FunctionGraph

Developer Guide

Issue 01

Date 2021-08-28

HUAWEI

HUAWEI TECHNOLOGIES CO., LTD.

Trademarks and Permissions


and other Huawei trademarks are trademarks 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 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.

Contents

1 Overview

1.1 Function Development

Supported Runtimes

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

NOTE

You are advised to use the latest runtime version.

Table 1-1 Runtime description

Runtime	Supported Version	SDK Download Link	Runtime Download Link
Node.js	6.10, 8.10, 10.16, and 12.13	-	-
Python	2.7 and 3.6	-	**Python 2.7 Runtime** **Python 3.6 Runtime** **NOTE** The Python runtime has integrated with common third-party SDKs listed in **Table 1-3**.

Runtime	Supported Version	SDK Download Link	Runtime Download Link
Java	8.0	Java SDK **NOTE** The Java runtime has integrated with Object Storage Service (OBS) SDKs.	-
Go	1.8 and 1.x	Go1.8.3 SDK Go1.x SDK	-
C#	.NET Core 2.0, .NET Core 2.1, and .NET Core 3.1	CsharpSDK	-
PHP	7.3	-	-
Custom	-	-	-

Third-Party Components Integrated with the Node.js Runtime

Table 1-2 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 SDK	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 API Gateway. 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

Non-Standard Libraries Integrated with the Python Runtime

Table 1-3 Non-standard libraries integrated with the Python Runtime

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

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.

Table 1-4 Download links of the sample project packages

Function	Project Package
Node.js function	**fss_examples_nodejs6.10.zip**
Python function	**fss_examples_python2.7.zip**
Java function	**fss_example_java8.jar**
Go function	**fss_examples_go1.8.zip**
C# function	**fss_example_csharp2.0** and **fss_example_csharp2.1**
PHP function	**fss_examples_php7.3.zip**

CLI

FunctionGraph provides the command line interface (CLI) for managing functions, triggers, and aliases, and invoking functions. CLI can run on a 64-bit Linux OS, 64-bit Windows OS, or macOS. **Table 1-5** provides the download links and MD5 verification codes for the CLI.

Table 1-5 Download links of the CLI

OS	Download Link	MD5 Verification Code
Linux	**CLI for Linux**	5d6332f839749ef80547f1b8561df9d3
Windows	**CLI for Windows**	73607b2fe289293ab49f17a0b4f61cf5
macOS	**CLI for macOS**	94d5667472a7f4f696d12e7834503229

HSS

Huawei Serverless Sandbox (HSS) is used to test functions locally, verify Huawei Serverless Application Model (HSAM) templates, and generate valid sample payloads for different event sources. **Table 1-6** provides the download links and MD5 verification codes for the HSS.

Table 1-6 Download links for the HSS

OS	Download Link	MD5 Verification Code
Linux	**HSS for Linux**	e70816fef93cf854c8c7d350e4a4a371
macOS	**HSS for macOS**	54b18c96e662f0b31a68b5967cf59f18

OS	Download Link	MD5 Verification Code
Windows	HSS for Windows	30b1184db23f250530c0794d77faf937

1.2 Supported Event Sources

This section describes the cloud services that can be configured as event sources for your FunctionGraph functions. After you preconfigure the event source mapping, these event sources automatically invoke the relevant function when detecting events.

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**.

DMS

Distributed Message Service (DMS) is a message queuing service that enables communication between distributed applications. If you create a function with a DMS trigger, messages automatically polled from a specified queue will be passed as a parameter (**DMS example event**) to invoke the function. For details, see **Using a DMS Trigger**.

API Gateway

API Gateway is an API hosting service that helps enterprises to build, manage, and deploy APIs at any scale. With API Gateway, your function can be invoked through HTTPS by using a custom REST API and a specified backend. You can map each API operation (such as, GET and PUT) to a specific function. API Gateway invokes the relevant function when an HTTPS request (**API Gateway example event**) is sent to the API backend. For details, see **Using an APIG Trigger**.

OBS

Object Storage Service (OBS) is a stable, secure, efficient, and easy-to-use cloud storage service. You can create a function to process OBS bucket events, for example, creating and deleting objects. 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**.

DIS

Data Ingestion Service (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 location-tracking events (**DIS example event**). FunctionGraph periodically polls the stream for new data records. For details, see **Using a DIS Trigger**.

Timer

You can schedule a timer (**timer example event**) to invoke your code based on a fixed rate of minutes, hours, or days or a cron expression. For details, see **Using a Timer Trigger**.

LTS

Log Tank Service (LTS) collects and stores logs, allowing you to query them in real time. If you create a function with an LTS trigger, subscribed logs collected by LTS will be passed as a parameter (**LTS example event**) to invoke the function. Then, the function processes or analyzes the logs, or loads the logs to other systems. For details, see **Using an LTS Trigger**.

CTS

Cloud Trace Service (CTS) collects operation records of subscribed cloud resources. If you create a function with a CTS trigger, collected operation records of specified cloud services will be passed as a parameter (**CTS example event**) to invoke the function. Then, the function analyzes and processes key information in the operation records, automatically recovers system or network modules, or reports alarms to service personnel by SMS or email. For details, see **Using a CTS Trigger**.

DDS

Document Database Service (DDS) is a MongoDB-compatible database service that provides a variety of functions including DB instance creation with a few clicks, scaling, disaster recovery, backup, restoration, and monitoring. If you create a DDS trigger for a function, any updates to the specified database table will trigger the function. For details, see **Using a DDS Trigger**.

DMS for Kafka

DMS for Kafka is a message queuing service that provides Kafka premium instances. If you create a Kafka trigger for a function, when a message is sent to a Kafka instance topic, FunctionGraph will retrieve the message and trigger the function to perform other operations. For details, see **Using a Kafka Trigger**.

Cloud Eye

Cloud Eye is a multi-dimensional resource monitoring platform. FunctionGraph is interconnected with Cloud Eye to report metrics, allowing you to view function metrics and alarm messages through Cloud Eye. For more information about metrics, see **Viewing Function Metrics**.

Example Events

- SMN example event

In the following example, the function name is **test** and the topic name is **serverless_Test**. **subject** is the message header, and **message** is the message body.

```
{
   "record": [{
      "event_version": "1.0",
      "smn": {
         "message_attributes": null,
         "subject": "This is the message title of smn trigger",
         "message_id": "a5dbd701d70d4ac2a153e7dd6dd9b601",
         "topic_urn": "urn:smn:xx-xxxxx-
x:bb8695913bb74682b9ca82a8803b60d8:serverless_Test",
         "type": "notification",
         "message": "This is the message body of smn trigger",
         "timestamp": "2017-11-15T04:02:24Z"
         },
      "event_source": "smn",
      "event_subscription_urn": "urn:fss:xx-xxxxx-
xxx:bb8695913bb74682b9ca82a8803b60d8:function:default:obsTrigger-Test1:latest"
   }],
   "functionname": "test",
   "requestId": "7c307f6a-cf68-4e65-8be0-4c77405a1b2c",
   "timestamp": "Wed Nov 15 2017 12:00:00 GMT+0800 (CST)"
}
```

- DMS example event

  In the following example, the function name is **test**. **Messages** is the message body, which includes **Body** and **Attributes**. The event also includes a queue ID and consumer group ID.

```
{
   "Tag": "latest",
   "QueueID": "48d11a53-e498-4964-84a2-aaa56442d5ca",
   "ConsumerGroupID": "g-61d9d290-7dae-4144-9ce7-00b9f159cdbe",
   "Messages": [{
      "Body": "This is dms trigger body",
      "Attributes": {
         "dmstrigger": "dmstrigger",
         "dmstrigger2": "dmstrigger"
      }
   }],
   "functionname": "test",
   "requestId": "57453335-ccf7-4d43-8639-79d108158749",
   "timestamp": "Wed Nov 15 2017 14:26:10 GMT+0800 (CST)"
}
```

- API Gateway example event

  NOTE

  When a function is invoked due to an API Gateway event, the request body is encrypted by default.

```
{
   "httpMethod": "GET",     //Request method
   "path": "/test/hello",   //Request path
   "pathParameters": {      //Path parameters
      "proxy": "hello"
   },
   "queryStringParameters": { //Query parameters
      "name": "me"
   },
   "headers": {              //Request header
```



```
        "x-stage": "RELEASE",
        "Host": "3f96e175-d5e4-4d4b-ae7e-f6d264e63b23-apigw.xx-xxx-x.xxx.com",
        "User-Agent": "lua-resty-http/0.10 (Lua) ngx_lua/10008",
        ...
    },
    "body": "...",              //Request body
    "isBase64Encoded":false/true,  //Indicates whether the request body is encoded using
Base64.
    "requestContext": {
        "stage": "test",          //Environment name
        "requestId": "dd4337362c02c7d77299e78781beb4b1",
        "apiId": "41b45ea3-70b5-11e6-b7bd-69b5aaebc7d9"
    },
}
```

The function returns characters strings by using the following structure.

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

- DIS example event

  In the following example, the stream name is **dis-swtest**. This example shows the format of a request received by a function associated with the DIS trigger.

```
{
    "ShardID": "shardId-0000000000",
    "Message": {
        "next_partition_cursor":
"eyJnZXRJdGVyYXRvclBhcmFtIjp7InN0cmVhbS1uYW1lIjoiZGlzLXN3dGVzdCIsInBhcnRpdGlvbi
1pZCI6InNoYXJkSWQtMDAwMDAwMDAwMCIsImN1cnNvci10eXBlIjoiVFJJTV9IT1JJWk9OIiwi
c3RhcnRpbmctc2VxdWVuY2UtbnVtYmVyIjoiNCJ9LCJnZW5lcmF0ZVRpbWVzdGFtcCI6MTUw
OTYwNjM5MjE5MX0",
        "records": [{
            "partition_key": "shardId_0000000000",
            "data": "PEJ1ZmZlcj48L0J1ZmZlcj4=",
            "sequence_number": "0"
        },
        {
            "partition_key": "shardId_0000000000",
            "data": "PEJ1ZmZlcj48L0J1ZmZlcj4=",
            "sequence_number": "1"
        },
        {
            "partition_key": "shardId_0000000000",
            "data": "PEJ1ZmZlcj48L0J1ZmZlcj4=",
            "sequence_number": "2"
        },
        {
            "partition_key": "shardId_0000000000",
            "data": "PEJ1ZmZlcj48L0J1ZmZlcj4=",
            "sequence_number": "3"
        }],
        "millis_behind_latest": ""
    },
    "Tag": "latest",
    "StreamName": "dis-swtest"
}
```

- Timer example event


```
{
  "version": "v1.0",
  "time": "2006-01-02T15:04:05-07:00",  // Local time
  "trigger_type": "TIMER",   // Trigger type
  "trigger_name": "Trigger name"
  "user_event": "User-defined event"   //Event configured when the timer trigger is created
}
```

- LTS example event

```
{
   "lts": {
   "data":
"eyJsb2dzIjpbIntcIm1lc3NhZ2VcIjpcIjIwMTgtMDYtMjYvMTg6NDA6NTMgW0lORl0gW2Nvbm
ZpZy5nbzo3Ml0gU3VjY2Vzc2Z1bGx5IGxvYWRlZCBnZW5lcmFsIGNvbmZpZ3VyYXRpb24gZml
sZVxcclxcblwiLFwidGltZVwiOjE1MzAwMDk2NTMwNTksXCJob3N0X25hbWVcIjpcImVjcy10Z
XN0YWdlbnQubm92YWxvY2FsXCIsXCJpcFwiOlwiMTkyLjE2OC4xLjk4XCIsXCJwYXRoXCI6XCIv
dXNyL2xvY2FsL3RlbGVzY29wZS9sb2cvY29tbW9uLmxvZ1wiLFwibG9nX3VpZFwiOlwiNjYzZD
Y5MzAtNzkyZC0xMWU4LThiMDgtMjg2ZWQ0ODhjZTcwXCIsXCJsaW5lX25vXCI6NjE1fSIsIntcI
m1lc3NhZ2VcIjpcIjIwMTgtMDYtMjYvMTg6NDA6NTMgW1dSTl0gW2NvbmZpZy5nbzo4Ml0g
VGhlIHByb2plY3RJZCBvciBpbnN0YW5jZUlkIG9mIGNvbmZpZy5qc29uIGlzIG5vdCBjb25zaXN0
ZW50IHdpdGggbWV0YWRhdGEsIHVzZSBtZXRhZGF0YS5cXG5cIixcInRpbWVcIjoxNTMwMDA
5NjUzMDU5LFwiaG9zdF9uYW1lXCI6XCJlY3MtdGVzdGFnZW50Lm5vdmFsb2NhbFwiLFwiaXB
cIjpcIjE5Mi4xNjguMS45OFwiLFwicGF0aFwiOlwiL3Vzci9sb2NhbC90ZWxlc2NvcGUvbG9nL2N
vbW1vbi5sb2dcIixcImxvZ191aWRcIjpcIjY2M2Q2OTMwLTc5MmQtMTFlOC04YjA5LTI4NmVk
NDg4Y2U3MFwiLFwibGluZV9ub1wiOjYxNn0iLCJ7XCJtZXNzYWdlXCIgSW4gY29uZi5qc29
uLCBwcm9qZWN0SWQgaXMgW10sIGluc2RhbmNlSWQgaXMgW10uIE1ldGFEYXRhIGlzIHs0
NTQzMjkzYS01YjJjLTQ0YzQtYjdhMC1kZTIxOGY3ZjJmYTYgNjI4MGUxNzBiZDkzNGY2MGE0Z
Dg1MWNmNWNhMDUxMjkgIH1cXHJcXG5cIixcInRpbWVcIjoxNTMwMDA5NjUzMDU5LFwia
G9zdF9uYW1lXCI6XCJlY3MtdGVzdGFnZW50Lm5vdmFsb2NhbFwiLFwiaXBcIjpcIjE5Mi4xNjgu
MS45OFwiLFwicGF0aFwiOlwiL3Vzci9sb2NhbC90ZWxlc2NvcGUvbG9nL2NvbW1vbi5sb2dcIix
cImxvZ191aWRcIjpcIjY2M2Q2OTMwLTc5MmQtMTFlOC04YjBhLTI4NmVkNDg4Y2U3MFwiLF
wibGluZV9ub1wiOjYxN30iXSwib3duZXIiOiI2MjgwZTE3MGJkOTM0ZjYwYTRkODUxY2Y1Y2E
wNTEyOSIsImxvZ19ncm91cF9pZCI6Ijk3YTlkMjg0LTQ0NDgtMTFlOC04ZmE0LTI4NmVkNDg4
Y2U3MCIsImxvZ190b3BpY19pZCI6IjFhOTY3NWE3LTc4NGQtMTFlOC05ZjcwLTI4NmVkNDg4
Y2U3MCJ9"
   }
}
```

- CTS example event

```
{
   "cts": {
      "time": "2018/06/26 08:54:07 GMT+08:00", //Timestamp of the sender
      "user": { //Information about the user that initiates the request
         "name": "userName",
         "id": "5b726c4fbfd84821ba866bafaaf56aax",
         "domain": {
            "name": "domainName",
            "id": "b2b3853af40448fcb9e40dxj89505ba"
         }
      },
      "request": {}, //Content of the trace request
      "response": {}, //Content of the trace response
      "code": 204, //Trace response codes, such as, 200 or 400
      "service_type": "FunctionGraph", //Name of the sender in abbreviated form, such as VPC and ECS
      "resource_type": "graph", //Resource type of the sender, such as VM and VPN
      "resource_name": "workflow-2be1", //Resource name, for example, the name of a VM on Elastic Cloud Server (ECS)
      "resource_id": "urn:fgs:xx-xxx-x:2d1d891d93054bbaa69b9e866c0971ac:graph:workflow-2be1", //Resource ID, for example, the ID of a VM on ECS
      "trace_name": "deleteGraph", //Trace name, such as startServer and shutDown
      "trace_type": "ConsoleAction", //Type of the trace source, such as ApiCall
```



```
        "record_time": "2018/06/26 08:54:07 GMT+08:00", //Time when CTS receives the trace
        "trace_id": "69be64a7-0233-11e8-82e4-e5d37911193e", //Trace ID
        "trace_status": "normal"
    }
}
```

- DDS example event

```
{
    "records": [
        {
            "event_source": "dds",
            "event_name": "insert",
            "region": "xx-xxxxx-x",
            "event_version": "1.0",
            "dds": {
                "size_bytes": "100",
                "token": {
                    "_data":
"825D8C2F4D0000001529295A100474039A3412A64BA89041DC952357FB4446645F696400
645D8C2F8E5BECCB6CF5370D6A0004"
                },
                "full_document": {
                    "_id": {
                        "$oid": "5d8c2f8e5beccb6cf5370d6a"
                    },
                    "name": "dds",
                    "age": {
                        "$numberDouble": "52.0"
                    }
                },
                "ns": {
                    "db": "functiongraph",
                    "coll": "person"
                }
            },
            "event_source_id": "e6065860-f7b8-4cca-80bd-24ef2a3bb748"
        }
    ]
}
```

- Kafka example event

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


1.3 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). **Table 1-7** describes the rules for packaging a function project.

Table 1-7 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 2.7	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 3.6	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 8	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 1.8	Not supported.	Compress project files into a ZIP file, and ensure that the name of the dynamic library file is consistent with the handler plugin name. For example, if the name of the dynamic library file is **testplugin.so**, set the handler plugin name to **testplugin.Handler**. **Handler** indicates the function handler.	Compress project files into a ZIP file and upload it to an OBS bucket.
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.
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
```

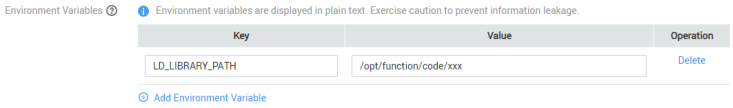
- Custom

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

1.4 Referencing DLLs in Functions

- 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.
- If the dependent **.so** file is stored in a non-root directory, specify the directory in the **LD_LIBRARY_PATH** variable on the **Configuration** tab page.

  In the following example, **/opt/function/code** indicates the project directory of the function code.



- 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 Initializer

2.1 What Is Initializer?

Introduction

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.

Application Scenarios

FunctionGraph executes a function in the following steps:

1. Allocate container resources to the function.
2. Download the 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

- Function initialization and request processing are isolated to enable clearer program logic and structure better and higher-performance code.
- Smooth function upgrade prevents performance loss caused by cold start initialization on the application layer. After new function instances are started,

FunctionGraph automatically executes the initialization logic and then processes requests.

- The overhead of application layer initialization is identified to accurately determine when resource scaling will be performed and how many resources will be required. 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 the definition of custom parameters, but 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.
- Time for execution

  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 is generated. 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.

2.2 Initializer Definition

This section describes how to define the initializer for the runtimes that support inline code editing.

Node.js

FunctionGraph supports the following Node.js runtimes:

- Node.js 6.1 (runtime = Node.js6)
- Node.js 8.9 (runtime = Node.js8)
- Nodejs10.16(runtime = Node.js10)
- Nodejs12.13(runtime = Node.js12)

Initializer syntax:

[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 as null as it is set in the previous example.

Python

FunctionGraph supports the following Python runtimes:

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

Initializer syntax:

[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:


```
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.

PHP

FunctionGraph supports the following PHP runtime:

- Php 7.2 (runtime = Php7.2)

Initializer syntax:

[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.

Java

FunctionGraph supports the following Java runtime:

- Java 8 (runtime = Java8)

  Initializer syntax:

  [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 the 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.

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

- context

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

3 Developing Functions

3.1 Developing Functions in Node.js

Function Syntax

NOTE

You are advised to use Node.js 12.13.

- Node.js 6.10

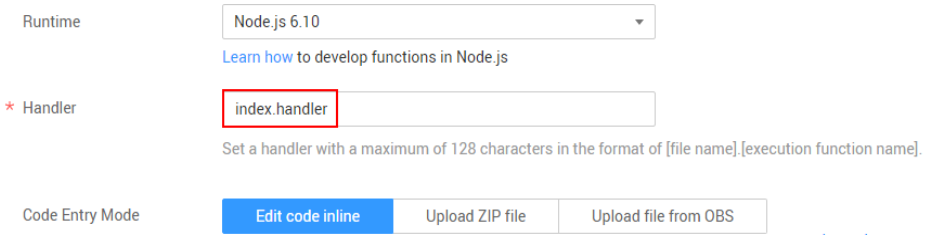
  Use the following syntax when creating a handler function in Node.js 6.10:

  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.
  - Function handler: **index.handler**.

    The function handler is in the format of *[File name].[Function name]*. For example, if you set the handler to **index.handler** in your function, as shown in **Figure 3-1**, FunctionGraph will load the handler function defined in the **index.js** file.

Figure 3-1 Handler of a Node.js function



- Node.js 8.10, Node.js 10.16, and Node.js 12.13

  Node.js 8.10, Node.js 10.16, and Node.js 12.13 are compatible with the APIs of Node.js 6.10, and supports an **async** handler.

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

  Responses are output through **return**.

SDK APIs

Table 3-1 describes the context methods provided by FunctionGraph.

Table 3-1 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) of an agency. If you use this method, you need to configure an agency for the function.
getSecretKey()	Obtains the SK (valid for 24 hours) of an agency. If you 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.

Method	Description
getCPUNumber()	Number of CPU millicores used by the function (1 core = 1000 millicores). The value of this field is proportional to that of **MemorySize**. By default, 100 CPU millicores are required for 128 MB memory. The number of CPU millicores is calculated as follows: Memory/128 x 100 + 200 (basic CPU millicores).
getProjectID()	Obtains a project ID.
getPackage()	Obtains a function group, that is, an app.
getToken()	Obtains the token (valid for 24 hours) of 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")

WARNING

Results returned by using the **getToken()**, **getAccessKey()**, and **getSecretKey()** methods contain sensitive information. Exercise caution when using these methods.

Developing a Node.js Function

Perform the following steps to develop a Node.js function:

Step 1 Create a function project.

1. Write the code for printing the context and event.

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

```
var return_helloworld = function (event, context) {
    const function_name = context.getFunctionName();
    const function_version = context.getVersion();
```



```
    const package = context.getPackage();
    const eventKey = event['key'];
    const logger = context.getLogger();
    logger.info(JSON.stringify({
        function_name,
        function_version,
        package,
        eventKey
    }));
}

module.exports = {
    print: return_helloworld
};
```

2. Define a FunctionGraph function.

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

```
var h = require('./helloworld.js');
exports.handler = function (event, context, callback) {
    h.print(event, context);
    callback(null, JSON.stringify(event));
}
```

NOTE

1. 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.
2. FunctionGraph can return only the following types of values:

   null: The HTTP response body is empty.

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

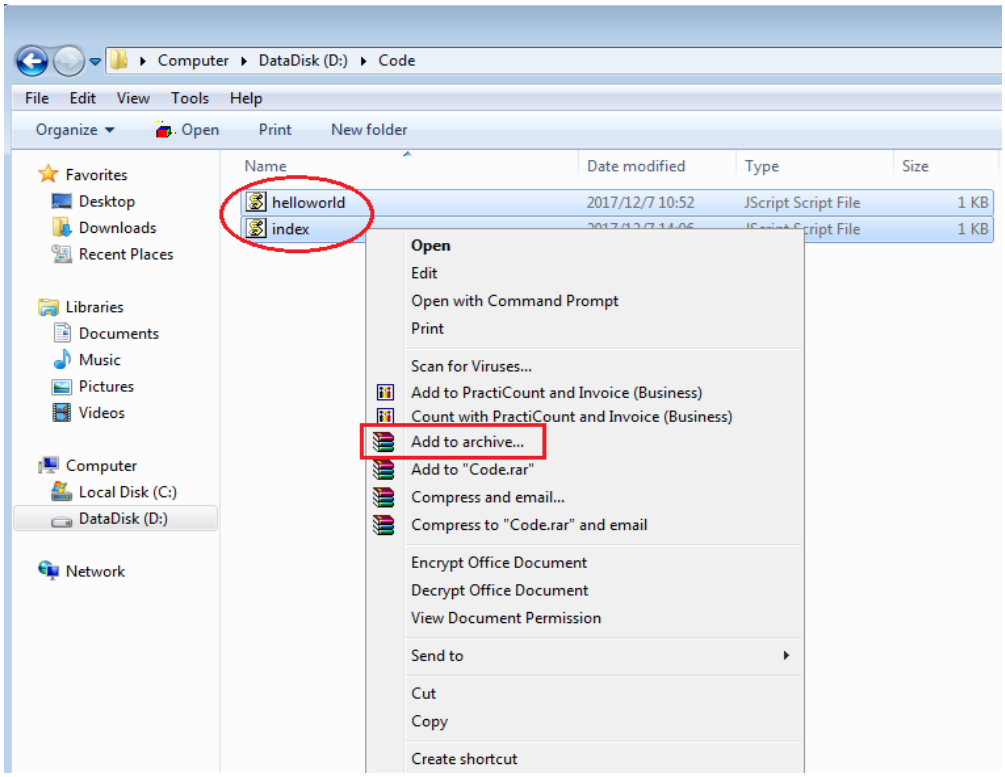
   string: The content in this string is the body of an HTTP response.

   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**.

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_nodejs6.10.zip** file, as shown in **Figure 3-2**.

Figure 3-2 Packaging the project files



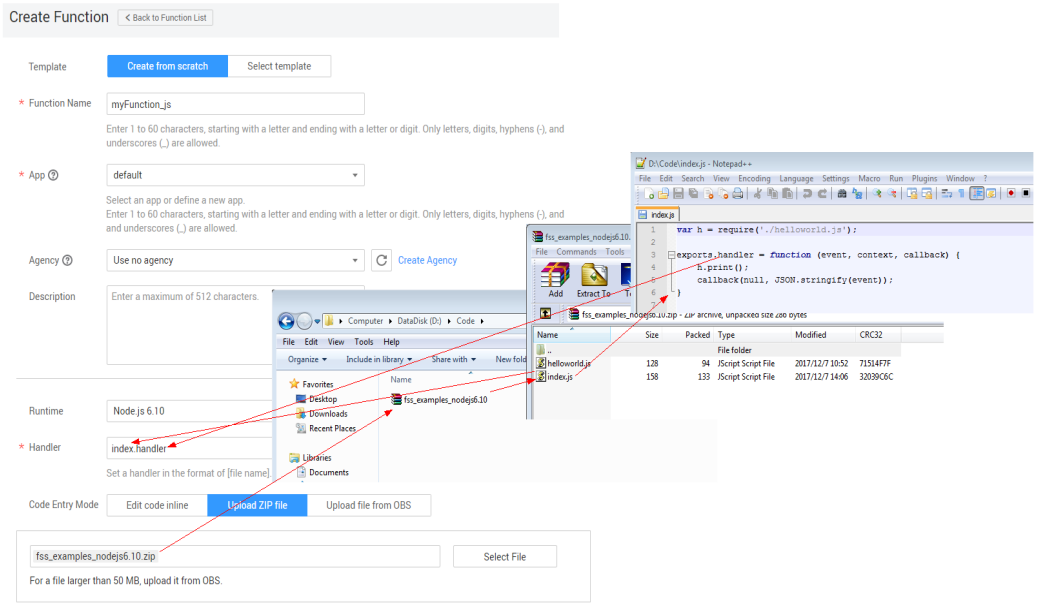
NOTICE

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.js** file, the handler of your FunctionGraph function, is under the root directory when the ZIP file is decompressed.

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

Log in to the FunctionGraph console, create a Node.js function, and upload the **fss_examples_nodejs6.10.zip** file, as shown in **Figure 3-3**.

Figure 3-3 Uploading the code package



NOTE

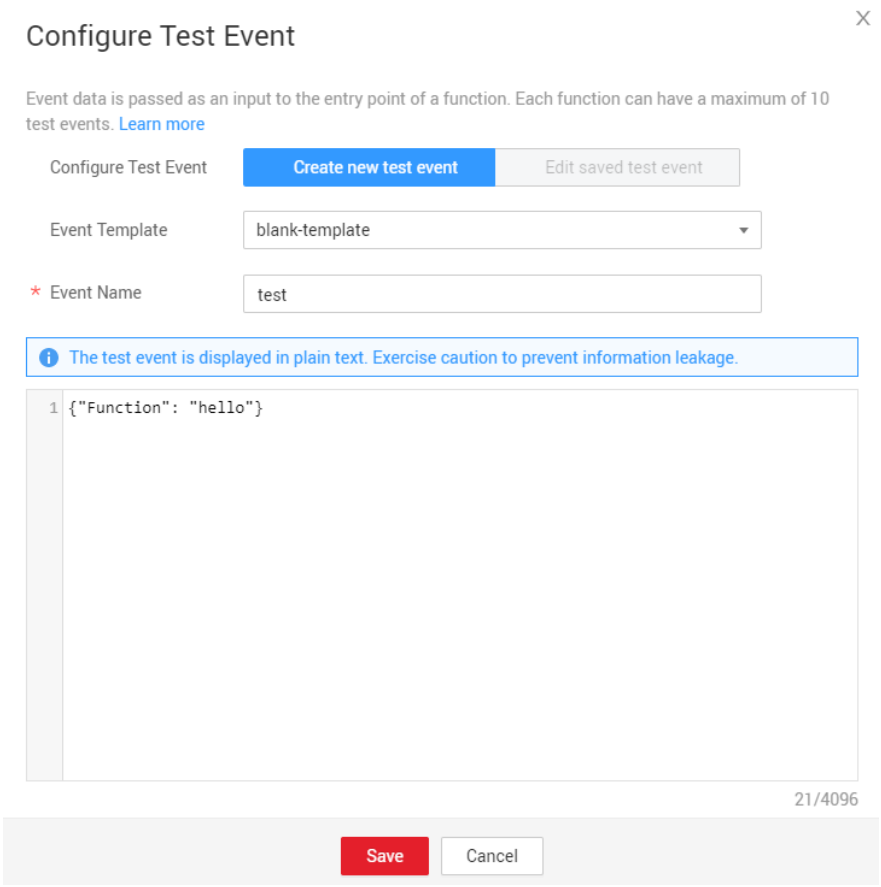
1. The **index** of the handler must be consistent with the file name of your function, because the file name will help to locate the function file.
2. The **handler** is a function name, which must be consistent with that defined in the **index.js** file.
3. After you upload the **fss_examples_nodejs6.10.zip** 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.js** file through **handler**, and then executes the function.

Step 4 Test the function.

1. Create a test event.

   On the function details page that is displayed, choose **Select test event** > **Configure test event**. Configure the test event information, as shown in **Figure 3-4**, and then click **Save**.

Figure 3-4 Configuring a test event

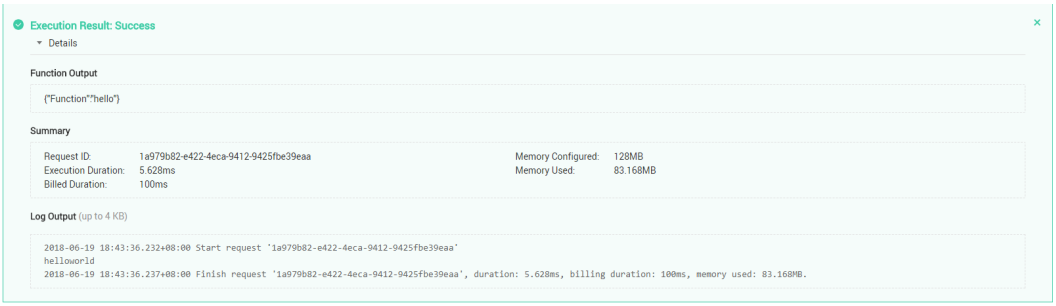


2. On the function details page, select the configured test event, and click **Test**.

Step 5 View the function execution result.

The function execution result consists of three parts: function output (returned by **callback**), summary, and logs (output by using the **console.log** or **getLogger()** method), as shown in **Figure 3-5**.

Figure 3-5 Test result



----End

Execution Result

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

Table 3-2 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** 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.

Example of the function output when a function fails to be executed:

```
{
    "errorMessage": "Unexpected identifier (./index.js:31)",
    "errorType": "SyntaxError"
}
```

Example of the log output when a function fails to be executed:

```
2020-07-14 10:29:10.375+08:00 Start invoke request '2a02e0e7-faa8-49aa-be15-d1c4cf0c7069', version:
latest
2020-07-14 10:29:10.381+08:00 2a02e0e7-faa8-49aa-be15-d1c4cf0c7069 Unexpected identifier (./index.js:
31)
./index.js:31
        function_version,
        ^^^^^^^^^^^^^^^^
SyntaxError: Unexpected identifier
2020-07-14 10:29:10.382+08:00 Finish invoke request '2a02e0e7-faa8-49aa-be15-d1c4cf0c7069'(invoke
Failed), duration: 6.632ms, billing duration: 100ms, memory used: 59.777MB
```

3.2 Developing Functions in Python

Function Syntax

NOTE

You are advised to use Python 3.6.

FunctionGraph supports Python 2.7 and Python 3.6.

Syntax for 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**.

SDK APIs

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

Table 3-3 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) of an agency. If you use this method, you need to configure an agency for the function.
getSecretKey()	Obtains the SK (valid for 24 hours) of an agency. If you 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()	Number of CPU millicores used by the function (1 core = 1000 millicores). The value of this field is proportional to that of **MemorySize**. By default, 100 CPU millicores are required for 128 MB memory. The number of CPU millicores is calculated as follows: Memory/128 x 100 + 200 (basic CPU millicores).
getProjectID()	Obtains a project ID.

Method	Description
getPackage()	Obtains a function group, that is, an app.
getToken()	Obtains the token (valid for 24 hours) of 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")

WARNING

Results returned by using the **getToken()**, **getAccessKey()**, and **getSecretKey()** methods contain sensitive information. Exercise caution when using these methods.

Developing a Python Function

Perform the following steps to develop a Python function:

NOTE

Python 2.7 is used an example.

Step 1 Create a 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 printhello():
       print 'hello world!'
   ```

2. Define a FunctionGraph function.

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

   ```
   import json
   import helloworld

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


NOTE

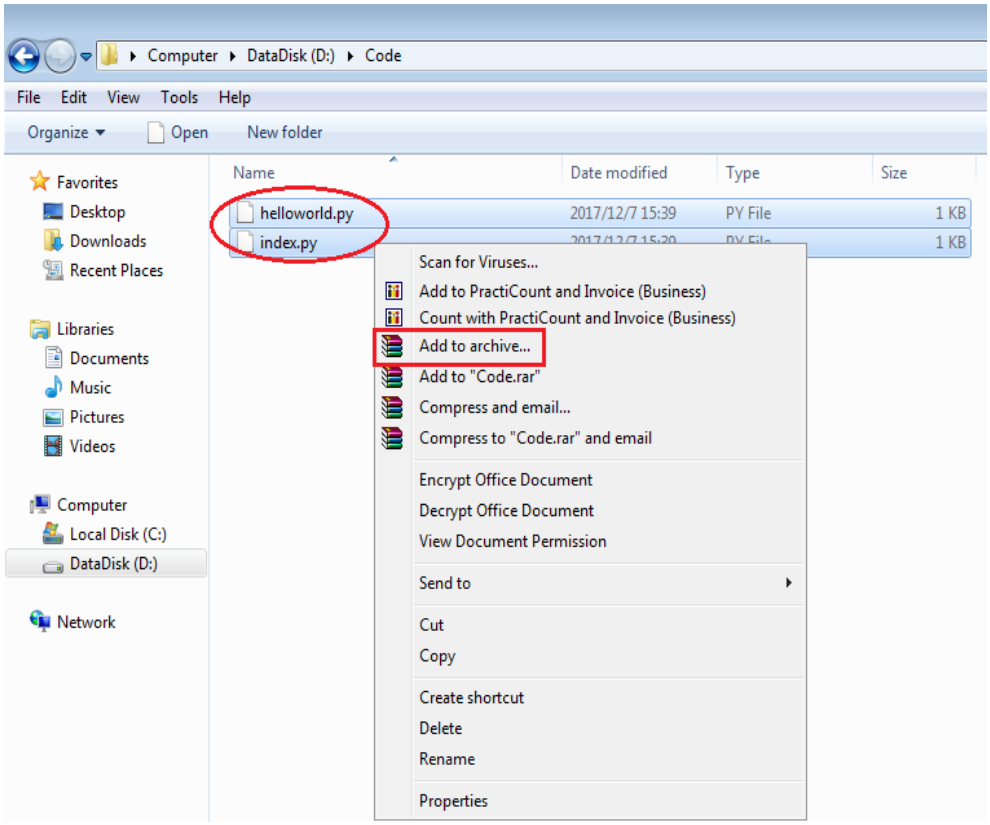
FunctionGraph can return only the following types of values:

- **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**.

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_python2.7.zip** file, as shown in **Figure 3-6**.

Figure 3-6 Packaging the project files



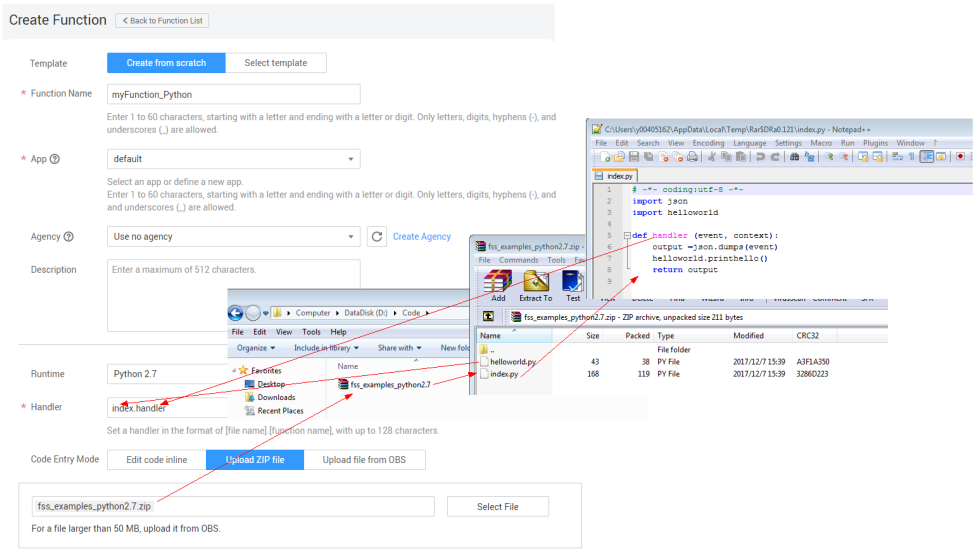
NOTICE

- 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.py** file, the handler of your FunctionGraph function, is under the root directory when the ZIP file is decompressed.
- 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 3 Create a FunctionGraph function and upload the code package.

Log in to the FunctionGraph console, create a Python function, and upload the **fss_examples_python2.7.zip** file, as shown in **Figure 3-7**.

Figure 3-7 Uploading the code package



NOTE

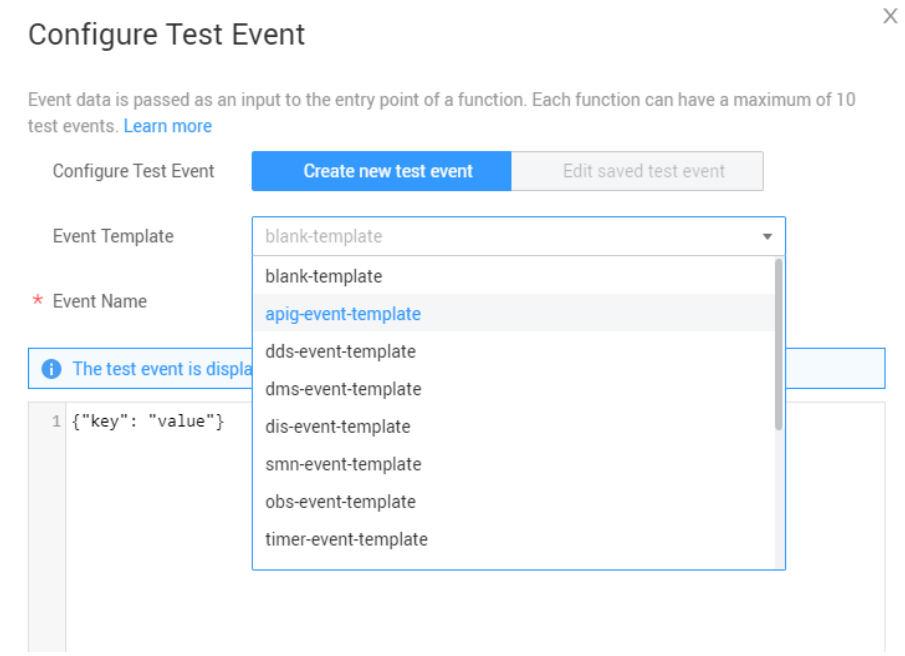
1. The **index** of the handler must be consistent with the file name of your function, because the file name will help to locate the function file.
2. The **handler** is a function name, which must be consistent with that defined in the **index.py** file.
3. After you upload the **fss_examples_python2.7.zip** 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 4 Test the function.

1. Create a test event.

   On the function details page that is displayed, choose **Select test event** > **Configure test event**. Configure the test event information, as shown in **Figure 3-8**, and then click **Save**.

Figure 3-8 Configuring a test event

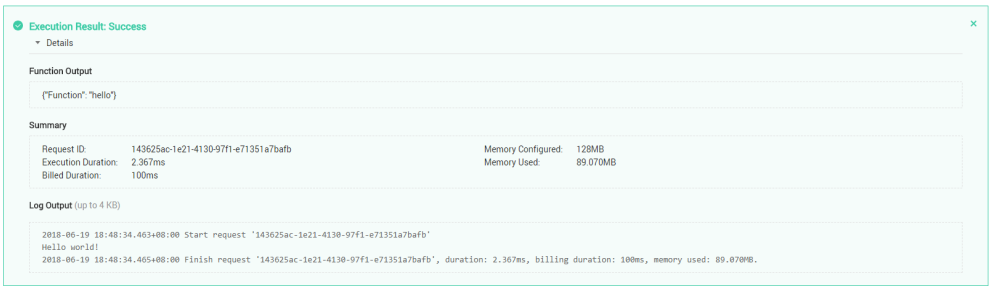


2. On the function details page, select the configured test event, and click **Test**.

Step 5 View the function execution result.

The function execution result consists of three parts: function output (returned by **callback**), summary, and logs (output by using the **console.log** or **getLogger()** method), as shown in **Figure 3-9**.

Figure 3-9 Test result



----End

Execution Result

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

Table 3-4 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.

Example of the function output when a function fails to be executed:

```
{
    "errorMessage": "Syntax error in module 'index'(invalid syntax (index.py, line 5))",
    "errorType": "SyntaxError",
    "stackTrace": [
        "File \"./index.py\", line 5",
        "  for i in range (1,1000000):"
    ]
}
```

Example of the log output when a function fails to be executed:

```
2020/07/27 14:28:14 GMT+08:00  Start invoke request '7b37809b-a086-49c9-9bfd-fdf55ec7f8d3', version:
latest
2020/07/27 14:28:14 GMT+08:00  7b37809b-a086-49c9-9bfd-fdf55ec7f8d3 Syntax error in module
'index'(invalid syntax (index.py, line 5))
Traceback (most recent call last):
  File "/opt/function/code/index.py", line 5
    for i in range (1,1000000):
                    ^
SyntaxError: invalid syntax
2020/07/27 14:28:14 GMT+08:00  Finish invoke request '7b37809b-a086-49c9-9bfd-fdf55ec7f8d3'(invoke
Failed), duration: 9.593ms, billing duration: 100ms, memory used: 65.000MB
```

3.3 Developing Functions in Java

3.3.1 Developing Functions in Java (Using Eclipse)

Function Syntax

The following is the syntax for creating a handler function in Java:

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**.

When creating a function in Java, define a handler in the format of *[Package name].[Class name].[Function name]*.

SDK APIs

The **Java SDK** provides context, event, and logging APIs.

- Event APIs

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

  a. **APIG trigger**

     i. APIGTriggerEvent methods

Table 3-5 APIGTriggerEvent methods

Method	Description
isBase64Encoded()	Checks whether the body of an event has been 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.

Method	Description
getRequestContext()	Obtains API Gateway configurations (returned as an **APIGRequestContext** object).
getHeaders()	Obtains the HTTP request header.
getQueryStringParameters()	Obtains query parameters. **NOTE** 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 encryption.
getUserData()	Obtains the user data set in the APIG custom authorizer.

Table 3-6 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.

ii. APIGTriggerResponse methods

Table 3-7 APIGTriggerResponse construction methods

Method	Description
APIGTriggerResponse()	Set the following parameters: • **headers**: null • **statusCode**: 200 • **body**: "" • **isBase64Encoded**: false

Method	Description
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 3-8 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 3-9 DISTriggerEvent methods

Method	Description
getShardID()	Obtains the partition ID.
getMessage()	Obtains the DIS message body (**DISMessage structure**).

Method	Description
getTag()	Obtains the version of a function.
getStreamName()	Obtains the stream name.

Table 3-10 DISMessage methods

Method	Description
getNextPatitionCursor()	Obtains the next partition cursor.
getRecords()	Obtains message records (**DISRecord structure**).
getMillisBehindLatest()	Reserved (Currently, **0** is returned.)

Table 3-11 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 3-12 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 the DMS message (**DMSMessage structure**).

Table 3-13 DMSMessage methods

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

d. **SMN trigger**

Table 3-14 SMNTriggerEvent method

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

Table 3-15 SMNRecord methods

Method	Description
getEventVersion()	Obtains the event version. (Currently, the 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 3-16 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 3-17 TimerTriggerEvent methods

Method	Description
getVersion()	Obtains the version. (Currently, the version is **v1.0**.)
getTime()	Obtains the current time.
getTriggerType()	Obtains the trigger type (**Timer**).
getTriggerName()	Obtains the trigger name.
getUserEvent()	Obtains the additional information of the trigger.

f. **LTS trigger**

Table 3-18 LTSTriggerEvent method

Method	Description
getLts()	Obtains LTS messages (**LTSBody structure**).

Table 3-19 LTSBody methods

Method	Description
getData()	Obtains original LTS messages.
getRawData()	Obtains UTF-8 messages decoded using Base64.

g. **CTS trigger**

Table 3-20 CTSTriggerEvent method

Method	Description
getCTS()	Obtains the CTS message body (CTS structure).

Table 3-21 CTS structure methods

Method	Description
getTime()	Obtains the time when a trace occurred.

Method	Description
getUser()	Obtains the information (user structure) about the user who triggered the trace.
getRequest()	Obtains the request content of the trace.
getResponse()	Obtains the response content of the trace.
getCode()	Obtains the response code.
getServiceType()	Obtains the name of the service involved in the trace.
getResourceType()	Obtains the type of the resource involved in the trace.
getResourceName()	Obtains the name of the resource involved in the trace.
getResourceId()	Obtains the ID of the resource involved in the trace.
getTraceName()	Obtains the trace name.
getTraceType()	Obtains the trace type, for example, **ConsoleAction**.
getRecordTime()	Obtains the time when CTS received the trace.
getTraceId()	Obtains the trace ID.
getTraceStatus()	Obtains the trace status.

Table 3-22 User methods

Method	Description
getName()	Obtains the username (an account can create multiple users).
getId()	Obtains the user ID.
getDomain()	Obtains the account information.

Table 3-23 Domain methods

Method	Description
getName()	Obtains the account name.
getId()	Obtains the account ID.

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 3-24 describes the context APIs provided by FunctionGraph.

  Table 3-24 Context methods

Method	Description
getRequestID( )	Obtains a request ID.
getRemainingTimeInMilligetRun-ningTimeInSecondsSeconds ( )	Obtains the remaining running time of a function.
getAccessKey( )	Obtains the AK (valid for 24 hours) of an agency. If you use this method, you need to configure an agency for the function.
getSecretKey( )	Obtains the SK (valid for 24 hours) of an agency. If you 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.

Method	Description
getCPUNumber( )	Number of CPU millicores used by the function (1 core = 1000 millicores). The value of this field is proportional to that of **MemorySize**. By default, 100 CPU millicores are required for 128 MB memory. The number of CPU millicores is calculated as follows: Memory/128 x 100 + 200 (basic CPU millicores).
getProjectID( )	Obtains a project ID.
getPackage( )	Obtains a function group, that is, an app.
getToken( )	Obtains the token (valid for 24 hours) of 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.

WARNING

Results returned by using the **getToken()**, **getAccessKey()**, and **getSecretKey()** methods contain sensitive information. Exercise caution when using these methods.

- Logging API

  Table 3-25 describes the logging API provided in the Java SDK.

  Table 3-25 Logging API

Method	Description
RuntimeLogger()	Records user input logs using the method **log(String string)**.

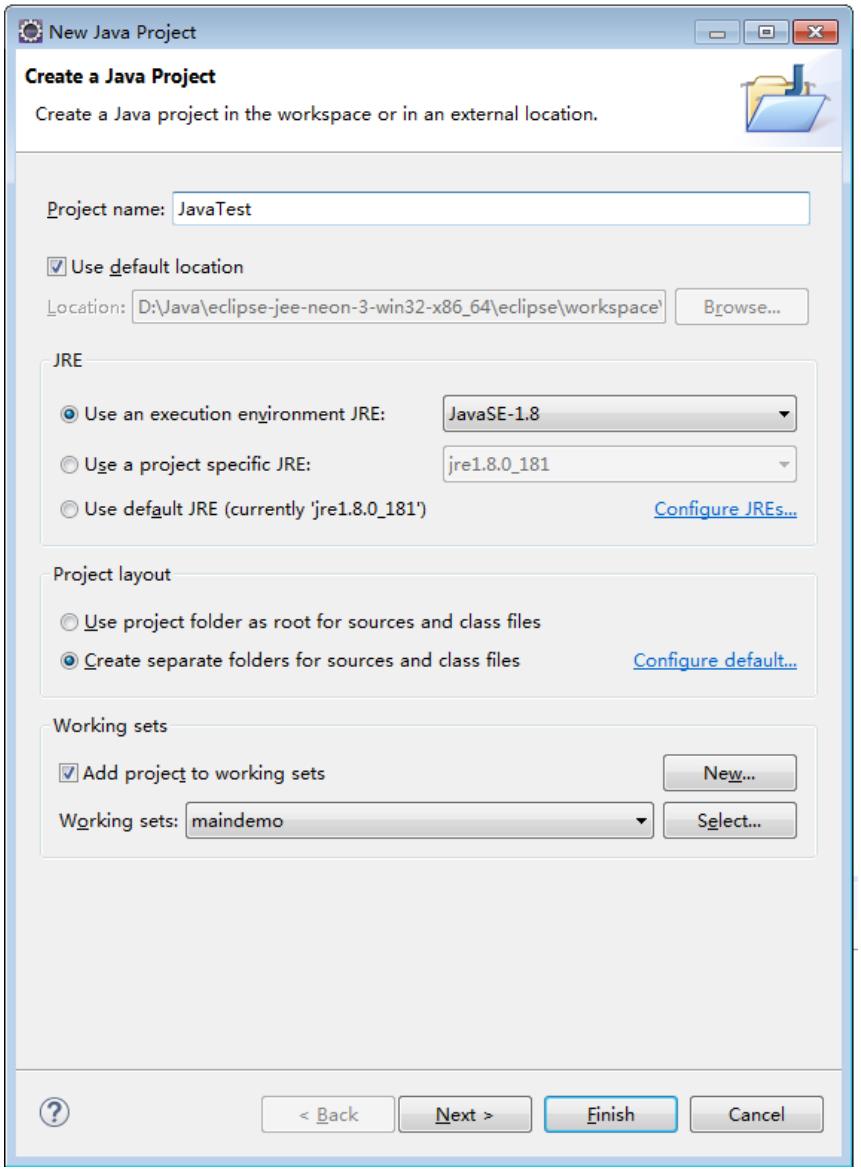
Developing a Java Function

Perform the following procedure to develop a Java function:

Step 1 Create a function project.

1. Configure Eclipse and create a Java project named JavaTest, as shown in **Figure 3-10**.

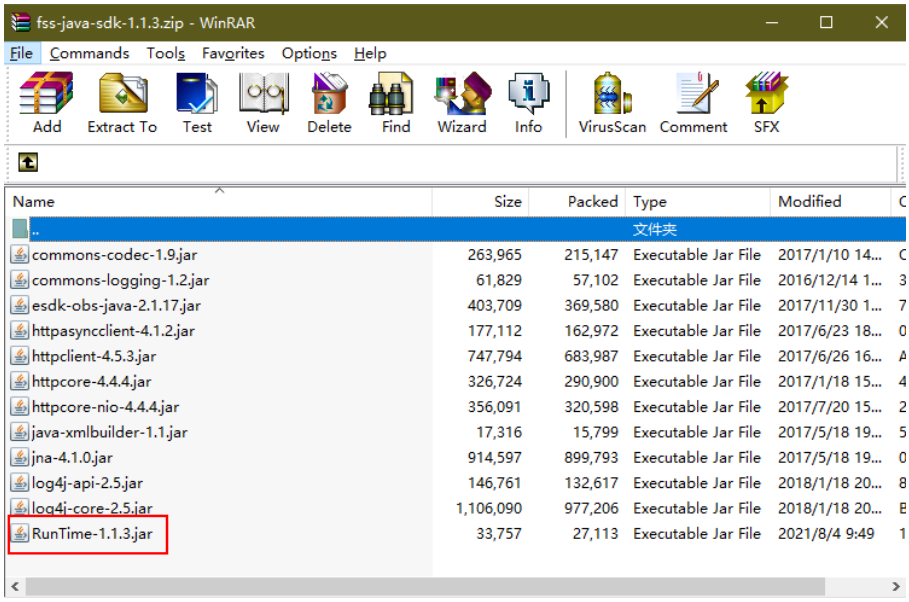
   Figure 3-10 Creating a project

2. Add a dependency to the project.

   Download the **Java SDK** to a local development environment, and decompress the SDK package, as shown in **Figure 3-11**.

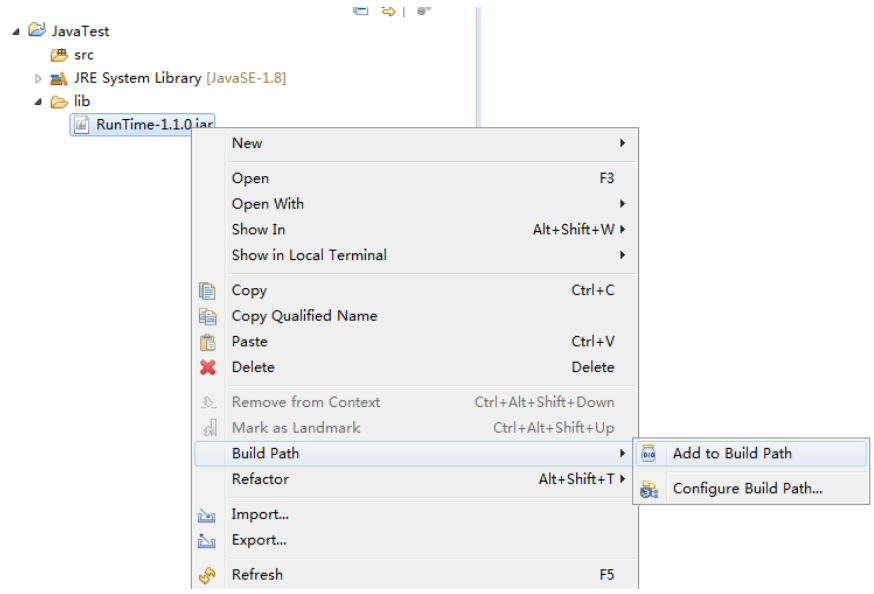
Figure 3-11 Decompressing the downloaded SDK



3. Configure the dependency.

   Create a folder named **lib** in the project directory, copy the **Runtime-1.1.3.jar** file in the SDK package to the **lib** folder, and add the JAR file as a dependency of the project, as shown in **Figure 3-12**.

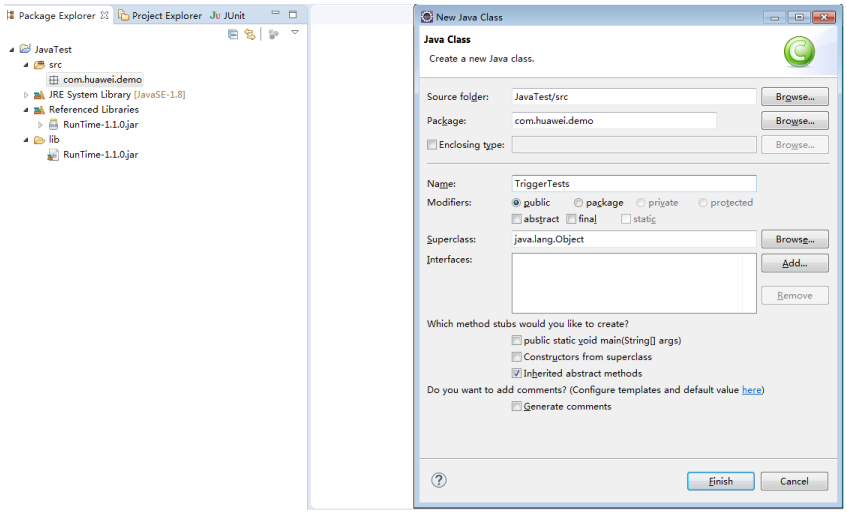
Figure 3-12 Configuring the dependency



Step 2 Create a function.

1. Create a package named **com.huawei.demo**, and then create a class named **TriggerTests** under the package, as shown in **Figure 3-13**.

Figure 3-13 Creating a package named com.huawei.demo



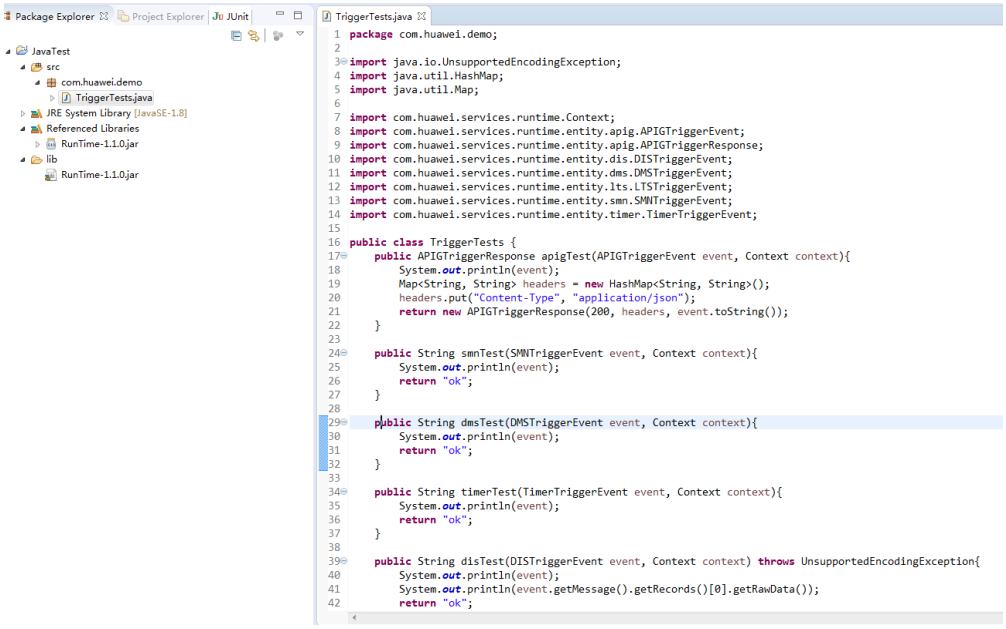
2. Define the function handler in **TriggerTests.java**, as shown in **Figure 3-14**.

```
package com.huawei.demo;

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

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

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

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

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

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

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

    public String ltsTest(LTSTriggerEvent event, Context context) throws
```



```
UnsupportedEncodingException {
        System.out.println(event);
        System.out.println("raw data: " + event.getLts().getRawData());
        return "ok";
    }
}
```

Figure 3-14 Defining the function handler



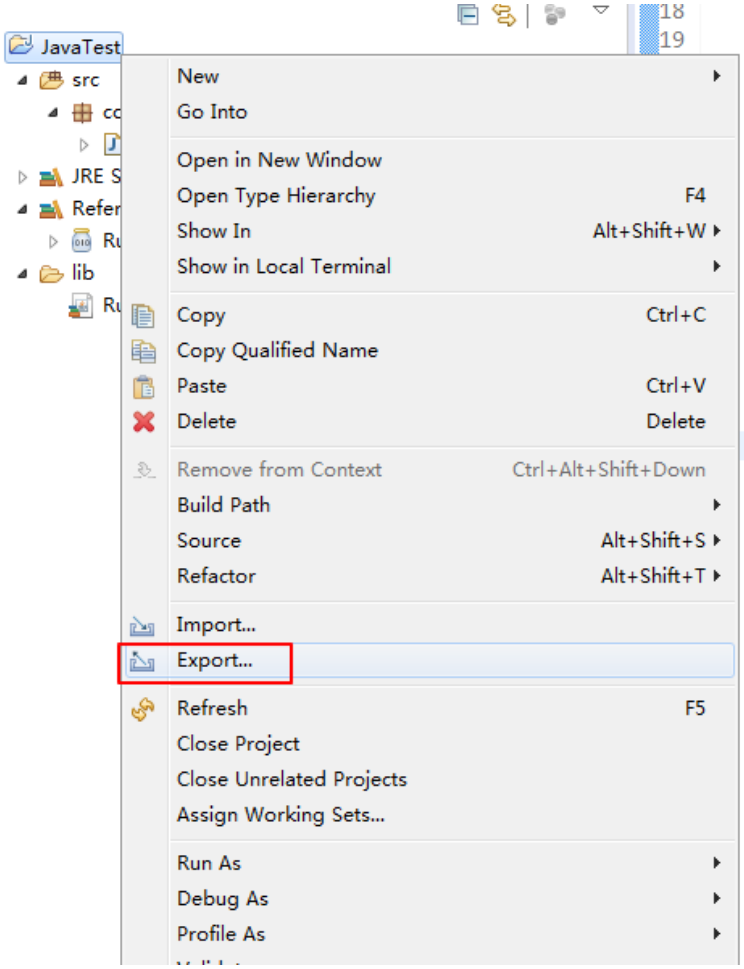
NOTE

The example code contains multiple handler functions, which use different types of triggers.

Step 3 Package the project files.

1. Right-click the **JavaTest** project and choose **Export**, as shown in **Figure 3-15**.

Figure 3-15 Packaging the project files



2. Export the project to a directory as a JAR file, as shown in **Figure 3-16** and **Figure 3-17**.

Figure 3-16 Selecting a format

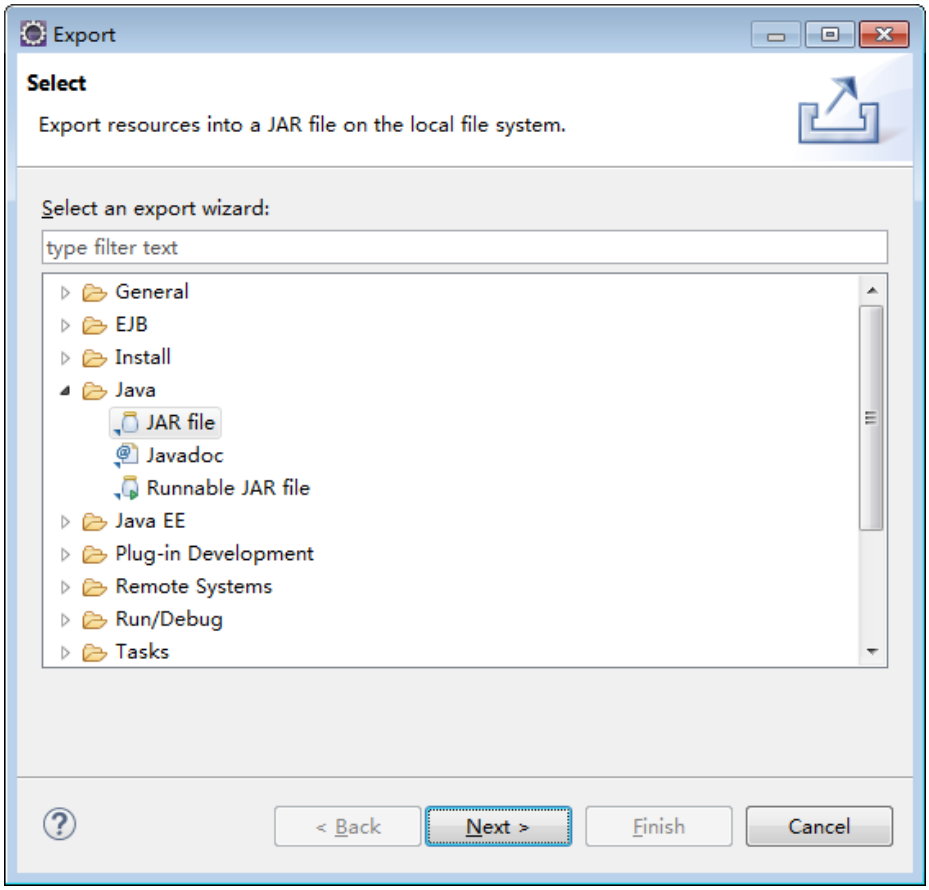
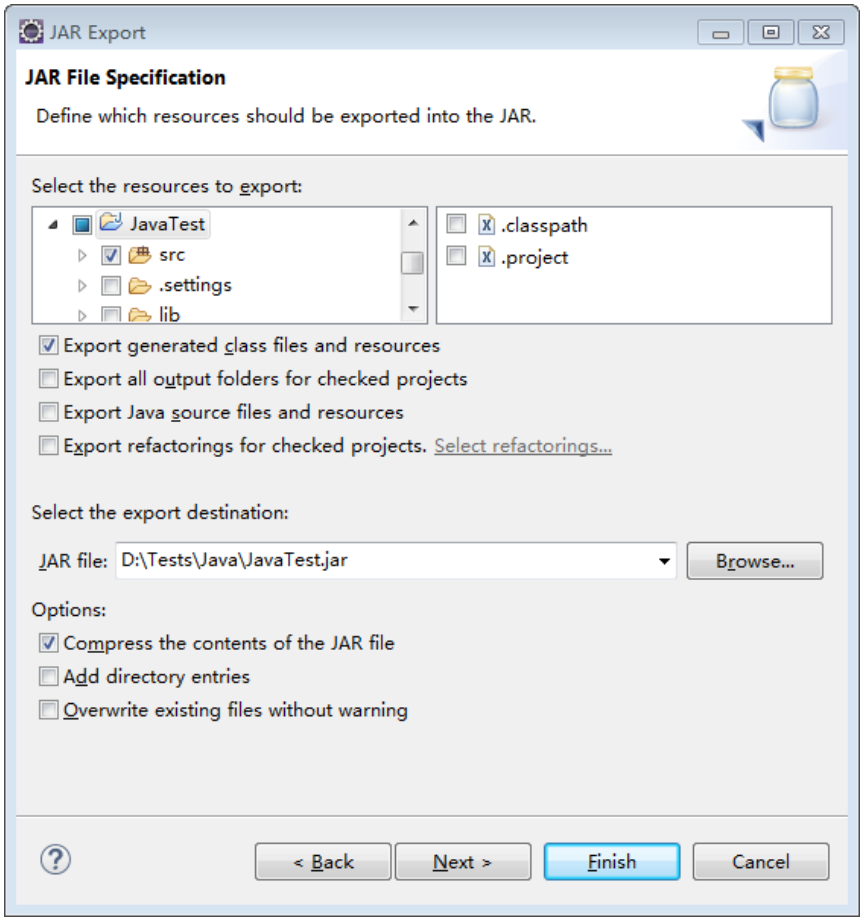
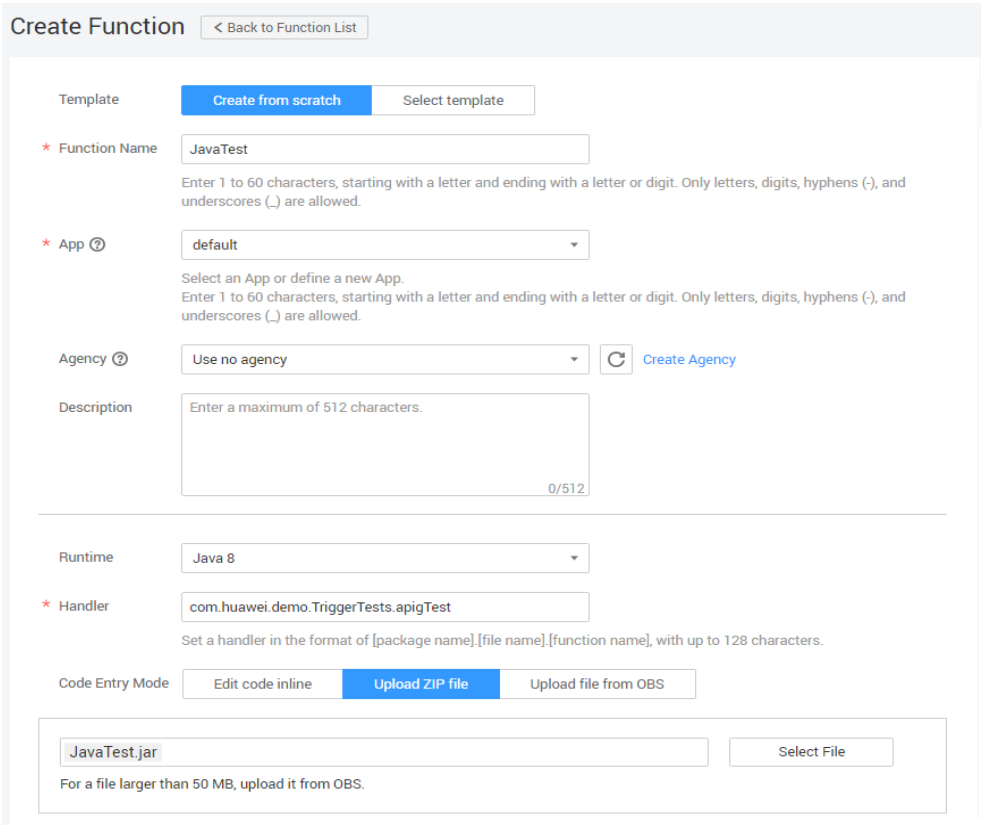


Figure 3-17 Specifying the destination path



Step 4 Log in to the FunctionGraph console, create a Java function, and upload the code package, as shown in **Figure 3-18**.

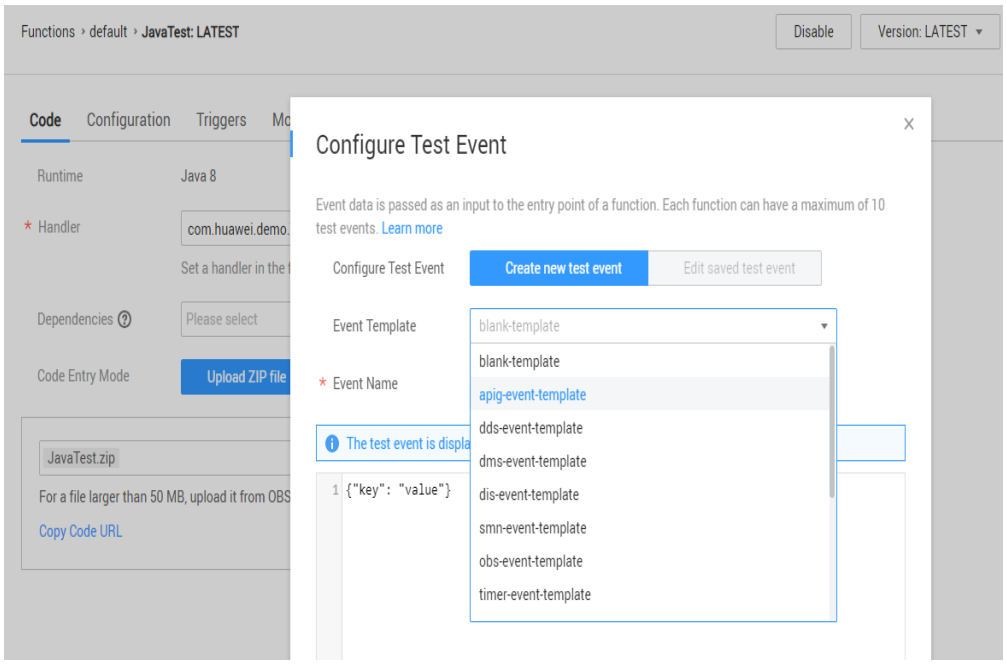
Figure 3-18 Creating a function



Step 5 Test the function.

1. Create a test event.
   Select **apig-event-template** and click **Save**.

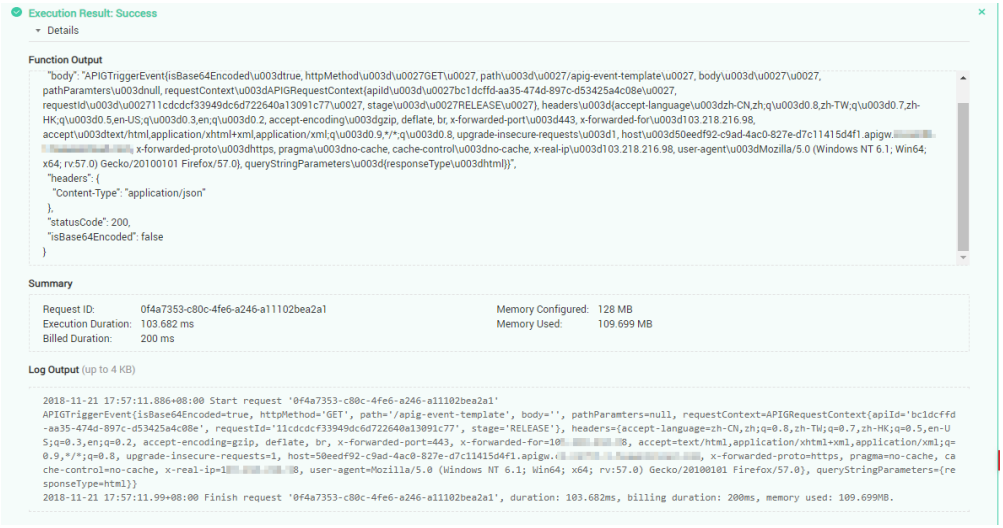
   Figure 3-19 Configuring a test event



2. Click **Test**.

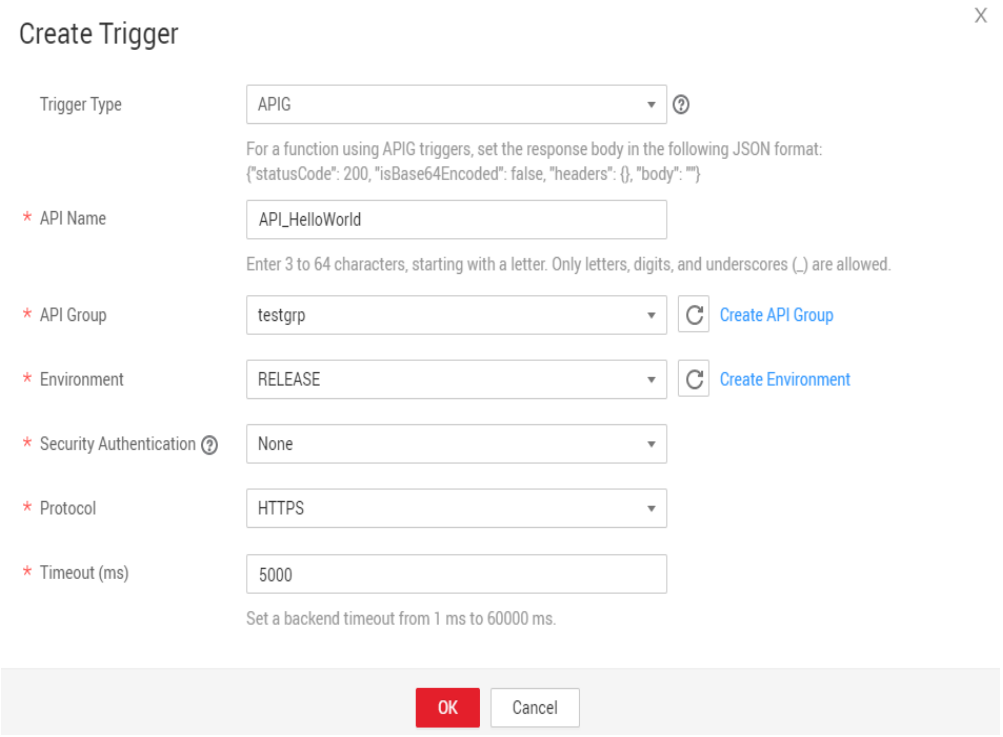
   The function execution result consists of three parts: function output (returned by **callback**), summary, and logs (output by using the **console.log** or **getLogger()** method), as shown in **Figure 3-20**.

   Figure 3-20 Execution result



3. Create an APIG trigger, as shown in **Figure 3-21**.

   Figure 3-21 Creating an APIG trigger



4. Access the API URL, as shown in **Figure 3-22**.

Figure 3-22 Accessing the API URL

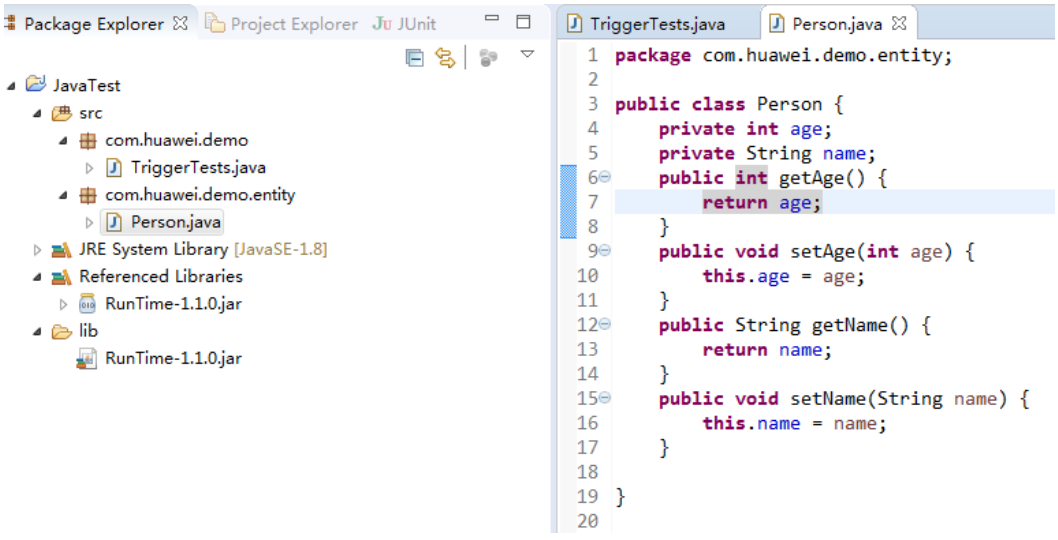
Change the handler to **com.huawei.demo.TriggerTests.smnTest** and change the event template to **smn-event-template**. **Figure 3-23** shows the execution result.

Figure 3-23 Execution result

Step 6 Use a user-defined parameter in the Java code.

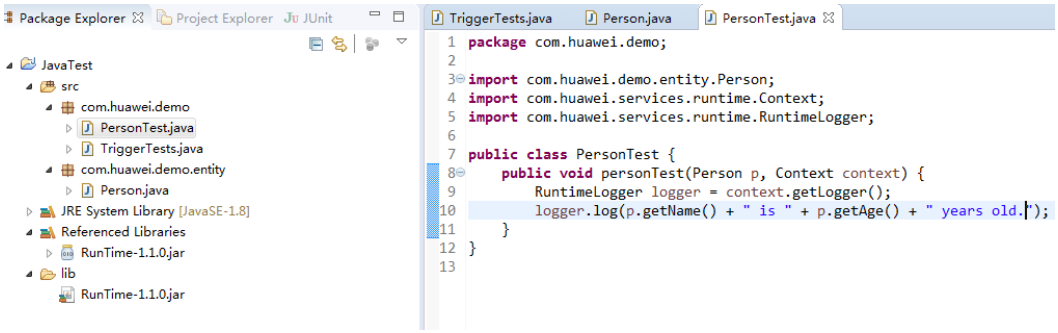
Create a Person class in the project, as shown in **Figure 3-24**.

Figure 3-24 Creating a Person class



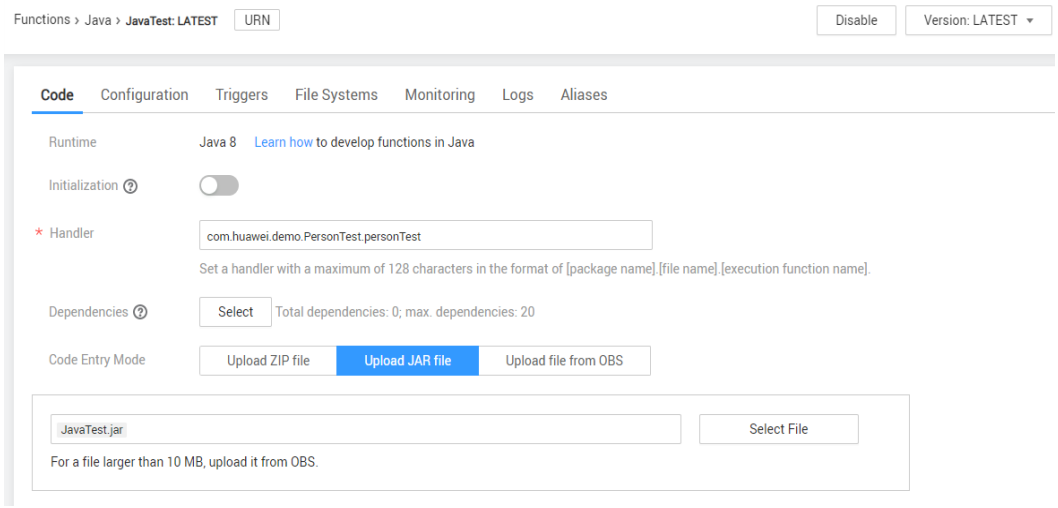
Create a class named **PersonTest.java**, and add a handler function to the class, as shown in **Figure 3-25**.

Figure 3-25 Creating a class named PersonTest.java



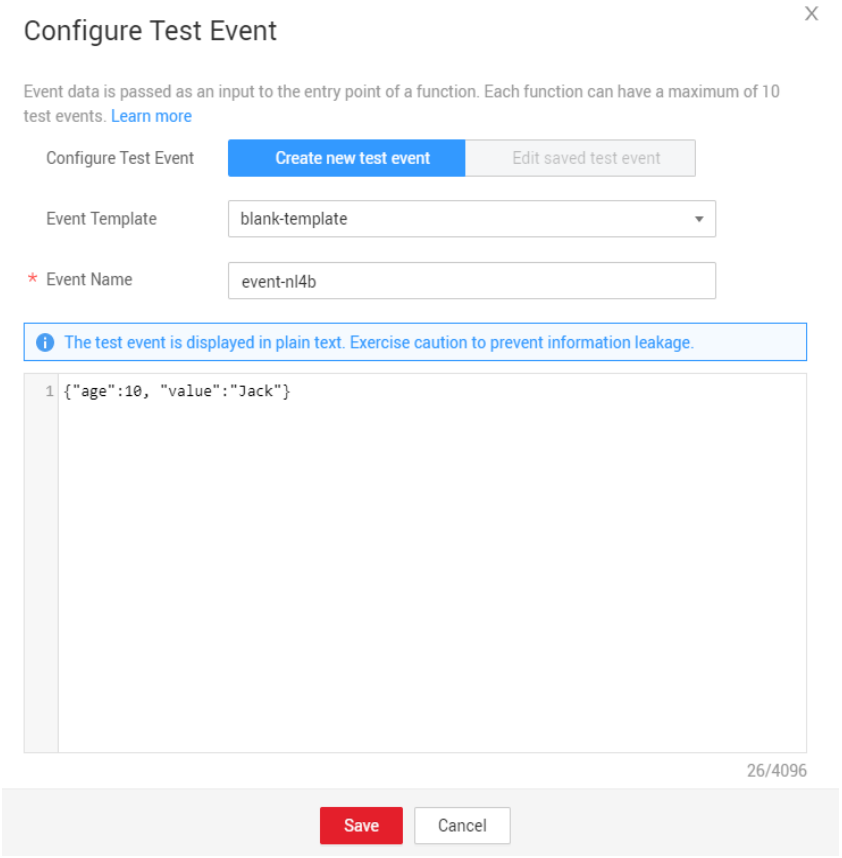
After exporting the new package, upload it to the function, change the function handler to **com.huawei.demo.PersonTest.personTest**, and click **Save**, as shown in **Figure 3-26**.

Figure 3-26 Uploading the code package



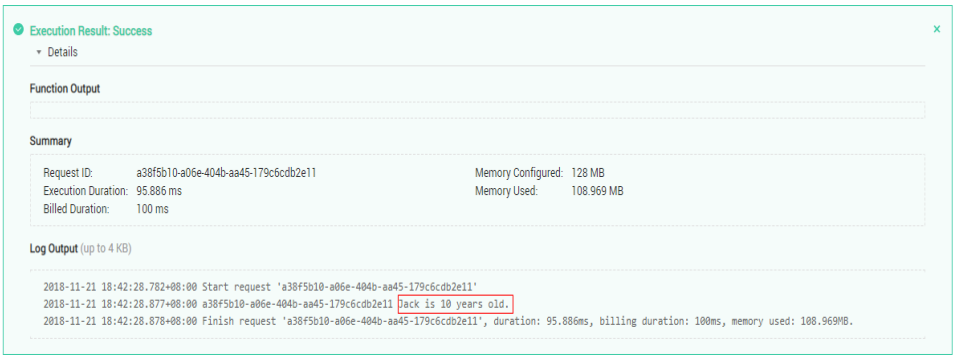
In the **Configure Test Event** dialog box, select **blank-template**, enter a test event, and click **Save**, as shown in **Figure 3-27**.

Figure 3-27 Configuring a test event



Click **Test**. **Figure 3-28** shows the execution result.

Figure 3-28 Execution result



----End

Execution Result

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

Table 3-26 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.

Example of the function output when a function fails to be executed:

```
{
  "errorMessage": "Exception: package.index",
  "stackTrace": []
}
```

Example of the log output when a function fails to be executed:

```
2020-07-27 15:16:55.819+08:00 Start invoke request '249d3490-d586-447e-92b7-945485126528', version: latest
2020-07-27 15:16:56.011+08:00 {
  "errorMessage": "Exception: package.index",
  "stackTrace": []
}
2020-07-27 15:16:56.114+08:00 Finish invoke request '249d3490-d586-447e-92b7-945485126528', duration: 294.518ms, billing duration: 300ms, memory used: 74.262MB.
```

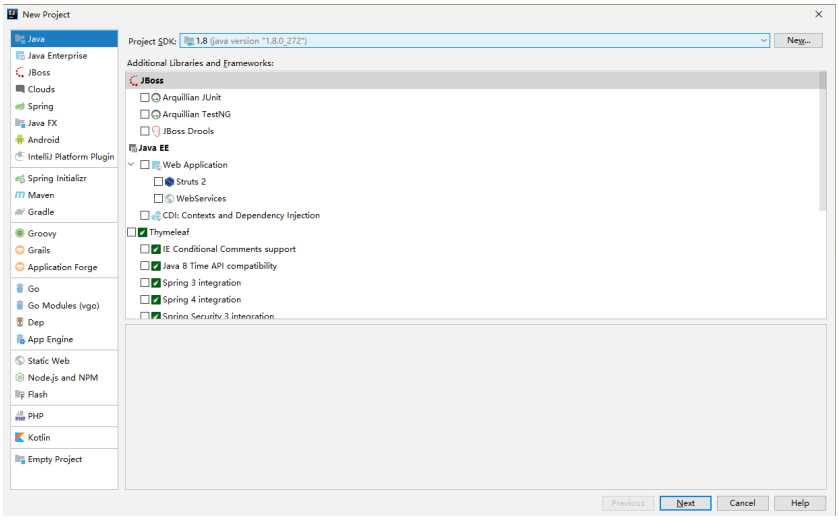
3.3.2 Developing Functions in Java (Using an IDEA Java Project)

Perform the following procedure to develop a Java function:

Step 1 Create a function project.

1. Configure the IDEA and create a Java project named **JavaTest**, as shown in **Figure 3-29**.

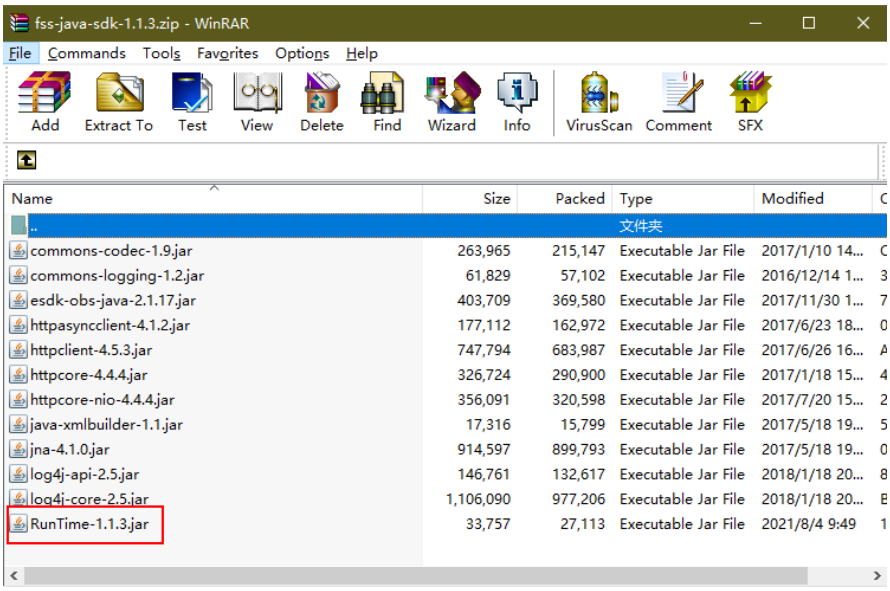
Figure 3-29 Creating a project



2. Add dependencies to the project.

   Download the **Java SDK** to a local development environment, and decompress the SDK package, as shown in **Figure 3-30**.

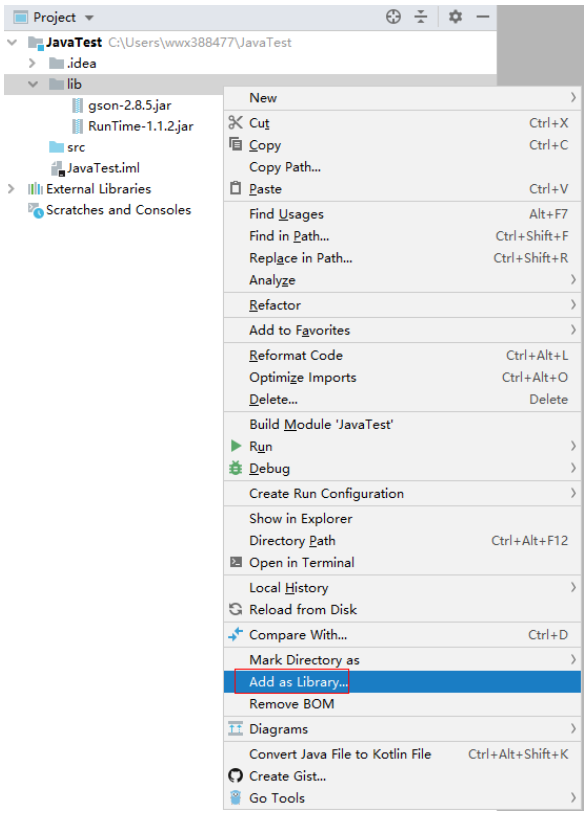
Figure 3-30 Decompressing the downloaded SDK



3. Configure dependencies.

   Create a folder named **lib** in the project directory, copy the **RunTime-1.1.3.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 3-31**.

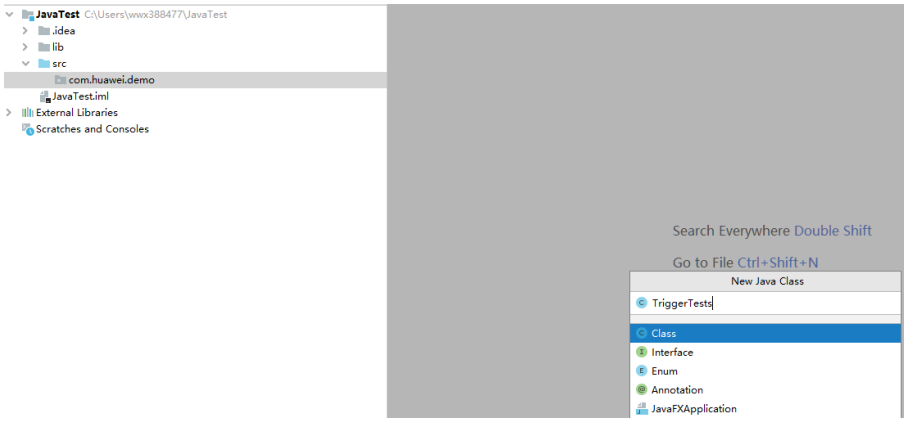
Figure 3-31 Configuring dependencies



Step 2 Create a function.

1. Create a package named **com.huawei.demo**, and then create a class named **TriggerTests** under the package, as shown in **Figure 3-32**.

Figure 3-32 Creating the TriggerTests class



2. Define the function handler in **TriggerTests.java**, as shown in **Figure 3-33**.

```
package com.huawei.demo;

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

import com.huawei.services.runtime.Context;
import com.huawei.services.runtime.entity.apig.APIGTriggerEvent;
import com.huawei.services.runtime.entity.apig.APIGTriggerResponse;
import com.huawei.services.runtime.entity.dis.DISTriggerEvent;
```

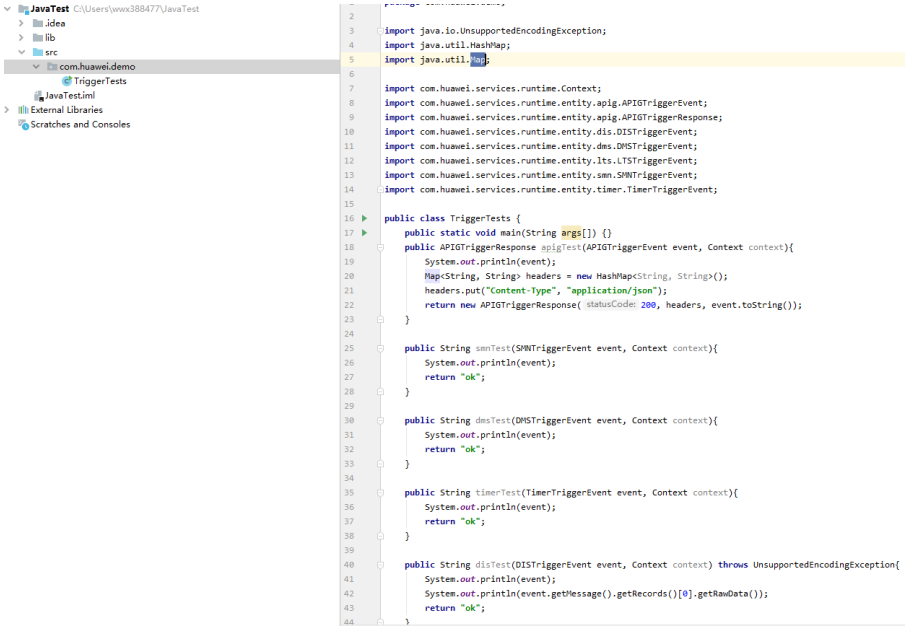


```
import com.huawei.services.runtime.entity.dms.DMSTriggerEvent;
import com.huawei.services.runtime.entity.lts.LTSTriggerEvent;
import com.huawei.services.runtime.entity.smn.SMNTriggerEvent;
import com.huawei.services.runtime.entity.timer.TimerTriggerEvent;

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

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

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

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

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

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

NOTE

A common Java project needs to be compiled using artifacts, and a main function needs to be defined.

Figure 3-33 Defining the function handler



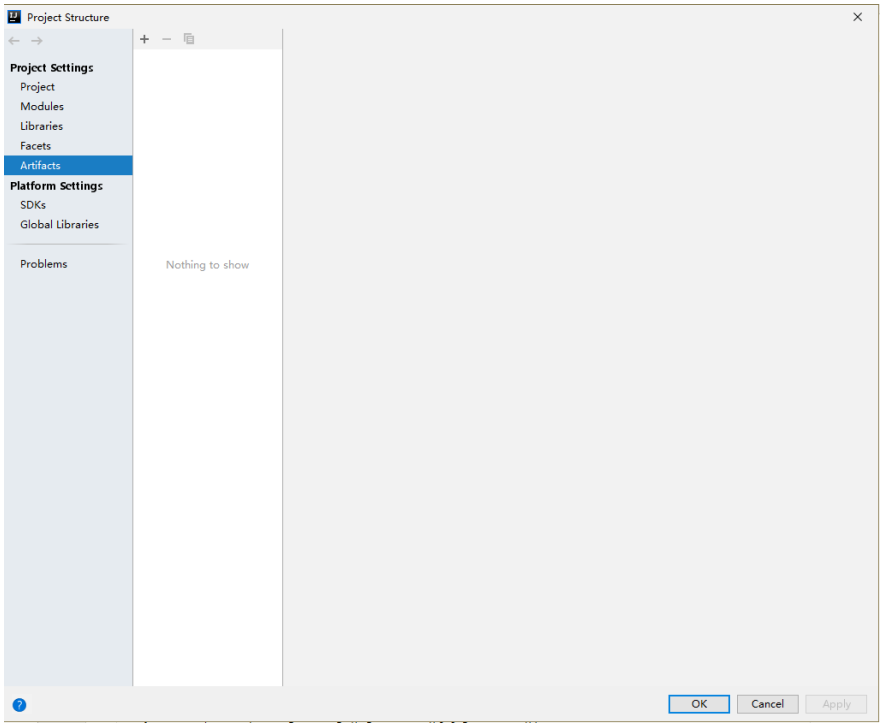
NOTE

The example code contains multiple handler functions, which use different types of triggers.

Step 3 Pack the project file.

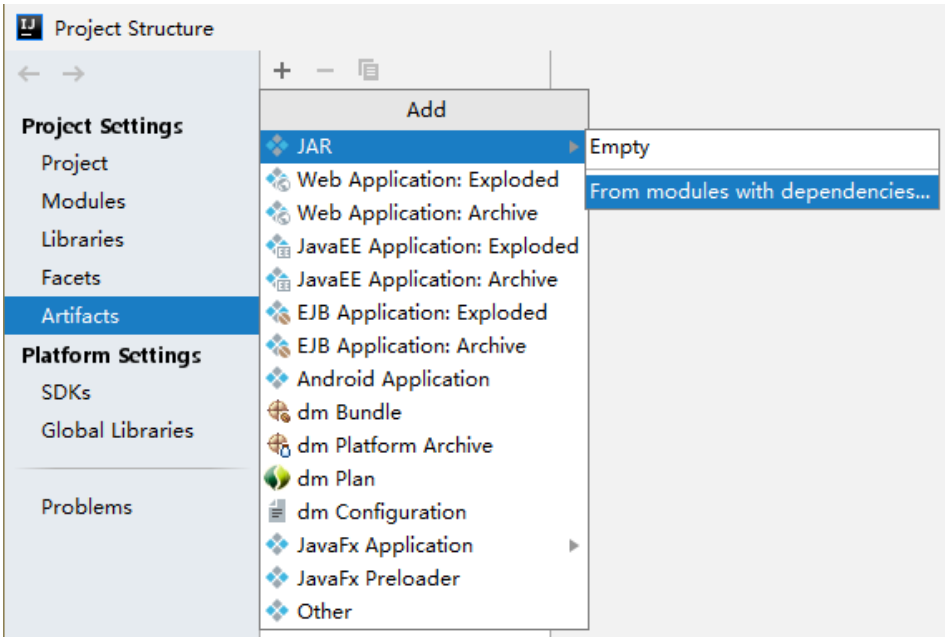
1. Choose **File** > **Project Structure**. The **Project Structure** page is displayed, as shown in **Figure 3-34**.

Figure 3-34 Going to the Project Structure page



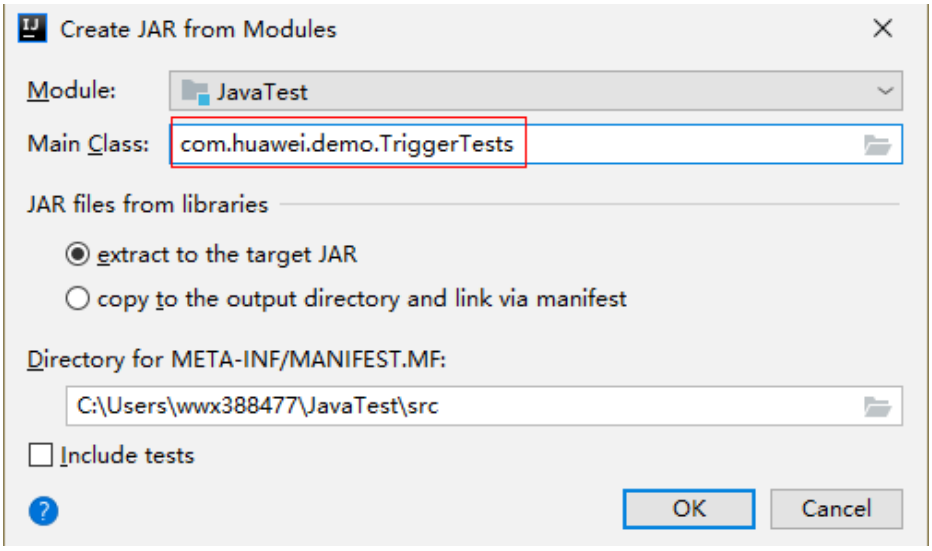
2. Choose **Artifacts** and click **+** to add artifacts, as shown in **Figure 3-35**.

Figure 3-35 Adding artifacts



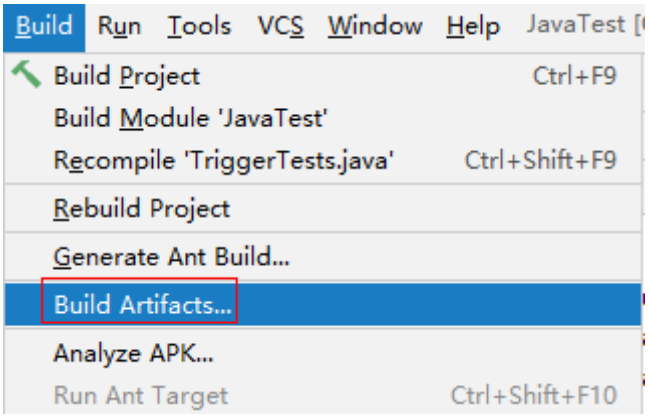
3. Add a main class, as shown in **Figure 3-36**.

Figure 3-36 Adding a main class



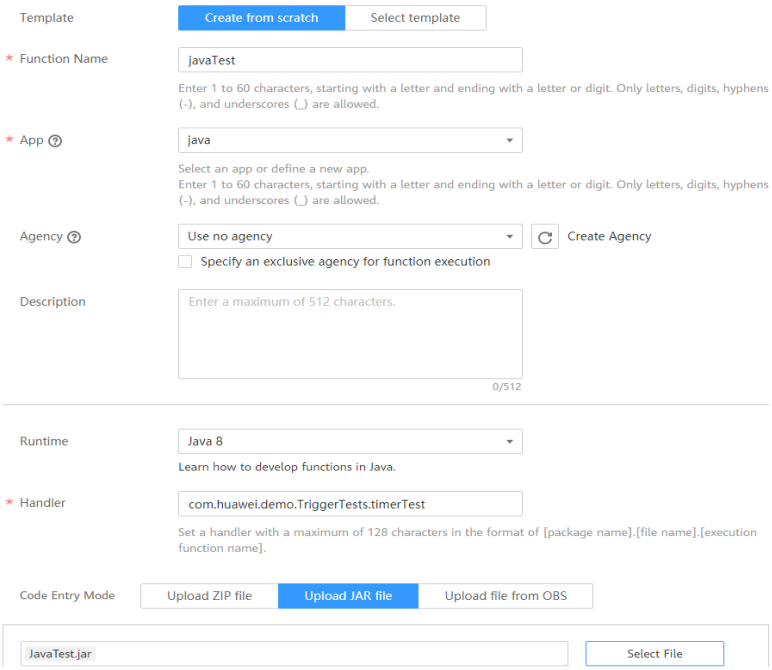
4. Choose **Build** > **Build Artifacts** to compile the JAR file, as shown in **Figure 3-37**.

Figure 3-37 Choosing Build Artifacts to compile the JAR file



Step 4 Log in to the FunctionGraph console, create a Java function, and upload the JAR file, as shown in **Figure 3-38**.

Figure 3-38 Creating a function

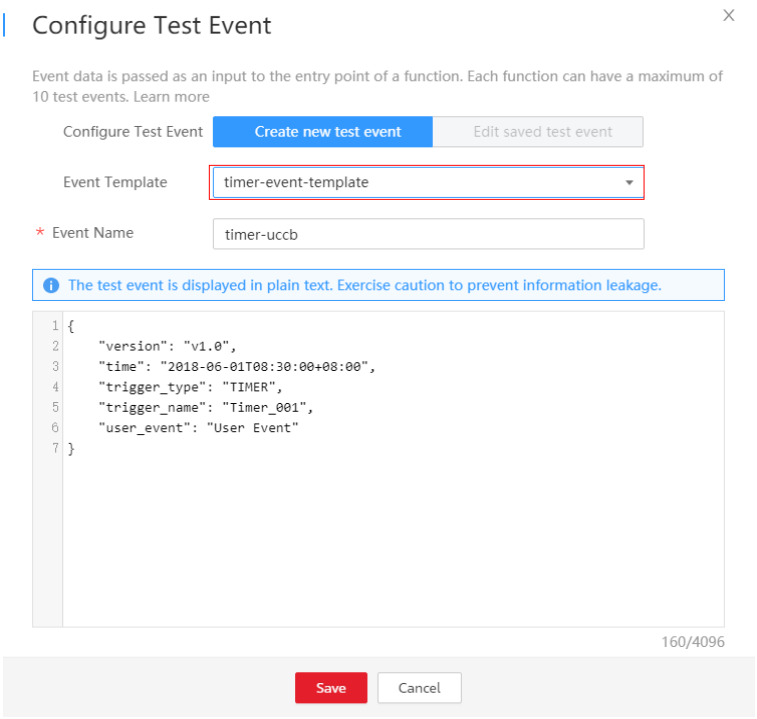


Step 5 Test the function.

1. Create a test event.

   Select **timer-event-template** from the **Event Template** drop-down list and click **Save**, as shown in **Figure 3-39**.

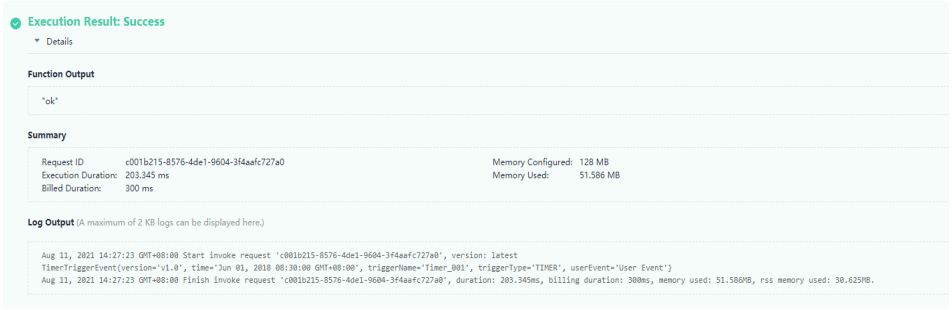
Figure 3-39 Configuring a test event



2. Click **Test**.

   The function execution result consists of three parts: function output (returned by **callback**), summary, and log (output by using the **console.log** or **getLogger()** method), as shown in **Figure 3-40**.

Figure 3-40 Execution result



----End

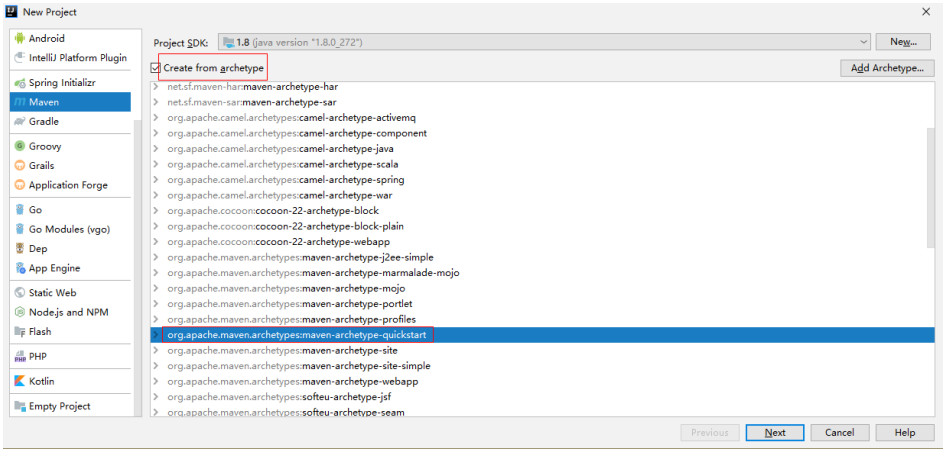
3.3.3 Developing Functions in Java (Using an IDEA Maven Project)

Perform the following procedure to develop a Java function:

Step 1 Create a function project.

1. Configure the IDEA and create a Maven project, as shown in **Figure 3-41**.

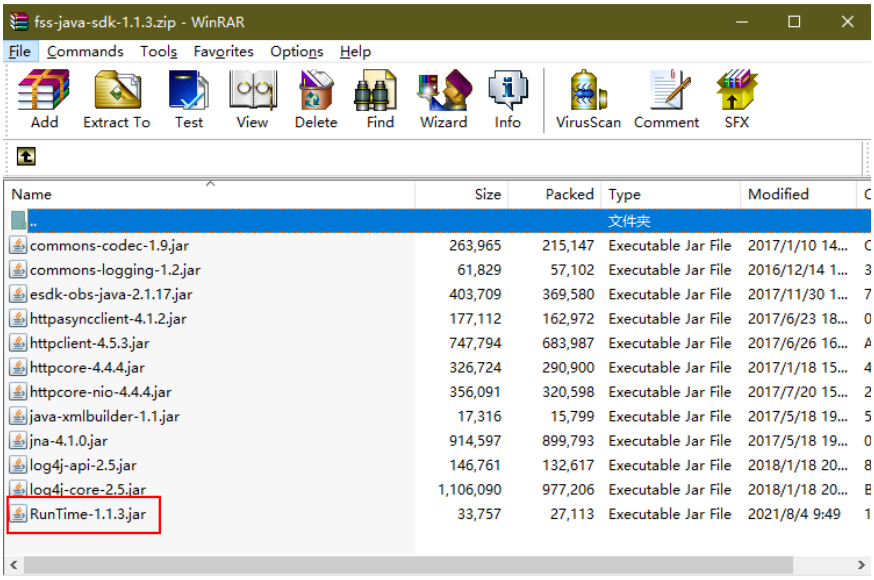
Figure 3-41 Creating a project



2. Add dependencies to the project.

   Download the **Java SDK** to a local development environment, and decompress the SDK package, as shown in **Figure 3-42**.

Figure 3-42 Decompressing the downloaded SDK



3. Copy **RunTime-1.1.3.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-1.1.3.jar -DgroupId=RunTime -DartifactId=RunTime -Dversion=1.1.3 -Dpackaging=jar

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

```
<dependency>
<groupId>Runtime</groupId>
<artifactId>Runtime</artifactId>
<version>1.1.3</version>
</dependency>
```

5. 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>
```

6. Add a plug-in to **pom.xml** to pack the code and dependencies together.

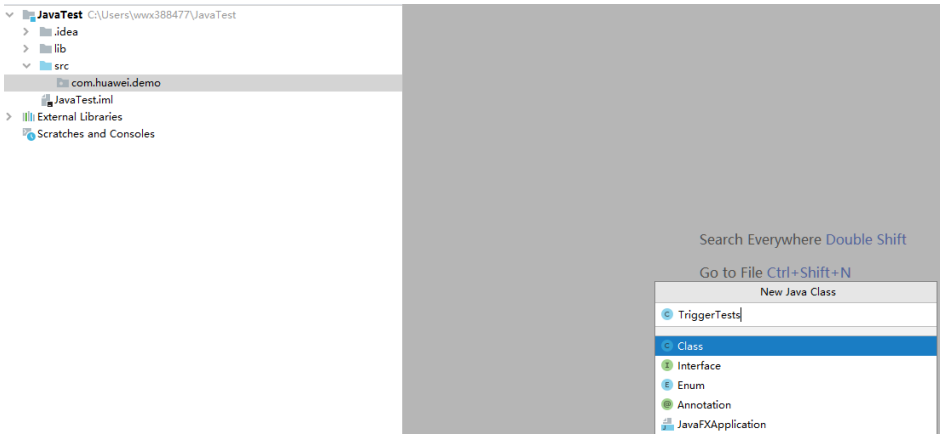
```
<plugin>
<artifactId> maven-assembly-plugin </artifactId>
<configuration>
<descriptorRefs>
<descriptorRef>jar-with-dependencies</descriptorRef>
</descriptorRefs>
<archive>
<manifest>
<mainClass>com.huawei.demo.TriggerTests</mainClass>
</manifest>
</archive>
<finalName>${project.name}</finalName>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

Replace *mainClass* with the actual class.

Step 2 Create a function.

1. Create a package named **com.huawei.demo**, and then create a class named **TriggerTests** under the package, as shown in **Figure 3-43**.

Figure 3-43 Creating the TriggerTests class



2. 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;
```



```
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";
    }
}
```

Step 3 Compile and pack the project file.

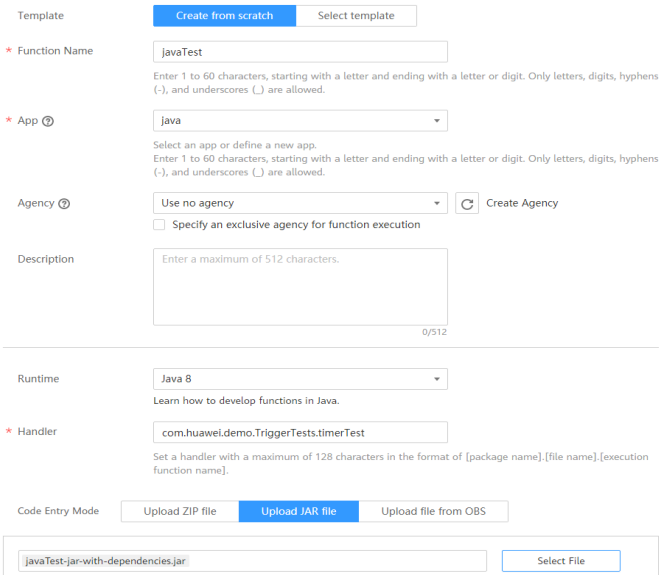
Run the following command to compile and pack the project file:

mvn package assembly:single

After compilation is complete, the **javaTest-jar-with-dependencies.jar** file is generated in the target directory.

Step 4 Log in to the FunctionGraph console, create a Java function, and upload the JAR file, as shown in **Figure 3-44**.

Figure 3-44 Creating a function

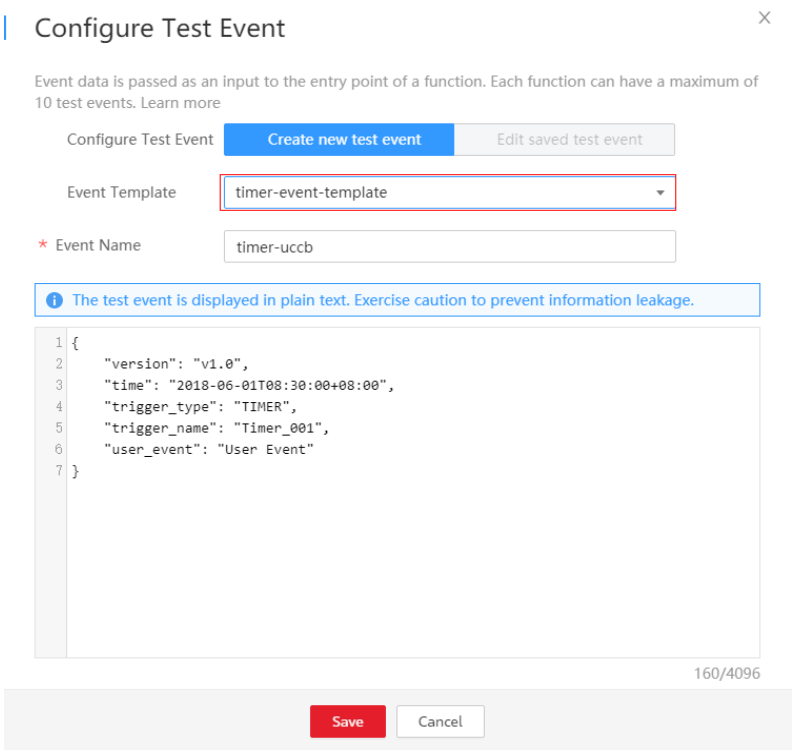


Step 5 Test the function.

1. Create a test event.

   Select **timer-event-template** from the **Event Template** drop-down list and click **Save**, as shown in **Figure 3-45**.

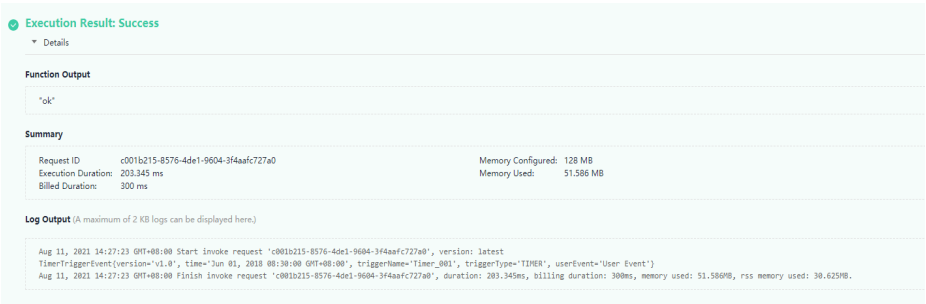
   Figure 3-45 Configuring a test event



2. Click **Test**.

The function execution result consists of three parts: function output (returned by **callback**), summary, and log (output by using the **console.log** or **getLogger()** method), as shown in **Figure 3-46**.

Figure 3-46 Execution result



----End

3.4 Developing Functions in Go 1.8.3

Function Syntax

NOTE

You are advised to use Go 1.x.

Syntax for creating a handler function in Go:

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

- **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, and must start with an uppercase letter.
- **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**.

SDK APIs

- Context APIs

  The Go SDK provides multiple context APIs and a logging API. The download link of the Go SDK is provided in **Table 1-1**. **Table 3-27** describes the context APIs provided by FunctionGraph.

Table 3-27 Context methods

Method	Description
GetRequestID() string	Obtains a request ID.
GetRemainingTimeInMilliSeconds()	Obtains the remaining running time of a function.

Method	Description
GetAccessKey() string	Obtains the AK (valid for 24 hours) of an agency. If you use this method, you need to configure an agency for the function.
GetSecretKey() string	Obtains the SK (valid for 24 hours) of an agency. If you use this method, you need to configure an agency for the function.
GetUserData(key string) string	Uses keys to obtain the values passed by environment variables.
GetFunctionName() string	Obtains the name of a function.
GetRunningTimeInSeconds() int	Obtains the timeout of a function.
GetVersion() string	Obtains the version of a function.
GetMemorySize() int	Obtains the allocated memory.
GetCPUNumber() int	Number of CPU millicores used by the function (1 core = 1000 millicores). The value of this field is proportional to that of **MemorySize**. By default, 100 CPU millicores are required for 128 MB memory. The number of CPU millicores is calculated as follows: Memory/128 x 100 + 200 (basic CPU millicores).
GetProjectID() string	Obtains a project ID.
GetPackage() string	Obtains a function group, that is, an app.
GetToken() string	Obtains the token (valid for 24 hours) of an agency. If you use this method, you need to configure an agency for the function.
GetLogger() context.RuntimeLogger	Obtains the **logger** method provided by the context. By default, information such as the time and request ID is output.

WARNING

Results returned by using the **GetToken()**, **GetAccessKey()**, and **GetSecretKey()** methods contain sensitive information. Exercise caution when using these methods.

- Logging API

  Table 3-28 describes the logging API provided in the Go SDK.

  Table 3-28 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**.

Developing a Go Function

Step 1 Create a Go project.

1. Log in to a Linux server on which the Go 1.8.3 SDK and required running environment have been installed. (Currently, Ubuntu 14.04, Ubuntu 16.04, SUSE 11.3, SUSE 12.0, and SUSE 12.1 are supported.) Then create the **/home/fssgo/src/go-runtime** directory, and decompress the **Go SDK** under this directory. Specifically, run the following two commands:

   $ mkdir -p /home/fssgo/src/go-runtime

   $ unzip fss-go-sdk.zip -d /home/fssgo/src/go-runtime

2. Create a function file under the **/home/fssgo/src/go-runtime** 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"
    "encoding/json"
    "go-runtime/go-api/context"
)

type Greeting struct {
```



```
    Name   string  `json:"name"`
}

// Handler function
func Handler(payload []byte, ctx context.RuntimeContext) (interface{}, error) {
    logger := ctx.GetLogger()

    var greeting Greeting
    err := json.Unmarshal(payload, &greeting)
    if err != nil {
        return "", fmt.Errorf("invalid request payload")
    }

    logger.Logf("hello %s.", greeting.Name)

    logger.Logf("RequestId: %s", ctx.GetRequestID())
    logger.Logf("FunctionName: %s", ctx.GetFunctionName())
    logger.Logf("Version: %s", ctx.GetVersion())
    return fmt.Sprintf("hello %s", greeting.Name), nil
}
```

NOTICE

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**.

Step 2 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/src/go-runtime

   $ go build --buildmode=plugin -o testplugin.so test.go

   After compilation, a file named **testplugin.so** is generated under the current directory.

3. Package the file.

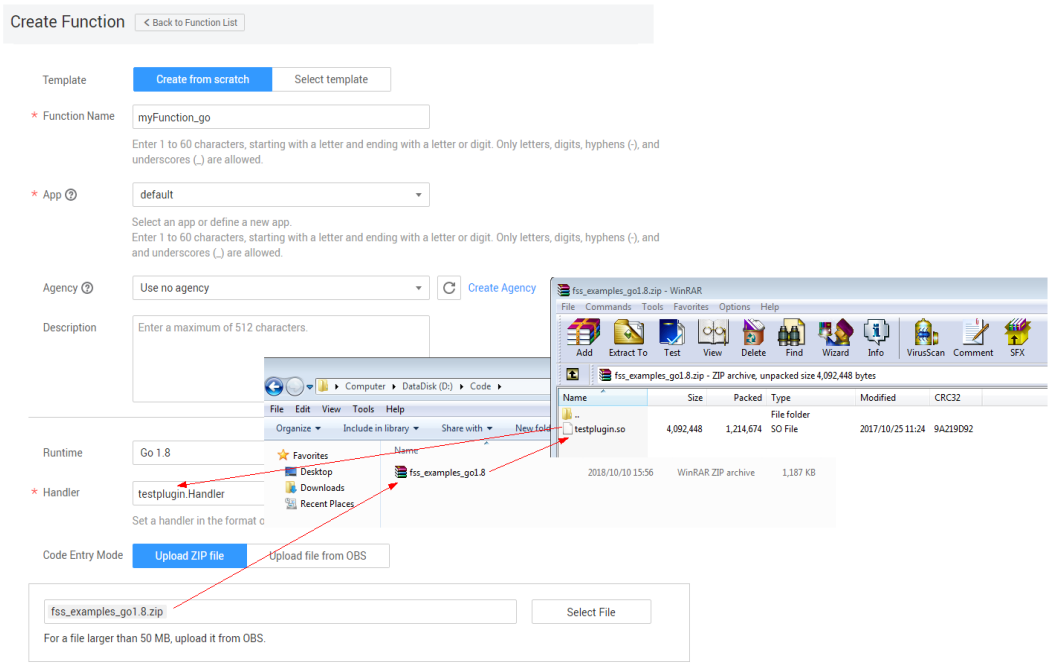
   $ zip fss_examples_go1.8.zip testplugin.so

   After the command is executed, a file named **fss_examples_go1.8.zip** is generated under the current directory.

Step 3 Create a function.

Log in to the FunctionGraph console, create a Go function, and upload the **fss_examples_go1.8.zip** file, as shown in **Figure 3-47**.

Figure 3-47 Uploading the code package



NOTE

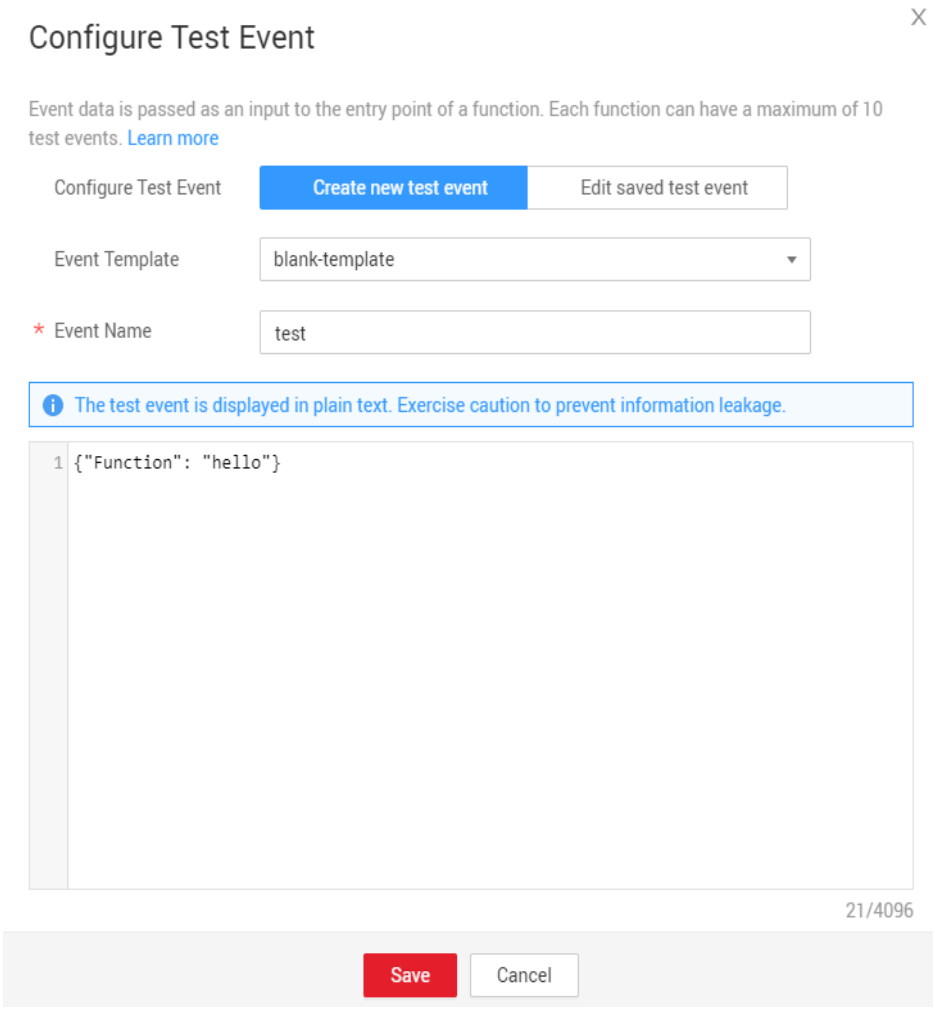
If you edit code in Go, package the compiled file into a ZIP file, and ensure that the name of the dynamic library file is consistent with the handler plugin name. For example, if the name of the dynamic library file is **testplugin.so**, set the handler plugin name to **testplugin.Handler**. The handler must be consistent with that defined in **Step 1**.

Step 4 Test the function.

1. Create a test event.

   On the function details page that is displayed, choose **Select test event** > **Configure test event**. Configure the test event information, as shown in **Figure 3-48**, and then click **Save**.

Figure 3-48 Configuring a test event

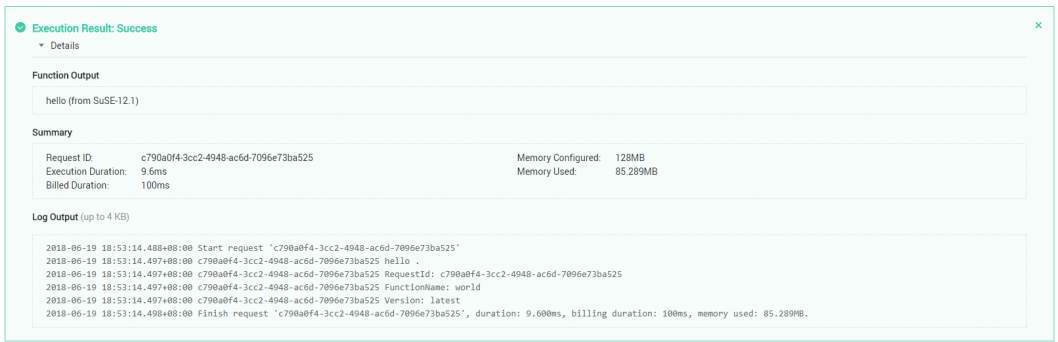


2. On the function details page, select the configured test event, and click **Test**.

Step 5 View the function execution result.

The function execution result consists of three parts: function output (returned by **callback**), summary, and logs (output by using the **console.log** or **getLogger()** method), as shown in **Figure 3-49**.

Figure 3-49 Execution result



----End

Execution Result

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

Table 3-29 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.

Example of the function output when a function fails to be executed:

```
{
    "errorMessage": "Function plugin 'plugin' not found(stat ${RUNTIME_CODE_ROOT}/plugin.so: no such file or directory)",
    "errorType": "FunctionLoadError"
}
```

Example of the log output when a function fails to be executed:

```
2020-07-27 15:20:53.292+08:00 Start invoke request '51a44941-577b-4e1f-b2f6-2f2c3ca2f5f7', version: latest
2020-07-27 15:20:53.292+08:00 51a44941-577b-4e1f-b2f6-2f2c3ca2f5f7 Function plugin 'plugin' not found(stat ${RUNTIME_CODE_ROOT}/plugin.so: no such file or directory)
2020-07-27 15:20:53.293+08:00 Finish invoke request '51a44941-577b-4e1f-b2f6-2f2c3ca2f5f7'(invoke Failed), duration: 1.447ms, billing duration: 100ms, memory used: 58.723MB
```

3.5 Developing Functions in Go 1.x

Function Syntax

Syntax for creating a handler function in Go:

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**.

SDK APIs

The Go SDK provides event, context, and logging APIs. **Go SDK**

- Event APIs

  Event structure definitions are added to the Go SDK. Currently, CTS, Kafka, DIS, DDS, SMN, LTS, 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 3-30 APIGTriggerEvent fields

Field Name	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 3-31 APIGRequestContext fields

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

ii. APIGTriggerResponse fields

Table 3-32 APIGTriggerResponse fields

Field Name	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 3-33 DISTriggerEvent fields

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

Table 3-34 DISMessage fields

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

Table 3-35 DISRecord fields

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

c. **Kafka Trigger Field Description**

Table 3-36 KAFKATriggerEvent fields

Field Name	Description
InstanceId	Instance ID.
Records	Message records (**Table 3-37**).
TriggerType	Trigger type (Kafka).
Region	region
EventTime	Time when an event occurred (seconds).
EventVersion	Event version.

Table 3-37 KAFKARecord parameters

Field Name	Description
Messages	DMS message body.
TopicId	Topic ID.

d. **SMN Trigger Field Description**

Table 3-38 SMNTriggerEvent fields

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

Table 3-39 SMNRecord fields

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

Table 3-40 SMNBody fields

Field Name	Description
TopicUrn	Topic URN.
TimeStamp	Message timestamp.
MessageAtrributes	Message attribute set.
Message	Message body.
Type	Message format.
MessageId	Message ID.
Subject	Message topic.

e. **Timer Trigger Field Description**

Table 3-41 TimerTriggerEvent fields

Field Name	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.

f. **LTS Trigger Field Description**

Table 3-42 LTSTriggerEvent fields

Field Name	Description
Lts	LTS message (**LTSBody structure**).

Table 3-43 LTSBody fields

Field Name	Description
Data	Original LTS message.

NOTE

LTSBody provides the GetRawData() function to return the message decoded using Base64.

g. **CTS Trigger Field Description**

Table 3-44 CTSTriggerEvent fields

Field Name	Description
CTS	CTS message body (**Table 3-45**).

Table 3-45 CTS structure fields

Field Name	Description
Time	Time when a trace occurred.
User	Information about the user who triggered the trace (**Table 3-46**).
Request	Request content of the trace.
Response	Response content of the trace.
Code	Response code.
ServiceType	Name of the service involved in the trace.
ResourceType	Type of the resource involved in the trace.
ResourceName	Name of the resource involved in the trace.
ResourceId	ID of the resource involved in the trace.

Field Name	Description
TraceName	Event name.
TraceType	Trace type, for example, **ConsoleAction**.
RecordTime	Time when CTS received the trace.
TraceId	Trace ID.
TraceStatus	Trace status.

Table 3-46 User fields

Field Name	Description
Name	Username (an account can create multiple users).
Id	User ID.
Domain	Account information (**Table 3-47**)

Table 3-47 Domain fields

Field Name	Description
Name	Account name.
Id	Account ID.

8. DDS Trigger Field Description

Table 3-48 DDS event structure fields

Field Name	Description
Records	DDS record (**Table 3-49**).

Table 3-49 DDS record body fields

Field Name	Description
EventName	Event name.
EventSource	Event source.

Field Name	Description
Region	region
Dds	Document data.
EventSourceId	Event source ID.

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 3-50 describes the context APIs provided by FunctionGraph.

  Table 3-50 Context methods

Method	Description
GetRequestID( )	Obtains a request ID.
GetRemainingTimeInMilligetRun-ningTimeInSecondsSeconds ( )	Obtains the remaining running time of a function.
GetAccessKey( )	Obtains the AK (valid for 24 hours) of an agency. If you use this method, you need to configure an agency for the function.
GetSecretKey( )	Obtains the SK (valid for 24 hours) of an agency. If you 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.

Method	Description
GetCPUNumber( )	Number of CPU millicores used by the function (1 core = 1000 millicores). The value of this field is proportional to that of **MemorySize**. By default, 100 CPU millicores are required for 128 MB memory. The number of CPU millicores is calculated as follows: Memory/128 x 100 + 200 (basic CPU millicores).
GetProjectID( )	Obtains a project ID.
GetPackage( )	Obtains a function group, that is, an app.
GetToken( )	Obtains the token (valid for 24 hours) of 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.

WARNING

Results returned by using the **GetToken()**, **GetAccessKey()**, and **GetSecretKey()** methods contain sensitive information. Exercise caution when using these methods.

- Table 3-51 describes the logging API provided in the Go SDK.

Table 3-51 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**.

Developing a Go Function

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

- If the Go version (1.11.1 or later) supports go mod, perform the following steps to compile and package code:

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 marked in red).

```
module test

go 1.14

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 3-50**. 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)
}
```

NOTICE

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 uses the 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.

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, which is used as the function entry.
2. Package the code.

$ zip fss_examples_go1.x.zip handler

----End

- If the Go version (earlier than 1.11.1) does not support go mod, perform the following steps to compile and package code:

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)
}
```

NOTICE

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 uses the 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.

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

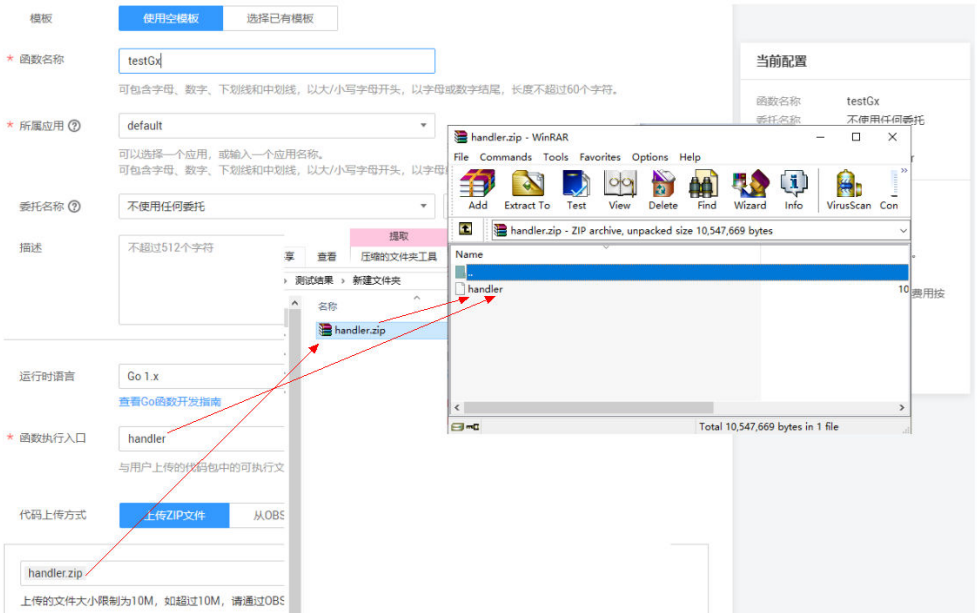
   The handler can be customized, which is used as the function entry.
3. Package the code.

   $ zip fss_examples_go1.x.zip handler

- Creating a function

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

Figure 3-50 Uploading the code package



NOTE

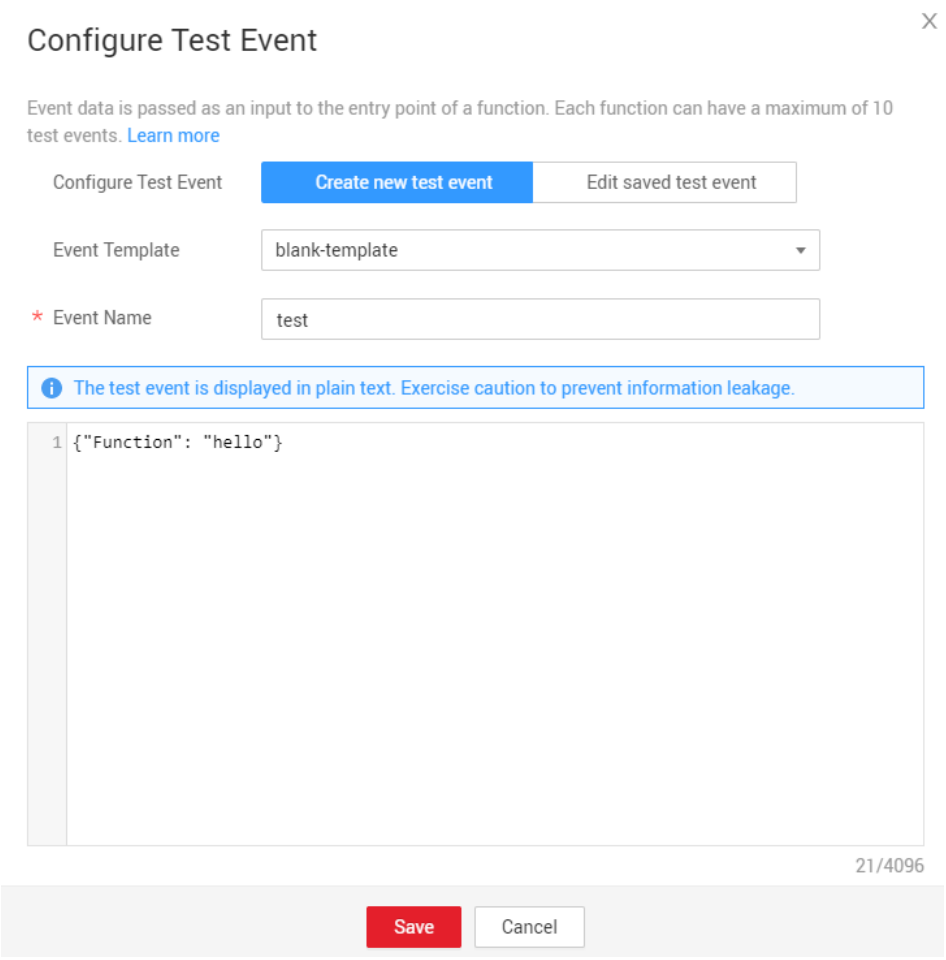
If you edit code in Go, package the compiled file into a ZIP file, and ensure that the name of the compiled file is consistent with the handler plugin name. For example, if the name of the binary file is **handler**, set the handler plugin name to **handler**. The handler must be consistent with that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

- Testing the function

  a. Create a test event.

  On the function details page that is displayed, choose **Select test event** > **Configure test event**. Configure the test event information, as shown in **Figure 3-51**, and then click **Save**.

Figure 3-51 Configuring a test event

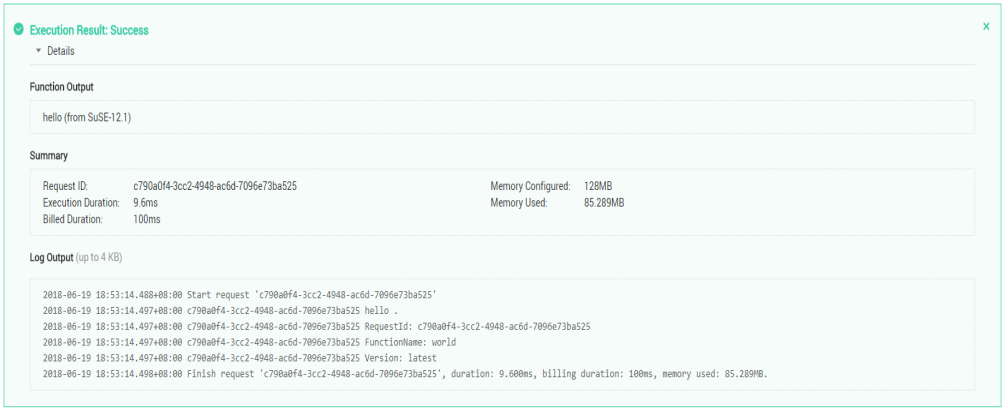


b. On the function details page, select the configured test event, and click **Test**.

- Executing the function

  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), as shown in **Figure 3-52**.

  Figure 3-52 Execution result



----End

Execution Result

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

Table 3-52 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.

Example of the function output when a function fails to be executed:

```
{   "errorMessage": "runtime error: invalid memory address or nil pointer dereference",
    "errorType": "panic",
    "stackTrace": [       "HandleRequest()",       "     go-runtime/main.go:15"   ]
}
```

Example of the log output when a function fails to be executed:

```
2020/08/12 09:17:07 GMT+08:00  Start invoke request 'e9d3c07f-583c-469b-866a-56a927576998', version: latest
2020/08/12 09:17:07 GMT+08:00  e9d3c07f-583c-469b-866a-56a927576998 payload:{"key": "value"}2020/08/12 09:17:07 GMT+08:00  Finish invoke request 'e9d3c07f-583c-469b-866a-56a927576998'(invoke Failed), duration: 39.413ms, billing duration: 100ms, memory used: 73.766MB
```

3.6 Developing Functions in C#

3.6.1 C# Function Development

Function Syntax

NOTE

You are advised to use .NET Core 3.1.

FunctionGraph supports C# (.NET Core 2.0), C# (.NET Core 2.1), and C# (.NET Core 3.1).

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.
- Event: event parameter defined for the 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
}
```

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 3-53 describes the provided context attributes.

Table 3-53 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	Allocated memory.
Int CpuNumber	Number of CPU millicores used by the function (1 core = 1000 millicores). The value of this field is proportional to that of **MemorySize**. By default, 100 CPU millicores are required for 128 MB memory. The number of CPU millicores is calculated as follows: Memory/128 x 100 + 200 (basic CPU millicores).
String Accesskey	Obtains the AK (valid for 24 hours) of an agency. If you use this method, you need to configure an agency for the function.
String Secretkey	Obtains the SK (valid for 24 hours) of an agency. If you use this method, you need to configure an agency for the function.
String Token	Obtains the token (valid for 24 hours) of 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 3-54 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)");

Method	Description

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);

Developing a C# Function

NOTE

If you use sample package **fss_example_csharp2.0**, skip **Step 1** and **Step 2**, go to **Step 3**, and modify the function handler to **MyCsharpPro::src.Program::myFunc**.

The following describes how to create a C# (.NET Core 2.0) function in Linux.

Step 1 Create a C# project.

Log in to a Linux server on which the .NET SDK and required running environment have been installed. Create the **/home/fsscsharp/src** directory, and decompress the **.dll package** to this directory, as shown in **Figure 3-53**.

This document uses the DLL function of the fssCsharp2.0-1.0.1 version as an example. The DLL functions of different versions are the same.

Figure 3-53 Decompressing the .dll package

```
root@SZX1000371099:/home/fsscsharp/src# pwd
/home/fsscsharp/src
root@SZX1000371099:/home/fsscsharp/src# ll
total 16
drwxr-xr-x 2 root root 4096 Oct 25 17:01 ./
drwxr-xr-x 6 root root 4096 Oct 25 09:48 ../
-rw-r--r-- 1 root root 5632 Oct 25 15:42 HC.Serverless.Function.Common.dll
root@SZX1000371099:/home/fsscsharp/src#
```

Run the **dotnet --info** command to check whether the .NET environment has been installed. The command output is as follows:

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

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

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

Microsoft .NET Core Shared Framework Host

  Version  : 2.0.9
  Build    : 1632fa1589b0eee3277a8841ce1770e554ece037
```

Run the following command to create and initialize a console application project:

"dotnet new console -n project_name"

Example command:

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

Create a handler function in the **Program.cs** code file in the **/home/fsscsharp/src/MyCsharpPro** directory. The input is the body of a client request, and the context is a runtime context object provided by FunctionGraph. For more information about context objects, see the context API description. The code is as follows:

```
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("Access Key:       {0}", context.AccessKey);
                sw.WriteLine("Secret Key:       {0}", context.SecretKey);
                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());
        }
    }
}
```

Step 2 Compile the C# project.

Manually add the **dll** reference (relative path of **dll** in HinPath) provided by FunctionGraph to the project configuration file **MyCsharpPro.csproj**. The code is as follows:

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



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

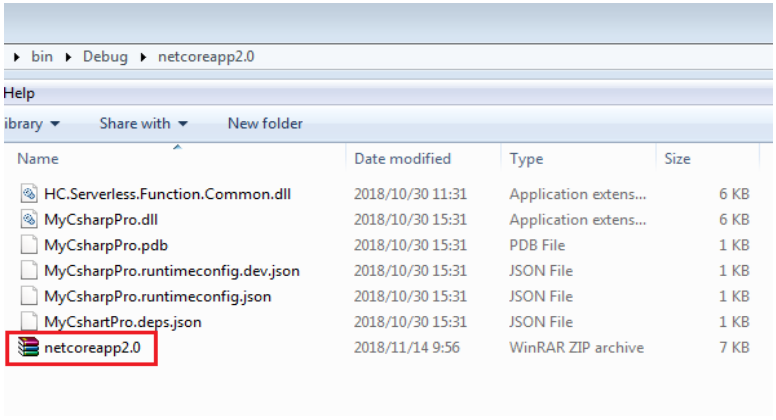
Run the **dotnet build** command to compile the project. The command output is as follows:

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

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

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

Time Elapsed 00:00:01.47
```

Step 3 Deploy the C# project to FunctionGraph.

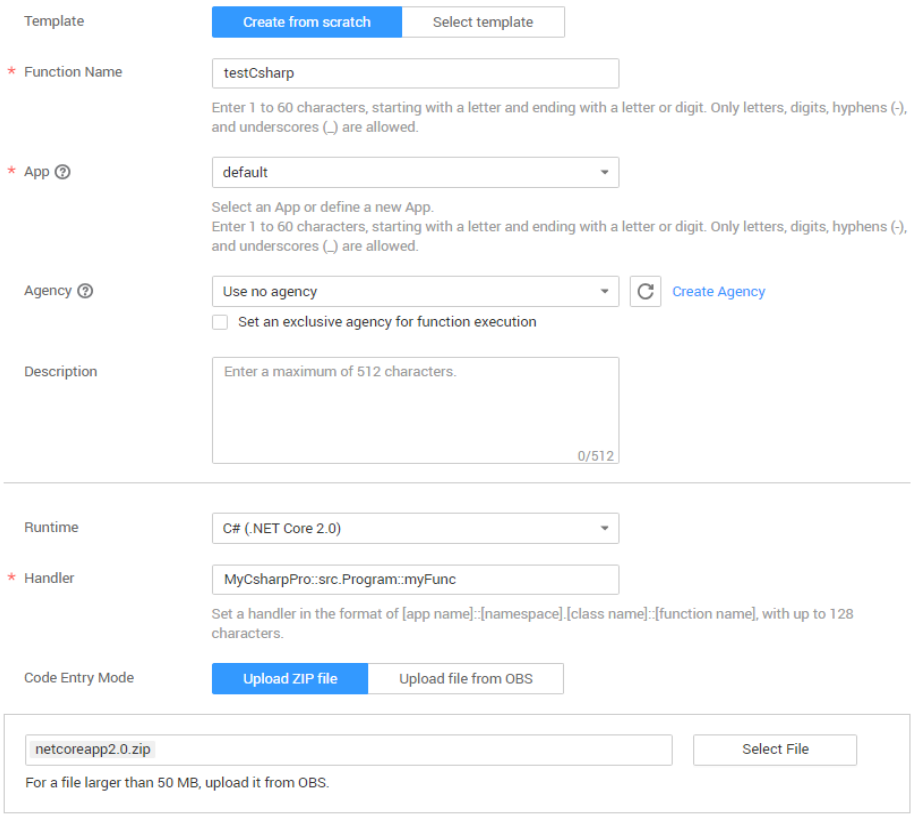
Use the SSH tool to copy and package the compiled files, as shown in **Figure 3-54**.

Figure 3-54 Packaging the compiled files



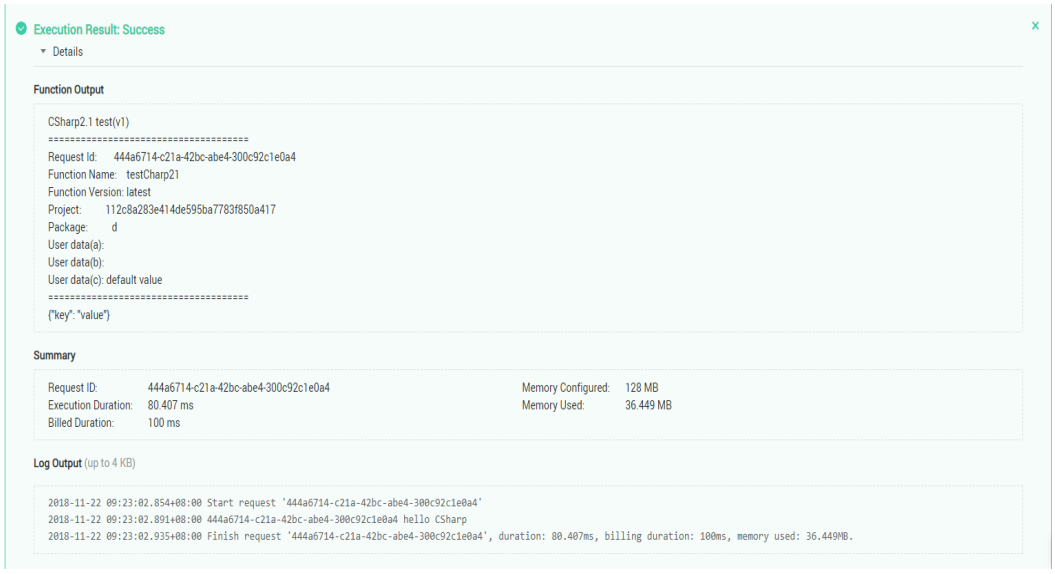
Log in to the FunctionGraph console and create a function, as shown in **Figure 3-55**.

Figure 3-55 Creating a function



Execute the function. The function execution result consists of three parts: function output (returned by **callback**), summary, and logs (output by using the **logf** method of the **context.Logger** object), as shown in **Figure 3-56**.

Figure 3-56 Execution result



----End

Execution Result

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

Table 3-55 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.

Example of the function output when a function fails to be executed:

```
{
  "errorMessage": "Assembly depends json file 'Myapp.deps.json' not found that should be along with Function Handler library.",
  "errorType": "ValidationException"
}
```

Example of the log output when a function fails to be executed:

```
2020-07-27 15:11:47.703+08:00 Start invoke request 'bba1ec4d-5f78-4d83-afae-fdf1c4fbf768', version: latest
2020-07-27 15:11:47.806+08:00 Finish invoke request 'bba1ec4d-5f78-4d83-afae-fdf1c4fbf768'(invoke Failed), duration: 103.364ms, billing duration: 200ms, memory used: 115.516MB
```

3.6.2 JSON Serialization and Deserialization

3.6.2.1 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 2.1. The procedure is similar for .NET Core 2.0 and .NET Core 3.1. The .NET SDK 2.1 has been installed in the execution environment.

Creating a Project

1. Run the following command to create the **/tmp/csharp/projets /tmp/csharp/release** directory:
   ```
   mkdir -p  /tmp/csharp/projets;mkdir -p  /tmp/csharp/release
   ```
2. Run the following command to go to the **/tmp/csharp/projets/** directory:
   ```
   cd /tmp/csharp/projets/
   ```
3. Create the project file **test.csproj** with the following content:
   ```
   <Project Sdk="Microsoft.NET.Sdk">
     <PropertyGroup>
       <TargetFramework>netcoreapp2.1</TargetFramework>
       <RootNamespace>test</RootNamespace>
       <AssemblyName>test</AssemblyName>
     </PropertyGroup>
     <ItemGroup>
       <Reference Include="HC.Serverless.Function.Common">
         <HintPath>HC.Serverless.Function.Common.dll</HintPath>
       </Reference>
       <Reference Include="HC.Serverless.Function.Common.JsonSerializer">
         <HintPath> HC.Serverless.Function.Common.JsonSerializer.dll</HintPath>
       </Reference>
     </ItemGroup>
   </Project>
   ```

Generating a Code Library

1. Upload the **.dll file package**.

   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/projets/** directory.
2. Create the **Class1.cs** file in the **/tmp/csharp/projets/** 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:

```
/home/tools/dotnetcore-sdk/dotnet-sdk-2.1.302-linux-x64/dotnet build /tmp/csharp/projets/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": "2.1.0"
      }
   }
}
```


NOTE

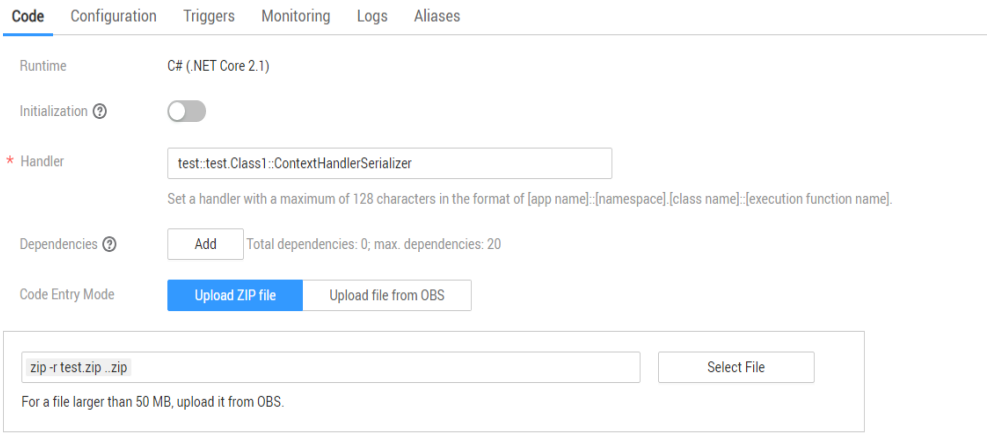
- The name of the ***.runtimeconfig.json** file is the name of an assembly.
- The **Version** parameter in the file indicates the version number of the target framework. If the framework is .NET Core 2.0, enter **2.0.0**. If the framework is .NET Core 2.1, enter **2.1.0**.
- For .NET Core 2.0, check whether **Newtonsoft.Json** is referenced in the ***.deps.json** file. If **Newtonsoft.Json** is not referenced, reference it manually.
  1. Reference the following content in **targets**:
     ```
     "Newtonsoft.Json/9.0.0.0": {
           "runtime": {
             "Newtonsoft.Json.dll": {
               "assemblyVersion": "9.0.0.0",
               "fileVersion": "9.0.1.19813"
             }
           }
         }
     ```
  2. Reference the following content in **libraries**:
     ```
     "Newtonsoft.Json/9.0.0.0": {
         "type": "reference",
         "serviceable": false,
         "sha512": ""
       }
     ```

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

Test Example

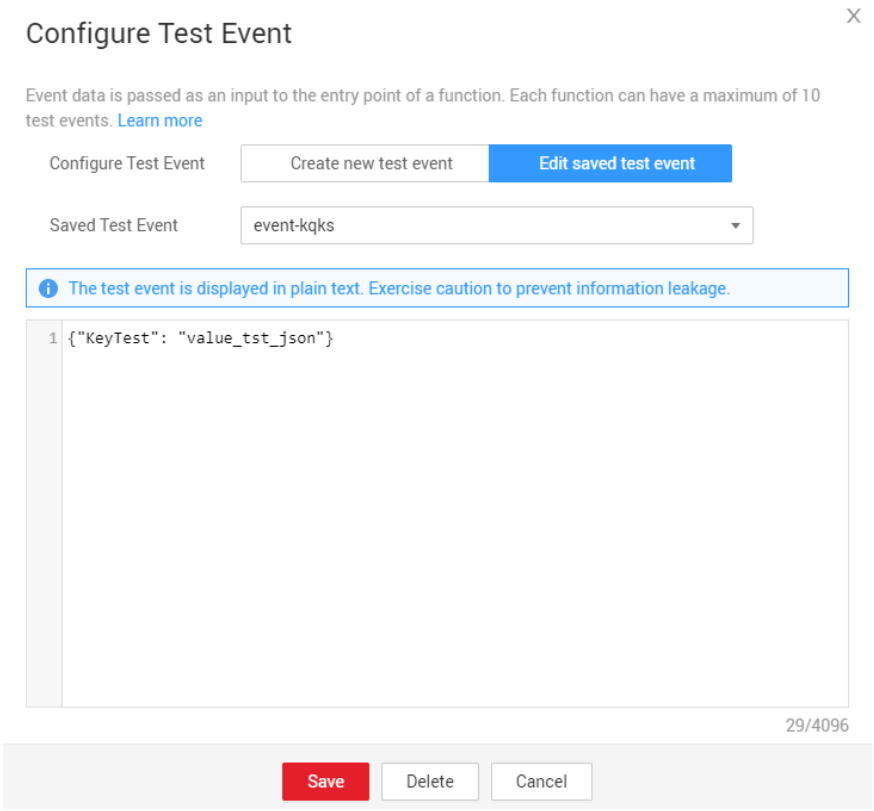
1. Create a C# (.NET 2.1) function on the FunctionGraph console and upload the **zip -r test.zip** file, as shown in **Figure 3-57**.

   Figure 3-57 Uploading the code package

2. Configure a test event, as shown in **Figure 3-58**. The key must be set to **KetTest**, and the value can be customized.

Figure 3-58 Configuring a test event

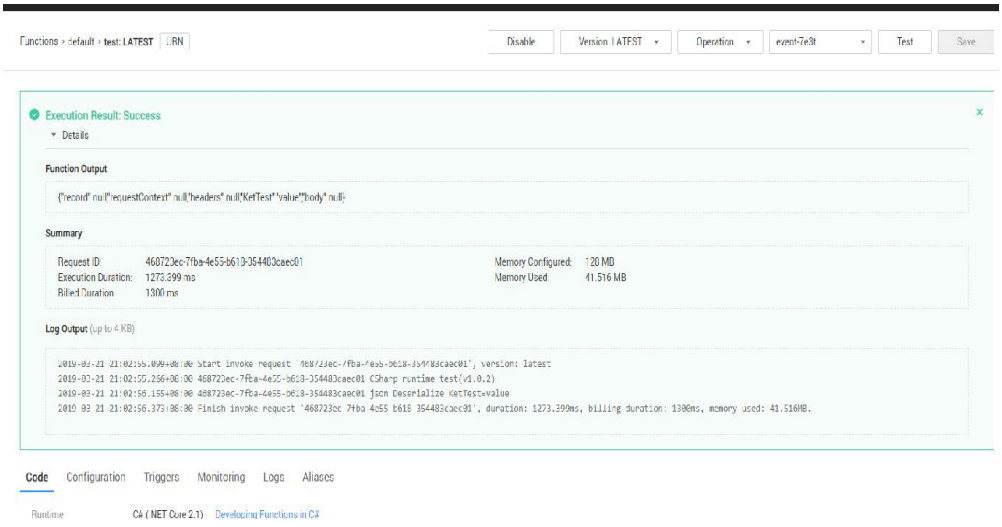


NOTE

- The attribute of the serialization class must be defined as **KetTest**.
- The test string must be in JSON format.

Figure 3-59 shows the test result.

Figure 3-59 Test result



3.6.2.2 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.

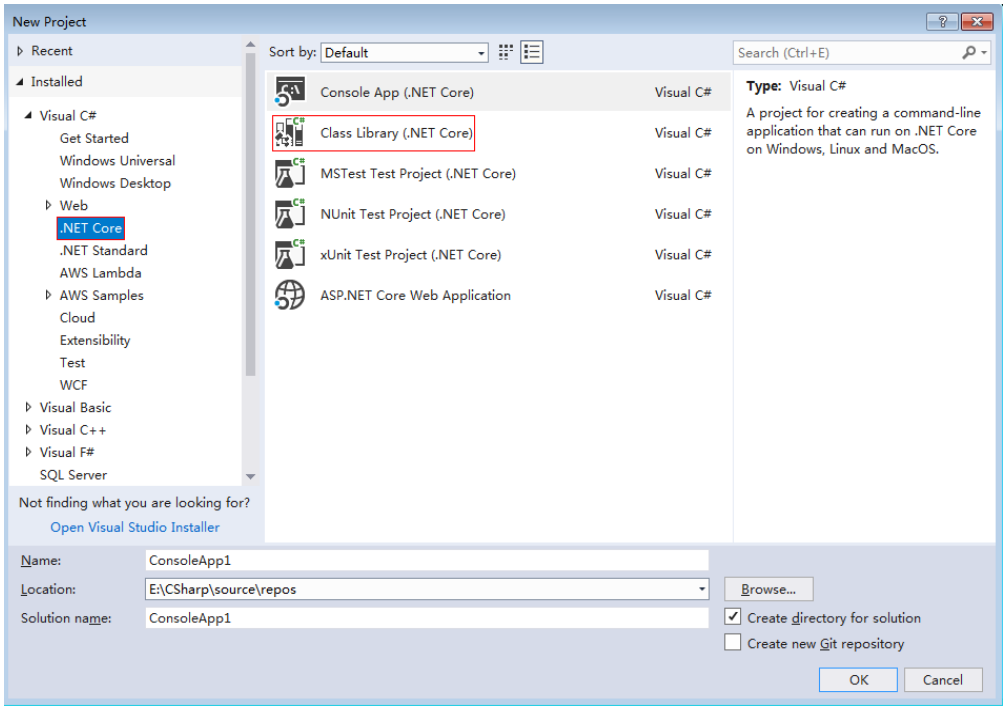
Stream Serialize<T>(T value): Serializes data to the returned response payload.

The following describes how to create a project named **test** based on .NET Core 2.0 by using Visual Studio 2017. The procedure is similar for .NET Core 2.1 and .NET Core 3.1.

Creating a Project

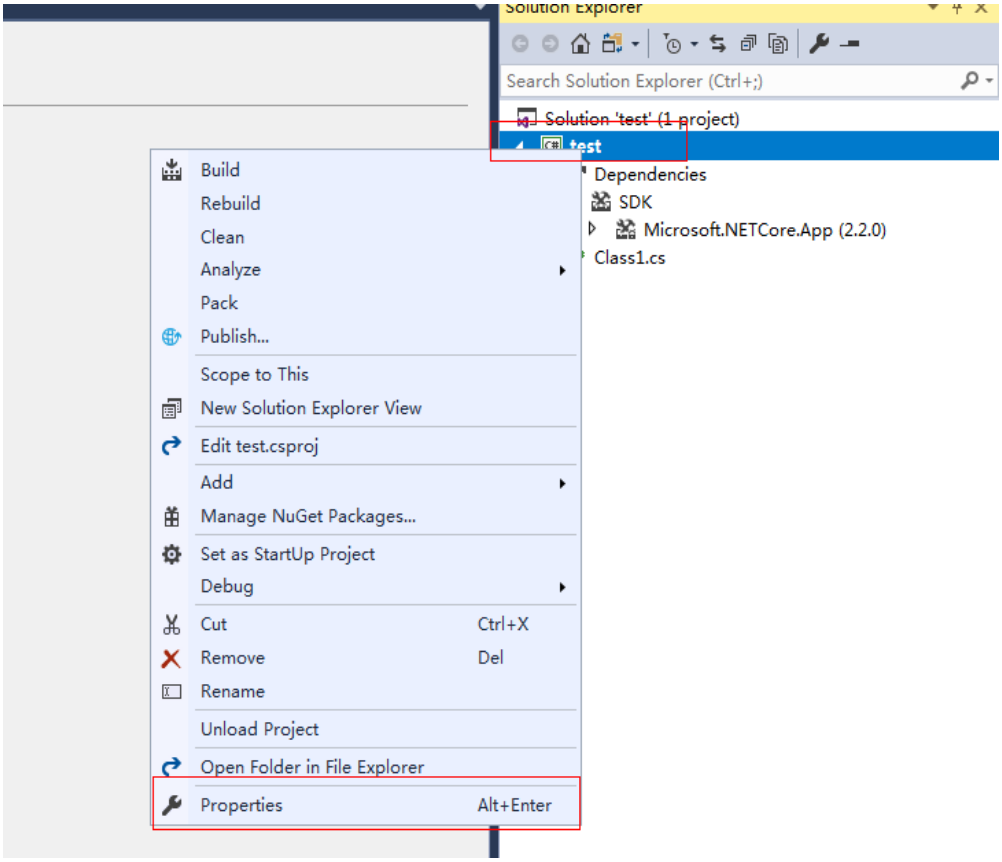
1. On the toolbar, choose **File** > **New** > **Project**, select **.NET Core**, select **Class Library (.NET Core)**, and change the project name to **test**, as shown in **Figure 3-60**.

   Figure 3-60 Creating a project

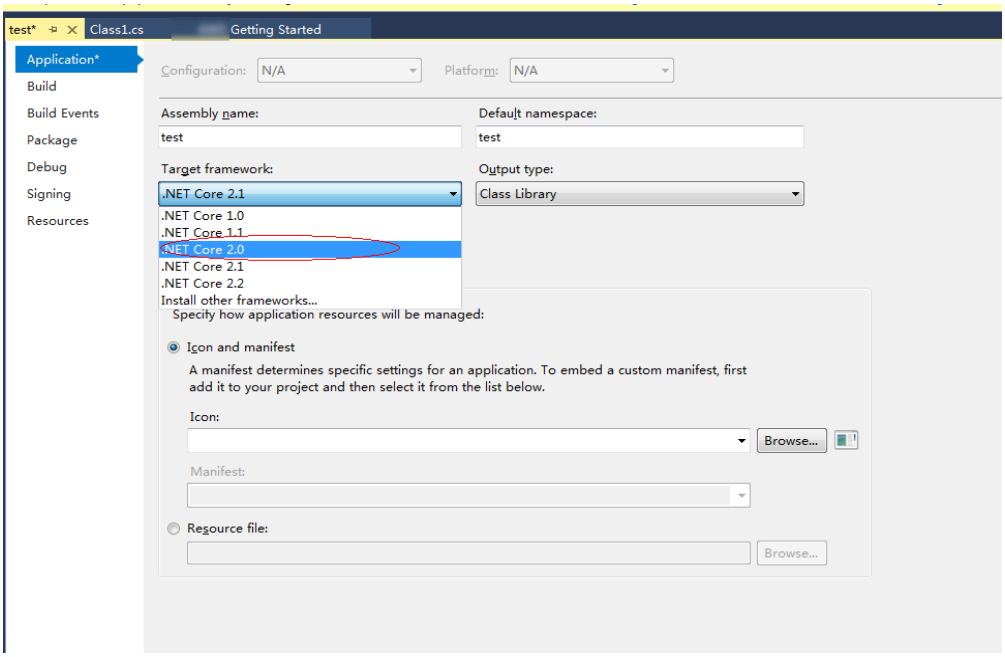
2. Select the **test** project in the navigation pane, right-click, and then choose **Properties**, as shown in **Figure 3-61**.

Figure 3-61 Properties



3. Choose **Application** and then set **Target framework** to **.NET Core 2.0**, as shown in **Figure 3-62**.

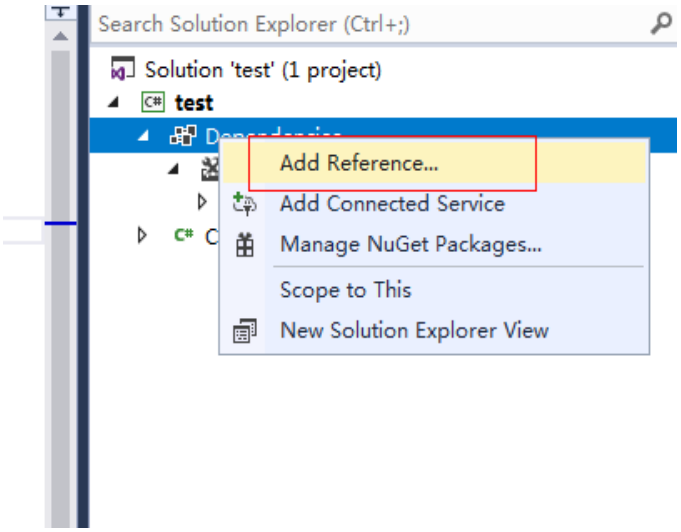
Figure 3-62 Selecting a target framework



Adding a Reference

1. Select the **test** project in **Search Solution Explorer**, right-click, and then choose **Add Reference** to reference the downloaded **.dll file package**, as shown in **Figure 3-63**.

   Figure 3-63 Adding a reference

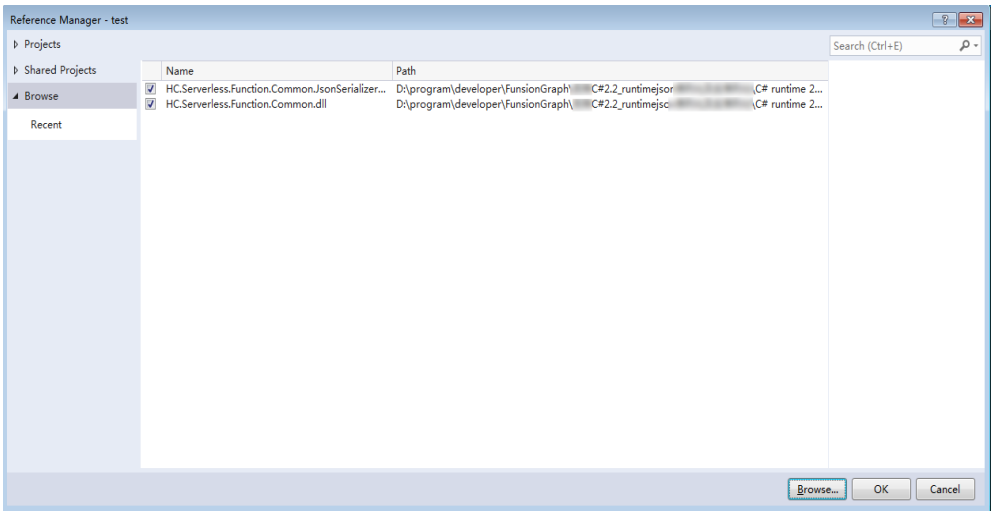
   

   NOTE

   Store **HC.Serverless.Function.Common.dll**, **HC.Serverless.Function.Common.JsonSerializer.dll**, and **Newtonsoft.Json.dll** in a lib file.

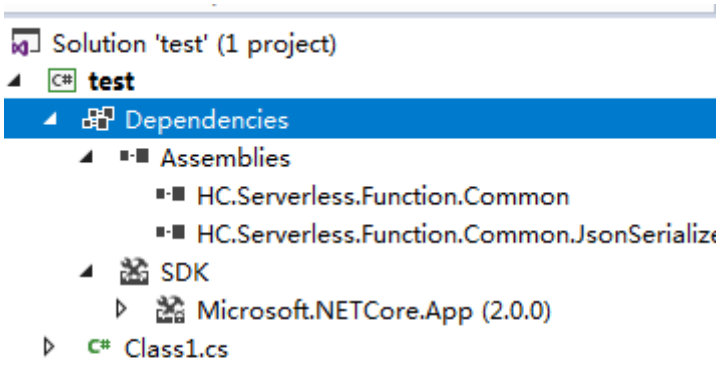
2. Choose **Browse**, click **Browse(B)**, reference the **HC.Serverless.Function.Common.dll** and **HC.Serverless.Function.Common.JsonSerializer.dll** files, and click **OK**, as shown in **Figure 3-64**.

   Figure 3-64 Referencing files

3. View the references, as shown in **Figure 3-65**.

Figure 3-65 References



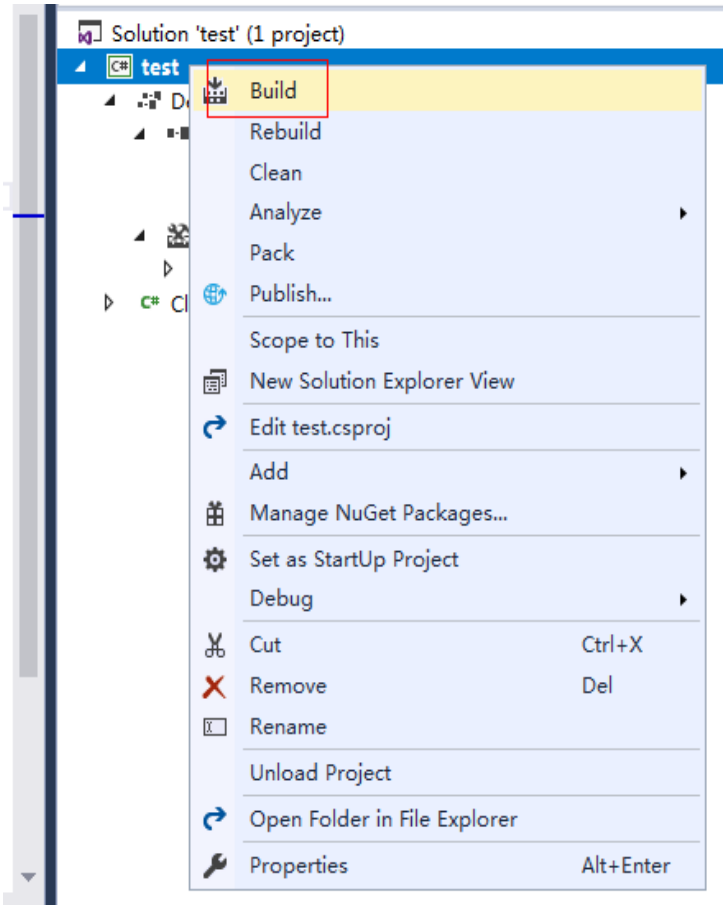
Packing Code

Sample code:

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

namespace test
{
    public class Class1
    {
        public Stream ContextHandlerSerializer(Stream input, IFunctionContext context)
        {
            var logger = context.Logger;
            logger.Logf("CSharp runtime test(v1.0.2)");
            JsonSerializer test = new JsonSerializer();
            TestJson Testjson = test.Deserialize<TestJson>(input);
            if (Testjson != null)
            {
                logger.Logf("json Deserialize KetTest={0}", Testjson.KetTest);
            }

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

        public class TestJson
        {
            public string KetTest { get; set; }//Define the attribute of the serialization class as KetTest.

        }
    }
}
```

1. Right-click the **test** project and choose **Build**, as shown in **Figure 3-66**.

Figure 3-66 Build



2. Copy the path **C:\Users\xxx\source\repos\test\test\bin\Release\netcoreapp2.0** of the .dll files, as shown in **Figure 3-67**.

Figure 3-67 Path of .dll files

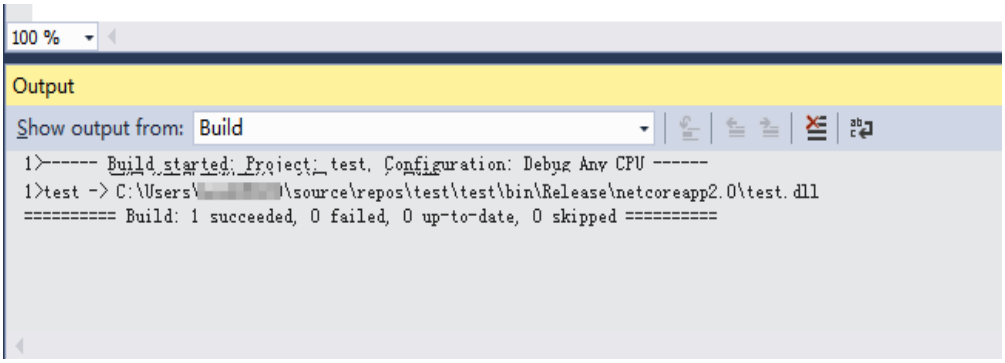
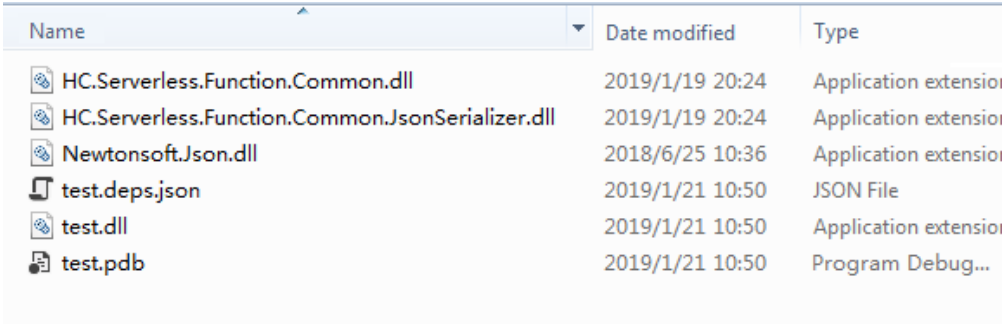


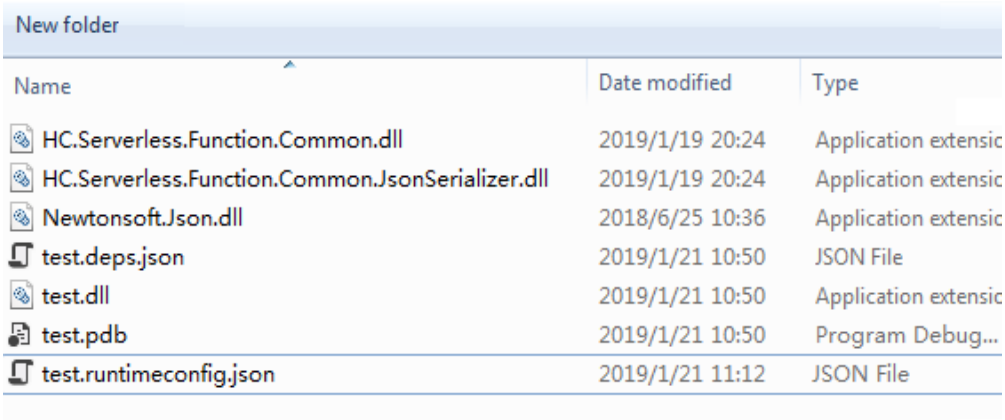
Figure 3-68 shows the files in the path.

Figure 3-68 Files



3. Create a **test.runtimeconfig.json** file in the path, as shown in **Figure 3-69**.

Figure 3-69 New file



Add the following content in the file:

```
{

"runtimeOptions": {

"framework": {

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

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

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. If the framework is .NET Core 2.0, enter **2.0.0**. If the framework is .NET Core 2.1, enter **2.1.0**.

4. Compress the file into **netcoreapp2.0.zip**.

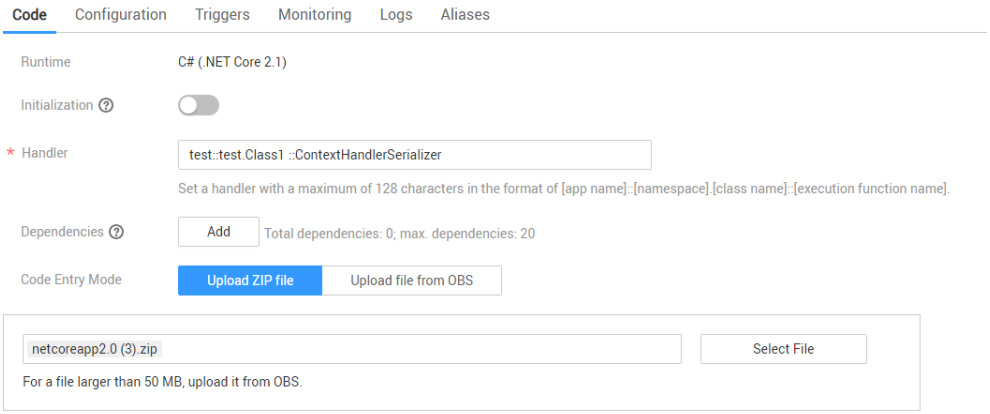
NOTE

The package name can be customized but must be ended with .zip.

Test Example

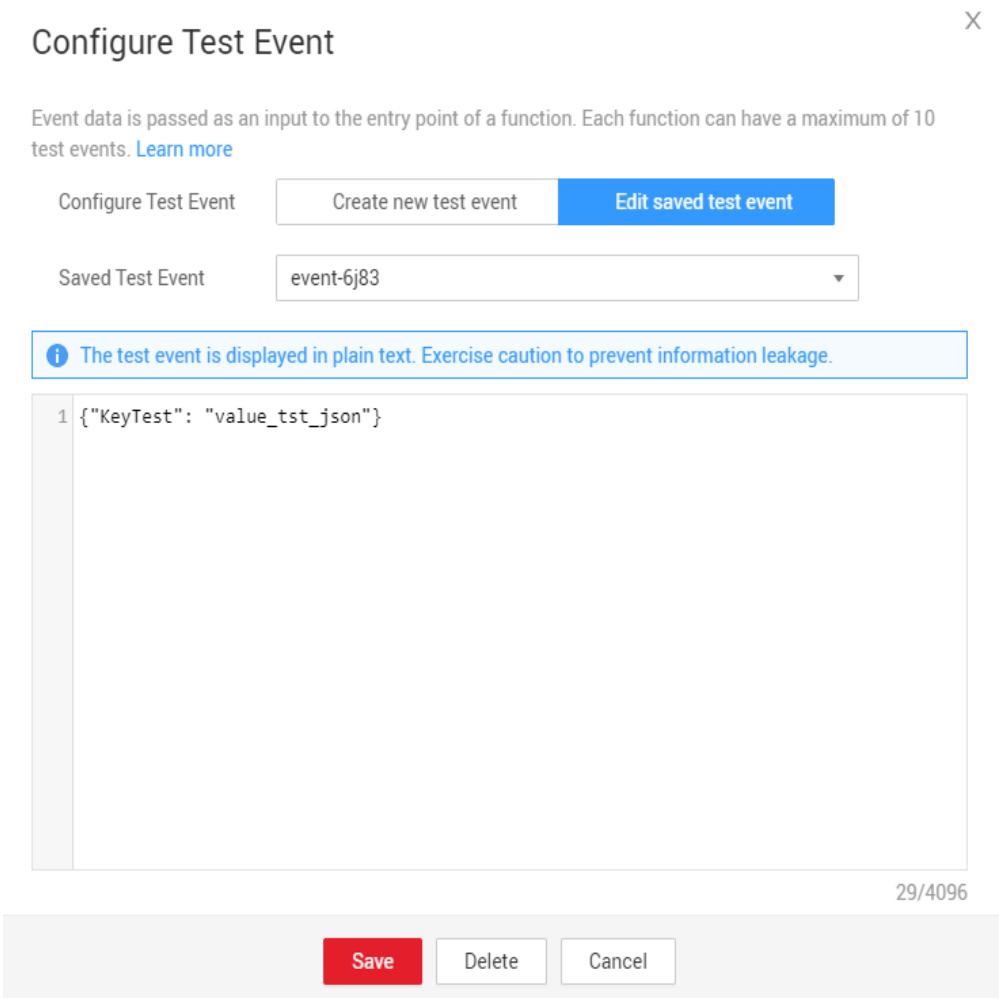
1. Create a C# (.NET 2.1) function on the FunctionGraph console and upload the code package, as shown in **Figure 3-70**.

   Figure 3-70 Uploading the code package

2. Configure a test event, as shown in **Figure 3-71**. The key must be set to **KetTest**, and the value can be customized.

Figure 3-71 Configuring a test event

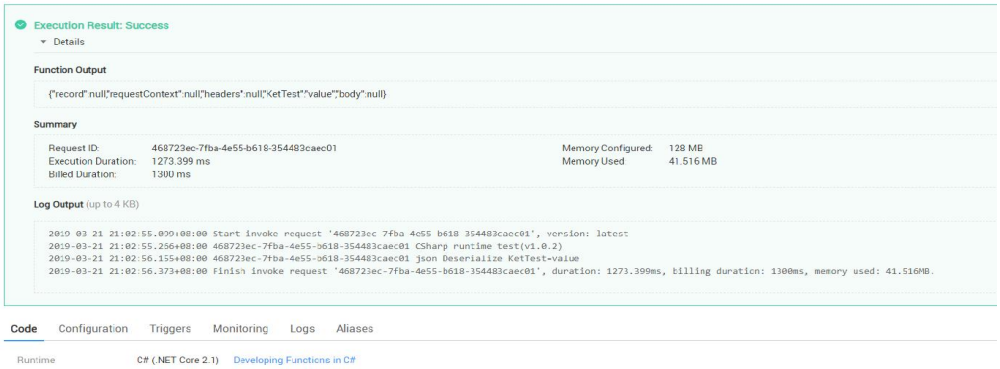


NOTE

- The attribute of the serialization class must be defined as **KetTest**.
- The test string must be in JSON format.

3. Figure 3-72 shows the test result.

Figure 3-72 Test result



3.7 Developing Functions in PHP

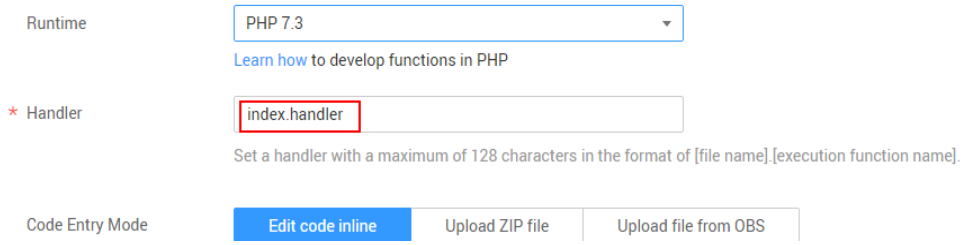
Function Syntax

Use the following syntax when creating a handler function in PHP 7.3:

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 function handler is in the format of *[File name].[Function name]*. For example, if you set the handler to **index.handler** in your function, as shown in **Figure 3-73**, FunctionGraph will load the handler function defined in the **index.js** file.

Figure 3-73 Handler of the function



SDK APIs

The following table describes the context methods provided by FunctionGraph.

Table 3-56 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) of an agency. If you use this method, you need to configure an agency for the function.
getSecretKey()	Obtains the SK (valid for 24 hours) of an agency. If you use this method, you need to configure an agency for the function.

Method	Description
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()	Number of CPU millicores used by the function (1 core = 1000 millicores). The value of this field is proportional to that of **MemorySize**. By default, 100 CPU millicores are required for 128 MB memory. The number of CPU millicores is calculated as follows: Memory/128 x 100 + 200 (basic CPU millicores).
getProjectID()	Obtains a project ID.
getPackage()	Obtains a function group, that is, an app.
getToken()	Obtains the token (valid for 24 hours) of 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")

NOTE

Results returned by using the **getToken()**, **getAccessKey()**, and **getSecretKey()** methods contain sensitive information. Exercise caution when using these methods.

Developing a PHP Function

Perform the following steps to develop 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;
}
```

   NOTE

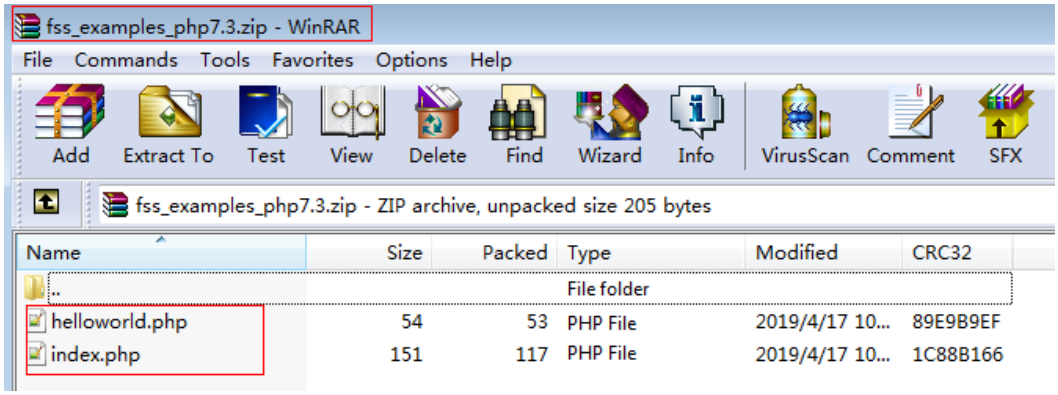
   FunctionGraph can return only the following types of values:

   - **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**.

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 3-74**.

Figure 3-74 Packaging the project files



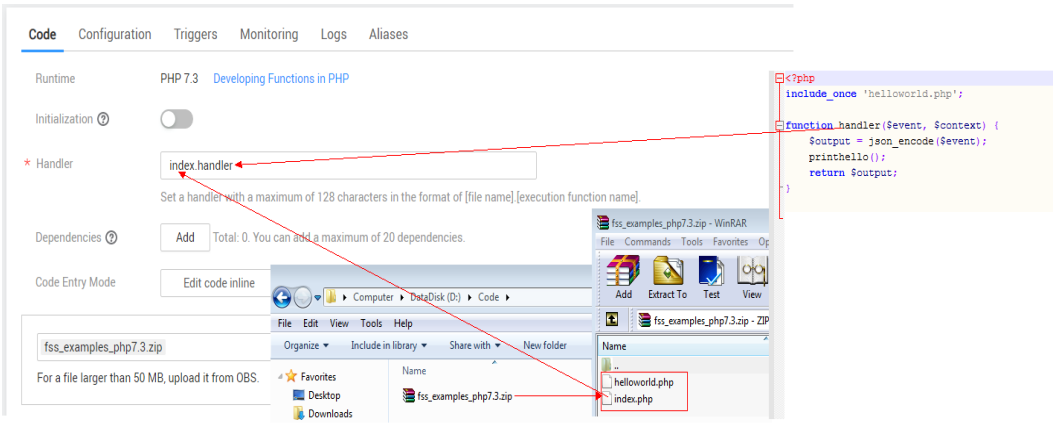
NOTE

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 ZIP file is decompressed.

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 3-75**.

Figure 3-75 Uploading the code package



NOTE

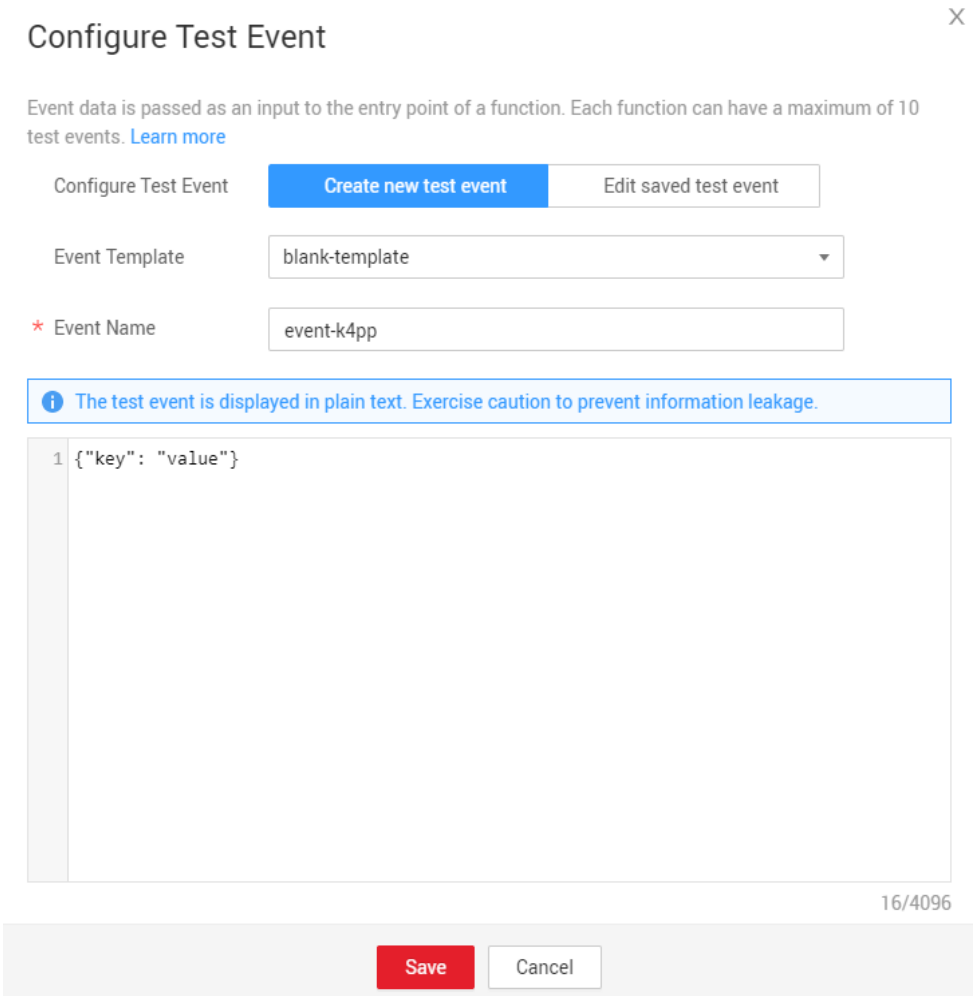
- The **index** of the handler must be consistent with the file name of your function, because the file name will help to locate the function file.
- The **handler** is a function name, which must be consistent with that defined in the **index.php** file.
- After you upload the **fss_examples_php7.3.zip** 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.

Step 4 Test the function.

1. Create a test event.

   On the function details page that is displayed, choose **Select test event** > **Configure test event**. Configure the test event information, as shown in **Figure 3-76**, and then click **Save**.

Figure 3-76 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** or **getLogger()** method), as shown in **Figure 3-77**.

Figure 3-77 Execution result

----End

Execution Result

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

Table 3-57 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.

Example of the function output when a function fails to be executed:

```
{
    "errorMessage": "syntax error, unexpected '$data' (T_VARIABLE), expecting ')'",
    "errorType": "ParseError",
    "stackTrace": {
        "file": "\/index.php",
        "line": 24
    }
}
```

Example of the log output when a function fails to be executed:

```
2020-07-27 14:40:03.817+08:00 Start invoke request '1512e67f-98c6-474e-b5c2-051ada1ccb94', version: latest
2020-07-27 14:40:03.819+08:00 Finish invoke request '1512e67f-98c6-474e-b5c2-051ada1ccb94'(invoke Failed), duration: 2.273ms, billing duration: 100ms, memory used: 54.320MB
```

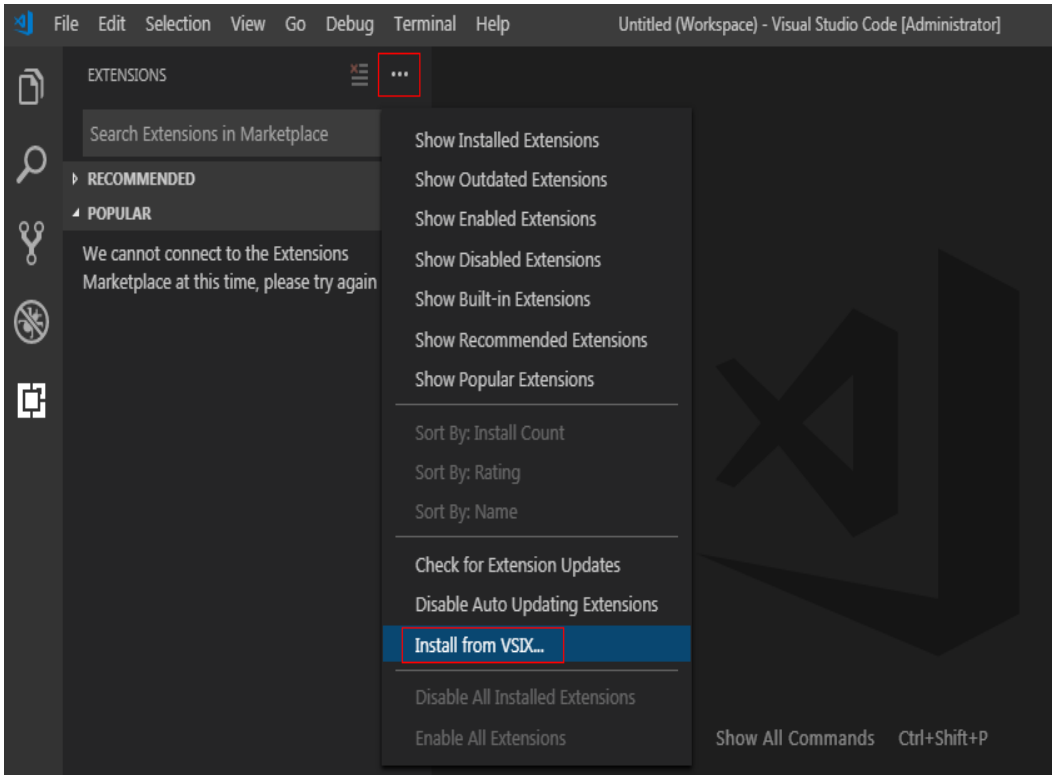

4 Developing Functions with Plug-ins

4.1 Visual Studio Code Plug-in

Step 1 Download the **Visual Studio Code plug-in**.

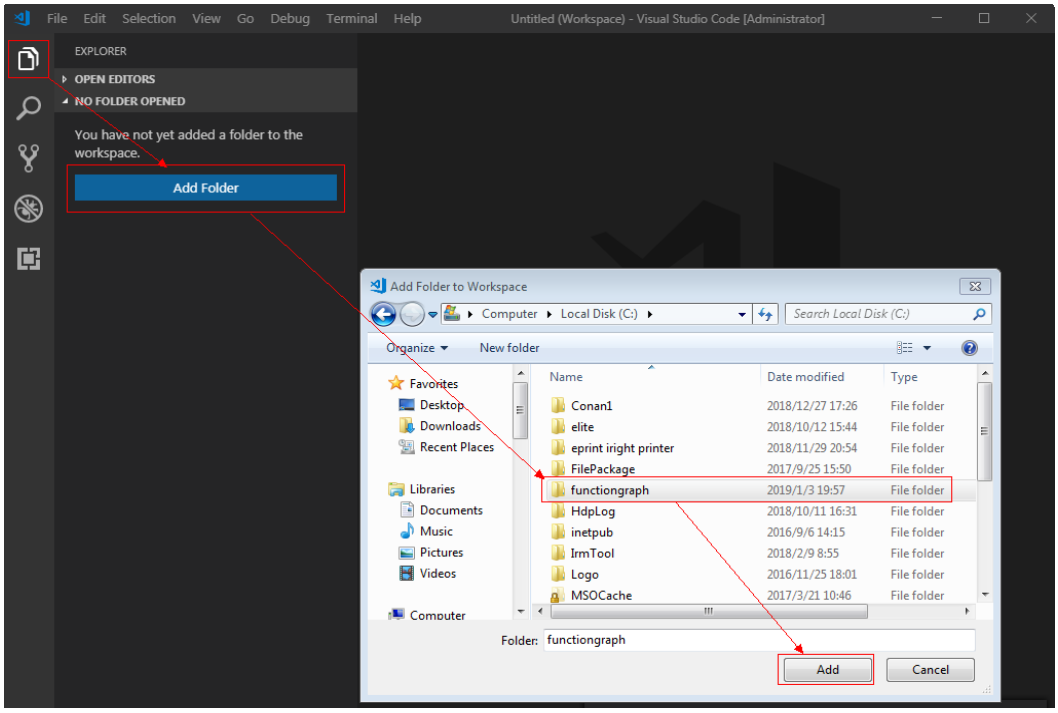
Step 2 Manually install **funcitongraph-tools**, as shown in **Figure 4-1**.

Figure 4-1 Installing funcitongraph-tools



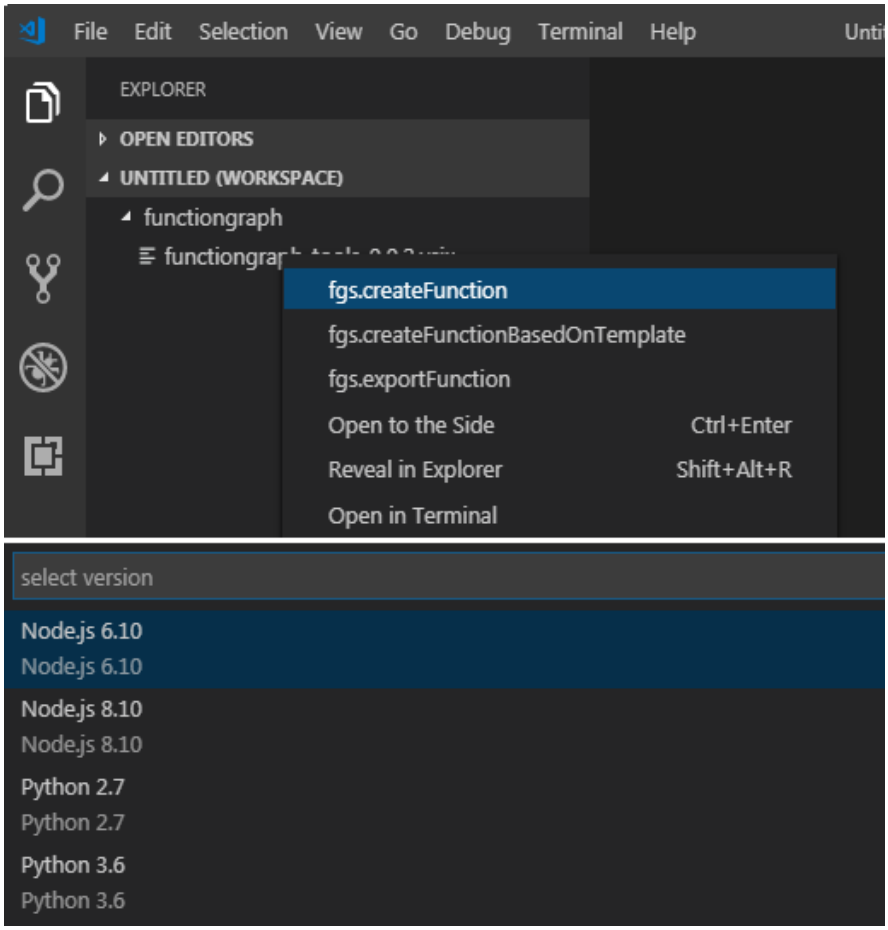
Step 3 Add the folder in which the installation file is located, as shown in **Figure 4-2**.

Figure 4-2 Adding the folder of the installation file



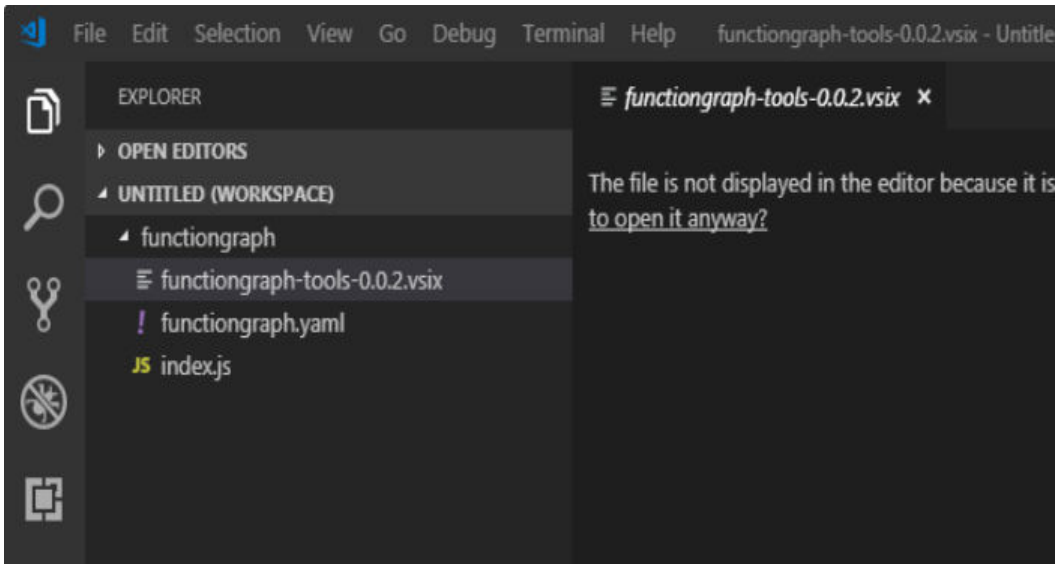
Step 4 Right-click, choose **fgs.createFunction** from the shortcut menu, and select a runtime to create a function, as shown in **Figure 4-3**.

Figure 4-3 Creating a function



After a function is created, the **functiongraph.yaml** and **index.js** files are automatically generated, as shown in **Figure 4-4**.

Figure 4-4 Configuration files



Step 5 Right-click and choose **fgs.createFunctionBasedOnTemplate** from the shortcut menu to create a function template.

Step 6 Right-click and choose **fgs.exportFunction** from the shortcut menu to export a function. Then you can upload the function to FunctionGraph.

----End

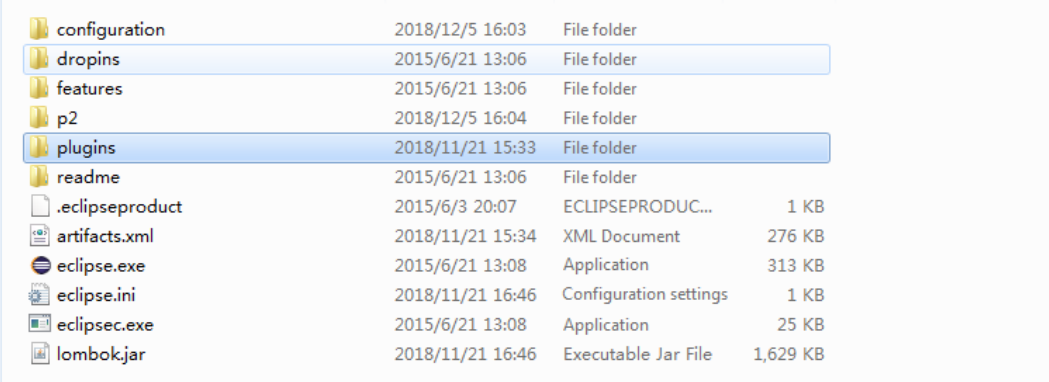
4.2 Eclipse Plug-in

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 Download the **Eclipse plug-in**.

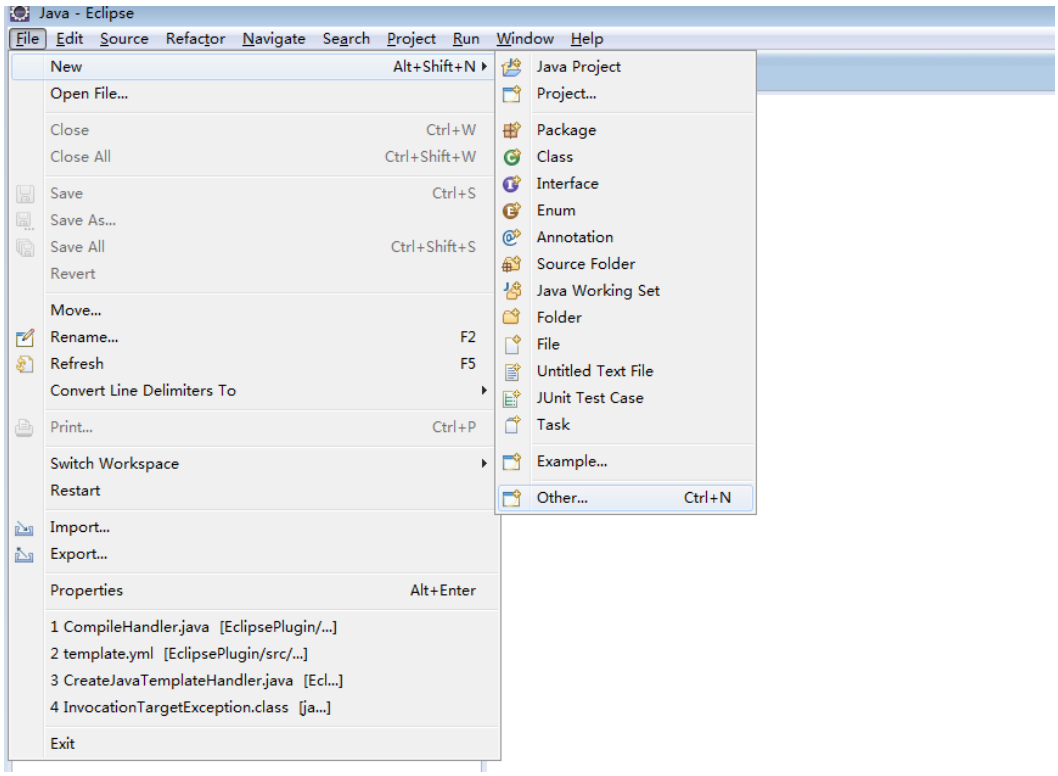
Step 2 Put the Eclipse plug-in package (.jar or .zip) in the **plugins** folder under the Eclipse installation directory. Then restart Eclipse. **Figure 4-5** shows the Eclipse installation directory.

Figure 4-5 Installing the Eclipse plug-in



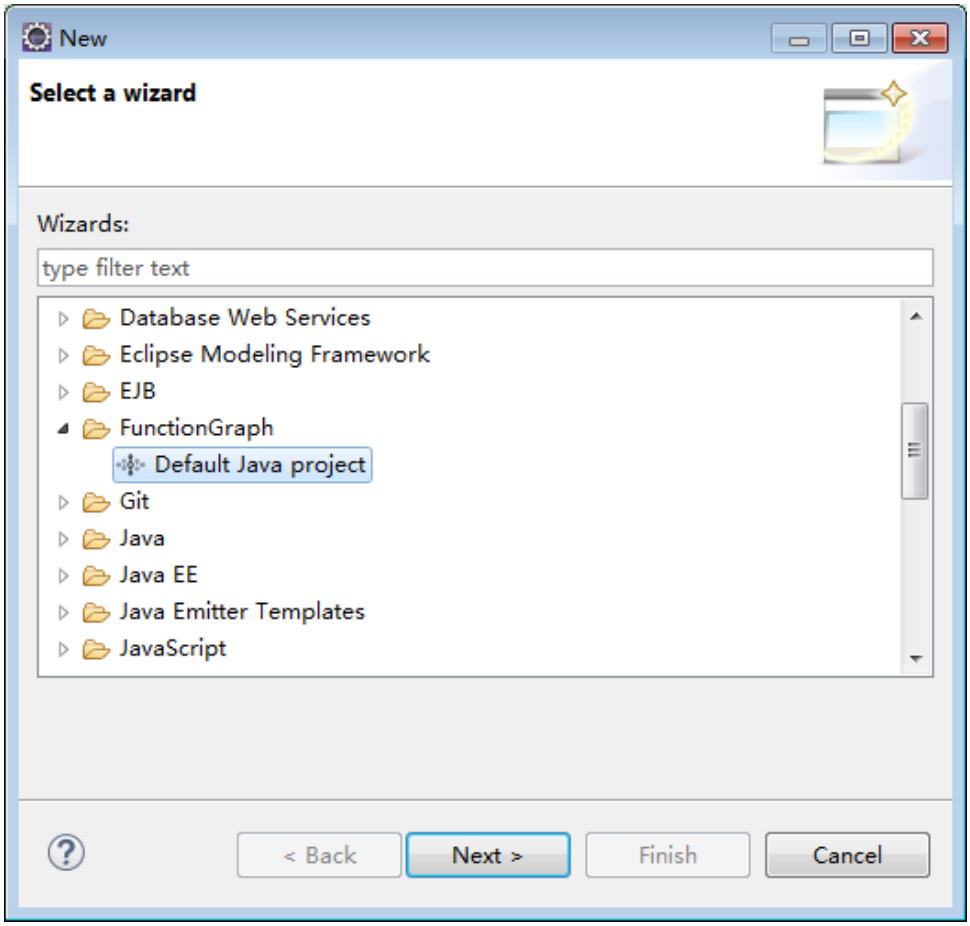
Step 3 Open Eclipse and choose **File** > **New** > **Other**, as shown in **Figure 4-6**.

Figure 4-6 Creating a template



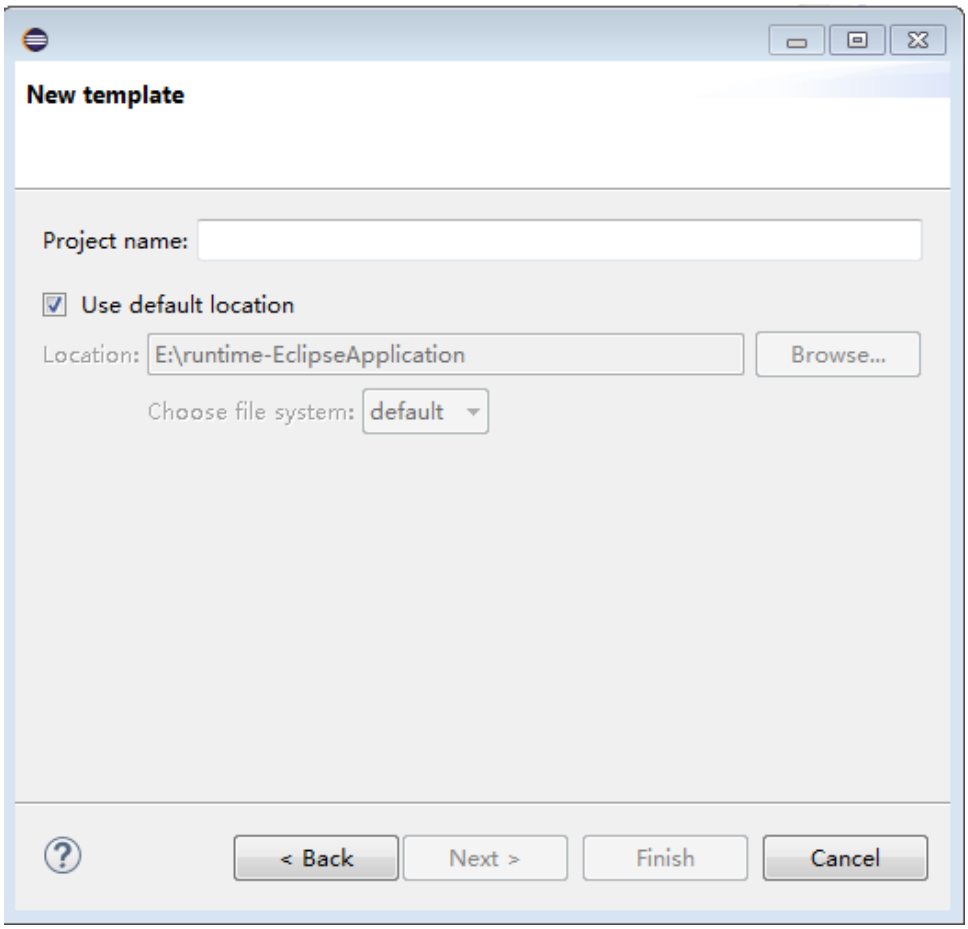
Step 4 Choose **FunctionGraph** > **Default Java project**, as shown in **Figure 4-7**.

Figure 4-7 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 4-8**.

Figure 4-8 Setting the project name and directory



----End

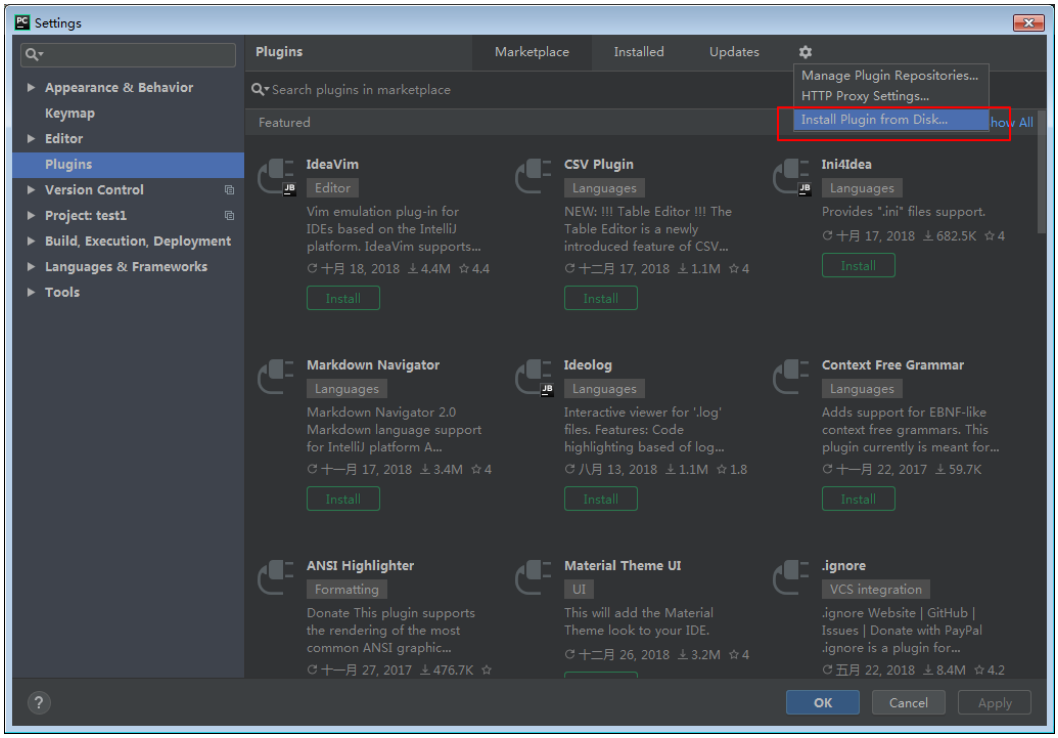
4.3 PyCharm Plug-in

With PyCharm, you can quickly generate Python templates, package project files, and deploy Python functions.

Step 1 Download the **PyCharm plug-in**.

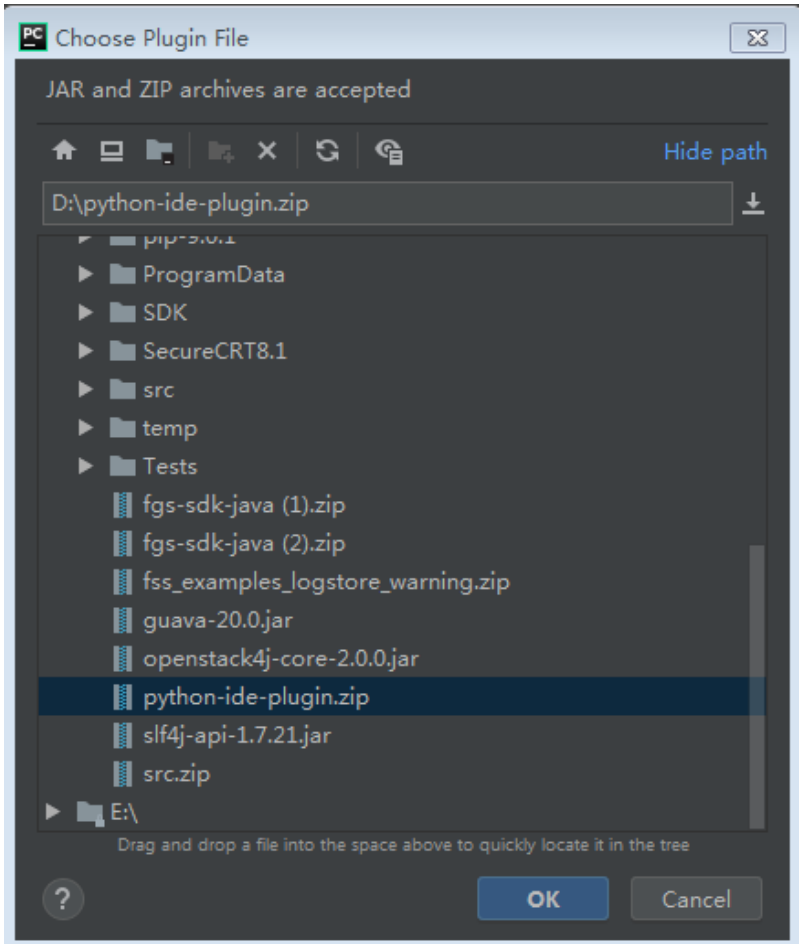
Step 2 Run JetBrains PyCharm. Choose **File** > **Settings**, choose **Plugins** in the left pane, and then click **Install Plugin from Disk** in the upper right corner, as shown in **Figure 4-9**.

Figure 4-9 Installing the plug-in



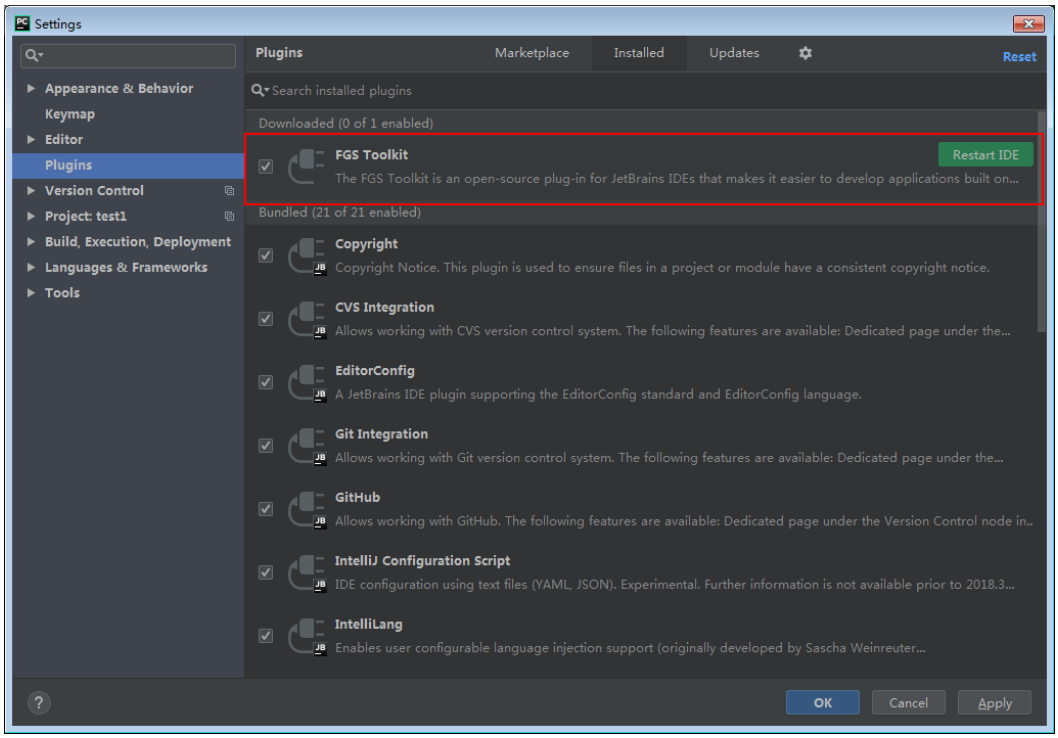
Step 3 Select the plug-in package you want to install, and click **OK**, as shown in **Figure 4-10**.

Figure 4-10 Selecting a plug-in package



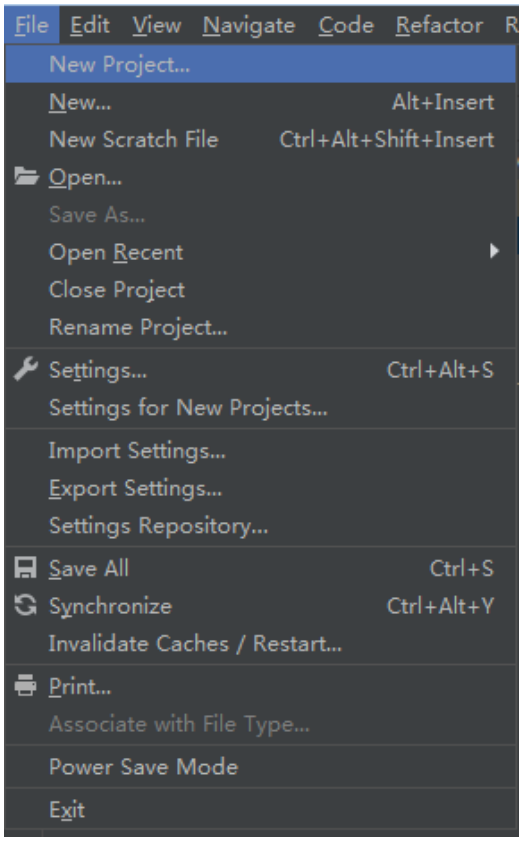
Step 4 In the plug-in list, select the desired plug-in and click **Restart IDE**, as shown in **Figure 4-11**.

Figure 4-11 Restarting the IDE



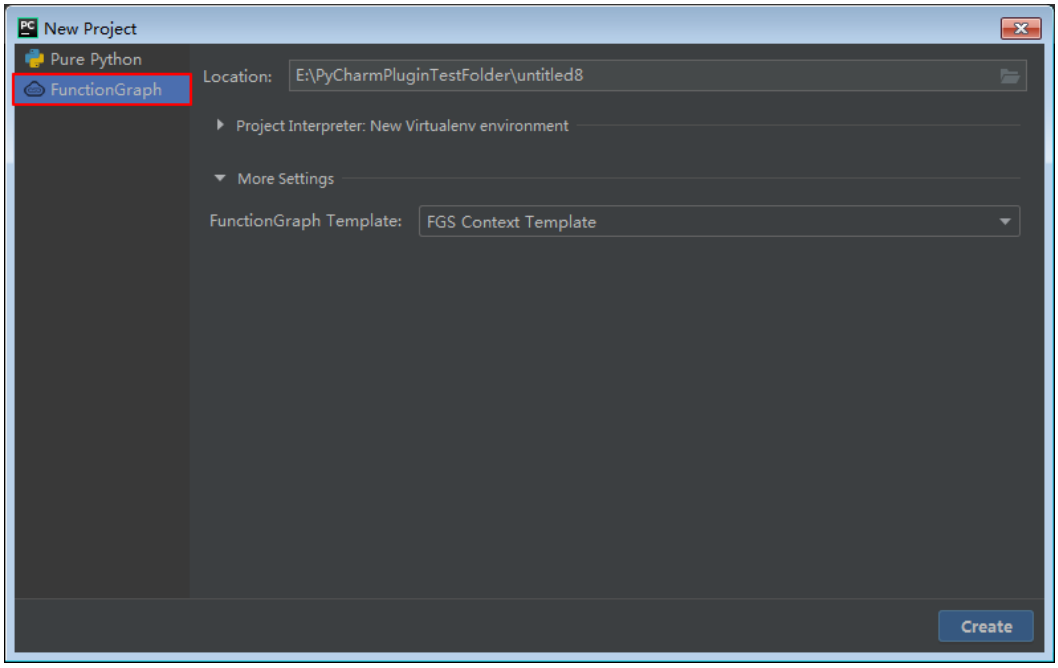
Step 5 Choose **File** > **New Project**, as shown in **Figure 4-12**.

Figure 4-12 Creating a project



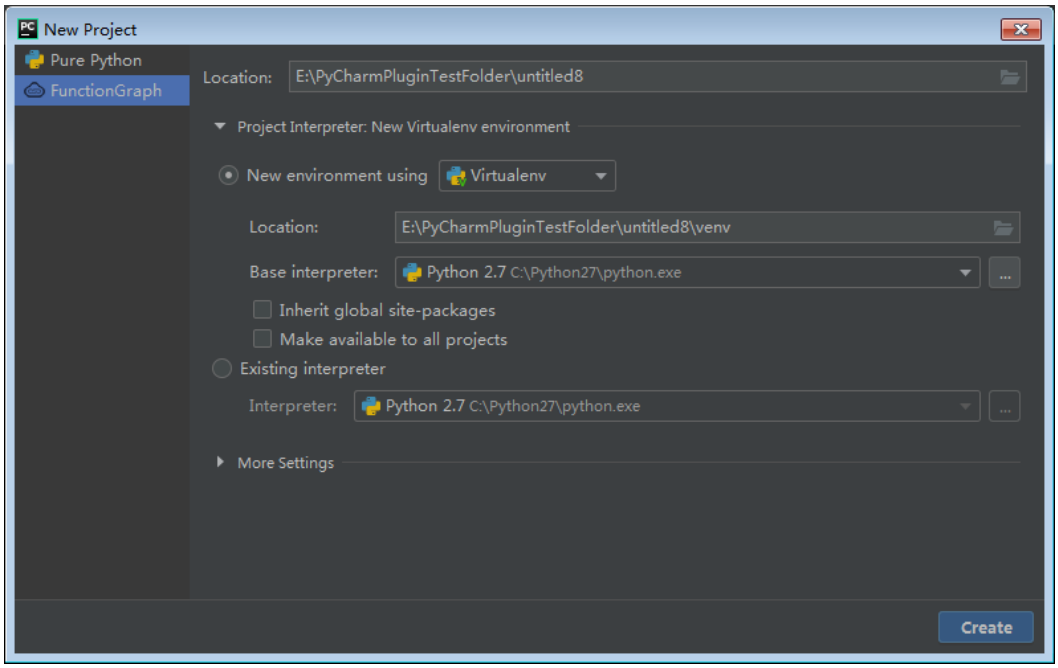
Step 6 On the displayed **New Project** page, choose **FunctionGraph**, as shown in **Figure 4-13**.

Figure 4-13 FunctionGraph



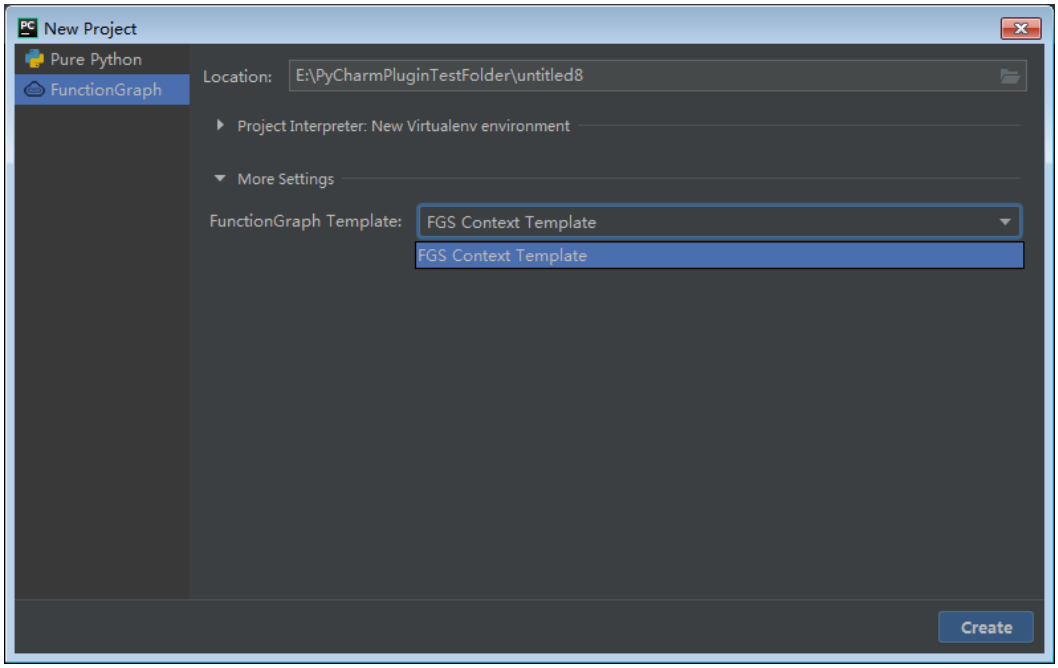
Step 7 Select the path in which the project will be stored in **Location**, and select a Python version in **Base interpreter**, as shown in **Figure 4-14**.

Figure 4-14 Selecting a version



Step 8 Select a template you want to create in the **More Settings** area, as shown in **Figure 4-15**.

Figure 4-15 Selecting a template



NOTE

Currently, only the Python 2.7 context template is supported.

Step 9 Click **Create**.

----End

