FunctionGraph

Developer Guide

Issue 01

Date 2026-01-04

HUAWEI

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Trademarks and Permissions


and other Huawei trademarks are the property of Huawei Technologies Co., Ltd.
All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei Cloud and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Huawei Cloud Computing Technologies Co., Ltd.

Address: Huawei Cloud Data Center Jiaoxinggong Road
Qianzhong Avenue
Gui'an New District
Gui Zhou 550029
People's Republic of China

Website: https://www.huaweicloud.com/intl/en-us/

Contents

1 Function Development Overview

1.1 Function Runtimes

Runtime

To create a function in FunctionGraph, you need to specify a runtime that passes events, context, and responses. Choose from a built-in runtime or customize your own.

Supported Runtimes

The Node.js, Java, Python, Go, C#, PHP, Cangjie, and custom runtimes are supported. **Table 1-1** lists the supported runtimes.

Table 1-1 Runtime description

Runtime	Supported Version	SDK	Development Overview
Node.js	6.10, 8.10, 10.16, 12.13, 14.18, 16.17, 18.15, 20.15	-	**Function Development Overview**
Python	2.7, 3.6, 3.9, 3.10, 3.12	-	**Function Development Overview**
Java	8, 11, 17, 21 (only available in **ME-Riyadh** and **TR-Istanbul**)	**Java SDK** (software package verification file: **fss-java-sdk_sha256**)	**Function Development Overview**

Runtime	Supported Version	SDK	Development Overview
C#	.NET Core 2.1, .NET Core 3.1, .NET Core 6.0, .NET Core 8.0 (only in **ME-Riyadh** and **TR-Istanbul**)	**C# SDK** (software package verification file: **fssCsharp_sha256**)	**C# Function Development Overview**
Go	1.x	**Go 1.x SDK** (software package verification file: **Go SDK_sha256**)	**Function Development Overview**
PHP	7.3 and 8.3	-	**PHP Function Development Overview**
Cangjie	1.0	-	-
Custom	-	-	-

Runtime Deprecation

For security and sustainable development, FunctionGraph will no longer provide technical support and security updates for some runtimes.

Table 1-2 shows the deprecation policy, which is divided into three stages.

Table 1-2 Deprecation stage description

Stage	Description
Stage 1: Notify customers 180 days in advance	You will be notified of the runtime deprecation through product notices, runtime deprecation marks, and emails by Huawei Cloud.
Stage 2: Deprecate runtimes	Security patches, feature updates, or technical support for these runtimes will no longer be provided by Huawei Cloud. Functions cannot be created using these runtimes. Existing functions can continue to update their code or configurations and run.

Stage	Description
Stage 3: 30 days after runtime deprecation	Security patches, feature updates, or technical support for these runtimes will no longer be provided by Huawei Cloud. Functions cannot be created or updated using these runtimes. Existing functions can continue running. You are advised to migrate your functions to the latest supported runtimes for technical support and security updates.

Table 1-3 is our runtime deprecation plan. Runtimes not included in this table are not subject to this deprecation plan.

Table 1-3 Runtime deprecation plan

Runtime	Stage 1	Stage 2	Stage 3
Node.js 6.10	December 15, 2025	June 15, 2026	July 15, 2026
Node.js 8.10	December 15, 2025	June 15, 2026	July 15, 2026
Python 2.7	December 15, 2025	June 15, 2026	July 15, 2026

Sample Project Packages

Table 1-4 provides the links for downloading the sample project packages mentioned in this document. You can download the project packages to a local path and upload them when creating functions.

For the operation process of each runtime, see the event function development sections.

Table 1-4 Download links of the sample project packages

Function	Project Package	Software Package Verification File
Node.js function	fss_examples_nodejs.zip	fss_examples_nodejs.sha256
Python function	fss_examples_python3.zip	fss_examples_python3_sha256
Java function	fss_example_java17.jar demo-with-dependencies_java17.jar (maven project)	fss_example_java_sha256 demo-with-dependencies_java17.jar.sha256

Function	Project Package	Software Package Verification File
PHP function	fss_examples_php7.3.zip	fss_examples_php7.3_sha256

Helpful Links

- For details about the trigger events supported by each runtime, see **Supported Trigger Events for FunctionGraph**.
- For details about the packaging specifications of each runtime project, see **Function Project Packaging Rules**.

1.2 Initializer

Overview

An initializer is a logic entry for initializing functions. For a function with an initializer, FunctionGraph invokes the initializer to initialize the function and then invokes the handler to process function requests. For a function without an initializer, FunctionGraph only invokes the handler to process function requests.

Applicable Scenario

FunctionGraph executes a function in the following steps:

1. Allocate container resources to the function.
2. Download function code.
3. Use the runtime to load the function code.
4. Initialize the function.
5. Process the function request and return the result.

Steps **1**, **2**, and **3** are performed during a systematic cold start, ensuring a stable latency through proper resource scheduling and process optimization. Step **4** is performed during an application-layer cold start in complex scenarios, such as loading large models for deep learning, building database connection pools, and loading function dependencies.

To reduce the latency caused by an application-layer cold start, FunctionGraph provides the initializer to identify function initialization logic for proper resource scheduling.

Benefits of the Initializer

- Isolate function initialization and request processing to enable clearer program logic and better structured and higher-performance code.
- Ensure a smooth function upgrade to prevent performance loss during the application layer's cold start initialization. Enable new function instances to automatically execute initialization logic before processing requests.

- Identify the overhead of application layer initialization, and accurately determine the time for resource scaling and the quantity of required resources. This feature makes request latency more stable when the application load increases and more function instances are required.
- If there are continuous requests and the function is not updated, the system may still reclaim or update existing containers. Although no code starts on the platform side, there are cold starts on the service side. The initializer can be used to ensure that requests can be processed properly.

Features of the Initializer

The initializer of each runtime has the following features:

- No custom parameters

  The initializer does not support custom parameters and only uses the variables in **context** for logic processing.
- No return values

  No values will be returned for initializer invocation.
- Initialization timeout

  You can set an initialization timeout (≤ 300s) different from the timeout for invoking the handler.
- Execution duration

  Function instances are processes that execute function logic in a container and automatically scale if the number of requests changes. When a new function instance is generated, the system invokes the initializer and then executes the handler logic if the invocation is successful.
- One-time execution

  After each function instance starts, the initializer can only be executed once. If an instance fails to execute the initializer, the instance is abandoned and another instance starts to execute the initializer. A maximum of three attempts are allowed. If the initializer is executed successfully, the instance will only process requests upon invocation and will no longer execute the initializer again within its lifecycle.
- Naming rule

  For all runtimes except Java, the initializer can be named in the format of *[File name].[Initializer name]*, which is similar with the format of a handler name. For Java, a class needs to be defined to implement the predefined initializer.
- Billing

  The initializer execution duration will be billed at the same rate as the function execution duration.

1.3 Supported Trigger Events for FunctionGraph

Supported Trigger Events

Table 1-5 lists the supported cloud services and trigger events for FunctionGraph. After an event source trigger is configured, the corresponding function is automatically invoked when an event is detected.

Table 1-5 Cloud service events supported by FunctionGraph

Cloud Service/ Feature	Trigger Event
Timer	You can schedule a timer (**timer example event**) to invoke function code based on a fixed rate of minutes, hours, or days or a cron expression. For details, see **Using a Timer Trigger**.
API Gateway (APIG)	FunctionGraph functions are invoked over HTTP or HTTPS by defining REST APIs and endpoints on APIG. You can map each API operation (such as, GET and PUT) to a specific function. APIG invokes the relevant function when an HTTPS request (**APIG example event**) is sent to the API endpoint. For details, see **Using an APIG (Dedicated) Trigger**.
ROMA Connect (APIC)	With API operations such as GET and PUT mapped to specific functions, APIC can invoke corresponding functions to perform operations when receiving relevant HTTPS or HTTP requests. For details, see **Using an APIC Trigger**.
Data Ingestion Service (DIS)	DIS can ingest large amounts of data in real time. You can create a function to automatically poll a DIS stream and process all new data records, such as website click streams, financial transactions, social media streams, IT logs, and data tracking events (**DIS example event**). FunctionGraph periodically polls the stream for new data records. For details, see **Using a DIS Trigger**.
Distributed Message Service (DMS) for Kafka	When a message is created in a Kafka topic, FunctionGraph consumes the message and triggers the function to perform other operations (**Kafka example event)**). For details about how to use Kafka triggers, see: • **Using a Kafka Trigger** • **Using an Open-Source Kafka Trigger**

Cloud Service/ Feature	Trigger Event
Distributed Message Service (DMS) for RabbitMQ	When a DMS for RabbitMQ trigger is used, FunctionGraph periodically polls for new messages in a specific topic bound to the exchange of a RabbitMQ instance and passes the messages as input parameters to invoke functions (**RabbitMQ example event**). For details, see **Using a DMS (for RabbitMQ) Trigger**.
GeminiDB MongoDB	If you create a GeminiDB Mongo trigger for a function, any updates **(GeminiDB MongoDB example event)** to the specified database table will trigger the function. For details, see **Using a GeminiDB Mongo Trigger**.
IoT Device Access (IoTDA)	FunctionGraph can use IoTDA trigger to track device properties, message reporting, and status changes as well as analyze, sort out, and measure data flows (**IoTDA example event**). For details, see **Using an IoTDA Trigger**.
Simple Message Notification (SMN)	Simple Message Notification (SMN) sends messages to email addresses, mobile phones, or HTTP/HTTPS URLs. If you create a function with an SMN trigger, messages published to a specified topic will be passed as a parameter (**SMN example event**) to invoke the function. Then, the function processes the event, for example, publishing messages to other SMN topics or sending them to other cloud services. For details, see **Using an SMN Trigger**.
Object Storage Service (OBS)	Create a function for processing OBS events, such as object creation or deletion (**OBS example events**). When an image is uploaded to a specified bucket, OBS invokes the function to read the image and create a thumbnail. For details, see: • **Using an OBS Trigger**.
EventGrid (EG)	EventGrid (EG) receives messages from event sources and passes the message payload as a parameter (**EG example event**) to invoke a function. Then, the function processes the events, for example, sending messages to other cloud services. EventGrid supports the following event sources: • **Creating an EG Trigger (OBS Application Service)** • **Creating an EG Trigger (RocketMQ custom event source)**

Trigger Event Examples

Timer

- TIMER example event. For details about the parameters, see **Table 1-6**.

```
{
    "version": "v2.0",
    "time": "2023-06-01T08:30:00+08:00",
    "trigger_type": "TIMER",
    "trigger_name": "Timer_001",
    "user_event": "User Event"
}
```

Table 1-6 Parameters of a timer example event

Parameter	Type	Example Value	Description
version	String	v2.0	Event version
time	String	2023-06-01T08:30:00+08:00	Time when an event occurs
trigger_type	String	TIMER	Trigger type
trigger_name	String	Timer_001	Trigger name
user_event	String	User Event	Additional information of the trigger

API Gateway (dedicated)

- API Gateway event example. For details about the parameters, see **Table 1-7**.

```
{
    "body": "",
    "requestContext": {
        "apiId": "bc1dcffd-aa35-474d-897c-d53425a4c08e",
        "requestId": "11cdcdcf33949dc6d722640a13091c77",
        "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": "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
}
```


Table 1-7 Parameters of an APIG example event

Parameter	Type	Example Value	Description
body	String	Example: {\"test\":\"body\"}"	Actual request in string format
requestContext	Map	Reference example code	Request information, including the API gateway configuration, request ID, authentication information, and source
httpMethod	String	GET	HTTP method
queryStringParameters	Map	Reference example code	Query strings configured in APIG and their actual values
pathParameters	Map	Reference example code	Path parameters configured in APIG and their actual values
headers	Map	Reference example code	Complete headers
path	String	/apig-event-template	Complete path
isBase64Encoded	Boolean	true	The default value is **true**.

Constraints:

- When calling a function using APIG, **isBase64Encoded** is valued **true** by default, indicating that the request body transferred to FunctionGraph is encoded using Base64 and must be decoded for processing.
- The function must return characters strings by using the following structure.

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

CTS

DDS

- DDS example event. For details about the parameters, see **Table 1-8**.

```
{
    "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"
    }
  ]
}
```


Table 1-8 Parameters of a DDS example event

Parameter	Type	Example Value	Description
region	String	ap-southeast-3	Region where the DDS instance is located
event_version	String	1.0	Event version
event_source	String	dds	Event source
event_name	String	insert	Event name
size_bytes	Int	100	Message bytes
token	String	Reference example code	Base64-encoded data
full_document	String	Reference example code	Complete file information
ns	String	Reference example code	Column name
event_source_id	String	Reference example code	Event source ID

DIS

- DIS example event. For details about the parameters, see **Table 1-9**.


```
{
    "ShardID": "shardId-0000000000",
    "Message": {
        "next_partition_cursor":
"eyJnZXRJdGVyYXRvclBhcmFtIjp7InN0cmVhbS1uYW1lIjoiZGlzLXN3dGVzdCIsInBhcnRpdGlvbi1pZCI6InN
oYXJkSWQtMDAwMDAwMDAwMCIsImN1cnNvci10eXBlIjoiVFJJTV9IT1JJWk9OIiwic3RhcnRpbmctc2Vxd
WVuY2UtbnVtYmVyIjoiNCJ9LCJnZW5lcmF0ZVRpbWVzdGFtcCI6MTUwOTYwNjM5MjE5MX0",
        "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"
}
```

Table 1-9 Parameters of a DIS example event

Parameter	Type	Example Value	Description
ShardID	String	shardId-0000000000	Partition ID
next_partition_cursor	String	Reference example code	Next partition cursor
Records	Map	Reference example code	Data records stored in a DIS stream
partition_key	String	Reference example code	Partition key
data	String	Reference example code	Data blocks, which are added by the data producer to the stream
sequence_number	Int	Reference example code	Record ID, which is automatically allocated by DIS
Tag	String	latest	Stream tag
StreamName	String	dis-swtest	Stream name

DMS (for Kafka)/Kafka (Open-Source)

- Kafka example event. For details about the parameters, see **Table 1-10**.

```
{
    "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"
    }
  ]
}
```

Table 1-10 Parameters of a Kafka example event

Parameter	Type	Example Value	Description
event_version	String	v1.0	Event version
event_time	String	Reference example code	Time when an event occurs
trigger_type	String	KAFKA	Event type
region	String	ap-southeast-3	Region where a Kafka instance resides
instance_id	String	81335d56-b9fe-4679-ba95-7030949cc76b	Kafka instance ID
messages	String	Reference example code	Message content
topic_id	String	topic-test	Message ID

DMS (for RabbitMQ)

- DMS for RabbitMQ example event. For details about the parameters, see **Table 1-11**.

```
{
  "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"
    }
```



```
  ]
}
```

Table 1-11 DMS for RabbitMQ parameters

Parameter	Type	Example Value	Description
event_version	String	v1.0	Event version
region	String	ap-southeast-3	Region where a RabbitMQ instance resides
instance_id	String	81335d56-b9fe-4679-ba95-7030949cc76b	RabbitMQ instance ID

GeminiDB MongoDB

- GeminiDB MongoDB example event. For details about the parameters, see **Table 1-12**.

```
{
   "records": [
      {
         "event_name": "\"insert\"",
         "event_version": "1.0",
         "event_source": "gauss_mongo",
         "region": "cn-north-xx",
         "gauss_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\"}"
}
```

Table 1-12 Parameters of a GeminiDB MongoDB example event

Parameter	Type	Example Value	Description
region	String	ap-southeast-3	Region where a GeminiDB instance resides
event_source	String	gemini_mongo	Event source
event_version	String	1.0	Event version
full_document	String	Reference example code	Complete file information

Parameter	Type	Example Value	Description
size_bytes	Int	100	Message bytes
token	String	Reference example code	Base64-encoded data
event_source_id	String	Reference example code	Event source ID
vernier	String	Reference example code	Cursor

GeminiDB DynamoDB

- IoTDA example event. For details about the parameters, see **Table 1-13**.

```
{
   "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"
      }
   }
}
```


Table 1-13 Parameters of an IoTDA example event

Parameter	Type	Example Value	Description
resource	string	device	Data source, which includes device, device property, device message, device message status, device status, batch task, product, and device asynchronous command status and run log.
event	string	create	Triggering event.
event_time	string	20240919T011335Z	Event triggering time in the character string format.
event_time_ms	string	2024-09-19T01:13:35.854Z	Event triggering time in datetime format.
request_id	string	75127474-1a26-4578-8847-3128d6101954	Request ID.
notify_data	Object. For details, see **Table 1-14**.	-	Message to push.

Table 1-14 NotifyData

Parameter	Type	Example Value	Description
body	Object. For details, see **Table 1-15**.	-	Message content.

Table 1-15 NotifyDataBody

Parameter	Type	Example Value	Description
app_id	string	3d40caf3ddfc4e83815b54b50f13aad7	Resource space ID.
app_name	string	DefaultApp_6439vdv2	Resource space name.
device_id	string	66eb7a0ffa8d9c36870c6892_ttytytytytytyt	Device ID, used to uniquely identify a device. The value of this parameter is specified during device registration or allocated by the platform. If the value is allocated by the platform, the value is in the format of *[product_id]_[node_id]*. Maximum length: 256 characters
node_id	string	ttytytytytytyt	Device identifier. This parameter is set to the IMEI, MAC address, or serial number. Maximum length: 64 characters
gateway_id	string	66eb7a0ffa8d9c36870c6892_ttytytytytytyt	Gateway ID, which is the device ID of the parent device. The gateway ID is the same as the device ID if the device is a directly connected device. If the device is an indirectly connected device, the gateway ID is the device ID of the directly connected device with which it associates.
node_type	string	GATEWAY	Device node type.
product_id	string	66eb7a0ffa8d9c36870c6892	Unique ID of the product associated with the device.
product_name	string	test	Name of the product associated with the device.

Parameter	Type	Example Value	Description
status	string	INACTIVE	Device status. • **ONLINE**: The device is online. • **OFFLINE**: The device is offline. • **ABNORMAL**: The device is abnormal. • **INACTIVE**: The device is not activated. • **FREEZED**: The device is frozen.
create_time	string	20240919T011335Z	Time when the device was registered on the platform. The value is in the format of yyyyMMdd'T'HHmmss'Z', for example, **20151212T121212Z**.
auth_info	Object. For details, see **Table 1-16**.	-	Access authentication information about the device.

Table 1-16 AuthInfo

Parameter	Type	Example Value	Description
auth_type	string	SECRET	Authentication mode. Secret authentication (**SECRET**) and certificate authentication (**CERTIFICATES**) are supported. If secret authentication is used, fill in **secret**. If certificate authentication is used, fill in **fingerprint**. If **auth_type** is not set, secret authentication is used by default.

Parameter	Type	Example Value	Description
secure_access	Boolean	false	Whether the device is connected to the platform using a secure protocol. The default value is **true**. • **true**: The device is connected to the platform using a secure protocol. • **false**: The device is connected to the platform using an insecure protocol.
timeout	Integer	0	Validity period of the device verification code, in seconds. The default value is **0**. If the device has not been connected to the platform within the validity period, the platform deletes the registration information of the device. If this parameter is set to **0**, the verification code is always valid. The recommended value is **0**. Note: This parameter is returned only when **timeout** is modified in the device registration or modification API. Minimum value: **0** Maximum value: **2147483647** Default value: **0**

For details about device messages, see the IoTDA official website. For example, **device addition notification**.

LTS

SMN

- SMN example event. For details about the parameters, see **Table 1-17**.

```
{
  "record": [
    {
      "event_version": "1.0",
      "smn": {
        "topic_urn": "topicUrn",
        "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": "functionUrn",
      "event_source": "smn"
    }
```



```
  ],
  "functionname": "test",
  "requestId": "7c307f6a-cf68-4e65-8be0-4c77405a1b2c",
  "timestamp": "Tue Jan 09 2018 15:11:40 GMT+0800 (CST)"
}
```

Table 1-17 Parameters of an SMN example event

Parameter	Type	Example Value	Description
event_version	String	1.0	Event version
topic_urn	String	Reference example code	Unique ID of an SMN event, which is generated by the SMN service
type	String	notification	Event type
requestId	String	Reference example code	Request ID, which is generated by FunctionGraph. The ID of each request is unique.
message_id	String	Reference example code	Message ID, which is generated by SMN. The ID of each message is unique.
message	String	this is smn message content	Message content
event_source	String	smn	Event source
event_subscription_urn	String	Reference example code	Function URN. The value is unique and can be obtained from the function details page.
timestamp	String	Tue Jan 09 2018 15:11:40 GMT+0800 (CST)	Time when an event occurs

OBS

- OBS example event. For details about the parameters, see **Table 1-18**.

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventTime": "2018-01-09T07:50:50.028Z",
      "requestParameters": {
        "sourceIPAddress": "103.218.216.125"
      },
      "s3": {
        "configurationId": "UK1DGFPYUKUZFHNQ00000160CC0B471D101ED30CE24DF4DB",
        "object": {
```



```
                "eTag": "9d377b10ce778c4938b3c7e2c63a229a",
                "sequencer": "00000000160D9E681484D6B4C0000000",
                "key": "job.png",
                "size": 777835
            },
            "bucket": {
                "arn": "arn:aws:s3:::syj-input2",
                "name": "functionstorage-template",
                "ownerIdentity": {
                    "PrincipalId": "0ed1b73473f24134a478962e631651eb"
                }
            }
        },
        "Region": "{region}",
        "eventName": "ObjectCreated:Post",
        "userIdentity": {
            "principalId": "9bf43789b1ff4b679040f35cc4f0dc05"
        }
    }
  ]
}
```


Table 1-18 Parameters of an OBS example event

Parameter	Type	Example Value	Description
eventVersion	String	2.0	Event version
eventTime	String	2018-01-09T07:50:50.028Z	Time when an event occurs. The ISO-8601 time format is used.
sourceIPAddress	String	103.218.216.125	Source IP address
s3	Map	Reference example code	OBS event content
object	Map	Reference example code	object parameter description
bucket	Map	Reference example code	bucket parameter description
arn	String	arn:aws:s3:::syj-input2	Bucket ID
ownerIdentity	Map	Reference example code	ID of the user who creates the bucket
Region	String	ap-southeast-3	Region where the bucket is located
eventName	String	ObjectCreated:Post	Event name

Parameter	Type	Example Value	Description
userIdentity	Map	Reference example code	ID of the Huawei Cloud account that initiates the request

EG

- EG example event. For details about the parameters, see **Table 1-19**.

  Custom RocketMQ event source

```
{
   "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"
}
```

  Custom RabbitMQ event source

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

  OBS application service event source

```
{
   "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":""
    }
  }
 ]
}
```


Table 1-19 Parameters of an EG example event

Parameter	Type	Example Value	Description
datacontenttype	String	application/json	Data type
data	Map	Reference example code	Data
subject	String	Reference example code	Target value
specversion	String	1.0	Version

Parameter	Type	Example Value	Description
id	String	Reference example code	Unique key
source	String	eventSourceName	Event source name
time	String	Reference example code	Subscription time
type	String	ROCKETMQ:CloudTrace:RocketmqCall	Subscription type

1.4 Function Project Packaging Rules

Packaging Rules

In addition to inline code editing, you can create a function by uploading a local ZIP file or JAR file, or uploading a ZIP file from Object Storage Service (OBS). For details, see **Creating a Deployment Package**. **Table 1-20** describes the rules for packaging a function project.

Table 1-20 Function project packaging rules

Runtime	JAR File	ZIP File	ZIP File on OBS
Node.js	Not supported.	• If the function project files are saved under the **~/Code/** directory, select and package all files under this directory to ensure that the function handler is under the root directory after the ZIP file is decompressed. • If the function project uses third-party dependencies, package the dependencies into a ZIP file, and import the ZIP file on the function code page. Alternatively, package the third-party dependencies and the function project files together.	Compress project files into a ZIP file and upload it to an OBS bucket.

Runtime	JAR File	ZIP File	ZIP File on OBS
PHP	Not supported.	• If the function project files are saved under the **~/Code/** directory, select and package all files under this directory to ensure that the function handler is under the root directory after the ZIP file is decompressed. • If the function project uses third-party dependencies, package the dependencies into a ZIP file, and import the ZIP file on the function code page. Alternatively, package the third-party dependencies and the function project files together.	Compress project files into a ZIP file and upload it to an OBS bucket.

Runtime	JAR File	ZIP File	ZIP File on OBS
Python	Not supported.	• If the function project files are saved under the **~/Code/** directory, select and package all files under this directory to ensure that the function handler is under the root directory after the ZIP file is decompressed. • If the function project uses third-party dependencies, package the dependencies into a ZIP file, and import the ZIP file on the function code page. Alternatively, package the third-party dependencies and the function project files together.	Compress project files into a ZIP file and upload it to an OBS bucket.
Java	If the function does not reference third-party components, compile only the function project files into a JAR file.	If the function references third-party components, compile the function project files into a JAR file, and compress all third-party components and the function JAR file into a ZIP file.	Compress project files into a ZIP file and upload it to an OBS bucket.

Runtime	JAR File	ZIP File	ZIP File on OBS
Go 1.x	Not supported.	Zip the compiled file and ensure that the name of the binary file is consistent with that of the handler. For example, if the name of the binary file is **Handler**, set the name of the handler to **Handler**.	Compress project files into a ZIP file and upload it to an OBS bucket.
C#	Not supported.	Compress project files into a ZIP file. The ZIP file must contain the following files: *Project_name*.**deps.json**, *Project_name*.**dll**, *Project_name*.**runtimeconfig.json**, *Project_name*.**pdb**, and **HC.Serverless.Function.Common.dll**.	Compress project files into a ZIP file and upload it to an OBS bucket.
Cangjie	Not supported.	Zip the compiled file and ensure that the name of the binary file is consistent with that of the handler. For example, if the name of the binary file is **libuser_func_test_success.so**, set the name of the handler to **libuser_func_test_success.so**.	Compress project files into a ZIP file and upload it to an OBS bucket.
Custom	Not supported.	Compress project files into a ZIP file. The ZIP file must contain a bootstrap file.	Compress project files into a ZIP file and upload it to an OBS bucket.

Example ZIP Project Packages

- Example directory of a Nods.js project package

```
Example.zip                Example project package
|--- lib                   Service file directory
```

```
|--- node_modules                NPM third-party component directory
|--- index.js                    .js handler file (mandatory)
|--- package.json                NPM project management file
```

- Example directory of a PHP project package

```
Example.zip                      Example project package
|--- ext                         Extension library directory
|--- pear                        PHP extension and application repository
|--- index.php                   PHP handler file
```

- Example directory of a Python project package

```
Example.zip                      Example project package
|--- com                         Service file directory
|--- PLI                         Third-party dependency PLI directory
|--- index.py                    .py handler file (mandatory)
|--- watermark.py                .py file for image watermarking
|--- watermark.png               Watermarked image
```

- Example directory of a Java project package

```
Example.zip                      Example project package
|--- obstest.jar                 Service function JAR file
|--- esdk-obs-java-3.20.2.jar    Third-party dependency JAR file
|--- jackson-core-2.10.0.jar     Third-party dependency JAR file
|--- jackson-databind-2.10.0.jar Third-party dependency JAR file
|--- log4j-api-2.12.0.jar        Third-party dependency JAR file
|--- log4j-core-2.12.0.jar       Third-party dependency JAR file
|--- okhttp-3.14.2.jar           Third-party dependency JAR file
|--- okio-1.17.2.jar             Third-party dependency JAR file
```

- Example directory of a Go project package

```
Example.zip                      Example project package
|--- testplugin.so               Service function package
```

- Example directory of a C# project package

```
Example.zip                                 Example project package
|--- fssExampleCsharp2.0.deps.json          File generated after project compilation
|--- fssExampleCsharp2.0.dll                File generated after project compilation
|--- fssExampleCsharp2.0.pdb                File generated after project compilation
|--- fssExampleCsharp2.0.runtimeconfig.json File generated after project compilation
|--- Handler                                Help file, which can be directly used
|--- HC.Serverless.Function.Common.dll      .dll file provided by FunctionGraph
```

- Example directory of a Cangjie project package

```
fss_example_cangjie.zip                     Example project package
|--- libuser_func_test_success.so           Service function package
```

- Custom

```
Example.zip                      Example project package
|--- bootstrap                   Executable boot file
```

1.5 Referencing DLLs in Functions

You can reference DLLs in functions as follows:

- By default, the root directory and the **lib** folder in this directory have been configured in the **LD_LIBRARY_PATH** environment variable. You only need to add dynamic link libraries (DLLs) here.
- You can directly modify the **LD_LIBRARY_PATH** variable in the code.

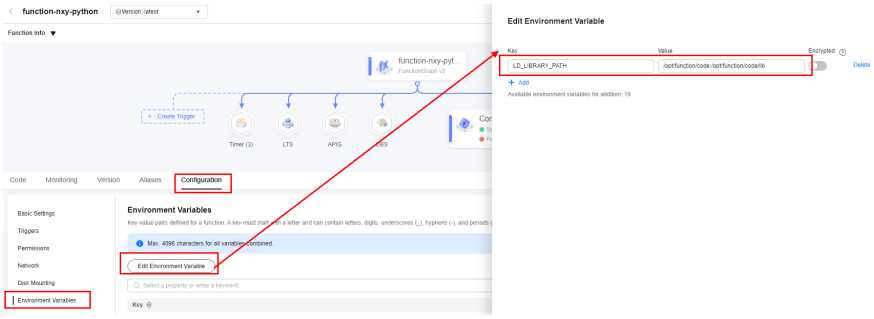
CAUTION

For Python, after the interpreter is started and initialized, the operation of modifying the **LD_LIBRARY_PATH** variable in the code does not take effect on the loading of dynamic link libraries (DLLs). Therefore, the path of the DLLs that Python code depends on must be configured before the interpreter is started.

- If the dependent **.so** file is stored in another directory, you can specify it when setting the **LD_LIBRARY_PATH** environment variable. For details, see **Configuring Environment Variables**.

  In the following example, **/opt/function/code** and **/opt/function/code/lib** indicate the project directories of the function code.

  Figure 1-1 Setting environment variables



- If a library in a mounted file system is used, specify its directory in the **LD_LIBRARY_PATH** variable on the **Configuration** tab page.

2 Node.js

2.1 Function Development Overview

FunctionGraph supports the following Node.js runtimes:

- Node.js 6.10
- Node.js 8.10
- Nodejs 10.16
- Nodejs 12.13
- Node.js 16.17
- Node.js 18.15
- Node.js 20.15

Function Syntax

Node.js 6.10

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

- **handler**: name of the function that FunctionGraph invokes to execute your code. The name must be consistent with that you define when creating a function.
- **event**: event parameter defined for the function. The parameter is in JSON format.
- **context**: runtime information provided for executing the function. For details, see **SDK APIs**.
- **callback**: used to return the defined **err** and **message** information to the frontend. The general syntax is **callback(err, message)**. You can define the error or message content, for example, a character string.

Node.js 8.10 and later

Node.js 8.10 and later are compatible with the APIs of Node.js 6.10, and supports an **async** handler. Responses are output through **return**.

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


The handler of a Node.js function is in the format of *[file name].[function name]*. You can configure the **handler** on the FunctionGraph console. For example, if you set the handler to **index.handler** in your function, FunctionGraph will load the handler function defined in the **index.js** file.

Node.js Initializer

For details about the initializer, see **Initializer**.

The initializer is in the format of *[File name].[Initializer name]*.

For example, if the initializer is named **index.initializer**, FunctionGraph loads the initializer function defined in the **index.js** file.

To use Node.js to build initialization logic, define a Node.js function as the initializer. The following is a simple initializer:

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

- Function name

  The function name **exports.initializer** must be the initializer function name specified for a function.

  For example, if the initializer is named **index.initializer**, FunctionGraph loads the initializer function defined in the **index.js** file.

- context

  The **context** parameter contains the runtime information about a function. For example, request ID, temporary AK, and function metadata.

- callback

  The **callback** parameter is used to return the invocation result. The signature of this parameter is **function(err, data)**, which is the same as that of the common **callback** parameter used in Node.js. If the value of **err** is not null, the function will return **HandledInitializationError**. The value of **data** is invalid because no value will be returned for function initialization. You can set the **data** parameter to null by referring to the previous example.

Third-Party Components Integrated with the Node.js Runtime

Table 2-1 Third-Party Components Integrated with the Node.js Runtime

Name	Usage	Version
q	Asynchronous method encapsulation	1.5.1
co	Asynchronous process control	4.6.0
lodash	Common tool and method library	4.17.10
esdk-obs-nodejs	OBS SDKs	2.1.5
express	Simplified web-based application development framework	4.16.4

Name	Usage	Version
fgs-express	Uses the Node.js application framework to run serverless applications and REST APIs in FunctionGraph and APIG. This component provides an example of using the Express framework to build serverless web applications or services and RESTful APIs.	1.0.1
request	Simplifies HTTP invocation and supports HTTPS and redirection.	2.88.0

SDK APIs

Table 2-2 describes the context methods provided by FunctionGraph.

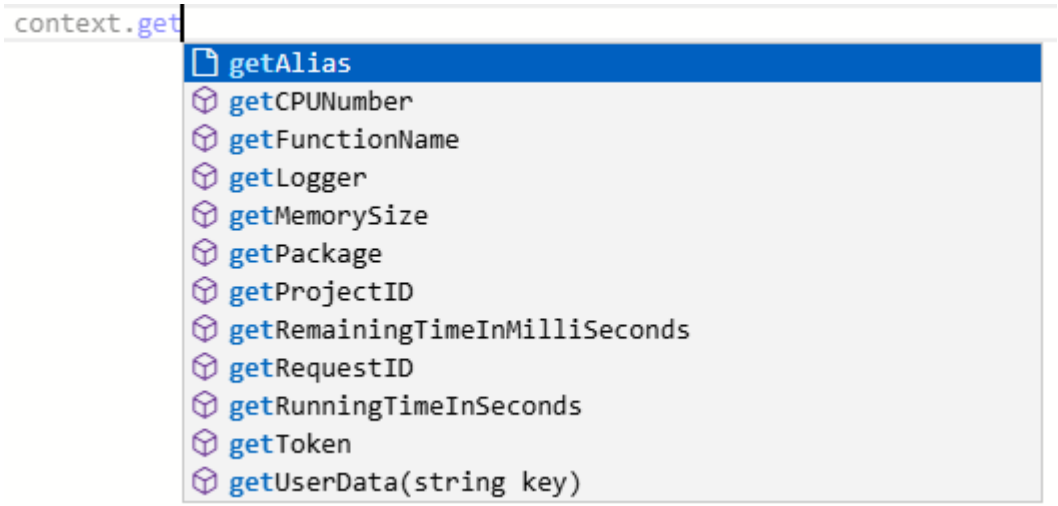
Table 2-2 Context methods

Method	Description
getRequestID()	Obtains a request ID.
getRemainingTimeInMilliSeconds ()	Obtains the remaining running time of a function.
getAccessKey()	Obtains the AK (valid for 24 hours) with an agency. If you use this method, you need to configure an agency for the function. **FunctionGraph has stopped maintaining the getAccessKey API in the Runtime SDK. You cannot use this API to obtain a temporary AK.**
getSecretKey()	Obtains the SK (valid for 24 hours) with an agency. If you use this method, you need to configure an agency for the function. **FunctionGraph has stopped maintaining the getSecretKey API in the Runtime SDK. You cannot use this API to obtain a temporary SK.**
getSecurityAccessKey()	Obtains the SecurityAccessKey (valid for 24 hours) with an agency. The cache duration is 10 minutes. That is, the same content is returned within 10 minutes. To use this method, you need to configure an agency for the function.
getSecuritySecretKey()	Obtains the SecuritySecretKey (valid for 24 hours) with an agency. The cache duration is 10 minutes. That is, the same content is returned within 10 minutes. To use this method, you need to configure an agency for the function.

Method	Description
getSecurityToken()	Obtains the SecurityToken (valid for 24 hours) with an agency. The cache duration is 10 minutes. That is, the same content is returned within 10 minutes. To use this method, you need to configure an agency for the function.
getUserData(string key)	Uses keys to obtain the values passed by environment variables.
getFunctionName()	Obtains the name of a function.
getRunningTimeInSeconds ()	Obtains the timeout of a function.
getVersion()	Obtains the version of a function.
getMemorySize()	Obtains the allocated memory.
getCPUNumber()	Obtains CPU usage of a function.
getPackage()	Obtains a function group.
getToken()	Obtains the token (valid for 24 hours) with an agency. If you use this method, you need to configure an agency for the function.
getLogger()	Obtains the **logger** method provided by the context and returns a log output class. Logs are output in the format of *Time-Request ID-Content* by using the **info** method. For example, use the **info** method to output logs: logg = context.getLogger() logg.info("hello")
getAlias()	Obtains function alias.

As shown in **Figure 2-1**, you can use the context class in the code editor on the FunctionGraph console.

Figure 2-1 Using the context class



Helpful Links

- For details about how to use Node.js to develop an event function, see **Developing a Node.js Event Function**.
- For details about how to use Node.js to develop an HTTP function, see **Developing an HTTP Function Using Node.js**.
- For details about how to create a dependency package for a Node.js function, see **Creating a Dependency for a Node.js Function**.
- For more information about function development, such as the supported runtimes, trigger events, function project packaging specifications, and DLL referencing, see **Function Development Overview**.

2.2 Developing a Node.js Event Function

You can develop a Node.js event function locally and upload the code file, or create a function on the FunctionGraph console and edit code online.

For details about the syntax and SDK APIs of Node.js functions, see **Function Development Overview**.

Constraints

- If the first parameter returned by **callback** is not **null**, the function execution fails and the HTTP error message defined in the second parameter is returned.
- When an APIG trigger is used, the response must be in the output format used in this example. The body only supports the following values:
  - **null**: The HTTP response body is empty.
  - **[]byte**: The content in this byte array is the body of an HTTP response.
  - **string**: The content in this string is the body of an HTTP response.
- When calling a function using APIG, **isBase64Encoded** is valued **true** by default, indicating that the request body transferred to FunctionGraph is encoded using Base64 and must be decoded for processing.

  The function must return characters strings by using the following structure.


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

Step 1: Creating a Node.js Function Project

1. Open the local text editor, write the function code, and name the file **index.js**.
   - The following is the code for the sync handler:

     ```
     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);
     }
     ```

   - The following is the code for async handler (with runtime 8.10 or later):

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

     If your Node.js function contains an asynchronous task, use **Promise** to execute the task in the current invocation. You can directly return the declared **Promise** or await to execute it.

     The asynchronous task can be executed only before the function responds to requests.

     ```
     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;
         // another way
         // res = await promise;
         // return res;
     }
     ```


Asynchronous function execution:

To execute the function asynchronously (respond immediately upon invocation while continuing task execution), you can call the API for **Executing a Function Asynchronously** through SDKs or APIs.

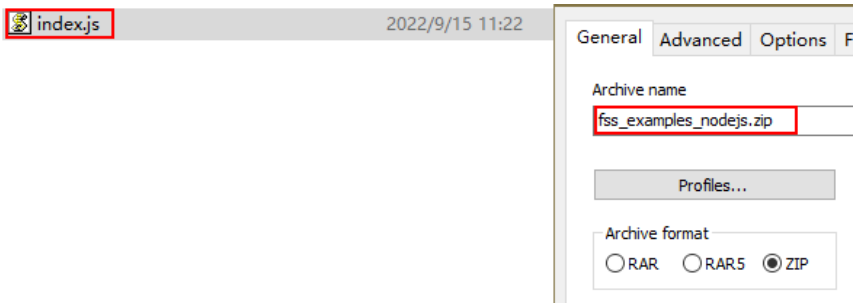
For an APIG trigger, click its name to go to the APIG console, and select **Asynchronous** for the **Invocation Mode**. For details, see **Asynchronous Invocation**.

2. Package the function project. The following uses an async handler as an example.

   After creating the function project, you get the following directory. Select all files under the directory and package them into the **fss_examples_nodejs.zip** file. Ensure that the function handler is under the root directory after the ZIP file is decompressed.

   You can also download the **Node.js function sample project package** and use it directly.

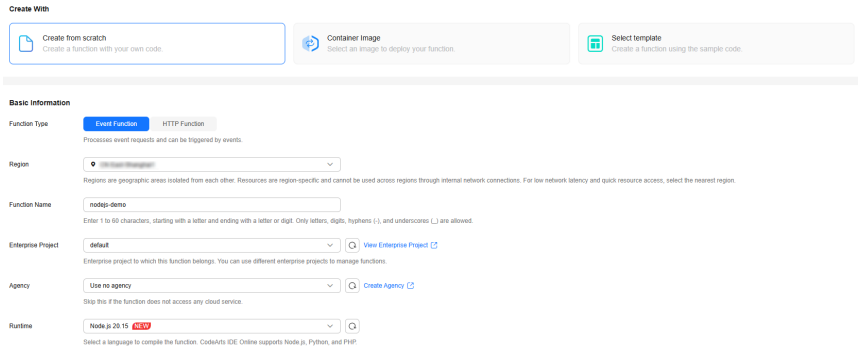
   Figure 2-2 Packaging the project files



Step 2: Create a Function

1. Log in to the **FunctionGraph console** and click **Create Function** in the upper right corner.
2. Create a Node.js event function from scratch and click **Create Now** as shown in**Figure 2-3**. The function details page is displayed.

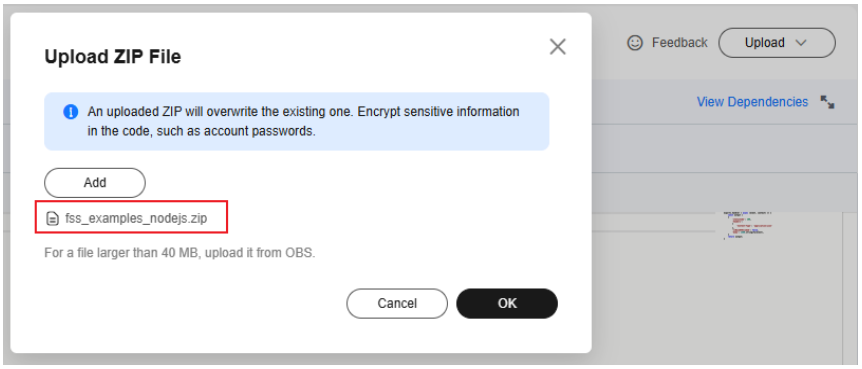
   Figure 2-3 Creating a Node.js function



3. Upload the **fss_examples_nodejs.zip** file packed in **Step 1: Creating a Node.js Function Project** by clicking **Upload > Local ZIP**, as shown in **Figure 2-4**.

The uploaded code will be automatically deployed on the FunctionGraph console. If you have modified the code, click **Deploy** again.

Figure 2-4 Uploading the code package

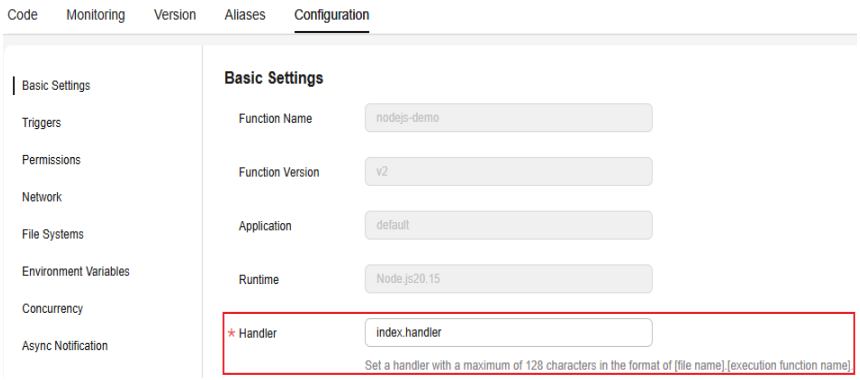


NOTE

Modifying the function handler:

In the navigation pane on the left of the FunctionGraph console, choose **Functions** > **Function List**. Click the name of the function to be set. On the function details page that is displayed, choose **Configuration** > **Basic Settings** and set the **Handler** parameter, as shown in **Figure 2-5**.

Figure 2-5 Handler parameter

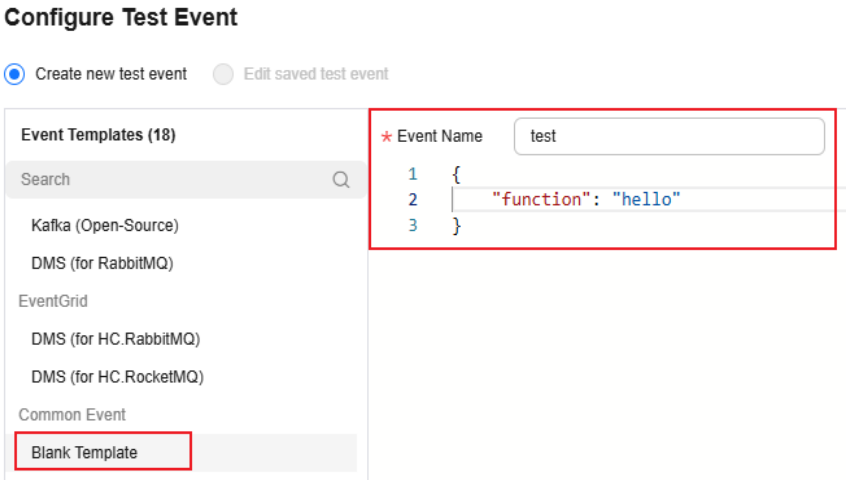


- The **index** of the handler must be consistent with the file name of your function in **Step 1: Creating a Node.js Function Project**, because the file name will help to locate the function file.
- The **handler** is a function name, which must be consistent with that defined in the **index.js** file in **Step 1: Creating a Node.js Function Project**.

Step 3: Testing the Function

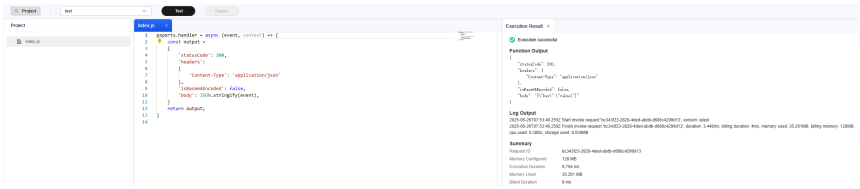
1. On the **Code** tab, click **Test**. In the **Configure Test Event** dialog box, set **test**, as shown in **Figure 2-6**, and click **Create**.

Figure 2-6 Configuring a test event



2. Select the configured test event **test** and click **Test**.
3. As shown in **Figure 2-7**, the **Execution Result** window is displayed on the right. You can check whether the function is executed successfully.

Figure 2-7 Test result



Function Execution Result Description

The execution result consists of the function output, summary, and log output.

Table 2-3 Description of the execution result

Parameter	Successful Execution	Failed Execution

Function Output	The defined function output information is returned.	A JSON file that contains **errorMessage** and **errorType** is returned. The format is as follows:
		{
		"errorMessage": "",
		"errorType":"",
		}
		errorMessage: Error message returned by the runtime.
		errorType: Error type.

Parameter	Successful Execution	Failed Execution
Summary	**Request ID**, **Memory Configured**, **Execution Duration**, **Memory Used**, and **Billed Duration** are displayed.	**Request ID**, **Memory Configured**, **Execution Duration**, **Memory Used**, and **Billed Duration** are displayed.
Log Output	Function logs are printed. A maximum of 4 KB logs can be displayed.	Error information is printed. A maximum of 4 KB logs can be displayed.

Helpful Links

- For details about how to use Node.js to develop an HTTP function, see **Developing an HTTP Function Using Node.js**.
- For details about how to create a Node.js function dependency, see **Creating a Dependency for a Node.js Function**.
- For more information about function development, such as the supported runtimes, trigger events, function project packaging specifications, and DLL referencing, see **Function Development Overview**.

2.3 Developing an HTTP Function Using Node.js

This section uses the Koa framework as an example to describe how to develop an HTTP function using Node.js.

Constraints

- HTTP functions can only use APIG or APIC triggers.

  According to the forwarding protocol between FunctionGraph and APIG/APIC, a valid HTTP function response must contain body(String), statusCode(int), headers(Map), and isBase64Encoded(boolean). By default, the response is encoded using Base64. The default value of isBase64Encoded is true. The same applies to other frameworks.
- By default, port 8000 is enabled for HTTP functions.
- When calling a function using the APIG trigger, **isBase64Encoded** is valued **true** by default, indicating that the request body transferred to FunctionGraph is encoded using Base64 and must be decoded for processing.

  The function must return characters strings by using the following structure.

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


Prerequisites

You have installed the Node.js environment on the local OS. You are advised to create Node.js dependencies in the **EulerOS environment**.

Deploying the Koa Framework Using an HTTP Function

Koa is a Node.js-based web development framework, which is mainly used to build efficient and scalable web applications.

1. Run the following command to create a project folder.
   ```
   mkdir koa-example && cd koa-example
   ```
2. Run the following commands to initialize the Node.js project and download the koa framework.
   ```
   npm init -y
   npm i koa
   ```
   After the commands are executed, the **node_modules** folder and the **package.json** and **package-lock.json** files are added to the folder.
3. Create the **index.js** file to reference the Koa framework. For details about how to use this framework, see **Koa's guide**.

   Sample code:
   ```
   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.')
   ```
4. You have prepared a bootstrap file as the startup file of the HTTP function. Add the following content in the file:
   ```
   /opt/function/runtime/nodejs20.15/rtsp/nodejs/bin/node $RUNTIME_CODE_ROOT/index.js
   ```
   - **/opt/function/runtime/nodejs20.15/rtsp/nodejs/bin/node**: path of the Node.js compilation environment.
   - *$RUNTIME_CODE_ROOT*: system variable that represents the **/opt/function/code** path for storing project code in a container.
   - **index.js**: project file created in **3**. You can also define a custom name.

   Table 2-4 lists the supported Node.js versions and the corresponding paths.

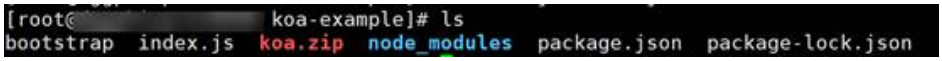
   Table 2-4 Node.js paths

Language	Path
Node.js 6	/opt/function/runtime/nodejs6.10/rtsp/nodejs/bin/node
Node.js 8	/opt/function/runtime/nodejs8.10/rtsp/nodejs/bin/node
Node.js 10	/opt/function/runtime/nodejs10.16/rtsp/nodejs/bin/node

Language	Path
Node.js 12	/opt/function/runtime/nodejs12.13/rtsp/nodejs/bin/node
Node.js 14	/opt/function/runtime/nodejs14.18/rtsp/nodejs/bin/node
Node.js 16	/opt/function/runtime/nodejs16.17/rtsp/nodejs/bin/node
Node.js 18	/opt/function/runtime/nodejs18.15/rtsp/nodejs/bin/node
Node.js 20	/opt/function/runtime/nodejs20.15/rtsp/nodejs/bin/node

5. Compress all project files and the bootstrap file into a ZIP package. Ensure that the project file is in the root directory after the decompression.

   Figure 2-8 Packaging all files

6. Log in to the **FunctionGraph console** and click **Create Function** in the upper right corner.
7. Create an HTTP function from scratch and upload the ZIP file to the **Code** tab.

   After the function is created, you can use the Koa framework to develop applications. The framework provides the infrastructure for processing requests, and your custom application code defines the specific service logic.

Helpful Links

- For details about how to develop functions in Node.js, see **Function Development Overview**.
- For details about how to use Node.js to develop an event function, see **Developing a Node.js Event Function**.
- For details about how to create a Node.js function dependency, see **Creating a Dependency for a Node.js Function**.
- For details about HTTP functions, see **Creating an HTTP Function**.
- For more information about function development, such as the supported runtimes, trigger events, function project packaging specifications, and DLL referencing, see **Function Development Overview**.

2.4 Node.js Function Template

Node.js Function

The following is the sample code template of a Node.js function:

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



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

When you create an empty Node.js event function on the FunctionGraph console, the preceding sample code is deployed by default.

Helpful Links

- For details about how to use Node.js to develop an event function, see **Developing a Node.js Event Function**.
- For details about how to use Node.js to develop an HTTP function, see **Developing an HTTP Function Using Node.js**.
- For details about how to create a dependency package for a Node.js function, see **Creating a Dependency for a Node.js Function**.
- For more information about function development, such as the supported runtimes, trigger events, function project packaging specifications, and DLL referencing, see **Function Development Overview**.

2.5 Creating a Dependency for a Node.js Function

You are advised to create function dependencies in Huawei Cloud EulerOS 2.0. If other OSs are used, the dynamic link library may not be found due to the differences between underlying dependency libraries.

Constraints

If the modules to be installed need dependencies such as **.dll**, **.so**, and **.a**, archive them to a **.zip** package.

Setting Up the EulerOS Environment

You are advised to create function dependencies in EulerOS. EulerOS is an enterprise-grade Linux OS based on open-source technology. It features high security, scalability, and performance, meeting customers' requirements for IT infrastructure and cloud computing services.

You can set up the **Huawei Cloud EulerOS** environment using the following methods:

- Buy a EulerOS ECS on Huawei Cloud by referring to **Purchasing and Logging In to a Linux ECS**. On the **Configure Basic Settings** page, select **Public Image**, and select **Huawei Cloud EulerOS** and an image version.
- Download the EulerOS image, and use virtualization software to set up the EulerOS VM on a local PC.

Creating a Dependency for a Node.js Function

Before creating a dependency, ensure that Node.js matching the function runtime has been installed in the environment. The following uses Node.js 20.15 as an example to describe how to create a MySQL dependency package.

Step 1 Run the following command to install the MySQL dependency package.

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

The **node_modules** folder is generated under the current directory.

Step 2 Run the following command to generate the ZIP dependency package.

```
zip -rq mysql-node20.15.zip node_modules
```

----End

To package multiple dependencies, follow the steps below:

Step 1 Create a **package.json** file with the following content: **Change the dependency version as required.**

```
{
  "name": "test",
  "version": "1.0.0",
  "dependencies": {
    "redis": "~2.8.0",
    "mysql": "~2.17.1"
  }
}
```

Step 2 Run the following command.

```
npm install --save
```

Step 3 Compress **node_modules** into a ZIP package. This generates a dependency that contains both MySQL and Redis.

```
zip -rq mysql-node20.15.zip node_modules
```

----End

Helpful Links

- For details about how to use Node.js to develop an event function, see **Developing a Node.js Event Function**.
- For details about how to use Node.js to develop an HTTP function, see **Developing an HTTP Function Using Node.js**.
- For details about how to create a dependency package for a Node.js function, see **Creating a Dependency for a Node.js Function**.
- For more information about function development, such as the supported runtimes, trigger events, function project packaging specifications, and DLL referencing, see **Function Development Overview**.

2.6 Developing a Node.js Function Using Huawei Cloud SDKs

Huawei Cloud **API Explorer** provides API reference documents and SDK code examples for each cloud service.

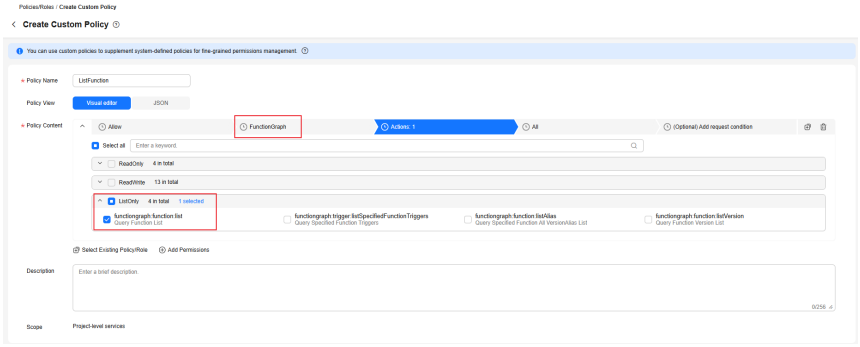
This section uses the API for querying the function list as an example to describe how to develop a Node.js function on the FunctionGraph console using Huawei Cloud SDKs.

Step 1: Creating a Function Agency

1. Log in to the **IAM console**.
2. In the navigation pane on the left, choose **Permissions** > **Policies/Roles**. On the displayed page, click **Create Custom Policy** in the upper right corner.
3. Take the agency to be created for the API used to query the function list as an example. Configure a custom policy that contains the permission for querying the function list, as shown in **Figure 2-9**, and click **OK**.

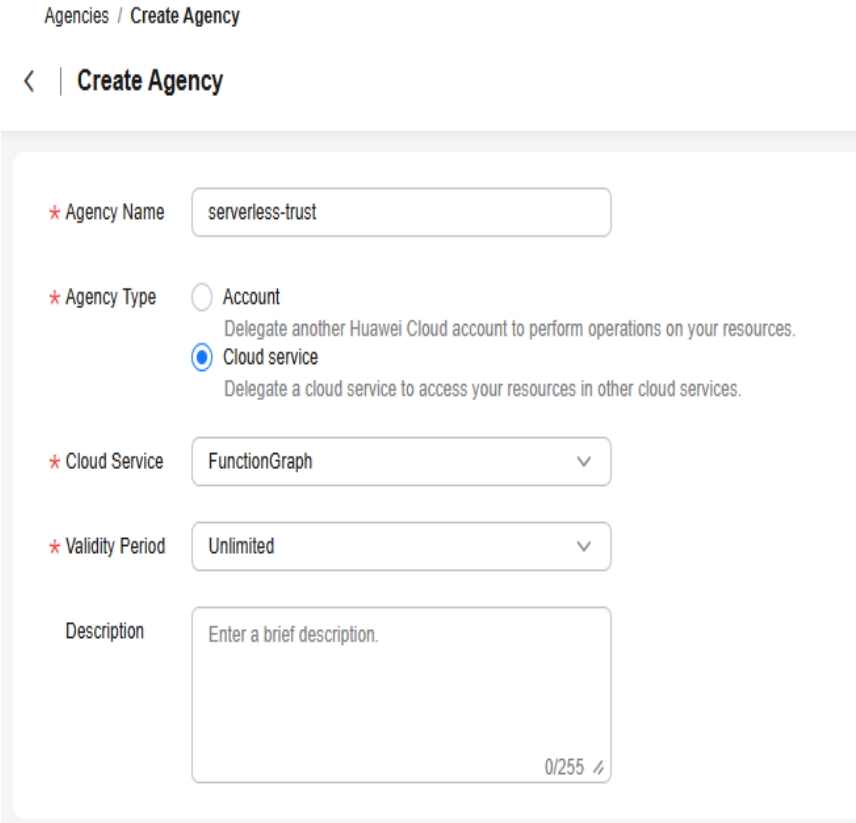
   For more information about permissions, see **Basic Concepts About Permissions**.

   Figure 2-9 Custom policy for querying a function list

4. In the navigation pane of the IAM console, choose **Agencies**. Then, click **Create Agency** in the upper right corner.
5. Configure agency parameters. After the parameters are configured, as shown in **Figure 2-10**, click **OK**. The system displays a message indicating that the creation is successful. Click **Authorize**.

Figure 2-10 Entering basic information



6. On the displayed page, select the custom policy created in 3, click **Next**, select the authorization scope based on the actual needs, and click **OK**.

Step 2: Creating a Node.js Function

1. Log in to the **FunctionGraph console** and click **Create Function** in the upper right corner.
2. Create a Node.js event function from scratch, select the agency created in **Step 1: Creating a Function Agency**, select the latest runtime version, and click **Create Function**.
3. On the **Code** tab, scroll down to the **Dependencies** area and click **Add**.
4. Select **Private** for **Type** and click **Create Dependency** as shown in **Figure 2-11**. The dependency creation page is displayed.

Figure 2-11 Selecting dependencies

5. Create the required Node.js dependency for the function by referring to **Huawei Cloud Node.js SDK** and **Configuring a Dependency for a Function**.

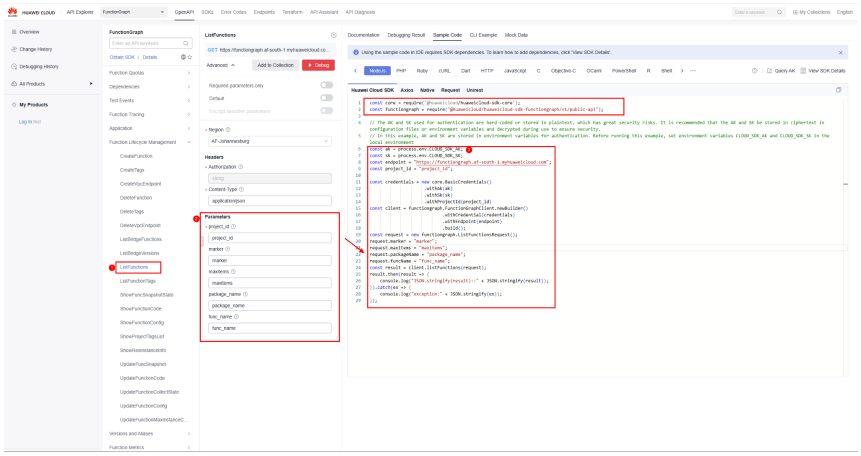
   Note that the runtime version of the dependency must be the same as that of the Node.js function.

6. After the dependency is created, return to **4** and add the created dependency.

Step 3: Obtaining SDK Sample Code from APIE

Step 1 Open **API Explorer**, select the required API, click the **Sample Code** tab, and select the **Node.js** language, as shown in **Figure 2-12**.

Figure 2-12 APIE sample code



1. The API for querying functions is used as an example.
2. Enter the parameters required by the API. For details about the parameters, see the corresponding section in the API reference. In this example, see **Querying Functions**.
3. Copy the code generated by API Explorer and paste it in the code editing box of the function created in **Step 2: Creating a Node.js Function**.

Step 2 You are advised to configure the AK/SK in the function environment variables. For details, see **Configuring Environment Variables**. And you can use the context.getUserData(string key) method to obtain the AK/SK in the code.

The modified code is as follows:

```
const core = require('@huaweicloud/huaweicloud-sdk-core');
const functiongraph = require("@huaweicloud/huaweicloud-sdk-functiongraph/v2/public-api");
exports.handler = async (event, context) => {
    const ak = context.getUserData("AK");
    const sk = context.getUserData("SK");
    const endpoint = "https://functiongraph.cn-north-4.myhuaweicloud.com";
    const project_id = "project_id";
    const credentials = new core.BasicCredentials()
        .withAk(ak)
        .withSk(sk)
        .withProjectId(project_id)
    const client = functiongraph.FunctionGraphClient.newBuilder()
        .withCredential(credentials)
        .withEndpoint(endpoint)
        .build();
```



```
    const request = new functiongraph.ListFunctionsRequest();
    request.marker = "marker";
    request.maxitems = "maxitems";
    request.packageName = "package_name";
    request.funcName = "func_name";
    const result = client.listFunctions(request);
    result.then(result => {
        console.log("JSON.stringify(result)::" + JSON.stringify(result));
    }).catch(ex => {
        console.log("exception:" + JSON.stringify(ex));
    });
    const output =
    {
        'statusCode': 200,
        'headers':
        {
            'Content-Type': 'application/json'
        },
        'isBase64Encoded': false,
        'body': JSON.stringify(event),
    }
    return output;
}
```

Step 3 (Optional) To use a more secure authentication mode, replace the following code:

```
const ak = context.getUserData("AK");
const sk = context.getUserData("SK");
const credentials = new core.BasicCredentials()
    .withAk(ak)
    .withSk(sk)
    .withProjectId(project_id)
```

with

```
const ak = context.getSecurityAccessKey();
const sk = context.getSecuritySecretKey();
const st = context.getSecurityToken();
const credentials = new core.BasicCredentials()
    .withAk(ak)
    .withSk(sk)
    .withProjectId(project_id)
    .with_security_token(st)
```

----End

3 Python

3.1 Function Development Overview

FunctionGraph supports the following Python runtimes:

- Python 2.7
- Python 3.6
- Python 3.9
- Python 3.10
- Python 3.12

Function Syntax

Use the following syntax when creating a handler function in Python:

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

- **handler**: name of the function that FunctionGraph invokes to execute your code. The name must be consistent with that you define when creating a function.
- **event**: event parameter defined for the function. The parameter is in JSON format.
- **Context**: runtime information provided for executing the function. For details, see **SDK APIs**.

The Python function handler is in the format of *[File name].[Function name]*. You can configure the **handler** on the FunctionGraph console.

Python Initializer

For details about the initializer, see **Initializer**.

The initializer is in the format of *[File name].[Initializer name]*.

For example, if the initializer is named **main.my_initializer**, FunctionGraph loads the my_initializer function defined in the **main.py** file.

To use Python to build initialization logic, define a Python function as the initializer. The following is a simple initializer (Python 3.12 is used as an example):

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

- Function name

  The function name **my_initializer** must be the initializer function name specified for a function. For example, if the initializer is named **main.my_initializer**, FunctionGraph loads the my_initializer function defined in the **main.py** file.

- context

  The **context** parameter contains the runtime information about a function. For example, request ID, temporary AK, and function metadata.

Non-standard Libraries Integrated with the Python Runtime

Table 3-1 lists the non-standard libraries integrated with Python, which can be directly declared and used in Python function code.

Table 3-1 Non-standard libraries integrated with Python

Library	Usage	Version
dateutil	Date and time processing	2.6.0
requests	HTTP library	2.7.0
httplib2	httpclient	0.10.3
numpy	Mathematical computation	1.13.1
redis	Redis client	2.10.5
obsclient	OBS client	-
smnsdk	SMN access	1.0.1

SDK APIs

Table 3-2 describes the context methods provided by FunctionGraph.

Table 3-2 Context methods

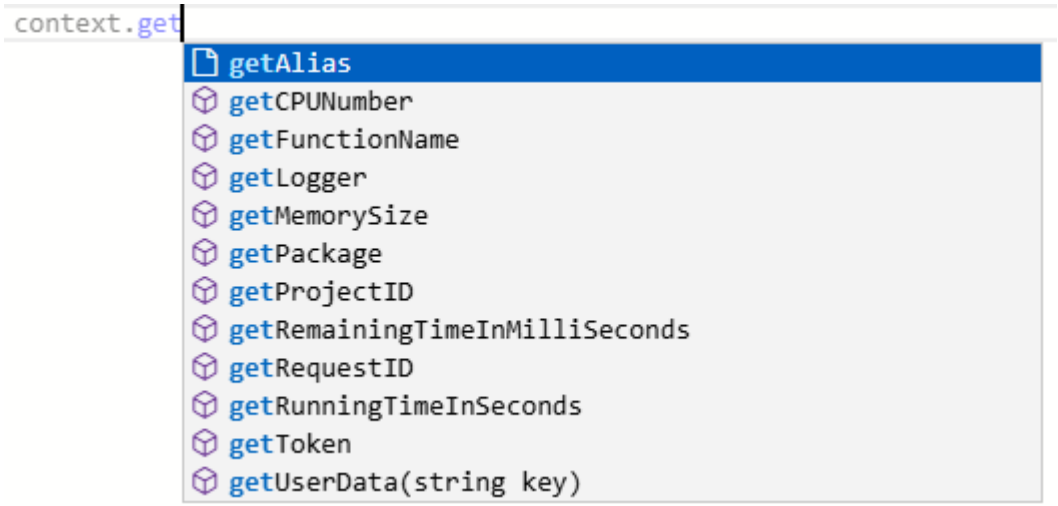
Method	Description
getRequestID()	Obtains a request ID.
getRemainingTimeInMilliSeconds ()	Obtains the remaining running time of a function.

Method	Description
getAccessKey()	Obtains the AK (valid for 24 hours) with an agency. If you use this method, you need to configure an agency for the function. **FunctionGraph has stopped maintaining the getAccessKey API in the Runtime SDK. You cannot use this API to obtain a temporary AK.**
getSecretKey()	Obtains the SK (valid for 24 hours) with an agency. If you use this method, you need to configure an agency for the function. **FunctionGraph has stopped maintaining the getSecretKey API in the Runtime SDK. You cannot use this API to obtain a temporary SK.**
getSecurityAccessKey()	Obtains the SecurityAccessKey (valid for 24 hours) with an agency. The cache duration is 10 minutes. That is, the same content is returned within 10 minutes. To use this method, you need to configure an agency for the function.
getSecuritySecretKey()	Obtains the SecuritySecretKey (valid for 24 hours) with an agency. The cache duration is 10 minutes. That is, the same content is returned within 10 minutes. To use this method, you need to configure an agency for the function.
getSecurityToken()	Obtains the SecurityToken (valid for 24 hours) with an agency. The cache duration is 10 minutes. That is, the same content is returned within 10 minutes. To use this method, you need to configure an agency for the function.
getUserData(string key)	Uses keys to obtain the values passed by environment variables.
getFunctionName()	Obtains the name of a function.
getRunningTimeInSeconds ()	Obtains the timeout of a function.
getVersion()	Obtains the version of a function.
getMemorySize()	Obtains the allocated memory.
getCPUNumber()	Obtains CPU usage of a function.
getPackage()	Obtains a function group.
getToken()	Obtains the token (valid for 24 hours) with an agency. If you use this method, you need to configure an agency for the function.

Method	Description
getLogger()	Obtains the **logger** method provided by the context and returns a log output class. Logs are output in the format of *Time-Request ID-Content* by using the **info** method. For example, use the **info** method to output logs: log = context.getLogger() log.info("test")
getAlias()	Obtains function alias.

As shown in **Figure 3-1**, you can use the context class in the code editor on the FunctionGraph console.

Figure 3-1 Using the context class



Helpful Links

- For details about how to use Python to develop an event function, see **Developing a Python Event Function**.
- For details about how to create a dependency for a Python function, see **Creating a Dependency for a Python Function**.
- For more information about function development, such as the supported runtimes, trigger events, function project packaging specifications, and DLL referencing, see **Function Development Overview**.

3.2 Developing a Python Event Function

You can develop a Python event function locally and upload the code file, or create a function on the FunctionGraph console and edit code online.

For details about the syntax and SDK APIs of Python functions, see **Function Development Overview**.

Constraints

- FunctionGraph can return only the following types of values:
  - **None**: The HTTP response body is empty.
  - **String**: The content in this string is the body of an HTTP response.
  - **Other**: For a value rather than **None** or **String**, FunctionGraph encodes the value in JSON, and uses the encoded object as the body of an HTTP response. The **Content-Type** header of the HTTP response is set to **application/json**.
- When calling a function using APIG, **isBase64Encoded** is valued **true** by default, indicating that the request body transferred to FunctionGraph is encoded using Base64 and must be decoded for processing.

  The function must return characters strings by using the following structure.

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

- When you write code in Python, do not name your package with the same suffix as a standard Python library, such as **json**, **lib**, and **os**. Otherwise, an error will be reported indicating a module loading failure.

Step 1: Creating a Python Function Project

1. Write code for printing text **helloworld**.

   Open the text editor, compile a HelloWorld function, and save the code file as **helloworld.py**. The code is as follows:

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

2. Define a FunctionGraph function.

   Open the text editor, write the function code, and save the function file as **index.py** under the same directory as the **helloworld.py** file. The function code is as follows:

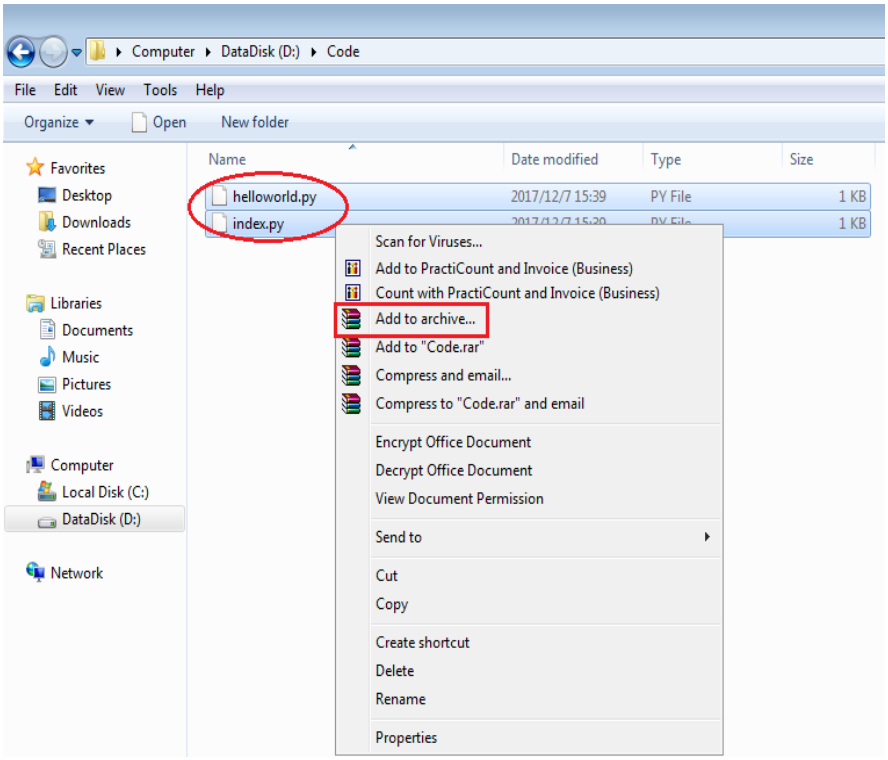
   ```
   # -*- coding:utf-8 -*-
   import json
   import helloworld

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

3. Package the project files.

   After creating the function project, you get the following directory. Select all files under the directory and package them into the **fss_examples_python3.zip** file, as shown in **Figure 3-2**. Ensure that the function handler is under the root directory after the ZIP file is decompressed.

   You can also download the **Python function sample project package** and use it directly.

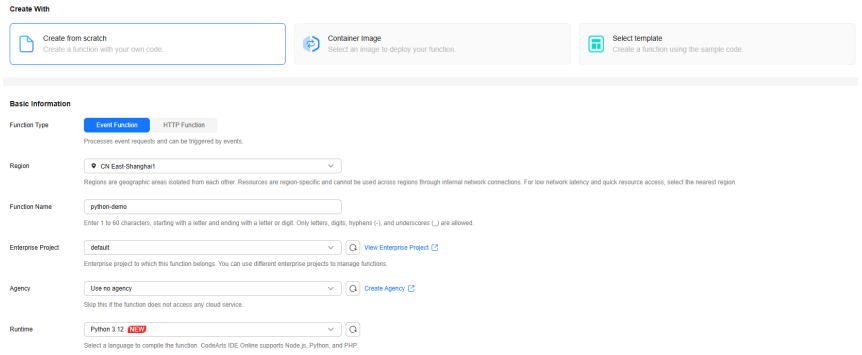
Figure 3-2 Packaging the project files



Step 2: Creating a Function

1. Log in to the **FunctionGraph console** and click **Create Function** in the upper right corner.
2. Create a Python event function from scratch and click **Create Now** as shown in**Figure 3-3**. The function details page is displayed.

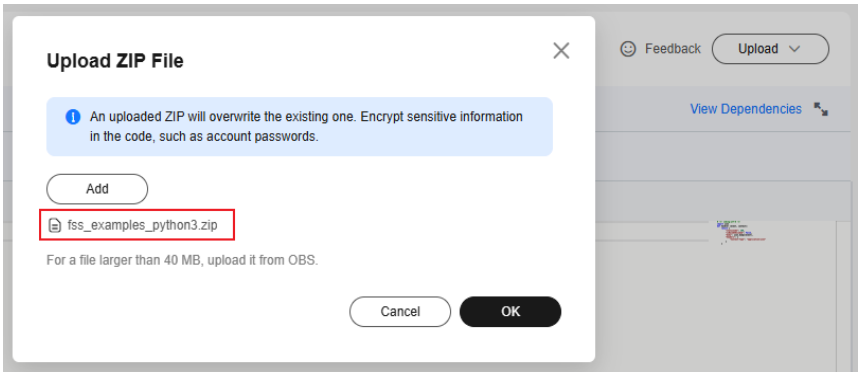
Figure 3-3 Creating a Python function



3. Upload the ZIP file packed in **Step 1: Creating a Python Function Project** on the **Code** tab, as shown in **Figure 3-4**.

   The uploaded code will be automatically deployed on the FunctionGraph console. If you have modified the code, click **Deploy** again.

Figure 3-4 Uploading a ZIP file

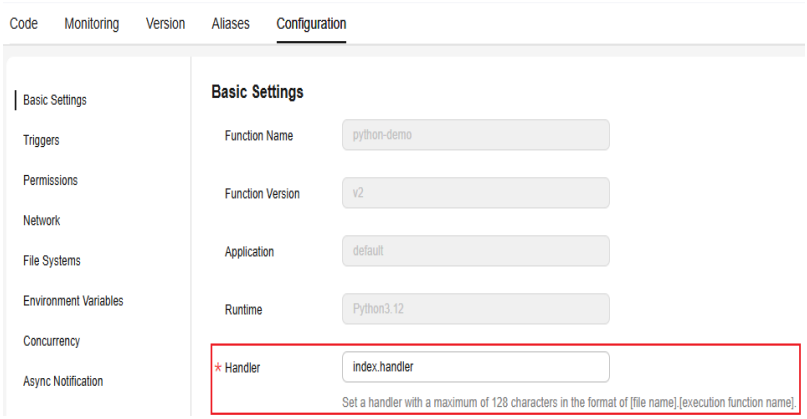


NOTE

Modifying the function handler:

In the navigation pane on the left of the FunctionGraph console, choose **Functions** > **Function List**. Click the name of the function to be set. On the function details page that is displayed, choose **Configuration** > **Basic Settings** and set the **Handler** parameter, as shown in **Figure 3-5**.

Figure 3-5 Function handler



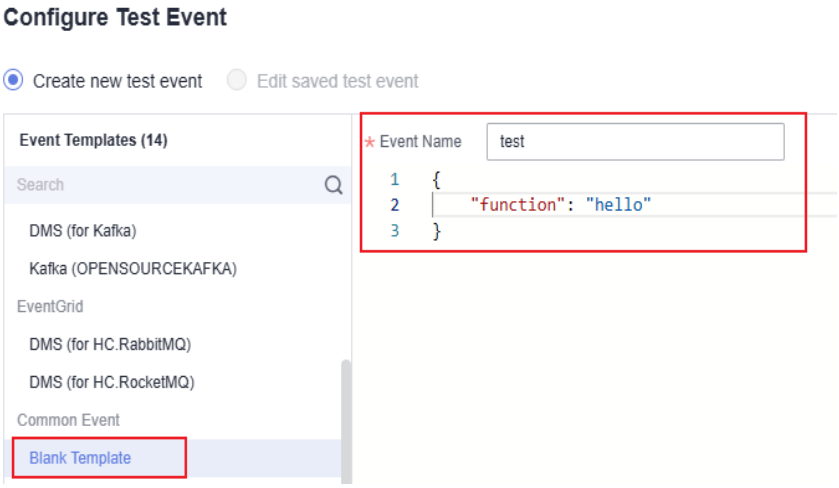
- The **index** of the handler must be consistent with the name of the file created in **Step 1: Creating a Python Function Project**, because the file name will help to locate the function file.
- The **handler** is a function name. It is used to find the handler of the function.

After you upload the **fss_examples_python3.zip** file to OBS, when the function is triggered, FunctionGraph decompresses the file to locate the function file through **index** and locate the function defined in the **index.py** file through **handler**, and then executes the function.

Step 3: Testing the Function

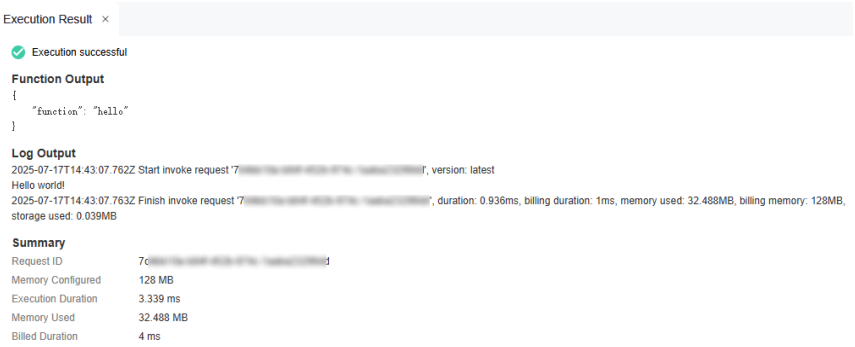
1. On the **Code** tab, click **Test**. In the displayed **Configure Test Event** dialog box, select **Blank Template**. Configure the test event **test** as shown in **Figure 3-6**. Click **Create**.

Figure 3-6 Configuring a test event



2. Select the configured test event **test** and click **Test**.
3. As shown in **Figure 3-7**, the **Execution Result** window is displayed on the right. You can check whether the function is executed successfully.

Figure 3-7 Test result



Function Execution Result

The execution result consists of the function output, summary, and log output.

Table 3-3 Description of the execution result

Parameter	Successful Execution	Failed Execution

Function Output	The defined function output information is returned.	A JSON file that contains **errorMessage**, **errorType**, and **stackTrace** is returned. The format is as follows:
		`{`
		`  "errorMessage": "",`
		`  "errorType": "",`
		`  "stackTrace": []`
		`}`
		errorMessage: Error message returned by the runtime.
		errorType: Error type.
		stackTrace: Stack error information returned by the runtime.

Summary	**Request ID**, **Memory Configured**, **Execution Duration**, **Memory Used**, and **Billed Duration** are displayed.	**Request ID**, **Memory Configured**, **Execution Duration**, **Memory Used**, and **Billed Duration** are displayed.
Log Output	Function logs are printed. A maximum of 4 KB logs can be displayed.	Error information is printed. A maximum of 4 KB logs can be displayed.

Helpful Links

- For details about how to create a Python function dependency, see **Creating a Dependency for a Python Function**.
- For more information about function development, such as the supported runtimes, trigger events, function project packaging specifications, and DLL referencing, see **Function Development Overview**.

3.3 Creating an Event Function Using a Container Image Built with Python

For details about how to use a container image to create and execute an event function, see **Creating an Event Function Using a Container Image and Executing the Function**. This section describes how to create an image using Python and verify the image locally.

Step 1: Creating an Image

Take the Linux x86 64-bit OS as an example. (There are no specific requirements for system configurations.)

1. Run the following command to create an empty folder named **custom_container_event_example**:
   ```
   mkdir custom_container_event_example && cd custom_container_event_example
   ```
2. Use Python to implement an HTTP server to process **init** and **invoke** requests and send responses.

   Run the following command to create a **main.py** file:
   ```
   touch main.py
   ```
   Introduce the Flask framework in the code and implement a function handler (method **POST** and path **/invoke**) and an initializer (method **POST** and path **/init**). Here is the code for **main.py**:
   ```
   import json

   from flask import Flask, request

   # Create a Flask application instance.
   app = Flask(__name__, template_folder='templates', static_folder='static')

   # Define the initialization API.
   @app.route('/init', methods=['POST'])
   def init():
       # Print the request path for debugging.
       print("***" + request.path + "***", flush=True)

       # Build response data.
       data = {
           "statusCode": 200,
           "isBase64Encoded": False,
           "body": json.dumps("init successes"),
           "headers": {
               "Content-Type": "application/json"
           }
       }
       return json.dumps(data)

   # Define the API.
   @app.route('/invoke', methods=['POST'])
   def invoke():
       print("***" + request.path + "***", flush=True)
       data = {
           "statusCode": 200,
           "isBase64Encoded": False,
           "body": json.dumps("invoke successes"),
           "headers": {
               "Content-Type": "application/json"
           }
       }
       return json.dumps(data)

   # Main program entry
   if __name__ == '__main__':
       app.run(host="0.0.0.0", port=8000)
   ```
3. Create a Dockerfile.
   ```
   touch Dockerfile
   ```
   The content of Dockerfile is as follows:
   ```
   FROM ubuntu:22.04

   ENV HOME=/home/custom_container
   ENV GROUP_ID=1003
   ENV GROUP_NAME=custom_container
   ENV USER_ID=1003
   ENV USER_NAME=custom_container

   RUN mkdir -m 550 ${HOME} && groupadd -g ${GROUP_ID} ${GROUP_NAME} && useradd -u ${USER_ID} -g ${GROUP_ID} ${USER_NAME}
   ```



```
RUN apt-get update &&  \
    apt-get install -y --no-install-recommends python3 pip &&  \
    apt-get clean

RUN pip3 install --verbose flask jsons requests --no-cache-dir

COPY main.py ${HOME}

RUN chown -R ${USER_ID}:${GROUP_ID} ${HOME}
RUN chmod -R 775 ${HOME}

USER ${USER_NAME}
WORKDIR ${HOME}
EXPOSE 8000
ENTRYPOINT ["python3", "main.py"]
```

Table 3-4 Instructions

Instruction	Description
FROM	Specifies base image **ubuntu:22.04**. The base image is mandatory and its value can be changed.
ENV	Sets environment variables **HOME** (**/home/custom_container**), **GROUP_NAME** and **USER_NAME** (**custom_container**), and **USER_ID** and **GROUP_ID** (**1003**). These environment variables are mandatory and their values can be changed.
RUN	Executes commands. The format is **RUN** *<command>*. For example, **RUN mkdir -m 550 ${HOME}** means to create directory **${HOME}** for user *${USER_NAME}* during container building.
COPY	Copies files or directories from the build context to the image. Copy **main.py** to the **${HOME}** directory of user *${USER_NAME}* in the container.
USER	Switches to user *${USER_NAME}*.
WORKDIR	Switches the working directory to the **${HOME}** directory of user *${USER_NAME}*.
EXPOSE	Informs Docker that the container listens on the specified network ports at runtime. Expose port 8000 of the container and do not change it.
ENTRYPOINT	Sets the executable command that is run each time a container starts. Run the **python3 main.py** command to start a container.

4. Run the following command to build an image:

```
docker build -t custom_container_event_example:latest .
```

In the preceding command, the image name is **custom_container_event_example**, the tag is **latest**, and the period (.) indicates the directory where the Dockerfile is located. Run the image build command to pack all files in the directory and send the package to a container engine to build an image.

Step 2: Performing Local Verification

1. Run the following command to start the Docker container:
   ```
   docker run -u 1003:1003 -p 8000:8000 custom_container_event_example:latest
   ```
2. Open a new Command Prompt, and send a message through port 8000 to access the **/init** directory specified in the template code.
   ```
   curl -XPOST -H 'Content-Type: application/json' localhost:8000/init
   ```
   The following information is returned based on the module code:
   ```
   {
       "statusCode": 200,
       "isBase64Encoded": False,
       "body": json.dumps("init successes"),
       "headers": {
           "Content-Type": "application/json"
       }
   }
   ```
3. Open a new Command Prompt, and send a message through port 8000 to access the **/invoke** directory specified in the template code.
   ```
   curl -XPOST -H 'Content-Type: application/json' -d '{"message":"HelloWorld"}' localhost:8000/invoke
   ```
   The following information is returned based on the module code:
   ```
   {
       "statusCode": 200,
       "isBase64Encoded": False,
       "body": json.dumps("invoke successes"),
       "headers": {
           "Content-Type": "application/json"
       }
   }
   ```

Step 3: Creating a Function

Once you build the container image locally, you can create a function on the console.

For details, see **Creating an Event Function Using a Container Image and Executing the Function**. Start from **Step 3: Upload the Image**.

Helpful Links

- For more information about function development, such as the supported runtimes, trigger events, function project packaging specifications, and DLL referencing, see **Function Development Overview**.
- For details about the syntax and SDK APIs of function development in Python, see **Function Development Overview**.

3.4 Creating an HTTP Function Using a Container Image Built with Python

For details about how to use a container image to create and execute an HTTP function, see **Creating an HTTP Function Using a Container Image and Executing the Function**. This section describes how to create an image using Python and verify the image locally.

Step 1: Creating an Image

Take the Linux x86 64-bit OS as an example. (There are no specific requirements for system configurations.)

1. Run the following command to create a folder:
   ```
   mkdir custom_container_http_example && cd custom_container_http_example
   ```
2. Use Python to implement an HTTP server to process HTTP requests and send responses.

   Run the following command to create a **main.py** file:

   ```
   touch main.py
   ```

   Introduce the Flask framework in the code and implement a function handler **index** (method **POST**). The handler can be compiled based on the actual service requirements. Here is the code for **main.py**:

   ```
   import json

   from flask import Flask, request

   # Create a Flask application instance.
   app = Flask(__name__, template_folder='templates', static_folder='static')

   # Define a route /index that handles POST requests.
   @app.route('/index', methods=['POST'])
   def index():
       # Print the request path for debugging.
       print("***" + request.path + "***", flush=True)

       # Build response data.
       data = {
           "statusCode": 200,
           "isBase64Encoded": False,
           "body": json.dumps(request.path + " success"),
           "headers": {
               "Content-Type": "application/json"
           }
       }
       return json.dumps(data)

   # Main program entry
   if __name__ == '__main__':
       app.run(host="0.0.0.0", port=8000)
   ```
3. Create a Dockerfile with the following content:
   ```
   FROM ubuntu:22.04

   ENV HOME=/home/custom_container
   ENV GROUP_ID=1003
   ENV GROUP_NAME=custom_container
   ENV USER_ID=1003
   ENV USER_NAME=custom_container

   RUN mkdir -m 550 ${HOME} && groupadd -g ${GROUP_ID} ${GROUP_NAME} && useradd -u ${USER_ID} -g ${GROUP_ID} ${USER_NAME}

   RUN apt-get update &&  \
       apt-get install -y --no-install-recommends python3 pip &&  \
       apt-get clean

   RUN pip3 install --verbose flask jsons requests --no-cache-dir

   COPY main.py ${HOME}

   RUN chown -R ${USER_ID}:${GROUP_ID} ${HOME}
   RUN chmod -R 775 ${HOME}
   ```



```
USER ${USER_NAME}
WORKDIR ${HOME}
EXPOSE 8000
ENTRYPOINT ["python3", "main.py"]
```

Table 3-5 Instructions

Instruction	Description
FROM	Specifies base image **ubuntu:22.04**. The base image is mandatory and its value can be changed.
ENV	Sets environment variables **HOME** (**/home/custom_container**), **GROUP_NAME** and **USER_NAME** (**custom_container**), and **USER_ID** and **GROUP_ID** (**1003**). These environment variables are mandatory and their values can be changed.
RUN	Executes commands. The format is **RUN** *<command>*. For example, **RUN mkdir -m 550 ${HOME}** means to create directory **${HOME}** for user *${USER_NAME}* during container building.
COPY	Copies files or directories from the build context to the image. Copy **main.py** to the **${HOME}** directory of user *${USER_NAME}* in the container.
USER	Switches to user *${USER_NAME}*.
WORKDIR	Switches the working directory to the **${HOME}** directory of user *${USER_NAME}*.
EXPOSE	Informs Docker that the container listens on the specified network ports at runtime. Expose port 8000 of the container and do not change it.
ENTRYPOINT	Sets the executable command that is run each time a container starts. Run the **python3 main.py** command to start a container.

4. Run the following command to build an image:
   ```
   docker build -t custom_container_http_example:latest .
   ```
   In the preceding command, the image name is **custom_container_http_example**, the tag is **latest**, and the period (.) indicates the directory where the Dockerfile is located. Run the image build command to pack all files in the directory and send the package to a container engine to build an image.

Step 2: Performing Local Verification

1. Run the following command to start the Docker container:
   ```
   docker run -u 1003:1003 -p 8000:8000 custom_container_http_example:latest
   ```
2. Open a new Command Prompt, and send a message through port 8000 to access the **/init** directory specified in the template code.
   ```
   curl -XPOST -H 'Content-Type: application/json' localhost:8000/index
   ```
   The following information is returned based on the module code:
   ```
   index successes
   ```


Step 3: Creating a Function

Once you build the container image locally, you can create a function on the console.

For details, see **Creating an HTTP Function Using a Container Image and Executing the Function**. Start from **Step 3: Upload the Image**.

Helpful Links

- For more information about function development, such as the supported runtimes, trigger events, function project packaging specifications, and DLL referencing, see **Function Development Overview**.
- For details about the syntax and SDK APIs of function development in Python, see **Function Development Overview**.

3.5 Python Function Template

Python Function

The following is a sample code template of a Python function:

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

When you create an empty Python event function on the FunctionGraph console, the preceding sample code is deployed by default.

3.6 Creating a Dependency for a Python Function

You are advised to create function dependencies in Huawei Cloud EulerOS 2.0. If other OSs are used, the dynamic link library may not be found due to the differences between underlying dependency libraries.

Constraints

If the modules to be installed need dependencies such as **.dll**, **.so**, and **.a**, archive them to a **.zip** package.

Setting Up the EulerOS Environment

You are advised to create function dependencies in EulerOS. EulerOS is an enterprise-grade Linux OS based on open-source technology. It features high security, scalability, and performance, meeting customers' requirements for IT infrastructure and cloud computing services.

You can set up the **Huawei Cloud EulerOS** environment using the following methods:

- Buy a EulerOS ECS on Huawei Cloud by referring to **Purchasing and Logging In to a Linux ECS**. On the **Configure Basic Settings** page, select **Public Image**, and select **Huawei Cloud EulerOS** and an image version.
- Download the EulerOS image, and use virtualization software to set up the EulerOS VM on a local PC.

Creating a Dependency for a Python Function

Before creating a dependency, ensure that Python matching the function runtime has been installed in the environment.

The following uses Python 3.12 as an example to describe how to create a PyMySQL dependency.

Step 1 Run the following command to install the PyMySQL dependency to the local **/tmp/pymysql** directory.

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

Step 2 Run the following command to access the specified directory.

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

Step 3 Go to the **site-packages** directory (generally **lib/python3.12/site-packages/**). If no dependency file exists in this directory, run the **find** command to find the file and go to its directory. Then run the following command to compress the dependency file.

The required dependency is generated.

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

----End

NOTE

To install the local wheel installation package, run the following commands:

```
pip install piexif-1.1.0b0-py2.py3-none-any.whl --root /tmp/piexif
//Replace piexif-1.1.0b0-py2.py3-none-any.whl with the actual installation package name.
```

Helpful Links

- For details about how to use Python to develop an event function, see **Developing a Python Event Function**.
- For details about how to create a Python function dependency, see **Creating a Dependency for a Python Function**.
- For more information about function development, such as the supported runtimes, trigger events, function project packaging specifications, and DLL referencing, see **Function Development Overview**.

3.7 Developing a Python Function Using the Huawei Cloud SDK

Huawei Cloud **API Explorer** provides API reference documents and SDK code examples for each cloud service.

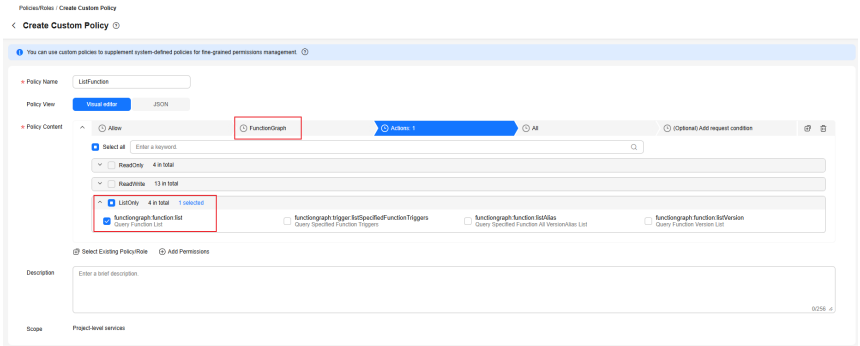
This section uses the API for querying the function list as an example to describe how to develop a Python function on the FunctionGraph console using Huawei Cloud SDKs.

Step 1: Creating a Function Agency

1. Log in to the **IAM console**.
2. In the navigation pane on the left, choose **Permissions** > **Policies/Roles**. On the displayed page, click **Create Custom Policy** in the upper right corner.
3. Take the agency to be created for the API used to query the function list as an example. Configure a custom policy that contains the permission for querying the function list, as shown in **Figure 3-8**, and click **OK**.

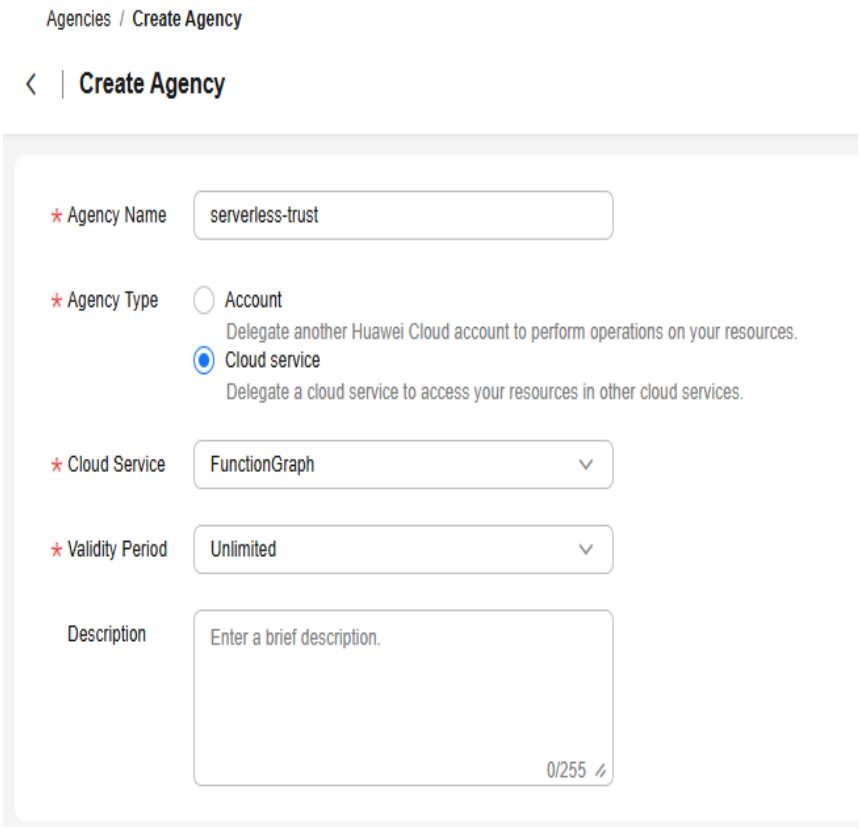
   For more information about permissions, see **Basic Concepts About Permissions**.

   Figure 3-8 Custom policy for querying a function list

4. In the navigation pane of the IAM console, choose **Agencies**. Then, click **Create Agency** in the upper right corner.
5. Configure agency parameters. After the parameters are configured, as shown in **Figure 3-9**, click **OK**. The system displays a message indicating that the creation is successful. Click **Authorize**.

Figure 3-9 Entering basic information

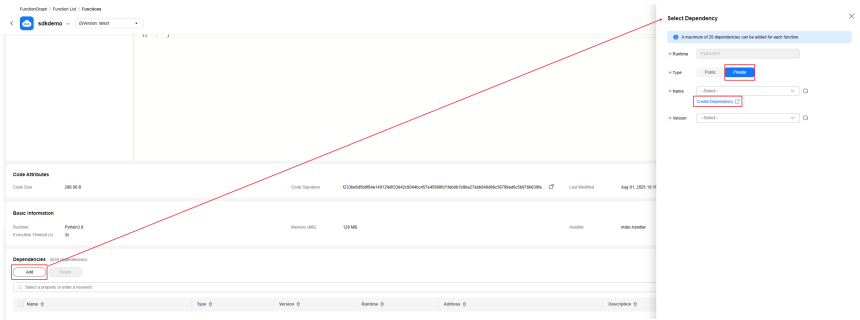


6. On the displayed page, select the custom policy created in 3, click **Next**, select the authorization scope based on the actual needs, and click **OK**.

Step 2: Creating a Python Function

1. Log in to the **FunctionGraph console** and click **Create Function** in the upper right corner.
2. Create a Python event function from scratch, select the agency created in **Step 1: Creating a Function Agency**, select the latest runtime version, and click **Create Function**.
3. On the **Code** tab, scroll down to the **Dependencies** area and click **Add**.
4. Select **Private** for **Type** and click **Create Dependency** as shown in **Figure 3-10**. The dependency creation page is displayed.

Figure 3-10 Selecting dependencies



5. Create the required Node.js dependency for the function by referring to **Huawei Cloud Python SDK** and **Creating a Dependency**.

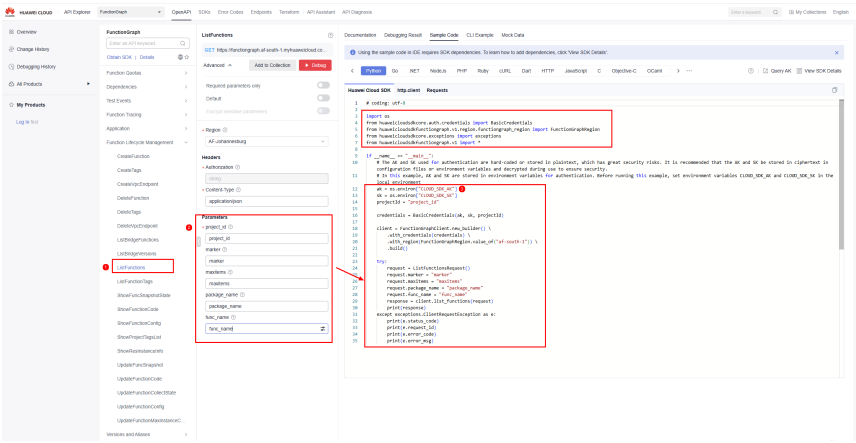
   Note that the runtime version of the dependency must be the same as that of the Python function.

6. After the dependency is created, return to **4** and add the created dependency.

Step 3: Obtaining SDK Sample Code from APIE

Step 1 Open **API Explorer**, select the required API, click the **Sample Code** tab, and select the **Python** language, as shown in **Figure 3-11**.

Figure 3-11 APIE sample code



1. The API for querying functions is used as an example.
2. Enter the parameters required by the API. For details about the parameters, see the corresponding section in the API reference. In this example, see **Querying Functions**.
3. Copy the code generated by API Explorer and paste it in the code editing box of the function created in **Step 2: Creating a Python Function**.

Step 2 You are advised to configure the AK/SK in the function environment variables. For details, see **Configuring Environment Variables**. And you can use the context.getUserData(string key) method to obtain the AK/SK in the code.

The modified code is as follows:

```
# -*- coding:utf-8 -*-
import json
import os
from huaweicloudsdkcore.auth.credentials import BasicCredentials
from huaweicloudsdkfunctiongraph.v2.region.functiongraph_region import FunctionGraphRegion
from huaweicloudsdkcore.exceptions import exceptions
from huaweicloudsdkfunctiongraph.v2 import *
def handler (event, context):
    ak = context.getUserData("AK")
    sk = context.getUserData("SK")
    projectId = "project_id"
    credentials = BasicCredentials(ak, sk, projectId)
    client = FunctionGraphClient.new_builder() \
        .with_credentials(credentials) \
        .with_region(FunctionGraphRegion.value_of("cn-north-4")) \
        .build()
```



```
try:
    request = ListFunctionsRequest()
    request.marker = "marker"
    request.maxitems = "maxitems"
    request.package_name = "package_name"
    request.func_name = "func_name"
    response = client.list_functions(request)
    print(response)
except exceptions.ClientRequestException as e:
    print(e.status_code)
    print(e.request_id)
    print(e.error_code)
    print(e.error_msg)
return {
    "statusCode": 200,
    "isBase64Encoded": False,
    "body": json.dumps(event),
    "headers": {
        "Content-Type": "application/json"
    }
}
```

Step 3 (Optional) To use a more secure authentication mode, replace the following code:

```
ak = context.getUserData("AK")
sk = context.getUserData("SK")
credentials = BasicCredentials(ak, sk, projectId)
```

with

```
ak = context.getSecurityAccessKey()
sk = context.getSecuritySecretKey()
st = context.getSecurityToken()
credentials = BasicCredentials(ak, sk, projectId).with_security_token(st)
```

----End

4 Java

4.1 Function Development Overview

FunctionGraph supports the following Java runtimes:

- Java 8
- Java 11
- Java 17
- Java 21 (only available in **ME-Riyadh** and **TR-Istanbul**)

Function Syntax

Java function syntax: *Scope Return parameter Function name (User-defined parameter, Context)*

- *Scope*: It must be defined as **public** for the function that FunctionGraph invokes to execute your code.
- *Return parameter*: user-defined output, which is converted into a character string and returned as an HTTP response. The HTTP response is a JSON string.
- *Function name*: user-defined function name.
- *User-defined parameter*: FunctionGraph supports only one user-defined parameter. For complex parameters, define them as an object and provide data through JSON strings. When invoking a function, FunctionGraph parses the JSON strings as an object.
- *Context*: runtime information provided for executing the function. For details, see **SDK APIs**.

The Java function handler is in the format of *[package name].[class name].[function name]*. Configure or modify handler parameters by referring to **function handler**.

Java Initializer

For details about the initializer, see **Initializer**.

Initializer format: *[Package name].[Class name].[Execution function name]*

For example, if the initializer is named **com.huawei.Demo.my_initializer**, FunctionGraph loads the my_initializer function defined in the **com.huawei** file.

To use Java to build initialization logic, define a Java function as the initializer. The following is a simple initializer:

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

- Function name

  The function name **my_initializer** must be the initializer function name specified for a function.

- context

  The **context** parameter contains the runtime information about a function. For example, request ID, temporary AK, and function metadata.

SDK APIs

The **Java SDK** (verification file: **fss-java-sdk-2.0.5.sha256**) provides context, event, and logging APIs.

- Event APIs

  Event structure definitions are added to the Java SDK. Currently, DMS, DIS, SMN, timer, APIG, and Kafka triggers are supported. The definitions make coding much simpler when triggers are required.

  a. **APIG trigger**

  - APIGTriggerEvent methods

    Table 4-1 APIGTriggerEvent methods

Method	Description
isBase64Encoded()	Checks whether the body of an event is encoded using Base64.
getHttpMethod()	Obtains the HTTP request method.
getPath()	Obtains the HTTP request path.
getBody()	Obtains the HTTP request body.
getPathParameters()	Obtains all path parameters.
getRequestContext()	Obtains APIG configuration information. (The **APIGRequestContext object** is returned.)
getHeaders()	Obtains the HTTP request header.

Method	Description
getQueryStringParameters()	Obtains query parameters. **The value of a query parameter cannot be an array. To support an array, customize the corresponding event structure.**
getRawBody()	Obtains the content before Base64 encoding.
getUserData()	Obtains the user data set in the APIG custom authorizer.

Table 4-2 APIGRequestContext methods

Method	Description
getApiId()	Obtains the API ID.
getRequestId()	Obtains the request ID of an API request.
getStage()	Obtains the name of the environment in which an API has been published.
getSourceIp()	Obtains the source IP address in the APIG custom authorizer.

- APIGTriggerResponse methods

Table 4-3 APIGTriggerResponse construction methods

Method	Description
APIGTriggerResponse()	Set the following parameters: **headers**: null **statusCode**: 200 **body**: "" **isBase64Encoded**: false
APIGTriggerResponse(statusCode, headers, body)	Set the value of **isBase64Encoded** to **false**, and use the input values of other parameters.
APIGTriggerResponse(statusCode, headers, isBase64Encoded, body)	Set the parameters in sequence.

Table 4-4 APIGTriggerResponse methods

Method	Description
setBody(String body)	Sets the message body.
setHeaders(Map<String,String> headers)	Sets the HTTP response header to be returned.
setStatusCode(int statusCode)	Sets the HTTP status code.
setBase64Encoded(boolean isBase64Encoded)	Configures Base64 encoding for the response body.
setBase64EncodedBody(String body)	Encodes the input with Base64 and configures it in the body.
addHeader(String key, String value)	Adds a group of HTTP headers.
removeHeader(String key)	Removes the specified header.
addHeaders(Map<String,String> headers)	Adds multiple headers.

NOTE

APIGTriggerResponse methods have the **headers** attribute, which can be initialized using the setHeaders method or a constructor function with the **headers** parameter.

b. **DIS trigger**

Table 4-5 DISTriggerEvent methods

Method	Description
getShardID()	Obtains the partition ID.
getMessage()	Obtains the DIS message body (**DISMessage structure**).
getTag()	Obtains the version of a function.
getStreamName()	Obtains the stream name.

Table 4-6 DISMessage methods

Method	Description
getNextPatitionCursor()	Obtains the next partition cursor.
getRecords()	Obtains message records (**DISRecord structure**).

Method	Description
getMillisBehindLatest()	Reserved (Currently, **0** is returned.)

Table 4-7 DISRecord methods

Method	Description
getPartitionKey()	Obtains the data partition.
getData()	Obtains data.
getRawData()	Obtains UTF-8 data strings decoded using Base64.
getSequenceNumber()	Obtains the sequence number (ID of each record).

c. **DMS trigger**

Table 4-8 DMSTriggerEvent methods

Method	Description
getQueueId()	Obtains the queue ID.
getRegion()	Obtains the region name.
getEventType()	Obtains the event type. ("MessageCreated" is returned.)
getConsumerGroupId()	Obtains the consumer group ID.
getMessages()	Obtains DMS messages (**DMSMessage structure**).

Table 4-9 DMSMessage methods

Method	Description
getBody()	Obtains the DMS message body.
getAttributes()	Obtains the message attribute set.

d. **SMN trigger**

Table 4-10 SMNTriggerEvent method

Method	Description
getRecord()	Obtains the message records (**SMNRecord structure**).

Table 4-11 SMNRecord methods

Method	Description
getEventVersion()	Obtains event version. (The current version is 1.0.)
getEventSubscriptionUrn()	Obtains the subscription Uniform Resource Name (URN).
getEventSource()	Obtains the event source.
getSmn()	Obtains the message body (**SMNBody structure**).

Table 4-12 SMNBody methods

Method	Description
getTopicUrn()	Obtains the topic URN.
getTimeStamp()	Obtains the timestamp of a message.
getMessageAtrributes()	Obtains the message attribute set.
getMessage()	Obtains the message body.
getType()	Obtains the message type.
getMessageId()	Obtains the message ID.
getSubject()	Obtains the message topic.

e. **Timer trigger**

Table 4-13 TimerTriggerEvent methods

Method	Description
getVersion()	Obtains the version name. (The current version is 1.0.)
getTime()	Obtains the current time.
getTriggerType()	Obtains trigger type (Timer).

Method	Description
getTriggerName()	Obtains the trigger name.
getUserEvent()	Obtains the additional information of the trigger.

f. **Kafka trigger**

Table 4-14 Kafka trigger

Method	Description
getEventVersion	Obtains event version.
getRegion	Obtains region information.
getEventTime	Obtains event time.
getTriggerType	Obtains trigger type
getInstanceId	Obtains the instance ID.
getRecords	Obtains record.

NOTE

1. When using an APIG trigger, set the first parameter of the handler function (for example, **handler**) to **handler(APIGTriggerEvent event, Context context)**.
2. The preceding TriggerEvent methods have corresponding **set** methods, which are recommended for local debugging. DIS and LTS triggers have getRawData() methods, but do not have setRawData() methods.

- Context APIs

  The context APIs are used to obtain the context, such as agency AK/SK, current request ID, allocated memory space, and number of CPUs, required for executing a function.

  Table 4-15 describes the context APIs provided by FunctionGraph.

  Table 4-15 Context methods

Method	Description
getRequestID( )	Obtains a request ID.
getRemainingTimeIn-MilliSeconds ( )	Obtains the remaining running time of a function.
getAccessKey( )	Obtains the AK (valid for 24 hours) with an agency. If you use this method, you need to configure an agency for the function. **FunctionGraph has stopped maintaining the getAccessKey API in the Runtime SDK. You cannot use this API to obtain a temporary AK.**

Method	Description
getSecretKey( )	Obtains the SK (valid for 24 hours) with an agency. If you use this method, you need to configure an agency for the function. **FunctionGraph has stopped maintaining the getSecretKey API in the Runtime SDK. You cannot use this API to obtain a temporary SK.**
getSecurityAccessKey( )	Obtains the SecurityAccessKey (valid for 24 hours) with an agency. The cache duration is 10 minutes. That is, the same content is returned within 10 minutes. To use this method, you need to configure an agency for the function.
getSecuritySecretKey( )	Obtains the SecuritySecretKey (valid for 24 hours) with an agency. The cache duration is 10 minutes. That is, the same content is returned within 10 minutes. To use this method, you need to configure an agency for the function.
getSecurityToken( )	Obtains the SecurityToken (valid for 24 hours) with an agency. The cache duration is 10 minutes. That is, the same content is returned within 10 minutes. To use this method, you need to configure an agency for the function.
getUserData(string key)	Uses keys to obtain the values passed by environment variables.
getFunctionName( )	Obtains the name of a function.
getRunningTimeInSeconds ( )	Obtains the timeout of a function.
getVersion( )	Obtains the version of a function.
getMemorySize( )	Obtains the allocated memory.
getCPUNumber( )	Obtains CPU usage of a function.
getPackage( )	Obtains a function group.
getToken( )	Obtains the token (valid for 24 hours) with an agency. If you use this method, you need to configure an agency for the function.
getLogger( )	Obtains the **logger** method provided by the context. By default, information such as the time and request ID is output.
getAlias()	Obtains function alias.

- Logging APIs

  Table 4-16 describes the logging API provided in the Java SDK.

Table 4-16 Logging API

Method	Description
RuntimeLogger()	Records user input logs. Method: log(String string).

4.2 Developing a Java Event Function

4.2.1 Developing Functions in Java (Using IDEA to Create a Java Project)

This section describes how to use IDEA to develop Java functions. For details about the Java syntax, initializer and SDK APIs, see **Function Development Overview**.

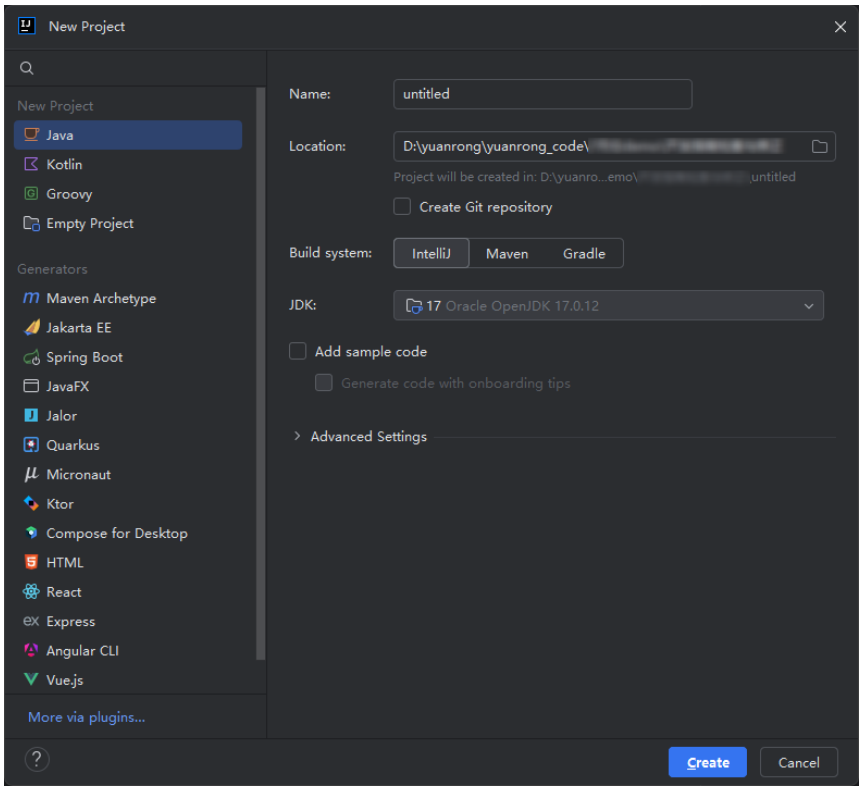
Procedure

You can create a Java project and Java function following the steps in this section, or download the **sample project package** and start from **Step 3: Creating and Testing a Java Function**.

Step 1: Creating a Java Project Using IDEA

1. Configure the IDEA.
   Create a Java project, as shown in **Figure 4-1**.

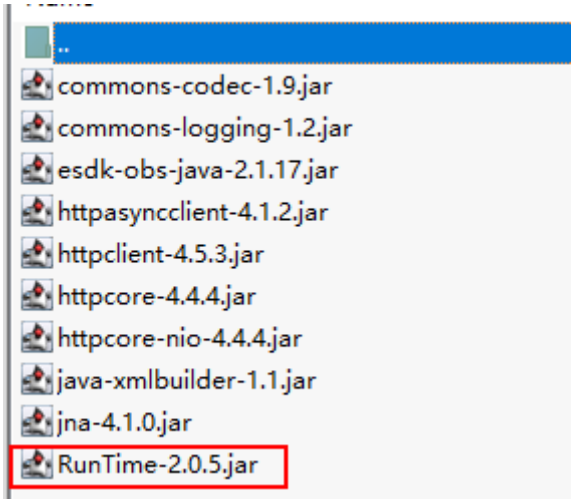
   Figure 4-1 Creating a project



2. Add dependencies to the project.

   Download the **Java SDK** to a local development environment, and decompress the SDK package, as shown in **Figure 4-2**.

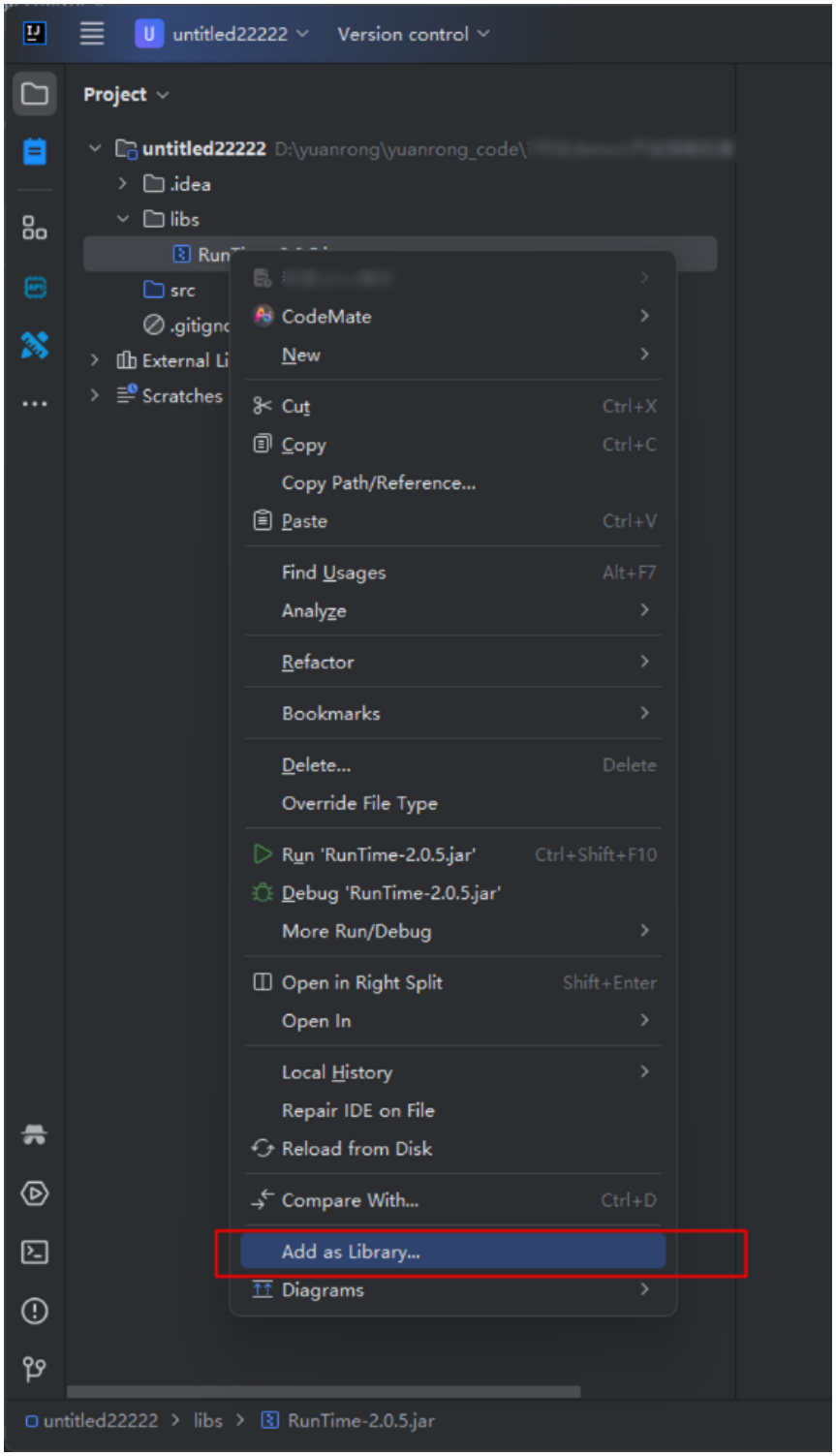
   Figure 4-2 Decompressing the downloaded SDK

3. Configure dependencies.

   Create a folder named **lib** in the project directory, copy the **Runtime2.0.5.jar** file and other required dependencies to the **lib** folder, and add the JAR files as the dependencies of the project, as shown in **Figure 4-3**.

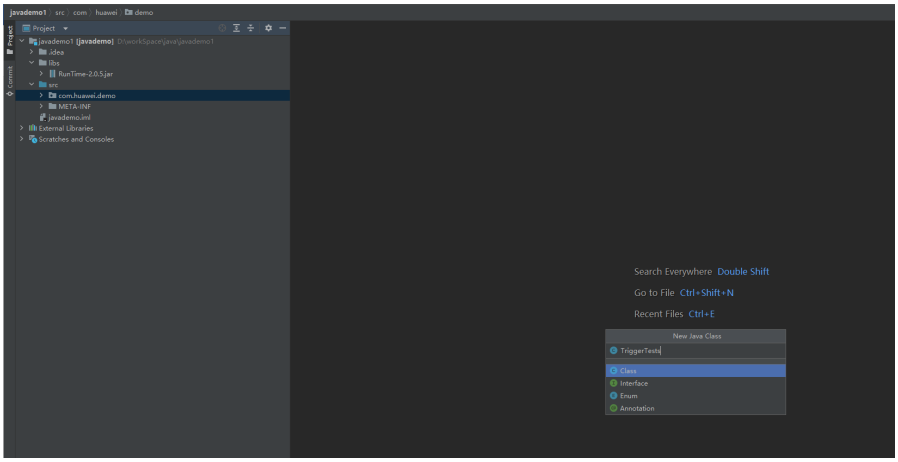
Figure 4-3 Configuring dependencies



4. Configure the function resources.

   Create a package named **com.huawei.demo**, and then create a class named **TriggerTests** under the package, as shown in **Figure 4-4**.

Figure 4-4 Creating the TriggerTests class



5. Configure the function code.

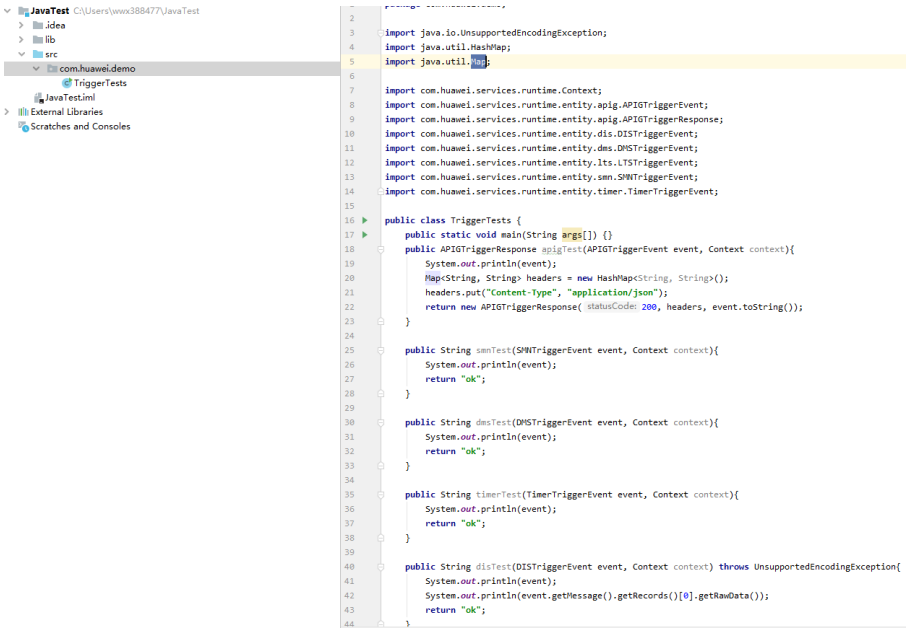
   Define the function handler in **TriggerTests.java** as shown in **Figure 4-5**. The sample code is as follows: **A common Java project needs to be compiled using artifacts, and a main function needs to be defined.**

```
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;
import com.huawei.services.runtime.entity.eventgrid.EventGridTriggerEvent;

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";
    }

    public String eventgridTest(EventGridTriggerEvent event, Context context){
        System.out.println(event);return "ok";
    }
}
```

Figure 4-5 Defining the function handler



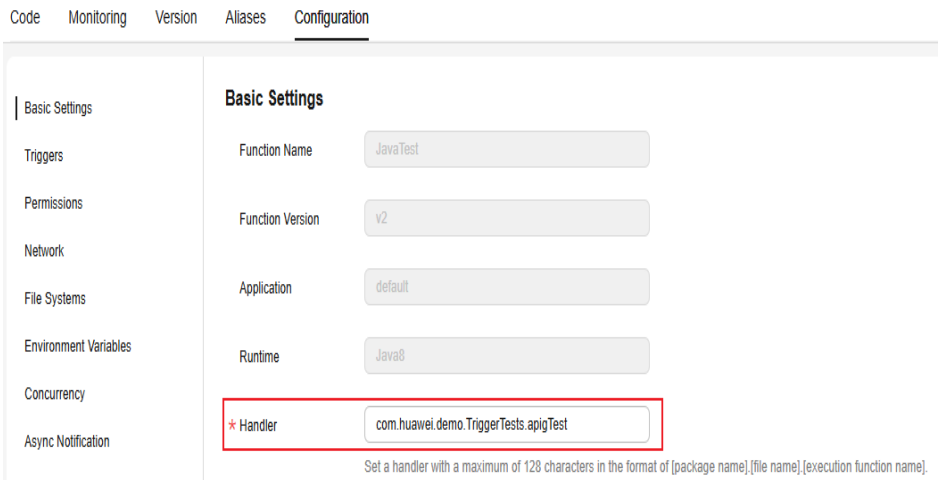
There are multiple handler functions with different trigger event types. You can modify the handler on the FunctionGraph console to test different handler functions. **For details about the constraints for the APIG event source, see Base64 Decoding and Response Structure.**

NOTE

Modifying the function handler:

In the navigation pane on the left of the FunctionGraph console, choose **Functions** > **Function List**. Click the name of the function to be set. On the function details page that is displayed, choose **Configuration** > **Basic Settings** and set the **Handler** parameter, as shown in **Figure 4-6**.

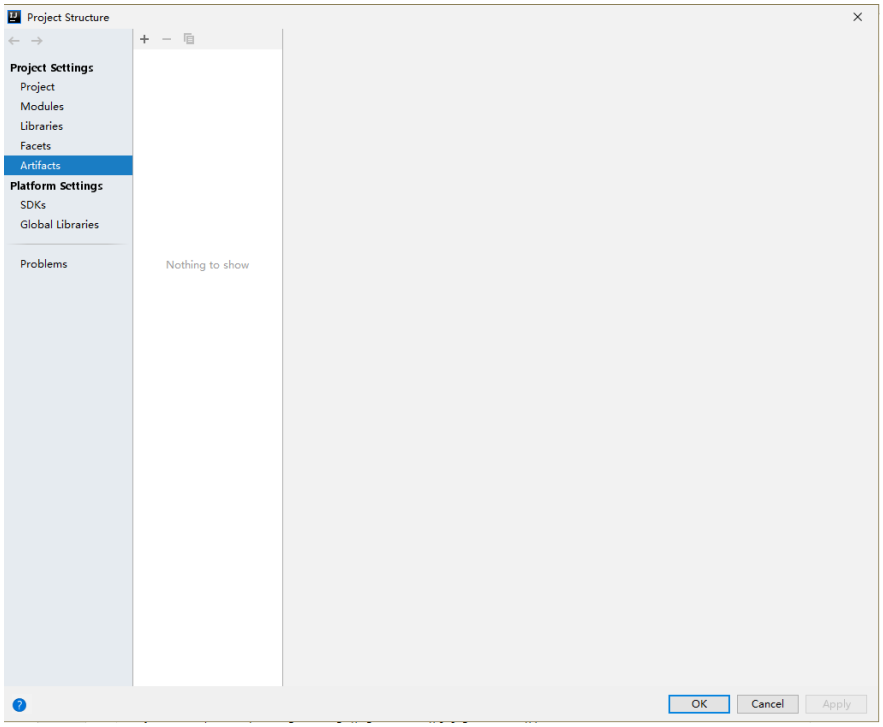
Figure 4-6 Function handler



Step 2: Packaging a Java Project

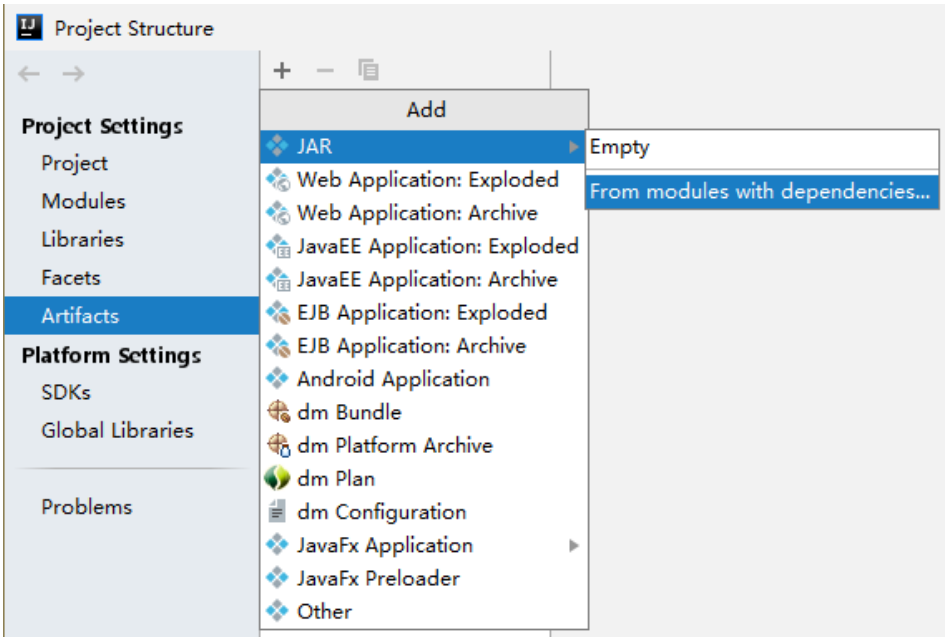
1. Choose **File** > **Project Structure**. The **Project Structure** page is displayed, as shown in **Figure 4-7**.

Figure 4-7 Going to the Project Structure page



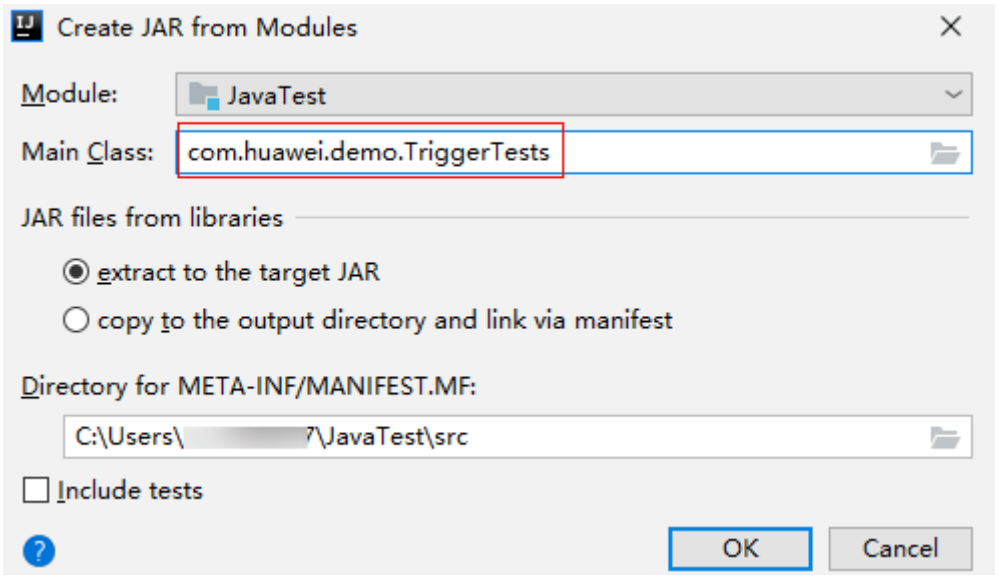
2. Choose **Artifacts** and click **+** to add artifacts, as shown in **Figure 4-8**.

Figure 4-8 Adding artifacts



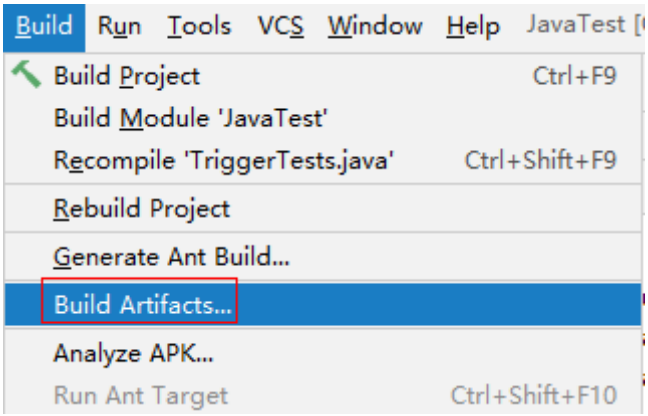
3. Add a main class, as shown in **Figure 4-9**.

Figure 4-9 Adding a main class



4. Choose **Build** > **Build Artifacts** to compile the JAR file, as shown in **Figure 4-10**.

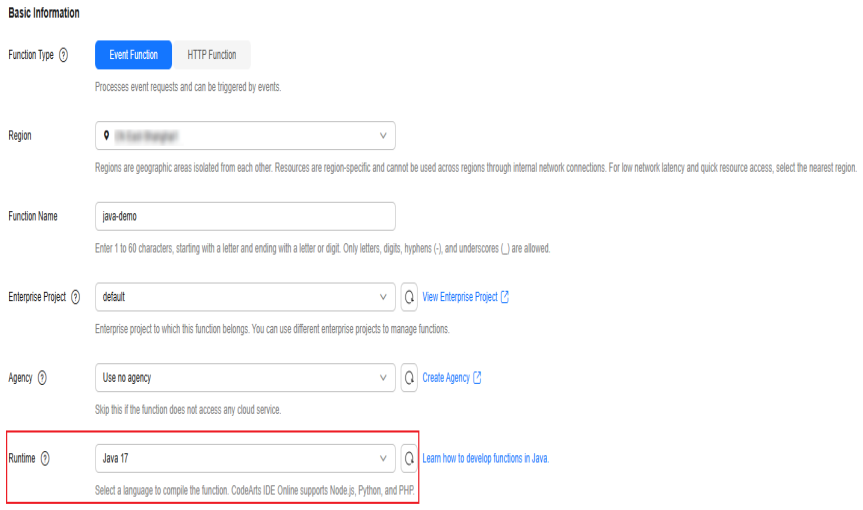
Figure 4-10 Choosing Build Artifacts to compile the JAR file



Step 3: Creating and Testing a Java Function

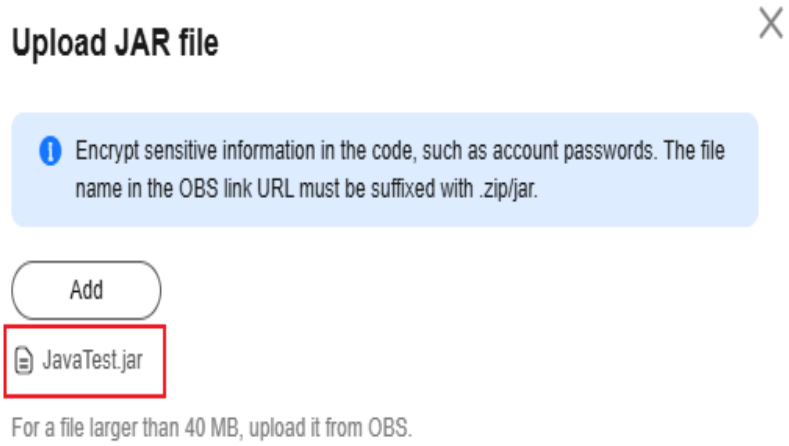
1. Log in to the **FunctionGraph console**. In the navigation pane on the left, choose **Functions** > **Function List**. On the displayed page, click **Create Function** in the upper right corner. On the displayed page, select **Create from scratch**.
2. Configure the basic function information, as shown in **Figure 4-11**. Set **Runtime** to **Java 17**. After the configuration is complete, click **Create Function** in the lower right corner.

Figure 4-11 Creating a Java function



3. On the function details page, click the **Code** tab, and click **Upload** > **Local JAR** on the right to upload the JAR file exported in **Step 2: Packaging a Java Project**, as shown in **Figure 4-12**.

Figure 4-12 Uploading a JAR file



4. After the upload is successful, choose **Configuration** > **Basic Settings**, modify the handler of the function to be tested, and click **Save**.
5. Return to the **Code** tab page, click **Test** in the code editing area, and click **Configure Test Event**. In the cloud event template list, select the event template to be tested and click **Create**.
6. Click **Test** and view the execution result.

   The function execution result consists of three parts: function output (returned by **callback**), summary, and logs (output by using the **console.log** or **getLogger()** method). For details, see **Table 4-17**.

Table 4-17 Description of the execution result

Parameter	Successful Execution	Failed Execution
Function Output	The defined function output information is returned.	A JSON file that contains **errorMessage** and **stackTrace** is returned. The format is as follows: `{ "errorMessage": "", "stackTrace": [] }` **errorMessage**: Error message returned by the runtime. **stackTrace**: Stack error information returned by the runtime.
Summary	**Request ID**, **Memory Configured**, **Execution Duration**, **Memory Used**, and **Billed Duration** are displayed.	**Request ID**, **Memory Configured**, **Execution Duration**, **Memory Used**, and **Billed Duration** are displayed.
Log Output	Function logs are printed. A maximum of 4 KB logs can be displayed.	Error information is printed. A maximum of 4 KB logs can be displayed.

NOTE

To test other event source triggers in the sample code, such as SMN, change the handler to **com.huawei.demo.TriggerTests.smnTest** on the **Basic Settings** page, and create an SMN test event to test the function.

4.2.2 Developing Functions in Java (Using an IDEA Maven Project)

This section describes how to develop functions in Java using an IDEA Maven project. For details about the Java syntax, initializer and SDK APIs, see **Function Development Overview**.

Procedure

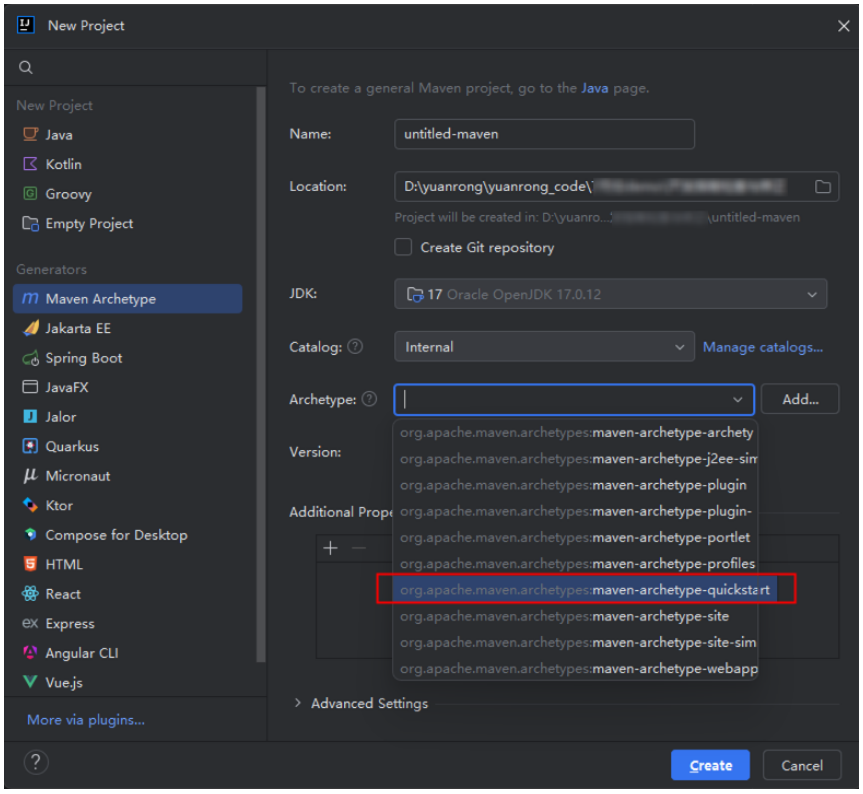
You can create a Java project and Java function following the steps in this section, or download the **Maven sample project package** and start from **Step 3: Creating and Testing a Java Function**.

Step 1: Creating a Maven Project Using IDEA

1. Create a function project.

   Configure IDEA and create a Maven project, as shown in **Figure 4-13**.

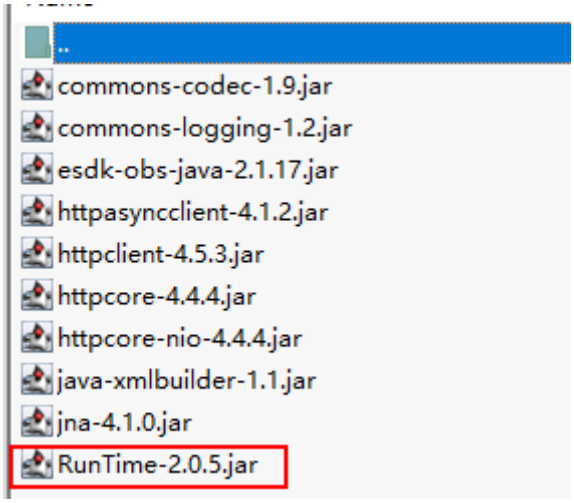
   Figure 4-13 Creating a project



2. Add dependencies to the project.

   Download the **Java SDK** to a local development environment, and decompress the SDK package, as shown in **Figure 4-14**.

Figure 4-14 Decompressing the downloaded SDK



a. Copy **Runtime-2.0.5.jar** in the package to a local directory, for example, **d:\\runtime**. Then, run the following command in the CLI to install the runtime to the local Maven repository:

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

b. Configure the dependency in the **pom.xml** file.

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

c. Configure other dependencies. The following uses the OBS dependency as an example.

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

d. Add a plug-in to **pom.xml** to pack the code and dependencies together. **Replace *mainClass* with the actual class.**

```
<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>
```

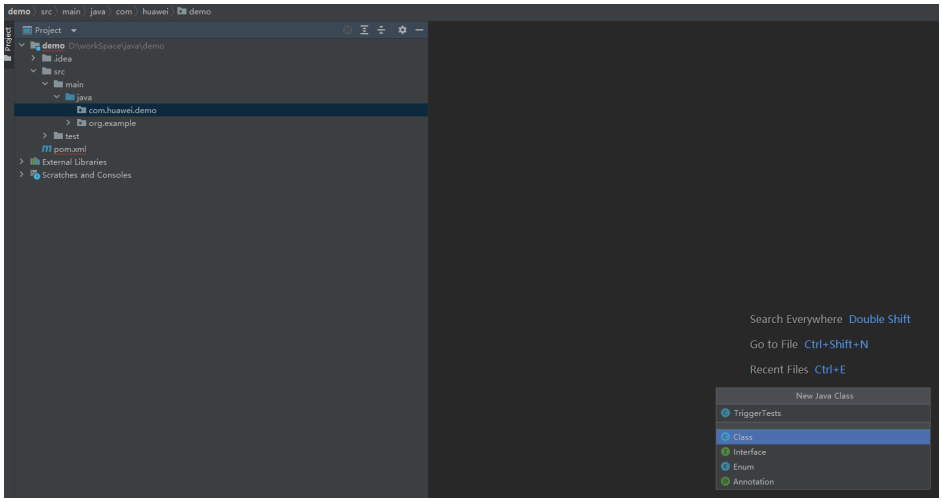


```
    </plugin>
  </plugins>
</build>
```

3. Configure the function resources.

   Create a package named **com.huawei.demo**, and then create a class named **TriggerTests** under the package, as shown in **Figure 4-15**.

   Figure 4-15 Creating the TriggerTests class



4. Configure the function code.

   Define the function handler in **TriggerTests.java**.

```
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;
import com.huawei.services.runtime.entity.eventgrid.EventGridTriggerEvent;

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";
    }

    public String eventgridTest(EventGridTriggerEvent event, Context context){
        System.out.println(event);return "ok";
    }
}
```

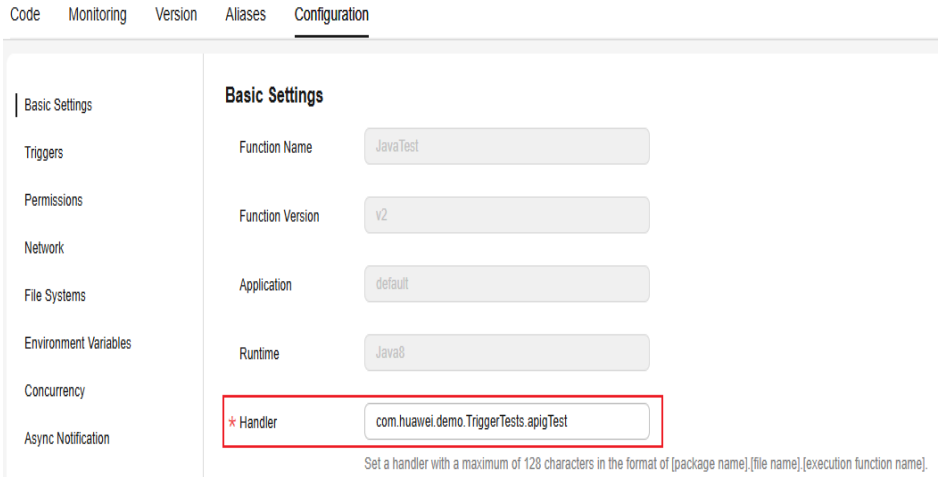
There are multiple handler functions with different trigger event types. You can modify the handler on the FunctionGraph console to test different handler functions. **For details about the constraints for the APIG event source, see Base64 Decoding and Response Structure.**

NOTE

Modifying the function handler:

In the navigation pane on the left of the FunctionGraph console, choose **Functions** > **Function List**. Click the name of the function to be set. On the function details page that is displayed, choose **Configuration** > **Basic Settings** and set the **Handler** parameter, as shown in **Figure 4-16**.

Figure 4-16 Function handler



5. Compile and pack the project file.

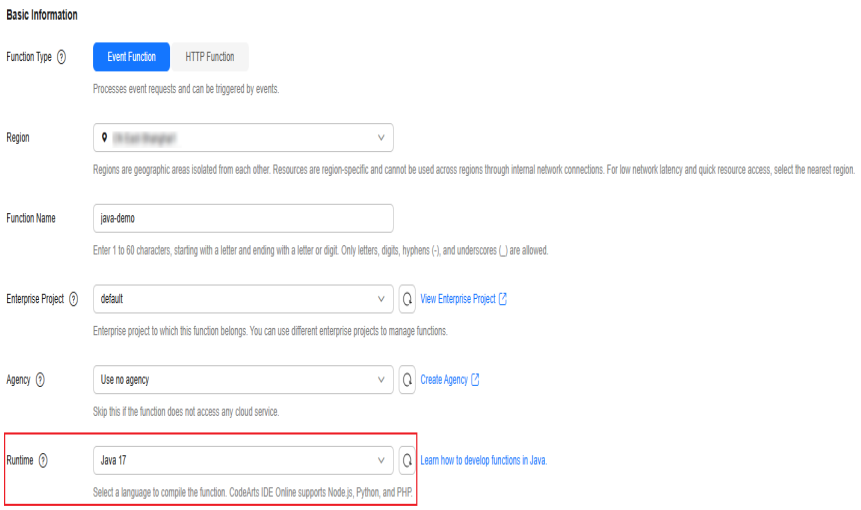
   Run the following command to compile and pack the project file: After compilation is complete, the **xx-jar-with-dependencies.jar** file is generated in the target directory.

```
mvn package assembly:single
```


Step 2: Creating and Testing a Java Function

1. Log in to the **FunctionGraph console**. In the navigation pane on the left, choose **Functions** > **Function List**. On the displayed page, click **Create Function** in the upper right corner. On the displayed page, select **Create from scratch**.
2. Configure the basic function information, as shown in **Figure 4-17**. Then, click **Create Function** in the lower right corner.

   Figure 4-17 Creating a Java function

3. On the function details page, click the **Code** tab, and click **Upload** > **Local JAR** on the right to upload the JAR file exported in **Step 1: Creating a Maven Project Using IDEA**.
4. After the upload is successful, choose **Configuration** > **Basic Settings**, modify the handler of the function to be tested, and click **Save**.
5. Return to the **Code** tab page, click **Test** in the code editing area, and click **Configure Test Event**. In the cloud event template list, select the event template to be tested and click **Create**.
6. Click **Test** and view the execution result.

   The function execution result consists of three parts: function output (returned by **callback**), summary, and logs (output by using the **console.log** or **getLogger()** method). For details, see **Table 4-18**.

Table 4-18 Description of the execution result

Parameter	Successful Execution	Failed Execution
Function Output	The defined function output information is returned.	A JSON file that contains **errorMessage** and **stackTrace** is returned. The format is as follows: `{ "errorMessage": "", "stackTrace": [] }` **errorMessage**: Error message returned by the runtime. **stackTrace**: Stack error information returned by the runtime.
Summary	**Request ID**, **Memory Configured**, **Execution Duration**, **Memory Used**, and **Billed Duration** are displayed.	**Request ID**, **Memory Configured**, **Execution Duration**, **Memory Used**, and **Billed Duration** are displayed.
Log Output	Function logs are printed. A maximum of 4 KB logs can be displayed.	Error information is printed. A maximum of 4 KB logs can be displayed.

NOTE

To test other event source triggers in the sample code, such as SMN, change the handler to **com.huawei.demo.TriggerTests.smnTest** on the **Basic Settings** page, and create an SMN test event to test the function.

4.3 Developing an HTTP Function Using Java

This section describes how to develop an HTTP function using Java. For details about HTTP function, see **Creating an HTTP Function**.

Constraints

- HTTP functions can only use APIG or APIC triggers.

  According to the forwarding protocol between FunctionGraph and APIG/APIC, a valid HTTP function response must contain body(String), statusCode(int), headers(Map), and isBase64Encoded(boolean). By default, the response is encoded using Base64. The default value of isBase64Encoded is true. The same applies to other frameworks. For details, see **Base64 Decoding and Return Structure**.
- By default, port **8000** is enabled for HTTP functions.
- **Table 4-15** describes the context methods provided by FunctionGraph.

Example of Using Java to Develop an HTTP Function

Overview

Usually, you may build Spring Boot applications using **SpringInitializr** or IntelliJ IDEA. This chapter uses the Spring.io project in **https://spring.io/guides/gs/rest-service/** as an example to deploy an HTTP function on FunctionGraph.

Procedure

This example describes how to use an existing Spring Boot project to build an HTTP function and deploy services on FunctionGraph.

For details, see **Building an HTTP Function with Spring Boot**.

4.4 Creating an Event Function Using a Container Image Built with Java

For details about how to use a container image to create and execute an event function, see **Creating an Event Function Using a Container Image and Executing the Function**. This section describes how to create an image using Java and verify the image locally.

Step 1: Creating an Image

Take the Linux x86 64-bit OS as an example. (There are no specific requirements for system configurations.)

1. Run the following command to create a folder:
   ```
   mkdir custom_container_event_example && cd custom_container_event_example
   ```
2. Use IntelliJ IDEA to create a Spring Boot project and select **Spring Web**, as shown in **Figure 4-19**.

Figure 4-18 Creating a Spring Boot project

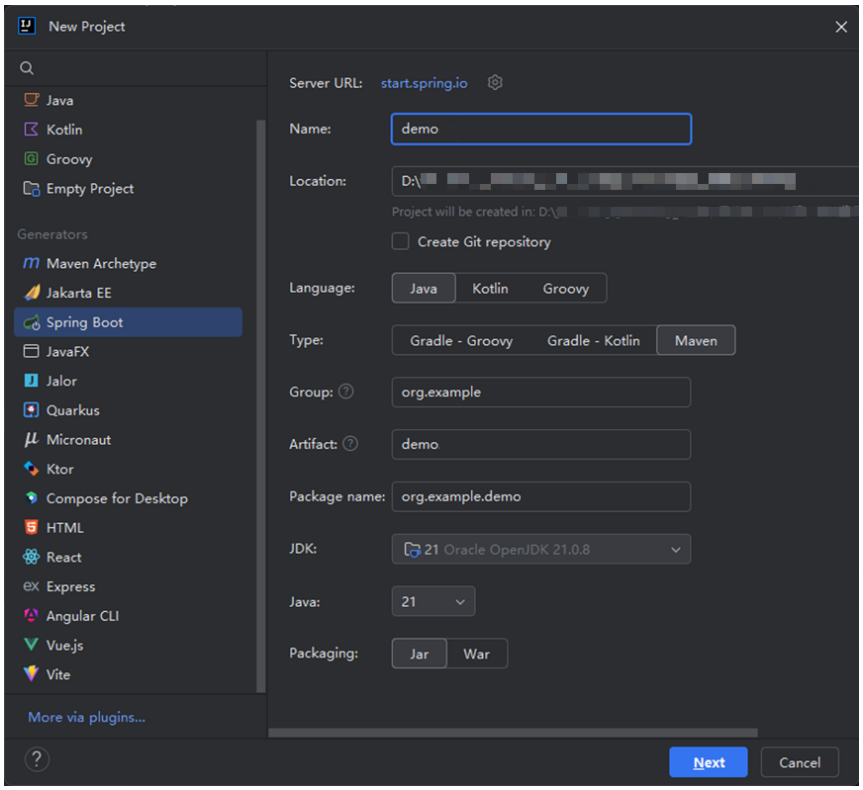
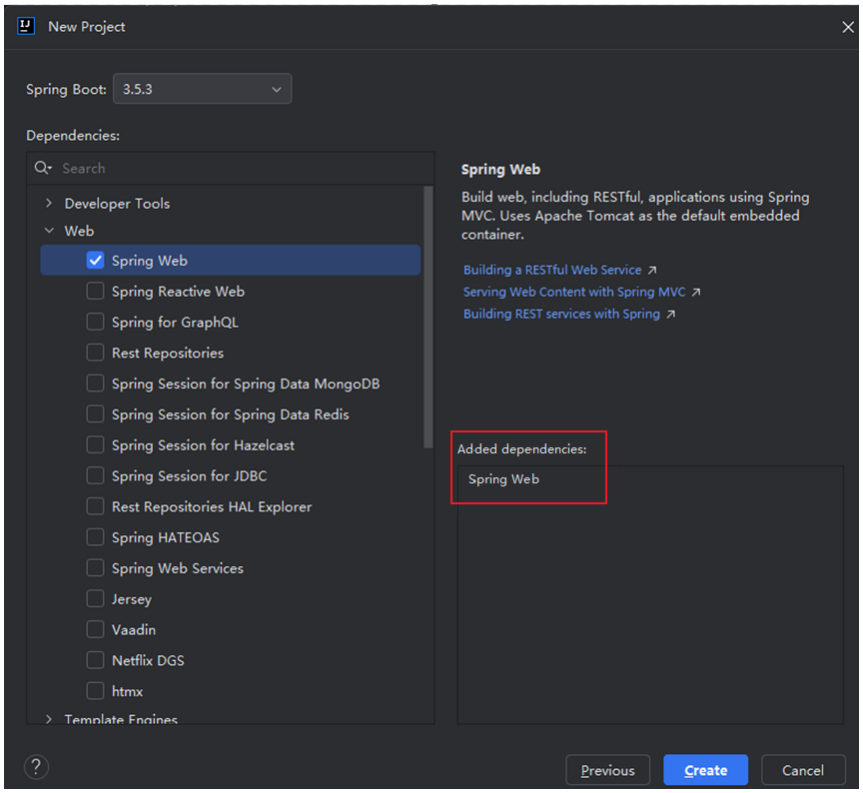


Figure 4-19 Selecting **Spring Web**



3. Use Java to implement an HTTP server demo to process HTTP requests and send responses.

Additionally create a controller package in the demo and implement a HelloWorld class to handle requests. The code is as follows:

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;


@RestController
public class HelloWorld {
    //Create a logger for recording debugging information.
    private static final Logger logger = LogManager.getLogger(HelloWorld.class);

    //Process the /init request.
    @RequestMapping("/init")
    public String init() {
        logger.debug("access init");
        return "hello init!!";
    }

    // Process the /invoke request.
    @RequestMapping("/invoke")
    public String invoke() {
        logger.debug("access invoke");
        return "hello invoke!!";
    }

}
```

4. Create a Dockerfile file and replace **demoSpringBoot-0.0.1-SNAPSHOT.jar** with the compiled JAR package.

```
FROM ubuntu:22.04

ENV HOME=/home/paas
ENV GROUP_ID=1003
ENV GROUP_NAME=paas_user
ENV USER_ID=1003
ENV USER_NAME=paas_user

RUN mkdir -m 550 ${HOME} && groupadd -g ${GROUP_ID} ${GROUP_NAME} && useradd -u ${USER_ID} -g ${GROUP_ID} ${USER_NAME}

RUN apt-get update &&  \
    apt-get install -y --no-install-recommends openjdk-21-jre-headless maven &&  \
    apt-get clean

COPY demoSpringBoot-0.0.1-SNAPSHOT.jar ${HOME}/

RUN chown -R ${USER_ID}:${GROUP_ID} ${HOME}
RUN chmod -R 775 ${HOME}

USER ${USER_NAME}
WORKDIR ${HOME}
EXPOSE 8000
ENTRYPOINT ["java", "-jar", "-Dfile.encoding=utf-8" ,"/home/paas/demoSpringBoot-0.0.1-SNAPSHOT.jar"]
```

Table 4-19 Parameter description

Parameter	Description
FROM	Specifies base image **ubuntu:22.04**. The base image is mandatory and its value can be changed.

Parameter	Description
ENV	Sets environment variables **HOME** (**/home/custom_container**), **GROUP_NAME** and **USER_NAME** (**custom_container**), and **USER_ID** and **GROUP_ID** (**1003**). These environment variables are mandatory and their values can be changed.
RUN	Executes commands. The format is **RUN** *<command>*. For example, **RUN mkdir -m 550 ${HOME}** means to create directory **${HOME}** for user *${USER_NAME}* during container building.
COPY	Copies files or directories from the build context to the image. Copy **demoSpringBoot-0.0.1-SNAPSHOT.jar** to the **${HOME}** directory of user *${USER_NAME}* in the container.
USER	Switches to user *${USER_NAME}*.
WORKDIR	Switches the working directory to the **${HOME}** directory of user *${USER_NAME}*.
EXPOSE	Expose port 8000 of the container. Do not change it.
ENTRYPOINT	Sets the executable command that is run each time a container starts. Run the following command to start a container: `java -jar -Dfile.encoding=utf-8 /home/paas/demoSpringBoot-0.0.1-SNAPSHOT.jar`

5. Run the following command to build an image:
   ```
   docker build -t custom_container_event_example:latest .
   ```
   In the preceding command, the image name is **custom_container_event_example**, the tag is **latest**, and the period (.) indicates the directory where the Dockerfile is located. Run the image build command to pack all files in the directory and send the package to a container engine to build an image.

Step 2: Performing Local Verification

1. Run the following command to start the Docker container:
   ```
   docker run -u 1003:1003 -p 8000:8000 custom_container_event_example:latest
   ```
2. Open a new Command Prompt, and send a message through port 8000 to access the **/*** directory specified in the template code.
   ```
   curl -XPOST -H 'Content-Type: application/json' localhost:8000/***
   ```
   The following information is returned based on the module code:
   ```
   hello invoke!!
   ```

Step 3: Creating a Function

Once you build the container image locally, you can create a function on the console.

For details, see **Creating an Event Function Using a Container Image and Executing the Function**. Start from **Step 3: Upload the Image**.

Helpful Links

- For more information about function development, such as the supported runtimes, trigger events, function project packaging specifications, and DLL referencing, see **Function Development Overview**.
- For details about the syntax and SDK APIs of function development in Java, see **Function Development Overview**.

4.5 Creating an HTTP Function Using a Container Image Built with Java

For details about how to use a container image to create and execute an HTTP function, see **Creating an HTTP Function Using a Container Image and Executing the Function**. This section describes how to create an image using Java and verify the image locally.

Step 1: Creating an Image

Take the Linux x86 64-bit OS as an example. (There are no specific requirements for system configurations.)

1. Run the following command to create a folder:
   ```
   mkdir custom_container_http_example && cd custom_container_http_example
   ```
2. Use IntelliJ IDEA to create a Spring Boot project and select **Spring Web**, as shown in **Figure 4-21**.

Figure 4-20 Creating a Spring Boot project

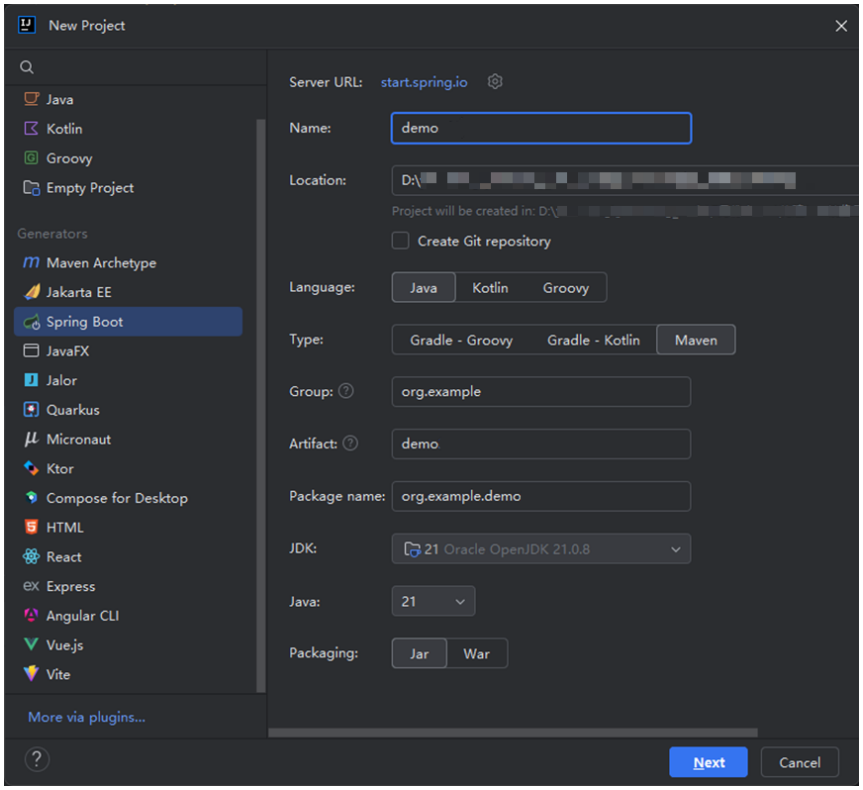
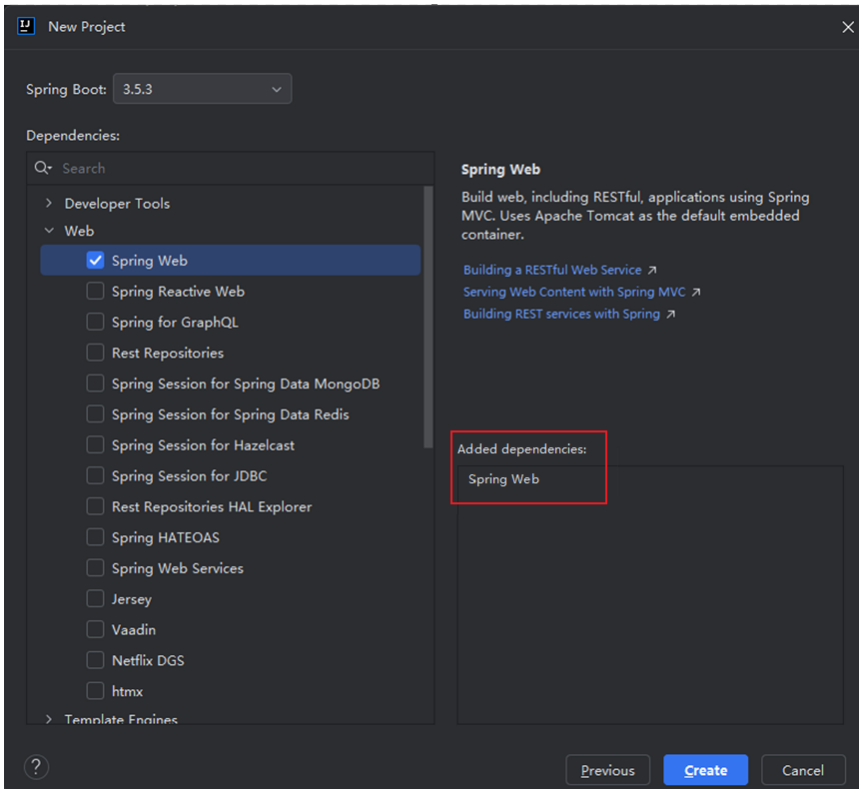


Figure 4-21 Selecting **Spring Web**



3. Use Java to implement an HTTP server demo to process HTTP requests and send responses.

   Additionally create a controller package in the demo and implement a HelloWorld class to handle requests. The code is as follows:

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;


@RestController
@RequestMapping("/hello")
public class HelloWorld {
    //Create a logger for recording debugging information.
    private static final Logger logger = LogManager.getLogger(HelloWorld.class);

    //Process the /init request.
   @RequestMapping("/world")
    public String world() {
        logger.debug("hello world");
        return "hello world!!";
    }

    // Process the /invoke request.
    @RequestMapping("/china")
    public String china() {
        logger.debug("hello china");
        return "hello china!!";
    }

}
```

4. Create a Dockerfile file and replace **demoSpringBoot-0.0.1-SNAPSHOT.jar** with the compiled JAR package.


```
FROM ubuntu:22.04

ENV HOME=/home/paas
ENV GROUP_ID=1003
ENV GROUP_NAME=paas_user
ENV USER_ID=1003
ENV USER_NAME=paas_user

RUN mkdir -m 550 ${HOME} && groupadd -g ${GROUP_ID} ${GROUP_NAME} && useradd -u ${USER_ID} -g ${GROUP_ID} ${USER_NAME}

RUN apt-get update &&  \
    apt-get install -y --no-install-recommends openjdk-21-jre-headless maven &&  \
    apt-get clean

COPY demoSpringBoot-0.0.1-SNAPSHOT.jar ${HOME}/

RUN chown -R ${USER_ID}:${GROUP_ID} ${HOME}
RUN chmod -R 775 ${HOME}

USER ${USER_NAME}
WORKDIR ${HOME}
EXPOSE 8000
ENTRYPOINT ["java", "-jar", "-Dfile.encoding=utf-8" ,"/home/paas/demoSpringBoot-0.0.1-SNAPSHOT.jar"]
```

Table 4-20 Parameter description

Parameter	Description
FROM	Specifies base image **ubuntu:22.04**. The base image is mandatory and its value can be changed.
ENV	Sets environment variables **HOME** (**/home/custom_container**), **GROUP_NAME** and **USER_NAME** (**custom_container**), and **USER_ID** and **GROUP_ID** (**1003**). These environment variables are mandatory and their values can be changed.
RUN	Executes commands. The format is **RUN** *<command>*. For example, **RUN mkdir -m 550 ${HOME}** means to create directory **${HOME}** for user *${USER_NAME}* during container building.
COPY	Copies files or directories from the build context to the image. Copy **demoSpringBoot-0.0.1-SNAPSHOT.jar** to the **${HOME}** directory of user *${USER_NAME}* in the container.
USER	Switches to user *${USER_NAME}*.
WORKDIR	Switches the working directory to the **${HOME}** directory of user *${USER_NAME}*.
EXPOSE	Expose port 8000 of the container. Do not change it.
ENTRYPOINT	Sets the executable command that is run each time a container starts. Run the following command to start a container: `java -jar -Dfile.encoding=utf-8 /home/paas/demoSpringBoot-0.0.1-SNAPSHOT.jar`

5. Run the following command to build an image:


```
docker build -t custom_container_http_example:latest .
```

In the preceding command, the image name is **custom_container_http_example**, the tag is **latest**, and the period (.) indicates the directory where the Dockerfile is located. Run the image build command to pack all files in the directory and send the package to a container engine to build an image.

Step 2: Performing Local Verification

1. Run the following command to start the Docker container:
   ```
   docker run -u 1003:1003 -p 8000:8000 custom_container_http_example:latest
   ```
2. Open a new Command Prompt, and send a message through port 8000 to access the **/hello/world** directory specified in the template code.
   ```
   curl -XPOST -H 'Content-Type: application/json' localhost:8000/hello/world
   ```
   The following information is returned based on the module code:
   ```
   hello world!!
   ```

Step 3: Creating a Function

Once you build the container image locally, you can create a function on the console.

For details, see **Creating an HTTP Function Using a Container Image and Executing the Function**. Start from **Step 3: Upload the Image**.

Helpful Links

- For more information about function development, such as the supported runtimes, trigger events, function project packaging specifications, and DLL referencing, see **Function Development Overview**.
- For details about the syntax and SDK APIs of function development in Java, see **Function Development Overview**.

4.6 Java Function Template

Java Function

The following is a sample code template of a Java function.

Each method corresponds to a specific trigger event, prints event information, and returns a response. You can create triggers and change the function handler to test Java functions.

Download the **Java SDK** to a local development environment, and decompress the SDK package. For complete Java function development example, see **Developing a Java Event Function** and **Developing an HTTP Function Using Java**.

```
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 com.huawei.services.runtime.entity.eventgrid.EventGridTriggerEvent;
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";
  }

  public String eventgridTest(EventGridTriggerEvent event, Context context){
    System.out.println(event);return "ok";
  }
}
```

4.7 Creating a Dependency for a Java Function

Constraints

If the modules to be installed need dependencies such as **.dll**, **.so**, and **.a**, archive them to a **.zip** package.

Dependency Creation Description

When you develop a function using Java, dependencies need to be compiled locally, compressed into a ZIP file, and uploaded through the ZIP file.

For details about how to create and add a dependency to a Java function, see **Developing Functions in Java (Using IDEA to Create a Java Project)**.

5 Go

5.1 Function Development Overview

Function Syntax

FunctionGraph supports Go 1.x. The following shows the function syntax.

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

- **Handler**: name of the handler function.
- **payload**: event parameter defined for the function. The parameter is in JSON format.
- **ctx**: runtime information provided for executing the function. For details, see **SDK APIs**.

The format of the Go function handler is the same as the name of the executable file in the code package. Ensure that the name of the dynamic library file is consistent with the plug-in name of the handler. For example, if the name of the dynamic library file is **testplugin.so**, set the handler name to **testplugin.Handler**. You can configure or modify the handler on the function details page on the FunctionGraph console.

SDK APIs

The Go SDK provides event, context, and logging APIs. **Download the Go SDK** (**Go SDK.sha256**).

- Event APIs

  Event structure definitions are added to the Go SDK. Currently, DIS, DDS, SMN, Timer, and APIG triggers are supported. The definitions make coding much simpler when triggers are required.

  a. **APIG trigger field description**

  i. APIGTriggerEvent fields

Table 5-1 APIGTriggerEvent fields

Field	Description
IsBase64Encoded	Whether the body of an event is encoded using Base64.
HttpMethod	HTTP request method.
Path	HTTP request path.
Body	HTTP request body.
PathParameters	All path parameters.
RequestContext	API Gateway configurations (**APIGRequestContext object**).
Headers	HTTP request header.
QueryStringParameters	Query parameters.
UserData	User data set in the APIG custom authorizer.

Table 5-2 APIGRequestContext fields

Field	Description
ApiId	API ID.
RequestId	API request ID.
Stage	Name of the environment in which an API has been published.

ii. APIGTriggerResponse fields

Table 5-3 APIGTriggerResponse fields

Field	Description
Body	Message body.
Headers	HTTP response header to be returned.
StatusCode	HTTP status code. Type: **int**.
IsBase64Encoded	Whether the body has been encoded using Base64. Type: **bool**.

NOTE

APIGTriggerEvent provides the GetRawBody() method to obtain the body decoded using Base64. APIGTriggerResponse provides the SetBase64EncodedBody() method to set the body encoded using Base64.

b. **DIS trigger field description**

Table 5-4 DISTriggerEvent fields

Field	Description
ShardID	Partition ID.
Message	DIS message body (**DISMessage structure**).
Tag	Function version.
StreamName	Stream name.

Table 5-5 DISMessage fields

Field	Description
NextPartitionCursor	Next partition cursor.
Records	Message records (**DISRecord structure**).
MillisBehindLatest	Reserved parameter.

Table 5-6 DISRecord fields

Field	Description
PartitionKey	Data partition.
Data	Data.
SequenceNumber	Sequence number (ID of each record).

c. **Kafka trigger field description**

Table 5-7 KAFKATriggerEvent fields

Field	Description
InstanceId	Instance ID.
Records	Message records (**Table 5-8**).

Field	Description
TriggerType	Trigger type (Kafka).
Region	Region.
EventTime	Time when an event occurred (seconds).
EventVersion	Event version.

Table 5-8 KAFKARecord parameters

Field	Description
Messages	DMS message body.
TopicId	Topic ID.

d. **SMN trigger field description**

Table 5-9 SMNTriggerEvent fields

Field	Description
Record	Message records (**SMNRecord structure**).

Table 5-10 SMNRecord fields

Field	Description
EventVersion	Event version. (Currently, the version is **1.0**.)
EventSubscriptionUrn	Subscription URN.
EventSource	Event source.
Smn	Message body (**SMNBody structure**).

Table 5-11 SMNBody fields

Field	Description
TopicUrn	Topic URN.
TimeStamp	Message timestamp.

Field	Description
MessageAtrributes	Message attribute set.
Message	Message body.
Type	Message format.
MessageId	Message ID.
Subject	Message topic.

e. **Timer trigger field description**

Table 5-12 TimerTriggerEvent fields

Field	Description
Version	Version. (Currently, the version is **v1.0**.)
Time	Current time.
TriggerType	Trigger type (Timer).
TriggerName	Trigger name.
UserEvent	Additional information about the trigger.

NOTE

1. When using an APIG trigger, set the first parameter of the handler function (for example, **handler**) to **handler(APIGTriggerEvent event, Context context)**. For details about the constraints, see **Base64 Decoding and Response Structure**.
2. The preceding TriggerEvent methods have corresponding **set** methods, which are recommended for local debugging. DIS and LTS triggers have getRawData() methods, but do not have setRawData() methods.

- Context APIs

  The context APIs are used to obtain the context, such as agency AK/SK, current request ID, allocated memory space, and number of CPUs, required for executing a function.

  Table 5-13 describes the context APIs provided by FunctionGraph.

Table 5-13 Context methods

Method	Description
getRequestID( )	Obtains a request ID.
getRemainingTimeInMilli-getRunningTimeInSe-condsSeconds ( )	Obtains the remaining running time of a function.

Method	Description
getAccessKey( )	Obtains the AK (valid for 24 hours) with an agency. If you use this method, you need to configure an agency for the function. **FunctionGraph has stopped maintaining the getAccessKey API in the Runtime SDK. You cannot use this API to obtain a temporary AK.**
getSecretKey( )	Obtains the SK (valid for 24 hours) with an agency. If you use this method, you need to configure an agency for the function. **FunctionGraph has stopped maintaining the getSecretKey API in the Runtime SDK. You cannot use this API to obtain a temporary SK.**
getSecurityAccessKey( )	Obtains the SecurityAccessKey (valid for 24 hours) with an agency. The cache duration is 10 minutes. That is, the same content is returned within 10 minutes. To use this method, you need to configure an agency for the function.
getSecuritySecretKey( )	Obtains the SecuritySecretKey (valid for 24 hours) with an agency. The cache duration is 10 minutes. That is, the same content is returned within 10 minutes. To use this method, you need to configure an agency for the function.
getSecurityToken( )	Obtains the SecurityToken (valid for 24 hours) with an agency. The cache duration is 10 minutes. That is, the same content is returned within 10 minutes. To use this method, you need to configure an agency for the function.
getUserData(string key)	Uses keys to obtain the values passed by environment variables.
getFunctionName( )	Obtains the name of a function.
getRunningTimeInSeconds( )	Obtains the timeout of a function.
getVersion( )	Obtains the version of a function.
getMemorySize( )	Obtains the allocated memory.
getCPUNumber( )	Obtains CPU usage of a function.
getPackage( )	Obtains a function group.

Method	Description
getToken( )	Obtains the token (valid for 24 hours) with an agency. If you use this method, you need to configure an agency for the function.
getLogger( )	Obtains the **logger** method provided by the context. By default, information such as the time and request ID is output.
getAlias()	Obtains function alias.

- Table 5-14 describes the logging API provided in the Go SDK.

Table 5-14 Logging API

Method	Description
RuntimeLogger()	• Records user input logs by using the method **Logf(format string, args ...interface{})**. • This method outputs logs in the format of *Time-Request ID-Output*, for example, **2017-10-25T09:10:03.328Z 473d369d-101a-445e-a7a8-315cca788f86 test log output**.

5.2 Developing a Go Event Function

For details about the syntax and SDK APIs of Go functions, see **Function Development Overview**.

Developing a Go Function

Log in to the Linux server where the Go 1.x SDK has been installed and perform the following steps to compile and package the Go function. (Currently, Ubuntu 14.04, Ubuntu 16.04, SUSE 11.3, SUSE 12.0, and SUSE 12.1 are supported.)

Go Versions Supporting go mod (1.11.1+)

Step 1 Create a temporary directory, for example, **/home/fssgo**, decompress the **Go SDK** of FunctionGraph to the created directory, and enable the go module function.

```
$ mkdir -p /home/fssgo
$ unzip functiongraph-go-runtime-sdk-1.0.1.zip -d /home/fssgo
$ export GO111MODULE="on"
```

Step 2 Generate the **go.mod** file in the **/home/fssgo** directory. Assume that the module name is **test**:

```
$ go mod init test
```


Step 3 Edit the **go.mod** file in the **/home/fssgo** directory as required (that is, add the content in bold).

```
module test

go 1.24.5

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

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

Step 4 Create a function file under the **/home/fssgo** directory and implement the following interface:

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

In this interface, **payload** is the body of a client request, and **ctx** is the runtime context object provided for executing a function. For more information about the methods, see **Table 5-13**. The following uses **test.go** as an example.

```
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)
}
```


Constraints:

1. If the **error** parameter returned by a function is not **nil**, the function execution fails.
2. If the **error** parameter returned by a function is **nil**, FunctionGraph supports only the following types of values:

   nil: The HTTP response body is empty.

   []byte: The content in this byte array is the body of an HTTP response.

   string: The content in this string is the body of an HTTP response.

   Other: FunctionGraph returns a value for JSON encoding, and uses the encoded object as the body of an HTTP response. The **Content-Type** header of the HTTP response is set to **application/json**.
3. **The preceding example uses an APIG trigger as an example. For other trigger types, you need to modify the content of the main function. For example, change the CTS trigger to runtime.Register(CtsTest). Currently, only one entry can be registered.**
4. **For details about the constraints for the APIG event source, see Base64 Decoding and Response Structure.**

Step 5 Compile and package the function code.

After completing the function code, compile and package it as follows:

1. Compile the code.

```
$ cd /home/fssgo
$ go build  -o handler test.go
```

   NOTE

   The **handler** can be customized on the function details page on the FunctionGraph console.

2. Package the code.

```
$ zip fss_examples_go1.x.zip handler
```

Step 6 Create a function.

Log in to the **FunctionGraph console**, create a Go 1.x function, and upload the code package **fss_examples_go1.x.zip**.

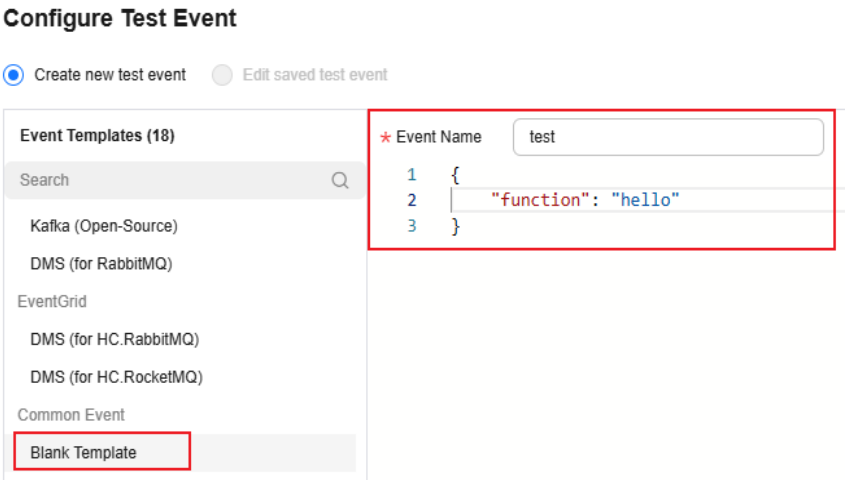
If you edit code in Go, package the compiled file into a ZIP file, and ensure that the name of the compiled file is consistent with the handler. For example, if the name of the binary file is **handler**, set the **Handler** to **handler**. The handler must be consistent with that defined in **Step 3.2**.

Step 7 Test the function.

1. Create a test event.

   On the function details page that is displayed, click **Configure Test Event**. Configure the test event information, as shown in **Figure 5-1**, and then click **Create**.

Figure 5-1 Configuring a test event



2. On the function details page, select the configured test event, click **Test**, and view the function execution result by referring to **Function Execution Result**.

----End

Go Versions not Supporting go mod (<1.11.1)

Step 1 Create a temporary directory, for example, **/home/fssgo/src/huaweicloud.com**, and decompress the **Go SDK** to the created directory.

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

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

Step 2 Create a function file under the **/home/fssgo/src** directory and implement the following interface:

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

In this interface, **payload** is the body of a client request, and **ctx** is the runtime context object provided for executing a function. For more information about the methods, see the SDK APIs. The following uses **test.go** as an example.

```
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)
}
```

Constraints:

1. If the **error** parameter returned by a function is not **nil**, the function execution fails.
2. If the **error** parameter returned by a function is **nil**, FunctionGraph supports only the following types of values:

   nil: The HTTP response body is empty.

   []byte: The content in this byte array is the body of an HTTP response.

   string: The content in this string is the body of an HTTP response.

   Other: FunctionGraph returns a value for JSON encoding, and uses the encoded object as the body of an HTTP response. The **Content-Type** header of the HTTP response is set to **application/json**.
3. **The preceding example uses an APIG trigger as an example. For other trigger types, you need to modify the content of the main function. For example, change the CTS trigger to runtime.Register(CtsTest). Currently, only one entry can be registered.**
4. **For details about the constraints for the APIG event source, see Base64 Decoding and Response Structure.**

Step 3 Compile and package the function code.

After completing the function code, compile and package it as follows:

1. Set environment variables **GOROOT** and **GOPATH**.

   ```
   $ export GOROOT=/usr/local/go (Assume that the Go SDK is installed under the /usr/local/go directory.)
   $ export PATH=$GOROOT/bin:$PATH
   $ export GOPATH=/home/fssgo
   ```
2. Compile the function code.

   ```
   $ cd /home/fssgo
   $ go build  -o handler test.go
   ```


> **NOTE**
>
> The **handler** can be customized on the function details page on the FunctionGraph console.

3. Package the code.

```
$ zip fss_examples_go1.x.zip handler
```

Step 4 Creating a function

Log in to the **FunctionGraph console**, create a Go 1.x function, and upload the code package **fss_examples_go1.x.zip**.

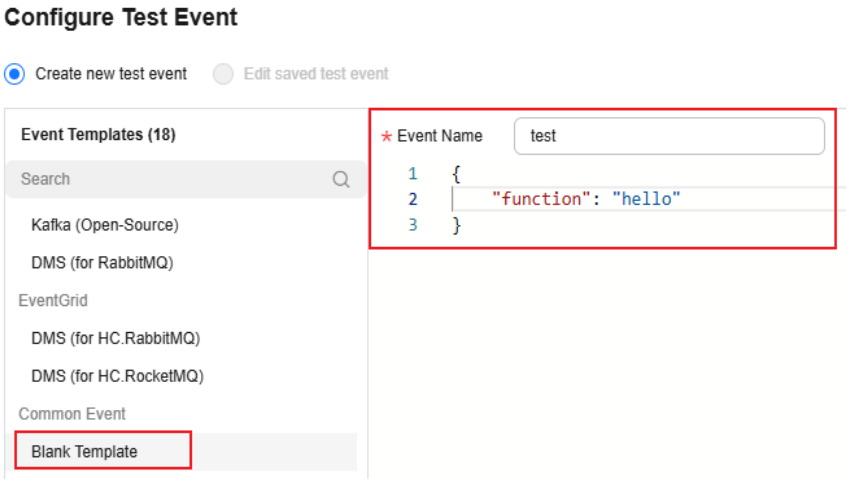
If you edit code in Go, package the compiled file into a ZIP file, and ensure that the name of the compiled file is consistent with the handler name. For example, if the name of the binary file is **handler**, set the **Handler** to **handler**. The handler must be consistent with that defined in **Step 3.2**.

Step 5 Test the function.

1. Create a test event.

   On the function details page that is displayed, click **Configure Test Event**. Configure the test event information, as shown in **Figure 5-2**, and then click **Create**.

   Figure 5-2 Configuring a test event



2. On the function details page, select the configured test event, click **Test**, and view the function execution result by referring to **Function Execution Result**.

----End

Function Execution Result

The execution result consists of the function output, summary, and log output.

Table 5-15 Description of the execution result

Parameter	Successful Execution	Failed Execution

Function Output	The defined function output information is returned.	A JSON file that contains **errorMessage** and **errorType** is returned. The format is as follows:
		{
		"errorMessage": "",
		"errorType":"",
		}
		errorMessage: Error message returned by the runtime.
		errorType: Error type.

Summary	**Request ID**, **Memory Configured**, **Execution Duration**, **Memory Used**, and **Billed Duration** are displayed.	**Request ID**, **Memory Configured**, **Execution Duration**, **Memory Used**, and **Billed Duration** are displayed.
Log Output	Function logs are printed. A maximum of 4 KB logs can be displayed.	Error information is printed. A maximum of 4 KB logs can be displayed.

5.3 Developing an HTTP Function Using Go

This section describes how to develop an HTTP function using Go. For details about HTTP function, see **Creating an HTTP Function**.

Constraints

- HTTP functions can only use APIG or APIC triggers.

  According to the forwarding protocol between FunctionGraph and APIG/APIC, a valid HTTP function response must contain body(String), statusCode(int), headers(Map), and isBase64Encoded(boolean). By default, the response is encoded using Base64. The default value of isBase64Encoded is true. The same applies to other frameworks. For details, see **Base64 Decoding and Return Structure**.
- By default, port **8000** is enabled for HTTP functions.
- **Table 5-13** describes the context methods provided by FunctionGraph.

Example of Using Go to Develop an HTTP Function

Overview

Since HTTP functions do not directly support Go code deployment, this chapter provides an example of using binary conversion to deploy a Go program on FunctionGraph.

Procedure

For details, see **Building an HTTP Function with Go**.

Sample Code

The code of the source file **main.go** is as follows:

```
// main.go
package main

import (
    "fmt"
    "net/http"

    "github.com/emicklei/go-restful"
)

func registerServer() {
    fmt.Println("Running a Go Http server at localhost:8000/")

    ws := new(restful.WebService)
    ws.Path("/")

    ws.Route(ws.GET("/hello").To(Hello))
    c := restful.DefaultContainer
    c.Add(ws)
    fmt.Println(http.ListenAndServe(":8000", c))
}

func Hello(req *restful.Request, resp *restful.Response) {
    resp.Write([]byte("nice to meet you"))
}

func main() {
    registerServer()
}
# bootstrap
/opt/function/code/go-http-demo
```

In **main.go**, an HTTP server is started using port **8000**, and an API whose path is **/hello** is registered. When the API is invoked, "nice to meet you" is returned.

Compiling and Packaging

1. On the Linux server, compile the preceding code using the **go build -o go-http-demo main.go** command.
2. Compress **go-http-demo** and **bootstrap** into a ZIP package named **xxx.zip**.

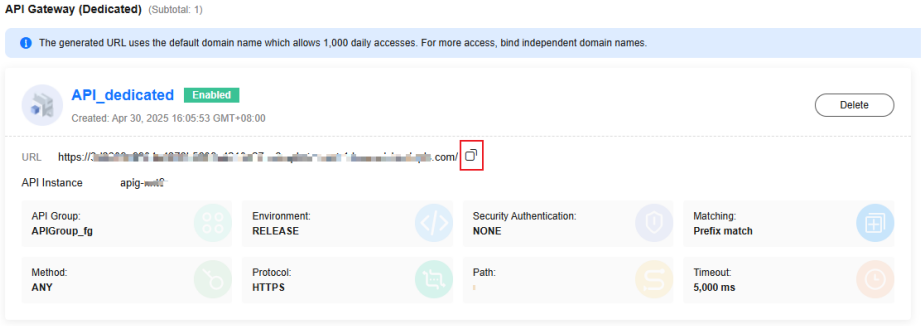
Creating an HTTP Function

Create an HTTP function in FunctionGraph, and upload the **xxx.zip** code package.

Creating an APIG Trigger

Create an APIG trigger, select a proper group and environment, and click **OK**. (You can select **None** for **Security Authentication** in the test phase.)

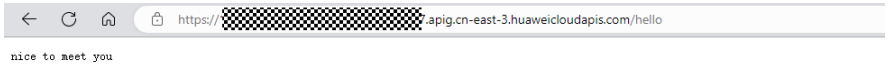
Figure 5-3 Copying the URL



Invocation Test

Copy the URL generated in **Creating an APIG Trigger** + *Path registered in the code* + **/hello** to the address box of the browser. The following information is displayed:

Figure 5-4 Returned result



5.4 Creating an Event Function Using a Container Image Built with Go

For details about how to use a container image to create and execute an event function, see **Creating an Event Function Using a Container Image and Executing the Function**. This section describes how to create an image using Go and verify the image locally.

Step 1: Creating an Image

Take the Linux x86 64-bit OS as an example. (There are no specific requirements for system configurations.)

1. Run the following command to create a folder:
   ```
   mkdir custom_container_event_example && cd custom_container_event_example
   ```
2. Implement an HTTP server to process **init** and **invoke** requests and give a response. Go is used as an example.

   Create a **server_demo.go** file, import the **gin** dependency package, and implement a function handler (method **POST** and path **/invoke**). Call the **init** function to initialize the configuration. Go runs this function automatically. Example code:
   ```
   package main

   import (
     "fmt"
     "net/http"
   ```

```
  "time"

  "github.com/gin-gonic/gin" // Import the Gin framework.
)

// Initialization function, which is executed when the program is started.
func init() {
  fmt.Println("init in main.go ")
}

// Logger is a middleware function used to record request and response information.
func Logger() gin.HandlerFunc {
  return func(c *gin.Context) {
    start := time.Now()

    reqBody, _ := c.GetRawData()
    fmt.Printf("[INFO] Request: %s %s %s\n", c.Request.Method, c.Request.RequestURI, reqBody)

    c.Next()

    end := time.Now()
    latency := end.Sub(start)
    respBody := string(rune(c.Writer.Size()))
    fmt.Printf("[INFO] Response: %s %s %s (%v)\n", c.Request.Method, c.Request.RequestURI,
respBody, latency)
  }
}

// invoke is a function that processes POST requests sent to the /invoke route.
func invoke(c *gin.Context) {
  println("hello world")
  c.String(http.StatusOK, "*** hello world ***")
  return
}

func main() {
  router := gin.Default() // Create a default Gin router.

  router.Use(Logger()) // Use the Logger middleware.

  router.POST("/invoke", invoke) // Register a route (/invoke) that processes HTTP POST requests.
When a client sends a POST request to /invoke, the Gin framework calls the invoke function to
process the request.
  err := router.Run(":8000") // Start the HTTP server and listen to port 8000.
  if err != nil {
    return
  }
}
```

3. Execute the following command to compile and generate the **server_demo** binary file:

```
go build server_demo.go
```

4. Create a Dockerfile and replace **server_demo** in the Dockerfile with the compiled file name. The file content is as follows:

```
FROM ubuntu:22.04

ENV HOME=/home/paas
ENV GROUP_ID=1003
ENV GROUP_NAME=paas_user
ENV USER_ID=1003
ENV USER_NAME=paas_user

RUN mkdir -m 550 ${HOME} && groupadd -g ${GROUP_ID} ${GROUP_NAME} && useradd -u ${USER_ID} -g ${GROUP_ID} ${USER_NAME}

COPY server_demo ${HOME}

RUN chown -R ${USER_ID}:${GROUP_ID} ${HOME}
RUN chmod -R 775 ${HOME}
```



```
USER ${USER_NAME}
WORKDIR ${HOME}
EXPOSE 8000
ENTRYPOINT ["./server_demo"]
```

Table 5-16 Parameter description

Parameter	Description
FROM	Specifies base image **ubuntu:22.04**. The base image is mandatory and its value can be changed.
ENV	Sets environment variables **HOME** (**/home/custom_container**), **GROUP_NAME** and **USER_NAME** (**custom_container**), and **USER_ID** and **GROUP_ID** (**1003**). These environment variables are mandatory and their values can be changed.
RUN	Executes commands. The format is **RUN** *<command>*. For example, **RUN mkdir -m 550 ${HOME}** means to create directory **${HOME}** for user *${USER_NAME}* during container building.
COPY	Copies files or directories from the build context to the image. Copy **server_demo.go** to the **${HOME}** directory of user *${USER_NAME}* in the container.
USER	Switches to user *${USER_NAME}*.
WORKDIR	Switches the working directory to the **${HOME}** directory of user *${USER_NAME}*.
EXPOSE	Informs Docker that the container listens on the specified network ports at runtime. Expose port 8000 of the container and do not change it.
ENTRYPOINT	Sets the executable command that is run each time a container starts. Run the **./server_demo** command to start a container.

5. Run the following command to build an image:
   ```
   docker build -t custom_container_event_example:latest .
   ```
   In the preceding command, the image name is **custom_container_event_example**, the tag is **latest**, and the period (.) indicates the directory where the Dockerfile is located. Run the image build command to pack all files in the directory and send the package to a container engine to build an image.

Step 2: Performing Local Verification

1. Run the following command to start the Docker container:
   ```
   docker run -u 1003:1003 -p 8000:8000 custom_container_event_example:latest
   ```
2. Open a new Command Prompt, and send a message through port 8000 to access the **/invoke** directory specified in the template code.
   ```
   curl -XPOST -H 'Content-Type: application/json' -d '{"message":"HelloWorld"}' localhost:8000/invoke
   ```


The following information is returned based on the module code:

```
*** hello world ***
```

Step 3: Creating a Function

Once you build the container image locally, you can create a function on the console.

For details, see **Creating an Event Function Using a Container Image and Executing the Function**. Start from **Step 3: Upload the Image**.

Helpful Links

- For more information about function development, such as the supported runtimes, trigger events, function project packaging specifications, and DLL referencing, see **Function Development Overview**.
- For details about the syntax and SDK APIs of function development in Go, see **Function Development Overview**.

5.5 Creating an HTTP Function Using a Container Image Built with Go

For details about how to use a container image to create and execute an HTTP function, see **Creating an HTTP Function Using a Container Image and Executing the Function**. This section describes how to create an image using Go and verify the image locally.

Step 1: Creating an Image

Take the Linux x86 64-bit OS as an example. (There are no specific requirements for system configurations.)

1. Run the following command to create a folder:
   ```
   mkdir custom_container_http_example && cd custom_container_http_example
   ```
2. Implement an HTTP server to process **init** and **invoke** requests and give a response. Go is used as an example.

   Create a **server_demo.go** file, import the **gin** dependency package, and implement a function handler index (method **POST** and path **/index**). **The handler can be compiled based on the actual service requirements.** Call the **init** function to initialize the configuration. Go runs this function automatically. Example code:

   ```
   package main

   import (
     "fmt"
     "net/http"
     "time"

     "github.com/gin-gonic/gin" // Import the Gin framework.
   )

   // Initialization function, which is executed when the program is started.
   func init() {
     fmt.Println("init in main.go ")
   }
   ```



```
// Logger is a middleware function used to record request and response information.
func Logger() gin.HandlerFunc {
  return func(c *gin.Context) {
    start := time.Now()

    reqBody, _ := c.GetRawData()
    fmt.Printf("[INFO] Request: %s %s %s\n", c.Request.Method, c.Request.RequestURI, reqBody)

    c.Next()

    end := time.Now()
    latency := end.Sub(start)
    respBody := string(rune(c.Writer.Size()))
    fmt.Printf("[INFO] Response: %s %s %s (%v)\n", c.Request.Method, c.Request.RequestURI,
respBody, latency)
  }
}

// index is a function that processes POST requests sent to the /index route.
func index(c *gin.Context) {
  println("hello world")
  c.String(http.StatusOK, "*** hello world ***")
  return
}

func main() {
  router := gin.Default() // Create a default Gin router.

  router.Use(Logger()) // Use the Logger middleware.

  router.POST("/index", index) // Register a route (/index) that processes HTTP POST requests. When
a client sends a POST request to /index, the Gin framework calls the index function to process the
request.
  err := router.Run(":8000") // Start the HTTP server and listen to port 8000.
  if err != nil {
    return
  }
}
```

3. Execute the following command to compile and generate the **server_demo** binary file:

```
go build server_demo.go
```

4. Create a Dockerfile and replace **server_demo** in the Dockerfile with the compiled file name. The file content is as follows:

```
FROM ubuntu:22.04

ENV HOME=/home/paas
ENV GROUP_ID=1003
ENV GROUP_NAME=paas_user
ENV USER_ID=1003
ENV USER_NAME=paas_user

RUN mkdir -m 550 ${HOME} && groupadd -g ${GROUP_ID} ${GROUP_NAME} && useradd -u ${USER_ID} -g ${GROUP_ID} ${USER_NAME}

COPY server_demo ${HOME}

RUN chown -R ${USER_ID}:${GROUP_ID} ${HOME}
RUN chmod -R 775 ${HOME}

USER ${USER_NAME}
WORKDIR ${HOME}
EXPOSE 8000
ENTRYPOINT ["./server_demo"]
```


Table 5-17 Parameter description

Parameter	Description
FROM	Specifies base image **ubuntu:22.04**. The base image is mandatory and its value can be changed.
ENV	Sets environment variables **HOME** (**/home/custom_container**), **GROUP_NAME** and **USER_NAME** (**custom_container**), and **USER_ID** and **GROUP_ID** (**1003**). These environment variables are mandatory and their values can be changed.
RUN	Executes commands. The format is **RUN** *<command>*. For example, **RUN mkdir -m 550 ${HOME}** means to create directory **${HOME}** for user *${USER_NAME}* during container building.
COPY	Copies files or directories from the build context to the image. Copy **server_demo.go** to the **${HOME}** directory of user *${USER_NAME}* in the container.
USER	Switches to user *${USER_NAME}*.
WORKDIR	Switches the working directory to the **${HOME}** directory of user *${USER_NAME}*.
EXPOSE	Informs Docker that the container listens on the specified network ports at runtime. Expose port 8000 of the container and do not change it.
ENTRYPOINT	Sets the executable command that is run each time a container starts. Run the **./server_demo** command to start a container.

5. Run the following command to build an image:
   ```
   docker build -t custom_container_http_example:latest .
   ```
   In the preceding command, the image name is **custom_container_http_example**, the tag is **latest**, and the period (.) indicates the directory where the Dockerfile is located. Run the image build command to pack all files in the directory and send the package to a container engine to build an image.

Step 2: Performing Local Verification

1. Run the following command to start the Docker container:
   ```
   docker run -u 1003:1003 -p 8000:8000 custom_container_http_example:latest
   ```
2. Open a new Command Prompt, and send a message through port 8000 to access the **/invoke** directory specified in the template code.
   ```
   curl -XPOST -H 'Content-Type: application/json' -d '{"message":"HelloWorld"}' localhost:8000/index
   ```
   The following information is returned based on the module code:
   ```
   *** hello world ***
   ```


Step 3: Creating a Function

Once you build the container image locally, you can create a function on the console.

For details, see **Creating an HTTP Function Using a Container Image and Executing the Function**. Start from **Step 3: Upload the Image**.

Helpful Links

- For more information about function development, such as the supported runtimes, trigger events, function project packaging specifications, and DLL referencing, see **Function Development Overview**.
- For details about the syntax and SDK APIs of function development in Go, see **Function Development Overview**.

6 C#

6.1 C# Function Development Overview

FunctionGraph supports the following C# runtimes:

- C#(.NET Core 2.1)
- C#(.NET Core 3.1)
- C#(.NET Core 6.0)
- C# (.NET Core 8.0) (only available in **ME-Riyadh** and **TR-Istanbul**)

Function Syntax

C# function syntax: *Scope Return parameter Function name (User-defined parameter, Context)*

- *Scope*: It must be defined as **public** for the function that FunctionGraph invokes to execute your code.
- *Return parameter*: user-defined output, which is converted into a character string and returned as an HTTP response.
- *Function name*: user-defined function name. The name must be consistent with that you define when creating a function.
- **context**: runtime information provided for executing the function. For details, see the description of SDK APIs.

  The **HC.Serverless.Function.Common** library needs to be referenced when you deploy a project in FunctionGraph. For details about the **IFunctionContext** object, see the context description.

  When creating a C# function, you need to define a method as the handler of the function. The method can access the function by using specified IFunctionContext parameters. Example:

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

For a C# function, the handler must be named in the format of *[assembly]*::*[namespace].[class name]*::*[execution function name]*. Example:

CsharpDemo::CsharpDemo.Program::MyFunc. You can configure or modify the handler on the **Basic Settings** page on the FunctionGraph console.

Function Handler

ASSEMBLY::NAMESPACE.CLASSNAME::METHODNAME

- **ASSEMBLY**: name of the .NET assembly file for your application, for example, **HelloCsharp**.
- **NAMESPACE** and **CLASSNAME**: names of the namespace and class to which the handler function belongs.
- **METHODNAME**: name of the handler function. Example:

  Set the handler to **HelloCsharp::Example.Hello::Handler** when you create a function.

SDK APIs

- Context APIs

  Table 6-1 describes the provided context attributes.

Table 6-1 Context objects

Attribute	Description
String RequestId	Request ID.
String ProjectId	Project ID.
String PackageName	Name of the group to which the function belongs.
String FunctionName	Function name.
String FunctionVersion	Function version.
Int MemoryLimitInMb	Obtains the allocated memory.
Int CpuNumber	Obtains CPU usage of a function.
String Accesskey	Obtains the AK (valid for 24 hours) with an agency. If you use this method, you need to configure an agency for the function. **FunctionGraph has stopped maintaining the String AccessKey API in the Runtime SDK. You cannot use this API to obtain a temporary AK.**
String Secretkey	Obtains the SK (valid for 24 hours) with an agency. If you use this method, you need to configure an agency for the function. **FunctionGraph has stopped maintaining the String SecretKey API in the Runtime SDK. You cannot use this API to obtain a temporary SK.**

Attribute	Description
String SecurityAccessKey	Obtains the SecurityAccessKey (valid for 24 hours) with an agency. The cache duration is 10 minutes. That is, the same content is returned within 10 minutes. To use this method, you need to configure an agency for the function.
String SecuritySecretKey	Obtains the SecuritySecretKey (valid for 24 hours) with an agency. The cache duration is 10 minutes. That is, the same content is returned within 10 minutes. To use this method, you need to configure an agency for the function.
String SecurityToken	Obtains the SecurityToken (valid for 24 hours) with an agency. The cache duration is 10 minutes. That is, the same content is returned within 10 minutes. To use this method, you need to configure an agency for the function.
String Token	Obtains the token (valid for 24 hours) with an agency. If you use this method, you need to configure an agency for the function.
Int RemainingTimeInMilli-Seconds	Remaining running time of a function.
String GetUserData(string key,string defvalue=" ")	Uses keys to obtain the values passed by environment variables.

- Logging APIs

  The following table describes the logging APIs provided in the C# SDK.

  Table 6-2 Logging APIs

Method	Description

Log(string message)	Creates a logger object by using context.
	var logger = context.Logger;
	logger.Log("hello CSharp runtime test(v1.0.2)");

Logf(string format, args ...interface{})	Creates a logger object by using context.
	var logger = context.Logger;
	var version = "v1.0.2"
	logger.Logf("hello CSharp runtime test({0})", version);

6.2 Developing a C# Event Function

6.2.1 Developing a C# Event Function Using IDE

This section describes how to develop a C# event function using IDE. For details about the C# syntax, initializer and SDK APIs, see **C# Function Development Overview**.

Constraints

When calling a function using APIG, **isBase64Encoded** is valued **true** by default, indicating that the request body transferred to FunctionGraph is encoded using Base64 and must be decoded for processing.

The function must return characters strings by using the following structure.

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

Viewing the.NET Version

Run the following command to view the installed .NET version:

```
dotnet --info
```

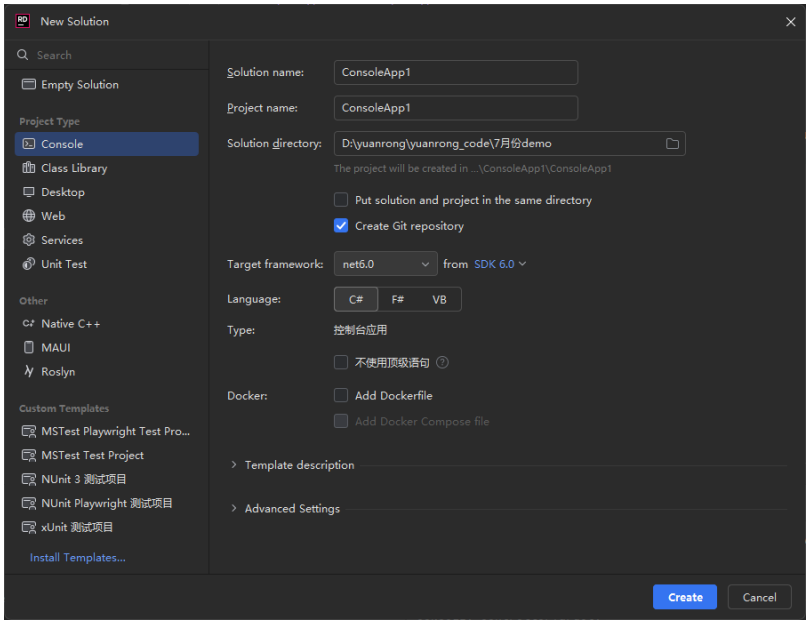
Figure 6-1 Viewing the.NET Version

```
Host:
  Version:      9.0.7
  Architecture: x64
  Commit:       3c298d9f00

.NET SDKs installed:
  6.0.428 [C:\Program Files\dotnet\sdk]
  9.0.302 [C:\Program Files\dotnet\sdk]

.NET runtimes installed:
  Microsoft.AspNetCore.App 6.0.36 [C:\Program Files\dotnet\shared\Microsoft.AspN
etCore.App]
  Microsoft.AspNetCore.App 8.0.18 [C:\Program Files\dotnet\shared\Microsoft.AspN
etCore.App]
  Microsoft.AspNetCore.App 9.0.7 [C:\Program Files\dotnet\shared\Microsoft.AspNe
tCore.App]
  Microsoft.NETCore.App 6.0.36 [C:\Program Files\dotnet\shared\Microsoft.NETCore
.App]
  Microsoft.NETCore.App 8.0.18 [C:\Program Files\dotnet\shared\Microsoft.NETCore
.App]
  Microsoft.NETCore.App 9.0.7 [C:\Program Files\dotnet\shared\Microsoft.NETCore.
App]
  Microsoft.WindowsDesktop.App 6.0.36 [C:\Program Files\dotnet\shared\Microsoft.
WindowsDesktop.App]
  Microsoft.WindowsDesktop.App 8.0.18 [C:\Program Files\dotnet\shared\Microsoft.
WindowsDesktop.App]
  Microsoft.WindowsDesktop.App 9.0.7 [C:\Program Files\dotnet\shared\Microsoft.W
indowsDesktop.App]
```

Step 1: Creating a Project

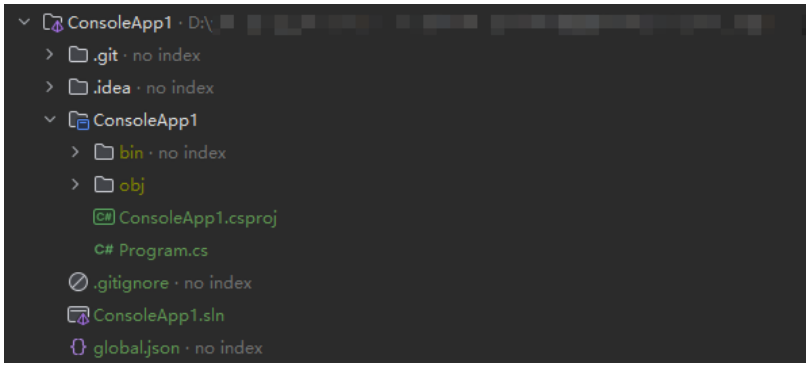
1. Open the IDE and create a C# compilation project, as shown in **Figure 6-2**. Select **net6.0** as the framework.

Figure 6-2 Creating a project



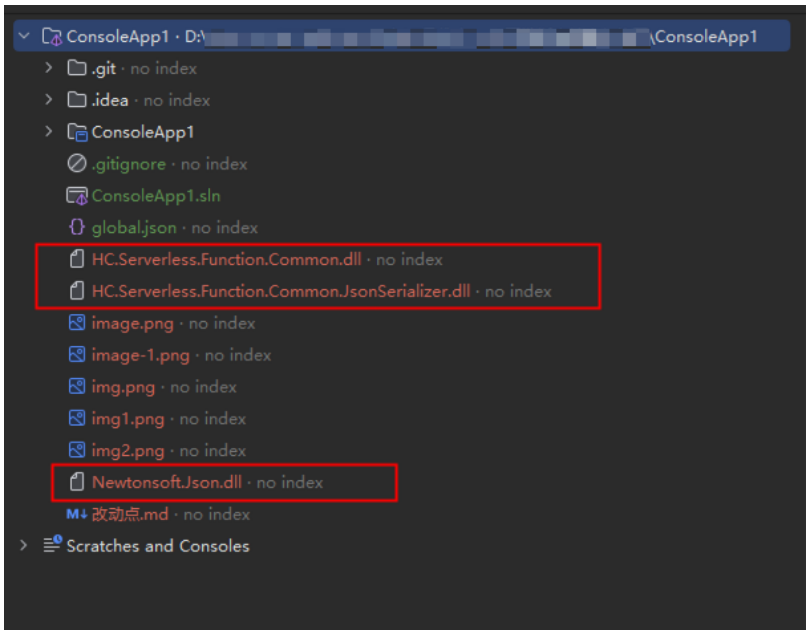
2. Check the directory after the project is created.

Figure 6-3 Project directory structure



3. Download the **.dll file** of the function and decompress the **.dll** file to the directory shown in **Figure 6-4**.

Figure 6-4 Decompressing the .dll file to the directory

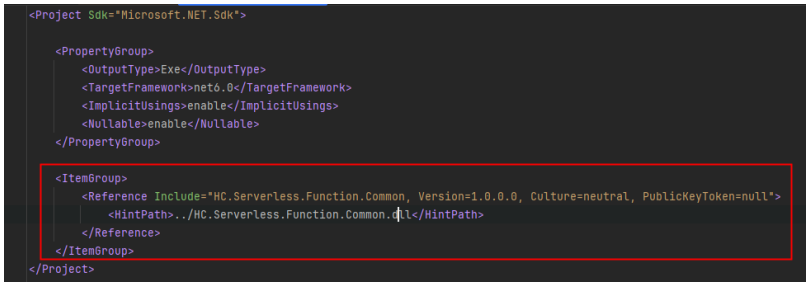


4. Edit the **ConsoleApp1.csproj** file in the project to enable the project to reference the downloaded **.dll** file. The following information is added to the file:

```
<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>
```

After the editing is complete, the content of the **cspoj file** is shown in **Figure 6-5**.

Figure 6-5 csproj file



5. Edit the **Program.cs** file in the project and replace the original code in the file with the following code:

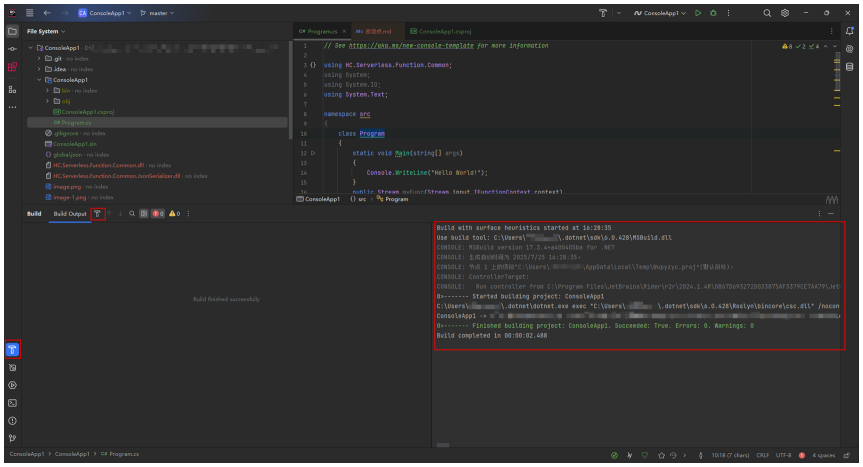
```
using HC.Serverless.Function.Common;
using System;
using System.IO;
using System.Text;

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());
        }
    }
}
```

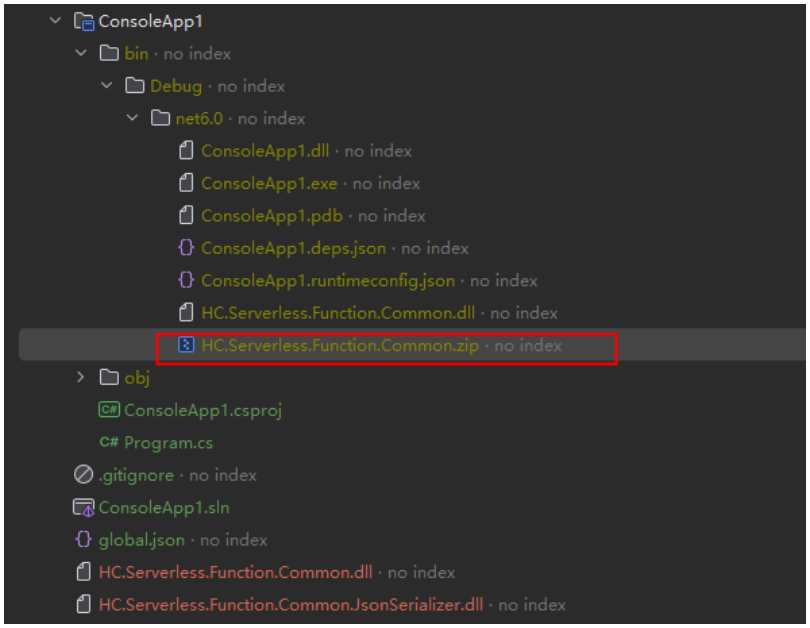
6. As shown in **Figure 6-6**, click the compilation button in the IDE to compile the C# project.

Figure 6-6 C# compilation button



7. Compress all compiled files in the **ConsoleApp1\bin\Debug\net6.0** directory into a .zip file, as shown in **Figure 6-7**. **Do not compress the entire folder. Ensure that files are displayed after the package is decompressed.**

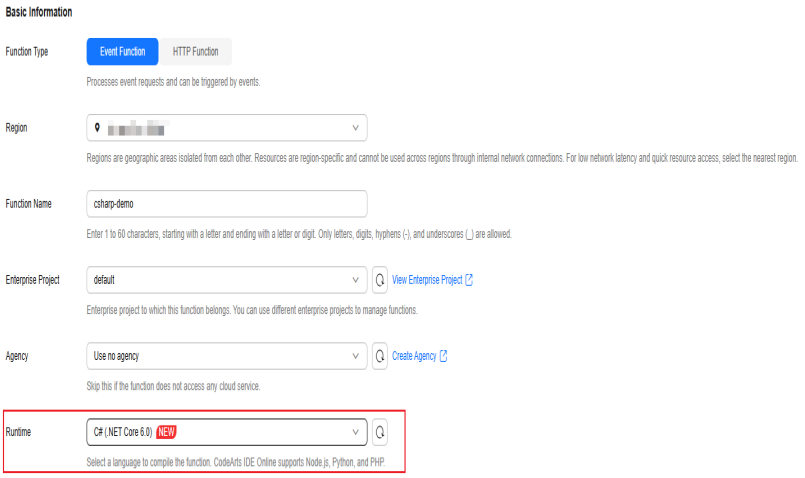
Figure 6-7 Compressing compiled files



Step 2: Testing the Function

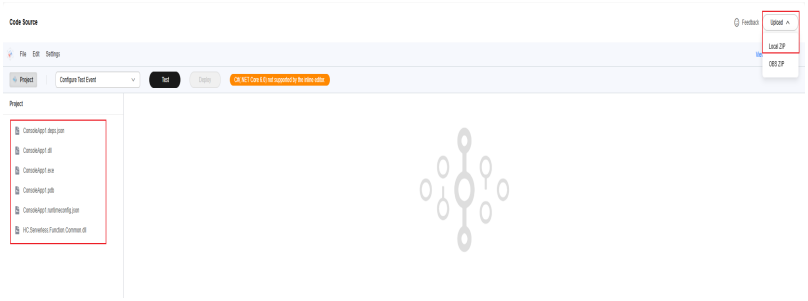
1. Log in to the **FunctionGraph console** and click **Create Function** in the upper right corner.
2. Create a C# event function from scratch and click **Create Now** as shown in**Figure 6-8**. The function details page is displayed.

Figure 6-8 Creating a function



3. Upload the ZIP file compressed in 7. After the code package is uploaded, it will be automatically deployed on the FunctionGraph console, as shown in **Figure 6-9**.

Figure 6-9 Uploading code

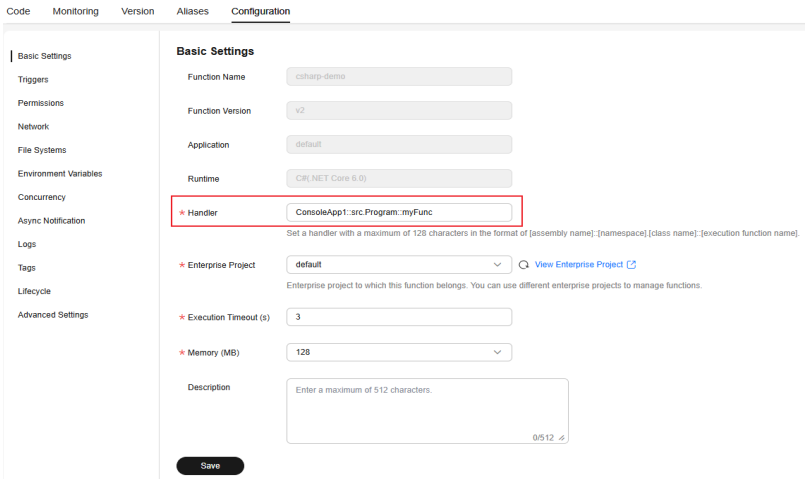


4. Choose **Configuration** > **Basic Settings**, set **Handler** to **ConsoleApp1::src.Program::myFunc**, and click **Save**, as shown in **Figure 6-10**.

   NOTE

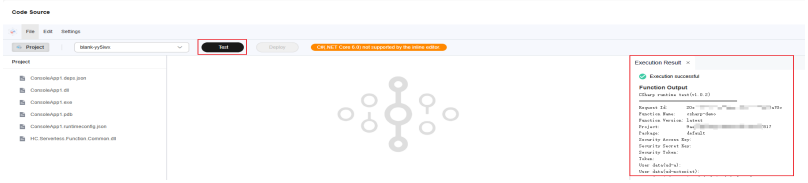
   The file generated after the C# project in the sample project package is compiled is **ConsoleApp1.dll**. Therefore, the assembly name of the function handler is **ConsoleApp1**.

Figure 6-10 Configuring the function handler



5. Return to the **Code** tab page, click **Test** to configure a blank test event.
6. Select the created blank test event, as shown in **Figure 6-11**, and click **Test** to view the function execution result.

Figure 6-11 Testing the function



Execution Result

The execution result consists of the function output, summary, and log output.

Table 6-3 Description of the execution result

Parameter	Successful Execution	Failed Execution

Function Output	The defined function output information is returned.	A JSON file that contains **errorMessage** and **errorType** is returned. The format is as follows:
		{
		"errorMessage": "",
		"errorType": ""
		}
		errorMessage: Error message returned by the runtime.
		errorType: Error type.

Summary	**Request ID**, **Memory Configured**, **Execution Duration**, **Memory Used**, and **Billed Duration** are displayed.	**Request ID**, **Memory Configured**, **Execution Duration**, **Memory Used**, and **Billed Duration** are displayed.
Log Output	Function logs are printed. A maximum of 4 KB logs can be displayed.	Error information is printed. A maximum of 4 KB logs can be displayed.

6.2.2 JSON Serialization and Deserialization

6.2.2.1 Developing a C# Function Using .NET Core CLI

C# supports JSON serialization and deserialization interfaces and provides the **HC.Serverless.Function.Common.JsonSerializer.dll** file.

The interfaces are as follows:

T Deserialize<T>(Stream ins): Deserializes data into objects of function programs.

Stream Serialize<T>(T value): Serializes data to the returned response payload.

The following shows how to create a project named **test** based on .NET Core 6.0. The procedure is similar for other versions. The .NET SDK has been installed in the execution environment.

Creating a Project

1. Run the following command to create the **/tmp/csharp/projects /tmp/csharp/release** directory:
   ```
   mkdir -p  /tmp/csharp/projects;mkdir -p  /tmp/csharp/release
   ```
2. Run the following command to go to the **/tmp/csharp/projects/** directory:
   ```
   cd /tmp/csharp/projects/
   ```
3. Create the project file **test.csproj** with the following content:
   ```
   <Project Sdk="Microsoft.NET.Sdk">
    <PropertyGroup>
   ```



```
    <TargetFramework>net6.0</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>
```

Generating a Code Library

1. Download the **dll files**.

   Upload the **HC.Serverless.Function.Common.dll**, **HC.Serverless.Function.Common.JsonSerializer.dll**, and **Newtonsoft.Json.dll** files in the package to the **/tmp/csharp/projects/** directory.

2. Create the **Class1.cs** file in the **/tmp/csharp/projects/** directory. The code is as follows:

```
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; }//Define the attribute of the serialization class as KetTest.

        }
    }
}
```

3. Run the following command to generate a code library:

```
dotnet build /tmp/csharp/projects/test.csproj -c Release -o /tmp/csharp/release
```

   NOTE

   .NET directory: /home/tools/dotnetcore-sdk/dotnet-sdk-2.1.302-linux-x64/dotnet

4. Run the following command to go to the **/tmp/csharp/release** directory:

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


5. View the compiled .dll files in the **/tmp/csharp/release** directory.

```
-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. Create the **test.runtimeconfig.json** file in the **/tmp/csharp/release** directory. The file content is as follows:

```
{
   "runtimeOptions": {
      "framework": {
         "name": "Microsoft.NETCore.App",
         "version": "6.0.0"
      }
   }
}
```

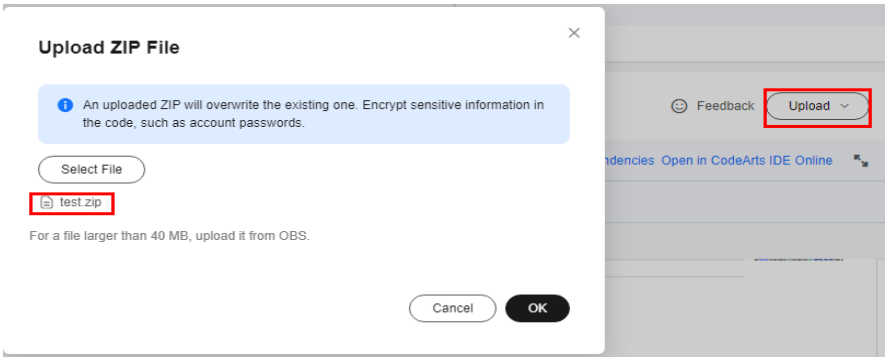
7. Run the following command to package the **test.zip** file in the **/tmp/csharp/release** directory:

```
zip -r test.zip ./*
```

Test Example

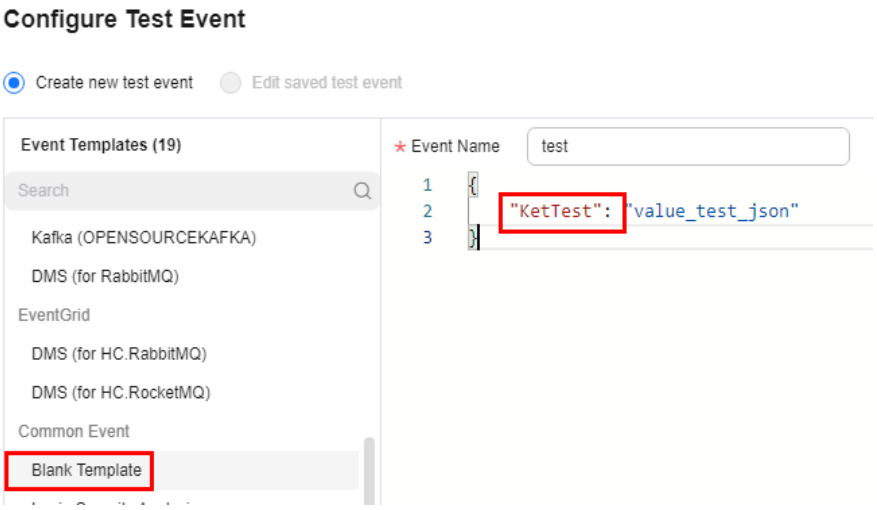
1. Create a C# (.NET 6.0) event function from scratch on the FunctionGraph console and upload the **test.zip** file, as shown in **Figure 6-12**.

Figure 6-12 Uploading the code package



2. Configure a test event, as shown in **Figure 6-13**. The key must be set to **KetTest**, and the value can be customized. (**The test string must be in JSON format.**)

Figure 6-13 Configuring a test event



NOTE

The attribute of the serialization class must be defined as **KetTest**.

3. Click **Test** and view the execution result.

6.2.2.2 Developing a C# Function Using Visual Studio

C# supports JSON serialization and deserialization interfaces and provides the **HC.Serverless.Function.Common.JsonSerializer.dll** file.

The interfaces are as follows:

T Deserialize<T>(Stream ins): Deserializes data into objects of function programs.

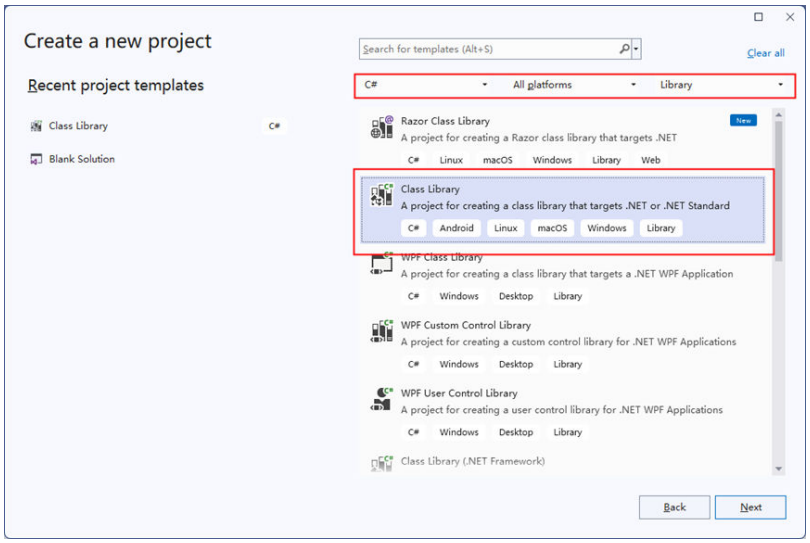
Stream Serialize<T>(T value): Serializes data to the returned response payload.

This section uses Visual Studio 2022 as an example to describe how to create a .NET Core 6.0 **ClassLibrary** project. The procedure for other runtime versions is similar.

Step 1: Creating a Project

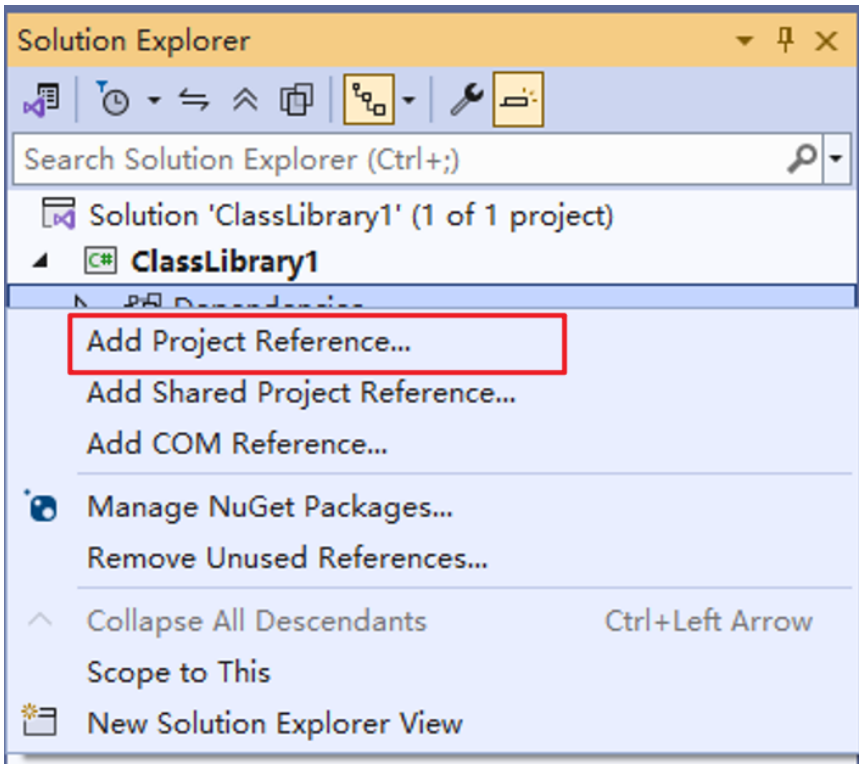
1. In Visual Studio, select **Create a new project**, select **Class Library** as shown in **Figure 6-14**, click **Next**, and select **.NET 6.0** to create a project.

Figure 6-14 Creating a project



2. As shown in **Figure 6-15**, right-click the **ClassLibrary** project in the Solution Explorer and choose **Add Project Reference** from the shortcut menu.

Figure 6-15 Adding a reference

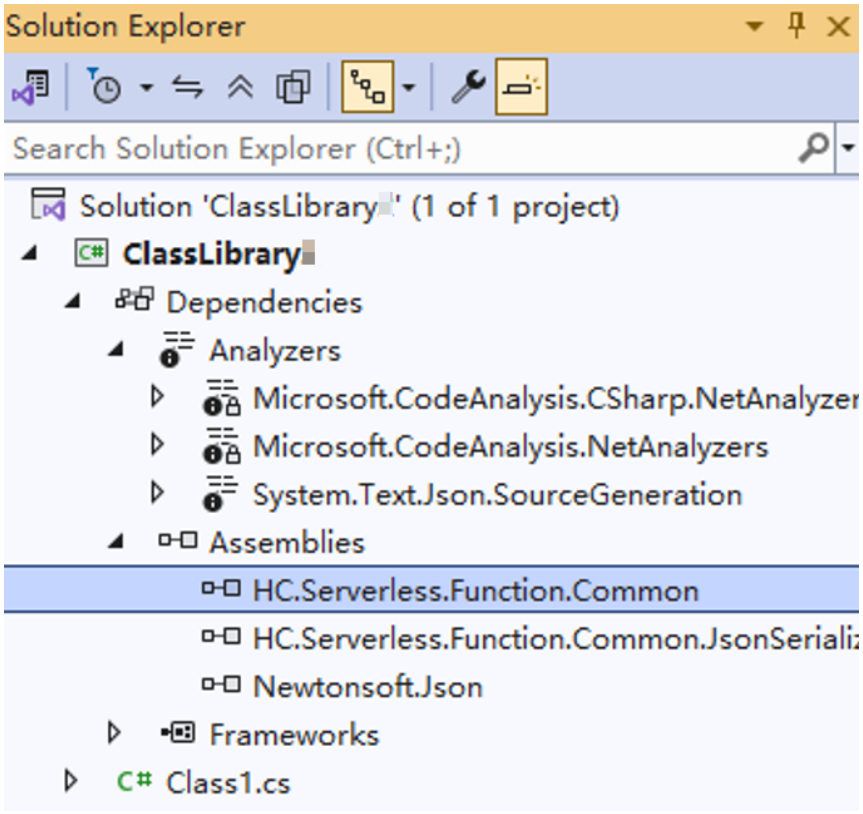


3. Choose **Browse**, click **Browse (B)**, select the three libraries in the downloaded **.dll file** for reference, and click **OK**.

   The .dll file contains three libraries: **HC.Serverless.Function.Common.dll**, **HC.Serverless.Function.Common.JsonSerializer.dll** and **Newtonsoft.Json.dll**.

4. View the references, as shown in **Figure 6-16**.

Figure 6-16 References



Step 2: Packaging Code

Sample code:

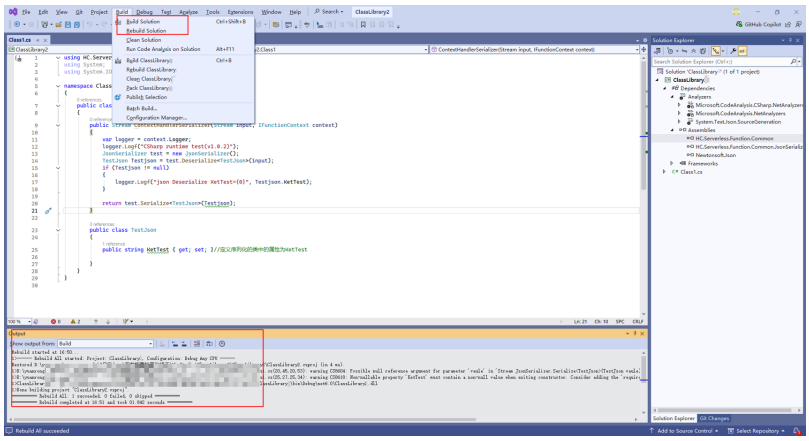
```
using HC.Serverless.Function.Common;
using System;
using System.IO;

namespace ClassLibrary2
{
    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; }//Define the attribute of the serialization class as KetTest.
```



```
    }
  }
}
```

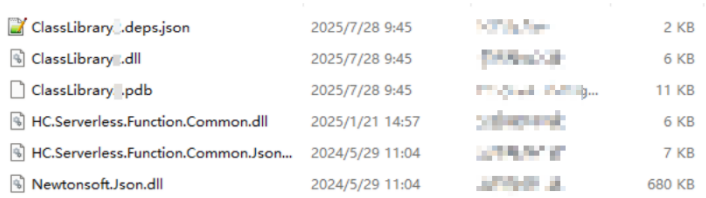
1. As shown in **Figure 6-17**, choose **Build** > **Build Solution** from the menu bar and copy the path of the generated .dll file in the output box.

   Figure 6-17 Generating files

2. View the generated file in the local folder, as shown in **Figure 6-18**.

   Figure 6-18 Files

3. Create the **ClassLibrary.runtimeconfig.json** file in the folder and add the following content to the file: After the operation is complete, there are seven files in total.

   ```
   {
     "runtimeOptions": {
       "framework": {
         "name": "Microsoft.NETCore.App",
         "version": "6.0.0"
       }
     }
   }
   ```

   NOTE

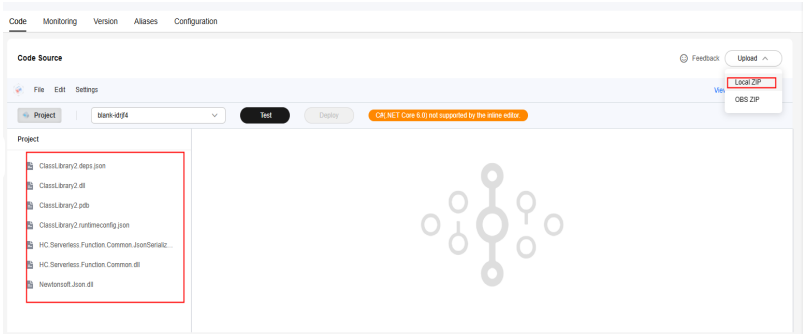
   - The name of the ***.runtimeconfig.json** file is the name of an assembly.
   - The **Version** parameter in the file indicates the version number of the target framework.

4. Compress the files in the folder into a .zip package. **Do not compress the entire folder. Ensure that seven files are displayed after the package is decompressed.**

Step 3: Testing the Function

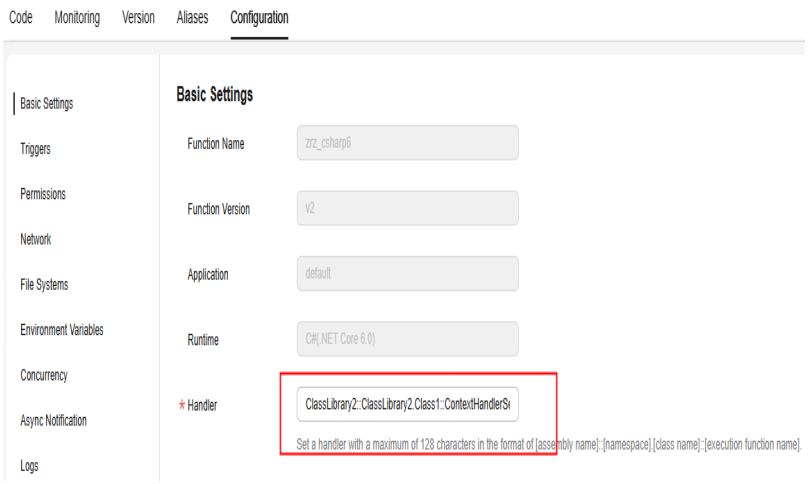
1. On the FunctionGraph console, create a C# (.NET Core 6.0) event function from scratch and upload the ZIP code package created in **4**, as shown in **Figure 6-19**.

   Figure 6-19 Uploading the code package

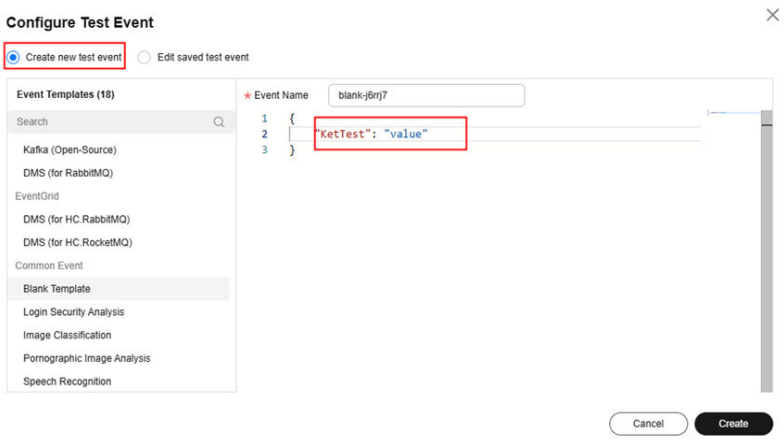
2. Choose **Configuration > Basic Settings**, set **Handler** to **ClassLibrary2::ClassLibrary2.Class1::ContextHandlerSerializer**, and click **Save**, as shown in **Figure 6-20**.

   Figure 6-20 Configuring the function handler

3. Return to the **Code** tab page, click **Test** to configure a test event. As shown in **Figure 6-21**, change **key** to **KetTest** to identify the corresponding value.

Figure 6-21 Configuring a test event

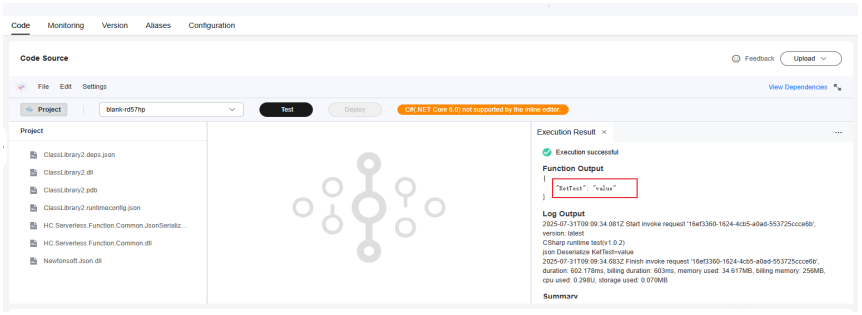


NOTE

The attribute of the serialization class must be defined as **KetTest**.

4. Click **Test** and view the execution result.

Figure 6-22 Viewing the execution result



7 PHP

7.1 PHP Function Development Overview

FunctionGraph supports the following PHP runtimes:

- PHP 7.3
- PHP 8.3

Function Syntax

Use the following syntax when creating a handler function in PHP:

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

- **$handler**: name of the function that FunctionGraph invokes to execute your code. The name must be consistent with that you define when creating a function.
- **$event**: event parameter defined for the function. The parameter is in JSON format.
- **$context**: runtime information provided for executing the function. For details, see **SDK APIs**.
- Function handler: **index.handler**.

The PHP function handler is in the format of *[File name].[Function name]*. You can configure or modify the **handler** on the FunctionGraph console.

PHP Initializer

For details about the initializer, see **Initializer**.

The initializer format of a PHP function is [*File name*].[*Initializer name*].

For example, if the initializer is named **main.my_initializer**, FunctionGraph loads the my_initializer function defined in the **main.php** file.

To use PHP to build initialization logic, define a PHP function as the initializer. The following is a simple initializer:

```
<?php
Function my_initializer($context) {
```



```
    echo 'hello world' . PHP_EOL;
    }
?>
```

- Function name

  The function name **my_initializer** must be the initializer function name specified for a function.

  For example, if the initializer is named **main.my_initializer**, FunctionGraph loads the my_initializer function defined in the **main.php** file.

- context

  The **context** parameter contains the runtime information about a function. For example, request ID, temporary AK, and function metadata.

SDK APIs

The following table describes the context methods provided by FunctionGraph.

Table 7-1 Context methods

Method	Description
getRequestID()	Obtains a request ID.
getRemainingTimeIn-MilliSeconds ()	Obtains the remaining running time of a function.
getAccessKey()	Obtains the AK (valid for 24 hours) with an agency. If you use this method, you need to configure an agency for the function. **FunctionGraph has stopped maintaining the getAccessKey API in the Runtime SDK. You cannot use this API to obtain a temporary AK.**
getSecretKey()	Obtains the SK (valid for 24 hours) with an agency. If you use this method, you need to configure an agency for the function. **FunctionGraph has stopped maintaining the getSecretKey API in the Runtime SDK. You cannot use this API to obtain a temporary SK.**
getSecurityAcces-sKey()	Obtains the SecurityAccessKey (valid for 24 hours) with an agency. The cache duration is 10 minutes. That is, the same content is returned within 10 minutes. To use this method, you need to configure an agency for the function.
getSecuritySecret-Key()	Obtains the SecuritySecretKey (valid for 24 hours) with an agency. The cache duration is 10 minutes. That is, the same content is returned within 10 minutes. To use this method, you need to configure an agency for the function.

Method	Description
getSecurityToken()	Obtains the SecurityToken (valid for 24 hours) with an agency. The cache duration is 10 minutes. That is, the same content is returned within 10 minutes. To use this method, you need to configure an agency for the function.
getUserData(string key)	Uses keys to obtain the values passed by environment variables.
getFunctionName()	Obtains the name of a function.
getRunningTimeIn-Seconds ()	Obtains the timeout of a function.
getVersion()	Obtains the version of a function.
getMemorySize()	Obtains the allocated memory.
getCPUNumber()	Obtains CPU usage of a function.
getPackage()	Obtains a function group.
getToken()	Obtains the token (valid for 24 hours) with an agency. If you use this method, you need to configure an agency for the function.
getLogger()	Obtains the **logger** method provided by the context and returns a log output class. Logs are output in the format of *Time-Request ID-Content* by using the **info** method. For example, use the **info** method to output logs: logg = context.getLogger()$ $logg->info("hello")
getAlias()	Obtains function alias.

7.2 Developing a PHP Event Function

You can develop a PHP event function locally and upload the code file, or create a function on the FunctionGraph console and edit code online.

For details about the syntax and SDK APIs of PHP functions, see **PHP Function Development Overview**.

Constraints

- FunctionGraph can return only the following types of values:
  - **Null**: The HTTP response body is empty.
  - **string**: The content in this string is the body of an HTTP response.

  - **Other**: FunctionGraph returns a value for JSON encoding, and uses the encoded object as the body of an HTTP response. The **Content-Type** header of the HTTP response is set to **text/plain**.
- **For details about the constraints for the APIG event source, see Base64 Decoding and Response Structure.**
- In this example, the function project files are saved under the **~/Code/** directory. Select and package all files under the directory to ensure that the **index.php** file, the handler of your FunctionGraph function, is under the root directory when the **fss_examples_php7.3.zip** file is decompressed.

Developing a PHP Function

Step 1 Create a function.

1. Write code for printing text **helloworld**.

   Open the text editor, compile a HelloWorld function, and save the code file as **helloworld.php**. The code is as follows:

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

2. Define a FunctionGraph function.

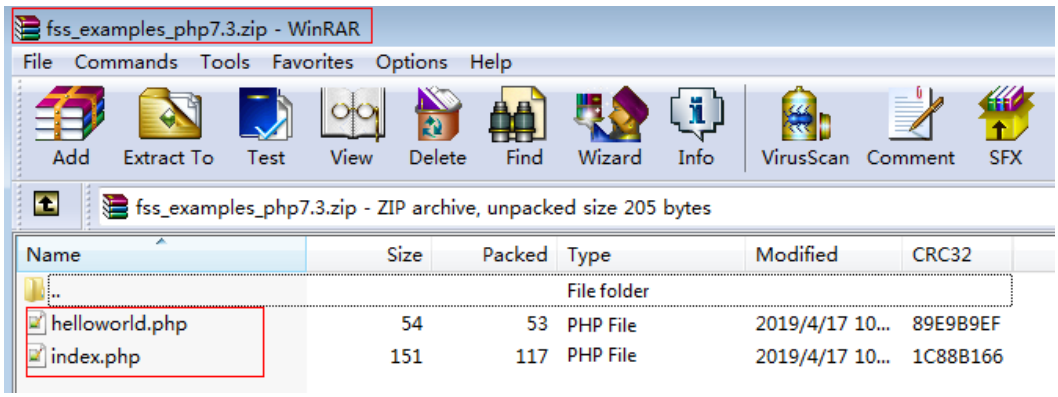
   Open the text editor, define a function, and save the function file as **index.php** under the same directory as the **helloworld.php** file. The function code is as follows:

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

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

Step 2 Package the project files.

After creating the function project, you get the following directory. Select all files under the directory and package them into the **fss_examples_php7.3.zip** file, as shown in **Figure 7-1**.

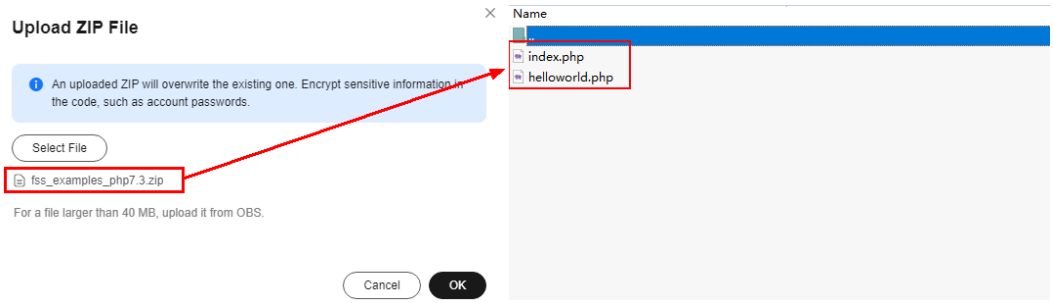
Figure 7-1 Packaging the project files



Step 3 Create a FunctionGraph function and upload the code package.

Log in to the FunctionGraph console, create a PHP function, and upload the **fss_examples_php7.3.zip** file, as shown in **Figure 7-2**.

Figure 7-2 Uploading the code package



- The **index** of the handler must be consistent with the name of the file created in **Step 1.2**, because the file name will help to locate the function file.
- The handler is a function name, which must be the same as that in the **index.php** file created in **Step 1.2**.

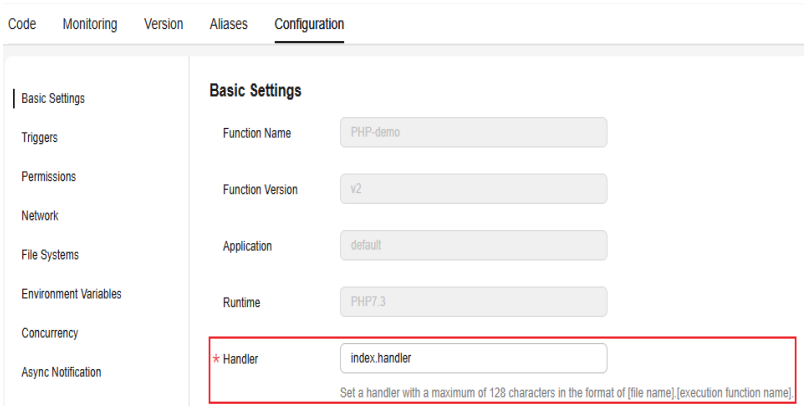
After you upload the **fss_examples_php7.3.zip** file to OBS, when the function is triggered, FunctionGraph decompresses the file to locate the function file through **index** and locate the function defined in the **index.php** file through **handler**, and then executes the function.

NOTE

Modifying the function handler:

In the navigation pane on the left of the FunctionGraph console, choose **Functions** > **Function List**. Click the name of the function to be set. On the function details page that is displayed, choose **Configuration** > **Basic Settings** and set the **Handler** parameter, as shown in **Figure 7-3**.

Figure 7-3 Function handler

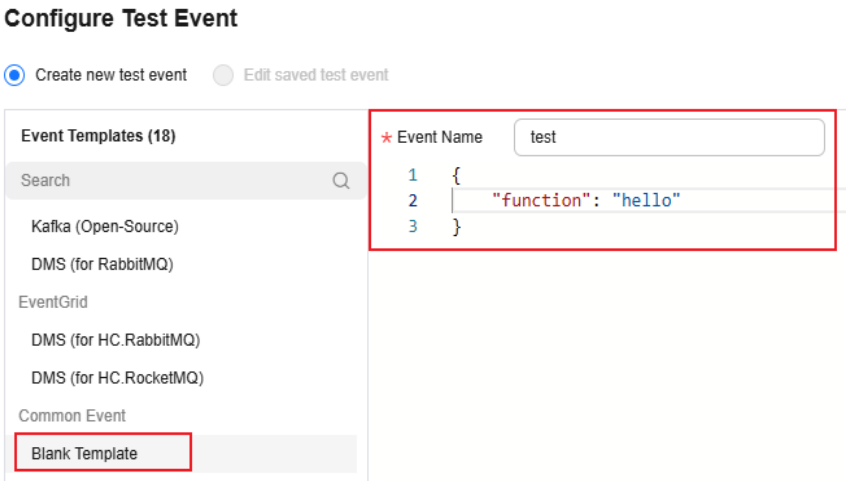


Step 4 Test the function.

1. Create a test event.

   On the function details page that is displayed, click **Configure Test Event**. Configure the test event information, as shown in **Figure 7-4**, and then click **Create**.

Figure 7-4 Configuring a test event



2. On the function details page, select the configured test event, and click **Test**.

Step 5 View the function execution result.

The function execution result consists of three parts: function output (returned by **return**), summary, and logs (output by using the **echo** method).

----End

Execution Result

The execution result consists of the function output, summary, and log output.

Table 7-2 Description of the execution result

Parameter	Successful Execution	Failed Execution

Function Output	The defined function output information is returned.	A JSON file that contains **errorMessage**, **errorType**, and **stackTrace** is returned. The format is as follows:
		{
		"errorMessage": "",
		"errorType": "",
		"stackTrace": {}
		}
		errorMessage: Error message returned by the runtime.
		errorType: Error type.
		stackTrace: Stack error information returned by the runtime.

Summary	**Request ID**, **Memory Configured**, **Execution Duration**, **Memory Used**, and **Billed Duration** are displayed.	**Request ID**, **Memory Configured**, **Execution Duration**, **Memory Used**, and **Billed Duration** are displayed.

Parameter	Successful Execution	Failed Execution
Log Output	Function logs are printed. A maximum of 4 KB logs can be displayed.	Error information is printed. A maximum of 4 KB logs can be displayed.

7.3 PHP Function Template

The following is a sample code template of a PHP function:

```
<?php
/*function initializer($context) {
    $output = 'hello Initializer';
    return $output;
}*/
function handler($event, $context) {
    $output = array(
        "statusCode" => 200,
        "headers" => array(
            "Content-Type" => "application/json",
        ),
        "isBase64Encoded" => false,
        "body" => json_encode($event),
    );
    return $output;
}
?>
```

When you create an empty PHP event function on the FunctionGraph console, the preceding sample code is deployed by default.

7.4 Creating a Dependency for a PHP Function

Setting Up the EulerOS Environment

You are advised to create function dependencies in Huawei Cloud EulerOS 2.0. If other OSs are used, the dynamic link library may not be found due to the differences between underlying dependency libraries.

EulerOS is an enterprise-grade Linux OS based on open-source technology. It features high security, scalability, and performance, meeting customers' requirements for IT infrastructure and cloud computing services.

You can set up the **Huawei Cloud EulerOS** environment using the following methods:

- Buy a EulerOS ECS on Huawei Cloud by referring to **Purchasing and Logging In to a Linux ECS**. On the **Configure Basic Settings** page, select **Public Image**, and select **Huawei Cloud EulerOS** and an image version.
- Download the EulerOS image, and use virtualization software to set up the EulerOS VM on a local PC.

Constraints

If the modules to be installed need dependencies such as **.dll**, **.so**, and **.a**, archive them to a **.zip** package.

Creating a Dependency for a PHP Function

Before creating a dependency, ensure that PHP matching the function runtime has been installed in the environment. The following uses PHP 7.3 as an example to describe how to install the protobuf3.19 dependency using Composer. Composer has been installed in the default environment. The procedure is the same for other PHP versions.

Step 1 Create the **composer.json** file with the following content:

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

Step 2 Run the following command:

```
Composer install
```

The **vendor** folder is generated with the **autoload.php**, **composer**, and **google** subfolders in the current directory.

Step 3 Run the following command to generate a ZIP package:

```
zip –rq vendor.zip vendor
```

----End

If multiple dependencies need to be packaged, specify them in the **composer.json** file, compress the **vendor** folder into a ZIP file and upload it.

NOTE

In PHP projects, third-party dependencies downloaded using Composer need to be loaded using **require "./vendor/autoload.php"**. By default, FunctionGraph stores the decompressed files in a directory at the same level as the project code directory.

8 Custom Runtime

Scenario

A runtime runs the code of a function, reads the handler name from an environment variable, and reads invocation events from the runtime APIs of FunctionGraph. The runtime passes event data to the function handler and returns the response from the handler to FunctionGraph.

FunctionGraph supports custom runtimes. You can use an executable file named **bootstrap** to include a runtime in your function deployment package. The runtime runs the function's handler method when the function is invoked.

Your runtime runs in the FunctionGraph execution environment. It can be a shell script or a binary executable file that is compiled in Linux.

Constraints

- Only ZIP packages can be uploaded using custom runtimes. All dependencies must be included in the ZIP package.
- The **bootstrap** file must be in the root directory of the ZIP package. Ensure that an executable file is generated after the ZIP package is decompressed.
- If you edit code in Go, zip the compiled file, and ensure that the name of the dynamic library file is consistent with the plugin name of the handler. For example, if the name of the dynamic library file is **testplugin.so**, set the handler name to **testplugin.Handler**.

Runtime File bootstrap

If there is a file named **bootstrap** in your function deployment package, FunctionGraph executes that file. If the **bootstrap** file is not found or not executable, your function will return an error when invoked.

The runtime code is responsible for completing initialization tasks. It processes invocation events in a loop until it is terminated.

The initialization tasks run once for each instance of the function to prepare the environment for handling invocations.

Runtime APIs

FunctionGraph provides HTTP runtime APIs to receive function invocation events and returns response data in the execution environment.

- **Next Invocation**

  Method – Get

  Path – http://$RUNTIME_API_ADDR/v1/runtime/invocation/request

  This API is used to retrieve an invocation event. The response body contains the event data. The following table describes additional data about the invocation contained in the response header.

  Table 8-1 Response header information

Parameter	Description
X-Cff-Request-Id	Request ID.
X-CFF-Access-Key	AK of the account. An agency must be configured for the function if this variable is used.
X-CFF-Auth-Token	Token of the account. An agency must be configured for the function if this variable is used.
X-CFF-Invoke-Type	Invocation type of the function.
X-CFF-Secret-Key	SK of the account. An agency must be configured for the function if this variable is used.
X-CFF-Security-Token	Security token of the account. An agency must be configured for the function if this variable is used.

- **Invocation Response**

  Method – POST

  Path – http://$RUNTIME_API_ADDR/v1/runtime/invocation/response/$REQUEST_ID

  This API is used to send a successful invocation response to FunctionGraph. After the runtime invokes the function handler, it publishes the response from the function to the invocation response path.

- **Invocation Error**

  Method – POST

  Path – http://$RUNTIME_API_ADDR/v1/runtime/invocation/error/$REQUEST_ID

  $REQUEST_ID is the value of variable **X-Cff-Request-Id** in the header of an event retrieval response. For more information, see **Table 8-1**.

  $RUNTIME_API_ADDR is a system environment variable. For more information, see **Table 8-2**.

This API is used to send an error invocation response to FunctionGraph. After the runtime invokes the function handler, it publishes the response from the function to the invocation response path.

Runtime Environment Variables

You can use both custom and runtime environment variables in function code. **Table 8-2** lists the runtime environment variables that are used in the FunctionGraph execution environment.

Table 8-2 Runtime environment variables

Key	Value Description
RUNTIME_PROJECT_ID	projectID
RUNTIME_FUNC_NAME	Function name
RUNTIME_FUNC_VERSION	Function version
RUNTIME_PACKAGE	App to which the function belongs
RUNTIME_HANDLER	Function handler
RUNTIME_TIMEOUT	Function timeout
RUNTIME_USERDATA	Value passed through an environment variable
RUNTIME_CPU	Number of allocated CPU cores
RUNTIME_MEMORY	Allocated memory
RUNTIME_CODE_ROOT	Directory that stores the function code
RUNTIME_API_ADDR	Host IP address and port of a custom runtime API

The value of a custom environment variable can be retrieved in the same way as the value of a FunctionGraph environment variable.

Example Description

This example contains one file called **bootstrap**. The file is implemented in Bash. The runtime loads function script from the deployment package. It uses two variables to find the script.

The **bootstrap** file is as follows:

```
#!/bin/sh
set -o pipefail
#Processing requests loop
while true
do
HEADERS="$(mktemp)"
  # Get an event
  EVENT_DATA=$(curl -sS -LD "$HEADERS" -X GET "http://$RUNTIME_API_ADDR/v1/runtime/invocation/
```



```
request")
  # Get request id from response header
  REQUEST_ID=$(grep -Fi x-cff-request-id "$HEADERS" | tr -d '[:space:]' | cut -d: -f2)
  if [ -z "$REQUEST_ID" ]; then
    continue
  fi
  # Process request data
  RESPONSE="Echoing request: hello world!"
  # Put response
  curl -X POST "http://$RUNTIME_API_ADDR/v1/runtime/invocation/response/$REQUEST_ID" -d
"$RESPONSE"
done
```

After loading the script, the runtime processes invocation events in a loop until it is terminated. It uses the API to retrieve invocation events from FunctionGraph, passes the events to the handler, and then sends responses back to FunctionGraph.

To obtain the request ID, the runtime saves the API response header in a temporary file, and then reads the request ID from the **x-cff-request-id** header field. The runtime processes the retrieved event data and sends a response back to FunctionGraph.

The following is an example of source code in Go. It can be executed only after compilation.

```
package main

import (
    "bytes"
    "encoding/json"
    "fmt"
    "io"
    "io/ioutil"
    "log"
    "net"
    "net/http"
    "os"
    "strings"
    "time"
)

var (
    getRequestUrl           = os.ExpandEnv("http://${RUNTIME_API_ADDR}/v1/runtime/invocation/request")
    putResponseUrl          = os.ExpandEnv("http://${RUNTIME_API_ADDR}/v1/runtime/invocation/response/
{REQUEST_ID}")
    putErrorResponseUrl     = os.ExpandEnv("http://${RUNTIME_API_ADDR}/v1/runtime/invocation/error/
{REQUEST_ID}")
    requestIdInvalidError   = fmt.Errorf("request id invalid")
    noRequestAvailableError = fmt.Errorf("no request available")
    putResponseFailedError  = fmt.Errorf("put response failed")
    functionPackage         = os.Getenv("RUNTIME_PACKAGE")
    functionName            = os.Getenv("RUNTIME_FUNC_NAME")
    functionVersion         = os.Getenv("RUNTIME_FUNC_VERSION")

    client = http.Client{
        Transport: &http.Transport{
            DialContext: (&net.Dialer{
                Timeout: 3 * time.Second,
            }).DialContext,
        },
    }
)

func main() {
    // main loop for processing requests.
    for {
        requestId, header, payload, err := getRequest()
```



```
        if err != nil {
            time.Sleep(50 * time.Millisecond)
            continue
        }

        result, err := processRequestEvent(requestId, header, payload)
        err = putResponse(requestId, result, err)
        if err != nil {
            log.Printf("put response failed, err: %s.", err.Error())
        }
    }
}

// event processing function
func processRequestEvent(requestId string, header http.Header, evtBytes []byte) ([]byte, error) {
    log.Printf("processing request '%s'.", requestId)
    result := fmt.Sprintf("function: %s:%s:%s, request id: %s, headers: %+v, payload: %s", functionPackage,
functionName,
        functionVersion, requestId, header, string(evtBytes))

    var event FunctionEvent
    err := json.Unmarshal(evtBytes, &event)
    if err != nil {
        return (&ErrorMessage{ErrorType: "invalid event", ErrorMessage: "invalid json formated
event"}).toJsonBytes(), err
    }

    return (&APIGFormatResult{StatusCode: 200, Body: result}).toJsonBytes(), nil
}

func getRequest() (string, http.Header, []byte, error) {
    resp, err := client.Get(getRequestUrl)
    if err != nil {
        log.Printf("get request error, err: %s.", err.Error())
        return "", nil, nil, err
    }
    defer resp.Body.Close()

    // get request id from response header
    requestId := resp.Header.Get("X-CFF-Request-Id")
    if requestId == "" {
        log.Printf("request id not found.")
        return "", nil, nil, requestIdInvalidError
    }

    payload, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        log.Printf("read request body error, err: %s.", err.Error())
        return "", nil, nil, err
    }

    if resp.StatusCode != 200 {
        log.Printf("get request failed, status: %d, message: %s.", resp.StatusCode, string(payload))
        return "", nil, nil, noRequestAvailableError
    }

    log.Printf("get request ok.")
    return requestId, resp.Header, payload, nil
}

func putResponse(requestId string, payload []byte, err error) error {
    var body io.Reader
    if payload != nil && len(payload) > 0 {
        body = bytes.NewBuffer(payload)
    }

    url := ""
    if err == nil {
        url = strings.Replace(putResponseUrl, "{REQUEST_ID}", requestId, -1)
```



```
    } else {
        url = strings.Replace(putErrorResponseUrl, "{REQUEST_ID}", requestId, -1)
    }

    resp, err := client.Post(strings.Replace(url, "{REQUEST_ID}", requestId, -1), "", body)
    if err != nil {
        log.Printf("put response error, err: %s.", err.Error())
        return err
    }
    defer resp.Body.Close()

    responsePayload, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        log.Printf("read request body error, err: %s.", err.Error())
        return err
    }

    if resp.StatusCode != 200 {
        log.Printf("put response failed, status: %d, message: %s.", resp.StatusCode, string(responsePayload))
        return putResponseFailedError
    }

    return nil
}

type FunctionEvent struct {
    Type string `json:"type"`
    Name string `json:"name"`
}

type APIGFormatResult struct {
    StatusCode      int               `json:"statusCode"`
    IsBase64Encoded bool              `json:"isBase64Encoded"`
    Headers         map[string]string `json:"headers,omitempty"`
    Body            string            `json:"body,omitempty"`
}

func (result *APIGFormatResult) toJsonBytes() []byte {
    data, err := json.MarshalIndent(result, "", "  ")
    if err != nil {
        return nil
    }

    return data
}

type ErrorMessage struct {
    ErrorType     string `json:"errorType"`
    ErrorMessage  string `json:"errorMessage"`
}

func (errMsg *ErrorMessage) toJsonBytes() []byte {
    data, err := json.MarshalIndent(errMsg, "", "  ")
    if err != nil {
        return nil
    }

    return data
}
```

Table 8-3 describes the environment variables used in the preceding code.

Table 8-3 Runtime environment variables

Environment Variable	Description
RUNTIME_FUNC_NAME	Function name

Environment Variable	Description
RUNTIME_FUNC_VERSION	Function version
RUNTIME_PACKAGE	App to which the function belongs

9 Development Tools

9.1 FunctionGraph and IaC

Combination of FunctionGraph and IaC

Most FunctionGraph functions do not run independently, but work with storage, gateways, databases, and message queues to form serviceless applications. Infrastructure as code (IaC) automates the frequent deployments and updates of functions and triggers. This approach shortens the development cycle, simplifies configuration management, and maintains the consistency of resource deployment.

Suitable IaC Tools for FunctionGraph

Resource Formation Service (RFS) is a new final-state cloud resource orchestration engine that fully supports Terraform (HCL and Provider), the industry's de facto standard for infrastructure as code. It automatically builds cloud resources in batches based on open ecosystem templates that use the HashiCorp Configuration Language (HCL) syntax. With RFS, you can create, manage, and upgrade cloud resources (such as FunctionGraph functions, APIG gateways, and database instances) efficiently, securely, and consistently.

Getting Started with RFS for FunctionGraph

Step 1 Create the Python script file **index.py** with the following content:

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

Step 2 Compress **index.py** into **index.zip**, **upload the ZIP file to an OBS bucket**, and obtain the file's **object link** in the bucket.

Step 3 Create the **variables.tf** file using the following content to define the parameters to be used in your RFS template.

```
variable "enterprise_project_id" {
  type        = string
  description = " Specifies enterprise_project_id"
  default     = "0"
}
variable "agency_name" {
  type        = string
  description = " Specifies the agency to which the function belongs."
  default     = ""
}
variable "region" {
  type        = string
  description = " Specifies the region."
  default     = "cn-north-7"
}
variable "code_url" {
  type        = string
  description = "code_url"
}
variable "apig_id" {
  type        = string
  description = "apig_id"
  default     = ""
}
```

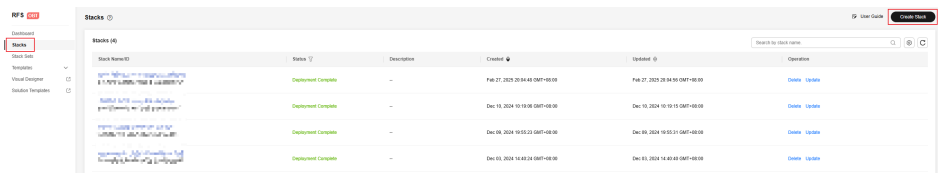
Step 4 Create the RFS template file **main.tf** using the following content to define functions and triggers.

```
variable "enterprise_project_id" {
  type        = string
  description = " Specifies enterprise_project_id"
  default     = "0"
}
variable "agency_name" {
  type        = string
  description = " Specifies the agency to which the function belongs."
  default     = ""
}
variable "region" {
  type        = string
  description = " Specifies the region."
  default     = "cn-north-7"
}
variable "code_url" {
  type        = string
  description = "code_url"
}
variable "apig_id" {
  type        = string
  description = "apig_id"
  default     = ""
}
```

Step 5 Compress **main.tf** and **variables.tf** into the **components.zip** file.

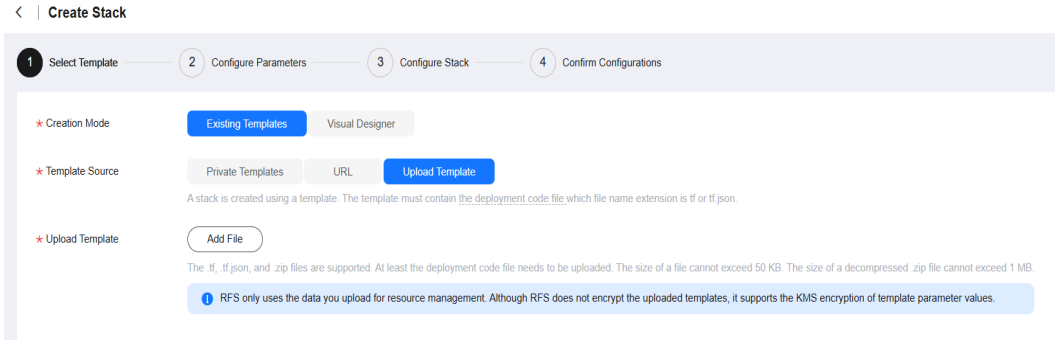
Step 6 Log in to the RFS console, and choose **Stacks** > **Create Stack** as shown in **Figure 9-1**. For details, see **Creating a Stack**.

Figure 9-1 RFS console



Step 7 Set parameters as shown in **Figure 9-2**, and upload **components.zip**. Then click **Next**.

Figure 9-2 Setting parameters



Step 8 On the **Configure Parameters** page, set the following parameters, and click **Next**.

- **enterprise_project_id**: enterprise project ID
- **agency_name**: function agency name
- **region**: region ID. Obtain it from **Regions and Endpoints**.
- **code_url**: OBS object link obtained in **step 2**
- **apig_id**: APIG gateway ID

Step 9 On the **Configure Stack** page, specify an IAM agency, retain the default values for other parameters, and click **Next** to confirm the deployment of this stack. The specified function and resource are automatically created on the FunctionGraph console.

Step 10 Log in to the **FunctionGraph console**, and click the function name in the function list to go to the details page. Choose **Configuration** > **Triggers**, locate the APIG trigger, copy its calling URL, and paste the URL into your browser or run the following **curl** command.

```
curl -s <Trigger_Invoke_URL>  # Replace <Trigger_Invoke_URL> with the calling URL of your APIG trigger.
```

The response is a JSON representation of the selected attributes in the original event object. It contains information about the request sent to the APIG endpoint. The following is an example response:

```
HTTP/1.1 200 OK
Content-Length: 658
Connection: keep-alive
Content-Type: application/json
Date: Wed, 12 Mar 2025 08:59:18 GMT
Server: api-gateway
Strict-Transport-Security: max-age=31536000; includeSubdomains;
X-Apig-Latency: 52
X-Apig-Ratelimit-Api: remain:97,limit:100,time:1 minute
X-Apig-Ratelimit-Api: remain:29973,limit:30000,time:1 second
X-Apig-Ratelimit-Api-Allenv: remain:199,limit:200,time:1 second
X-Apig-Ratelimit-Api-Allenv: remain:29973,limit:30000,time:1 second
X-Apig-Ratelimit-User: remain:3995,limit:4000,time:1 second
X-Apig-Upstream-Latency: 14
X-Cff-Billing-Duration: 1
X-Cff-Invoke-Summary:
{"funcDigest":"e6e9c99b8f5b9d6f9408d5210263330","duration":0.756,"billingDuration":1,"memorySize":128,
"memoryUsed":33.207,"podName":"pool22-300-128-fusion-844bdc7755-
bh55w","gpuMemorySize":0,"ephemeralStorage":512}
X-Cff-Request-Id: b43781ee-49f3-4762-8c24-236c718d3391
```



```
X-Content-Type-Options: nosniff
X-Download-Options: noopen
X-Frame-Options: SAMEORIGIN
X-Func-Err-Code: 0
X-Is-Func-Err: false
X-Request-Id: 90a48d7a4c699780579f4edc8983cdaf
X-Xss-Protection: 1; mode=block;

{"requestContext": {"requestId": "90a48d7a4c699780579f4edc8983cdaf", "apiId":
"01127600bb9f4d2ca8e532d1c378d8c8", "stage": ":DEBUG"}, "queryStringParameters": {}, "path": "/nxy-
sasa", "httpMethod": "GET", "isBase64Encoded": true, "headers": {"host":
"47f32d1efa1742f5a7ee5b720ca9c4a5.apig.cn-east-3.huaweicloudapis.com", "content-type": "application/
json", "x-forwarded-host": "47f32d1efa1742f5a7ee5b720ca9c4a5.apig.cn-east-3.huaweicloudapis.com",
"user-agent": "APIGatewayDebugClient/1.0", "x-forwarded-port": "443", "x-forwarded-proto": "https", "x-
request-id": "90a48d7a4c699780579f4edc8983cdaf", "x-apig-mode": "debug"}, "body": "",
"pathParameters": {}}
```

----End

9.2 Local Debugging with VSCode

Introduction

Huawei Cloud FunctionGraph is a Visual Studio Code (VSCode) plug-in of Huawei Cloud serverless products. With this plug-in, you can:

- Quickly create local functions.
- Run and debug local functions and deploy them to the cloud.
- Pull the function list from the cloud, call cloud functions, and upload ZIP packages to the cloud.

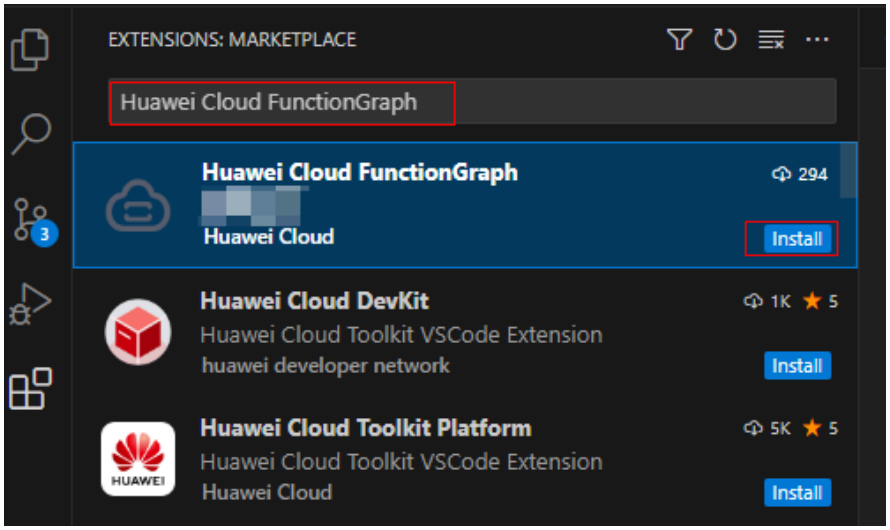
Prerequisites

You have downloaded the **VSCode tool (later than 1.63.0)** and installed it.

Installing the Plug-in

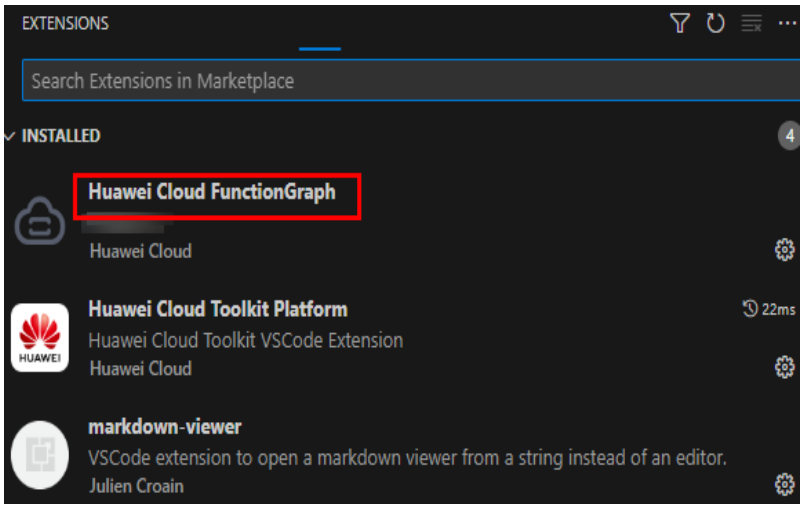
1. Open VSCode, search for **Huawei Cloud FunctionGraph** in the app store, and install it.

   Figure 9-3 Searching for and installing Huawei Cloud FunctionGraph



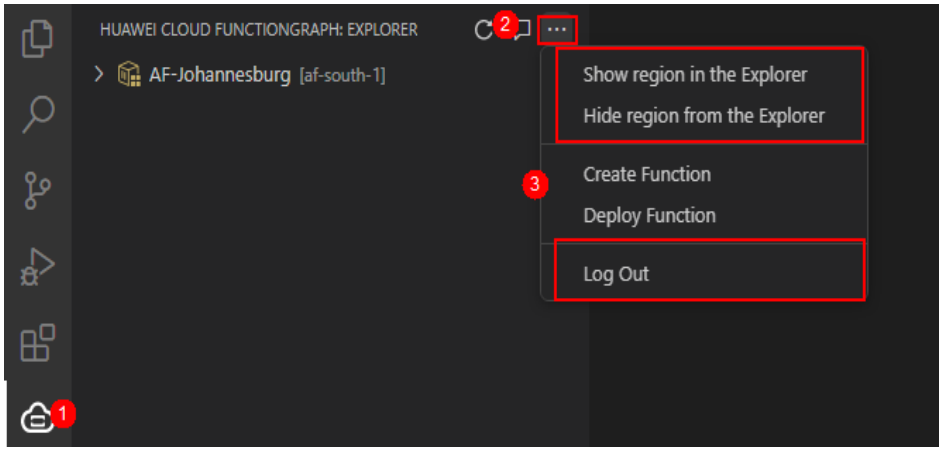
2. After the installation is successful, **Huawei Cloud FunctionGraph** is displayed in the plug-in list.

Figure 9-4 Installed plug-ins



Logging In to FunctionGraph Plug-in

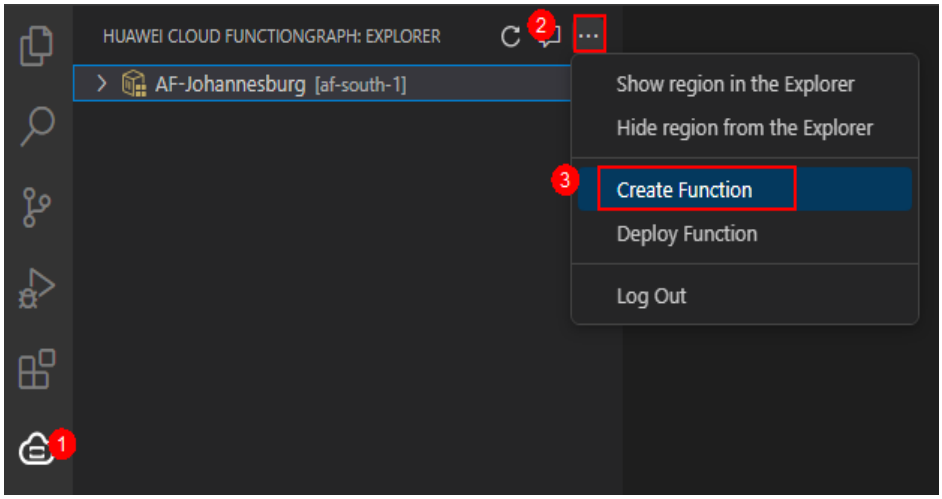
1. Click the **Huawei Cloud FunctionGraph** plug-in icon, click the login link, and select a login mode. If you select login with AK/SK, obtain an AK/SK. For details, see **Creating an Access Key**.
2. Select a region to view function information.
3. Show or hide desired regions, or log out of your account by referring to the following figure.
   - **Show region in the Explorer**: Show the regions where you need to perform operations.
   - **Hide region from the Explorer**: Hide the regions you do not need.
   - **Log Out**: Log out of your account.



Creating a Function

1. On the plug-in panel, select **Create Function** or press **Ctrl+Shift+p** to search for the **Create Function** command. Then, specify the runtime, template,

function name, and local folder as prompted. A function is created in the specified folder.



2. After the local function is created, the handler file is automatically opened.
3. You can customize the function's configuration by modifying the automatically generated configuration file. The parameters are as follows:

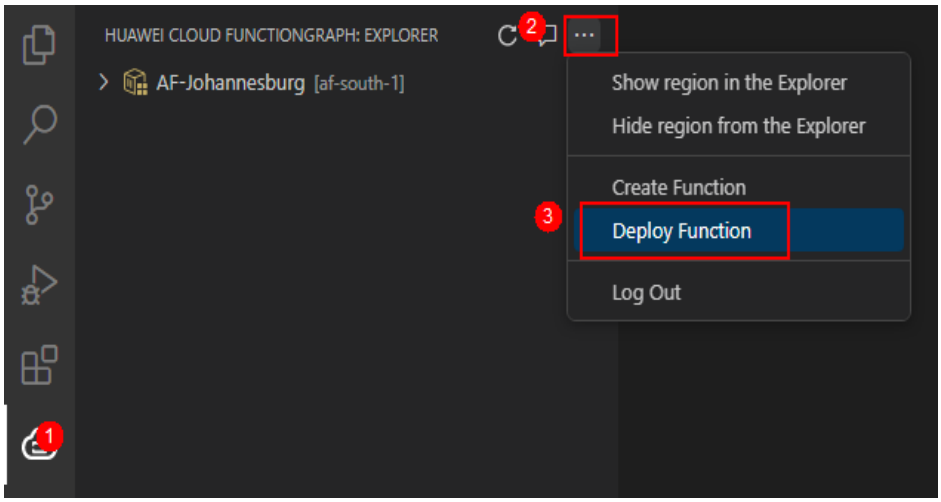
```
HcCrmTemplateVersion: v2
Resources:
  Type: HC::Serverless::Function
  Properties:
    Name: functionName //Function name
    Handler: handler // Handler of the function
    Runtime: runtime // Function runtime
    CodeType: inline // Default
    CodeFileName: index.zip // Default
    CodeUrl: ""
    Description: '' // Function runtime
    MemorySize: 128 // Memory for executing the function
    Timeout: 30 // Function timeout, in seconds.
    Version: latest // Default
    Environment:
      Variables: {} // Environment variables
    InitializerHandler: "" // Function initializer
    InitializerTimeout: 0 // Initialization timeout of the function
    EnterpriseProjectId: "0" // Enterprise project
    FuncType: v2
    URN: "" // Function URN, which is generated after the function is downloaded.
```

Deploying a Function

- **Prerequisites**

  Ensure that the function code path is correct. The code of Node.js, Python, and PHP functions is stored in the **src** directory, and the code of other functions is stored in the root directory.

On the plug-in panel, select **Deploy Function** or press **Ctrl+Shift+p** to search for the **Deploy Function** command, and select a function and region as prompted.



- If the deployment is successful, a success message is displayed in the lower right corner of the page. Switch to the target region to view the deployment result.
- If the deployment fails, view the error log in the **Output** area and rectify the fault.

Local Debugging

1. Node.js
   - **Prerequisites**

     Node.js has been installed in the local environment.
   - **Default mode**

     Click **Local Debug** of the handler method, configure the event content, and click **Invoke** for debugging.

     Figure 9-5 Clicking Local Debug

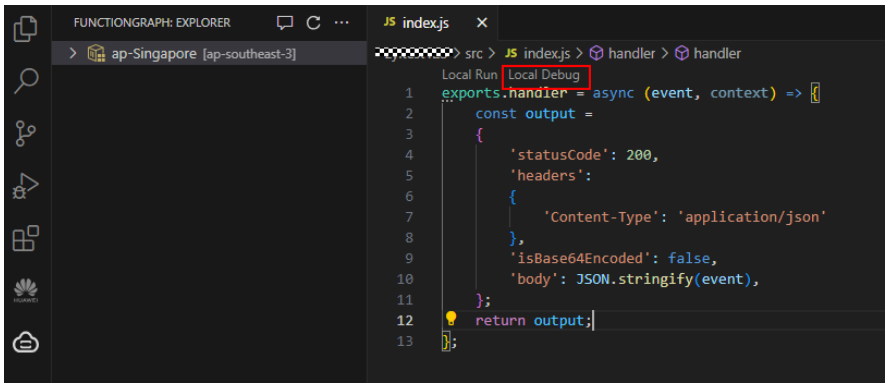
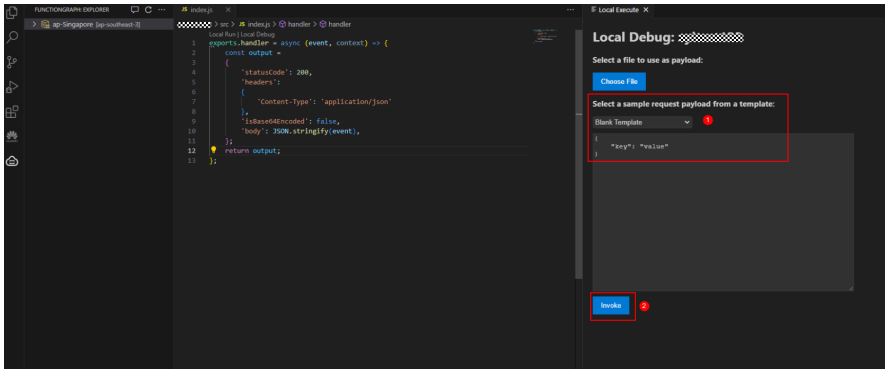


Figure 9-6 Configuring the event content



– **Debugging with VS Code**

Create the **main.js** file in the function folder and copy the following content to this file. Click the  icon on the left. Then, click **Add Config**, select Node.js, and press **F5** to start debugging.

```
const handler = require('./index'); //Path of the function handler file. Modify it as required.
const event = { 'hello': 'world' }; //Test event. Modify it as required.
const context = {}; //Context class.
console.log(handler.handler(event, context));
```

2. Python

– **Prerequisites**

Python has been installed in the local environment.

Create the **main.py** file in the function folder and copy the following content to this file. Click the  icon on the left. Then, click **Add Config**, select Python, and press **F5** to start debugging.

```
import sys
import index #Path of the function handler file. Modify it as required.

#The main method is used for debugging, and event is the selected debugging event.
if __name__ == '__main__':
....event = { 'hello': 'world' } #Test event. Modify it as required.
    context = ''
    content = index.handler(event, context)
....print('Returned value:')
    print(content)
```

3. Java

– **Prerequisites**

Java has been installed. VS Code supports Java testing.

In the **test** directory of the function folder, open the **TriggerTestsTest.java** file, and click the  icon on the left. Then, click **Add Config**, select Java, and press **F5** to start debugging.

Other Functions

- **Opening in Portal**

  Right-click the target function, and choose **Open in Portal**. The function details page is displayed.

- **Executing a Cloud Function**
  a. Right-click the target function and choose **Invoke Function...** from the shortcut menu.
  b. In the **Invoke Function** panel, select the event to be passed and click **Invoke**. The function log and result are displayed in the **Output** area.
- **Downloading a Cloud Function**
  – **Prerequisites**
    You have been granted the permission (**obs:object:GetObject**) for obtaining bucket objects.

  Right-click the function you want to download and choose **Download...** from the shortcut menu. The function code is downloaded from the cloud to the specified local path, and the handler file is automatically opened.
- **Updating a Cloud Function**
  Right-click the target function, choose **Upload Function...** from the shortcut menu, and select a ZIP package to upload.
- **Deleting a Cloud Function**
  a. Right-click the function to be deleted and choose **Delete...** from the shortcut menu.
  b. In the confirmation dialog box, click **Delete** to delete the function.
- **Copying URN**
  Right-click the target function and choose **Copy URN** from the shortcut menu.

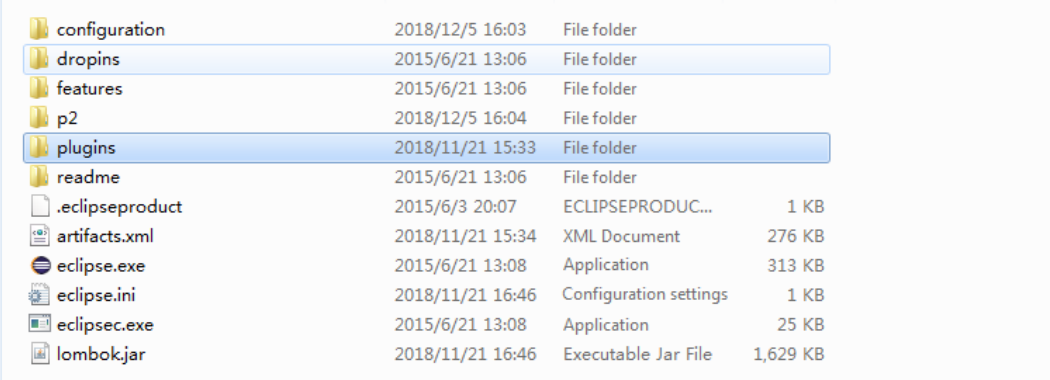
9.3 Eclipse Plug-in

Currently, FunctionGraph does not provide Java function templates and only allows you to upload Java function packages through OBS. With the Eclipse plug-in, you can quickly create Java function templates, package function project files, upload function packages to OBS, and deploy functions.

Step 1 Obtain the **Eclipse plug-in** (software package verification file: **Eclipse plug-in.sha256**).

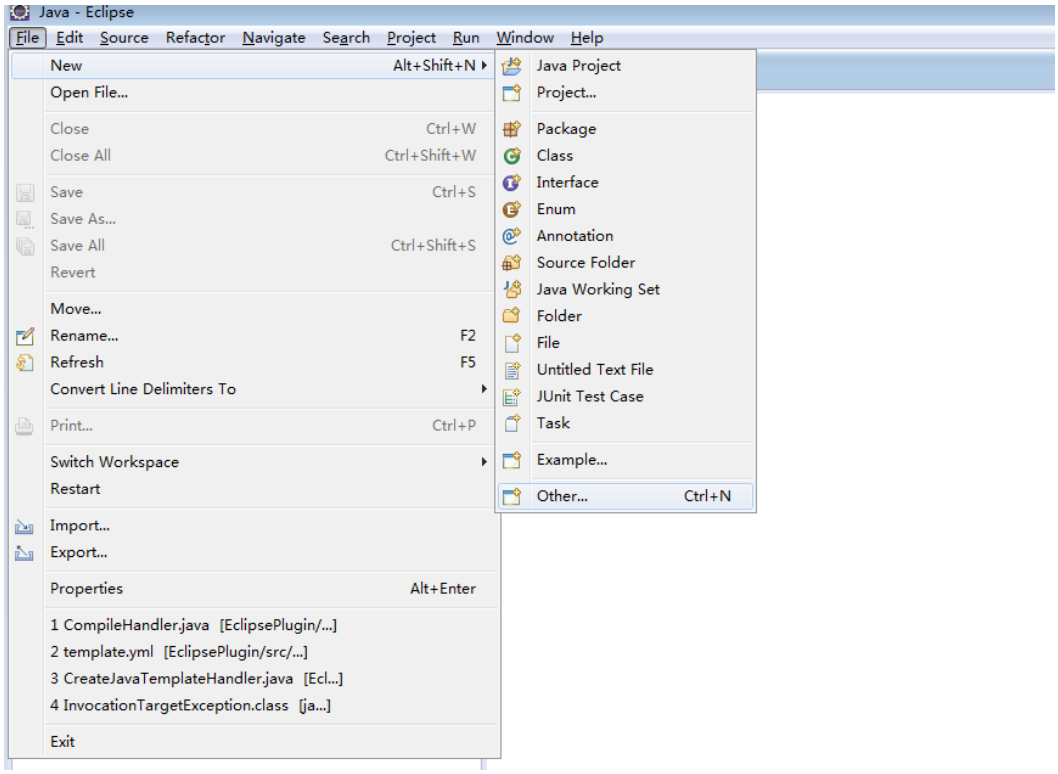
Step 2 Put the Eclipse plug-in package (.jar or .zip) in the **plugins** folder under the Eclipse installation directory. Then restart Eclipse. **Figure 9-7** shows the Eclipse installation directory.

Figure 9-7 Installing the Eclipse plug-in



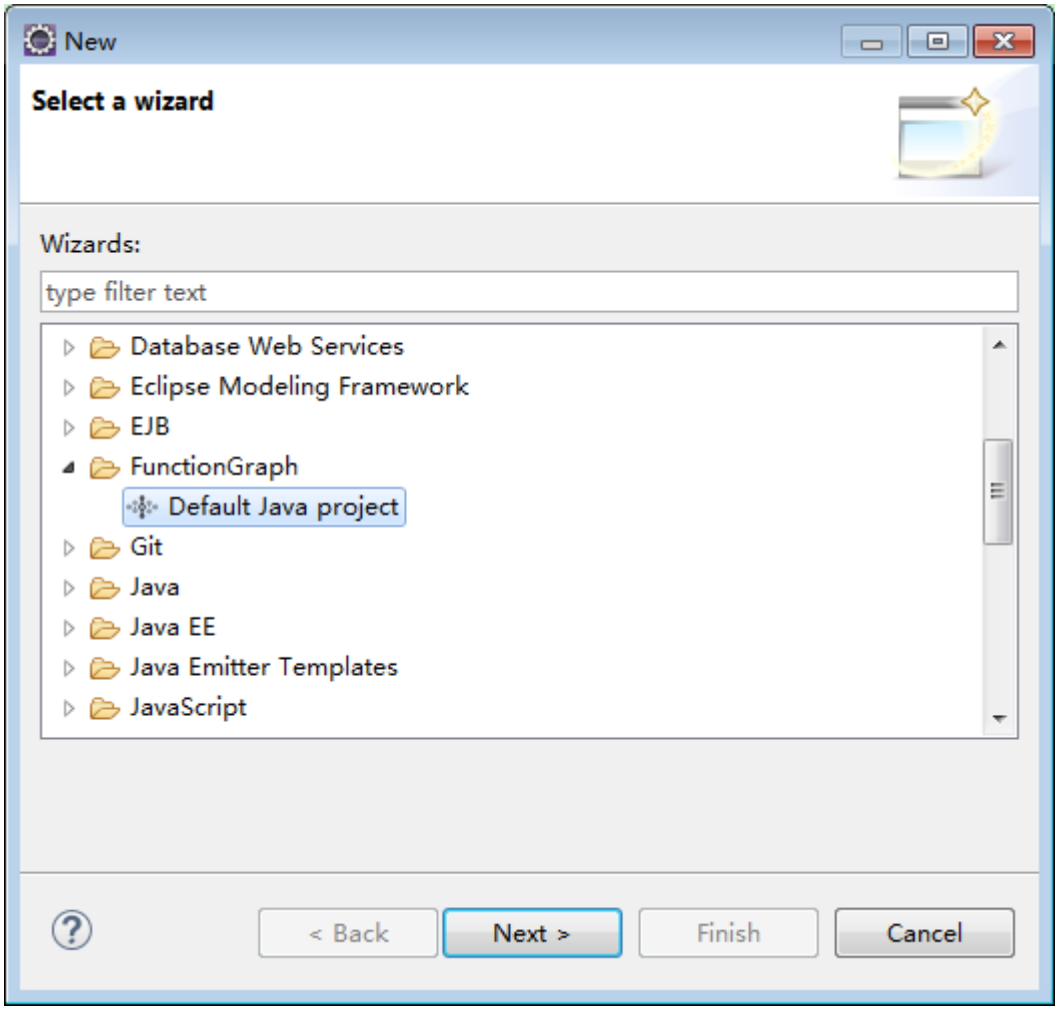
Step 3 Open Eclipse and choose **File** > **New** > **Other**, as shown in **Figure 9-8**.

Figure 9-8 Creating a template



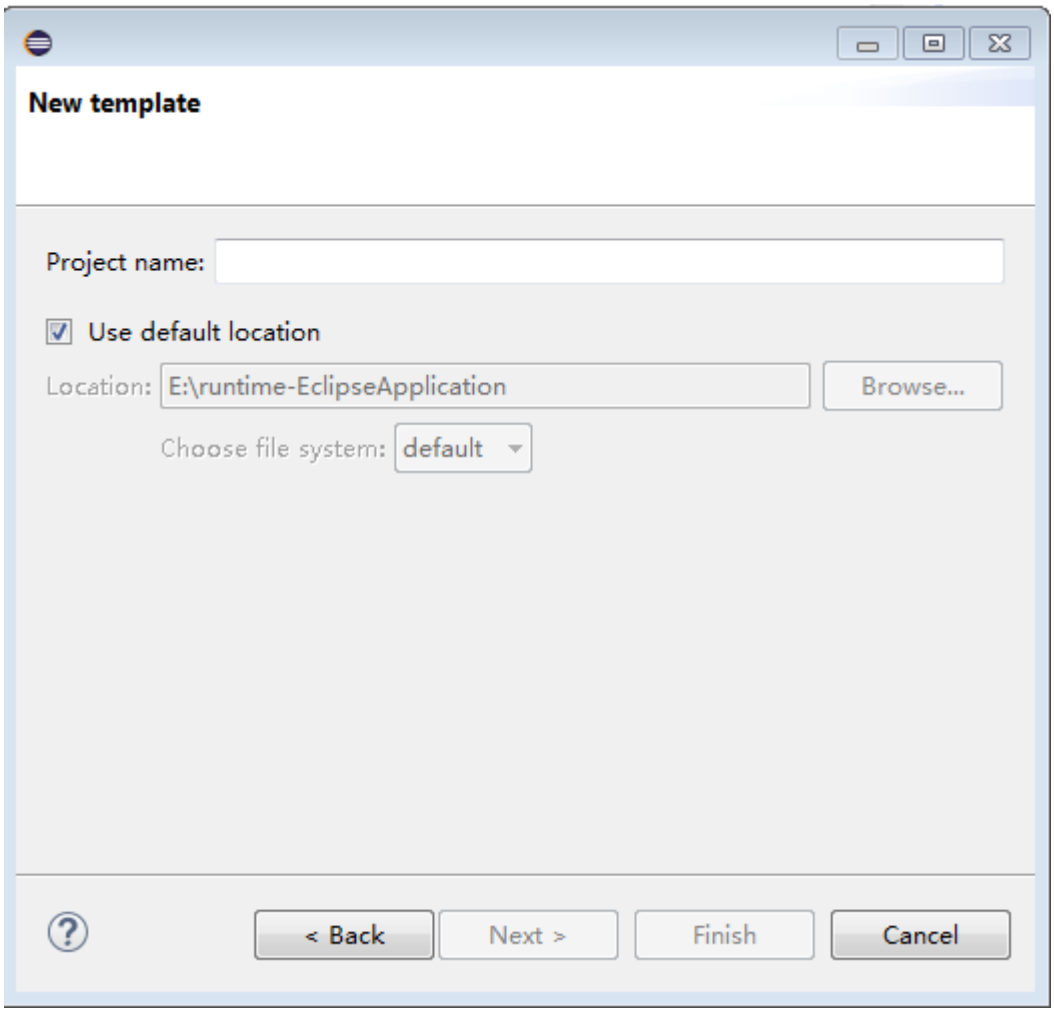
Step 4 Choose **FunctionGraph** > **Default Java project**, as shown in **Figure 9-9**.

Figure 9-9 Selecting the default Java template



Step 5 Enter a project name, specify a project directory (or use the default directory), and click **Finish**, as shown in **Figure 9-10**.

Figure 9-10 Setting the project name and directory



----End

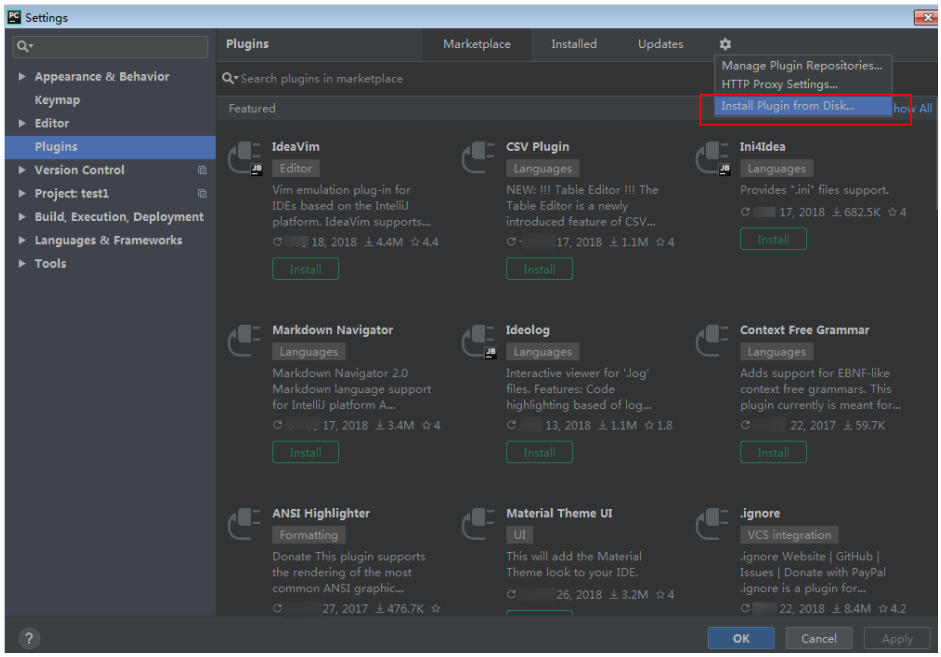
9.4 PyCharm Plug-in

With PyCharm, you can quickly generate Python templates, package project files, and deploy Python functions.

Step 1 Obtain the **plug-in** (**Plug-in.sha256**) .

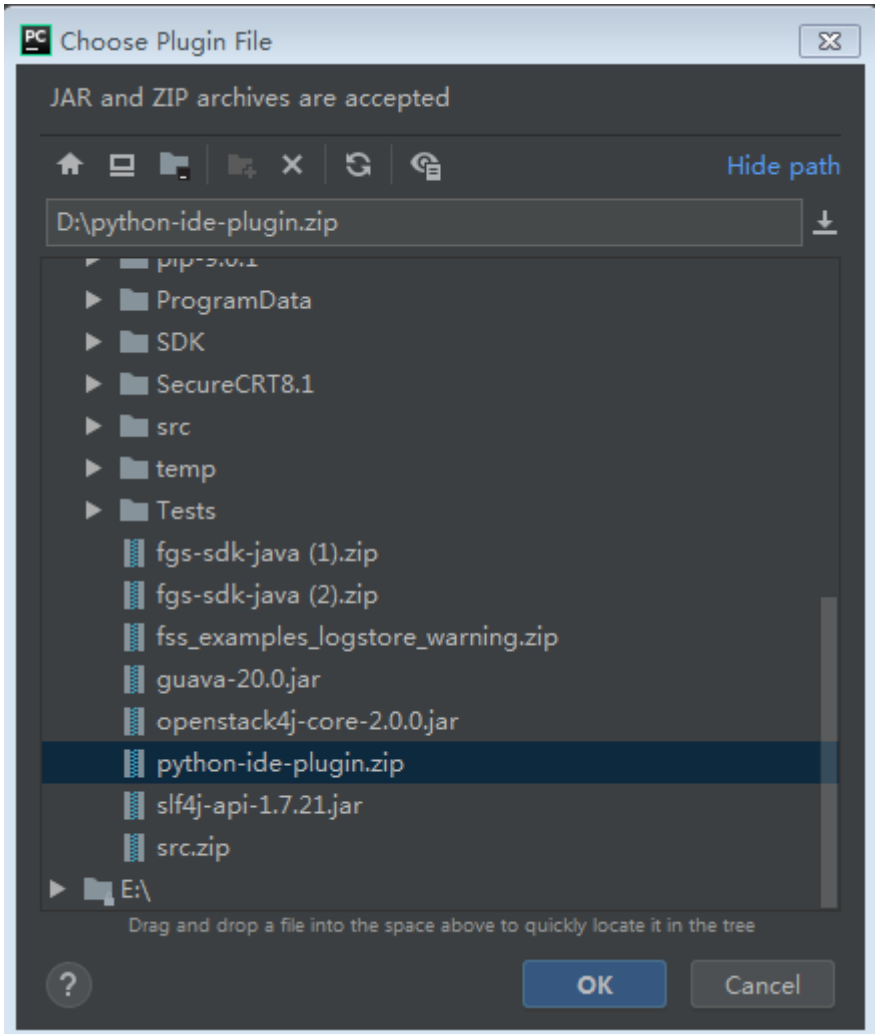
Step 2 Run JetBrains PyCharm. Choose **File** > **Settings**, choose **Plugins** in the left pane, and then click **Install Plugin from Disk** in the upper right corner, as shown in **Figure 9-11**.

Figure 9-11 Installing the plug-in



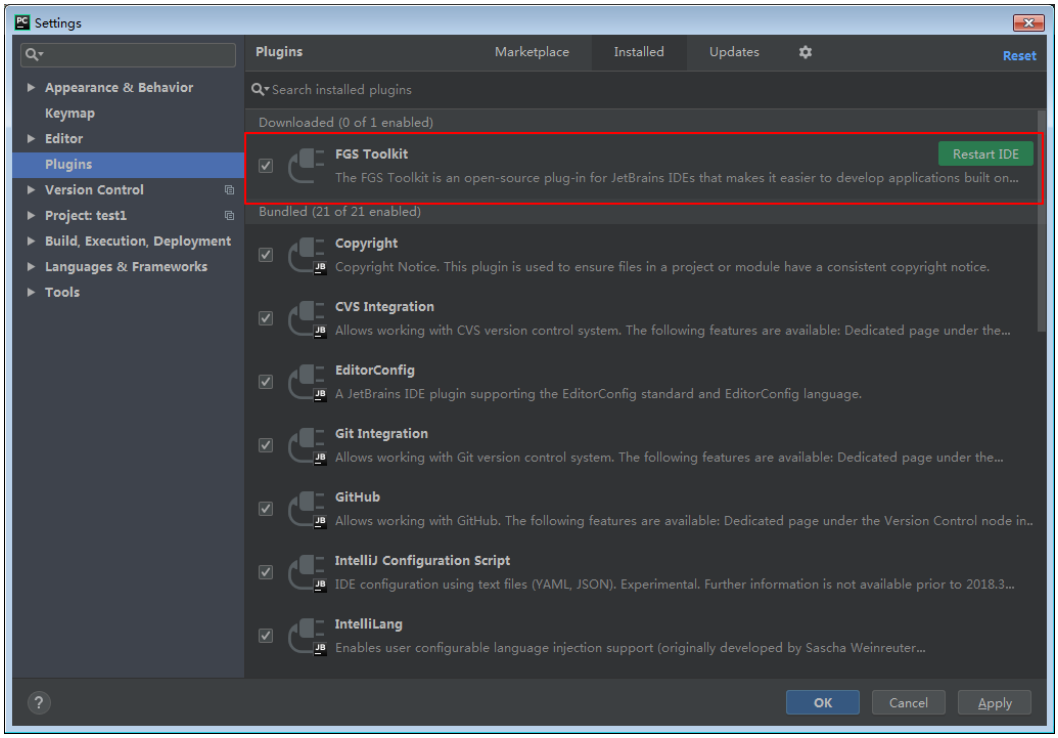
Step 3 Select the plug-in package you want to install, and click **OK**, as shown in **Figure 9-12**.

Figure 9-12 Selecting a plug-in package



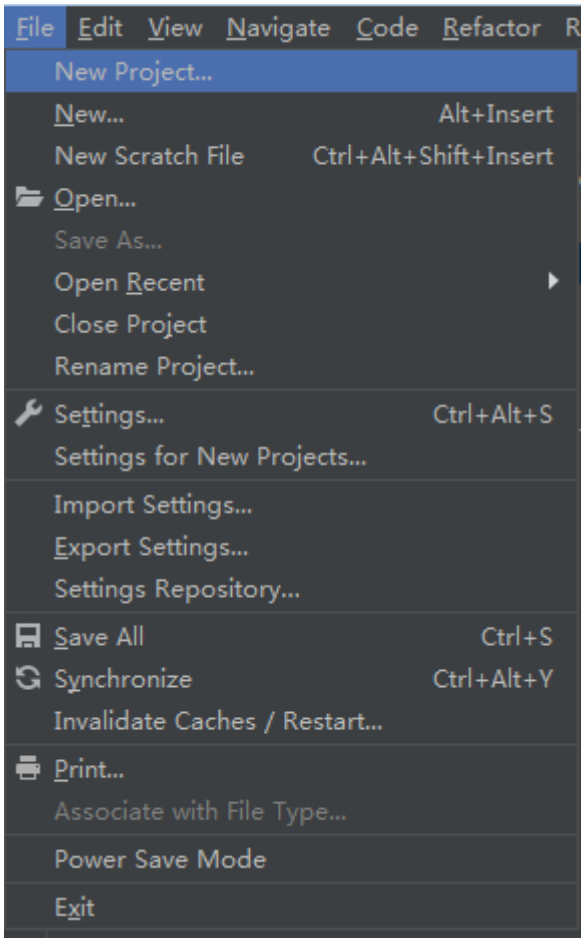
Step 4 In the plug-in list, select the desired plug-in and click **Restart IDE**, as shown in **Figure 9-13**.

Figure 9-13 Restarting the IDE



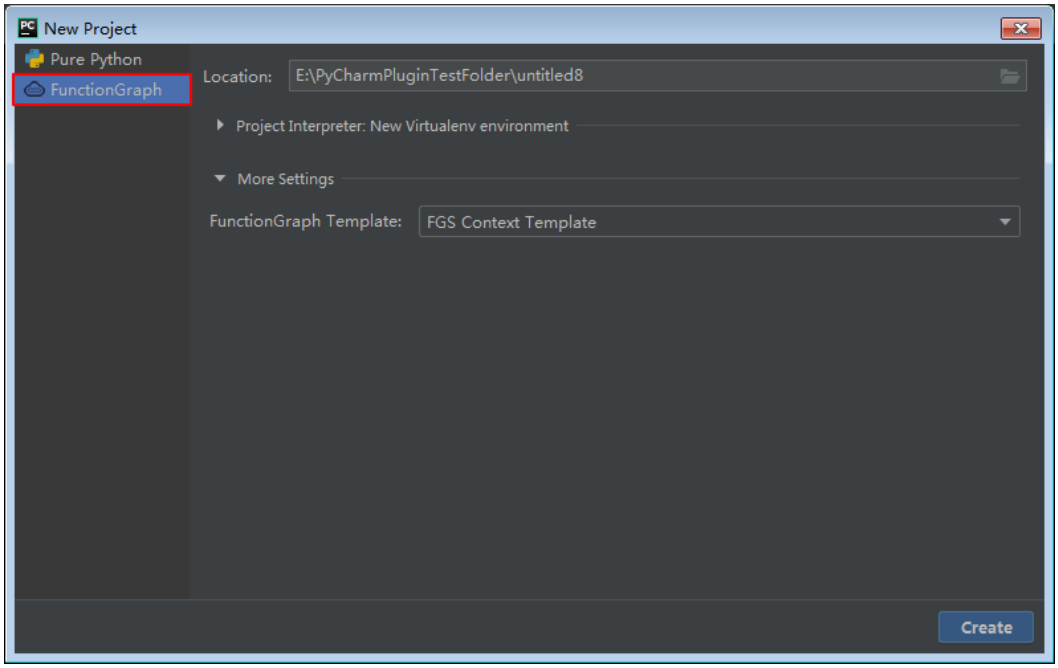
Step 5 Choose **File** > **New Project**, as shown in **Figure 9-14**.

Figure 9-14 Creating a project



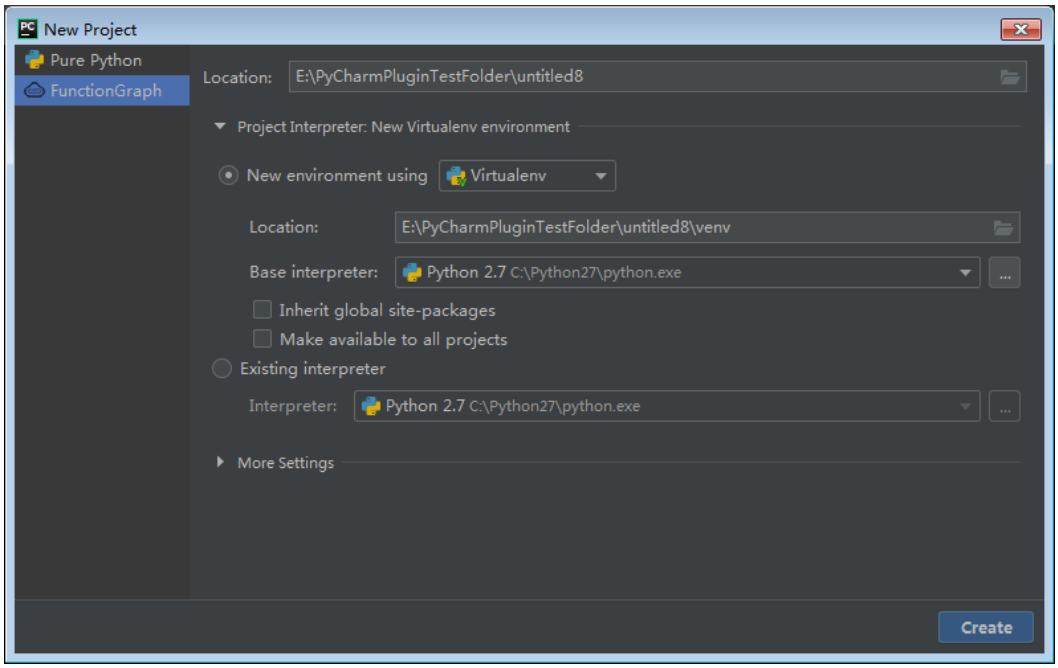
Step 6 On the displayed **New Project** page, choose **FunctionGraph**, as shown in **Figure 9-15**.

Figure 9-15 FunctionGraph



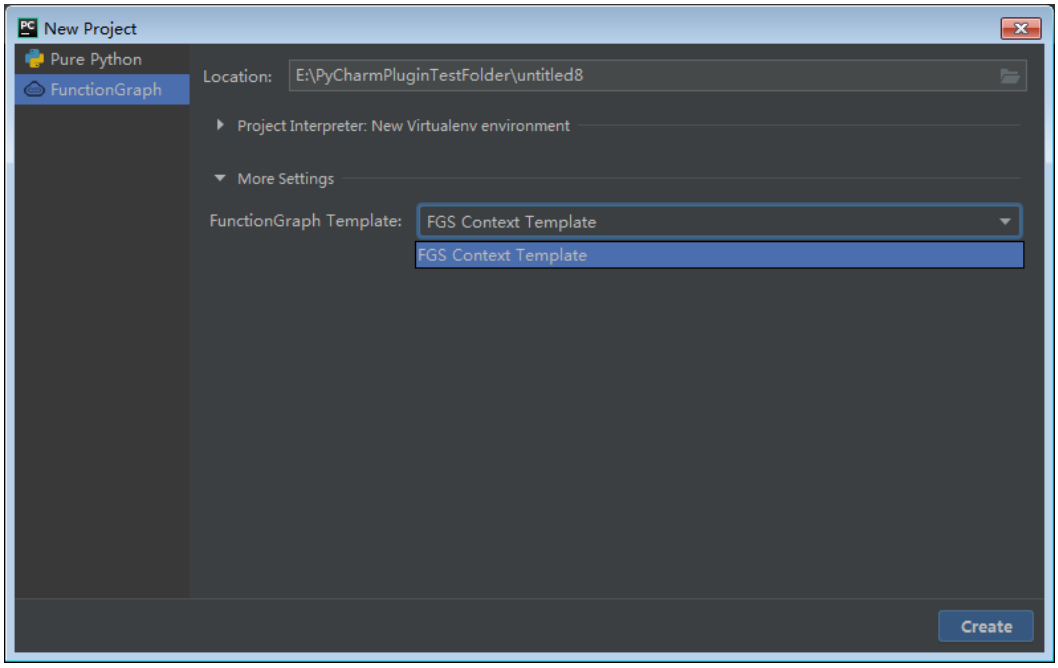
Step 7 Select the path in which the project will be stored in **Location**, and select a Python version in **Base interpreter**, as shown in **Figure 9-16**.

Figure 9-16 Selecting a version



Step 8 Select a template you want to create in the **More Settings** area, as shown in **Figure 9-17**.

Figure 9-17 Selecting a template



NOTE

Currently, only the Python 2.7 context template is supported.

Step 9 Click **Create**.

----End

9.5 Serverless Devs

9.5.1 Introduction

Component Description

Huawei Cloud FunctionGraph component is a tool used to manage the lifecycle of Huawei Cloud function applications. It is developed based on **Serverless Devs**. By configuring the resource configuration file **s.yaml**, you can easily and quickly deploy applications on **Huawei Cloud FunctionGraph**.

Prerequisites

Node.js has been installed on the local host.

Getting Started

Step 1 Run the **npm install -g @serverless-devs/s** command to install the Serverless Devs developer tool.

After the installation is complete, you need to configure the key. For details, see **Key Configuration**.

Step 2 Initialize a Hello World project by running **s init start-fg-http-Nodejs14**.

Step 3 After the initialization is complete, enter the project and run **s deploy** to deploy a function.

----End

Using Commands

The capabilities of the FunctionGraph component are listed in **Table 9-1**.

Table 9-1 Component capabilities

Build & Deploy	Publish & Configure	Other Functions
deploy	version	Project Migration fun2s
remove	alias	-

When using the FunctionGraph component, you also need to compile resource description files. For details about YAML specifications of the FunctionGraph component, see **Huawei Cloud FunctionGraph YAML Specifications**.

9.5.2 Key Configuration

Obtaining Key Information

1. Log in to Huawei Cloud and choose **My Account** > **My Credentials** in the upper right corner. The **My Credentials** page is displayed.
2. In the navigation pane on the left, choose **Access Keys**. Click **Create Access Key** to generate a new access key and download and save it.

Configuring the AK/SK

Using Wizard

Run **config add** to add a key:

```
$ 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)
```

After selecting a provider, you will be prompted to enter the corresponding information:

```
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
```

Using a Command

Add a key using a command:

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

Or:

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

9.5.3 Using Commands

9.5.3.1 deploy

deploy Commands

deploy commands are used to deploy the function resources declared in a YAML file (see **YAML File**) to the cloud.

deploy Command Parsing

You can run **deploy -h** or **deploy --help** to view the documentation.

This command contains two subcommands: **deploy function** and **deploy trigger**.

deploy function

- **deploy function** is used to deploy a function.

  You can run **deploy function -h** or **deploy function --help** to view the documentation.

  Example

  If a resource description file (YAML) is available, you can directly run the **s deploy function** command to deploy the function. The following is an example of the description file (YAML):

```
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

- **deploy trigger** is used to deploy a trigger of a function.

  You can run **deploy trigger -h** or **deploy trigger --help** to view the documentation.

  Example

  If the resource description file (YAML) is available, you can directly run the **s deploy trigger** command to deploy the trigger. The following is an example of the description file (YAML):


```
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
```

When deploying service resources, you may need to perform some interactive operations. For details, see the interaction involved during deployment in **Precautions**.

Parameter Parsing

Table 9-2 Parameter description

Parameter Name	Abbreviation	Required in YAML	Description
type	-	No	Deployment type, which can be **code** or **config**.

Examples

If the resource description file (YAML) is available, you can directly run the **s deploy** command to deploy resources. The following is an example of the resource description file (YAML):

```
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
```

Precautions

You may have some special requirements during resource deployment. See the following description:

- To only deploy or update the code, add the **--type code** parameter.
- To only deploy or update the configuration, add the **--type config** parameter.

9.5.3.2 version

version Commands

version commands are used to view aliases, publish, and delete a function version.

Command Parsing

You can run **version -h** or **version --help** to view the documentation.

This command includes two subcommands:

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

version list

version list is used to list all published versions of a service.

You can run **version list -h** or **version list --help** to view the documentation.

This command also supports some global parameters, such as **-a/--access** and **--debug**. For details, see **Global Parameters of Serverless Devs**.

Table 9-3 Parameter description

Parameter Name	Abbreviation	Required in YAML	Required in CLI	Description
region	-	No	Yes	Region
function-name	-	No	Yes	Function name
table	-	No	Yes	Output in table

Example

- If you have a resource description file (YAML), run **s version list** to list all published versions of a function.
- If you use CLI (without a YAML resource description file), specify a region and service name. For example, **s cli fgs version list --region cn-north-4 --function-name fg-test**.

NOTICE

When using CLI, if the key is not **default**, add the **access** parameter. For example, **s cli fgs version list --region cn-north-4 --function-name fg-test --access xxxx**.

Execution result of the preceding command:

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

If the **--table** parameter is used, the following output is displayed.

Figure 9-18 Output example:

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

version publish

version publish is used to publish a version.

You can run **version publish -h** or **version publish --help** to view the documentation.

This command also supports some global parameters, such as **-a/--access** and **--debug**. For details, see **Global Parameters of Serverless Devs**.

Table 9-4 Parameter description

Parameter Name	Abbreviation	Required in YAML	Required in CLI	Description
region	-	No	Yes	Region
function-name	-	No	Yes	Function name
version-name	-	No	Yes	Version
description	-	No	Yes	Description

Example

- If you have a resource description file (YAML), run **s version publish** to publish a version.
- If you use CLI (without a YAML resource description file), specify a region and service name. For example, **s cli fgs version publish --region cn-north-4 --**

function-name fg-test --version-name 1 --description "test publish version".

NOTICE

When using CLI, if the key is not **default**, add the **access** parameter. For example, **s cli fgs version publish --region cn-north-4 --function-name fg-test --version-name 1 --description "test publish version" --access xxxx**.

Execution result of the preceding command:

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

9.5.3.3 Project Migration fun2s

fun2s is used to convert the configuration of a function into the **s.yaml** format so that it can be identified by Serverless Devs.

- Command Parsing
  - Command Parsing
  - Examples

Command Parsing

You can run **fun2s -h** or **fun2s --help** to view the documentation.

This command also supports some global parameters, such as **-a/--access** and **--debug**. For details, see Global Parameters of Serverless Devs.

Table 9-5 Parameter description

Parameter Name	Abbreviation	Required in CLI	Description
region	-	Yes	Region
function-name	-	Yes	Function name
target	-	No	Path of the generated Serverless Devs configuration file (**s.yaml** by default)

Examples

In the Funcraft project, run **fun2s** to convert a function into the YAML format. For example:

```
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
```


In this way, you can convert the original function configuration into **s.yaml** so that it complies with Serverless Devs specifications.

After conversion (**s.yaml**):

```
edition: 1.0.0
name: transform_fun
access: default
vars:
  region: cn-north-4
  functionName: fgs-deploy-test
services:
  component-test: #  Service name
    component: fgs # Component name
    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 remove

remove Commands

remove commands are used to remove a deployed resource. **Resources cannot be recovered once removed.**

Command Parsing

You can run **remove -h** or **remove --help** to view the documentation.

This command includes four subcommands:

- **function: Delete a specified function**.
- **trigger: Delete a specified trigger**.
- **version: Delete a specified version**.
- **alias: Delete a specified alias**.

Table 9-6 Parameter description

Parameter Name	Abbreviation	Required in YAML	Description
assume-yes	y	No	During interaction, **y** is selected by default.

Example

If a resource description file (YAML) is available, run the **s remove** command to delete the resource. The following is an example of the resource description file (YAML):

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

remove function

remove function is used to delete a specified function. This will also delete all versions, aliases, and triggers of the function.

You can run **remove function -h** or **remove function --help** to view the documentation.

Table 9-7 Parameter description

Parameter Name	Abbreviation	Required in YAML	Required in CLI	Description
region	-	No	Yes	Region
function-name	-	No	Yes	Function name
assume-yes	y	No	Yes	During interaction, **y** is selected by default.

Example

- If you have a resource description file (YAML), run **s remove function** to delete a specified function.
- If you use CLI (without a YAML resource description file), specify the name and region of the service. For example, **s cli fgs remove function --region cn-north-4 --function-name fgs-test**.

NOTICE

When using CLI, if the key is not **default**, add the **access** parameter. For example, **s cli fgs remove function --region cn-north-4 --function-name fgs-test --access xxxx**.

Execution result of the preceding command:

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

remove trigger

remove trigger is used to delete a specified trigger.

You can run **remove trigger -h** or **remove trigger --help** to view the documentation.

This command also supports some global parameters, such as **-a/--access** and **--debug**. For details, see **Global Parameters of Serverless Devs**.

Table 9-8 Parameter description

Parameter Name	Abbreviation	Required in YAML	Required in CLI	Description
region	-	No	Yes	Region
function-name	-	No	Yes	Function name
version-name	-	No	No	Version. Default: **latest**
trigger-type	-	No	Yes	Trigger type
trigger-name	-	No	Yes	Trigger name. APIG: API name; OBS: bucket name; TIMER: trigger name
assume-yes	y	No	No	During interaction, **y** is selected by default.

Example

- If you have a resource description file (YAML), run **s remove trigger** to delete a trigger declared in the file.
- If you use CLI (without a YAML resource description file), specify the name and region of the service. For example, **s cli fgs remove trigger --region cn-north-4 --function-name fgs-test --trigger-type APIG --trigger-name fgs-test-trigger**.

Execution result of the preceding command:

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

remove version

remove version is used to delete a specified version.

You can run **remove version -h** or **remove version --help** to view the documentation.

This command also supports some global parameters, such as **-a/--access** and **--debug**. For details, see **Global Parameters of Serverless Devs**.

Table 9-9 Parameter description

Parameter Name	Abbreviation	Required in YAML	Required in CLI	Description
region	-	No	Yes	Region

Parameter Name	Abbreviation	Required in YAML	Required in CLI	Description
function-name	-	No	Yes	Service name
version-name	-	Yes	Yes	Version name, which cannot be **latest**.

Example

- If you have a resource description file (YAML), run **s remove version --version-name versionName** to delete a specified versionName.
- If you use CLI (without a YAML resource description file), specify the region and name of the service. For example, **s cli fgs remove version --region cn-north-4 --function-name fgs-test --version-name v1**.

Execution result of the preceding command:

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

remove alias

remove alias is used to delete a specified alias.

You can run **remove alias -h** or **remove alias --help** to view the documentation.

This command also supports some global parameters, such as **-a/--access** and **--debug**. For details, see **Global Parameters of Serverless Devs**.

Table 9-10 Parameter description

Parameter Name	Abbreviation	Required in YAML	Required in CLI	Description
region	-	No	Yes	Region
function-name	-	No	Yes	Service name
alias-name	-	Yes	Yes	Alias

Example

- If you have a resource description file (YAML), run **s remove alias --alias-name aliasName** to delete a specified alias.
- If you use CLI (without a YAML resource description file), specify the region and name of the service. For example, **s cli fgs remove alias --region cn-north-4 --function-name fgs-test --alias-name pre**.

Execution result of the preceding command:


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

9.5.3.5 alias

alias commands are used to view, publish, modify, and delete a function alias.

Command Parsing

You can run **alias -h** or **alias --help** to view the documentation.

This command includes four subcommands:

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

alias get

alias get is used to obtain details about a specified alias of a service.

You can run **alias get -h** or **alias get --help** to view the documentation.

This command also supports some global parameters, such as **-a/--access** and **--debug**. For details, see **Global Parameters of Serverless Devs**.

Table 9-11 Parameter description

Parameter Name	Abbreviation	Required in YAML	Required in CLI	Description
region	-	No	Yes	Region
function-name	-	No	Yes	Function name
alias-name	-	Yes	Yes	Alias

Example

- If you have a resource description file (YAML), run **s alias get --alias-name aliasName** to obtain the details about a specified alias.
- If you use CLI (without a YAML resource description file), specify the region and name of the service. For example, **s cli fgs alias get --region cn-north-4 --function-name fg-test --alias-name pre**.

NOTICE

When using CLI, if the key is not **default**, add the **access** parameter. For example, **s cli fgs alias get --region cn-north-4 --function-name fg-test --alias-name pre --access xxxx**.

Execution result of the preceding command:

```
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

alias list is used to list the aliases.

You can run **alias list -h** or **alias list --help** to view the documentation.

This command also supports some global parameters, such as **-a/--access** and **--debug**. For details, see **Global Parameters of Serverless Devs**.

Table 9-12 Parameter description

Parameter Name	Abbreviation	Required in YAML	Required in CLI	Description
region	-	No	Yes	Region
function-name	-	No	Yes	Function name
table	-	No	No	Output in table

Example

- If you have a resource description file (YAML), run **s alias list** to list the aliases.
- If you use CLI (without a YAML resource description file), specify the region and name of the service. For example, **s cli fgs alias list --region cn-north-4 --function-name fg-test**.

NOTICE

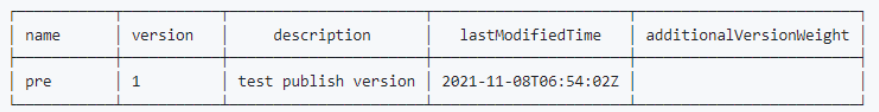
When using CLI, if the key is not **default**, add the **access** parameter. For example, **s cli fgs alias list --region cn-north-4 --function-name fg-test --access xxxx**.

Execution result of the preceding command:

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

If the **--table** parameter is specified, as shown in **Figure 9-19**:

Figure 9-19 Output example:



name	version	description	lastModifiedTime	additionalVersionWeight
pre	1	test publish version	2021-11-08T06:54:02Z	

alias publish

alias publish is used to publish and update an alias.

You can run **alias publish -h** or **alias publish --help** to view the documentation.

This command also supports some global parameters, such as **-a/--access** and **--debug**. For details, see **Global Parameters of Serverless Devs**.

Table 9-13 Parameter description

Parameter Name	Abbreviation	Required in YAML	Required in CLI	Description
region	-	No	Yes	Region
function-name	-	No	Yes	Function name
alias-name	-	Yes	Yes	Alias
version-name	-	No	Yes	Version corresponding to the alias
description	-	No	No	Alias description
gversion	-	No	No	ID of an additional version for dark launch. This is required only when a weight is specified.
weight	-	No	No	Weight of the dark launch version. This parameter is required when the dark launch version ID is specified.

Example

- If you have a resource description file (YAML), run **s alias publish** to publish or update an alias.
- If you use CLI (without a YAML resource description file), specify the region and name of the service. For example, **s cli fgs alias publish --region cn-north-4 --function-name fg-test --alias-name pre --version-name 1**.

NOTICE

When using CLI, if the key is not **default**, add the **access** parameter. For example, **s cli fgs alias publish --region cn-north-4 --function-name fg-test --alias-name pre --version-name 1 --access xxxx**.

Execution result of the preceding command:

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

To upgrade an alias, specify the alias and update the desired parameters. For example, to add a description for the preceding **pre** alias, specify the **--description** parameter and run the previous command again. The output is as follows:

```
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

For details, see **remove alias**.

9.5.3.6 YAML File

Complete YAML Configuration

The YAML fields of Huawei Cloud FunctionGraph component are as follows:

```
edition: 1.0.0 #  YAML specifications version for the command line, which complies with the Semantic Versioning specifications.
name: fg-test #  Project name
access: "default" #  Key alias

vars: # Global variables
   region: "cn-east-3"
   functionName: "start-fg-event-Nodejs14"

services:
   component-test: #  Service name
      component: fgs # Component name
      props:
         region: ${vars.region}
         function:
            functionName: ${vars.functionName} # Function name
            handler: index.handler # Handler
            memorySize: 256 # Memory required for the function
            timeout: 30 # Timeout for executing the function
            runtime: Node.js14.18 # Runtime
            agencyName: fgs-vpc-test # Agency name
            environmentVariables: # Environment variables
               test: test
               hello: world
            vpcId: xxx-xxx # Unique ID of a Virtual Private Cloud (VPC)
            subnetId: xxx-xxx # Subnet ID
            concurrency: 10 # Maximum number of instances for the function
```



```
    concurrentNum: 10 # Maximum number of concurrent requests per instance
    codeType: zip # Function code type
    dependVersionList: # Dependency ID
      - xxx-xxx
    code: # Local code address
      codeUri: ./code
  trigger:
    triggerTypeCode: TIMER # Trigger type
    status: DISABLED # Trigger status
    eventData: # Trigger configuration
      name: APIG_test  # API name
      groupName: APIGroup_xxx # Group name
      auth: IAM # Security authentication
      protocol: HTTPS # Request protocol
      timeout: 5000 # Backend timeout duration
```

Table 9-14 Parameter description

Parameter	Required	Type	Description
region	True	Enum	Region
function	True	**Struct**	Function
triggers	False	**Struct**	Triggers

Description of function Fields

The **function** fields in the YAML file are described in **Table 9-15**.

Table 9-15 Description of function fields

Parameter Name	Required	Type	Description
functionName	True	String	Function name.
handler	True	String	Handler of the function in the format of "xx.xx". It must contain a period (.).
runtime	True	String	Function runtime.
package	False	String	Package (or group) to which the function belongs. The default value is **default**.
memorySize	True	Number	Memory size (MB) allocated to the function. The options include 128, 256, 512, 768, 1024, 1280, 1536, 1792, 2048, 2560, 3072, 3584, and 4096.
timeout	True	Number	Maximum duration the function can be executed. Value range: 3s–900s.

Parameter Name	Required	Type	Description
Code Type	True	String	Function code type. Options: • **inline**: inline code • **zip**: ZIP file • **obs**: function code stored in an OBS bucket • **jar**: JAR file, mainly for Java functions
codeUrl	False	String	If **CodeType** is set to **obs**, enter the OBS URL of the function code package. If **CodeType** is not set to **obs**, leave this parameter blank.
environmentVariables	False	**Struct**	Environment variable. A maximum of 20 environment variables can be defined. The total length cannot exceed 4 KB.
agencyName	False	String	Name of an agency created in IAM. This is required when the function needs to access other services.
vpcId	False	String	Unique ID of a VPC. **agencyName** is required if you set this parameter. Log in to the **VPC Console** to view the VPC ID.
subnetId	False	String	Subnet ID. **agencyName** is required if you set this parameter. To obtain the subnet ID, log in to the **VPC Subnet page**.
dependVersionList	False	List<String>	Dependency package IDs.
code	False	**Struct**	Local code address, which is required if **CodeType** is set to **zip**.
concurrency	False	Number	Maximum number of instances in which a function can run. Range: –1 to 1,000. **–1**: The function can run in any number of instances. **0**: The function is disabled.
concurrentNum	False	Number	Maximum number of concurrent requests per instance. Range: –1 to 1,000.
description	False	String	Brief description of **function**.

- Func Code parameters

Table 9-16 Func Code parameters

Parameter Name	Required	Type	Description
codeUri	True	String	Local code address

- Environment variables

  Object format. For example:

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

  Do not write sensitive information to **s.yaml** in plaintext.

Example

```
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
```

Description of triggers Fields

The **triggers** fields in the YAML file are described in **Table 9-17**.

Table 9-17 trigger parameter description

Parameter Name	Required	Type	Description
triggerTypeCode	True	String	Trigger type
status	False	Enum	Trigger status. Options: **ACTIVE** (default) and **DISABLED**
eventData	True	Struct	Trigger configurations, including **APIG** and **timer** triggers

- APIG trigger

  Table 9-18 APIG trigger parameters

Parameter Name	Required	Type	Description
name	False	String	API name. The function name is used by default.

Parameter Name	Required	Type	Description
groupName	False	String	Group. The first one is selected by default.
auth	False	Enum	Authentication mode. Default: **IAM**. API authentication mode. Options: • **App**: AppKey and AppSecret high security authentication. This authentication mode is recommended. For details, see **App Authentication**. • **IAM**: IAM authentication. Only IAM users are allowed to access the system. The security level is medium. For details, see **IAM Authentication**. • **None**: No authentication. This mode grants access permissions to all users.
protocol	False	Enum	Request protocol. Default: **HTTPS**. Options: • HTTP • HTTPS
timeout	False	Number	Backend timeout in milliseconds. Range: 1–60,000. Default: **5000**.

Example:

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

- Timer trigger

Table 9-19 Timer trigger parameter description

Parameter Name	Required	Type	Description
name	False	String	Timer name.
scheduleType	True	Enum	Trigger rule. The value can be **Rate** or **Cron**.

Parameter Name	Required	Type	Description
schedule	True	String	Timer rule content.
userEvent	False	String	Additional information, which will be put into the **user_event** field of the timer event source.

Example:

```
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 Huawei Cloud FunctionGraph YAML Specifications

Field Parsing

Table 9-20 Parameter description

Parameter Name	Required	Type	Description
region	True	Enum	Enum
function	True	Struct	Function
trigger	False	Struct	Triggers

Complete YAML Configuration

The YAML fields of Huawei Cloud FunctionGraph component are as follows:

```
edition: 1.0.0 #  YAML specifications version for the command line, which complies with the Semantic Versioning specifications.
name: fg-test # Project name
access: "default" # Key alias

vars: # Global variable
    region: "cn-east-3"
    functionName: "start-fg-event-Nodejs14"

services:
```



```
component-test: # Service name
component: fgs # Component name
    props:
        region: ${vars.region}
        function:
            functionName: ${vars.functionName} # Function name
            handler: index.handler # Handler of the function
            memorySize: 256 # Memory required for the function
            timeout: 30 # Timeout for executing the function
            runtime: Node.js14.18 # Runtime
            agencyName: fgs-vpc-test # Agency name
            environmentVariables: # Environment variables
                test: test
                hello: world
            vpcId: xxx-xxx # Unique ID of a Virtual Private Cloud (VPC)
            subnetId: xxx-xxx #  Subnet ID
            concurrency: 10 # Maximum number of instances of a single function
            concurrentNum: 10 # Maximum number of concurrent tasks of a single instance
            codeType: zip # Code type of the function
            dependVersionList: # Dependency ID
                - xxx-xxx
            code: # Local code address
                codeUri: ./code
        trigger:
            triggerTypeCode: TIMER # Trigger type
            status: DISABLED # Trigger status
            eventData: # Trigger configuration
                name: APIG_test  # API name
                groupName: APIGroup_xxx # Group name
                auth: IAM # Security authentication
                protocol: HTTPS # Request protocol
                timeout: 5000 # Backend timeout duration
```

9.5.5 Global Parameters of Serverless Devs

Table 9-21 Global parameters of Serverless Devs

Parameter Name	Abbreviation	Default Value	Description	Remarks
template	t	**s.yaml** or **s.yml**	Specify a resource description file.	-
access	a	**access** information specified in the YAML file or **default**	Specify the key information for this deployment.	You can use the key information configured by running the **config** command and the key information configured to environment variables.

Parameter Name	Abbreviation	Default Value	Description	Remarks
skip-actions	-	-	Skip the **actions** module set in the YAML file.	-
debug	-	-	Enable the Debug Mode	After the Debug Mode is enabled, you can view more information about the tool execution process.
output	o	**default**	Specify the data output format.	The **default**, JSON, YAML, and RAW formats are supported.
version	v	-	View version information	-
help	h	-	View help information	-

9.6 Serverless Framework

9.6.1 Usage Guide

Welcome to Huawei Cloud FunctionGraph serverless usage guide.

NOTE

Before using the CLI, you need to provide **Huawei Cloud Credentials**.

9.6.1.1 Introduction

The Serverless Framework helps you develop and deploy serverless applications using FunctionGraph. It is a CLI that offers structure, automation, and best practices out-of-the-box, allowing you to focus on building sophisticated, event-driven, serverless architectures, comprised of **Function** and **Events**.

The Serverless Framework is different from other application frameworks because:

- It manages your code as well as your infrastructure.
- It supports multiple languages (Node.js, Python, Java, and more).

Core Concepts

Here are the main concepts of the Framework and how they pertain to FunctionGraph.

Function

Function refers to a **Huawei Cloud FunctionGraph function**. It is an independent unit of deployment, like a microservice. It is merely code deployed in the cloud and mostly written to perform a single job such as:

- Saving a user to the database
- Processing files in the database

You can perform multiple tasks in your code, but we do not recommend doing so without good reason. Separation of concerns is best and the Framework is designed to help you easily develop and deploy functions, as well as manage them.

Events

Anything that triggers a Huawei Cloud FunctionGraph function to execute is regarded by the Framework as an **Event**. Events are platform events on Huawei Cloud FunctionGraph including API Gateway (APIG) service and API (for example, REST API), OBS bucket (for example, image uploaded into a bucket).

When you define an event for your FunctionGraph in the Serverless Framework, the Framework will automatically translate the event with its function into the corresponding cloud resources. This way the event is configured so that your functions can listen to it.

Service

Service is the Framework's unit of organization. You can consider it as a project file, though you can have multiple services for a single application. It is where you define your functions, the events that trigger them, and the resources your functions use, all in one file entitled **serverless.yml** (or **serverless.json**).

```
# 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
```

When you deploy with the Framework by running **serverless deploy**, everything in **serverless.yml** is deployed at once.

Plug-in

You can overwrite or extend the functionality of the Framework using **Plug-ins**. Every **serverless.yml** can contain a **plugins:** attribute, which features multiple plug-ins.

```
# serverless.yml

plugins:
  - serverless-huawei-functions
```

9.6.1.2 Quick Start

This guide is designed to help you get started as quickly as possible.

Initial Setup

There are a few prerequisites you need to install and configure:

- Install Node.js 14.x or later version on your local machine. For details, see **Installing Node.js and NPM**.
- Install the Serverless Framework open-source CLI 3.28.1 or later version. For details, see **Installing the Open-source CLI of the Serverless Framework.**.

If you already have these prerequisites set up, you can skip ahead to deploy an example service.

Installing Node.js and NPM

Step 1 Install Node.js and NPM. For details about the download address, see the **download description**.

Step 2 At the end, you can run **Node -v** from your command line and you will get a result like this:

```
$ Node -v
vx.x.x
```

You can also run **npm -v** from your command line and you will get a result like this:

```
$ npm -v
x.x.x
```

----End

Installing the Open-source CLI of the Serverless Framework.

Step 1 Run this command in your terminal:

```
npm install -g serverless
```

Step 2 After the installation is complete, you can run **serverless -v** from your command line and you will get a result like this:

```
$ serverless -v
x.x.x
```

----End

Creating and Deploying a Serverless Service

The setup is completed, now you can create and deploy a serverless service.

Step 1 Create a new service.

1. Create a new service with the **huawei-Nodejs** template.
```
serverless create --template-url https://github.com/zy-linn/examples/tree/v3/legacy/huawei-Nodejs --path my-service
```
2. Install the dependencies.
```
cd my-service
npm install
```

Step 2 Set up the credentials. For details, see **credential settings**.

Step 3 Update the **serverless.yml**.

Update the **region** and **credentials** in **serverless.yml**.

Step 4 Deploy.

Use this command to deploy a service for the first time or to deploy all changes in the service after modifying the functions, events, or resources in the **serverless.yml** file. For details about this command, see **Deploy**.
```
serverless deploy
```

----End

9.6.1.3 Installation

Serverless is a Node.js CLI tool, so the first thing you need is to install Node.js on your machine.

Go to the **official Node.js website**, download and follow the **download description** to install Node.js on your local machine.

You can verify whether the Node.js is installed successfully by running **Node --version** in your terminal. If installed, you can see the corresponding Node.js version number printed out.

Installing the Serverless Framework

Step 1 Next, install the Serverless Framework via **npm** which was already installed when you installed Node.js.

Step 2 Open a terminal and type **npm install -g serverless** to install Serverless.
```
npm install -g serverless
```

Step 3 Once the installation is done, you can verify whether Serverless is installed successfully by running the following command in your terminal:
```
serverless
```

To see which Serverless version of, run:
```
serverless --version
```

----End

Installing the FunctionGraph Plug-ins

To install the latest package from npm, run:


```
npm i --save serverless-huawei-functions
```

Setting up FunctionGraph

To run Serverless commands that issue requests to Huawei Cloud, you will need to set up your Huawei Cloud credentials For details, see **credential settings**

9.6.1.4 Credentials

The Serverless Framework needs access to your Huawei Cloud credentials so that it can create and manage resources on your behalf.

Creating a Huawei Cloud Account

Open **Huawei Cloud official website**, and then choose **Sign Up**. For details, see **Registering a HUAWEI ID and Enabling Huawei Cloud Services**.

Getting the Credentials

You need to create credentials so that Serverless can use them to create resources in your project.

Step 1 Go to the **Access Keys** page to get the access keys of your Huawei Cloud account.

Step 2 Create a file named **credentials** containing the credentials that you have collected.

```
access_key_id=<collected in step 1>
secret_access_key=<collected in step 1>
```

Step 3 Save the credentials file somewhere secure. We recommend creating a folder in your root folder and putting it there, like **~/.fg/credentials**. Remember the path you saved it to.

----End

Updating the provider Configuration in serverless.yml

Open your **serverless.yml** file and update the **provider** section with the path to your credentials file (the path should be absolute). The result should be like this:

```
provider:
  name: huawei
  runtime: Node.js14.18
  credentials: ~/.fg/credentials
```

9.6.1.5 Service

A service is like a project. It is where you define your FunctionGraph functions and the events that trigger them, all in a file called **serverless.yml**.

To build your first Serverless Framework project, create a service.

Organizing

For the beginners, you can use a single service to define all of the functions and events.

```
myService/
  serverless.yml  # Contains all functions and infrastructure resources
```


As your application grows, you can split it into multiple services. Most users organize their services by workflows or data models, and group the functions related to those workflows and data models together.

```
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
```

This makes sense since related functions usually use common infrastructure resources, and users want to keep those functions and resources together as a single unit of deployment for better organization and separation of concerns.

Creating

To create a service, use the **create** command. You can also input a path to create a directory and auto-name your service:

```
# 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 is an available runtime of FunctionGraph.

Check out the **Create** for all the details and options.

Contents

You will see the following files in your working directory:

- serverless.yml
- src/index.js

serverless.yml

Each service configuration is managed in the **serverless.yml** file. The main responsibilities of this file are:

- Declare a Serverless service.
- Define one or more functions in the service:
  - Define the provider the service will be deployed to (and the runtime if provided).
  - Define any custom plug-ins to be used.
  - Define events that trigger functions to execute (such as HTTP requests).
  - Allow events listed in the **events** section to automatically create the resources required for the event upon deployment.
  - Allow flexible configuration using Serverless variables.

You can see the name of the service, the provider configuration, and the first function inside the **functions** definition. Any further service configuration will be done in this file.

```
# 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

The **index.js** file contains your exported functions.

Deploying

When you deploy a service, all of the functions and events in your **serverless.yml** are translated into calls to the Huawei Cloud API to dynamically define those resources.

To deploy a service, use the **deploy** command:

```
serverless deploy
```

Check out the **deployment guide** to learn more about deployments and how they work. Or, check out the **deploy command docs** for all the details and options.

Removing

To easily remove your service on Huawei Cloud, you can use the **remove** command.

Run **serverless remove** to trigger the removal process.

You will be notified of the process in the console when the removal starts. A success message is printed once the whole service is removed.

Only the service on your provider's infrastructure is removed during the removal process. The service directory will still remain on your local machine so you can still modify and (re)deploy it to another stage, region, or provider later on.

Version Pinning

The Serverless Framework is usually installed globally via **npm install -g serverless**. Therefore, the Serverless CLI is available for all your services.

Installing tools globally has the downside that the version cannot be pinned inside the **package.json**. This can lead to issues if you upgrade Serverless, but your colleagues or CI system do not. You can use a feature in your **serverless.yml** without worrying that your CI system will deploy with an old version of Serverless.

- **Pinned version**

  To configure version pinning, define a **frameworkVersion** attribute in your **serverless.yml**. Whenever you run a Serverless command from the CLI, it checks whether your current Serverless version is in the **frameworkVersion** range. The CLI uses **Semantic Versioning** so you can pin it to an exact version or provide a range. In general, we recommend pinning to an exact version to ensure everybody in your team has the exact same setup and no unexpected problems occur.

  Examples

Explicit version

```
# serverless.yml

frameworkVersion: '2.1.0'
```

Version range

```
# serverless.yml

frameworkVersion: ^2.1.0 # >=2.1.0 && <3.0.0
```

Installing Serverless in an Existing Service

If you already have a Serverless service, and prefer to lock down the framework version using **package.json**, then you can install Serverless as follows:

```
# from within a service
npm install serverless --save-dev
```

- **Invoke serverless locally**

  To execute the locally installed Serverless, you have to reference the binary out of the **Node_ modules** directory. An example is as follows:

  ```
  Node ./Node_modules/serverless/bin/serverless deploy
  ```

9.6.1.6 Functions

If you are using Huawei Cloud FunctionGraph as a provider, all functions inside the service are Huawei Cloud FunctionGraph functions.

Configuration

All of the Huawei Cloud FunctionGraph in your Serverless service can be found in **serverless.yml** under the **functions** attribute.

```
# serverless.yml
service: fg-service

provider:
  name: huawei

plugins:
  - serverless-huawei-functions

functions:
  first:
    handler: index.handler
```

Handler

The **handler** attribute should be the function name you have exported in your handler file.

For example, when you export a function with the name **handler** in **index.js**, your **handler** should be **handler: index.handler**.

```
// index.js
exports.handler = (event, context, callback) => {};
```

Memory Size and Timeout

The **memorySize** and **timeout** for the functions can be specified at the provider or function level. The provider definition enables all functions to share this

configuration, whereas the function definition indicates that this configuration is only valid for the function.

The default **memorySize** is 256 MB and the default **timeout** is 30s if not specified.

```
# serverless.yml

provider:
  memorySize: 512
  timeout: 90

functions:
  first:
    handler: first
  second:
    handler: second
    memorySize: 256
    timeout: 60
```

Handler Signature

The signature of an event handler is:

```
function (event, context) { }
```

- **event**

  If the function is triggered by an APIG event specified, the **event** passed to the handler will be:

  ```
  // 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**

  The **context** parameter contains the runtime information about the function. For example, request ID, temporary AK, and function metadata. See **Developing an Event Function**.

9.6.1.7 Events

Simply put, events are the things that trigger your functions to run.

If you are using Huawei Cloud as your provider, events in the service are limited to the Huawei Cloud APIG and OBS. For details, see **Event list**.

Upon deployment, the Framework will set up the corresponding event configuration your function should listen to.

Configuration

Events belong to each function and can be found in the **events** attribute in **serverless.yml**.

```
# 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
```

NOTE

Currently, only one event definition per function is supported.

Types

The Serverless Framework supports OBS and APIG events of the FunctionGraph. For details, see **Event list**.

Deployment

To deploy or update your functions and events, run:

```
serverless deploy
```

9.6.1.8 Deploy

The Serverless Framework is designed to provision functions, events and resources of FunctionGraph safely and quickly.

Deploying All

This is the main method for deploying with the Serverless Framework:

```
serverless deploy
```

Use this method when you have updated your function, event, or resource configuration in **serverless.yml** and you want to deploy that change (or multiple changes at the same time) to Huawei Cloud.

Working Principles

The Serverless Framework translates all syntax in **serverless.yml** to a configuration template.

1. The provider plugin parses **serverless.yml** configuration and translates it to Huawei Cloud resources.
2. The code of your functions is then packaged into a directory, zipped, and uploaded to the deployment bucket.
3. Resources are deployed.

   NOTE

   Use this in your CI/CD system, as it is the safest method for deployment.

Check out the **deploy command docs** for all the details and options.

9.6.1.9 Package

Packaging CLI Command

Using the Serverless CLI tool, you can package your project without deploying it to Huawei Cloud. This is best used with CI/CD workflows to ensure consistent deployable artifacts.

Running the following command will build and save all of the deployment artifacts in the **.serverless** directory of the service:

```
serverless package
```

Packaging Configuration

If you want to have more control over function artifacts and package methods.

You can use the **patterns** configuration.

- **Patterns**

  Patterns allow you to define globs that will be excluded/included from the resulting artifact. If you want to exclude files, you can use a global pattern prefixed with **!**, such as **!exclude-me/****. Serverless Framework will run the global patterns so that you can always re-include previously excluded files and directories.

  Examples

  Exclude all **Node_modules** but then re-include a specific module (**Node-fetch** in this case) using **exclude**

  ```
  package:
    patterns:
      - '!Node_modules/**'
      - 'Node_modules/Node-fetch/**'
  ```

  Exclude all files but **handler.js**

  ```
  package:
    patterns:
      - '!src/**'
      - src/function/handler.js
  ```

  NOTE

  If you want to exclude directories, use the correct global syntax. The following is an example:

  ```
  package:
    patterns:
      - '!tmp/**'
      - '!.git/**'
  ```


- Artifact

  For complete control over the packaging process, you can specify your own artifact ZIP file.

  Serverless will not zip your service if this is configured and therefore **patterns** will be ignored. Either you use **artifact** or **patterns**.

  The artifact option is especially useful if your development environment allows you to generate a deployable artifact like Maven does for Java.

  Example

```
service: my-service
package:
  patterns:
    - '!tmp/**'
    - '!.git/**'
    - some-file
  artifact: path/to/my-artifact.zip
```

- **Package functions separately**

  If you want even more control over your functions for deployment, you can configure them to be packaged separately. This allows for more control to optimize your deployment. To enable individual packaging, set **individually** to **true** in the packaging settings of the service or the function.

  Then for every function you can use the same **patterns** or **artifact** configuration options. The **patterns** options will be merged with the service options to create a **patterns** configuration for each function during packaging.

```
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'
```

  You can also select which functions to be packaged separately, and have the rest use the service package by setting the **individually** to **true**.

```
service: my-service
functions:
  hello:
    handler: handler.hello
  world:
    handler: handler.hello
    package:
      individually: true
```

- **Development Dependencies**

  Serverless will auto-detect and exclude development dependencies based on the runtime your service is using to ensure that only the production relevant packages and modules are included in your ZIP file. This drastically reduces the overall size of the deployment package which will be uploaded to the cloud provider.

You can opt out of automatic exclusion of development dependency by setting the **excludeDevDependencies** to **false**:

```
package:
  excludeDevDependencies: false
```

9.6.1.10 Variables

The Serverless Framework provides a powerful variable system which allows you to add dynamic data into your **serverless.yml**. With Serverless variables, you will be able to do the following:

- Reference & load environment variables.
- Reference & load variables from CLI options.
- Recursively reference properties of any type from the same **serverless.yml** file.
- Recursively reference properties of any type from other **YAML** or **JSON** files.
- Recursively nest variable references for ultimate flexibility.
- Combine multiple variable references to overwrite each other.

Constraints

You can only use variables in **values** attribute instead of key attribute in **serverless.yml**. So you cannot use variables to generate dynamic logical IDs in the custom resources.

Referencing Environment Variables

To reference environment variables, use the *${env:someProperty}* syntax in your **serverless.yml**.

```
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 Reference

Welcome to the Serverless CLI reference for FunctionGraph!

NOTE

Before continuing, **Huawei Cloud credentials** are required for using the CLI.

9.6.2.1 Create

Creates a new service in the current working directory based on the specified template.

- Create a service in the current working directory:
  ```
  serverless create --template-url https://github.com/zy-linn/examples/tree/v3/legacy/huawei-Nodejs
  ```
- Create a service in a new folder using a custom template:
  ```
  serverless create --template-url https://github.com/zy-linn/examples/tree/v3/legacy/huawei-Nodejs --path my-service
  ```

Options

- **--template-url** or **-u**: A URL pointing to a remotely hosted template. **Required if --template and --template-path are not present.**
- **--template-path**: The local path of your template. **Required if --template and --template-url are not present.**
- **--path** or **-p**: The path where the service is created.
- **--name** or **-n**: The name of the service in **serverless.yml**.

Examples

- **Create a new service**
  ```
  serverless create --template-url https://github.com/zy-linn/examples/tree/v3/legacy/huawei-Nodejs --name my-special-service
  ```
  This example generates a Node.js runtime in the current working directory for the service with Huawei as the provider.
- **Create a named service in a (new) directory**
  ```
  serverless create --template-url https://github.com/zy-linn/examples/tree/v3/legacy/huawei-Nodejs --path my-new-service
  ```
  This example will generate a Node.js runtime in the **my-new-service** directory for the service with Huawei as the provider. This directory will be automatically created if it does not exist. Otherwise Serverless will use the already present directory.

  Additionally, Serverless will rename the service according to the path you provide. In this example, the service will be renamed to **my-new-service**.

9.6.2.2 Install

install Command

Install a service from a GitHub URL in the current working directory.
```
serverless install --url https://github.com/some/service
```

Options

- **--url** or **-u**: The services GitHub URL, which is required.
- **--name** or **-n**: Name for the service.

Examples

- **Install a service from a GitHub URL**
  ```
  serverless install --url https://github.com/zy-linn/examples/tree/v3/legacy/huawei-Nodejs
  ```


This example will download the **.zip** file of the **huawei-Nodejs** service from GitHub, create a new directory named **huawei-Nodejs** in the current working directory, and unzip the file in this directory.

- **Install a service from a GitHub URL with a new service name**
  ```
  serverless install --url https://github.com/zy-linn/examples/tree/v3/legacy/huawei-Nodejs--name my-huawei-service
  ```
  The execution process is as follows:
  a. Download the **.zip** file of the **huawei-Nodejs** service from GitHub.
  b. Create a new directory with the name **my-huawei-service** in the current working directory.
  c. Unzip the files in this directory.
  d. Rename the service to **my-huawei-service** if **serverless.yml** exists in the service root.
- **Install a service from a directory in a GitHub URL**
  ```
  serverless install --url https://github.com/zy-linn/examples/tree/v3/legacy/huawei-Nodejs
  ```
  In this example, the **huawei-Nodejs** service will be downloaded from GitHub.

9.6.2.3 Package

The **serverless package** command packages your entire infrastructure into the **.serverless** directory by default and makes it ready for deployment.

package Command

```
serverless package
```

In this example, your service will be packaged. The package will be generated in the default **.serverless** directory.

9.6.2.4 Deploy

deploy Command

The **serverless deploy** command deploys your entire service via the Huawei Cloud API. Run this command when you have edited the **serverless.yml** file.

```
serverless deploy
```

Artifacts

After running the **serverless deploy** command, all created deployment artifacts will be placed in the **.serverless** folder of the service.

9.6.2.5 Info

By default, the **serverless info** command is used to display information about deployed services.

info Command

Run this command in the service directory to display information about deployed services.

```
serverless info
```


9.6.2.6 Invoke

invoke Command

Invoke a deployed function. You can send event data, read logs, and view other important information about function calls.

```
serverless invoke --function functionName
```

Options

- **--function** or **-f**: The name of the function that you want to invoke. **Required**
- **--data** or **-d**: Data you want to pass into the function.
- **--path** or **-p**: The path to a JSON file which contains the input data to be transferred to the invoked function. This path is relative to the root directory of the service.

Examples

- **Simple function invocation**

  ```
  serverless invoke --function functionName
  ```

  In this example the deployed function will be invoked and the result of the invocation will be displayed in the terminal.

- **Function invocation with data**

  ```
  serverless invoke --function functionName --data '{"name": "Bob"}'
  ```

  In this example, the function will be invoked with the provided data and the result will be displayed in the terminal.

- **Function invocation with passed data**

  ```
  serverless invoke --function functionName --path lib/event.json
  ```

  In this example, the **JSON** data in the **lib/event.json** file will be passed (relative to the root of the service) while invoking the specified or deployed function.

  Example of **event.json**

  ```
  {
    "key": "value"
  }
  ```

9.6.2.7 Logs

logs Command

View the logs of a specific function.

```
serverless logs --function functionName
```

Options

- **--function** or **-f**: The function for obtaining the logs. **Required**.
- **--count** or **-c**: The number of logs to display.

Example

Retrieve logs

```
serverless logs --function functionName
```


This will display logs for the specified function.

9.6.2.8 Remove

remove Command

The **serverless remove** command will remove the deployed service defined in your current working directory from the provider.

```
serverless remove
```

NOTE

Only the deployed service and all its resources will be removed. The code on the local computer will be retained.

Example

Remove service

```
serverless remove
```

This command will remove the deployed service in your current working directory.

9.6.3 Event list

9.6.3.1 APIG Events

FunctionGraph can create function-based API endpoints through APIG.

To create HTTP endpoints as event sources for FunctionGraph, use the **HTTP** event syntax.

HTTP Endpoint

In this setting, the **first** function should be run when someone accesses the API endpoint via a **GET** request. You can get the URL for the endpoint by running the **serverless info** command after deploying your service.

Here is an example:

```
# 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
```

NOTE

For details about the handler signature used for such events, see the **Handler**.

9.6.3.2 OBS Events

Huawei Cloud functions can be triggered by different event sources, which can be defined and configured through **events**.

OBS Events

In this example, an OBS event is set. Each time an object is uploaded to **my-service-resource**, the event triggers the **first** function.

```
# 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 Automated Deployment

10.1 Preparing an Environment

This section uses a Linux host as an example to describe how to set up a function CI/CD environment using KooCLI and **CodeArts**.

ECS

This server functions as a CodeArts task deployment host to deploy and update FunctionGraph functions.

- Specifications: 1 vCPU | 1 GiB
- Image: CentOS 8.2 64 bits
- Other: Configure an EIP because you will need to install the Python library and CodeArts and configure the ECS as a deployment host.
- CodeArts uses this server as a deployment host through port 22 over SSH. For high security, add the following IP addresses to the security group of the server. Otherwise, authorization will fail.

  42.202.130.147

  49.4.3.11

  122.112.212.206

  139.159.226.153

  49.4.85.127

  124.70.46.237

Configuring a Security Group for the Deployment Host

1. Go to the **Create IP Address Group** page.
2. Set the parameters as prompted. For details about the parameters, see **Creating an IP Address Group** . Then, click **Create Now**.
   - **Name**: Enter **ipGroup-clouddeploy**.
   - **IP Address**:
     ```
     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. Return to the network console, choose **Access Control** > **Security Groups** in the navigation pane on the left, and click **Create Security Group**. For details about how to configure a security group, see **Creating a Security Group**. Then click **Create Now**.
   - **Name**: Enter **functions-deploy**.
   - **Enterprise Project**: Enter **default**.
4. Go to the details page of **functions-deploy**, click the **Inbound Rules** tab, and click **Add Rule** to add an inbound rule.

   Set **Priority** to **1**, **Protocol & Port** to **22**, and **Source** to the IP address group **ipGroup-clouddeploy**. Then click **OK**.

   Figure 10-1 Adding an inbound rule

5. Return to the ECS console, click the ECS name, and change the security group to **functions-deploy**.

Installing Python Libraries

Run the following commands to install the pyyaml and pycryptodome libraries: The two libraries will respectively parse the **cam.yaml** configuration file of your function and encrypts and decrypts the function's environment variables.

```
pip3 install pyyaml
pip3 install pycryptodome
```

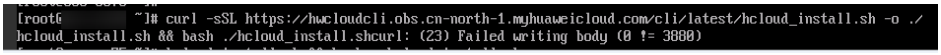
Installing KooCLI

1. Install KooCLI.

   Remotely log in to the purchased ECS, and run the following command to install KooCLI:

   ```
   curl -sSL https://ap-southeast-3-hwcloudcli.obs.ap-southeast-3.myhuaweicloud.com/cli/latest/hcloud_install.sh -o ./hcloud_install.sh && bash ./hcloud_install.sh
   ```

   Figure 10-2 Installing KooCLI

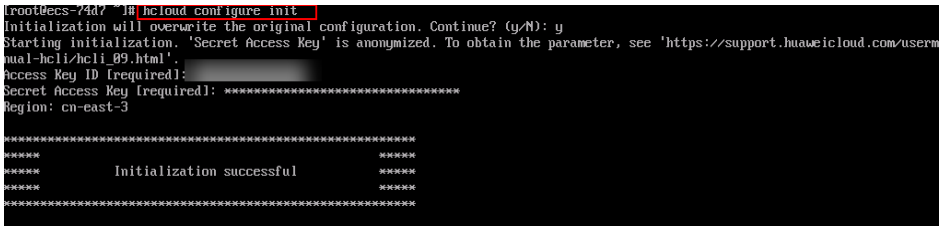
2. Initialize KooCLI.

   Run the following command to initialize KooCLI:

   ```
   hcloud configure init
   ```

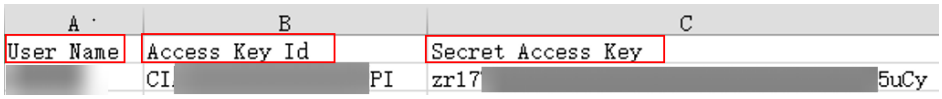
   Enter an access key ID (AK), secret access key (SK), and region name. For details about how to obtain them, see **3** and **4**.

Figure 10-3 Initializing KooCLI



3. Obtain an access key (AK/SK).
   - If you have access to the console, log in, and create an access key on the **My Credentials** page. For details, see **Creating an Access Key**. An AK/SK file is downloaded. Generally, it is named **credentials.csv**. As shown in the following figure, the file contains a username, AK, and SK.

     Figure 10-4 Content of the credentials.csv file

   - If you do not have access to the console, request the administrator to create an access key for you on the IAM console in case your access key is lost or needs to be reset. For details, see **Managing Access Keys for an IAM User**.
4. Obtain a region name.

   For details, see **Regions and Endpoints**.

Figure 10-5 Obtaining region information

Region Name	Region
AF-Johannesburg	af-south-1
AP-Bangkok	ap-southeast-2
AP-Singapore	ap-southeast-3

10.2 Hosting Function Code with CodeArts

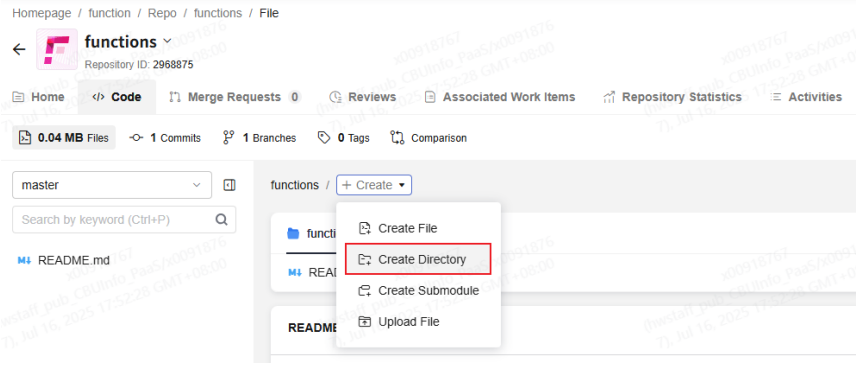
10.2.1 Step 1: Create a Project

1. Log in to the **CodeArts console** and access the **CodeArts** page. Click **Go to Workspace**.
2. On the **Project** tab page, choose **Templates** > **Scrum** and click **Select**.
3. Enter project name **function** and retain the default settings for other parameters.
4. Click **OK**.

10.2.2 Step 2: Host Function Code

1. On the function project page, choose **Code** > **Repo** in the navigation pane on the left and click **Create Repository**.
2. Set **Repository Type** to **Custom** and click **Next**.
3. Set **Repository Name** to **functions**, retain the default values for other parameters, and click **OK**.
4. Go to the **functions** repository. Create a **deploy** directory for storing the **deploy.py code** of the function.

Figure 10-6 Creating a directory

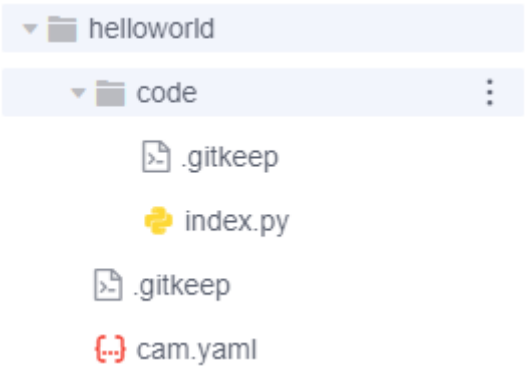


NOTE

The **deploy.py** script will read the function configuration file **cam.yaml** and construct an **hcloud** command to update the function code and configuration. For details about the configurations in **cam.yaml**, see **Analyzing cam.yaml**. Logs generated when executing this script will be recorded in the **/home/function/deploy/function.log** file.

5. Create another directory named **helloworld**, with a complete structure.

Figure 10-7 Complete function directory structure

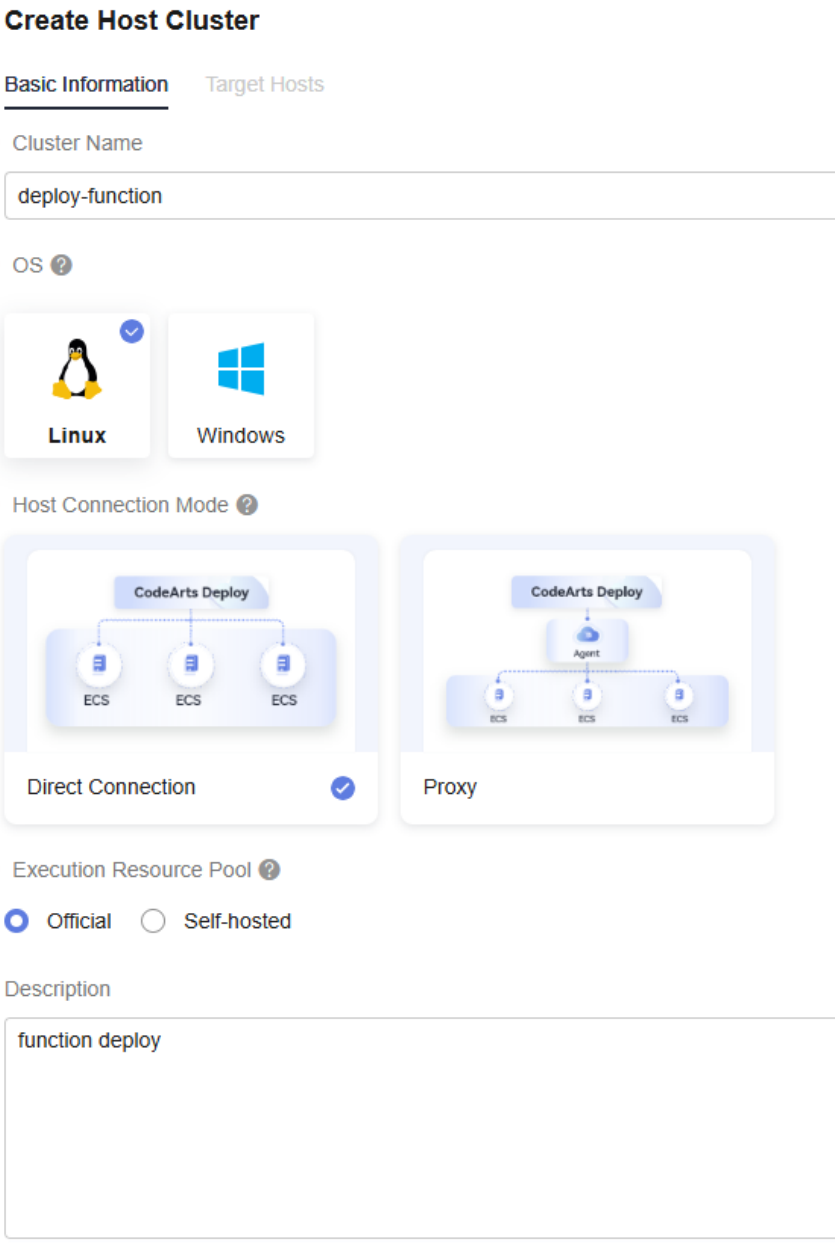


- **helloworld**: the **helloworld** function
- **cam.yaml**: the configuration file
- **code**: function code directory, which stores the **index.py** file

10.2.3 Step 3: Configure a Deployment Host

1. In the navigation pane on the left, choose **Settings** > **General** > **Basic Resources**. On the displayed page, click **Create Host Cluster**.
2. Enter **deploy-function** in the **Cluster Name** text box, set other parameters as shown in **Figure 10-8**, and click **Save**.

   Figure 10-8 Creating a host cluster



3. On the **Target Hosts** tab page, click **Add Host**.

   Select **Importing ECS**, import the ECS cloud server in **Preparing an Environment**, enter the username, password, and SSH port number 22 of the server, and click **OK**.

4. If connectivity verification is successful, the host is added.

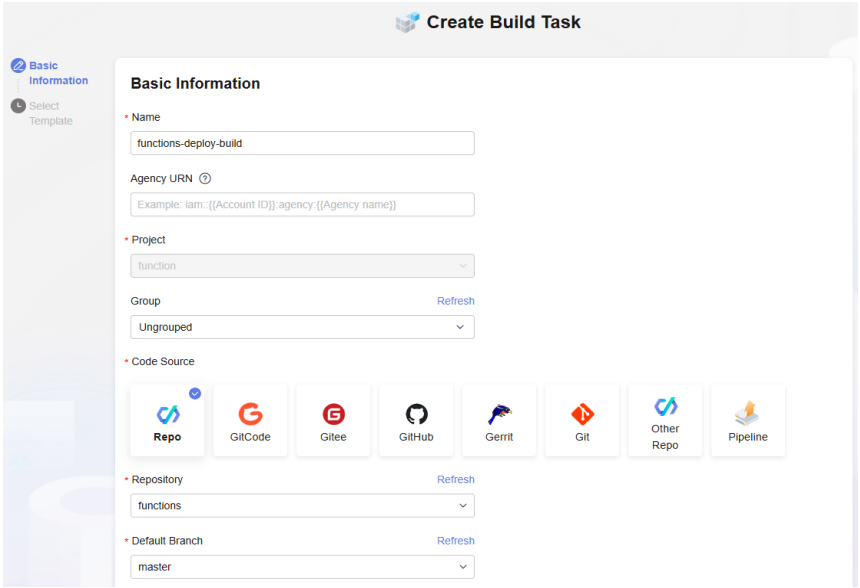
10.2.4 Step 4: Set Up a Pipeline for Updating the Function Deployment Script

The pipeline allows you to release the function deployment script **deploy.py** to the deployment host for function updates.

Creating a Build Task

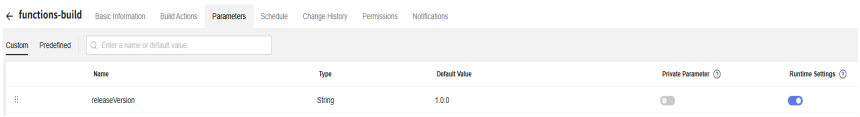
1. Log in to the **CodeArts** console, choose **CodeArts Build** in the navigation pane, and click **Go to CodeArts Build**.
2. On the **Build** page, click **Create Task**. Set basic information by referring to **Figure 10-9** and click **Next**.

Figure 10-9 Creating a build task



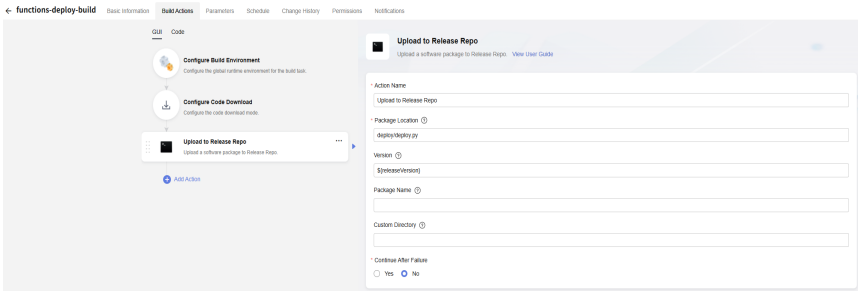
3. Select **Blank Template** and click **OK**.
4. Click the **Parameters** tab, add the **codeBranch** and **releaseVersion** parameters in**Custom** page, and enable **Runtime Settings**.

Figure 10-10 Configuring the version parameter



5. Click the **Build Actions** tab, click **Add Build Actions**, and select **Upload to Release Repo**. The **Upload to Release Repo** action is displayed in the left pane.
6. Click the **Upload to Release Repo** action, and set parameters.

Figure 10-11 Setting parameters

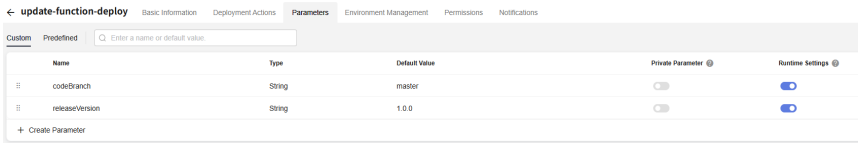


- **Action Name**: Enter **Upload to Release Repo**.
- **Package Location**: Enter **deploy/deploy.py**.
- **Version**: Enter **${releaseVersion}**.

Creating a Deployment Task

1. Return to the CodeArts console. In the navigation pane, choose **Deploy**. Click **Create Application**.
2. On the **Basic Information** tab page, set the application basic information. Set **Name** to **update-function-deploy**, retain the default values for other parameters, and click **Next**.
3. Select **Blank Template** and click **OK**.
4. On the **Deployment Actions** tab page, add **Select Deployment Source**.
5. Set **Source** to **Artifact** and configure the deployment source.
   - **Environment**: Select the host cluster **deploy-function**.
   - **Software Package**: Enter **/functions-deploy-build/${releaseVersion}/deploy.py**.
   - **Download Path**: Enter **/home/function/deploy**.
6. Click the **Parameters** tab, add the **codeBranch** and **releaseVersion** parameters in**Custom** page, and enable **Runtime Settings**.

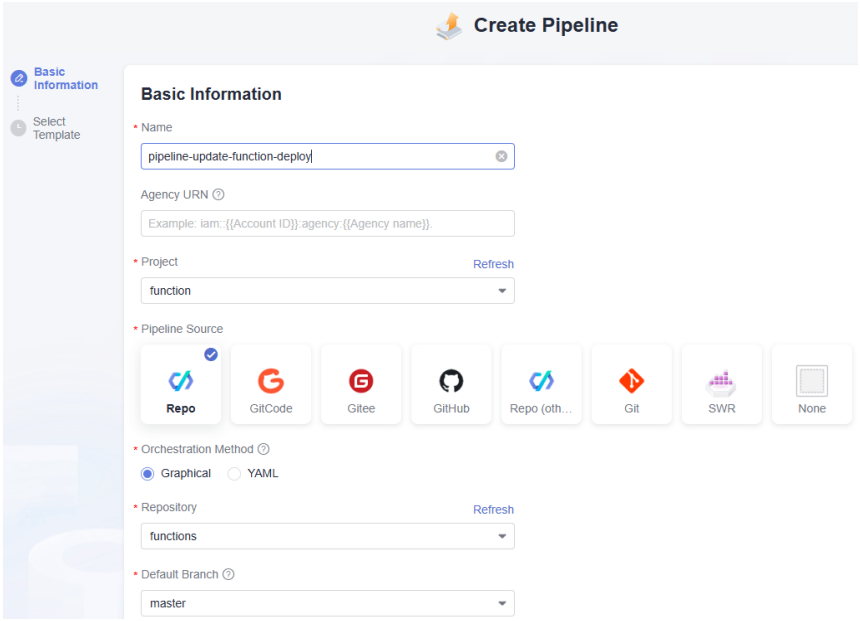
Figure 10-12 Configuring the version parameter



Configuring a Pipeline

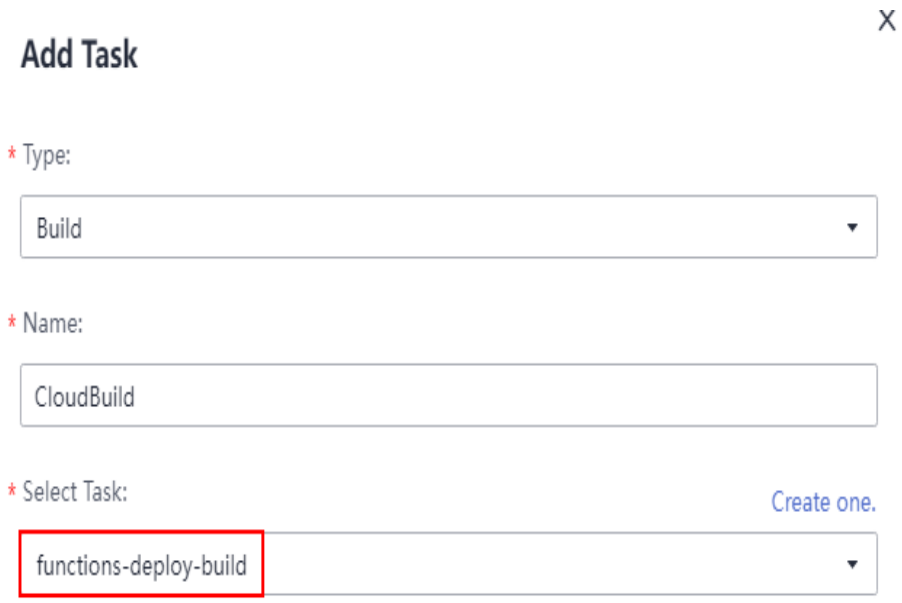
1. Return to the CodeArts console. In the navigation pane, choose **CodeArts Pipeline**. Click **Go to CodeArts Pipeline**.
2. Access the pipeline console and click **Create Pipeline**.
3. On the **Basic Information** page, set the pipeline name to **pipeline-update-function-deploy**, set other parameters by referring to **Figure 10-13**, and click **Next**.

Figure 10-13 Creating a pipeline



4. Configure **Build and Check**.
   a. Add a build task, and select the **function-deploy-build** task.

      Figure 10-14 Adding a task



   b. Set **releaseVersion** as a pipeline parameter.

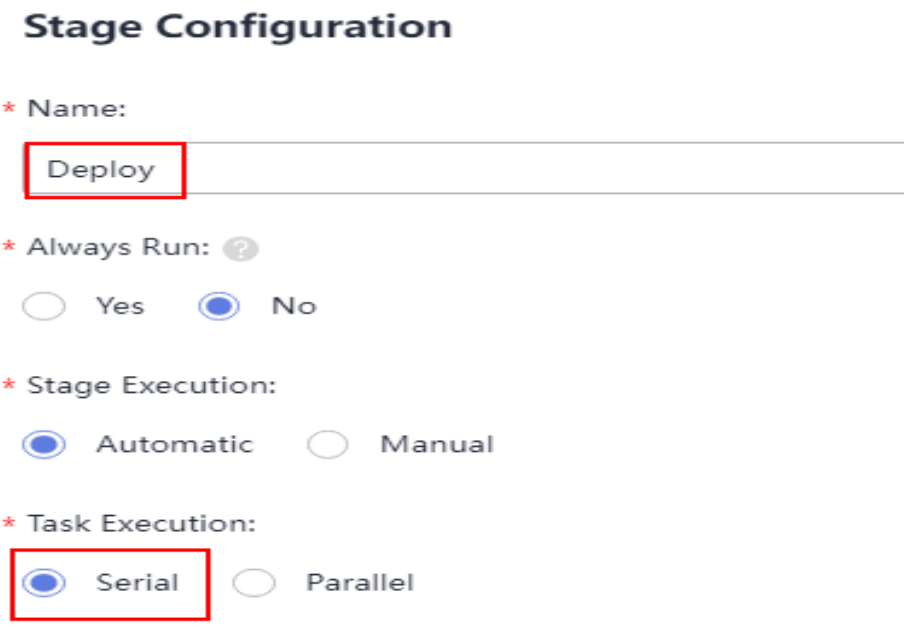
Figure 10-15 Setting releaseVersion

c. Click **Save**.

5. Configure a deployment task.

a. Add a stage named **Deploy** after **Build_and_Check**, set **Task Execution** to **Serial**, and click **Save**.

Figure 10-16 Configuring the stage



b. Click **Add Task** to add a deployment task named **DeployScript**, and select the **update-function-deploy** task.

Figure 10-17 Adding a task

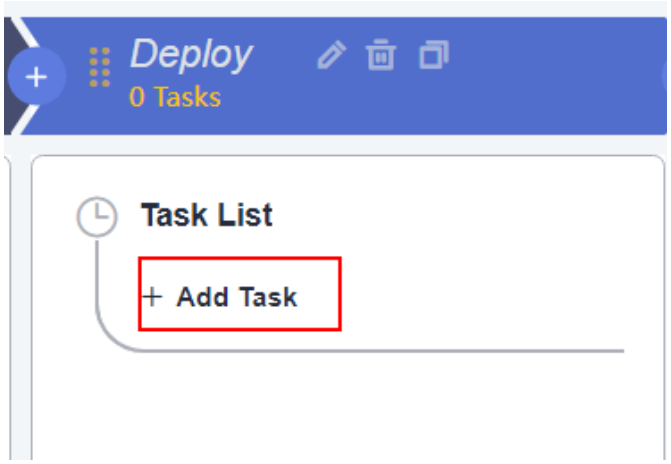
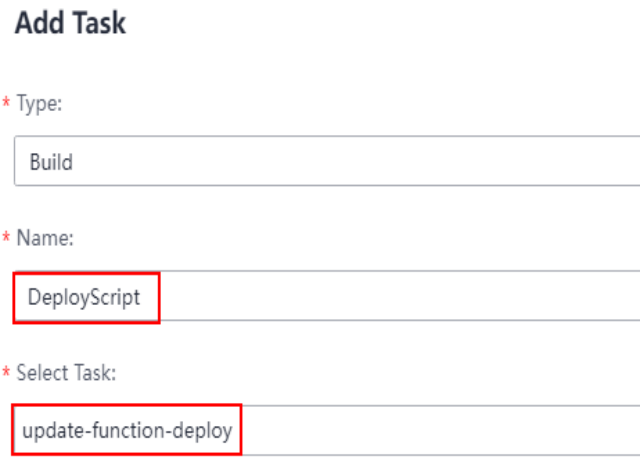
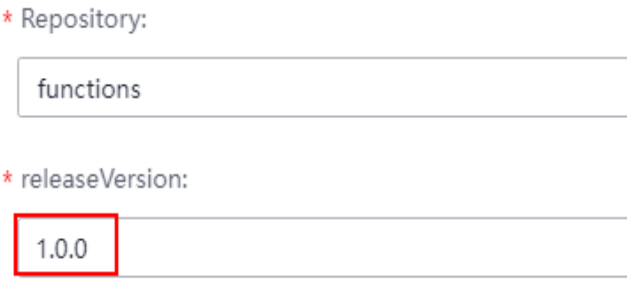


Figure 10-18 Configuring the task



Set **releaseVersion** as a pipeline parameter.

Figure 10-19 Setting releaseVersion

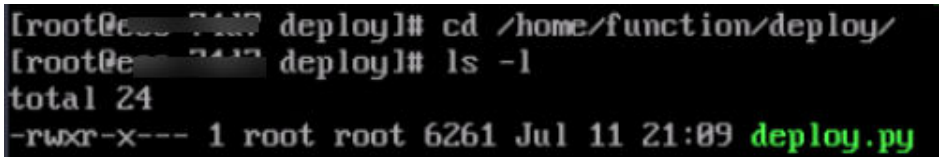


c. Click **Save**.

6. On the **Basic Information** tab page, change the pipeline task name to **pipeline-update-function-deploy** and click **Save**.
7. Execute the pipeline.

a. Set the runtime parameter **releaseVersion** to **1.0.0** and click **Execute**.
b. Wait till the **deploy.py** script is released.

Figure 10-20 Execution successful



10.2.5 Step 5: Set Up a Function Update Pipeline

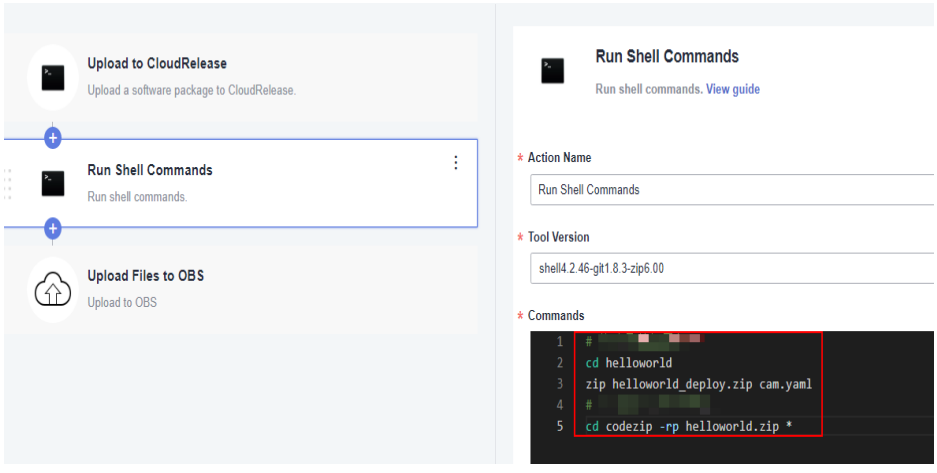
The pipeline allows you to release and update the **helloworld** function code in the **functions** repository to FunctionGraph.

Creating a Build Task

1. Choose **Build & CloudArtifact** > **CloudBuild**, and click **Create Task**.
2. Select **functions** for **Source Code Repository** and **Blank Template** for the template.
3. Add these three actions: **Run Shell Commands**, **Upload Files to OBS**, and **Upload Deployment Package to CloudRelease**.
   a. Configure the **Run Shell Commands** action.

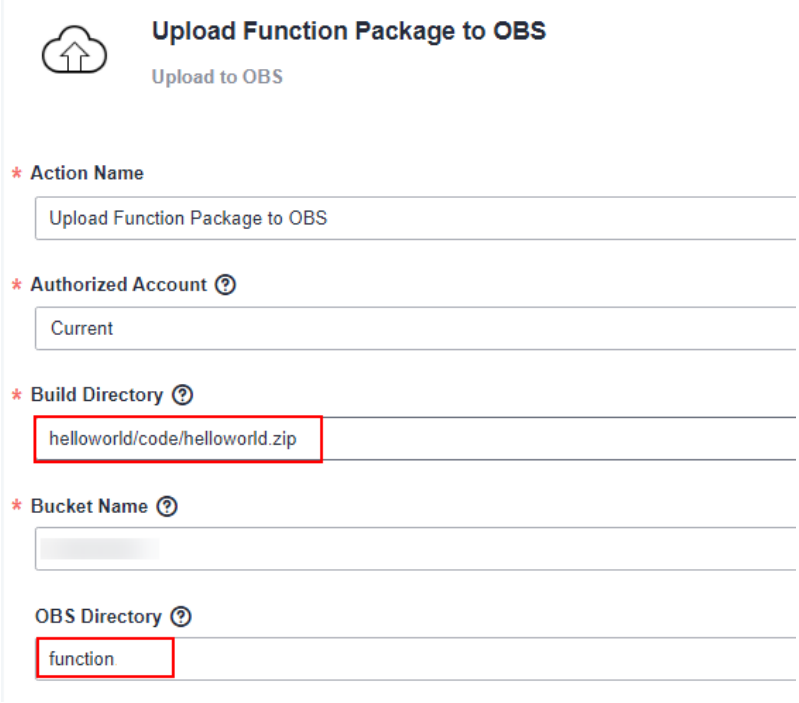
```
# Build a function deployment package.
cd helloworld
zip helloworld_deploy.zip cam.yaml
# Build a function code package.
cd code
zip -rp helloworld.zip *
```

Figure 10-21 Run Shell Commands



   b. Configure the **Upload Files to OBS** action.

Figure 10-22 Upload Files to OBS



- **Action Name**: Enter **Upload Function Package to OBS**.
- **Build Directory**: Enter **helloworld/code/helloworld.zip**.
- **Bucket Name**: Specify a private bucket to store the function code ZIP package.
- **OBS Directory**: Enter **function**.

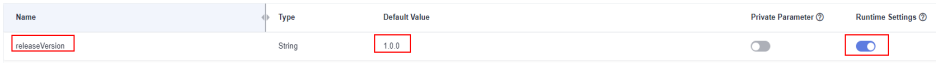
c. Configure the **Upload Deployment Package to CloudRelease** action.

Figure 10-23 Upload Deployment Package to CloudRelease

- **Action Name**: Enter **Upload Deployment Package to CloudRelease**.
- **Package Location**: Enter **helloworld/helloworld_deploy.zip**.
- **Version**: Enter **${releaseVersion}**.

4. On the **Parameters** tab page, add **releaseVersion** and enable **Runtime Settings**.

   Figure 10-24 Setting parameters

5. On the **Basic Information** tab page, change the task name to **pipeline-update-function-deploy** and click **Save**.

Creating a Deployment Task

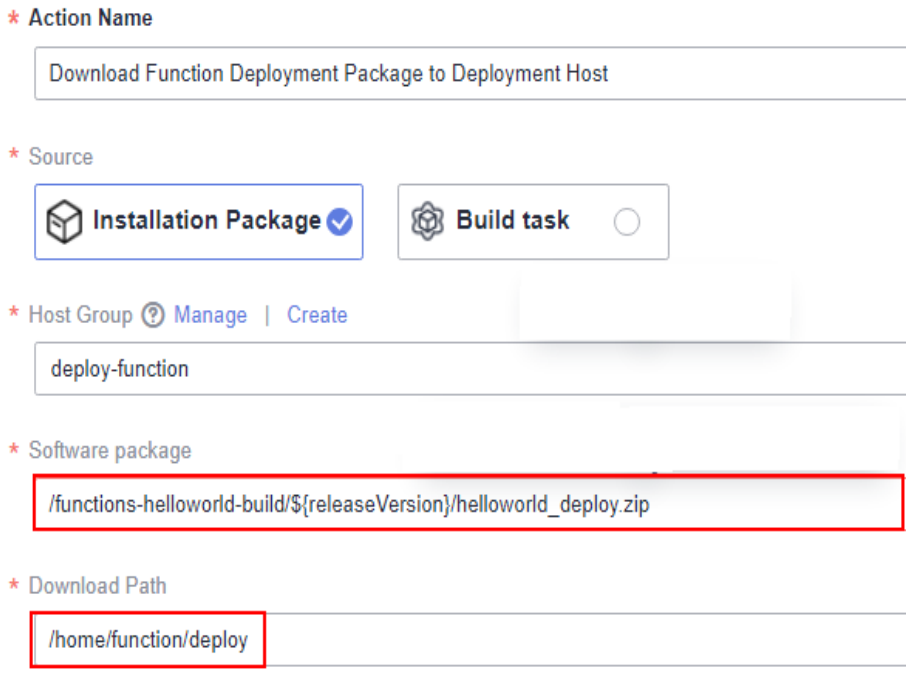
1. Choose **Build & CloudArtifact** > **CloudDeploy**, and click **Create Task**.
2. Select **Blank Template** and click **Next**.
3. Add the actions **Select Deployment Source** and **Run Shell Commands**.

   Figure 10-25 Adding deployment actions

   a. Configure the **Select Deployment Source** action.

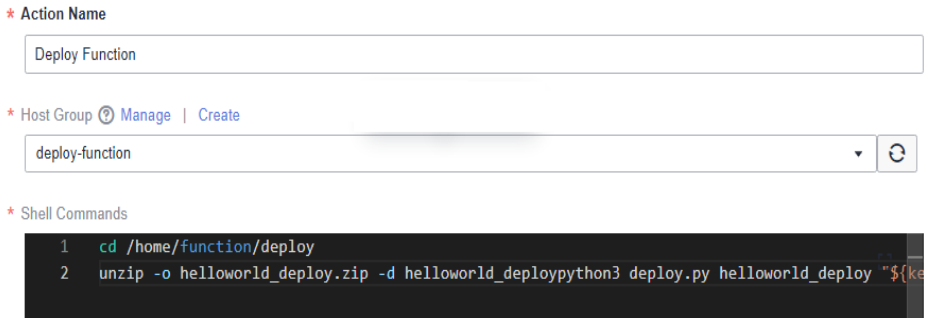
Figure 10-26 Setting action name to "Download Function Deployment Package to Deployment Host"



- **Action Name**: Enter **Download Function Deployment Package to Deployment Host**.
- **Host Group**: Select **deploy-function**.
- **Software package**: Select **/functions-helloworld-build/${releaseVersion}/helloworld_deploy.zip**.
- **Download Path**: Enter **/home/function/deploy**.

b. Configure the **Run Shell Commands** action.

Figure 10-27 Setting action name to "Deploy Function"



- **Action Name**: Enter **Deploy Function**.
- **Host Group**: Select **deploy-function**.

- **Shell Commands**:

```
cd /home/function/deploy
unzip -o helloworld_deploy.zip -d helloworld_deploy
python3 deploy.py helloworld_deploy "${key}"
```

4. Add two parameters.
   - **releaseVersion**: Use the default value **1.0.0** and enable **Runtime Settings**.
   - **key**: Enter a password and enable **Private Parameter**.

   Figure 10-28 Setting parameters

5. On the **Basic Information** tab page, change the task name to **update-function-deploy** and click **Save**.

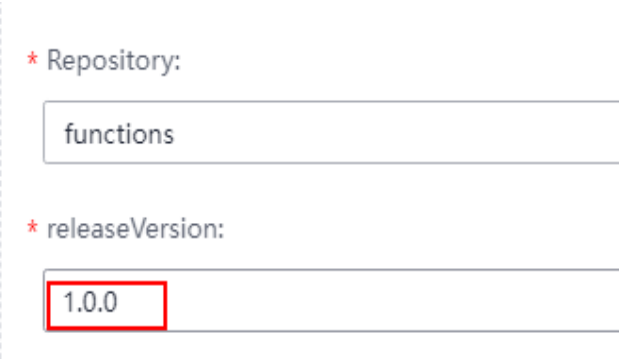
Configuring a Pipeline

1. Choose **Build & CloudArtifact** > **CloudPipeline**, and click **Create Pipeline**.
2. Select **functions** for **Source Code Repository** and **Blank Template** for the template.
3. Configure **Build and Check**.
   a. Add a build task, and select the **functions-helloworld-build** task.

   Figure 10-29 Adding a task

   b. Set **releaseVersion** as a pipeline parameter.

Figure 10-30 Setting releaseVersion

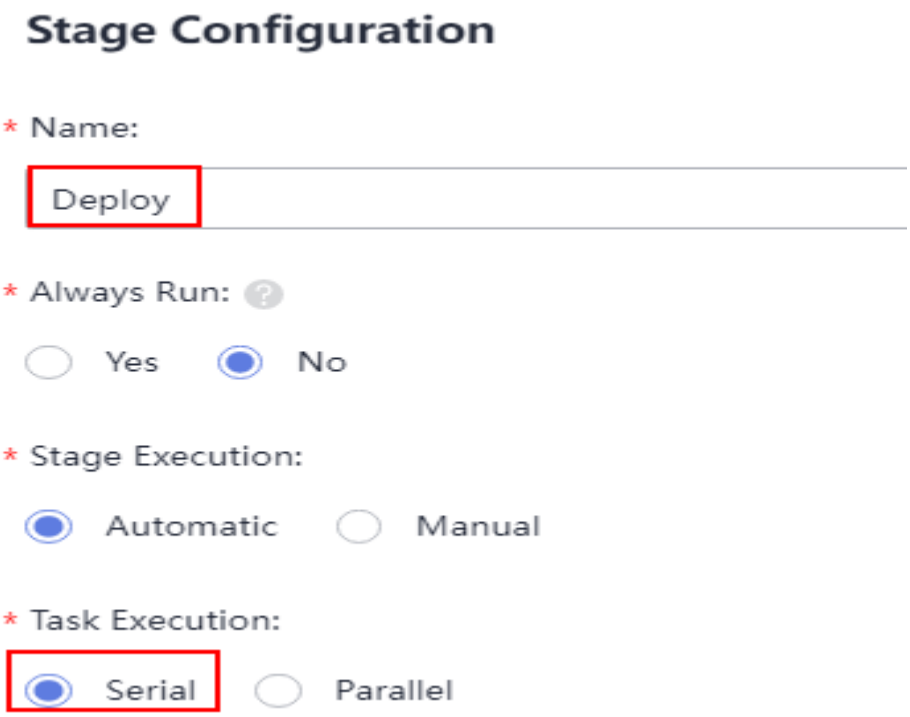


c. Click **Save**.

4. Configure a deployment task.

a. Add a stage named **Deploy** after **Build_and_Check**, set **Task Execution** to **Serial**, and click **Save**.

Figure 10-31 Configuring the stage

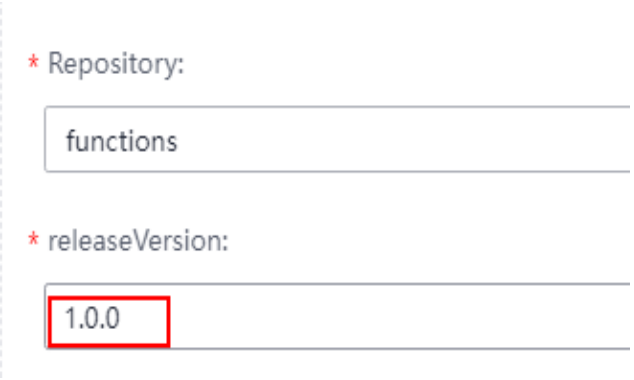


b. Click **Add Task** to add a deployment task named **DeployhelloworldScript**, and select the **update-function-deploy** task.

Figure 10-32 Adding a task

c. Set **releaseVersion** as a pipeline parameter.

Figure 10-33 Setting releaseVersion



d. Click **Save**.

5. On the **Basic Information** tab page, change the pipeline task name to **pipeline-update-function-helloworld** and click **Save**.
6. Execute the pipeline.

   Set the runtime parameter **releaseVersion** to **1.0.0** and click **Execute**.

10.3 Sample Code of deploy.py

Sample Code

The following is a sample code of **deploy.py** for automated deployment.

This example is used for automating the deployment and update of Huawei Cloud FunctionGraph functions, covering both configuration and code updates. The script parses configuration files, runs update commands via the CLI, decrypts encrypted data, and records logs for traceability. For details, see the code comments.

```
# -*-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):
    # Function handler
    handler = new_config['handler']
    # Runtime (required and not modifiable)
    runtime = new_config['runtime']
    # Memory
    memory_size = new_config['memorySize']
    # Timeout
    timeout = new_config['timeout']
    # Project ID
    project_id = new_config['projectID']
    # Command for updating the function configuration
    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}"'

    # Environment variables
    # Environment variables are directly overwritten. Manually configured variables that are not updated to
the cam.yaml file will be lost.

    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}'

    # Keep this part if a VPC is used.
    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"]}'

    # Keep "xrole": "function-admin" and "app_xrole": "function-admin" if an agency is specified.
    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}"'

    # Configure the initializer and initialization timeout.
    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}'

    # Concurrency settings
    strategy_config = new_config.get('strategyConfig', None)
    if strategy_config is not None:
        concurrency = strategy_config.get('concurrency', None)
        # Maximum number of concurrent requests per instance
        concurrent_num = strategy_config.get('concurrentNum', None)
        update_cmd = update_cmd + \
                f' --strategy_config.concurrency="{concurrency}" ' \
                f'--strategy_config.concurrent_num={concurrent_num}'

    # Keep this part if a file system is mounted to the function.
    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)
        # Return an error if the function code has no changes to update.
        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']
    # Obtain the function URN.
    function_urn = "urn:fss:" + region + ":" + project_id + \
                   ":function:default:" + function_name + ":latest"
    logging.info(f"start to deploy functionURN:{function_urn}")

    # Query the function configuration.
    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)
    # Check whether a VPC and an agency have been configured for the function. If yes, they must be
included during function updates.
    old_function_config = json.loads(result)
    check_result("query function config", result)

    # Query the function code. Keep this part if a dependency is bound to the function.
    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)

    # Update the function code.
    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)

    # Update the function configuration.
    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 Analyzing cam.yaml

Example

```
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
```

Parameter Description

Function configuration is included in **properties** of the **cam.yaml** file. The following table details the function configuration.

Parameter	Mandatory	Can Be Updated	Description
region	Yes	No	Region where the function is located.
codeUri	Yes	No	Location of the function code. It is an OBS address where the function code package is stored.
projectID	Yes	No	Project ID.
handler	Yes	Yes	Function handler.
runtime	Yes	No	Execution environment. Options: Python 2.7 Python 3.6 PHP 7.3 Java 8 Node.js 6.10 Node.js 8.10 C# (.NET Core 2.0) C# (.NET Core 2.1) C# (.NET Core 3.1) Custom
memorySize	Yes	Yes	Memory size. Unit: MB. Enumerated values: 128, 256, 512, 768, 1024, 1280, 1536, 1792, 2048, 2560, 3072, 3584, 4096
timeout	Yes	Yes	Timeout. Value range: 3s to 900s.
userData	No	Yes	Name/value information defined for the function.
encryptedUserData	No	Yes	Name/value information to be encrypted.
initializerTimeout	No	Yes	Initialization timeout. Value range: 1s to 300s.

Parameter	Mandatory	Can Be Updated	Description
initializerHandler	No	Yes	Function initializer. It must be in the format of "xx.xx". For example, if the function file name is **myfunction.js** and the initializer function is **initializer**, the initializer is **myfunction.initializer**.
concurrentNum	No	Yes	Maximum number of concurrent requests per instance.
concurrency	No	Yes	Maximum number of instances per function. Value **0** indicates that a function is disabled, and value **–1** indicates that there is no instance limit. For example, the value **100** means that a function can have a maximum 100 instances, including common and reserved instances.

NOTE

1. Currently, the **cam.yaml** file does not support the update of VPC, agency, file system, and dynamic memory settings. If a function uses a VPC, agency, file system, or dynamic memory settings, configure them on the function details page. These settings will be kept when executing the function update pipeline.
2. To avoid displaying **encryptedUserData** in plaintext in the **cam.yaml** file, the CI/CD process encrypts its value using AES with Galois/Counter Mode (GCM). The ciphertext is the value of **encryptedUserData** in the **cam.yaml** file. This value is transferred in ciphertext in both the **functions** repository and the function update pipeline. It is decrypted and updated when the function is deployed. Therefore, the key for AES encryption must be provided when the function update pipeline is executed. Example:

   Value of **encryptedUserData** in plaintext:

   ```
   '{"password":"123"}'
   ```

   After AES-GCM encryption:

   ```
   {"nonce": "ZEUOREFaiahRbMz+K9xQwA==", "header": "aGVhZGVy", "ciphertext": "SCxXsffvpU1BF2Ci8a2RedNQ", "tag": "a+EYRVPOsQ+YpQkMuFg1wA=="} ciphertext is the encrypted value.
   ```

Keep the key for AES encryption properly.

Python AES-GCM example: **https://pycryptodome.readthedocs.io/en/latest/src/cipher/modern.html?highlight=GCM#gcm-mode**

The AES-GCM encryption script is as follows:

```
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)
```

To execute the script, run the following command on an ECS:

python3 aes_gcm_encrypt_tool.py *"16-byte key"* **'{"password":"123"}'**

