ALIBABA CLOUD

阿里云

函数计算

代码开发

文档版本：20220713

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.代码开发概述

本文列举目前函数计算支持的多语言运行时信息。

背景信息

运行时提供针对不同语言的、在执行环境中运行的环境。运行时作为函数计算和您的函数之间的接力员，传递函数调用的事件（event）、上下文信息（context）和响应。您可以使用函数计算提供的运行时或构建您自己的运行时，还可以自行构建容器镜像。

函数计算运行时

Node.js

- 事件函数
- HTTP函数
- Initializer函数
- 运行环境

Python

- 事件函数
- HTTP函数
- Initializer函数
- 运行环境

PHP

- 事件函数
- HTTP函数
- Initializer函数
- 运行环境

Java

- 事件函数
- HTTP函数
- Initializer函数
- 运行环境

C#

- 事件函数
- HTTP函数
- Initializer函数
- 运行环境

Go

- 入门概述
- 事件函数
- HTTP函数
- Initializer函数

Custom Runtime

以下为多语言使用示例的文档列表，更多信息，请参见环境说明。

- **事件函数**
- **HTTP函数**

Custom Container

以下为多语言使用示例的文档列表，更多信息，请参见Custom Container简介。

- **事件函数**
- **HTTP函数**

2.基础信息

本文介绍使用函数计算编写代码相关的基础概念信息，包括请求处理程序、函数实例生命周期回调方法、日志记录和错误处理等。

请求处理程序

在创建函数时，您需要指定请求处理程序。函数计算的运行时会加载并调用您的请求处理程序处理请求。请求处理程序包含以下两种类型：

- 事件请求处理程序
  用于处理除HTTP触发器以外的各种事件源触发的事件请求。
- HTTP请求处理程序
  用于处理HTTP触发器触发的请求调用。更多信息，请参见配置HTTP触发器并使用HTTP触发。

您可以在函数计算控制台配置**请求处理程序（函数入口）**。具体操作，请参见更新函数。

函数实例生命周期回调方法

函数的按量实例动态按需创建，闲置时会被冻结，冻结一段时间会被销毁，您配置的实例生命周期回调方法在实例状态发生变化时被调用。函数计算支持Initializer、PreFreeze和PreStop三种生命周期回调方法。更多信息，请参见函数实例生命周期 。

Initializer回调

Initializer回调是在函数实例启动成功之后，执行请求处理程序之前执行。函数计算保证在一个实例生命周期内，成功执行且只能成功执行一次Initializer回调。例如，您的Initializer回调首次执行失败后系统会重试，直到成功为止，然后再执行您的请求处理程序。您可以将数据库场景下连接池构建、函数依赖库加载等耗时较长的业务逻辑放到Initializer回调中，避免每次运行函数都会做重复的操作，降低函数延时。

PreFreeze回调

PreFreeze回调在函数实例冻结前执行，您可以在该回调完成实例冻结前的必要操作，例如等待指标发送成功等。

PreStop回调

PreStop回调在函数实例销毁前执行，您可以在该回调完成实例销毁前的必要操作，例如关闭数据库链接，以及上报、更新状态等。

日志记录

> **说明**
> - 您需要配置服务级别的日志库，函数计算会将函数日志发送到指定日志库中。更多信息，请参见配置日志。
> - 通过控制台创建服务时，函数计算默认为您配置日志库。

函数计算与日志记录集成，函数计算会将函数调用的记录以及函数代码中打印的日志全部存储到日志库中，您可以使用函数计算提供的日志语句记录函数日志，方便调试及定位问题。各种编程语言的打印日志语句，如下表所示：

开发语言	编程语言内嵌的打印日志语句	函数计算提供的日志语句	参考文档

开发语言	编程语言内嵌的打印日志语句	函数计算提供的日志语句	参考文档
Node.js	console.log()	context.logger.info()	打印日志
Python	print()	logging.getLogger().info()	打印日志
Java	System.out.println()	context.getLogger().info()	编译部署代码包
PHP	echo "" . PHP_EOL	$GLOBALS['fcLogger']->info()	环境说明
C#	Console.WriteLine("")	context.Logger.LogInformation()	打印日志

使用编程语言内嵌的打印输出语句打印的日志也会被收集到日志库里，而使用函数计算提供的日志语句打印的每条日志前都会带上请求ID，方便日志的筛选。

```
#使用编程语言内嵌的打印输入语句打印的日志
# print('hello world')
message:  hello world
#使用函数计算提供的日志语句打印的日志
# logger.info('hello world')
message:  2020-03-13T04:06:49.099Z f84a9f4f-2dfb-41b0-9d6c-1682a2f3a650 [INFO] hello world
```

日志组成

函数运行日志包括服务名称、函数名称、当前执行版本、别名和代码日志。

函数运行日志数据结构如下所示：

```
__source__:
__tag__:__receive_time__:  1584072413
__topic__:  myService
functionName:  myFunction
message:  2020-03-13T04:06:49.099Z f84a9f4f-2dfb-41b0-9d6c-1682a2f3a650 [INFO] hello world
qualifier:  LATEST
serviceName:  myService
versionId:
```

- 在函数开始执行时，系统会打印 `FC Invoke Start RequestId: f84a9f4f-2dfb-41b0-9d6c-1682a2f3a650` ，标志函数执行开始。
- 在函数执行完成时，系统会打印 `FC Invoke End RequestId: f84a9f4f-2dfb-41b0-9d6c-1682a2f3a650` ，标志函数执行结束。

错误处理

函数计算的错误类型有两种，分别是 `HandledInvocationError` 和 `UnhandledInvocationError` 。

- HandledInvocationError

只有在Node.js中通过 `callback` 返回的错误是 `HandledInvocationError` ，错误信息在响应内容中体现。

```
'use strict';
  module.exports.handler = function(event, context, callback) {
    console.log('hello world');
    callback('this is error', 'hello world');
  }
```

响应内容如下所示：

```
{"errorMessage":"this is error"}
```

- UnhandledInvocationError
  除了 `HandledInvocationError` ，其余的错误都是 `UnhandledInvocationError` 。
  `UnhandledInvocationError` 的错误 `stackTrace` 会打印到日志中，您可以进入日志查看上下文，找到对应的 `stackTrace` 。

3.Go

3.1. 环境说明

本文介绍在函数计算中使用Go语言编写函数的运行环境信息。

Go运行时

函数计算目前支持Go 1.x版本，推荐使用Go 1.8或以上版本。

名称	操作系统	架构
Go 1.x	Linux	x86_64

> **注意** Go运行时目前使用Linux操作系统，暂不支持ARM64架构。

Go SDK与工具

函数计算提供以下Go SDK与工具：

- FC SDK for Go：Go运行时编程模型的具体实现，函数计算平台依赖该包运行您的请求处理程序（Handler）。
- fccontext：访问函数执行上下文信息（Context）的辅助库。
- examples：使用Go运行时的简单示例。

相关文档

- 请求处理程序（Handler）
- 事件请求处理程序（Event Handler）
- HTTP请求处理程序（HTTP Handler）
- 上下文
- 编译部署代码包
- 日志
- 错误处理
- 函数实例生命周期回调

3.2. 请求处理程序（Handler）

本文介绍在函数计算FC（Function Compute）中使用Go运行时开发请求处理程序的相关概念和方法。

什么是请求处理程序

FC函数的请求处理程序，是函数代码中处理请求的方法。当您的FC函数被调用时，函数计算会运行您提供的Handler方法处理请求。您可以通过函数计算控制台的**函数入口**配置Handler。

对Go语言的FC函数而言，您的请求处理程序被编译为一个可执行的二进制文件。您只需要将FC函数的请求处理程序配置项设置为该可执行文件的文件名即可。

关于FC函数的具体定义和相关操作，请参见管理函数。

配置说明

请求处理程序的具体配置均需符合函数计算平台的配置规范。配置规范因请求处理程序类型而异。

请求处理程序分为事件请求处理程序（Event Handler）和HTTP请求处理程序（HTTP Handler）；其中事件请求由各种事件源触发生成，HTTP请求则由HTTP触发器触发生成。两种请求处理程序的详细解释，请参见请求处理程序类型。

请求处理程序的具体配置示例，请分别参见事件请求处理程序和HTTP请求处理程序。

更多信息

如何将您的代码包编译部署至函数计算平台，请参见编译部署代码包。

3.3. 事件请求处理程序（Event Handler）

本文介绍Go事件请求处理程序的结构和特点。

使用示例

在Go语言的代码中，您需要引入官方的SDK库 `aliyun/serverless/fc-runtime-go-sdk/fc` ，并实现 `handler` 函数和 `main` 函数。示例如下：

```
package main
import (
    "fmt"
    "context"
    "github.com/aliyun/fc-runtime-go-sdk/fc"
)
type StructEvent struct {
    Key string `json:"key"`
}
func HandleRequest(ctx context.Context, event StructEvent) (string, error) {
    return fmt.Sprintf("hello, %s!", event.Key), nil
}
func main() {
    fc.Start(HandleRequest)
}
```

传入的 `event` 参数是一个包含 `key` 属性的JSON字符串，示例如下。

```
{
  "key": "value"
}
```

具体的示例解析如下：

- `package main` ：在Go语言中，Go应用程序都包含一个名为 `main` 的包。
- `import` ：需要引用函数计算依赖的包，主要包括以下包：
  - `github.com/aliyun/fc-runtime-go-sdk/fc` ：函数计算Go语言的核心库。
  - `context` ：函数计算Go语言的Context对象。
- `func HandleRequest(ctx context.Context, event StructEvent) (string, error)` ：处理事件请求的方法（即Handler），需包含将要执行的代码，参数含义如下：
  - `ctx context.Context` ：为您的FC函数调用提供在调用时的运行上下文信息。更多信息，请参见上下文。
  - `event StructEvent` ：调用函数时传入的数据，可以支持多种类型。

  - `string, error`：返回两个值，字符串和错误信息。更多信息，请参见错误处理。
  - `return fmt.Sprintf("Hi, %s !", event.Key), nil`：简单地返回`hello`信息，其中包含传入的`event`。`nil`表示没有报错。
- `func main()`：运行FC函数代码的入口点，Go程序必须包含`main`函数。通过添加代码`fc.Start(HandleRequest)`，您的程序即可运行在阿里云函数计算平台。

> **注意** HTTP请求处理程序和事件请求处理程序的启动方法不同。如果是事件请求处理程序，您需要在`main`函数中调用`fc.Start`函数。如果是HTTP请求处理程序，您需要在`main`函数中调用`fc.StartHttp`函数。

Event Handler签名

有效的Event Handler签名如下：

- `func ()`
- `func () error`
- `func (InputType) error`
- `func () (OutputType, error)`
- `func (InputType) (OutputType, error)`
- `func (context.Context) error`
- `func (context.Context, InputType) error`
- `func (context.Context) (OutputType, error)`
- `func (context.Context, InputType) (OutputType, error)`

其中，`InputType`和`OutputType`与`encoding/json`标准库兼容。

Event Handler的使用需遵循以下规则：

- Handler必须是一个函数。
- Handler支持0～2个输入参数。如果有2个参数，则第一个参数必须是`context.Context`。
- Handler支持0～2个返回值。如果有1个返回值，则必须是`error`类型；如果有2个返回值，则第2个返回值必须是`error`。

事件函数的Handler示例代码：

- event-struct.go：`event`为Struct类型的示例代码。
- event-string.go：`event`为String类型的示例代码。
- event-map.go：`event`为`map[string]interface{}`类型的示例代码。

更多Handler示例，请参见examples。

Context

Context的详细使用方法，请参见上下文。

3.4. HTTP请求处理程序（HTTP Handler）

您可以使用HTTP Handler更方便地处理HTTP请求。当调用函数时，FC运行您提供的执行方法来处理请求。本文介绍Go HTTP Handler的结构和特点。

使用示例

在Go语言的代码中，您需要引入官方的SDK库 `aliyun/serverless/fc-runtime-go-sdk/fc` ，并实现 `handler` 函数和 `main` 函数。示例如下：

```
package main
import (
    "context"
    "fmt"
    "net/http"
    "io/ioutil"
    "github.com/aliyun/fc-runtime-go-sdk/fc"
)
func HandleHttpRequest(ctx context.Context, w http.ResponseWriter, req *http.Request) error
{
  body, err := ioutil.ReadAll(req.Body)
  if err != nil {
    w.WriteHeader(http.StatusBadRequest)
    w.Header().Add("Content-Type", "text/plain")
    w.Write([]byte(err.Error()))
    return nil
  }
    w.WriteHeader(http.StatusOK)
    w.Header().Add("Content-Type", "text/plain")
    w.Write([]byte(fmt.Sprintf("Hi, %s!\n", body)))
    return nil
}
func main() {
    fc.StartHttp(HandleHttpRequest)
}
```

示例解析如下：

- `package main` ：在Go语言中，Go应用程序都包含一个名为 `main` 的包。
- `import` ：需要引用函数计算依赖的包，主要包括以下包：
  - `github.com/aliyun/fc-runtime-go-sdk/fc` ：函数计算Go语言的核心库。
  - `context` ：函数计算Go语言的Context对象。
  - `net/http` ：HTTP Handler中需要用到的HTTP包中Request和ResponseWriter接口。
- `HandleHttpRequest(ctx context.Context, w http.ResponseWriter, req *http.Request) error` ：处理HTTP请求的方法（即HTTP Handler），需包含将要执行的代码，参数含义如下：
  - `ctx context.Context` ：提供了函数在调用时的运行上下文信息，详细信息，请参见上下文。
  - `w http.ResponseWriter` ：HTTP Handler的响应接口，可以设置状态行、消息头和响应正文。具体支持格式，请参见响应接口。
  - `req *http.Request` ：HTTP Handler的请求接口，包含请求行、请求头和请求正文。具体方法，请参见请求结构体。
  - `w.WriteHeader(http.StatusOK)` ：填入响应的HTTP状态码。

- `w.Header().Add("Content-Type", "text/plain")`：填入响应的消息头。
- `w.Write([]byte(fmt.Sprintf("Hi, %s!\n", body)))`：填入响应的消息体。
- `return nil`：简单的错误信息，`nil`表示没有错误发生。如果设置了错误信息，则认为是函数错误。

- `func main()`：运行FC函数代码的入口点，Go程序必须包含`main`函数。在`main()`中调用`fc.StartHttp(HandleHttpRequest)`，您的程序即可运行在阿里云的函数计算平台。

> **注意** HTTP请求处理程序和事件请求处理程序的启动方法不同。如果是事件请求处理程序，您需要在`main`函数中调用`fc.Start`函数。如果是HTTP请求处理程序，您需要在`main`函数中调用`fc.StartHttp`函数。

Handler

Go的HTTP Handler的定义，是参考Go标准库HTTP中的Handler interface设计，并在此基础上新增一个`context`参数。定义如下：

```
function(ctx context.Context, w http.ResponseWriter, req *http.Request) error
```

函数定义中包含以下三部分内容：

- `context`：为您的FC函数调用提供运行时信息。更多信息，请参见上下文。
- `http.Request`：请求结构体。详细信息，请参见请求结构体。
- `http.ResponseWriter`：响应接口。详细信息，请参见响应接口。

请求结构体

`http.Request`是Go标准库HTTP中的定义，目前支持的参数和方法如下表所示。

参数	类型	描述
Method	String	HTTP请求方法，例如PUT、POST、DELETE等。
URL	*url.URL	请求地址信息。
Header	http.Header	HTTP请求头部的键值对。
Body	io.ReadCloser	请求结构体。
ContentLength	Int64	请求结构体数据长度。

响应接口

实现了`http.ResponseWriter`声明的三个方法，示例如下：

```
type ResponseWriter interface {
    Header() Header
    Write([]byte) (int, error)
    WriteHeader(statusCode int)
}
```

说明如下：

- `WriteHeader(statusCode int)`：设置状态码。
- `Header() Header`：获取并设置响应头信息。
- `Write([]byte) (int, error)`：设置响应体。

限制说明

- 请求限制
  如果超过以下限制，会返回`400`状态码和`InvalidArgument`错误码。

字段	限制说明	HTTP状态码	错误码
headers	请求头中的所有键和值的总大小不能超过4 KB。	400	InvalidArgument
path	请求路径以及所有查询参数的总大小不能超过4 KB。		
body	同步调用请求的Body的总大小不能超过16 MB，异步调用请求的Body的总大小不能超过128 KB。		

- 响应限制
  如果超过以下限制，会返回`502`状态码和`BadResponse`错误码。

字段	限制说明	HTTP状态码	错误码
headers	响应头中的所有键和值对的大小不能超过4 KB。	502	BadResponse
body	同步调用响应的Body的总大小不能超过16 MB，异步调用响应的Body的总大小不能超过128 KB。		

Context

Context的详细使用方法，请参见上下文。

3.5. 上下文

本文介绍在函数计算中使用Go运行时开发代码时，所涉及的Context（上下文）的相关概念和使用示例。

什么是上下文

当函数计算运行您的函数时，它会将上下文对象（`context.Context`）传递到执行方法中。该对象包含有关调用、服务、函数、链路追踪和执行环境等信息。

事件请求处理程序（Event Handler）和HTTP请求处理程序（HTTP Handler）都支持上下文对象作为传入参数，且格式和内容相同。上下文对象主要提供了以下参数。

Context信息

字段	说明
变量	
RequestID	本次调用请求的唯一ID。您可以记录该ID，当函数调用出现问题时方便查询。
Credentials	函数计算服务通过扮演服务角色而获取的一组临时密钥，其有效时间是5分钟。您可以在代码中使用 `Credentials` 去访问相应的服务例如OSS，这就避免了您把自己的AccessKey信息编码在函数代码里。详细信息，请参见通过RAM角色实现跨云账号授权。
Function	当前调用的函数的一些基本信息，例如函数名、函数入口、函数内存和超时时间。
Service	当前调用的函数所在的服务信息，包含服务名称、接入的日志服务SLS的Project和Logstore信息，以及服务的版本和别名信息。其中 `qualifier` 表示调用函数时指定的服务版本或别名，`version_id` 表示实际调用的服务版本。
Region	当前调用的函数所在地域ID，例如在华东2（上海）地域调用，则地域ID为cn-shanghai。详细信息，请参见服务地址。
AccountId	函数所属的阿里云账号ID（主账号ID）。
方法	
deadline	返回函数执行的超时时间，格式为Unix时间戳，单位：毫秒。

完整的数据结构，请参见context.go。

使用示例

示例一：打印Context信息

首先，函数的 `handler` 需要包含 `context` 参数，函数计算会把Context信息中的变量信息插入到 `context` 的取值中。然后，需要 `import aliyun/fc-runtime-go-sdk/fccontext` ，通过 `fccontext.FromContext` 方法获取 `fccontext` 。

```
package main
import (
    "context"
    "encoding/json"
    "log"
    "github.com/aliyun/fc-runtime-go-sdk/fc"
    "github.com/aliyun/fc-runtime-go-sdk/fccontext"
)
func main() {
    fc.Start(echoContext)
}
func echoContext(ctx context.Context) (string, error) {
    fctx, _ := fccontext.FromContext(ctx)
    log.Println(fctx.AccountId)
    log.Printf("%#v\n", fctx)
    res, _ := json.Marshal(fctx)
    return string(res), nil
}
```

示例二：获取函数剩余执行时间

以下示例展示了如何使用 `deadline` 获取函数剩余执行时间。

```
package main
import (
    "context"
    "fmt"
    "log"
    "time"
    "github.com/aliyun/fc-runtime-go-sdk/fc"
)
func LongRunningHandler(ctx context.Context) (string, error) {
    deadline, _ := ctx.Deadline()
    fmt.Printf("now: %s\ndeadline: %s\n", time.Now().String(), deadline.String())
    deadline = deadline.Add(-100 * time.Millisecond)
    timeoutChannel := time.After(time.Until(deadline))
    for {
        select {
        case <-timeoutChannel:
            return "Finished before timing out.", nil
        default:
            log.Print("hello!")
            time.Sleep(50 * time.Millisecond)
        }
    }
}
func main() {
    fc.Start(LongRunningHandler)
}
```

3.6. 编译部署代码包

Go是静态编译型语言，您需在本地自行编译程序并打包为.zip文件。本文介绍如何将函数计算官方Go SDK库与您的代码一同打包。

前提条件

安装Go语言环境。函数计算已支持Go 1.x版本，推荐使用Go 1.8或以上版本。

在Linux或macOS下编译打包

1. 下载函数计算Go SDK库。

```
go get github.com/aliyun/fc-runtime-go-sdk/fc
```

2. 在文件所在目录下，执行如下命令编译文件。

```
GOOS=linux go build main.go
```

> **说明**
> - `main.go` 仅为示例，需替换为您实际的文件名。
> - 编译完成后，该目录下生成与文件同名的二进制文件。

设置 `GOOS=linux`，确保编译后的可执行文件与函数计算平台的Go运行系统环境兼容，尤其是在非Linux环境中编译时。
补充说明如下：

- 针对Linux操作系统，建议使用纯静态编译，配置 `CGO_ENABLED=0`，确保可执行文件不依赖任何外部依赖库（如libc库），避免出现编译环境和Go运行时环境依赖库的兼容问题。示例如下：

```
GOOS=linux CGO_ENABLED=0 go build main.go
```

- 针对M1 macOS（或其他ARM架构的机器），配置 `GOARCH=amd64`，实现跨平台编译，示例如下：

```
GOOS=linux GOARCH=amd64 go build main.go
```

3. 打包上一步生成的二进制文件。

```
zip fc-golang-demo.zip main
```

在Windows下编译打包

1. 编译可执行文件。
   i. 同时按下**Win+R**键打开运行窗口。
   ii. 输入*cmd*，然后按下**Enter**键。
   iii. 在弹出的命令提示符窗口中，执行以下命令。

```
set GOOS=linux
set GOARCH=amd64
go build -o main main.go
```

2. 使用build-fc-zip工具打包。
   i. 使用**go get**方式从GitHub下build-fc-zip工具。

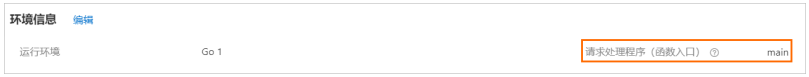
```
go get -u github.com/aliyun/fc-runtime-go-sdk/cmd/build-fc-zip
```

ii. 使用build-fc-zip工具打包。如果您使用的go的默认安装方式，则该工具通常会安装在*%USERPROFILE%\go\bin*目录下。

```
~\go\bin\build-fc-zip.exe -output main.zip main
```

配置FC函数处理程序

1. 创建服务。
2. 创建函数，选择**运行环境**为Go 1。



Go是编译型语言，需要在本地编译后以上传ZIP包的形式上传可执行的二进制文件。在函数计算控制台的请求处理程序配置中，Go语言的FC函数**请求处理程序**需要直接设置为 `[文件名]` 。该文件名是指编译后的二进制文件名称，当函数被调用时，函数计算平台会直接执行该二进制文件。

- 如果编译生成的二进制文件存放在ZIP包的根目录，如下图所示。此时，需要将FC函数**请求处理程序**设置为 `main` 。

```
→  code zip -sf fc-golang-demo.zip
Archive contains:
  main
Total 1 entries (6657596 bytes)
```

- 如果编译生成的二进制文件没有放到ZIP包的根目录，而是放到例如*bin/*目录下，如下图所示。此时，**请求处理程序**需要设置为 `bin/main` 。

```
→  code zip -sf fc-golang-demo.zip
Archive contains:
  bin/main
Total 1 entries (6657596 bytes)
```

其他部署方式：

- 使用函数计算SDK部署函数
- 使用Serverless Devs部署函数

3.7. 日志

本文介绍Go运行环境的日志打印相关内容。

日志打印

您可以使用 `context.GetLogger()` 方法打印日志，也可以使用log库或fmt库中的方法打印日志，或者其他写入到stdout或stderr的日志库打印日志。

使用context.GetLogger() 方法打印日志

使用该方法打印的每条日志中都包含日志级别、RequestId、时间、文件名和行号等信息，代码示例如下所示。

```
package main
import (
    "context"
    "github.com/aliyun/fc-runtime-go-sdk/fc"
    "github.com/aliyun/fc-runtime-go-sdk/fccontext"
)
func HandleRequest(ctx context.Context) (string, error) {
    fctx, _ := fccontext.FromContext(ctx)
       fctx.GetLogger().Debug("this is Debug log")
    fctx.GetLogger().Debugf("Hi, %s\n", "this is Debugf log")
    fctx.GetLogger().Info("this is Info log")
    fctx.GetLogger().Infof("Hi, %s\n", "this is Infof log")
    fctx.GetLogger().Warn("this is Warn log")
    fctx.GetLogger().Warnf("Hi, %s\n", "this is Warnf log")
    fctx.GetLogger().Error("this is Error log")
    fctx.GetLogger().Errorf("Hi, %s\n", "this is Errorf log")
    return "Hello world", nil
}
func main() {
    fc.Start(HandleRequest)
}
```

输出的日志内容如下所示。

```
FC Invoke Start RequestId: 1e9a87a5-fe0f-4904-a6f4-1d2728514129
2022-04-20T04:28:41.79Z d32c01bc-4397-4f52-a9ca-e374c28f96c1 [DEBUG] main.go:16: this is De
bug log
2022-04-20T04:28:41.79Z d32c01bc-4397-4f52-a9ca-e374c28f96c1 [DEBUG] main.go:17: Hi, this i
s Debugf log
2022-04-20T04:28:41.79Z d32c01bc-4397-4f52-a9ca-e374c28f96c1 [INFO] main.go:19: this is Inf
o log
2022-04-20T04:28:41.79Z d32c01bc-4397-4f52-a9ca-e374c28f96c1 [INFO] main.go:20: Hi, this is
Infof log
2022-04-20T04:28:41.79Z d32c01bc-4397-4f52-a9ca-e374c28f96c1 [WARN] main.go:22: this is War
n log
2022-04-20T04:28:41.79Z d32c01bc-4397-4f52-a9ca-e374c28f96c1 [WARN] main.go:23: Hi, this is
Warnf log
2022-04-20T04:28:41.791Z d32c01bc-4397-4f52-a9ca-e374c28f96c1 [ERROR] main.go:25: this is E
rror log
2022-04-20T04:28:41.791Z d32c01bc-4397-4f52-a9ca-e374c28f96c1 [ERROR] main.go:26: Hi, this
is Errorf log
FC Invoke End RequestId: 1e9a87a5-fe0f-4904-a6f4-1d2728514129
```

使用log库打印日志

使用该方法打印的日志库包含日期和时间信息，代码示例如下所示。

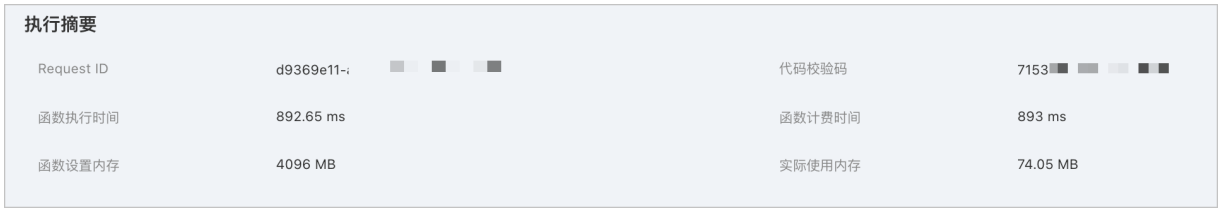
```
package main
import (
    "log"
    "aliyun/serverless/fc-runtime-go-sdk/fc"
)
func HandleRequest() (string, error) {
    log.Println("hello world")
    return "hello world", nil
}
func main() {
    fc.Start(HandleRequest)
}
```

输出的日志内容如下所示。

```
FC Invoke Start RequestId: a15f8f85-9ed3-47c9-a61c-75678db61831
2022/02/17 04:29:02.228870 hello world
FC Invoke End RequestId: a15f8f85-9ed3-47c9-a61c-75678db61831
```

执行摘要

Go运行时会记录每次调用的Start和End行，以及执行摘要信息。



执行摘要的参数说明如下：

参数	说明
Request ID	调用的唯一请求ID。
代码校验码	调用所使用代码包的校验码。
函数执行时间	处理程序实际运行所花费的时间。
函数计费时间	针对调用计费的时间量。
函数设置内存	分配给函数的内存量。
实际使用内存	函数实际使用的最大内存量。

3.8. 错误处理

本文介绍Go运行环境的错误处理相关内容。

发生异常时，函数调用响应的HTTP Header中会包含 `X-Fc-Error-Type` ，例如 `X-Fc-Error-Type: UnhandledInvocationError` 。函数计算的错误类型的更多信息，请参见错误处理。

函数计算返回错误信息的方式如下：

- 在入口函数直接返回错误信息，示例如下。

```
package main
import (
    "errors"
    "fmt"
    "aliyun/serverless/fc-runtime-go-sdk/fc"
)
func HandleRequest() error {
    fmt.Println("hello world")
    return errors.New("something is wrong")
}
func main() {
    fc.Start(HandleRequest)
}
```

调用函数时收到的响应如下所示。

```
{
  "errorMessage": "something is wrong!",
  "errorType": "errorString"
}
```

- 使用panic抛出错误信息，示例如下。

```
package main
import (
    "fmt"
    "aliyun/serverless/fc-runtime-go-sdk/fc"
)
func HandleRequest() error {
    fmt.Println("hello world")
    panic("Error: something is wrong")
    return nil
}
func main() {
    fc.Start(HandleRequest)
}
```

调用函数时收到的响应如下所示（示例中省略了部分堆栈信息）。

```
{
  errorMessage: 'Error: something is wrong',
  errorType: 'string',
  stackTrace: [
    {
      path: 'aliyun/serverless/fc-runtime-go-sdk/fc/errors.go',
      line: 39,
      label: 'fcPanicResponse'
    },
    {
      path: 'aliyun/serverless/fc-runtime-go-sdk/fc/function.go',
      line: 84,
      label: '(*Function).Invoke.func1'
    },
    ...
    ...
    ...
    { path: 'code/main.go', line: 22, label: 'main' },
    { path: 'runtime/proc.go', line: 255, label: 'main' },
    { path: 'runtime/asm_amd64.s', line: 1581, label: 'goexit' }
  ]
}
```

- 建议不要使用包含 `os.Exit(1)` 代码的错误处理代码，例如 `log.Fatal` 。

> **注意** 使用该方法无法获取退出时的报错信息和堆栈信息。

```
package main
import (
    "fmt"
    "log"
    "aliyun/serverless/fc-runtime-go-sdk/fc"
)
func HandleRequest() error {
    fmt.Println("hello world")
    log.Fatal("something is wrong")
    return nil
}
func main() {
    fc.Start(HandleRequest)
}
```

调用函数时收到的响应如下所示。

```
{
  errorMessage: 'Process exited unexpectedly before completing request (duration: 0ms, ma
xMemoryUsage: 8MB)'
}
```

3.9. 函数实例生命周期回调

本文介绍Go实现函数实例生命周期回调的方法。

背景信息

当您实现并配置函数实例生命周期回调后，函数计算系统将在相关实例生命周期事件发生时调用对应的回调程序。函数实例生命周期涉及Initializer、PreFreeze和PreStop三种回调。更多信息，请参见函数实例生命周期回调。

> **注意** 如您需要使用PreFreeze和PreStop回调，请将fc-runtime-go-sdk升级到v0.1.0及以上版本。

回调方法签名

- 初始化回调程序（Initializer回调）是在函数实例启动成功之后，运行请求处理程序（Handler）之前执行。函数计算保证在一个实例生命周期内，成功且只成功执行一次Initializer回调。例如您的Initializer回调第一次执行失败了，系统会重试，直到成功为止，然后再执行您的请求处理程序。因此，您在实现Initializer回调时，需要保证它被重复调用时的正确性。
- 预冻结回调程序（PreFreeze回调）在函数实例冻结前执行。
- 预停止回调程序（PreStop回调）在函数实例销毁前执行。

Initializer回调、PreFreeze回调和PreStop回调方法签名相同，均只有一个Context输入参数，没有返回参数。定义如下：

```
function(ctx context.Context)
```

回调方法实现

在代码中实现生命周期回调方法，并使用相应函数进行注册。三种回调方法的注册方法如下：

```
// 注册Initializer回调方法
fc.RegisterInitializerFunction(initialize)
// 注册PreStop回调方法
fc.RegisterPreStopFunction(preStop)
// 注册PreFreeze回调方法
fc.RegisterPreFreezeFunction(preFreeze)
```

示例程序如下所示。

```
package main
import (
    "context"
    "log"
    "github.com/aliyun/fc-runtime-go-sdk/fc"
    "github.com/aliyun/fc-runtime-go-sdk/fccontext"
)
func HandleRequest(ctx context.Context) (string, error) {
    return "hello world!", nil
}
func preStop(ctx context.Context) {
    log.Print("this is preStop handler")
    fctx, _ := fccontext.FromContext(ctx)
    fctx.GetLogger().Infof("context: %#v\n", fctx)
}
func preFreeze(ctx context.Context) {
    log.Print("this is preFreeze handler")
    fctx, _ := fccontext.FromContext(ctx)
    fctx.GetLogger().Infof("context: %#v\n", fctx)
}
func initialize(ctx context.Context) {
    log.Print("this is initialize handler")
    fctx, _ := fccontext.FromContext(ctx)
    fctx.GetLogger().Infof("context: %#v\n", fctx)
}
func main() {
    fc.RegisterInitializerFunction(initialize)
    fc.RegisterPreStopFunction(preStop)
    fc.RegisterPreFreezeFunction(preFreeze)
    fc.Start(HandleRequest)
}
```

示例解析如下：

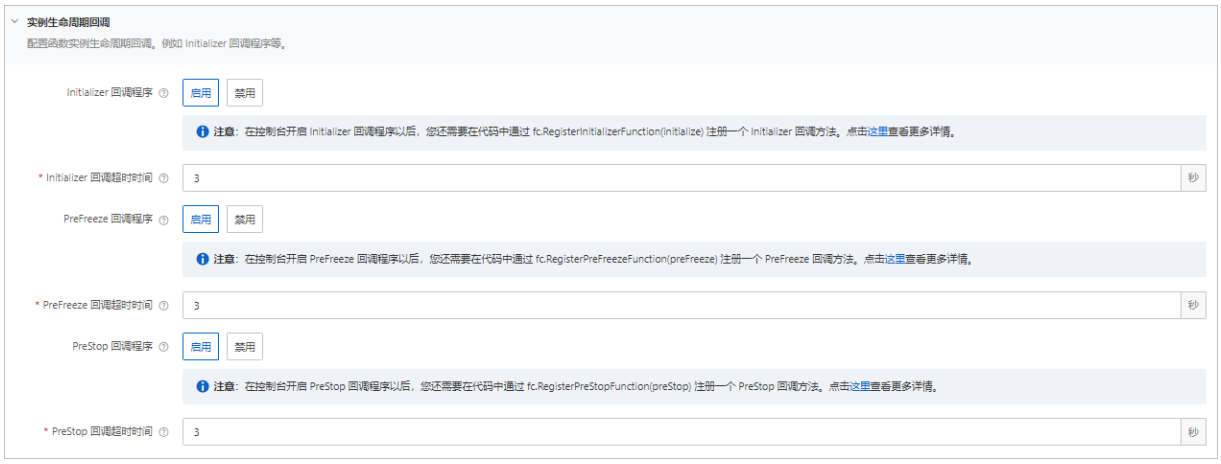
- `func initialize(ctx context.Context)`：Initializer回调方法。其中`ctx context.Context`参数提供了函数在调用时的运行信息。更多信息，请参见上下文。
- `func preStop(ctx context.Context)`：PreStop回调方法。其中`ctx context.Context`参数提供了函数在调用时的运行信息。更多信息，请参见上下文。
- `func preFreeze(ctx context.Context)`：PreFreeze回调方法。其中`ctx context.Context`参数提供了函数在调用时的运行信息。更多信息，请参见上下文。
- func main()：运行FC函数代码的入口点，Go程序必须包含`main`函数。通过添加代码`fc.Start(HandleRequest)`，设置请求处理程序的执行方法；通过添加代码`fc.RegisterInitializerFunction(initialize)`，注册Initializer回调方法。同理，注册PreStop和PreFreeze回调方法。

> 注意
>
> - 上述示例仅适用于事件请求处理程序（Event Handler）。如果是HTTP请求处理程序（HTTP Handler），您需要将示例中的 `fc.Start(HandleRequest)` 换成 `fc.StartHttp(HandleRequest)` 。
> - 注册生命周期回调方法必须在 `fc.Start(HandleRequest)` 或 `fc.StartHttp(HandleRequest)` 之前执行，否则会导致注册失败。

配置生命周期回调函数

通过控制台配置

在函数计算控制台的FC函数配置中，启用**Initializer回调程序**、**PreFreeze回调程序**和**PreStop回调程序**。示例如下：



具体步骤，请参见函数实例生命周期。

通过Serverless Devs配置

使用Serverless Devs配置

如果使用Serverless Devs工具，需要在 `s.yaml` 配置文件中添加**Initializer 回调程序**、**PreFreeze 回调程序**和**PreStop 回调程序**。

- Initializer回调配置
  在function配置下添加initializer和initializationTimeout两个字段。
- PreFreeze回调配置
  在function配置下添加instanceLifecycleConfig.preFreeze字段，包括handler和timeout两个字段。
- PreStop回调配置
  在function配置下添加instanceLifecycleConfig.preStop字段，包括handler和timeout两个字段。

> 注意 handler字段需配置为非空字符串，在 `s.yaml` 文件中使用默认值 `"true"` 时，必须携带双引号。

具体的示例如下所示。

```
edition: 1.0.0
name: hello-world  #  项目名称
access: default    #  密钥别名
vars: # 全局变量
  region: cn-shanghai # 地域
  service:
    name: fc-example
    description: 'fc example by serverless devs'
services:
  helloworld: # 业务名称或模块名称
    component: fc
    actions: # 自定义执行逻辑
      pre-deploy: # 在deploy之前运行
        - run: mvn package # 要运行的命令行
          path: ./ # 命令行运行的路径
    props: #  组件的属性值
      region: ${vars.region}
      service: ${vars.service}
      function:
        name: golang-lifecycle-hook-demo
        description: 'fc example by serverless devs'
        runtime: java8
        codeUri: ./code
        handler: main
        memorySize: 128
        timeout: 60
        initializationTimeout: 60
        initializer: "true"
        instanceLifecycleConfig:
          preFreeze:
            handler: "true"
            timeout: 30
          preStop:
            handler: "true"
            timeout: 30
```

关于Serverless Devs的YAML配置规范，请参见Serverless Devs操作命令。

查看日志

您可以在高级日志中，通过实例ID来查询实例生命周期回调函数的相关日志。具体操作，请参见函数实例生命周期。

示例程序

函数计算为您提供了Initializer回调的示例程序。该示例为您展示了如何使用Go运行时的Initializer回调来初始化MySQL连接池。在本示例中，MySQL数据库配置在函数的环境变量配置中（参考s.yaml）。Initializer回调从环境变量中获取数据库配置，创建MySQL连接池并测试连通性。

详细信息，请参见go-initializer-mysql。

4.Custom Runtime

4.1. 环境说明

本文介绍在函数计算中使用Custom Runtime编写函数的运行环境信息。

背景信息

Custom Runtime是自定义运行环境。基于Custom Runtime您可以打造属于您的运行环境，例如：

- 定制个性化语言，例如Rust。
- 定制编程语言指定版本的运行环境，例如Node.js 16。

容器环境

Custom Runtime的容器环境如下：

- 操作系统版本：Debian 9。
- 架构：x86_64。
- 用户权限：
  - 2021年12月01日00:00:00之后创建的函数，函数的执行用户为root。
  - 2021年12月01日00:00:00之前创建的函数，函数的执行用户为非root。
- 目录权限：
  - 2021年12月01日00:00:00之后创建的函数，所有目录均可写。
  - 2021年12月01日00:00:00之前创建的函数，只有*/tmp*目录可写。
- 代码在容器内目录位置：*/code*。

环境信息

Custom Runtime内置以下语言版本。您可以直接创建以下语言版本的Custom Runtime，无需安装第三方解释器：

- Python 3.7.4
- Node.js 10.16.2
- OpenJDK 1.8.0
- Ruby 2.7
- PowerShell 7.1.0
- Nginx 1.10.3
- PHP 7.4.12
  其中，PHP 7.4.12内置的扩展列表如下：

bcmath	calendar	Core
ctype	curl	date
dom	exif	FFI
fileinfo	filter	ftp
gd	gettext	hash

iconv	imagick	imap
intl	json	libxml
mbstring	mcrypt	memcached
mysqli	mysqlnd	openSSL
pcntl	pcre	PDO
pdo_mysql	pdo_pgsql	pdo_sqlite
pgsql	Phar	posix
protobuf	readline	redis
Reflection	session	shmop
SimpleXML	soap	sockets
sodium	SPL	sqlite3
standard	swoole	sysvmsg
sysvsem	sysvshm	tokenizer
xml	xmlreader	xmlrpc
xmlwriter	xsl	Zend OPcache
zip	zlib	无

使用非内置编程语言

当您打算使用某种语言打造Custom Runtime，但该语言不是Custom Runtime的内置语言时，您需要将该语言的解析器或运行时和代码文件一起打包部署到函数计算，实现您的预期目标。例如当运行环境是Node.js 16时，您需要先下载Node.js 16所需的解释器到代码中，然后再将打包后的代码部署到函数计算。具体操作如下所示：

1. 下载Linux-x64版本的 `node` 到代码包目录。

   ```
   wget http://mirrors.nju.edu.cn/nodejs/v16.14.2/node-v16.14.2-linux-x64.tar.gz -O node-v16.14.2-linux-x64.tar.gz && tar -zxvf node-v16.14.2-linux-x64.tar.gz && rm -rf node-v16.14.2-linux-x64.tar.gz
   ```

2. 设置Custom Runtime启动命令使用 `node` 。

```
customRuntimeConfig:
  command:
    - /code/node-v16.14.2-linux-x64/bin/node
  args:
    - 'server.js'
# 您也可以给函数设置环境变量 PATH=/code/node-v16.14.2-linux-x64/bin:/usr/local/bin/apache-maven/bin:/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/local/ruby/bin
# 这样就可以直接使用node启动HTTP Server
```

4.2. 基本原理

针对Custom Runtime，您的代码文件ZIP包是一个HTTP Server程序。本文介绍冷启动Custom Runtime的基本原理和HTTP Server配置要求。

基本原理

针对Custom Runtime，您的代码文件ZIP包是一个HTTP Server程序，您只需设置函数配置中的**启动命令**和**启动参数**完成HTTP Server的启动。函数计算冷启动Custom Runtime时，会调用您设置的**启动命令**和**启动参数**启动您自定义的HTTP Server，该HTTP Server接管了来自函数计算的所有请求。HTTP Server的默认端口是9000，如果您的HTTP Server是其他端口，比如8080，您可以设置函数配置中的监听端口为8080。

例如，函数的程序包名称为*function.zip*，该运行时各个语言程序包内的文件形式和相应的**启动命令**和**启动参数**示例如下：

Java 8或Spring Boot | Python 3.7 | Node.js 10 | PHP 7.4

```
.
├── demo.jar


customRuntimeConfig:
  command:
    - java
  args:
    - '-jar'
    - 'demo.jar'
```

> **说明** `customRuntimeConfig` 是函数的自定义启动命令配置。其中 `command` 是容器的入口命令列表，`args` 是启动参数列表。函数计算依次把 `command` 和 `args` 列表中的内容进行拼接，形成完整的启动命令。如果未配置启动命令及启动参数，HTTP Server将默认从*/code/bootstrap*启动。

HTTP Server配置要求

创建HTTP Server时您需要满足以下要求：

- Custom Runtime启动的服务一定要监听 `0.0.0.0:CAPort` 或 `*:CAPort` 端口。如果您使用 `127.0.0.1:CAPort` 端口，会导致请求超时，出现以下错误：

```
{
  "ErrorCode":"FunctionNotStarted",
  "ErrorMessage":"The CA's http server cannot be started:ContainerStartDuration:250000000. Ping CA failed due to: dial tcp 21.0.XX.XX:9000: getsockopt: connection refused Logs: 2019-11-29T09:53:30.859837462Z Listening on port 9000"
}
```

Custom Runtime的监听端口，即函数属性监听端口默认是9000。如果Custom Runtime使用默认的监听端口，那么您实现的Custom Runtime的HTTP Server监听的端口也必须是9000。如果Custom Runtime使用的监听端口是8080，那么您实现的Custom Runtime的HTTP Server监听的端口也必须是8080。

- Connection需要设置为Keep-Alive，Server端请求超时时间需设置在24小时（函数最大运行时间）及以上。示例如下：

```
//例如Node.js使用express时。
var server = app.listen(PORT, HOST);
server.timeout = 0; // never timeout
server.keepAliveTimeout = 0; // keepalive, never timeout
```

- HTTP Server需要在120秒内启动完毕。

4.3. 请求处理程序（Handler）

本文介绍在函数计算中使用Custom Runtime运行时开发请求处理程序的相关概念和方法。

什么是请求处理程序

FC函数的请求处理程序，是函数代码中处理请求的方法。当您的FC函数被调用时，函数计算会运行您提供的Handler方法处理请求。您可以通过函数计算控制台的**请求处理程序（函数入口）**配置Handler。

对Custom Runtime语言的FC函数而言，由于您本身已经实现了一个HTTP Server，因此，函数入口配置的Handler在绝大部分场景是无用的，即随意设置一个有效字符串即可。您可以在HTTP Server的逻辑中通过 `x-fc-function-handler` 这个Header获取函数入口配置的Handler来做您的自定义处理。

> **说明** 函数实例生命周期回调函数，包括Initializer函数、PreFreeze函数和PreStop函数配置的Handler同理。具体信息，请参见函数计算公共请求头。

关于FC函数的具体定义和相关操作，请参见管理函数。

配置说明

请求处理程序的具体配置均需符合函数计算平台的配置规范。配置规范因请求处理程序类型而异。

请求处理程序分为事件请求处理程序（Event Handler）和HTTP请求处理程序（HTTP Handler）；其中事件请求由各种事件源触发生成，HTTP请求则由HTTP触发器触发生成。两种请求处理程序的详细解释，请参见请求处理程序类型。

请求处理程序的具体配置示例，请分别参见事件请求处理程序（Event Handler）和HTTP请求处理程序（HTTP Handler）。

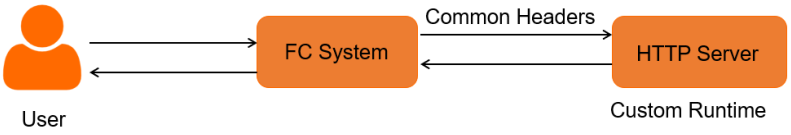
4.4. 事件请求处理程序（Event Handler）

本文介绍Custom Runtime中事件请求处理程序的结构特点、使用示例和常见问题。

背景信息

在Custom Runtime中，函数计算会将Common Headers、Body、POST、*/invoke*和*/initialize*路径转发给您

实现的HTTP Server。Common Headers里面的信息可以构造类似官方Runtime中的入参 `context` ，而您调用函数的请求体可以构造类似官方Runtime中的入参 `event` 。

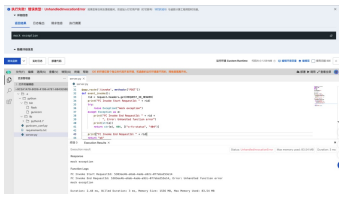


函数调用说明

当函数是事件请求处理程序时，Http Server仅需实现Path为 `/invoke` 和Method为 `POST` 的对应逻辑即可。

Path	输入请求	预期响应

Path	输入请求	预期响应

POST `/invoke`	• 请求体：函数输入（InvokeFunction时指定的Payload）。
	• 请求头：Common Request Headers。具体信息，请参见函数计算公共请求头。
	注意 `Content-Type` 为 `application/octet-stream` 。
		响应体：函数Handler的返回值，包括响应码和响应头。
		• 响应码 `StatusCode`
		  ○ `200` ：成功状态。
		  ○ `404` ：失败状态。
		• 响应头 `x-fc-status`
		  ○ `200` ：成功状态。
		  ○ `404` ：失败状态。
		通过Headers中的 `x-fc-status` 响应，向函数计算汇报本地函数是否执行成功。
		• 不设置 `x-fc-status` ：函数计算默认本次调用是成功执行的，但是您的函数可能有异常，没有向函数计算汇报，函数计算会认为这次函数执行没有报错，在业务逻辑上可能没有影响，但是在监控可观测性上会有影响。如下图所示：
		 
		• 设置 `x-fc-status` ：当您的函数存在异常，通过 `x-fc-status` 响应向函数计算汇报本次函数执行失败，并将错误堆栈信息打印到日志中。如下图所示：
		 
		说明 在返回的HTTP响应中，建议您同时设置 `StatusCode` 和 `x-fc-status` 。

多语言使用示例

常见问题

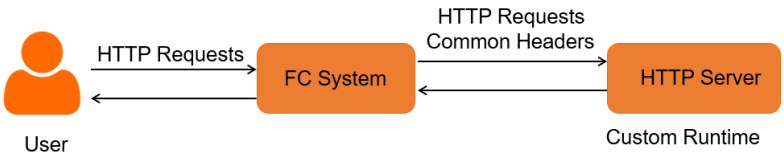
- Custom Runtime的监听端口一定要和HTTP Server的监听端口一致吗？
- Custom Runtime启动的服务中调用第三方服务时，出现FunctionNotStarted错误怎么办？
- 当我实现的HTTP Server在120s内无法成功启动怎么办？
- 遇到Process exited unexpectedly before completing request怎么办？
- 当我使用浏览器或cURL方式访问函数时出现404怎么办？

4.5. HTTP请求处理程序（HTTP Handler）

本文介绍Custom Runtime中HTTP请求处理程序的结构特点、调用说明、使用限制、使用示例和常见问题。

背景信息

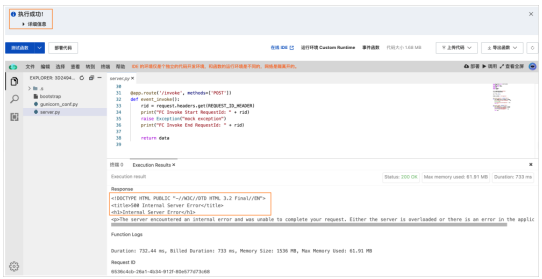
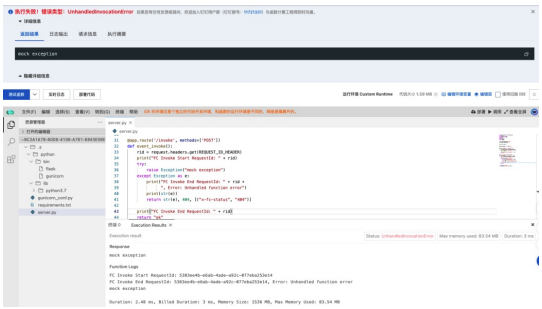
函数计算会将用户的请求，包括Method、Path、Body、Query和Headers以及函数计算生成的Common Header透传给HTTP Server。您可以平滑迁移已有的HTTP Web应用。



函数调用说明

当HTTP请求处理程序被调用时，和调用一个Web API方式一样，您可以直接使用cURL、Postman或浏览器等方式直接请求调用。如果您是通过浏览器访问HTTP触发器的，对应的函数被强制下载时，请参见解决方法。

Header	描述
（可选）x-fc-base-path	当您尚未配置自定义域名时，x-fc-base-path的值是 `/2016-08-15/proxy/${servicename}/${functionname}/` 。

Header	描述
（可选）x-fc-status	当您的HTTP函数不是应用一键迁移，而是单纯新开发的Web API时，其性质和事件函数类似。 通过Headers中的 `x-fc-status` 响应，向函数计算汇报本地函数是否执行成功。 • 不设置 `x-fc-status` ：函数计算默认本次调用是成功执行的，但是您的函数可能有异常，没有向函数计算汇报，函数计算会认为这次函数执行没有报错，在业务逻辑上可能没有影响，但是在监控可观测性上会有影响。如下图所示：  • 设置 `x-fc-status` ：当您的函数存在异常，通过 `x-fc-status` 响应向函数计算汇报本次函数执行失败，并将错误堆栈信息打印到日志中。如下图所示：  **说明** 在返回的HTTP响应中，建议您同时设置 `StatusCode` 和 `x-fc-status` 。

使用限制

- 一个HTTP函数的一个版本或别名，最多只能创建一个HTTP类型的触发器。详细信息，请参见管理版本和管理别名。
- HTTP Request限制
  - Request Headers不支持以x-fc-开头的自定义字段和以下自定义字段：
    - connection
    - keep-alive

- 如果Request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：包括所有的Query Params，Path的总大小不得超过4 KB。
  - Body大小：同步调用请求的Body的总大小不得超过16 MB，异步调用请求的Body的总大小不得超过128 KB。

- HTTP Response限制
  - Response Headers不支持以x-fc-开头的自定义字段和以下自定义字段：
    - connection
    - content-length
    - date
    - keep-alive
    - server
    - 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 Headers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。详细信息，请参见配置自定义域名。

  - 如果Response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。

- 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。详细信息，请参见配置自定义域名。
  您还可以使用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。详细信息，请参见使用函数计算作为API网关后端服务。

多语言使用示例

常见问题

- Custom Runtime的监听端口一定要和HTTP Server的监听端口一致吗？
- Custom Runtime启动的服务中调用第三方服务时，出现FunctionNotStarted错误怎么办？
- 当我实现的HTTP Server在120s内无法成功启动怎么办？
- 遇到Process exited unexpectedly before completing request怎么办？
- 当我使用浏览器或cURL方式访问函数时出现404怎么办？

4.6. 函数实例生命周期回调

本文介绍Custom Container Runtime实现函数实例生命周期回调的方法。

回调方法

当您实现并配置函数实例生命周期回调后，函数计算将在相关实例生命周期事件发生时调用对应的回调程序。函数实例生命周期涉及Initializer、PreFreeze和PreStop三种回调。更多信息，请参见函数实例生命周期回调。

Path	输入请求	期望的响应

（可选）POST `/initialize`	请求体：无。
	请求头：Common Request Headers。具体信息，请参见函数计算公共请求头。
		响应体：函数 `Initializer` 的返回值。
		StatusCode
		• 200：成功状态。
		• 404：失败状态。
		Python语言中关于 `initialize` 的示例代码如下：
		`@app.route('/initialize', methods=['POST'])`
		`def init_invoke():`
		`rid = request.headers.get(x-fc-request-id)`
		`print("FC Initialize Start RequestId: " + rid)`
		`# do your things`
		`print("FC Initialize End RequestId: " + rid)`
		`return "OK"`

（可选）GET `/pre-freeze`	• 请求体：无。 • 请求头：Common Request Headers。具体信息，请参见函数计算公共请求头。	响应体：函数PreFreeze的返回值。 StatusCode • 200：成功状态。 • 404：失败状态。
（可选）GET `/pre-stop`	• 请求体：无。 • 请求头：Common Request Headers。具体信息，请参见函数计算公共请求头。	响应体：函数PreStop的返回值。 StatusCode • 200：成功状态。 • 404：失败状态。

如果您想在Custom Runtime中使用Initializer回调方法，您只需在您的HTTP Server中实现Path为 `/initialize` 和Method为 `POST` 的对应逻辑即可。示例代码，请参见上表中关于 `initialize` 的示例代码。

> **注意** 如果创建的函数不设置Initializer，就无需实现 `/initialize` 。此时，即使HTTP Server实现了 `/initialize` ，代码中的 `/initialize` 逻辑也无法被调用执行。

PreFreeze和PreStop回调方法的使用，同Initializer回调方法。

4.7. Spec细则

本文介绍Custom Runtime的公共请求头、响应码和响应头以及日志格式，您可以参见这些信息打造属于您的自定义运行环境。

函数计算公共请求头

Custom Runtime从函数计算中接收到的公共请求头如下表所示。如果您需要访问阿里云其他服务，您可能需要用到临时AccessKey的Headers。如果您需要迁移已有的应用，可忽略下文的内容。

> **说明**
> - 事件函数和HTTP函数均包含Common Headers。
> - 公共请求头是函数计算自动生成的，主要包含权限信息和函数的基本信息等。

Header	描述
x-fc-request-id	Request ID。
x-fc-access-key-id	临时AccessKey ID。
x-fc-access-key-secret	临时AccessKey Secret。
x-fc-security-token	临时Security Token。
x-fc-function-handler	函数的Handler，如果Runtime本身就是函数（例如Custom Runtime或者Custom Container函数），则该值无意义，设置为一个随机字符串即可。
x-fc-function-memory	函数最大能使用的内存。
x-fc-function-initializer	Initializer函数的Handler，如果Runtime本身就是函数（例如Custom Runtime或者Custom Container函数），则该值无意义，设置为一个随机字符串即可。
x-fc-initialization-timeout	Initializer函数执行的超时时间。
x-fc-instance-lifecycle-pre-stop-handler	PreStop函数的Handler，如果Runtime本身就是函数（例如Custom Runtime或者Custom Container函数），则该值无意义，设置为一个随机字符串即可。
x-fc-instance-lifecycle-pre-freeze-handler	PreFreeze函数的Handler，如果Runtime本身就是函数（例如Custom Runtime或者Custom Container函数），则该值无意义，设置为一个随机字符串即可。
x-fc-region	函数所在的地域。
x-fc-account-id	函数所有者的UID。
x-fc-qualifier	函数调用时指定的服务版本或别名。更多信息，请参见灰度发布示例。
x-fc-version-id	函数调用时指定的服务版本。
x-fc-function-name	函数名称。
x-fc-service-name	函数所在的服务的名字。
x-fc-service-logproject	函数所在服务配置的日志项目。

Header	描述
x-fc-service-logstore	函数所在服务配置的日志库。
x-fc-control-path	函数的请求类型。 对于Custom Runtime或Custom Container，您可以根据Headers中的参数来判断函数调用是HTTP函数调用还是事件函数调用。参数信息如下： • */invoke*：该请求为事件函数调用。*/invoke*表示是Invoke函数调用请求。 • */http-invoke*：该请求为HTTP函数调用。*/http-invoke*表示是HTTP invoke函数调用请求，函数计算会将您的请求（包括Path、Body和Headers）加上Common Headers后转发给Custom Runtime或Custom Container，Custom Runtime或Custom Container返回的响应头和响应体则会被返回给客户端。 • */initialize*：*/initialize*表示第一次创建执行环境时，函数计算自动发起的Initialize函数调用请求。在容器的生命周期内，有且仅成功调用一次，类似于Class构造函数。

函数计算响应码和响应头

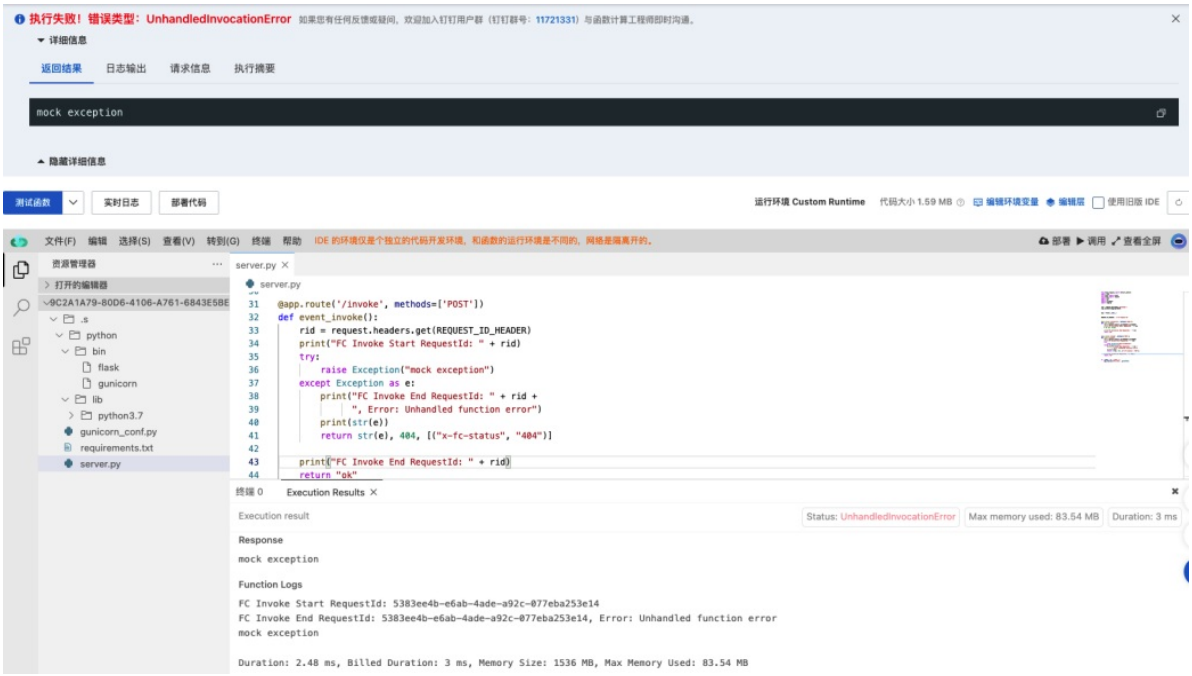
Custom Runtime本质是您实现的HTTP Server，因此每一次函数调用都是一次HTTP请求，即每次响应都有响应码和响应头。

- 响应码 `StatusCode`
  - `200`：成功状态。
  - `404`：失败状态。
- 响应头 `x-fc-status`
  - `200`：成功状态。
  - `404`：失败状态。

通过Headers中的 `x-fc-status` 响应，向函数计算汇报本地函数是否执行成功。

- 不设置 `x-fc-status`：函数计算默认本次调用是成功执行的，但是您的函数可能有异常，没有向函数计算汇报，函数计算会认为这次函数执行没有报错，在业务逻辑上可能没有影响，但是在监控可观测性上会有影响。如下图所示：

- 设置 `x-fc-status` ：当您的函数存在异常，通过 `x-fc-status` 响应向函数计算汇报本次函数执行失败，并将错误堆栈信息打印到日志中。如下图所示：



> **说明** 在返回的HTTP响应中，建议您同时设置 `StatusCode` 和 `x-fc-status` 。

函数日志格式

建议您在创建服务时启用日志功能，Custom Runtime中所有打印到标准输出（Stdout）的日志会自动收集到您指定的日志服务中。具体步骤，请参见配置日志。

函数计算在其他运行环境，即除Custom Runtime和Custom Container以外的运行环境中调用函数时，如果请求头中包含 `x-fc-log-type" = "Tail"` ，那么返回的响应头包含 `x-fc-log-result` 的内容就是函数执行时打印的日志，日志上限为4 KB。您可以在函数计算控制台函数执行结果中查看该日志。如果您希望在控制台的执行结果中看到Custom Runtime的运行日志，需要在代码中记录请求开始和结束的日志。

日志内容	是否必选	代码
启动Runtime	否 **说明** 该内容为函数冷启动的标志。	`FunctionCompute ${runtime} runtime inited.` **说明** `${runtime}` 为自定义语言类型，建议您避免使用函数计算官方语言名称，例如Node.js、Python或PHP等。
Invoke开始的日志	是	`FC Invoke Start RequestId: ${RequestId}`
Invoke结束的日志	是	`FC Invoke End RequestId: ${RequestId}`
Initialize开始的日志	否 **说明** 如果是Initializer函数，则需要记录。	`FC Initialize Start RequestId: ${RequestId}`
Initialize结束的日志	否 **说明** 如果是Initializer函数，则需要记录。	`FC Initialize End RequestId: ${RequestId}`

除了以上特殊的信息外，推荐您在自己的日志中包含请求ID，方便日后诊断问题。推荐日志格式为 `$utcdatetime(yyyy-MM-ddTHH:mm:ss.fff) $requestId [$Level] $message` 。

说明 不同语言下指定日志级别的接口不同，请您根据实际运行环境设置。更多信息，请参见基础信息。

相关文档

- 环境说明
- 基本原理
- 事件请求处理程序（Event Handler）
- HTTP请求处理程序（HTTP Handler）
- 函数实例生命周期回调

5.Custom Container

5.1. Custom Container简介

本文介绍Custom Container（自定义容器运行环境）的背景信息、基本原理、HTTP Server配置要求、日志格式、冷启动优化、计费说明及使用限制。

背景信息

在云原生时代，容器镜像已经逐渐变成了软件部署和开发的标准工具。为了优化开发者体验、提升开发和交付效率，函数计算提供了Custom Container。开发者将容器镜像作为函数的交付物，通过HTTP协议和函数计算系统交互。使用Custom Container具有以下优势：

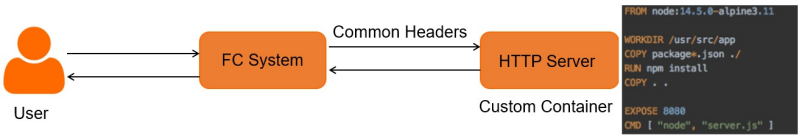
- 低成本迁移，无需修改代码或重新编译二进制、共享对象（*.so），保持开发和线上环境一致。
- 避免代码和依赖分离，简化分发和部署的步骤。
- 容器镜像天然的分层缓存，提高代码上传和拉取效率。
- 标准可复用的第三方库引用、分享、构建、代码上传、存储和版本管理，丰富的开源生态CI/CD交付体验。

基本原理

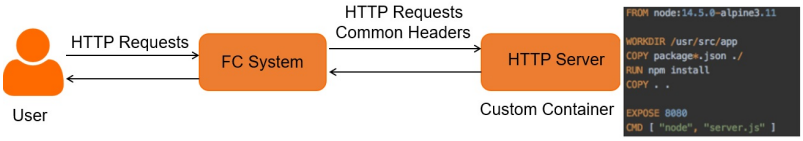
Custom Container工作原理与Custom Runtime基本相同。函数计算系统初始化执行环境实例前会扮演该函数的服务角色（Service Role），获得临时用户名和密码并拉取镜像。拉取成功后根据指定的启动命令Command、参数Args及CAPort端口（默认9000）启动您定义的HTTP Server。然后这个HTTP Server接管了函数计算系统的所有请求，包括来自您的事件函数和HTTP函数的调用。

您在开发函数具体的逻辑之前，一般需要确认开发的是事件函数还是HTTP函数，原理如下所示：

- 事件函数



- HTTP函数



HTTP Server配置要求

- Custom Container启动的服务一定要监听 `0.0.0.0:CAPort` 或 `*:CAPort` 端口。如果您使用 `127.0.0.1:CAPort` 端口，会导致请求超时，出现以下错误：

```
{
  "ErrorCode":"FunctionNotStarted",
  "ErrorMessage":"The CA's http server cannot be started:ContainerStartDuration:2500000000. Ping CA failed due to: dial tcp 21.0.XX.XX:9000: getsockopt: connection refused Logs: 2019-11-29T09:53:30.859837462Z Listening on port 9000"
}
```

Custom Container的监听端口，即函数属性CAPort默认是9000。如果Custom Container使用默认的监听端口，那么实现的Custom Container的HTTP Server监听的端口也必须是9000。 如果Custom Container使用的监听端口是8080，那么实现的Custom Container的HTTP Server监听的端口也必须是8080。

- Connection需要设置为Keep-Alive，Server端请求超时时间需设置在15分钟及以上。示例如下：

```
//例如Node.js使用Express时。
var server = app.listen(PORT, HOST);
server.timeout = 0; // never timeout
server.keepAliveTimeout = 0; // keepalive, never timeout
```

- HTTP Server需要在120秒内启动完毕。

公共请求头

Custom Container的公共请求头和Custom Runtime的公共请求头一致。详细信息，请参见函数计算公共请求头。

日志格式

Custom Container的日志格式与Custom Runtime的日志格式一致。详细信息，请参见函数日志格式。

冷启动优化最佳实践

相比于代码包，容器镜像依赖的基础环境带来了额外的数据下载和解压的时间，为了更好的冷启动体验，推荐您使用以下最佳实践：

- 容器镜像地址推荐使用与函数计算同地域的VPC镜像地址，例如*registry-vpc.cn-hangzhou.aliyuncs.com/fc-demo/helloworld:v1beta1*，以获得最短的镜像拉取延时和稳定性。
- 镜像最小化，基于类似Alpine或Ubuntu这样的最小镜像或者是其他镜像中精简版本构建自定义镜像。仅保留必要的依赖，删除不必要的文档、数据和其他文件。
- 容器镜像配合预留实例一起使用。更多信息，请参见弹性管理（含预留模式）。
- 在资源允许和线程安全的情况下，搭配使用单实例多并发功能，可避免不必要的冷启动，同时降低成本。更多信息，请参见设置实例并发数。
- 函数创建或更新后，函数计算将默认开启镜像启动加速功能。更多信息，请参见镜像启动加速（ACR个人版）和镜像启动加速（ACR企业版）。

计费说明

Custom Container的计费项与其他类型Runtime的计费项一致。更多信息，请参见计费概述。

其中，镜像资源使用量中的执行时间是从仓库拉取镜像到拉取镜像结束的时长。例如，1024 MB内存的实例拉取镜像耗时10秒，则本次拉取镜像的资源消耗量为10 GB-秒。

容器镜像在一段时间一定范围内有缓存，因此并不是每次冷启动一定会产生镜像拉取的费用。

使用限制

- 镜像大小限制
  ACR企业版的标准版和高级版最大支持10 GB的未解压镜像，ACR企业版的基础版和ACR个人版最大支持3 GB的未解压镜像。
- 镜像启动加速
  函数创建或更新后，需等待加速镜像准备完成后，才能在函数计算控制台触发函数调用。
- 镜像仓库
  支持拉取阿里云容器镜像服务的企业版和个人版的镜像。详细信息，请参见什么是容器镜像服务ACR。
- 镜像访问
  目前仅ACR个人版公开镜像支持跨账号跨地域读取，其余镜像仅支持同账号同地域下私有镜像仓库读取。

- 容器内文件读写权限
  容器run-as-user UID默认是Root。如果您在*Dockerfile*中指定了具体的使用者，则会以指定的使用者运行该容器镜像。
- 容器可写层存储空间限制
  排除只读镜像层，容器产生的数据最大不能超过512 MB。

  > **说明** 容器可写层数据不是持久化的，数据会随着容器被销毁而删除。如果需要持久化存储，可以考虑使用函数计算挂载NAS。具体操作，请参见配置NAS文件系统。或者使用其他共享存储服务，例如对象存储或表格存储等。

- 镜像架构限制
  目前函数计算仅支持AMD64镜像架构，因此，针对搭载Apple芯片的Mac电脑（或其他ARM架构的机器），构建镜像时需要指定镜像的编译平台为Linux/Amd64。参考命令如 `docker build --platform linux/amd64 -t $IMAGE_NAME .` 。

  > **说明** 构建完成后，可执行 `docker inspect` 进行检查。如果返回内容包含 `"Architecture": "amd64"` ，则表示构建的镜像正确。

相关文档

- 请求处理程序（Handler）
- 事件请求处理程序（Event Handler）
- HTTP请求处理程序（HTTP Handler）
- 创建Custom Container函数
- 镜像启动加速（ACR个人版）
- 镜像启动加速（ACR企业版）
- 函数实例生命周期回调

5.2. 请求处理程序（Handler）

本文介绍在函数计算中使用Custom Container运行时开发请求处理程序的相关概念和方法。

什么是请求处理程序

FC函数的请求处理程序，是函数代码中处理请求的方法。当您的FC函数被调用时，函数计算会运行您提供的Handler方法处理请求。您可以通过函数计算控制台的**请求处理程序（函数入口）**配置Handler。

对于Custom Container语言的FC函数，由于您本身已经实现了一个HTTP Server，因此，您在函数计算控制台**请求处理程序（函数入口）**配置的Handler在绝大部分场景是无用的，即随意设置一个有效字符串即可。您可以在HTTP Server的逻辑中通过 `x-fc-function-handler` 这个Header获取**请求处理程序（函数入口）**配置的Handler来做您的自定义处理。

关于FC函数的具体定义和相关操作，请参见管理函数。

配置说明

请求处理程序的具体配置均需符合函数计算平台的配置规范。配置规范因请求处理程序类型而异。

请求处理程序分为事件请求处理程序（Event Handler）和HTTP请求处理程序（HTTP Handler）；其中事件请求由各种事件源触发生成，HTTP请求则由HTTP触发器触发生成。两种请求处理程序的详细解释，请参见请求处理程序类型。

请求处理程序的具体配置示例，请分别参见事件请求处理程序（Event Handler）和HTTP请求处理程序（HTTP Handler）。

5.3. 事件请求处理程序（Event Handler）

本文介绍Custom Container中事件请求处理程序的结构特点、使用示例和常见问题。

背景介绍

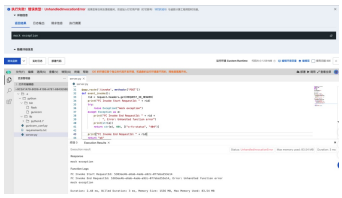
在Custom Container中，函数计算会将Common Headers、Body、POST方法以及*/invoke*、*/initialize*路径转发给容器中的HTTP Server。您可以选择实现类似官方支持的Runtime（例如Golang Runtime）的 `context` 和 `event` 函数签名。您也可以直接使用入参请求头（Headers）和请求体 （Body）来编写函数的业务逻辑。更多信息，请参见Custom Runtime事件函数调用。

函数调用说明

当函数是事件请求处理程序时，Http Server仅需实现Path为 `/invoke` 和Method为 `POST` 的对应逻辑即可。

Path	输入请求	预期响应

Path	输入请求	预期响应

POST `/invoke`	• 请求体：函数输入（InvokeFunction时指定的Payload）。
	• 请求头：Common Request Headers。具体信息，请参见函数计算公共请求头。
	注意 `Content-Type` 为 `application/octet-stream` 。
		响应体：函数Handler的返回值，包括响应码和响应头。
		• 响应码 `StatusCode`
		  ○ `200` ：成功状态。
		  ○ `404` ：失败状态。
		• 响应头 `x-fc-status`
		  ○ `200` ：成功状态。
		  ○ `404` ：失败状态。
		通过Headers中的 `x-fc-status` 响应，向函数计算汇报本地函数是否执行成功。
		• 不设置 `x-fc-status` ：函数计算默认本次调用是成功执行的，但是您的函数可能有异常，没有向函数计算汇报，函数计算会认为这次函数执行没有报错，在业务逻辑上可能没有影响，但是在监控可观测性上会有影响。如下图所示：
		 
		• 设置 `x-fc-status` ：当您的函数存在异常，通过 `x-fc-status` 响应向函数计算汇报本次函数执行失败，并将错误堆栈信息打印到日志中。如下图所示：
		 
		说明 在返回的HTTP响应中，建议您同时设置 `StatusCode` 和 `x-fc-status` 。

代码示例

在以下Node.js Express示例中，POST方法和*/initialize*路径会在函数实例初始化时被函数计算调用，POST方法和*/invoke*路径为函数计算被调用时的Handler，通过 `req.headers` 以及 `req.body` 获取 `context` 和 `event` 并将函数返回结果通过HTTP Response结构体输出。

```
'use strict';
const express = require('express');
// Constants
const PORT = 9000;
const HOST = '0.0.0.0';
const app = express();
// Parse request body as JSON
app.use(express.json())
// initialize example, need config Initializer in function meta
app.post('/initialize', (req, res) => {
  console.log(req.body)
  res.send('Hello FunctionCompute, /initialize\n');
});
// Event function invocation
app.post('/invoke', (req, res) => {
  console.log(req.body)
  res.send('Hello FunctionCompute, event function\n');
});
var server = app.listen(PORT, HOST);
console.log(`Running on http://${HOST}:${PORT}`);
server.timeout = 0; // never timeout
server.keepAliveTimeout = 0; // keepalive, never timeout
```

多语言使用示例

常见问题

- Custom Container的监听端口一定要和HTTP Server的监听端口一致吗？
- Custom Container Runtime启动的服务中调用第三方服务时，出现FunctionNotStarted错误怎么办？
- 当我实现的HTTP Server在120s内无法成功启动怎么办？
- 遇502报错且报错信息为Process exited unexpectedly before completing request怎么办？
- 当我使用浏览器或cURL方式访问函数时出现404怎么办？
- 如未使用阿里云的公网容器镜像，我需要给函数计算的服务角色授予什么权限？

5.4. HTTP请求处理程序（HTTP Handler）

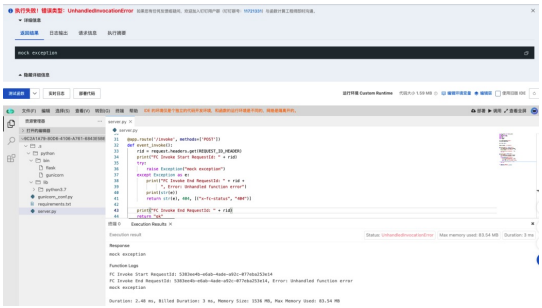
本文介绍Custom Container中HTTP请求处理程序的结构特点、调用说明、使用限制、使用示例和常见问题。

背景信息

函数计算会将用户的请求，包括Method、Path、Body、Query和Headers以及函数计算系统生成的Common Header转发给HTTP Server。您可以平滑迁移已有的HTTP Web应用。更详细的介绍，请参见Custom Runtime的HTTP函数。

函数调用说明

当HTTP请求处理程序被调用时，和调用一个Web API方式一样，您可以直接使用cURL、Postman或浏览器等方式直接请求调用。如果您是通过浏览器访问HTTP触发器的，对应的函数被强制下载时，请参见解决方法。

Header	描述
（可选）x-fc-base-path	当您尚未配置自定义域名时，x-fc-base-path的值是 `/2016-08-15/proxy/${servicename}/${functionname}/` 。
（可选）x-fc-status	当您的HTTP函数不是应用一键迁移，而是单纯新开发的Web API时，其性质和事件函数类似。 通过Headers中的 `x-fc-status` 响应，向函数计算汇报本地函数是否执行成功。 • 不设置 `x-fc-status` ：函数计算默认本次调用是成功执行的，但是您的函数可能有异常，没有向函数计算汇报，函数计算会认为这次函数执行没有报错，在业务逻辑上可能没有影响，但是在监控可观测性上会有影响。如下图所示：  • 设置 `x-fc-status` ：当您的函数存在异常，通过 `x-fc-status` 响应向函数计算汇报本次函数执行失败，并将错误堆栈信息打印到日志中。如下图所示：  **说明** 在返回的HTTP响应中，建议您同时设置 `StatusCode` 和 `x-fc-status` 。

使用限制

- 一个HTTP函数的一个版本或别名，最多只能创建一个HTTP类型的触发器。详细信息，请参见管理版本和管理别名。
- HTTP Request限制
  - Request Headers不支持以x-fc-开头的自定义字段和以下自定义字段：
    - connection
    - keep-alive
  - 如果Request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：包括所有的Query Params，Path的总大小不得超过4 KB。
    - Body大小：同步调用请求的Body的总大小不得超过16 MB，异步调用请求的Body的总大小不得超过128 KB。
- HTTP Response限制
  - Response Headers不支持以x-fc-开头的自定义字段和以下自定义字段：
    - connection
    - content-length
    - date
    - keep-alive
    - server
    - 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 Headers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。详细信息，请参见配置自定义域名。
  - 如果Response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。
- 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。详细信息，请参见配置自定义域名。
  您还可以使用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。详细信息，请参见使用函数计算作为API网关后端服务。

代码示例

在以下Node.js Express示例中，GET和POST方法分别路由至不同的Handler。您也可以根据需要做任意的Path和Handler映射。

```
'use strict';
const express = require('express');
// Constants
const PORT = 9000;
const HOST = '0.0.0.0';
// HTTP function get
const app = express();
// Parse request body as JSON
app.use(express.json())
app.get('/*', (req, res) => {
  console.log(req.body)
  res.send('Hello FunctionCompute, http GET');
});
app.post('/*', (req, res) => {
  console.log(req.body)
  res.send('Hello FunctionCompute, http POST');
});
app.listen(PORT, HOST);
console.log(`Running on http://${HOST}:${PORT}`);
```

多语言使用示例

常见问题

- Custom Container的监听端口一定要和HTTP Server的监听端口一致吗？
- Custom Container Runtime启动的服务中调用第三方服务时，出现FunctionNotStarted错误怎么办？
- 当我实现的HTTP Server在120s内无法成功启动怎么办？
- 遇502报错且报错信息为Process exited unexpectedly before completing request怎么办？
- 当我使用浏览器或cURL方式访问函数时出现404怎么办？
- 如未使用阿里云的公网容器镜像，我需要给函数计算的服务角色授予什么权限？

5.5. 创建Custom Container函数

本文介绍如何在函数计算控制台或使用Serverless Devs创建Custom Container函数。

前提条件

- 容器镜像服务
  - 创建实例

    > **说明** 您可以创建个人版或企业版实例，推荐您创建企业版实例。

  - 创建命名空间
  - 创建镜像仓库
- Serverless Devs
  - 安装Serverless Devs和Docker
  - 配置Serverless Devs。

在控制台创建函数

本文以在*/tmp*目录中执行为例，介绍如何创建函数，假设函数计算的地域为华南1（深圳），镜像仓库名称*nodejs-express*。

1. 将您的函数镜像推送至默认实例镜像仓库。
   i. 执行以下命令进入*/tmp*目录。
   ```
   cd /tmp
   ```
   ii. 在*/tmp*目录中执行以下命令克隆示例工程。
   ```
   git clone https://github.com/awesome-fc/custom-container-docs.git
   ```
   iii. 执行以下命令，进入项目目录。
   ```
   cd custom-container-docs/nodejs-express
   ```
   iv. 执行以下命令指定镜像仓库。
   ```
   export IMAGE_NAME="registry.cn-shenzhen.aliyuncs.com/fc-demo/nodejs-express:v0.2"
   ```
   v. 执行以下命令打包镜像。
   ```
   docker build -t $IMAGE_NAME .
   ```
   > **说明** 针对搭载Apple芯片的Mac电脑（或其他ARM架构的机器），构建镜像时需要指定镜像的编译平台为Linux/Amd64。实现跨平台编译，示例代码如 `docker build --platform linux/amd64 -t $IMAGE_NAME .` 。
   vi. 执行以下命令推送镜像。
   ```
   docker push $IMAGE_NAME
   ```
2. 创建服务并为服务设置权限。
   i. 在函数计算控制台创建服务。具体操作步骤，请参见创建服务。
   ii. 为目标服务绑定权限策略AliyunContainerRegistryReadOnlyAccess或者AliyunContainerRegistryFullAccess。具体操作步骤，请参见授予函数计算访问其他云服务的权限。
   函数计算需要使用上述策略中的权限去获取容器镜像服务中默认实例的临时账号，然后利用该临时账号推送位于您的私有镜像仓库中的镜像。
3. 创建函数。
   i. 登录函数计算控制台。
   ii. 在左侧导航栏，单击**服务及函数**。
   iii. 在顶部菜单栏，选择地域。
   iv. 在**服务列表**页面，单击目标服务。
   v. 在**函数管理**页面，单击**创建函数**。
   vi. 在**创建函数**页面，选择**使用容器镜像创建**，在**基本设置**区域设置相关参数，然后单击**创建**。

参数	是否必填	操作	示例值

参数	是否必填	操作	示例值
函数名称	否	填写自定义的函数名称。 **说明** 如果不填写名称，函数计算会自动为您创建。	Function
容器镜像	是	单击**选择 ACR 中的容器镜像**，在弹出的**选择容器镜像**对话框，选择已创建的**容器镜像实例**和**ACR镜像仓库**，然后在下方选择镜像区域找到目标镜像并在其右侧**操作**列单击**选择**。	test-registry-vpc.cn-chengdu.cr.aliyuncs.com/test/registry:latest
启动命令	否	容器的启动命令。如果不填写，则默认使用镜像中的Entrypoint/CMD。	python app.py
监听端口	是	容器镜像中的HTTP Server所监听的端口。默认端口为9000。	9000
请求处理程序类型	是	选择请求处理程序类型。 ■ **处理事件请求**：通过定时器、调用API/SDK或其他阿里云服务的触发器来触发函数执行。 ■ **处理 HTTP 请求**：用于处理HTTP请求或Websocekt请求的函数。	通过事件触发
实例类型	是	选择适合您的实例类型。 ■ **弹性实例** ■ **性能实例** ■ **GPU实例**（公测中） 更多信息，请参见实例类型及使用模式。关于各种实例类型的计费详情，请参见计费概述。 **说明** 如需使用GPU实例，请提交工单申请。具体操作，请参见实例类型。	弹性实例
内存规格	是	设置函数执行内存。 ■ **选择输入**：在下拉列表中选择所需内存。 ■ **手动输入**：仅适用于弹性实例，单击**手动输入内存大小**，可自定义函数执行内存。取值范围[128, 3072]，单位为MB。 **说明** 输入的内存必须为64 MB的倍数。	512 MB

参数	是否必填	操作	示例值
GPU显存规格	是	实例类型选择**GPU实例**时必填。	8 G
实例并发度	是	设置函数实例的并发度，具体信息，请参见设置实例并发数。	1

> **说明** 您也可以在**配置触发器**区域为即将创建的函数配置触发器，具体操作步骤，请参见触发器管理。

创建完成后，您可以在目标服务下的函数列表中看到刚才创建的函数。

使用Serverless Devs创建

使用Serverless Devs可以一键构建、推送容器镜像并部署函数。

1. 执行以下命令，初始化项目。

```
s init start-fc-custom-container-event-nodejs14
```

输出示例：

```
Serverless Awesome: https://github.com/Serverless-Devs/package-awesome
Please input your project name (init dir) start-fc-custom-container-event-nodejs14
file decompression completed

    ____  _     _ ___  _ _     _        _____ ____
   /  _ \/ \   / \\  \///\ /\/ \  /|  /    //   _\
   | / \|| |   | | \  / | | ||| |\ ||  |  __\|  /
   | |-||| |_/\| | / /  | \_/|| | \||  | |   |  \__
   \_/ \|\____/\_//_/   \____/\_/  \|  \_/   \____/
please select credential alias default
   Welcome to the Aliyun FC start application
    This application requires to open these services:
        FC : https://fc.console.aliyun.com/
        ACR: https://cr.console.aliyun.com/
    * 额外说明：s.yaml中声明了actions：
       部署前执行：s build --use-docker --dockerfile ./code/Dockerfile
      如果不需要每次都构建项目，或者部署前不需要构建，或者已经手动构建了，可以注释掉这部分内容。
      > PS：部署的时候还需要修改s.yaml中image字段为自己的ACR配置的地址。
    * 项目初始化完成，您可以直接进入项目目录下，并使用s deploy进行项目部署。
 Thanks for using Serverless-Devs
 You could [cd /test/test1/start-fc-custom-container-event-nodejs14] and enjoy your ser
verless journey!
 If you need help for this example, you can use [s -h] after you enter folder.
 Document Star：https://github.com/Serverless-Devs/Serverless-Devs
 Do you want to deploy the project immediately? No
```

2. 执行以下命令，进入项目目录。

```
cd start-fc-custom-container-event-nodejs14
```

3. 编辑*s.yaml*文件。

   将该文件内的 `image` 的参数信息修改为您的ACR镜像地址。

4. 执行以下命令，部署项目。

```
s deploy
```

输出示例：

```
[2021-12-15 07:54:30] [INFO] [S-CLI] - Start ...
[2021-12-15 07:54:30] [INFO] [S-CLI] - Start the pre-action
[2021-12-15 07:54:30] [INFO] [S-CLI] - Action: s build --use-docker --dockerfile ./code
/Dockerfile
[2021-12-15 07:54:31] [INFO] [S-CLI] - Start ...
[2021-12-15 07:54:32] [INFO] [FC-BUILD] - Build artifact start...
[2021-12-15 07:54:32] [INFO] [FC-BUILD] - Use docker for building.
[2021-12-15 07:54:32] [INFO] [FC-BUILD] - Building image...
Sending build context to Docker daemon   5.12kB
Step 1/7 : FROM node:14.5.0-alpine3.11
 ---> 5d97b3d11dc1
......
Step 7/7 : ENTRYPOINT [ "node", "server.js" ]
 ---> Using cache
 ---> a5ef1c015e7e
Successfully built a5ef1c015e7e
Successfully tagged registry.cn-hangzhou.aliyuncs.com/fc-example/test:nginx
SECURITY WARNING: You are building a Docker image from Windows against a non-Windows Do
cker host. All files and directories added to build context will have '-rwxr-xr-x' perm
issions. It is recommended to double check and reset permissions for sensitive files an
d directories.
Build image(registry.cn-hangzhou.aliyuncs.com/fc-example/test:nginx) successfully
[2021-12-15 07:54:33] [INFO] [FC-BUILD] - Build artifact successfully.
Tips for next step
======================
* Invoke Event Function: s local invoke
* Invoke Http Function: s local start
* Deploy Resources: s deploy
End of method: build
[2021-12-15 07:54:33] [INFO] [S-CLI] - End the pre-action
[2021-12-15 07:54:34] [INFO] [FC-DEPLOY] - Using region: cn-hangzhou
[2021-12-15 07:54:34] [INFO] [FC-DEPLOY] - Using access alias: default
[2021-12-15 07:54:34] [INFO] [FC-DEPLOY] - Using accessKeyID: LTAI4G4cwJkK4Rza6xd9****
[2021-12-15 07:54:34] [INFO] [FC-DEPLOY] - Using accessKeySecret: eCc0GxSpzfq1DVspnqqd6
nmYNN****
[2021-12-15 07:54:34] [INFO] [FC-DEPLOY] - Checking Service hello-world-service exists
[2021-12-15 07:54:35] [INFO] [FC-DEPLOY] - Setting role: AliyunFCDefaultRole
[2021-12-15 07:54:35] [INFO] [RAM] - Checking Role AliyunFCDefaultRole exists
[2021-12-15 07:54:35] [INFO] [RAM] - Updating role: AliyunFCDefaultRole
[2021-12-15 07:54:35] [INFO] [RAM] - Checking Plicy AliyunFCDefaultRolePolicy exists
[2021-12-15 07:54:35] [INFO] [FC-DEPLOY] - Checking Function nodejs14-event-function ex
ists
[2021-12-15 07:54:36] [INFO] [FC-DEPLOY] - Using image registry: registry.cn-hangzhou.a
liyuncs.com
[2021-12-15 07:54:36] [INFO] [FC-DEPLOY] - Try to use a temporary token for docker logi
n
Login to registry: registry.cn-hangzhou.aliyuncs.com with user: cr_temp_user
Pushing docker image: registry.cn-hangzhou.aliyuncs.com/fc-example/test:nginx...
The push refers to repository [registry.cn-hangzhou.aliyuncs.com/fc-example/test]
cdf38e7753b7: Layer already exists
```

```
43128f71725b: Layer already exists
0fb36a16ab83: Layer already exists
dd966b9fd474: Layer already exists
a1915d7a1111: Layer already exists
c4491b3ee709: Layer already exists
9fb10d900487: Layer already exists
3e207b409db3: Layer already exists
nginx: digest: sha256:02b69157def85ceb72f32cb1c5845d00e1d8df19caf6eaf720a9bc77bb57db76
size: 1991
√ Make service hello-world-service success.
√ Make function hello-world-service/nodejs14-event-function success.
[2021-12-15 07:54:39] [INFO] [FC-DEPLOY] - Checking Service hello-world-service exists
[2021-12-15 07:54:39] [INFO] [FC-DEPLOY] - Checking Function nodejs14-event-function ex
ists
There is auto config in the service: hello-world-service
Tips for next step
======================
* Display information of the deployed resource: s info
* Display metrics: s metrics
* Display logs: s logs
* Invoke remote function: s invoke
* Remove Service: s remove service
* Remove Function: s remove function
* Remove Trigger: s remove trigger
* Remove CustomDomain: s remove domain
helloworld:
  region:   cn-hangzhou
  service:
    name: hello-world-service
  function:
    name:       nodejs14-event-function
    runtime:    custom-container
    handler:    not-used
    memorySize: 256
    timeout:    60
```

5. 执行以下命令，调试函数。

```
s invoke -e "{\"key\":\"val\"}"
```

输出示例：

```
[2021-12-15 08:00:17] [INFO] [S-CLI] - Start ...
========= FC invoke Logs begin =========
FC Invoke Start RequestId: 768945c8-f92d-428e-89c2-ecd50883****
{"key":"val"}
FC Invoke End RequestId: 768945c8-f92d-428e-89c2-ecd50883****
Duration: 3.05 ms, Billed Duration: 4 ms, Memory Size: 256 MB, Max Memory Used: 10.77 M
B
========= FC invoke Logs end =========
FC Invoke Result:
OK
End of method: invoke
```

5.6. 镜像启动加速（ACR个人版）

容器镜像相比于代码包有更好的可移植性和更丰富的工具链生态，但其自带的与应用无关的数据极易使镜像臃肿，GB级镜像会导致分钟级冷启动。开启镜像加速可分两阶段共提速约90%，将分钟级的镜像拉取缩短至秒级。本文介绍使用ACR个人版实现镜像启动加速的使用原理等。

使用原理

对于运行环境为Custom Container且使用阿里云容器镜像服务ACR个人版的函数，函数计算默认为其开启镜像启动加速功能。每次创建或更新该类函数时，函数计算会扮演服务RAM角色，使用临时的AccessKey拉取镜像，并且转存至函数计算系统的加速镜像缓存服务中。转存结束后，函数中的自定义容器镜像启动速度将会得到显著提升。

注意事项

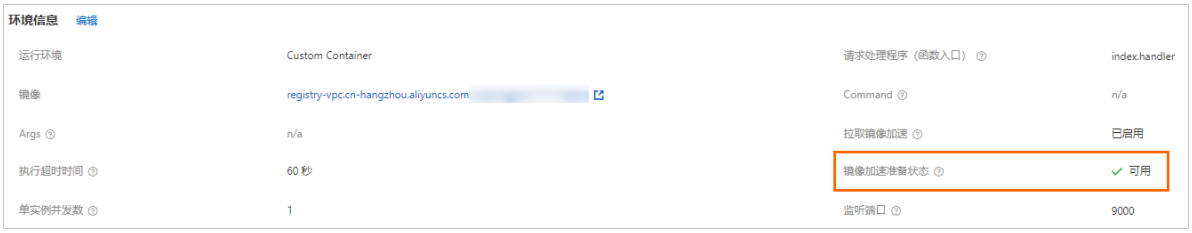
- 默认支持镜像加速的地域有华北2（北京）、华北3（张家口）、华东1（杭州）、华东2（上海）、华南1（深圳）、中国香港、新加坡（新加坡）、美国（硅谷）和美国（弗吉尼亚）。
- 如果您同意使用镜像加速，则代表您同意并授权函数计算拉取您仓库内的镜像并转存至函数计算服务内部的加速缓存服务中。镜像数据将被加密，并且具备网络隔离和身份鉴权，以保证足够的数据安全性。请确保将镜像转存在函数计算系统内部存储服务的操作符合您所在机构的安全规范和指导。
- 创建或更新使用ACR个人版镜像的函数后，由于转存镜像也有一定的时间消耗，在加速镜像可用之前，该函数在函数计算控制台禁止使用。加速镜像通常会在创建或更新函数后5分钟内完成。

查看镜像加速准备状态

您可以通过以下任一方式查看镜像加速准备状态，来判断当前被加速的镜像是否可用。

- 在函数计算控制台的函数详情页面的**函数配置**页签，找到**环境信息**区域，即可查看**镜像加速准备状态**，取值如下：
  - **准备中**：对应 `Preparing` 。
  - **可用**：对应 `Ready` 。
  - **失败**：对应 `Failed` 。

  示例如下：



- 调用GetFunction接口，根据返回的accelerationInfo的status参数判断当前镜像加速准备状态。主要的状态如下：
  - `Preparing` ：加速正在准备中，这时调用会使用没有加速效果的原始镜像。
  - `Ready` ：加速完成，后续函数调用会有加速效果。
  - `Failed` ：加速失败。

版本控制最佳实践

如果您使用了ACR个人版，更新函数镜像会触发新的加速镜像转存。在加速镜像可用前的一段时间内，函数调用会拉取原始镜像从而失去函数加速运行效果。对此，您可参照以下流程通过管理函数和管理版本的方式来体系化地发布函数：

1. 更新函数，此时会连带更新服务的LATEST版本。
2. 等待镜像加速准备状态从**准备中**状态至**可用**后，发布一个新的版本。
3. 将别名切换到新的服务版本。

查看加速效果

冷启动镜像加速的效果分为两个阶段。

示例项目puppeteer-pdf中提供了一个使用Node.js Express结合Puppeteer将网页转换成PDF的示例。

加速前，端到端耗时66.51s。加速后，第一阶段仅需15.2s，速度提升77.1%；第二阶段加速后缩短至4.3s，冷启动速度再次提升71.7%，总共提升93.5%。示例代码如下：

```
time curl -H "x-fc-invocation-target: 2016-08-15/proxy/CustomContainerDemo/puppeteer-pdf-no
-accl" https://$ACCOUNT_ID.$REGION.fc.aliyuncs.com/generate-pdf\?url\=http://example.com -o
/tmp/fc-demo-puppeteer-pdf-no-accl.pdf
# Time spent: 0.06s user 0.09s system 0% cpu 1:06.51 total time
# 加速镜像第一次冷启动
curl -H "x-fc-invocation-target: 2016-08-15/proxy/CustomContainerDemo/puppeteer-pdf-accl" h
ttps://$ACCOUNT_ID.$REGION.fc.aliyuncs.com/generate-pdf\?url\=http://example.com -o /tmp/fc
-demo-puppeteer-pdf-accl.pdf
# Time spent: 0.05s user 0.06s system 0% cpu 15.200 total time
# 相隔一段时间后的冷启动
curl -H "x-fc-invocation-target: 2016-08-15/proxy/CustomContainerDemo/puppeteer-pdf-accl" h
ttps://$ACCOUNT_ID.$REGION.fc.aliyuncs.com/generate-pdf\?url\=http://example.com -o /tmp/fc
-demo-puppeteer-pdf-accl.pdf
# Time spent: 0.05s user 0.06s system 0% cpu 4.300 total time
```

> **说明** 在测试过程中可能存在误差，请以实际情况为准。

5.7. 镜像启动加速（ACR企业版）

容器镜像相比于代码包有更好的可移植性和更丰富的工具链生态，但其自带的与应用无关的数据极易使镜像臃肿，GB级镜像会导致分钟级冷启动。开启镜像加速可分两阶段共提速约90%，将分钟级的镜像拉取缩短至秒级。本文介绍使用ACR企业版实现镜像启动加速的使用原理和配置方法等。

使用优势（相比ACR个人版）

除了拥有ACR个人版的所有镜像加速功能外，ACR企业版镜像还具备以下优势：

- 单独的网络访问，可配置独立的VPC安全规则管理仓库实例的网络访问。更多信息，请参见配置专有网络的访问控制。
- 独享的带宽，拉取镜像更加敏捷。
- 镜像仓库自带的镜像转换功能，避免函数计算加速镜像转换完成前偶发的未加速的冷启动。

使用原理

对于运行环境为Custom Container且使用容器镜像服务ACR企业版镜像的函数，函数计算会在执行函数请求时扮演服务RAM角色，使用临时的AccessKey拉取加速镜像。以此，函数中的自定义容器镜像拉取及容器启动速度会得到显著提升。

注意事项

- 默认支持镜像加速的地域有华北2（北京）、华北3（张家口）、华东1（杭州）、华东2（上海）、华南

1（深圳）、中国香港、新加坡（新加坡）、美国（硅谷）和美国（弗吉尼亚）。

- 函数计算在解析ACR企业版镜像域名时，使用镜像仓库实例配置的专有网络默认解析或云解析PrivateZone自动解析的访问IP地址。
- 在创建或更新使用ACR企业版（基础版）镜像的函数后，请等待加速镜像生成，加速镜像通常会在创建或更新函数后5分钟内完成。函数计算将在函数调用时拉取您的仓库中的加速镜像。
- 在创建或更新使用ACR企业版（标准版或高级版）镜像的函数后，函数计算将在函数调用时拉取您的仓库中的加速镜像。如果您的仓库内不存在加速镜像，则需要您开启镜像仓库的镜像加速功能生成加速镜像。具体操作步骤，请参见配置方法。

配置方法

1. 开启镜像加速。
   ACR企业版的标准版和高级版实例均提供了自带的加速镜像转换功能。镜像加速可在创建或更新仓库时开启，具体步骤，请参见转换加速镜像。
2. 选择加速镜像。
   在创建或更新函数时，请优先使用 `_accelerated` 结尾的加速镜像并开启镜像加速拉取。当函数配置完成后，可直接调用到加速镜像，确保函数调用自始至终具备加速效果。您可以在创建或更新函数时，通过以下方式选择加速镜像：
   - 通过函数计算控制台选择加速镜像。详细操作，请参见使用控制台创建函数。
   - 通过Serverless Devs配置选择加速镜像。详细信息，请参见YAML规范。

     ```
     customContainerConfig:
       image: registry-vpc.<regionId>.aliyuncs.com/fc-demo/python-flask:[镜像版本号_accelerated]
     ```

   - 通过SDK配置。

其他操作

您可以通过登录函数计算控制台或调用GetFunction接口查看镜像加速的准备状态。具体步骤，请参见查看镜像加速准备状态。

5.8. 函数实例生命周期回调

本文介绍Custom Container Runtime实现函数实例生命周期回调的方法。

回调方法

当您实现并配置函数实例生命周期回调后，函数计算将在相关实例生命周期事件发生时调用对应的回调程序。函数实例生命周期涉及Initializer、PreFreeze和PreStop三种回调。更多信息，请参见函数实例生命周期回调。

Path	输入请求	期望的响应

Path	输入请求	期望的响应

（可选）POST `/initialize`	请求体：无。
	请求头：Common Request Headers。具体信息，请参见函数计算公共请求头。
		响应体：函数 `Initializer` 的返回值。
		StatusCode
		• 200：成功状态。
		• 404：失败状态。
		Python语言中关于 `initialize` 的示例代码如下：
		`@app.route('/initialize', methods=['POST'])`
		`def init_invoke():`
		`  rid = request.headers.get(x-fc-request-id)`
		`  print("FC Initialize Start RequestId: " + rid)`
		`  # do your things`
		`  print("FC Initialize End RequestId: " + rid)`
		`  return "OK"`

（可选）GET `/pre-freeze`	• 请求体：无。 • 请求头：Common Request Headers。具体信息，请参见函数计算公共请求头。	响应体：函数PreFreeze的返回值。 StatusCode • 200：成功状态。 • 404：失败状态。
（可选）GET `/pre-stop`	• 请求体：无。 • 请求头：Common Request Headers。具体信息，请参见函数计算公共请求头。	响应体：函数PreStop的返回值。 StatusCode • 200：成功状态。 • 404：失败状态。

如果您想在Custom Runtime中使用Initializer回调方法，您只需在您的HTTP Server中实现Path为 `/initialize` 和Method为 `POST` 的对应逻辑即可。示例代码，请参见上表中关于 `initialize` 的示例代码。

> **注意** 如果创建的函数不设置Initializer，就无需实现 `/initialize` 。此时，即使HTTP Server实现了 `/initialize` ，代码中的 `/initialize` 逻辑也无法被调用执行。

PreFreeze和PreStop回调方法的使用，同Initializer回调方法。

6.编程模型扩展

6.1. 功能简介

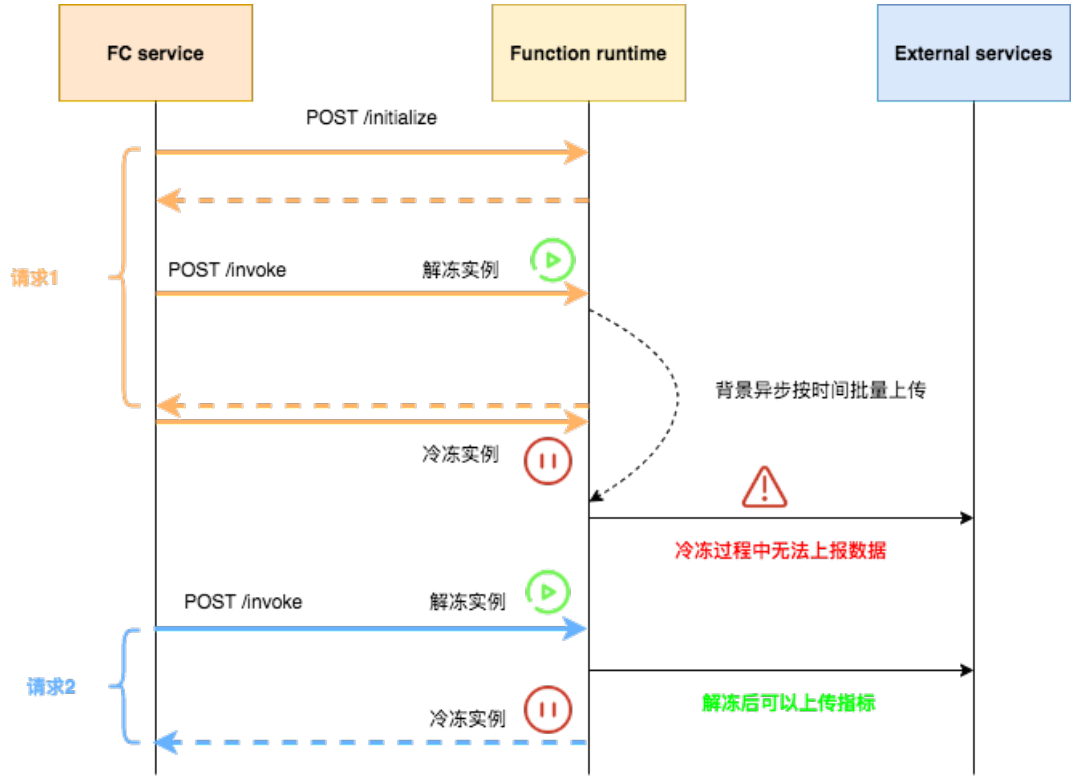
本文介绍函数计算基于传统常驻应用所拓展的运行时扩展功能，帮助您消除闲置成本。

常驻应用与FaaS运行环境

传统常驻的虚拟机或者托管容器类服务通常从实例启动到结束作为计费区间，即使该时间段没有业务请求仍然需要付费。函数计算提供1 ms计费粒度，且只在有实际请求的区间内计费，在请求以外的时间段内实例会被冷冻。这样基本消除了完全事件驱动的计费模型的闲置成本，然而冷冻机制也会打破一些传统架构下long-running进程的假设，加大应用迁移的难度。例如常用的开源分布式链路追踪Tracing Analysis库或者是第三方APM解决方案由于函数计算特殊的运行环境无法正确上报数据。

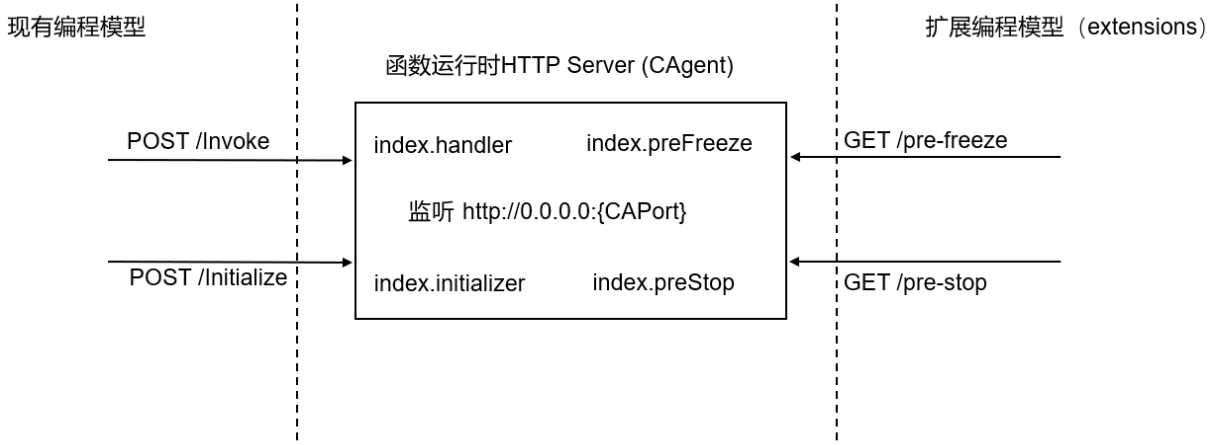
下列痛点都阻碍传统应用平滑迁移至Serverless架构：

- 异步背景指标数据延迟或丢失：如果在请求期间没有发送成功，则可能被延迟至下一次请求，或者数据点被丢弃。
- 同步发送指标增加延迟：如果在每个请求结束后都调用类似Flush接口，不仅增加了每个请求的延迟，对于后端服务也产生了不必要的压力。
- 函数优雅下线：实例关闭时应用有清理连接，关闭进程，上报状态等需求。在函数计算中实例下线时机开发者无法掌握，也缺少Webhook通知函数实例下线事件。

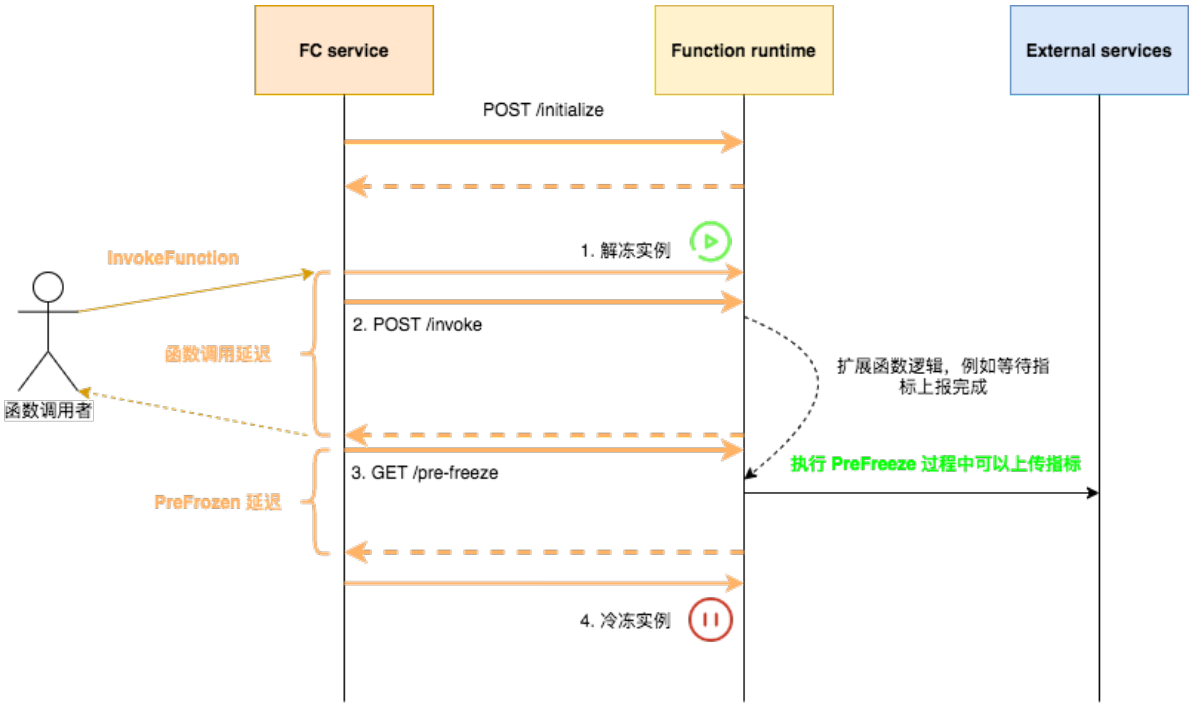


编程模型扩展

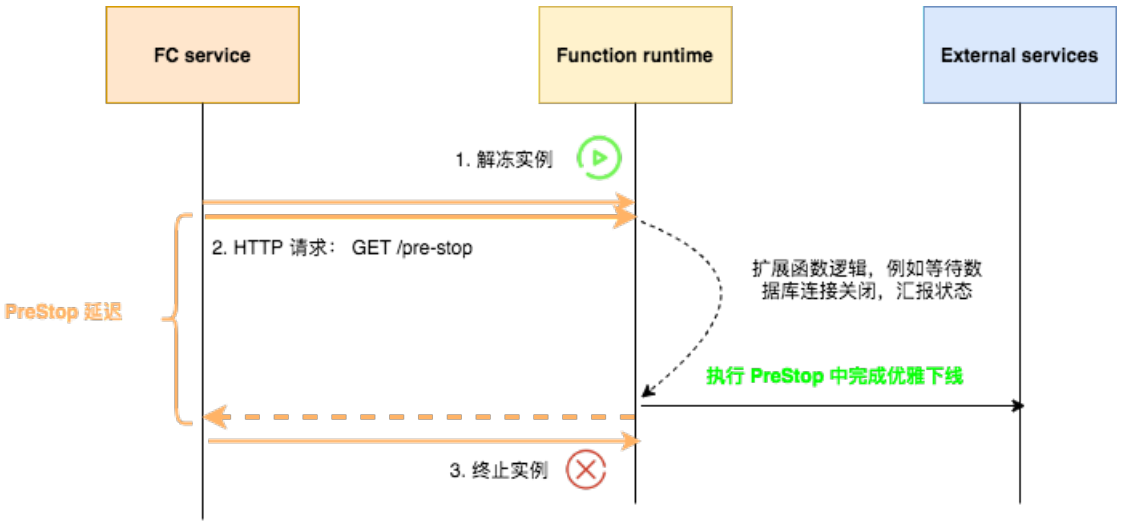
函数计算针对上述痛点发布了运行时扩展（runtime extensions）功能。该功能在现有的HTTP服务编程模型上扩展，在已有的HTTP服务器的模型中增加了PreFreeze和PreStop webhooks。扩展开发者实现HTTP handler，监听函数实例生命周期事件。



- PreFreeze：在每次函数计算服务决定冷冻当前函数实例前，函数计算服务会调用HTTP GET /pre-freeze路径，扩展开发者负责实现相应逻辑以确保完成实例冷冻前的必要操作，例如等待指标发送成功等。函数调用InvokeFunction的时间不包PreFreeze hook的执行时间。

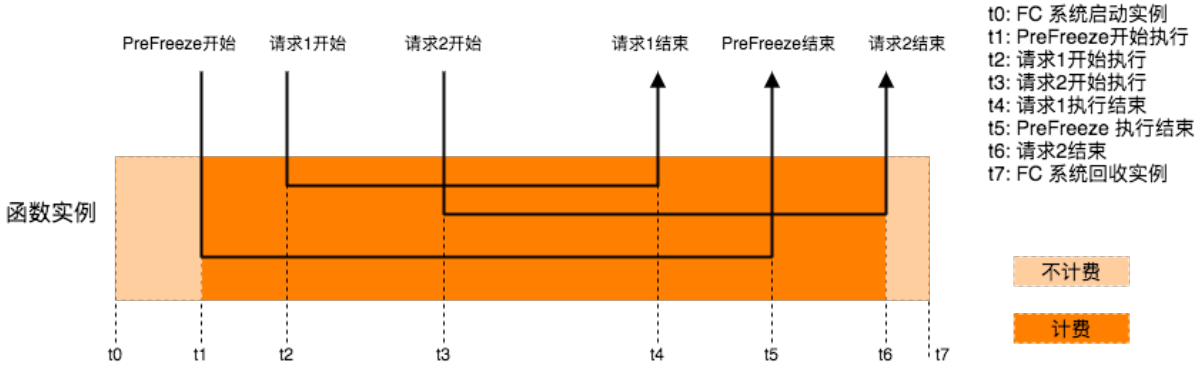


- PreStop：在每次函数计算决定停止当前函数实例前，函数计算服务会调用HTTP GET /pre-stop路径，扩展开发者负责实现相应逻辑以确保完成实例释放前的必要操作，如关闭数据库链接，以及上报、更新状态等。



计费说明

唤起PreFreeze或PreStop中产生的同InvokeFunction计费方式一致。扩展HTTP hooks请求数不做计费。扩展在单实例多并发的场景下依然适用，假设同时有多个Invoke请求在相同实例执行，在所有请求都结束后即将冷冻实例之前会调用一次PreFreeze hook。以下图为例，函数规格为1 GB，从t1 PreFreeze开始到t6请求2结束的时间段（假设为1秒），则实例执行时间为t6-t1，消耗1s * 1 GB=1 CU。



7.代码开发FAQ

7.1. 咨询类FAQ

7.1.1. 我可以使用什么语言编写函数？

函数计算支持的语言列表，请参见代码开发概述。

7.1.2. 函数计算如何保证代码的安全？

函数计算对代码进行加密并将其存储在对象存储OSS中。每当使用代码时，函数计算都执行完整性检查。代码执行与他自己的文件系统和网络命名空间相隔离。

7.1.3. 使用函数Context参数中的AccessKey ID等信息访问其他云资源时，收到The Access Key ID does not exist的报错怎么办？

函数Context参数中提供了访问云资源的临时密钥，包含AccessKey ID、AccessKey Secret及Security Token，如果遗漏了Security Token，会收到The Access Key ID does not exist的报错。

以下是在Python函数中访问OSS代码的示例。

```
import json
import oss2
def my_handler(event, context):
    evt = json.loads(event)
    creds = context.credentials
    # 身份验证时，请不要遗漏了security_token!
    # Do not miss the "security_token" for the authentication!
    auth = oss2.StsAuth(creds.access_key_id, creds.access_key_secret, creds.security_token)
    bucket = oss2.Bucket(auth, evt['endpoint'], evt['bucket'])
    bucket.put_object(evt['objectName'], evt['message'])
    return 'success'
```

7.1.4. 函数计算只支持Node.js，用C++写的程序怎么运行？

函数计算会根据您的需求不断拓展支持的语言种类。当前函数计算支持的编程语言详情请参见代码开发概述，如果您的程序是用函数计算还未支持的语言实现的，您可以采用以下做法：

- 用函数计算支持的语言改写。Node.js、Python等语言包含了非常丰富的类库，开发效率很高。
- 使用Custom Runtime自定义开发环境。
- 将C/C++等程序编译为可执行文件，通过Fork等系统调用的方式运行可执行文件。
- 将C/C++模块编译为Shared Library，在Python等语言中通过Binding的方式调用。

下表总结了以上方法的优缺点。

构建方式	实施难度	性能损失	适用场景
重写逻辑	取决于逻辑复杂度	取决于语言以及具体应用场景	适用于逻辑不太复杂的场景。
Custom Runtime	低	低	适用于所有场景。
调用可执行文件	低	高	适合对延时不敏感的场景，例如异步后台文件处理等。
调用Shared Library	高	低	适合性能要求很高的场景。

如果以上方法仍不能解决您的问题，请联系我们，为您定制其他方法。

7.1.5. 函数计算的运行环境中所依赖的包如何自动安装？

函数计算要求您上传的代码包中包含了所有的依赖。不同的语言包管理机制不同，例如在Node.js中，您可以使用npm将依赖的包安装到代码目录中并打包上传。函数计算支持通过控制台和Serverless Devs安装第三方依赖。具体信息，请参见为函数安装第三方依赖。

7.2. PHP运行环境FAQ

7.2.1. 函数计算PHP运行环境支持HTTP触发器吗？

支持。更多信息，请参见HTTP触发器概述。

7.2.2. PHP运行环境支持使用第三方扩展吗？

支持。详细信息，请参见PHP运行环境和PHP运行环境动态加载卸载内置扩展。

7.2.3. PHP运行环境如何加载卸载内置扩展？

具体操作，请参见PHP运行环境动态加载卸载内置扩展。

7.2.4. PHP运行环境内置的Tablestore PHP SDK使用有问题怎么办？

由于内置Protobuf扩展和Tablestore依赖的PHP实现的Protobuf有冲突，导致PHP运行环境内置的Tablestore PHP SDK使用有问题。解决方法，请参见PHP运行环境动态加载卸载内置扩展。同时，PHP运行环境动态加载卸载内置扩展，建议您通过环境变量裁剪不必要的扩展，优化运行环境的启动速度。

7.2.5. PHP运行环境中Notice或Warning导致某些第三方库（aliyun-openapi-php-sdk）不能使用怎么办？

PHP运行环境对异常处理严格，针对一些第三方库的引入，可能会出现因Warning或Notice而无法使用的情况。函数计算允许您通过自定义 `set_error_handler` 来覆盖运行环境中的默认处理。详细信息，请参见函数计算PHP Runtime-Exception处理。

7.2.6. 使用PHP运行环境HTTP触发器时，出现Cannot modify header information - headers already sent by (output started at ...怎么办？

具体原因和解决办法，请参见解决方案。

7.2.7. 使用PHP运行环境HTTP触发器时，想更改Session目录怎么办？

关于如何更改Session目录，请参见更改方式。

7.2.8. PHP运行环境开发Web时，怎么支持Rewrite？

详细信息，请参见PHP运行环境中HTTP Web的Rewrite浅解和方案。

7.2.9. 当我需要使用MongoDB等其他PHP非内置扩展怎么办？

您可以在PHP Runtime镜像内执行相关命令安装MongoDB扩展。详细操作，请参见安装PHP非内置扩展。

7.2.10. PHP文件中Require_once的使用示例

- *a.php*使用示例如下：

```
<?php
$appcode = 123456;
class Foo {
   public $name = 'FooClass';
   function sayhi() {
        print 'Foo say hello!';
    }
}
$foo = new Foo;
```

- *b.php*使用示例如下：

```
<?php
require_once __DIR__ . '/a.php';
function handler($event, $context) {
    echo $GLOBALS['appcode'] . PHP_EOL;
    $GLOBALS['foo']->sayhi();
    $foo2 = new Foo;
    $foo2->sayhi();
    return $GLOBALS['appcode'];
}
```

7.3. Custom Runtime FAQ

7.3.1. Custom Runtime的监听端口一定要和HTTP Server的监听端口一致吗？

是的。Custom Runtime的监听端口（CAPort）默认是9000。如果Custom Runtime使用默认的监听端口，那么您实现的Custom Runtime的HTTP Server监听的端口也必须是9000。 如果Custom Runtime使用的监听端口是8080，那么您实现的Custom Runtime的HTTP Server监听的端口也必须是8080。

Custom Runtime启动的HTTP Server一定要监听 `0.0.0.0:CAPort` 或 `*:CAPort` 端口。如果您使用 `127.0.0.1:CAPort` 端口，会导致请求超时，出现以下错误：

```
{
  "ErrorCode":"FunctionNotStarted",
  "ErrorMessage":"The CA's http server cannot be started:ContainerStartDuration:25000000000
. Ping CA failed due to: dial tcp 21.0.5.7:9000: getsockopt: connection refused Logs : 2019
-11-29T09:53:30.859837462Z Listening on port 9000"
}
```

7.3.2. Custom Runtime的bootstrap文件是Shell脚本时，出现CAExited怎么办？

当Custom Runtime的*bootstrap*文件是Shell脚本，且出现以下错误时，Custom Runtime的*bootstrap*文件一定要添加 `#!/bin/bash` 。

```
{
  "ErrorCode":"CAExited",
    "ErrorMessage":"The CA process either cannot be started or exited:ContainerStartDuratio
n:25037266905. CA process cannot be started or exited already: rpc error: code = 106 desc =
ContainerStartDuration:25000000000. Ping CA failed due to: dial tcp 21.0.7.2:9000: i/o time
out Logs : 2019-11-29T07:27:50.759658265Z panic: standard_init_linux.go:178: exec user proc
ess caused \"exec format error\"
}
```

如果是二进制可执行文件，例如Go，C++直接编译出来的目标二进制文件，则不需要添加 `#!/bin/bash` 。

7.3.3. Custom Runtime的bootstrap文件没有可执行权限，出现以下错误怎么办？

Custom Runtime的*bootstrap*文件，一定要具备777或755权限，否则会出现以下错误：

```
{
  "ErrorCode":"CAFilePermission",
    "ErrorMessage":"The CA process cannot be started due to bootstrap file don't have execu
te permissions
}
```

您可以在打包文件前执行chmod 777 bootstrap或chmod 755 bootstrap命令获取权限。

7.3.4. Custom Runtime启动的服务中调用第三方服务时，出现FunctionNotStarted错误怎么办？

您需要查看该第三方服务是否有网络限制，如果存在网络限制或网络超时的情况，则会导致没有完成启动HTTP Server的逻辑，出现如下异常。

```
{
    "ErrorCode":"FunctionNotStarted",
    "ErrorMessage":"The CA's http server cannot be started:ContainerStartDuration:250000000. Ping CA failed due to: dial tcp 21.0.3.1:9000: getsockopt: connection refused"}
```

7.3.5. 当我实现的HTTP Server在120s内无法成功启动怎么办？

您可以优化HTTP Server的启动速度，例如调大函数执行内存。

> **说明** 当您优化了HTTP Server的启动速度后，仍无法成功启动HTTP Server，请提交工单。

7.3.6. 当我的操作系统是Windows时，对bootstrap文件的格式有什么要求吗？

如果您使用的是Windows操作系统，您需要确保*bootstrap*的文件格式为UNIX格式。

7.3.7. 遇到Process exited unexpectedly before completing request怎么办？

当遇到以上错误提示时，您可以从以下三种情况排查：

- 当Custom Runtime出现这种情况时，您可以查看Connection及Server端的设置。
  Connection需要设置为Keep-Alive，Server端请求超时时间需设置在15分钟及以上。示例如下：

  ```
  var server = app.listen(PORT, HOST);
  server.timeout = 0; // never timeout
  server.keepAliveTimeout = 0; // keepalive, never timeout
  ```

- 调用函数的Client端主动Cancel导致的，例如函数的执行时长是10s，但是Client端，例如SDK调用函数自己设置的Timeout是5s。建议调用函数的Client端设置的Timeout稍微大于函数设置的Timeout。
- 函数本身逻辑的问题，导致执行环境退出。

7.3.8. 当我使用浏览器或cURL方式访问函数时出现404怎么办？

问题现象

我创建了一个Custom Runtime的HTTP函数，其中服务名为*CustomDemo*、函数名为*func-http*，并且设置了匿名的HTTP触发器，实现Custom Runtime的HTTP Server的路由代码示例如下：

```
@app.route('/test', methods = ['POST','GET'])
def test():
```

当我使用cURL工具或浏览器等方式访问HTTP函数的URL时，遇到 `404` 报错。

- 使用cURL工具访问HTTP函数。

```
curl  -v  https://164901546557****.cn-hangzhou.fc.aliyuncs.com/2016-08-15/proxy/CustomDemo/func-http/test
```

- 使用浏览器访问HTTP函数。

```
https://164901546557****.cn-hangzhou.fc.aliyuncs.com/2016-08-15/proxy/CustomDemo/func-http/test
```

> 说明 HTTP函数的URL格式为：`https://<account_id>.<region_id>.fc.aliyuncs.com/<version>/proxy/<serviceName>/<functionName>/<path>` 。

解决方案

操作步骤

1. 您可以选择以下任意方式解决该问题：
   - 使用函数计算为新建HTTP触发器分配的子域名重新访问。具体信息，请参见步骤三：测试函数。
     子域名格式如下：

     ```
     https://<subdomain>.<region_id>.fcapp.run/[action?queries]
     ```

     本文的访问示例如下：

     ```
     https://funcname-svcname-khljsjksld.cn-shanghai.fcapp.run/action?hello=world
     ```

   - 在访问的命令中增加名为 `x-fc-invocation-target` 的Header。访问格式如下：

     ```
     curl -v "x-fc-invocation-target: 2016-08-15/proxy/$ServiceName/$functionName" https://<account_id>.<region_id>.fc.aliyuncs.com/$path
     ```

     本文的访问示例如下：

     ```
     curl -v -H "x-fc-invocation-target: 2016-08-15/proxy/CustomDemo/func-http" https://164901546557****.cn-hangzhou.fc.aliyuncs.com/test
     ```

   - 为您的函数绑定自定义域名，绑定成功后再执行以下命令重新访问即可。关于绑定域名的操作步骤，请参见配置自定义域名。
     假设域名是 `example.com` ，访问格式如下：

     ```
     curl -v  https://example.com/$path
     ```

     本文的访问示例如下：

     ```
     curl -v  https://example.com/test
     ```

> **注意** 您需要将路径 `/$path` 设置为绑定的自定义域名中设置的与函数名称和服务名称对应的路径。更多信息，请参见路由匹配规则。

- 修改您的函数代码，并且成功部署函数后，重新使用默认的URL访问即可。函数代码修改示例如下：

```
@app.route('/2016-08-15/proxy/CustomDemo/func-http/test', methods = ['POST','GET'])
def test():
```

本文的访问示例如下：

```
curl  -v  https://164901546557****.cn-hangzhou.fc.aliyuncs.com/2016-08-15/proxy/CustomDemo/func-http/test
```

7.3.9. 遇502报错且报错信息为Process exited unexpectedly before completing request怎么办？

可能原因

HTTP Server连接主动关闭，主动关闭的可能原因如下：

- 连接未设置Keep-Alive。
- 空闲一段时间后，主动关闭。
- 读写超时或错误时关闭。

解决方案

操作步骤

1. 当前的函数计算使用Keep-Alive连续访问Custom Runtime内的HTTP Server，对于幂等请求例如GET、HEAD、OPTIONS或TRACE等，在连接失败时例如 `EOF` 、 `connection reset by peer` 等，会主动重试。但对于非幂等请求例如POST、PATCH等，在连接失败时会直接返回502报错。为避免502报错，Custom Runtime的服务端需要设置以下两类参数：
   - 将连接模式Connection设置为Keep-Alive。
   - 关闭IDLE超时时间或将IDLE超时时间设置为15分钟以上。

   对于不同的HTTP Server框架以上两种参数的配置方式可能会不一样，例如GoFrame框架，不仅需要将 `SetIdletimeout` 设置为0，还需要设置 `ReadTimeout` 和 `python uvicorn` 参数， `python uvicorn` 还需要在命令行中设置 `--timeout-keep-alive` 等参数。建议您自行验证，对于Keep-Alive模式的HTTP客户端在进行稀疏性调用时，是否会触发HTTP server主动关闭连接。

可能原因

函数本身原因导致进程退出，可能原因如下：

- 主动调用 `exit` 等接口退出。
- 运行过程中出现的 `exception` 未被捕获。

解决方案

操作步骤

1. 您可以按照以下方式检查您的代码：
   - 检查您的代码中是否存在主动退出的逻辑。

- 在运行环境进程顶层增加异常捕获或覆盖，避免发生 `exception` 时进程退出。

7.4. Custom Container FAQ

7.4.1. Custom Container的监听端口一定要和HTTP Server的监听端口一致吗？

是的。Custom Container的监听端口（CAPort）默认是9000。如果Custom Container使用默认的监听端口，那么您实现的Custom Container 的HTTP Server监听的端口也必须是9000。 如果Custom Container使用的监听端口是8080，那么您实现的Custom Container的HTTP Server监听的端口也必须是8080。

Custom Container启动的HTTP Server一定要监听0.0.0.0:CAPort或*:CAPort端口。如果您使用127.0.0.1:CAPort端口，会导致请求超时，出现以下错误：

```
{
  "ErrorCode":"FunctionNotStarted",
  "ErrorMessage":"The CA's http server cannot be started:ContainerStartDuration:25000000000
. Ping CA failed due to: dial tcp 21.0.5.7:9000: getsockopt: connection refused Logs : 2019
-11-29T09:53:30.859837462Z Listening on port 9000"
}
```

7.4.2. Custom Container Runtime启动的服务中调用第三方服务时，出现FunctionNotStarted错误怎么办？

您需要查看该第三方服务是否有网络限制，如果存在网络限制或网络超时的情况，则会导致没有完成启动HTTP Server的逻辑，出现如下异常。

```
{
    "ErrorCode":"FunctionNotStarted",
    "ErrorMessage":"The CA's http server cannot be started:ContainerStartDuration:250000000
00. Ping CA failed due to: dial tcp 21.0.3.1:9000: getsockopt: connection refused"}
```

7.4.3. 当我实现的HTTP Server在120s内无法成功启动怎么办？

您可以优化HTTP Server的启动速度，例如调大函数执行内存。

> **说明** 当您优化了HTTP Server的启动速度后，仍无法成功启动HTTP Server，请提交工单。

7.4.4. 遇502报错且报错信息为Process exited unexpectedly before completing request怎么办？

可能原因

HTTP Server连接主动关闭，主动关闭的可能原因如下：

- 连接未设置Keep-Alive。

- 空闲一段时间后，主动关闭。
- 读写超时或错误时关闭。

解决方案

操作步骤

1. 当前的函数计算使用Keep-Alive连续访问Custom Container内的HTTP Server，对于幂等请求例如GET、HEAD、OPTIONS或TRACE等，在连接失败时例如 `EOF` 和 `connection reset by peer` 等，会主动重试。但对于非幂等请求例如POST、PATCH等，在连接失败时会直接返回502报错。为避免502报错，Custom Container的服务端需要设置以下两类参数：
   - 将连接模式Connection设置为Keep-Alive。
   - 关闭IDLE超时时间或将IDLE超时时间设置为15分钟以上。

   对于不同的HTTP Server框架以上两种参数的配置方式可能会不一样，例如GoFrame框架，不仅需要将 `SetIdletimeout` 设置为0，还需要设置 `ReadTimeout` 和 `python uvicorn` 参数， `python uvicorn` 还需要在命令行中设置 `--timeout-keep-alive` 等参数。建议您自行验证，对于Keep-Alive模式的HTTP客户端在进行稀疏性调用时，是否会触发HTTP server主动关闭连接。

可能原因

函数本身原因导致进程退出，可能原因如下：

- 主动调用 `exit` 等接口退出。
- 运行过程中出现的 `exception` 未被捕获。

解决方案

操作步骤

1. 您可以按照以下方式检查您的代码：
   - 检查您的代码中是否存在主动退出的逻辑。
   - 在运行环境进程顶层增加异常捕获或覆盖，避免发生 `exception` 时进程退出。

7.4.5. 当我使用浏览器或cURL方式访问函数时出现404怎么办？

问题现象

我创建了一个Custom Container Runtime的HTTP函数，其中服务名为*CustomDemo*、函数名为*func-http*，并且设置了匿名的HTTP触发器，实现Custom Container Runtime的HTTP Server的路由代码示例如下：

```
@app.route('/test', methods = ['POST','GET'])
def test():
```

当我使用cURL工具或浏览器等方式访问HTTP函数的URL时，遇到 `404` 报错。

- 使用cURL工具访问HTTP函数。

```
curl  -v  https://164901546557****.cn-hangzhou.fc.aliyuncs.com/2016-08-15/proxy/CustomDemo/func-http/test
```

- 使用浏览器访问HTTP函数。

```
https://164901546557****.cn-hangzhou.fc.aliyuncs.com/2016-08-15/proxy/CustomDemo/func-http/test
```

> **说明** HTTP函数的URL格式为：`https://<account_id>.<region_id>.fc.aliyuncs.com/<version>/proxy/<serviceName>/<functionName>/<path>`。

解决方案

操作步骤

1. 您可以选择以下任意方式解决该问题：
   - 使用函数计算为新建HTTP触发器分配的子域名重新访问。具体信息，请参见步骤三：测试函数。
     子域名格式如下：

     ```
     https://<subdomain>.<region_id>.fcapp.run/[action?queries]
     ```

     本文的访问示例如下：

     ```
     https://funcname-svcname-khljsjksld.cn-shanghai.fcapp.run/action?hello=world
     ```

   - 在访问的命令中增加名为`x-fc-invocation-target`的Header。访问格式如下：

     ```
     curl -v "x-fc-invocation-target: 2016-08-15/proxy/$ServiceName/$functionName" https://<account_id>.<region_id>.fc.aliyuncs.com/$path
     ```

     本文的访问示例如下：

     ```
     curl -v -H "x-fc-invocation-target: 2016-08-15/proxy/CustomDemo/func-http" https://164901546557****.cn-hangzhou.fc.aliyuncs.com/test
     ```

   - 为您的函数绑定自定义域名，绑定成功后再执行以下命令重新访问即可。关于绑定域名的操作步骤，请参见配置自定义域名。
     假设域名是`example.com`，访问格式如下：

     ```
     curl -v  https://example.com/$path
     ```

     本文的访问示例如下：

     ```
     curl -v  https://example.com/test
     ```

     > **注意** 您需要将路径`/$path`设置为绑定的自定义域名中设置的与函数名称和服务名称对应的路径。更多信息，请参见路由匹配规则。

   - 修改您的函数代码，并且成功部署函数后，重新使用默认的URL访问即可。函数代码修改示例如下：

     ```
     @app.route('/2016-08-15/proxy/CustomDemo/func-http/test', methods = ['POST','GET'])
     def test():
     ```

     本文的访问示例如下：

     ```
     curl  -v  https://164901546557****.cn-hangzhou.fc.aliyuncs.com/2016-08-15/proxy/CustomDemo/func-http/test
     ```

7.4.6. 如未使用阿里云的公网容器镜像，我需要给函数计算的服务角色授予什么权限？

您需要给服务角色添加 `AliyunContainerRegistryReadOnlyAccess` 或 `AliyunContainerRegistryFullAccess` 权限。关于授权的操作步骤，请参见授予函数计算访问其他云服务的权限。