天翼云

天翼云函数工作流

用户使用指南

目 录

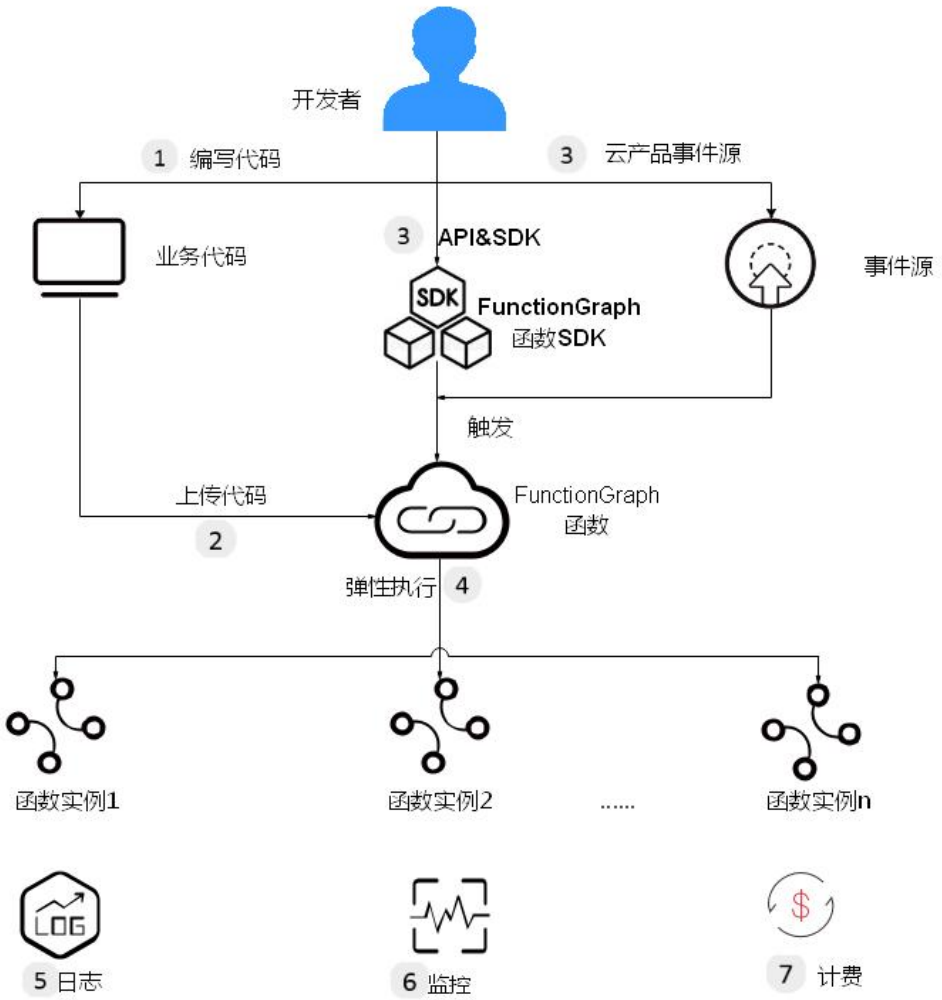
1 产品介绍

1.1 什么是 FunctionGraph

FunctionGraph 是一项基于事件驱动的函数托管计算服务。使用 FunctionGraph 函数，只需编写业务函数代码并设置运行的条件，无需配置和管理服务器等基础设施，函数以弹性、免运维、高可靠的方式运行。

函数使用流程如图 1-1 所示。

图 1-1 函数使用流程



①编写代码

用户编写业务代码，目前支持 Node.js、Python、Java、Go 等语言。

②上传代码

目前支持在线编辑、上传 ZIP 或 JAR 包，从 OBS 引用 ZIP 包等，详情请参考表 1-2。

③API 和云产品事件源触发函数执行

通过 RESTful API 或者云产品事件源触发函数执行，生成函数实例，实现业务功能。

④弹性执行

函数在执行过程中，会根据请求量弹性扩容，支持请求峰值的执行，此过程用户无需配置，由 FunctionGraph 完成，并发数限制请参考 1.6 约束与限制。

⑤查看日志

FunctionGraph 函数实现了与云日志服务的对接，您无需配置，即可查看函数运行日志信息。

⑥查看监控

FunctionGraph 函数实现了与云监控服务的对接，您无需配置，即可查看图形化监控信息。

1.2 产品功能

函数管理

提供控制台管理函数。

- 函数支持 Node.js、Java、Python、Go 等多种运行时语言，同时支持用户自定义运行时，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 运行时语言说明

运行时语言	支持版本
Node.js	6.10、8.10、10.16、12.13、14.18
Python	2.7、3.6、3.9
Java	8.0、11
Go	1.x
定制运行时	-

- 函数支持多种代码导入方式

  支持在线编辑代码、OBS 文件引入、上传 ZIP 包、上传 JAR 包等方式。不同运行时支持的代码上传方式如表 1-2 所示。

表 1-2 代码上传方式说明

运行时	在线编辑	上传 ZIP 文件	上传 JAR 包	从 OBS 上传文件
Node.js	支持	支持	不支持	支持
Python	支持	支持	不支持	支持
Java	不支持	支持	支持	支持
Go	不支持	支持	不支持	支持

运行时	在线编辑	上传 ZIP 文件	上传 JAR 包	从 OBS 上传文件
定制运行时	支持	支持	不支持	支持

日志和监控

提供调用函数的监控指标和运行日志的采集和展示，实时的图形化监控指标展示，在线查询日志，方便用户查看函数运行状态和定位问题。

日志的查询过程请参考 3.7.2.2 管理函数日志。

单个监控指标请参考 3.7.1.1 监控信息说明。

初始化功能

引入 initializer 接口：

- 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，更精准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。

函数流

函数流是用来编排 FunctionGraph 函数的工具，可以将多个函数编排成一个协调多个分布式函数任务执行的工作流。

用户通过在可视化的编排页面，将事件触发器、函数和流程控制器通过连线关联在一个流程图中，每个节点的输出作为连线下一个节点的输入。编排好的流程会按照流程图中设定好的顺序依次执行，执行成功后支持查看工作流的运行记录，方便您轻松地诊断和调试。

函数流功能特性和优势：

- 功能特性
  a. 函数可视化编排
  b. 函数流执行引擎
  c. 错误处理
  d. 可视化监控
- 优势
  a. 使用更少代码快速构建应用程序

函数流允许用户将函数组合编排成一个完整的应用程序，而无需进行代码编写。可以实现快速构建，快速上线。当业务调整时，可以快速调整流程，完成快速上线，无需编写任何代码。

b. 完善的错误处理机制

支持对流程中发生的错误进行捕获和重试，用户可以进行灵活的异常处理。

c. 可视化的编排和监控体验

通过拖拽进行流程编排，学习成本低，可以快速上手。

监控页面使用流程可视化的查看方式，可以做到快速识别问题位置。

HTTP 函数

HTTP 函数专注于优化 Web 服务场景，用户可以直接发送 HTTP 请求到 URL 触发函数执行。在函数创建编辑界面增加类型。HTTP 函数只允许创建 APIG/APIC 的触发器类型，其他触发器不支持。

自定义镜像

支持用户直接打包上传容器镜像，由平台加载并启动运行，调用方式与 HTTP 函数类似。与原本上传代码方式相比，用户可以使用自定义的代码包，不仅灵活也简化了用户的迁移成本。

1.3 产品优势

无服务器管理

自动运行用户代码，用户无需配置或管理服务器，专注于业务创新。

高弹性

根据请求的并发数量自动调度资源运行函数，实现透明、准确和实时的伸缩，应付业务峰值的访问。

用户无需关心峰值和空闲时段的资源需要申请多少资源，系统根据请求的数量自动扩容/缩容。自动负载均衡将请求分发到函数运行实例。

事件触发

通过事件触发机制，集成多种云服务，满足不同场景需求，获得高效的开发体验。

与云日志服务、云监控服务对接，无需任何配置，即可查询函数日志和监控告警信息，快速排查故障。

高可用

函数运行实例出现异常，系统会启动新的实例处理后续的请求，故障函数实例占用资源将会回收使用。

动态资源指定

函数执行时可根据业务需要动态指定资源规格，最小化资源占用，灵活调度节省成本。

1.4 应用场景

函数工作流应用场景，如实时文件处理、实时数据流处理、Web 移动应用后端和人工智能场景。

场景一：事件驱动类应用

以事件驱动的方式执行服务，按需供给，开发者无需关注业务波峰波谷，节省闲时成本，最终降低运维成本。比如文件处理、图片处理、视频直播/转码、实时数据流处理、IoT 规则/事件处理等。

- **实时文件处理**

  客户端上传文件到 OBS，触发 FunctionGraph 函数，在上传数据后立即进行处理。可以使用 FunctionGraph 实时创建图像缩略图、转换视频编码、进行数据文件汇聚、筛选等。

  其优势有：

  - 灵活扩展，业务爆发时可以自动调度资源运行更多函数实例以满足处理需求。
  - 事件触发，通过上传文件到 OBS，触发 FunctionGraph 函数进行文件处理。
  - 按需收费，只有对函数处理文件数据的时间进行计费，无需购买冗余的资源用于非峰值处理。

- **实时数据流处理**

  使用 FunctionGraph 和 DIS 处理实时流数据，跟踪应用程序活动、顺序事务处理、分析数据流、整理数据、生成指标、筛选日志、建立索引、分析社交媒体以及遥测和计量 IoT 设备数据。

  其优势有：

  - 事件触发，通过 DIS 流采集数据，批量数据通过事件触发处理函数进行处理。
  - 灵活扩展，业务爆发时可以自动调度资源运行更多函数实例以满足处理需求。
  - 按需收费，只有对函数处理文件数据的时间进行计费，无需购买冗余的资源用于非峰值处理。

场景二：Web 类应用

使用 FunctionGraph 和其他云服务或租户 VM 结合，用户可以快速构建高可用，自动伸缩的 Web/移动应用后端。比如小程序、网页/App、聊天机器人、BFF 等。

其优势有：

- 高可用，利用 OBS，Cloud Table 的高可用性实现网站数据的高可靠性，利用 API Gateway 和 FunctionGraph 的高可用性实现网站逻辑的高可用。
- 灵活扩展，业务爆发时可以自动调度资源运行更多函数实例以满足处理需求。
- 按需收费，只有对函数处理文件数据的时间进行计费，无需购买冗余的资源用于非峰值处理。

场景三：AI 类应用

各行各业智能化深入带来更多的应用开发场景，通常需要集成各类服务快速上线。比如三方服务集成、AI 推理、人脸识别、车牌识别。

其优势有：

- 快速搭建，用户上传图像后触发函数工作流执行调用文字识别/内容检测服务针对图像进程处理，并将结果以 JSON 结构化数据返回。按需使用函数与多个智能服务集成，形成丰富的应用处理场景。并随时根据业务改变对函数处理过程做调整，实现业务灵活变更。
- 简化运维，用户只需开通相关云服务并在函数服务中编写业务逻辑，无需配置或管理服务器，专注于业务创新。业务爆发时可以自动调度资源运行更多函数实例以满足处理需求。
- 按需计费，只有对函数执行的时间及各智能服务处理进行计费，无需购买冗余的资源用于非峰值处理。

1.5 函数类型

1.5.1 事件函数

概述

FunctionGraph 支持事件类型函数。事件是指用于触发函数，通常为 JSON 格式的请求。用户作为事件源（事件的生产者），可以通过云服务平台或 Cloud IDE 触发函数并进行执行。在函数创建界面可以选择函数类型，事件类型的函数不受触发器类型的限制，当前 FunctionGraph 支持的所有类型触发器均可用于触发事件函数。

说明

1. FunctionGraph 原生支持事件类型函数，在函数创建界面默认选择该类型；
2. 测试函数时在参数配置界面输入用户指定的事件 JSON 即可完成函数触发；
3. 用户也可以通过 FunctionGraph 支持的触发器进行事件函数触发；

优势

- 单机编程体验，简单易用

  事件类型函数可以在 FunctionGraph 函数界面或 Cloud IDE 界面进行函数编辑或代码包上传，一键式完成函数云上部署，用户无需关心并处理函数的并发、故障恢复等问题。
- 高性能极速运行时

事件函数提供毫秒级函数启动、函数扩容、函数调用，秒级故障中断检测及秒级故障恢复。

- 便捷完备的工具链

  提供完备的日志、调用链、debug 及监控能力，支撑开发者“三步”上线函数应用。

限制

事件函数受限于事件格式（事件源），开发者在开发过程中需要遵循函数平台的函数开发规则。

1.5.2 HTTP 函数

概述

FunctionGraph 支持两种函数类型，事件函数和 HTTP 函数。HTTP 函数专注于优化 Web 服务场景，用户可以直接发送 HTTP 请求到 URL 触发函数执行，从而使用自己的 Web 服务。HTTP 函数只允许创建 APIG/APIC 的触发器类型，其他触发器不支持。

说明

1. HTTP 函数支持 HTTP/1.1 协议。
2. 在函数创建页面，新增一种函数类型“HTTP 函数”；
3. HTTP 函数执行入口需要设置为 bootstrap，用户直接写启动命令，**端口统一开放成 8000；**
4. **若运行用户 JAR 包，bootstrap 中建议增加 JVM 参数-Dfile.encoding=utf-8，否则可能会出现中文乱码。**

优势

- 丰富的框架支持

  您可以使用常见的 Web 框架（例如 Nodejs Web 框架：Express、Koa）编写 Web 函数，也可以将您本地的 Web 框架服务以极小的改造量快速迁移上云。

- 减少请求处理环节

  函数可以直接接收并处理 HTTP 请求，API 网关不再需要做 json 格式转换，减少请求处理环节，提升 Web 服务性能。

- 编写体验舒适化

  HTTP 函数的编写体验更贴近编写原生 Web 服务，可以使用 Node.js 原生接口，保证和本地开发服务体验一致。

限制

- HTTP 函数只允许创建 APIG 共享版、APIG 专享版、APIC 的触发器类型，其他触发器不支持。
- 同一个函数支持绑定多个 API 触发器，但所有 API 都必须在一个 APIG 服务下。
- 针对 http 函数，用户的 http 响应体不超过 6M。
- 不支持长时运行和异步调用，不支持重试。

1.6 约束与限制

帐户资源限制

表 1-3 帐户资源说明表

资源	限制
单个帐户下最大允许创建的函数个数	400
单个函数下最大允许创建的版本个数	10
单个函数下最大允许创建的别名个数	10
前端页面展示时，单个代码部署包大小（压缩为.zip/.jar 文件）	3MB
前端页面上传时，单个代码部署包大小（压缩为.zip/.jar 文件）	10MB
调用函数接口时，在线编辑单个函数代码部署包大小（压缩为.zip/.jar 文件）	50MB
调用函数接口时，单个代码部署包原始代码大小	• zip 格式：解压后原始代码大小为 1500M • OBS 桶：最大可上传 300M 压缩后的代码包
单个帐户下最大允许部署包大小	10 GB
单个帐户下函数并发执行数	100
单个帐户下创建预留实例个数	90（单个租户下函数并发执行数 *90%）
单个函数下所有环境变量的大小	总长度不能超过 4096 个字符

函数运行资源限制

表 1-4 函数运行资源限制说明

资源	默认值
临时磁盘空间（“/tmp”空间）	512MB
文件描述符	1024
进程和线程数（总和）	1024
单个请求最大执行时长	900 秒

资源	默认值
函数同步调用请求正文有效负载大小	6MB
函数同步调用响应正文有效负载大小	6MB
函数异步调用请求正文有效负载大小	256KB
函数导入的资源大小	zip 格式压缩文件，大小 50MB 以内
函数导出资源包大小	50MB 以内

说明

- 函数同步调用响应正文有效负载大小：返回的字符串或返回体序列化后的 json 字符串默认不大于 6MB。具体数据大小会随 FunctionGraph 系统后台设置产生变化，因为系统后台判断的是序列化之后的数据大小，所以会存在字节级别的误差，误差范围为 6MB±100bytes。
- FunctionGraph 控制台不建议调用执行时间超过 90 秒的函数；若需要调用执行时间超过 90 秒的函数，请使用异步调用的方式。

1.7 权限管理

如果您需要对 FunctionGraph 的函数资源，给企业中的员工设置不同的访问权限，以达到不同员工之间的权限隔离，您可以使用统一身份认证服务（Identity and Access Management，简称 IAM）进行精细的权限管理。该服务提供用户身份认证、权限分配、访问控制等功能，可以帮助您安全的控制公有云资源的访问。

通过 IAM，您可以在帐号中给员工创建 IAM 用户，并使用策略来控制他们对云资源的访问范围。例如您的员工中有负责软件开发的人员，您希望他们拥有 FunctionGraph 的使用权限，但是不希望他们拥有删除等高危操作的权限，那么您可以使用 IAM 为开发人员创建用户，通过授予仅能使用 FunctionGraph，但是不允许删除的权限策略，控制他们对 FunctionGraph 资源的使用范围。

如果帐号已经能满足您的要求，不需要创建独立的 IAM 用户进行权限管理，您可以跳过本章节，不影响您使用 FunctionGraph 服务的其它功能。

FunctionGraph 权限

默认情况下，新建的 IAM 用户没有任何权限，您需要将其加入用户组，并给用户组授予策略，才能使得用户组中的用户获得策略定义的权限，这一过程称为授权。授权后，用户就可以基于策略对云服务进行操作。

FunctionGraph 资源通过物理区域划分，为项目级服务。授权时，“作用范围”需要选择“区域级项目”，然后在各区域对应的项目中设置相关权限，并且该权限仅对此项目生效；如果在“所有项目”中设置权限，则该权限在所有区域项目中都生效。访问 FunctionGraph 时，需要先切换至授权区域。

根据授权精细程度分为角色和策略。

- 角色：IAM 最初提供的一种根据用户的工作职能定义权限的粗粒度授权机制。该机制以服务为粒度，提供有限的服务相关角色用于授权。由于各服务之间存在业务依赖关系，因此给用户授予角色时，可能需要一并授予依赖的其他角色，才能正确完成业务。角色并不能满足用户对精细化授权的要求，无法完全达到企业对权限最小化的安全管控要求。
- 策略：IAM 最新提供的一种细粒度授权的能力，可以精确到具体服务的操作、资源以及请求条件等。基于策略的授权是一种更加灵活的授权方式，能够满足企业对权限最小化的安全管控要求。

如表 1-6 所示，包括了 FunctionGraph 的所有系统权限。

表 1-5 系统权限说明

系统角色/策略名称	描述	类别	依赖关系
FunctionGraph FullAccess	函数工作流服务所有权限	系统策略	无
FunctionGraph ReadOnlyAccess	函数工作流服务只读权限	系统策略	无
FunctionGraph CommonOperations	函数工作流（FunctionGraph）调用者，具有查询函数和触发器，以及调用函数的权限	系统策略	无

列出了 FunctionGraph 常用操作与系统权限的授权关系，您可以参照该表选择合适的系统权限。

表 1-6 常用操作与系统权限之间的关系

操作	FunctionGraph ReadOnlyAccess	FunctionGraph CommonOperations	FunctionGraph FullAccess
创建函数	×	×	√
查询函数	√	√	√
修改函数	×	×	√
删除函数	×	×	√
调用函数	×	√	√
查看函数日志	√	√	√
查看函数指标数据	√	√	√

1.8 基本概念

函数

函数是处理事件的自定义代码。

事件源

事件源是发布事件的公有云服务或自定义应用程序。

同步调用

同步调用指的是客户端请求需要明确等到响应结果，也就是说这样的请求必须得调用到用户的函数，并且等到调用完成才返回。

异步调用

异步调用是指客户端不关注请求调用的结果，服务端收到请求后将请求排队，排队成功后请求就返回，服务端在空闲的情况下会逐个处理排队的请求。

触发器

触发函数执行的事件。

函数流

用户通过在 UI 界面拖拽组件、配置组件和连接组件进行可视化编排，创建函数流任务，完成复杂场景的编排。

单实例多并发

单实例多并发是指单个实例可以同时处理的请求数量。

自定义镜像函数

用户直接打包上传容器镜像，由平台加载并启动运行。

自定义运行

自定义函数执行的脚本和文件。

函数日志

函数调用过程中产生的日志信息。

函数监控

函数执行过程中的监控信息。

函数版本

函数从开发、测试、生产过程中发布一个或多个版本，实现对函数代码的管理。对于发布的每个版本的函数、环境变量会另存为相应版本的快照，函数代码发布后，可以根据实际需要修改版本配置信息。

函数别名

用户可以创建别名，指向特定函数版本。别名的优势在于：如果需要回滚到之前的函数版本，则可以将相应别名指向该版本，不再需要修改代码信息。

函数别名支持绑定两个版本，一个对应版本和开启灰度版本，并且支持配置同一个别名下两个不同版本分流权重。

依赖包

依赖包管理模块统一管理用户所有的依赖包，用户可以通过本地上传和 obs 地址的形式上传依赖包，并为依赖包命名。

bootstrap 文件

bootstrap 文件是 HTTP 函数的启动文件，HTTP 函数仅支持读取 bootstrap 作为启动文件名称，其它名称将无法正常启动服务。

1.9 与其他服务的关系

FunctionGraph 服务与以下云服务的对接，实现相关功能，如表 1-8 所示。

表 1-7 对接服务

服务名称	实现功能
API 网关（API Gateway）	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。
对象存储服务（OBS）	构建 FunctionGraph 函数来处理 OBS 存储桶事件，例如对象事件或删除事件。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。
云监控服务（CES）	FunctionGraph 函数实现了与云监控服务对接，函数上报云监控服务的监控指标，用户可以通过云监控服务来查看函数产生的监控指标和告警信息。
虚拟私有云（VPC）	函数支持用户创建虚拟私有云（VPC）并访问自己 VPC 内的资源，同时支持通过 SNAT 方式绑定 EIP 访问外网。

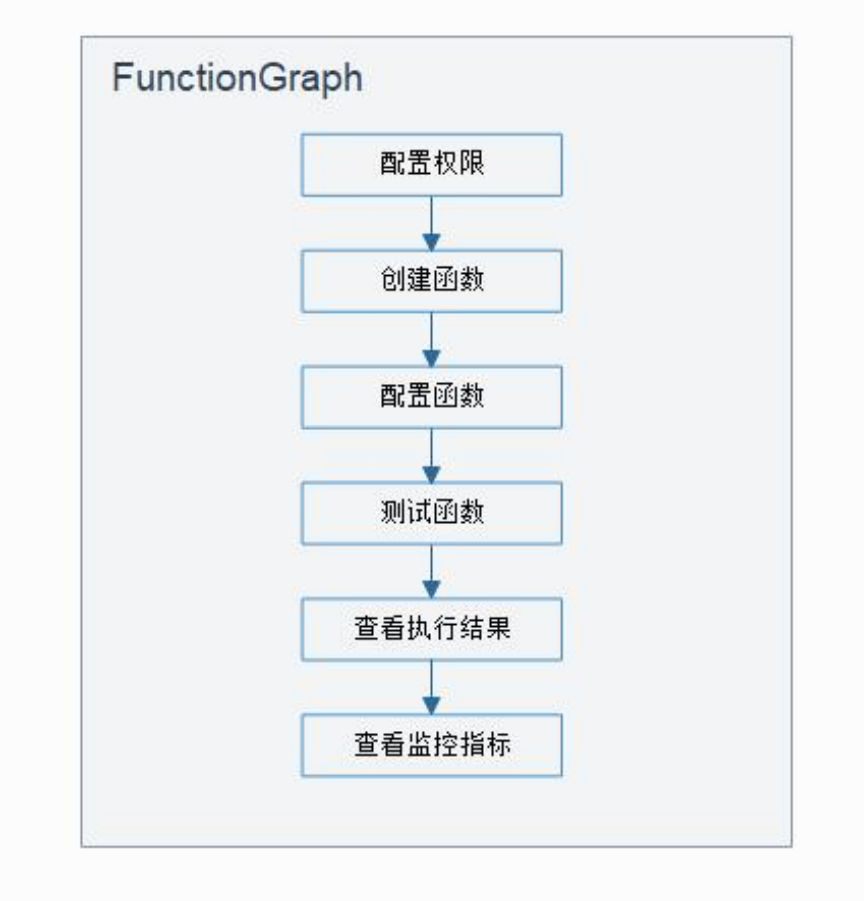
2 快速入门

2.1 FunctionGraph 入门简介

使用流程

函数工作流 FunctionGraph 是一项基于事件驱动的函数托管计算服务。使用 FunctionGraph 函数，只需编写业务函数代码并设置运行的条件，无需配置和管理服务器等基础设施，函数以弹性、免运维、高可靠的方式运行。

使用 FunctionGraph 快速创建函数的流程如下：



1. 配置权限：确保登录的用户已有“FunctionGraph Administrator”权限。
2. 创建函数：选择使用空白模板创建函数、示例代码创建函数、容器镜像部署函数。
3. 配置函数：配置代码源或修改其他参数配置。
4. 测试函数：创建测试事件来调试函数。
5. 查看执行结果：在函数详情页面，根据配置的测试事件，查看执行结果。
6. 查看监控指标：在函数详情页面的“监控”页签，查看函数监控指标。

2.2 使用空白模板创建函数

概述

本章节介绍如何在函数工作流控制台使用空白模板快速开发一个简单的 Hello World 函数 。以创建 HelloWorld 函数为例，介绍函数的创建及测试过程，供您快速体验 FunctionGraph 函数的基本功能。

步骤一：准备环境

本章节所有操作均默认具有操作权限，请确保您登录的用户已有“FunctionGraph Administrator”权限，即 FunctionGraph 服务管理员权限。

步骤二：创建函数

1. 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。
2. 单击右上方的“创建函数”，进入“创建函数”页面，开始创建空白函数。
3. 参考图 2-1，函数名称输入“HelloWorld”，其他参数保持默认，完成后单击“创建函数”。

图 2-1 基本信息配置

4. 配置代码源，复制如下代码至代码窗，单击“部署”。

   样例代码实现的功能是：获取测试事件，打印测试事件信息。

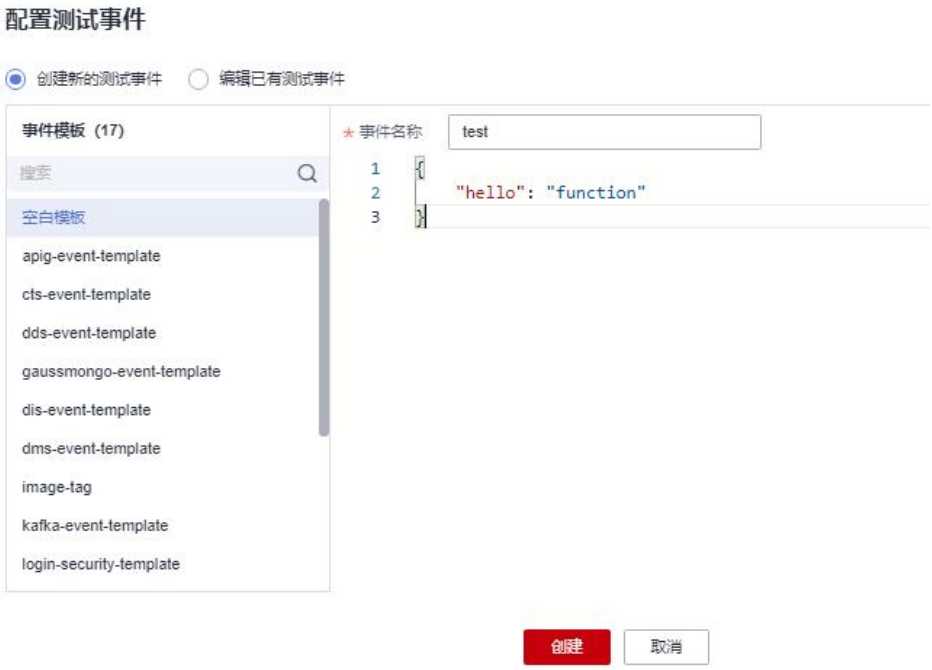
```
exports.handler = function (event, context, callback) {
    const error = null;
    const output = `Hello message: ${JSON.stringify(event)}`;
    callback(error, output);
}
```

步骤三：测试函数

1. 在函数详情页，单击“测试”，在弹窗中创建新的测试事件。
2. 选择“空白模板”，事件名称输入“test”，测试事件修改为如下所示，完成后单击“创建”。

```
{
    "hello": "function"
}
```

图 2-2 配置测试事件



步骤四：查看执行结果

单击 test 事件的“测试”，执行后，在右侧查看执行结果。

- “函数返回”显示函数的返回结果。
- “日志”部分显示函数执行过程中生成的日志。
- “执行摘要”部分显示“日志”中的关键信息。

图 2-3 查看执行结果

说明

此页面最多显示 2K 日志，了解函数更多日志信息，请参考 3.7.2.1 查看函数日志。

步骤五：查看监控指标

在函数详情页面，选择“监控”页签。

- 在“监控”页签，先选择“指标”，再选择时间粒度（5 分钟、15 分钟、1 小时），查看函数运行状态。
- 可以查看的指标有：调用次数、错误次数、运行时间（包括最大运行时间、最小运行时间、平均运行时间）、被拒绝次数。

步骤六：删除函数

1. 在函数详情页面，单击右上角的“操作 > 删除函数”。
2. 在确认框继续单击“确认”，及时释放资源。

2.3 使用模板创建函数

概述

FunctionGraph 平台提供了函数模板，本章节介绍如何在创建函数时选择模板，实现模板代码、运行环境自动填充，快速构建应用程序。

步骤一：准备环境

本章节所有操作均默认具有操作权限，请确保您登录的用户已有“FunctionGraph Administrator”权限，即 FunctionGraph 服务管理员权限。

步骤二：创建函数

1. 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。
2. 单击右上方的“创建函数”，进入“创建函数”页面，使用模板创建函数。
3. 参考图 2-4，选择如下模板并单击“使用模板”。

图 2-4 选择模板

4. 函数名称输入“context”，“委托名称”选择已创建的任意委托，其他设置保持不变，单击“创建函数”。

说明

若不配置委托，在触发函数时，执行结果会返回

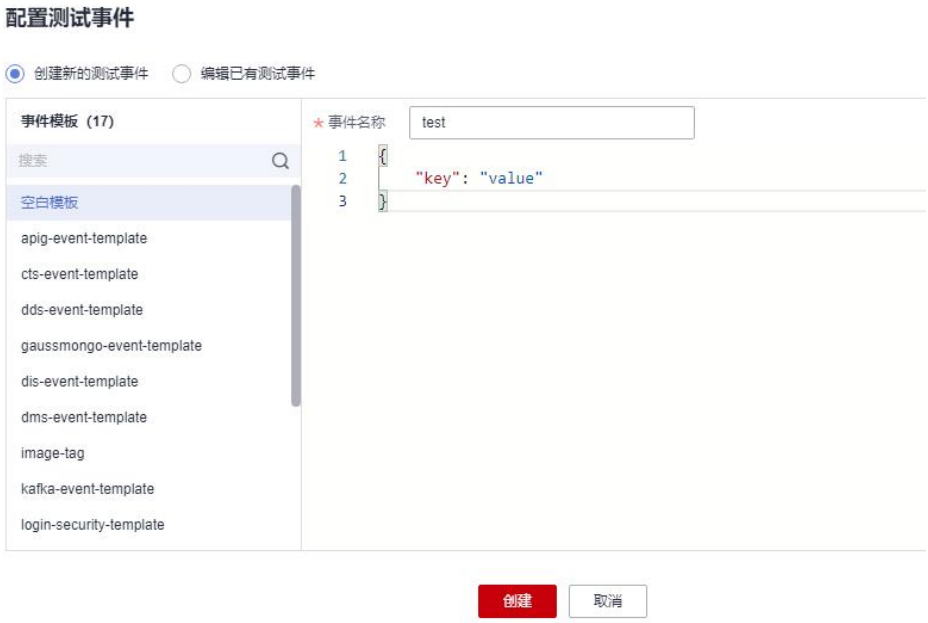
```
Failed to access other services because no temporary AK, SK, or token has been obtained. Please set an agency.
```

图 2-5 填写基本信息

步骤三：测试函数

1. 在函数详情页，单击“测试”，在弹窗中创建新的测试事件。
2. 选择“空白模板”，事件名称输入“test”，完成后单击“创建”。

图 2-6 配置测试事件



步骤四：查看执行结果

单击 test 事件的“测试”，成功执行后，在右侧查看执行结果。

- “函数返回”显示函数的返回结果。
- “日志”部分显示函数执行过程中生成的日志。
- “执行摘要”部分显示“日志”中的关键信息。

说明

此页面最多显示 2K 日志，了解函数更多日志信息，请参考 3.7.2.1 查看函数日志。

步骤五：查看监控指标

在函数详情页面，选择“监控”页签。

- 在“监控”页签，先选择“指标”，再选择时间粒度（5 分钟、15 分钟、1 小时），查看函数运行状态。
- 可以查看的指标有：调用次数、错误次数、运行时间（包括最大运行时间、最小运行时间、平均运行时间）、被拒绝次数。

步骤六：删除函数

1. 在函数详情页面，单击右上角的“操作 > 删除函数”。
2. 在确认框继续单击“确认”，及时释放资源。

2.4 使用容器镜像部署函数

2.4.1 开发 HTTP 函数示例

概述

使用自定义镜像开发 HTTP 函数时，用户需要在镜像中实现一个 http server，并监听 8000 端口接收请求。备注：HTTP 函数只支持 APIG 触发器。

步骤一：准备环境

本章节所有操作均默认具有操作权限，请确保您登录的用户已有“FunctionGraph Administrator”权限，即 FunctionGraph 服务管理员权限。

步骤二：制作镜像

以在 linux x86 64 位系统上制作镜像为例。

1. 创建一个空文件夹

```
mkdir custom_container_http_example && cd custom_container_http_example
```

2. 以 Nodejs 语言为例，实现一个 Http Server。

   创建一个 main.js 文件，引入 express 框架，接收 POST 请求，打印请求 Body 到标准输出并返回 Hello FunctionGraph, method POST 给客户端。

```
const express = require('express');

const PORT = 8000;

const app = express();
app.use(express.json());

app.post('/*', (req, res) => {
   console.log('receive', req.body);
   res.send('Hello FunctionGraph, method POST');
});

app.listen(PORT, () => {
  console.log(`Listening on http://localhost:${PORT}`);
});
```

3. 创建一个 package.json 文件，此文件用于向 npm 提供信息，使其能够识别项目以及处理项目的依赖关系。

```
{
  "name": "custom-container-http-example",
  "version": "1.0.0",
  "description": "An example of a custom container http function",
  "main": "main.js",
  "scripts": {},
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
```

```
    "express": "^4.17.1"
  }
}
```

- name：值为项目名。
- version：值为项目版本。
- main：列举文件为库的主入口。
- dependencies：列出 npm 上可用的项目的所有依赖项。

4. 创建 Dockerfile 文件

```
FROM node:12.10.0

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

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

COPY --chown=${USER_ID}:${GROUP_ID} main.js ${HOME}
COPY --chown=${USER_ID}:${GROUP_ID} package.json ${HOME}

RUN cd ${HOME} && npm install

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

RUN find ${HOME} -type d | xargs chmod 500 &&
find ${HOME} -type f | xargs chmod 500

USER ${USER_NAME}
WORKDIR ${HOME}

EXPOSE 8000
ENTRYPOINT ["node", "main.js"]
```

- FROM：指定基础镜像为 node:12.10.0，基础镜像必须设置，值可修改。
- ENV：设置环境变量，设置 HOME 环境变量为/home/custom_container，设置 GROUP_NAME 和 USER_NAME 为 custom_container，USER_ID 和 GROUP_ID 为 1003，这些环境变量必须设置，值可修改。
- RUN：格式为 RUN <命令>，例如 RUN mkdir -m 550 ${HOME}表示构建容器时创建${USER_NAME}用户的 home 目录。
- USER：切换${USER_NAME}用户。
- WORKDIR：切换工作目录到${USER_NAME}用户的 home 目录下。
- COPY：将 main.js 和 package.json 拷贝到容器的${USER_NAME}用户的 home 目录下。
- EXPOSE：暴露容器的 8000 端口，请勿修改。
- ENTRYPOINT：使用 node main.js 命令启动容器，请勿修改。

说明

1. 可以使用任意基础镜像。
2. 在云上环境会默认使用 uid 1003，gid 1003 启动容器。uid、gid 可以在函数页面的设置 > 常规设置 >容器镜像覆盖板块中修改，但不可以是 root 或其他保留 id。

5. 构建镜像

   指定镜像的名称为 custom_container_http_example，版本为 latest，“.”指定 Dockerfile 所在目录，镜像构建命令将该路径下所有的内容打包给容器引擎帮助构建镜像。

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

步骤三：本地验证

1. 启动 docker 容器

```
docker run -u 1003:1003 -p 8000:8000 custom_container_http_example:latest
```

2. 打开一个新的命令行窗口，向开放的 8000 端口发送消息，访问模板代码中指定的 /invoke 路径

```
curl -XPOST -H 'Content-Type: application/json' -d '{"message":"HelloWorld"}' localhost:8000/helloworld
```

按照模块代码中返回

```
Hello FunctionGraph, method POST
```

3. 在容器启动端口可以看到

```
receive {"message":"HelloWorld"}
```

```
[root@ecs-74d7 ~]# docker run -u 1003:1003 -p 8000:8000 custom_container_http_example:latest
Listening on http://localhost:8000
receive { message: 'HelloWorld' }
```

或者使用 docker logs 命令获取容器的日志

```
[root@ecs-74d7 custom_container_http_example]# docker logs 1354c3580638
Listening on http://localhost:8000
receive { message: 'HelloWorld' }
[root@ecs-74d7 custom_container_http_example]#
```

步骤四：上传镜像

1. 登录容器镜像服务控制台，在左侧导航栏选择“我的镜像”。
2. 单击右上角的“客户端上传”或“页面上传”。
3. 根据指示上传镜像。

4. 上传成功后，在“我的镜像”界面可查看。

步骤五：创建函数

1. 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。
2. 单击右上方的“创建函数”，进入“创建函数”页面，使用容器镜像部署函数。
3. 填写基本信息。

- 函数类型：选择“HTTP 函数”
- 函数名称：输入“custom_container_http”
- 容器镜像：选择上一步上传到 SWR 的镜像。
- 现有委托：使用包含 SWR Admin 权限的委托。

4. 完成后单击“创建函数”。

步骤六：测试函数

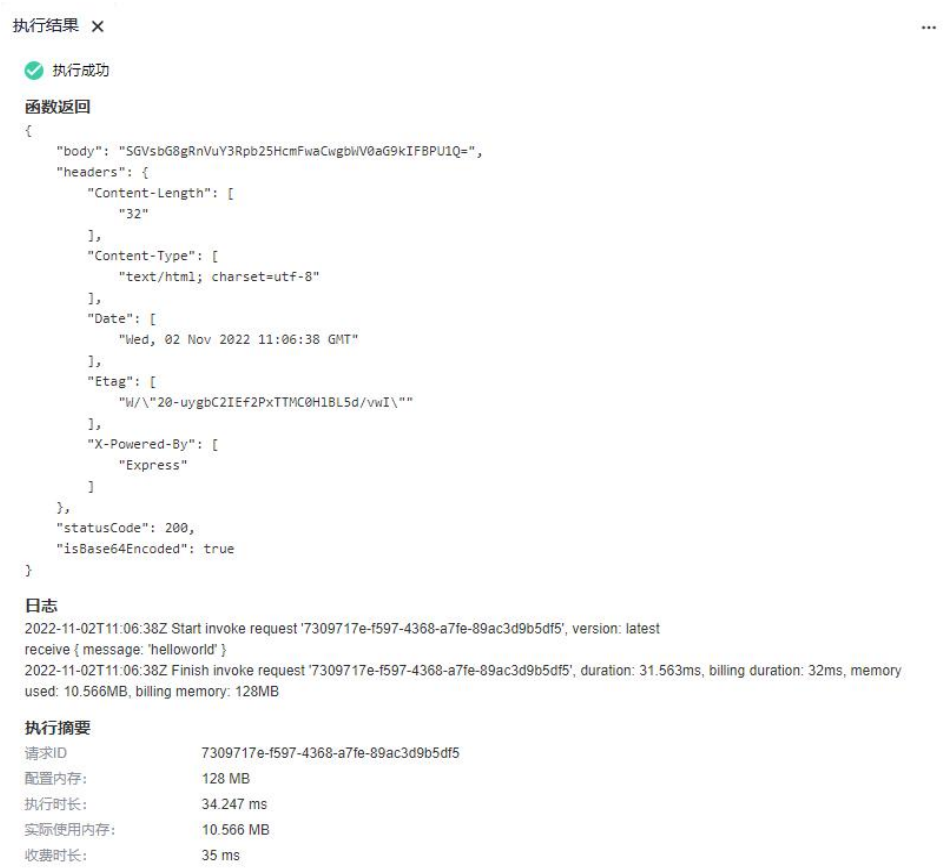
1. 在函数详情页，单击“测试”，在弹窗中创建新的测试事件。
2. 选择“apig-event-template”，事件名称输入“helloworld”，测试事件修改为如下所示，完成后单击“创建”。

```
{
    "body": "{\"message\": \"helloworld\"}",
    "requestContext": {
        "requestId": "11cdcdcf33949dc6d722640a13091c77",
        "stage": "RELEASE"
    },
    "queryStringParameters": {
        "responseType": "html"
    },
    "httpMethod": "POST",
    "pathParameters": {},
    "headers": {
        "Content-Type": "application/json"
    },
    "path": "/helloworld",
    "isBase64Encoded": false
}
```

步骤七：查看执行结果

单击 helloworld 事件的“测试”，执行后，在右侧查看执行结果，执行结果如下图。

图 2-7 执行结果



- “函数返回”显示函数的返回结果。
- “日志”部分显示函数执行过程中生成的日志。
- “执行摘要”部分显示“日志”中的关键信息。

说明

此页面最多显示 2K 日志，了解函数更多日志信息，请参考 3.7.2.1 查看函数日志。

步骤八：查看监控指标

在函数详情页面，选择“监控”页签。

- 在“监控”页签，先选择“指标”，再选择时间粒度（5 分钟、15 分钟、1 小时），查看函数运行状态。
- 可以查看的指标有：调用次数、错误次数、运行时间（包括最大运行时间、最小运行时间、平均运行时间）、被拒绝次数。

步骤九：删除函数

1. 在函数详情页面，单击右上角的“操作 > 删除函数”。
2. 在确认框继续单击“确认”，及时释放资源。

2.4.2 开发事件函数示例

概述

使用自定义镜像开发事件函数时，用户需要在镜像中实现一个 http server，并监听 8000 端口接收请求。其中，请求路径/init 默认为函数初始化入口，请根据需要实现该接口。请求路径/invoke 为函数执行入口，触发器事件转到该接口处理。

步骤一：准备环境

本章节所有操作均默认具有操作权限，请确保您登录的用户已有“FunctionGraph Administrator”权限，即 FunctionGraph 服务管理员权限。

步骤二：制作镜像

以在 linux x86 64 位系统上制作镜像为例。

1. 创建一个空文件夹

```
mkdir custom_container_event_example && cd custom_container_evnet_example
```

2. 以 Nodejs 语言为例，实现一个 Http Server，处理函数初始化 init 请求和函数调用 invoke 请求并响应。

   创建一个 main.js 文件，引入 express 框架，实现 Method 为 POST 和 Path 为/invoke 的函数执行入口，实现 Method 为 POST 和 Path 为/init 的函数初始化入口。

```
const express = require('express');

const PORT = 8000;

const app = express();
app.use(express.json());

app.post('/init', (req, res) => {
  console.log('receive', req.body);
  res.send('Hello init\n');
});

app.post('/invoke', (req, res) => {
  console.log('receive', req.body);
  res.send('Hello invoke\n');
});

app.listen(PORT, () => {
  console.log(`Listening on http://localhost:${PORT}`);
});
```

3. 创建一个 package.json 文件，此文件用于向 npm 提供信息，使其能够识别项目以及处理项目的依赖关系。

```
{
  "name": "custom-container-event-example",
  "version": "1.0.0",
  "description": "An example of a custom container event function",
  "main": "main.js",
```

```
  "scripts": {},
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
     "express": "^4.17.1"
  }
}
```

- name：值为项目名。
- version：值为项目版本。
- main：列举文件为库的主入口。
- dependencies：列出 npm 上可用的项目的所有依赖项。

4. 创建 Dockerfile 文件

```
FROM node:12.10.0

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

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

COPY --chown=${USER_ID}:${GROUP_ID} main.js ${HOME}
COPY --chown=${USER_ID}:${GROUP_ID} package.json ${HOME}

RUN cd ${HOME} && npm install

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

RUN find ${HOME} -type d | xargs chmod 500 && \
find ${HOME} -type f | xargs chmod 500

USER ${USER_NAME}
WORKDIR ${HOME}

EXPOSE 8000
ENTRYPOINT ["node", "main.js"]
```

- FROM：指定基础镜像为 node:12.10.0，基础镜像必须设置，值可修改。
- ENV：设置环境变量，设置 HOME 环境变量为/home/custom_container，设置 GROUP_NAME 和 USER_NAME 为 custom_container，USER_ID 和 GROUP_ID 为 1003，这些环境变量必须设置，值可修改。
- RUN：格式为 RUN <命令>，例如 RUN mkdir -m 550 ${HOME}表示构建容器时创建${USER_NAME}用户的 home 目录。
- USER：切换${USER_NAME}用户。
- WORKDIR：切换工作目录到${USER_NAME}用户的 home 目录下。

- COPY：将 main.js 和 package.json 拷贝到容器的${USER_NAME}用户的 home 目录下。
- EXPOSE：暴露容器的 8000 端口，请勿修改。
- ENTRYPOINT：使用 node /home/tester/main.js 命令启动容器。

说明

1. 可以使用任意基础镜像。
2. 在云上环境会默认使用 uid 1003，gid 1003 启动容器。uid、gid 可以在函数页面的设置 > 常规设置 >容器镜像覆盖板块中修改，但不可以是 root 或其他保留 id。

5. 构建镜像

   指定镜像的名称为 custom_container_event_example，版本为 latest，“.”指定 Dockerfile 所在目录，镜像构建命令将该路径下所有的内容打包给容器引擎帮助构建镜像。

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

步骤三：本地验证

1. 启动 docker 容器

```
docker run -u 1003:1003 -p 8000:8000 custom_container_event_example:latest
```

2. 打开一个新的命令行窗口，向开放的 8000 端口发送消息，访问模板代码中指定的 /init 路径

```
curl -XPOST -H 'Content-Type: application/json' localhost:8000/init
```

   按照模块代码中返回

```
Hello init
```

3. 打开一个新的命令行窗口，向开放的 8000 端口发送消息，访问模板代码中指定的 /invoke 路径

```
curl -XPOST -H 'Content-Type: application/json' -d '{"message":"HelloWorld"}'
localhost:8000/invoke
```

   按照模块代码中返回

```
Hello invoke
```

4. 在容器启动端口可以看到

```
Listening on http://localhost:8000
receive {}
receive { message: 'HelloWorld' }
```

```
[root@ecs-74d7 ~]# docker run -u 1003:1003 -p 8000:8000 custom_container_event_example:latest
Listening on http://localhost:8000
receive {}
receive { message: 'HelloWorld' }
```

   或者使用 docker logs 命令获取容器的日志

```
[root@ecs-74d7 custom_container_event_example]# docker logs 5560e1ec09d3
Listening on http://localhost:8000
receive {}
receive { message: 'HelloWorld' }
[root@ecs-74d7 custom_container_event_example]#
```

步骤四：上传镜像

1. 登录容器镜像服务控制台，在左侧导航栏选择“我的镜像”。
2. 单击右上角的“客户端上传”或“页面上传”。
3. 根据指示上传镜像。

4. 上传成功后，在“我的镜像”界面可查看。

步骤五：创建函数

1. 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。
2. 单击右上方的“创建函数”，进入“创建函数”页面，使用容器镜像部署函数。
3. 填写基本信息。
   - 函数类型：选择“事件函数”
   - 函数名称：输入“custom_container_event”
   - 容器镜像：选择上一步上传到 SWR 的镜像。
   - 现有委托：使用包含 SWR Admin 权限的委托。
4. 完成后单击“创建函数”。
5. 在函数详情页“设置 > 高级设置”，开启“初始化函数”，即调用 init 接口进行初始化。

步骤六：测试函数

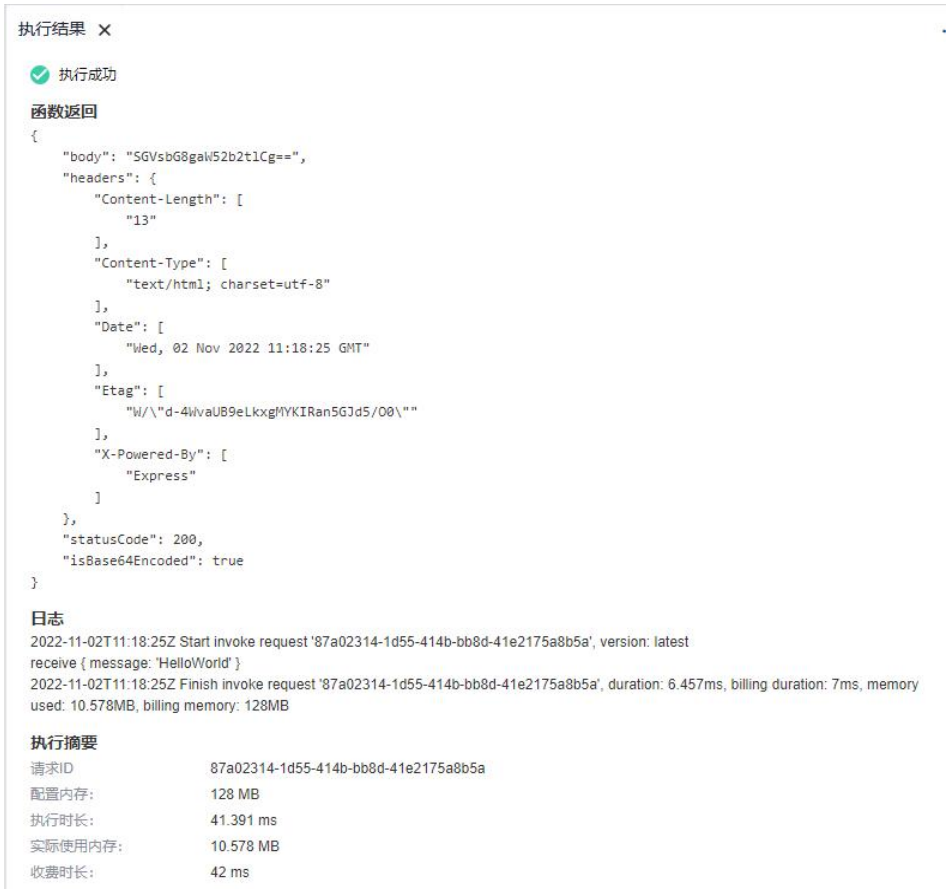
1. 在函数详情页，单击“测试”，在弹窗中创建新的测试事件。
2. 选择“空白模板”，事件名称输入“helloworld”，测试事件修改为如下所示，完成后单击“创建”。

```
{
    "message": "HelloWorld"
}
```

步骤七：查看执行结果

单击 helloworld 事件的“测试”，执行后，在右侧查看执行结果，执行结果如下图。

图 2-8 执行结果



- “函数返回”显示函数的返回结果。
- “日志”部分显示函数执行过程中生成的日志。
- “执行摘要”部分显示“日志”中的关键信息。

说明

此页面最多显示 2K 日志，了解函数更多日志信息，请参考 3.7.2.1 查看函数日志。

步骤八：查看监控指标

在函数详情页面，选择“监控”页签。

- 在“监控”页签，先选择“指标”，再选择时间粒度（5 分钟、15 分钟、1 小时），查看函数运行状态。
- 可以查看的指标有：调用次数、错误次数、运行时间（包括最大运行时间、最小运行时间、平均运行时间）、被拒绝次数。

步骤九：删除函数

1. 在函数详情页面，单击右上角的“操作 > 删除函数”。
2. 在确认框继续单击“确认”，及时释放资源。

3 用户指南

3.1 使用前必读

3.1.1 FunctionGraph 使用流程

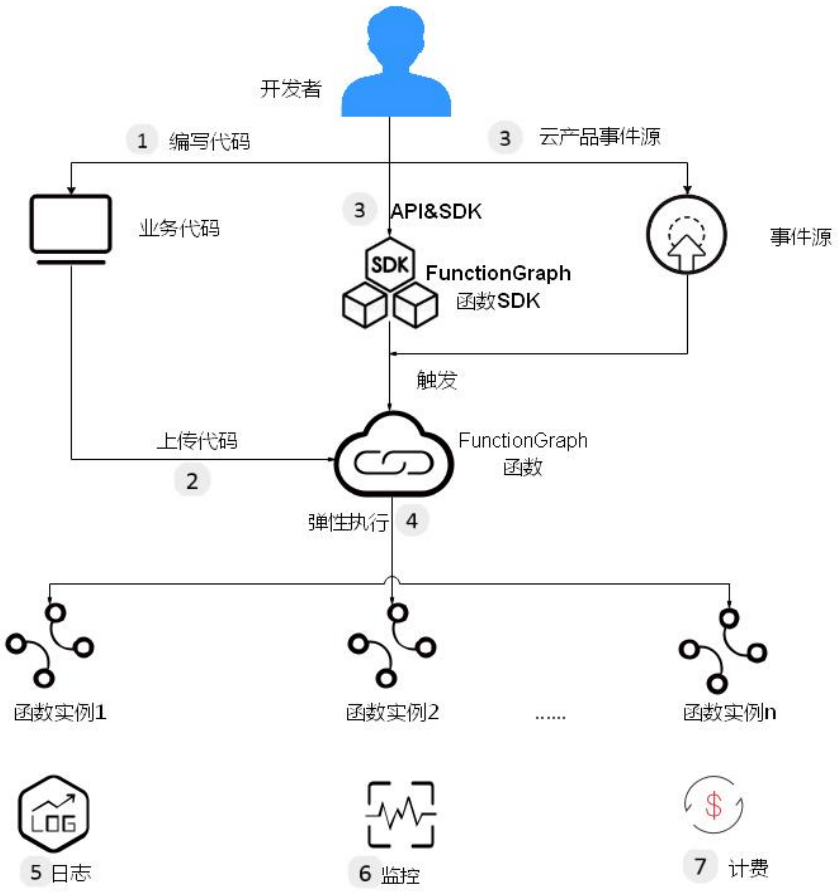
函数工作流 FunctionGraph 是一项基于事件驱动的函数托管计算服务。使用 FunctionGraph 函数，只需编写业务函数代码并设置运行的条件，无需配置和管理服务器等基础设施，函数以弹性、免运维、高可靠的方式运行。

函数使用流程

函数使用流程如图 3-1 所示。

1. 用户编写业务程序代码，打包上传至 FunctionGraph 函数，添加事件源（如 SMN、OBS 和 APIG 等），完成应用程序构建部署。
2. 通过 RESTful API 或者云产品事件源触发函数，生成函数实例，实现业务功能，函数在运行过程中的资源调度由 FunctionGraph 来管理。
3. 用户可以查看函数运行日志和监控信息，按照代码运行情况收费，代码未运行时不产生费用。

图 3-1 函数使用流程



说明如下：

1. 编写代码

   用户编写代码，目前支持 Node.js、Python、Java、Go 等语言。

2. 上传代码

   上传代码，目前支持在线编辑、上传 ZIP 或 JAR 包，从 OBS 引用 ZIP 包等，详情请参见 3.2.1 创建程序包。

3. API 和云产品事件源触发函数执行

   通过 API 和云产品事件源触发函数执行，触发方法请参见 3.5 配置触发器。

4. 弹性执行

   函数在执行过程中，会根据请求量弹性扩容，支持请求峰值的执行，此过程用户无需配置，由 FunctionGraph 完成。

5. 查看日志

   FunctionGraph 函数实现了与云日志服务的对接，您无需配置，即可查看函数运行日志信息，请参见 3.7.2 日志。

6. 查看监控

FunctionGraph 函数实现了与云监控服务的对接，您无需配置，即可查看图形化监控信息，请参见 3.7.1 指标。

总览页面介绍

登录 FunctionGraph 控制台，在左侧导航栏选择“总览”，进入“总览”页面。

- 可以查看函数数量/配额信息、代码存储/存储配额、函数月度调用次数/月度资源用量。

图 3-2 月度统计

- 可以查看租户层面的监控信息（调用次数、错误次数、运行时间、被拒绝次数）。运行监控指标说明如表 3-1 所示。

表 3-1 监控指标说明表

指标	单位	说明
调用次数	次	函数总的调用请求数，包含了错误和被拒绝的调用。异步调用在该请求实际被系统执行时才开始计数。
运行时间	毫秒	最大运行时间为某统计粒度（周期）下，即某一时间段内所有函数单次执行最大的运行时间。 最小运行时间为某统计粒度（周期）下，即某一时间段内所有函数单次执行最小的运行时间。 平均运行时间为某统计粒度（周期）下，即某一时间段内所有函数单次执行平均的运行时间。
错误次数	次	指发生异常请求的函数不能正确执行完并且返回 200，都计入错误次数。函数自身的语法错误或自身执行错误也会计入该指标。
被拒绝次数	次	由于并发请求太多，系统流控而被拒绝的请求次数。

- 可以查看函数流指标：调用次数、运行时间、错误次数、运行中

指标	单位	说明

指标	单位	说明
调用次数	次	函数流总的调用请求数，包含了正确、错误和运行中的调用。异步函数流在请求被系统执行时才开始计数。
运行时间	毫秒	时间段内单次函数流执行平均的运行时间。
错误次数	次	指发生异常请求的函数流不能正确执行完，会计入错误次数。
运行中	个	正在运行中的函数流的数量。

3.1.2 支持的编程语言

3.1.2.1 Node.js 语言

√表示支持，×表示不支持

语言版本	是否支持
Node.js 6.10	√
Node.js 8.10	√
Node.js 10.16	√
Node.js 12.13	√
Node.js 14.18	√

3.1.2.2 Python 语言

√表示支持，×表示不支持

语言版本	是否支持
Python 2.7	√
Python 3.6	√
Python 3.9	√

3.1.2.3 Java 语言

√表示支持，×表示不支持

语言版本	是否支持

语言版本	是否支持
Java 8	√
Java 11	√

3.1.2.4 Go 语言

√表示支持，×表示不支持

语言版本	是否支持
Go 1.x	√

3.1.2.5 定制运行时语言

场景说明

运行时负责运行函数的设置代码、从环境变量读取处理程序名称以及从 FunctionGraph 运行时 API 读取调用事件。运行时会将事件数据传递给函数处理程序，并将来自处理程序的响应返回给 FunctionGraph。

FunctionGraph 支持自定义编程语言运行时。您可以使用可执行文件（名称为 bootstrap）的形式将运行时包含在函数的程序包中，当调用一个 FunctionGraph 函数时，它将运行函数的处理程序方法。

自定义的运行时在 FunctionGraph 执行环境中运行，它可以是 Shell 脚本，也可以是可在 linux 可执行的二进制文件。

说明

在本地开发程序之后打包，必须是 ZIP 包（Java、Node.js、Python、Go）或者 JAR 包（Java），上传至 FunctionGraph 即可运行，无需其它的部署操作。制作 ZIP 包的时候，单函数入口文件必须在根目录，保证解压后，直接出现函数执行入口文件，才能正常运行。

对于 Go runtime，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 testplugin.so，则“函数执行入口”命名为 testplugin.Handler。

运行时文件 bootstrap 说明

如果程序包中存在一个名为 bootstrap 的文件，FunctionGraph 将执行该文件。如果引导文件未找到或不是可执行文件，函数在调用后将返回错误。

运行时代码负责完成一些初始化任务，它将在一个循环中处理调用事件，直到它被终止。

初始化任务将对函数的每个实例运行一次以准备用于处理调用的环境。

运行时接口说明

FunctionGraph 提供了用于自定义运行时的 HTTP API 来接收来自函数的调用事件，并在 FunctionGraph 执行环境中发送回响应数据。

- **获取调用**

  方法 – Get

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

  该接口用来获取下一个事件，响应正文包含事件数据。响应标头包含信息如下。

表 3-2 响应标头信息说明

参数	说明
X-Cff-Request-Id	请求 ID。
X-CFF-Access-Key	租户 AccessKey，使用该特殊变量需要给函数配置委托。
X-CFF-Auth-Token	Token，使用该特殊变量需要给函数配置委托。
X-CFF-Invoke-Type	函数执行类型。
X-CFF-Secret-Key	租户 SecretKey，使用该特殊变量需要给函数配置委托。
X-CFF-Security-Token	Security token，使用该特殊变量需要给函数配置委托。

- **调用响应**

  方法 – POST

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

  该接口将正确的调用响应发送到 FunctionGraph。在运行时调用函数处理程序后，将来自函数的响应发布到调用响应路径。

- **错误上报**

  方法 – POST

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

  $REQUEST_ID 为获取事件的响应 header 中 X-Cff-Request-Id 变量值，说明请参见表 3-2。

  $RUNTIME_API_ADDR 为系统环境变量，说明请参见表 3-3。

  该接口将错误的调用响应发送到 FunctionGraph。在运行时调用函数处理程序后，将来自函数的响应发布到调用响应路径。

运行时环境变量说明

下面是 FunctionGraph 执行环境中运行时相关的环境变量列表，除此之外，还有用户自定义的环境变量，都可以在函数代码中直接使用。

表 3-3 环境变量说明

键	值说明
RUNTIME_PROJECT_ID	projectID
RUNTIME_FUNC_NAME	函数名称
RUNTIME_FUNC_VERSION	函数的版本
RUNTIME_PACKAGE	函数组
RUNTIME_HANDLER	函数执行入口
RUNTIME_TIMEOUT	函数超时时间
RUNTIME_USERDATA	用户通过环境变量传入的值
RUNTIME_CPU	分配的 CPU 数
RUNTIME_MEMORY	分配的内存
RUNTIME_CODE_ROOT	包含函数代码的目录
RUNTIME_API_ADDR	自定义运行时 API 的主机和端口

用户定义的环境变量也同 FunctionGraph 环境变量一样，可通过环境变量获取方式直接获取用户定义环境变量。

示例说明

此示例包含 1 个文件（bootstrap 文件），该文件都在 Bash 中实施。

运行时将从部署程序包加载函数脚本。它使用两个变量来查找脚本。

引导文件 bootstrap 内容如下：

```
#!/bin/sh
set -o pipefail
#Processing requests loop
while true
do
HEADERS="$(mktemp)"
  # Get an event
  EVENT_DATA=$(curl -sS -LD "$HEADERS" -X GET
"http://$RUNTIME_API_ADDR/v1/runtime/invocation/request")
  # Get request id from response header
  REQUEST_ID=$(grep -Fi x-cff-request-id "$HEADERS" | tr -d '[:space:]' | cut -d: -
f2)
  if [ -z "$REQUEST_ID" ]; then
```

```
  continue
 fi
 # Process request data
 RESPONSE="Echoing request: hello world!"
 # Put response
 curl -X POST
"http://$RUNTIME_API_ADDR/v1/runtime/invocation/response/$REQUEST_ID" -d
"$RESPONSE"
done
```

加载脚本后，运行时将在一个循环中处理事件。它使用运行时 API 从 FunctionGraph 检索调用事件，将事件传递到处理程序，并将响应发布回给 FunctionGraph。

为了获取请求 ID，运行时会将来自 API 响应的标头保存到临时文件，并从该文件读取 x-cff-request-id 读取请求头的请求唯一标识。将获取到的事件数据做处理并响应发布返回 FunctionGraph。

go 源码示例，需要通过编译后才可执行。

```
package main

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

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

```
    functionName           = os.Getenv("RUNTIME_FUNC_NAME")
    functionVersion        = os.Getenv("RUNTIME_FUNC_VERSION")

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

func main() {
    // main loop for processing requests.
    for {
        requestId, header, payload, err := getRequest()
        if err != nil {
            time.Sleep(50 * time.Millisecond)
            continue
        }

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

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

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

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

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

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

    payload, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        log.Printf("read request body error, err: %s.", err.Error())
        return "", nil, nil, err
    }

    if resp.StatusCode != 200 {
        log.Printf("get request failed, status: %d, message: %s.",
resp.StatusCode, string(payload))
        return "", nil, nil, noRequestAvailableError
    }

    log.Printf("get request ok.")
    return requestId, resp.Header, payload, nil
}

func putResponse(requestId string, payload []byte, err error) error {
    var body io.Reader
    if payload != nil && len(payload) > 0 {
        body = bytes.NewBuffer(payload)
    }

    url := ""
    if err == nil {
        url = strings.Replace(putResponseUrl, "{REQUEST_ID}", requestId, -
```

```
1)
    } else {
        url = strings.Replace(putErrorResponseUrl, "{REQUEST_ID}",
requestId, -1)
    }

    resp, err := client.Post(strings.Replace(url, "{REQUEST_ID}",
requestId, -1), "", body)
    if err != nil {
        log.Printf("put response error, err: %s.", err.Error())
        return err
    }
    defer resp.Body.Close()

    responsePayload, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        log.Printf("read request body error, err: %s.", err.Error())
        return err
    }

    if resp.StatusCode != 200 {
        log.Printf("put response failed, status: %d, message: %s.",
resp.StatusCode, string(responsePayload))
        return putResponseFailedError
    }

    return nil
}

type FunctionEvent struct {
    Type string `json:"type"`
    Name string `json:"name"`
}

type APIGFormatResult struct {
    StatusCode     int             `json:"statusCode"`
    IsBase64Encoded bool            `json:"isBase64Encoded"`
    Headers        map[string]string `json:"headers,omitempty"`
    Body           string          `json:"body,omitempty"`
}

func (result *APIGFormatResult) toJsonBytes() []byte {
    data, err := json.MarshalIndent(result, "", "  ")
    if err != nil {
```

```
        return nil
    }

    return data
}

type ErrorMessage struct {
    ErrorType    string `json:"errorType"`
    ErrorMessage  string `json:"errorMessage"`
}

func (errMsg *ErrorMessage) toJsonBytes() []byte {
    data, err := json.MarshalIndent(errMsg, "", "  ")
    if err != nil {
        return nil
    }

    return data
}
```

代码中的环境变量说明如下，请参见表 3-4。

表 3-4 环境变量说明

环境变量	说明
RUNTIME_FUNC_NAME	函数名称
RUNTIME_FUNC_VERSION	函数版本
RUNTIME_PACKAGE	函数组

3.2 构建函数

3.2.1 创建程序包

要创建 FunctionGraph 函数，首先需要创建函数部署程序包（包含代码和所有依赖项的文件）。用户可以自行创建部署程序包或直接在 FunctionGraph 函数控制台在线编辑代码，控制台将创建并上传部署程序包，从而实现 FunctionGraph 函数的创建。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。如果用户代码是上传 zip 包的方式，则前端进行相应解压展示，提供编辑能力。

说明

- 用户在本地开发程序之后打包，必须是 ZIP 包（Java、Node.js、Python、Go）或者 JAR 包（Java），上传至 FunctionGraph 即可运行，无需其它的部署操作。

- 制作 ZIP 包的时候，单函数入口文件必须在根目录，保证解压后，直接出现函数执行入口文件，才能正常运行。
- 对于 Go runtime，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 testplugin.so，则“函数执行入口”命名为 testplugin.Handler。
- 对于 Java runtime，由于 Java 是编译型语言，所以不能在线编辑代码。如果函数没有引入其他第三方件，可以选择上传函数 jar 包。如果函数中引入其他三方件，则需要制作包含所有依赖三方件和函数 jar 的 zip 包，选择上传 zip 文件。

FunctionGraph 函数支持的上传程序包的方式如表 3-5。

表 3-5 代码上传方式说明

运行时	在线编辑	上传 ZIP 文件	上传 JAR 包	从 OBS 上传文件
Node.js	支持	支持	不支持	支持
Python	支持	支持	不支持	支持
Java	不支持	支持	支持	支持
GO	不支持	支持	不支持	支持
定制运行时	支持	支持	不支持	支持

须知

上传代码时，如果代码中包含敏感信息（如帐户密码等），请您自行加密，以防止信息泄露。

表 3-6 函数代码上传方式表

代码上传方式	操作
在线编辑	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。如果用户代码是上传 zip 包的方式，则前端进行相应解压展示，并支持用户在函数详情页的“代码”页签，进行在线编辑修改。 • 文件：支持创建文件和文件夹功能。其中包括新建文件，新建文件夹、保存、关闭所有文件功能。 • 编辑：支持在编码框中，对代码进行撤销、恢复、剪切、复制、粘贴、查找、替换操作。 • 设置：支持设置编码框中代码字体大小、自动格式化和编码框主题颜色。
上传 ZIP 文件	1. 在函数详情页的“代码”页签，选择“上传自 > Zip 文件”。 2. 单击“添加文件”上传本地代码至平台。上传的 zip 文件大小限制为 50M，如超过 50M，请通过 OBS 上传。

代码上传方式	操作
从 OBS 上传文件	1. 在函数详情页的“代码”页签，选择“上传自 > OBS 地址”。 2. 单击“添加文件”上传本地代码至平台。

Node.js 程序包

在线编辑

FunctionGraph 服务预装了适用于 Node.js 的开发工具包，如果自定义代码只需要软件开发工具包库，则可以使用 FunctionGraph 控制台的内联编辑器。使用控制台可以编辑代码并将代码上传到 FunctionGraph，控制台会将代码及相关的配置信息压缩到 FunctionGraph 服务能够运行的部署程序包中。

上传程序包

如果编写的代码需要用到其他资源（如使用图形库进行图像处理），则需要先创建 FunctionGraph 函数部署程序包，然后再使用控制台上传部署程序包。Node.js 编程语言支持以下两种方式上传程序包。

须知

- 制作 zip 包的时候，单函数入口文件必须在根目录，保证解压后，直接出现函数执行入口文件，才能正常运行。
- 解压后的源代码不能超过 1.5G，超大代码请联系专员。

- 直接上传程序包

  在创建部署程序包后，可直接从本地上传 ZIP 程序包，ZIP 程序包大小限制为 50MB，如果超过该限制，请使用 OBS 存储桶。
- 上传至 OBS 存储桶

  在创建部署程序包后，可先将.zip 文件上传到要在其中创建 FunctionGraph 函数的区域中的 OBS 存储桶中，然后指定 FunctionGraph 函数中设置程序包的 OBS 存储地址，OBS 中 ZIP 包大小限制为 300MB。

Python 程序包

在线编辑

FunctionGraph 服务预装了适用于 Python 的开发工具包，如果自定义代码只需要软件开发工具包库，则可以使用 FunctionGraph 控制台的内联编辑器。使用控制台可以编辑代码并将代码上传到 FunctionGraph，控制台会将代码及相关的配置信息压缩到 FunctionGraph 服务能够运行的部署程序包中。

使用 Python 语言在线编辑代码，需要输出中文时，请在编辑器中增加如下代码：

```
# -*- coding:utf-8 -*-
import json
def handler (event, context):
```

```
output = 'Hello message: ' + json.dumps(event,ensure_ascii=False)
return output
```

上传程序包

如果编写的代码需要用到其他资源（如使用图形库进行图像处理），则需要先创建 FunctionGraph 函数部署程序包，然后再使用控制台上传部署程序包。Python 编程语言支持以下两种方式上传程序包。

须知

- 制作 zip 包的时候，单函数入口文件必须在根目录，保证解压后，直接出现函数执行入口文件，才能正常运行。
- 解压后的源代码不能超过 1.5G，超大代码请联系专员。
- 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。

- 直接上传程序包

  在创建部署程序包后，可直接从本地上传 ZIP 程序包，ZIP 程序包大小限制为 50MB，如果超过该限制，请使用 OBS 存储桶。
- 上传至 OBS 存储桶

  在创建部署程序包后，可先将.zip 文件上传到要在其中创建 FunctionGraph 函数的区域中的 OBS 存储桶中，然后指定 FunctionGraph 函数中设置程序包的 OBS 存储地址，OBS 中 ZIP 包大小限制为 300MB。

Java 程序包

由于 Java 是编译型语言，所以不能在线编辑代码，只能上传程序包，部署程序包可以是.zip 文件或独立的 jar 文件。

上传 Jar 包

- 如果函数没有引入其他依赖包，可以直接上传函数 jar 包。
- 如果函数引入了其他依赖包，可以先将依赖包上传至 OBS 桶，创建函数时设置依赖包，并上传函数 jar 包。

上传 zip

如果函数中引入其他三方件，也可以制作包含所有依赖三方件和函数 jar 的 zip 包，选择上传 zip 文件。

Java 编程语言支持以下两种方式上传程序包。

须知

- 制作 zip 包的时候，单函数入口文件必须在根目录，保证解压后，直接出现函数执行入口文件，才能正常运行。
- 解压后的源代码不能超过 1.5G，超大代码请联系专员。

- 直接上传程序包

  在创建部署程序包后，可直接从本地上传 ZIP 程序包，ZIP 程序包大小限制为 50MB，如果超过该限制，请使用 OBS 存储桶。

- 上传至 OBS 存储桶

  在创建部署程序包后，可先将.zip 文件上传到要在其中创建 FunctionGraph 函数的区域中的 OBS 存储桶中，然后指定 FunctionGraph 函数中设置程序包的 OBS 存储地址，OBS 中 ZIP 包大小限制为 300MB。

GO 语言程序包

上传程序包

只能上传程序包，部署程序包必须是.zip 文件。Go 编程语言支持以下两种方式上传程序包。

须知

- 制作 zip 包的时候，单函数入口文件必须在根目录，保证解压后，直接出现函数执行入口文件，才能正常运行。
- 解压后的源代码不能超过 1.5G，超大代码请联系专员。

- 直接上传程序包

  在创建部署程序包后，可直接从本地上传 ZIP 程序包，ZIP 程序包大小限制为 50MB，如果超过该限制，请使用 OBS 存储桶。

- 上传至 OBS 存储桶

  在创建部署程序包后，可先将.zip 文件上传到要在其中创建 FunctionGraph 函数的区域中的 OBS 存储桶中，然后指定 FunctionGraph 函数中设置程序包的 OBS 存储地址，OBS 中 ZIP 包大小限制为 300MB。

定制运行时程序包

在线编辑

使用控制台可以编辑代码并将代码上传到 FunctionGraph，控制台会将代码及相关的配置信息压缩到 FunctionGraph 服务能够运行的部署程序包中。

上传程序包

如果编写的代码需要用到其他资源（如使用图形库进行图像处理），则需要先创建 FunctionGraph 函数部署程序包，然后再使用控制台上传部署程序包。定制运行时支持以下两种方式上传程序包。

须知

- 制作 zip 包的时候，单函数入口文件必须在根目录，保证解压后，直接出现函数执行入口文件，才能正常运行。
- 解压后的源代码不能超过 1.5G，超大代码请联系专员。

- 直接上传程序包

  在创建部署程序包后，可直接从本地上传 ZIP 程序包，ZIP 程序包大小限制为 50MB，如果超过该限制，请使用 OBS 存储桶。

- 上传至 OBS 存储桶

  在创建部署程序包后，可先将.zip 文件上传到要在其中创建 FunctionGraph 函数的区域中的 OBS 存储桶中，然后指定 FunctionGraph 函数中设置程序包的 OBS 存储地址，OBS 中 ZIP 包大小限制为 300MB。

3.2.2 使用空白模板创建函数

3.2.2.1 创建事件函数

概述

函数是处理事件的自定义代码，您可以使用空白模板函数创建函数，根据实际业务场景进行函数配置。

由于 FunctionGraph 承担计算资源的管理工作，在函数完成编码以后，需要为函数设置运算资源等信息，目前主要是在 FunctionGraph 函数控制台完成。

创建函数时可以使用空模板，也可以 3.2.3 使用示例模板创建函数、3.2.4 使用容器镜像部署函数。

说明

使用空模板创建函数时，需要设置基础配置信息和代码信息，如表 3-7 所示，带*参数为必填项。

每个 FunctionGraph 函数都运行在其自己的环境中，有其自己的资源和文件系统。

前提条件

1. 了解函数支持的编程语言，具体请参见 3.1.2 支持的编程语言。
2. 创建程序包，具体请参见 3.2.1 创建程序包。
3. 创建委托（可选，根据实际情况，确定是否需创建委托），具体请参见 3.3.3 配置委托权限。

操作步骤

1. 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。
2. 单击右上方的“创建函数”，进入“创建函数”页面。
3. 选择“创建空白函数”，参见表 3-7 填写函数信息，带*参数为必填项。

表 3-7 函数基础配置信息表

参数	说明
*函数类型	• 事件函数：通过触发器来触发函数执行。 • HTTP 函数：用户可以直接发送 HTTP 请求到 URL 触发函数执行。 说明 • HTTP 函数当前不区分编程语言，函数执行入口必须在 bootstrap 文件中设置，用户直接写启动命令，端口统一开放成 8000。 • HTTP 函数只允许创建 APIG/APIC 的触发器类型，其他触发器不支持。 • HTTP 函数的使用说明请参见 3.2.2.2 创建 HTTP 函数。
*区域	选择要部署代码的区域。
*函数名称	函数名称，命名规则如下： • 可包含字母、数字、下划线和中划线，长度不超过 60 个字符。 • 以大/小写字母开头，以字母或数字结尾。
委托名称	用户委托函数工作流服务去访问其他的云服务，则需要提供权限委托，创建委托，请参见 3.3.3 配置委托权限。 如果用户函数不访问任何云服务，则不用提供委托名称。
*企业项目	选择已创建的企业项目，将函数添加至企业项目中，默认选择“default”。
运行时	选择用来编写函数的语言。 须知 控制台代码编辑器仅支持 Node.js、Python。

4. 填写完成后单击“创建函数”,页面跳转至代码配置页面，继续配置代码源。

配置代码源

1. 您可以根据所选的运行时语言 Runtime，参见 3.2.1 创建程序包，选择适合的方式进行代码源部署，完成后单击“部署”。

   以下图为例，运行时语言为“Node.js 10.16”，可以选择“在线编辑”、“Zip 文件”、“OBS 地址”三种方式进行代码源部署。

2. 代码若有修改，请修改完成后再次单击“部署”，重新部署代码。

查看代码信息

1. 查看代码属性

   代码属性展示最新部署代码的大小及上次修改时间。

图 3-3 查看代码属性

2. 查看基本信息

函数创建完成后，各语言默认内存和执行超时时间如图 3-4 所示，请根据实际业务评估，若需修改“函数执行入口”、“内存（MB）”“执行超时时间（秒）”，可单击“编辑”，在常规设置中修改配置信息，具体请参见 3.3.2 配置常规信息。

图 3-4 编辑基本信息

须知

函数一旦创建，便不能修改运行时语言。

3.2.2.2 创建 HTTP 函数

概述

HTTP 函数专注于优化 Web 服务场景，用户可以直接发送 HTTP 请求到 URL 触发函数执行，从而使用自己的 Web 服务。

说明

- HTTP 函数当前不区分编程语言，函数执行入口必须在 bootstrap 文件中设置，用户直接写启动命令，端口统一开放成 8000，绑定 IP 为 127.0.0.1。
- bootstrap 文件是 HTTP 函数的启动文件，HTTP 函数仅支持读取 bootstrap 作为启动文件名称，其它名称将无法正常启动服务。
- HTTP 函数支持多种开发语言。
- 用户函数需要返回一个合法的 http 响应报文。
- 该章节均以 java 为样例，若需要使用其他语言，则更换语言路径即可，代码包路径无需更换。其他各语言路径请参见表 3-8。

前提条件

1. 准备一个 java 的 jar 包。
2. 准备一个 bootstrap 启动文件，作为 HTTP 函数的启动文件。

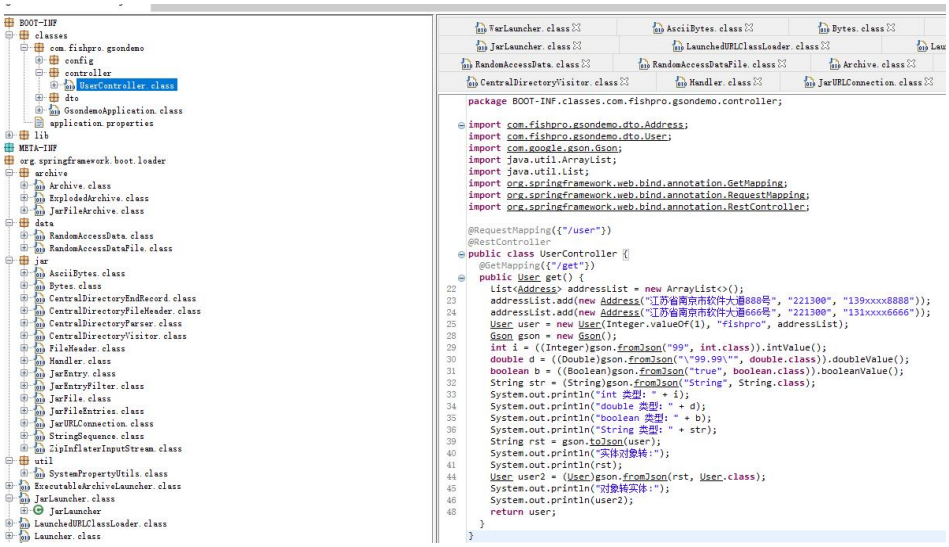
示例：

bootstrap 文件内容如下

```
/opt/function/runtime/java8/rtsp/jre/bin/java -jar -Dfile.encoding=utf-8
/opt/function/code/gsondemo-0.0.1-SNAPSHOT.jar
```

- /opt/function/runtime/java8/rtsp/jre/bin/java：表示 java 所在路径。

- -Dfile.encoding=utf-8：JVM 参数，添加此参数，可以避免中文乱码。
- /opt/function/code：表示函数代码包所在路径
- gsondemo-0.0.1-SNAPSHOT.jar：示例 jar 包，提供的服务路径为“/user/get”。



若需要使用其他语言，则参见表 3-8 更换语言路径，代码包路径无需更换。

表 3-8 多语言路径说明

语言	路径
Java8	/opt/function/runtime/java8/rtsp/jre/bin/java
Java11	/opt/function/runtime/java11/rtsp/jre/bin/java
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
Python2.7	/opt/function/runtime/python2.7/rtsp/python/bin/python
Python3.6	/opt/function/runtime/python3.6/rtsp/python/bin/python3
Python3.9	/opt/function/runtime/python3.9/rtsp/python/bin/python3

操作步骤

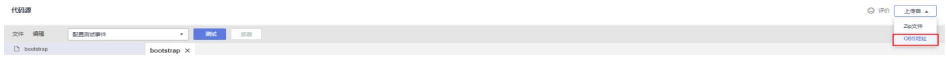
1. 创建函数
   a. 创建 HTTP 函数，详细配置信息请参见 3.2.2.1 创建事件函数，如下参数需注意。
      - 函数类型：HTTP 函数

- 区域：选择要部署代码的区域

b. 上传代码，此处以“从 OBS 地址”上传为例，完成后单击“部署”。

将提前准备好的 jar 包和 bootstrap 文件打包成 zip 包，代码上传方式选择“OBS 地址”。

图 3-5 上传自 OBS 地址



2. 创建触发器

说明

HTTP 函数只允许创建 APIG 的触发器类型，其他触发器不支持。

a. 进入函数详情页面，选择“设置 > 触发器”页签，单击“创建触发器”。

b. 配置触发器信息，此处以创建“API 网关服务（APIG 专享版）”触发器为例，其他配置信息请参见 3.5.3 使用 APIG（专享版）触发器。

说明

示例中“安全认证”暂时选择“None”，用户在配置时应根据实际情况选择。

- App：采用 Appkey&Appsecret 认证，安全级别高，推荐使用。
- IAM：IAM 认证，只允许 IAM 用户能访问，安全级别中等。
- None：无认证模式，所有用户均可访问。

c. 配置完成后，单击“确定”。API 触发器创建完成后，会在 API 网关生成 API“API_test_http”。

3. 发布 API

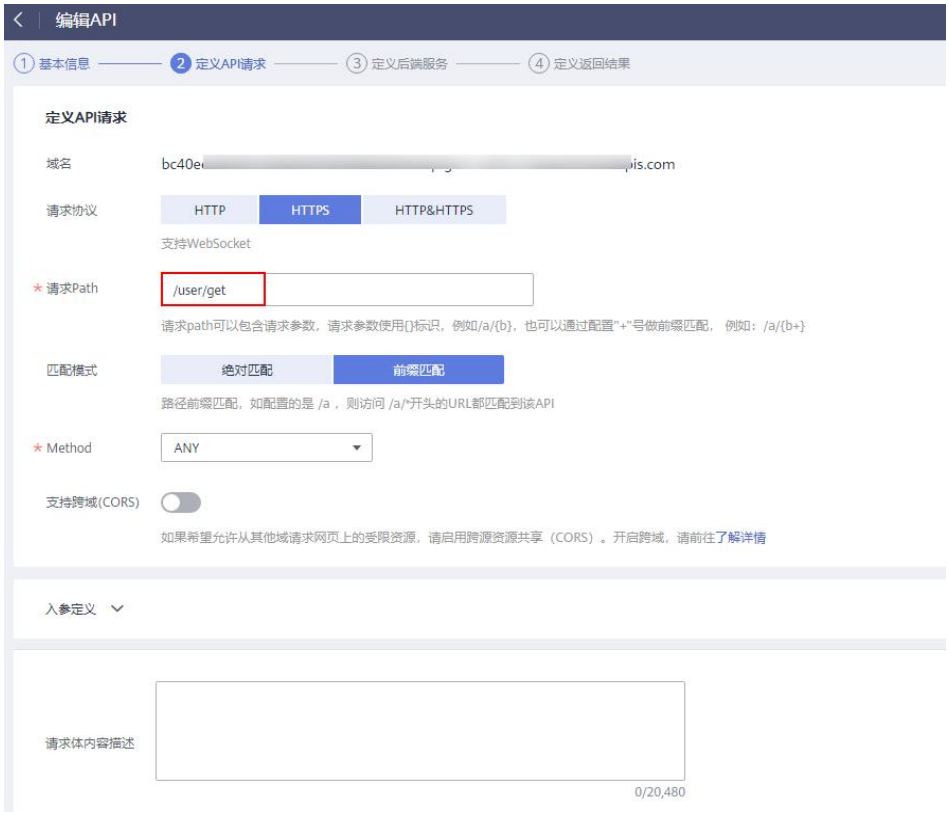
a. 单击“触发器”页签下的 API 名称，跳转至 API 的总览页面。

b. 单击右上方的“编辑”，进入“基本信息”页面。

图 3-6 编辑 API

c. 单击“下一步”，进入“定义 api 请求”页面，修改“请求 Path”为“/user/get”并单击“立即完成”。

图 3-7 定义 API 请求



d. 单击“发布 API”，在发布页面继续单击“发布”。

4. 触发函数

a. 返回函数工作流控制台，在左侧导航栏选择“函数 > 函数列表”，单击创建的 HTTP 函数进入函数详情页。

b. 选择“设置 > 触发器”，复制“调用 URL”，在浏览器访问。

图 3-8 复制 URL

c. 查看请求结果。

图 3-9 查看请求结果

函数公共请求头

HTTP 函数请求头默认携带如下字段。

表 3-9 默认请求头

字段	描述
X-CFF-Request-Id	当前请求 ID
X-CFF-Memory	分配的内存
X-CFF-Timeout	函数超时时间
X-CFF-Func-Version	函数版本
X-CFF-Func-Name	函数名称
X-CFF-Project-Id	ProjectID
X-CFF-Package	函数组
X-CFF-Region	当前 region

3.2.3 使用示例模板创建函数

概述

FunctionGraph 平台提供了函数模板，在创建函数时选择模板，实现模板代码、运行环境自动填充，快速构建应用程序。

创建函数

1. 登录函数工作流控制台，在左侧的导航栏选择“函数模板”。
2. 在“函数模板”界面，“云服务”选择“函数工作流”，模板选择 Python 2.7 的“context 使用指导”，单击“使用模板”。

说明

此处以 Python 2.7 的“context 使用指导”举例，请您根据实际需求选择模板。

3. 选择函数模板后，会加载模板内置的代码、配置信息，进入到“创建函数”界面。

4. 输入函数名称“context”，选择已创建的委托，其他设置保持不变，单击“创建函数”，进入配置详情页。

说明

若不配置委托，在触发函数时，执行结果会返回

```
Failed to access other services because no temporary AK, SK, or token has been obtained. Please set an agency.
```

5. 完成后单击“保存”。

触发函数

1. 在“context”函数的“代码”页签，单击“测试”。
2. 在弹出的“配置测试事件”对话框中，选择“空白模板”，再单击“创建”。
3. 继续单击“测试”，等待测试完成，查看测试结果。

图 3-10 执行成功结果

3.2.4 使用容器镜像部署函数

概述

用户直接打包上传容器镜像，由平台加载并启动运行。与原本上传代码方式相比，用户可以使用自定义的代码包，不仅灵活也简化了用户的迁移成本。您可以选择“事件函数”类型创建自定义镜像函数，也可以选择“HTTP 函数”类型创建自定义镜像函数。

支持的功能：

- **下载用户镜像**

  用户镜像储存在自己的 SWR 服务中，需要 SWR Admin 权限才能下载，FunctionGraph 会在创建 pod 前使用 swr api 生成并设置好临时登录指令。

- **环境变量**

  设置 FunctionGraph 函数的加密配置和环境变量，具体请参见 3.3.6 配置环境变量。

- **挂载外部数据盘**

  支持挂载外部数据盘，具体请参见 3.3.5 配置磁盘挂载。

- **预留实例**

  支持预留实例，具体请参见预留实例。

说明

用户容器会使用属主 1003、属组 1003 启动，与其他类型的函数相同。

前提条件

请参见 3.3.3 配置委托权限，创建一个包含“SWR Admin 容器镜像服务（SWR）管理员”权限的委托，因为用户镜像储存在 SWR 服务中，只有拥有“SWR Admin”权限，才能调用与获取，拉取镜像。

操作步骤

1. 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。
2. 单击右上方的“创建函数”，进入“创建函数”页面。
3. 选择“容器镜像”，参见表 3-10。

图 3-11 创建容器镜像


基本信息
* 函数类型
事件函数
HTTP函数
* 区域
不同区域的资源之间内网不互通。请就近选择靠近您业务的区域，可以降低网络时延、提高访问速度。
* 函数名称
请输入函数名称
可包含字母、数字、下划线和中划线，以大小写字母开头，以字母或数字结尾，长度不超过60个字符。
* 企业项目
default
查看企业项目
* 容器镜像
请输入镜像URL
查看镜像
容器镜像覆盖
CMD
请输入CMD
Args
请输入Args
Working Dir
请输入Working Dir
用户ID
请输入用户ID
用户组ID
请输入用户组ID
权限
不使用任何委托
使用现有委托
* 现有委托
all
创建委托
用户镜像储存在SWR服务中，所选现有委托需要包含SWR Admin权限。

表 3-10 配置信息

参数	说明
*函数类型	选择函数类型。 • 事件函数：通过触发器来触发函数执行。 • HTTP 函数：用户可以直接发送 HTTP 请求到 URL 触发函数执行。 说明 • 自定义容器镜像需包含 HTTP Server，监听端口为 8000。 • HTTP 函数只允许创建 APIG/APIC 的触发器类型，其他触发器不支持。 • 事件函数需创建一个 HTTP Server 并实现 Method 为 POST 和 Path 为/invoke 的函数执行入口，可实现 Method 为 POST 和 Path 为/init 的函数初始化入口。
*区域	选择要部署代码的区域。
*函数名称	函数名称，命名规则如下： • 可包含字母、数字、下划线和中划线，长度不超过 60 个字符。 • 以大/小写字母开头，以字母或数字结尾。
*企业项目	选择已创建的企业项目，将函数添加至企业项目中，默认选择“default”。
容器镜像	输入镜像 URL，即用于函数的容器镜像的位置。您可以单击“查看镜像”，查看自有镜像及共享镜像。
容器镜像覆盖	• CMD：容器的启动命令，例如"/bin/sh"。该参数为可选参数，不填写，则默认使用镜像中的 Entrypoint/CMD。字符串数组，以逗号分开。 • Args：容器的启动参数，例如"-args,value1"。该参数为可选参数，不填写，则默认使用镜像中的 CMD。字符串数组，以逗号分开。 • Working Dir：容器的工作目录。该参数为可选参数，不填写，则默认使用镜像中的配置。文件夹路径，以/开头。 • 用户 ID：镜像运行时的用户 ID，若不填写，默认为 1003。 • 用户组 ID：镜像运行时的用户组 ID，若不填写，默认为 1003。
现有委托	选择包含 SWR Admin 权限的委托。

说明

- Command、Args、Working dir 三个参数之和不能超过 5120。
- 初次执行时需要从 SWR 中拉取镜像，且冷启动时需要启动容器，所以自定义镜像冷启动比较慢。后续每次冷启动，如果节点上没有镜像，都需要从 SWR 中拉取。
- 镜像需要选择为“公开”。
- 自定义容器镜像开放端口限定为 8000。
- 可支持的镜像包最大为 2G，当镜像包过大时可以采取一些方式缩容，比如在线题库场景中，可以把原本加载在容器中的题库数据通过外部文件系统挂载盘方式挂载到容器中。

- FunctionGraph 通过 LTS 日志采集容器输出到控制台的所有日志，可以通过标准输出或者开源日志框架重定向到控制台的方式打印业务信息。打印的内容建议包括系统时间、组件名称、代码行、关键数据等来方便定位。
- oom 错误时，内存占用大小可以在函数执行结果中查看。
- 用户函数需要返回一个合法的 http 响应报文。

3.3 配置函数

3.3.1 配置初始化

概述

初始化函数在函数实例启动成功后执行，执行成功后，实例才能开始调用请求处理函数处理请求。FunctionGraph 保证一个函数实例在生命周期内，初始化函数成功执行且只能成功执行一次。

应用场景

多个请求处理可以共享的业务逻辑适合放到初始化函数，以降低函数时延，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。

前提条件

已创建函数。

初始化函数

步骤 1 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。

步骤 2 选择待配置的函数，单击进入函数详情页。

步骤 3 选择“设置 > 高级设置”，开始配置。

图 3-12 开启初始化配置

表 3-11 初始化配置参数说明

参数	说明
配置初始化函数	如需初始化，请开启此参数。
初始化超时时间（秒）	函数初始化的超时时间，如开启函数初始化功能则设置，不开启则不设置。 函数初始化超时时间设置范围为 1-300 秒。
函数初始化入口	在函数配置页面中，可以选择开启函数初始化功能。各 runtime 的函数初始化入口命名规范与原有函数执行入口保持一致。如 Node.js 和 Python 函数，命名规则：[文件名].[初始化函数名]。 说明 如不开启函数初始化功能则无需配置函数初始化入口。

说明

- 开启函数初始化功能后，各 runtime 的函数初始化入口命名规范与原有函数执行入口保持一致。如 Node.js 和 Python 函数，命名规则：[文件名].[初始化函数名]。
- 函数代码配置信息请参见 3.2.1 创建程序包。

----结束

3.3.2 配置常规信息

概述

在函数创建完成后，“内存”、“函数执行入口”、“执行超时时间”会根据您所选的“运行时语言”默认填写。如果您需要贴合实际业务场景修改配置，请参见本章节进行配置。

前提条件

已创建函数。

操作步骤

1. 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。
2. 选择待配置的函数，单击进入函数详情页。
3. 选择“设置 > 常规设置”，参见表 3-12 填写函数信息，带*参数为必填项。

表 3-12 基本信息配置说明

参数	说明
所属应用	当前创建的新函数所属应用均为“default”应用，且无法更改。

参数	说明
	须知 “应用”实际作用就是文件夹功能。新版本里会逐步弱化并下线老界面的“应用"概念，未来会通过标签分组的方式来管理函数的分类等。
*函数执行入口	• Node.js、Python 和 PHP 函数执行入口的命名规则：[文件名].[执行函数名]，必须包含“.”。 例如：myfunction.handler。 • Java 函数执行入口的命名规则：[包名].[类名].[执行函数名]。 例如：com.xxxxx.exp.Myfunction.myHandler。 • Go 函数执行入口的命名规则：与用户上传的代码包中的可执行文件名保持一致。 例如：用户编译的可执行文件名为 handler，则填 handler。
*企业项目	选择已创建的企业项目，将函数添加至企业项目中，默认选择“default”。
*执行超时时间（秒）	在“配置”页签，函数运行的超时时间，超时的函数将被强行停止。如果执行时间超过 90 秒，请采用异步调用的方式。 函数超时时间设置范围为 3~900 秒。
内存（MB）	函数实例内存规格，取值范围：128、256、512、768、1024、1280、1536、1792、2048、2560、3072、3584、4096。
描述	填写对函数的描述，不超过 512 个字符。

4. 填写完成后单击“保存”。

3.3.3 配置委托权限

概述

大部分场景下，FunctionGraph 都需要与其他云服务协同工作，通过创建云服务委托，让 FunctionGraph 以您的身份使用其他云服务，代替您进行一些资源运维工作。

应用场景

若您在 FunctionGraph 服务中使用如下场景，请先 3.3.3 配置委托权限，对应授权项参见表 3-13 进行选择。

表 3-13 常见授权项选择

场景	授权项	说明
使用自定义镜像	SWR Admin	SWR Admin：容器镜像服务（SWR）管理员，拥有该服务下的所有权限。 如何创建自定义镜像，请参见 3.2.4 使用容器镜像部署函数。
挂载 sfs turbo 文件系统	SFS Administrator 或 Tenant administrator	SFS Administrator：弹性文件服务（SFS）管理员，拥有该服务下的所有权限。 Tenant administrator：全部云服务管理员（除 IAM 管理权限），拥有该权限的用户可以对企业拥有的所有云资源执行任意操作。
挂载 ECS 共享目录	Tenant Guest 及 VPC Administrator	Tenant Guest：全部云服务只读权限（除 IAM 权限） VPC Administrator：网络管理员 需要给函数设置委托至少拥有 Tenant Guest 以及 VPC Administrator 权限。
使用 APM 调用链	APM Administrator	APM Administrator：应用性能管理服务管理员。
配置跨域 VPC 访问	VPC Administrator	拥有 VPC Administrator 权限的用户可以对 VPC 内所有资源执行任意操作。在函数配置跨 VPC 访问时，则函数必须配置具备 VPC 管理权限的委托。 如何配置跨域 VPC 访问，请参见 3.3.4 配置网络。

创建委托

按照如下参数设置委托。

1. 登录统一身份认证服务（IAM）控制台。
2. 在统一身份认证服务（IAM）的左侧导航窗格中，选择“委托”页签，单击右上方的“+创建委托”。

图 3-13 创建委托

3. 开始配置委托。

图 3-14 填写基本信息

- 委托名称：serverless-trust。
- 委托类型：选择“云服务”。
- 云服务：选择“函数工作流 FunctionGraph”。
- 持续时间：选择“永久”。
- 描述：填写描述信息。

4. 单击“下一步”，进入委托选择页面，在右方搜索框中搜索需要添加的权限并勾选。此处以添加 Tenant Administrator 权限为例。

图 3-15 选择策略

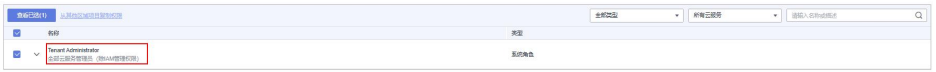


表 3-14 委托权限示例

权限名称	使用描述/场景
Tenant Administrator	全部云服务管理员（除 IAM 管理权限），拥有该权限的用户可以对企业拥有的所有云资源执行任意操作。

5. 单击“下一步”，选择权限的作用范围。

配置函数委托

1. 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。
2. 选择待配置的函数，单击进入函数详情页。
3. 选择“设置 > 权限”，单击“创建委托”，根据实际业务场景，配置函数委托。

表 3-15 配置函数委托参数说明

参数	说明
函数配置委托	选择已创建的函数委托。
函数执行委托	勾选“为函数执行单独设置委托”，需配置此参数。

说明

- 在创建函数过程中选择委托时，勾选“为函数执行单独设置委托”时，弹出“函数执行委托”，函数执行委托与函数配置委托可独立设置，这将减小不必要的性能损耗；不勾选时，函数执行委托和函数配置委托将使用同一委托，即使用同一个选择的委托或不使用任何委托。

4. 配置完成后单击“保存”。

修改委托

修改委托：如果需要修改委托的权限、持续时间、描述等，可以在 IAM 控制台修改委托。

注意

FunctionGraph 相关委托修改后，约 10 分钟生效（如 context.getToken 更新）。

3.3.4 配置网络

访问公网

函数创建成功后，默认具有公网访问权限，即函数可直接访问公网上的服务。函数访问公网上的服务需要固定公网出口 IP 的场景（例如被访问服务需要白名单验证），可以通过开启 VPC，在 VPC 内配置 NAT 网关绑定 EIP 的方式实现。

访问 VPC

函数支持用户创建虚拟私有云（VPC）并访问自己 VPC 内的资源。VPC 开启后，函数不再具有默认的公网访问权限，如果需要访问公网，可通过在 VPC 内配置 NAT 网关绑定 EIP 的方式实现。

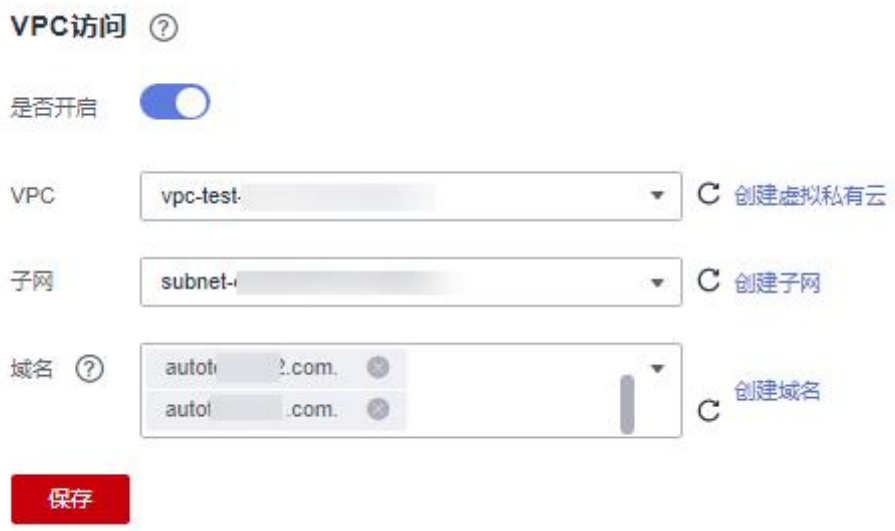
相关权限

配置委托权限请参见 3.3.3 配置委托权限。

操作步骤

1. 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。
2. 选择待配置的函数，单击进入函数详情页。
3. 选择“设置 >VPC”，开启 VPC，配置 VPC 和子网。

图 3-16 配置 VPC



4. 配置完成后单击“保存”。

配置固定公网 IP

函数需要在 VPC 内访问公网或者需要固定公网 IP 的场景，可以选择给 VPC 添加 NAT 网关并绑定 EIP 的方式。

前提条件

1. 已创建虚拟私有云和子网。
2. 已申请弹性云公网 IP。

创建 NAT 网关步骤如下

1. 登录 NAT 网关控制台，单击“创建 NAT 网关”。
2. 在 NAT 网关创建页面，输入相关信息，选择已创建的虚拟私有云及子网（此处以 vpc-01 为例），在确认规格信息后提交，完成创建。具体操作步骤请参见创建公网 NAT 网关。
3. 创建完成后，单击 NAT 网关名称进入详情页面，选择“添加 SNAT 规则”，单击“确定”完成配置。

网络限制

默认 NAT 访问限制：当前函数默认的 NAT 访问带宽在多个租户间共享，带宽小，仅适合小量调用的测试业务场景使用；如果对带宽、性能、可靠性有高要求的生产业务场景，需开启函数访问 VPC，在 VPC 内添加 NAT 网关并绑定 EIP，分配独占的外网访问带宽。

3.3.5 配置磁盘挂载

概述

FuncitonGraph 提供了文件系统挂载功能，多个函数可以通过共用一个文件系统，实现文件共享。相比于对单个函数实例分配的临时磁盘空间限制，可以极大扩展函数的执行和存储空间。

场景介绍

目前 FunctionGraph 函数支持以下文件系统配置。

- SFS Turbo 文件系统

  SFS Turbo 分为 SFS Turbo 标准型、SFS Turbo 标准型-增强版、SFS Turbo 性能型和 SFS Turbo 性能型-增强版。SFS Turbo 为用户提供一个完全托管的共享文件存储，能够弹性伸缩至 320TB 规模，具备高可用性和持久性，为海量的小文件、低延迟高 IOPS 型应用提供有力支持。适用于多种应用场景，包括高性能网站、日志存储、压缩解压、DevOps、企业办公、容器应用等。

- ECS 共享目录

  ECS 共享目录是通过 nfs 服务，把 ECS 上的指定目录设置为共享文件系统，函数（和 ECS 相同的 VPC 配置）可以挂载对应目录进行读写等操作，实现计算资源的动态扩展。此类型适合业务不太频繁的场景。

使用文件系统挂载功能具有以下优势：

- 函数执行空间相比于/tmp，可以极大扩展存储空间。
- 多个函数之间可以共享访问已经配置好的文件系统。
- ECS 计算资源动态扩展，利用 ECS 已有的存储能力实现更大的计算能力。

说明

您可以在**/tmp** 路径下写临时文件，最大不能超过 512MB。

创建委托

为函数添加文件系统配置需要先给函数设置相关服务的委托。

创建委托时，委托类型选择云服务，云服务选择 FunctionGraph，因为委托数目有限，而且目前界面上不支持修改，建议可以创建一个权限较大的委托（Tenant Administrator），可以支持在函数中操作当前区域内的所有资源，请参见 3.3.3 配置委托权限。

添加 sfs turbo 文件系统

设置委托

挂载 sfs turbo 文件系统需要给函数设置委托（至少拥有 sfs administrator 以及 VPC administrator 权限）。如果没有对应权限的委托，需要新创建。

设置 VPC

sfs turbo 涉及 VPC 内部网络访问，添加 sfs turbo 文件系统前需要给函数配置 sfs turbo 对应的 VPC。

1. 在弹性文件服务中，获取需要挂载的文件系统的 VPC 和子网信息。
2. 参见 3.3.4 配置网络开启 VPC 访问，输入 1 中获取的 VPC 和子网。

添加挂载-SFS Turbo

添加 sfs turbo 和添加 sfs 过程相似，只要选好需要挂载的文件系统，设置好函数访问路径即可。

添加 ECS 共享目录

添加委托

挂载 ECS 共享目录需要给函数设置委托（至少拥有 tenant guest 以及 VPC administrator 权限），如果没有对应权限的委托，需要新创建。

配置 VPC

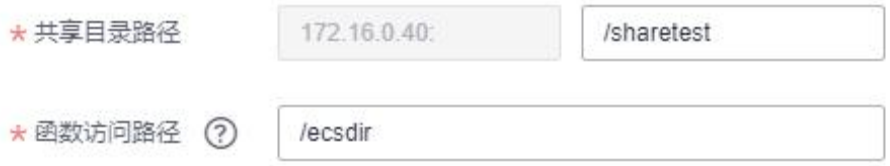
添加 ECS 共享目录前，也需要给函数配置 ECS 对应的 VPC，可以到 ECS 详情页的“基本信息”页签中查看“虚拟私有云”。单击虚拟私有云名称，进入虚拟私有云的详情页，查看子网。

获取到这两个信息后，可以在函数配置中配置对应的 VPC。

添加挂载-ECS

需要在界面上输入 ECS 上的共享目录路径信息和函数访问路径。

图 3-17 填写路径信息



后续操作

当函数挂载了文件系统配置后，对函数访问路径的读写就相当于对相关文件系统的读写。

如果把日志路径配置为函数访问路径的子目录，就可以轻松实现函数日志的持久化。

ECS 创建 nfs 共享目录

1. **Linux 系统**
   - CentOS、SUSE、Euler OS、Fedora 或 OpenSUSE 等系统
     i. 配置 yum 源

     ①在/etc/yum.repos.d 目录下创建文件 euleros.repo（文件名可随意取，但是必须以“.repo”结尾）。

     ②使用如下命令进入 euleros.repo 编辑配置信息。

     ```
     vi /etc/yum.repos.d/euleros.repo
     ```

     Euler 2.0SP3 yum 配置信息如下：

     ```
     [base]
     name=EulerOS-2.0SP3 base
     baseurl=http://repo.cloud.com/euler/2.3/os/x86_64/
     enabled=1
     gpgcheck=1
     gpgkey=http://repo.cloud.com/euler/2.3/os/RPM-GPG-KEY-EulerOS
     ```

     Euler 2.0SP5 yum 配置信息如下：

     ```
     [base]
     name=EulerOS-2.0SP5 base
     baseurl=http://repo.cloud.com/euler/2.5/os/x86_64/
     enabled=1
     gpgcheck=1
     gpgkey=http://repo.cloud.com/euler/2.5/os/RPM-GPG-KEY-EulerOS
     ```

说明

参数说明

name：仓库的名称。

baseurl：仓库的地址。

- 使用 http 协议的网络地址：http://path/to/repo
- 使用本地仓库地址：file:///path/to/local/repo

gpgcheck：表示是否进行 gpg（GNU Private Guard）校验，以确定 RPM 包来源的有效性和安全性。gpgcheck 设置为 1 表示进行 gpg 校验，0 表示不进行 gpg 校验。如果没有这一项，默认是检查的。

③保存配置的 repo 文件。

④执行如下命令清理缓存。

```
yum clean all
```

ii. 使用如下命令安装 nfs-utils

```
yum install nfs-utils
```

iii. 设置共享文件夹

打开/etc/exports，比如要把/sharedata 目录设置为共享目录，可以填入如下内容：

/sharedata　192.168.0.0/24(rw,sync,no_root_squash)

说明

上述内容的含义是：把/sharedata 这个目录共享给 192.168.0.0/24 这个子网段的其他服务器。

命令输入完成后，可以执行命令 exportfs -v 显示共享的目录，从而判断是否设置成功。

iv. 使用如下命令启动 nfs 服务

```
systemctl start rpcbind
service nfs start
```

v. 修改共享目录

比如需要新增/home/myself/download 到共享目录，可以在/etc/exports 中新增如下内容。

/home/myself/download　192.168.0.0/24(rw,sync,no_root_squash)

然后重启 nfs 服务。

```
service nfs restart
```

或者用如下命令，无需重启 nfs 服务。

```
exportfs -rv
```

vi. 设置 rpcbind 开机启动（可选）

如果需要设置 rpcbind 服务开机启动，可执行如下命令。

```
systemctl enable rpcbind
```

- **Ubuntu 系统**

i. 使用如下命令安装 nfs-kernel-server

```
sudo apt-get update
sudo apt install nfs-kernel-server
```

ii. 设置共享文件夹

打开/etc/exports，比如要把/sharedata 目录设置为共享目录，可以填入如下内容。

/sharedata　192.168.0.0/24(rw,sync,no_root_squash)

说明

上述内容的含义是：把/sharedata 这个目录共享给 192.168.0.0/24 这个子网段的其他服务器。

命令输入完成后，可以执行命令 exportfs -v 显示共享的目录，从而判断是否设置成功。

iii. 启动 nfs 服务

```
service nfs-kernel-server restart
```

iv. 修改共享目录

比如需要新增/home/myself/download 到共享目录，可以在/etc/exports 中新增如下内容：

/home/myself/download　192.168.0.0/24(rw,sync,no_root_squash)

然后重启 nfs 服务

```
service nfs restart
```

或者用如下命令，无需重启 nfs 服务：

```
exportfs -rv
```

2. **Windows 系统**

1. 安装 nfs server 软件

目前可用的收费软件有：hanewin nfs server，可到对应官网下载。

免费的有：FreeNFS、winnfsd 等，可到 sourceforge 上下载。

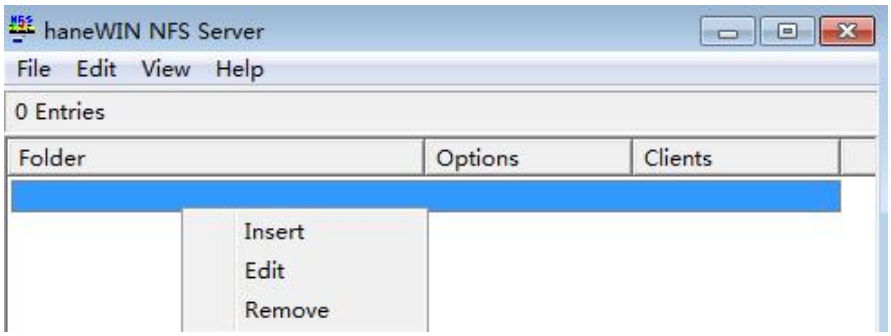
2. 打开 nfs 功能

- 如果是 winnfsd，可参见：https://github.com/winnfsd/winnfsd。
- 如果是 hanewin nfs server，可以参见如下步骤。

i. 以 Windows 系统管理员身份运行 nfsctl.exe

ii. 在空白处右键，然后 Insert，完成设置

图 3-18 Insert



3.3.6 配置环境变量

概述

环境变量可以在不修改代码的情况下，将动态参数传递到函数，调整函数的执行行为。

应用场景

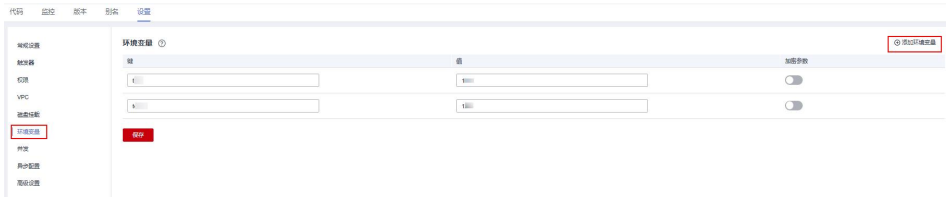
- 区分多环境：相同的函数逻辑，可根据部署环境的不同，配置不同的环境变量以区分。例如，通过环境变量给测试和开发环境配置不同的数据库。

- 配置加密：函数中访问其他服务的认证信息，例如账号和密码，ak/sk，可通过配置加密环境变量，在代码中动态获取，保证敏感数据的安全。
- 动态配置：函数逻辑中需要动态调整的配置，例如查询周期、超时时间，可提取为环境变量避免业务每次变化都需要修改代码。

操作步骤

设置 FunctionGraph 函数的加密配置和环境变量，无需对代码进行任何更改，可以将设置动态参数传递到函数代码和库。

图 3-19 添加环境变量



例如 Node.js 语言加密配置和环境变量的值(value)可以通过 Context 类中的 getUserData(string key)获取。

警告

- 设置加密配置、环境变量时，用户自定义的键(key)/值(value)，键(key)输入规范：可包含字母、数字、下划线_，以大/小写字母开头。
- 设置“键”和“值”的总长度不超过 4096 个字符。
- 设置环境变量时，FunctionGraph 会明文展示所有输入信息，请不要输入敏感信息（如帐户密码等），以防止信息泄露。
- 打开加密开关之后，界面上会对键值进行加密，参数传输过程中键值也处于加密状态。

预设值

环境变量存在如下预设值，您无法配置和预设值同名的环境变量。

表 3-16 预设值及说明

环境变量名	含义	获取方式和默认值
RUNTIME_PROJECT_ID	函数的项目 ID	Context 类提供接口或通过系统环境变量获取
RUNTIME_FUNC_NAME	函数名称	Context 类提供接口或通过系统环境变量获取
RUNTIME_FUNC_VERSI	函数版本	Context 类提供接口或通过

环境变量名	含义	获取方式和默认值
ON		系统环境变量获取
RUNTIME_HANDLER	函数执行入口	通过系统环境变量获取
RUNTIME_TIMEOUT	函数执行的超时时间	通过系统环境变量获取
RUNTIME_USERDATA	用户通过环境变量传入的值	Context 类提供接口或通过系统环境变量获取
RUNTIME_CPU	函数占用的 CPU 资源，取值与 MemorySize 成比例	Context 类提供接口或通过系统环境变量获取
RUNTIME_MEMORY	函数配置的内存大小	Context 类提供接口或通过系统环境变量获取 单位 MB
RUNTIME_MAX_RESP_BODY_SIZE	最大返回值限制	通过系统环境变量获取 系统默认为 6291456 Byte
RUNTIME_INITIALIZER_HANDLER	函数初始化入口	通过系统环境变量获取
RUNTIME_INITIALIZER_TIMEOUT	函数初始化超时时间	通过系统环境变量获取
RUNTIME_ROOT	Runtime 包的路径	通过系统环境变量获取 系统默认路径为 /home/snuser/runtime
RUNTIME_CODE_ROOT	代码在容器中的存放目录	通过系统环境变量获取 系统默认路径为 /opt/function/code
RUNTIME_LOG_DIR	系统日志在容器中存放的目录	通过系统环境变量获取 系统默认路径为 /home/snuser/log

示例

使用环境变量设置以下信息：安装文件的目录、存储输出的位置、存储连接和日志记录设置等。这些设置与应用程序逻辑解耦，在需要变更设置时，无需更新函数代码。

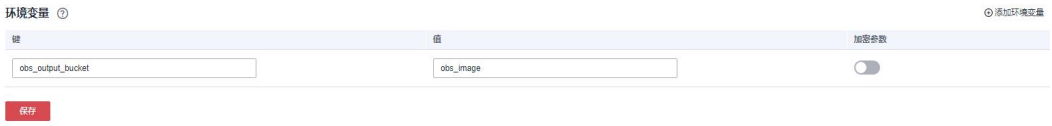
在如下函数代码片段中，参数“obs_output_bucket”为图片处理后存储地址。

```
def handler(event, context):
    srcBucket, srcObjName = getObsObjInfo4OBSTrigger(event)
    obs_address = context.getUserData('obs_address')
    outputBucket = context.getUserData('obs_output_bucket')
    if obs_address is None:
```

```
    obs_address = '{obs_address_ip}'
if outputBucket is None:
    outputBucket = 'casebucket-out'

ak = context.getAccessKey()
sk = context.getSecretKey()

# download file uploaded by user from obs
GetObject(obs_address, srcBucket, srcObjName, ak, sk)

outFile = watermark_image(srcObjName)

# 将转换后的文件上传到新的obs桶中
PostObject (obs_address, outputBucket, outFile, ak, sk)

return 'OK'
```

通过设置环境变量 obs_output_bucket，可以灵活设置存储输出图片的 OBS 桶。

图 3-20 环境变量



3.3.7 配置函数异步

概述

函数可以被同步或异步调用，异步调用场景下，FunctionGraph 持久化请求后立即返回，不等待请求最终处理完成，用户无法实时感知请求处理结果。如果您希望异步请求处理失败后重试或者希望获取异步处理结果通知，可通过函数异步配置项进行设置。

应用场景

- 失败重试：用户代码异常造成的失败，FunctionGraph 默认不重试。如果函数中有需要重试的场景，例如调用三方服务经常失败，可配置重试提升成功率。
- 结果通知：FunctionGraph 可自动通知异步函数的执行结果给下游服务做进一步处理。例如执行失败信息保存到 OBS，后续分析失败原因；执行成功信息推送到 DIS 或再次触发函数做处理等。

操作步骤

步骤 1 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。

步骤 2 选择待配置的函数，单击进入函数详情页。

步骤 3 选择“设置 > 异步配置”，在“异步配置”页签，单击“配置异步调用”。

图 3-21 配置异步调用

步骤 4 填写配置参数，参见如下表 3-17，例如设置目标服务为函数工作流（FunctionGraph）。

图 3-22 填写配置参数

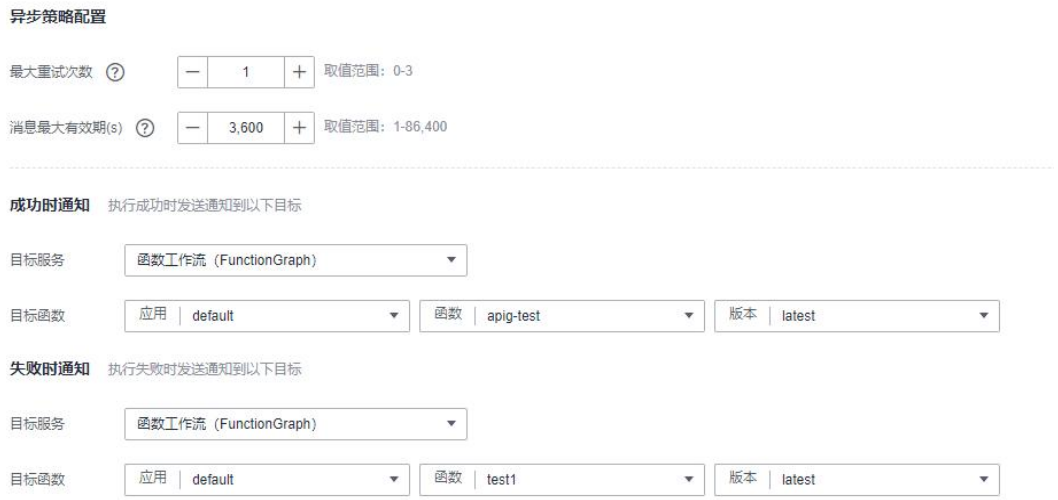


表 3-17 参数说明

参数	说明
异步策略配置	• 最大重试次数：异步调用失败后最大重试次数，默认为 1 次，取值范围：0-3。 • 消息最大有效期（s）：消息最大存活时长，取值范围：1-86400。
成功时通知	目标服务：执行成功时发送通知到以下目标服务 1. 函数工作流（FunctionGraph） 2. 对象存储服务（OBS） 3. 消息通知服务（SMN）

参数	说明
失败时通知	目标服务：执行失败时发送通知到以下目标服务 1. 函数工作流（FunctionGraph） 2. 对象存储服务（OBS） 3. 消息通知服务（SMN）

步骤 5 填写完成后单击“确定”。

说明

1. 异步配置通知到目标服务时，需配置具有目标服务操作权限的函数委托。
2. 当您在配置异步执行通知目标时，请务必保证不要出现循环调用的情况。例如：您为函数 A 配置了成功调用时的异步通知目标为函数 B，为函数 B 配置了成功调用时的异步通知目标为函数 A，当您异步触发函数 A 并且执行成功后，则可能出现 A—>B—>A.....循环调用的情况。

----结束

配置说明

异步调用目标的配置说明参见表 3-18，异步调用目标的事件内容参见如下示例：

```
{
       "timestamp": "2020-08-20T12:00:00.000Z",
    "request context": {
           "request id": "1167bf8c-87b0-43ab-8f5f-26b16c64f252",
           "function urn": "urn:fss:xx-xxxx-x:xxxxxxx:function:xxxx:xxxx:latest",
           "condition": "",
           "approximate invoke count": 0
     },
       "request_payload": "",
       "response_context": {
           "status_code": 200,
           "function_error": ""
      },
     "response_payload": "hello world!"
}
```

表 3-18 配置参数说明

参数	说明
timestamp	调用时间戳。
request_context	请求上下文。
request_context.request_id	异步调用的请求 ID。
request_context. function_urn	异步执行的函数 URN。
request_context.condition	调用错误类别。

参数	说明
request_context. approximate_invoke_count	异步调用的执行次数。当该值大于 1 时，说明函数计算对您的函数进行了重试。
request_payload	请求函数的原始负载。
response_context	返回上下文。
response_context.statusCode	调用函数的返回码（系统）。当该返回码不为 200 时，说明出现了系统错误。
response_context.function_error	调用错误信息。
response_payload	执行函数返回的原始负载。

3.3.8 配置单实例多并发

说明

该特性仅 FunctionGraph v2 版本支持。

概述

默认情况下，每个函数实例同一时刻只处理一个请求，多并发时，例如并发三个请求，FunctionGraph 会启动三个函数实例处理请求。FunctionGraph 推出的单实例多并发能力，可以让您在一个实例上并发处理多个请求。

应用场景

单实例多并发适合函数处理逻辑中有较长时间等待下游服务响应的场景，也适合函数逻辑中初始化时间较长的场景，具备以下优势：

- 降低冷启动概率，优化函数处理时延：例如并发三个请求，不配置单实例多并发，FunctionGraph 默认启动三个实例处理请求，会有三次冷启动。若配置了单实例支持三并发，三个并发请求，FunctionGraph 只启动一个实例处理请求，减少了两次冷启动。
- 减少总请求处理时长，节省费用：单实例单并发下，多个请求的总处理时长为每个请求的处理时长相加。

单实例单并发与单实例多并发的对比

当一个函数执行需要花费 5 秒，若配置为单实例单并发，三次函数调用请求分别在三个函数实例执行，总执行时长为 15 秒。

若配置为单实例多并发，设置单实例并发数为 5，即单个实例最多支持 5 个并发请求，如果有三次函数调用请求，将在一个实例内并发处理，总执行时间为 5 秒。

说明

单实例并发数大于 1，在您设置的“单函数最大实例数”范围内，超过单实例并发处理能力时会自动扩容新实例。

表 3-19 单并发与多并发对比

对比项	单实例单并发	单实例多并发
日志打印	-	Node.js Runtime 使用 console.info()函数，Python Runtime 使用 print()函数，Java Runtime 使用 System.out.println()函数打印日志，该方式会把当前请求的 Request ID 包含在日志内容中。当多请求在同一个实例并发处理时，当前请求可能有很多个，继续使用这些函数打印日志会导致 Request ID 错乱。此时应该使用 context.getLogger()，获取一个日志输出对象，通过这个日志输出对象打印日志，例如 Python Runtime： log = context.getLogger() log.info("test")
共享变量	不涉及	单实例多并发处理时，修改共享变量会导致错误。这要求您在编写函数时，对于非线程安全的变量修改要进行互斥保护。
监控指标	按实际情况进行监控。	相同负载下，函数的实例数明显减少。
流控错误	不涉及	太多请求时，body 中的 errorcode 为“FSS.0429” ，响应头中的 status 为 429 ，错误信息提示：Your request has been controlled by overload sdk, please retry later。

配置单实例多并发

1. 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。
2. 选择待配置的函数，单击进入函数详情页。
3. 选择“设置 > 并发”，开始配置。

   参见表 3-20 进行配置，完成后单击“保存”。

图 3-23 并发基础配置

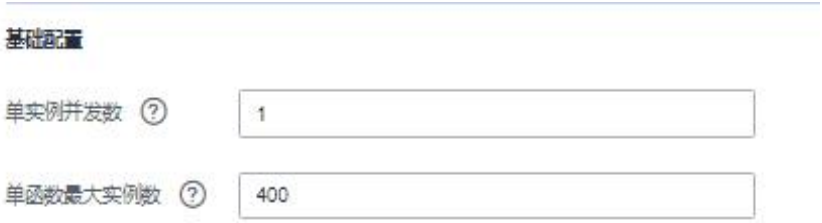


表 3-20 参数说明

参数	说明
单实例并发数	单个实例支持的请求并发数。取值范围为 1-1000。
单函数最大实例数	单个函数的运行实例数，默认值 400，最大值为 1000；-1 表示不限制实例数；0 代表禁用。 **说明** 超过实例数限制处理能力的请求会被直接丢弃，而不是重试。 当前超过实例数限制导致的请求错误不会直接显示在函数日志中，您可以通过 3.3.7 配置函数异步来获取错误详细信息。

配置约束

- 对于 Python 函数，由于 Python GIL 锁导致实例上的线程被绑定到一个核上，造成多并发无法使用多核，即使配置更大资源规格也无法提升函数处理性能。
- 对于 Node.js 函数，由于 V8 引擎的单进程单线程，造成多并发无法使用多核，即使配置更大资源规格也无法提升函数处理性能。

3.3.9 版本管理

概述

在函数从开发、测试、生产过程中，可以发布一个或多个版本，实现对函数代码的管理。对于发布的每个版本的函数、环境变量会另存为相应版本的快照，函数代码发布后，您可以根据实际需要修改版本配置信息。

函数创建以后，默认版本为 latest 版本，每个函数都有一个 latest 版本。函数代码发布后，您可以根据实际需要修改版本配置信息。

说明

版本相当于函数服务的快照，可对应代码里的 tag，函数版本会对应函数的配置、代码等，新版本默认不绑定触发器。当用户新建版本后，对应版本的配置（如环境变量等）、代码等都无法更新，从而保证版本的稳定性、可追溯性等。

发布版本

1. 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。
2. 选择待配置的函数，单击进入函数详情页。
3. 在“版本”页签下，单击“发布新版本”。

图 3-24 发布新版本参数配置

- 版本号：您自定义的版本号，用于区分不同的版本。当您未设置时，系统以时间生成版本号，例如：v20220510-190658。
- 描述：对于版本的描述信息，可以不填。

4. 单击“确定”，系统自动完成版本发布，当前函数版本也会切换至新创建的版本。

说明

- 单个函数最多可以发布 10 个版本。
- latest 版本设置了预留实例，能修改函数配置。新发布的非 latest 版本默认不带预留实例。
- 基于 latest 创建的新版本默认不会挂载磁盘，如果不绑定触发器就无法单独设置环境变量。

删除版本

1. 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。
2. 选择待配置的函数，单击进入函数详情页。
3. 在“latest 版本”的“版本”页签下，选择需要删除的函数版本。

图 3-25 删除版本

说明

- latest 版本不能删除。
- 如果函数版本关联了别名，则删除版本时会把关联的别名删除。

4. 单击弹框中的“确认”，删除函数版本。

⚠ 警告

删除版本将永久删除关联的代码、配置、别名及事件源映射，但不会删除日志。删除操作无法恢复，请谨慎操作。

3.3.10 别名管理

概述

别名指向函数的特定版本，推荐您创建别名并把别名暴露给客户端（例如绑定触发器到别名上而不是某个版本上）。这样，通过修改在别名上配置的版本，可以实现版本的更新和回滚，客户端无感知。一个别名支持配置最多两个版本，在不同的版本上可以分配不同的权重，实现灰度发布。

创建别名

1. 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。
2. 选择待配置的函数，单击进入函数详情页。
3. 在“别名”页签下，单击“创建别名”。

图 3-26 创建别名

- 别名名称：您自定义的别名名称，用于区分不同的别名。
- 对应版本：选择需要关联的版本。
- 开启灰度版本：选择是否开启灰度版本，开启灰度版本后，一个别名可以同时关联两个版本，根据设置的权重比例，函数切换部分主版本的请求到灰度版本运行。
- 灰度版本：选择需要关联的灰度版本，latest 版本不能作为灰度版本。
- 权重：为灰度版本设置权重，支持输入 0-100 的整数。
- 描述：对于别名的描述信息。

4. 单击“确定”，完成别名的创建。

说明

单个函数最多可以创建 10 个别名。

修改别名

1. 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。
2. 选择待配置的函数，单击进入函数详情页。
3. 在“latest 版本”的“别名”页签下，选择需要修改的函数别名。

图 3-27 修改别名

4. 单击“确定”，完成函数别名修改。

删除别名

1. 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。
2. 选择待配置的函数，单击进入函数详情页。
3. 在“latest 版本”的“别名”页签下，选择需要删除的函数别名。

图 3-28 删除别名

4. 单击弹框中的“确认”，删除函数版本。

3.3.11 配置动态内存

概述

默认情况下，一个函数唯一绑定了一个资源规格。开启动态内存可以让您在处理指定请求时，设置本次处理函数实例使用的资源规格，如果您不指定，函数将使用默认配置的资源规格。

应用场景

以使用函数做视频转码为例：视频文件大小从 MB 到 GB，不同编码格式和分辨率对转码需要的计算资源要求差别很大。为了保证转码性能，通常需要配置一个很大的资源规格，但是在处理低分辨率（例如短视频）视频时，会造成资源浪费。您可以把转码业务实现为元数据获取和转码两个函数，根据元数据信息指定转码函数的资源规格，最小化资源占用，达到更低的成本开销。

前提条件

已创建函数，若未创建，请参见 3.2.2 使用空白模板创建函数。

操作步骤

步骤 1 登录 FunctionGraph 控制台，在左侧导航栏选择“函数 > 函数列表”，单击已创建的函数名称。

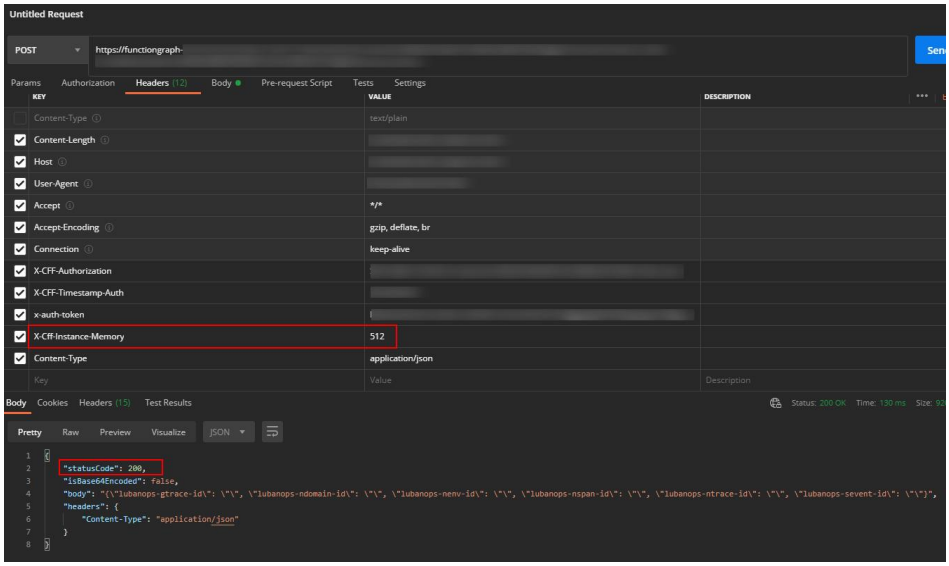
图 3-29 选择已创建的函数

步骤 2 在“设置 > 高级设置”页签下，开启“动态内存”。

步骤 3 通过本地工具调用同步执行函数或异步执行函数接口，然后在请求头的数据结构中添加请求头“X-Cff-Instance-Memory”，值可以设置为 128、256、512、768、1024、1280、1536、1792、2048、2560、3072、3584、4096。

此处以通过 postman 调用为例，在“Headers”中添加请求头“X-Cff-Instance-Memory”，设置 value 指为 512，调用成功返回“200”。

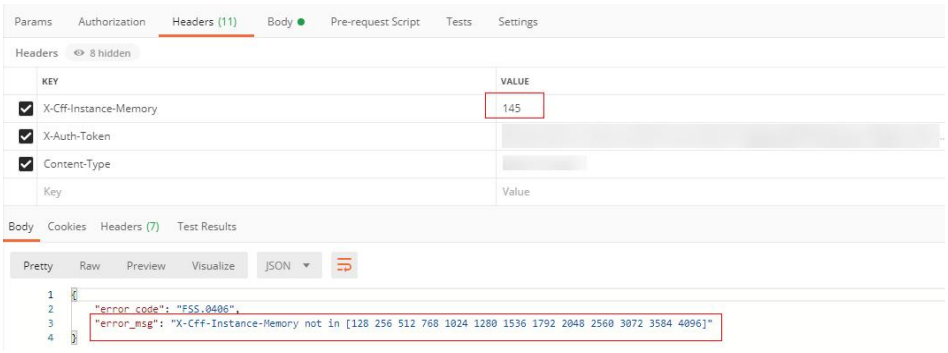
图 3-30 添加请求头并调用



说明

- 未开启动态内存，调用接口时默认取创建函数时设置的内存大小；
- 若配置了动态内存，未设置 value 值，调用同步执行接口或异步执行接口时仍默认取创建函数设置的内存大小，调用成功返回“200”。
- 若配置了动态内存，内存值设置错误，未包含在 128、256、512、768、1024、1280、1536、1792、2048、2560、3072、3584、4096 中，调用接口时，返回错误码“FSS.0406”，您只需重新设置 value 值即可调用成功。

图 3-31 调用失败



----结束

3.3.12 配置心跳函数

概述

心跳函数用于检测用户函数运行时的异常，例如以下场景：

- 用户函数死锁，无法正常运行。
- 用户函数内存溢出，无法正常运行。
- 用户函数网络异常，无法正常运行。

在配置了自定义心跳函数后，当用户函数运行时，FunctionGraph 每隔 5s 向函数实例发送一次心跳请求，触发心跳函数。如果心跳请求返回异常，FunctionGraph 会认为函数实例异常，终止此函数实例。

FunctionGraph 心跳请求的超时时间是 3s，如果连续 6 次心跳请求未响应，函数实例将被终止。

约束条件

1. **当前心跳函数只支持 Java 语言**。
2. 心跳函数入口需要与函数执行入口在同一文件下。

   Java 心跳函数格式为：

   ```
   public boolean heartbeat() {
       // 自定义检测逻辑
       return true
   ```

3. 心跳函数目前无输入参数，返回值为 bool 类型。

操作步骤

步骤 1 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。

步骤 2 选择待配置的函数，单击进入函数详情页。

步骤 3 选择“设置 > 高级设置”，开始配置。

步骤 4 开启“配置心跳函数”开关，并填写心跳函数入口。

图 3-32 配置心跳函数

表 3-21 心跳函数配置说明

参数	说明
配置心跳函数	开启心跳函数，FunctionGraph 将检测用户函数运行时的异常场景。
心跳函数入口	心跳函数入口需要与函数执行入口在同一文件下。 格式为[包名].[类名].[执行函数名]，不超过 128 个字符。

步骤 5 配置完成后单击“保存”。

----结束

3.4 在线调试

注意事项

事件数据作为 event 参数传入入口函数，配置后保存可以持久化，以便下次测试使用。每个函数最多可配置 10 个测试事件。

创建测试事件

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

步骤 2 单击函数名称，进入函数详情界面。

步骤 3 在函数详情页，选择函数版本，单击“测试”，弹出“配置测试事件”页。

步骤 4 在“配置测试事件”界面填写测试信息，如表 3-22 所示，带*参数为必填项。

表 3-22 测试信息

参数	说明

参数	说明
配置测试事件	可创建新的测试事件也可编辑已有的测试事件。 默认值为：“创建新的测试事件”。
事件模板	使用空白模板需要编辑测试事件。 使用已有模板会自动加载相对应的测试事件，事件模板说明如表 3-23 所示。
*事件名称	事件名称必须以大写或小写字母开头，支持字母（大写或小写），数字和下划线“_”（或中划线“-”），并以字母或数字结尾，长度为 1-25 个字符，例如 even-123test。
测试事件	输入测试事件。

表 3-23 事件模板说明

模板名称	模板说明
timer-event-template	模拟 TIMER 事件，触发函数。

步骤 5 单击“保存”，完成测试事件创建。

----结束

测试函数

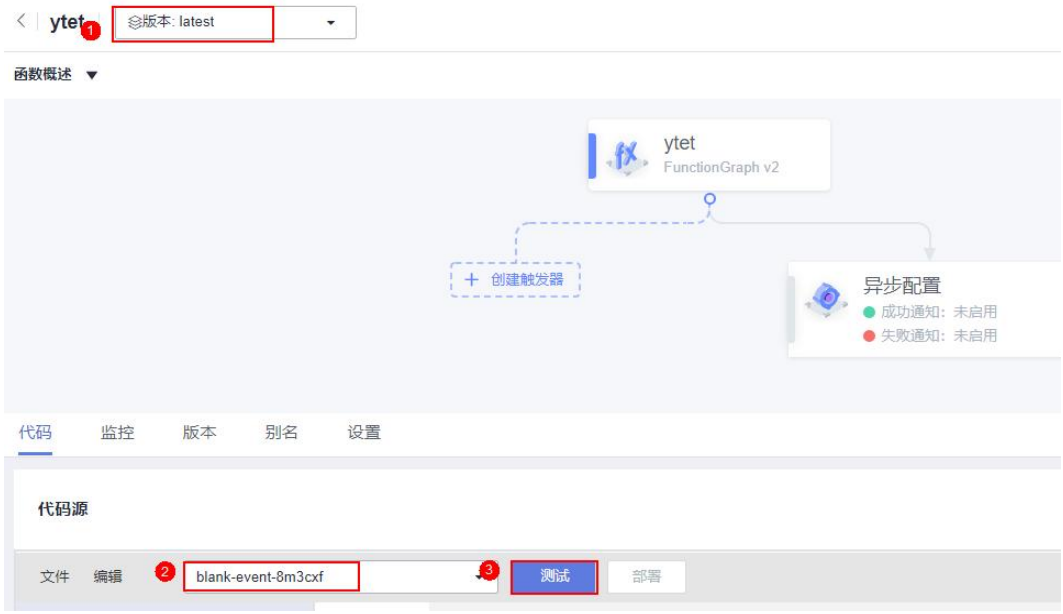
函数创建以后，可以在线测试函数能否正常运行，验证能否实现预期功能。

步骤 1 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。

步骤 2 单击函数名称，进入函数详情界面。

步骤 3 在函数详情页，选择函数版本，选择测试事件，单击“测试”。

图 3-33 选择测试事件



步骤 4 单击“测试”，可以得到函数运行结果。

说明

“日志”页签最多显示 2K 日志，如需查看完整日志，请参见 3.7.2.2 管理函数日志的操作。

----结束

修改测试事件

步骤 1 登录 FunctionGraph 控制台，进入“函数”界面。

步骤 2 选择“函数列表”，单击函数名称，进入函数详情界面。

步骤 3 在函数详情页，选择函数版本，单击“配置测试事件”，弹出“配置测试事件”页。

步骤 4 在“配置测试事件”界面修改测试信息，如表 3-24 所示。

表 3-24 测试信息

参数	说明
创建新的测试事件	重新创建新的测试事件。
编辑已有测试事件	修改已有的测试事件。
测试事件	修改测试事件代码。

步骤 5 单击“保存”，完成配置修改。

----结束

删除测试事件

步骤 1 登录 FunctionGraph 控制台，进入“函数”界面。

步骤 2 选择“函数列表”，单击函数名称，进入函数详情界面。

步骤 3 在函数详情页，选择函数版本，单击“请选择测试事件 > 请配置测试事件”，弹出“配置测试事件”页。

步骤 4 在“配置测试事件”界面选择测试信息，如表 3-25 所示。

表 3-25 测试信息

参数	说明
请配置测试事件	选择“编辑已有的测试事件”。
已保存测试事件	选择需要删除的测试事件。

步骤 5 单击“删除”，完成配置删除。

----结束

3.5 配置触发器

3.5.1 触发器管理

停用/启用触发器

已经创建的触发器，通过设置停用/启用，控制触发器的状态。**OBS 触发器、APIG 触发器创建以后，不能停用，只能删除。**

步骤 1 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。

步骤 2 单击函数名称，进入函数详情界面。

步骤 3 选择“设置 > 触发器”，进入“触发器”页签，在需要停用/启用的触发器所在行，单击“停用”/“启用”，停用/启用触发器。

----结束

删除触发器

已经创建的触发器，如果不再使用，可以执行删除操作。

步骤 1 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。

步骤 2 单击函数名称，进入函数详情界面。

步骤 3 单击“触发器”，进入“触发器”页签，在需要删除的触发器所在行，单击“删除”，删除触发器。

----结束

3.5.2 使用定时触发器

本节介绍创建定时触发器，按照设置的频率，定期触发函数运行，供用户了解定时触发器的使用方法。

前提条件

已经创建函数，创建过程请参见 3.2.2 使用空白模板创建函数。

创建定时触发器

步骤 1 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。

步骤 2 选择待配置的函数，单击进入函数详情页。

步骤 3 选择“设置 > 触发器”，单击“创建触发器”，弹出“创建触发器”对话框。

图 3-34 创建触发器

步骤 4 设置以下信息。

- 触发器类型：选择“定时触发器 (TIMER)”。
- 定时器名称：您自定义的定时器名称，例如：Timer。
- 触发规则：固定频率和 Cron 表达式。
  - 固定频率：固定时间间隔触发函数，该类型下支持配置单位为分、时、天，每种类型仅支持整数配置，其中分钟支持范围(0，60]，小时支持范围(0，24]，天支持范围(0，30]。
  - Cron 表达式：设置更为复杂的函数执行计划，例如：周一到周五上午 08:30:00 执行函数等，具体请参见 3.5.4 附录：函数定时触发器 Cron 表达式规则。
- 是否开启：是否开启定时触发器。
- 附加信息：如果用户配置了触发事件，会将该事件填写到 TIMER 事件源的“user_event”字段。

步骤 5 单击“确定”，完成定时触发器的创建。

----结束

查看函数运行结果

函数的定时触发器创建以后，每隔一分钟执行一次函数，可以查看函数运行日志。

步骤 1 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。

步骤 2 选择函数，单击进入函数详情页。

步骤 3 选择“监控 > 日志”，查询函数运行日志。

----结束

3.5.3 使用 APIG（专享版）触发器

本节介绍创建 APIG 触发器，使用 API 调用函数运行。供用户了解 APIG 触发器的使用方法。

前提条件

已经创建 API 分组，此处以 APIGroup_test 分组为例。

创建 APIG 触发器

步骤 1 登录，在左侧的导航栏选择“函数 > 函数列表”。

步骤 2 单击右上方的“创建函数”，进入“创建函数”页面。

步骤 3 设置以下函数信息。

- 函数名称：输入您自定义的函数名称，例如：apig。
- 委托名称：选择“不使用任何委托”。
- 企业项目：选择“default”。
- 运行时语言：选择“Python 2.7”。

步骤 4 单击“创建函数”，完成函数的创建。

步骤 5 在“代码”页签下，复制如下代码至代码窗并单击“部署”。

```
# -*- coding:utf-8 -*-
import json
def handler (event, context):
   body = "<html><title>Functiongraph Demo</title><body><p>Hello,
FunctionGraph!</p></body></html>"
   print(body)
   return {
      "statusCode":200,
      "body":body,
      "headers": {
         "Content-Type": "text/html",
      },
      "isBase64Encoded": False
   }
```

步骤 6 选择“设置 > 触发器”，单击“创建触发器”，弹出“创建触发器”对话框。

图 3-35 创建触发器

步骤 7 设置以下触发器信息。

表 3-26 触发器信息

字段	填写说明
触发器类型	选择“API 网关服务（APIG 专享版）”。
实例	选择所属实例，若无实例，可单击“创建实例”完成创建。
API 名称	您自定义的 API 名称，例如：API_apig。
分组	API 分组相当于一个 API 集合，API 提供方以 API 分组为单位，管理分组内的所有 API。 选择“APIGroup_test”。
发布环境	API 可以同时提供给不同的场景调用，如生产、测试或开发。API 网关服务提供环境管理，在不同的环境定义不同的 API 调用路径。 选择“RELEASE”，才能调用。
安全认证	API 认证方式： • App： 采用 Appkey&Appsecret 认证，安全级别高，推荐使用。 • IAM： IAM 认证，只允许 IAM 用户能访问，安全级别中等。 • None： 无认证模式，所有用户均可访问。 选择“None”。
请求协议	分为两种类型： • HTTP • HTTPS 选择“HTTPS”。
后端超时(毫秒)	输入“5000”。

步骤 8 单击“确定”，完成触发器的创建。

说明

1. “调用 URL”即 APIG 触发器调用地址。
2. API 触发器创建完成后，会在 API 网关生成名为 API_apig 的 API，单击 API 名称，跳转至 API 网关服务。

----结束

调用函数

步骤 1 在浏览器地址栏输入 APIG 触发器的调用地址 URL，按“Enter”。

步骤 2 函数执行完毕，得到返回结果，如图 3-36 所示。

图 3-36 返回结果

说明

1. FunctionGraph 函数对 APIG 调用的传入值为函数自带的事件模板，您可以参见表 3-23。
2. FunctionGraph 函数对来自 APIG 调用的返回结果进行了封装，APIG 触发器要求函数的返回结果中必须包含 body(String)、statusCode(int)、headers(Map)和 isBase64Encoded(boolean)，才可以正确返回。

----结束

查看函数运行结果

步骤 1 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。

步骤 2 选择函数，单击进入函数详情页。

步骤 3 选择“监控 > 日志”，查询函数运行日志。

----结束

3.5.4 附录：函数定时触发器 Cron 表达式规则

函数 Cron 表达式下支持如下几种配置方式。

- @every 格式

  @every *N*Unit，其中 N 表示一个正整数，Unit 可以为 ns, µs, ms, s, m, h，表示每隔 N 个 Unit 时间触发一次函数如表 3-27 所示。

表 3-27 表达式示例

表达式	含义

表达式	含义
@every 30m	每隔 30 分钟触发一次函数
@every 1.5h	每隔 1.5 小时触发一次函数
@every 2h30m	每隔 2 小时 30 分钟触发一次函数

- 标准 cron 表达式

  cron 表达式格式要求“秒 分 时 日 月 周(可选)”，每个字段间以空格隔开，其中各字段说明如表 3-28 所示。

表 3-28 cron 表达式字段说明

字段	说明	取值范围	允许的特殊字符
秒	必选	0-59	, - * /
分钟	必选	0-59	, - * /
时	必选	0-23	, - * /
日(Day of month)	必选	1-31	, - * ? /
月	必选	1-12 或者 Jan-Dec（英文不区分大小写）如表 3-29 所示。	, - * /
星期几(Day of week)	可选	0-6 或者 Sun-Sat（0 表示星期天，英文不区分大小写），如表 3-30 所示。	, - * ? /

表 3-29 月份字段取值说明

月份	数字	英文简写
1 月	1	Jan
2 月	2	Feb
3 月	3	Mar
4 月	4	Apr
5 月	5	May
6 月	6	Jun

月份	数字	英文简写
7 月	7	Jul
8 月	8	Aug
9 月	9	Sep
10 月	10	Oct
11 月	11	Nov
12 月	12	Dec

表 3-30 星期字段取值说明

星期	数字	英文简写
星期一	1	Mon
星期二	2	Tue
星期三	3	Wed
星期四	4	Thu
星期五	5	Fri
星期六	6	Sat
星期日	0	Sun

cron 表达式字段特殊字符说明如表 3-31 所示。

表 3-31 特殊字符说明

特殊字符	含义	说明
*	表示该字段中的所有值	在“分钟”字段中表示每一分钟都执行。
,	指定多个值（可以不连续）	在“月”字段中指定“Jan,Apr,Jul,Oct”或者“1,4,7,10”，表示 1 月，4 月，7 月和 10 月，在“星期几”字段中指定“Sat,Sun”或者“6,0”表示周六，周日。
-	指定一个范围	在“分钟”字段中使用 0-3，表示从 0 分到 3 分
?	指定一个或另一个	仅“日”和“星期几”字段可以指定。例如，如果指定了一个特定的日期，但你不关心该日期对应星期几，那么“星期几”字段就可以使

特殊字符	含义	说明
		用该特殊字符。
/	表示起步和步幅，n/m 表示从 n 开始，每次增加 m	在“分钟”字段 1/3 表示在满足其它字段情况下的绝对时间 1 分（例如 00:01:00）开始，每隔 3 分钟触发一次。

cron 表达式配置示例如表 3-32 所示。

表 3-32 cron 表达式配置示例

配置示例	示例说明
0 15 2 * * ?	每天凌晨 02:15:00 执行
0 30 8 ? * Mon-Fri	周一到周五上午 08:30:00 执行
0 45 7 1-3 * ?	每月 1, 2, 3 号上午 07:45:00 执行
0 0/3 * ? * Mon,Wed,Fri,Sun	周一、三、五、日，每隔三分钟执行一次
0 0/3 9-18 ? * Mon-Fri	周一到周五 09:00-18:00 之间每隔三分钟执行一次
0 0/30 * * * ?	每 30 分钟执行一次

3.6 调用函数

3.6.1 同步调用

同步调用指的是客户端触发函数后，需阻塞等待函数调用结果返回的场景。您也可以使用同步执行函数接口同步触发函数。同步调用场景下，函数最大运行时长限制为 15 分钟。

3.6.2 异步调用

异步调用指的是客户端触发函数后，FunctionGraph 持久化请求并立即返回，客户端不等待请求最终处理完成，用户无法实时感知请求处理结果。FunctionGraph 最终将异步请求排队，在服务端空闲的情况下逐个处理。如果您希望获取异步请求结果通知或者设置异步请求失败重试，请参见 3.3.7 配置函数异步。

- 以下触发器：默认异步调用，用户不可修改。

表 3-33 调用方式

事件源	调用方式

事件源	调用方式
定时器 TIMER	异步调用

- 您也可以使用异步执行函数 API 接口异步触发函数。异步调用场景下，函数最大运行时长限制为 12 小时（通过白名单配置）。

说明

如果函数执行端到端时延超过 90s，建议使用异步不使用同步，否则会因为网关限制，超过 90s 后无法收到同步响应。

3.6.3 重试机制

函数在同步调用或异步调用执行失败时，您可以参见以下重试机制进行操作。

- 同步调用

  同步调用执行失败，建议您自行尝试重试。

- 异步调用

  异步调用可在界面配置最大重试次数和消息最大有限期。函数平台会根据您配置的最大重试次数和消息最大有限期（最大有限期为 24 小时），进行重试。重试次数和配置的最大重试次数一致，重试有效期和配置的消息最大有效期一致。

3.7 监控

3.7.1 指标

3.7.1.1 监控信息说明

FunctionGraph 函数实现了与云监控服务的对接，用户无需任何配置，即可查询函数监控信息。

查看函数监控信息

FunctionGraph 会统计函数的运行时指标，显示的指标是函数运行时活动的聚合视图。要查看不同函数版本的指标，可在查看指标前切换函数版本，查询不同版本对应的监控信息。

1. 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。
2. 选择待配置的函数，单击进入函数详情页。
3. 选择“监控 > 指标”，选择时间粒度（最近 1 天、最近 3 天、自定义），查看函数运行状态。

说明

可以查看的指标有：调用次数、错误次数、运行时间（包括最大运行时间、最小运行时间、平均运行时间）、被拒绝次数、资源统计。

指标说明

运行监控指标说明如表 3-34 所示。

表 3-34 监控指标说明表

指标	单位	说明
调用次数	次	函数总的调用请求数，包含了错误和被拒绝的调用。异步调用在该请求实际被系统执行时才开始计数。
运行时间	毫秒	最大运行时间为某统计粒度（周期）下，即某一时间段内单次函数执行最大的运行时间。 最小运行时间为某统计粒度（周期）下，即某一时间段内单次函数执行最小的运行时间。 平均运行时间为某统计粒度（周期）下，即某一时间段内单次函数执行平均的运行时间。
错误次数	次	指发生异常请求的函数不能正确执行完并且返回200，都计入错误次数。函数自身的语法错误或自身执行错误也会计入该指标。
被拒绝次数	次	由于并发请求太多，系统流控而被拒绝的请求次数。
资源统计	个	该函数的请求并发数和预留实例数。

3.7.1.2 FunctionGraph 服务的监控指标参考

功能说明

本节定义了 FunctionGraph 上报云监控服务的监控指标的命名空间，监控指标列表和维度定义，用户可以通过云监控服务提供管理控制台或 API 接口来检索 FunctionGraph 产生的监控指标和告警信息。

命名空间

SYS.FunctionGraph

函数监控指标

表 3-35 FunctionGraph 支持的监控指标

指标 ID	指标名称	指标含义	取值范围	测量对象	监控周期（原始指标）
count	调用次数	该指标用于统计函	≥ 0	函数	1 分钟

指标 ID	指标名称	指标含义	取值范围	测量对象	监控周期（原始指标）
		数调用次数。 单位：次	counts		
failcount	错误次数	该指标用于统计函数调用错误次数。 以下两种情况都会记入错误次数： • 函数请求异常，导致无法执行完成且返回 200。 • 函数自身语法错误或者自身执行错误。 单位：次	≥ 0 counts	函数	1 分钟
rejectcount	被拒绝次数	该指标用于统计函数调用被拒绝次数。 被拒绝次数是指并发请求太多，系统流控而被拒绝的请求次数。 单位：次	≥ 0 counts	函数	1 分钟
concurrency	并发数	该指标用于统计函数同时调用处理的最大并发请求个数。 单位：个	≥ 0 counts	函数	1 分钟
reservedinstancenum	预留实例个数	该指标用于统计函数配置的预留实例个数。 单位：个	≥ 0 counts	函数	1 分钟
duration	平均运行时间	该指标用于统计函数调用平均运行时间。 单位：毫秒	≥ 0 ms	函数	1 分钟
maxDuration	最大运行时间	该指标用于统计函数调用最大运行时间。 单位：毫秒	≥ 0 ms	函数	1 分钟

指标 ID	指标名称	指标含义	取值范围	测量对象	监控周期（原始指标）
minDuration	最小运行时间	该指标用于统计函数最小运行时间。 单位：毫秒	≥ 0 ms	函数	1 分钟

表 3-36 函数流支持的监控指标

指标 ID	指标名称	指标含义	取值范围	测量对象	监控周期（原始指标）
count	调用次数	用于统计函数流调用次数。 单位：次	≥ 0 counts	函数流	1 分钟
fail_count	错误次数	该指标用于统计函数调用错误次数。 单位：次	≥ 0 counts	函数流	1 分钟
running_count	正在运行数量	该指标用于统计正在运行状态的函数流。 单位：个	≥ 0 counts	函数流	1 分钟
reject_count	被拒绝次数	该指标用于统计函数流调用被拒绝次数。 单位：个	≥ 0 counts	函数流	1 分钟
duration	平均运行时间	该指标用于统计函数流调用平均耗时。 单位：毫秒	≥ 0 ms	函数流	1 分钟

维度

key	value
package-functionname	应用名-函数名。 示例：default-myfunction_Python。
graph_name	函数流。

3.7.1.3 创建告警规则

函数及触发器创建以后，可以实时监控函数被调用及运行情况。

监控函数

不同版本函数的监控信息做了区分，查询函数指标之前设置函数版本，可以查询不同版本对应的监控信息。

操作步骤

函数实现与云监控服务的对接，函数上报云监控服务的监控指标，用户可以通过云监控服务来查看函数产生的监控指标和告警信息。

步骤 1 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。

步骤 2 单击函数名称，进入函数详情界面。

步骤 3 选择函数对应的版本或者别名，选择“监控 > 指标”。

步骤 4 单击“创建告警规则”，弹出“创建告警规则”对话框。

步骤 5 输入告警参数，单击“下一步”。如图 3-37 所示。

图 3-37 创建告警规则

步骤 6 输入告警规则名称，单击“确定”。

----结束

监控指标说明

告警监控指标如表 3-37 所示。

表 3-37 告警监控指标说明表

指标名称	显示名	描述	单位	上限值	下限值	建议阈值	值类型	所属维度
count	调用次数	该指标用于统计函数调用次数	次数	-	0	-	int	package-functionname
failcount	错误次数	该指标用于统计函数调用错误次数	次数	-	0	-	int	package-functionname
rejectcount	被拒绝次数	该指标用于统计函数调用被拒绝次数	次数	-	0	-	int	package-functionname
duration	平均运行时间	该指标用于统计函数调用平均运行时间	毫秒	-	0	-	int	package-functionname
maxDuration	最大运行时间	该指标用于统计函数调用最大运行时间	毫秒	-	0	-	int	package-functionname
minDuration	最小运行时间	该指标用于统计函数调用最小运行时间	毫秒	-	0	-	int	package-functionname

3.7.2 日志

3.7.2.1 查看函数日志

FunctionGraph 函数实现了与云日志服务的对接，用户无需任何配置，即可查询函数日志信息。

函数日志信息

在 FunctionGraph 函数控制台，可以通过以下两种方式查看函数日志。

- 在测试页签查看日志

  函数创建完成后，可以测试函数，在执行结果页，可以查看函数测试日志。操作步骤请参见 3.4 在线调试。

  此处最多显示 2KB 字节日志，如果日志太多，可以去函数详情页日志页签查询日志。

- 在日志页签查看日志

  在函数详情页“监控 > 日志”页签，查询日志信息，操作步骤请参见 3.7.2.2 管理函数日志。

3.7.2.2 管理函数日志

云日志服务（LTS）管理函数日志

FunctionGraph 支持开通云日志服务(LTS)，使用更丰富的函数日志管理功能。开通云日志服务后，FunctionGraph 会自动创建 1 个日志组，在这个日志组下会创建 20 个日志流，函数的日志会随机出现在某个日志流中，比如函数 A 第一次执行将日志存放在了日志流 A 中，那么以后都会固定在日志流 A 中，但是 1 个日志流中可能包含多个函数的日志。

说明

- 默认创建的 20 个日志流，您无法自定义。您可以在函数的“日志”页签下，单击“F12”，找到 query 接口里的日志流 ID，再到 lts 里找到对应的日志流 ID。

- 若在 LTS 控制台误删函数日志组，之前的日志数据不可找回，FunctionGraph 服务不感知该操作。此时您可以通过修改函数常规设置中的描述信息，保存后触发重建函数日志组。

步骤 1 设置查询条件。

- 请求列表：支持设置请求 ID、调用结果（执行成功、执行失败）、原因分析（初始化失败、加载失败、系统错误、调用超时、内存超限、磁盘超限、代码异常）
- 请求日志：支持关键字、请求 ID、实例 ID

表 3-38 调用结果

调用结果	说明
执行成功	函数执行成功打印的日志。
执行失败	函数执行失败打印的日志，包函调用超时、内存超限、磁盘超限、代码异常四种情况。 若想查看调用超时的日志信息，请将“日志类型”切换为调用超时，另外 3 种执行失败下的日志类型查看方法相同。

表 3-39 原因分析

原因分析	说明
初始化失败	函数初始化失败打印的日志。
加载失败	runtime 加载用户函数文件失败打印的日志
系统错误	内部错误。
调用超时	函数调用时间超过配置的“执行超时时间”打印的日志。
内存超限	函数内存大小超过配置的“内存”大小打印的日志。
磁盘超限	磁盘超出限制大小打印的日志。
代码异常	代码出现异常情况打印的日志。

说明

- 支持的时间条件：最近 1 小时、最近 1 天、最近 3 天及自定义。
- 您可以单击“到 LTS 进行日志分析等更多操作”，前往 LTS 控制台管理函数日志。

----结束

3.8 函数管理

概述

函数是实现某一功能所需代码、运行时、资源、设置的组合，是可以独立运行的最小单元。函数通过 Trigger 触发，自行调度所需资源及环境，实现预期功能。

导出函数

FunctionGraph 支持将已创建的函数导出。

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

步骤 2 在“函数”页面，单击函数名称，进入函数详情页面。

步骤 3 在函数详情页面，选择函数版本，单击操作列表中的“导出函数”，即可将该函数导出。

说明

- 同一时段单个用户只能并发导出一个函数。
- 导出函数资源包大小 50MB 以内。

- 导出的函数资源名称为函数名+函数代码的 MD5 值.zip。
- 导出的函数资源中配置信息不包含别名信息。

----结束

禁用函数

用户可以根据实际情况将函数禁用，禁用期间函数不能执行。

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

步骤 2 在“函数列表”，单击要禁用的函数名称，进入“函数详情”页面。

步骤 3 单击“禁用函数”。

步骤 4 单击“确定”，函数被禁用。

说明

- 只能禁用“latest”版本的函数，不能禁用已经发布的版本的函数。
- 基于已禁用的“latest”版本重新发布新版本，发布后的新版本也处于禁用状态且不能启用。
- 当函数处于禁用状态时可以修改代码，不能执行函数。

----结束

启用函数

用户可以根据实际情况将已禁用的函数重新启用。

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

步骤 2 在“函数列表”，单击要启用的函数名称，进入“函数详情”页面。

步骤 3 单击“启用函数”，函数被启用。

----结束

删除函数

对于已经不再使用的函数，可以进行删除操作，及时释放资源。

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

步骤 2 在“函数列表”单击要删除的函数名称，进入“函数详情”页面。

步骤 3 在右上方选择搜索字段（可选项：函数名称、运行时语言、应用名称），输入搜索关键字，单击“ ”，搜索待删除函数。

步骤 4 选择待删除函数，单击操作栏的“删除”，弹出“删除函数”页。

步骤 5 在“删除函数”页，单击“确定”，完成函数删除。

----结束

3.9 依赖包管理

概述

函数代码一般包含公共库和业务逻辑两部分。对于公共库，您可以打包成依赖包单独管理，共享给多个函数使用，同时也减少了函数代码包部署、更新时的体积。

FunctionGraph 也提供了一些公共依赖包，公共依赖包在平台内部缓存，消除了冷启动加载的影响，推荐您优先使用。

依赖包管理模块统一管理用户所有的依赖包，用户可以通过本地上传和 obs 地址的形式上传依赖包，并为依赖包命名。同时支持用户针对同一依赖包进行迭代更新，即同一依赖包可拥有多个版本，便于用户对依赖包进行系统化管理。

函数依赖包生成示例请参见如何制作函数依赖包。

说明

- 依赖包内文件名不能以~结尾。
- 依赖包当前文件限制数为 30000。
- 如果函数配置了私有依赖包且依赖包很大的话，建议在函数详情页的“设置 > 常规设置”重新设置函数执行时间，在原基础上增加超时时间。

创建依赖包

步骤 1 登录 FunctionGraph 控制台，在左侧导航栏选择“函数 > 依赖包管理”，进入“依赖包管理”界面。

步骤 2 单击的“创建依赖包”，弹出“创建依赖包”对话框。

步骤 3 设置以下信息。

表 3-40 依赖包配置参数说明

参数	说明
依赖包名称	自定义的依赖包名称，用于识别不同的依赖包。
代码上传方式	分为上传 ZIP 文件和从 OBS 上传文件。 • 上传 ZIP 文件：需单击“添加文件”，上传 ZIP 文件。 • OBS 链接 URL：需填写“OBS 链接 URL”。
运行时语言	选择运行时语言。
描述	对于依赖包的描述信息，可以不填。

步骤 4 单击“确定”，完成依赖包的创建。默认首次创建的依赖包版本为“1”。

步骤 5 单击列表中的依赖包名称，进入版本历史界面，可以查看当前依赖包下的所有版本和版本相关信息。当前支持针对同一依赖包，进行不同版本的系统化管理。

- 单击“创建版本”，填写相关信息，可以创建新的依赖包版本。
- 单击具体的版本号，可以查看版本地址。
- 单击版本号所在行的删除，可以删除该版本。

----结束

配置函数依赖

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

步骤 2 单击函数名称，进入函数详情界面。

步骤 3 在“代码”页签，单击“代码依赖包”所在行的“添加依赖包”，弹出“选择依赖包”对话框。

步骤 4 选择依赖包，单击“确定”。

表 3-41 依赖包配置说明

参数	说明
运行时语言	默认展示当前函数的运行时语言，无法修改。
依赖包源	根据实际业务，选择“公共依赖包”或“私有依赖包”。
依赖包名称	选择当前运行时语言下的依赖包。
版本	选择当前依赖包的具体版本。

说明

- 一个函数最多可添加 20 个依赖包。
- 除了您自行创建的依赖包（私有依赖包）以外，FunctionGraph 还提供了一些常见的公共依赖包，您可以直接选择使用。

----结束

删除依赖包

依赖包当前无法在界面直接删除，若需删除，请删除依赖包下的所有版本。当所有版本全部删除完成后，依赖包会自动删除。

步骤 1 登录 FunctionGraph 控制台，在左侧导航栏选择“函数 > 依赖包管理”，进入“依赖包管理”界面。

步骤 2 单击依赖包名称，进入版本历史管理界面。

步骤 3 单击版本号所在行的删除，可以删除该版本，存在多个版本请重复此操作。

图 3-38 删除依赖包版本

说明

如果函数正在使用此依赖包，则无法删除。

----结束

引入依赖库

支持的依赖库说明

FunctionGraph 支持引入标准库及第三方依赖库。

- 标准库

  对于标准库，无论是在线编辑或是线下开发打包上传至 FunctionGraph，均可以直接在代码中引入，使用其功能。

- FunctionGraph 支持的非标准库

  FunctionGraph 内置一些三方件，如表 3-42、表 3-43 所示。像标准库一样，在编写代码时直接引入，使用其功能。

表 3-42 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

表 3-43 Python Runtime 支持的非标准库

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	访问云 smn 服务	1.0.1

- 其他第三方库（除了上面表格列举的非标准三方库，FunctionGraph 没有内置别的非标准三方库）

  将依赖的第三方库打包，上传至 OBS 桶，在创建函数时配置依赖包的 OBS 存储地址，在函数代码中即可使用其功能。

引入依赖库示例

处理图片的函数代码如下。

```
# -*- coding: utf-8 -*-
from PIL import Image, ImageEnhance

from com.obs.client.obs_client import ObsClient

import sys
import os

current_file_path = os.path.dirname(os.path.realpath(__file__))
# append current path to search paths, so that we can import some third
party libraries.
sys.path.append(current_file_path)
region = 'your region'
obs_server = 'obs.xxxxxxcloud.com'
def newObsClient(context):
    ak = context.getAccessKey()
    sk = context.getSecretKey()
    return ObsClient(access_key_id=ak, secret_access_key=sk,
server=obs_server,
                     path_style=True, region=region, ssl_verify=False,
max_retry_count=5, timeout=20)
def downloadFile(obsClient, bucket, objName, localFile):
```

```
    resp = obsClient.getObject(bucket, objName, localFile)
    if resp.status < 300:
        print 'download file', file, 'succeed'
    else:
        print('download failed, errorCode: %s, errorMessage: %s,
requestId: %s' % resp.errorCode, resp.errorMessage,
              resp.requestId)
def uploadFileToObs(client, bucket, objName, file):
    resp = client.putFile(bucket, objName, file)
    if resp.status < 300:
        print 'upload file', file, 'succeed'
    else:
        print('upload failed, errorCode: %s, errorMessage: %s,
requestId: %s' % resp.errorCode, resp.errorMessage,
              resp.requestId)
def getObjInfoFromObsEvent(event):
    s3 = event['Records'][0]['s3']
    eventName = event['Records'][0]['eventName']
    bucket = s3['bucket']['name']
    objName = s3['object']['key']
    print "*** obsEventName: %s, srcBucketName: %s, objName: %s",
eventName, bucket, objName
    return bucket, objName
def set_opacity(im, opacity):
    """设置透明度"""
    if im.mode != "RGBA":
        im = im.convert('RGBA')
    else:
        im = im.copy()
    alpha = im.split()[3]
    alpha = ImageEnhance.Brightness(alpha).enhance(opacity)
    im.putalpha(alpha)
    return im
def watermark(im, mark, opacity=0.6):
    """添加水印"""
    try:
        if opacity < 1:
            mark = set_opacity(mark, opacity)
        if im.mode != 'RGBA':
            im = im.convert('RGBA')
        if im.size[0] < mark.size[0] or im.size[1] < mark.size[1]:
            print "The mark image size is larger size than original image
file."
```

```
            return False
        x = (im.size[0] - mark.size[0]) / 2
        y = (im.size[1] - mark.size[1]) / 2
        layer = Image.new('RGBA', im.size, )
        layer.paste(mark, (x, y))
        return Image.composite(layer, im, layer)
    except Exception as e:
        print ">>>>>>>>>>> WaterMark EXCEPTION:  " + str(e)
        return False
def watermark_image(localFile, fileName):
    im = Image.open(localFile)
    watermark_image_path = os.path.join(current_file_path, "watermark.png")
    mark = Image.open(watermark_image_path)
    out = watermark(im, mark)
    print "**********finish water mark"
    name = fileName.split('.')
    outFileName = name[0] + '-watermark.' + name[1]
    outFilePath = "/tmp/" + outFileName
    if out:
        out = out.convert('RGB')
        out.save(outFilePath)
    else:
        print "Sorry, Save watermarked file Failed."
    return outFileName, outFilePath
def handler(event, context):
    srcBucket, srcObjName = getObjInfoFromObsEvent(event)
    outputBucket = context.getUserData('obs_output_bucket')
    client = newObsClient(context)
    # download file uploaded by user from obs
    localFile = "/tmp/" + srcObjName
    downloadFile(client, srcBucket, srcObjName, localFile)
    outFileName, outFile = watermark_image(localFile, srcObjName)
    # 将转换后的文件上传到新的obs桶中
    uploadFileToObs(client, outputBucket, outFileName, outFile)
    return 'OK'
```

对于标准库和 FunctionGraph 支持的非标准库，可以直接引入。

对于 FunctionGraph 暂没有内置的非标准三方库，通过以下步骤引入。

1. 将依赖的库文件压缩成 ZIP 包，上传至 OBS 存储桶，获得依赖包的 OBS 存储链接。
2. 登录 FunctionGraph 控制台，在左侧导航栏选择“函数 > 依赖包管理”，进入“依赖包管理”界面。
3. 选择“创建依赖包”，弹出“创建依赖包”对话框。

4. 输入依赖包名称、运行时语言和 OBS 存储链接，单击“确定”。

图 3-39 设置依赖包

5. 进入函数详情页面，在“代码”页签，单击“依赖代码包”所在行的“添加依赖包”，选择 4 中创建的依赖包，单击“确定”。

图 3-40 添加依赖包

6. 单击“保存”，完成函数的修改。

警告

各个依赖包和代码包之间尽量不要有相同的目录或文件，比如依赖包 depends.zip，里面有 index.py 这个文件，如果代码采用在线编辑方式，函数执行入口为 index.handler，这样在函数执行的时候会产生一个代码文件 index.py，跟依赖包里面的 index.py 文件同名，两个文件可能会因覆盖合并而出错。

3.10 预留实例管理

概述

函数工作流提供了按量和预留两种类型的实例。

- 按量实例是由函数工作流根据用户使用函数的实际情况来创建和释放，当函数工作流收到函数的调用请求时，自动为此请求分配执行环境。
- 预留实例是将函数实例的创建和释放交由用户管理，当您为某一函数创建了预留实例，函数工作流收到此函数的调用请求时，会优先将请求转发给您的预留实例，当请求的峰值超过预留实例处理能力时，剩余部分的请求将会转发给按量实例，由函数工作流自动为您分配执行环境。

  预留实例在创建完成后，会自动加载该函数的代码、依赖包以及执行初始化入口函数，且预留实例会常驻环境，消除冷启动对业务的影响。

  预留实例当前支持配置固定数量的预留实例，也支持配置定时伸缩的预留实例。

说明

用户默认没有权限使用预留实例，如果需要使用预留实例功能，请在工单系统提交工单添加白名单。

配置固定数量的预留实例

直接创建固定个数的预留实例前，确保 FunctionGraph 控制台已存在需要创建预留实例的目标函数。

1. 登录函数工作流控制台，在左侧的导航栏选择“函数 > 函数列表”。
2. 选择待配置的函数，单击进入配置详情页。
3. 选择“设置 > 并发”，单击“添加”，开始配置。

   图 3-41 单击“添加”

4. 参见表 3-44，填写参数。

   您可以给函数对应的版本或者别名创建指定数量的预留实例，其中预留实例的数量不能超过并发实例数配额和单函数最大实例数。

图 3-42 基础配置

表 3-44 基础配置说明

参数	说明
函数名称	展示当前配置预留实例的函数的名称。
类型	根据实际业务情况，选择“版本”或“别名”。
选择版本	仅当类型选择“版本”时，需设置此参数。
选择别名	仅当类型选择“别名”时，需设置此参数。
最小实例数	设置最小实例数，输入值不能超过 1000。配置最小实例数后，函数工作流会为您创建固定数目的函数实例，并且在您将最小实例数设置为 0 之前预留实例会持续运行。
闲置模式	开启此参数，表示预留实例在无调用的时候暂停 CPU，节省资源，降低费用成本。

说明

别名和对应的版本不可以同时配置预留实例。比如，latest 版本对应的别名为 1.0，在 latest 版本下进行了预留实例配置，则在别名 1.0 下不能再进行预留实例配置，反之同理。

5. 配置完成后，单击“确定”，在“预留实例策略配置”列表展示已添加的“策略配置”。

图 3-43 列表展示

配置定时伸缩的预留实例

用户配置预留实例时，能够配置指定的时间段、cron 表达式及其对应的预留实例数量。函数服务能够在该时间段中，根据 cron 表达式更新预留实例的数量，如果时间段超过了该时间段，则将预留实例数量调整到配置的固定值的预留实例数量。

1. 参见表 3-44 进行基础配置，完成后单击“添加策略”，进行弹性预留策略配置。

图 3-44 添加策略

2. 参见表 3-45，填写参数。

表 3-45 弹性策略配置说明

参数	说明
策略名称	自定义策略名称。
Cron 表达式(UTC)	您可以参见 3.5.4 附录：函数定时触发器 Cron 表达式规则，填写此参数。
生效时间	生效时间为本地时间，即 cron 表达式的生效时间窗。 只有当时间在时间窗内时，该弹性策略才会生效，当该函数的所有弹性策略的生效时间窗都不生效时，那么预留实例数就会还原到基础配

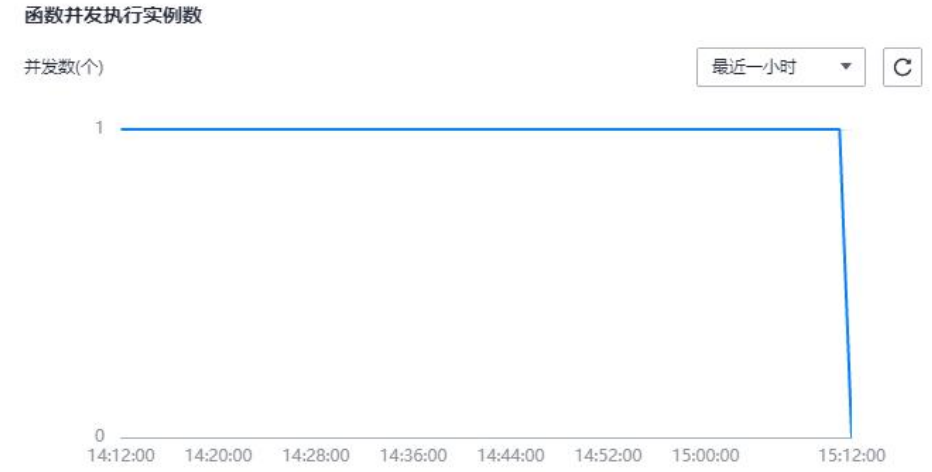
参数	说明
	置中的最小实例数。
最小实例数	需要创建的预留实例数。 根据实际业务场景，填写当前策略生效时创建的预留实例的个数。 说明 最小实例数必须大于或等于基础配置里的最小实例数。

3. 配置完成后，单击“确定”，在“预留实例策略配置”列表展示已添加的“策略配置”。

图 3-45 列表展示

4. 单击“操作 > 编辑”，修改弹性策略信息、添加策略。
5. 单击“操作 > 删除”，删除版本或别名下的预留实例策略。
6. 预留实例将根据添加的弹性策略配置执行，您可以在“预留实例策略配置”列表，单击“限定符”，选择“弹性策略名称”，查看函数并发执行实例数。

图 3-46 查看并发执行实例数



3.11 函数流管理

3.11.1 函数流简介

函数流是一个面向无服务器计算领域，编排无服务器分布式应用的工作流服务。基于该服务，用户可以通过 Low Code 以及可视化的方式将多个独立的无服务器函数用顺序、分支、并行等方式轻松编排成一个完整的应用，并提供监控和管理平台，用于诊断和调试应用。

本章节主要介绍函数流组件、组件编排规则、表达式运算符和配置示例。

组件说明

函数流提供多种类型的组件，用户可以通过拖拽组件、配置组件和连接组件进行可视化编排，实现函数任务流的编排。使用函数流功能，请先了解表 3-46。

表 3-46 组件说明

类型	名称	说明
函数组件	函数	FunctionGraph 函数，如何创建函数请参见 3.2.2 使用空白模板创建函数。
流程控制器	子流程	把已创建的“函数流”任务作为“子流程”组合成一个新的函数流任务。
	并行分支	用于创建多个并行分支的控制器，以便同时执行多个分支任务，并可根据分支执行结束后控制下一步流程。
	开始节点	只能加入触发器，用于标识流程的开始，一个流程只能有一个开始节点。
	异常处理	用于控制函数执行失败后的下一步流程。
	循环节点	用于对数组中每个元素进行循环处理。每次循环会执行一次循环内部的子流程。
	时间等待	用于控制当前流程在指定时间延迟后再调用下一个流程。
	服务节点	用于对多个函数构成的复杂操作进行抽象，可以将多个函数操作合并成一个原子节点进行管理。
	条件分支	用于根据条件判断是否执行下一分支。
	结束节点	用于标识流程的结束。

编排规则

- 设计的函数流必须是一个有向无环图，从开始节点出发，开始节点后续必须且只能连接一个节点（除了异常处理和结束节点）；流程必须在某一个节点结束，结束流程有两种形式：

a. 流程中存在的节点没有任何后继节点，且后续节点非条件分支，并行分支或开始节点。

b. 流程中存在结束节点，且结束节点后续无其他节点。

- 组件设计规则

表 3-47 触发器和函数

参数	说明	创建函数流时，是否必选
触发器	• 当前允许流程中配置 0-10 个触发器。 • 触发器必须配置在开始节点内。 • 触发器不允许连接其他任何节点，也不允许被其他节点连接。	否
函数	• 当前允许流程中配置 0-99 个函数节点。 • 当函数连接异常处理节点时，最多可以再连接一个非开始节点和非异常处理节点。 • 当函数不连接异常处理节点时，只能连接一个非开始节点。	否

表 3-48 流程控制器

参数	说明	创建函数流时，是否必选
子流程	该节点选择已创建的函数流任务。	否
并行分支	• 用于标识节点后面的分支会并行执行。 • 后继节点允许连接 1-20 个节点（除了异常处理，开始节点和结束节点），至少连接一个节点。	否
开始节点	• 用于标识流程开始，每个流程必须有且只能有一个开始节点。 • 开始节点后面必须接 1 个节点，后续节点类型不能是结束节点或者异常处理。	必选
异常处理	后面可以接 0-10 个节点，后继节点不能是开始节点，结束节点和异常处理节点。	否
循环节点	用来对数组中每个元素进行循环处理。每次循环会执行一次循环内部的子流程。 循环节点内部子流程需要满足如下规则： 1. 只能有一个起始节点（没有前驱节点），起	否

参数	说明	创建函数流时，是否必选
	始节点只能使用函数，时间等待节点。 2. 循环节点内部只允许编排函数，时间等待，异常处理节点。	
时间等待	后面可以连接 0 个或 1 个节点，节点类型不能是开始节点和异常处理节点。	否
服务节点	服务节点由多个函数节点组成，后续节点可以是结束节点或异常处理节点。	否
条件分支	后面可以连接 2-20 个后继节点，后继节点类型不能为开始节点，结束节点和异常处理节点。	否
结束节点	后面不能接任何节点。	否

表达式运算符说明

异常处理和条件分支的表达式的结构为 [JsonPath] + [逻辑运算符] + [对比数据]，简单示例：$.age >= 20

JsonPath 说明

Operator	Supported	Description
$	Y	执行查询的 root，所有正则表达式由此启动。
@	Y	过滤正在处理的当前位置。
.	Y	子节点。
[(,)]	Y	数组索引。
[start:end]	Y	数组切片运算符。
[?()]	Y	过滤表达式。 表达式必须计算为布尔值。

参见示例

- 简单取值：JSON 数据样例

```
{
    "fruits": [ "apple", "orange", "pear" ],
    "vegetables": [{
    "veggieName": "potato",
    "veggieLike": true
     },
```

```
    {
       "veggieName": "broccoli",
       "veggieLike": false
    }]
}
```

$.fruits 表达式含义：取出 fruits 下对应的所有 value。

$.fruits 解析结果：["apple","orange","pear"]

- 简单过滤：JSON 数据样例

```
{
     "fruits": [ "apple", "orange", "pear" ],
     "vegetables": [{
     "veggieName": "potato",
     "veggieLike": true
     },
    {
       "veggieName": "broccoli",
       "veggieLike": false
    }]
}
```

表达式：$.vegetables[?(@.veggieLike == true)].veggieName

表达式含义：取出 key 值 vegetables 对应的所有 value，并根据过滤条件输出 veggieLike 为 True 的 veggieName。

```
取值结果：[potato]
```

逻辑运算符说明

使用以下数据作为例子中的输入参数：

```
{
  "name" : "apple",
  "weight": 13.4,
  "type": [3,4,6,8],
  "obj": {
       "a" : 1
   }
}
```

支持的运算符如下：

符号	作用	例子	返回值	备注
==	相等	$.name == 'apple'	true	支持的数据类型包括：int,float,string,bool,nil
!=	不等	$.name != 'apple'	false	支持的数据类型包括：int,float,string,bool,nil
<	小于	$.weight < 12	false	只支持数字类型

符号	作用	例子	返回值	备注
>	大于	$.weight > 12	true	只支持数字类型
<=	小于等于	$.weight <= 13.4	true	只支持数字类型
>=	大于等于	$.weight >= 13.4	true	只支持数字类型
'*'	通配符	$.weight == '*'	true	只支持在==比较中使用
||	或	$.name == 'apple' || $.weight < 12	true	支持使用（）的复杂与或逻辑
&&	且	$.name == 'apple' && $.weight < 12	false	支持使用（）的复杂与或逻辑

说明

- 字符串格式常量需要使用‘’包含，例如：‘apple’
- jsonpath 表达式中不能出现上述保留字符'=', '!=', '<', '>', '|', '&'

配置示例

示例一

1. 新建三个函数， Runtime 均使用 python 3.9，代码功能及内容如下所示 。
   - 函数 1：函数执行返回 result 的值为函数调用事件内的 input 输入值

```
import json
def handler (event, context):
    input = event.get('input',0)
    return {
        "result": input
        }
```

   - 函数 2：函数执行返回 result 的值为函数调用事件内的 input 输入值+2 的结果值

```
import json
def handler (event, context):
    input = event.get('input',0)
    return {
        "result": input+2
        }
```

   - 函数 3：函数执行返回 result 的值为函数调用事件内的 input 输入值平方的结果值

```
import json
def handler (event, context):
    input = event.get('input',0)
    return {
        "result": input*input
        }
```

2. 在函数流编排区域拖拽组件，“并行分支”节点配置如下。

图 3-47 并行分支节点配置

3. 函数节点配置如表 3-47 所示，在 3 个函数节点均需配置

图 3-48 函数节点配置

4. 保存函数流，启动执行时，定义输入值如下所示。

```
{
  "input":3
}
```

5. 单击函数流任务名称，查看执行结果。

图 3-49 执行结果

示例二

1. 在函数编排区域编排与拖拽组件，服务节点选择“串行模式”。

图 3-50 服务节点配置

2. 函数节点分别选择示例一中创建的函数 2、函数 3，函数流的配置如下。

图 3-51 函数 2 配置

图 3-52 函数 3 配置

3. 保存函数流，启动执行时，定义输入值如下所示。

```
{
  "input":3
}
```

4. 单击函数流任务名称，查看执行结果。

图 3-53 执行结果

示例三

1. 新建两个函数， Runtime 均使用 Python 3.9，代码内容相同。

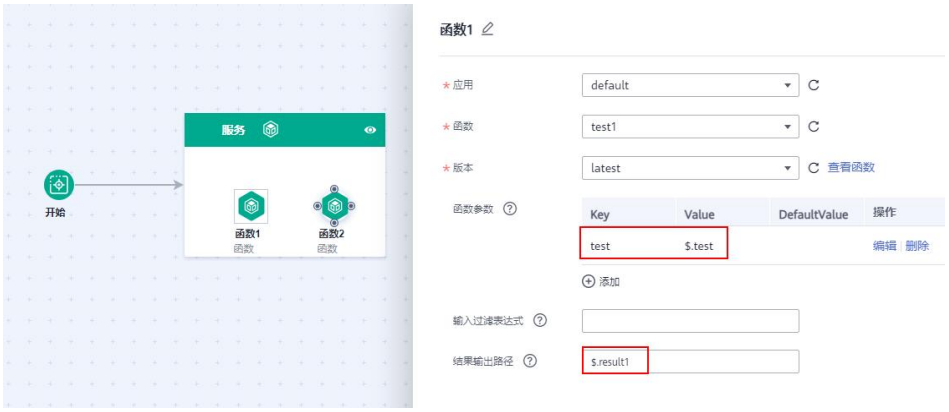
```
import json
def handler (event, context):
    print(event)
    return {"result":"success"}
```

2. 在函数流编排区域拖拽组件，服务节点的配置如下。

图 3-54 服务节点配置

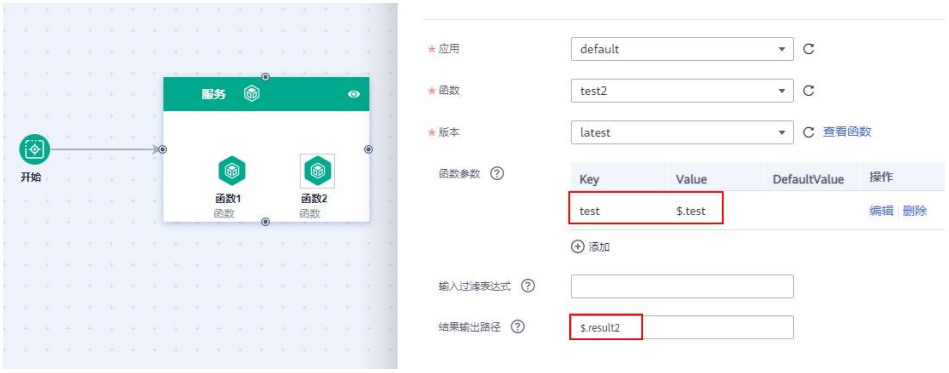
3. 函数 1 节点配置如下。

图 3-55 函数 1 节点配置



4. 函数 2 节点配置如下。

图 3-56 函数 2 节点配置



5. 保存函数流，启动执行时，定义输入值如下所示。

```
{
  "test":123
}
```

6. 单击函数流任务名称，查看执行结果，可以看到两个函数执行结果合并，分别为 result1 和 result2。

图 3-57 执行结果

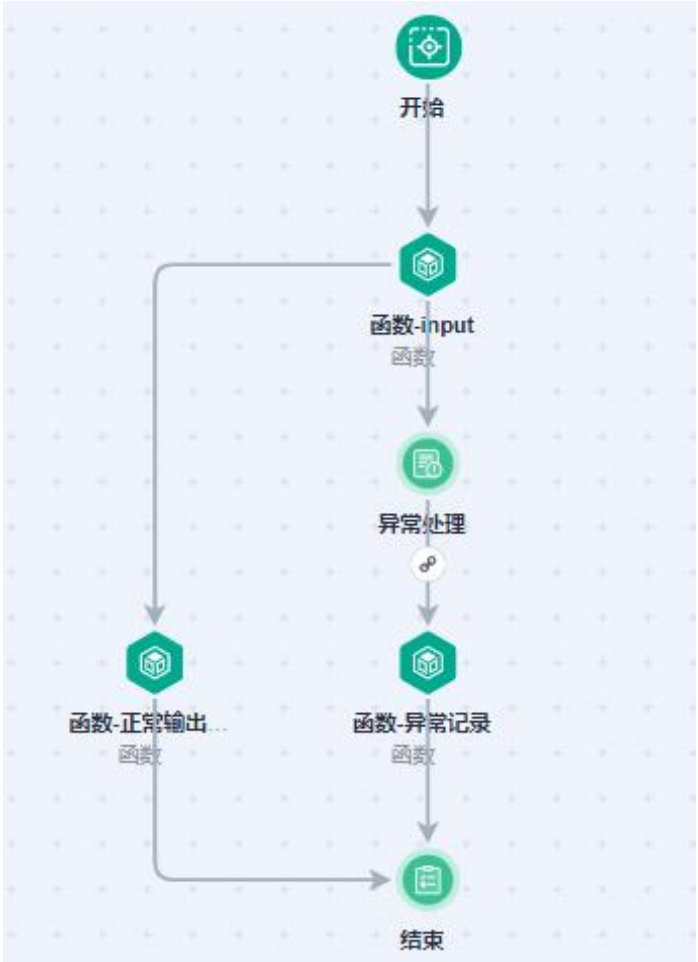
说明

如果两个函数执行返回的输出值结构一致，会导致函数执行结果被覆盖。

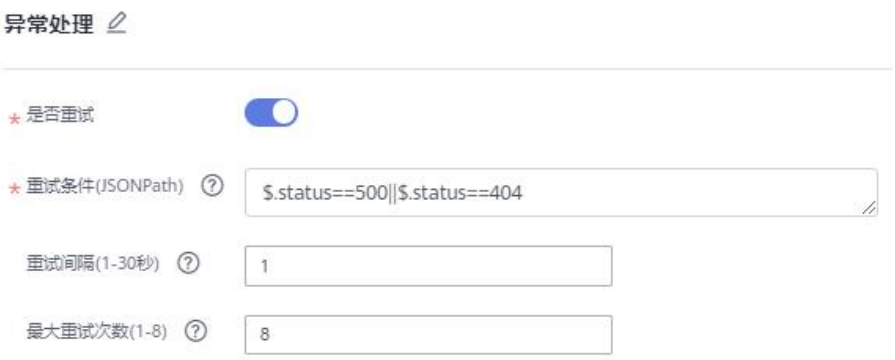
示例 4

当函数流里面的函数执行异常时，可以通过“异常处理”来处理执行失败的函数并可添加重试。函数执行失败可分为多少情况：函数执行异常；函数内部业务失败并在返回内容中添加了错误码，例如 status，200 代表成功，500 和 404 等代表失败。

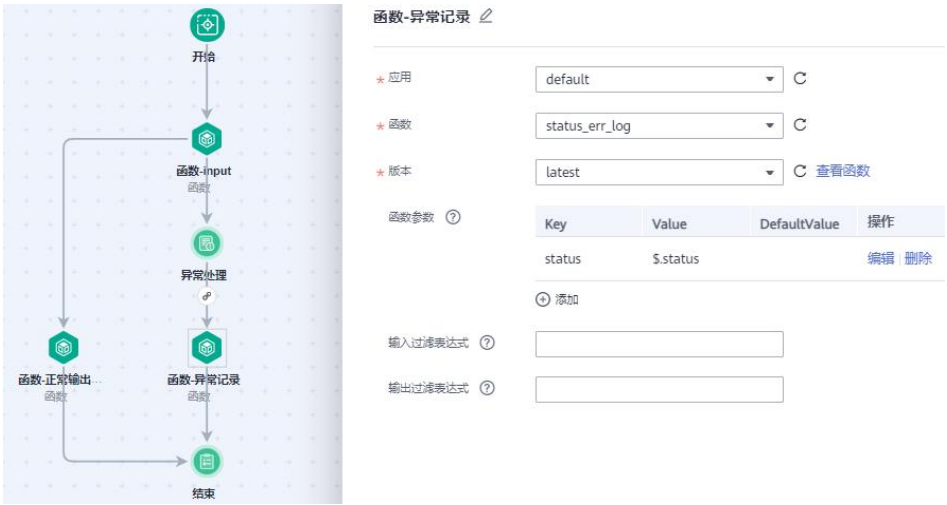
1. 在函数流编排区域拖拽组件，各节点功能如下。
   - 函数-input：从 event 取出 input 输入值，作为函数返回 status 的输出值；
   - 异常处理：开启重试机制，当函数返回的 status 为 500 或 404 时进行重试，重试间隔 1s，最大重试次数 8 次；
   - 函数-异常记录：当经过 8 次重试函数返回的 status 依旧为 500 或 404 时，进行异常记录；
   - 函数-正常输出：如果“函数-input”返回的 status 不为 500 或 404 时，执行“函数-正常输出”。



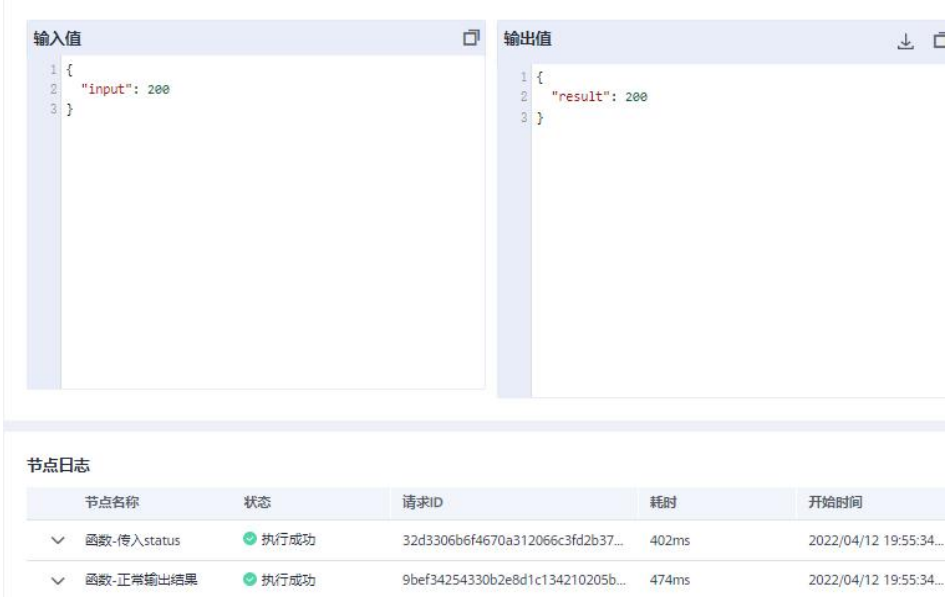
2. 配置异常处理，重试条件：$.status==500||$.status==404。



3. 添加重试之后依旧失败的处理逻辑即“函数-异常记录”。



4. 输入 200 执行成功。



5. 输入 500，异常处理逻辑进行重试和记录。

3.11.2 创建函数流任务

本章节主要介绍如何创建函数流任务和编排函数流任务。您可以根据实际业务场景来创建标准函数流或快速函数流。

- 标准模式面向普通的业务场景，支持长时间任务，支持执行历史持久化和查询，只支持异步调用，在函数流运行记录页面查询执行结果。
- 快速模式面向业务执行时长较短，需要极致性能的场景，只支持流程执行时长低于 5 分钟的场景，不支持执行历史持久化，支持同步和异步调用。通过同步执行函数流接口进行函数流的同步执行，接口直接返回函数流执行结果，同时日志页面查看上报到 LTS 的函数流执行日志。

说明

快速函数流限时免费，欢迎体验！

前提条件

- 已经在 FunctionGraph 控制台创建函数，创建过程请参见 3.2.2 使用空白模板创建函数。

操作步骤

步骤 1 登录 FunctionGraph 控制台，进入“函数流”页面。

步骤 2 在“函数流”页面，单击“创建标准函数流”或“创建快速函数流”，进入新建函数流页面。

步骤 3 编排函数流任务，请您根据实际应用进行函数流编排。

1. 在函数流页面，通过拖拽组件进行流程编排。

   以图 3-58 为例，将开始节点、函数、结束节点拖入编辑框内，并用连接线连接好。

   图 3-58 编排函数流

2. 分别单击编辑框中的每个节点进行编辑。配置函数参数，参数说明如表 3-49 所示，带*参数为必填项。

   说明

   配置函数前确保已创建好函数，示例中的函数节点选择函数 2（函数执行返回 result 的值为函数调用事件内的 input 输入值+2 的结果值），参见图 3-59 配置。

   图 3-59 函数节点配置

表 3-49 函数参数说明

参数	说明
*应用	函数所属应用，用户创建函数时可以进行分组，每个函数应用下面可以创建多个函数，在函数创建时可以指定其归属于某个函数应用。
*函数	FunctionGraph 中对应的函数。 说明 仅对于 Go 函数支持返回流式数据：在函数详情页的“设置 > 高级设置”下，打开“返回流式数据”开关即可。
*版本	FunctionGraph 中函数对应的版本。
函数参数	流程中以 json 格式作为 body 参数在执行时传入函数。 Key：填写参数 Value:填写参数值 DefaultValue：设置默认值，参数未获取到值时，默认获取默认值 操作：编辑或删除设置的参数
输入过滤表达式(JSONPath)	基于上一个流程的 json 输出参数，可以使用 JSONPath 格式来选择性的过滤出当前流程的输入参数。
输出过滤表达式(JSONPath)	基于当前流程的 json 输出参数，可以使用 JSONPath 格式来选择性的过滤出下一流程的输出参数。

3. 若您的函数流任务中配置了流程控制器，请参见表 3-50 进行配置，带*参数为必填项。

表 3-50 流程控制器参数说明

类型	参数	说明
子流程	选择子流程	选择已创建的函数流任务。
	是否等待子流程完成	默认选择“是”。
	输入过滤表达式(JSONPath)	基于上一个流程的 json 输出参数，可以使用 JSONPath 格式来选择性的过滤出当前流程的输入参数。
	输出过滤表达式(JSONPath)	基于当前流程的 json 输出参数，可以使用 JSONPath 格式来选择性的过滤出下一流程的输出参数。

类型	参数	说明
并行分支	*分支执行完成条件	• 所有分子执行完成：2 个或 2 个以上分支时选择该条件 • 一个分支执行完成：只有 1 个分支时选择该条件 • 指定数目分支执行完成：2 个或 2 个以上分支时其中某个分支可以选择该条件
	输入过滤表达式(JSONPath)	基于上一个流程的 json 输出参数，可以使用 JSONPath 格式来选择性的过滤出当前流程的输入参数。
	输出过滤表达式(JSONPath)	基于当前流程的 json 输出参数，可以使用 JSONPath 格式来选择性的过滤出下一流程的输出参数。
	指定分支执行完成数目	当“分支执行完成条件”选择指定数目分支执行完成时，支持自定义执行完成的数目。
	*结果输出路径	输入并行分支执行结果输出位置，输入值作为 key，并行分支执行结果作为 value，以 json 形式输出。若未填写，默认输出路径为：result。
开始节点	加入触发器	用于标识流程的开始，一个流程只能有一个开始节点。如何创建函数流触发器，请参见 3.11.4 创建函数流触发器。
异常处理	*是否重试	默认关闭，开启后可以控制函数执行失败后的下一步流程。 • 重试条件(JSONPath)：例如: $.status == 500 • 重试间隔(1-30 秒)：默认重试间隔 1S • 最大重试次数(1-8)：默认重试次数 3 次
循环节点	*遍历数组路径（JSONPath）	需要遍历的数组类型变量地址。
	*迭代变量名称	每次循环迭代，引用数组中元素的参数名称。
	*结果输出路径（JSONPath）	指定全部迭代分支执行结果数组的输出位置。
	并发迭代数目	并发运行迭代分支的数目，限制 0-100, 0 代表并发拉起的数目无限制。

类型	参数	说明
	并发迭代时间间隔（秒）	每次迭代间隔的时间。
	输入过滤表达式（JSONPath）	基于上一个流程的 json 输出参数，可以使用 JSONPath 格式来选择性的过滤出当前流程的输入参数。
	输出过滤表达式（JSONPath）	基于当前流程的 json 输出参数，可以使用 JSONPath 格式来选择性的过滤出下一流程的输出参数。
时间等待	*延迟时间（秒）	默认 1000 秒。
服务节点	执行模式	定义服务节点中函数的执行顺序。 • 串行模式：服务中的函数节点按照连线顺序依次执行，可以严格保证函数的执行顺序 • 并行模式：服务中的函数节点并行执行，不保证内部函数节点的执行顺序
	输入过滤表达式	通过 JSONPath 表达式对节点的输入信息进行过滤。
	输出过滤表达式	通过 JSONPath 表达式对节点的输出信息进行过滤。
条件分支	*分支类型	• 条件分支 • 默认分支 当一个分支选择条件分支时，必须要有一个分支选择默认分支。
	表达式	选择“条件分支”，需要输入 JSONPath 类型表达式。
	输入过滤表达式	通过 JSONPath 表达式对节点的输入信息进行过滤。
	输出过滤表达式	通过 JSONPath 表达式对节点的输出信息进行过滤。
结束节点	流程结束的标志	后面不能接任何节点。

4. 流程中的所有节点参数配置完成后，单击右上角的“保存”。

说明

函数流中的节点改动后，必须先保存信息，再启动函数流任务。

5. 在新建函数流页面，填写相关信息，单击“确定”，函数流保存成功。

表 3-51 输入配置信息

参数	说明
*名称	输入函数流名称。
*企业项目	选择企业项目。
日志记录	创建快速函数流，保存时需要选择此参数。 • ALL：为所有事件启用日志记录 • ERROR：仅启用错误日志记录 • NONE：关闭日志记录
合并参数	将上一个节点的输出与下一个节点的输入合并为输入。
描述	输入函数流的简要描述。

图 3-60 新建标准函数流

6. 单击“启动”，在弹出的启动执行页面，支持输入定义值或者直接启动。此处选择“输入定义值”。

```
{
  "input":3
}
```

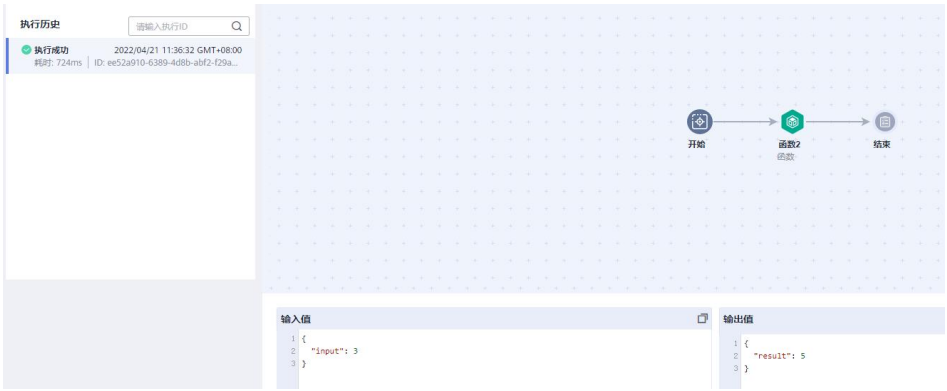
图 3-61 启动执行配置

说明

输入定义值必须是 JSON 格式的内容。

7. 单击“开始执行”，页面右上角提示“启动函数流 xxx 成功”。
8. 单击函数流任务名称，进入函数流任务详情，查看函数流执行结果。

图 3-62 查看执行结果



----结束

3.11.3 函数流执行历史管理

执行历史查询

步骤 1 登录 FunctionGraph 控制台，在左侧导航栏选择“函数流”，进入“函数流”页面。

步骤 2 在“函数流”流程列表页面，单击需要查看执行历史的流程，进入流程详情页面。

步骤 3 切换至“运行记录”页签，查看执行历史运行记录。

步骤 4 左侧为执行历史记录列表，展示最近 100 次执行记录，支持根据流程执行 ID 进行查询。

- 单击左侧的执行记录，中间画布展示流程的执行结果，如果节点执行成功，图标背景为绿色，如果执行失败背景为红色。

图 3-63 执行失败

- 画布下方输入输出展示区默认展示流程的输入和输出，点击上方任意节点，展示节点的输入和输出。

说明

对于函数流及函数流各个节点，若输出值中有字段的值为 null，则该字段会被直接过滤，不予展示。

- 最下方日志展示流程从开始到结束所有节点的执行记录。

----结束

失败流程重试

步骤 1 登录 FunctionGraph 控制台，在左侧导航栏选择“函数流”，进入“函数流”页面。

步骤 2 在“函数流”流程列表页面，单击需要查看执行历史的流程，进入流程详情页面。

步骤 3 切换至“运行记录”页签，查看执行历史运行记录。

步骤 4 在失败的记录右侧单击重试图标，重试成功后会生成一条执行记录。

----结束

运行中流程终止

步骤 1 登录 FunctionGraph 控制台，在左侧导航栏选择“函数流”，进入“函数流”页面。

步骤 2 在“函数流”流程列表页面，单击需要查看执行历史的流程，进入流程详情页面。

步骤 3 切换至“运行记录”页签，单击执行中任务的停止图标，终止成功后流程会进入取消状态。

----结束

3.11.4 创建函数流触发器

本节介绍创建函数流触发器，函数流触发器当前支持 APIG 触发器、定时触发器。

创建定时触发器

步骤 1 登录 FunctionGraph 控制台，进入“函数流”页面。

步骤 2 在“函数流”流程列表页面，选择需要创建触发器的流程，单击“编辑”，进入编辑页面。

步骤 3 单击“开始”节点，在右侧弹出的属性页面添加触发器，触发器类型选择“定时触发器”。

步骤 4 填写触发器配置信息。如表 3-52 所示，带*参数为必填项。

表 3-52 定时触发器配置信息

配置项	说明
*触发规则	定时触发器的触发规则，当前只支持 Cron 表达式
*Cron 表达式	用于表示任务调度的表达式，能够表示特定周期进行的特定的时间、日期等。具体请参见 3.5.4 附录：函数定时触发器 Cron 表达式规则。
附加信息	附加信息为 json 格式，输入必须包含 input，在 input 内输入需要的 json 体。input 的内容会作为流程的输入参数。

步骤 5 单击“创建”，完成定时触发器创建。

----结束

创建 APIG（共享版）触发器

说明

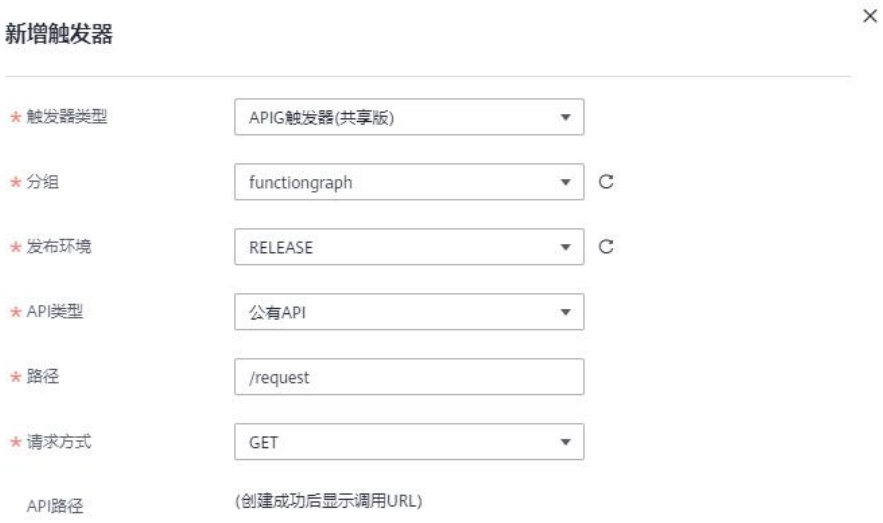
首次使用 API 网关的用户不再支持共享版服务，老用户仍可继续使用共享版服务。即 API 网关当前已不提供共享版，目前只有存量用户可以使用共享版。

函数流 APIG 触发器目前仅支持 IAM 认证方式。

步骤 1 登录 FunctionGraph 控制台，进入“函数流”页面。

步骤 2 在“函数流”流程列表页面，选择需要创建触发器的流程，单击“编辑”，进入编辑页面。

步骤 3 单击“开始”节点，在右侧弹出的属性页面添加触发器，触发器类型选择“APIG 触发器（共享版）”。



步骤 4 填写触发器配置信息。如表 3-53 所示，带*参数为必填项。

表 3-53 APIG 触发器（共享版）信息

字段	填写说明
*分组	API 分组相当于一个 API 集合，API 提供方以 API 分组为单位，管理分组内的所有 API。 选择“APIGroup_test”。
*发布环境	API 可以同时提供给不同的场景调用，如生产、测试或开发。API 网关服务提供环境管理，在不同的环境定义不同的 API 调用路径。 选择“RELEASE”，才能调用。
*API 类型	API 类型：
*路径	接口请求的路径。 格式如：/users/projects
*请求方式	接口调用方式：GET、POST、DELETE、PUT、PATCH、HEAD、OPTIONS、ANY 其中 ANY 表示该 API 支持任意请求方法。

步骤 5 单击“创建”，完成 APIG（共享版）触发器创建。

----结束

创建 APIG（专享版）触发器

说明

- 函数流 APIG 触发器目前仅支持 IAM 认证方式。

- 前提条件：需要预先创建 APIG 专享版实例。

步骤 1 登录 FunctionGraph 控制台，进入“函数流”页面。

步骤 2 在“函数流”流程列表页面，选择需要创建触发器的流程，单击“编辑”，进入编辑页面。

步骤 3 单击“开始”节点，在右侧弹出的属性页面添加触发器，触发器类型选择“APIG 触发器(专享版）”。

步骤 4 填写触发器配置信息。如表 3-54，带*参数为必填项。

表 3-54 APIG 触发器（专享版）信息

字段	填写说明
*实例	专享版 APIG 实例名称
*分组	API 分组相当于一个 API 集合，API 提供方以 API 分组为单位，管理分组内的所有 API。 选择“APIGroup_test”。
*发布环境	API 可以同时提供给不同的场景调用，如生产、测试或开发。API 网关服务提供环境管理，在不同的环境定义不同的 API 调用路径。 选择“RELEASE”，才能调用。
*API 类型	API 类型：
*路径	接口请求的路径。 格式如：/users/projects
*请求方式	接口调用方式：GET、POST、DELETE、PUT、PATCH、HEAD、OPTIONS、ANY

字段	填写说明
	其中 ANY 表示该 API 支持任意请求方法。

步骤 5 单击“创建”，完成 APIG（专享版）触发器创建。

----结束

4 常见问题

4.1 通用问题

4.1.1 FunctionGraph 是什么？

函数工作流（FunctionGraph）是一项基于事件驱动的函数托管计算服务。使用 FunctionGraph 函数，只需编写业务函数代码并设置运行的条件，无需配置和管理服务器等基础设施，函数以弹性、免运维、高可靠的方式运行。

4.1.2 使用 FunctionGraph 是否需要开通计算、存储、网络等服务？

用户使用 FunctionGraph 时，不需要开通或者预配置计算、存储、网络等服务，由 FunctionGraph 提供和管理底层计算资源，包括服务器 CPU、内存、网络和其他配置/资源维护、代码部署、弹性伸缩、负载均衡、安全升级、资源运行情况监控等，用户只需要按照 FunctionGraph 支持的编程语言提供程序包，上传即可运行。

4.1.3 使用 FunctionGraph 开发程序之后是否需要部署？

用户在本地开发程序之后打包，必须是 ZIP 包（Java、Node.js、Python、Go）或者 JAR 包（Java），上传至 FunctionGraph 即可运行，无需其它的部署操作。

制作 ZIP 包的时候，单函数入口文件必须在根目录，保证解压后，直接出现函数执行入口文件，才能正常运行。

对于 Go runtime，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 testplugin.so，则“函数执行入口”命名为 testplugin.Handler。

4.1.4 FunctionGraph 函数支持哪些编程语言？

FunctionGraph 目前支持的编程语言，如表 4-1 所示。

表 4-1 支持的编程语言和版本

语言	支持版本
Python	2.7、3.6
Node.js	6.10、8.10、10.16、12.13
Java	8
Go	1.8、1.x

4.1.5 FunctionGraph 函数分配磁盘空间有多少？

对于每个 FunctionGraph 函数分配了 512MB 临时存储空间，单个租户下最大允许部署包大小为 10G，更多函数的资源限制，请参考使用限制。

4.1.6 FunctionGraph 函数是否支持版本控制？

FunctionGraph 函数支持版本控制。

4.1.7 函数中如何读写文件？

函数工作目录权限说明

函数可以读取代码目录下的文件，函数工作目录在入口文件的上一级，例如用户上传了文件夹 backend，需要读取与入口文件同级目录的文件 test.conf，可以用相对路径“code/backend/test.conf”，或者使用绝对路径（相关目录为 RUNTIME_CODE_ROOT 环境变量对应的值）。如果需要写文件（如创建新文件或者下载文件等），可以在/tmp 目录下进行或者使用函数提供的挂载文件系统功能。

说明

- 若容器回收，文件的读写就会失效。
- 函数目前不支持持久化。

典型场景

- 需要对 OBS 上的文件进行处理，可以先把文件下载到/tmp 目录。
- 函数运行过程中产生了一些数据想保存到 OBS，可以先在/tmp 目录下创建新文件，然后把这些数据写到里面，接下来上传该文件到 OBS。

4.1.8 FunctionGraph 函数是否支持扩展？

FunctionGraph 目前已经集成了一些非标准库如：redis、http、obs_client 等，开发函数时可以直接使用。

用户可以通过维护属于自己的依赖代码库，供所有函数使用，请参考 3.9 依赖包管理。

4.1.9 IAM 子使用 FunctionGraph 需要设置哪些权限？

使用 IAM 子登录、使用函数工作流服务，对函数和函数下的触发器进行增删改查，如果出现权限不足情况，需要主对 IAM 子所属的用户组设置相应的权限，如创建 OBS 桶和 OBS 触发器，需要对 OBS 服务设置 Tenant Administrator 权限。其他操作按照最小授权原则设置相应的权限。

4.1.10 如何制作函数依赖包？

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

说明

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

为 Python 函数制作依赖包

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

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

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

执行成功后，执行以下命令。

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

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

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

所生成的包即为最终需要的依赖包。

说明

如果需要安装存放在的本地 wheel 安装包，直接输入：

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

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。

为 Java 函数制作依赖包

使用 Java 编译型语言开发函数时，依赖包需要在本地编译。

4.1.11 FunctionGraph 配额

FunctionGraph 服务的账户资源配额请参考账户资源限制。

4.1.12 函数工作流的常见使用场景？

1. Web 类应用：比如小程序、网页/App、聊天机器人、BFF 等。
2. 事件驱动类应用：文件处理、图片处理、视频直播/转码、实时数据流处理、IoT 规则/事件处理等。
3. AI 类应用：三方服务集成、AI 推理、车牌识别。

4.1.13 已创建的函数是否支持修改函数名称?

不支持，函数一旦创建完成，就不能修改函数名称。

4.1.14 同步调用响应未收到的可能原因？

如果函数执行端到端时延超过 90s，建议使用异步不使用同步，否则会因为网关限制，超过 90s 后无法收到同步响应。

4.1.15 自定义运行时，都能操作哪些目录？

目前默认只能操作/tmp 目录，在/tmp 下可以写文件（如创建新文件或者下载文件等）。

4.1.16 用户想使用 vpc 功能，但不想配置 VPC Administrator 委托，应配置哪些授权项？

用户若不想配置 VPC Administrator 委托，可授予最小权限，如表 4-2 所示。

表 4-2 授权项配置

权限	授权项
删除端口	vpc:ports:delete
查询端口	vpc:ports:get
创建端口	vpc:ports:create
查询 VPC	vpc:vpcs:get
查询子网	vpc:subnets:get

4.1.17 函数执行超时的可能原因有哪些？

- 自身代码执行逻辑超时，建议优化代码或增加超时时间。
- 网路请求超时，建议增加超时时间。
- 函数进行冷启动时，Java 加载类时间过长，建议增加超时时间或增加内存。

4.2 创建函数

4.2.1 能否在函数代码中使用线程和进程？

用户可使用编程语言和操作系统的功能，在函数中创建额外的线程和进程。

4.2.2 FunctionGraph 函数工程打包有哪些规范（限制）？

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。

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 4-3 所示。

表 4-3 函数工程打包规范

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

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
```

- 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
```

4.2.3 FunctionGraph 如何隔离代码？

每个 FunctionGraph 函数都运行在其自己的环境中，有其自己的资源和文件系统。

4.3 触发器管理

4.3.1 使用 APIG 触发器调用一个返回 String 的 FunctionGraph 函数，报 500 错误，该如何解决？

FunctionGraph 函数对来自 APIG 调用的返回结果进行了封装，APIG 触发器要求函数的返回结果中必须包含 body(String)、statusCode(int)、headers(Map)和 isBase64Encoded(boolean)，才可以正确返回。

Node.js 函数 APIG 触发器调用返回结果定义示例如下：

```
exports.handler = function (event, context, callback) {
    const response = {
```

```
        'statusCode': 200,
        'isBase64Encoded': false,
        'headers': {
            "Content-type": "application/json"
        },
        'body': 'Hello, FunctionGraph with APIG',
    }
    callback(null, response);
}
```

Java 函数 APIG 触发器调用返回结果定义示例如下：

```
import java.util.Map;

public HttpTriggerResponse index(String event, Context context){
        String body = "<html><title>FunctionStage</title>"
                + "<h1>This is a simple APIG trigger test</h1><br>"
                + "<h2>This is a simple APIG trigger test</h2><br>"
                + "<h3>This is a simple APIG trigger test</h3>"
                + "</html>";
        int code = 200;
        boolean isBase64 = false;
        Map<String, String> headers = new HashMap<String, String>();
        headers.put("Content-Type", "text/html; charset=utf-8");
        return new HttpTriggerResponse(body, headers, code, isBase64);
    }


class HttpTriggerResponse {
    private String body;
    private Map<String, String> headers;
    private int statusCode;
    private boolean isBase64Encoded;
    public HttpTriggerResponse(String body, Map<String,String> headers,
int statusCode, boolean isBase64Encoded){
        this.body = body;
        this.headers = headers;
        this.statusCode = statusCode;
        this.isBase64Encoded = isBase64Encoded;
    }
}
```

4.4 函数执行

4.4.1 FunctionGraph 函数的执行需要多长时间？

调用函数的执行时间在 900 秒内，FunctionGraph 函数默认的执行超时时间为 3 秒，您可以自行设置执行超时时间为 3～900 秒之间的任何整数。如果执行超时时间设置为 3 秒，超过 3 秒后，函数将终止执行。

4.4.2 FunctionGraph 函数的执行包含了哪些过程？

FunctionGraph 函数的执行过程包含两步：

1. 选择一个相应内存的空闲实例。
2. 执行用户的指定运行代码。

4.4.3 FunctiongGraph 函数的并发处理过程是什么？

FunctionGraph 会根据实际的请求情况自动弹性伸缩函数实例，并发变高时，会分配更多的函数实例来处理请求，并发减少时，相应的实例也会变少。

4.4.4 FunctiongGraph 函数如何处理长时间不执行的实例？

如果一个函数在一段时间内一直没有执行，那么所有与之相关的实例都会被释放。

4.4.5 首次访问函数慢，如何优化？

如果您使用的是 C#或者 Go 语言，因为机制原因，启动速度会比其他语言慢。此时，您可以通过以下设置，增加运行速度。

- 适当增加函数的内存。
- 精简函数代码，例如：删除不必要的依赖包。
- 使用 C#语言时，除了以上两种方法，在非并发场景下，您还可以通过以下方法增加运行速度。

  创建一个一分钟一次的定时触发器，确保至少有一个存活的实例。

4.4.6 怎样获取在函数运行过程中实际使用了多少内存？

函数调用的返回信息中会包含最大内存消耗等信息。也可以在执行结果界面查看。

4.4.7 如何读取函数的请求头？

函数入口中的第一个参数里面包含请求头，您可以打印函数执行结果，从而获取想要的字段。

如下图，event 为函数入口的第一个参数，headers 为请求头。

```
index.py ×
1  # -*- coding:utf-8 -*-
2  import json
3  def handler (event, context):
4      body = "<html><title>Functiongraph Demo</title><body><p>Hello, FunctionGraph!
5      print(body)
6      return {
7          "statusCode":200,
8          "body":body,
9          "headers":{
10             "Content-Type": "text/html",
11         },
12         "isBase64Encoded": False
13     }
```

4.4.8 为什么函数实际使用内存大于预估内存，甚至触发 OOM？

1. 函数调用过程中，运行时会解析和缓存传入的 event 事件， 这部分操作会消耗额外的内存。
2. 函数调用结束后，回收的内存首先会放入内部内存池中，并不一定归还给操作系统，导致内存偏高，在高并发场景下这种现象会更加明显。

4.4.9 函数内存超限返回“runtime memory limit exceeded”，如何查看内存占用大小？

请在函数请求返回界面查看。

图 4-2 查看 oom 内存大小

4.4.10 用户使用相同的镜像名更新镜像，预留实例无法自动更新，会一直使用老镜像，应如何处理？

建议使用非 latest 的镜像 tag 管理镜像更新，避免使用完全相同的镜像名。

4.5 函数配置

4.5.1 FunctionGraph 函数是否支持环境变量？

创建函数时可以设置环境变量，无需对代码进行任何更改，可以设置动态参数，传递到函数代码和库，请参考 3.3.6 配置环境变量。

4.5.2 能否在函数环境变量中存储敏感信息？

定义环境变量时，系统会明文展示所有输入信息，请不要输入敏感信息（如账户密码等），以防止信息泄露。