



# 函数 workflow

开发指南

天翼云科技有限公司

<b>1 概述</b> .....	<b>4</b>
1.1 函数开发简介 .....	4
1.2 函数支持的事件源 .....	5
1.3 函数工程打包规范 .....	15
1.4 在函数中引入动态链接库 .....	19
<b>2 函数初始化入口 Initializer</b> .....	<b>20</b>
<b>3 Node.js</b> .....	<b>22</b>
3.1 开发事件函数 .....	22
3.2 开发 HTTP 函数 .....	28
3.3 nodejs 模板 .....	30
3.4 制作依赖包 .....	31
<b>4 Python</b> .....	<b>32</b>
4.1 开发事件函数 .....	32
4.2 python 模板 .....	38
4.3 制作依赖包 .....	38
<b>5 Java</b> .....	<b>40</b>
5.1 开发事件函数 .....	40
5.1.1 Java 函数开发指南（使用 Eclipse 工具） .....	40
5.1.2 Java 函数开发指南（使用 IDEA 工具普通 Java 项目） .....	56
5.1.3 Java 函数开发指南（使用 IDEA 工具 maven 项目） .....	63
5.2 java 模板.....	67
<b>6 Go</b> .....	<b>68</b>
6.1 开发事件函数 .....	68
<b>7 C#</b> .....	<b>79</b>
7.1 开发事件函数 .....	79
7.1.1 C#函数开发.....	79
7.1.2 函数支持 json 序列化和反序列化 .....	85
<b>8 PHP</b> .....	<b>96</b>
8.1 开发事件函数 .....	96
8.2 制作依赖包 .....	101
<b>9 开发工具</b> .....	<b>102</b>
9.1 CodeArts IDE Online .....	102

9.1.1 CodeArts IDE Online 在线管理函数.....	102
9.1.2 附录：CodeArts IDE Online 使用方法.....	110
9.2 Eclipse-plugin.....	119
9.3 PyCharm-Plugin .....	122

---

# 1 概述

---

## 1.1 函数开发简介

### 函数支持的运行时语言

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

#### 说明

建议使用相关语言的最新版本。

表1-1 运行时说明

运行时语言	支持版本	SDK 下载
Node.js	6.10、8.10、10.16、12.13、14.18、16.17、18.15	-
Python	2.7、3.6、3.9、3.10	-
Java	8、11	-
Go	1.8、1.x	-
C#	.NET Core 2.0 、.NET Core 2.1、.NET Core 3.1	-
PHP	7.3	-
定制运行时	-	-
Cangjie	1.0	-

## Node.js Runtime 集成的三方件

表1-2 Node.js Runtime 集成的三方件

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

## Python Runtime 集成的非标准库

表1-3 Python Runtime 集成的非标准库

模块	功能	版本号
dateutil	日期/时间处理	2.6.0
requests	http 库	2.7.0
httplib2	httpclient	0.10.3
numpy	数学计算	1.13.1
redis	redis 客户端	2.10.5
obsclient	OBS 客户端	-
smnsdk	访问 SMN 服务	1.0.1

## 1.2 函数支持的事件源

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

## API 网关服务 (APIG 专享版)

可以通过 HTTPS 调用 FunctionGraph 函数，使用 API Gateway 自定义 REST API 和终端节点来实现。可以将各个 API 操作（如 GET 和 PUT）映射到特定的 FunctionGraph 函数，当向该 API 终端节点发送 HTTPS 请求时（[APIG 示例事件](#)），API Gateway 会调用相应的 FunctionGraph 函数。

## 对象存储服务 OBS

可以编写 FunctionGraph 函数来处理 OBS 存储桶事件，例如对象创建事件或对象删除事件。当用户将一张照片上传到存储桶时，OBS 存储桶调用 FunctionGraph 函数，实现读取图像和创建照片缩略图。

表1-4 OBS 支持事件类型

事件	说明
ObjectCreated	表示所有创建对象的操作，包含 Put、Post、Copy 对象以及合并段。
Put	使用 Put 方法上传对象。
Post	使用 Post 方法上传对象。
Copy	使用 copy 方法复制对象。
CompleteMultipartUpload	表示合并分段任务。
ObjectRemoved	表示删除对象。
Delete	指定对象版本号删除对象。
DeleteMarkerCreated	不指定对象版本号删除对象。

### 说明

多个事件类型可以作用于同一个目标对象，例如：同时选择“事件类型”复选框中的 Put、Copy、Delete 等方法作用于某目标对象，则用户往该桶中上传、复制、删除符合前后缀规则的目标对象时，均会发送事件通知给用户。ObjectCreated 包含了 Put、Post、Copy 和 CompleteMultipartUpload，如果选择了 ObjectCreated，则不能再选择 Put、Post、Copy 或 CompleteMultipartUpload。同理如果选择了 ObjectRemoved，则不能再选择 Delete 或 DeleteMarkerCreated。

## 定时触发器 TIMER

可以使用 TIMER 的计划事件功能定期调用您的代码，可以指定固定频率（分钟、小时、天数）或指定 cron 表达式定期调用函数。

## 日志触发器 LTS

可以编写 FunctionGraph 函数来处理云日志服务订阅的日志，当云日志服务采集到订阅的日志后，可以通过将采集到的日志作为参数传递（[LTS 示例事件](#)）来调用

FunctionGraph 函数，FunctionGraph 函数代码可以对其进行自定义处理、分析或将其加载到其他系统。

## 云审计服务触发器 CTS

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

## 文档数据库服务 DDS

使用 DDS 触发器，每次更新数据库中的表时，都可以触发 FunctionGraph 函数以执行额外的工作。

## 分布式消息服务 Kafka 版

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

## 分布式消息服务 RabbitMQ 版

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

## 示例事件

- OBS 示例事件

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventTime": "2018-01-09T07:50:50.028Z",
      "requestParameters": {
        "sourceIPAddress": "103.218.216.125"
      },
      "s3": {
        "configurationId":
"UK1DGFPPYUKUZFHNQ00000160CC0B471D101ED30CE24DF4DB",
        "object": {
          "eTag": "9d377b10ce778c4938b3c7e2c63a229a",
          "sequencer": "00000000160D9E681484D6B4C000000",
          "key": "job.png",
          "size": 777835
        },
        "bucket": {
          "arn": "arn:aws:s3:::syj-input2",
          "name": "functionstorage-template",
          "ownerIdentity": {
            "PrincipalId": "0ed1b73473f24134a478962e631651eb"
          }
        }
      }
    }
  ]
}
```

```

        }
    },
    "Region": "{region}",
    "eventName": "ObjectCreated:Post",
    "userIdentity": {
        "principalId": "9bf43789b1ff4b679040f35cc4f0dc05"
    }
}
]
}

```

表1-5 参数说明

参数	类型	示例值	描述
eventVersion	String	2.0	事件协议的版本。
eventTime	String	2018-01-09T07:50:50.028Z	事件产生的时间。使用 ISO-8601 标准时间格式。
sourceIPAddress	String	103.218.216.125	请求的源 IP 地址
s3	Map	参考示例	OBS 事件内容
object	Map	参考示例	object 参数内容
bucket	Map	参考示例	bucket 参数内容
arn	String	arn:aws:s3:::syj-input2	Bucket 的唯一标识符
ownerIdentity	Map	参考示例	创建 Bucket 的用户 ID
Region	String	/	Bucket 所在的地域
eventName	String	ObjectCreated:Post	配置的触发函数的事件
userIdentity	Map	参考示例	请求发起者的帐号 ID

- APIG 示例事件

```

{
  "body": "{\"test\":\"body\"}",
  "requestContext": {
    "apiId": "bc1dcffd-aa35-474d-897c-d53425a4c08e",
    "requestId": "11cdcdef33949dc6d722640a13091c77",
    "stage": "RELEASE"
  },
  "queryStringParameters": {
    "responseType": "html"
  },
  "httpMethod": "GET",

```



```

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

```

**说明**

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

```

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

```

表1-6 参数说明

参数	类型	示例值	描述
body	String	"{\"test\":\"body\"}"	记录实际请求转换为 String 字符串后的内容。
requestContext	Map	参考示例	请求来源的 API 网关的配置信息、请求标识、认证信息、来源信息。
httpMethod	String	GET	记录实际请求的 HTTP 方法
queryStringParameters	Map	参考示例	记录在 API 网关中配置过的 Query 参数



```
Mjg2ZWQ0ODhjZTcwIiwKICAgICJsb2dfdG9waWNfaWQiOiAiMWE5Njc1YTctNzg0ZC0xMWU4LTlmNzA
tMjg2ZWQ0ODhjZTcwIgotfQ=="
}
}
```

表1-8 Event 中涉及的参数解释

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

• CTS 示例事件

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

表1-9 参数说明

参数	类型	示例值	描述
User	Map	参考示例	本次请求的发起用户信息
Request	Map	参考示例	事件请求内容
Response	Map	参考示例	事件响应内容
Code	Int	204	事件响应码，例如 200、400

参数	类型	示例值	描述
service_type	String	vpc	发送方的简写, 比如 vpc, ecs 等等
resource_type	String	VPC	发送方资源类型, 比如 vm, vpn 等等
resource_name	String	workflow-2be1	资源名称, 例如 ecs 服务中某个虚拟机的名称
trace_name	String	deleteGraph	事件名称, 比如 :startServer, shutDown 等
trace_type	String	ConsoleAction	事件发生源头类型, 例如 ApiCall
record_time	string	2018/06/26 08:54:07 GMT+08:00	cts 服务接受到这条 trace 的时间
trace_id	String	69be64a7-0233-11e8-82e4-e5d37911193e	事件的唯一标识符
trace_status	String	normal	事件的状态

- DDS 示例事件

```
{
  "records": [
    {
      "event source": "dds",
      "event name": "insert",
      "region": "{region}",
      "event version": "1.0",
      "dds": {
        "size bytes": "100",
        "token": "{ \" data\": \\\"825D8C2F4D0000001529295A100474039A3412A64BA89041DC952357FB4446645F696400645D8C2F8E5BECCB6CF5370D6A0004\\\" }",
        "full document": "{ \" id\": { \" $oid\": \\\"5d8c2f8e5beccb6cf5370d6a\\\" }, \\\" name\": \\\" dds\\\", \\\" age\": { \\\" $numberDouble\\\": \\\"52.0\\\" } }",
        "ns": "{ \\\" db\\\": \\\" functiongraph\\\", \\\" coll\\\": \\\" person\\\" }"
      },
      "event source id": "e6065860-f7b8-4cca-80bd-24ef2a3bb748"
    }
  ]
}
```

表1-10 参数说明

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

- Kafka 示例事件

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

表1-11 参数说明

参数	类型	示例值	描述
event_version	String	v1.0	事件协议的版本
event_time	String	2018-01-09T07:50:50.028Z	事件发生的时间

参数	类型	示例值	描述
trigger_type	String	KAFKA	事件类型
region	String	/	Kafka 实例所在的地域
instance_id	String	81335d56-b9fe-4679-ba95-7030949cc76b	创建的 Kafka 实例的唯一标识符。
messages	String	参考示例	消息内容
topic_id	String	topic-test	消息的唯一标识符

- RabbitMQ 示例事件

```

{
  "event version": "v1.0",
  "event time": 1576737962,
  "trigger type": "RABBITMQ",
  "region": "{region}",
  "records": [
    {
      "messages": [
        "rabbitmq message1",
        "rabbitmq message2",
        "rabbitmq message3",
        "rabbitmq message4",
        "rabbitmq message5"
      ],
      "instance id": "81335d56-b9fe-4679-ba95-7030949cc76b",
      "exchange": "exchange-test"
    }
  ]
}

```

表1-12 参数说明

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

## 1.3 函数工程打包规范

### 打包规范说明

函数除了支持在线编辑代码，还支持上传 ZIP、JAR、引入 OBS 文件等方式上传代码，函数工程的打包规范说明如表 1-13 所示。

表1-13 函数工程打包规范

编程语言	JAR 包	ZIP 包	OBS 文件
Node.js	不支持该方式	<ul style="list-style-type: none"> <li>假如函数工程文件保存在“~/Code/”文件夹下，在打包的时候务必进入 Code 文件夹下选中所有工程文件进行打包，这样做的目的是：入口函数是程序执行的入口，确保解压后，入口函数所在的文件位于根目录。</li> <li>如果函数工程引入了第三方依赖，可以将第三方依赖打成 ZIP 包，在函数代码界面设置外部依赖包；也可以将第三方依赖和函数工程文件一起打包。</li> </ul>	将工程打成 ZIP 包，上传到 OBS 存储桶。
PHP	不支持该方式	<ul style="list-style-type: none"> <li>假如函数工程文件保存在“~/Code/”文件夹下，在打包的时候务必进入 Code 文件夹下选中所有工程文件进行打包，这样做的目的是：入口函数是程序执行的入口，确保解压后，入口函数所在的文件位于根目录。</li> <li>如果函数工程引入了第三方依赖，可以将第三方依赖打成 ZIP 包，在函数代码界面设置外部依赖包；也可以将第三方依赖和函数工程文件一起打包。</li> </ul>	将工程打成 ZIP 包，上传到 OBS 存储桶。

编程语言	JAR 包	ZIP 包	OBS 文件
Python 2.7	不支持该方式	<ul style="list-style-type: none"> <li>假如函数工程文件保存在“~/Code/”文件夹下，在打包的时候务必进入 Code 文件夹下选中所有工程文件进行打包，这样做的目的是：入口函数是程序执行的入口，确保解压后，入口函数所在的文件位于根目录。</li> <li>如果函数工程引入了第三方依赖，可以将第三方依赖打成 ZIP 包，在函数代码界面设置外部依赖包；也可以将第三方依赖和函数工程文件一起打包。</li> </ul>	将工程打成 ZIP 包，上传到 OBS 存储桶。
Python 3.6	不支持该方式	<ul style="list-style-type: none"> <li>假如函数工程文件保存在“~/Code/”文件夹下，在打包的时候务必进入 Code 文件夹下选中所有工程文件进行打包，这样做的目的是：入口函数是程序执行的入口，确保解压后，入口函数所在的文件位于根目录。</li> <li>如果函数工程引入了第三方依赖，可以将第三方依赖打成 ZIP 包，在函数代码界面设置外部依赖包；也可以将第三方依赖和函数工程文件一起打包。</li> </ul>	将工程打成 ZIP 包，上传到 OBS 存储桶。
Java 8	如果函数没有引用第三方件，可以直接将函数工程编译成 Jar 包。	如果函数引用第三方件，将函数工程编译成 Jar 包后，将所有依赖三方件和函数 jar 包打成 ZIP 包。	将工程打成 ZIP 包，上传到 OBS 存储桶。
Go 1.8	不支持该方式	必须在编译之后打 zip 包，编译后的动态库文件名称必须与函数执行入口的插件名称保持一致，例如：动态库名称为	将工程打成 ZIP 包，上传到 OBS 存储桶。



编程语言	JAR 包	ZIP 包	OBS 文件
		testplugin.so, 则“函数执行入口”命名为 testplugin.Handler, 其中 Handler 为入口函数。	
Go 1.x	不支持该方式	必须在编译之后打 zip 包, 编译后的二进制文件必须与执行函数入口保持一致, 例如二进制名称为 Handler, 则执行入口为 Handler。	将工程打成 ZIP 包, 上传到 OBS 存储桶。
C#	不支持该方式	必须在编译之后打 zip 包, 必须包含“工程名.deps.json”, “工程名.dll”, “工程名.runtimeconfig.json”, “工程名.pdb”和“HC.Serverless.Function.Common.dll”文件。	将工程打成 ZIP 包, 直接上传到 OBS 存储桶。
定制运行时	不支持该方式	打 zip 包, 必须包含“bootstrap”可执行引导文件。	将工程打成 ZIP 包, 直接上传到 OBS 存储桶。
Cangjie 1.0	不支持该方式	必须在编译之后打 zip 包, 编译后的二进制文件必须与执行函数入口保持一致, 例如二进制名称为 libuser_func_test_success.so, 则执行入口为 libuser_func_test_success.so。	将工程打成 ZIP 包, 上传到 OBS 存储桶。

## ZIP 工程包示例

- Nods.js 工程 ZIP 包目录示例

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

- PHP 工程 ZIP 包目录示例

Example.zip	示例工程包
--- ext	扩展库目录

--- pear	PHP扩展与应用仓库
--- index.php	入口PHP文件

● Python 工程 ZIP 包目录示例

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

● Java 工程 ZIP 包目录示例

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

● Go 工程 ZIP 包目录示例

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

● C#工程 ZIP 包目录示例

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

● Cangjie 工程 ZIP 包目录示例

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

● 定制运行时

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

## 1.4 在函数中引入动态链接库

- 函数运行环境中已经默认将代码根目录和根目录下的 lib 目录加入到 LD\_LIBRARY\_PATH 中，只需要将动态链接库放到此处即可。
- 在代码中直接修改 LD\_LIBRARY\_PATH 环境变量。
- 如果依赖的.so 文件放在其他目录，可以在配置页面设置 LD\_LIBRARY\_PATH 环境变量指明对应的目录。其中，**/opt/function/code**、**/opt/function/code/lib** 表示函数代码的工程目录。
- 如果使用了挂载文件系统中的库，可以在配置页面设置 LD\_LIBRARY\_PATH 环境变量指明挂载文件系统中对应的目录。

# 2 函数初始化入口 Initializer

## 概述

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

## 适用场景

用户函数执行调度包括以下几个阶段：

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

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

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

## 引入 `initializer` 接口的价值

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

## initializer 接口规范

各个 runtime 的 initializer 接口有以下共性：

- 无自定义参数  
Initializer 不支持用户自定义参数，只能获取 FunctionGraph 提供的 context 参数中的变量进行相关逻辑处理。
- 无返回值  
开发者无法从 invoke 的响应中获取 initializer 预期的返回值。
- 超时时间  
开发者可单独设置 initializer 的超时时间，与 handler 的超时相互独立，但最长不超过 300 秒。
- 执行时间  
运行函数逻辑的进程称之为函数实例，运行在容器内。FunctionGraph 会根据用户负载伸缩函数实例。每当有新函数实例创建时，系统会首先调用 initializer。系统保证一定 initializer 执行成功后才会执行 handler 逻辑。
- 最多成功执行一次  
FunctionGraph 保证每个函数实例启动后只会成功执行一次 initializer。如果执行失败，那么该函数实例执行失败，选取下一个实例重新执行，最多重试 3 次。一旦执行成功，在该实例的生命周期内不会再执行 initializer，收到 Invoke 请求之后只执行请求处理函数。
- initializer 入口命名  
除 Java 外，其他 runtime 的 initializer 入口命名规范与原有的执行函数命名保持一致，格式为 [文件名].[initializer 名]，其中 initializer 名可自定义。Java 需要定义一个类并实现函数计算预定义的初始化接口。
- 计量计费  
Initializer 的执行时间也会被计量，用户需要为此付费，计费方式同执行函数。

# 3 Node.js

## 3.1 开发事件函数

### 函数定义

#### 📖 说明

建议使用 Node.js 12.13 版本。

- Node.js 6.10 函数定义

Node.js 6.10 函数的接口定义如下所示。

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

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

函数执行入口格式为 “[文件名].[函数名]”。例如创建函数时设置为 `index.handler`, 那么 FunctionGraph 会去加载 `index.js` 中定义的 `handler` 函数。

- Node.js 8.10、Node.js 10.16、Node.js 12.13、Node.js 14.18 函数定义

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

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

通过 `return` 进行返回。

### Node.js 的 initializer 入口介绍

FunctionGraph 目前支持以下 Node.js 运行环境:

- Node.js 6.10 (runtime = Node.js 6)
- Node.js 8.10 (runtime = Node.js 8)
- Node.js 10.16 (runtime = Node.js 10)
- Node.js 12.13 (runtime = Node.js 12)

- Node.js14.18(runtime = Node.js14)
- Node.js16.17(runtime = Node.js16)
- Node.js18.15(runtime = Node.js18)

Initializer 入口格式为:

**[文件名].[initializer 名]**

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

在函数服务中使用 Node.js 编写 initializer 逻辑, 需要定义一个 Node.js 函数作为 initializer 入口, 一个最简单的 initializer 示例如下。

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

- 函数名  
exports.initializer 需要与实现 initializer 接口时的 Initializer 字段相对应。  
示例: 创建函数时指定的 Initializer 入口为 index.initializer, 那么 FunctionGraph 会去加载 index.js 中定义的 initializer 函数。
- context 参数  
context 参数中包含一些函数的运行时信息。例如: request id、临时 AccessKey、function meta 等。
- callback 参数  
callback 参数用于返回调用函数的结果, 其签名是 function(err, data), 与 Nodejs 中惯用的 callback 一样, 它的第一个参数是 error, 第二个参数 data。如果调用时 error 不为空, 则函数将返回 HandledInitializationError, 由于屏蔽了初始化函数的返回值, 所以 data 中的数据是无效的, 可以参考上文的示例设置为空。

## SDK 接口

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

表3-1 Context 类上下文方法说明

方法名	方法说明
getRequestID()	获取请求 ID。
getRemainingTimeInMilliseconds ()	获取函数剩余运行时间。
getAccessKey()	获取用户委托的 AccessKey (有效期 24 小时), 使用该方法需要给函数配置委托。
getSecretKey()	获取用户委托的 SecretKey (有效期 24 小时), 使用该方法需要给函数配置委托。
getUserData(string key)	通过 key 获取用户通过环境变量传入的值。

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

 **警告**

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

## 开发 Node.js 函数

如下为本地开发后上传实例，也可以直接在页面创建在线编辑。

### 步骤 1 创建函数工程

#### 1. 创建函数代码（同步形式入口函数）

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

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



```
}  
  callback(error, output);  
}
```

### 📖 说明

1. `callback` 返回的第一个参数不为 `null`，则认为函数执行失败，会返回定义在第二个参数的 HTTP 错误信息。
2. 注意，当使用 APIG 触发器时，函数返回必须使用示例中 `output` 的格式，函数 `Body` 参数仅支持返回如下几种类型的值。

`null`: 函数返回的 HTTP 响应 `Body` 为空。

`[]byte`: 函数返回的 HTTP 响应 `Body` 内容为该字节数组内容。

`string`: 函数返回的 HTTP 响应 `Body` 内容为该字符串内容。

### 2. 创建函数代码(异步形式的入口函数, 运行时 8.10 及以上支持)

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

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

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

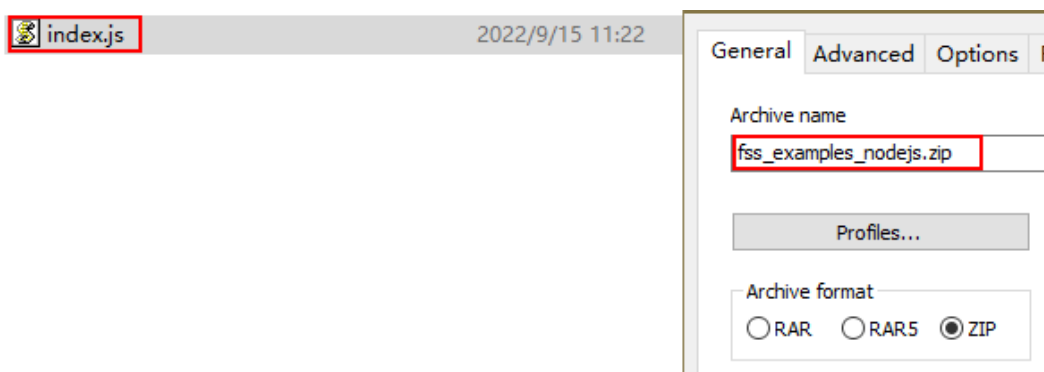
```
    }, 2000)  
  })  
  return promise  
  // anthor way  
  // res = await promise;  
  // return res  
}
```

如果期望函数先响应，随后继续执行任务。可以通过 SDK/API 调用。

### 步骤 2 工程打包

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

图3-1 打包



### 须知

本例函数工程文件保存在“~/Code/”文件夹下，在打包的时候务必进入 Code 文件夹下选中所有工程文件进行打包，这样做的目的是：定义了 FunctionGraph 函数的 index.js 是程序执行入口，确保 fss\_examples\_nodejs.zip 解压后，index.js 文件位于根目录。

### 步骤 3 创建 FunctionGraph 函数，上传程序包

登录 FunctionGraph 控制台，创建 Node.js 函数，上传 fss\_examples\_nodejs.zip 文件。

#### 说明

1. 函数设置中，图 3-2 中的 index 与步骤创建函数工程中创建的函数文件名保持一致，通过该名称找到 FunctionGraph 函数所在文件。
2. 图 3-2 中的 handler 为函数名，与步骤创建函数工程中创建的 index.js 文件中的函数名保持一致。

在函数 workflow 控制台左侧导航栏选择“函数 > 函数列表”，单击需要设置的“函数名称”进入函数详情页，选择“设置 > 常规设置”，配置“函数执行入口”参数，如图 3-2 所示。其中参数值为“index.handler”格式，“index”和“handler”支持自定义命名。

图3-2 函数执行入口参数

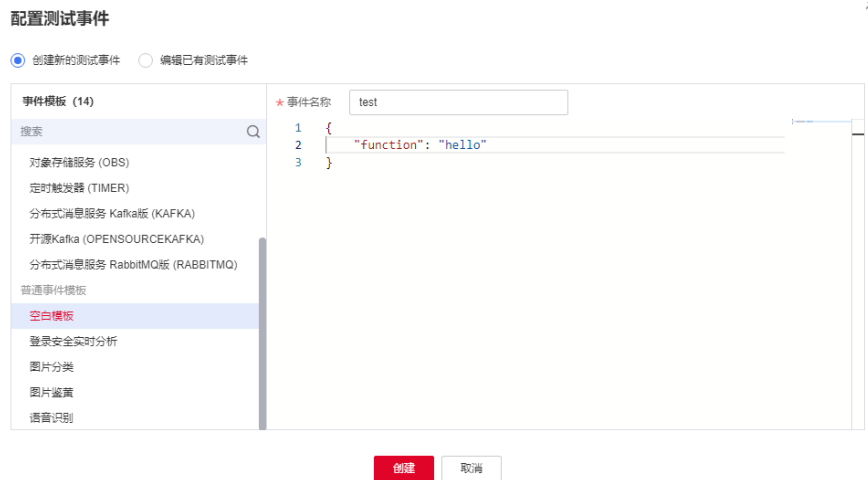


#### 步骤 4 测试函数

##### 1. 创建测试事件。

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

图3-3 配置测试事件

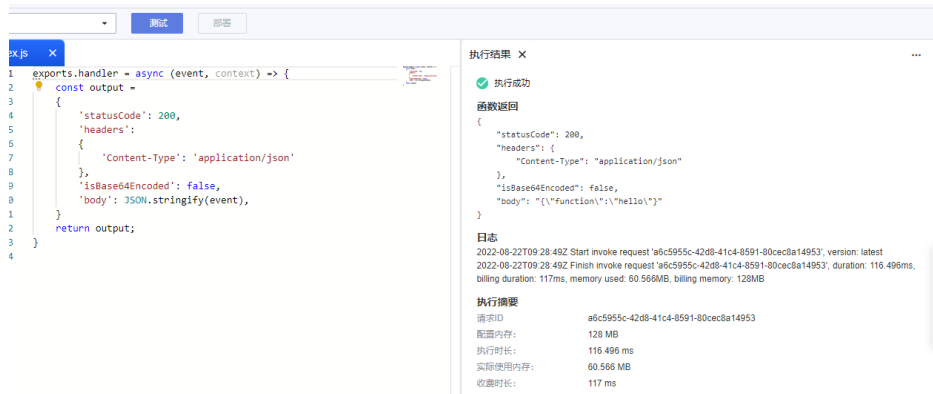


##### 2. 在函数详情页，选择已配置测试事件，单击“测试”。

#### 步骤 5 函数执行

函数执行结果分为三部分，分别为函数返回（由 `callback` 返回）、执行摘要、日志输出（由 `console.log` 或 `getLogger()`方法获取的日志方法输出），如图 3-4 所示。

图3-4 测试结果



### ---结束

## 执行结果

执行结果由 3 部分组成：函数返回、执行摘要和日志。

表3-2 执行结果说明

参数项	执行成功	执行失败
函数返回	返回函数中定义的返回信息。	返回包含错误信息和错误类型的 JSON 文件。格式如下： <pre>{  "errorMessage": "",  "errorType": ""}</pre> errorMessage: Runtime 返回的错误信息 errorType: 错误类型
执行摘要	显示请求 ID、配置内存、执行时长、实际使用内存和收费时长。	显示请求 ID、配置内存、执行时长、实际使用内存和收费时长。
日志	打印函数日志，最多显示 4KB 的日志。	打印报错信息，最多显示 4KB 的日志。

## 3.2 开发 HTTP 函数

本章节通过 HTTP 函数部署 koa 框架。

### 前提条件

1. 准备一个 bootstrap 启动文件，作为 HTTP 函数的启动文件。举例如下：

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

- /opt/function/runtime/nodejs14.18/rtsp/nodejs/bin/node: 表示 nodejs 编译环境所在路径。
- \$RUNTIME\_CODE\_ROOT: 系统变量, 表示容器中项目代码存放路径 /opt/function/code。
- index.js: 项目入口文件, 可自定义名称。

目前支持的 Nodejs 语言和对应的路径请参见表 3-3。

表3-3 Nodejs 语言对应路径

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

2. 在 Linux 机器上安装 node 环境并准备 nodejs 项目文件。

### Koa Web 应用

- a. 创建项目文件夹。

```
mkdir koa-example && cd koa-example
```

- b. 初始化 nodejs 项目, 下载 koa 框架, 文件夹中会新增 node\_modules 文件夹和 package.json、package-lock.json 文件。

```
npm init -y  
npm i koa
```

- c. 创建 index.js 文件, 在 index.js 文件中引入 koa 框架。

代码示例:

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

### 📖 说明

- HTTP 函数只能绑定 APIG/APIC 触发器，根据函数和 APIG/APIC 之间的转发协议，函数的返回合法的 http 相应报文中必须包含 body(String)、statusCode(int)、headers(Map)和 isBase64Encoded(boolean)，HTTP 函数会默认对返回结果做 Base64 编码，isBase64Encoded 默认为 true，其它框架同理。
- HTTP 函数默认开放端口为 8000。

d. 创建 bootstrap 文件。

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

3. 把项目文件和 bootstrap 文件打包成 zip 包。以 koa 框架为例：

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

## 3.3 nodejs 模板

主调函数：

```
// funcName: nodejscaller  
const { Function } = require("function-javascript-sdk");  
  
module.exports.newStateRouter = async (event, context) => {  
  func = new Function(context, "nodejsstateful", "test1");  
  await func.init();  
  instanceID = func.getInstanceID();  
  return instanceID;  
}  
  
module.exports.bindStateRouter = async (event, context) => {  
  func = new Function(context);  
  // bind  
  await func.getInstance("nodejsstateful", "test1");  
  instanceID = func.getInstanceID();  
  return instanceID;  
}  
  
module.exports.invoke = async (event, context) => {  
  var func = new Function(context);  
  // bind  
  await func.getInstance("nodejsstateful", "test1");  
  var obj = await func.invoke({});  
  var result = await obj.get();  
  return result;  
}  
  
module.exports.terminate = async (event, context) => {  
  var func = new Function(context);  
  // bind  
  await func.getInstance("nodejsstateful", "test1");  
  var obj = await func.terminate();  
  var result = await obj.get();  
  return result;  
}
```

## 3.4 制作依赖包

### 为 Nodejs 函数制作依赖包

需要先保证环境中已经安装了对应版本的 Nodejs。

为 Nodejs 8.10 安装 MySQL 依赖包，可以执行如下命令。

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

可以看到当前目录下会生成一个 `node_modules` 文件夹。

- Linux 系统

Linux 系统下可以使用以下命令生成 zip 包。

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

即可生成最终需要的依赖包。

- windows 系统

用压缩软件将 `node_modules` 目录压缩成 zip 文件即可。

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

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

```
npm install --save
```

#### 说明

不要使用 **CNPM** 命令制作 nodejs 依赖包。

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

Nodejs 其他版本制作依赖包过程与上述相同。

# 4 Python

## 4.1 开发事件函数

### 函数定义

#### 📖 说明

建议使用 Python 3.6 版本。

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

函数有明确的接口定义，如下所示。

`def handler (event, context)`

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

### Python 的 initializer 入口介绍

FunctionGraph 目前支持以下 Python 运行环境。

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

Initializer 入口格式为:

**[文件名].[initializer 名]**

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

在 FunctionGraph 中使用 Python 编写 initializer, 需要定义一个 Python 函数作为 initializer 入口, 一个最简单的 initializer (以 Python 2.7 版本为例) 示例如下。

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

- 函数名



my\_initializer 需要与实现 initializer 接口时的 Initializer 字段相对应，实现 initializer 接口时指定的 Initializer 入口为 main.my\_initializer ，那么函数服务会去加载 main.py 中定义的 my\_initializer 函数。

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

## SDK 接口

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

表4-1 Context 类上下文方法说明

方法名	方法说明
getRequestID()	获取请求 ID。
getRemainingTimeInMilliseconds ()	获取函数剩余运行时间。
getAccessKey()	获取用户委托的 AccessKey（有效期 24 小时），使用该方法需要给函数配置委托。
getSecretKey()	获取用户委托的 SecretKey（有效期 24 小时），使用该方法需要给函数配置委托。
getUserData(string key)	通过 key 获取用户通过环境变量传入的值。
getFunctionName()	获取函数名称。
getRunningTimeInSeconds ()	获取函数超时时间。
getVersion()	获取函数的版本。
getMemorySize()	分配的内存。
getCPUNumber()	获取函数占用的 CPU 资源。
getPackage()	获取函数组。
getToken()	获取用户委托的 token（有效期 24 小时），使用该方法需要给函数配置委托。
getLogger()	<p>获取 context 提供的 logger 方法，返回一个日志输出类，通过使用其 info 方法按“时间-请求 ID-输出内容”的格式输出日志。</p> <p>如调用 info 方法输出日志：</p> <pre>log = context.getLogger() log.info("test")</pre>

 **警告**

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

## 开发 Python 函数

开发 Python 函数步骤如下。

 **说明**

以下示例使用的 Python 2.7 版本。

### 步骤 1 创建函数工程

#### 1. 编写打印 helloworld 的代码

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

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

#### 2. 定义 FunctionGraph 函数

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

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

 **说明**

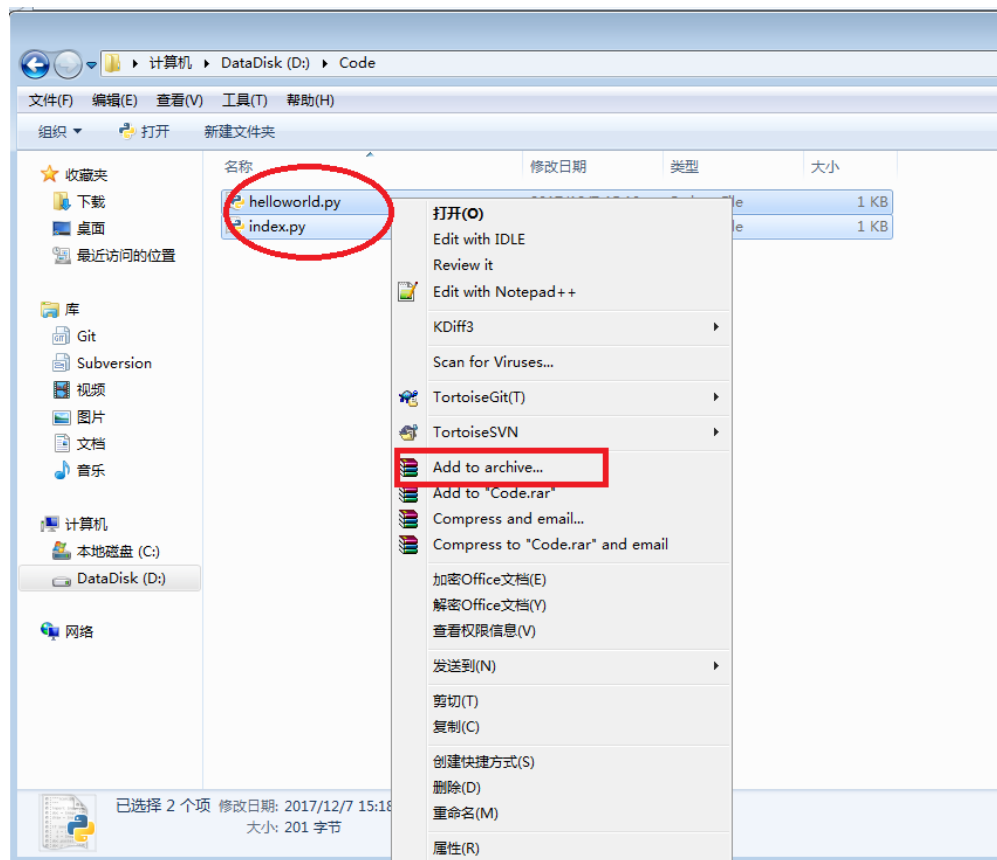
函数仅支持返回如下几种类型的值。

- None: 函数返回的 HTTP 响应 Body 为空。
- String: 函数返回的 HTTP 响应 Body 内容为该字符串内容。
- 其他: 当函数返回值的类型不为 None 和 String 时，函数会将返回值作为对象进行 json 编码，并将编码后的内容作为 HTTP 响应的 Body，同时设置响应的“Content-Type”头为“application/json”。

### 步骤 2 工程打包

函数工程创建以后，可以得到以下目录，选中工程所有文件，打包命名为“fss\_examples\_python2.7.zip”，如图 4-1 所示。

图4-1 打包



### 须知

- 本例函数工程文件保存在“~/code/”文件夹下，在打包的时候务必进入 code 文件夹下选中所有工程文件进行打包，这样做的目的是：由于定义了 FunctionGraph 函数的 index.py 是程序执行入口，确保 fss\_examples\_python2.7.zip 解压后，index.py 文件位于根目录。
- 用 Python 语言写代码时，自己创建的包名不能与 Python 标准库同名，否则会提示 module 加载失败。例如“json”、“lib”、“os”等。

### 步骤 3 创建 FunctionGraph 函数，上传程序包

登录 FunctionGraph 控制台，创建 Python 函数，上传 SDK 文件。

### 说明

1. 图 4-2 中的 index 与步骤定义 FunctionGraph 函数的文件名保持一致，通过该名称找到 FunctionGraph 函数所在文件。
2. 图 4-2 中的 handler 为函数名，与步骤定义 FunctionGraph 函数中创建的 index.py 文件中的 handler 名称保持一致。
3. 函数执行过程为：用户上传 fss\_examples\_python2.7.zip 保存在 OBS 中，触发函数后，解压缩 zip 文件，通过 index 匹配到 FunctionGraph 函数所在文件，通过 handler 匹配到 index.py 文件中定义的 FunctionGraph 函数，找到程序执行入口，执行函数。

在函数 workflow 控制台左侧导航栏选择“函数 > 函数列表”，单击需要设置的“函数名称”进入函数详情页，选择“设置 > 常规设置”，配置“函数执行入口”参数，如图 4-2 所示。其中参数值为“index.handler”格式，“index”和“handler”支持自定义命名。

图4-2 函数执行入口参数



### 步骤 4 测试函数

1. 创建测试事件。

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

图4-3 配置测试事件



2. 在函数详情页，选择已配置测试事件，单击“测试”。

### 步骤 5 函数执行

函数执行结果分为三部分，分别为函数返回（由 `callback` 返回）、执行摘要、日志输出（由 `print()` 方法获取的日志方法输出），如图 4-4 所示。

图4-4 测试结果

```

函数返回
{
  "function": "hello"
}

日志
2022-07-26T02:49:13Z Start invoke request '4bc4328d-843c-4714-afe1-71eb03a981db', version: latest
Hello world!
2022-07-26T02:49:13Z Finish invoke request '4bc4328d-843c-4714-afe1-71eb03a981db', duration: 6.228ms, billing duration: 7ms, memory used: 10.582MB, billing memory: 128MB

执行摘要
请求ID          4bc4328d-843c-4714-afe1-71eb03a981db
配置内存:       128 MB
执行时长:       6.228 ms
实际使用内存:   10.582 MB
收费时长:       7 ms
    
```

---结束

## 执行结果

执行结果由 3 部分组成：函数返回、执行摘要和日志。

表4-2 执行结果说明

参数项	执行成功	执行失败
函数返回	返回函数中定义的返回信息。	返回包含错误信息、错误类型和堆栈异常报错信息的 JSON 文件。格式如下： <pre>{   "errorMessage": "",   "errorType": "",   "stackTrace": [] }</pre> <b>errorMessage:</b> Runtime 返回的错误信息 <b>errorType:</b> 错误类型 <b>stackTrace:</b> Runtime 返回的堆栈异常报错信息
执行摘要	显示请求 ID、配置内存、执行时长、实际使用内存和收费时长。	显示请求 ID、配置内存、执行时长、实际使用内存和收费时长。
日志	打印函数日志，最多显示 4KB 的日志。	打印报错信息，最多显示 4KB 的日志。

## 4.2 python 模板

主调函数:

```
# funcName: pythoncaller

from functionsdk import Function
def newStateRouter(event, context):
    func = Function(context, "pythonstateful", "test1")
    instanceID = func.get_instance_id()
    return instanceID

def bindStateRouter(event, context):
    func = Function(context)
    # bind
    func.get_instance("pythonstateful", "test1")
    instanceID = func.get_instance_id()
    return instanceID

def invoke(event, context):
    func = Function(context)
    # bind
    func.get_instance("pythonstateful", "test1")
    obj = func.invoke("{\"key\":\"value\"}")
    result = obj.get()
    return result

def terminate(event, context):
    func = Function(context)
    # bind
    func.get_instance("pythonstateful", "test1")
    obj = func.terminate()
    result = obj.get()
    return result
```

## 4.3 制作依赖包

### 为 Python 函数制作依赖包

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

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

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

执行成功后，执行以下命令。

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

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

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

所生成的包即为最终需要的依赖包。

### 说明

如果需要安装存放在本地 wheel 安装包，直接输入：

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

# 5 Java

## 5.1 开发事件函数

### 5.1.1 Java 函数开发指南（使用 Eclipse 工具）

#### 函数定义

函数有明确的接口定义，如下：

*作用域 返回参数 函数名（函数参数，Context 参数）*

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

创建 Java 函数时，函数入口参数需要提供函数完整的名字空间，参数格式为：包名.类名.函数名。

#### Java 的 initializer 入口介绍

函数服务目前支持以下 Java 运行环境。

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

Initializer 格式为：

**[包名].[类名].[执行函数名]**

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

在函数服务中使用 Java 实现 initializer 接口，需要定义一个 java 函数作为 initializer 入口，一个最简单的 initializer 示例如下。

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



- 函数名  
my\_initializer 需要与实现 initializer 接口时的 initializer 字段相对应。  
示例：实现 initializer 接口时指定的 Initializer 入口为 com.telecom.Demo.my\_initializer，那么 FunctionGraph 会去加载 com.telecom 包，Demo 类中定义的 my\_initializer 函数。
- context 参数  
context 参数中包含一些函数的运行时信息，例如：request id、临时 AccessKey、function meta 等。

## SDK 接口

FunctionGraph 函数 JavaSDK 提供了 Event 事件接口、Context 接口和日志记录接口。

- Event 事件接口  
Java SDK 加入了触发器事件结构体定义，目前支持 CTS、LTS、TIMER、APIG、Kafka。在需要使用触发器的场景时，编写相关代码更简单。
  - a. **APIG 触发器相关方法说明**
    - i. APIGTriggerEvent 相关方法说明

表5-1 APIGTriggerEvent 相关方法说明

方法名	方法说明
isBase64Encoded()	Event 中的 body 是否是 base64 编码
getHttpMethod()	获取 Http 请求方法
getPath()	获取 Http 请求路径
getBody()	获取 Http 请求 body
getPathParameters()	获取所有路径参数
getRequestContext()	获取相关的 APIG 配置（返回表 5-2）
getHeaders()	获取 Http 请求头
getQueryStringParameters()	获取查询参数  说明 当前查询参数不支持取值为数组，如果查询参数的取值需要为数组，请自定义对应的触发器事件结构体。
getRawBody()	获取 base64 编码前的内容
getUserData()	获取 APIG 自定义认证中设置的 userdata

表5-2 APIGRequestContext 相关方法说明

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

ii. APIGTriggerResponse 相关方法说明

表5-3 APIGTriggerResponse 构造方法说明

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

表5-4 APIGTriggerResponse 相关方法说明

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

**说明**

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

b. 定时触发器相关方法说明

表5-5 TimerTriggerEvent 相关方法说明

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

c. LTS 触发器相关方法说明

表5-6 LTSTriggerEvent 相关方法说明

方法名	方法说明
getLts()	获取 LTS 消息（表 5-7）

表5-7 LTSBody 相关方法说明

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

d. CTS 触发器相关方法说明

表5-8 CTSTriggerEvent 说明

方法名	方法说明
getCTS()	获取 CTS 消息体（CTS 结构）

表5-9 CTS 结构相关方法说明

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

表5-10 User 方法说明

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

表5-11 Domain 方法说明

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

e. **Kafka** 触发器相关方法说明

表5-12 **Kafka** 触发器相关方法说明

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

 **说明**

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

• **Context** 接口

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

Context 接口说明如表 5-13 所示。

表5-13 Context 类上下文方法说明

方法名	方法说明
getRequestID()	获取请求 ID。
getRemainingTimeInMilliseconds()	获取函数剩余运行时间。
getAccessKey()	获取用户委托的 AccessKey（有效期 24 小时），使用该方法需要给函数配置委托。
getSecretKey()	获取用户委托的 SecretKey（有效期 24 小时），使用该方法需要给函数配置委托。
getUserData(string key)	通过 key 获取用户通过环境变量传入的值。
getFunctionName()	获取函数名称。
getRunningTimeInSeconds()	获取函数超时时间。
getVersion()	获取函数的版本。
getMemorySize()	分配的内存。

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

 **警告**

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

- 日志接口

Java SDK 日志接口日志说明如表 5-14 所示。

表5-14 日志接口说明表

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

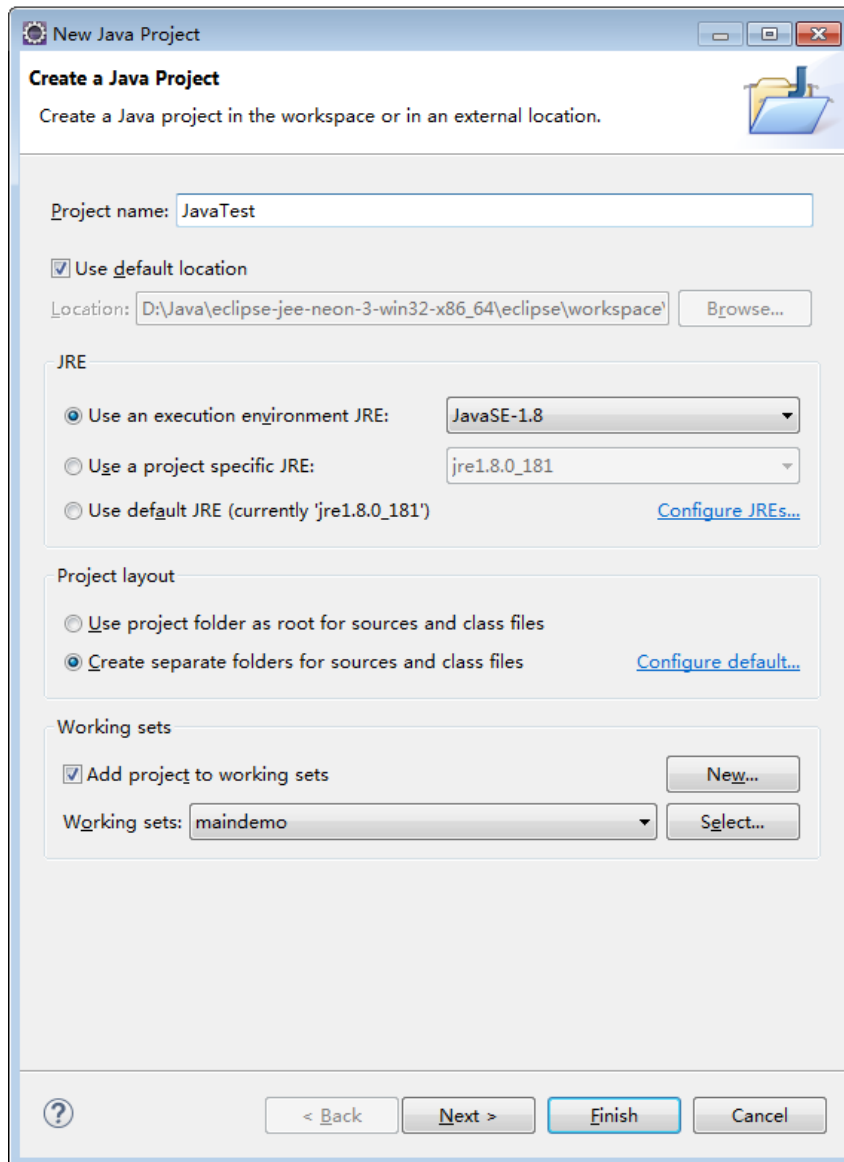
## 开发 Java 函数

开发 Java 函数，以下给出 demo 实例步骤（使用 Eclipse 工具）：

### 步骤 1 创建函数工程

1. 配置 Eclipse，创建 java 工程 JavaTest，如图 5-1 所示。

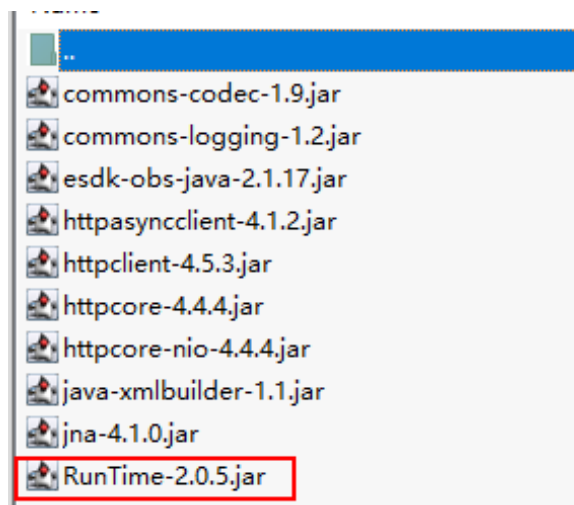
图5-1 创建工程



2. 添加工程依赖

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

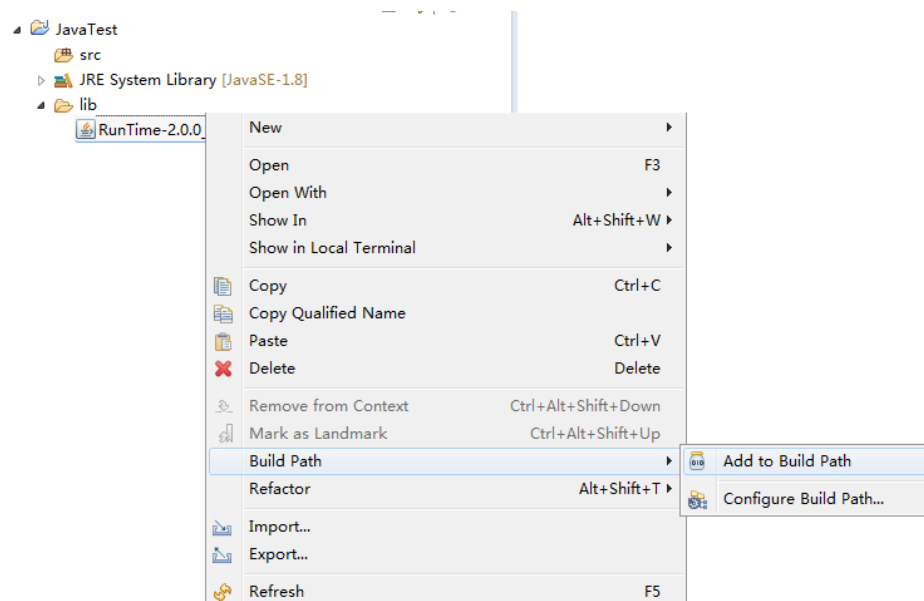
图5-2 下载 SDK 解压



### 3. 配置依赖

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

图5-3 配置依赖



### 步骤 2 创建本地函数

1. 创建包 com.telecom.demo，并在包下创建 TriggerTests 类。
2. 在 TriggerTests.java 中定义函数运行入口，示例代码如下，如图 5-4 所示。

```
package com.telecom.demo;

import java.io.UnsupportedEncodingException;
```



```
import java.util.HashMap;
import java.util.Map;

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

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

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

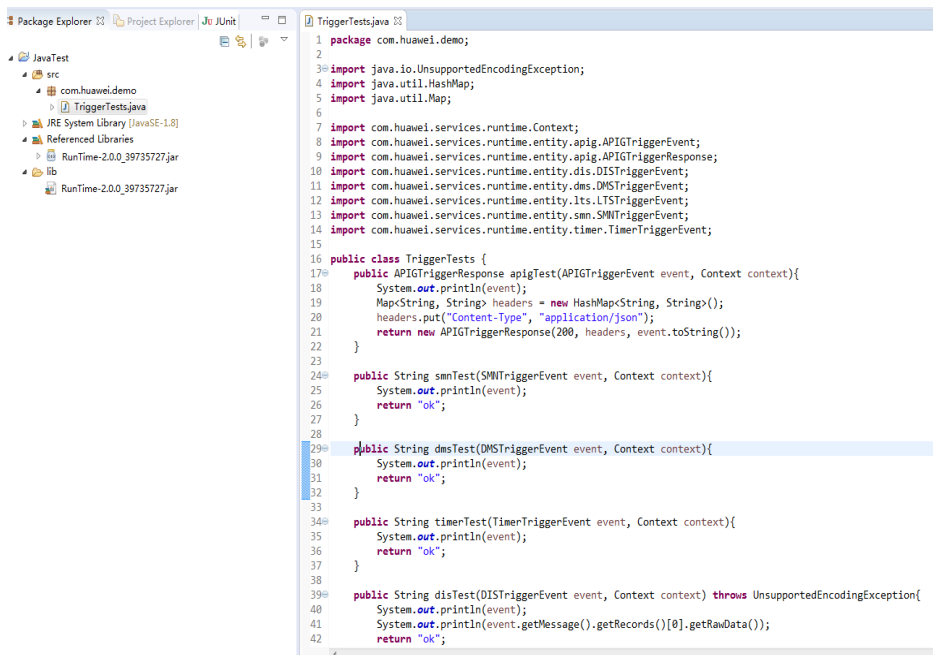
    public String dmsTest(DMSTriggerEvent event, Context context){
        System.out.println(event);
        return "ok";
    }

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

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

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

图5-4 定义函数运行入口



```
1 package com.huawei.demo;
2
3 import java.io.UnsupportedEncodingException;
4 import java.util.HashMap;
5 import java.util.Map;
6
7 import com.huawei.services.runtime.Context;
8 import com.huawei.services.runtime.entity.apig.APIGTriggerEvent;
9 import com.huawei.services.runtime.entity.apig.APIGTriggerResponse;
10 import com.huawei.services.runtime.entity.dis.DISTriggerEvent;
11 import com.huawei.services.runtime.entity.dms.DMSTriggerEvent;
12 import com.huawei.services.runtime.entity.its.LTSTTriggerEvent;
13 import com.huawei.services.runtime.entity.smn.SMNTriggerEvent;
14 import com.huawei.services.runtime.entity.timer.TimerTriggerEvent;
15
16 public class TriggerTests {
17     public APIGTriggerResponse apigTest(APIGTriggerEvent event, Context context){
18         System.out.println(event);
19         Map<String, String> headers = new HashMap<String, String>();
20         headers.put("Content-Type", "application/json");
21         return new APIGTriggerResponse(200, headers, event.toString());
22     }
23
24     public String smnTest(SMNTriggerEvent event, Context context){
25         System.out.println(event);
26         return "ok";
27     }
28
29     public String dmsTest(DMSTriggerEvent event, Context context){
30         System.out.println(event);
31         return "ok";
32     }
33
34     public String timerTest(TimerTriggerEvent event, Context context){
35         System.out.println(event);
36         return "ok";
37     }
38
39     public String disTest(DISTriggerEvent event, Context context) throws UnsupportedEncodingException{
40         System.out.println(event);
41         System.out.println(event.getMessage().getRecords()[0].getRawData());
42         return "ok";
43     }
44 }
```

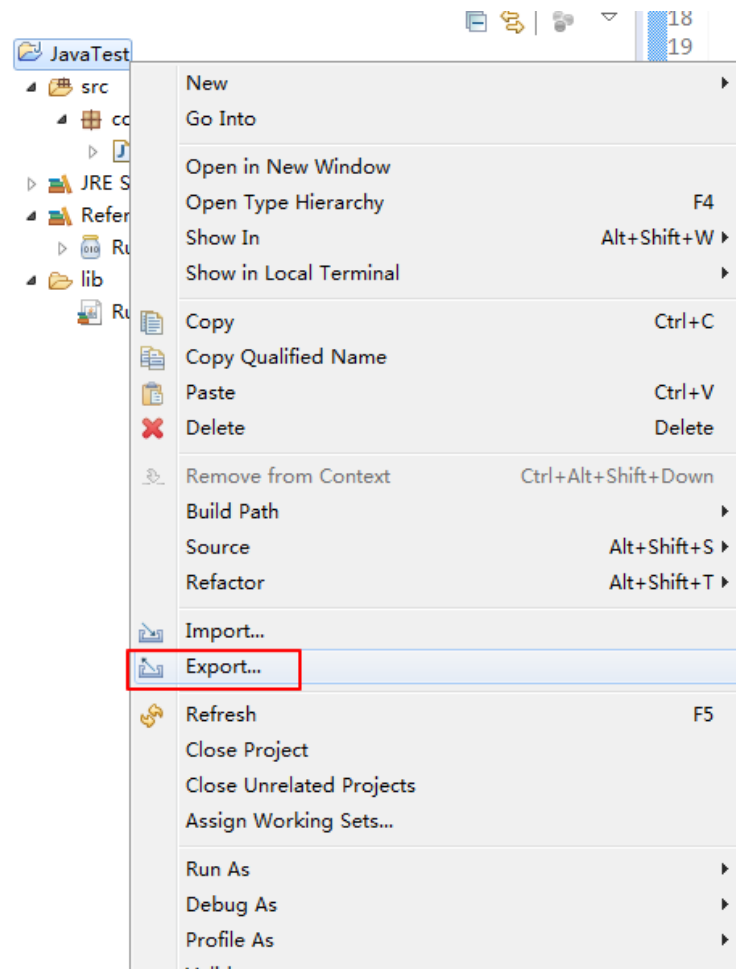
**说明**

上图所示的代码中添加了多个入口函数，分别使用了不同的触发器事件类型。

**步骤 3 工程打包**

1. 右击工程，选择“Export”，如图 5-5 所示。

图5-5 打包



2. 选择导出为 jar，设置导出目录，如图 5-6、图 5-7 所示。

图5-6 选择

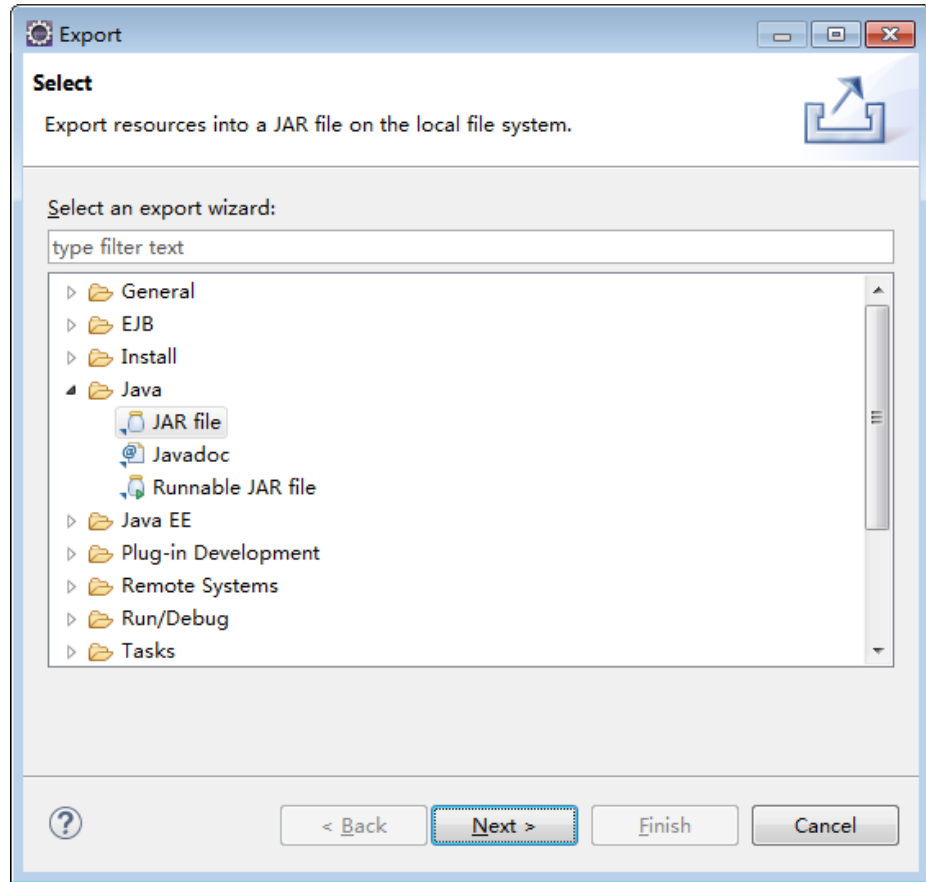
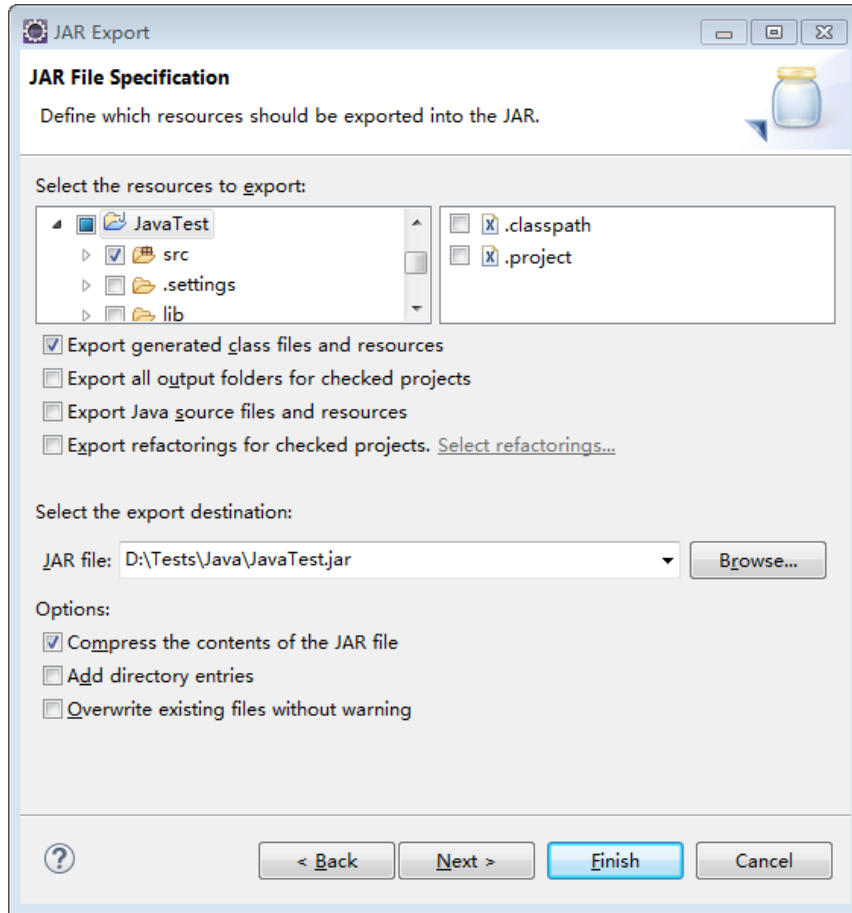


图5-7 导出



步骤 4 创建 Java 函数，上传程序包。

步骤 5 测试函数

1. 创建测试事件。

在事件模板中选择“API 网关服务 (APIG 专享版)”，并保存。

2. 单击“测试”并执行。

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

3. 创建一个 APIG 触发器。
4. 访问调用 URL，如图 5-8 所示。

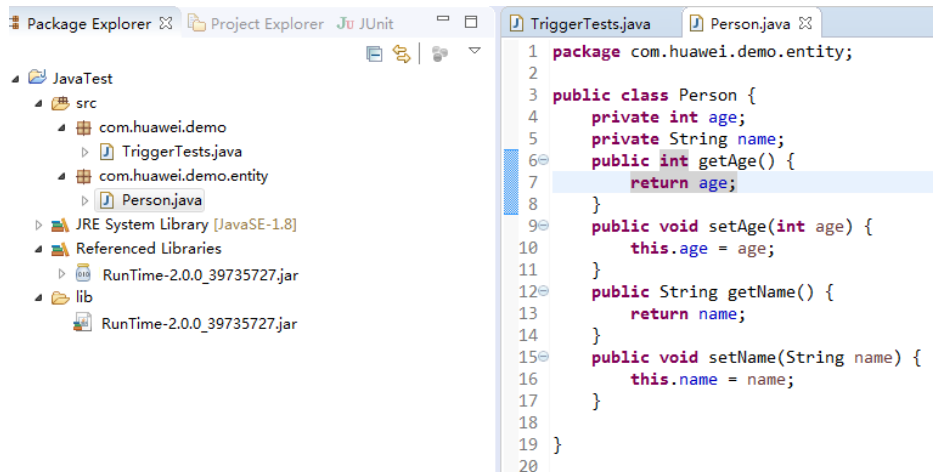
图5-8 访问调用 URL



把入口改成 com.telecom.demo.TriggerTests.smnTest，并把事件模板改成 smn。

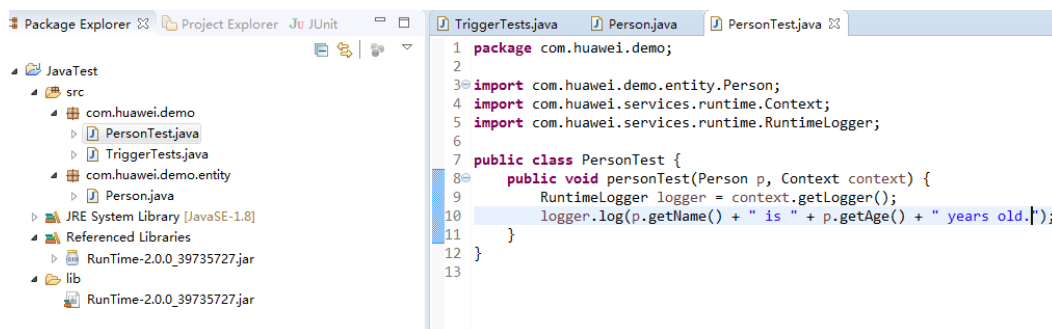
步骤 6 在 Java 中使用自定义参数类型  
在工程中新建 Person 类，如图 5-9 所示。

图5-9 新建 Person 类



新建 PersonTest.java，并在其中加入入口函数，如图 5-10 所示。

图5-10 新建 PersonTest.java



导出新包后，先在函数配置详情界面上传 JAR 包，再修改函数执行入口为“com.telecom.demo.PersonTest.personTest”并保存。

图5-11 上传 JAR 包



打开配置测试事件对话框，选择空白模板，输入测试事件内容。

单击“创建”后执行测试。

---结束

## 执行结果

执行结果由 3 部分组成：函数返回、执行摘要和日志。

表5-15 执行结果说明

参数项	执行成功	执行失败
函数返回	返回函数中定义的返回信息。	返回包含错误信息和堆栈异常报错信息的 JSON 文件。格式如下： <pre>{   "errorMessage": "",   "stackTrace": [] }</pre> errorMessage: Runtime 返回的错误信息 stackTrace: Runtime 返回的堆栈异常报错信息
执行摘要	显示请求 ID、配置内存、执行时长、实际使用内存和收费时长。	显示请求 ID、配置内存、执行时长、实际使用内存和收费时长。
日志	打印函数日志，最多显示 4KB 的日志。	打印报错信息，最多显示 4KB 的日志。

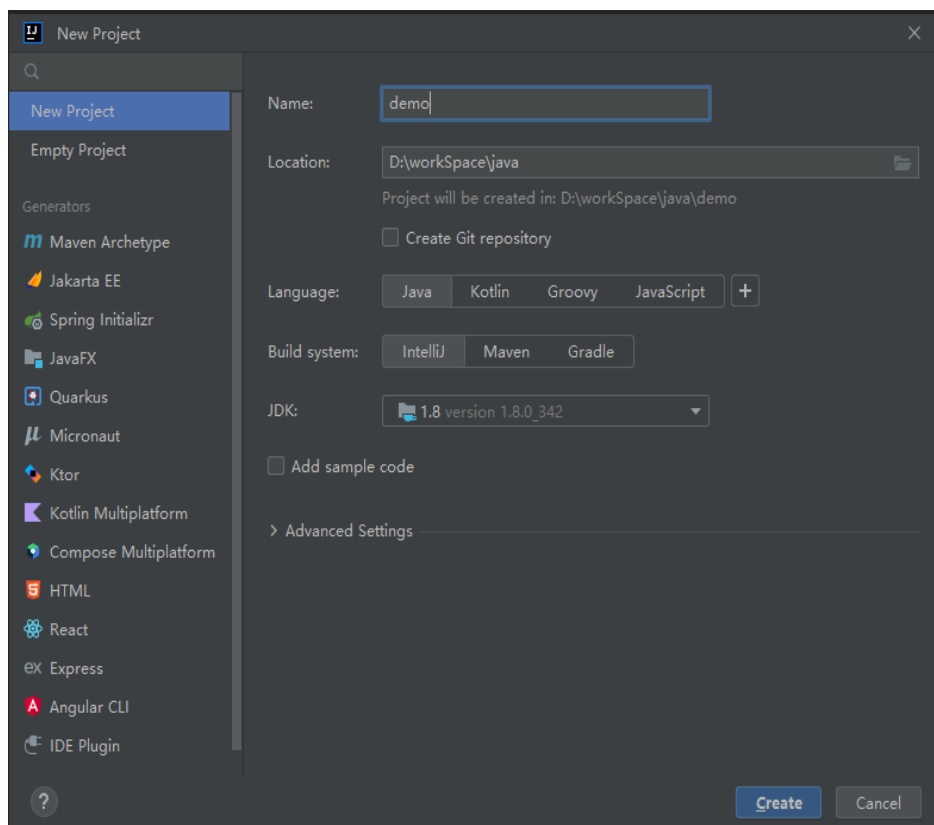
## 5.1.2 Java 函数开发指南（使用 IDEA 工具普通 Java 项目）

开发 Java 函数，以下给出 Demo 实例步骤：

### 步骤 1 创建函数工程

1. 配置 idea，创建 java 工程 JavaTest，如图图 5-12 所示

图5-12 创建工程

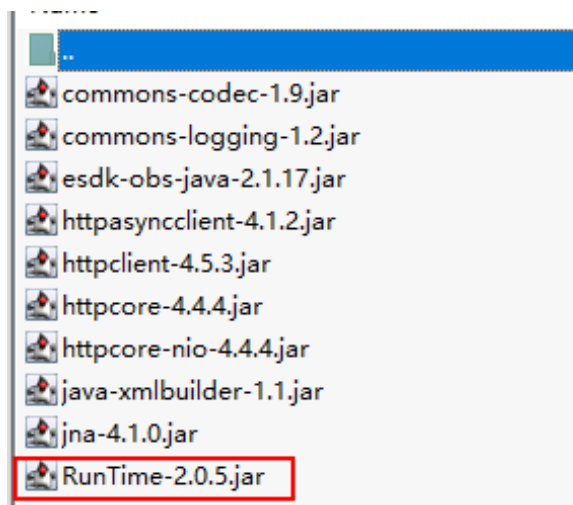


2. 添加工程依赖

根据 Java SDK 下载提供的 SDK 地址，下载 JavaRuntime SDK 到本地开发环境解压，如图 5-13 所示。



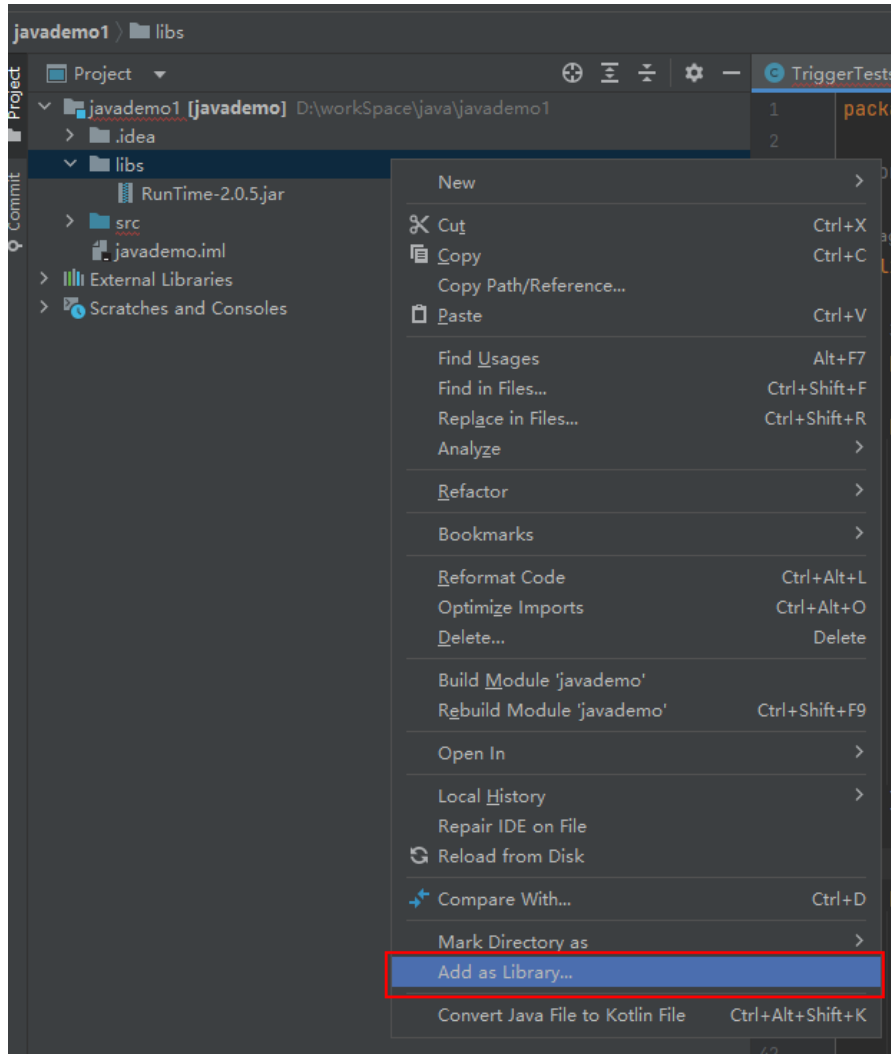
图5-13 下载 SDK 解压



3. 配置依赖

在工程目录下创建 lib 目录，将 zip 中的 Runtime2.0.5.jar 和代码所需要的三方依赖包拷贝到该目录，并把该 jar 添加为工程依赖。如图 5-14 所示：

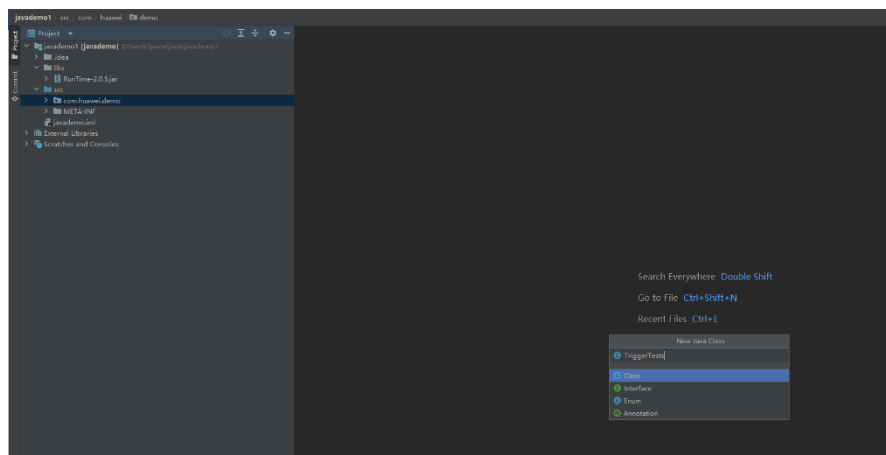
图5-14 配置依赖



## 步骤 2 创建本地函数

1. 创建包 `com.telecom.demo`，并在包下创建 `TriggerTests` 类，如图图 5-15 所示。

图5-15 创建 TriggerTests 类



2. 在 TriggerTests.java 中定义函数运行入口，示例代码如下，如图 5-16。

```
package com.telecom.demo;

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

import com.telecom.services.runtime.Context;
import com.telecom.services.runtime.entity.apig.APIGTriggerEvent;
import com.telecom.services.runtime.entity.apig.APIGTriggerResponse;
import com.telecom.services.runtime.entity.dis.DISTTriggerEvent;
import com.telecom.services.runtime.entity.dms.DMSTTriggerEvent;
import com.telecom.services.runtime.entity.lts.LTSTTriggerEvent;
import com.telecom.services.runtime.entity.smn.SMNTriggerEvent;
import com.telecom.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(DMSTTriggerEvent event, Context context){
        System.out.println(event);
        return "ok";
    }

    public String timerTest(TimerTriggerEvent event, Context context){
```

```
        System.out.println(event);
        return "ok";
    }

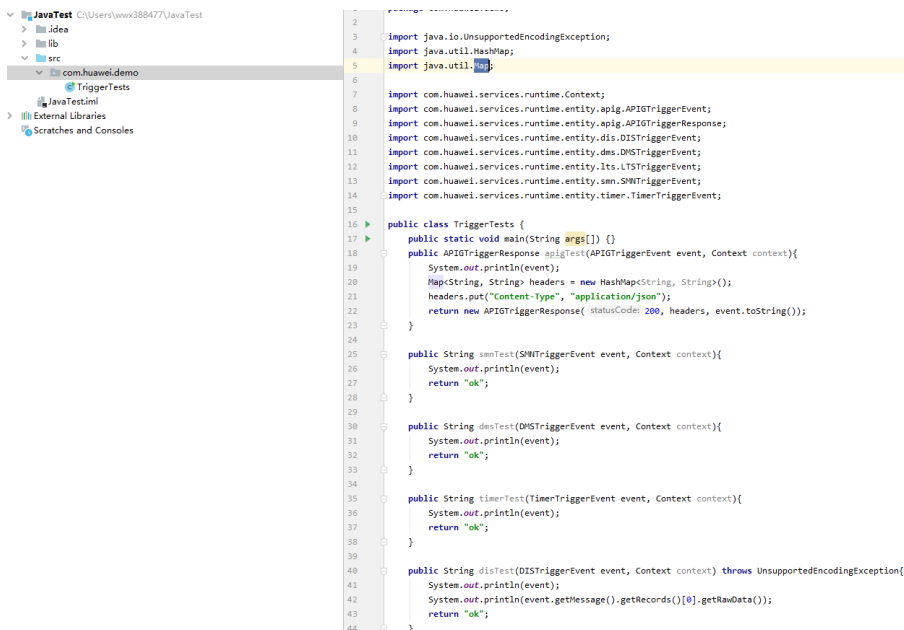
    public String disTest(DISTriggerEvent event, Context context) throws
    UnsupportedEncodingException{
        System.out.println(event);
        System.out.println(event.getMessage().getRecords()[0].getRawData());
        return "ok";
    }

    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";
    }
}
```

**说明**

普通 java 项目需要通过 Artifacts 来进行编译，需要定义一个 main 函数。

图5-16 定义函数运行入口



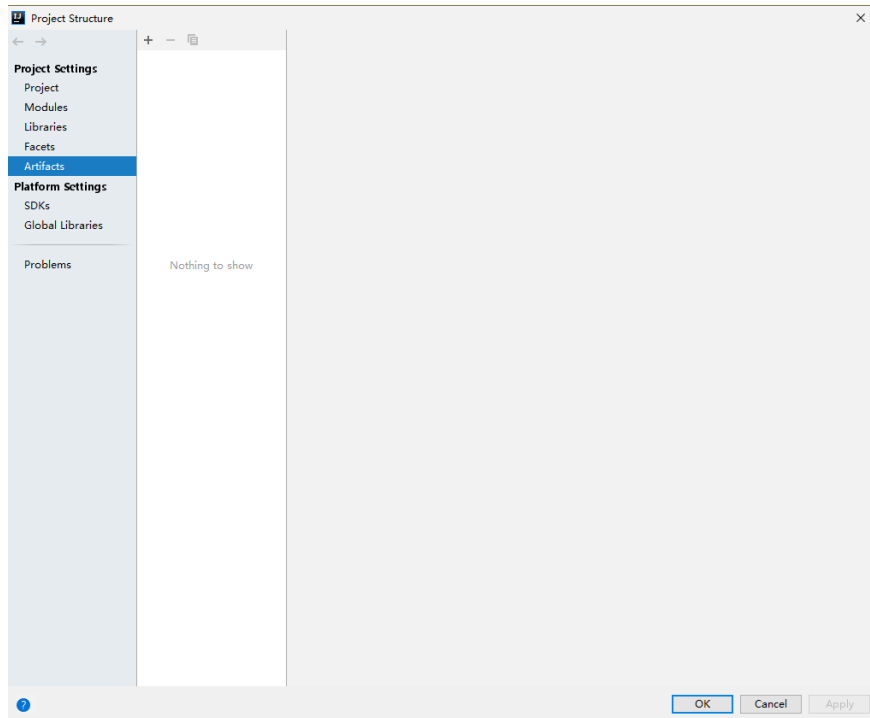
**说明**

上图所示的代码中添加了多个入口函数，分别使用了不同的触发器事件类型。

**步骤 3 工程打包**

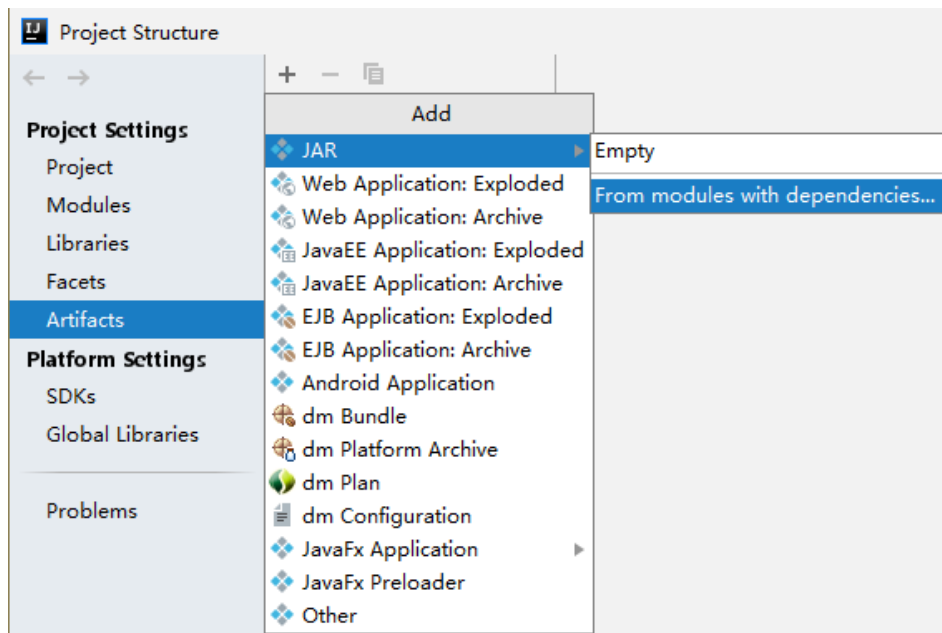
1. 单击 File->Project Structure 打开 Project Structure 窗口，如图 5-17 所示。

图5-17 Project Structure



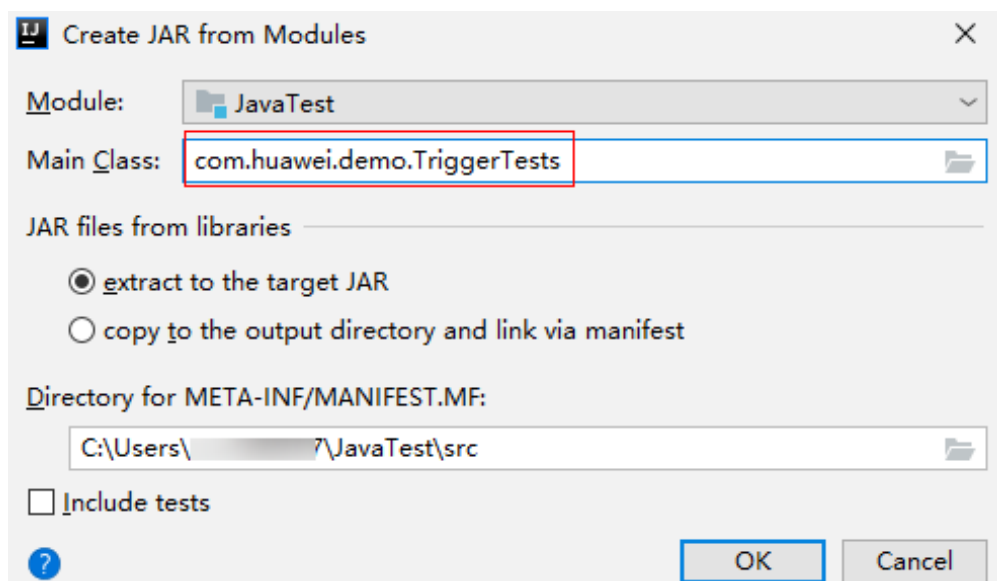
2. 选择上图中的“Artifacts”，单击“+”，进入添加“Artifacts”窗口，如图 5-18 所示。

图5-18 添加 Artifacts



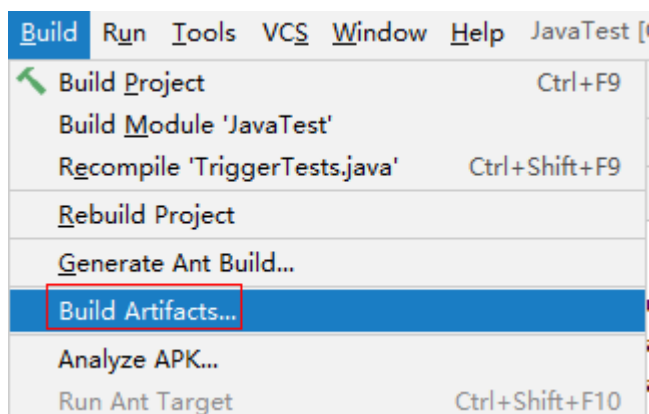
3. 添加"Main Class"，如图 5-19 所示。

图5-19 添加 Main Class



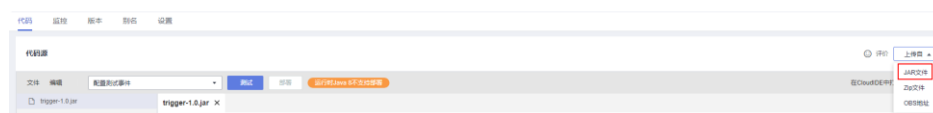
4. 单击"Build"->"Build Artifacts"来编译 Jar 包，如图 5-20 所示。

图5-20 Build Artifacts



- 步骤 4 创建 Java 函数，使用 Jar 包的方式上传代码包，如图 5-21 所示。

图5-21 上传 jar 包



- 步骤 5 测试函数

1. 创建测试事件。  
在事件模板中选择“timer-event-template”，并保存。

- 2. 单击“测试”并执行。

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

---结束

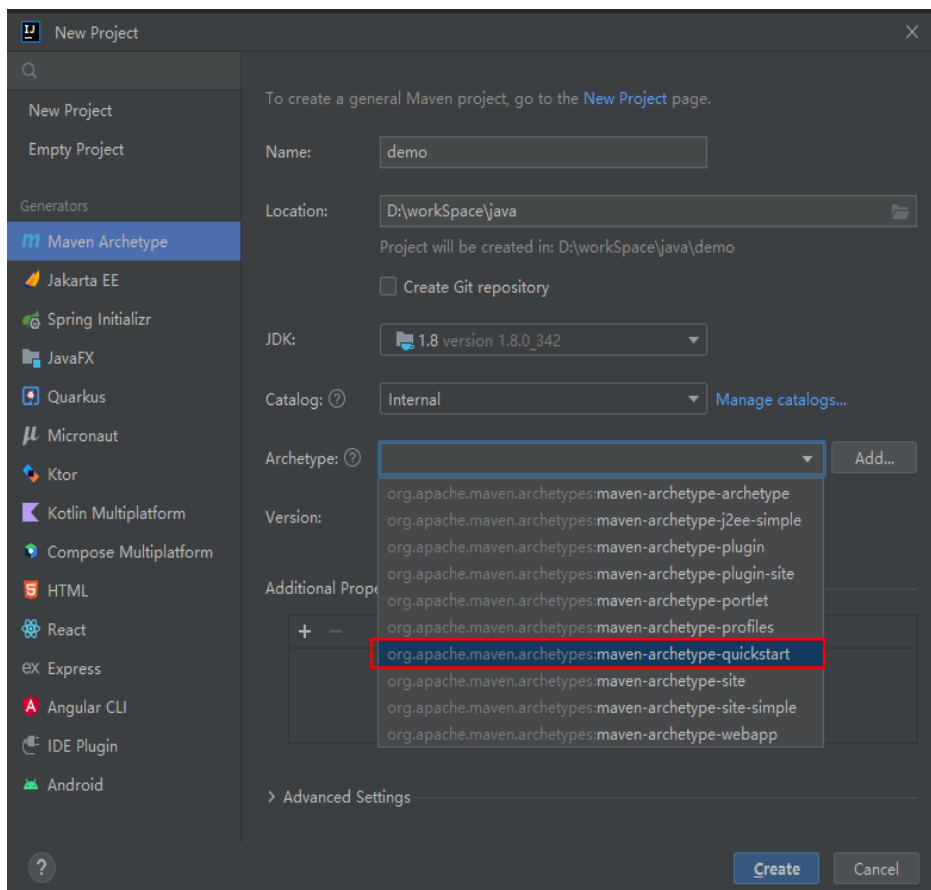
### 5.1.3 Java 函数开发指南（使用 IDEA 工具 maven 项目）

开发 Java 函数，以下给出 Demo 实例步骤：

#### 步骤 1 创建函数工程

- 1. 配置 idea，创建 maven 工程，如图 5-22 所示。

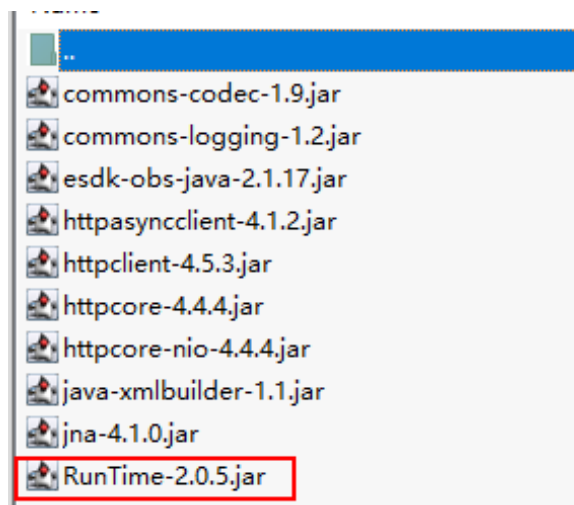
图5-22 创建工程



- 2. 添加工程依赖。

根据 Java SDK 下载提供的 SDK 地址，下载 JavaRuntime SDK 到本地开发环境解压，如图 5-23 所示。

图5-23 下载 SDK 解压



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

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

4. 在 pom.xml 中配置 dependency，如下所示。

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

5. 配置其他的依赖包，以 obs 依赖包为例，如下所示。

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

6. 在 pom.xml 中添加插件用来将代码和依赖包打包到一起。

```
<build>  
<plugins>  
<plugin>  
<artifactId>maven-assembly-plugin</artifactId>  
<configuration>  
<descriptorRefs>  
<descriptorRef>jar-with-dependencies</descriptorRef>  
</descriptorRefs>  
<archive>  
<manifest>  
<mainClass>com.telecom.demo.TriggerTests</mainClass>  
</manifest>  
</archive>  
<finalName>${project.name}</finalName>  
</configuration>  
</executions>
```



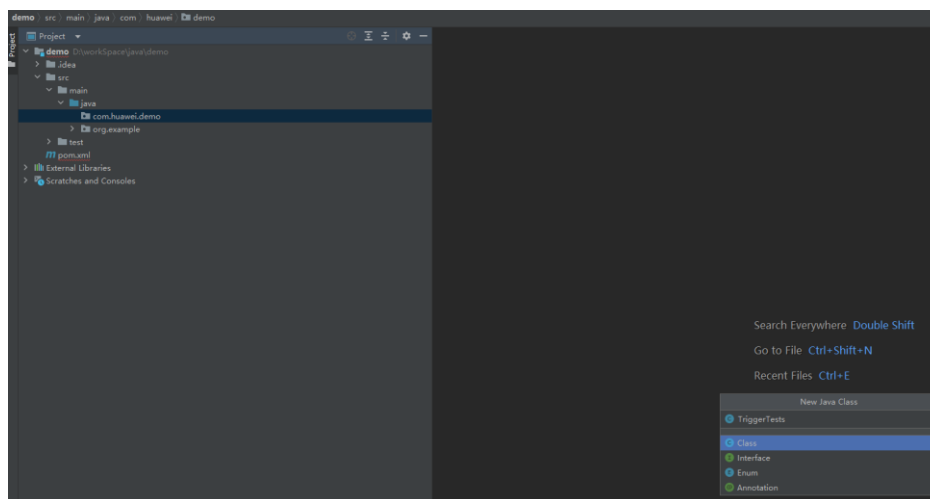
```
<execution>
  <id>make-assembly</id>
  <phase>package</phase>
  <goals><goal>single</goal>
  </goals>
</execution>
</executions>
</plugin>
</plugins>
</build>
```

请把 mainClass 替换为真实的类。

## 步骤 2 创建本地函数

1. 创建包 com.telecom.demo，并在包下创建 TriggerTests 类，如图图 5-24 所示。

图5-24 创建 TriggerTests 类



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

```
package com.telecom.demo;

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

import com.telecom.services.runtime.Context;
import com.telecom.services.runtime.entity.apig.APIGTriggerEvent;
import com.telecom.services.runtime.entity.apig.APIGTriggerResponse;
import com.telecom.services.runtime.entity.dis.DISTTriggerEvent;
import com.telecom.services.runtime.entity.dms.DMSTTriggerEvent;
import com.telecom.services.runtime.entity.lts.LTSTTriggerEvent;
import com.telecom.services.runtime.entity.smn.SMNTriggerEvent;
import com.telecom.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";
    }
}
```

### 步骤 3 工程编译打包。

在命令行窗口执行如下命令进行编译打包。

```
mvn package assembly:single
```

编译完成后在 target 目录会生成一个 demo-jar-with-dependencies.jar。

### 步骤 4 创建 java 函数，上传 jar 包。

### 步骤 5 测试函数

#### 1. 创建测试事件。

在事件模板中选择“timer-event-template”，并保存。

#### 2. 单击“测试”并执行。

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

----结束

## 5.2 java 模板

主调函数:

```
// funcName: javacaller
import com.google.gson.JsonObject;
import com.telecom.function.Function;
import com.telecom.function.runtime.StateReader;
import com.telecom.services.runtime.Context;
import com.telecom.function.ObjectRef;

public class JavaHandler {
    // 初始化, 调用 new Function instanceName 为 test1 为 javastateful 函数初始化状态路由
    public String newStateRouter(Object event, Context context) {
        Function func = new Function(context, "javastateful", "test1");
        String instanceID = func.getInstanceID();
        return instanceID;
    }

    // 绑定已创建的状态路由
    public String bindStateRouter(Object event, Context context) {
        Function func = new Function(context);
        // bind
        func.getInstance("javastateful", "test1");
        String instanceID = func.getInstanceID();
        return instanceID;
    }

    // 绑定路由后进行调用
    public Object invoke(Object event, Context context) {
        Function func = new Function(context);
        // bind
        func.getInstance("javastateful", "test1");
        ObjectRef<Object> obj = func.invoke("{\"key\":\"value\"}");
        // 通过 object 对象获取执行结果
        Object result = obj.get();
        return result;
    }

    // 删除状态实例
    public Object terminate(Object event, Context context) {
        Function func = new Function(context);
        // bind
        func.getInstance("javastateful", "test1");
        ObjectRef<Object> obj = func.terminate();
        // 通过 object 对象获取执行结果
        Object result = obj.get();
        return result;
    }
}
```

# 6 Go

## 6.1 开发事件函数

### 函数定义

函数有明确的接口定义，如下所示：

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

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

### SDK 接口

FunctionGraph 函数 GoSDK 提供了 Event 事件接口、Context 接口和日志记录接口。

- Event 事件接口  
Go SDK 加入了触发器事件结构体定义，目前支持 CTS、DDS、Kafka、LTS、TIMER、APIG。在需要使用触发器的场景时，编写相关代码更简单。
  - a. **APIG 触发器相关字段说明**
    - i. APIGTriggerEvent 相关字段说明

表6-1 APIGTriggerEvent 相关字段说明

字段名	字段描述
IsBase64Encoded	Event 中的 body 是否是 base64 编码
HttpMethod	Http 请求方法
Path	Http 请求路径
Body	Http 请求 body
PathParameters	所有路径参数
RequestContext	相关的 APIG 配置（表 6-2）
Headers	Http 请求头
QueryStringParameters	查询参数

字段名	字段描述
UserData	APIG 自定义认证中设置的 userdata

表6-2 APIGRequestContext 相关字段说明

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

ii. APIGTriggerResponse 相关字段说明

表6-3 APIGTriggerResponse 相关字段说明

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

 说明

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

b. KAFKA 触发器相关字段说明

表6-4 KAFKATriggerEvent 相关字段说明

字段名	字段描述
InstanceId	实例 ID
Records	消息记录 (表 6-5)
TriggerType	触发器类型, 返回 KAFKA
Region	region
EventTime	事件发生时间, 秒数
EventVersion	事件版本

表6-5 KAFKARecord 相关字段说明

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

c. 定时触发器相关字段说明

表6-6 TimerTriggerEvent 相关字段说明

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

d. LTS 触发器相关字段说明

表6-7 LTSTriggerEvent 相关字段说明

字段名	字段描述
Lts	LTS 消息（表 6-8）

表6-8 LTSBody 相关字段说明

字段名	字段描述
Data	LTS 原始消息

 说明

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

e. CTS 触发器相关字段说明

表6-9 CTSTriggerEvent 字段说明

字段名	字段说明
CTS	CTS 消息体（表 6-10）

表6-10 CTS 结构相关字段说明

字段名	字段描述
Time	事件产生时间
User	触发该事件的用户信息（表 6-11）
Request	事件请求内容
Response	事件响应内容
Code	响应码
ServiceType	事件触发的服务名称
ResourceType	事件触发的资源类型
ResourceName	事件触发的资源名称
ResourceId	事件触发资源的唯一标识
TraceName	事件名称
TraceType	事件触发的方式(如 ConsoleAction: 代表前台操作)
RecordTime	CTS 服务接收事件时间
TraceId	当前事件的唯一标识
TraceStatus	事件状态

表6-11 User 字段说明

字段名	字段描述
Name	用户名（同一账号可以创建多个子用户）
Id	用户 ID
Domain	账号信息（表 6-12）

表6-12 Domain 字段说明

字段名	字段描述
Name	账号名称
Id	账号 ID

**说明**

1. 例如使用 APIG 触发器时，只需要把入口函数（假如函数名为 handler）的第一个参数按照如下方式设置：handler(APIGTriggerEvent event, Context context)。
  2. 关于所有 TriggerEvent，上面提到的 TriggerEvent 方法均有与之对应的 set 方法，建议在本地调试时使用；DIS 和 LTS 均有对应的 getRawData()方法，但无与之相应的 setRawData()方法。
- Context 接口  
Context 接口提供函数获取函数执行上下文，例如，用户委托的 AccessKey/SecretKey、当前请求 ID、函数执行分配的内存空间、CPU 数等。  
Context 接口说明如表 6-13 所示。

表6-13 Context 类上下文方法说明

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

**警告**

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

- 日志接口 Go SDK 日志接口日志说明如表 6-14 所示。



表6-14 日志接口说明表

方法名	方法说明
RuntimeLogger()	<ul style="list-style-type: none"> <li>记录用户输入日志对象，包含方法如下：Logf(format string, args ...interface{})</li> <li>该方法会将内容输出到标准输出，格式：“时间-请求 ID-输出内容”，示例如下： 2017-10-25T09:10:03.328Z 473d369d-101a-445e-a7a8-315cca788f86 test log output。</li> </ul>

## 开发 Go 函数

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

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

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

```
$ mkdir -p /home/fssgo
```

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

```
$ export GO111MODULE="on"
```

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

```
$ go mod init test
```

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

```
module test

go 1.14

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

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

**步骤 4** 在目录“/home/fssgo”下创建函数文件，并实现如下接口：

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

其中 payload 为客户端请求的 body 数据，ctx 为 FunctionGraph 函数服务提供的运行时上下文对象，具体提供的方法可以参考表 6-13

### 须知

1. 如果函数返回的 `error` 参数不是 `nil`，则会认为函数执行失败。
2. 如果函数返回的 `error` 为 `nil`，FunctionGraph 函数服务仅支持返回如下几种类型的值。  
`nil`: 返回的 HTTP 响应 Body 为空。  
`[]byte`: 返回的 HTTP 响应 Body 内容为该字节数组内容。  
`string`: 返回的 HTTP 响应 Body 内容为该字符串内容。  
其它: FunctionGraph 函数服务会将返回值作为对象进行 json 编码，并将编码后的内容作为 HTTP 响应的 Body，同时设置响应的 "Content-Type" 头为 "application/json"。
3. 上面的例子是 APIG 触发器的事件类型，如果是其他触发器类型需要修改 `main` 函数的内容，例如 `cts` 触发器修改为 `runtime.Register(CtsTest)`，目前只支持注册一个入口。

### 步骤 5 编译和打包

函数代码编译完成后，按照如下方式编译和打包。

1. 编译

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

#### 📖 说明

`handler` 可以自定义，后面作为函数入口

2. 打包:

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

----结束

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

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

```
$ mkdir -p /home/fssgo/src/telecomcloud.com  
$ unzip functiongraph-go-runtime-sdk-1.0.1.zip -d /home/fssgo/src/telecomcloud.com
```

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

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

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

```
package main  
  
import (  
    "fmt"  
    "telecomcloud.com/go-runtime/go-api/context"  
    "telecomcloud.com/go-runtime/pkg/runtime"
```

```
"telecomcloud.com/go-runtime/events/apig"  
"telecomcloud.com/go-runtime/events/cts"  
"telecomcloud.com/go-runtime/events/dds"  
"telecomcloud.com/go-runtime/events/dis"  
"telecomcloud.com/go-runtime/events/kafka"  
"telecomcloud.com/go-runtime/events/lts"  
"telecomcloud.com/go-runtime/events/smn"  
"telecomcloud.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)
}
```

### 须知

1. 如果函数返回的 `error` 参数不是 `nil`，则会认为函数执行失败。
2. 如果函数返回的 `error` 为 `nil`，FunctionGraph 函数服务仅支持返回如下几种类型的值。
  - `nil`: 返回的 HTTP 响应 Body 为空。
  - `[]byte`: 返回的 HTTP 响应 Body 内容为该字节数组内容。
  - `string`: 返回的 HTTP 响应 Body 内容为该字符串内容。
  - 其它: FunctionGraph 函数服务会将返回值作为对象进行 json 编码，并将编码后的内容作为 HTTP 响应的 Body，同时设置响应的 "Content-Type" 头为 "application/json"。
3. 上面的例子是 APIG 触发器的事件类型，如果是其他触发器类型需要修改 `main` 函数的内容，例如 `cts` 触发器修改为 `runtime.Register(CtsTest)`，目前只支持注册一个入口。

### 步骤 3 编译和打包

函数代码编译完成后，按照如下方式编译和打包。

1. 设置 `GOROOT` 和 `GOPATH` 环境变量：

```
$ export GOROOT=/usr/local/go （假设 Go 安装到了 /usr/local/go 目录）
$ export PATH=$GOROOT/bin:$PATH
$ export GOPATH=/home/fssgo
```
2. 编译：

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

#### 📖 说明

`handler` 可以自定义，后面作为函数入口

3. 打包：

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

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

#### 📖 说明

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

- 测试函数
  - a. 创建测试事件。  
在函数详情页，单击“配置测试事件”，弹出“配置测试事件”页，输入测试信息如图 6-1 所示，单击“创建”。

图6-1 配置测试事件



- b. 在函数详情页，选择已配置测试事件，单击“测试”。
- 函数执行
  - 函数执行结果分为三部分，分别为函数返回（由 `callback` 返回）、执行摘要、日志输出（由 `fmt.Println()`方法获取的日志方法输出）。

---结束

## 执行结果

执行结果由 3 部分组成：函数返回、执行摘要和日志。

表6-15 执行结果说明

参数项	执行成功	执行失败
函数返回	返回函数中定义的返回信息。	返回包含错误信息和错误类型的 JSON 文件。格式如下： <pre>{   "errorMessage": "",   "errorType": "" }</pre> errorMessage: Runtime 返回的错误信息 errorType: 错误类型
执行摘要	显示请求 ID、配置内存、执行时长、实际使用内存和收费时长。	显示请求 ID、配置内存、执行时长、实际使用内存和收费时长。
日志	打印函数日志，最多显示 4KB 的日志。	打印报错信息，最多显示 4KB 的日志。

# 7 C#

## 7.1 开发事件函数

### 7.1.1 C#函数开发

#### 函数定义

##### 📖 说明

建议使用.NET Core 3.1 版本。

对于 C#，FunctionGraph 运行时目前支持 C#(.NET Core 2.0)、C#(.NET Core 2.1)、C#(.NET Core 3.1)版本。

作用域 返回参数 函数名 (函数参数, Context 参数)

- 作用域：提供给 FunctionGraph 调用的用户函数必须定义为 public。
- 返回参数：用户定义，FunctionGraph 负责转换为字符串，作为 HTTP Response 返回。
- 函数名：用户自定义函数名称，需要和函数执行入口处用户自定义的入口函数名称一致。

在函数工作流控制台左侧导航栏选择“函数 > 函数列表”，单击需要设置的“函数名称”进入函数详情页，选择“设置 > 常规设置”，配置“函数执行入口”参数，如图 7-1 所示。其中参数值为“CsharpDemo::CsharpDemo.Program::MyFunc”（[程序集名]::[命名空间].[类名]::[执行函数名]）格式。

图7-1 函数执行入口参数



- 执行事件体：函数执行界面由用户输入的执行事件参数。
- 上下文环境（context）：Runtime 提供的函数执行上下文，相关属性定义在对象说明中。

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

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

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

## 函数 Handler 定义

ASSEMBLY::NAMESPACE.CLASSNAME::METHODNAME

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

## SDK 接口

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

表7-1 Context 对象说明

属性名	属性说明
String RequestId	请求 ID。
String ProjectId	Project Id
String PackageName	函数所在分组名称
String FunctionName	函数名称
String FunctionVersion	函数版本
Int MemoryLimitInMb	分配的内存。
Int CpuNumber	获取函数占用的 CPU 资源。
String Accesskey	获取用户委托的 AccessKey（有效期 24 小时），使用该方法需要给函数配置委托。
String Secretkey	获取用户委托的 SecretKey（有效期 24 小时），使用该方法需要给函数配置委托。
String Token	获取用户委托的 Token（有效期 24 小时），使用该方法



属性名	属性说明
	需要给函数配置委托。
Int RemainingTimeInMilliseconds	函数剩余运行时间
String GetUserData(string key,string defvalue=" ")	通过 key 获取用户通过环境变量传入的值。

- 日志接口  
    FunctionGraph 中 C# SDK 中接口日志说明如所示。

表7-2 日志接口说明

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

## 开发 C#函数

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

### 步骤 1 创建 C#编译工程

登录已经安装了.NET SDK 和运行环境的 linux 服务器，创建目录“/home/fsscsharp/src”，将 FunctionGraph 函数 dll 解压到该目录。如图 7-2 所示。

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

图7-2 函数解压

```

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#
    
```

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

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

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

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

Microsoft .NET Core Shared Framework Host

  Version : 2.0.9
  Build   : 1632fa1589b0eee3277a8841ce1770e554ece037
```

创建并初始化 console application 工程，命令如下：

“dotnet new console -n project\_name”

示例命令：

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

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

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

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

```
sw.WriteLine("CSharp runtime test(v1.0.2)");
sw.WriteLine("=====");
sw.WriteLine("Request Id:      {0}", context.RequestId);
sw.WriteLine("Function Name:   {0}", context.FunctionName);
sw.WriteLine("Function Version: {0}", context.FunctionVersion);
sw.WriteLine("Project:         {0}", context.ProjectId);
sw.WriteLine("Package:         {0}", context.PackageName);
sw.WriteLine("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());
}
}
```

## 步骤 2 编译 C#工程

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

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

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

用“`dotnet build`”命令编译工程，回显信息如下:

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

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

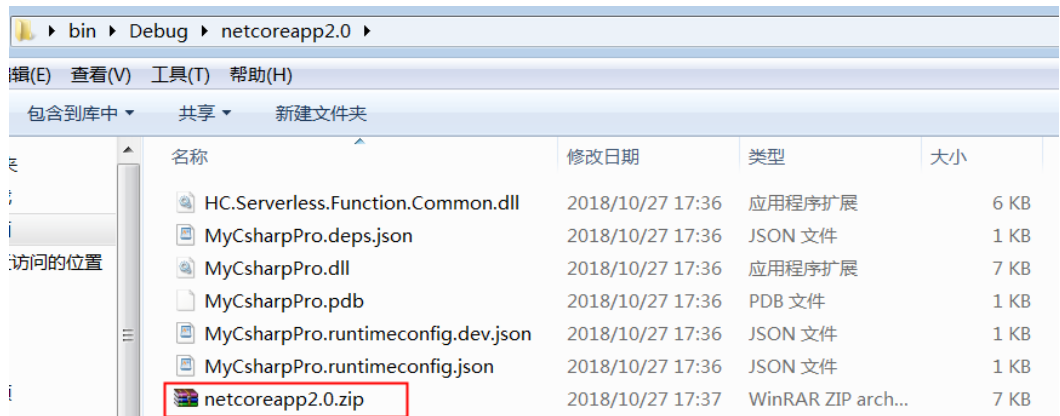
Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:01.47
```

### 步骤 3 部署 C#工程到 FunctionGraph 服务

用 ssh 工具将编译后的文件拷贝并打包，如图 7-3 所示。

图7-3 打包文件



创建函数并上传上一步的 zip 包。

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

---结束

## 执行结果

执行结果由 3 部分组成：函数返回、执行摘要和日志。

表7-3 执行结果说明

参数项	执行成功	执行失败
函数返回	返回函数中定义的返回信息。	返回包含错误信息和错误类型的 JSON 文件。格式如下： <pre>{     "errorMessage": "",</pre>

参数项	执行成功	执行失败
		<pre>"errorType": "" }</pre> <p>errorMessage: Runtime 返回的错误信息 errorType: 错误类型</p>
执行摘要	显示请求 ID、配置内存、执行时长、实际使用内存和收费时长。	显示请求 ID、配置内存、执行时长、实际使用内存和收费时长。
日志	打印函数日志, 最多显示 4KB 的日志。	打印报错信息, 最多显示 4KB 的日志。

## 7.1.2 函数支持 json 序列化和反序列化

### 7.1.2.1 使用 NET Core CLI

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

提供的接口如下:

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

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

本例以 .NET Core2.1 创建 “test” 工程为例说明, .NET Core2.0、.NET Core3.1 方法类似。执行环境已装有 .NET SDK2.1。

### 新建项目

1. 创建目录 “/tmp/csharp/projects /tmp/csharp/release” , 执行命令如下:

```
mkdir -p /tmp/csharp/projects;mkdir -p /tmp/csharp/release
```

2. 进入 “/tmp/csharp/projects/” 目录, 执行命令如下:

```
cd /tmp/csharp/projects/
```

3. 新建工程文件 “test.csproj” , 文件内容如下:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.1</TargetFramework>
    <RootNamespace>test</RootNamespace>
    <AssemblyName>test</AssemblyName>
  </PropertyGroup>
  <ItemGroup>
    <Reference Include="HC.Serverless.Function.Common">
      <HintPath>HC.Serverless.Function.Common.dll</HintPath>
    </Reference>
    <Reference Include="HC.Serverless.Function.Common.JsonSerializer">
```

```
<HintPath>  
HC.Serverless.Function.Common.JsonSerializer.dll</HintPath>  
</Reference>  
</ItemGroup>  
</Project>
```

## 生成代码库

1. 上传 dll 文件。  
将 HC.Serverless.Function.Common.dll、HC.Serverless.Function.Common.JsonSerializer.dll、Newtonsoft.Json.dll 文件上传至目录 “/tmp/csharp/projects/”。
2. 在 “/tmp/csharp/projects/” 路径下，新建 “Class1.cs” 文件，代码如下：

```
using HC.Serverless.Function.Common;  
using System;  
using System.IO;  
  
namespace test  
{  
    public class Class1  
    {  
        public Stream ContextHandlerSerializer(Stream input, IFunctionContext  
context)  
        {  
            var logger = context.Logger;  
            logger.Logf("CSharp runtime test(v1.0.2)");  
            JsonSerializer test = new JsonSerializer();  
            TestJson Testjson = test.Deserialize<TestJson>(input);  
            if (Testjson != null)  
            {  
                logger.Logf("json Deserialize KetTest={0}", Testjson.KetTest);  
            }  
            else  
            {  
                return null;  
            }  
  
            return test.Serialize<TestJson>(Testjson);  
        }  
  
        public class TestJson  
        {  
            public string KetTest { get; set; } //定义序列化的类中的属性为 KetTest  
        }  
    }  
}
```

3. 执行以下命令，生成代码库：

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

### 📖 说明

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

4. 执行以下命令, 进入 “/tmp/csharp/release” 路径。

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

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

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

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

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

### 📖 说明

- \*.runtimeconfig.json 文件的名称为程序集的名称。
- 文件内容中的 version: 项目属性中的目标框架的版本号, 2.0 则为 2.0.0 , 2.1 则为 2.1.0。
- 当目标框架为 .NET Core 2.0 时, 要注意生成 \*.deps.json 文件中是否已引入 Newtonsoft.Json。如果没有引入, 则需要自己手动引入, 如下所示:
- 需要在 “targets” 中引入以下内容:

```
"Newtonsoft.Json/9.0.0.0": {
  "runtime": {
    "Newtonsoft.Json.dll": {
      "assemblyVersion": "9.0.0.0",
      "fileVersion": "9.0.1.19813"
    }
  }
}
```

1. 在 “libraries” 引入以下内容:

```
"Newtonsoft.Json/9.0.0.0": {
  "type": "reference",
  "serviceable": false,
  "sha512": ""
}
```

```
}
```

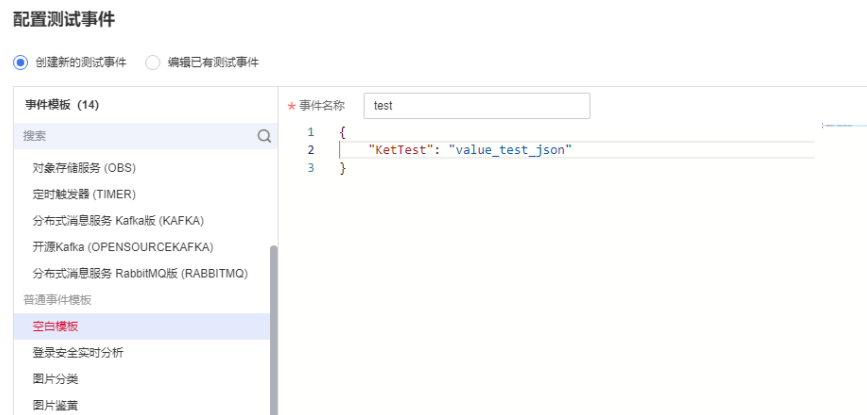
7. 在“/tmp/csharp/release”路径下，执行如下命令，打包 test.zip 代码库压缩包。

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

## 测试示例

1. 在天翼云 FunctionGraph 控制台新建一个 C# (.NET 2.1) 函数，上传打包好的“test.zip”压缩包。
2. 配置一个测试事件。如图 7-4 所示。其中的 key 必须设置为“KetTest”，value 可以自定义。

图7-4 配置测试事件



### 说明

- KetTest: 定义序列化的类中的属性为 KetTest。
  - 测试串必需为 json 格式。
3. 单击“测试”，查看测试执行结果。

## 7.1.2.2 使用 Visual Studio

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

提供的接口如下：

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

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

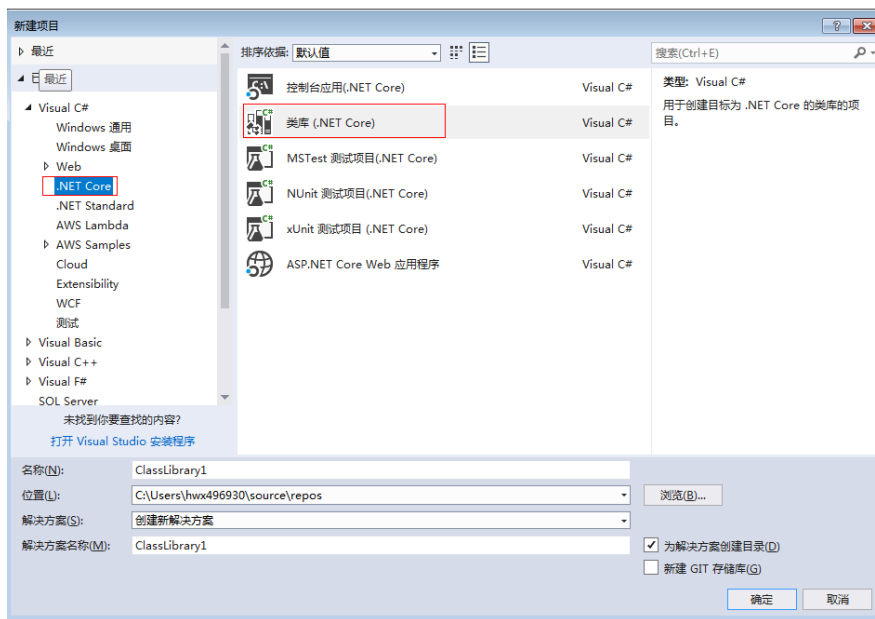
本例以 Visual Studio 2017 新建一个 .NET Core 2.0 的“test”工程，.NET Core 2.1、.NET Core 3.1 类似。

## 新建项目

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

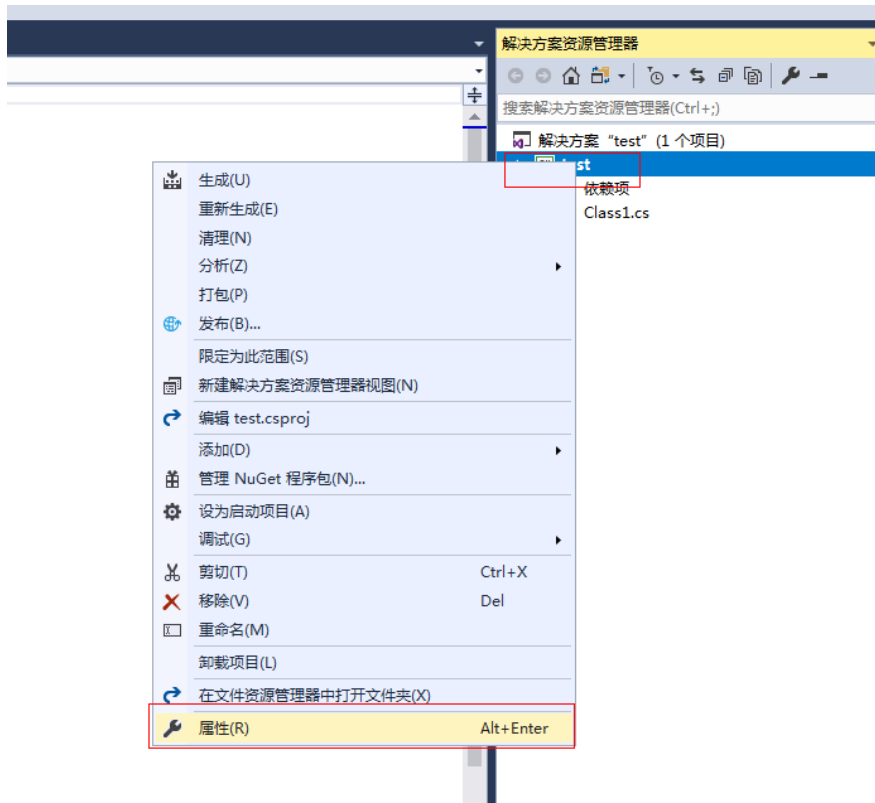


图7-5 新建项目



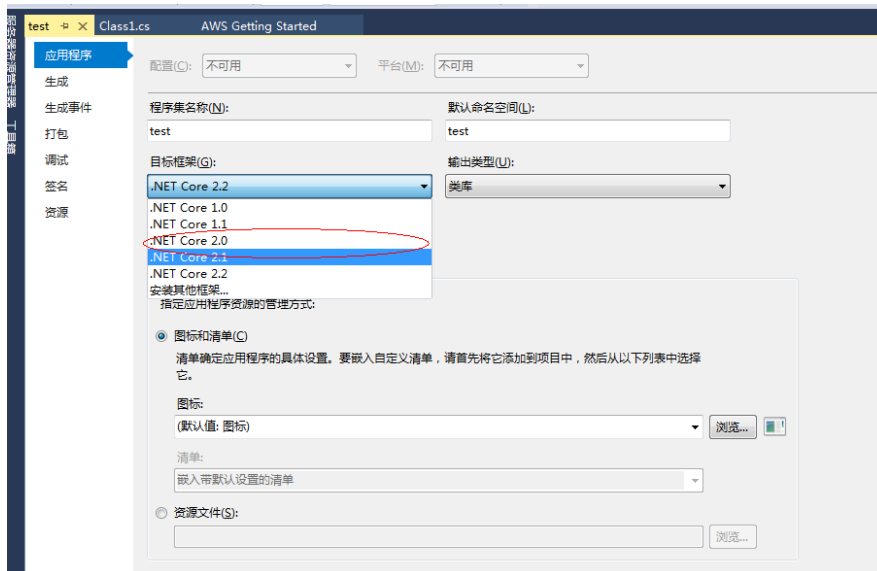
2. 导航栏中选择“test”项目，单击鼠标右键，选择“属性”，打开属性界面。如图7-6所示。

图7-6 属性



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

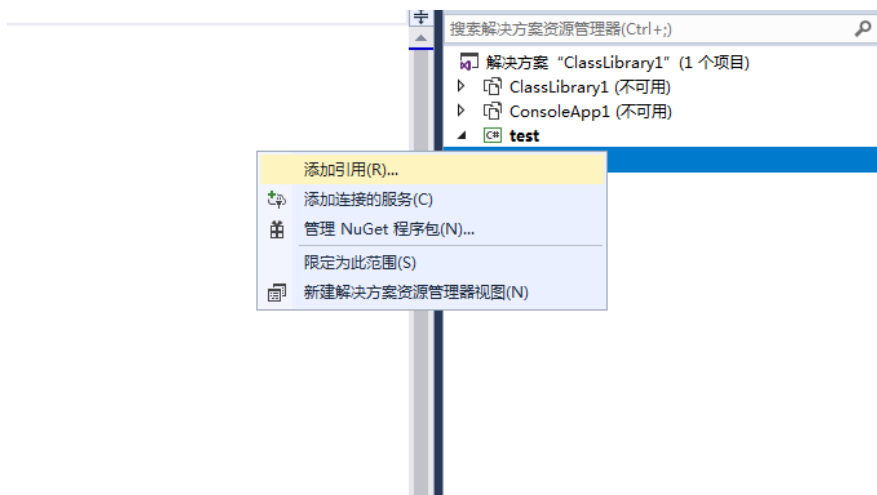
图7-7 选择目标框架



## 添加引用

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

图7-8 添加引用

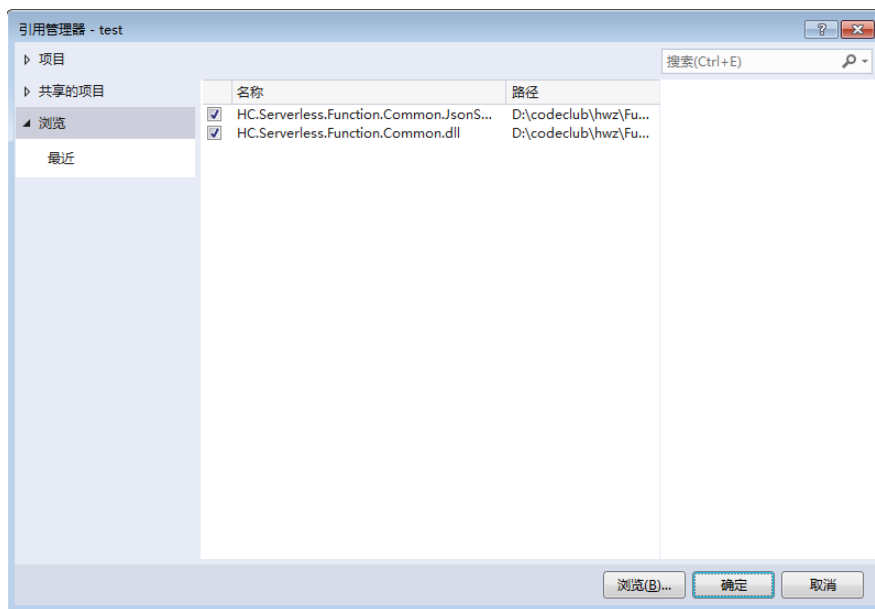


### 说明

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

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

图7-9 引用文件



3. 引用成功后界面如图 7-10 所示。

图7-10 完成引用



## 打包代码

本例所用示例代码如下：

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

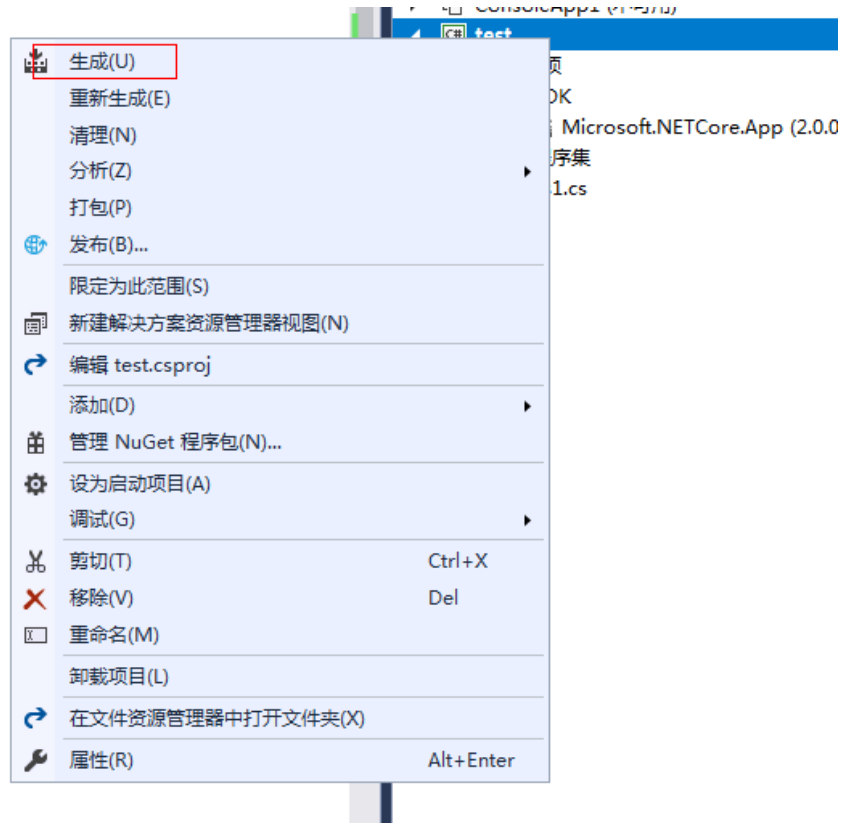
```
namespace test
{
    public class Class1
    {
        public Stream ContextHandlerSerializer(Stream input, IFunctionContext
context)
        {
            var logger = context.Logger;
            logger.Logf("CSharp runtime test (v1.0.2)");
            JsonSerializer test = new JsonSerializer();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

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

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

1. 右击“test”工程，选择“生成”，如图 7-11 所示。

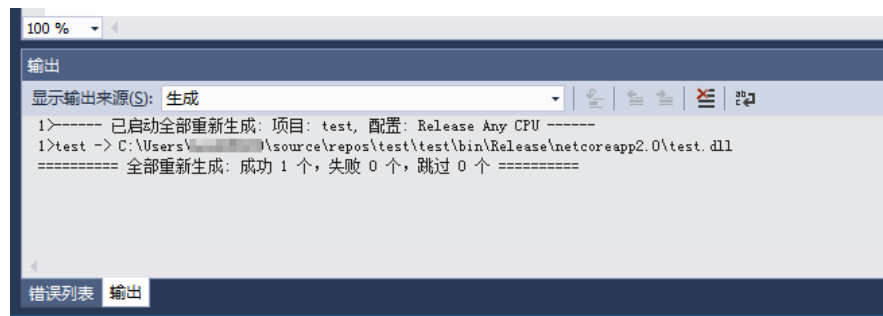
图7-11 生成文件



2. 拷贝生成 dll 文件的路径

“C:\Users\xxx\source\repos\test\test\bin\Release\netcoreapp2.0\”，如图 7-12 所示。

图7-12 生成路径



该路径下的文件如图 7-13 所示。

图7-13 文件

名称	修改日期	类型
HC.Serverless.Function.Common.dll	2019/1/19 20:24	应用程序扩展
HC.Serverless.Function.Common.JsonSerializer.dll	2019/1/19 20:24	应用程序扩展
Newtonsoft.Json.dll	2018/6/25 10:36	应用程序扩展
test.deps.json	2019/1/21 10:50	JSON File
test.dll	2019/1/21 10:50	应用程序扩展
test.pdb	2019/1/21 10:50	Program Debug...

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

图7-14 新建文件

名称	修改日期	类型
HC.Serverless.Function.Common.dll	2019/1/19 20:24	应用程序扩展
HC.Serverless.Function.Common.JsonSerializer.dll	2019/1/19 20:24	应用程序扩展
Newtonsoft.Json.dll	2018/6/25 10:36	应用程序扩展
test.deps.json	2019/1/21 10:50	JSON File
test.dll	2019/1/21 10:50	应用程序扩展
test.pdb	2019/1/21 10:50	Program Debug...
test.runtimeconfig.json	2019/1/21 11:12	JSON File

文件内容如下：

```
{  
  
  "runtimeOptions": {  
  
    "framework": {  
  
      "name": "Microsoft.NETCore.App",  
  
      "version": "2.0.0"  
    }  
  }  
}
```

**说明**

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

4. 将文件打包为 netcoreapp2.0.zip 压缩包。

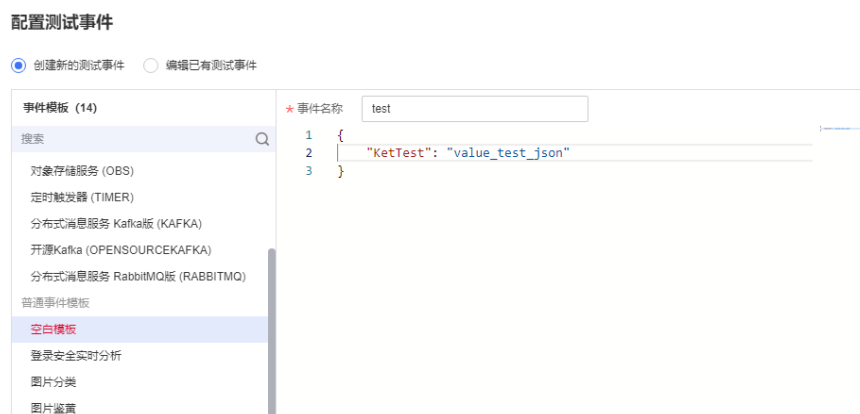
### 说明

压缩包文件名称可随意，但是一定为.zip 格式。

## 测试示例

1. 在天翼云 FunctionGraph 控制台新建一个 C# (.NET 2.1) 函数，上传打包完成的代码包。
2. 配置一个测试事件。如图 7-15 所示。其中的 key 必须设置为“KetTest”，value 可以自定义。

图7-15 配置测试事件



### 说明

- KetTest: 定义序列化的类中的属性为 KetTest.
  - 测试串必需为 json 格式。
3. 单击“测试”，查看测试执行结果。

# 8 PHP

## 8.1 开发事件函数

### 函数定义

PHP 7.3 函数的接口定义如下所示：

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

- 入口函数名（`$handler`）：入口函数名称，需和函数执行入口处用户自定义的入口函数名称一致。
- 执行事件（`$event`）：函数执行界面由用户输入的执行事件参数，格式为 JSON 对象。
- 上下文环境（`$context`）：Runtime 提供的函数执行上下文，其接口定义在 [SDK 接口说明](#)。
- 函数执行入口：`index.handler`。
- 函数执行入口格式为 “[文件名].[函数名]”。例如创建函数时设置为 `index.handler`，那么 FunctionGraph 会去加载 `index.php` 中定义的 handler 函数。

### PHP 的 initializer 入口介绍

函数服务目前支持以下 PHP 运行环境。

- Php 7.3 (runtime = Php7.3)

Initializer 格式为：

**[文件名].[initializer 名]**

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

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

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

- 函数名  
`my_initializer` 需要与实现 initializer 接口时的 `initializer` 字段相对应。



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

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

## SDK 接口

Context 类中提供了许多上下文方法供用户使用，其声明和功能如所示。

表8-1 Context 类上下文方法说明

方法名	方法说明
getRequestID()	获取请求 ID。
getRemainingTimeInMilliseconds ()	获取函数剩余运行时间。
getAccessKey()	获取用户委托的 AccessKey（有效期 24 小时），使用该方法需要给函数配置委托。
getSecretKey()	获取用户委托的 SecretKey（有效期 24 小时），使用该方法需要给函数配置委托。
getUserData(string key)	通过 key 获取用户通过环境变量传入的值。
getFunctionName()	获取函数名称。
getRunningTimeInSeconds ()	获取函数超时时间。
getVersion()	获取函数的版本。
getMemorySize()	分配的内存。
getCPUNumber()	获取函数占用的 CPU 资源。
getPackage()	获取函数组。
getToken()	获取用户委托的 token（有效期 24 小时），使用该方法需要给函数配置委托。
getLogger()	<p>获取 context 提供的 logger 方法，返回一个日志输出类，通过使用其 info 方法按“时间-请求 ID-输出内容”的格式输出日志。</p> <p>如调用 info 方法输出日志：</p> <pre>logg = context.getLogger() \$logg-&gt;info("hello")</pre>

### 📖 说明

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

## 开发 PHP 函数

开发 PHP 函数步骤如下：

### 步骤 1 创建函数

1. 编写打印 helloworld 的代码。

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

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

2. 定义 FunctionGraph 函数

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

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

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

### 📖 说明

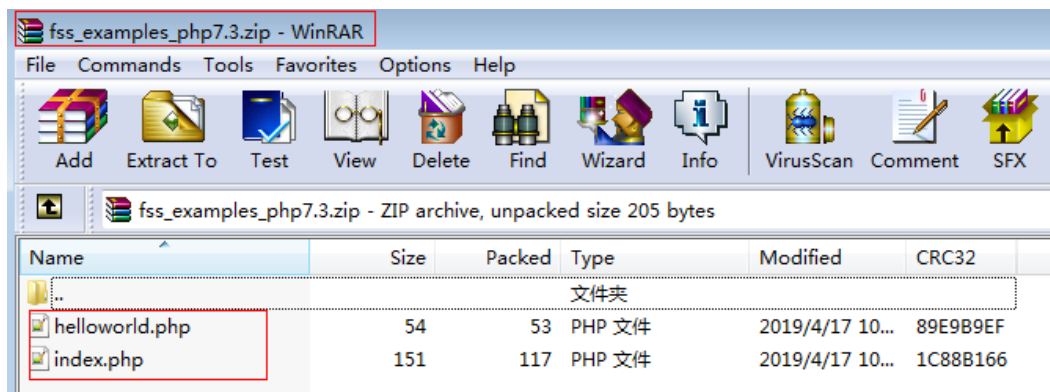
函数仅支持返回如下几种类型的值。

- Null: 函数返回的 HTTP 响应 Body 为空。
- string: 函数返回的 HTTP 响应 Body 内容为该字符串内容。
- 其他: 函数会返回值作为对象进行 json 编码，并将编码后的内容作为 HTTP 响应的 Body，同时设置响应的“Content-Type”头为“text/plain”。

### 步骤 2 工程打包

函数工程创建以后，可以得到以下目录，选中工程所有文件，打包命名为“fss\_examples\_php7.3.zip”，如图 8-1 所示。

图8-1 工程打包



### 说明

本例函数工程文件保存在“~/Code/”文件夹下，在打包的时候务必进入 Code 文件夹下选中所有工程文件进行打包，这样做的目的是：由于定义了 FunctionGraph 函数的 index.php 是程序执行入口，确保 fss\_examples\_php7.3.zip 解压后，index.php 文件位于根目录。

### 步骤 3 创建 FunctionGraph 函数，上传程序包

登录 FunctionGraph，创建 PHP 函数，上传 SDK 文件。

### 说明

- 图 8-2 中的 index 与步骤定义 FunctionGraph 函数的文件名保持一致，通过该名称找到 FunctionGraph 函数所在文件。
- 图 8-2 中的 handler 为函数名，与步骤定义 FunctionGraph 函数中创建的 index.php 文件中的函数名保持一致。
- 函数执行过程为：用户上传 fss\_examples\_php7.3.zip 保存在 OBS 中，触发函数后，解压缩 zip 文件，通过 index 匹配到 FunctionGraph 函数所在文件，通过 handler 匹配到 index.php 文件中定义的 FunctionGraph 函数，找到程序执行入口，执行函数。

在函数工作流控制台左侧导航栏选择“函数 > 函数列表”，单击需要设置的“函数名称”进入函数详情页，选择“设置 > 常规设置”，配置“函数执行入口”参数，如图 8-2 所示。其中参数值为“index.handler”格式，“index”和“handler”支持自定义命名。

图8-2 函数执行入口参数

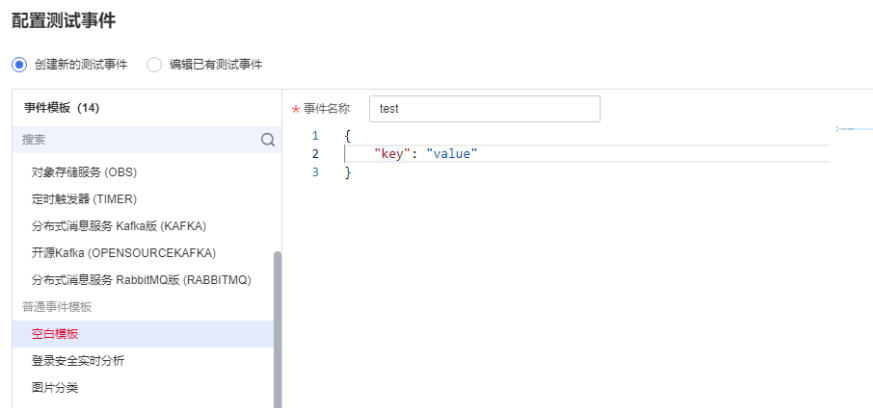


#### 步骤 4 测试函数

##### 1. 创建测试事件。

在函数详情页，单击“配置测试事件”，弹出“配置测试事件”页，输入测试信息如图 8-3 所示，单击“创建”。

图8-3 配置测试事件



##### 2. 在函数详情页，选择已配置测试事件，单击“测试”。

#### 步骤 5 执行函数

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

---结束

## 执行结果

执行结果由 3 部分组成：函数返回、执行摘要和日志。

表8-2 执行结果说明

参数项	执行成功	执行失败
函数返回	返回函数中定义的返回信息。	返回包含错误信息、错误类型和堆栈异常报错信息的 JSON 文件。格式如下： <pre>{   "errorMessage": "",   "errorType": "",   "stackTrace": {} }</pre> errorMessage: Runtime 返回的错误信息 errorType: 错误类型 stackTrace: Runtime 返回的堆栈异常报错信息
执行摘要	显示请求 ID、配置内存、执行时长、实际使用内存和收费时长。	显示请求 ID、配置内存、执行时长、实际使用内存和收费时长。
日志	打印函数日志, 最多显示 4KB 的日志。	打印报错信息, 最多显示 4KB 的日志。

## 8.2 制作依赖包

### 为 PHP 函数制作依赖包

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

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

执行如下命令。

```
Composer install
```

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

- Linux 系统

Linux 系统下可以使用以下命令生成 zip 包。

```
zip -rq vendor.zip vendor
```

- windows 系统

用压缩软件将 vendor 目录压缩成 zip 文件即可。

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

### 📖 说明

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

# 9 开发工具

## 9.1 CodeArts IDE Online

### 9.1.1 CodeArts IDE Online 在线管理函数

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

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

### 📖 说明

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

### 前提条件

1. 新建实例选择 All In One 实例，直接包含了需要的 Java、Python 等插件。
2. 已经在 FunctionGraph 控制台创建函数。

### FunctionGraph 控制台创建函数

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

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

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

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

图9-1 创建新实例



步骤 4 进入 CodeArts IDE Online 在线编辑页面。

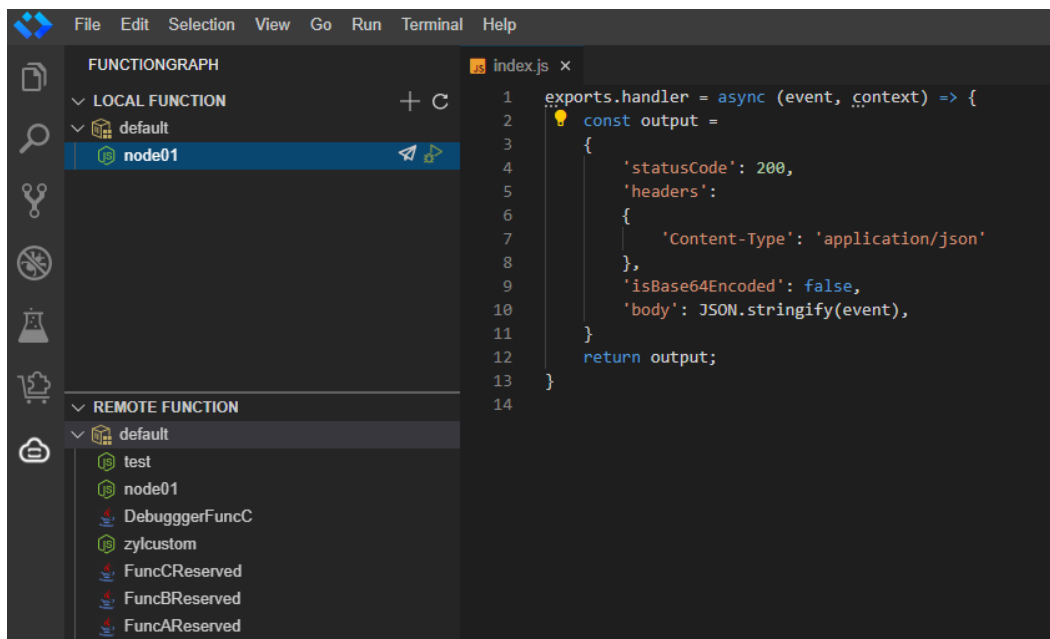
#### 📖 说明

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

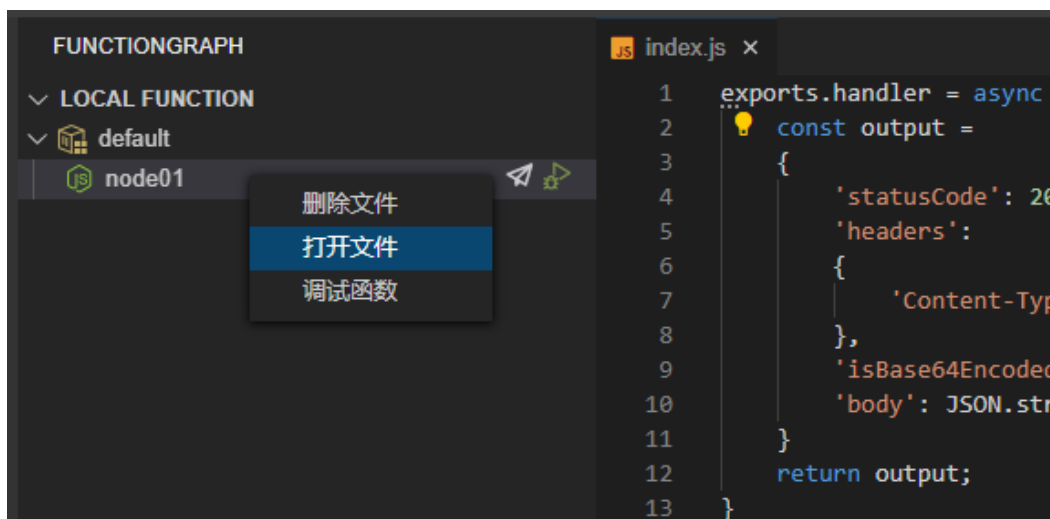
步骤 5 在编辑页面左侧导航栏单击 FunctionGraph 插件后，左侧编辑框中的 REMOTE FUNCTION 文件中即可看到在 FunctionGraph 控制台创建好的所有函数和应用。

#### 📖 说明

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



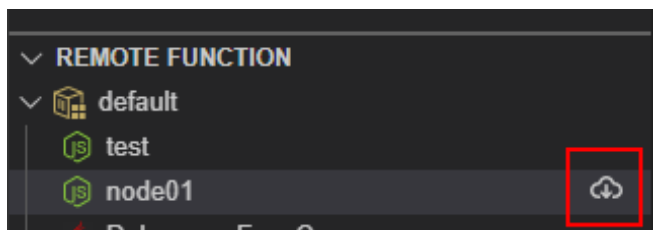
步骤 6 在 LOCAL FUNCTION 文件中，在编辑框调试函数，双击“node01”函数或者右键选择“打开文件”，在右侧编辑框打开文件。



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

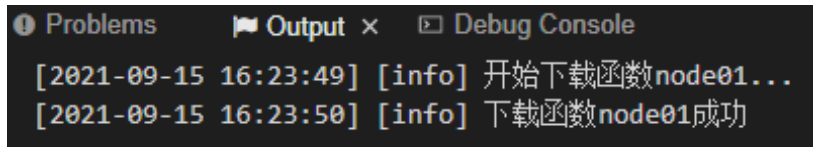
### 说明

以下示例仅供参考，具体请以实际创建函数为准。

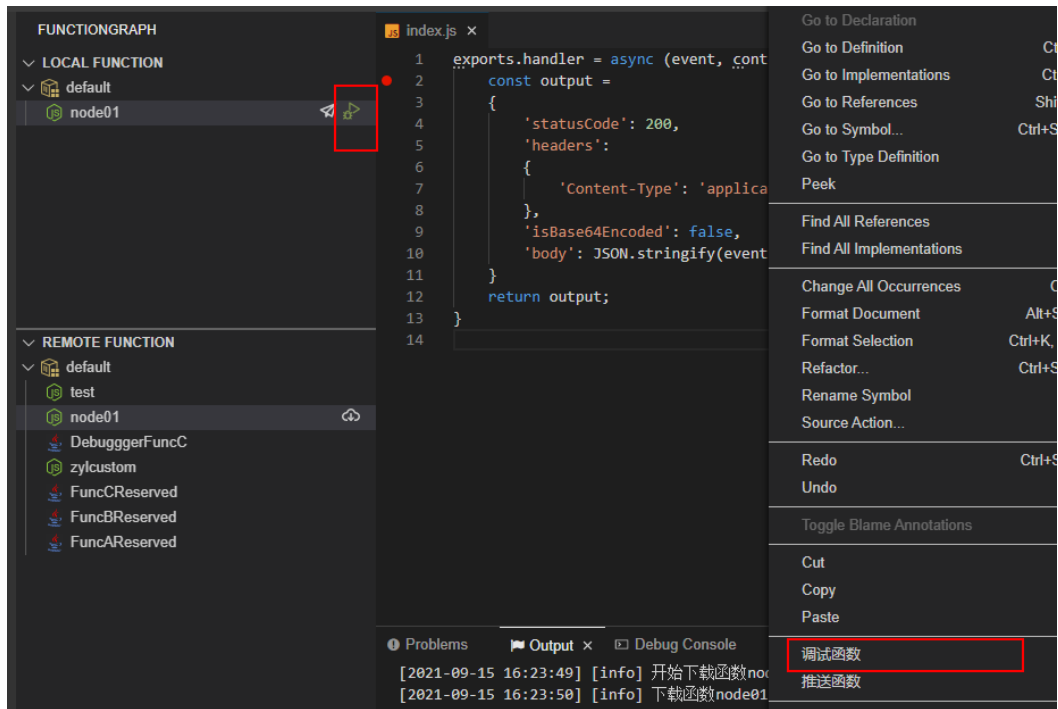


步骤 7 下载成功后，右侧输出控制台提示下载成功。

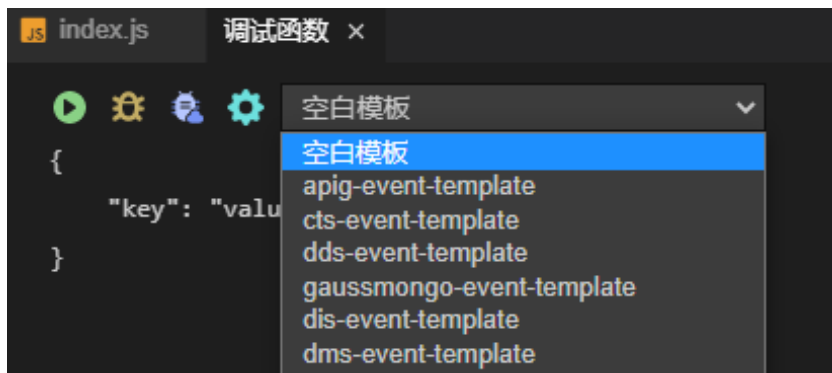


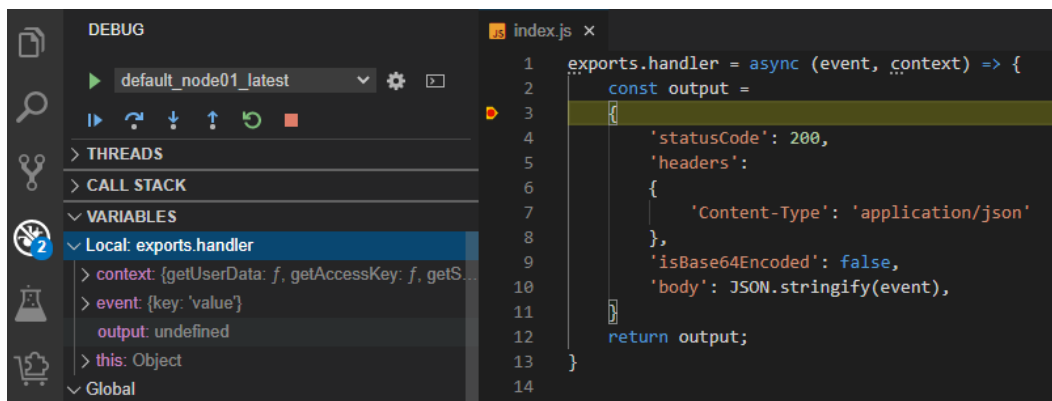


步骤 8 打完断点，点击调试图标或者右键选择“调试函数”。

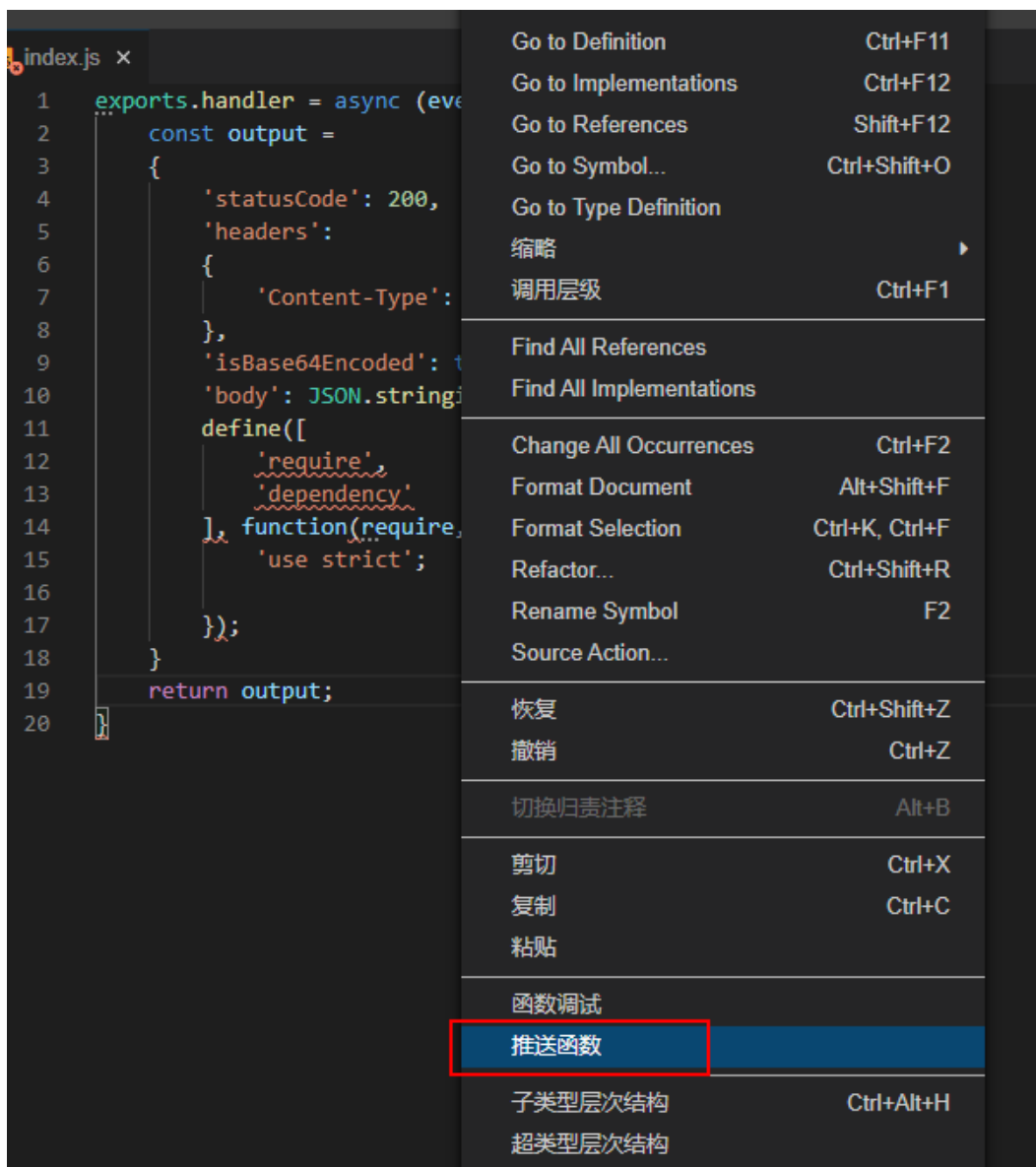


步骤 9 进入函数调试页面，选择测试事件，点击“调试”。





步骤 10 修改函数代码后，点击“推送函数”图标或者右键选择“推送函数”。



步骤 11 页面下方输出控制台提示推送成功。

```
Problems Output x Debug Console
[2021-09-15 17:52:22] [info] 推送函数中，函数名称为node01
[2021-09-15 17:52:22] [info] 函数没有依赖
[2021-09-15 17:52:23] [info] 推送函数成功
```

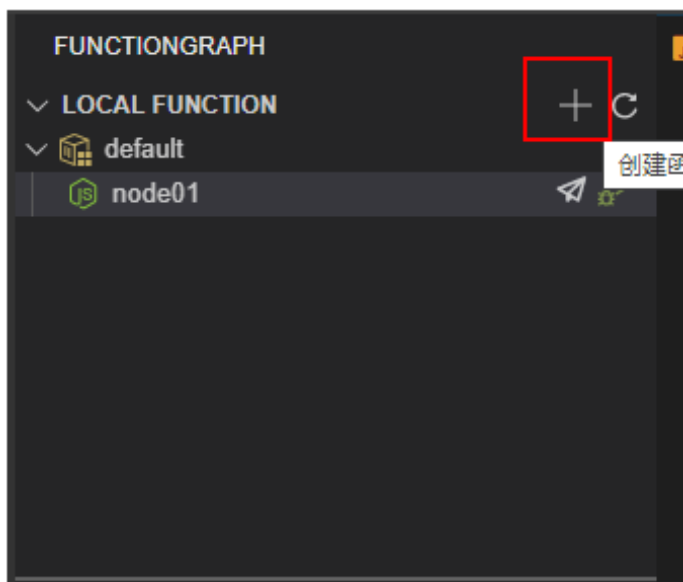
步骤 12 返回 FunctionGraph 控制台，查看函数，确认已合入修改的内容。

---结束

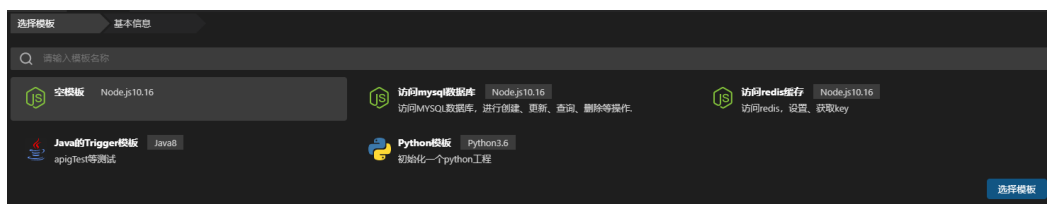
## CodeArts IDE Online 本地创建函数

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

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



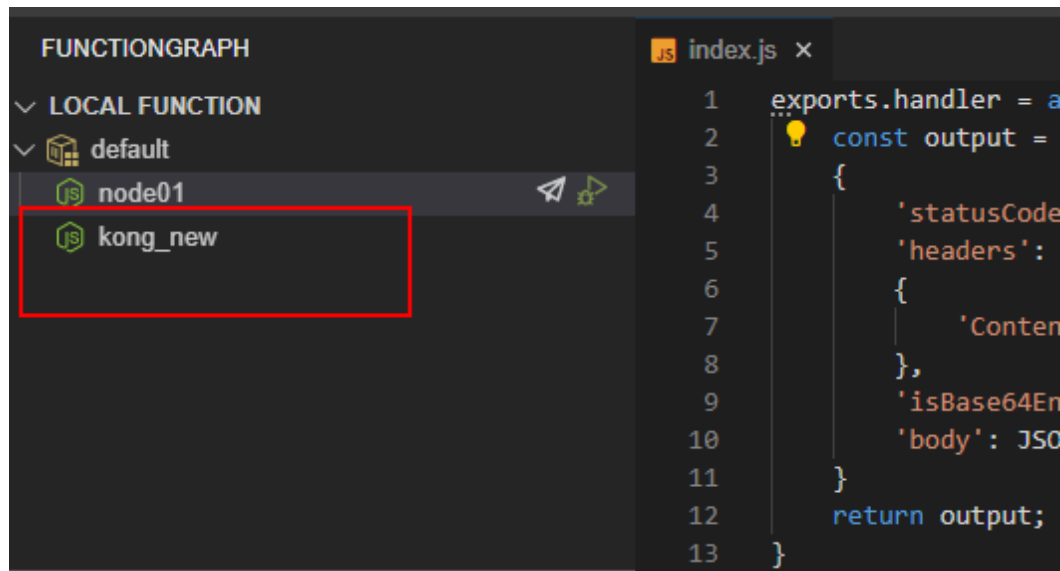
步骤 2 选择模板，创建函数。



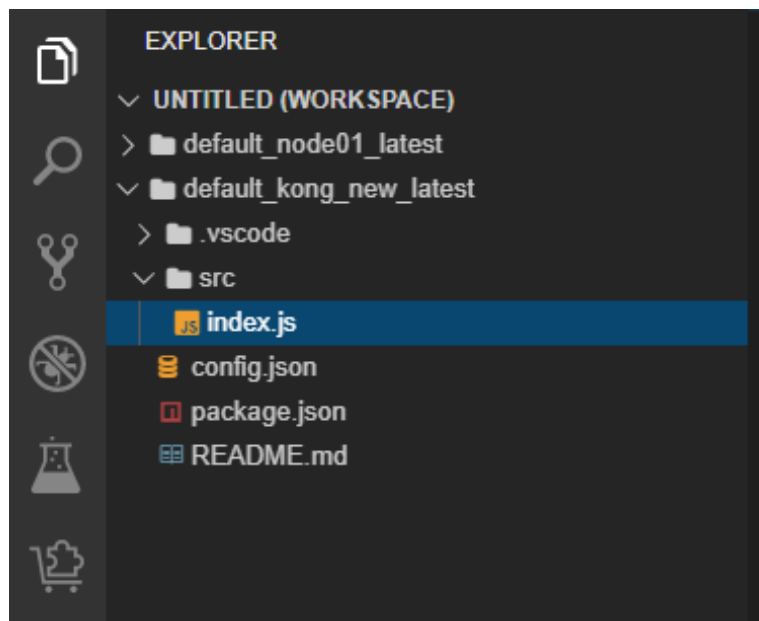
步骤 3 例如选择空模板创建，输入函数名称，比如“kong\_new”，单击“创建函数”。



步骤 4 创建成功后，左侧编辑框即可看到刚创建的函数。



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

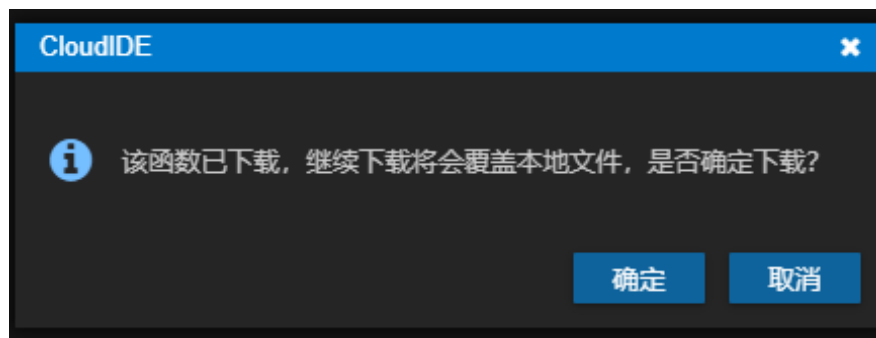


---结束

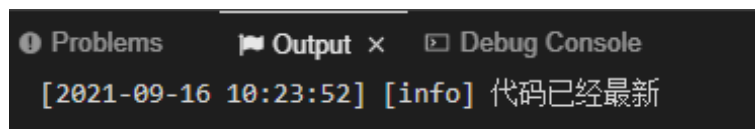
### 函数下载、推送代码比较流程

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

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

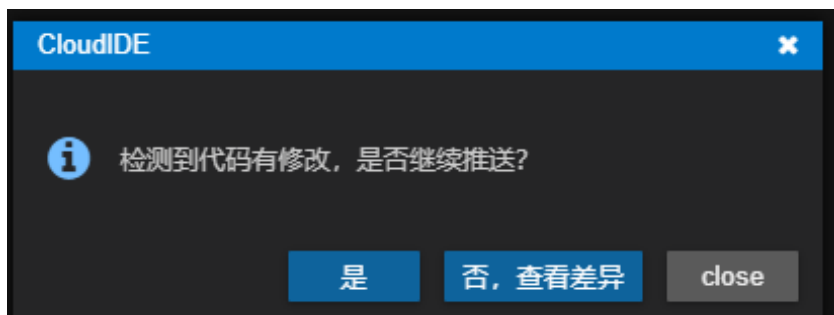


步骤 2 函数 node01 已经存在在 REMOTE FUNCTION，不做修改直接推送，远程和本地的代码会比较，提示“代码已经最新”，不必推送。

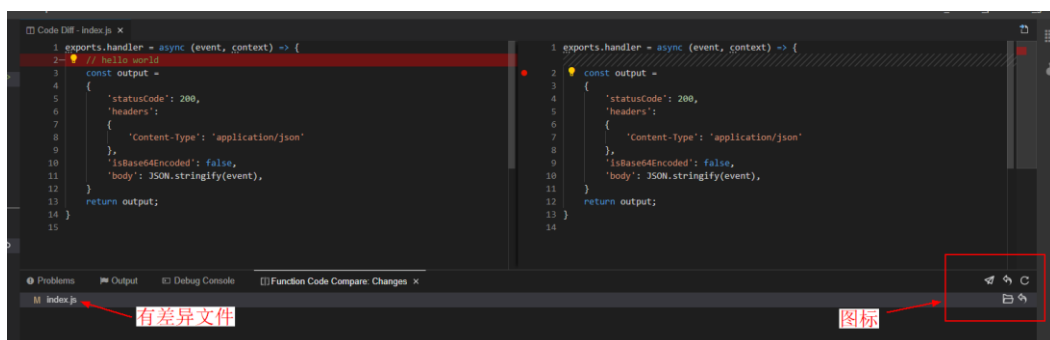


步骤 3 函数 node01 做出修改，删除第二行的“// hello world”，推送提示如下。

- 是：直接推送，本地代码将会覆盖远程代码。



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



---结束

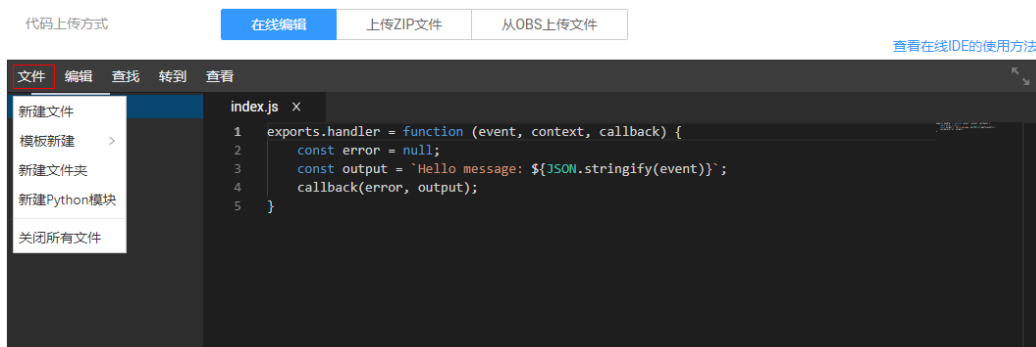
## 9.1.2 附录: CodeArts IDE Online 使用方法

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

### 目录管理

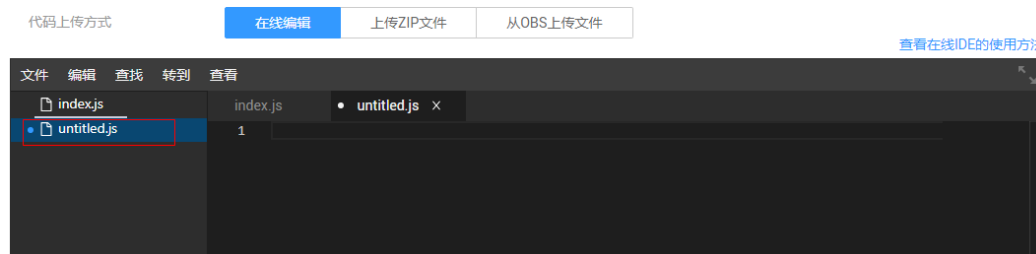
在编辑器菜单栏中选择“文件”, 可以管理文件夹目录, 如图 9-2 所示。

图9-2 文件



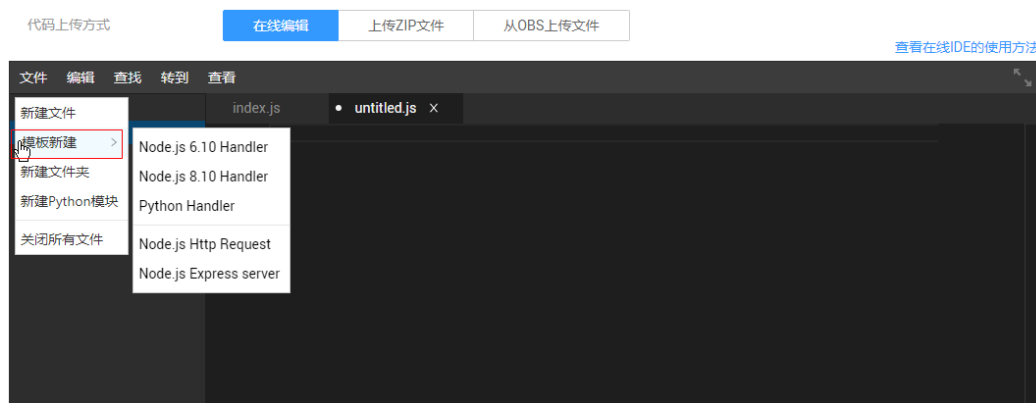
在展开的下拉菜单中选择“新建文件”，可以新建文件，并命名，如图 9-3 所示。

图9-3 新建文件



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

图9-4 新建模板



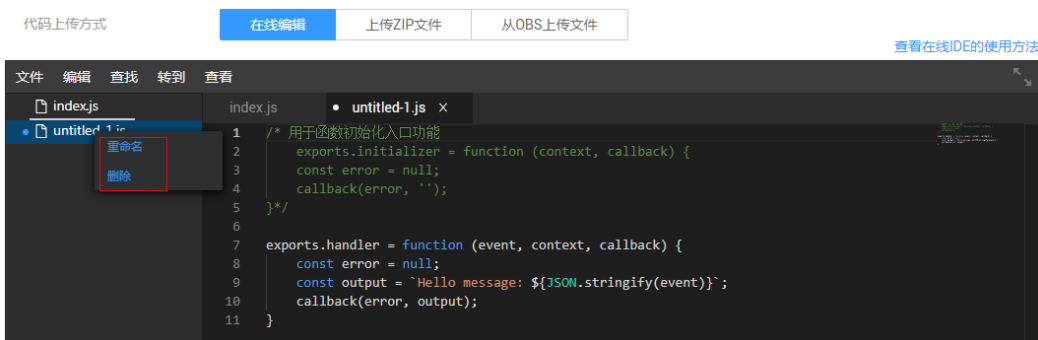
选择需要的模板，创建在线函数，并命名，如图 9-5 所示。

图9-5 创建文件



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

图9-6 文件重命名

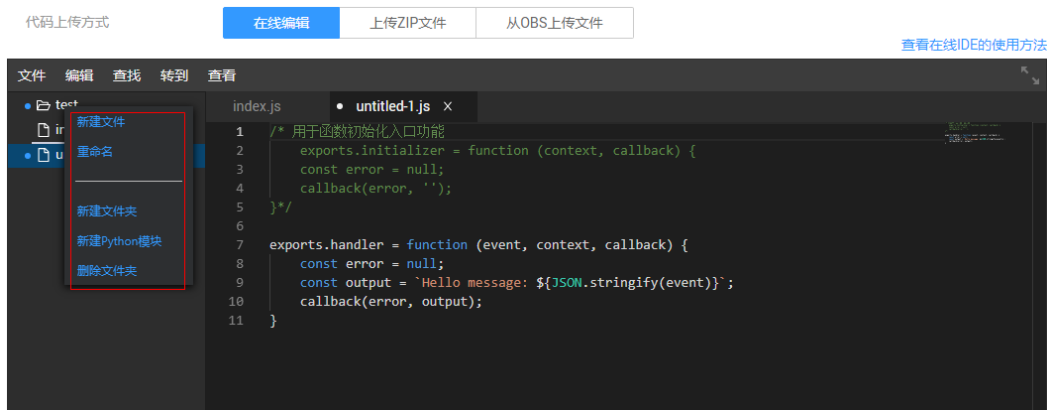


在左侧“文件”下拉菜单中选择“新建文件夹”，新建文件夹并命名，如图 9-2 中所示。

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



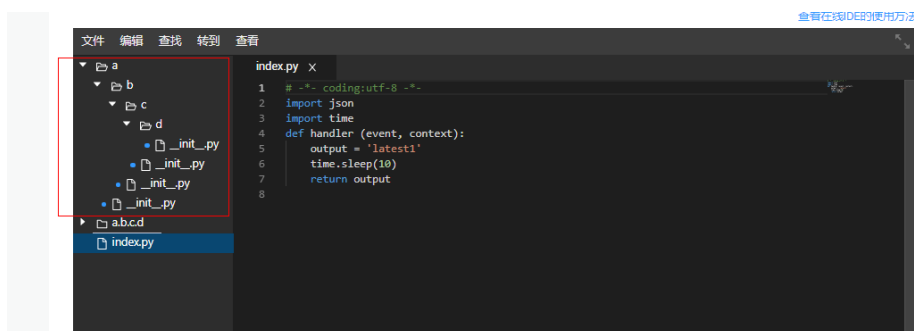
图9-7 编辑文件夹



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

支持快速创建 Python 模块，支持多创建层级，如图 9-8 所示。

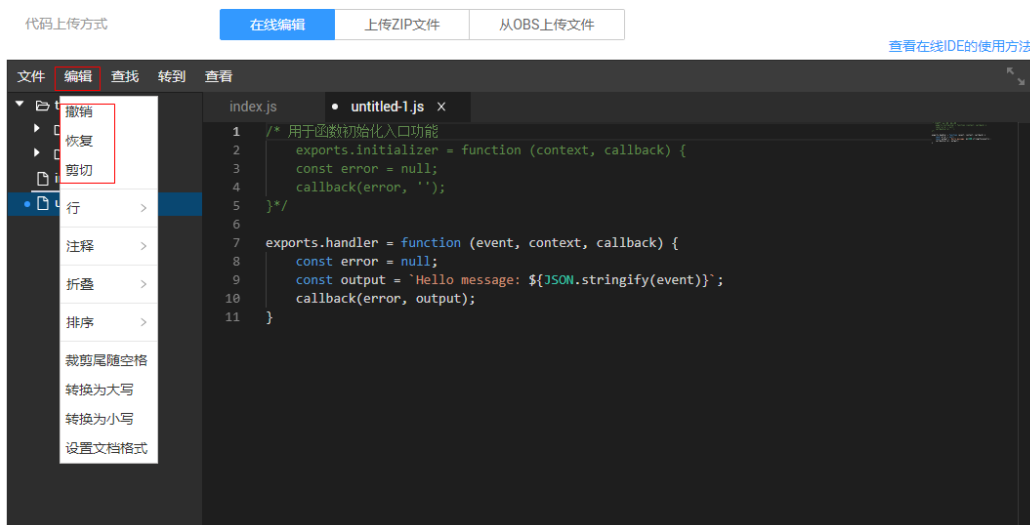
图9-8 多层次模块



## 代码在线编辑

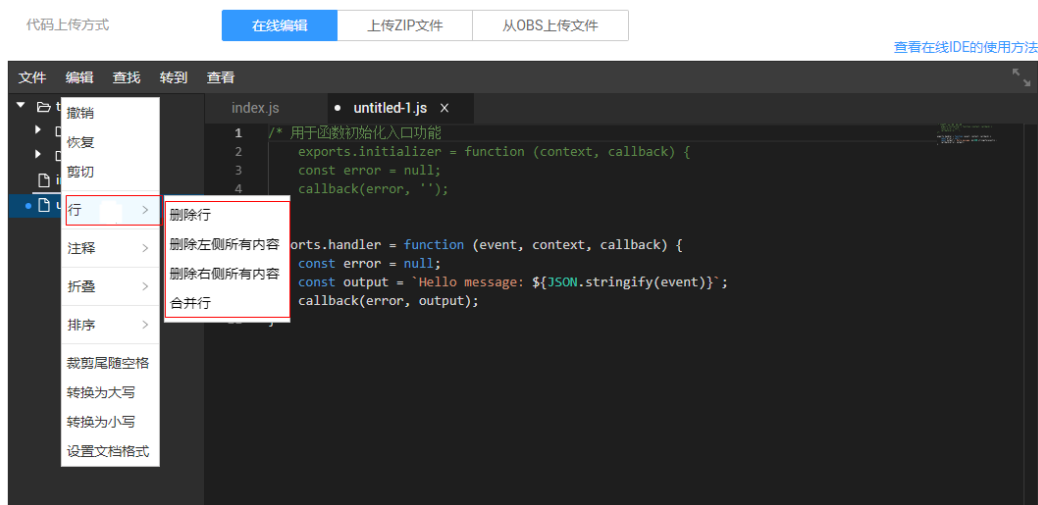
在编辑器菜单栏中选择“编辑”，可以在线编辑代码。在下拉菜单中可以撤销操作或恢复上一步操作，也可以剪切内容等，如图 9-9 所示。

图9-9 在线编辑



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

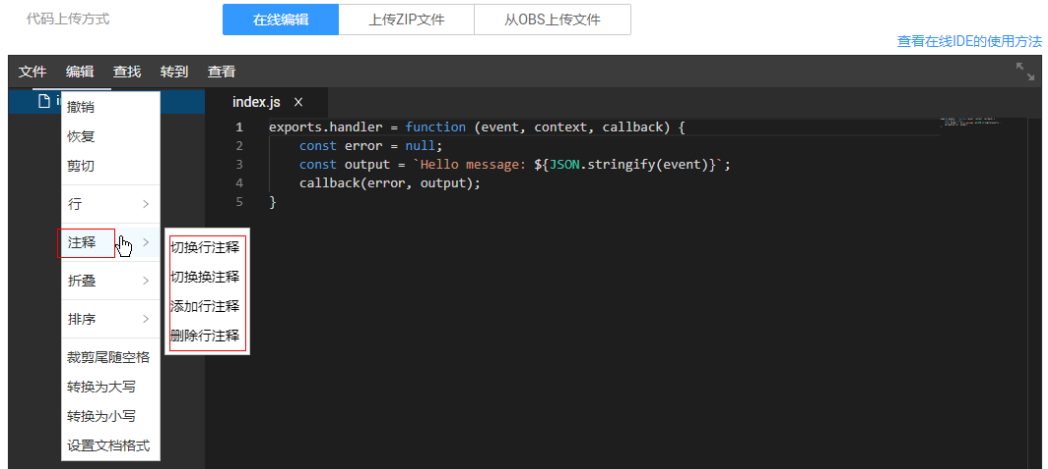
图9-10 编辑行



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

在“编辑”下拉菜单中选择“注释”，可以编辑注释，如图 9-11 所示。

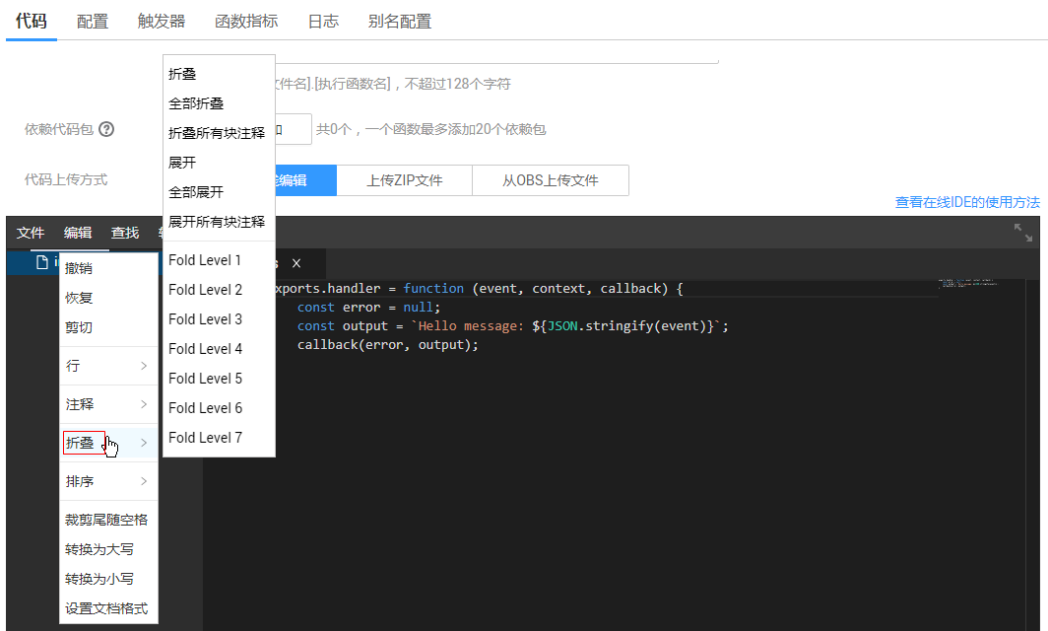
图9-11 编辑注释



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

在“编辑”下拉菜单中选择“折叠”，可以展开或折叠代码，如图 9-12 所示。

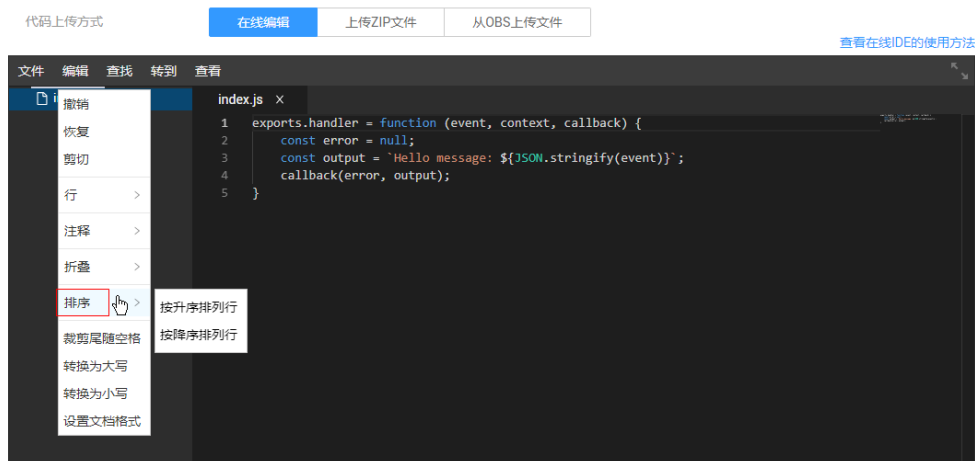
图9-12 展开或折叠代码



代码可以全部展开或折叠，按层级折叠请选择“Fold Level”。

选择“排序”对代码行进行排序，如图 9-13 所示。

图9-13 排序

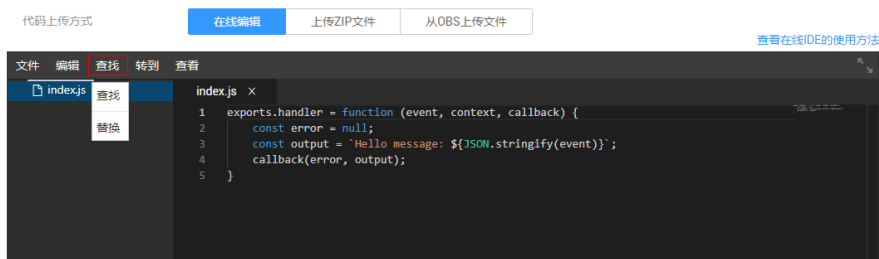


“编辑”菜单栏中同时可以裁剪尾随空格、转换大小写、设置文档格式等。

## 查找及替换

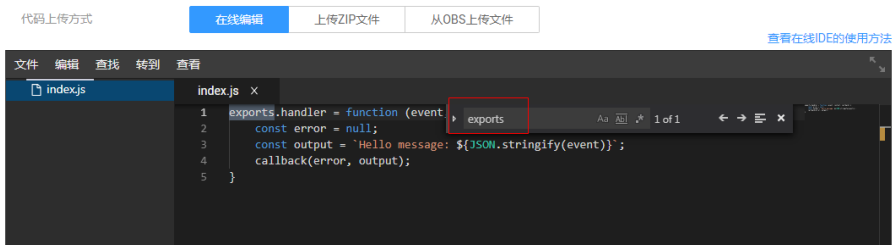
在编辑器菜单栏中选择“查找”，可以进行查找或替换。如图 9-14 所示。

图9-14 查找



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

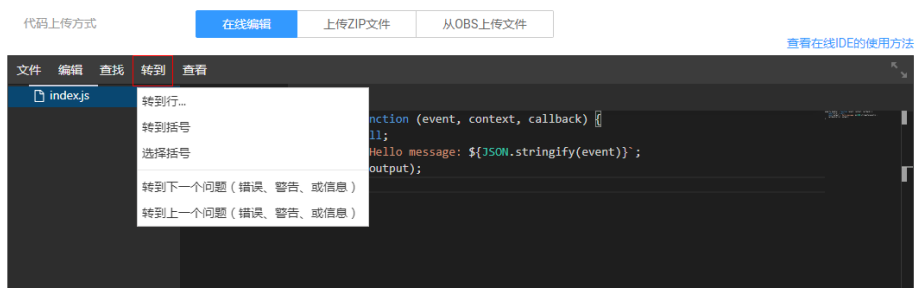
图9-15 查找代码



## 跳转

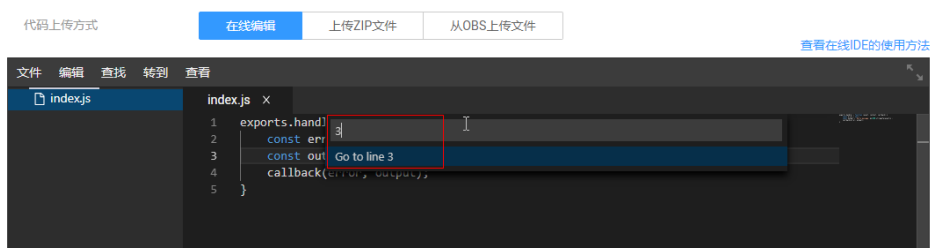
在编辑菜单栏中选择“转到”，可以跳转到相应的代码位置，如图 9-16 所示。

图9-16 转到



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

图9-17 跳转

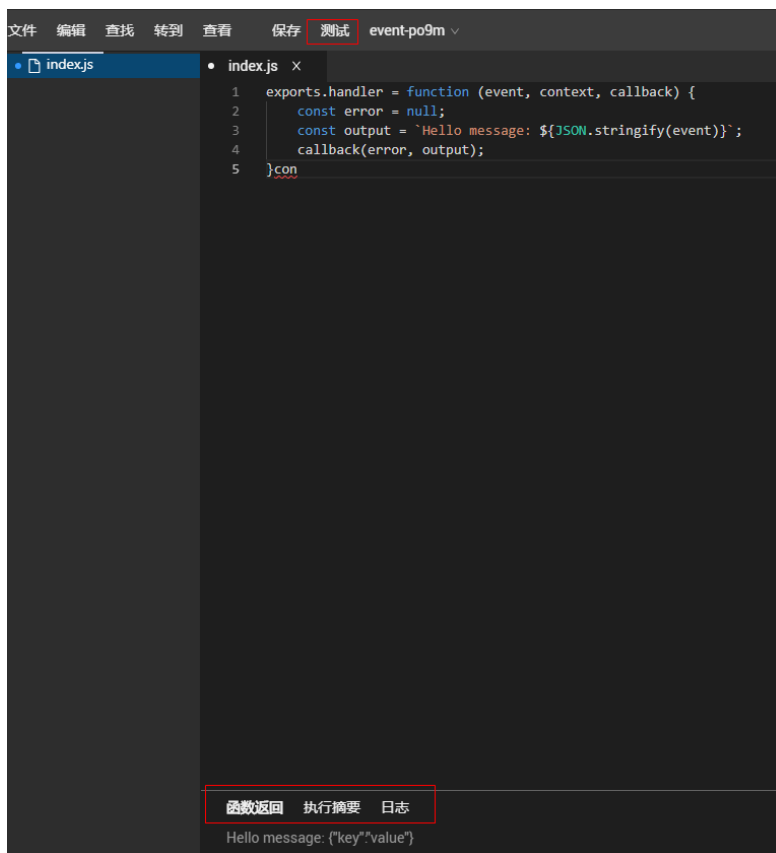


当代码比较多时，选择一个反括号“{”，点击“转到括号”可跳转到相应的括号“}”位置，当代码中有错误时，点击“转到下一个问题（错误、告警、或信息）”跳转到下一个错误的地方。

## 函数在线测试

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

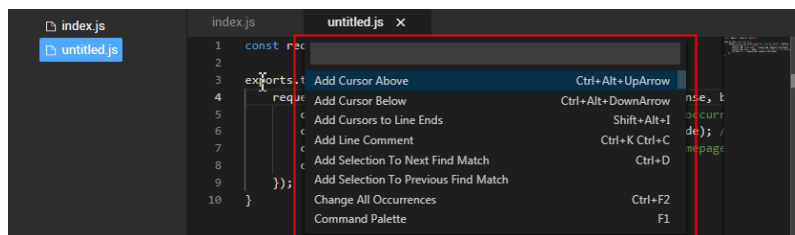
图9-18 函数在线测试



## 通用功能

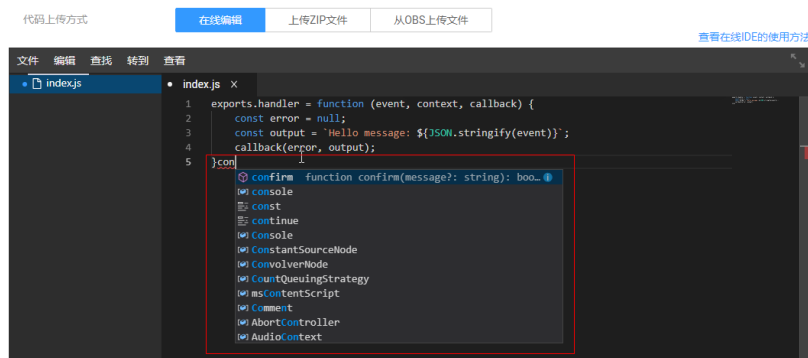
1. 在编辑菜单栏中选择“查看”，在下拉的菜单栏中选择“主题”可以更换编辑器主题，选择“显示命令面板”可以查看所有命令，如图 9-19 所示。

图9-19 查看所有命令



2. 编辑器具有代码智能提示功能，如图 9-20 所示。

图9-20 智能提示



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

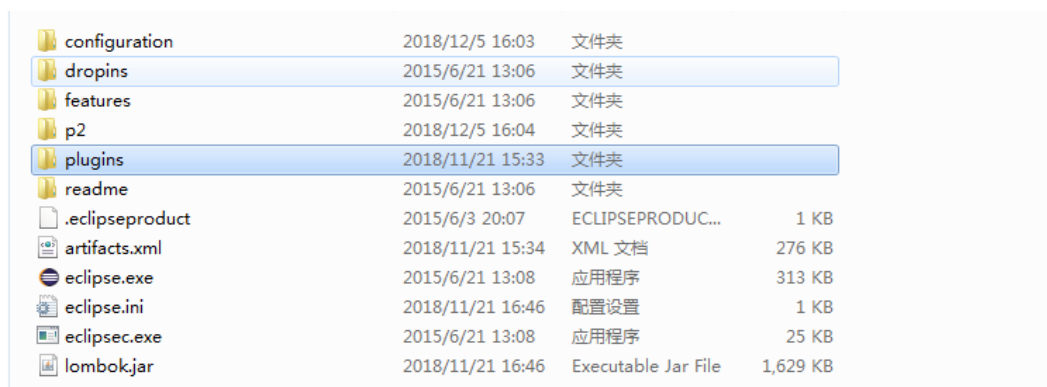
## 9.2 Eclipse-plugin

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

步骤 1 获取 Eclipse 插件。

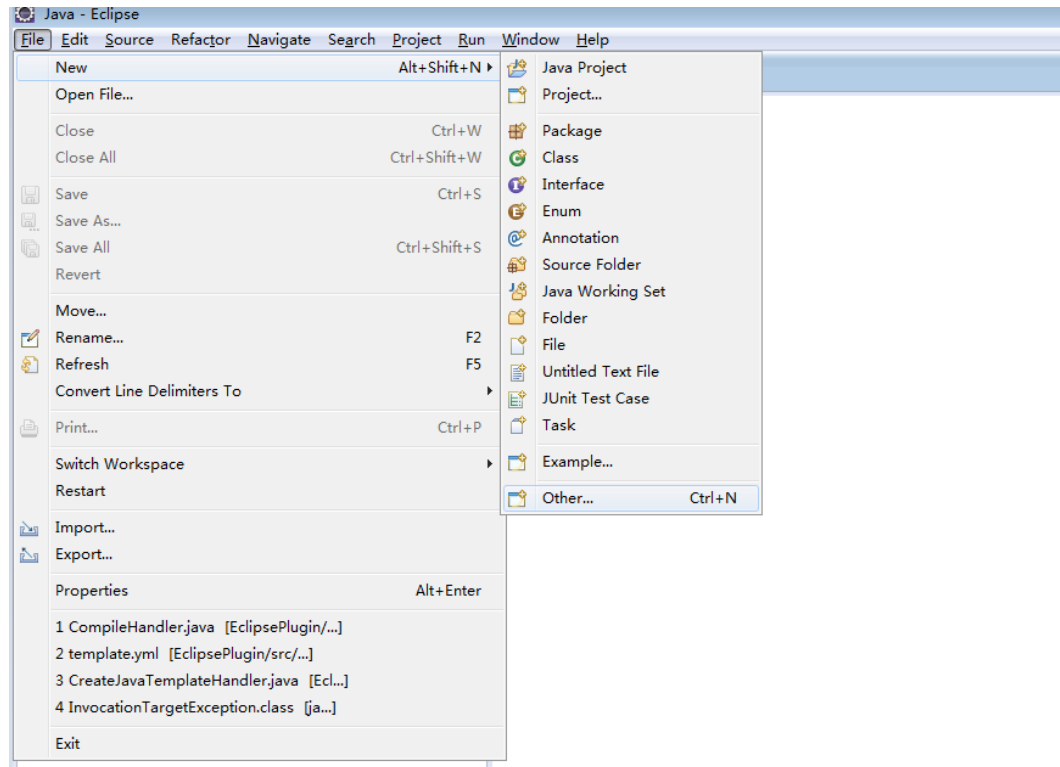
步骤 2 将获取的 Eclipse 插件 jar/zip 包，放入 Eclipse 安装目录下的 plugins 文件夹中，重启 Eclipse，即可开始使用 Eclipse 插件。如图 9-21 所示。

图9-21 安装插件



步骤 3 打开 Eclipse，单击“File”，选择“New > Other”，如图 9-22 所示。

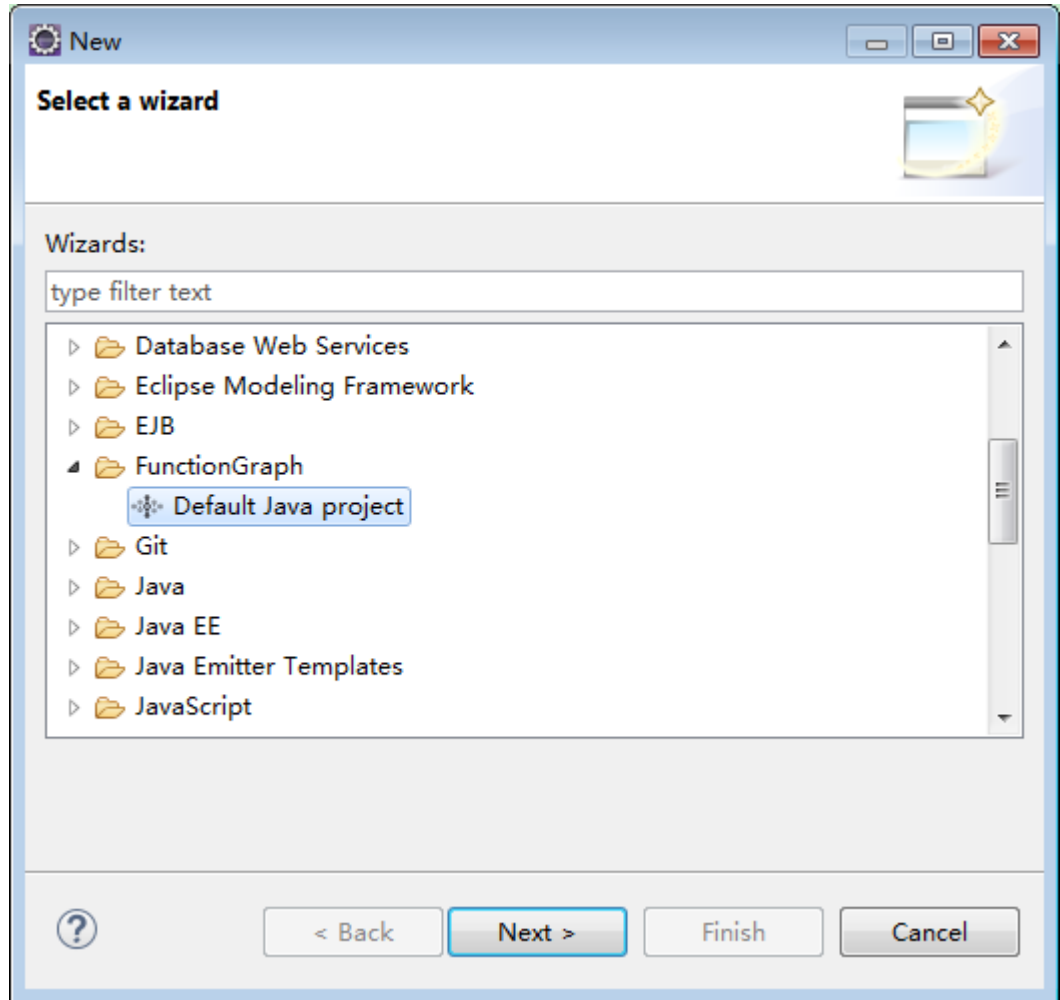
图9-22 新建模板



步骤 4 选择 “FunctionGraph” 文件下的 “Default Java project” 节点。如图 9-23 所示。

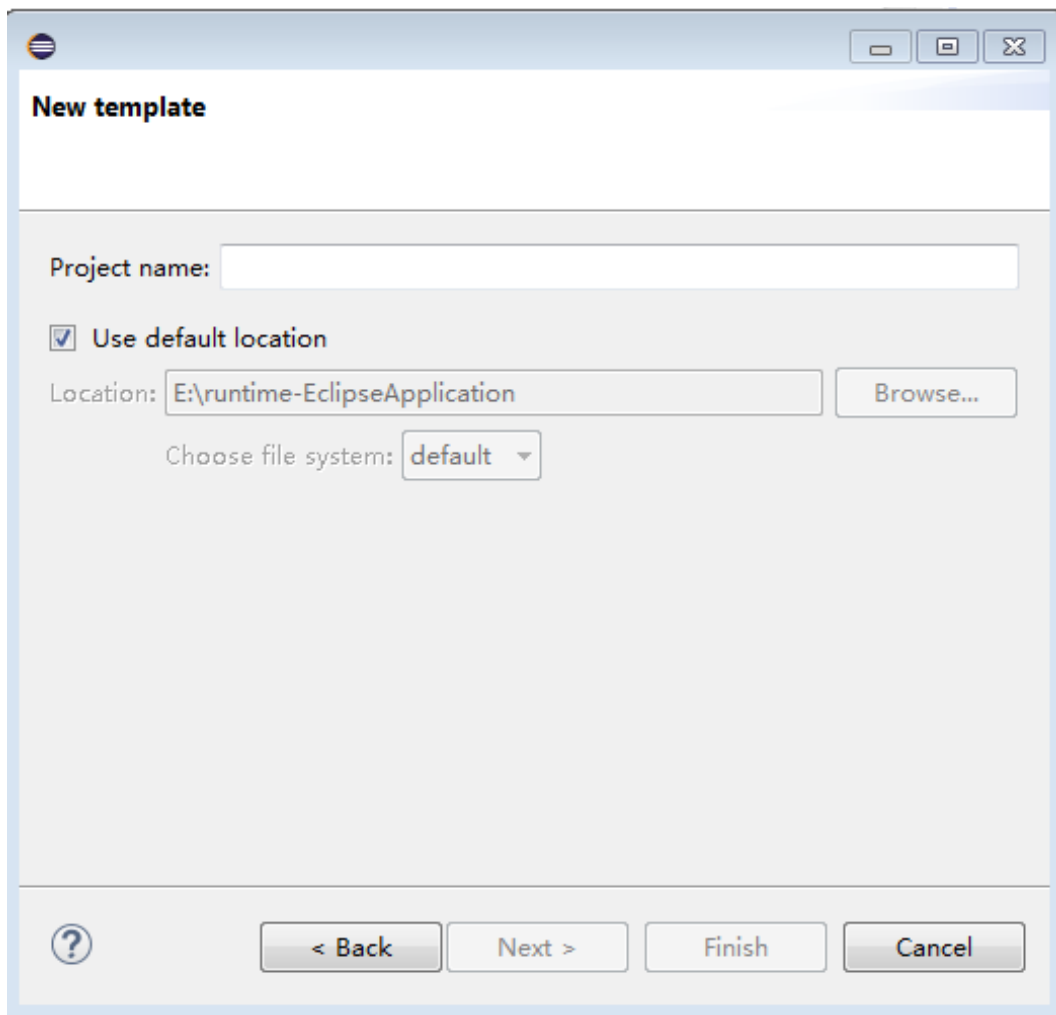


图9-23 选择默认 Java 模板



步骤 5 输入工程名称，选择工程目录（也可以使用默认目录），单击“Finish”完成模板创建。如图 9-24 所示。

图9-24 完成创建



---结束

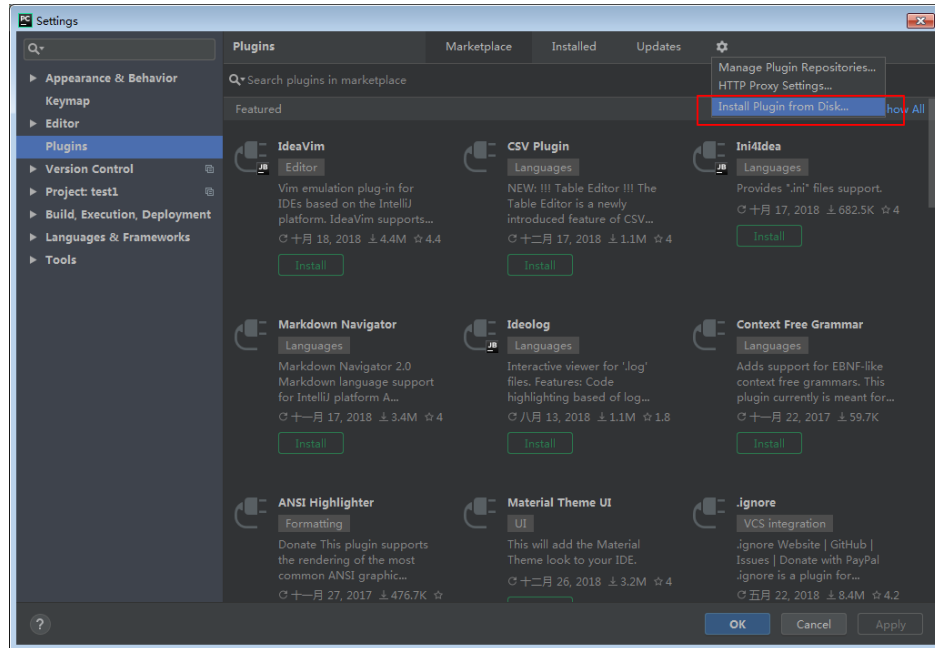
### 9.3 PyCharm-Plugin

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

步骤 1 获取插件。

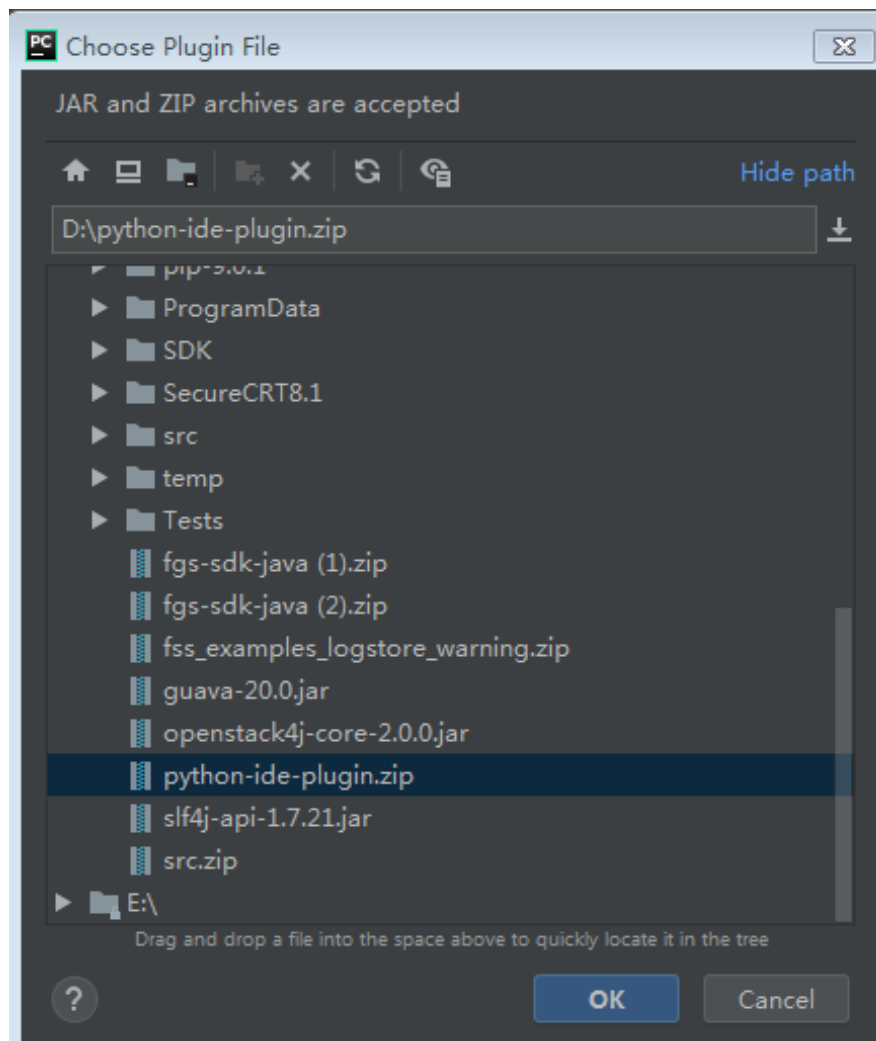
步骤 2 打开 JetBrains PyCharm，单击“File”菜单，选择“Settings”，在弹出界面的菜单中选择“Plugins”页面，单击右上角设置按钮中的“Install plugin from disk...”，如图 9-25 所示。

图9-25 安装 Plugins



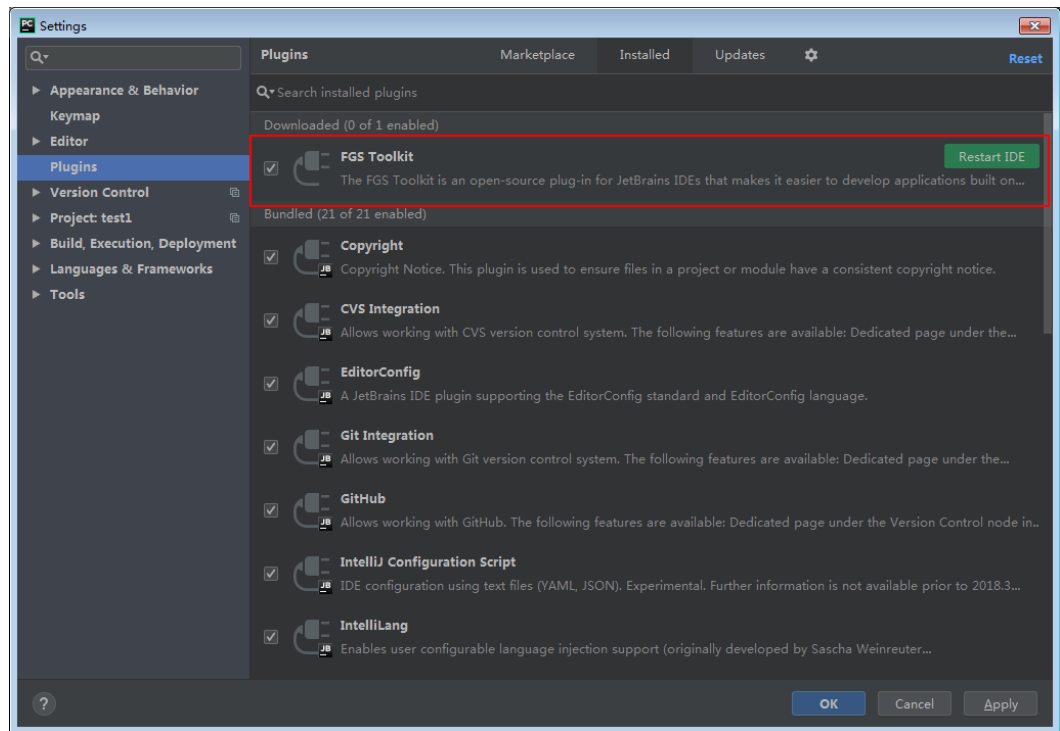
步骤 3 在弹出的界面中，选择插件包，单击“OK”，如图 9-26 所示。

图9-26 选择插件包



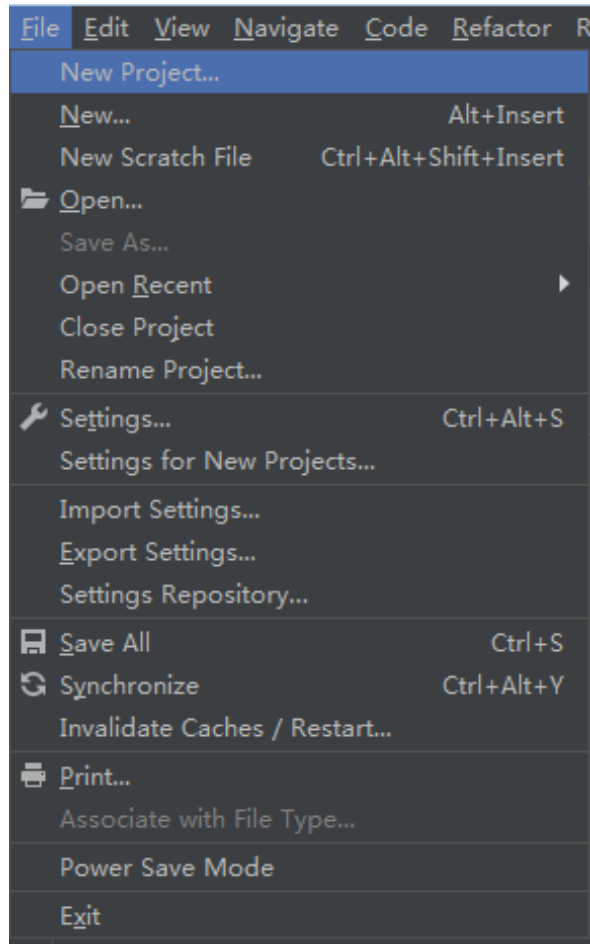
步骤 4 在插件列表中，勾选插件名称，单击“Restart IDE”，如图 9-27 所示。

图9-27 重启 IDE



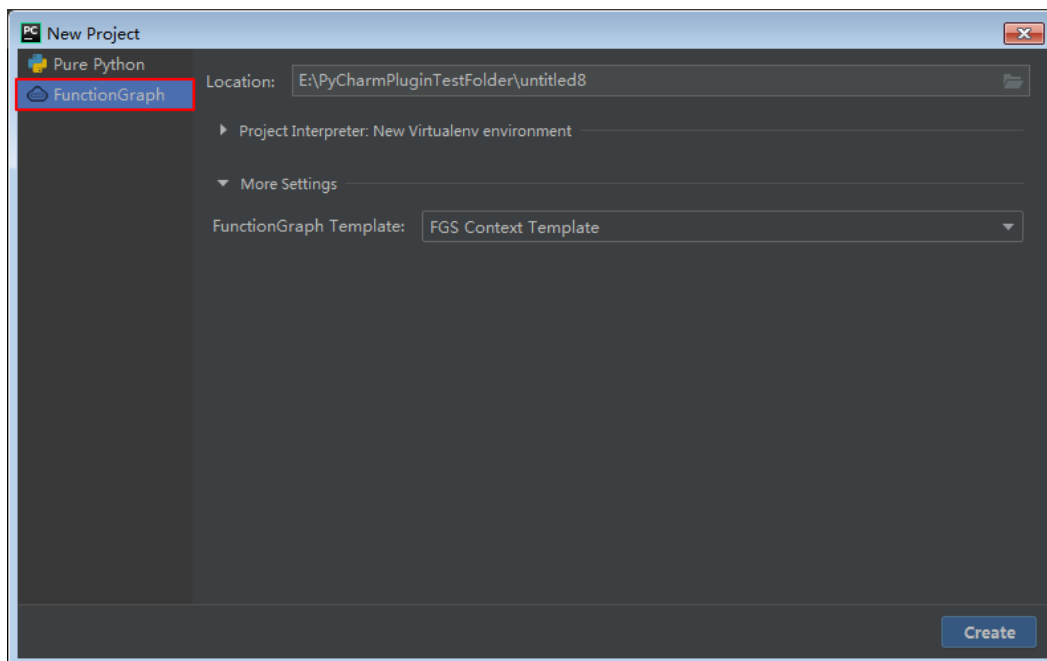
步骤 5 单击“File”菜单，选择“New Project”，如图 9-28 所示。

图9-28 新建工程



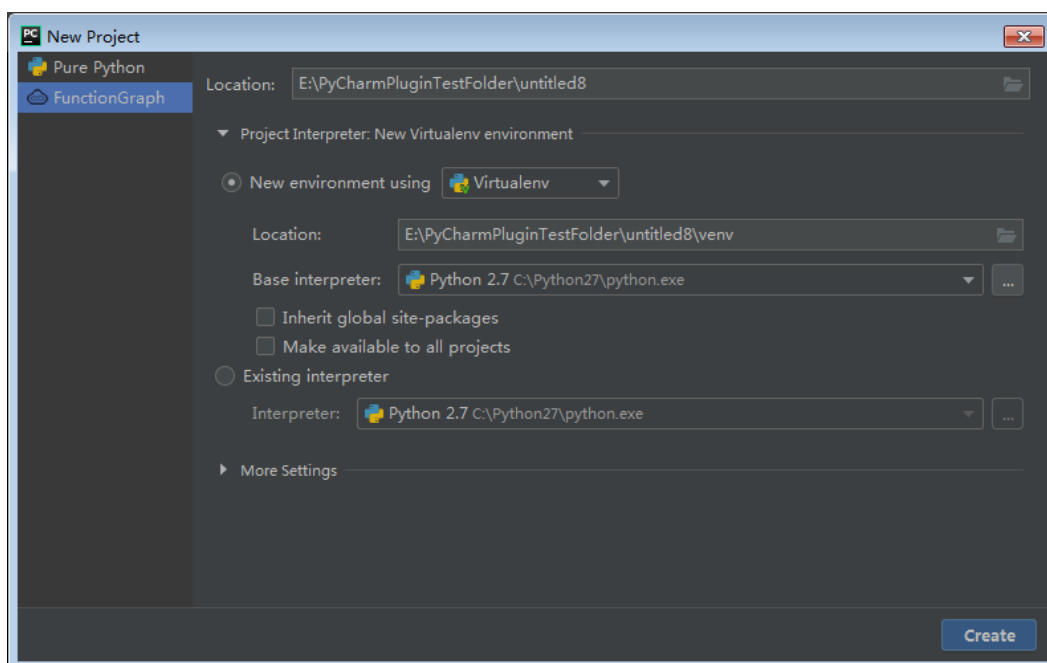
步骤 6 在弹出的新建工程页面中，选择“FunctionGraph”，如图 9-29 所示。

图9-29 FunctionGraph



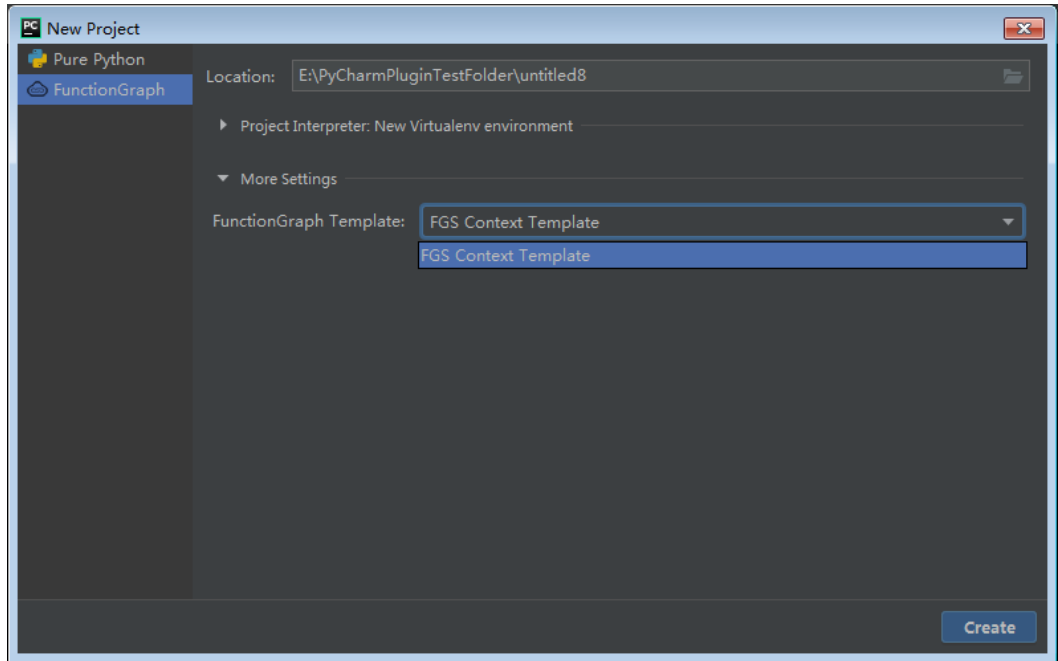
步骤 7 在“Location”栏中选择工程的路径，在“Project Interpreter: New Virtualenv environment”中选择使用 python 的版本。如图 9-30 所示。

图9-30 选择版本



步骤 8 在“More Settings”中选择要创建的模板，如图 9-31 所示。

图9-31 选择模板



**说明**

目前仅支持 python 2.7 的 Context 模板。

步骤 9 单击“Create”，完成创建。

---结束



