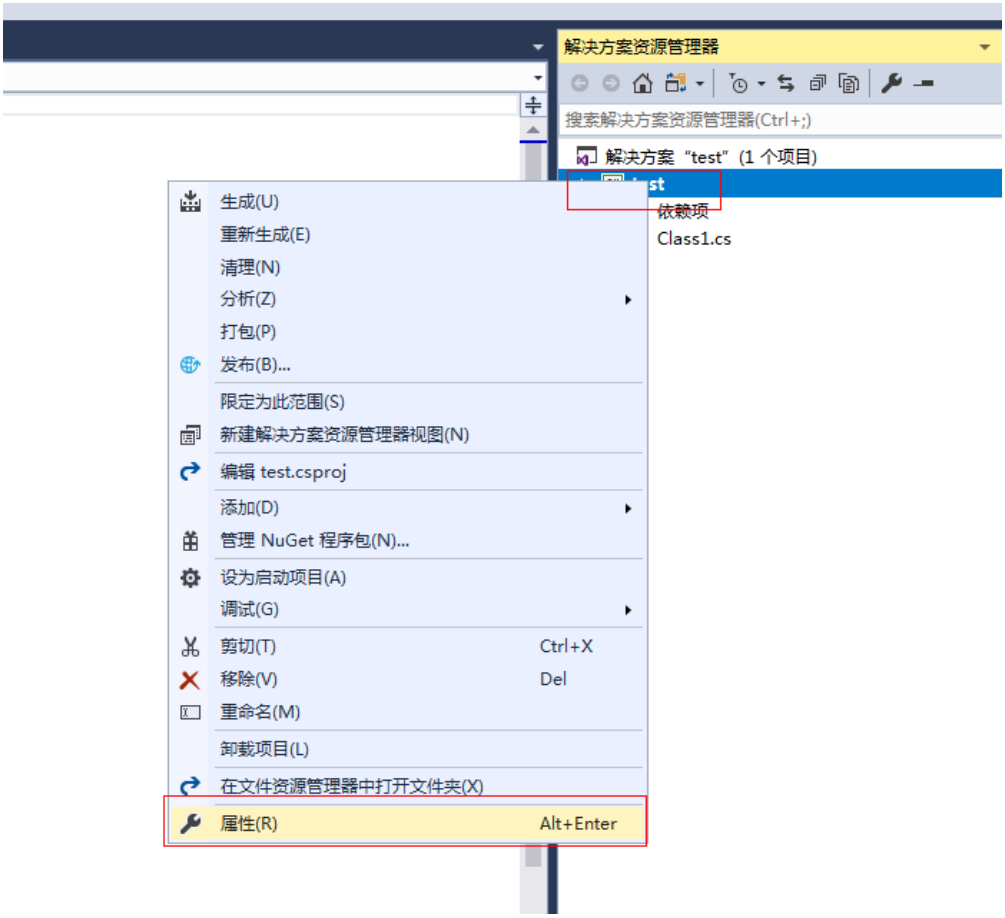
1. 在工具栏中选择“文件 > 新建 > 项目”，选择“.NET Core”，选择“类库(.NET Core)”，并将名称修改为“test”。如图7-5所示。

图 7-5 新建项目



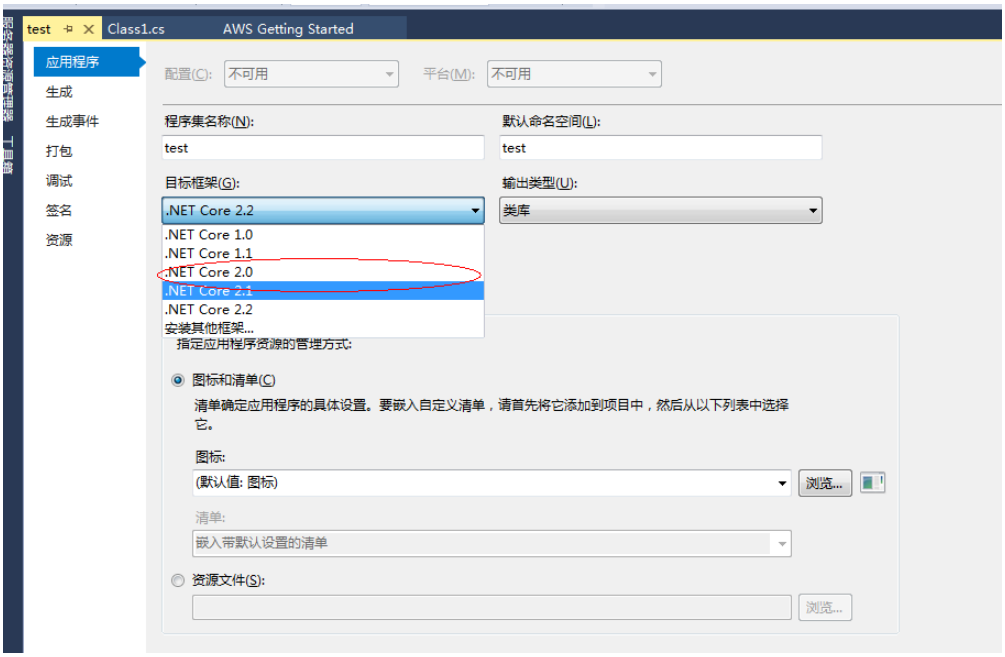
2. 导航栏中选择“test”项目，单击鼠标右键，选择“属性”，打开属性界面。如图7-6所示。

图 7-6 属性



3. 在属性界面选择“应用程序”，选择目标框架为“.NET Core 2.0”，如图7-7所示。

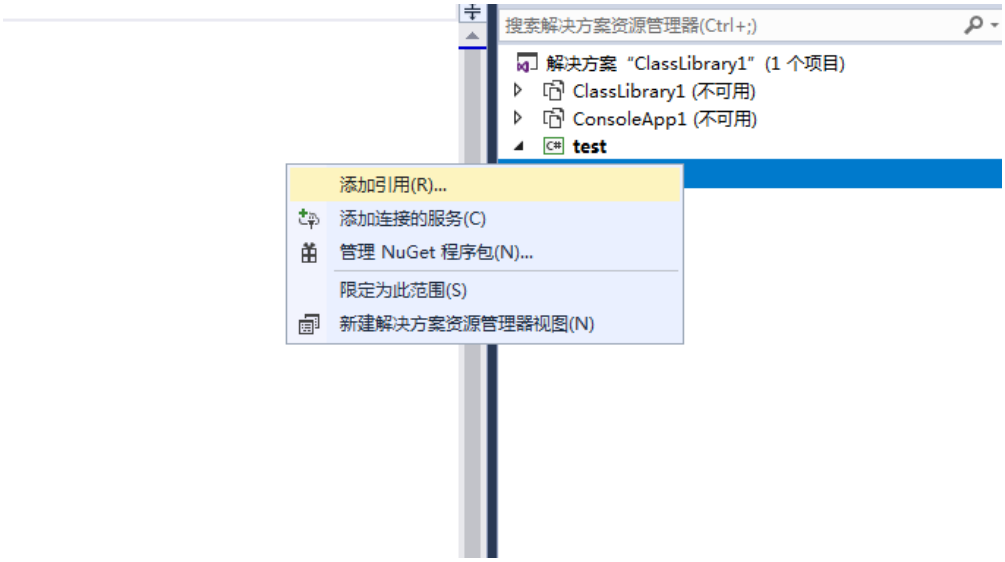
图 7-7 选择目标框架



添加引用

1. 选择解决方案资源管理器中“test”工程，单击鼠标右键，选择“添加引用”，下载**dll文件**引用进来。如**图7-8**所示。

   图 7-8 添加引用

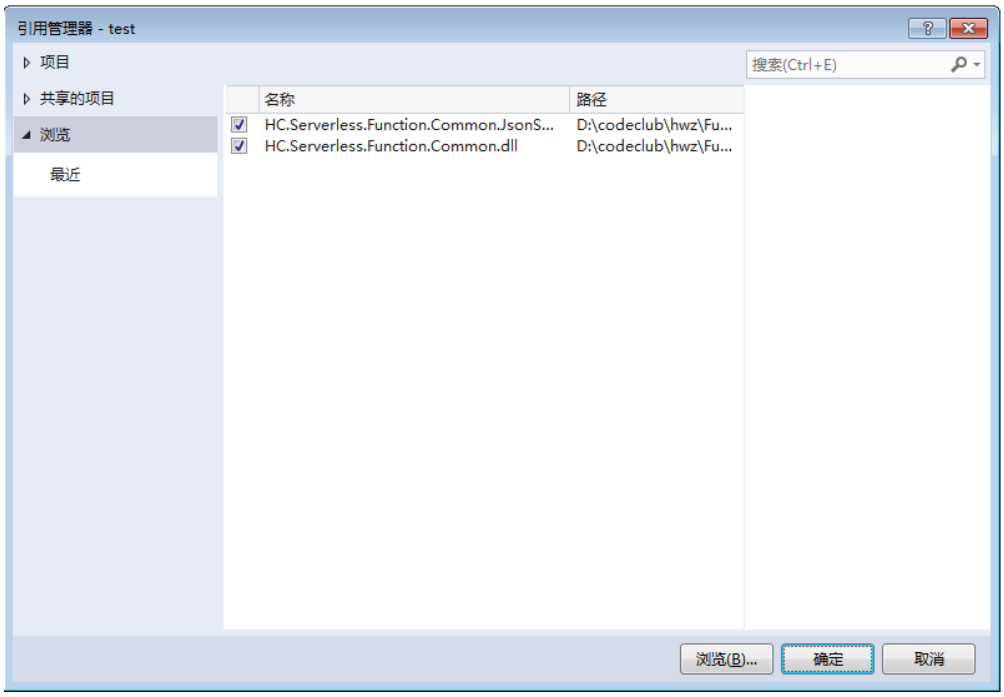
   

   说明

   所引用的dll下载后放在一个lib文件中，一共有三个库：
   HC.Serverless.Function.Common.dll、
   HC.Serverless.Function.Common.JsonSerializer.dll、Newtonsoft.Json.dll。

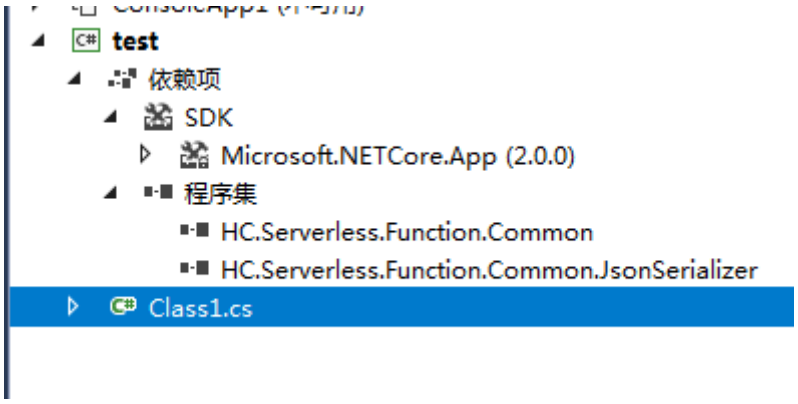
2. 选择“浏览”，单击“浏览(B) ”，把HC.Serverless.Function.Common.dll和HC.Serverless.Function.Common.JsonSerializer.dll引用进来，单击“确定”。如**图7-9**所示。

图 7-9 引用文件



3. 引用成功后界面如图7-10所示。

图 7-10 完成引用



打包代码

本例所用示例代码如下：

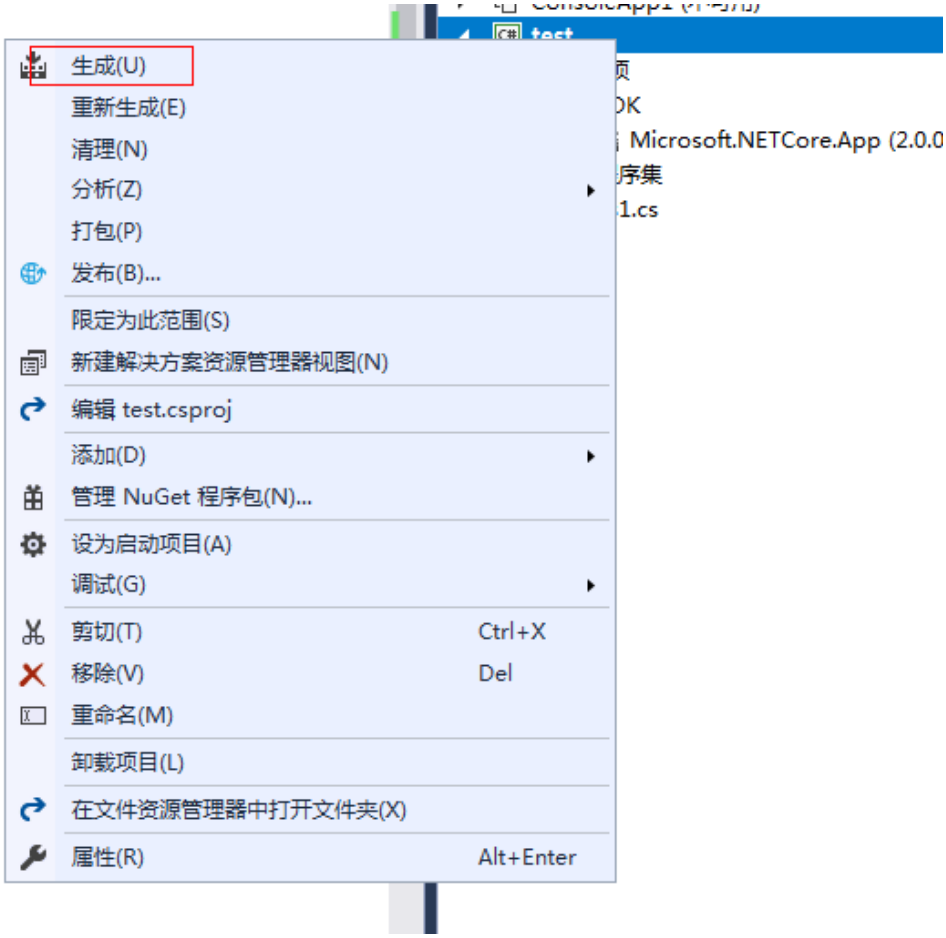
```
using HC.Serverless.Function.Common;
using System;
using System.IO;

namespace test
{
    public class Class1
    {
        public Stream ContextHandlerSerializer(Stream input, IFunctionContext context)
        {
            var logger = context.Logger;
            logger.Logf("CSharp runtime test(v1.0.2)");
            JsonSerializer test = new JsonSerializer();
            TestJson Testjson = test.Deserialize<TestJson>(input);
```



```
        if (Testjson != null)
        {
            logger.Logf("json Deserialize KetTest={0}", Testjson.KetTest);
        }

        return test.Serialize<TestJson>(Testjson);
    }

    public class TestJson
    {
        public string KetTest { get; set; }//定义序列化的类中的属性为KetTest

    }
  }
}
```

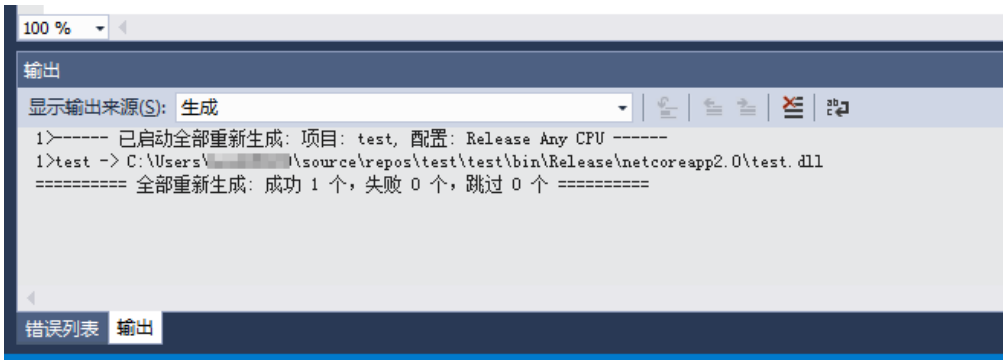
1. 右击“test”工程，选择“生成”，如图7-11所示。

图 7-11 生成文件



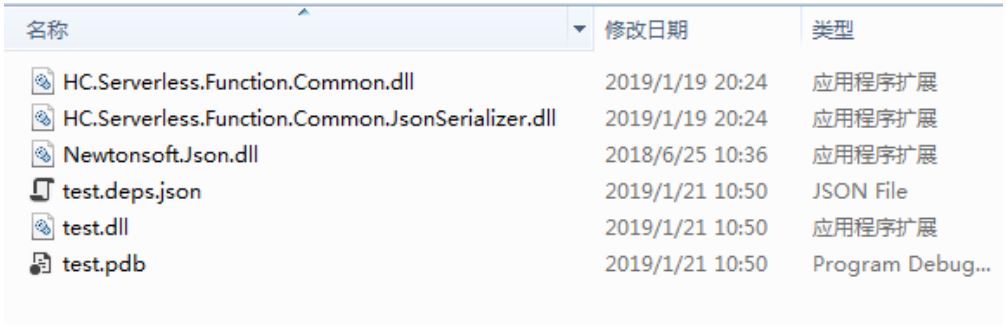
2. 拷贝生成dll文件的路径“C:\Users\xxx\source\repos\test\test\bin\Release\netcoreapp2.0\”，如图7-12所示。

图 7-12 生成路径



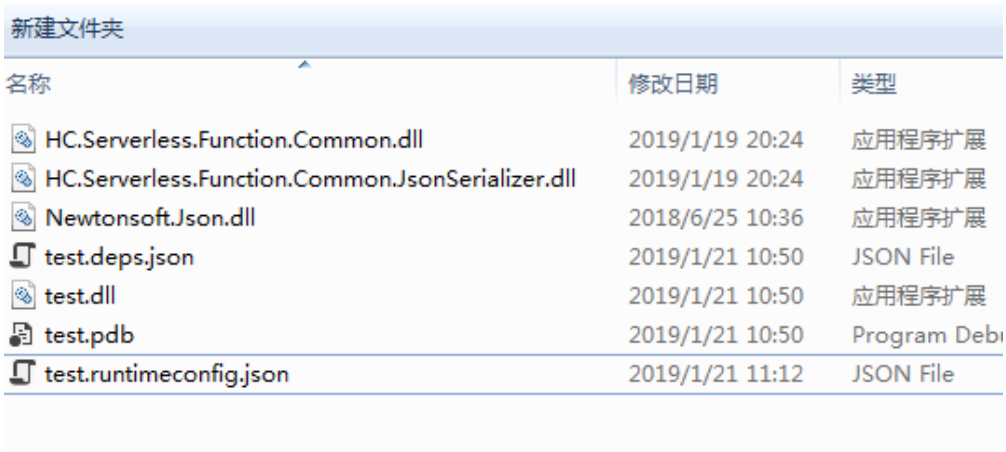
该路径下的文件如图7-13所示。

图 7-13 文件



3. 在该路径新建“test.runtimeconfig.json”文件，如图7-14所示。

图 7-14 新建文件



文件内容如下：

```
{

"runtimeOptions": {

"framework": {

"name": "Microsoft.NETCore.App",
```



```
"version": "2.0.0"
      }
   }
}
```

说明

- *.runtimeconfig.json文件的名称为程序集的名称。
- 文件内容中的version为项目属性中的目标框架的版本号，2.0则为2.0.0 ，2.1则为2.1.0。

4. 将文件打包为netcoreapp2.0.zip压缩包。（**压缩包文件名称可随意，但是一定为.zip格式。**）

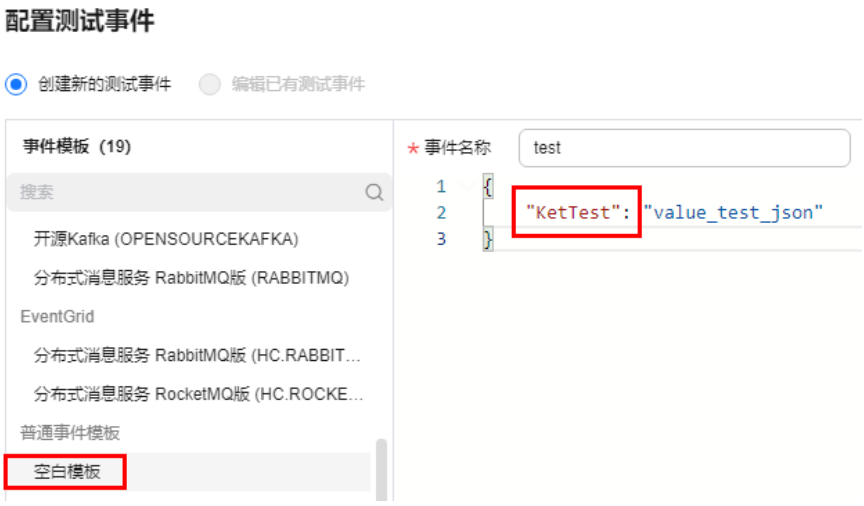
测试示例

1. 在华为云FunctionGraph控制台新建一个C#（.NET 2.1）函数，上传打包完成的代码包，如图7-15所示。

图 7-15 上传代码包

2. 配置一个测试事件。如图7-16所示。其中的key必须设置为“KetTest”，value可以自定义。（**测试串必须为json格式。**）

图 7-16 配置测试事件



说明

KetTest：定义序列化的类中的属性为KetTest.

3. 单击“测试”，查看测试执行结果。

8 PHP

8.1 开发事件函数

PHP 函数接口定义

FunctionGraph运行时支持PHP 7.3版本，PHP 7.3函数的接口定义如下所示：

```
function handler($event, $context)
```

- 入口函数名（$handler）：入口函数名称， 需和函数执行入口处用户自定义的入口函数名称一致。
- 执行事件（$event）：函数执行界面由用户输入的执行事件参数， 格式为JSON对象。
- 上下文环境（$context）：Runtime提供的函数执行上下文，其接口定义在**SDK接口**说明。
- 函数执行入口： index.handler。

PHP函数的函数执行入口参数格式为：**[文件名].[函数名]**，请参考**函数执行入口**通过FunctionGraph控制台进行配置或修改。

约束与限制

getToken()、getAccessKey()和getSecretKey()方法返回的内容包含敏感信息，请谨慎使用，避免造成用户敏感信息的泄露。

PHP 的 initializer 入口介绍

函数工作流服务目前支持以下PHP运行环境。

- Php 7.3 (runtime = Php7.3)

Initializer格式为：**[文件名].[initializer名]**

示例：创建函数时指定的initializer为main.my_initializer，那么FunctionGraph会去加载main.php中定义的my_initializer函数。

在函数工作流服务中使用PHP实现initializer接口，需要定义一个PHP函数作为initializer入口，一个最简单的initializer示例如下。


```
<?php
Function my_initializer($context) {
   echo 'hello world' . PHP_EOL;
   }
?>
```

- 函数名

  my_initializer需要与实现initializer接口时的initializer字段相对应。

  示例：实现initializer接口时指定的Initializer入口为main.my_initializer，那么FunctionGraph会去加载main.php中定义的my_initializer函数。

- context参数

  context参数中包含一些函数的运行时信息，例如：request id、临时AccessKey、function meta等。

SDK 接口

Context类中提供了许多上下文方法供用户使用，其声明和功能如所示。

表 8-1 Context 类上下文方法说明

方法名	方法说明
getRequestID()	获取请求ID。
getRemainingTimeInMilliSeconds ()	获取函数剩余运行时间。
getAccessKey()	获取用户委托的AccessKey（有效期24小时），使用该方法需要给函数配置委托。 **当前函数工作流已停止维护Runtime SDK 中getAccessKey接口，您将无法使用getAccessKey获取临时AK。**
getSecretKey()	获取用户委托的SecretKey（有效期24小时），使用该方法需要给函数配置委托。 **当前函数工作流已停止维护Runtime SDK 中getSecretKey接口，您将无法使用getSecretKey获取临时SK。**
getSecurityAccessKey()	获取用户委托的SecurityAccessKey（有效期24小时），使用该方法需要给函数配置委托。
getSecuritySecretKey()	获取用户委托的SecuritySecretKey（有效期24小时），使用该方法需要给函数配置委托。
getSecurityToken()	获取用户委托的SecurityToken（有效期24小时），使用该方法需要给函数配置委托。
getUserData(string key)	通过key获取用户通过环境变量传入的值。
getFunctionName()	获取函数名称。
getRunningTimeInSeconds ()	获取函数超时时间。

方法名	方法说明
getVersion()	获取函数的版本。
getMemorySize()	分配的内存。
getCPUNumber()	获取函数占用的CPU资源。
getPackage()	获取函数组。
getToken()	获取用户委托的token（有效期24小时），使用该方法需要给函数配置委托。
getLogger()	获取context提供的logger方法，返回一个日志输出类，通过使用其info方法按“时间-请求ID-输出内容”的格式输出日志。 如调用info方法输出日志： logg = context.getLogger()$ $logg->info("hello")
getAlias	获取函数的别名

开发 PHP 函数

约束与限制：

- 函数仅支持返回如下几种类型的值。
  - Null：函数返回的HTTP响应Body为空。
  - string：函数返回的HTTP响应Body内容为该字符串内容。
  - 其他：函数会返回值作为对象进行json编码，并将编码后的内容作为HTTP响应的Body，同时设置响应的“Content-Type”头为“text/plain”。
- 当函数的事件源是APIG时，相关约束条件请参考**Base64解码和返回结构体的说明**。
- 本例函数工程文件保存在“~/Code/”文件夹下，在打包的时候务必进入Code文件夹下选中所有工程文件进行打包，这样做的目的：由于定义了FunctionGraph函数的index.php是程序执行入口，确保fss_examples_php7.3.zip解压后，index.php文件位于根目录。

开发PHP函数步骤如下：

步骤1 创建函数

1. 编写打印helloworld的代码。

   打开文本编辑器，编写helloworld函数，代码如下，文件命名为“helloworld.php”，保存文件。

   ```
   <?php
   function printhello() {
   echo 'Hello world!';
   }
   ```

2. 定义FunctionGraph函数

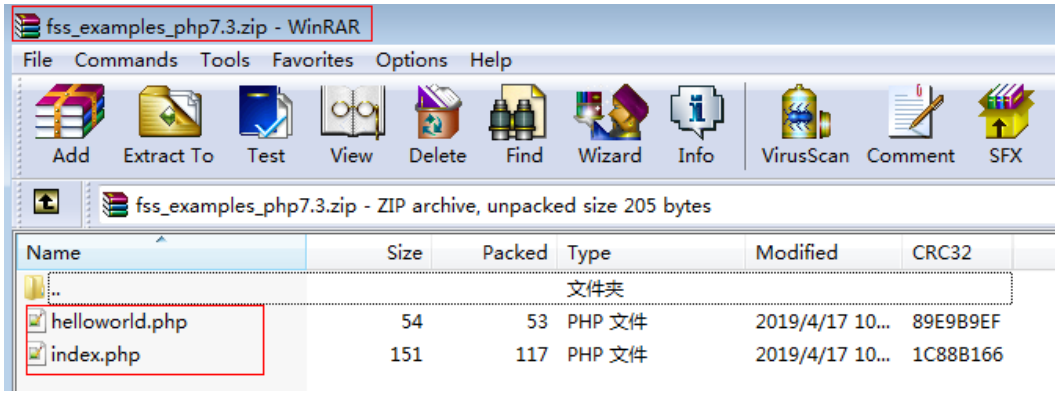
   打开文本编辑，定义函数，代码如下，文件命名为index.php，保存文件（与helloworld.php保存在同一文件夹下）。


```
<?php
include_once 'helloworld.php';

function handler($event, $context) {
    $output = json_encode($event);
    printhello();
    return $output;
}
```

步骤2 工程打包

函数工程创建以后，可以得到以下目录，选中工程所有文件，打包命名为“fss_examples_php7.3.zip”，如图8-1所示。

图 8-1 工程打包



步骤3 创建FunctionGraph函数，上传程序包

登录FunctionGraph，创建PHP函数，上传fss_examples_php7.3.zip文件。如 图8-2 所示。

图 8-2 上传程序包

函数执行入口中的index与步骤定义FunctionGraph函数的文件名保持一致，通过该名称找到FunctionGraph函数所在文件。

函数执行入口中的handler为函数名，与步骤定义FunctionGraph函数中创建的index.php文件中的函数名保持一致。

函数执行过程为：用户上传fss_examples_php7.3.zip保存在OBS中，触发函数后，解压缩zip文件，通过index匹配到FunctionGraph函数所在文件，通过handler匹配到index.php文件中定义的FunctionGraph函数，找到程序执行入口，执行函数。

在函数工作流控制台左侧导航栏选择“函数 > 函数列表”，单击需要设置的“函数名称”进入函数详情页，选择“设置 > 常规设置”，配置“函数执行入口”参数，如图

8-3所示。其中参数值为“index.handler”格式，“**index**”和“**handler**”支持自定义命名。

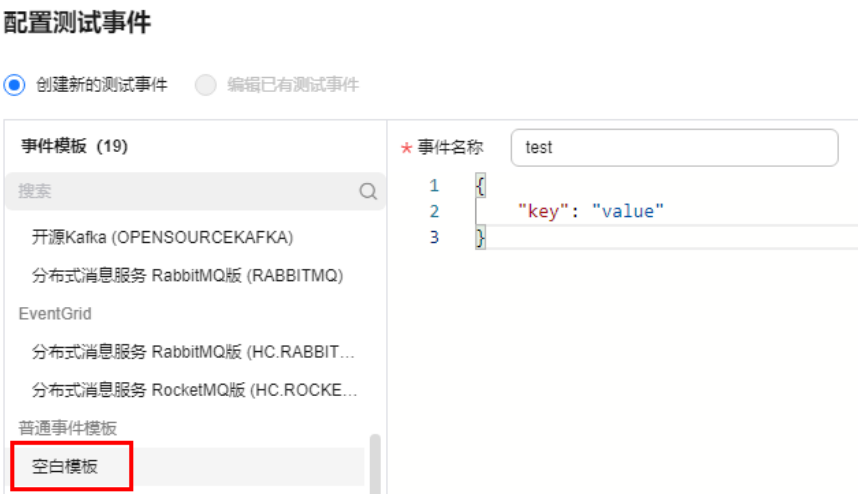
图 8-3 函数执行入口参数

步骤4 测试函数

1. 创建测试事件。

   在函数详情页，单击“配置测试事件”，弹出“配置测试事件”页，输入测试信息如图8-4所示，单击“创建”。

   图 8-4 配置测试事件



2. 在函数详情页，选择已配置测试事件，单击“测试”。

步骤5 执行函数

函数执行结果分为三部分，分别为函数返回（由return返回）、执行摘要、日志输出（由echo方法获取的日志方法输出）。

----结束

执行结果

执行结果由3部分组成：函数返回、执行摘要和日志。

表 8-2 执行结果说明

参数项	执行成功	执行失败

函数返回	返回函数中定义的返回信息。	返回包含错误信息、错误类型和堆栈异常报错信息的JSON文件。格式如下：
		{
		"errorMessage": "",
		"errorType": "",
		"stackTrace": {}
		}
		errorMessage：Runtime返回的错误信息
		errorType：错误类型
		stackTrace：Runtime返回的堆栈异常报错信息

执行摘要	显示请求ID、配置内存、执行时长、实际使用内存和收费时长。	显示请求ID、配置内存、执行时长、实际使用内存和收费时长。
日志	打印函数日志，最多显示4KB的日志。	打印报错信息，最多显示4KB的日志。

8.2 制作依赖包

制作函数依赖包推荐在Huawei Cloud EulerOS 2.0环境中进行。使用其他系统打包可能会因为底层依赖库的原因，运行出问题，比如找不到动态链接库。

约束与限制

如果安装的依赖模块需要添加依赖库，请将依赖库归档到zip依赖包文件中，例如，添加.dll、.so、.a等依赖库。

为 PHP 函数制作依赖包

制作函数依赖包推荐在Huawei Cloud EulerOS 2.0环境中进行。

为php7.3通过composer安装protobuf3.19依赖包，默认环境中已经安装了composer和对应版本的php。

新建一个composer.json文件，在composer.json中填入以下内容。

```
{
    "require": {
        "google/protobuf": "^3.19"
    }
}
```

执行如下命令。

```
Composer install
```

可以看到当前目录底下生成一个vendor文件夹，文件夹中有autoload.php、composer和google三个文件夹。

- Linux系统

Linux系统下可以使用以下命令生成zip包。

```
zip –rq vendor.zip vendor
```

- windows系统

  用压缩软件将vendor目录压缩成zip文件即可。

  如果要安装多个依赖包，在composer.json文件中指定需要的依赖，把生成的vendor文件整体打包成zip上传。

说明

php工程代码中使用通过composer下载的第三方依赖时，需要通过require "./vendor/autoload.php" 加载，平台默认把上传的zip包解压后的内容置于项目代码的同级目录下。

9 开发工具

9.1 CodeArts IDE Online

9.1.1 CodeArts IDE Online 在线管理函数

用户通过CodeArts IDE Online在线管理函数，调试方便，界面友好，帮忙用户快速创建函数。CodeArts IDE Online工具支持以下功能：

1. 用户在FunctionGraph控制台创建函数后，将函数下载到CodeArts IDE Online在线进行编辑，编辑完成后，再将修改好的函数推送到FunctionGraph控制台。
2. 用户在CodeArts IDE Online创建函数并完成编辑，再将函数推送到FunctionGraph控制台。

说明

该功能当前仅“华南-广州、华北-北京四”区域支持。

约束与限制

当前仅Node.js、Java、Python语言支持CodeArts IDE Online在线管理函数。

前提条件

1. 新建实例选择All In One实例，直接包含了需要的Java、Python等插件。
2. 已经在FunctionGraph控制台创建函数，创建过程请参见**构建函数**。
3. 使用CodeArts IDE Online如遇缺少相关操作权限时，权限授权详情请参考**自定义权限策略**。

FunctionGraph 控制台创建函数

用户在FunctionGraph控制台创建函数后，将函数下载到CodeArts IDE Online在线进行编辑，编辑完成后，再将修改好的函数推送到FunctionGraph控制台。

步骤1 登录FunctionGraph控制台，在左侧导航栏选择“函数 > 函数列表”。

步骤2 创建函数完成后，在“代码”页签，单击“在CodeArts IDE Online中打开”或者直接在线编辑。

步骤3 在新打开的“选择实例”页面，单击“创建新实例”，输入实例名称，单击“确定”。

图 9-1 创建新实例

步骤4 进入CodeArts IDE Online在线编辑页面。

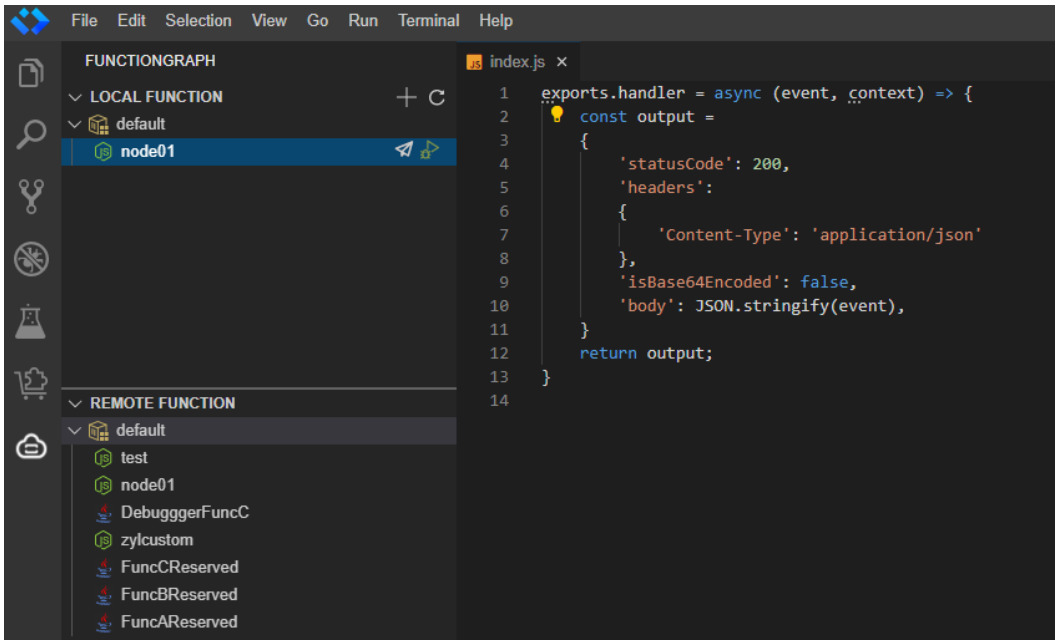
说明

首次进入CodeArts IDE Online在线编辑页面，提示选择切换为中文语言。

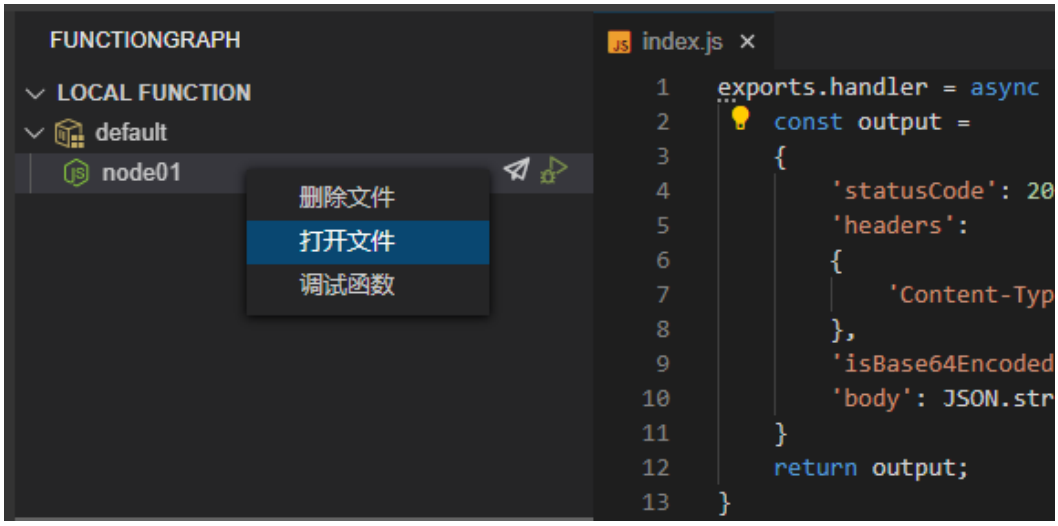
步骤5 在编辑页面左侧导航栏单击FunctionGraph插件后，左侧编辑框中的REMOTE FUNCTION文件中即可看到在FunctionGraph控制台创建好的所有函数和应用。

说明

选择从哪个函数进入CodeArts IDE Online在线编辑页面的，插件自动下载并打开该函数，显示在LOCAL FUNCTION目录下。



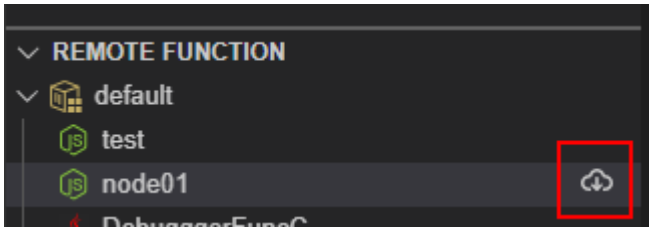
步骤6 在LOCAL FUNCTION文件中，在编辑框调试函数，双击“node01”函数或者右键选择“打开文件”，在右侧编辑框打开文件。



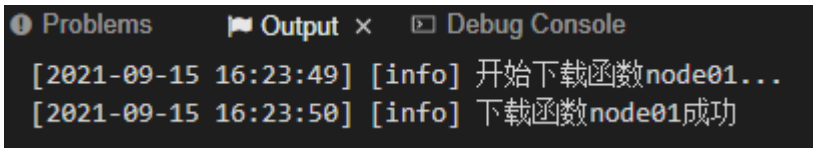
或者在REMOTE FUNCTION文件中添加需要编辑的函数，并下载到CodeArts IDE Online进行在线编辑。

说明

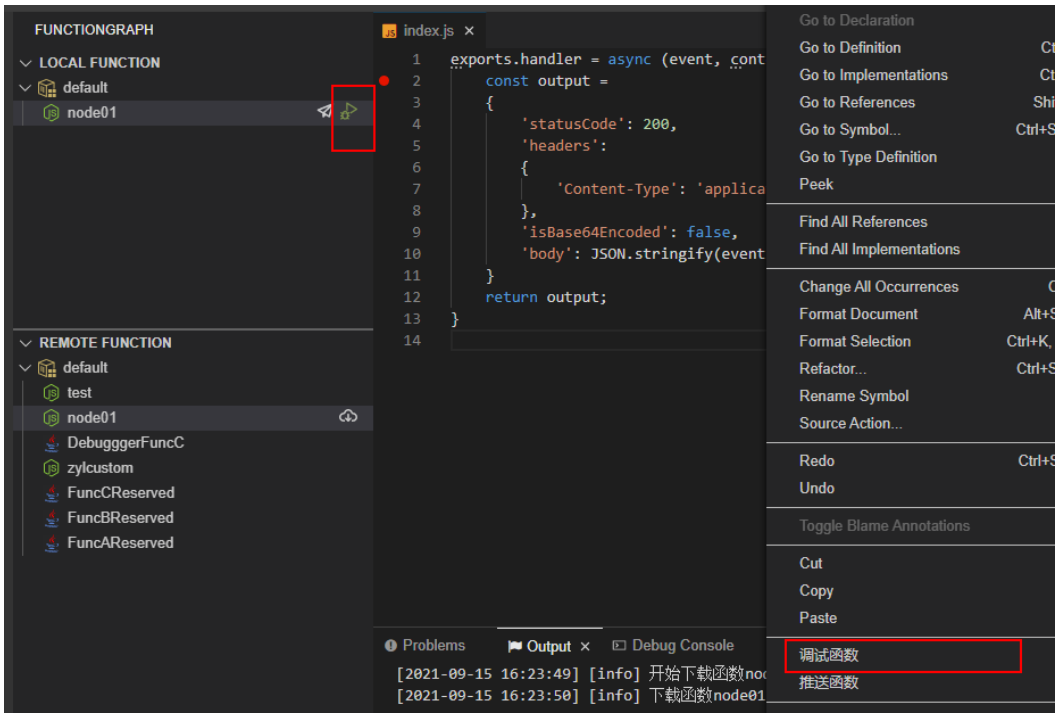
以下示例仅供参考，具体请以实际创建函数为准。



步骤7 下载成功后，右侧输出控制台提示下载成功。



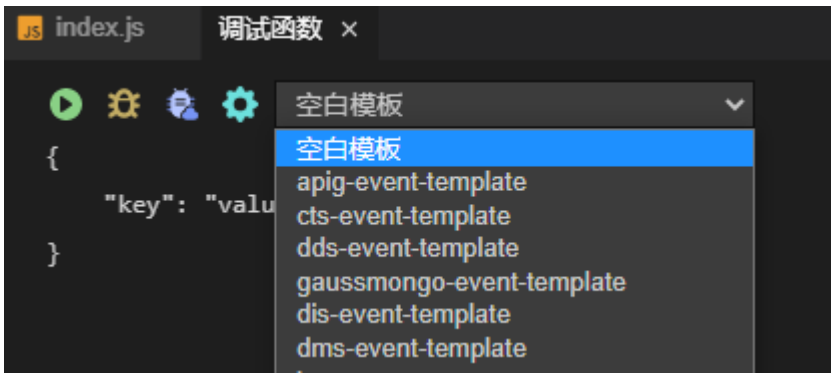
步骤8 打完断点，单击调试图标或者右键选择“调试函数”。

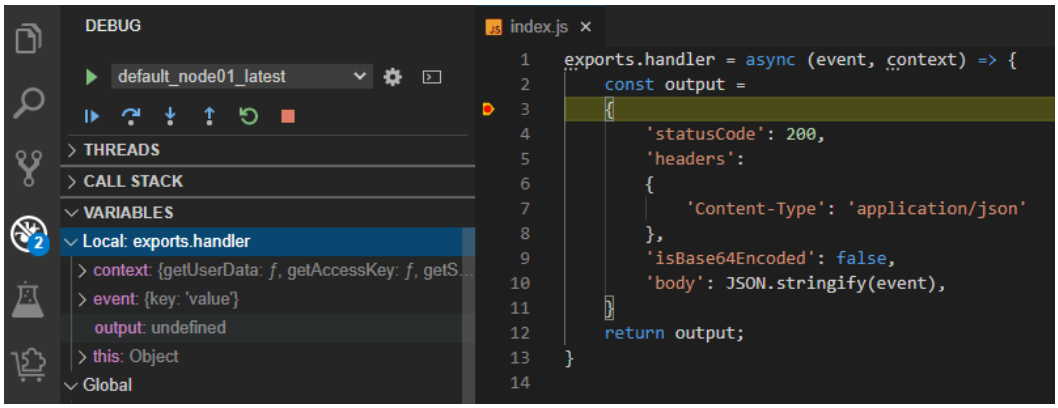


步骤9 进入函数调试页面，选择测试事件，单击“调试”。

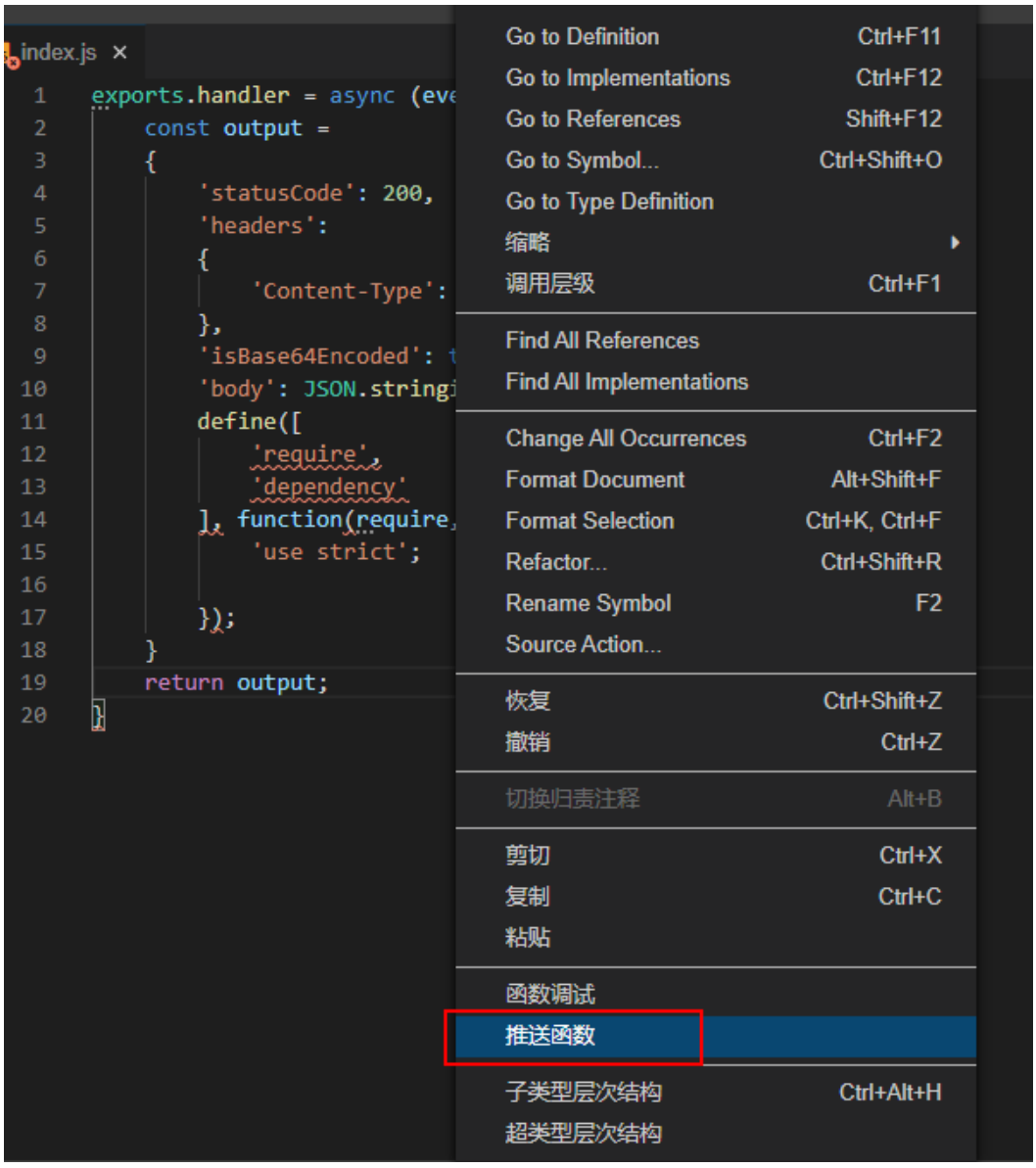
说明

配置测试事件和测试函数请参考**测试管理**。

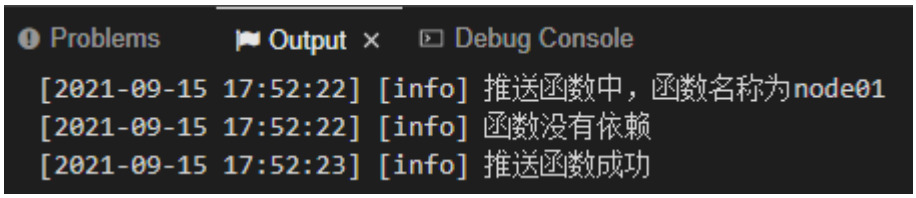




步骤10 修改函数代码后，单击“推送函数”图标或者右键选择“推送函数”。



步骤11 页面下方输出控制台提示推送成功。



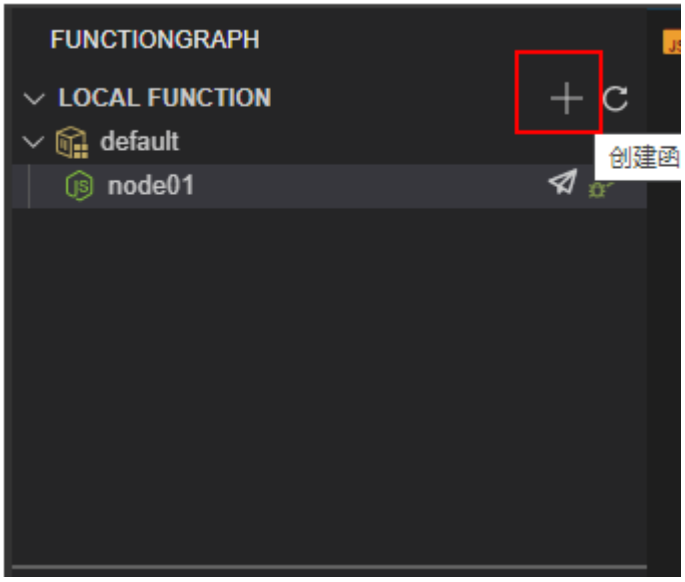
步骤12 返回FunctionGraph控制台，查看函数，确认已合入修改的内容。

----结束

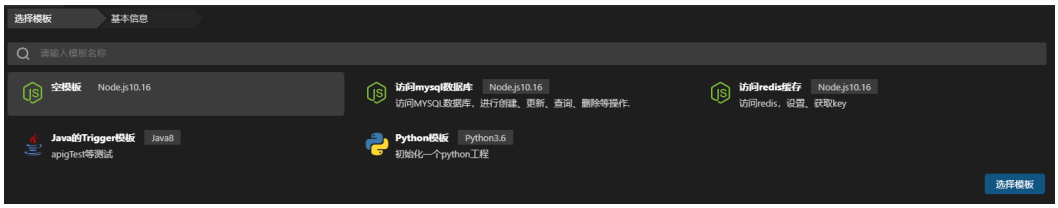
CodeArts IDE Online 本地创建函数

用户在CodeArts IDE Online本地创建函数并完成编辑，再将函数推送到FunctionGraph控制台。以下示例仅供参考，具体请以实际创建函数为准。

步骤1 在CodeArts IDE Online编辑框的LOCAL FUNCTION打开创建函数。

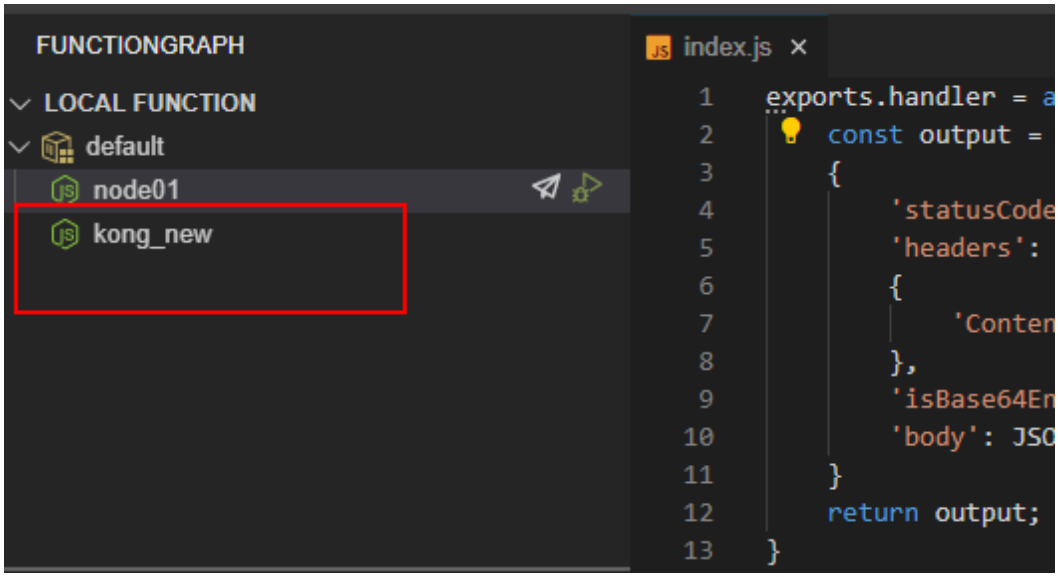


步骤2 选择模板，创建函数。

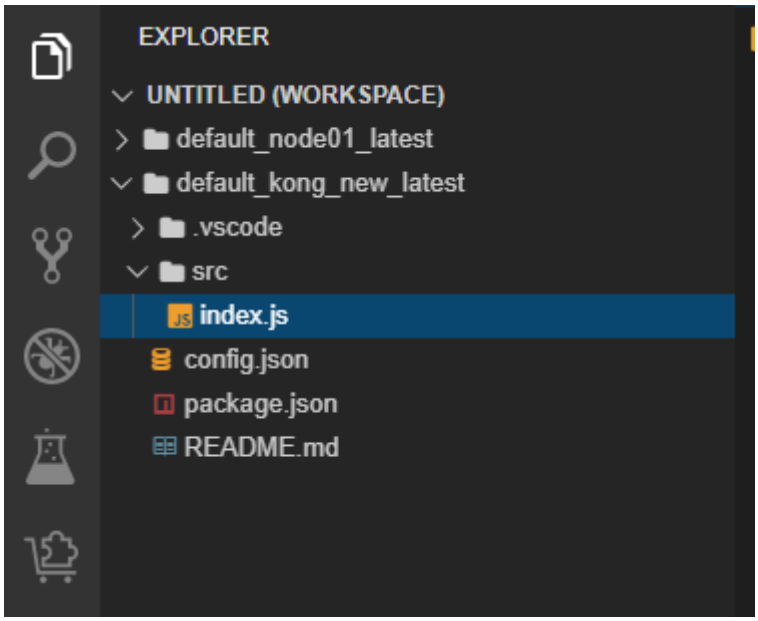


步骤3 例如选择空模板创建，输入函数名称，比如“kong_new”，单击“创建函数”。

步骤4 创建成功后，左侧编辑框即可看到刚创建的函数。



步骤5 在EXPLORER可以看到完整的kong_new函数，其中函数代码只是 “index.js”，其余的都是配置文件，可以不关心。

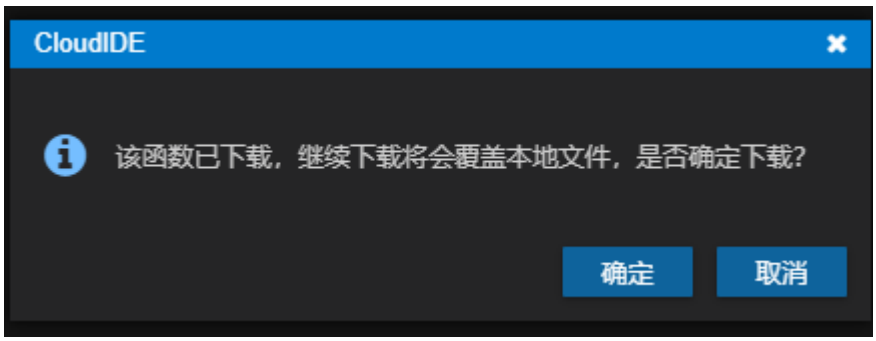


----结束

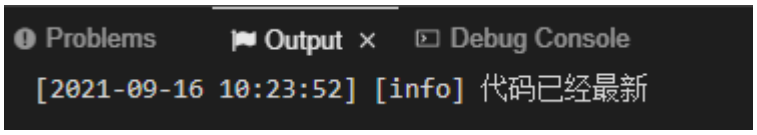
函数下载、推送代码比较流程

鉴于本地代码与远程代码可能存在不同，会存在新代码覆盖老代码的情况，所以当下载、推送时候，都会有弹框提示。

步骤1 函数 node01 已经存在LOCAL FUNCTION，此时再下载，可能会覆盖本地函数，下载前有提示。

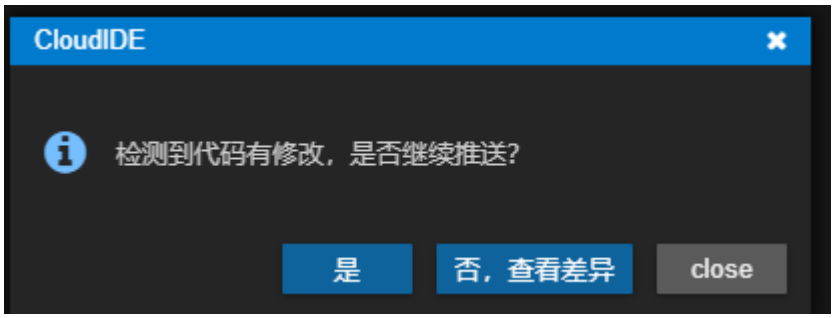


步骤2 函数 node01 已经存在在REMOTE FUNCTION，不做修改直接推送，远程和本地的代码会比较，提示“代码已经最新”，不必推送。

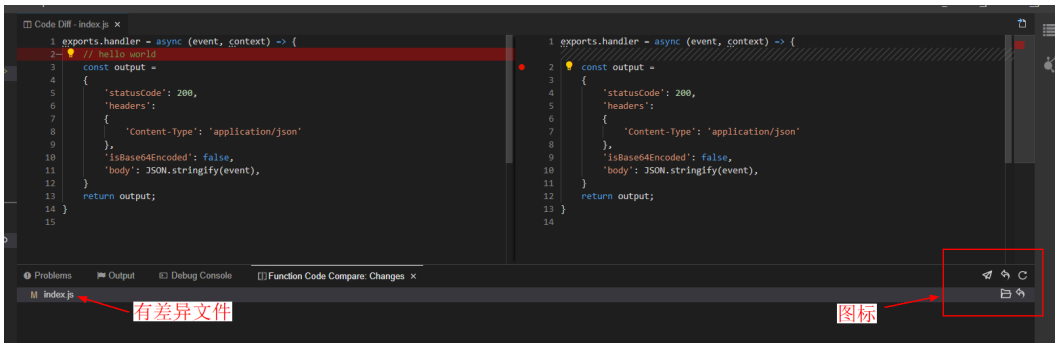


步骤3 函数 node01 做出修改，删除第二行的“// hello world”，推送提示如下。

- 是：直接推送，本地代码将会覆盖远程代码。



- 否，查看差异：弹出有差异的文件，单击“index.js”打开比较差异的页面，可以看出最新文件少了第二行。右侧图标依次为“继续推送”、“取消推送”、“刷新”、“打开文件”、“放弃修改”，鼠标指针放上均有提示。



----结束

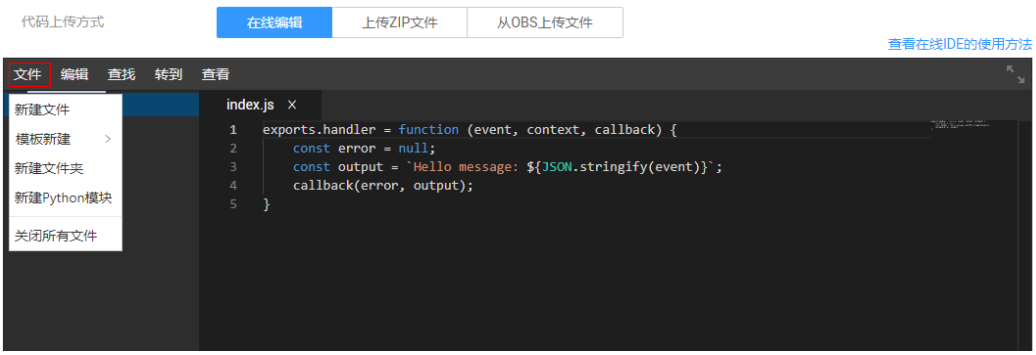
9.1.2 附录：CodeArts IDE Online 使用方法

用户在编辑函数代码时支持类似工程方式的管理，可以创建文件、文件夹并对其进行编辑。使用函数工作流控制台中的在线代码编辑器，可以在线编写函数代码，如果代码是上传zip包的方式，则前端进行相应解压展示，并支持在线编辑修改。同时，在线代码编辑器支持在线测试和保存，可以查看函数执行的返回、执行摘要和日志，该功能需要在编辑器全屏模式下使用。

目录管理

在编辑器菜单栏中选择“文件”，可以管理文件夹目录，如图9-2所示。

图 9-2 文件

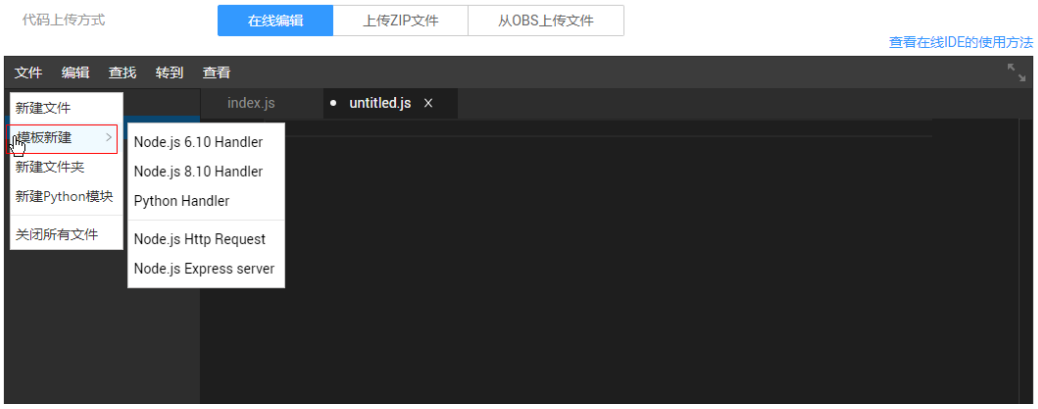


在展开的下拉菜单中选择“新建文件”，可以新建文件，并命名，如图9-3所示。

图 9-3 新建文件

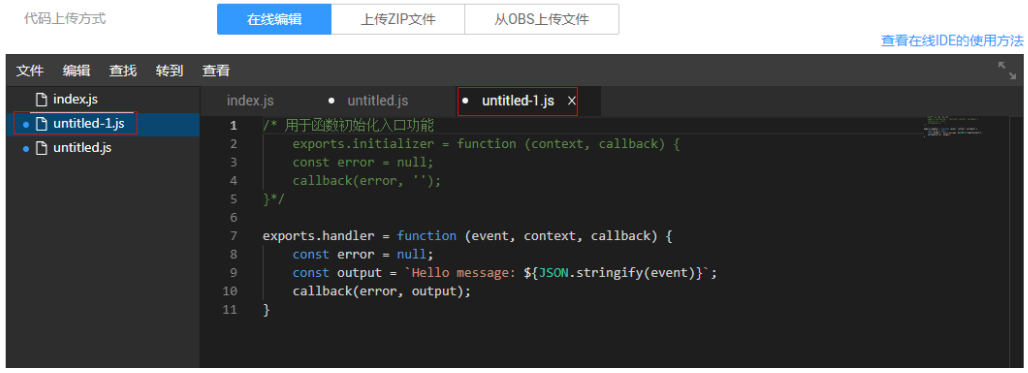
在展开的下拉菜单中选择“新建模板”，可以选择模板在线创建函数，如图9-4所示。

图 9-4 新建模板



选择需要的模板，创建在线函数，并命名，如图9-5所示。

图 9-5 创建文件



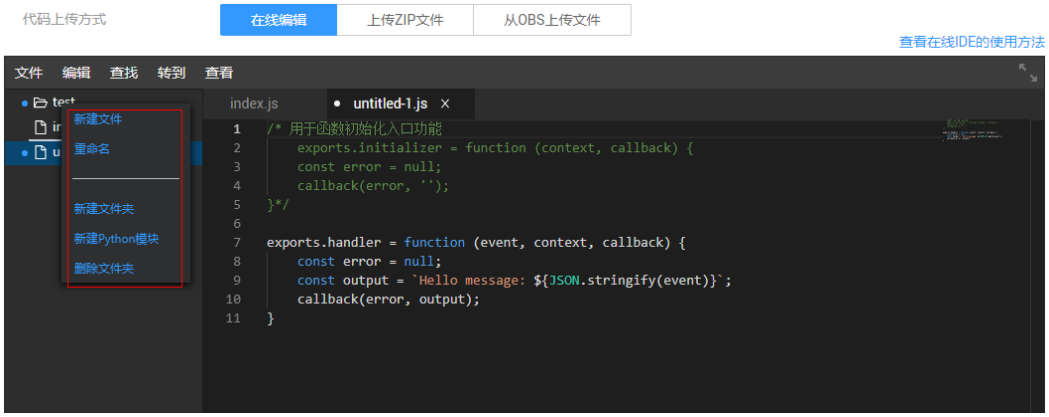
单击右键，选择“重命名”可以对文件和文件夹进行重命名，选择“删除”可以删除，如图9-6所示。

图 9-6 文件重命名

在左侧“文件”下拉菜单中选择“新建文件夹”，新建文件夹并命名，如图9-2中所示。

单击右键，选择“新建文件”可以在该文件夹目录下新建文件，在该菜单栏中，可以对文件夹重命名、新建Python模块、删除文件夹，如图9-7所示。

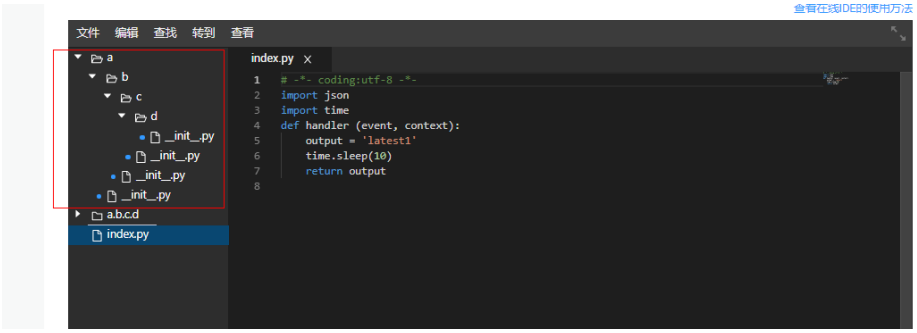
图 9-7 编辑文件夹



在左侧“文件”下拉菜单中选择“关闭所有文件”，将所有打开的文件关闭，如图9-2中所示。

支持快速创建Python模块，支持多创建层级，如图9-8所示。

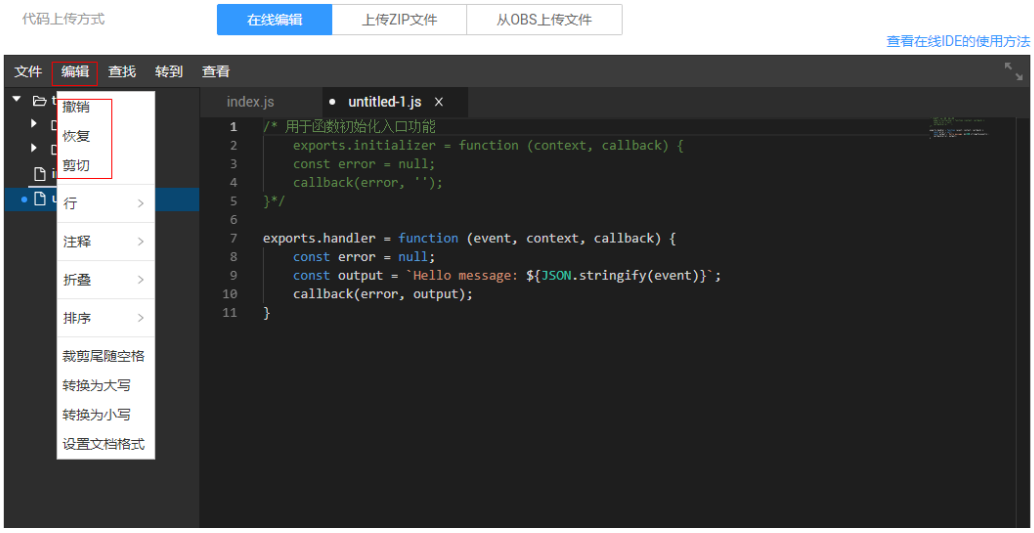
图 9-8 多层级模块



代码在线编辑

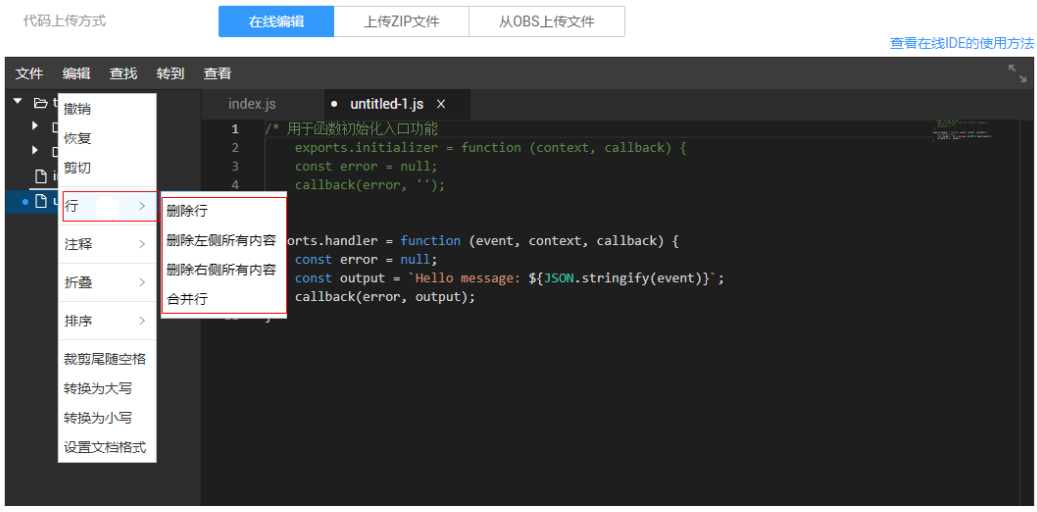
在编辑器菜单栏中选择“编辑”，可以在线编辑代码。在下拉菜单中可以撤销操作或恢复上一步操作，也可以剪切内容等，如图9-9所示。

图 9-9 在线编辑



在下拉菜单中选择“行”，可以对代码以“行”为单位进行编辑，如图9-10所示。

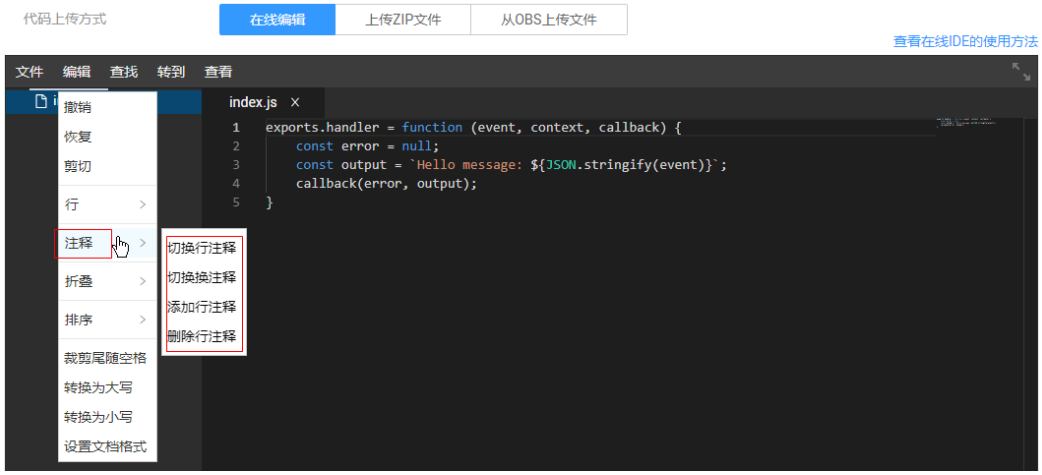
图 9-10 编辑行



在“行”命令的展开菜单中，选择“删除行”，可以删除整行代码。选择“删除左侧所有内容”，可以将鼠标光标所在位置左侧的代码删除，同样选择“删除右侧所有内容”，将右侧代码删除。选择“合并行”，将光标所在代码行的下一行代码合入到该行。

在“编辑”下拉菜单中选择“注释”，可以编辑注释，如图9-11所示。

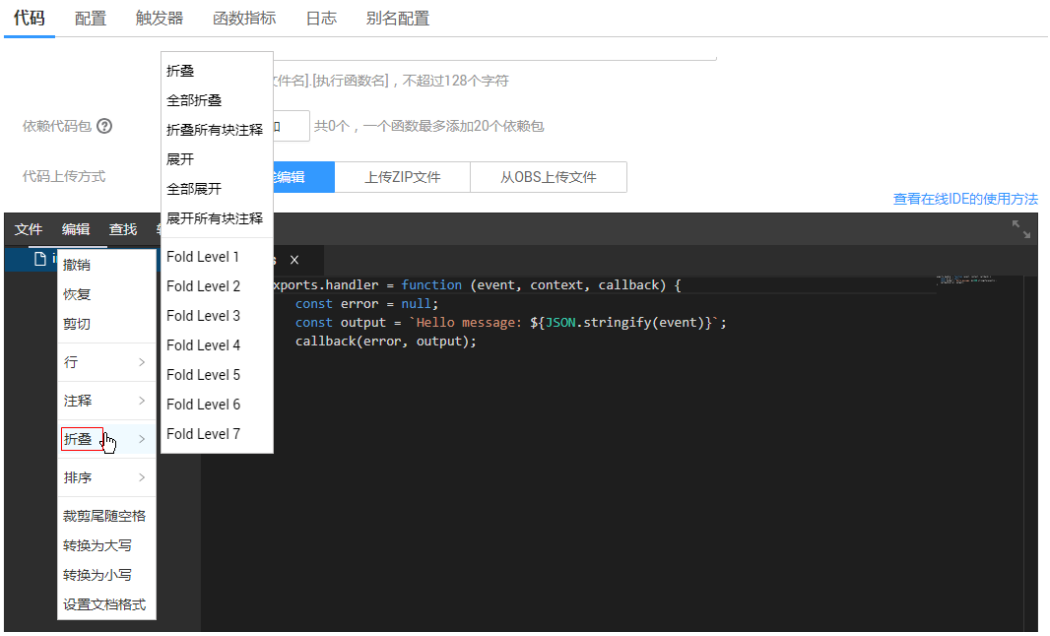
图 9-11 编辑注释



在“注释”命令的展开菜单中，选择“切换行注释”打开某一行代码的注释，选择“切换块注释”打开某一块代码的注释，选择“添加行注释”增加一行注释，选择“删除行注释”删除一行注释。

在“编辑”下拉菜单中选择“折叠”，可以展开或折叠代码，如图9-12所示。

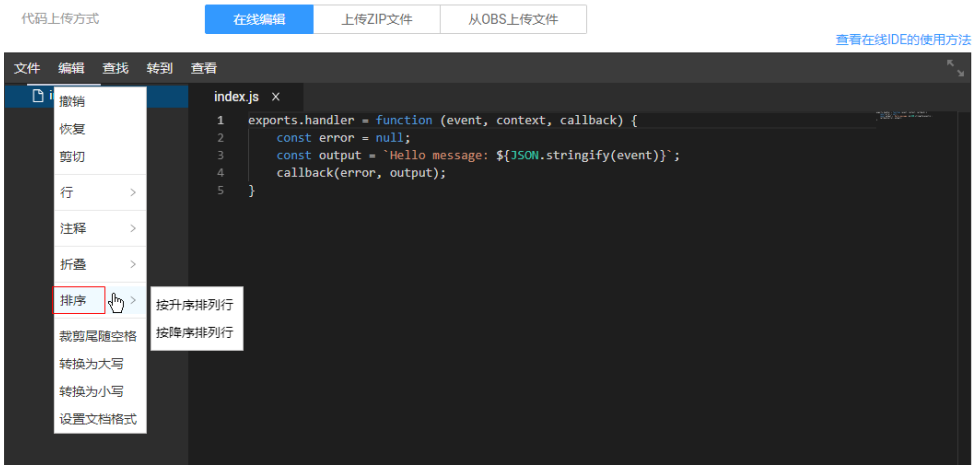
图 9-12 展开或折叠代码



代码可以全部展开或折叠，按层级折叠请选择“Fold Level”。

选择“排序”对代码行进行排序，如图9-13所示。

图 9-13 排序

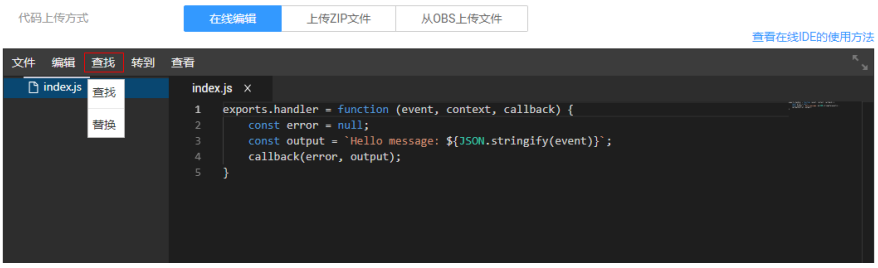


“编辑”菜单栏中同时可以裁剪尾随空格、转换大小写、设置文档格式等。

查找及替换

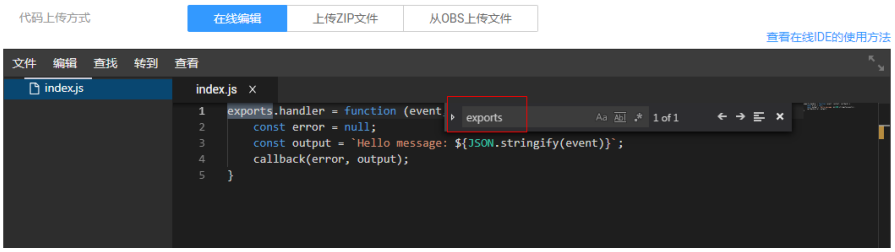
在编辑器菜单栏中选择“查找”，可以进行查找或替换。如图9-14所示。

图 9-14 查找



在展开的下拉菜单中选择“查找”或“替换”，输入内容，进行查找或替换代码，如图9-15所示。

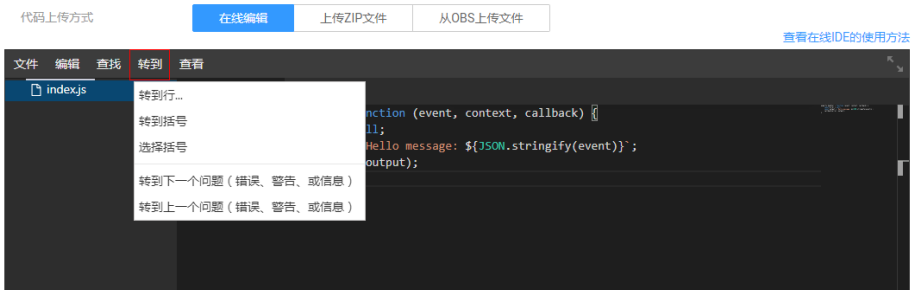
图 9-15 查找代码



跳转

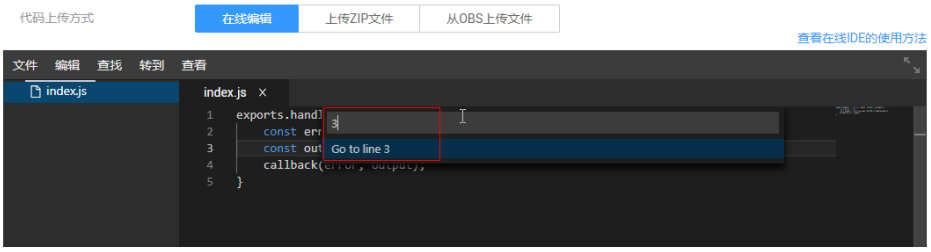
在编辑菜单栏中选择“转到”，可以跳转到相应的代码位置，如图9-16所示。

图 9-16 转到



在展开的下拉菜单中选择“转到行”，可以跳转到对应的代码行，如图9-17所示。

图 9-17 跳转

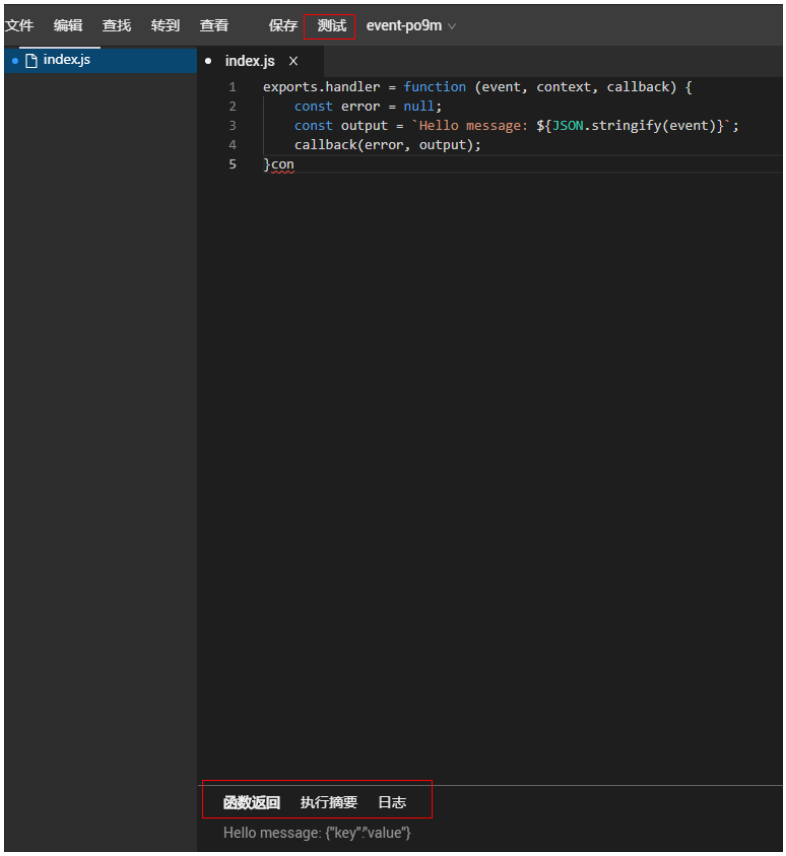


当代码比较多时，选择一个反括号“{”，单击“转到括号”可跳转到相应的括号“}”位置，当代码中有错误时，单击“转到下一个问题（错误、告警、或信息）”跳转到下一个错误的地方。

函数在线测试

在编辑器全屏模式下，函数配置测试事件，单击“测试”测试函数，显示函数返回、执行摘要和日志，如图9-18所示。

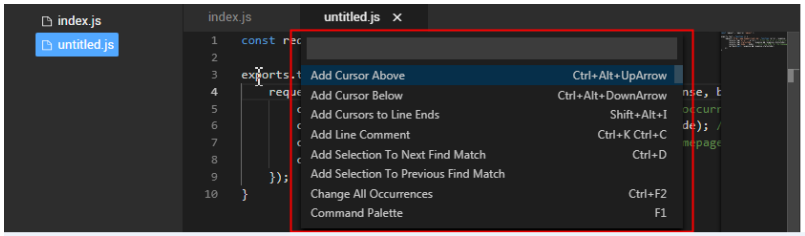
图 9-18 函数在线测试



通用功能

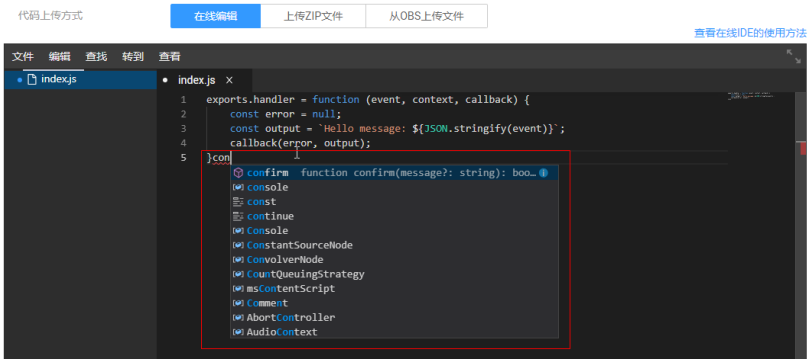
1. 在编辑菜单栏中选择“查看”，在下拉的菜单栏中选择“主题”可以更换编辑器主题，选择“显示命令面板”可以查看所有命令，如图9-19所示。

   图 9-19 查看所有命令

2. 编辑器具有代码智能提示功能，如图9-20所示。

图 9-20 智能提示



3. 支持左侧工作区域调整宽度，文件可以拖拽移动，同时本地文件可以拖拽上传。

9.2 VSCode 本地调试

概述

Huawei Cloud FunctionGraph是华为云Serverless产品的VSCode插件。通过该插件，您可以：

- 快速地在本地创建函数
- 运行调试本地函数、部署本地函数至云端
- 拉取云端的函数列表、调用云端函数、上传ZIP包至云端

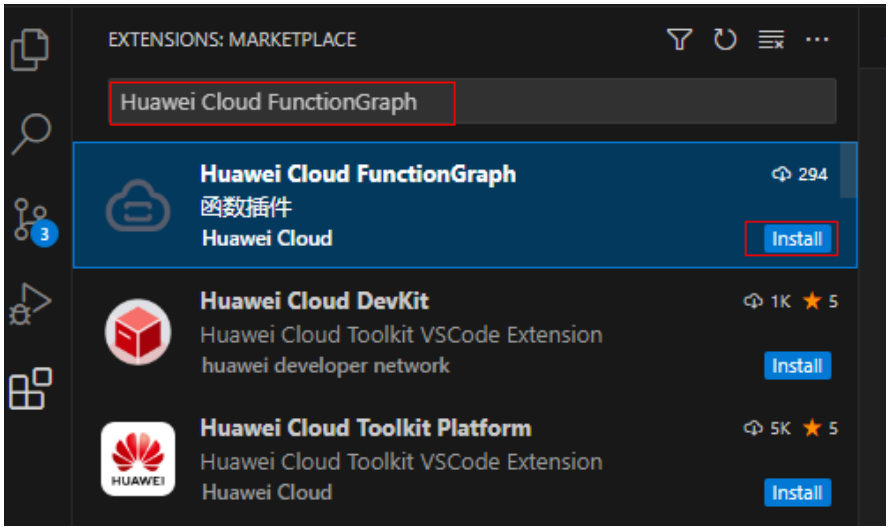
前提条件

下载 **Visual Studio Code**（1.63.0版本以上）并安装。

安装插件

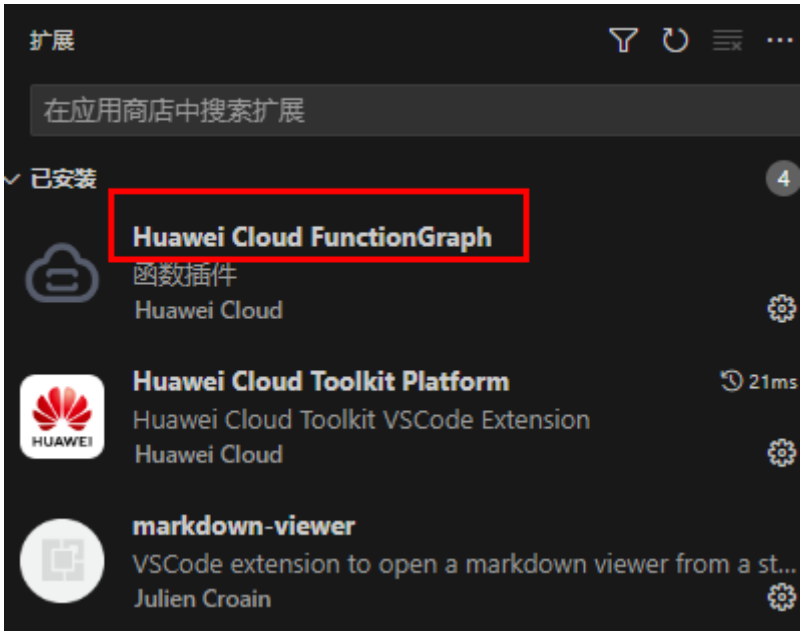
1. 打开Visual Studio Code工具，在应用商店中搜索“Huawei Cloud FunctionGraph”并进行安装。

图 9-21 搜索并安装



2. 安装成功后，Huawei Cloud FunctionGraph插件展示在已安装列表中。

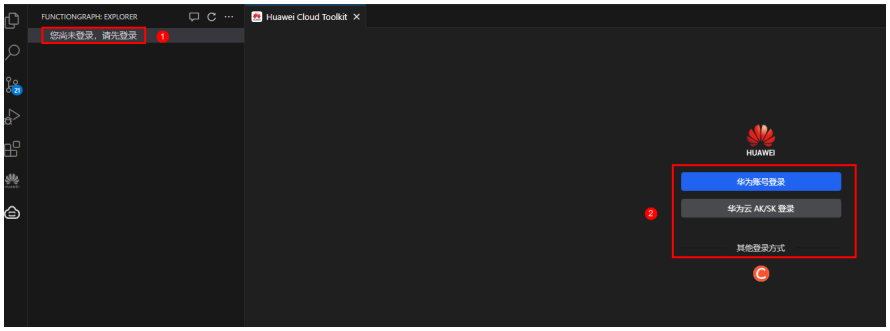
图 9-22 已安装列表展示



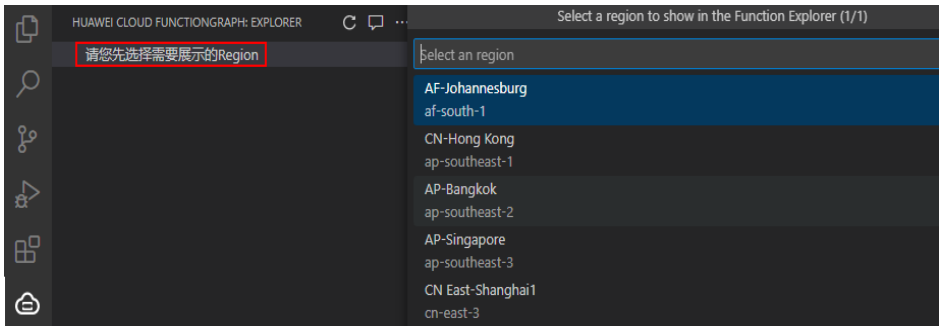
登录函数插件

1. 单击Huawei Cloud FunctionGraph插件图标，单击“您尚未登录，请先登录”，弹出登录界面，根据页面提示选择登录方式。若选择“华为云 AK/SK登录”，需先获取账号的AK/SK，请参见**新增访问密钥**。

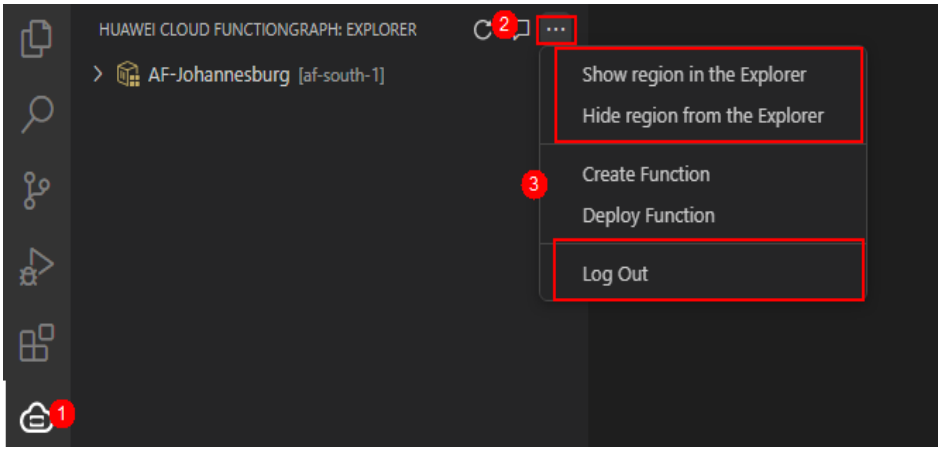
图 9-23 使用 AK/SK 登录



2. 您可以选择需要展示的区域Region，查看不同区域的函数信息。

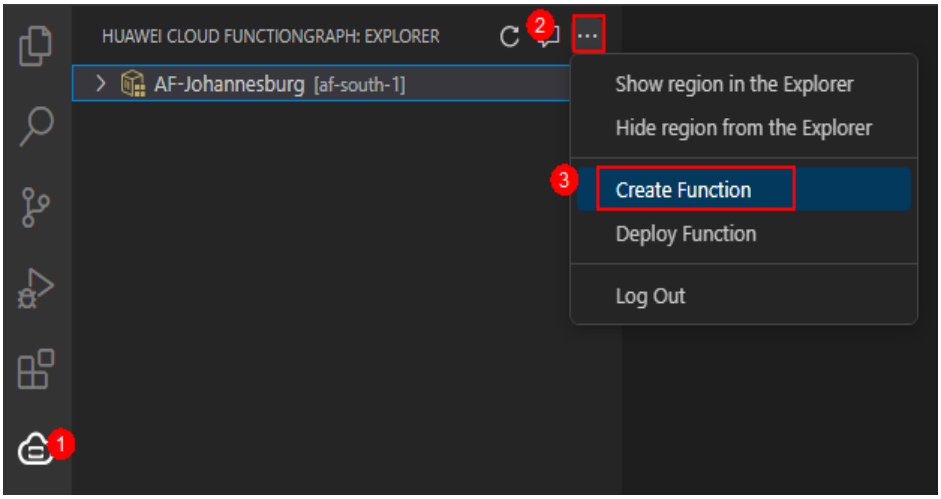


3. 您也可以参考下图，进行更多区域的展示和隐藏、以及账号退出操作。
   - Show region in the Explorer：选择需要操作的Region。
   - Hide region from the Explorer：隐藏暂时不关注的Region。
   - Log Out：退出登录。



创建函数

1. 在插件面板中选择“Create Function”，或“Ctrl+Shift+p”搜索“Create Function”命令，按照提示依次选择或输入“运行时”、“模板”、“函数名称”、“本地文件夹”。



2. 本地函数创建成功后，会自动打开入口文件。
3. 自动生成配置文件，可以通过修改文件配置函数信息，参数如下：

```
HcCrmTemplateVersion: v2
Resources:
    Type: HC::Serverless::Function
    Properties:
        Name: functionName //函数名称
        Handler: handler //函数执行入口
        Runtime: runtime //函数运行时
        CodeType: inline //默认固定
        CodeFileName: index.zip //默认固定
        CodeUrl: ""
        Description: '' //函数运行时
        MemorySize: 128 //函数执行内存
        Timeout: 30 //函数超时时间(s)
        Version: latest //默认固定
```



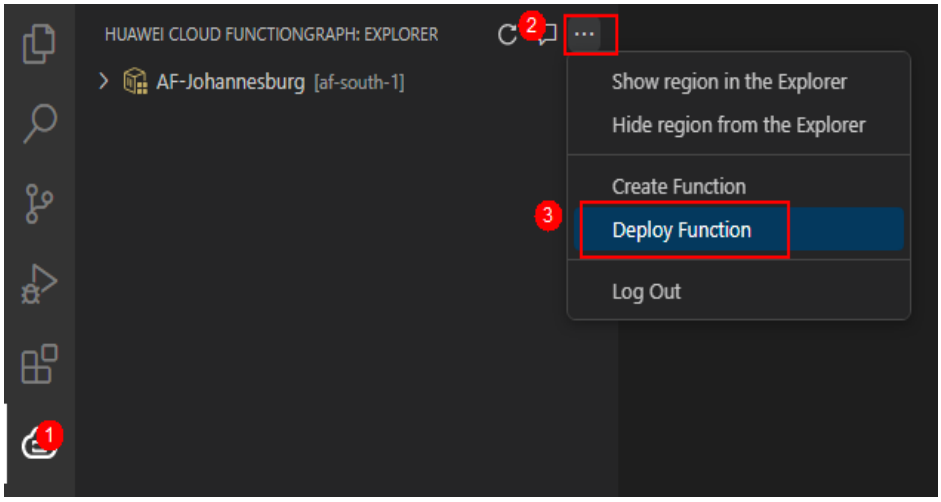
```
Environment:
  Variables: {} // 环境变量
InitializerHandler: "" //函数初始化入口
InitializerTimeout: 0 //函数初始化超时时间
EnterpriseProjectId: "0" //企业项目
FuncType: v2
URN: "" //函数URN，函数下载后生成
```

部署函数

- **前提**

  确保函数代码路径正确。Nodejs、Python和PHP运行时函数代码在src目录下，其余运行时函数代码在根目录下。

在插件面板中选择“Deploy Function”，或“Ctrl+Shift+p”搜索“Deploy Function”命令，按照提示依次选择“需要部署的函数”、“Region”。



- 部署成功：界面右下角弹出成功提示，切换至部署“Region”查看。
- 部署失败：在“OUTPUT”下查看错误日志并解决。

本地调试

1. Nodejs
   - **前提**

     本地环境已安装Nodejs。
   - **默认模式**

     单击handler方法的Local Debug，配置事件内容，单击 Invoke，进行调试。

图 9-24 单击 Local Debug

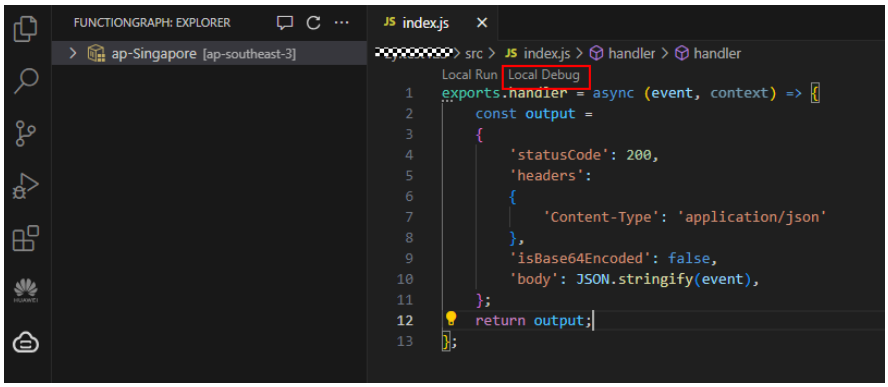
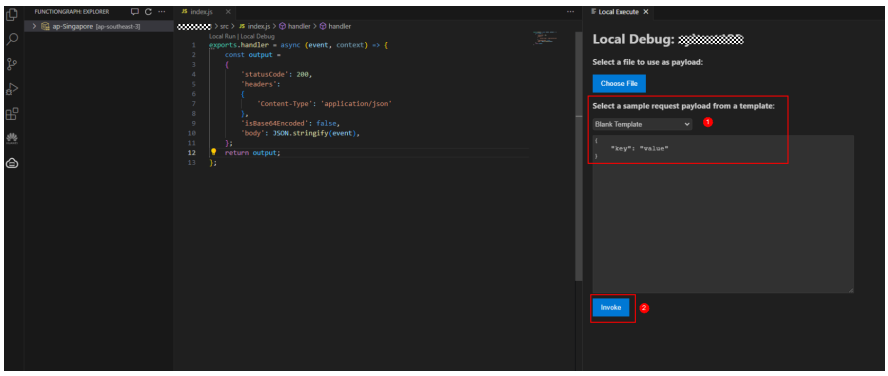


图 9-25 配置事件内容



- **VSCode自带调试能力**

  在函数文件夹下新建main.js文件，并将下面内容复制到main.js文件，单击左侧的运行和调试图标，选择添加配置，进行配置，选择Nodejs，按“F5”进行调试。

```
const handler = require('./index'); // 函数入口文件路径，根据具体情况修改
const event = { 'hello': 'world' }; // 测试事件内容，根据具体情况修改
const context = {}; // Context类
console.log(handler.handler(event, context));
```

2. Python

   - **前提**

     本地环境已安装Python。

   在函数文件夹下新建main.py文件，并将下面内容复制到main.py文件，单击左侧的运行和调试图标，选择添加配置，进行配置，选择Python，按“F5”进行调试。

```
import sys
import index # 函数入口文件路径，根据具体情况修改

# main方法用于调试，event是选择的调试事件
if __name__ == '__main__':
    event = { 'hello': 'world' } # 测试事件内容，根据具体情况修改
    context = ''
    content = index.handler(event, context)
    print('函数返回：')
    print(content)
```


3. Java
   - **前提**

     已安装Java，VSCode已支持java的运行测试。

   在函数文件夹下的test目录下，打开TriggerTestsTest.java文件，单击左侧的运行和调试图标  ，选择添加配置，进行配置，选择Java，按“F5”进行调试。

其余功能

- **跳转到界面打开**

  选择您需要打开的函数，鼠标右键单击“Open in Portal”，会在浏览器中打开该函数的详情页面。

- **执行云端函数**

  a. 选择您需要操作的函数，鼠标右键单击“Invoke Function...”。

  b. 在Invoke Function面板中，选择需要传入的事件，单击“Invoke”，函数的日志以及结果会输出在OUTPUT中。

- **下载云端函数**
  - **前提**

    用户已添加获取桶对象(obs:object:GetObject)的权限。

  选择您需要操作的函数，鼠标右键单击“Download...”，选择您要下载的路径，函数代码会从云端下载到本地并自动打开入口文件。

- **更新云端函数**

  选择您需要操作的函数，鼠标右键单击“Upload Function...”，选择您想要上传的ZIP包。

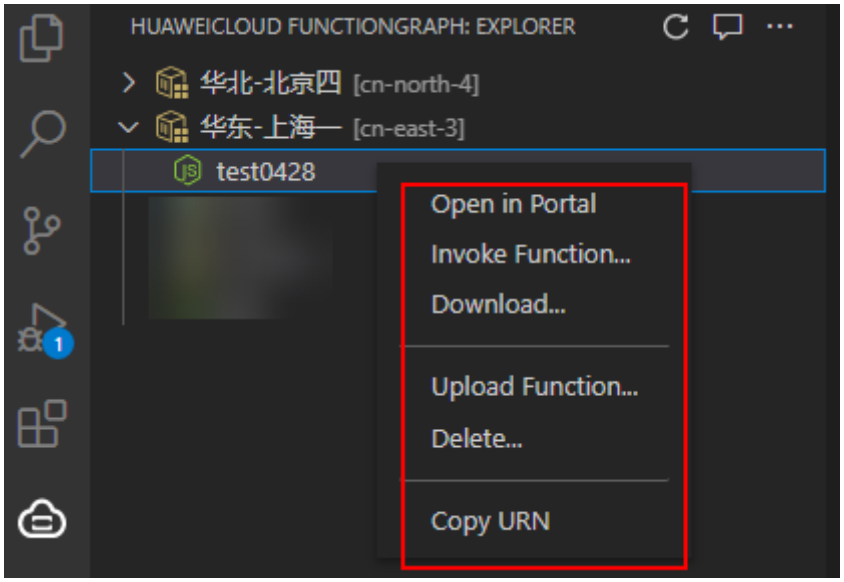
- **删除云端函数**

  a. 选择您需要删除的函数，鼠标右键单击“Delete...”。

  b. 在确认框中选择"Delete"，删除函数。

- **复制URN**

  选择您需要复制URN的函数，鼠标右键单击“Copy URN”直接获取。



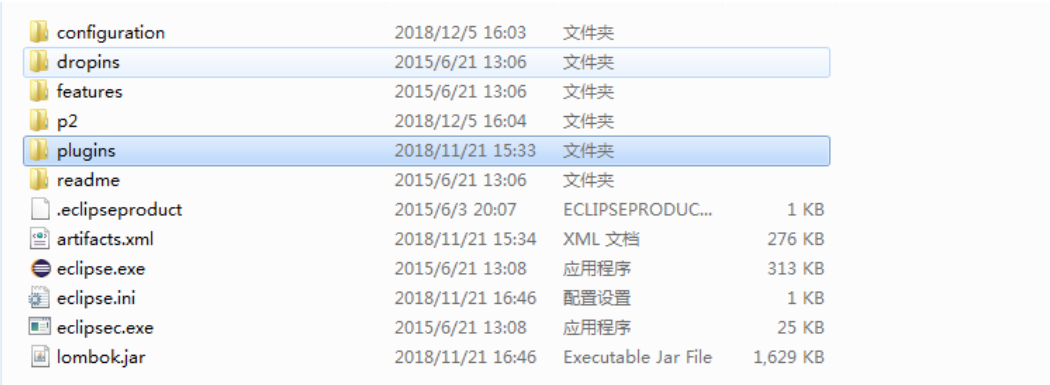
9.3 Eclipse-plugin

当前java没有对应的模板功能，且只支持传包到OBS上，不支持在线编辑，所以需要一个插件，能够支持在java的主流开发工具（Eclipse）上，实现一键创建java模板、java打包、上传到OBS和部署。

步骤1 获取**Eclipse 插件**（软件包校验文件：**Eclipse插件.sha256**）。

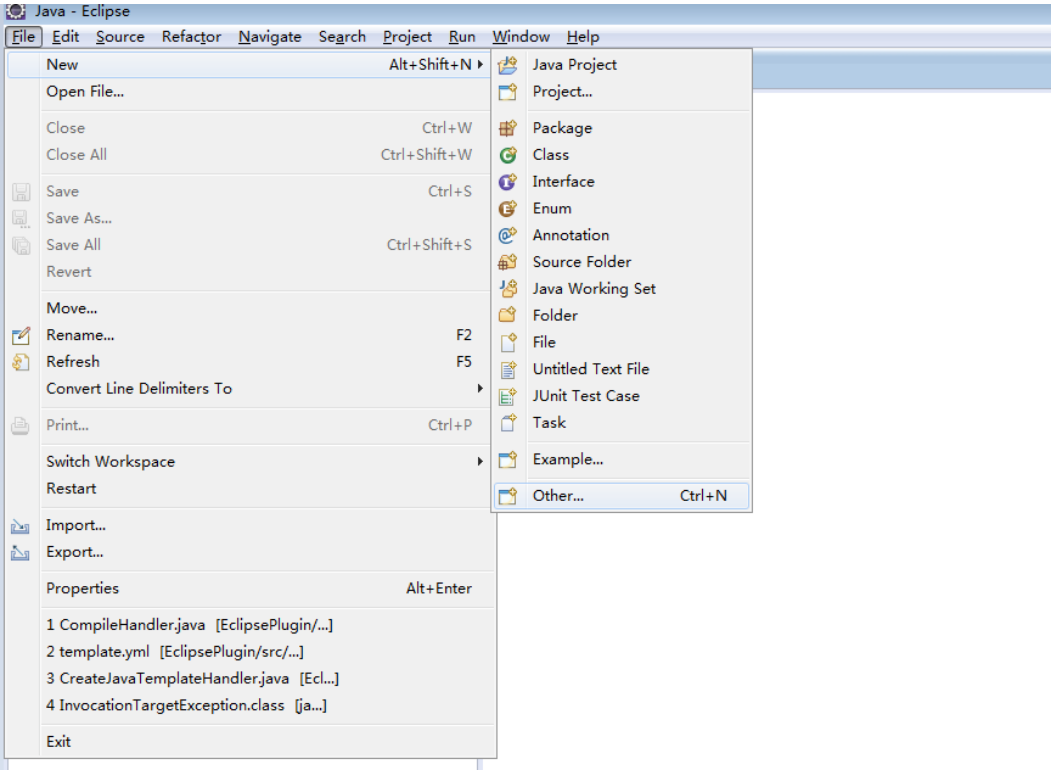
步骤2 将获取的Eclipse插件jar/zip包，放入Eclipse安装目录下的plugins文件夹中，重启Eclipse，即可开始使用Eclipse插件。如**图9-26**所示。

图 9-26 安装插件



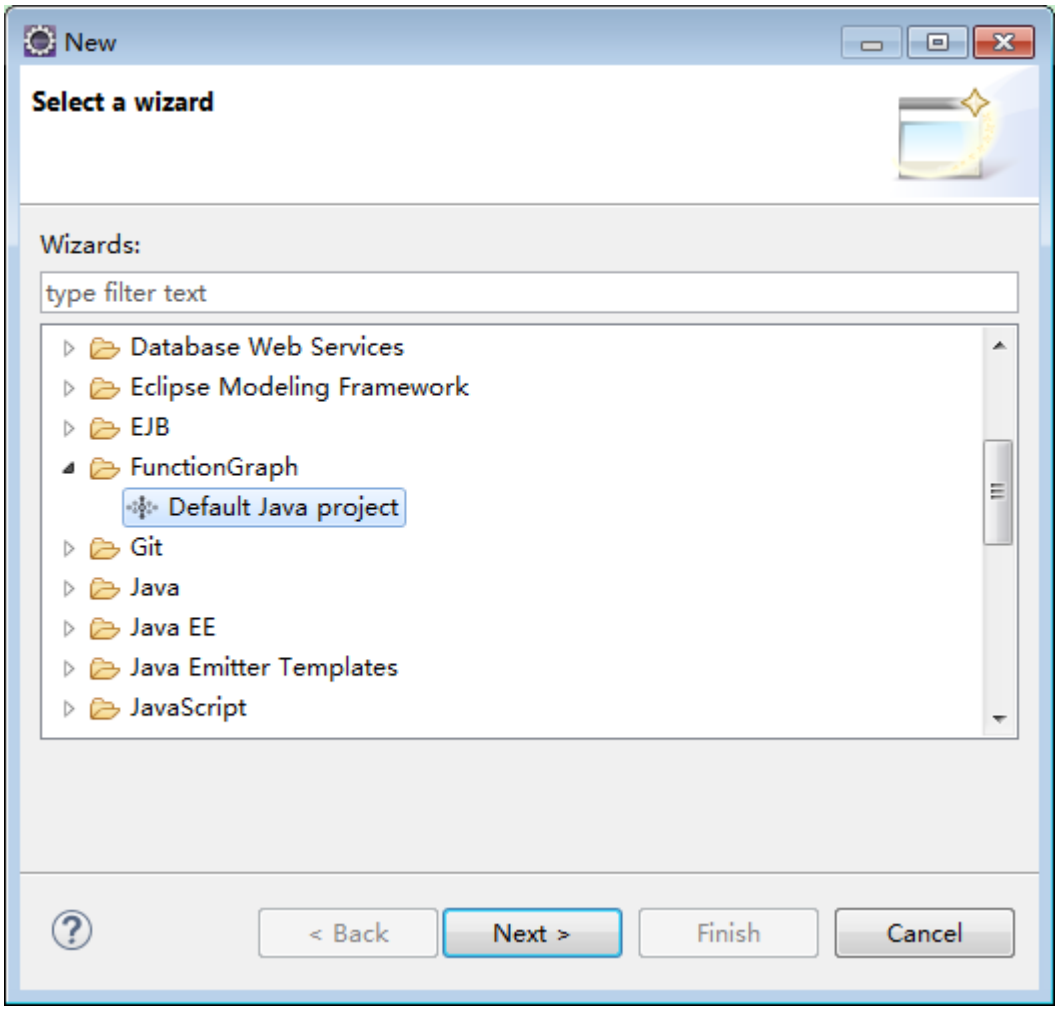
步骤3 打开Eclipse，单击“File”，选择“New > Other”，如**图9-27**所示。

图 9-27 新建模板



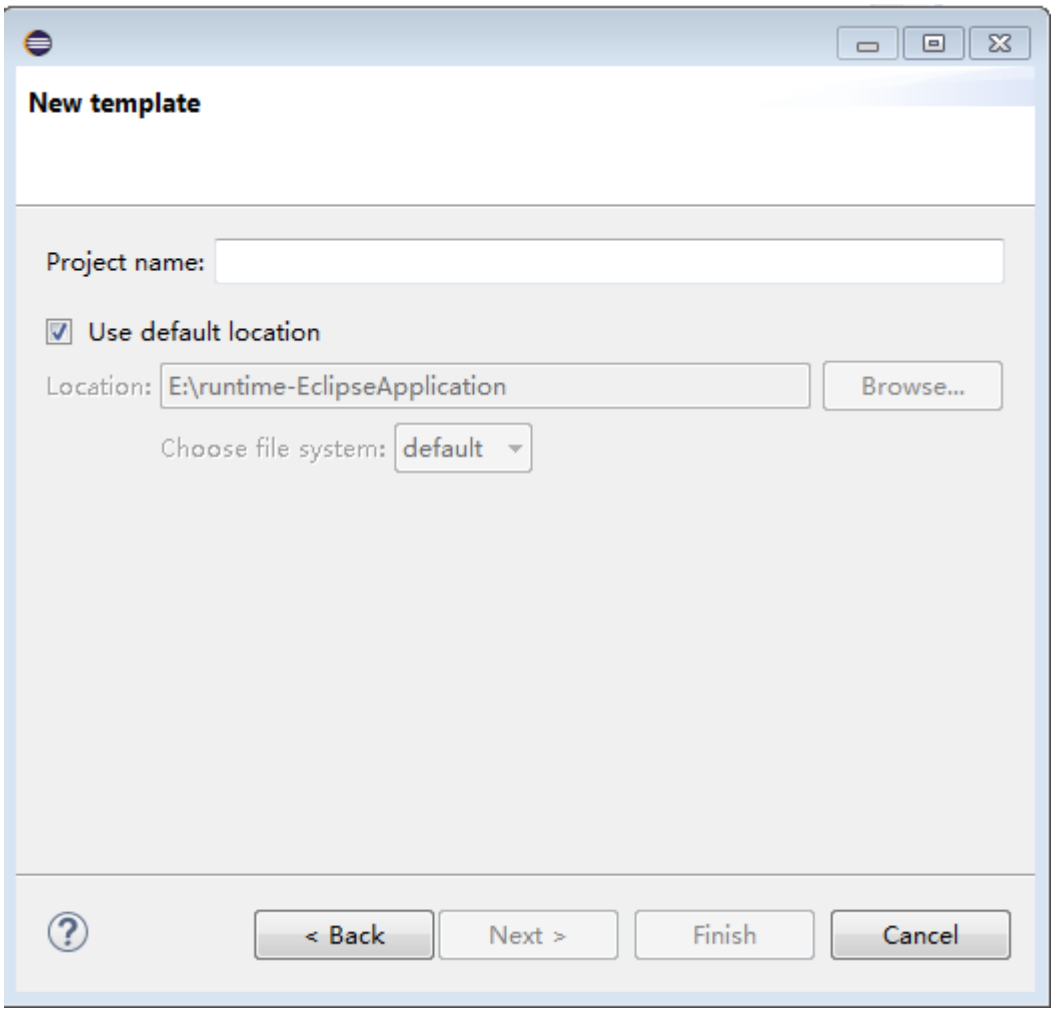
步骤4 选择 “FunctionGraph”文件下的“Default Java project”节点。如图9-28所示。

图 9-28 选择默认 Java 模板



步骤5 输入工程名称，选择工程目录（也可以使用默认目录），单击“Finish”完成模板创建。如图9-29所示。

图 9-29 完成创建



----结束

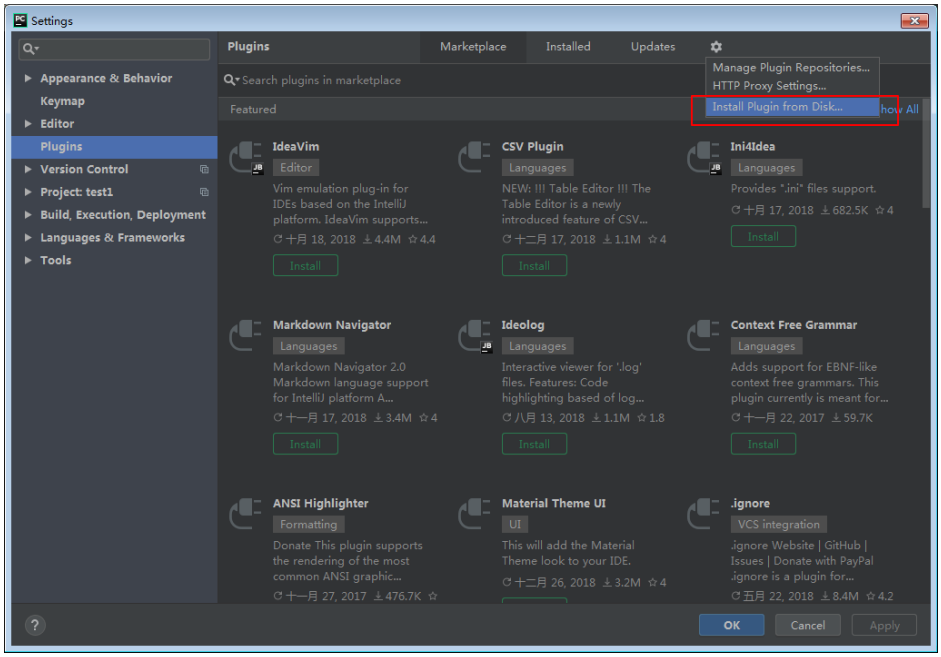
9.4 PyCharm-Plugin

在Python主流开发工具（PyCharm）上实现一键生成python模板工程、打包、部署等功能。

步骤1 获取**插件**（**插件.sha256**）。

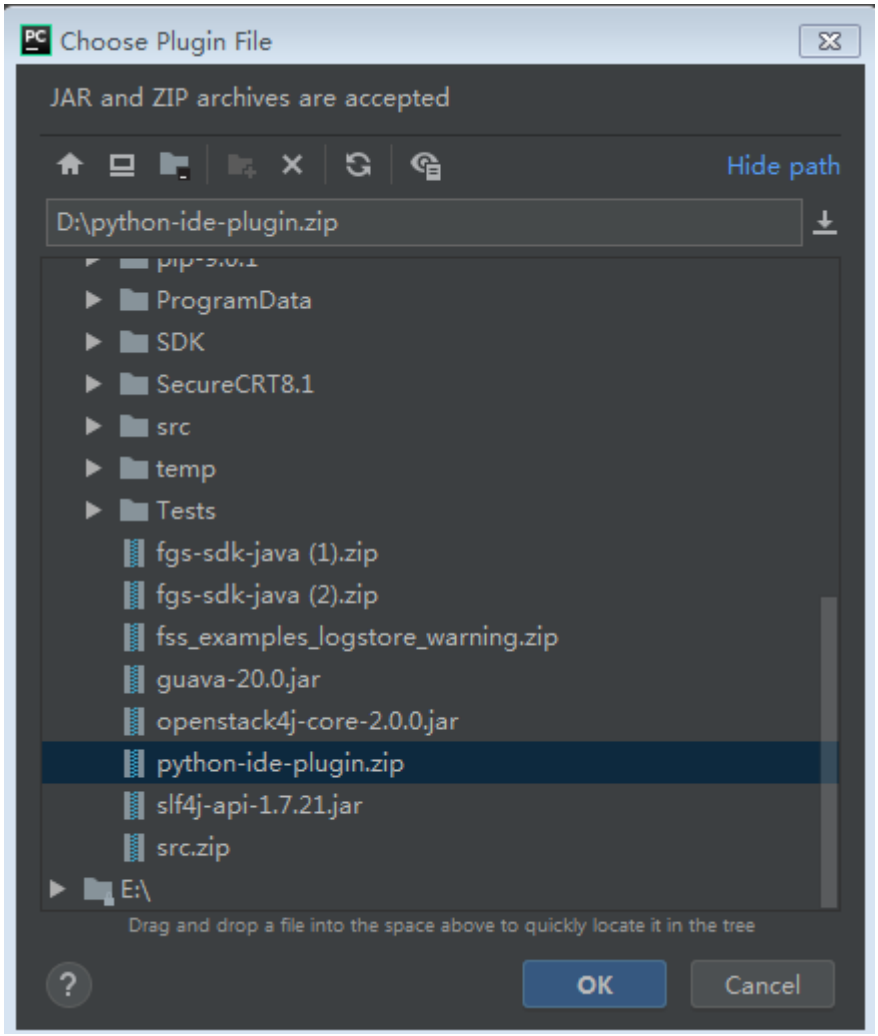
步骤2 打开JetBrains PyCharm，单击“File”菜单，选择“Settings ”，在弹出界面的菜单中选择“Plugins”页面，单击右上角设置按钮中的“Install plugin from disk... ”，如**图9-30**所示。

图 9-30 安装 Plugins



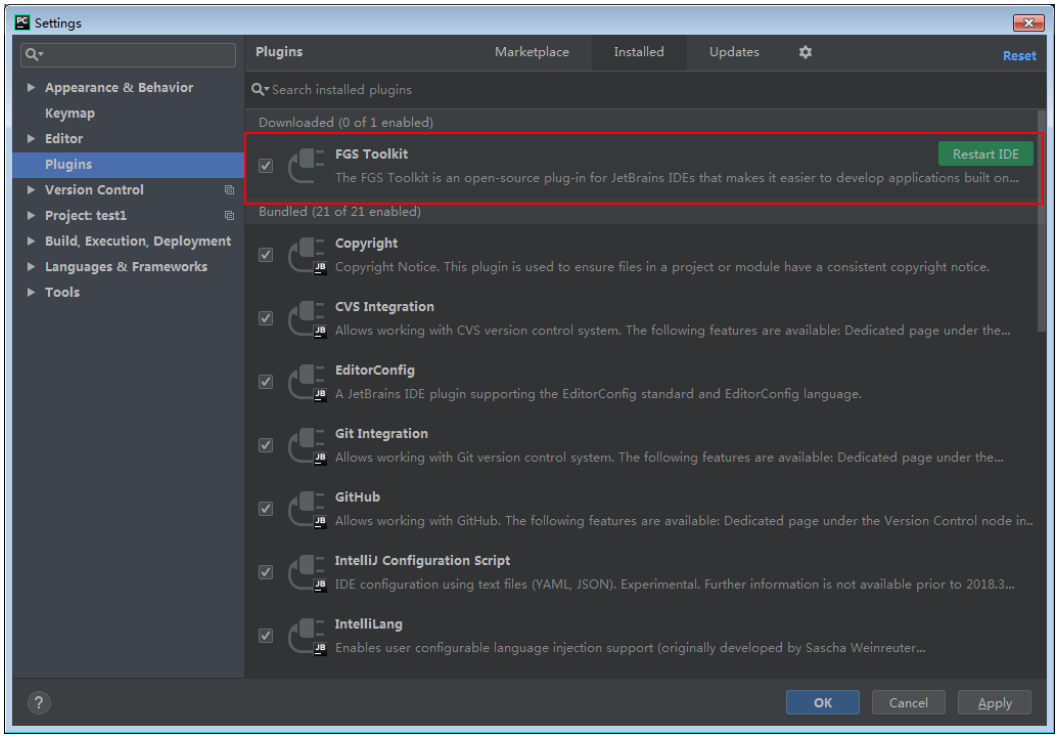
步骤3 在弹出的界面中，选择插件包，单击“OK”，如图9-31所示。

图 9-31 选择插件包



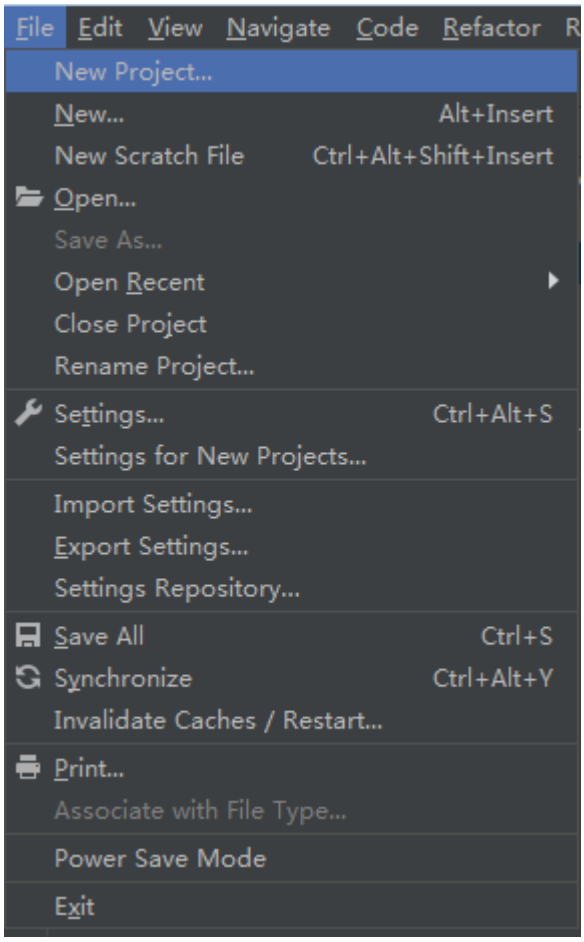
步骤4 在插件列表中，勾选插件名称，单击“Restart IDE”，如图9-32所示。

图 9-32 重启 IDE



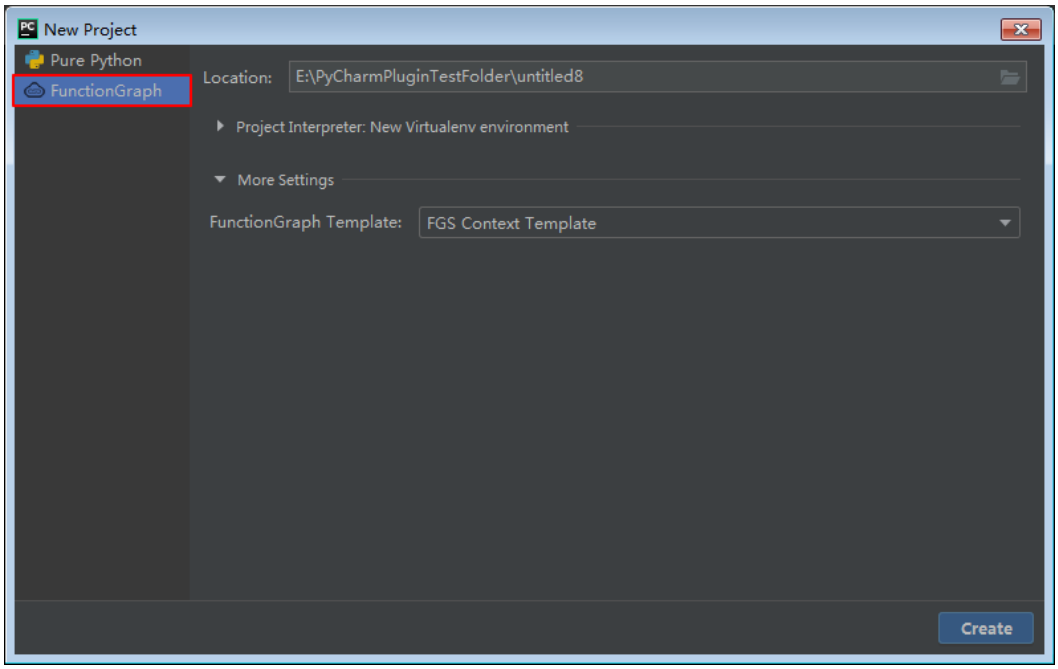
步骤5 单击“File”菜单，选择“New Project”，如图9-33所示。

图 9-33 新建工程



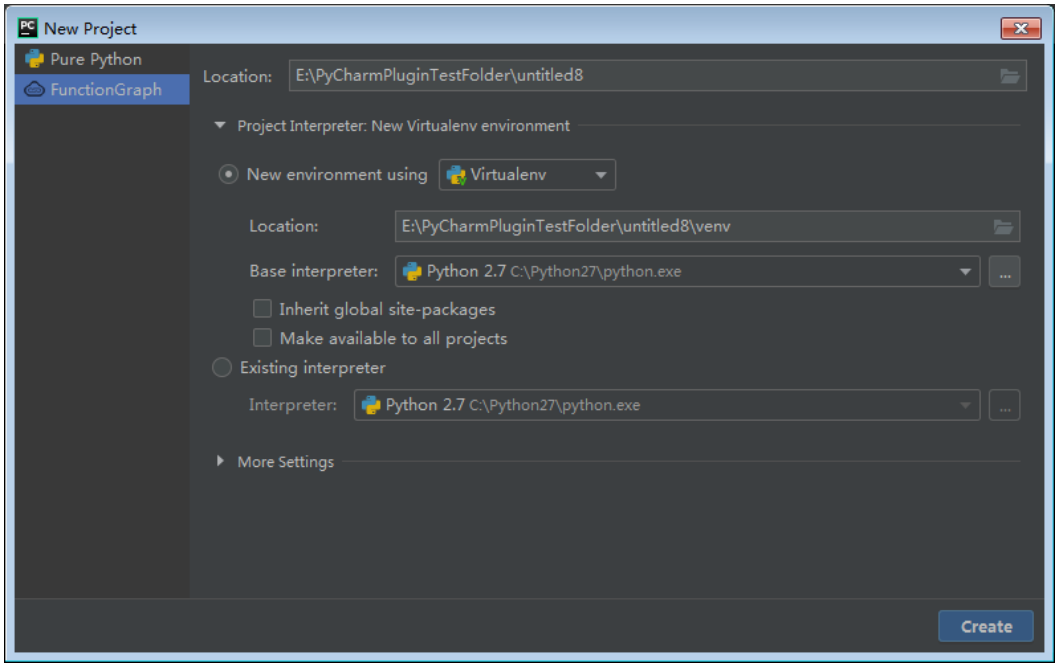
步骤6 在弹出的新建工程页面中，选择“FunctionGraph”，如图9-34所示。

图 9-34 FunctionGraph



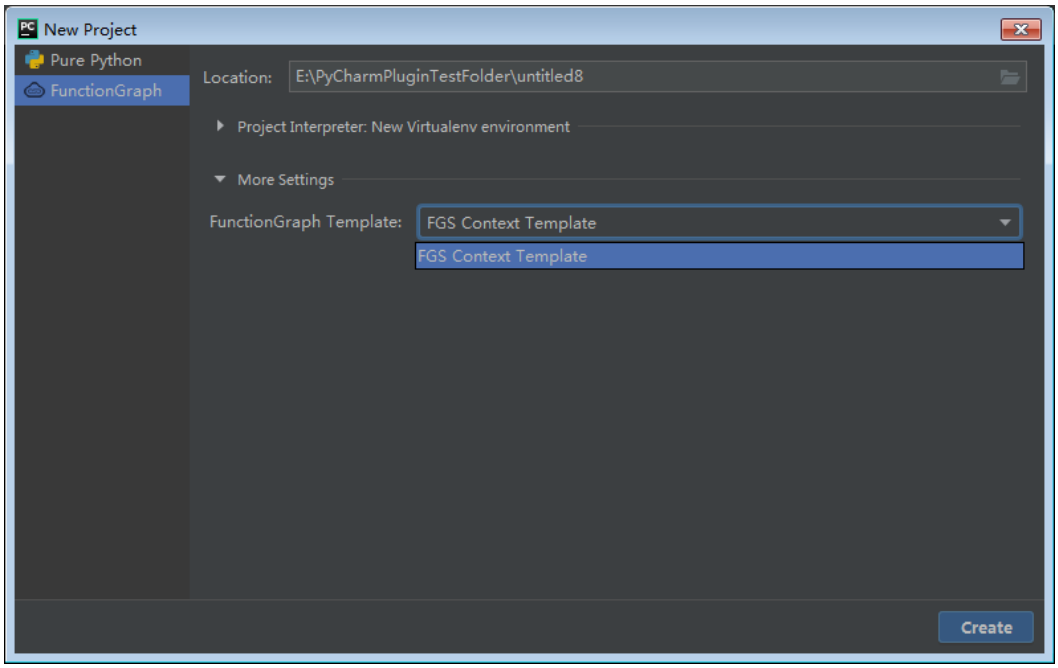
步骤7 在“Location”栏中选择工程的路径，在“Project Interpreter: New Virtualenv environment”中选择使用python的版本。如图9-35所示。

图 9-35 选择版本



步骤8 在“More Settings”中选择要创建的模板，如图9-36所示。

图 9-36 选择模板



说明

目前仅支持python 2.7的Context模板。

步骤9 单击“Create”，完成创建。

----结束

9.5 Serverless Devs

9.5.1 概览

组件说明

华为云函数工作流（FunctionGraph）组件是一个用于支持华为云函数应用生命周期的工具，基于**Serverless Devs**进行开发，通过配置资源配置文件s.yaml，您可以简单快速地部署应用到**华为云函数工作流平台**。

前提条件

本地已安装nodejs。

快速开始

步骤1 安装Serverless Devs 开发者工具： **npm install -g @serverless-devs/s**。

安装完成还需要配置密钥，可以参考**密钥配置文档**。

步骤2 初始化一个函数计算的 Hello World 项目：**s init start-fg-http-nodejs14**

步骤3 初始化完成之后，进入项目，执行**s deploy**部署函数。

----结束

指令使用方法

华为云函数工作流（FunctionGraph）组件全部支持的能力如**表9-1**所示：

表 9-1 组件支持能力介绍

构建&部署	发布&配置	其他功能
部署deploy	**版本 version**	**项目迁移fun2s**
删除remove	**别名alias**	-

在使用华为云函数工作流（FunctionGraph）组件时，还会涉及到资源描述文件的编写，关于华为云函数工作流（FunctionGraph）组件的Yaml规范可以参考**华为云函数工作流（FunctionGraph）Yaml规范**章节。

9.5.2 密钥配置文档

获取密钥信息

1. 登录华为云后台，单击右上角“账号中心 > 我的凭证 ”，进入“我的凭证”界面。
2. 在左侧导航栏进入“访问密钥”界面，单击“新增访问密钥”生成新的密钥并下载保存。

配置密钥

引导式配置

可以通过执行config add直接进行密钥的添加：

```
$ s config add

? Please select a provider: (Use arrow keys)
  Alibaba Cloud (alibaba)
  AWS (aws)
  Azure (azure)
  Baidu Cloud (baidu)
  Google Cloud (google)
> Huawei Cloud (huawei)
  Tencent Cloud (tencent)
  Custom (others)
```

当您选择某个选项之后，系统会进行交互式引导：

```
s config add

? Please select a provider: Huawei Cloud (huawei)
? AccessKeyID **********
? SecretAccessKey **********
? Please create alias for key pair. If not, please enter to skip default
```

命令式配置

可以通过命令式直接进行密钥的添加：

```
$ s config add --AccessKeyID ****** --SecretAccessKey ******
```

或：

```
$ s config add -kl AccessKeyID,SecretAccessKey -il ${AccessKeyID},${SecretAccessKey}
```

9.5.3 指令使用方法

9.5.3.1 部署 deploy

deploy 命令

deploy 命令是对函数资源进行部署的命令，即将本地在 **Yaml文件** 中声明的资源部署到线上。

deploy 命令解析

当执行命令**deploy -h/deploy --help**时，可以获取帮助文档。

在该命令中，包括了两个子命令：deploy function命令和deploy trigger命令。

deploy function命令

- deploy function命令，是部署函数的命令。

  当执行命令**deploy function -h/deploy function --help**时，可以获取帮助文档。

  操作案例：

  有资源描述文件（Yaml）时，可以直接执行**s deploy function**进行函数的部署，描述文件（Yaml）示例：

```
fgs-deploy-test:
  region:   cn-north-4
  function:
    functionName: fgs-deploy-test
    handler: index.handler
    memorySize: 128
    timeout: 30
    runtime: Node.js14.18
    package: default
    codeType: zip
    code:
    codeUri: ./code
```

deploy trigger命令

- deploy trigger命令，是部署函数触发器的命令。

  当执行命令**deploy trigger -h/deploy trigger --help**时，可以获取帮助文档。

  操作案例：

  有资源描述文件（Yaml）时，可以直接执行**s deploy trigger**进行触发器的部署，描述文件（Yaml）示例：

```
fgs-deploy-test:
  region:   cn-north-4
  trigger:
    triggerTypeCode: APIG
    status: ACTIVE
    eventData:
      name: APIG_test
      groupName: APIGroup_xxx
      auth: IAM
      protocol: HTTPS
      timeout: 5000
```

  在进行服务资源部署时，可能会涉及到交互式操作，相关的描述参考 **deploy 命令 注意事项**中的在部署时可能会涉及到交互式操作。

参数解析

表 9-2 参数说明

参数全称	参数缩写	Yaml模式下是否必填	参数含义
type	-	选填	部署类型，可以选择code, config

操作案例

有资源描述文件（Yaml）时，可以直接执行s deploy 进行资源部署，描述文件（Yaml）示例：


```
fgs-deploy-test:
  region:   cn-north-4
  function:
    functionName: fgs-deploy-test
    handler: index.handler
    memorySize: 128
    timeout: 30
    runtime: Node.js14.18
    package: default
    codeType: zip
    code:
    codeUri: ./code
  trigger:
    triggerTypeCode: APIG
    status: ACTIVE
    eventData:
      name: APIG_test
      groupName: APIGroup_xxx
      auth: IAM
      protocol: HTTPS
      timeout: 5000
```

注意事项

在进行资源部署时，会涉及到一定的特殊情况，可以参考以下描述：

- 只需要部署/更新代码，可以增加--type code参数；
- 只需要部署/更新配置，可以增加--type config参数；

9.5.3.2 版本 version

version 命令

version 命令是进行函数版本操作的命令；主要包括别名的查看、发布、删除等功能。

命令解析

当执行命令**version -h/version --help**时，可以获取帮助文档。

在该命令中，包括了两个子命令：

- **version list命令**
- **version publish命令**

version list 命令

version list命令，是查看服务已发布的版本列表的命令。

当执行命令**version list -h/version list --help**时，可以获取帮助文档。

当前命令还支持部分全局参数（例如-a/--access, --debug等），详情可参考**Serverless Devs 全局参数**。

表 9-3 参数说明

参数全称	参数缩写	Yaml模式下是否必填	Cli模式下是否必填	参数含义
region	-	选填	必填	地区
function-name	-	选填	必填	函数名称
table	-	选填	必填	是否以表格形式输出

操作案例：

- 有资源描述文件（Yaml）时，可以直接执行**s version list**查看当前函数所发布的版本列表；
- 纯命令行形式（在没有资源描述Yaml文件时），需要指定服务所在地区以及服务名称，例如**s cli fgs version list --region cn-north-4 --function-name fg-test**

须知

执行cli 模式时，如果密钥信息不是default，需要添加 access参数，例如**s cli fgs version list --region cn-north-4 --function-name fg-test --access xxxx**

上述命令的执行结果示例：

```
fg-test:
 -
   version:       1
   description:       test publish version
   lastModifiedTime: 2021-11-08T06:07:00Z
```

如果指定了--table参数，则输出示例：

图 9-37 输出示例

```
┌─────────┬──────────────────┬──────────────────────┐
│ version │     description  │   lastModifiedTime   │
├─────────┼──────────────────┼──────────────────────┤
│ 1       │ test publish version │ 2021-11-08T06:07:00Z │
└─────────┴──────────────────┴──────────────────────┘
```

version publish 命令

version publish命令，是用于发布版本的命令。

当执行命令**version publish -h/version publish --help**时，可以获取帮助文档。

当前命令还支持部分全局参数（例如-a/--access, --debug等），详情可参考 **Serverless Devs 全局参数**。

表 9-4 参数说明

参数全称	参数缩写	Yaml模式下是否必填	Cli模式下是否必填	参数含义
region	-	选填	必填	地区
function-name	-	选填	必填	函数名称
version-name	-	选填	必填	版本号
description	-	选填	必填	版本描述

操作案例：

- 有资源描述文件（Yaml）时，可以直接执行**s version publish**进行版本的发布；
- 纯命令行形式（在没有资源描述Yaml文件时），需要指定服务所在地区以及服务名称，例如**s cli fgs version publish --region cn-north-4 --function-name fg-test --version-name 1 --description "test publish version"**

须知

执行cli 模式时，如果密钥信息不是default，需要添加 access参数，例如**s cli fgs version publish --region cn-north-4 --function-name fg-test --version-name 1 --description "test publish version" --access xxxx**

上述命令的执行结果示例：

```
fg-test:
  version:       1
  description:     test publish version
  lastModifiedTime: 2021-11-08T06:07:00Z
```

9.5.3.3 项目迁移 fun2s

fun2s 命令是将函数的配置信息转换成 Serverless Devs 所识别的 s.yaml的命令。

- 命令解析
  - 命令解析
  - 操作案例

命令解析

当执行命令**fun2s -h/fun2s --help**时，可以获取帮助文档。

当前命令还支持部分全局参数（例如-a/--access, --debug等），详情可参考 Serverless Devs 全局参数。

表 9-5 参数说明

参数全称	参数缩写	Cli模式下必填	参数含义
region	-	必填	地区
function-name	-	必填	函数名称
target	-	选填	生成的 Serverless Devs 的配置文档路径（默认是s.yaml）

操作案例

可以在 Funcraft 项目目录下，通过fun2s命令，实现Yaml规范转换，例如：

```
s cli fgs fun2s --region cn-north-4 --function-name fgs-deploy-test --target ./s.yml

Tips for next step
======================
* Deploy Function: s deploy -t ./s.yml
```

此时，就可以将原有的函数配置转换成支持 Serverless Devs 规范的 s.yaml。

转换后（s.yaml）：

```
edition: 1.0.0
name: transform_fun
access: default
vars:
  region: cn-north-4
  functionName: fgs-deploy-test
services:
  component-test: #  服务名称
    component: fgs # 组件名称
    props:
      region: ${vars.region}
      function:
        functionName: ${vars.functionName}
        handler: index.handler
        memorySize: 256
        timeout: 300
        runtime: Node.js14.18
        codeType: zip
        code:
          codeUri: ./code
```

9.5.3.4 删除 remove

remove 命令

remove 命令是对已经部署的资源进行移除的操作。**资源一旦移除可能无法恢复，所以在使用移除功能时，请您慎重操作。**

命令解析

当执行命令**remove -h/remove --help**时，可以获取帮助文档。

在该命令中，包括了四个子命令：

- **function：删除指定的函数**
- **trigger：删除指定的触发器**
- **version：删除指定的版本**
- **alias：删除指定的别名**

表 9-6 参数说明

参数全称	参数缩写	Yaml模式下必填	参数含义
assume-yes	y	选填	在交互时，默认选择y

操作案例：

有资源描述文件（Yaml）时，可以直接执行**s remove**进行资源删除，描述文件（Yaml）示例：

```
Function [myFunction] deleted successfully.
```

remove function 命令

remove function命令，是删除指定函数的命令。默认会把整个函数删除，包含所有的版本、别名以及触发器。

当执行命令**remove function -h/remove function --help**时，可以获取帮助文档。

表 9-7 参数说明

参数全称	参数缩写	Yaml模式下必填	Cli模式下必填	参数含义
region	-	选填	必填	地区
function-name	-	选填	必填	函数名
assume-yes	y	选填	必填	在交互时，默认选择y

操作案例：

- 有资源描述文件（Yaml）时，可以直接执行**s remove function**删除指定的函数；
- 纯命令行形式（在没有资源描述Yaml文件时），需要指定服务所在地区以及服务名称，例如**s cli fgs remove function --region cn-north-4 --function-name fgs-test**

须知

执行cli 模式时，如果密钥信息不是default，需要添加 access参数，例如**s cli fgs remove function --region cn-north-4 --function-name fgs-test --access xxxx**

上述命令的执行结果示例：

```
Function [fg-test] deleted.
```

remove trigger 命令

remove trigger命令，是删除指定触发器的命令。

当执行命令**remove trigger -h/remove trigger --help**时，可以获取帮助文档。

当前命令还支持部分全局参数（例如-a/--access, --debug等），详情可参考 **Serverless Devs 全局参数**。

表 9-8 参数说明

参数全称	参数缩写	Yaml模式下必填	Cli模式下必填	参数含义
region	-	选填	必填	地区
function-name	-	选填	必填	函数名
version-name	-	选填	选填	指定版本，不设置默认为latest版本
trigger-type	-	选填	必填	触发器类型
trigger-name	-	选填	必填	触发器名，APIG为API名称，OBS为桶名，TIMER为触发器名称
assume-yes	y	选填	选填	在交互时，默认选择y

操作案例：

- 有资源描述文件（Yaml）时，可以直接执行**s remove trigger**删除Yaml中声明的触发器；
- 纯命令行形式（在没有资源描述Yaml文件时），需要指定服务所在地区以及服务名称，例如**s cli fgs remove trigger --region cn-north-4 --function-name fgs-test --trigger-type APIG --trigger-name fgs-test-trigger**

上述命令的执行结果示例：

```
Trigger [fgs-test-trigger] deleted.
```

remove version 命令

remove version命令，是用户删除指定已发布的版本命令。

当执行命令**remove version -h/remove version --help**时，可以获取帮助文档。

当前命令还支持部分全局参数（例如-a/--access, --debug等），详情可参考**Serverless Devs 全局参数**。

表 9-9 参数说明

参数全称	参数缩写	Yaml模式下必填	Cli模式下必填	参数含义
region	-	选填	必填	地区
function-name	-	选填	必填	服务名
version-name	-	必填	必填	版本名称，不能为latest

操作案例：

- 有资源描述文件（Yaml）时，可以直接执行**s remove version --version-name versionName**删除指定versionName的版本；
- 纯命令行形式（在没有资源描述Yaml文件时），需要指定服务所在地区以及服务名称，例如**s cli fgs remove version --region cn-north-4 --function-name fgs-test --version-name v1**

上述命令的执行结果示例：

```
Version [v1] deleted.
```

remove alias 命令

remove alias命令，是删除指定服务别名的命令。

当执行命令**remove alias -h/remove alias --help**时，可以获取帮助文档。

当前命令还支持部分全局参数（例如-a/--access, --debug等），详情可参考**Serverless Devs 全局参数**。

表 9-10 参数说明

参数全称	参数缩写	Yaml模式下必填	Cli模式下必填	参数含义
region	-	选填	必填	地区
function-name	-	选填	必填	服务名
alias-name	-	必填	必填	别名

操作案例：

- 有资源描述文件（Yaml）时，可以直接执行**s remove alias --alias-name aliasName**删除指定别名；

- 纯命令行形式（在没有资源描述Yaml文件时），需要指定服务所在地区以及服务名称，例如**s cli fgs remove alias --region cn-north-4 --function-name fgs-test --alias-name pre**

上述命令的执行结果示例：

```
Alias [pre] deleted.
```

9.5.3.5 别名 alias

alias命令是对函数别名操作的命令；主要包括别名的查看、发布、修改、删除等功能。

命令解析

当执行命令**alias -h/alias --help**时，可以获取帮助文档。

在该命令中，包括了四个子命令：

- **alias get命令**
- **alias list命令**
- **alias publish命令**
- **remove alias命令**

alias get 命令

alias get命令，是获取服务指定别名详情的命令。

当执行命令**alias get -h/alias get --help**时，可以获取帮助文档。

当前命令还支持部分全局参数（例如-a/--access, --debug等），详情可参考**Serverless Devs 全局参数**。

表 9-11 参数说明

参数全称	参数缩写	Yaml模式下必填	Cli模式下必填	参数含义
region	-	选填	必填	地区
function-name	-	选填	必填	函数名称
alias-name	-	必填	必填	别名

操作案例：

- 有资源描述文件（Yaml）时，可以直接执行**s alias get --alias-name aliasName**进行指定的别名详情获取；
- 纯命令行形式（在没有资源描述Yaml文件时），需要指定服务所在地区以及服务名称，例如**s cli fgs alias get --region cn-north-4 --function-name fg-test --alias-name pre**

须知

执行cli 模式时，如果密钥信息不是default，需要添加 access参数，例如**s cli fgs alias get --region cn-north-4 --function-name fg-test --alias-name pre --access xxxx**

上述命令的执行结果示例：

```
fg-test:
  aliasName:              pre
  versionId:              1
  description:            test publish version
  additionalVersionWeight:
  createdTime:            2021-11-08T06:51:36Z
  lastModifiedTime:       2021-11-08T06:54:02Z
```

alias list 命令

alias list命令，是进列举别名列表的命令。

当执行命令**alias list -h/alias list --help**时，可以获取帮助文档。

当前命令还支持部分全局参数（例如-a/--access, --debug等），详情可参考 **Serverless Devs 全局参数**。

表 9-12 参数说明

参数全称	参数缩写	Yaml模式下必填	Cli模式下必填	参数含义
region	-	选填	必填	地区
function-name	-	选填	必填	函数名称
table	-	选填	选填	是否以表格形式输出

操作案例：

- 有资源描述文件（Yaml）时，可以直接执行**s alias list**获取别名列表；
- 纯命令行形式（在没有资源描述Yaml文件时），需要指定服务所在地区以及服务名称，例如**s cli fgs alias list --region cn-north-4 --function-name fg-test**

须知

执行cli 模式时，如果密钥信息不是default，需要添加 access参数，例如**s cli fgs alias list --region cn-north-4 --function-name fg-test --access xxxx**

上述命令的执行结果示例：

```
fg-test:
  -
    aliasName:              pre
    versionId:              1
    description:            test publish version
```



```
lastModifiedTime:       2021-11-08T06:54:02Z
additionalVersionWeight:
```

如果指定了--table参数，如图9-38所示：

图 9-38 输出示例

```
┌──────┬─────────┬──────────────────┬──────────────────────┬──────────────────────┐
│ name │ version │ description      │ lastModifiedTime     │ additionalVersionWeight │
├──────┼─────────┼──────────────────┼──────────────────────┼──────────────────────┤
│ pre  │ 1       │ test publish version │ 2021-11-08T06:54:02Z │                      │
└──────┴─────────┴──────────────────┴──────────────────────┴──────────────────────┘
```

alias publish 命令

alias publish命令，是对别名进行发布和更新的命令。

当执行命令**alias publish -h/alias publish --help**时，可以获取帮助文档。

当前命令还支持部分全局参数（例如-a/--access, --debug等），详情可参考 **Serverless Devs 全局参数**。

表 9-13 参数说明

参数全称	参数缩写	Yaml模式下必填	Cli模式下必填	参数含义
region	-	选填	必填	地区
function-name	-	选填	必填	函数数名
alias-name	-	必填	必填	别名
version-name	-	选填	必填	别名对应的版本名称
description	-	选填	选填	别名描述
gversion	-	选填	选填	灰度版本 Id。灰度版本权重填写时必填
weight	-	选填	选填	灰度版本权重。灰度版本 Id 填写时必填

操作案例：

- 有资源描述文件（Yaml）时，可以直接执行**s alias publish**进行别名的发布或者更新；
- 纯命令行形式（在没有资源描述Yaml文件时），需要指定服务所在地区以及服务名称，例如 **s cli fgs alias publish --region cn-north-4 --function-name fg-test --alias-name pre --version-name 1**

须知

执行cli 模式时，如果密钥信息不是default，需要添加 access参数，例如**s cli fgs alias publish --region cn-north-4 --function-name fg-test --alias-name pre --version-name 1 --access xxxx**

上述命令的执行结果示例：

```
fg-test:
  aliasName:              pre
  versionId:              1
  description:
  additionalVersionWeight:
  createdTime:            2021-11-08T06:51:36Z
  lastModifiedTime:       2021-11-08T06:51:36Z
```

如果需要对别名进行升级，只需要指定别名之后，进行相对应的参数更新，例如针对上述的pre别名，指定--description参数后再次执行上述命令，执行示例：

```
fc-deploy-test:
  aliasName:              pre
  versionId:              1
  description:            test publish version
  additionalVersionWeight:
  createdTime:            2021-11-08T06:51:36Z
  lastModifiedTime:       2021-11-08T06:54:02Z
```

remove alias 命令

具体命令详情请参考**remove alias命令**。

9.5.3.6 Yaml 文件

Yaml 完整配置

华为云函数工作流（FunctionGraph）组件的Yaml字段如下：

```
edition: 1.0.0 #  命令行YAML规范版本，遵循语义化版本（Semantic Versioning）规范
name: fg-test #  项目名称
access: "default" #  秘钥别名

vars: # 全局变量
   region: "cn-east-3"
   functionName: "start-fg-event-nodejs14"

services:
   component-test: #  服务名称
      component: fgs # 组件名称
      props:
         region: ${vars.region}
         function:
            functionName: ${vars.functionName} # 函数名
            handler: index.handler # 函数执行入口
            memorySize: 256 # 函数消耗的内存
            timeout: 30 # 函数执行超时时间
            runtime: Node.js14.18 # 运行时
            agencyName: fgs-vpc-test # 委托名称
            environmentVariables: # 环境变量
               test: test
               hello: world
            vpcId: xxx-xxx # 虚拟私有云唯一标识
            subnetId: xxx-xxx # 子网编号
            concurrency: 10 # 单函数最大实例数
            concurrentNum: 10 # 单实例最大并发数
            codeType: zip # 函数代码类型
```



```
        dependVersionList: # 依赖包，取依赖包的ID
          - xxx-xxx
        code: # 本地代码地址
          codeUri: ./code
    trigger:
        triggerTypeCode: TIMER # 触发器类型
        status: DISABLED # 触发器状态
        eventData: # 触发器配置
          name: APIG_test  # API名称
          groupName: APIGroup_xxx # 分组名称
          auth: IAM # 安全认证
          protocol: HTTPS # 请求协议
          timeout: 5000 # 后端超时时间
```

表 9-14 参数说明

参数	必填	类型	参数描述
region	True	Enum	地域
function	True	**Struct**	函数
triggers	False	**Struct**	触发器

function 字段介绍

Yaml文件中function字段说明请参考**表9-15**。

表 9-15 function 字段说明

参数名	必填	类型	参数描述
functionName	True	String	函数名称。
handler	True	String	函数执行入口，规则：xx.xx，必须包含“.”。
runtime	True	String	运行时。 目前支持如下： • Node.js14.18、Node.js12.13、Node.js10.16、Node.js8.10、Node.js6、Node.js4.4 • Python3.9、Python3.6、Python2.7 • Java11、Java8 • Go1.x、Go1.8 • PHP7.3 • http • Custom
package	False	String	函数所属的分组Package，用于用户针对函数的自定义分组，默认为default。

参数名	必填	类型	参数描述
memorySize	True	Number	函数消耗的内存，单位M。 取值范围为：128、256、512、768、1024、1280、1536、1792、2048、2560、3072、3584、4096。
timeout	True	Number	函数执行超时时间，超时函数将被强行停止，范围3～900秒。
CodeType	True	String	函数代码类型。 • inline: UI在线编辑代码。 • zip: 函数代码为zip包。 • obs: 函数代码来源于obs存储。 • jar: 函数代码为jar包，主要针对Java函数。
codeUrl	False	String	当CodeType为obs时，该值为函数代码包在OBS上的地址，CodeType为其他值时，该字段为空。
environmentVariables	False	**Struct**	环境变量。最多定义20个，总长度不超过4KB。
agencyName	False	String	委托名称，需要IAM支持，并在IAM界面创建委托，当函数需要访问其他服务时，必须提供该字段。
vpcId	False	String	虚拟私有云唯一标识。配置时，agencyName必填。虚拟私有云标识请登录**虚拟私有云页面**查看。
subnetId	False	String	子网编号。配置时，agencyName必填。子网编号请登录**虚拟私有云子网页面**查看。
dependVersionList	False	List<String>	依赖包，取依赖包的ID。
code	False	**Struct**	本地代码地址，当CodeType为zip时必填。
concurrency	False	Number	单函数最大实例数，取值-1到1000。 -1代表该函数实例数无限制；0代表该函数被禁用。
concurrentNum	False	Number	单实例最大并发数，取值-1到1000。
description	False	String	function 的简短描述。

- Func Code参数说明：

  表 9-16 Func Code 参数说明

参数名	必填	类型	参数描述
codeUri	True	String	本地代码地址

- Environment Variables参数说明：

  Object 格式，例如：

  ```
  DB_connection: jdbc:mysql://ip:port/dbname
  ```

  当然不推荐通过明文将敏感信息写入到s.yaml。

参考案例：

```
function:
  functionName: event-function
  description: this is a test
  runtime: Node.js14.18
  handler: index.handler
  memorySize: 128
  timeout: 60
  code:
    codeUri: ./code
  environmentVariables:
    test: 123
    hello: world
```

triggers 字段介绍

Yaml文件中triggers字段说明请参考**表9-17**。

表 9-17 trigger 参数说明

参数名	必填	类型	参数描述
triggerTypeCode	True	String	触发器类型。
status	False	Enum	触发器状态，取值为 ACTIVE、DISABLED，默认为 ACTIVE。
eventData	True	Struct	触发器配置，包括**•APIG触发器**, **•TIMER触发器**。

- APIG触发器

  表 9-18 APIG 参数说明

参数名	必填	类型	参数描述
name	False	String	API名称，默认使用函数名。
groupName	False	String	分组，默认选择当前第一个。

参数名	必填	类型	参数描述
auth	False	Enum	安全认证，默认为 IAM。 API认证方式： • App： 采用Appkey&Appsecret认证，安全级别高，推荐使用，详情请参见**APP认证**。 • IAM： IAM认证，只允许IAM用户能访问，安全级别中等，详情请参见**IAM认证**。 • None： 无认证模式，所有用户均可访问。
protocol	False	Enum	请求协议，默认为 HTTPS。 分为两种类型： • HTTP • HTTPS
timeout	False	Number	后端超时时间，单位为毫秒，取值范围为 1 ~ 60000。默认为 5000。

参考案例：

```
trigger:
  triggerTypeCode: APIG
  status: ACTIVE
  eventData:
    name: APIG_test
    groupName: APIGroup_xxx
    auth: IAM
    protocol: HTTPS
    timeout: 5000
```

- TIMER触发器

表 9-19 TIMER 参数说明

参数名	必填	类型	参数描述
name	False	String	定时器名称。
scheduleType	True	Enum	触发规则，取值为 Rate、**Cron**。
schedule	True	String	定时器规则内容。
userEvent	False	String	附加信息，如果用户配置了触发事件，会将该事件填写到TIMER事件源的“user_event”字段。

参考案例：

```
trigger:
  triggerTypeCode: TIMER
  status: ACTIVE
  eventData:
```



```
        name: Timer-xxx
        scheduleType: Rate
        schedule: 3m
        userEvent: xxxx

    trigger:
      triggerTypeCode: TIMER
      status: ACTIVE
      eventData:
        name: Timer-xxx
        scheduleType: Cron
        schedule: 0 15 2 * * ?
        userEvent: xxxx
```

9.5.4 华为云函数工作流（FunctionGraph）Yaml 规范

字段解析

表 9-20 参数说明

参数名	必填	类型	参数描述
region	True	Enum	Enum
function	True	Struct	函数
trigger	False	Struct	触发器

Yaml 完整配置

华为云函数工作流（FunctionGraph）组件的Yaml字段如下：

```
edition: 1.0.0 #  命令行YAML规范版本，遵循语义化版本（Semantic Versioning）规范
name: fg-test #  项目名称
access: "default" #  秘钥别名

vars: # 全局变量
  region: "cn-east-3"
  functionName: "start-fg-event-nodejs14"

services:
  component-test: #  服务名称
    component: fgs # 组件名称
    props:
      region: ${vars.region}
      function:
        functionName: ${vars.functionName} # 函数名
        handler: index.handler # 函数执行入口
        memorySize: 256 # 函数消耗的内存
        timeout: 30 # 函数执行超时时间
        runtime: Node.js14.18 # 运行时
        agencyName: fgs-vpc-test # 委托名称
        environmentVariables: # 环境变量
          test: test
          hello: world
        vpcId: xxx-xxx # 虚拟私有云唯一标识
        subnetId: xxx-xxx # 子网编号
        concurrency: 10 # 单函数最大实例数
        concurrentNum: 10 # 单实例最大并发数
        codeType: zip # 函数代码类型
        dependVersionList: # 依赖包，取依赖包的ID
          - xxx-xxx
        code: # 本地代码地址
```



```
            codeUri: ./code
        trigger:
            triggerTypeCode: TIMER # 触发器类型
            status: DISABLED # 触发器状态
            eventData: # 触发器配置
                name: APIG_test  # API名称
                groupName: APIGroup_xxx # 分组名称
                auth: IAM # 安全认证
                protocol: HTTPS # 请求协议
                timeout: 5000 # 后端超时时间
```

9.5.5 Serverless Devs 全局参数

表 9-21 Serverless Devs 全局参数介绍

参数全称	参数缩写	默认取值	参数含义	备注
template	t	s.yaml/s.yml	指定资源描述文件	-
access	a	yaml中所指定的access信息/default	指定本次部署时的密钥信息	可以使用通过config命令配置的密钥信息，以及配置到环境变量的密钥信息
skip-actions	-	-	跳过yaml所设置的actions模块	-
debug	-	-	开启Debug模式	开启Debug模式后可以查看到更多的工具执行过程信息
output	o	default	指定数据的输出格式	支持default, json, yaml, raw格式
version	v	-	查看版本信息	-
help	h	-	查看帮助信息	-

9.6 Serverless Framework

9.6.1 使用指南

欢迎使用华为云函数工作流Serverless使用指南。

说明

您在继续操作之前，使用CLI需要先提供**华为云用户凭证**。

9.6.1.1 简介

Serverless Framework帮助您使用华为云函数工作流开发和部署无服务器应用。它是一个CLI，提供开箱即用的结构、自动化功能和最佳实践，您可以专注于构建复杂的、事件驱动的、无服务器架构，由**函数**和**事件**组成。

Serverless Framework与其他应用程序框架不同，因为它：

- 管理您的代码和基础设施。
- 支持多种语言（Node.js、Python、Java等）。

核心概念

以下将介绍Framework的主要概念，以及它们与华为云函数工作流的关系。

函数

函数是**华为云函数工作流函数**。它是一个独立的部署单元，就像微服务一样。它只是部署在云中的代码，主要是为了执行单个任务而编写，例如：

- 将用户保存到数据库。
- 处理数据库中的文件。

您可以在代码中执行多个任务，但不建议在没有充分理由的情况下这样做。分离关注点是最好的，Framework旨在帮助您轻松开发和部署函数，以及管理它们。

事件

任何触发华为云函数工作流的函数执行的事物都被Framework视为**事件**。事件是指华为云函数工作流上的平台事件，例如：API网关服务和API（例如，REST API）、OBS桶（例如，上传到桶中的镜像）等等。

在Serverless Framework中为华为云函数工作流定义事件时，Framework会自动将事件及其函数转换为相应的云资源。这样就可以配置事件，以便您的函数可以侦听它。

服务

服务是Framework的组织单位。您可以将其视为项目文件，单个应用可以拥有多个服务。可以在服务中定义函数、触发它们的事件以及函数使用的资源，所有这些都在一个名为serverless.yml（或serverless.json）的文件中，例如：

```
# serverless.yml

service: fgs

functions: # Your "Functions"
  hello_world:
    events: # The "Events" that trigger this function
      - apigw:
          env_id: DEFAULT_ENVIRONMENT_RELEASE_ID
          env_name: RELEASE
          req_method: GET
          path: /test
          name: API_test
```

通过运行serverless deploy使用Framework进行部署时，serverless.yml中的所有内容都会同时部署。

插件

可以使用**插件**覆盖或扩展Framework的功能。每个serverless.yml都可以包含一个“plugins:属性”，该属性具有多个插件。

```
# serverless.yml

plugins:
  - serverless-huawei-functions
```

9.6.1.2 快速入门

本章节旨在帮助您尽快了解Serverless Framework的使用。

初始化设置

您需要安装和配置以下几个前提条件：

- 在本地计算机上安装Node.js 14.x或更高版本，详情请参见**安装Node.js和NPM**。
- 安装Serverless Framework开源CLI版本3.28.1或更高版本，详情请参见**安装Serverless Framework的开源CLI**。

如果已经具备了这些前提条件，则可以跳过部署示例服务。

安装 Node.js 和 NPM

步骤1 安装Node.js和NPM，下载地址请参考**下载说明**。

步骤2 最后，您应该能够从命令行中运行node -v，并获得以下结果：

```
$ node -v
vx.x.x
```

同时，您还能够从命令行中运行npm -v，并获得以下结果：

```
$ npm -v
x.x.x
```

----结束

安装 Serverless Framework 的开源 CLI

步骤1 在终端中运行如下命令：

```
npm install -g serverless
```

步骤2 安装完成后，您能够从命令行中运行serverless -v，并获得以下结果：

```
$ serverless -v
x.x.x
```

----结束

创建并部署 serverless 服务

当前您已经完成了设置，可以开始创建和部署serverless服务。

步骤1 创建新服务。

1. 使用huawei-nodejs模板创建新服务。

   ```
   serverless create --template-url https://github.com/zy-linn/examples/tree/v3/legacy/huawei-nodejs --path my-service
   ```


2. 安装依赖项。
```
cd my-service
npm install
```

步骤2 设置凭证，详情请参考**凭证设置**。

步骤3 更新serverless.yml。

更新项目serverless.yml中的region和credentials。

步骤4 部署。

使用如下命令的场景为首次部署服务，以及在更改serverless.yml中的函数、事件或资源之后，希望同时部署服务中的所有更改。该命令详情请参考**Deploy命令**。
```
serverless deploy
```

----结束

9.6.1.3 安装

Serverless是一个Node.js CLI工具，因此您需要先在计算机上安装Node.js。

请访问**Node.js官方网站**，下载并按照**安装说明**在本地计算机上安装Node.js。

您可以通过在终端中运行node --version来验证Node.js是否安装成功，即可以看到打印出来的对应Node.js版本号。

安装 Serverless Framework

步骤1 通过**npm**安装Serverless Framework，它在安装Node.js时已经安装。

步骤2 打开终端，输入npm install -g serverless安装Serverless。
```
npm install -g serverless
```

步骤3 安装完成后，可以通过在终端中运行以下命令来验证Serverless是否安装成功。
```
serverless
```

查看安装的Serverless版本，请运行：
```
serverless --version
```

----结束

安装华为云函数工作流提供商插件

从npm安装最新的软件包，请运行：
```
npm i --save serverless-huawei-functions
```

设置华为云函数工作流

运行向华为云发出请求的Serverless命令，需要在您的计算机上设置华为云凭证，具体详情请参考**设置华为云凭证**。

9.6.1.4 凭证

Serverless Framework需要访问您的华为云账号的凭证，代表您创建和管理资源。

创建华为云账号

打开**华为云官网**，选择“注册”，详情请参考**注册华为账号并开通华为云**。

获取凭证

您需要创建凭证，以便Serverless可以使用它们在项目中创建资源。

步骤1 进入“访问密钥”页面，获取您华为云账号的访问密钥。

步骤2 创建一个名为credentials的文件，其中包含您收集的凭证。

```
access_key_id=<collected in step 1>
secret_access_key=<collected in step 1>
```

步骤3 将凭证文件保存在安全的地方。建议在根文件夹中创建一个文件夹用于放置凭证，如：~/.fg/credentials，并记住其保存路径。

----结束

更新 serverless.yml 中的 provider 配置

打开您的serverless.yml文件，并使用凭证文件的路径更新provider部分（这里需要使用绝对路径）。结果应该类似于如下：

```
provider:
  name: huawei
  runtime: Node.js14.18
  credentials: ~/.fg/credentials
```

9.6.1.5 服务

service就像一个项目，您可以在服务中定义华为云函数工作流的函数和触发它们的events，所有这些都在一个名为serverless.yml的文件中。

若您要构建第一个Serverless Framework项目，请先创建一个service。

组织

在最初使用应用时，建议您可以使用单个服务来定义该项目的所有函数和事件。

```
myService/
  serverless.yml  # Contains all functions and infrastructure resources
```

但是，随着应用增多，您可以将其拆分为多个服务。大多数用户按工作流或数据模型组织他们的服务，并在服务中将与这些工作流和数据模型相关的函数进行分组。

```
users/
  serverless.yml # Contains 4 functions that do Users CRUD operations and the Users database
posts/
  serverless.yml # Contains 4 functions that do Posts CRUD operations and the Posts database
comments/
  serverless.yml # Contains 4 functions that do Comments CRUD operations and the Comments database
```

这是有意义的，因为相关函数通常使用公共基础设施资源，并且用户希望将这些函数和资源作为单个部署单元放在一起，以便更好地组织和分离关注点。

创建

使用create命令创建服务。您可以输入路径创建目录并将服务自动命名：

```
# Create service with Node.js template in the folder ./my-service
serverless create --template-url https://github.com/zy-linn/examples/tree/v3/legacy/huawei-nodejs --path
my-service
```

huawei-nodejs是华为云函数工作流的可用运行时。

有关所有详细信息和选项，请查看创建。

目录

您将在工作目录中看到以下文件：

- serverless.yml
- src/index.js

serverless.yml

每个service配置都在serverless.yml文件中管理。本文件的主要用途是：

- 声明Serverless服务。
- 在服务中定义一个或多个函数：
  - 定义服务将被部署到的提供商（如果有运行时，也要定义）。
  - 定义要使用的任何自定义插件。
  - 定义触发每个函数执行的事件（如HTTP请求）。
  - 允许“events”部分中列出的事件在部署时自动创建事件所需的资源。
  - 允许使用Serverless变量进行灵活配置。

您可以看到服务名称、提供商配置，以及functions定义中的第一个函数。任何后续的服务配置都将在此文件中完成。

```
# serverless.yml
service: my-fc-service

provider:
  name: huawei
  runtime: Node.js14.18
  credentials: ~/.fg/credentials # path must be absolute

plugins:
  - serverless-huawei-functions

functions:
  hello_world:
    handler: index.handler
```

index.js

index.js文件包含您导出的函数。

部署

部署服务时，serverless.yml中的所有函数和事件都会转换为对华为云API的调用，用于动态定义这些资源。

使用deploy命令部署服务：

```
serverless deploy
```

查看**部署指南**，了解有关部署的更多信息及其工作原理。有关所有详细信息和选项，请查看**deploy命令文档**。

移除

为了方便地在华为云上移除您的服务，可以使用remove命令。

运行serverless remove触发移除进程。

Serverless开始移除时，会在控制台中通知您进程。移除整个服务后，打印成功消息。

移除过程将仅移除提供商基础设施上的服务。服务目录仍将保留在本地计算机上，因此您仍可以在稍后修改并（重新）将其部署到另一个环境、区域或提供商。

版本固定

Serverless Framework通常通过npm install -g serverless全局安装。因此，您的所有服务都可以使用Serverless CLI。

全局安装工具的缺点是无法将版本固定在package.json内部。如果升级Serverless，但您的同事或CI系统不升级，这可能会导致问题。您可以在serverless.yml中使用某个特性，而不必担心CI系统会使用旧版本的Serverless进行部署。

- **固定版本**

  要配置版本固定，请在serverless.yml中定义frameworkVersion属性。每当您从CLI运行Serverless命令时，它都会检查当前Serverless版本是否匹配frameworkVersion的范围。CLI使用**语义化版本**，因此您可以将其固定为明确的版本或提供版本范围。一般来说，建议固定到明确的版本，以确保团队中的每个人都有完全相同的设置，并且不会发生意想不到的问题。

  示例

  明确的版本。

  ```
  # serverless.yml

  frameworkVersion: '2.1.0'
  ```

  版本范围。

  ```
  # serverless.yml

  frameworkVersion: ^2.1.0 # >=2.1.0 && <3.0.0
  ```

在现有服务中安装 Serverless

如果您已经有Serverless服务，并且更愿意使用package.json锁定框架版本，那么您可以按以下方式安装Serverless：

```
# from within a service
npm install serverless --save-dev
```

- **本地调用Serverless**

  要执行本地安装的Serverless，您必须引用node_ modules目录中的二进制文件，示例如下：

  ```
  node ./node_modules/serverless/bin/serverless deploy
  ```

9.6.1.6 函数

如果您以华为云函数工作流作为提供商，则服务的所有函数都属于华为云函数工作流中的函数。

配置

您的Serverless服务中有关华为云函数工作流的所有内容都可以在functions属性下的serverless.yml中找到。

```
# serverless.yml
service: fg-service

provider:
  name: huawei

plugins:
```



```
  - serverless-huawei-functions

functions:
  first:
    handler: index.handler
```

执行入口

handler属性应该是您在入口文件中导出的函数名称。

例如，当您导出函数并以index.js中的handler命名时，您的handler应该是handler: index.handler。

```
// index.js
exports.handler = (event, context, callback) => {};
```

内存大小和超时

函数的memorySize和timeout可以在提供商或函数层面指定。提供商层面的定义允许所有函数共享此配置，而函数层面的定义意味着此配置仅对当前函数有效。

如果未指定，默认memorySize为256MB，timeout为30s。

```
# serverless.yml

provider:
  memorySize: 512
  timeout: 90

functions:
  first:
    handler: first
  second:
    handler: second
    memorySize: 256
    timeout: 60
```

执行入口签名

事件执行入口的签名如下：

```
function (event, context) { }
```

- **event**

  如果函数由指定的APIG事件触发，则传递给执行入口的event如下：

  ```
  // JSON.parse(event)
  {
    events: {
      "body": "",
      "requestContext": {
          "apiId": "xxx",
          "requestId": "xxx",
          "stage": "RELEASE"
      },
      "queryStringParameters": {
          "responseType": "html"
      },
      "httpMethod": "GET",
      "pathParameters": {},
      "headers": {
          "accept-language": "zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2",
          "accept-encoding": "gzip, deflate, br",
          "x-forwarded-port": "443",
          "x-forwarded-for": "xxx",
          "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
          "upgrade-insecure-requests": "1",
          "host": "xxx",
  ```



```
      "x-forwarded-proto": "https",
      "pragma": "no-cache",
      "cache-control": "no-cache",
      "x-real-ip": "xxx",
      "user-agent": "Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:57.0) Gecko/20100101 Firefox/57.0"
    },
    "path": "/apig-event-template",
    "isBase64Encoded": true
  }
}
```

- **context**

  context参数包含有关函数的运行时信息。例如：请求ID、临时AK、函数元数据。具体详情请参见**开发事件函数**。

9.6.1.7 事件

简单地说，事件主要用于触发函数运行。

如果您选择华为云作为提供商，则服务中的events仅限于华为云API网关（APIG）和OBS，具体详情请参见**事件列表**。

部署后，Framework将设置您的function应该侦听的相应事件配置。

配置

事件属于每个函数，可以在serverless.yml的events属性中找到。

```
# serverless.yml
functions:
  first: # Function name
    handler: index.http # Reference to file index.js & exported function 'http'
    events:
      - apigw:
          env_id: DEFAULT_ENVIRONMENT_RELEASE_ID
          env_name: RELEASE
          req_method: GET
          path: /test
          name: API_test
```

说明

目前，每个函数只支持一个事件定义。

类型

Serverless Framework支持华为云函数工作流的obs和APIG事件，详细信息请参见**事件列表**。

部署

要部署或更新函数和事件，请运行：

```
serverless deploy
```

9.6.1.8 部署

Serverless Framework旨在安全、快速地为您创建华为云函数工作流的函数、事件和资源。

全量部署

如下是使用Serverless Framework执行部署的主要方法：

```
serverless deploy
```

当您在serverless.yml中更新了函数、事件或资源配置，并且希望将该更改（或多个更改）部署到华为云时，请使用此方法。

工作原理

Serverless Framework将serverless.yml中的所有语法转换为华为云部署管理的配置模板。

1. 提供商插件解析serverless.yml配置并转换为华为云资源。
2. 然后将函数的代码打包到目录中，压缩并上传到部署桶中。
3. 资源部署完成。

说明

建议在CI/CD系统中使用此方法，因为它是最安全的部署方法。

有关所有详细信息和选项，请查看**deploy命令文档**。

9.6.1.9 打包

打包 CLI 命令

使用Serverless CLI工具，可以将项目打包，而无需将其部署到华为云。建议与CI/CD工作流一起使用，以确保可部署产物一致。

运行以下命令将在服务的.serverless目录中构建和保存所有部署产物：

```
serverless package
```

打包配置

有时，您可能希望对函数产物以及它们的打包方式有更多的控制。

您可以使用patterns配置来更多地控制打包过程。

- **Patterns**

  您可以定义将从结果产物中排除/包括的全局模式。如果您希望排除文件，可以使用前缀为“!”的全局模式，如：!exclude-me/**。Serverless Framework将运行全局模式，以便您始终可以重新包含以前排除的文件和目录。

  示例

  排除所有node_modules，然后专门使用exclude重新包含的特定模块（在本例中为node-fetch）：

  ```
  package:
    patterns:
      - '!node_modules/**'
      - 'node_modules/node-fetch/**'
  ```

  排除handler.js以外的所有文件：

  ```
  package:
    patterns:
      - '!src/**'
      - src/function/handler.js
  ```


说明

如果要排除目录，请不要忘记使用正确的全局语法，可参考如下：

```
package:
  patterns:
    - '!tmp/**'
    - '!.git/**'
```

- 产物

  要完全控制打包过程，您可以指定自己的产物ZIP文件。

  如果配置了此功能，Serverless将不会压缩您的服务，因此将忽略patterns，可以选择使用产物或模式进行部署。

  如果您的开发环境允许您像Maven为Java一样生成可部署的产物，则产物选项能起到很大帮助。

  示例

```
service: my-service
package:
  patterns:
    - '!tmp/**'
    - '!.git/**'
    - some-file
  artifact: path/to/my-artifact.zip
```

- **分别打包函数**

  如果您希望对部署的函数进行更多的控制，您可以配置将它们分别进行打包。这样可以通过更多控制，对部署进行优化。要启用单独打包，请在服务或函数的打包设置中将individually设置为true。

  然后，对于每个函数，您都可以使用适用于整个服务的patterns或artifact配置选项。patterns选项将与服务选项合并，在打包期间为每个函数创建一个patterns配置。

```
service: my-service
package:
  individually: true
  patterns:
    - '!excluded-by-default.json'
functions:
  hello:
    handler: handler.hello
    package:
      # We're including this file so it will be in the final package of this function only
      patterns:
        - excluded-by-default.json
  world:
    handler: handler.hello
    package:
      patterns:
        - '!some-file.js'
```

  您还可以选择需要单独打包的函数，并通过在函数级别设置individually标志，让其余的函数使用服务包：

```
service: my-service
functions:
  hello:
    handler: handler.hello
  world:
    handler: handler.hello
    package:
      individually: true
```

- **开发依赖**

  Serverless将根据您的服务正在使用的运行时来自动检测和排除开发依赖项。从而确保ZIP文件中仅包含与生产相关的软件包和模块。这样做可以大幅减小上传到云提供商的部署包总大小。

可以通过将excludeDevDependency包配置设置为false来选择退出自动排除开发依赖项：

```
package:
  excludeDevDependencies: false
```

9.6.1.10 变量

Serverless Framework提供了一个强大的变量系统，您可以将动态数据添加到serverless.yml中。使用Serverless变量，您将能够执行以下操作：

- 引用并加载环境变量中的变量。
- 引用并加载CLI选项中的变量。
- 递归引用同一serverless.yml文件中的任何类型的属性。
- 递归引用其他YAML/JSON文件中的任何类型的属性。
- 递归地嵌套变量引用，提高灵活性。
- 合并多个变量引用以相互覆盖。

约束与限制

只能在serverless.yml的**values**属性中使用变量，而不能使用键属性。因此，您不能在自定义资源部分中使用变量生成动态逻辑ID。

从环境变量中引用变量

要引用环境变量中的变量，请在serverless.yml中使用${env:someProperty}语法，如下：

```
service: new-service

provider:
  name: huawei
  runtime: Node.js14.18
  credentials: ~/.fg/credentials # path must be absolute
  environment:
    variables:
      ENV_FIRST: ${env:TENCENTCLOUD_APPID}

plugins:
  - serverless-huawei-functions

functions:
  hello:
    handler: index.hello
```

9.6.2 CLI 参考

欢迎使用华为云函数工作流Serverless命令行参考。

说明

您在继续操作之前，使用命令行需要提供**华为云凭证**。

9.6.2.1 创建

根据指定的模板在当前工作目录下创建新服务。

- 在当前工作目录中创建服务：

```
serverless create --template-url https://github.com/zy-linn/examples/tree/v3/legacy/huawei-nodejs
```


- 使用自定义模板在新文件夹中创建服务：
```
serverless create --template-url https://github.com/zy-linn/examples/tree/v3/legacy/huawei-nodejs --path my-service
```

选项

- --template-url或-u：指向远程托管模板的URL。**如果未指定--template和--template-path，则该选项必填。**
- --template-path：模板的本地路径。**如果未指定--template和--template-url，则该选项必填。**
- --path或-p：新建服务所在路径。
- --name或-n：serverless.yml中服务的名称。

示例

- **创建新服务**
```
serverless create --template-url https://github.com/zy-linn/examples/tree/v3/legacy/huawei-nodejs --name my-special-service
```
此示例将为服务生成Node.js运行时。华为作为提供商，该运行时将在当前工作目录中生成。
- **在（新）目录中创建指定名称的服务**
```
serverless create --template-url https://github.com/zy-linn/examples/tree/v3/legacy/huawei-nodejs --path my-new-service
```
此示例将为服务生成Node.js运行时。华为作为提供商，该运行时将在my-new-service目录中生成；如不存在该目录，则会自动生成。在其他情况下Serverless将使用已经存在的目录。

此外，Serverless将根据您提供的路径将服务重命名。在此示例中，服务将重命名为my-new-service。

9.6.2.2 安装

serverless install 命令

在当前工作目录中从GitHub URL安装服务，如下：
```
serverless install --url https://github.com/some/service
```

选项

- --url或-u：GitHub的服务URL，必填。
- --name或-n：服务名称。

示例

- **从GitHub URL安装服务**
```
serverless install --url https://github.com/zy-linn/examples/tree/v3/legacy/huawei-nodejs
```
本示例将从GitHub下载huawei-nodejs服务的.zip文件，在当前工作目录下创建一个名为huawei-nodejs的新目录，并将文件解压到该目录下。
- **使用新服务名称从GitHub URL安装服务**
```
serverless install --url https://github.com/zy-linn/examples/tree/v3/legacy/huawei-nodejs--name my-huawei-service
```
此示例执行过程如下：

a. 从GitHub下载huawei-nodejs服务的.zip文件。
b. 在当前工作目录中创建名为my-huawei-service的新目录。
c. 在此目录中解压文件。
d. 如果根服务中存在serverless.yml，则将服务重命名为my-huawei-service。

- **从GitHub URL中的目录安装服务**

```
serverless install --url https://github.com/zy-linn/examples/tree/v3/legacy/huawei-nodejs
```

本示例将从GitHub下载huawei-nodejs服务。

9.6.2.3 打包

默认情况下，severless package命令将所有基础设施资源打包到.serverless目录中用于部署。

serverless package 命令

```
serverless package
```

在该示例中，您的服务会被打包。生成的软件包将默认位于服务的.serverless目录。

9.6.2.4 部署

serverless deploy 命令

serverless deploy命令通过华为云API部署整个服务。当您编辑serverless.yml文件更改了服务时，请运行此命令。

```
serverless deploy
```

产物

执行serverless deploy命令后，所有创建的部署产物都将放置在服务的.serverless目录。

9.6.2.5 信息

默认情况下，serverless info命令用于显示有关已部署服务的信息。

serverless info 命令

在服务目录中运行此命令即可显示已部署服务的信息。

```
serverless info
```

9.6.2.6 调用

serverless invoke 命令

调用已部署的函数。您可以发送事件数据、读取日志和查看函数调用的其他重要信息。

```
serverless invoke --function functionName
```

选项

- --function或-f：要调用的服务中函数的名称，**必填**。

- --data或-d：传递给函数的数据。
- --path或-p：JSON文件的路径，其中包含要传递给所调用函数的输入数据。此路径是相对于服务根目录的相对路径。

示例

- **简单的函数调用**
  ```
  serverless invoke --function functionName
  ```
  本示例将调用部署的函数，并在终端中输出调用的结果。
- **带数据的函数调用**
  ```
  serverless invoke --function functionName --data '{"name": "Bob"}'
  ```
  此示例将使用提供的数据调用函数，并在终端中输出调用的结果。
- **带传递数据的函数调用**
  ```
  serverless invoke --function functionName --path lib/event.json
  ```
  此示例将在调用指定/部署的函数时传递lib/event.json文件（相对于服务的根目录的相对路径）中的JSON数据。

  event.json示例：
  ```
  {
    "key": "value"
  }
  ```

9.6.2.7 日志

serverless logs 命令

查看特定函数的日志。
```
serverless logs --function functionName
```

选项

- --函数或-f：获取日志的函数，必填。
- --count或-c：显示的日志数。

示例

检索日志
```
serverless logs --function functionName
```

这将显示指定函数的日志。

9.6.2.8 移除

serverless remove 命令

serverless remove命令将从提供商中移除当前工作目录中定义的已部署服务。
```
serverless remove
```

说明

该命令将仅移除已部署的服务及其所有资源，本地计算机上的代码将会保留。

示例

服务移除

```
serverless remove
```

此示例将移除当前工作目录中已部署的服务。

9.6.3 事件列表

9.6.3.1 APIG 网关事件

华为云函数工作流可以通过API网关（APIG）创建基于函数的API终端节点。

要创建HTTP终端节点作为华为云函数工作流的事件源，请使用http事件语法。

HTTP 终端节点

此设置指定当有人通过GET请求访问函数API终端节点时，应运行first函数。您可以在部署服务后运行serverless info命令来获取终端节点的URL。

以下是一个例子：

```
# serverless.yml

functions:
  hello:
    handler: index.hello
    events:
      - apigw:
          env_id: DEFAULT_ENVIRONMENT_RELEASE_ID
          env_name: RELEASE
          req_method: GET
          path: /test
          name: API_test
```

说明

请参考有关**函数执行入口**的文档，了解用于此类事件的入口函数签名。

9.6.3.2 OBS 事件

华为云函数可以由不同的event源触发。这些事件源可以通过event定义和配置。

OBS 事件

此示例设置一个OBS事件，每当对象上传到my-service-resource时，该事件将触发first函数。

```
# serverless.yml

functions:
  first:
    handler: index.first
    events:
      - obs:
          bucket: bucket
          events:
            - s3:ObjectCreated:Put
            - s3:ObjectCreated:Post
// index.js

exports.first = async (event, context) => {
  const response = {
```



```
    statusCode: 200,
    body: JSON.stringify({
      message: 'Hello!',
    }),
  };

  return response;
};
```


10 自动化部署

10.1 部署环境准备

本章节以Linux主机为例，指导您基于KooCLI和**软件开发生产线CodeArts**搭建一套FunctionGraph函数的CI/CD。

云服务器 ECS

该服务器作为CodeArts部署任务的部署主机，用于部署更新FunctionGraph函数。

- 规格：1vCPUs | 1GiB
- 镜像：CentOS 8.2 64bit
- 其他：需要配置弹性公网IP，因为要安装python库和CodeArts，配置该服务器为部署主机。
- 注意：因为CodeArts配置该服务器为部署主机是通过SSH协议22端口，如果您对安全有较高的要求，至少需要将以下IP地址加入安全组并放开限制，否则将无法进行授信。

  42.202.130.147

  49.4.3.11

  122.112.212.206

  139.159.226.153

  49.4.85.127

  124.70.46.237

部署主机安全组配置流程

1. 进入**创建IP地址组页面**。
2. 根据界面提示设置IP地址组参数，具体参数详细说明请参见**创建IP地址组**，完成后单击“保存”。
   - 名称：ipGroup-clouddeploy
   - IP地址：

     ```
     42.202.130.147
     49.4.3.11
     122.112.212.206
     ```

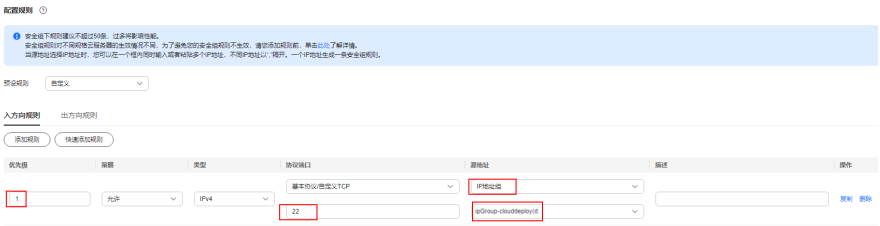


```
139.159.226.153
49.4.85.127
124.70.46.237
```

3. 返回网络控制台，在左侧导航栏选择“访问控制 > 安全组”，单击“创建安全组”，具体安全组配置详情请参见**创建安全组**，完成后单击“确定”。
   - 名称：functions-deploy
   - 企业项目：default
4. 在“入方向规则”页签下，单击“添加规则”，给functions-deploy安全组添加一个入方向规则。

   优先级为1，协议端口号为22，源地址选择已创建的ipGroup-clouddeploy的IP地址组，完成后单击“确定”。

图 10-1 添加入方向规则



5. 返回弹性云服务器页面，单击部署主机ECS的名称，将部署主机的安全组切换为functions-deploy安全组。

安装 Python 库

执行如下命令，安装pyyaml库和pycryptodome库。对函数的cam.yaml配置文件进行解析，对函数的加密环境变量进行加解密。

```
pip3 install pyyaml
pip3 install pycryptodome
```

安装 KooCLI 命令行工具

1. **安装KooCLI命令行工具**

   远程登录购买的ECS云服务器，执行如下命令安装KooCLI：

   ```
   curl -sSL https://hwcloudcli.obs.cn-north-1.myhuaweicloud.com/cli/latest/hcloud_install.sh -o ./hcloud_install.sh && bash ./hcloud_install.sh
   ```

图 10-2 安装命令行工具

```
[root@                    y ~]# curl -sSL https://hwcloudcli.obs.cn-north-1.myhuaweicloud.com/cli/latest/hcloud_install.sh -o ./hcloud_in
stall.sh && bash ./hcloud_install.sh
Download HCloud CLI to the default '/usr/local/hcloud/' directory? To change directory, enter 'n'. [Y/n] y
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 3489k  100 3489k    0     0  10.1M      0 --:--:-- --:--:-- --:--:-- 10.2M
hcloud
README.md
OpenSourceSoftwareNotice.md
Create symbolic link for HCloud CLI in '/usr/local/bin/'? [Y/n] y
Turn on autocomplete for HCloud CLI? [Y/n] y
HCloud CLI installed. Current version: 3.2.7
[root@                  oy ~]#
```

2. **初始化KooCLI命令行工具**

   使用如下命令初始化KooCLI命令行工具：

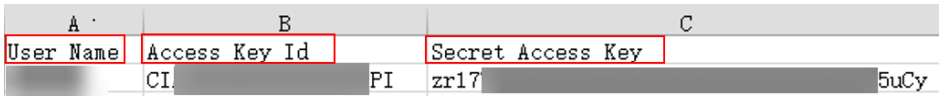
   ```
   hcloud configure init
   ```


需要输入Access Key ID、Secret Access key和Region Name，获取方法请参见3、4。

图 10-3 初始化 KooCLI 命令行工具

3. **获取访问密钥（Access Key ID和Secret Access key）**
   - 如果您有登录密码，可以登录控制台，可以在我的凭证中获取自己的访问密钥AK/SK。请参见：**新增访问密钥**。可以下载得到AK/SK文件，文件名一般为：credentials.csv。如下图所示，文件包含了用户名称（User Name），AK（Access Key Id），SK（Secret Access Key）。

     图 10-4 credentials.csv 文件内容

   - 如果您没有登录密码，不能登录控制台，在访问密钥异常丢失或者需要重置时，可以请账号管理员在IAM中生成您的访问密钥，并发送给您。请参见：**管理IAM用户访问密钥**。
4. **获取Region Name**

   请参见：**地区和终端节点**。

图 10-5 获取区域

10.2 使用 CodeArts 托管函数代码

10.2.1 步骤一：新建项目

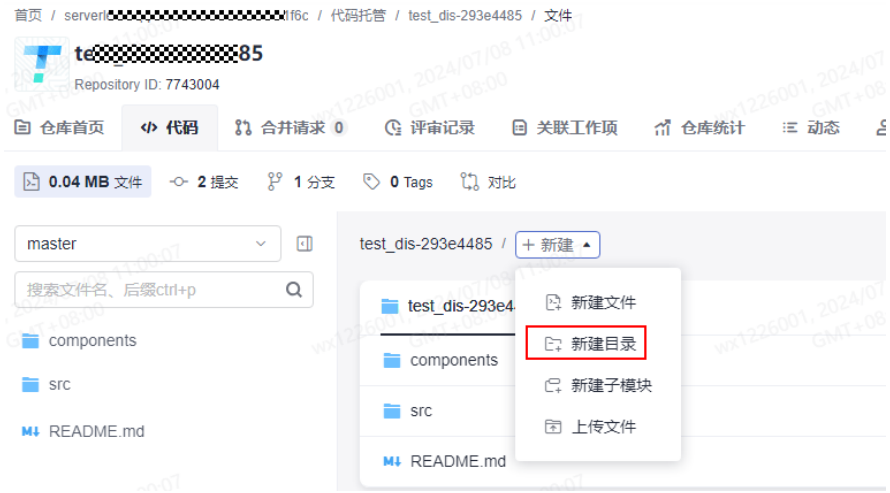
1. 登录**软件开发生产线CodeArts**控制台，进入CodeArts操作页面。
2. 单击“立即使用”，跳转至“新建项目”页面。
3. 单击“新建项目”，选择“项目模板 > Scrum”。

4. 输入项目名称“function”，其他配置保持默认。
5. 完成后单击“确定”。

10.2.2 步骤二：函数代码托管

1. 在CodeArts界面，在“代码 > 代码托管”页签下，单击“立即使用”。
   创建一个专属于函数的仓库，填写代码仓库名称“functions”，其他配置保持默认。
2. 进入1创建的functions仓库。先新建一个deploy目录，用于存放用户来部署函数的**deploy.py脚本**。

图 10-6 新建目录

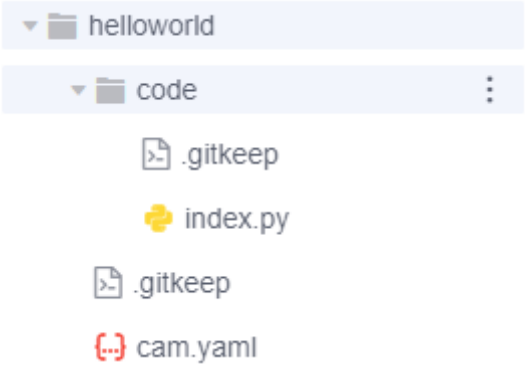


说明

执行deploy.py脚本时读取函数配置文件cam.yaml，构造hcloud命令更新函数代码和函数配置，cam.yaml详细配置请参见**cam.yaml解析**。执行deploy.py脚本日志会写入/home/function/deploy/function.log日志文件中。

3. 再创建一个helloworld目录，函数目录完整结构如下：

图 10-7 函数目录完整结构



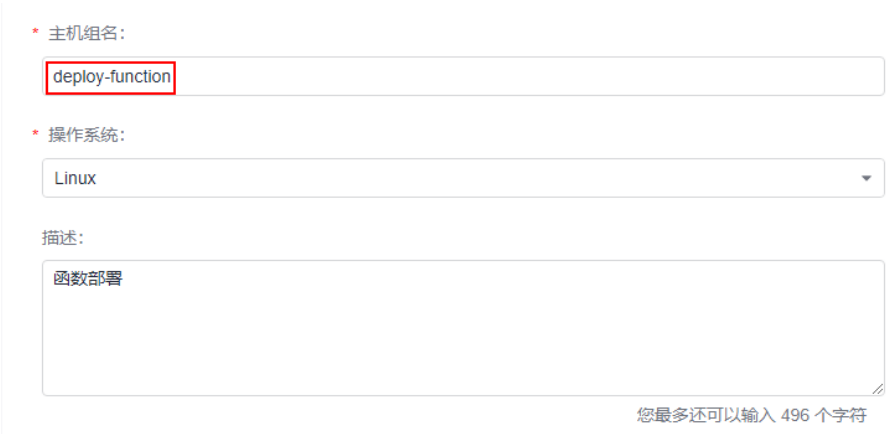
- helloworld：代表helloworld函数
- cam.yaml：函数配置文件

- code：函数代码目录，存放index.py函数代码

10.2.3 步骤三：配置部署主机

1. 在CodeArts界面，在“设置 > 通用设置”页签下，选择“主机组管理”，单击“新建主机组”。
2. 输入主机组名“deploy-function”，单击“保存”：

图 10-8 填写主机组名



3. 在跳转界面“主机信息”页签下，单击“导入ECS”。
   将**部署环境准备**的ECS云服务器导入，输入该服务器的用户名、密码、ssh端口号22、勾选《隐私政策声明》，完成后单击“添加”。

图 10-9 导入 ECS

4. “**连通性验证**”显示“验证成功”。

图 10-10 连通性验证成功

10.2.4 步骤四：搭建函数部署脚本更新流水线

此流水线的主要作用是将函数部署脚本deploy.py发布到部署主机上，供函数更新流水线使用。

新建构建任务

1. 在“构建&制品 > 编译构建”页面，单击“新建任务”。
2. 源码仓库选择“functions仓库”，构建模板选择“空白构建模板”，完成单击“确定”。
3. 构建步骤，只添加“上传软件包到软件发布库”，添加成功后将在左侧区域将展示“上传软件包到软件发布库”。
4. 在左侧区域单击“上传软件包到软件发布库”，进行相关配置。

图 10-11 配置参数

- 步骤显示名称：上传deploy.py到发布库
- 构建包路径：deploy/deploy.py
- 发布版本号：${releaseVersion}

5. 在“参数设置”中配置releaseVersion，开启“运行时设置”。

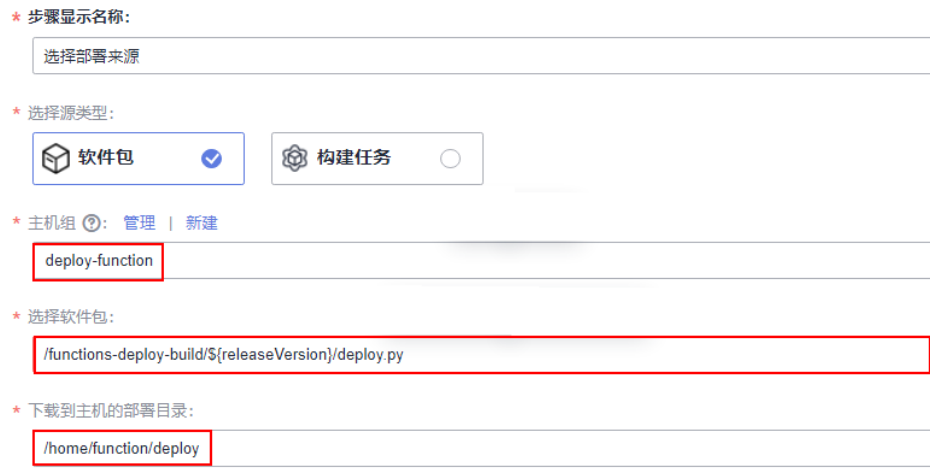
图 10-12 版本号配置

6. 在“基础信息”页签下，更新任务名称为functions-deploy-build，并单击“保存”。

新建部署任务

1. 在“构建&发布 > 部署”页面，单击“新建任务”。
2. “部署模板”选择“空白模板”，单击“下一步”。
3. 添加步骤，只添加“选择部署来源”。
4. 对选择部署来源进行配置。

图 10-13 配置部署来源



- 主机组：选择主机组deploy-function
- 选择软件包：输入/functions-deploy-build/${releaseVersion}/deploy.py
- 下载到主机的部署目录为：/home/function/deploy

5. 在“参数设置”中配置releaseVersion，开启“运行时设置”。

图 10-14 参数设置

6. 在“基础信息”页签下，更新该构建任务名称为update-function-deploy，并单击“保存”。

配置流水线

1. 在“构建&发布 > 流水线”页面，单击“新建流水线”。
2. 选择创建的functions仓库，构建模板选择“空白构建模板”。
3. 配置“构建和检查”。
   a. 添加构建任务，类型为构建，选择要添加的任务function-deploy-build任务。

图 10-15 添加任务

b. 其中releaseVersion设置为流水线参数。

图 10-16 releaseVersion 参数设置

c. 单击“保存”，保存构建任务。

4. 配置部署任务

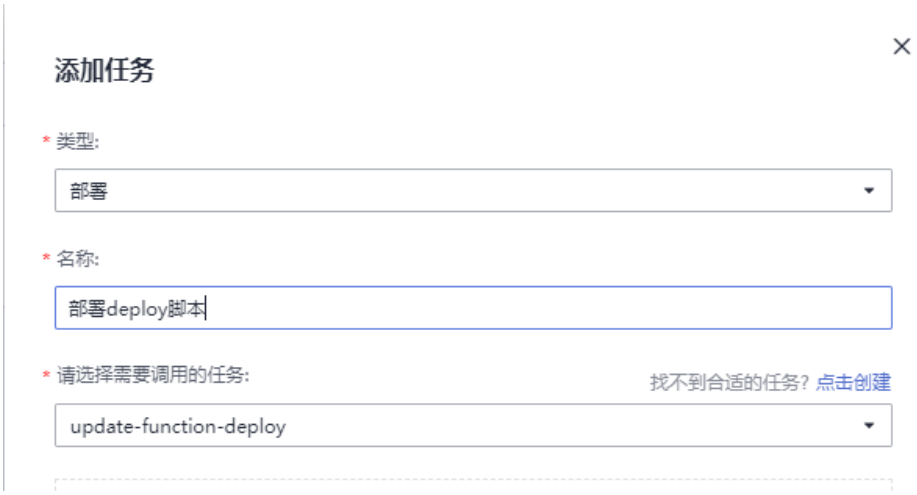
a. 在构建阶段后新建一个阶段，名称为部署，任务串行执行，完成后单击“保存”。

图 10-17 阶段配置

b. 单击“添加任务”，添加一个类型为部署的任务，输入名称为部署deploy脚本，选择需要调用的任务update-function-deploy。

图 10-18 添加任务

图 10-19 配置添加任务



其中releaseVersion设置为流水线参数。

图 10-20 releaseVersion 参数设置

c. 单击“保存”，保存部署任务。

5. 在“基础信息”页签下，更新流水线名称为pipeline-update-function-deploy，并单击“保存”流水线。
6. 执行流水线
   a. 运行时参数配置releaseVersion为1.0.0，单击“执行”。

图 10-21 运行时参数配置

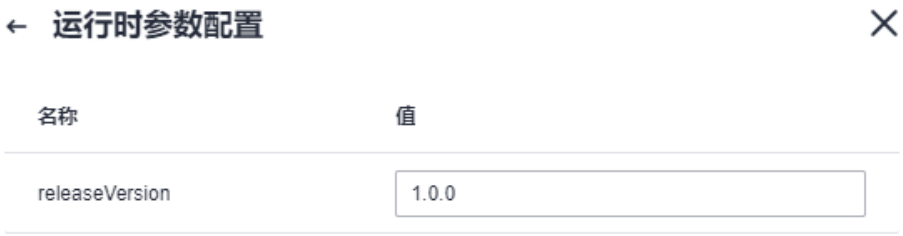
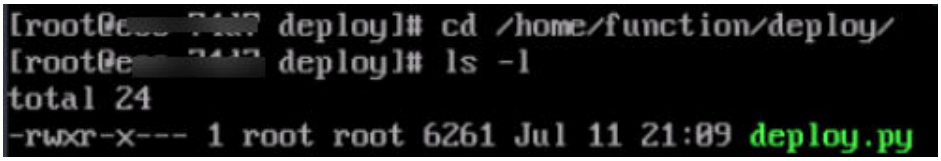


图 10-22 流水线

b. deploy脚本发布成功。

图 10-23 执行成功



10.2.5 步骤五：搭建函数更新流水线

此流水线的主要作用是将functions仓库的helloworld函数代码配置发布更新到FunctionGraph平台。

新建构建任务

1. 在“构建&发布 > 编译构建”页面，单击“新建任务”。
2. 源码仓库选择functions仓库，构建模板选择“空白构建模板”。
3. 构建步骤，添加三个构建步骤“执行shell命令”、“上传文件到obs”和“上传软件包到软件发布库”。

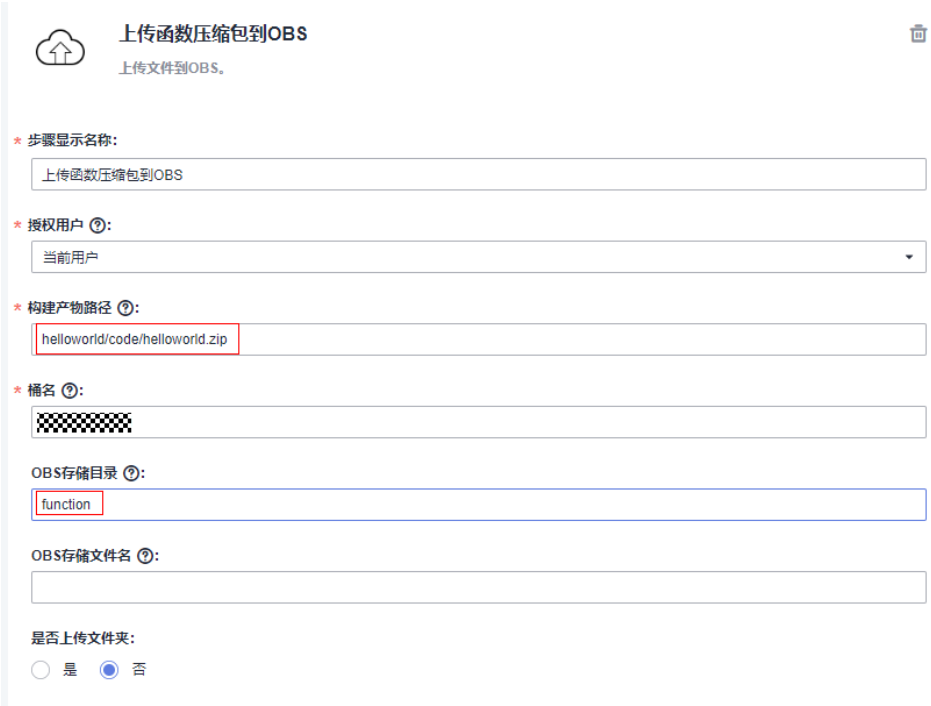
   a. 执行shell命令

```
# 构建函数部署包
cd helloworld
zip helloworld_deploy.zip cam.yaml
# 构建函数代码压缩包
cd code
zip -rp helloworld.zip *
```


图 10-24 执行 shell 命令

b. 上传函数压缩包到OBS

图 10-25 上传函数压缩包到 OBS



- 步骤显示名称：上传函数压缩包到OBS
- 构建产物路径：输入helloworld/code/helloworld.zip
- 桶名：选择一个私有桶存储函数代码zip包
- OBS存储目录：function

c. 上传部署包到软件发布库

图 10-26 上传部署包到软件发布库

- 步骤显示名称：上传部署包到软件发布库
- 构建包路径：helloworld/helloworld_deploy.zip
- 发布版本号：${releaseVersion}

4. 在“参数设置”中配置releaseVersion，开启“运行时设置”。

图 10-27 参数设置

5. 在“基础信息”页签下，更新任务名称为pipeline-update-function-deploy，并单击“保存”。

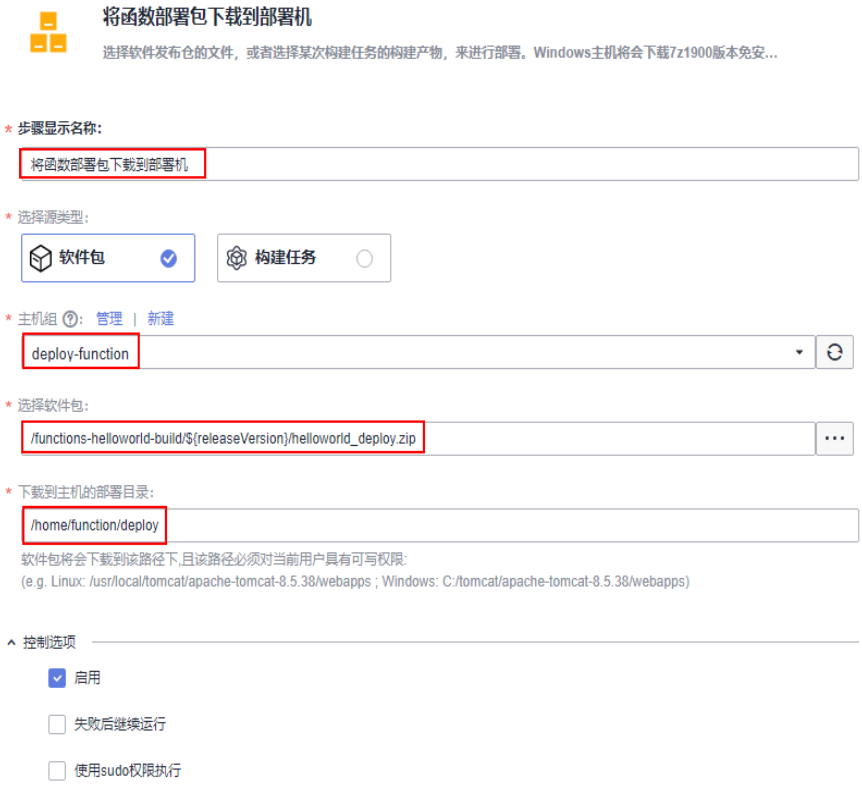
新建部署任务

1. 在“构建&发布 > 部署”页面，单击“新建任务”。
2. “部署模板”选择“空白模板”，单击“下一步”
3. 部署步骤。选择添加步骤“选择部署来源”和“执行shell命令”。

图 10-28 添加部署步骤

a. 选择部署来源

图 10-29 将函数部署包下载到部署机

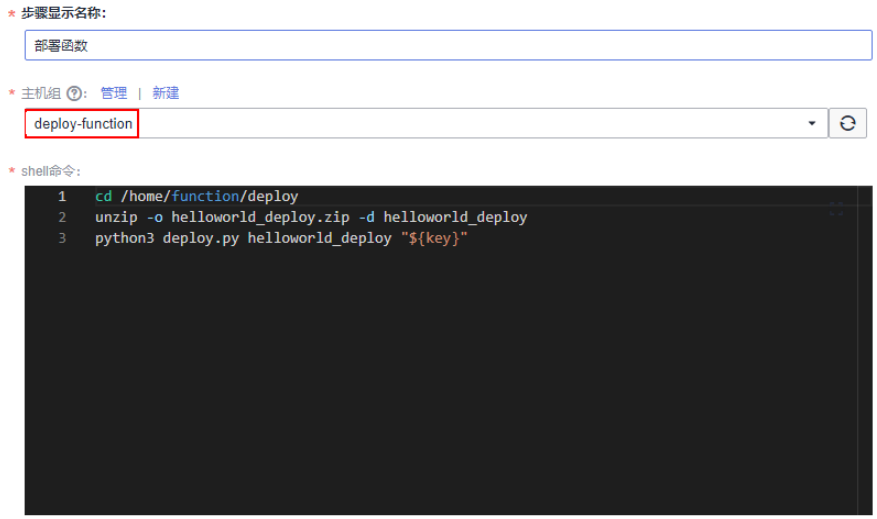


- 步骤显示名称：将函数部署包下载到部署机
- 主机组：deploy-function
- 选择软件包：/functions-helloworld-build/${releaseVersion}/helloworld_deploy.zip

- 下载到主机的部署目录：/home/function/deploy

b. 执行shell命令

图 10-30 部署函数

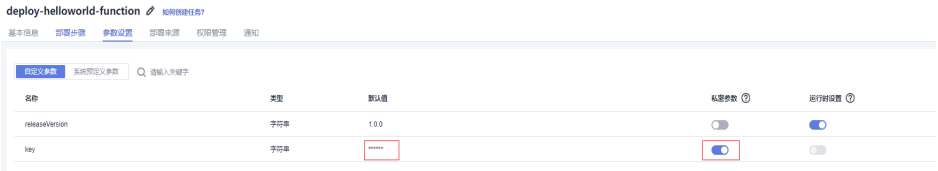


- 步骤显示名称：部署函数
- 主机组：选择deploy-function
- shell命令：

```
cd /home/function/deploy
unzip -o helloworld_deploy.zip -d helloworld_deploy
python3 deploy.py helloworld_deploy "${key}"
```

4. 添加两个“参数设置”。
   - releaseVersion：默认值为1.0.0，开启“运行时设置”。
   - key：默认值输入密码，开启“私密参数”。

图 10-31 参数设置



5. 在“基础信息”页签下，更新任务名称为update-function-deploy，并单击“保存”。

配置流水线

1. 在“构建&发布 > 流水线”页面，单击“新建流水线”。
2. 源码仓库选择functions仓库，构建模板选择“空白构建模板”。
3. 配置“构建和检查”。
   a. 添加构建任务，类型为构建，选择要添加的任务functions-helloworld-build任务。

图 10-32 添加任务

b. 其中releaseVersion设置为流水线参数。

图 10-33 releaseVersion 参数设置

c. 单击“保存”，保存构建任务。

4. 配置部署任务

a. 在构建阶段后新建一个阶段，名称为部署，任务串行执行，完成后单击“保存”。

图 10-34 阶段配置

b. 单击“添加任务”，添加一个类型为部署的任务，输入名称为部署helloworld脚本，选择需要调用的任务update-function-deploy。

图 10-35 添加任务

c. 其中releaseVersion设置为流水线参数。

图 10-36 releaseVersion 参数设置

d. 单击“保存”，保存部署任务。

5. 在“基础信息”页签下，更新流水线名称为pipeline-update-function-helloworld，单击“保存”。
6. 执行流水线。

   运行时参数配置releaseVersion输入1.0.0，单击“执行”。

图 10-37 运行时参数配置

10.3 deploy.py 代码示例

```
# -*-coding:utf-8 -*-

import os
import sys
import json
import logging
import subprocess
from yaml import load
from base64 import b64decode
from Crypto.Cipher import AES

# need: pip install pyyaml
try:
    from yaml import CLoader as Loader, CDumper as Dumper
except ImportError:
    from yaml import Loader, Dumper

logging.basicConfig(level=logging.INFO,
                    filename='function.log',
                    filemode='a',
                    format='%(asctime)s - %(pathname)s[line:%(lineno)d] - %(levelname)s: %(message)s')


def decrypt(json_input, key):
    # We assume that the key was securely shared beforehand
    try:
        b64 = json.loads(json_input)
        json_k = ['nonce', 'header', 'ciphertext', 'tag']
        jv = {k: b64decode(b64[k]) for k in json_k}
        cipher = AES.new(key.encode(), AES.MODE_GCM, nonce=jv['nonce'])
        cipher.update(jv['header'])
        plaintext = cipher.decrypt_and_verify(jv['ciphertext'], jv['tag'])
        return plaintext.decode()
    except (ValueError, KeyError) as e:
        raise e


def generate_update_function_config_cmd(new_config, old_config, key):
    # 函数执行入口
    handler = new_config['handler']
    # 函数runtime配置（必填但不支持修改）
    runtime = new_config['runtime']
    # 函数内存规格配置
    memory_size = new_config['memorySize']
    # 函数执行超时配置
    timeout = new_config['timeout']
    # 函数所属project_id
    project_id = new_config['projectID']
    # 拼装更新函数配置命令
    update_cmd = f'hcloud FunctionGraph UpdateFunctionConfig' \
```



```
            f' --cli-region="{region}"' \
            f' --function_urn="{function_urn}"' \
            f' --project_id="{project_id}"' \
            f' --handler="{handler}"' \
            f' --timeout={timeout}' \
            f' --memory_size={memory_size}' \
            f' --runtime="{runtime}"' \
            f' --func_name="{function_name}"'

# 用户环境变量配置
# 更新用户环境变量为直接覆盖，如果有手动在函数界面配置环境变量没有更新到cam.yaml文件内
# 则手动添加环境变量配置则丢失
user_data = new_config.get('userData', None)
if user_data is not None:
    user_date_json_str = json.dumps(user_data)
    user_date_json_str = json.dumps(user_date_json_str)
    update_cmd = update_cmd + f' --user_data={user_date_json_str}'

encrypted_user_data = new_config.get('encryptedUserData', None)
if encrypted_user_data is not None:
    encrypted_user_data = decrypt(encrypted_user_data, key)
    encrypted_user_date_json_str = json.dumps(encrypted_user_data)
    update_cmd = update_cmd + \
            f' --encrypted_user_data={encrypted_user_date_json_str}'

# 如果有vpc则保留
vpc_config = old_config.get('func_vpc', None)
if vpc_config is not None:
    update_cmd = update_cmd + \
            f' --func_vpc.vpc_name={vpc_config["vpc_name"]}' \
            f' --func_vpc.vpc_id={vpc_config["vpc_id"]}' \
            f' --func_vpc.subnet_id={vpc_config["subnet_id"]}' \
            f' --func_vpc.cidr={vpc_config["cidr"]}' \
            f' --func_vpc.subnet_name={vpc_config["subnet_name"]}' \
            f' --func_vpc.gateway={vpc_config["gateway"]}'

# 如果有委托配置则保留 "xrole": "function-admin"和"app_xrole": "function-admin",
xrole_config = old_config.get('xrole', None)
if xrole_config is not None:
    update_cmd = update_cmd + f' --xrole="{xrole_config}"'

app_xrole_config = old_config.get('app_xrole', None)
if app_xrole_config is not None:
    update_cmd = update_cmd + f' --app_xrole="{app_xrole_config}"'

# 配置初始化入口和初始化超时时间
initializer_handler = new_config.get('initializerHandler', None)
initializer_timeout = new_config.get('initializerTimeout', None)
if initializer_handler is not None and initializer_timeout is not None:
    update_cmd = update_cmd + \
            f' --initializer_handler="{initializer_handler}" ' \
            f'--initializer_timeout={initializer_timeout}'

# 并发配置
strategy_config = new_config.get('strategyConfig', None)
if strategy_config is not None:
    concurrency = strategy_config.get('concurrency', None)
    # 单实例并发数
    concurrent_num = strategy_config.get('concurrentNum', None)
    update_cmd = update_cmd + \
            f' --strategy_config.concurrency="{concurrency}" ' \
            f'--strategy_config.concurrent_num={concurrent_num}'

# 如果有磁盘挂载则保留
mount_config = old_config.get('mount_config', None)
if mount_config is not None:
    mount_user = mount_config["mount_user"]
    update_cmd = update_cmd + \
            f' --mount_config.mount_user.user_id={mount_user["user_id"]}' \
```



```
                f' --mount_config.mount_user.user_group_id={mount_user["user_group_id"]}'
        func_mounts = mount_config["func_mounts"]
        i = 1
        for func_mount in func_mounts:
            update_cmd = update_cmd + \
                    f' --mount_config.func_mounts.{i}.mount_resource="{func_mount["mount_resource"]}"' \
                    f' --mount_config.func_mounts.
{i}.mount_share_path="{func_mount["mount_share_path"]}"' \
                    f' --mount_config.func_mounts.{i}.mount_type="{func_mount["mount_type"]}"' \
                    f' --mount_config.func_mounts.{i}.local_mount_path="{func_mount["local_mount_path"]}"'
            i = i + 1

    return update_cmd


def exec_cmd(cmd):
    proc = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE,
                            stderr=subprocess.STDOUT)
    outs, _ = proc.communicate()
    return outs.decode('UTF-8')


def check_result(stage, exec_result):
    if "USE_ERROR" in exec_result:
        error_info = f"failed to {stage}: {exec_result}"
        logging.error(error_info)
        raise Exception(error_info)

    if "FSS.0409" in exec_result:
        error_info = f"failed to {stage}: {exec_result}"
        logging.error(error_info)
        # 返回错误为函数代码没变不需要更新错误则返回
        return

    try:
        result_object = json.loads(exec_result)
    except Exception:
        error_info = f"failed to {stage}: {exec_result}"
        logging.error(error_info)
        raise Exception(error_info)

    if "error_code" in result_object:
        error_message = result_object["error_msg"]
        error_info = f"failed to {stage}: {error_message}"
        logging.error(error_info)
        raise Exception(error_info)


def generate_update_function_code_cmd():
    cmd = \
        f'hcloud FunctionGraph UpdateFunctionCode --cli-region="{region}"' \
        f' --function_urn="{function_urn}" --project_id="{project_id}"' \
        f' --code_url="{code_url}" --func_code.link="" --func_code.file="" --code_type="obs" '

    depend_list = old_function_code.get("depend_list", None)
    if depend_list is not None and len(depend_list) > 0:
        i = 1
        for depend_id in depend_list:
            cmd = cmd + f'--depend_list.{i}="{depend_id}"'

    return cmd


if __name__ == '__main__':
    deploy_function_path = sys.argv[1]
    key = sys.argv[2]
    f = open(os.path.join(deploy_function_path, "cam.yaml"))
    data = load(f, Loader=Loader)
    function_config = data['components'][0]
```



```
    function_name = function_config['name']
    function_properties = function_config['properties']
    region = function_properties['region']
    code_url = function_properties['codeUri']
    project_id = function_properties['projectID']
    # 拼接获取函数urn
    function_urn = "urn:fss:" + region + ":" + project_id + \
                   ":function:default:" + function_name + ":latest"
    logging.info(f"start to deploy functionURN:{function_urn}")

    # 查询函数的配置信息
    query_function_config_cmd = \
        f'hcloud FunctionGraph ShowFunctionConfig --cli-region="{region}"' \
        f' --function_urn="{function_urn}" --project_id="{project_id}"'
    result = exec_cmd(query_function_config_cmd)
    # 主要是查看函数是否有配置VPC和委托，如果有更新函数配置时需要带上，避免更新导VPC或委托配置丢失
    old_function_config = json.loads(result)
    check_result("query function config", result)

    # 查询函数代码，主要是函数绑定依赖包信息保留
    query_function_code_cmd = \
        f'hcloud FunctionGraph ShowFunctionCode --cli-region="{region}"' \
        f' --function_urn="{function_urn}" --project_id="{project_id}"'
    result = exec_cmd(query_function_code_cmd)
    old_function_code = json.loads(result)
    logging.info("query function %s code result: %s", function_urn, result)
    check_result("query function code", result)

    # 更新函数代码
    query_function_code_cmd = generate_update_function_code_cmd()
    result = exec_cmd(query_function_code_cmd)
    logging.info("update function %s code result: %s", function_urn, result)
    check_result("update function code", result)

    # 更新函数配置
    update_function_config_cmd = generate_update_function_config_cmd(
        function_properties, old_function_config, key)
    result = exec_cmd(update_function_config_cmd)
    logging.info("update function %s config result: %s", function_urn, result)
    check_result("update function config", result)

    logging.info(f"succeed to deploy function {function_urn}")
```

10.4 cam.yaml 解析

示例

```
metadata:
  description: This is an example application for FunctionGraph.
  author: Serverless team
  homePageUrl: https://www.huaweicloud.com/product/functiongraph.html
  version: 1.0.0
components:
  - name: helloworld
    type: Huawei::FunctionGraph::Function
    properties:
      region: cn-east-4
      codeUri: https://test-wkx.obs.cn-north-4.myhuaweicloud.com/helloworld.zip
      projectID: 0531e14952000f742f3ec0088c4b25cf
      handler: index.handler
      runtime: Python3.9
      memorySize: 256
      timeout: 60
      userData:
        key1: value1
        key2: value2
      encryptedUserData: '{"nonce": "ZEUOREFaiahRbMz+K9xQwA==", "header": "aGVhZGVy", "ciphertext":
```

```
"SCxXsffvpU1BF2Ci8a2RedNQ", "tag": "a+EYRVPOsQ+YpQkMuFg1wA=="}'
    initializerTimeout: 30
    initializerHandler: index.init_handler
    strategyConfig:
      concurrency: 80
      concurrentNum: 20
```

详解

函数配置在cam.yaml的properties属性下，当前支持的函数配置详解如下：

参数	是否必须	是否更新	描述
region	是	否	调用函数所在region。
codeUri	是	否	函数代码地址。该值为函数代码包在OBS上的地址。
projectID	是	否	租户Project ID。
handler	是	是	函数执行入口。
runtime	是	否	FunctionGraph函数的执行环境支持Node.js6.10、Python2.7、Python3.6、PHP7.3、Java8、Node.js 8.10、C#.NET Core 2.0、C#.NET Core 2.1。 Python2.7: Python语言2.7版本。 Python3.6: Python语言3.6版本。 PHP7.3: Php语言7.3版本。 Java8: Java语言8版本。 Node.js6.10: Nodejs语言6.10版本。 Node.js8.10: Nodejs语言8.10版本。 C#(.NET Core 2.0): C#语言2.0版本。 C#(.NET Core 2.1): C#语言2.1版本。 C#(.NET Core 3.1): C#语言3.1版本。 Custom: 自定义运行时。
memorySize	是	是	函数内存，单位M。 枚举值： 128、256、512、768、1024、1280、1536、1792、2048、2560、3072、3584、4096
timeout	是	是	函数运行超时时间，单位秒，范围3～900秒。
userData	否	是	用户自定义的name/value信息，在函数中使用的参数。
encryptedUserData	否	是	用户自定义的name/value信息，用于需要加密的配置。

参数	是否必须	是否更新	描述
initializerTimeout	否	是	初始化超时时间,超时函数将被强行停止,范围1~300秒。
initializerHandler	否	是	函数初始化入口,规则:xx.xx,必须包含“.”。举例:对于node.js函数:myfunction.initializer,则表示函数的文件名为myfunction.js,初始化的入口函数名为initializer。
concurrentNum	否	是	函数单实例并发数。
concurrency	否	是	单函数最大实例数，0禁用函数，-1无限制，例100，该函数最大实例数100（普通实例+预留实例）。

说明

1. 当前cam.yaml不支持**VPC、委托、磁盘挂载和动态内存**配置的更新，如果函数需要使用**VPC、委托或者磁盘挂载和动态内存**请在函数界面手动配置，在使用函数更新流水线时会保留**VPC、委托、磁盘挂载和动态内存**配置，不会覆盖掉。
2. 为了避免在cam.yaml中明文显示函数的加密配置-**encryptedUserData**，该CICD使用了AES对称加密的GCM模式对**encryptedUserData**明文内容进行加密，加密输出配置为cam.yaml中**encryptedUserData**项对应的值。在functions仓库和“函数更新流水线”中encryptedUserData的值以密文传输，在最后部署更新函数时解密更新，所以在执行“函数更新流水线”时需提供AES加密时使用的Key。示例如下：

   encryptedUserData明文：

   ```
   '{"password":"123"}'
   ```

   使用AES-GCM加密后：

   ```
   {"nonce": "ZEUOREFaiahRbMz+K9xQwA==", "header": "aGVhZGVy", "ciphertext": "SCxXsffvpU1BF2Ci8a2RedNQ", "tag": "a+EYRVPOsQ+YpQkMuFg1wA=="}，其中ciphertext为加密后的密文。
   ```

AES加密使用的Key需妥善保管。

Python AES-GCM使用示例：**https://pycryptodome.readthedocs.io/en/latest/src/cipher/modern.html?highlight=GCM#gcm-mode**

AES-GCM加密脚本如下：

```
import json
from base64 import b64encode
from Crypto.Cipher import AES
import sys

if __name__ == '__main__':
    key = sys.argv[1].encode()
    data = sys.argv[2].encode()
    header = b"header"
    cipher = AES.new(key, AES.MODE_GCM)
    cipher.update(header)
    ciphertext, tag = cipher.encrypt_and_digest(data)
    json_k = ['nonce', 'header', 'ciphertext', 'tag']
    json_v = [b64encode(x).decode('utf-8') for x in
              [cipher.nonce, header, ciphertext, tag]]
```



```
    result = json.dumps(dict(zip(json_k, json_v)))
    print(result)
```

使用方式为在ECS云服务器上执行如下命令：

python3 aes_gcm_encrypt_tool.py "16个字节的key" '{"password":"123"}'，在ECS云服务器上执行。

