ALIBABA CLOUD

阿里云

函数计算

触发器管理

文档版本：20201215

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
Courier字体	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.触发器简介

触发器是触发函数执行的方式。在事件驱动的计算模型中，事件源是事件的生产者，函数是事件的处理者，而触发器提供了一种集中、统一的方式来管理不同的事件源。在事件源中，当事件发生时，如果满足触发器定义的规则，事件源会自动调用触发器所对应的函数。

什么是触发器

函数计算提供了一种事件驱动的计算模型。函数的执行是由事件驱动的。函数的执行可以通过函数计算控制台、fcli工具、VSCode插件或SDK触发，也可以由其它一些事件源来触发。您可以在指定函数中创建触发器，该触发器描述了一组规则，当某个事件满足这些规则，事件源就会触发相应的函数。

触发器的基本属性

- triggerName：触发器名称。
- triggerType：触发器类型。例如，OSS触发器、HTTP触发器、定时触发器等。
- sourceArn：触发函数执行的资源描述符。
  - 涉及到阿里云其他服务触发函数计算执行时需要设置。例如，OSS触发器、SLS触发器等。
  - 不涉及到阿里云其他服务触发函数计算执行时不需要设置。例如，定时触发器，HTTP触发器等。

  示例：OSS触发器的sourceArn格式为：`acs:oss:region:accountId:bucketName`。
- invocationRole：触发角色，事件源需要扮演一个角色来触发函数的执行，要求这个角色有触发函数执行的权限。详情请参见权限简介。
- qualifier：触发的服务版本或别名。更多版本和别名的使用请参见版本简介。
- triggerConfig：触发器的配置信息，各触发器的配置信息请参考该触发器对应的文档。

场景示例

- 示例一：对象存储OSS（Object Storage Service）中的图片状态变更触发函数执行
  某应用使用对象存储OSS存放上传的图片，您可以通过直接调用函数的方式去下载图片进行处理，并将结果存入OSS或者其他服务。如果OSS能够帮助我们关注新上传的图片，并且自动去调用相应函数，您就不需要再去自己调用函数了，从而简化了开发和使用流程。OSS触发器的作用就是关注这些事件并调用函数计算的函数。配置了OSS触发器后，当有新图片上传，OSS触发器会自动触发函数下载并处理图片。
- 示例二：日志服务SLS（Log Service）中日志更新触发函数执行
  某应用使用日志服务SLS定时采集更新的日志，您可以通过直接调用函数对增量的日志进行查询，分析。如果SLS能够帮助我们关注更新的日志，并自动调用相应的函数，您就不需要再去自己调用函数。SLS触发器的作用就是关注这些事件并调用函数计算的函数。配置了SLS触发器后，当有日志更新，SLS触发器会自动触发函数消费增量的日志。
- 示例三：在指定时间触发函数执行
  某应用需要每隔1小时收集一次数据。您可以每隔1小时通过直接调用函数收集数据并处理。如果函数计算中的函数能每隔1小时自动执行，您就不需要再去关注时间。定时触发器的作用就是关注时间事件并调用函数计算的函数。配置了定时触发器后，在指定的时间，定时触发器会自动触发函数收集和处理数据。

除上述场景外，函数计算还支持很多其他事件源，详情请参见触发器列表，后续会陆续集成其他事件源。

2.触发器列表

函数计算支持的触发器分为单向集成触发器和双向集成触发器。
单向和双向集成触发器的区别如下：

- 双向集成触发器：您既可以在函数计算又可以在事件源端配置触发器。
- 单向集成触发器：您只需要在事件源端配置触发器，无需在函数计算配置。

双向集成触发器

触发器名称	文档链接	示例链接
OSS事件触发器	OSS触发器概述	使用OSS触发函数计算示例
HTTP触发器	HTTP触发器概述	基于HTTP触发器搭建Web Server
定时触发器	定时触发器概述	使用定时触发器触发函数计算示例
MNS主题触发器	MNS主题触发器概述	使用MNS主题触发器触发函数计算示例
Tablestore触发器	Tablestore触发器概述	使用函数计算做数据清洗
CDN事件触发器	CDN事件触发器概述	CDN事件触发器示例
SLS触发器	SLS触发器	SLS触发器示例

单向集成触发器

触发器名称	示例链接
API网关触发器	API网关触发器
IoT触发器	IoT触发器
云监控触发器	云监控触发器
消息队列Kafka版Connector触发器	创建FC Sink Connector
事件总线EventBridge触发器	路由到函数计算

触发器简易预览

图解函数计算&事件源服务

3.触发器管理

触发器（Trigger）定义了函数可以被触发执行的方式，是函数级别的。本文介绍如何在函数计算控制台上创建、更新、删除触发器。

前提条件

1. 创建服务
2. 创建函数

创建触发器

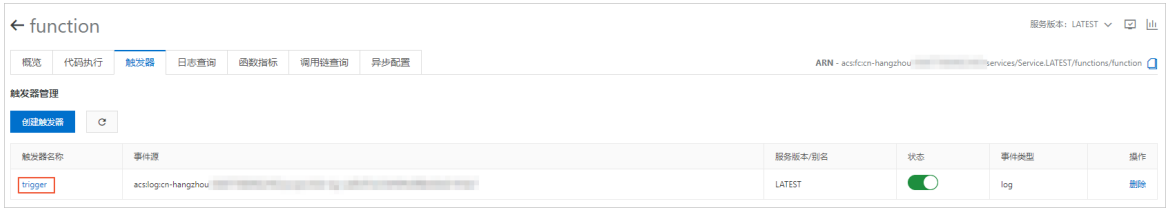
- 创建OSS触发器
- 创建HTTP触发器
- 创建MNS主题触发器
- 创建日志服务触发器
- 创建TableStore触发器
- 创建CDN事件触发器
- 创建定时触发器

更新触发器

1. 登录函数计算控制台。
2. 在顶部菜单栏，选择地域。
3. 在左侧导航栏，单击**服务/函数**。
4. 找到目标服务下的目标函数，单击函数名称。

> **说明** 若目标服务存在多个版本，需要选择LATEST版本。

5. 在函数详情页面，单击**触发器**页签。
6. 在触发器列表中，单击目标触发器名称。然后在**修改触发器**区域，根据需要更新触发器的配置。



删除触发器

1. 登录函数计算控制台。
2. 在顶部菜单栏，选择地域。
3. 在左侧导航栏，单击**服务/函数**。
4. 找到目标服务下的目标函数，单击函数名称。

> **说明** 若目标服务存在多个版本，需要选择LATEST版本。

5. 在函数详情页面的触发器列表中，单击目标触发器**操作**列中的**删除**。然后，单击**确认**。

相关文档

使用fcli命令行工具管理触发器的具体信息，请参见初次使用fcli。

4.触发器事件格式

对于不同的触发器，其触发的事件传递到函数接口的事件（event）对象格式不同，本文列举了不同触发器的事件对象格式。

event格式列表

触发器类型	event格式说明
OSS触发器	event格式说明
HTTP触发器	HTTP触发器有别于其他触发器，函数签名是请求（Request）和响应（Response）对象，而不是事件（event）对象。所以HTTP触发器没有事件格式。详情请参见HTTP触发器概述。
MNS触发器	event格式说明
SLS触发器	event格式说明
Tablestore触发器	event格式说明
CDN触发器	event格式说明
定时触发器	event格式说明

IoT触发器	IoT hub传入函数计算的event内容是没有封装的IoT hub消息内容。例如，您可以通过以下Java示例向IoT Topic推送消息。
	`PubRequest request = new PubRequest();`
	`request.setProductKey("VHo5FRjudkZ");`
	`request.setMessageContent(Base64.getEncoder().encodeToString("{\"hello\":\"world\"}".getBytes()));`
	`request.setTopicFullName("/VHo5FRjudkZ/deviceName/update");`
	`request.setQos(0);`
	`PubResponse response = client.getAcsResponse(request);`
	`System.out.println(response.getSuccess());`
	`System.out.println(response.getErrorMessage());`
	在函数计算函数内部收到的事件为：
	`{`
	`"hello": "world"`
	`}`
	更多IoT触发器的内容请参见数据转发到函数计算。

相关参考文档

- 什么是对象存储OSS

- 什么是日志服务
- 什么是表格存储
- 什么是物联网平台
- 什么是阿里云CDN

5.OSS触发器

5.1. OSS触发器概述

对象存储服务OSS（Object Storage Service）是阿里云提供的海量、安全、低成本、高可靠的云存储服务。OSS与函数计算集成后，OSS事件能触发相关函数执行，实现对OSS中的数据进行自定义处理。

背景信息

OSS和函数计算通过OSS事件触发器实现无缝集成，您可以编写函数对OSS事件进行自定义处理，当OSS系统捕获到指定类型的事件后，OSS事件触发相应的函数执行。例如，您可以设置函数来处理PutObject事件，当您调用OSS的PutObject API接口上传图片到OSS后，相关联的函数会自动被触发来处理该图片。

OSS和函数计算集成后，您可以自由的调用各种函数处理图像或音频数据，再把结果写回到多种存储服务中。整个架构中，您只需要专注于函数逻辑的编写，系统将以实时的、可靠的、大规模并行的方式处理海量的数据。

地域限制

以下地域（Region）不支持使用OSS触发器：

- 西南1（成都）
- 英国（伦敦）
- 马来西亚（吉隆坡）

OSS事件定义

当OSS系统捕获到相关事件后，会将事件信息编码为JSON字符串，传递给事件处理函数。OSS事件通知格式的详细信息，请参见事件通知消息格式。已支持的OSS事件定义如下表所示。

事件名称	说明
oss:ObjectCreated:PutObject	调用PutObject接口上传文件，更多信息，请参见PutObject。 API执行成功后触发函数。
oss:ObjectCreated:PutSymlink	调用PutSymlink接口针对OSS上的TargetObject创建软链接，您可以通过该软链接访问TargetObject，更多信息，请参见PutSymlink。 API执行成功后触发函数。
oss:ObjectCreated:PostObject	调用PostObject接口使用HTML表单上传文件到指定的Bucket，更多信息，请参见PostObject。 API执行成功后触发函数。
oss:ObjectCreated:CopyObject	调用CopyObject接口拷贝一个在OSS上已经存在的对象，更多信息，请参见CopyObject。 API执行成功后触发函数。
oss:ObjectCreated:InitiateMultipartUpload	使用MultipartUpload模式传输数据前，必须先调用InitiateMultipartUpload接口来通知OSS初始化一个MultipartUpload事件，更多信息，请参见InitiateMultipartUpload。 API执行成功后触发函数。
oss:ObjectCreated:UploadPart	初始化一个Multipart Upload之后，可以根据指定的对象名和Upload ID来分块（Part）上传数据，更多信息，请参见UploadPart。 API执行成功后触发函数。

事件名称	说明
oss:ObjectCreated:UploadPartCopy	UploadPartCopy通过从一个已存在的Object中拷贝数据来上传一个Part，更多信息，请参见UploadPartCopy。 API执行成功后触发函数。
oss:ObjectCreated:CompleteMultipartUpload	在将所有数据Part都上传完成后，必须调用CompleteMultipartUpload接口来完成整个文件的MultipartUpload。更多信息，请参见CompleteMultipartUpload。 API执行成功后触发函数。
oss:ObjectCreated:AppendObject	调用AppendObject接口以追加写的方式上传文件，更多信息，请参见AppendObject。 API执行成功后触发函数。
oss:ObjectCreated:*	任何上述ObjectCreated类型的API执行成功后都会触发函数。
oss:ObjectRemoved:DeleteObject	调用DeleteObject接口删除某个对象，更多信息，请参见DeleteObject。 API执行成功后触发函数。
oss:ObjectRemoved:DeleteObjects	调用DeleteObjects接口批量删除文件。 API执行成功后触发函数。
oss:ObjectRemoved:AbortMultipartUpload	调用 `AbortMultipartUpload` 接口可以根据用户提供的Upload ID中止其对应的MultipartUpload事件，更多信息，请参见AbortMultipartUpload。 API执行成功后触发函数。

> **说明** 一个Bucket最多能创建10个触发规则事件。

触发规则

使用OSS触发器一定要避免循环触发。一个典型的循环触发场景是OSS的某个Bucket上传文件触发函数执行，这个函数又生成了一个或多个文件，写回到OSS的Bucket里，这个写入动作又触发了函数执行，形成了链状循环。

上述过程类似于一个无限递归，为了避免这种循环触发函数产生不必要的费用，强烈建议您配置前缀或后缀，例如将触发函数的Bucket目录前缀设置成*src/*，生成的文件写入的目录前缀设置为*dst/*，这样生成的文件就不会再次触发函数。

一个Bucket的不同触发器如果指定了相同的事件类型，则前缀和后缀不能重复。假如一个触发器配置了oss:ObjectCreated:PutObject事件，*source*前缀和*zip*后缀。下表列出了是否可以给该Bucket配置其它触发器。

事件	前缀	后缀	是否合法	说明
oss:ObjectCreated:PutObject	source	zip	否	前后缀一样
oss:ObjectCreated:PutObject	source	无	否	前缀重合，后缀不设置包含了后缀为zip的对象。
oss:ObjectCreated:PutObject	无	zip	否	后缀重合，前缀不设置包含了前缀为source的对象。

事件	前缀	后缀	是否合法	说明
oss:ObjectCreated:PutObject	source1	1zip	是	不重合
oss:ObjectCreated:PutObject	source	zip1	是	后缀不重合
oss:ObjectCreated:PutObject	1source	zip	是	前缀不重合
oss:ObjectCreated:PostObject	source	zip	是	事件不同

说明

如果您需要让一个触发器触发多个函数，可以结合使用函数计算和Serverless工作流服务。先通过触发器触发一个函数，该函数启动Serverless工作流流程，在流程里您可以调用多个函数。更多信息，请参见示例。

6.HTTP触发器

6.1. HTTP触发器概述

HTTP触发器通过发送HTTP请求触发函数执行，主要适用于快速构建Web服务等场景。HTTP触发器支持HEAD、POST、PUT、GET和DELETE方式触发函数。

HTTP触发器优势

HTTP触发器与API网关触发器均可应用于Web应用的创建。相较于API网关触发器，HTTP触发器有以下优势：

- 简化了开发人员的学习成本和调试过程，帮助开发人员快速使用函数计算搭建Web service和API。
- 支持选择您熟悉的HTTP测试工具验证函数计算侧的功能和性能。
- 减少请求处理环节，HTTP触发器支持更高效的请求、响应格式，不需要编码或解码成JSON格式，性能更优。
- 方便对接其他支持Webhook回调的服务，例如CDN回源、MNS等。

使用限制

- 函数设置HTTP触发器后不能设置其他类型触发器。
- 每个函数只能创建一个HTTP触发器。
- 若您使用了版本管理功能，每个函数对应的版本或别名只能创建一个HTTP触发器。即一个函数的一个版本或一个别名对应可以创建一个HTTP触发器。版本和别名的详细信息请参见版本简介。
- Request Headers key中包含以下字段会被忽略，不支持用户自定义。另外，以x-fc-开头的Key也不支持自定义。
  - accept-encoding
  - connection
  - keep-alive
  - proxy-authorization
  - te
  - trailer
  - transfer-encoding
- Response Headers key中包含以下字段会被忽略，不支持自定义。另外，以 `x-fc-` 开头的Key也不支持自定义。
  - connection
  - content-length
  - content-encoding
  - date
  - keep-alive
  - proxy-authenticate
  - server
  - trailer
  - transfer-encoding
  - upgrade

- content-disposition:attachment

> **说明** 从安全角度考虑，使用函数计算默认的aliyuncs.com域名，服务端会在response header中强制添加content-disposition: attachment字段，此字段会使得返回结果在浏览器中以附件的方式下载。如果要移除该限制，需设置自定义域名，详情请参见绑定自定义域名。

- Request限制项。
  如果超过以下限制，会返回 `400` 状态码和 `InvalidArgument` 错误码。
  - headers大小：headers中的所有Key和Value的总大小不得超过4 KB。
  - path大小：包括各个query params，path的总大小不得超过4 KB。
  - body大小：HTTP body的总大小不得超过6 MB。
- Response限制项。
  如果超过以下限制，会返回 `502` 状态码和 `BadResponse` 错误码。
  - headers大小：headers中的所有Key和Value的总大小不得超过4 KB。
  - body大小：HTTP body的总大小不得超过6 MB。
- 其他使用说明。
  您可以通过绑定自定义域名为HTTP函数映射不同的HTTP访问路径，详情请参见绑定自定义域名。除此外还可以使用API网关，后端服务类型选择HTTP服务，设置HTTP函数为后端服务地址，实现类似功能，详情请参见使用函数计算作为API网关后端服务。

HTTP触发器的入口函数形式

设置HTTP触发器的函数和普通函数的入口函数是有差异的，普通函数（左）与设置HTTP触发器的函数（右）的对比示例如下图所示。如果您设置HTTP触发器后依然使用普通接口，请及时修改。

- Node.js

```node.js
module.exports.handler = function(event, context, callback) {
    console.log('hello world');
    callback(null, 'hello world');
};
```

```node.js
module.exports.handler = function(req, resp, context) {
    console.log("hello world");
    resp.send("hello world");
}
```

- Python

```python
# -*- coding: utf-8 -*-
import logging

def handler(event, context):
  logger = logging.getLogger()
  logger.info('hello world')
  return 'hello world'
```

```python
# -*- coding: utf-8 -*-
HELLO_WORLD = b"Hello world!\n"

def handler(environ, start_response):
    context = environ['fc.context']
    request_uri = environ['fc.request_uri']
    for k, v in environ.items():
      if k.startswith("HTTP_"):
        # process custom request headers
        pass
    # do something here
    status = '200 OK'
    response_headers = [('Content-type', 'text/plain')]
    start_response(status, response_headers)
    return [HELLO_WORLD]
```

其他更多内容请参见以下文档：

- Node.js HTTP函数
- Python HTTP函数
- PHP HTTP函数
- Java HTTP函数
- C# HTTP函数

- 简介

6.2. HTTP触发器示例

6.2.1. 示例简介

函数计算支持HTTP触发器，配置HTTP触发器的函数可以通过HTTP请求被触发执行。此时函数可以看做一个Web Server，对HTTP请求进行处理，并将处理结果返回给调用端。

示例场景

您可以配置一个HTTP触发器，将HTTP Request请求中的参数作为Response的信息返回给调用端。

配置触发器流程

函数计算支持多种方式配置触发器：

- 通过控制台配置触发器
  i. 创建触发器
  ii. 编写函数
  iii. 测试函数
- 通过Funcraft工具配置触发器，详情请参见fun。
- 通过fcli工具配置触发器，详情请参见初次使用fcli。
- 通过SDK配置触发器，详情请参见SDK列表。

6.2.2. 创建触发器

本文介绍如何通过函数计算控制台创建HTTP触发器。

前提条件

创建服务

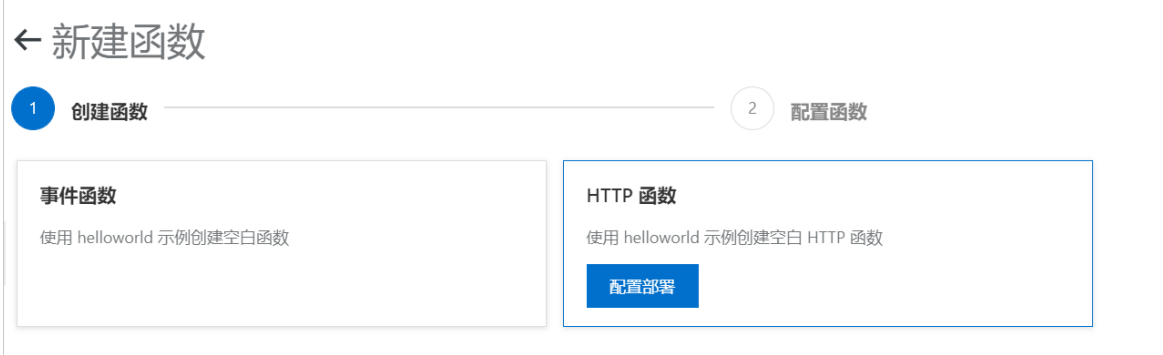
背景信息

创建HTTP触发器与创建其他类型的触发器不同。HTTP触发器相关参数是在创建函数的过程中配置，而其他类型的触发器相关参数是在已创建的函数中配置。

操作步骤

1. 登录函数计算控制台。
2. 在顶部菜单栏，选择地域。
3. 在左侧导航栏中，单击**服务/函数**。在**服务/函数**页面右上角单击**新增函数**。

4. 在**新增函数**页面，单击**HTTP函数**，然后单击**配置部署**。



5. 在**配置函数**区域，填写函数相关信息。

参数	操作	本文示例
所在服务	在列表中选择函数所在的服务。	service
函数名称	填写自定义的函数名称。	function
运行环境	选择您熟悉的语言，例如Python、Java、PHP、Node.js等。	python3
函数入口	填写函数入口。格式为[文件名].[函数名]。	index.handler
函数执行内存	设置函数执行内存。默认内存大小为512 MB，最大为3072 MB。	512 MB
超时时间	设置超时时间。默认超时时间为60秒，最长为600秒。 超过设置的超时时间，函数将以执行失败结束。	60
实例并发度	单个实例能够并发处理的请求数。	Python运行环境不支持配置

6. 在**配置触发器**区域，填写触发器相关信息。

参数	操作	本文示例
触发器名称	填写自定义的触发器名称。	HTTP-Trigger

参数	操作	本文示例
认证方式	选择鉴权类型，取值： ○ **anonymous**：不需要身份验证，支持匿名访问，安全性低，任何人都可以发HTTP请求调用您的函数。 ○ **function**：需要通过身份验证，不支持匿名访问，安全性高。详情请参见签名认证。 **说明** HTTP Request的header中需要传入Authorization和Date信息。其中Date为GMT格式，且参与签名的运算，服务器端会以Date的时间计算签名，并与传入的Authorization的值进行比对，若签名比对成功且当前时间与Date时间相差15 min以内，才判定认证通过。	anonymous
请求方式	HTTP触发器支持的访问方式，可以多选，取值： ○ **GET** ○ **POST** ○ **PUT** ○ **DELETE** ○ **HEAD** ○ **PATCH**	GET、POST

7. 单击**完成**。

在目标服务下可以看到已创建的函数。

单击目标函数，然后单击**触发器**页签，可以看到已创建好的HTTP触发器。

后续步骤

1. 编写函数
2. 测试函数

6.2.3. 编写函数

完成创建HTTP触发器后，您可以开始编写函数代码。本文介绍如何使用函数计算控制台编写函数。

前提条件

创建触发器

操作步骤

1. 登录函数计算控制台。
2. 在顶部菜单栏，选择地域。
3. 在左侧导航栏，单击**服务/函数**。
4. 找到目标服务下的目标函数，单击函数名称。

5. 单击**代码执行**页签，在代码编辑器中编写代码。代码示例如下：

Node.js | Python | PHP

```
var getRawBody = require('raw-body')
module.exports.handler = function (request, response, context) {
  // get request info
  getRawBody(request, function (err, data) {
    var params = {
      path: request.path,
      queries: request.queries,
      headers: request.headers,
      method: request.method,
      body: data,
      url: request.url,
      clientIP: request.clientIP,
    }
    // you can deal with your own logic here
    // set response
    var respBody = new Buffer.from(JSON.stringify(params));
    // var respBody = new Buffer( )
    response.setStatusCode(200)
    response.setHeader('content-type', 'application/json')
    response.send(respBody)
  })
};
```

后续步骤

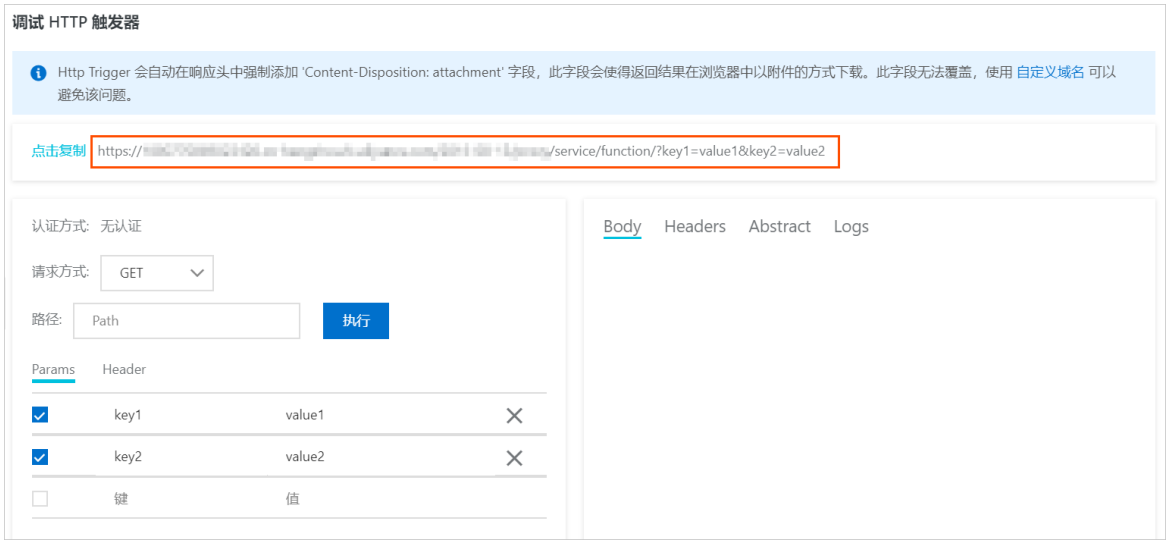
测试函数

6.2.4. 测试函数

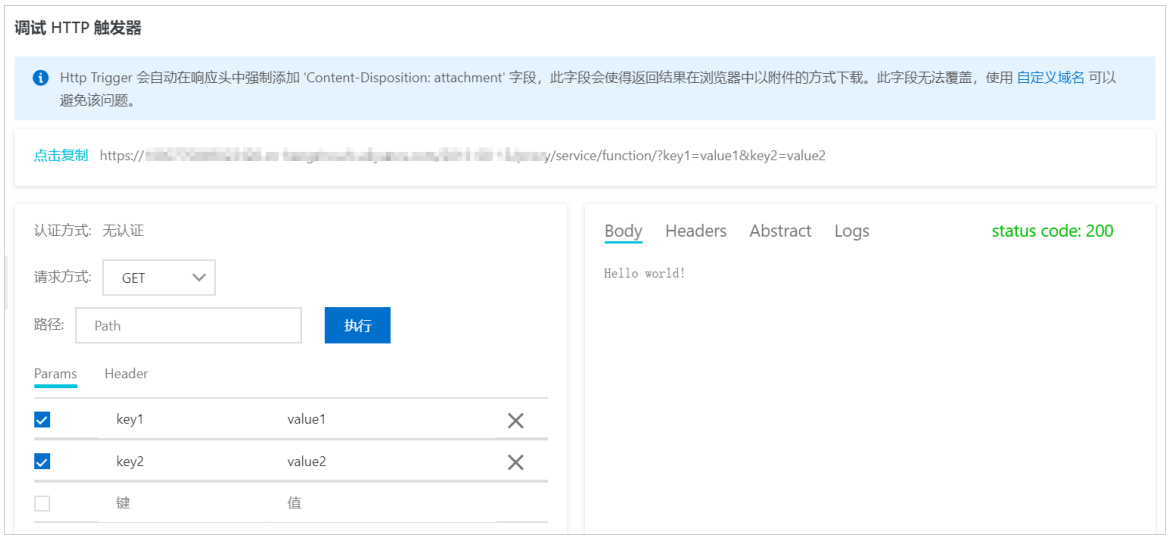
您需要向函数发送HTTP请求测试函数执行是否符合预期。您可以使用函数计算控制台或浏览器发送HTTP请求来测试函数的正确性。

使用控制台测试函数

1.
2. 在顶部菜单栏，选择地域。
3. 在左侧导航栏，单击**服务/函数**。
4. 找到目标服务下的目标函数，单击目标函数名称。
5. 单击**代码执行**页签，向下滚动页面至**调试HTTP触发器**区域。
6. 分别在Params和Header页签下的**键**、**值**文本框中输入键值对。
   输入后可以看到函数计算平台自动为您组装的HTTP触发器URL。



7. 单击**执行**。
   您可以看到函数的执行结果。



使用浏览器测试函数

1. 组装HTTP触发器URL。HTTP触发器URL的组装规则如下：

   ```
   <account_id>.<region>.fc.aliyuncs.com/<version>/proxy/<serviceName>/<functionName>/[action?queries]
   ```

   示例如下：

   ```
   123456.cn-shanghai.fc.aliyuncs.com/2016-08-15/proxy/serviceName/functionName/action?hello=world
   ```

   URL中的参数说明如下：

参数	说明

参数	说明
account_id	阿里云账号ID。 您可以在安全设置中获取您的账号ID。若您的账号是子账号，将鼠标移至控制台右上角的头像处可以看到您的阿里云账号ID。
region	函数计算服务所在的地域。
version	函数计算的API版本。
action	自定义的请求路径。
serviceName	服务名称。
functionName	函数名称。
queries	查询参数。

2. 将HTTP触发器URL输入浏览器地址栏，按回车键执行。
   浏览器中会返回执行结果。

6.2.5. 问题诊断

为什么HTTP函数无法执行？

- 如果是您新创建的触发器，会有10s左右的缓存更新时间，请稍后再试。
- 请检查函数的入口函数是否正确，HTTP函数与普通函数的入口函数不同。不同语言的HTTP函数请参见以下文档：
  - Node.js HTTP函数
  - Python HTTP函数
  - PHP HTTP函数
  - Java HTTP函数
  - C# HTTP函数
  - Custom Runtime HTTP函数

为什么函数无法结束？

请检查是否调用返回函数。

- Node.js需调用 `response.send()` 。
- Python需调用 `return` 。
- PHP需调用 `return new Response()` 。
- Java需调用 `HttpServletResponse`
- C#需调用 `return` 。
- Custom Runtime以各语言示例为准。

错误排查

错误主要分为以下两种：

- 请求错误是指发送的Request不符合标准，在Response里报错状态码为4xx。
- 函数错误即编写的函数有问题，会报5xx状态码。

下表描述了请求错误和函数错误可能出现的场景，以便您迅速排查问题。

错误类型	X-Fc-Error-Type	HTTP状态码	原因分析	是否计费
请求错误	FcCommonError	400	您的请求超过Request限制项的限制。详情请参见使用限制。	否
	FcCommonError	400	调用需要身份认证的函数的Request没有传入Date信息或Authorization信息。	否
	FcCommonError	403	调用需要身份认证的函数的Request的签名错误，即Authorization不正确，由于Date参与签名计算，且超过15 min，签名失效，一种常见的原因是使用需要访问认证的HTTP触发器，Request header中发送的Date据当前时间超过15 min，导致签名失效。	否
	FcCommonError	403	您的Request请求使用了HTTP触发器中未配置的请求方法。例如，HTTP触发器中的请求方法只配置了GET方法，却发送POST方法的HTTP请求。	否
	FcCommonError	404	向没有设置HTTP触发器的函数发送HTTP请求。	否
用户流控	FcCommonError	429	用户被流控，可减小并发量或者联系函数计算开发团队提高并发度。	否

错误类型	X-Fc-Error-Type	HTTP状态码	原因分析	是否计费
函数错误	UnhandledInvocationError	502	函数的返回值超过Response限制项的限制。详情请参见使用限制。	是
	UnhandledInvocationError	502	函数代码有语法错误或者异常。	是
	UnhandledInvocationError	502	向未使用HTTP入口函数的函数发送HTTP请求。	是
系统错误	FcCommonError	500	函数计算系统错误，可重试解决。	否
系统流控	FcCommonError	503	函数计算系统流控。可用指数退避方式重试。	否

如果问题还未能解决，请联系我们。

7.MNS主题触发器

7.1. MNS主题触发器概述

阿里云消息服务MNS（Message Notification Service）与阿里云函数计算集成后，将MNS Topic（主题）作为事件源接入函数计算（即Topic的订阅者是函数）。通过消息服务主题触发器（以下简称“MNS主题触发器”）触发函数执行，可以对发布在Topic上的消息进行自定义处理。

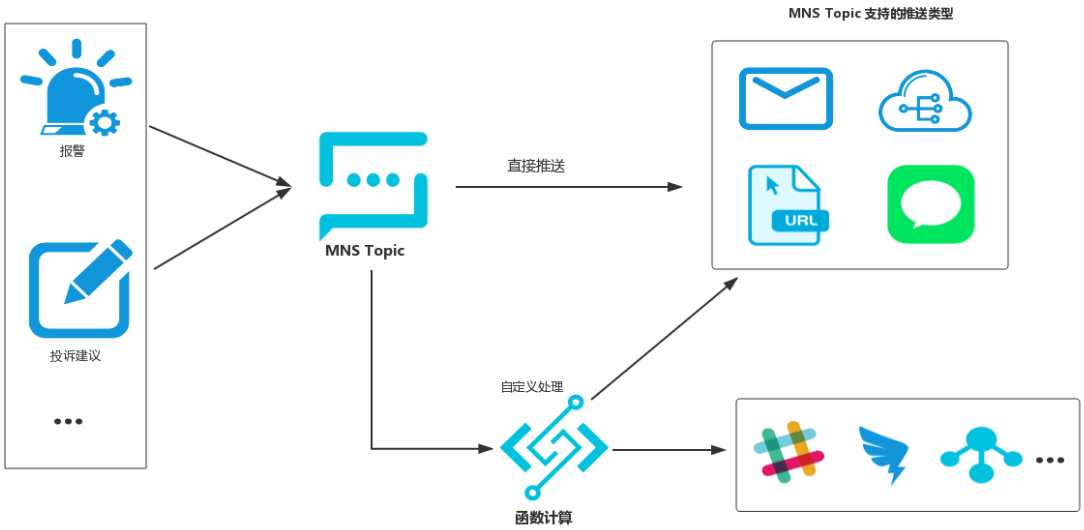
背景介绍

MNS是一种高效、可靠、安全、便捷、可弹性扩展的分布式消息服务。MNS能够帮助应用开发者在他们应用的分布式组件上自由地传递数据、通知消息，构建松耦合系统。在MNS中，Topic是发布消息的目的地。发布者可以通过PublishMessage接口向Topic发布消息，Topic的订阅者接收该消息。接口信息请参见PublishMessage，更多MNS的内容请参见MNS产品文档。

配置一个MNS主题触发器，相当于将函数注册为这个MNS主题的订阅者，当发布者向MNS主题发布消息的时候，就会把消息内容通知给函数，即触发函数执行，同时消息内容作为函数入口的event参数，详情请参见函数入参。

目前，MNS可以支持特定的订阅对象，但是不够灵活。而MNS与函数计算集成有以下优势：

- 可以实现对消息进行一些高阶处理再发送邮件或者短信。
- HTTP Endpoint不需要有自建的服务。
- 支持丰富的自定义处理。例如，把消息发送给Slack或者对于特定的消息进行持久化存储。



注意事项

- 强烈建议MNS Topic和函数计算的函数在相同的地域。
  虽然 MNS主题触发器支持Topic和函数在不同的地域，但是在不同的地域会增加网络延时，尤其当函数所在的地域和MNS Topic所在的地域分别是国内和国外时。
- 避免出现循环调用的情况。
  编写函数时，注意不要出现以下逻辑：Topic A触发函数B，函数B又Publish新的Message到Topic A，从而造成函数无限循环调用。

7.2. MNS主题触发器示例

7.2.1. 示例简介

通过消息服务主题触发器（以下简称“MNS主题触发器”）可以触发函数执行，进而实现在函数中对发布在Topic中的消息进行自定义处理。本文以当MNS Topic收到消息时，通过触发器触发函数执行为例，介绍如何配置MNS主题触发器。

示例场景

您可以配置一个MNS主题触发器，相当于将函数注册为这个MNS Topic的订阅者，当发布者向MNS Topic发布消息时，就会把消息内容通知给函数，同时消息内容作为函数入口的event参数触发函数执行 。

配置触发器

函数计算支持多种方式配置触发器：

- 通过控制台配置触发器
  i. 创建触发器
  ii. 编写函数
  iii. 调试函数
- 通过Funcraft工具配置触发器，详情请参见fun。
- 通过fcli工具配置触发器，详情请参见初次使用fcli。
- 通过SDK配置触发器，详情请参见SDK列表。

7.2.2. 创建触发器

本文介绍如何通过函数计算控制台创建MNS主题触发器。

前提条件

- 创建服务
- 创建函数
- 创建主题

创建触发器

1. 登录函数计算控制台。
2. 在顶部菜单栏，选择地域。
3. 在左侧导航栏，单击**服务/函数**。
4. 找到目标服务下的目标函数，单击函数名称。

5. 单击**触发器**页签，然后单击**创建触发器**。

6. 在**创建触发器**面板填写相关信息。然后，单击**确定**。

参数	操作
服务类型	选择**MNS主题触发器**。
触发器名称	填写自定义的触发器名称。
触发版本/别名	填写触发器版本，默认值为**LATEST**。详情请参见版本简介。
MNS Topic所在区域	选择Topic所在的地域。 强烈建议MNS Topic和函数计算的函数在相同的地域，否则会增加网络延时。
Topic	在列表中选择已创建的Topic。
过滤标签	填写消息过滤标签。 只有收到包含了此处设置的过滤标签字符串的消息时，才会触发函数执行。
重试策略	选择重试策略。取值： ○ **退避重试** ○ **指数衰减** 如何选择重试策略，请参见NotifyStrategy。
Event格式	选择Event格式。取值： ○ **STREAM** ○ **JSON**
角色创建方式	i. 在列表中选择**快捷授权**。 ii. 单击**点击授权**。 iii. 单击**同意授权**。 触发器角色为**AliyunMNSNotificationRole**。

在触发器列表中可以查看创建好的触发器。

后续步骤

1. 编写函数

2. 调试函数

7.2.3. 编写函数

完成MNS主题触发器创建后，您可以开始写入函数代码。本文介绍如何使用函数计算控制台编写函数。

前提条件

创建触发器

编写函数代码（Python）

1. 登录函数计算控制台。
2. 在顶部菜单栏，选择地域。
3. 在左侧导航栏中，单击**服务/函数**。在**服务列表**区域单击目标服务。
4. 在**函数列表**页签找到目标函数，单击函数名称。

5. 单击**代码执行**页签，在代码编辑器中编写代码。本文以Python函数代码为例。以下示例代码可以作为MNS主题触发器的函数模板。

```
import json
import logging
def handler(event, context):
 logger = logging.getLogger()
 logger.info("mns_topic trigger event = {}".format(event))
 # 例如，将事件记录到表格存储。
 return "OK"
```

event格式说明

发布在MNS Topic上的消息根据notifyContentFormat进行处理，即入口函数的event。更多信息，请参见NotifyContentFormat。

- 创建触发器时，若**Event格式**设置为**STREAM**。
  - 当消息中不含消息属性（MessageAttributes）时，Event格式如下。

> **说明** 当消息中不含消息属性（MessageAttributes）时，Event的内容格式为JSON字符串。

```
# 消息正文。
'hello topic'
```

- 当消息中含有消息属性（MessageAttributes）时，Event格式如下。

  > **说明** Event的内容中包含MessageAttributes相关的键值对。更多信息，请参见PublishMessage。

```
{
  "body": "hello topic",
  "attrs": {
    "Extend": "{\\"key\\":\\"value\\"}"
  }
}
```

- 创建触发器时，若event格式设置为JSON。
  - 当消息中不含消息属性（MessageAttributes）时，Event格式如下。

```
{
  "TopicOwner": "118620210433****",
  "Message": "hello topic",
  "Subscriber": "118620210433****",
  "PublishTime": 1550216480040,
  "SubscriptionName": "test-fc-subscibe",
  "MessageMD5": "BA4BA9B48AC81F0F9C66F6C909C3****",
  "TopicName": "test-topic",
  "MessageId": "2F5B3C082B923D4EAC694B76D928****"
}
```

- 当消息中含有消息属性（MessageAttributes）时，Event格式如下。

> **说明** Event的内容中包含MessageAttributes相关的键值对。更多信息，请参见PublishMessage。

```
{
  "key": "value",
  "TopicOwner": "118620210433****",
  "Message": "hello topic",
  "Subscriber": "118620210433****",
  "PublishTime": 1550216302888,
  "SubscriptionName": "test-fc-subscibe",
  "MessageMD5": "BA4BA9B48AC81F0F9C66F6C909C3****",
  "TopicName": "test-topic",
  "MessageId": "2F5B3C281B283D4EAC694B742528****"
}
```

后续步骤

调试函数

7.2.4. 调试函数

完成函数编写后，您需要调试函数以验证代码的正确性。在实际操作过程中当MNS Topic收到消息时会自动触发函数执行。本文介绍如何通过函数计算控制台模拟触发事件调试函数。

前提条件

1. 创建触发器
2. 编写函数

操作步骤

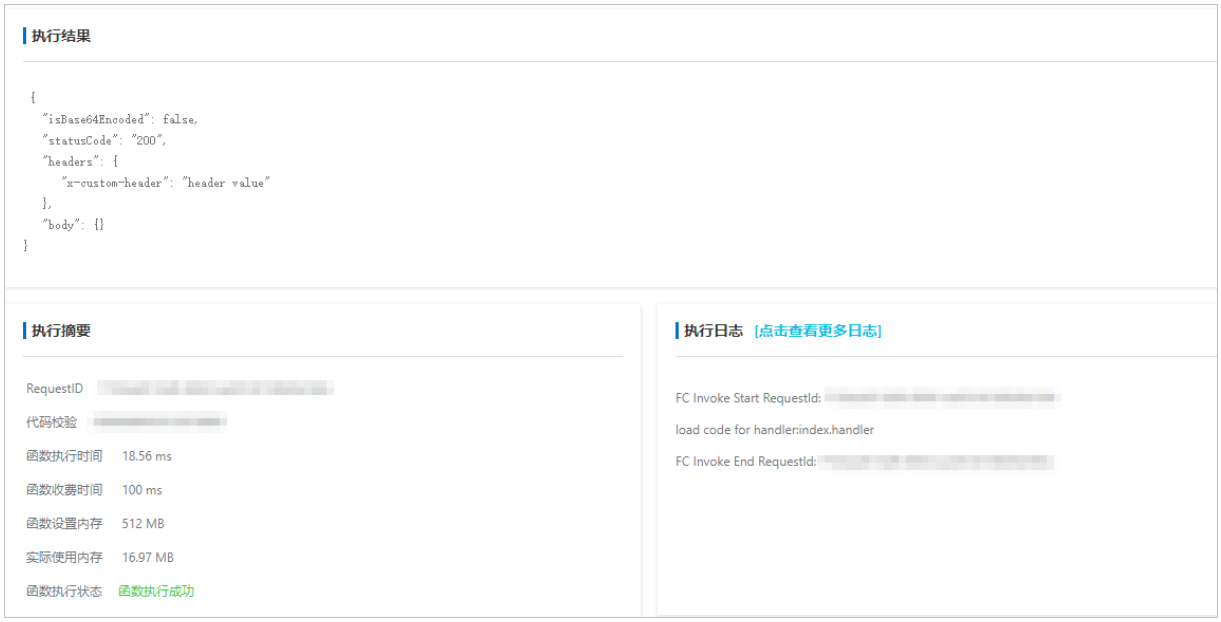
1. 登录函数计算控制台。
2. 在顶部菜单栏，选择地域。
3. 在左侧导航栏，单击**服务/函数**。
4. 找到目标服务下的目标函数，单击函数名称。

5. 在**代码执行**页签，单击**触发事件**。

6. 按照event事件格式在代码编辑器中编写event。更多信息，请参见event格式说明。然后单击**确定**。
7. 在**代码执行**页签下，单击**执行**。

执行结果

在**代码执行**页签可以看到执行成功的提示。



8.日志服务触发器

8.1. 日志服务触发器概述

当日志服务SLS（Log Service）定时获取更新的数据时，通过日志服务触发器触发函数消费增量的日志数据，并完成对数据的自定义加工。

使用场景

日志服务触发器可以实现函数计算与日志服务的集成，集成的使用场景如下：

- 数据清洗、加工场景。
  通过日志服务，快速完成日志采集、加工、查询、分析。

- 数据投递场景。
  为数据的目的端落地提供支撑，构建云上大数据产品间的数据管道。

8.2. 日志服务触发器示例

8.2.1. 示例介绍

通过结合函数计算服务与日志服务可以实现流式的全托管数据加工服务。您可以在函数中对日志数据进行自定义处理。本文以日志服务调用函数计算，函数计算获取日志并打印为例，介绍如何配置日志服务触发器。

示例场景

您可以配置一个日志服务触发器，该触发器将定时获取更新的数据并触发函数执行，增量消费日志服务Logstore中的数据，在函数里完成自定义加工任务（例如，数据清洗和加工）以及将数据投递给第三方服务。本示例中只演示如何获取日志数据并打印。

> **说明** 用于数据加工的函数可以是日志服务提供的模板，也可以是您的自定义函数。

触发器配置流程

函数计算支持多种方式配置触发器：

- 通过控制台配置触发器
  i. 创建触发器
  ii. 编写函数
  iii. 调试函数
- 通过Funcraft工具配置触发器，详情请参见fun。
- 通过fcli工具配置触发器，详情请参见初次使用fcli。
- 通过SDK配置触发器，详情请参见SDK列表。

更多信息

更多示例，请参见开发指南。

8.2.2. 创建触发器

您可以通过创建日志服务触发器将日志服务与函数计算连接起来，当有新日志产生时触发函数执行，对日志进行处理。本文介绍如何在函数计算控制台上创建日志服务触发器。

前提条件

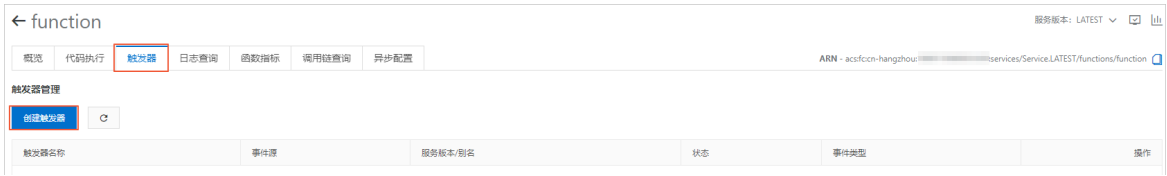
- 创建服务
- 创建函数
- 创建日志项目和日志仓库

> **说明** 需要创建一个日志项目和两个日志仓库。一个日志仓库用于处理日志及数据源，另一个日志仓库用于存储函数计算产生的日志。

创建触发器

1. 登录函数计算控制台。
2. 在顶部菜单栏，选择地域。
3. 在左侧导航栏，单击**服务/函数**。
4. 找到目标服务下的目标函数，单击函数名称。

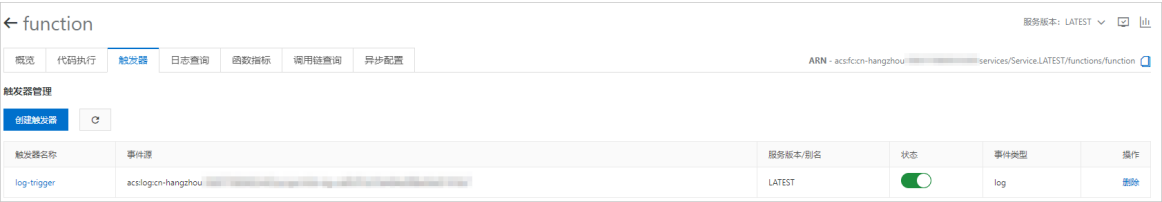
5. 单击**触发器**页签，然后单击**创建触发器**。



6. 在**创建触发器**面板填写相关信息。然后，单击**确定**。

参数	操作
服务类型	选择**日志服务触发器**。
触发器名称	填写自定义的触发器名称。
触发版本/别名	填写触发器版本，默认值为**LATEST**。详情请参见版本简介。
日志项目名称	选择已创建的日志项目。
日志仓库名称	选择已创建的日志仓库，当前触发器会定时从该日志仓库中订阅数据到函数服务进行自定义加工。
触发器日志	选择已创建的日志仓库，日志服务触发函数执行过程的日志会记录到该日志仓库中。
触发间隔	填写日志服务触发函数运行的时间间隔。 取值范围：[3,600]，单位：秒。
重试次数	填写单次触发允许的最大重试次数。 取值范围：[0,100]。
函数配置	如果您想传入自定义参数，可以在此处配置。该参数将作为函数Event的Parameter传入函数。 默认值为空（{}）。
角色创建方式	i. 在列表中选择**快捷授权**。 ii. 单击**点击授权**。 iii. 单击**同意授权**。 触发器角色为**AliyunLogETLRole**。

在触发器列表中可以查看创建好的触发器。



后续步骤

1. 编写函数
2. 调试函数

8.2.3. 编写函数

完成日志触发器创建后，您可以开始编写函数代码。本文介绍如何使用函数计算控制台编写函数，实现日志服务收集增量日志时触发该函数，函数计算获取对应日志，然后打印收集的日志。

前提条件

创建触发器

编写函数代码（Python）

1. 登录函数计算控制台。
2. 在顶部菜单栏，选择地域。
3. 在左侧导航栏，单击**服务/函数**。
4. 找到目标服务下的目标函数，单击函数名称。

5. 单击**代码执行**页签，在代码编辑器中编写代码。本文以Python函数代码为例。以下示例代码可以作为提取大部分逻辑日志的模板。其中 accessKeyId 和 accessKey 可以填写在代码内，也可以从 context 和 creds 中获取。

```
# -*- coding: utf-8 -*-
import logging
import json
from aliyun.log import LogClient
from time import time
def logClient(endpoint, creds):
 logger = logging.getLogger()
 logger.info('creds info')
 logger.info(creds.access_key_id)
 logger.info(creds.access_key_secret)
 logger.info(creds.security_token)
 accessKeyId = 'your accessKeyId'
 accessKey = 'your accessKeyId scr'
 client = LogClient(endpoint, accessKeyId, accessKey)
 return client
def handler(event, context):
 logger = logging.getLogger()
 logger.info('start deal SLS data')
 logger.info(event.decode().encode())
 info_arr = json.loads(event.decode())
 fetchdata(info_arr['source'],context)
 return 'hello world'
```

```
def fetchdata(event,context):
 logger = logging.getLogger()
 endpoint = event['endpoint']
 creds = context.credentials
 client = logClient(endpoint, creds)
 if client == None :
   logger.info("client creat failed")
   return False
 project = event['projectName']
 logstore = event['logstoreName']
 start_cursor = event['beginCursor']
 end_cursor = event['endCursor']
 loggroup_count = 10
 shard_id = event['shardId']
 while True:
   res = client.pull_logs(project, logstore, shard_id, start_cursor, loggroup_count, end_cursor)
   res.log_print()
   next_cursor = res.get_next_cursor()
   if next_cursor == start_cursor :
     break
   start_cursor = next_cursor
  #log_data =  res.get_loggroup_json_list()
 return True
```

event格式说明

event是函数计算的入口参数。具体格式为如下。

```
{
  "parameter": {},
  "source": {
    "endpoint": "http://cn-shanghai-intranet.log.aliyuncs.com",
    "projectName": "log-com",
    "logstoreName": "log-en",
    "shardId": 0,
    "beginCursor": "MTUyOTQ4MDIwOTY1NTk3ODQ2Mw==",
    "endCursor": "MTUyOTQ4MDIwOTY1NTk3ODQ2NA=="
  },
  "jobName": "1f7043ced683de1a4e3d8d70b5a412843d817a39",
  "taskId": "c2691505-38da-4d1b-998a-f1d4bb8c9994",
  "cursorTime": 1529486425
}
```

参数	描述
parameter	您配置触发器时填写的函数配置。
source	设置函数读取的日志块信息。 • endpoint：日志服务Project所属的阿里云地域。 • projectName：日志服务Project名称。 • logstoreName：Logstore名称。 • shardId：Logstore中一个确定的Shard。 • beginCursor：开始消费数据的位置。 • endCursor：停止消费数据的位置。
jobName	日志服务ETL Job名字，函数配置的日志服务触发器对应一个日志服务的ETL Job。
taskId	对于ETL Job而言，taskId是一个确定性的函数调用标识。
cursorTime	最后一条日志到达日志服务端的unix_timestamp。

后续步骤

调试函数

8.2.4. 调试函数

完成函数编写后，您需要调试函数以验证代码的正确性。在实际操作过程中当日志仓库中有新增日志进入时会自动触发函数的执行。本文介绍如何通过函数计算控制台调试函数。

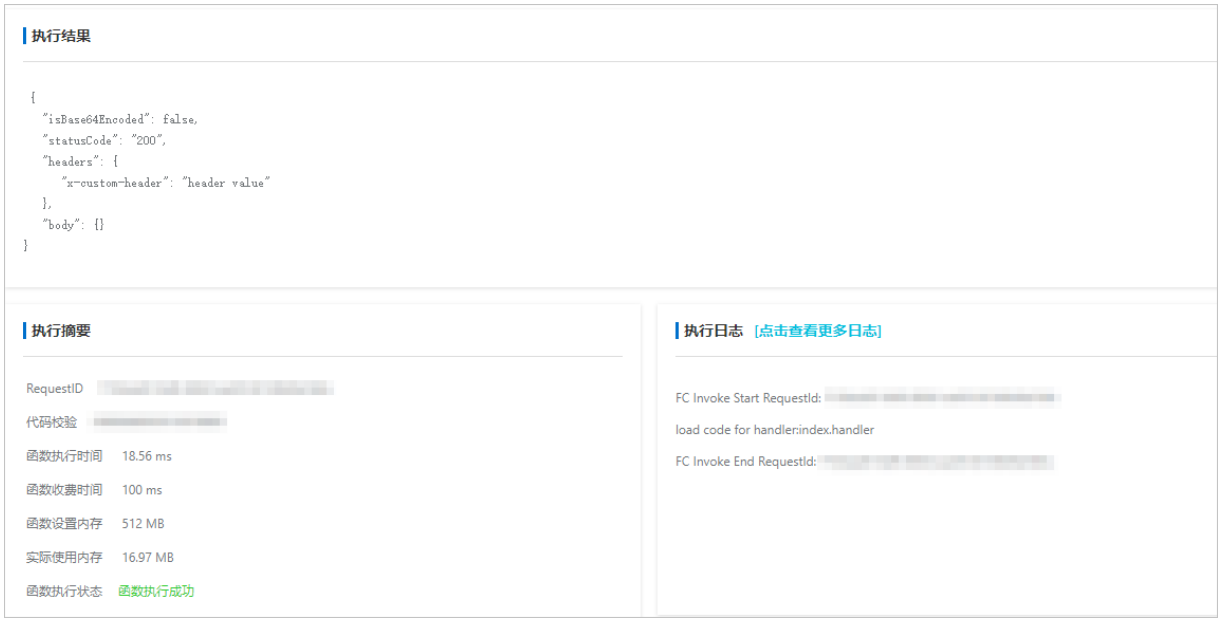
操作步骤

1. 登录函数计算控制台。
2. 在顶部菜单栏，选择地域。
3. 在左侧导航栏，单击**服务/函数**。
4. 找到目标服务下的目标函数，单击函数名称。

5. 在**代码执行**页签，单击**触发事件**。
6. 按照event事件格式在代码编辑器中编写event。更多信息，请参见event格式说明。然后单击**确定**。
7. 在**代码执行**页签下，单击**执行**。

执行结果

在**代码执行**页签可以看到执行成功的提示。


执行结果
{
"isBase64Encoded": false,
"statusCode": "200",
"headers": {
"x-custom-header": "header value"
},
"body": {}
}
执行摘要
RequestID
代码校验
函数执行时间 18.56 ms
函数收费时间 100 ms
函数设置内存 512 MB
实际使用内存 16.97 MB
函数执行状态 函数执行成功
执行日志 [点击查看更多日志]
FC Invoke Start RequestId:
load code for handler:index.handler
FC Invoke End RequestId:

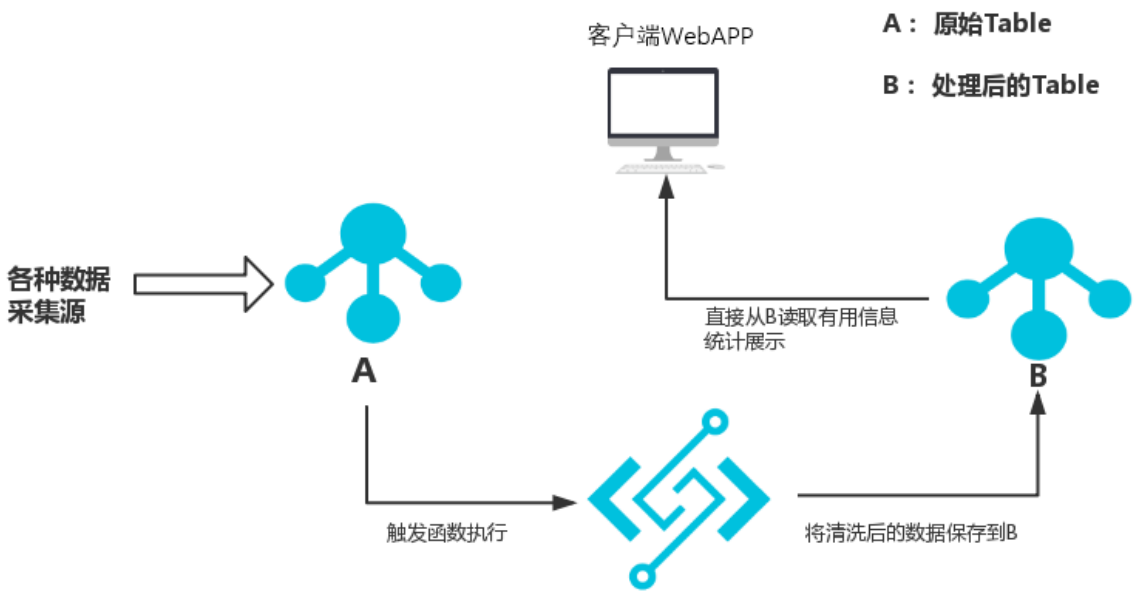
9.Tablestore触发器

9.1. Tablestore触发器概述

阿里云表格存储Tablestore是构建在阿里云飞天分布式系统之上的分布式NoSQL数据存储服务。Tablestore与阿里云函数计算集成，将Tablestore作为事件源接入函数计算，当Tablestore中的数据发生变更时，就会以数据变更信息作为参数触发函数的执行。本文介绍如何在函数计算中使用表格存储触发器（简称Tablestore触发器）。

典型场景

典型的使用场景如下图所示。



原始信号源数据存储到原始Table A，当Table A中的数据发生变更时会触发函数自定义清洗数据，将清洗后的数据存入Table B，您可以直接读取Table B的数据统计展示，完成一个弹性可伸缩的Serverless Web运用。

首次使用Tablestore触发器请务必先阅读地域限制和注意事项。

地域限制

目前支持Tablestore触发器的地域为：华北2（北京）、华东1（杭州）、华东2（上海）、华南1（深圳）、日本（东京）、新加坡、澳大利亚（悉尼）、德国（法兰克福）、中国香港。

注意事项

- 您编写函数的时候，请注意不要出现以下逻辑：表格存储Table A触发函数B，函数B又更新Table A的数据，从而造成函数无限循环调用。
- 若您需要使用内网访问Tablestore触发器对应的函数，您可以使用VPC Endpoint：{instance}.{region}.vpc.tablestore.aliyuncs.com，不可以使用Tablestore内网Endpoint，详情请参见国内VPC功能开启可用性影响说明。
- 触发的函数执行时间不能超过一分钟。
- 若函数执行出现异常，函数将无限重试直到Tablestore中的日志数据过期。

更多信息

- 使用函数计算
- 表格存储触发函数计算示例之Node.js、PHP、Java和C# Runtime
- 联系我们

9.2. Tablestore触发器示例

9.2.1. 示例简介

Tablestore触发器可以在Tablestore中的数据更新时自动触发函数执行，然后根据您的函数的逻辑实现在函数中对这些数据进行自定义处理。本文以使用Tablestore触发器触发函数获取数据为例，介绍如何配置Tablestore触发器。

示例场景

您可以配置一个Tablestore触发器，当Tablestore中有数据变更时，触发函数获取变更的数据。

配置触发器

函数计算支持多种方式配置触发器：

- 通过控制台配置触发器
  i. 创建触发器
  ii. 编写函数
  iii. 调试函数
- 通过Funcraft工具配置触发器，详情请参见fun。
- 通过fcli工具配置触发器，详情请参见初次使用fcli。
- 通过SDK配置触发器，详情请参见SDK列表。

9.2.2. 创建触发器

本文介绍如何在函数计算控制台创建Tablestore触发器。

前提条件

- 创建函数计算资源
  i. 创建服务
  ii. 创建函数
- 创建Tablestore资源
  i. 创建Tablestore实例
  ii. 创建数据表

创建触发器

1. 登录函数计算控制台。
2. 在顶部菜单栏，选择地域。
3. 在左侧导航栏，单击**服务/函数**。
4. 找到目标服务下的目标函数，单击函数名称。

5. 单击**触发器**页签，然后单击**创建触发器**。

6. 在**创建触发器**区域填写相关信息。然后单击**确定**。

参数	操作
服务类型	选择**表格存储触发器**。
触发器名称	自定义填写触发器名称。
触发版本/别名	填写触发器版本或别名，默认值为**LATEST**。详情请参见版本简介。
实例	在列表中选择已创建的Tablestore实例。
表格	在列表中选择已创建的表格。
角色创建方式	如果您已经拥有了一个满足条件的角色，您可以直接使用该角色。如果您没有角色，您可以参考以下步骤创建角色。 i. 在列表中选择**快捷授权**。 ii. 单击**点击授权**。 iii. 单击**同意授权**。 触发器角色为**AliyunTableStoreStreamNotificationRole**。

后续步骤

1. 编写函数
2. 调试函数

9.2.3. 编写函数

完成TableStore触发器创建后，您可以开始编写函数代码。本文介绍如何使用函数计算控制台编写函数。

前提条件

创建触发器

编写函数代码（Python）

1. 登录函数计算控制台。
2. 在顶部菜单栏，选择地域。
3. 在左侧导航栏，单击**服务/函数**。
4. 找到目标服务下的目标函数，单击函数名称。

5. 单击**代码执行**页签，在代码编辑器中编写代码。本文以Python函数代码为例。 如果您想使用其他运行环境，代码示例请参见表格存储触发函数计算示例之 Nodejs/Php/Java/C# Runtime。

```
import logging
import cbor
import json
def get_attribute_value(record, column):
    attrs = record[u'Columns']
    for x in attrs:
        if x[u'ColumnName'] == column:
            return x['Value']
def get_pk_value(record, column):
    attrs = record[u'PrimaryKey']
    for x in attrs:
        if x['ColumnName'] == column:
            return x['Value']
def handler(event, context):
    logger = logging.getLogger()
    logger.info("Begin to handle event")
    #records = cbor.loads(event)
    records = json.loads(event)
    for record in records['Records']:
        logger.info("Handle record: %s", record)
        pk_0 = get_pk_value(record, "pk_0")
        attr_0 = get_attribute_value(record, "attr_0")
    return 'OK'
```

event格式说明

表格存储触发器使用CBOR格式对增量数据进行编码构成函数计算的event，增量数据的具体数据格式如下：

```
{
  "Version": "string",
  "Records": [
    {
      "Type": "string",
      "Info": {
        "Timestamp": int64
      },
      "PrimaryKey": [
        {
          "ColumnName": "string",
          "Value": formated_value
        }
      ],
      "Columns": [
        {
          "Type": "string",
          "ColumnName": "string",
          "Value": formated_value,
          "Timestamp": int64
        }
      ]
    }
  ]
}
```

- Version： payload版本号，目前为Sync-v1。类型为String。
- Records： 数据表中的增量数据行数组。
  - Type： 数据行类型，包含PutRow 、UpdateRow和DeleteRow。类型为String。
  - Info： 数据行基本信息。
    - Timestamp：该行的最后修改UTC时间。类型为Int64。
  - PrimaryKey： 主键列数组。
    - ColumnName： 主键列名称。类型为String。
    - Value： 主键列内容。类型为formated_value，支持Integer、String和Blob。
  - Colunms： 属性列数组。
    - Type： 属性列类型，包含Put、DeleteOneVersion和DeleteAllVersions。类型为String。
    - ColumnName： 属性列名称。类型为String。
    - Value： 属性列内容。类型为formated_value，支持Integer、Boolean、Double、String和Blob。
    - Timestamp： 属性列最后修改UTC时间。类型为Int64。

Event示例如下：

```
{
  "Version": "Sync-v1",
  "Records": [
    {
      "Type": "PutRow",
      "Info": {
        "Timestamp": 1506416585740836
      },
      "PrimaryKey": [
        {
          "ColumnName": "pk_0",
          "Value": 1506416585881590900
        },
        {
          "ColumnName": "pk_1",
          "Value": "2017-09-26 17:03:05.8815909 +0800 CST"
        },
        {
          "ColumnName": "pk_2",
          "Value": 1506416585741000
        }
      ],
      "Columns": [
        {
          "Type": "Put",
          "ColumnName": "attr_0",
          "Value": "hello_table_store",
          "Timestamp": 1506416585741
        },
        {
          "Type": "Put",
          "ColumnName": "attr_1",
          "Value": 1506416585881590900,
          "Timestamp": 1506416585741
        }
      ]
    }
  ]
}
```

后续步骤

调试函数

9.2.4. 调试函数

完成函数编写后，您需要调试函数以验证代码的正确性。在实际操作过程中当Tablestore中有数据更新时会自动触发函数执行。本文介绍如何通过函数计算控制台模拟触发事件调试函数。

前提条件

1. 创建触发器
2. 编写函数

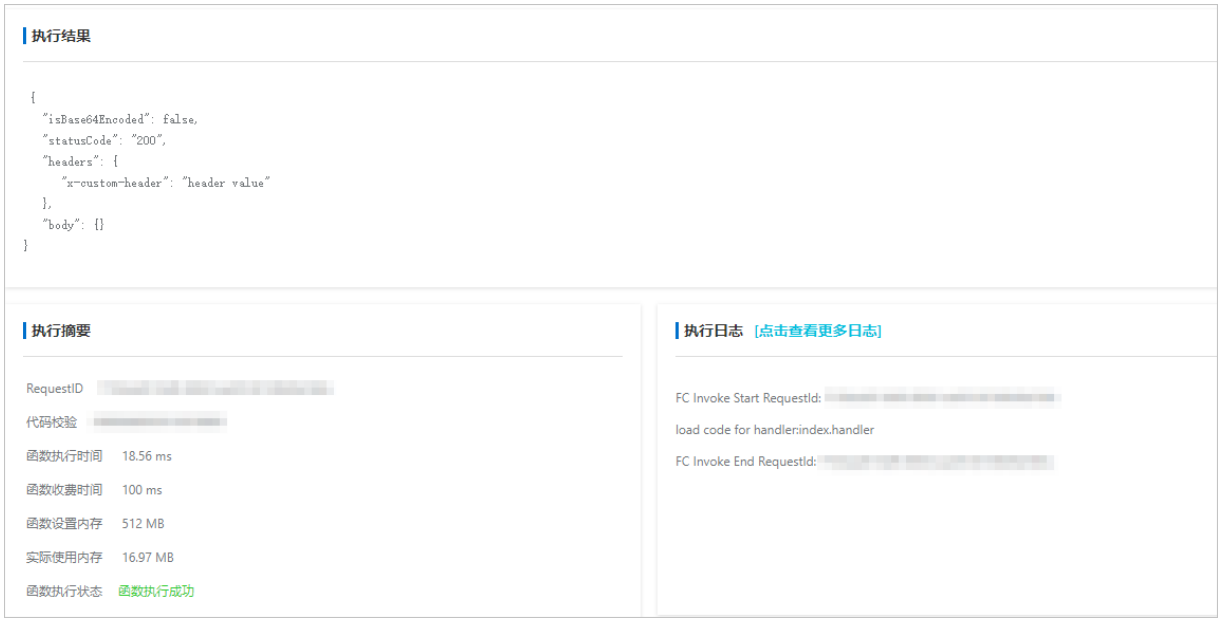
操作步骤

1. 登录函数计算控制台。
2. 在顶部菜单栏，选择地域。
3. 在左侧导航栏，单击**服务/函数**。
4. 找到目标服务下的目标函数，单击函数名称。

5. 在**代码执行**页签，单击**触发事件**。
6. 按照event事件格式在代码编辑器中编写event。更多信息，请参见event格式说明。然后单击**确定**。
7. 在**代码执行**页签下，单击**执行**。

执行结果

在**代码执行**页签可以看到执行成功的提示。


执行结果
{
"isBase64Encoded": false,
"statusCode": "200",
"headers": {
"x-custom-header": "header value"
},
"body": {}
}
执行摘要
RequestID
代码校验
函数执行时间 18.56 ms
函数收费时间 100 ms
函数设置内存 512 MB
实际使用内存 16.97 MB
函数执行状态 函数执行成功
执行日志 [点击查看更多日志]
FC Invoke Start RequestId:
load code for handler:index.handler
FC Invoke End RequestId:

10.CDN事件触发器

10.1. CDN事件触发器概述

阿里云内容分发网络CDN（Content Delivery Network）和函数计算无缝集成，您可以编写函数对CDN事件进行自定义处理。当CDN系统捕获到指定类型的、满足过滤条件的事件后，通过CDN事件触发器触发函数执行。

背景信息

阿里云CDN是建立并覆盖在承载网之上、由分布在不同区域的边缘节点服务器群组成的分布式网络。阿里云CDN可以替代传统以Web Server为中心的数据传输模式，将源站资源缓存到阿里云全国各地的边缘服务器，供您就近快速获取，提升用户体验，降低源站压力。在函数计算中通过配置内容分发网络事件触发器（以下简称“CDN事件触发器”）集成CDN服务可以实现您对CDN的各类事件进行自定义处理。例如，您可以设置函数和对应的CDN触发器来处理www.taobao.com域名下的资源刷新事件，当该域名下有资源刷新事件时，CDN事件触发器会自动触发函数执行。

使用场景

CDN事件触发器可以实现函数计算与CDN服务的集成，集成的使用场景如下：

- CDN在预热（CachedObjectsPushed）和刷新（CachedObjectsRefreshed）用户数据后，通过触发器执行函数。用户可以及时得知资源预热刷新的状态并进行下一步处理，避免不断轮询列表查询最新状态。
- 当在CDN上发现违禁内容（CachedObjectsBlocked）时，通过触发器执行函数直接去源站删除资源。您不需要等待CDN团队响应，可以及时去源站删除资源。
- 日志文件生成后（LogFileCreated），通过触发器执行函数处理日志。您不需要长时间等待日志，可以及时转存或处理日志。
- 当某加速域名被停用（CdnDomainStopped）或者被启用（CdnDomainStarted），通过触发器执行函数及时作出相应的处理。

CDN事件定义

当CDN系统捕获到相关事件后，会将事件信息编码为JSON字符串，传递给函数进行处理。CDN事件触发器当前支持的事件及版本如下表所示。

事件名称	事件版本	过滤参数	参考文档
CachedObjectsRefreshed	1.0.0	domain	刷新节点上的文件内容
CachedObjectsBlocked	1.0.0	domain	CDN封禁资源
CachedObjectsPushed	1.0.0	domain	预热源站内容到缓存节点
LogFileCreated	1.0.0	domain	获取加速域名的日志信息
CdnDomainStarted	1.0.0	domain	启用状态为停用的加速域名
CdnDomainStopped	1.0.0	domain	停用加速域名
CdnDomainAdded	1.0.0	domain	添加加速域名
CdnDomainDeleted	1.0.0	domain	删除已添加的加速域名

10.2. CDN事件触发器示例

10.2.1. 示例简介

阿里云CDN和函数计算无缝集成，您可以为CDN的各种事件设置处理函数，并允许用户通过事件中的域名等参数进行过滤，只接收指定域名下的数据。当CDN系统捕获到指定类型的、满足过滤条件的事件后，会自动调用函数处理。

示例场景

本示例使用OSS作为源站，您也可以使用自己的Web服务器。

您可以在OSS上创建存储空间，获取域名作为源站。然后为源站添加加速域名。CDN事件触发器配置完成后，每当CDN服务在您过滤器指定的加速域名下生成一个离线日志文件，CDN事件就会触发函数计算转存CDN离线日志。

通过示例您将了解如何使用CDN事件触发器处理CDN事件。

配置触发器

函数计算支持多种方式配置触发器：

- 通过控制台配置触发器
  - 创建触发器
  - 编写函数
  - 测试函数
- 通过Funcraft工具配置触发器，详情请参见fun。
- 通过fcli工具配置触发器，详情请参见初次使用fcli。
- 通过SDK配置触发器，详情请参见SDK列表。

10.2.2. 创建触发器

本文介绍如何在函数计算控制台上创建CDN事件触发器。

前提条件

1. 创建服务
2. 创建函数
3. 创建存储空间
4. 添加加速域名

操作步骤

1. 登录函数计算控制台。
2. 在顶部菜单栏，选择地域。
3. 在左侧导航栏，单击**服务/函数**。
4. 找到目标服务下的目标函数，单击函数名称。

5. 单击**触发器**页签，然后单击**创建触发器**。

6. 在**创建触发器**面板填写相关信息。然后，单击**确定**。

参数	操作
服务类型	选择**CDN 事件触发器**。
触发器名称	填写自定义的触发器名称。
触发版本/别名	填写触发器版本，默认值为**LATEST**。更多信息，请参见版本简介。
触发事件	选择一个触发事件类型。 如何选择触发事件类型，请参见CDN事件定义。 本示例中选择**LogFileCreated**。
触发事件版本	填写版本号*1.0.0*。 **说明** 目前仅支持1.0.0事件版本。
备注	填写触发器备注信息。
过滤器	○ **过滤参数**：选择**domain**。 ○ **过滤参数值**：填写您在CDN控制台添加域名时设置的加速域名名称。
角色创建方式	i. 在列表中选择**快捷授权**。 ii. 单击**点击授权**。 iii. 单击**同意授权**。 触发器角色为**AliyunCDNEventNotificationRole**。

在触发器列表中可以看到创建好的触发器。

后续步骤

1. 编写函数
2. 测试函数

10.2.3. 编写函数

完成CDN事件触发器后，您可以开始写入函数代码。本文介绍如何使用函数计算控制台编写函数，实现CDN事件触发函数计算转存CDN离线日志。

前提条件

创建触发器

操作步骤

1. 登录函数计算控制台。
2. 在顶部菜单栏，选择地域。
3. 在左侧导航栏，单击**服务/函数**。
4. 找到目标服务下的目标函数，单击函数名称。

5. 单击**代码执行**页签，在代码编辑器中编写代码。
   本文以Python函数代码为例。

```
import json
import oss2
import requests
def handler(event, context):
  creds = context.credentials
  auth = oss2.StsAuth(creds.access_key_id, creds.access_key_secret, creds.security_token)
  bucket = oss2.Bucket(auth, 'your endpoint', 'your bucket name')
 # Parse the event to get the source object info.
 eventObj = json.loads(event)["events"]
 fileUrl = eventObj[0]["eventParameter"]["filePath"]
 file = requests.get(fileUrl)
 result = bucket.put_object("your log name", fileUrl)
 return result.status
```

event格式说明

event是函数计算的入口参数，具体格式为如下。其中，对于不同的CDN事件类型，eventParameter中包含的键值对不同。

```
{
  "events": [
    {
      "eventName": "LogFileCreated",//事件类型
      "eventSource": "cdn",//事件源名称
      "region": "cn-hangzhou",//地域，默认为“cn-hangzhou”。
      "eventVersion": "1.0.0",//事件版本
      "eventTime": "2018-06-14T15:31:49+08:00",//事件发生时间
      "traceId": "c6459282-6a4d-4413-894c-e4ea39686738" //事件源传递过来的ID，用于排查问题。
      "userIdentity": {
        "aliUid": "1xxxxxxxxxxx" //用户ID
      },
      "resource": {
        "domain": "example.com"//域名
      },
      "eventParameter": {
        "key": "value"
      },
    }
  ]
}
```

- LogFileCreated事件的event示例。

替换 filePath 为您CDN日志的路径，或任何测试文件。

```
{
  "events": [
    {
      "eventName": "LogFileCreated",//事件类型
      "eventSource": "cdn",//事件源名称
      "region": "cn-hangzhou",//地域，默认为“cn-hangzhou”。
      "eventVersion": "1.0.0",//事件版本
      "eventTime": "2018-06-14T15:31:49+08:00",//事件发生时间
      "traceId": "c6459282-6a4d-4413-894c-e4ea39686738" //事件源传递过来的ID，用于排查问题。
      "userIdentity": {
        "aliUid": "1xxxxxxxxxxx" //用户ID
      },
      "resource": {
        "domain": "example.com"//域名
      },
      "eventParameter": {
        "domain": "example.com",//域名
        "endTime": 1528959900,//日志文件的结束时间
        "fileSize": 1788115,//日志文件大小
        "filePath": "http://cdnlog.cn-hangzhou.oss.aliyun-inc.com/www.aliyun.com/2017_12_27/www.aliyun.com_2017_12_27_0800_0900.gz?OSSAccessKeyId=xxxx&Expires=xxxx&Signature=xxxx",//日志文件地址
        "startTime": 1528959600//日志文件的起始时间
      },
    }
  ]
}
```

- CachedObjectsRefreshed、CachedObjectsPushed和CachedObjectsBlocked事件的event示例。

```
{
  "events": [
    {
      "eventName": "CachedObjectsRefreshed",//事件类型
      "eventVersion": "1.0.0", // 事件版本，目前都是1.0.0版本。
      "eventSource": "cdn", // 事件源名称
      "region": "cn-hangzhou", // 地域，默认为"cn-hangzhou"。
      "eventTime": "2018-03-16T14:19:55+08:00",//事件发生时间
      "traceId": "cf89e5a8-7d59-4bb5-a33e-4c3d08e25acf",//事件源传递过来的ID，用于排查问题
      "resource": {
        "domain": "example.com"//资源所在的域名
      },
      "eventParameter": {
        "objectPath": [
          "/2018/03/16/13/33b430c57e7.mp4",//资源标识
          "/2018/03/16/14/4ff6b9bd54d.mp4"//资源标识
        ],
        "createTime": 1521180769,//刷新开始时间
        "domain": "example.com",//资源所在的域名
        "completeTime": 1521180777,//刷新结束时间
        "objectType": "File", //刷新类型，取值为File，Directory
        "taskId": 2089687230 //资源刷新任务ID
      },
      "userIdentity": {
        "aliUid": "1xxxxxxxxxx" //用户ID
      }
    }
  ]
}
```

- CdnDomainStarted和CdnDomainStopped事件的event示例。

```
{ "events": [
   {
     "eventName": "CdnDomainStarted", //事件类型
     "eventVersion": "1.0.0", //事件版本
     "eventSource": "cdn", //事件源名称
     "region": "cn-hangzhou", //区域，默认为“cn-hangzhou”。
     "eventTime": "2018-03-16T14:19:55+08:00",//事件发生时间
     "traceId": "cf89e5a8-7d59-4bb5-a33e-4c3d08e25acf",//事件源传递过来的ID，用于排查问题。
     "resource": {
        "domain": "chongshi.alicdn.com"
     },
     "eventParameter": {
        "domain": "chongshi.alicdn.com",
        "status": "online" //Domain状态
     },
     "userIdentity": {
        "aliUid": "12345678"//用户ID
     }
   }
 ]
}
```

- CdnDomainAdded和CdnDomainDeleted事件的event示例。

```
{ "events": [
   {
     "eventName": "CdnDomainStarted", //事件类型
     "eventVersion": "1.0.0", //事件版本
     "eventSource": "cdn", //事件源名称
     "region": "cn-hangzhou", //地域，默认为“cn-hangzhou”。
     "eventTime": "2018-03-16T14:19:55+08:00",//事件发生时间
     "traceId": "cf89e5a8-7d59-4bb5-a33e-4c3d08e25acf",//事件源传递过来的ID，用于排查问题。
     "resource": {
        "domain": "chongshi.alicdn.com"
     },
     "eventParameter": {
        "domain": "chongshi.alicdn.com",
     },
     "userIdentity": {
        "aliUid": "12345678"//用户ID
     }
   }
 ]
}
```

后续步骤

测试函数

10.2.4. 测试函数

完成函数编写后，您需要调试函数以验证代码的正确性。在实际操作过程中当发生CDN事件时，触发器会自动触发函数的执行。本文介绍如何通过函数计算控制台模拟CDN事件来调试函数。

前提条件

1. 创建触发器
2. 编写函数

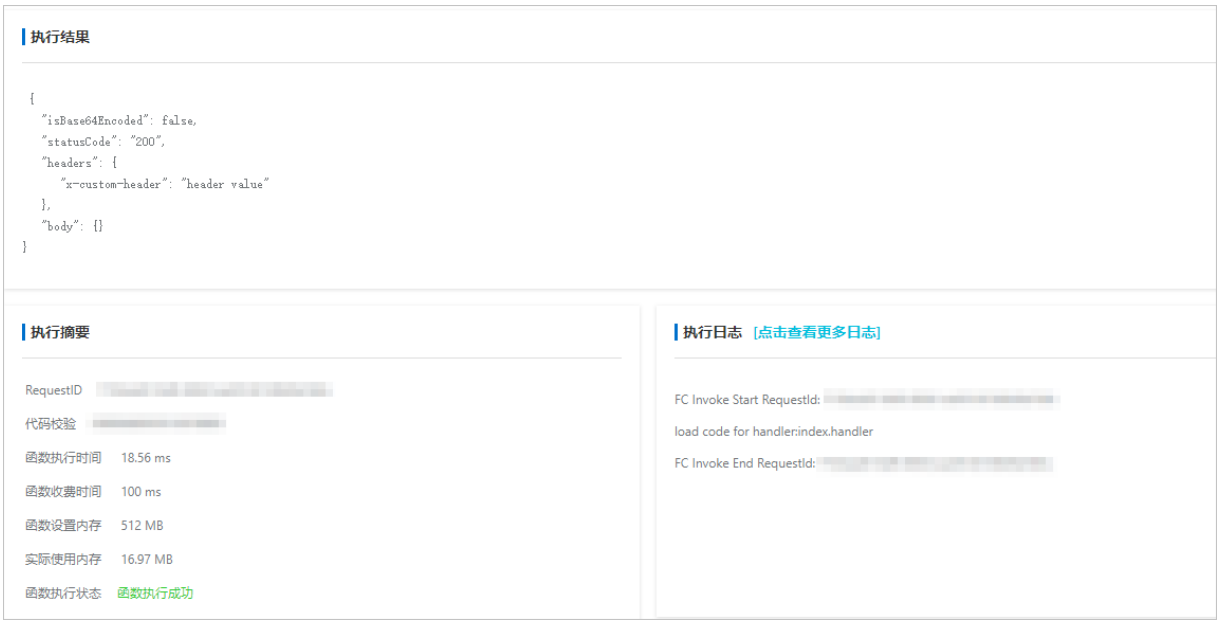
操作步骤

1. 登录函数计算控制台。
2. 在顶部菜单栏，选择地域。
3. 在左侧导航栏，单击**服务/函数**。
4. 找到目标服务下的目标函数，单击函数名称。

5. 在**代码执行**页签，单击**触发事件**。
6. 按照event事件格式在代码编辑器中编写event。更多信息，请参见event格式说明。然后单击**确定**。
7. 在**代码执行**页签下，单击**执行**。

执行结果

在**代码执行**页签可以看到执行成功的提示。



11.定时触发器

11.1. 定时触发器概述

函数计算支持配置定时触发器（Time Trigger），可以在指定的时间点自动触发函数执行。

使用场景

定时触发器会在指定时间自动触发函数执行。定时任务的场景非常广泛，包括但不限于以下使用场景：

- 批量数据的定时处理，例如每1小时收集全量数据并生成报表。
- 日常行为的调度，例如整点发送优惠券。
- 与业务解耦的异步任务，例如每天0点清理数据。

11.2. 定时触发器示例

11.2.1. 示例简介

您可以配置定时触发器实现在指定的时间点触发函数执行。

示例场景

您可以在函数计算控制台配置每5分钟触发一次函数执行的定时触发器。

配置触发器

函数计算支持多种方式配置触发器：

- 通过函数计算控制台配置触发器
  i. 创建触发器
  ii. 编写函数
  iii. 调试函数
- 通过Funcraft工具配置触发器，详情请参见fun。
- 通过fcli工具配置触发器，详情请参见初次使用fcli。
- 通过SDK配置触发器，详情请参见SDK列表。

11.2.2. 创建触发器

本文介绍如何在函数计算控制台创建每5分钟触发一次函数执行的定时触发器。

前提条件

- 创建服务
- 创建函数

创建触发器

1. 登录函数计算控制台。
2. 在顶部菜单栏，选择地域。
3. 在左侧导航栏，单击**服务/函数**。
4. 找到目标服务下的目标函数，单击函数名称。

5. 单击**触发器**页签，然后单击**创建触发器**。

6. 在**创建触发器**区域填写相关信息。然后单击**确定**。

参数	操作	本文示例
服务类型	选择**定时触发器**。	定时触发器
触发器名称	填写自定义的触发器名称。	my_trigger
触发版本/别名	填写触发器版本，默认值为**LATEST**。详情请参见版本简介。	LATEST
时间配置	根据需要选择配置方式： ○ **时间间隔**：在文本框输入正整数n，表示每n分钟触发一次函数执行。 ○ **Cron表达式**：在文本框中输入Cron表达式，到达指定的时间点触发一次函数执行。标准的Cron表达式形式为：Seconds Minutes Hours Day-of-month Month Day-of-week。 **说明** Cron以UTC时间运行，即北京时间减去8个小时。	时间间隔：5分钟
启动触发器	打开**启动触发器**开关。	打开开关
触发消息	输入自定义的参数。该触发消息将会作为event中payload的值。	{awesome-fc}

Cron表达式（Seconds Minutes Hours Day-of-month Month Day-of-week）的字段说明如下：

○ 字段说明

字段名	取值范围	允许的特殊字符
Seconds	0 ~ 59	无
Minutes	0 ~ 59	, - * /

字段名	取值范围	允许的特殊字符
Hours	0 ~ 23	, - * /
Day-of-month	1 ~ 31	, - * ? /
Month	1 ~ 12或JAN ~ DEC	, - * /
Day-of-week	1 ~ 7或MON ~ SUN	, - * ?

- 特殊字符说明

字符名	定义	示例
*	表示任一，每一。	在Minutes字段中：0表示每分钟的0秒都执行。
,	表示列表值	在Day-of-week字段中：`MON,WED,FRI`表示星期一，星期三和星期五。
-	表示一个范围	在Hours字段中：10-12表示UTC时间从10点到12点。
?	表示不确定的值	与其他指定值一起使用。例如，如果指定了一个特定的日期，但您不在乎它是星期几，那么在Day-of-week字段中就可以使用。
/	表示一个值的增加幅度，n/m表示从n开始，每次增加m。	在minute字段中：3/5表示从3分开始，每隔5分钟执行。

- 常用示例
  下表的第一列为北京时间，第二列为北京时间对应的UTC时间（即北京时间减去8小时）。例如，您希望工作流在北京时间12:00被调度，则需先将时间转换为UTC时间4:00，对应的Cron表达式为0 0 4 * * *。

示例（北京时间）	转换为UTC时间	表达式
每天12:00调度函数	每天4:00调度函数	0 0 4 * * *
每天12:30调度函数	每天4:30调度函数	0 30 4 * * *
每小时的26分，29分，33分调度函数	每小时的26分，29分，33分调度函数	0 26,29,33 * * * *
周一到周五的每天12:30调度函数	周一到周五的每天4:30调度函数	0 30 4 ? * MON-FRI
周一到周五的每天12:00 ~ 14:00每5分钟调度函数	周一到周五的每天4:00 ~ 6:00每5分钟调度函数	0 0/5 4-6 ? * MON-FRI
一月到四月每天12:00调度函数	一月到四月每天4:00调度函数	0 0 4 ? JAN,FEB,MAR,APR *

在触发器列表中可以看到创建好的触发器。

后续步骤

1. 编写函数
2. 调试函数

11.2.3. 编写函数

完成定时触发器创建后，您可以开始编写函数代码。本文介绍如何使用函数计算控制台编写函数，实现定时触发函数执行。

前提条件

创建触发器

编写函数代码（Python）

1. 登录函数计算控制台。
2. 在顶部菜单栏，选择地域。
3. 在左侧导航栏，单击**服务/函数**。
4. 找到目标服务下的目标函数，单击函数名称。

5. 单击**代码执行**页签，在代码编辑器中编写代码。本文以Python函数代码为例，示例代码如下。

```
import json
import logging
logger = logging.getLogger()
def handler(event, context):
  logger.info('event: %s', event)
  evt = json.loads(event)
  triggerName = evt["triggerName"]
  triggerTime = evt["triggerTime"]
  message = evt["payload"]
  creds = context.credentials
  logger.info('access_key_id: %s', creds.access_key_id)
  logger.info("message = %s", message)
```

event格式说明

定时触发器会按照以下event格式来触发函数。

```
{
  "triggerTime":"2018-02-09T05:49:00Z",
  "triggerName":"my_trigger",
  "payload":"awesome-fc"
}
```

参数	描述
triggerTime	函数被触发的时间。
triggerName	定时触发器的名称。
payload	您在触发器配置里自定义的输入参数。

后续步骤

调试函数

11.2.4. 调试函数

完成函数编写后，您需要调试函数以验证代码的正确性。在实际操作过程中当到达指定的时间，定时触发器会自动触发函数执行。本文介绍如何通过函数计算控制台调试函数。

前提条件

1. 创建触发器
2. 编写函数

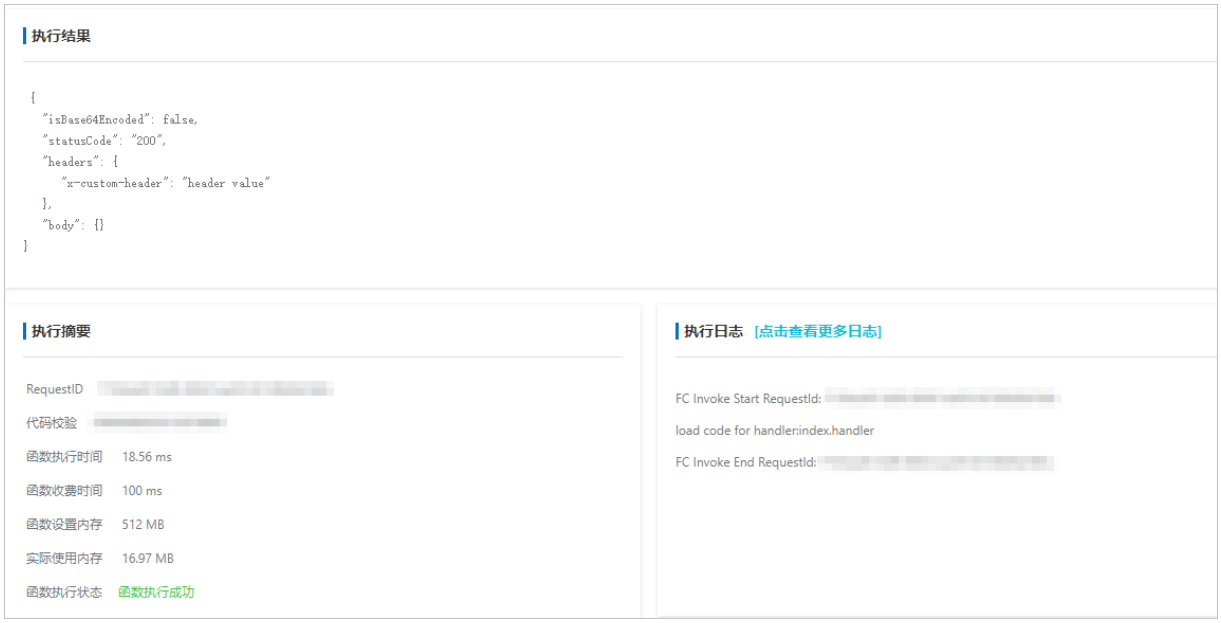
操作步骤

1. 登录函数计算控制台。
2. 在顶部菜单栏，选择地域。
3. 在左侧导航栏，单击**服务/函数**。
4. 找到目标服务下的目标函数，单击函数名称。

5. 在**代码执行**页签，单击**触发事件**。
6. 按照event事件格式在代码编辑器中编写event。更多信息，请参见event格式说明。然后单击**确定**。
7. 在**代码执行**页签下，单击**执行**。

执行结果

在**代码执行**页签可以看到执行成功的提示。



12.API网关触发器

12.1. API网关触发器概述

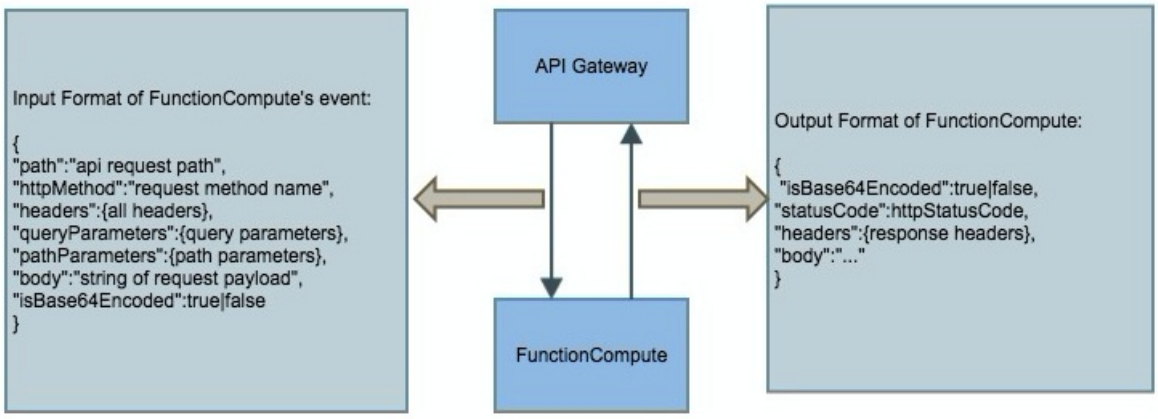
函数计算支持API网关作为事件源，当有请求到达后端服务设置为函数计算的API网关时，API网关会触发函数的执行。函数计算会将执行结果返回给API网关。

使用场景

API网关触发器与HTTP触发器类似，可应用于搭建Web应用。相较于HTTP触发器，您可以使用API网关进行IP白名单或黑名单设置等高级操作。

实现原理

API网关调用函数计算服务时，会将API的相关数据转换为Map形式传给函数计算服务。函数计算服务处理后，按照Output Format格式返回statusCode、headers、body等相关数据。API网关再将函数计算返回的内容映射到statusCode、header、body等位置返回给客户端。



12.2. API网关触发器示例

12.2.1. 示例简介

函数计算支持API网关作为事件源，本文介绍配置API网关触发函数计算执行的流程。配置完成后，函数将获取API网关传入的参数，并将参数的处理结果返回给API网关。

配置流程

1. 编写函数
2. 调试函数
3. 创建API

更多信息

问题诊断

12.2.2. 编写函数

本文介绍如何使用函数计算控制台编写函数。本文提供的代码仅为示例，您可以根据实际情况编写业务代码。

前提条件

1. 创建服务
2. 创建函数

event格式说明

函数计算作为API网关的后端服务时，API网关会把请求参数通过一个固定结构作为event传给函数计算，函数计算通过如下结构去获取参数，然后进行处理。

```
{
  "path":"api request path",
  "httpMethod":"request method name",
  "headers":{all headers,including system headers},
  "queryParameters":{query parameters},
  "pathParameters":{path parameters},
  "body":"string of request payload",
  "isBase64Encoded":"true|false, indicate if the body is Base64-encode"
}
```

> 说明
>
> - 如果 `isBase64Encoded` 的值为 `true` ，表示API网关传给函数计算的body内容已Base64编码。函数计算需要先对body内容Base64解码后再处理。
> - 如果 `isBase64Encoded` 的值为 `false` ，表示API网关没有对body内容Base64编码，在函数中可以直接获取body内容。

编写函数代码

1. 登录函数计算控制台。
2. 在顶部菜单栏，选择地域。
3. 在左侧导航栏，单击**服务/函数**。
4. 找到目标服务下的目标函数，单击函数名称。

5. 单击**代码执行**页签，在代码编辑器中编写代码。不同语言的示例代码如下：
   - Node.js

```
module.exports.handler = function(event, context, callback) {
  var event = JSON.parse(event);
  var content = {
    path: event.path,
    method: event.method,
    headers: event.headers,
    queryParameters: event.queryParameters,
    pathParameters: event.pathParameters,
    body: event.body
  // 您可以在这里编写您自己的逻辑。
  }
  var response = {
      isBase64Encoded: false,
      statusCode: '200',
      headers: {
        'x-custom-header': 'header value'
      },
      body: content
    };
  callback(null, response)
};
```

- Python

```
# -*- coding: utf-8 -*-
import json
def handler(event, context):
  event = json.loads(event)
  content = {
    'path': event['path'],
    'method': event['httpMethod'],
    'headers': event['headers'],
    'queryParameters': event['queryParameters'],
    'pathParameters': event['pathParameters'],
    'body': event['body']
  }
  # 您可以在这里编写您自己的逻辑。
  rep = {
    "isBase64Encoded": "false",
    "statusCode": "200",
    "headers": {
      "x-custom-header": "no"
    },
    "body": content
  }
  return json.dumps(rep)
```

- PHP

```php
<?php
function handler($event, $context) {
  $event   = json_decode($event, $assoc = true);
  $content = [
    'path'        => $event['path'],
    'method'      => $event['httpMethod'],
    'headers'     => $event['headers'],
    'queryParameters' => $event['queryParameters'],
    'pathParameters'  => $event['pathParameters'],
    'body'        => $event['body'],
  ];
  $rep = [
    "isBase64Encoded" => "false",
    "statusCode"    => "200",
    "headers"       => [
      "x-custom-header" => "no",
    ],
    "body"        => $content,
  ];
  return json_encode($rep);
}
```

- Java
  使用Java编程时，必须要实现一个类，需要实现函数计算预定义的handler，目前有两个预定义的handler可以实现（任选其一即可）。函数计算Java运行环境，请参见Java运行环境。
  - （推荐）使用PojoRequestHandler<I, O> handler。

```
import com.aliyun.fc.runtime.Context;
import com.aliyun.fc.runtime.PojoRequestHandler;
import java.util.HashMap;
import java.util.Map;
public class ApiTriggerDemo implements PojoRequestHandler<ApiRequest, ApiResponse> {
  public ApiResponse handleRequest(ApiRequest request, Context context) {
    // 获取API请求信息。
    context.getLogger().info(request.toString());
    String path = request.getPath();
    String httpMethod = request.getHttpMethod();
    String body = request.getBody();
    context.getLogger().info("path：" + path);
    context.getLogger().info("httpMethod：" + httpMethod);
    context.getLogger().info("body：" + body);
    // 您可以在这里编写您自己的逻辑。
    // API返回示例。
    Map headers = new HashMap();
    boolean isBase64Encoded = false;
    int statusCode = 200;
    String returnBody = "";
    return new ApiResponse(headers,isBase64Encoded,statusCode,returnBody);
  }
}
```

- 两个 `POJO` 类、 `ApiRequest` 类和 `ApiResponse` 类定义如下。

  > **说明** `POJO` 类的 `set()` 和 `get()` 方法要写全。

```
import java.util.Map;
public class ApiRequest {
  private String path;
  private String httpMethod;
  private Map headers;
  private Map queryParameters;
  private Map pathParameters;
  private String body;
  private boolean isBase64Encoded;
  @Override
  public String toString() {
    return "Request{" +
        "path='" + path + '\'' +
```

```
        ", httpMethod='" + httpMethod + '\'' +
        ", headers=" + headers +
        ", queryParameters=" + queryParameters +
        ", pathParameters=" + pathParameters +
        ", body='" + body + '\'' +
        ", isBase64Encoded=" + isBase64Encoded +
        '}';
}
public String getPath() {
  return path;
}
public void setPath(String path) {
  this.path = path;
}
public String getHttpMethod() {
  return httpMethod;
}
public void setHttpMethod(String httpMethod) {
  this.httpMethod = httpMethod;
}
public Map getHeaders() {
  return headers;
}
public void setHeaders(Map headers) {
  this.headers = headers;
}
public Map getQueryParameters() {
  return queryParameters;
}
public void setQueryParameters(Map queryParameters) {
  this.queryParameters = queryParameters;
}
public Map getPathParameters() {
  return pathParameters;
}
public void setPathParameters(Map pathParameters) {
  this.pathParameters = pathParameters;
}
public String getBody() {
  return body;
```

```
    }
    public void setBody(String body) {
        this.body = body;
    }
    public boolean getIsBase64Encoded() {
        return this.isBase64Encoded;
    }
    public void setIsBase64Encoded(boolean base64Encoded) {
        this.isBase64Encoded = base64Encoded;
    }
}
```

```
import java.util.Map;
public class ApiResponse {
  private Map headers;
  private boolean isBase64Encoded;
  private int statusCode;
  private String body;
  public ApiResponse(Map headers, boolean isBase64Encoded, int statusCode, String body) {
    this.headers = headers;
    this.isBase64Encoded = isBase64Encoded;
    this.statusCode = statusCode;
    this.body = body;
  }
  public Map getHeaders() {
    return headers;
  }
  public void setHeaders(Map headers) {
    this.headers = headers;
  }
  public boolean getIsBase64Encoded() {
    return isBase64Encoded;
  }
  public void setIsBase64Encoded(boolean base64Encoded) {
    this.isBase64Encoded = base64Encoded;
  }
  public int getStatusCode() {
    return statusCode;
  }
  public void setStatusCode(int statusCode) {
    this.statusCode = statusCode;
  }
  public String getBody() {
    return body;
  }
  public void setBody(String body) {
    this.body = body;
  }
}
```

- *pom.xml*文件如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/ma
ven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>apiTrigger</groupId>
  <artifactId>apiTrigger</artifactId>
  <version>1.0-SNAPSHOT</version>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>com.aliyun.fc.runtime</groupId>
      <artifactId>fc-java-core</artifactId>
      <version>1.0.0</version>
    </dependency>
  </dependencies>
</project>
```

- 使用StreamRequestHandler handler。
  使用该handler，需要将输入的 `InputStream` 转换为对应的 `POJO` 类，示例代码如下。
  *pom.xml*文件配置与使用PojoRequestHandler<I, O> handler相同。

```
import com.aliyun.fc.runtime.Context;
import com.aliyun.fc.runtime.StreamRequestHandler;
import com.aliyun.fc.runtime.Context;
import com.google.gson.Gson;
import java.io.*;
import java.util.Base64;
```

```
import java.util.HashMap;
import java.util.Map;
public class ApiTriggerDemo2 implements StreamRequestHandler {
  public void handleRequest(InputStream inputStream, OutputStream outputStream, Context co
ntext) {
    try {
      // 将InputStream转化成字符串。
      BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(inputStrea
m));
      StringBuffer stringBuffer = new StringBuffer();
      String string = "";
      while ((string = bufferedReader.readLine()) != null) {
        stringBuffer.append(string);
      }
      String input = stringBuffer.toString();
      context.getLogger().info("inputStream: " + input);
      Request req = new Gson().fromJson(input, Request.class);
      context.getLogger().info("input req: ");
      context.getLogger().info(req.toString());
      String bodyReq = req.getBody();
      Base64.Decoder decoder = Base64.getDecoder();
      context.getLogger().info("body: " + new String(decoder.decode(bodyReq)));
      // 您可以在这里处理您自己的逻辑。
      // 返回结构。
      Map headers = new HashMap();
      headers.put("x-custom-header", " ");
      boolean isBase64Encoded = false;
      int statusCode = 200;
      Map body = new HashMap();
      Response resp = new Response(headers, isBase64Encoded, statusCode, body);
      String respJson = new Gson().toJson(resp);
      context.getLogger().info("outputStream: " + respJson);
      outputStream.write(respJson.getBytes());
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      try {
        outputStream.close();
        inputStream.close();
      } catch (IOException e) {
        e.printStackTrace();
```

```
        }
      }
    }
    class Request {
      private String path;
      private String httpMethod;
      private Map headers;
      private Map queryParameters;
      private Map pathParameters;
      private String body;
      private boolean isBase64Encoded;
      @Override
      public String toString() {
        return "Request{" +
            "path='" + path + '\'' +
            ", httpMethod='" + httpMethod + '\'' +
            ", headers=" + headers +
            ", queryParameters=" + queryParameters +
            ", pathParameters=" + pathParameters +
            ", body='" + body + '\'' +
            ", isBase64Encoded=" + isBase64Encoded +
            '}';
      }
      public String getBody() {
        return body;
      }
    }
    // 函数计算需要以以下JSON格式返回对API网关的响应。
    class Response {
      private Map headers;
      private boolean isBase64Encoded;
      private int statusCode;
      private Map body;
      public Response(Map headers, boolean isBase64Encoded, int statusCode, Map body) {
        this.headers = headers;
        this.isBase64Encoded = isBase64Encoded;
        this.statusCode = statusCode;
        this.body = body;
      }
    }
}
```

函数计算的返回参数格式

函数计算需要将输出的内容通过以下的JSON格式返回给API网关，然后由API网关解析。

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

> **说明** isBase64Encoded表示函数计算返回给API网关的编码形式。
>
> - 当body内容为二进制编码时，需在函数计算中对body内容Base64编码，设置isBase64Encoded的值为true。
> - 如果body内容无需Base64编码，设置isBase64Encoded的值为false。
>
> API网关会对isBase64Encoded的值为true的body内容Base64解码后，再返回给客户端。
> 如果函数计算返回不符合格式要求的返回结果，API网关将返回503 Service Unavailable给客户端。

12.2.3. 调试函数

完成函数编写后，您需要调试函数以验证代码的正确性。API网关触发函数执行时，API网关的信息以event作为输入参数传给函数，您可以将API网关传入的event信息作为参数，调试函数代码编写是否正确。本文介绍如何通过函数计算控制台调试函数。

前提条件

编写函数

操作步骤

1. 登录函数计算控制台。
2. 在顶部菜单栏，选择地域。
3. 在左侧导航栏，单击**服务/函数**。
4. 找到目标服务下的目标函数，单击函数名称。

5. 在**代码执行**页签，单击**触发事件**。
6. 按照event事件格式在代码编辑器中编写event，然后单击**确定**。event格式示例如下所示。

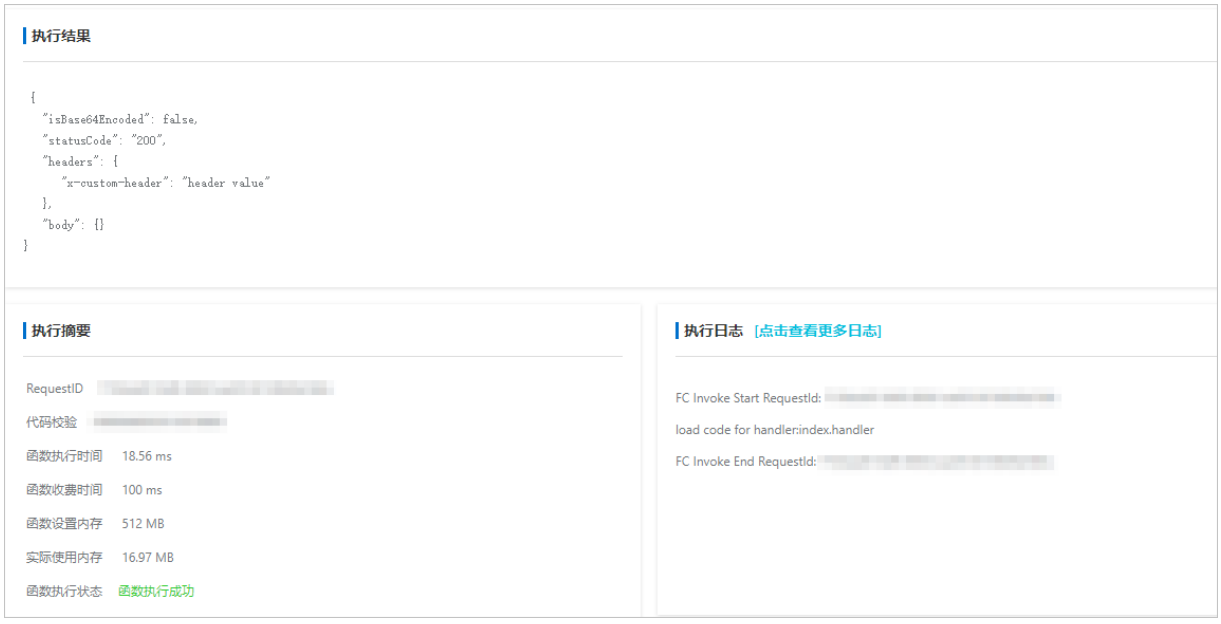
```
{
  "path":"api request path",
  "httpMethod":"request method name",
  "headers":{all headers,including system headers},
  "queryParameters":{query parameters},
  "pathParameters":{path parameters},
  "body":"string of request payload",
  "isBase64Encoded":"true|false, indicate if the body is Base64-encode"
}
```

> 说明
>
> - 如果 isBase64Encoded 的值为 true ，表示API网关传给函数计算的body内容已Base64编码。函数计算需要先对body内容Base64解码后再处理。
> - 如果 isBase64Encoded 的值为 false ，表示API网关没有对body内容Base64编码，在函数中可以直接获取body内容。

7. 在**代码执行**页签下，单击**执行**。

执行结果

在**代码执行**页签可以看到执行成功的提示。



12.2.4. 创建API

使用API网关触发函数执行，需要在API中集成函数计算，即需要将函数计算设置为API的后端服务，并发布API。本文介绍如何在API网关控制台创建、调试并发布API。

步骤一：创建分组

分组是API的管理单元，创建API之前，您需要先创建分组。若已有分组，则跳过该步骤。

1. 登录API 网关控制台。
2. 选择API所在地域。

> **说明** 如果函数计算与API不在同一地域，API将通过公网访问您的函数计算服务。若您对数据安全和网络延迟有较高要求，请选择API与函数计算为同一地域。

3. 选择**开放API > 分组管理**。
4. 单击**创建分组**。
5. 选择**实例**并填写**分组名称**，然后单击**确定**。

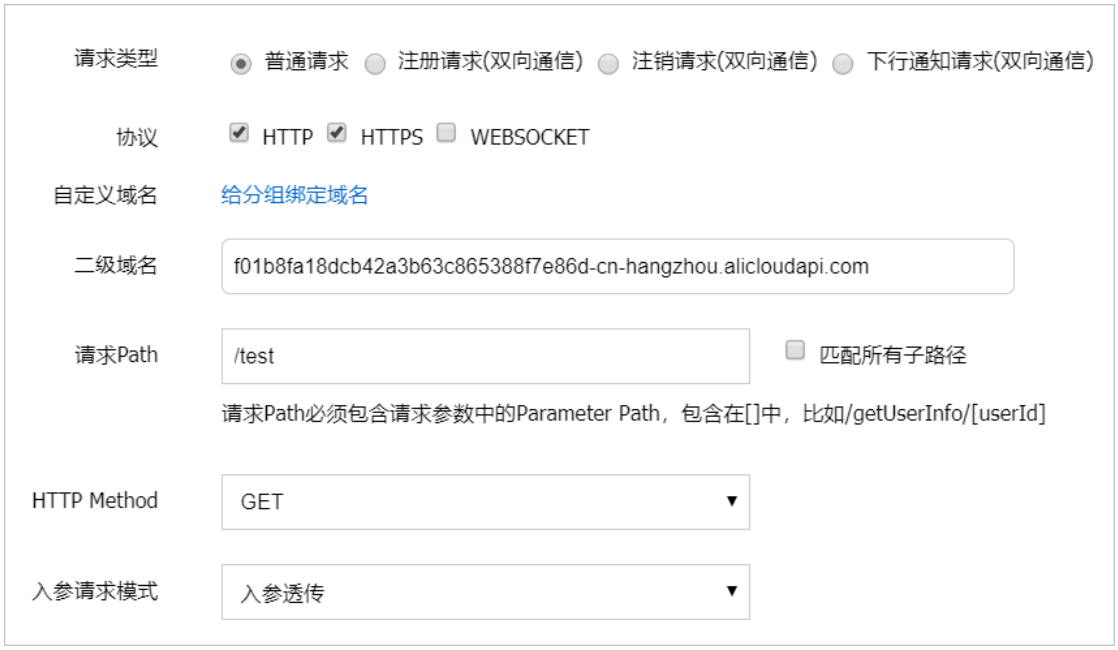
您可以在API分组列表中看到已创建的API分组。

步骤二：创建API

1. 在左侧导航栏，选择**开放API > API列表**。
2. 单击**创建API**。
3. 根据配置向导，填写相关信息。具体参数解释请参见创建API。

i. 按提示填写相应信息，然后单击**下一步**。

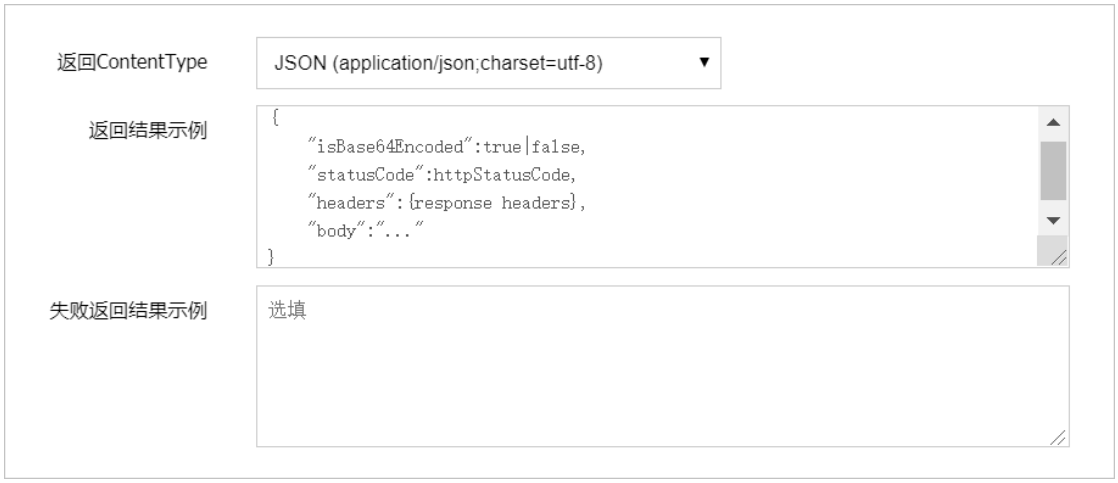
ii. 定义API请求，然后单击**下一步**。



iii. 定义API后端服务，单击**下一步**。

参数	操作
后端服务类型	选择**函数计算**。
函数类型	选择函数计算中对接API网关的函数类型，取值： ■ **事件函数** ■ **HTTP函数**
区域	选择函数计算中对接API网关的函数所在的地域。
服务名称	填写函数计算中对接API网关的函数所在的服务名称。
函数名称	填写函数计算中对接API网关的函数名称。
函数别名	填写函数别名，默认值为**LATEST**。如需填写其他合法别名，请确保您已预先在函数计算控制台创建该合法别名。操作步骤请参见别名操作。
角色Arn	单击**获取授权**会自动分配默认角色。
后端超时	设置后端超时时间。

iv. 定义返回结果，单击**创建**。



API创建成功后，您可以在**API列表**页面查看到该 API。

调试API

调试API的操作步骤请参见调试API。

发布API

1. 在API列表中，找到目标API。
2. 单击**操作**列的**发布**。
3. 在**发布API**对话框中，选择要发布的环境，填写备注信息，然后单击**发布**。

发布完成后，您就可以通过API网关提供的API来访问函数。

更多信息

- 问题诊断
- 使用函数计算做为API后端服务

12.2.5. 问题诊断

如果您在使用函数计算作为API网关后端服务的过程中遇到一些问题，可参考本文进行问题诊断。

API网关触发函数执行时报503，查看函数日志，函数已经执行成功了，这是怎么回事？

API网关和函数计算的对接有格式要求，如果函数计算返回给API网关的结果没有按规定的格式返回，那么API网关就认为后端服务不可用。API网关和函数计算的对接格式要求请参见event格式说明和函数计算的返回参数格式。

如何设置返回响应的content-type？

如图所示，在API设置的时候可以设置响应的content-type。更多详细内容请参见创建API。

API网关触发函数计算执行，已经调通的函数，一段时间不调用，再次调用会报503，这是什么原因？

一段时间不调用后，函数重新调用需要准备执行环境，有冷启动时延，在API网关设置的超时时间内没有调用完，API网关会认为后端服务不可用。延长API网关的超时时间即可解决问题。

为什么函数中接收到API网关传过来的body是经过了Base64编码的？

API网关对FORM形式的body传输是不进行Base64编码的（使用FORM形式需要在API网关选择入参映射），其他形式body都会进行Base64编码，避免内容传输错误或者丢失。建议您在使用时，先判断event中isBase64是否为true。如果isBase64为true，则body需要在函数中进行解码。API网关传给函数计算的event格式请参见event格式说明。

13.云监控触发器

13.1. 云监控触发器概述

阿里云云监控为您提供了IT设施基础监控，外网网络质量拨测监控，基于事件、自定义指标、日志等内容的业务监控等服务。将云监控集成函数计算后，您可以使用丰富的事件触发自定义的函数执行，以实现更多云资源的自动化处理。

使用场景

云监控触发器可实现对因系统错误或实例错误而重启的机器的自动化处理。例如通过云监控中的ECS重启事件触发函数执行，自动查找ECS挂接的云盘，并给云盘自动创建快照。

事件类型

影响	事件类型	事件参数	应对建议
实例重启	因系统维护实例重启	*SystemMaintenance.Reboot*	在用户操作窗口期合适的时间点： 1. 重启ECS实例或者修改预约重启实例。更多信息，请参见修改预约重启时间。 **说明** 必须在ECS控制台重启实例或调用API RebootInstance，在实例内部重启无效。更多信息，请参见重启实例。 2. 在应用层面，切换流量。或从负载均衡实例中移除有计划维护的ECS实例，避免影响您的业务。 3. （可选）为实例挂载的云盘创建快照备份数据。
实例异常重启	因系统错误实例重启	*SystemFailure.Reboot*	更多信息，请参见实例自动恢复事件。
	因实例错误实例重启	*InstanceFailure.Reboot*	当您收到事件通知时，实例正在或已完成重新启动，建议您： • 查看实例系统日志和屏幕截图排查故障，检查系统发生崩溃的原因，避免再次引发崩溃问题。更多信息，请参见系统日志和屏幕截图。 • 验证实例和应用是否恢复正常。
实例重新部署	因系统维护实例重新部署	*SystemMaintenance.Redeploy*	更多信息，请参见重新部署本地盘实例。
	因系统错误实例重新部署	*SystemFailure.Redeploy*	更多信息，请参见重新部署本地盘实例。
实例停止	因包年包月期限到期实例停止	*InstanceExpiration.Stop*	续费资源。更多信息，请参见续费概述。
	因账号欠费按量付费资源停止	*AccountUnbalanced.Stop*	建议您及时为账号充值，避免因支付方式余额不足而停止实例。

影响	事件类型	事件参数	应对建议
实例释放	因包年包月期限到期实例释放	*InstanceExpiration.Delete*	续费资源。更多信息，请参见续费概述。
	因账号欠费按量付费资源释放	*AccountUnbalanced.Delete*	建议您及时为账号充值，避免因支付方式余额不足而停止实例。
实例释放	因实例创建失败而自动释放	*SystemFailure.Delete*	更多信息，请参见实例创建失败事件。
云盘性能影响	云盘性能受到严重影响	*Stalled*	在应用层面，隔离对该云盘的读写操作。或从负载均衡实例中暂时移除该ECS实例。
本地盘受损	本地盘出现损坏	*ErrorDetected*	更多信息，请参见本地盘实例系统事件概述。
实例重启，受损本地盘被隔离	因系统维护计划重启并隔离坏盘	*SystemMaintenance.RebootAndIsolateErrorDisk*	更多信息，请参见隔离损坏的本地盘（CLI）。
实例重启，本地盘恢复正常	因系统维护重启并重新初始化坏盘	*SystemMaintenance.RebootAndReInitErrorDisk*	更多信息，请参见隔离损坏的本地盘（CLI）。
磁盘卸载异常	因系统维护清理残留磁盘	*SystemMaintenance.CleanInactiveDisks*	登录ECS控制台，查看待处理事件，按照提示处理事件。更多信息，请参见查看系统事件。
突发性能实例发生性能受限	因可用CPU积分不足，突发性能实例的性能无法超过基准性能	*N/A*	您可以通过以下任一方式应对： • 开启无性能约束模式。更多信息，请参见切换性能模式。 • 升级实例配置到更高的规格。更多信息，请参见升降配方式概述。
受损本地盘被隔离	因系统维护隔离坏盘	SystemMaintenance.IsolateErrorDisk	更多信息，请参见隔离损坏的本地盘（CLI）。
本地盘恢复正常	因系统维护重新初始化坏盘	SystemMaintenance.ReInitErrorDisk	更多信息，请参见隔离损坏的本地盘（CLI）。

更多信息

关于系统事件的信息，请参见系统事件概述。

13.2. 云监控触发器示例

13.2.1. 示例简介

函数计算支持监控事件作为事件源，本文介绍使用云监控服务触发函数计算的操作流程。配置完成后，指定的云监控事件会触发函数执行，实现自动化式的自定义处理。
云监控 示例 函数计算

示例场景

假设一台云服务器ECS发生因系统错误而重启，运维人员或者ECS用户可能会紧急响应，人工做一些验证或者创建快照的处理。在本示例中，通过云监控中的ECS重启事件触发函数执行，自动查找出ECS挂接的云盘，并给云盘自动创建了快照。

操作流程

1. 编写函数
2. 测试函数
3. 配置云监控对接函数计算

13.2.2. 编写函数

本文介绍如何使用函数计算控制台编写函数，以实现ECS重启后，函数自动查找出ECS挂接的云盘，并给云盘自动创建快照。

前提条件

1. 创建服务
2. 创建函数

编写函数代码

1. 登录函数计算控制台。
2. 在顶部菜单栏，选择地域。
3. 在左侧导航栏，单击**服务/函数**。
4. 找到目标服务下的目标函数，单击函数名称。

5. 单击**代码执行**页签，在代码编辑器中编写代码。
   本文以Python为例，示例代码如下。

```
# -*- coding: utf-8 -*-
import logging
import json, random, string, time
from aliyunsdkcore import client
from aliyunsdkecs.request.v20140526.DeleteSnapshotRequest import DeleteSnapshotRequest
```

```
from aliyunsdkecs.request.v20140526.CreateSnapshotRequest import CreateSnapshotRequest
from aliyunsdkecs.request.v20140526.DescribeDisksRequest import DescribeDisksRequest
from aliyunsdkcore.auth.credentials import StsTokenCredential
LOGGER = logging.getLogger()
clt = None
def handler(event, context):
 creds = context.credentials
 sts_token_credential = StsTokenCredential(creds.access_key_id, creds.access_key_secret, creds.security_token)
 '''
 {
  "product": "ECS",
  "content": {
    "executeFinishTime": "2018-06-08T01:25:37Z",
    "executeStartTime": "2018-06-08T01:23:37Z",
    "ecsInstanceName": "timewarp",
    "eventId": "e-t4nhcpqcu8fqushpn3mm",
    "eventType": "InstanceFailure.Reboot",
    "ecsInstanceId": "i-bp18l0uopocfc98xxxx"
  },
  "resourceId": "acs:ecs:cn-hangzhou:12345678:instance/i-bp18l0uopocfc98xxxx",
  "level": "CRITICAL",
  "instanceName": "instanceName",
  "status": "Executing",
  "name": "Instance:SystemFailure.Reboot:Executing",
  "regionId": "cn-hangzhou"
 }
 '''
 evt = json.loads(event)
 content = evt.get("content");
 ecsInstanceId = content.get("ecsInstanceId");
 regionId = evt.get("regionId");
 global clt
 clt = client.AcsClient(region_id=regionId, credential=sts_token_credential)
 name = evt.get("name");
 name = name.lower()
 if name in [ 'Instance:SystemFailure.Reboot:Executing'.lower(), "Instance:InstanceFailure.Reboot:Executing".lower()]:
  pass
  # do other things
 if name in ['Instance:SystemFailure.Reboot:Executed'.lower(), "Instance:InstanceFailure.Reboot:Execu
```

```
ted".lower()]:
  request = DescribeDisksRequest()
  request.add_query_param("RegionId", regionId)
  request.set_InstanceId(ecsInstanceId)
  response = _send_request(request)
  disks = response.get('Disks').get('Disk', [])
  for disk in disks:
   diskId = disk["DiskId"]
   SnapshotId = create_ecs_snap_by_id(diskId)
   LOGGER.info("Create ecs snap sucess, ecs id = %s , disk id = %s ", ecsInstanceId, diskId)
def create_ecs_snap_by_id(disk_id):
  LOGGER.info("Create ecs snap, disk id is %s ", disk_id)
  request = CreateSnapshotRequest()
  request.set_DiskId(disk_id)
  request.set_SnapshotName("reboot_" + ''.join(random.choice(string.ascii_lowercase) for _ in range(
6)))
  response = _send_request(request)
  return response.get("SnapshotId")
# send open api request
def _send_request(request):
  request.set_accept_format('json')
  try:
    response_str = clt.do_action_with_exception(request)
    LOGGER.info(response_str)
    response_detail = json.loads(response_str)
    return response_detail
  except Exception as e:
    LOGGER.error(e)
```

13.2.3. 配置云监控对接函数计算

使用云监控触发器函数执行，需要将云监控服务与函数计算对接，即创建事件报警。本文介绍如何在云监控控制台上配置对接函数计算。

创建事件报警

1. 登录云监控控制台。
2. 在左侧导航栏，单击**事件监控**。
3. 在**事件监控**页面，单击**报警规则**页签，单击右上角的**创建事件报警**。
4. 在**创建/修改事件报警**面板，配置报警参数。

参数说明如下。

参数	说明
报警规则名称	输入创建的报警规则名称。
事件类型	选择**系统事件**。
产品类型	选择您需要的产品类型。
事件类型	选择需要报警的事件类型。

参数	说明
事件等级	选择事件等级。
事件名称	输入事件名称。
资源范围	○ **全部资源**：任何资源发生相关事件，都会按照配置发送通知。 ○ **应用分组**：只有指定分组内的资源发生相关事件时，才会发送通知。
报警方式	选中**函数计算**，然后选择函数所在**地域**、**服务**和**函数**。

5. 单击**确定**。

后续步骤

测试函数

13.2.4. 测试函数

完成函数编写和对接配置后，您需要调试函数以验证代码的正确性。云监控服务提供事件报警调试功能。本文介绍如何在云监控控制台调试函数。

前提条件

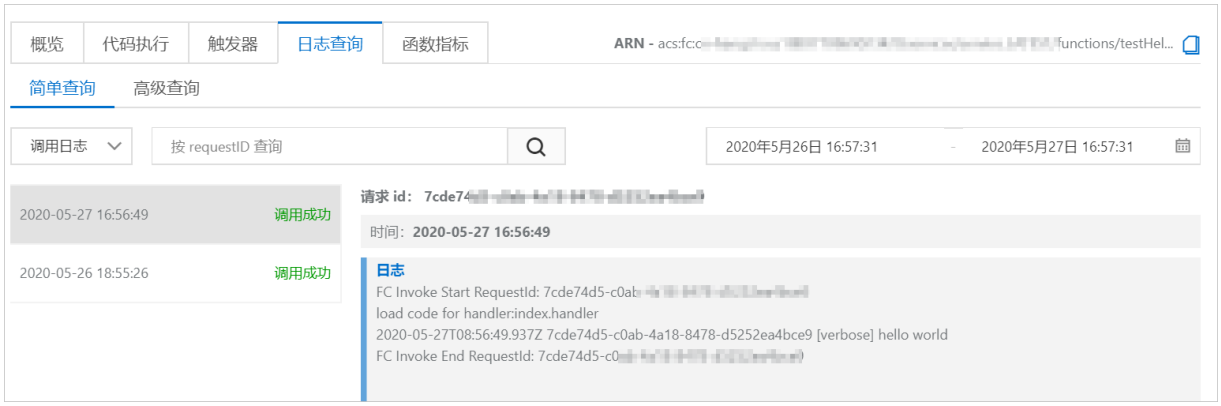
1. 编写函数
2. 配置云监控对接函数计算

操作步骤

1. 登录云监控控制台。
2. 在左侧导航栏，选择**报警服务 > 报警规则**。
   进入**报警规则列表**页面。
3. 单击**事件报警**页签。
4. 找到目标规则，单击**操作**列中的**调试**。
5. 单击**确定**。

执行结果

登录函数计算控制台，找到目标服务下的目标函数，在**日志查询**页签可以看到函数执行成功的日志。


概览
代码执行
触发器
日志查询
函数指标
ARN - acs:fc:c
functions/testHel...
简单查询
高级查询
调用日志
按 requestID 查询
2020年5月26日 16:57:31
2020年5月27日 16:57:31
2020-05-27 16:56:49
调用成功
2020-05-26 18:55:26
调用成功
请求 id: 7cde74
时间：2020-05-27 16:56:49
日志
FC Invoke Start RequestId: 7cde74d5-c0ab
load code for handler:index.handler
2020-05-27T08:56:49.937Z 7cde74d5-c0ab-4a18-8478-d5252ea4bce9 [verbose] hello world
FC Invoke End RequestId: 7cde74d5-c0