ALIBABA CLOUD

Alibaba Cloud

Serverless 工作流

流程定义语言

文档版本：20210118

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.基本介绍 ---- 05
2.输入和输出 ---- 07
3.任务步骤 ---- 15
4.等待步骤 ---- 20
5.传递步骤 ---- 22
6.选择步骤 ---- 23
7.并行步骤 ---- 27
8.并行循环步骤 ---- 29
9.成功步骤 ---- 32
10.失败步骤 ---- 34

1.基本介绍

本文介绍了流程定义语言的基本知识和相关使用示例。

基本知识

流程定义语言FDL（Flow Definition Language）用来描述和定义业务逻辑，在执行流程时，Serverless工作流服务会根据流程定义依次执行相关步骤。在FDL中，一个流程（Flow）通常包含若干步骤（Step），这些步骤可以是简单的原子步骤，如任务（ `task` ）、成功（ `succeed` ）、失败（ `fail` ）、等待（ `wait` ）和传递（ `pass` ）等；也可以是复杂的控制步骤，如选择（ `choice` ）、并行（ `parallel` ）和并行循环（ `foreach` ）等。这些步骤可以组合使用以构建复杂的业务逻辑，例如一个并行步骤的分支可以是一个顺序步骤。同时，步骤执行可能出现错误，但是FDL提供了错误重试（ `retry` ）和捕获（ `catch` ）能力。

FDL 的步骤类似于编程语言中的函数（Function），组合类似于函数调用。步骤之间通过输入（Input）和输出（Output）传递数据，每个步骤由本地（Local）变量保存数据。如果一个步骤包含另一个步骤，则称外层步骤为父步骤，被包含步骤为子步骤。

在定义流程时，您可以：

- 通过传递（ `pass` ）步骤来做占位符规划流程基本结构。
- 通过任务（ `task` ）步骤去调用函数计算服务的函数。
- 如果需要等待一段时间可以通过等待（ `wait` ）步骤暂停流程的执行。
- 通过选择（ `choice` ）步骤来定义不同执行路径。
- 通过成功（ `succeed` ）步骤或者失败（ `fail` ）步骤来提前终止流程步骤。
- 通过并行（ `parallel` ）步骤来并行执行多个分支。
- 通过并行循环（ `foreach` ）步骤来并行处理数组数据。

流程包含以下属性：

- version（必需）：流程版本，仅支持v1。
- type（必需）：flow表示是流程类型。
- steps（必需）：定义了流程的多个串行步骤。一个步骤执行完成后，如果成功，则会执行下一个步骤。如果需要提前结束，可以使用步骤的结束（ `end` ）属性，或者使用成功和失败步骤。
- inputMappings（可选）：输入映射。输入映射中引用的 `$input` 是 `StartExecution` API请求的 `Input` 参数。
- outputMappings（可选）：输出映射。输入映射中的引用的 `$local` 是一个JSON对象，记录了串行的每个步骤的执行结果。

  > 说明 如果未指定输出映射，将 `$local` 作为最终流程输出。

- timeoutSeconds（可选）：流程超时时间。如果流程执行超过指定的超时时间，则流程执行超时。

示例

- 以下的示例流程包含一个任务步骤，这个步骤调用一个函数计算的函数。

```
version: v1
type: flow
steps:
 - type: task
   name: hello
   resourceArn: acs:fc:{region}:{accountID}:services/fnf_test/functions/hello
```

- 以下示例流程包含两个子步骤（ `step1` 和 `step4` ），其中第一个步骤（ `step1` ）又包含了两个子步骤（ `step2` 和 `step3` ）。

```
version: v1
type: flow
steps:
 - type: parallel
   name: step1
   branches:
     - steps:
       - type: pass
         name: step2
     - steps:
       - type: pass
         name: step3
 - type: pass
   name: step4
```

更多信息

流程定义语言的特性，请参见以下文档：

- 输入和输出
- 任务步骤
- 等待步骤
- 传递步骤
- 选择步骤
- 并行步骤
- 并行循环步骤
- 成功步骤
- 失败步骤

2.输入和输出

本文介绍了输入和输出的基本知识，您可以参见本文示例进行操作。

流程和步骤

通常流程的多个步骤之间需要传递数据。和函数式编程语言类似，FDL的步骤类似于函数，它接受输入（Input），并返回输出（Output），输出会保存在父步骤（调用者）的本地（Local）变量里。其中，输入和输出的类型必须是JSON对象格式，本地变量的类型因步骤而异。例如任务步骤把调用函数计算函数的返回结果作为本地变量，并行步骤把它所有分支的输出（数组）作为本地变量。步骤的输入、输出和本地变量总大小不能超过32 KiB，否则会导致流程执行失败。

如果一个步骤包含另一个步骤，则称外层步骤为父步骤，被包含步骤为子步骤。最外层步骤的父步骤是流程。如果两个步骤的父步骤相同，则这两个步骤是同级步骤。

流程和步骤都有输入、输出和本地变量。他们的转换关系如下：

- 步骤输入映射（`inputMappings`）将父步骤的输入和本地变量映射为子步骤的输入。
- 步骤输出映射（`outputMappings`）将当前步骤的输入和本地变量映射为当前步骤的输出。
- 流程输入映射（`inputMappings`）将用户执行流程时的输入映射为流程的输入。
- 流程输出映射（`outputMappings`）将流程输入和本地变量映射为流程的输出。

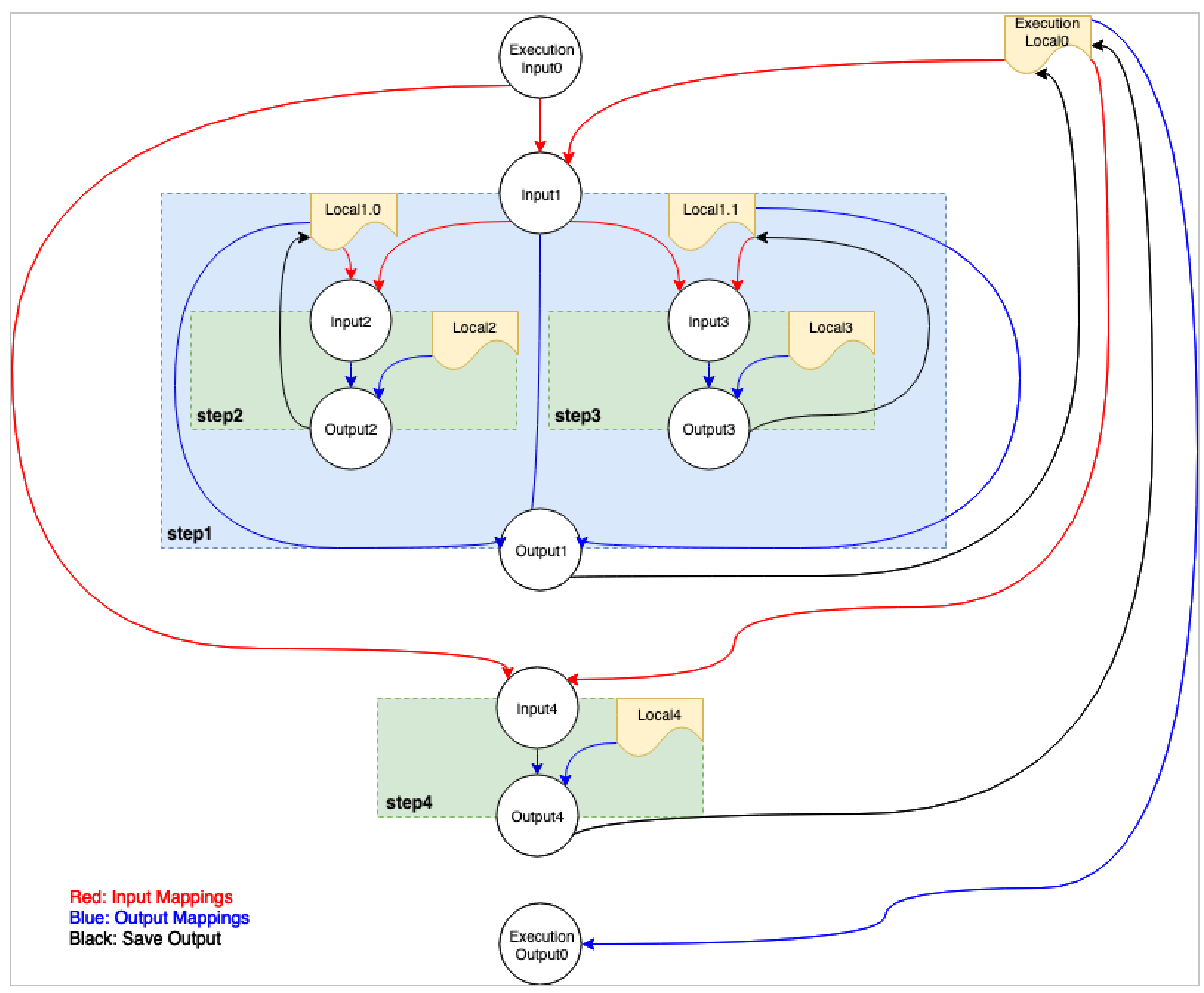
一个父步骤的本地变量保存了其所有子步骤输出的合并，如果输出中有同名键值，则后执行步骤会覆盖前执行步骤的结果。在大多数情况下，您不需要指定输入和输出映射，而使用默认的映射行为。他们的转换关系如下：

- 在不指定输入映射情况下，子步骤的输入是其父步骤输入和父步骤本地变量的合并（如果本地变量和输入有同名键值，则本地变量会覆盖输入）。
- 在不指定输出映射情况下，除并行步骤和并行循环步骤外的其它步骤都会将本地变量作为输出。

如果您想更好地控制输入和输出，则需要了解详细的映射规则。

下面示例流程的输入输出映射如图所示。其中step1是step2和step3的父步骤。step1和step4是最外层步骤。

```
version: v1
type: flow
steps:
  - type: parallel
    name: step1
    branches:
      - steps:
        - type: pass
          name: step2
      - steps:
        - type: pass
          name: step3
  - type: pass
    name: step4
```



为了便于理解，这些映射可以用下面的代码描述。

```
func flow(input0 Input) (output0 Output) {
 local0 := {}
 input1 := buildInput(step1InputMappings, input0, local0)
 output1 := step1(input1)
 save(local0, output1)
 input4 := buildInput(step4InputMappings, input0, local0)
 output4 := step4(input4)
 save(local0, output4)
 return buildOutput(flowOutputMappings, input0, local0)
}
func step1(input1 Input) (output1 Output) {
 local10 := {}
 input2 := buildInput(step2InputMappings, input1, local10)
 output2 := step2(input2)
 save(local10, output2)
 local11 := {}
 input3 := buildInput(step3InputMappings, input1, local11)
 output3 := step3(input3)
 save(local11, output3)
 return buildOutput(step1OutputMappings, [local10, local11])
}
func step2(input2 Input) (output2 Output) {
}
func step3(input3 Input) (output3 Output) {
}
func step4(input4 Input) (output4 Output) {
}
```

在这个例子中，流程包含了两个子步骤： `step1` 和 `step4` 。 `step1` 是一个并行步骤，它包含 `step2` 和 `step3` 两个子步骤。

1. 系统在开始执行流程时，根据流程的输入映射将 `StartExecution` 输入转换为流程输入（ `input0` ）。
2. 在 `flow` 开始时，其 `local0` 为空。
3. 系统根据 `step1` 的输入映射（ `step1InputMappings` ）来为它准备输入 `input1` ，映射的源来自 `flow` 的输入 `input0` 和本地变量 `local0` 。
4. 调用 `step1` ，传入输入 `input1` ， `step1` 返回输出 `output1` 。
   - 在 `step1` 开始时，其 `local0` 为空 。因为 `step1` 是一个并行步骤，所以每个分支都对应一个本地变量，从而避免并发访问的问题。
   - 系统根据 `step2` 的输入映射（ `step2InputMappings` ）来为它准备输入 `input2` ，映射的源来自 `ste`

p1 的输入 `input1` 和本地变量 `local10` 。

- 调用 `step2` ，传入输入 `input2` ， `step2` 返回输出 `output2` 。
- 系统将 `step2` 的输出保存到 `step1` 的本地变量 `local10` 。
- 同样的，系统调用 `step3` 并将结果保存在 `step1` 的本地变量 `local11` 。

5. 系统将 `step1` 的输出保存到 `flow` 的本地变量 `local0` 。
6. 类似的，系统根据 `step4` 的输入映射来为它准备输入 `input4` ，映射的源来自 `flow` 的输入 `input0` 和本地变量 `local0` 。

> **说明** 这时的本地变量 `local0` 可能已经包含了 `step1` 的输出，从而实现了 `step1` 和 `step4` 的数据传递。

7. 调用 `step4` ，传入输入 `input4` ， `step4` 返回输出 `output4` 。
8. 系统将 `step4` 的输出保存到 `flow` 的本地变量 `local0` 。
9. 最后，系统根据流程的输出映射将 `local0` 转换为流程输出。

类型定义

输入映射和输出映射都是由 `target` 和 `source` 组成的数组类型，其中 `source` 定义了参数来源，根据不同的映射取值不同，例如 `$input.key` 表示参数来自输入 `input` 的 `$.key` 所对应的值， `target` 定义了目标参数名称。当 `source` 以 `$` 开始表示该值通过JSON Path方式指定（可以通过这个工具调试JSON Path），系统会根据该路径解析为具体值，否则认为该值是常量类型。

- 来源（Source）

  来源支持使用常量作为 `source` 值，可以使用数字（ `number` ）、字符串（ `string` ）、布尔（ `boolean` ）、数组（ `array` ）、对象（ `object` ）或者 `null` 类型。

  下面示例映射的source使用了不同类型的常量，其产生的输出如下所示。

```
outputMappings:
 - target: int_key
   source: 1
 - target: bool_key
   source: true
 - target: string_key
   source: abc
 - target: float_key
   source: 1.234
 - target: null_key
   source: null
 - target: array1
   source: [1, 2, 3]
 - target: array2
   source:
     - 1
     - 2
     - 3
 - target: object1
   source: {a: b}
 - target: object2
   source:
     a:
```

```
{
 "array1": [1, 2, 3],
 "array2": [1, 2, 3],
 "bool_key": true,
 "float_key": 1.234,
 "int_key": 1,
 "null_key": null,
 "object1": {
   "a": "b"
 },
 "object2": {
   "a": "b"
 },
 "string_key": "abc"
}
```

- 目标

目标只能是一个字符串类型常量。

输入映射

输入映射将父步骤的输入（ `$input` ）、父步骤的本地变量（ `$local` ）、或者常量转换为子步骤的输入。如果没有指定输入映射，父步骤的输入和父步骤的本地变量合并后会被当做子步骤的输入。如果父步骤的输入和父步骤的本地变量中有相同名称，则新的输入中采用本地变量中的名或值。

```
inputMappings:
 - target: key1
   source: $input.key1
 - target: key2
   source: $local.key2
 - target: key3
   source: literal
```

输入 `$input`	本地变量 `$local`	输入映射	子步骤输入
`{ "key1":"value1" }`	`{ "key2":"value2" }`	`inputMappings: - target: key1 source: $input.key1 - target: key2 source: $local.key2 - target: key3 source: literal`	`{ "key1":"value1" "key2":"value2" "key3":"literal" }`
`{ "key1":"value1" }`	`{ "key2":"value2" }`	无	`{ "key1":"value1" "key2":"value2" }`
`{ "key1":"value1" }`	`{ "key1":"value2" }`	无	`{ "key1":"value2" }`

输出映射

输出映射将当前步骤的输入（ `$input` ）、本地变量（ `$local` ）、或者常量转换为本地步骤的输出。如果没有指定输出映射，对于选择步骤和循环步骤，它们的本地变量会被当做其输出，任务步骤会将具体任务执行结果作为输出。由于并行（Parallel）和并行循环（ForEach）步骤的本地变量是数组类型，所以您需要定义输出映射将数组转换结果成JSON对象格式，默认不会输出它们的本地变量。具体介绍可参见步骤描述。

```
outputMappings:
 - target: key1
   source: $input.key1
 - target: key2
   source: $local.key2
 - target: key3
   source: literal
```

输入 `$input`	本地变量 `$local`	输出映射	步骤输出
`{ "key1":"value1" }`	`{ "key2":"value2" }`	`outputMappings: - target: key1 source: $input.key1 - target: key2 source: $local.key2 - target: key3 source: literal`	`{ "key1":"value1" "key2":"value2" "key3":"literal" }`
`{ "key1":"value1" }`	`[ { ” key2” :” value2.1” }, { ” key2” :” value2.2” } ]`	`outputMappings: - target: key1 source: $input.key1 - target: key2 source: $local[*].key2 - target: key3 source: literal`	`{ ” key1” :” value1” , ” key2” :[ “value2.1” ,” value2.2” ], ” key3” :” literal” }`

输入 `$input`	本地变量 `$local`	输出映射	步骤输出

{
"key1":"value1"
}
	{
	"key2":"value2"
	}
		无	{
			"key2":"value2"
			}

输出如何保存到父步骤本地变量

子步骤的输出（ `$output` ）会被并入父步骤的本地变量。如果二者有重复名称，则输出中的名或值会覆盖本地变量相应名或值。

输出 `$output`	父步骤本地变量 `$local`	更改后父步骤本地变量

{
"key1":"value1"
}
	{
	"key2":"value2"
	}
		{
		” key1” :” value1”
		},
		{
		” key2” :” value2”
		}

{
"key1":"value11"
}
	{
	” key1” :” value1”
	},
	{
	” key2” :” value2”
	}
		{
		” key1” :” value11”
		},
		{
		” key2” :” value2”
		}

3.任务步骤

本文介绍了任务步骤和其相关使用示例。

参数说明

任务（Task）步骤定义了函数计算服务的函数调用信息，执行任务步骤会调用相应的函数。

任务步骤包含以下属性：

- `type` ：task表示该步骤是任务步骤。
- `name` ：步骤名称。
- `resourceArn` ：资源标识，当前支持函数、消息队列、Serverless工作流流程，例如 `acs:fc:cn-shanghai:18807708****3420:services/fnf_test/functions/hello` 。更多信息，请参见服务集成。
- （可选）pattern：集成服务的执行模式，不同的集成服务支持不同的执行模式。默认值为 `requestResponse` ，枚举值如下：
  - `requestResponse` ：同步等待任务执行结束后继续该步骤。
  - `sync` ：异步提交任务后，等待任务执行结束，获得任务返回结果后继续该步骤。
  - `waitForCallback` ：异步提交任务后（例如完成函数调用后）当前步骤暂停，直到收到关于该任务的回调或该任务超时。
- （可选） `timeoutSeconds` ：任务超时时间。如果任务执行超过指定的超时时间，则任务步骤执行超时。
- （可选） `end` ：当前步骤结束后是否继续执行其后定义的步骤。
- （可选） `inputMappings` ：输入映射。任务步骤的输入会被用作函数调用的event，更多信息，请参见InvokeFunction。
- （可选） `outputMappings` ：输出映射。其中的 `$local` 是函数调用的返回结果，其格式必须是JSON类型。

> **说明** 如果未指定输出映射，本步骤默认将 `$local` 作为输出。

- （可选） `errorMappings` ：错误映射。本映射仅在步骤执行出错，且配置 `catch` 时生效。可以通过 `$local.cause` 及 `$local.error` 将错误信息映射至输出，并传递给下个步骤。

> **说明** `$local.error` 及 `$local.code` 为系统预留字段，mapping中的 `source` 字段必须取这两个值，详细信息，请参见本文示例。另外， `errorMappings` 为可选字段，如果未指定则发生错误后无法在下一步骤获取相关错误信息。

- `retry` ：定义了一组重试策略，其中每一个重试策略包含以下属性：
  - `errors` ：定义了一个或者多个错误，更多信息，请参见错误定义。
  - `intervalSeconds` ：重试的初始间隔时间，最大值是86400秒，默认值是1秒。
  - `maxIntervalSeconds` ：重试的最长间隔时间，最大值和默认值是86400秒。
  - `maxAttempts` ：最多重试次数，默认值是3次。

- multiplier ：后一次重试比前一次重试间隔时间的倍数，默认值是2。

- catch ：定义了一组捕获策略，其中每一个捕获策略包含如下属性：
  - errors ：定义了一个或者多个错误，详情请参见下面表格。
  - goto ：跳转目的步骤名称。

  说明 该目的步骤只能是和当前任务步骤并列的一个步骤。

错误定义

函数执行状态	FC响应HTTP Code	FC响应	Serverless工作流步骤失败（**用于重试和捕获**）	是否需要重试
没有执行	429	ResourceExhausted	FC.ResourceExhausted	是
没有执行	4xx，非429	ServiceNotFound，FunctionNotFound，InvalidArgument等	FC.ServiceNotFound，FC.FunctionNotFound，FC.InvalidArgument等	否
不确定	500	InternalServerError	FC.InternalServerError	是
没有执行	503	ResourceThrottled	FC.ResourceThrottled	是
函数执行成功，返回错误。	200	用户自定义错误，包含errorType。	errorType	根据业务决定
函数执行异常，返回错误。	200	无errorType	FC.Unknown	是
函数执行成功，返回非JSON对象。	200	无errorType	FC.InvalidOutput	否

示例

- 简单任务步骤

  下面的示例流程包含一个任务步骤。

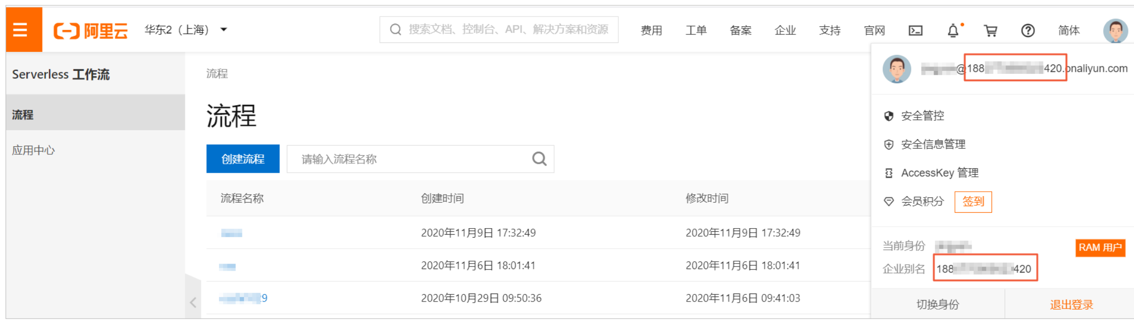
  - 如果流程的输入是 {"name": "function flow"} ，输出是 {"hello": "function flow"} 。
  - 如果流程没有指定输入或者输入不包含 name key，则任务步骤执行失败，从而导致整个流程执行失败。

- 流程定义

```
version: v1
type: flow
steps:
 - type: task
  name: hello
  resourceArn: acs:fc:{region}:{accountID}:services/fnf_test/functions/hello
```

`resourceArn` 的参数说明如下：

- `{region}`：需替换成您选择的地域，例如`cn-shanghai`。
- `{accountID}`：需替换成您的账号ID。您可以在Serverless工作流控制台的**流程**页面，单击右上角的头像查看，如下图所示。



- 函数定义

```
import json
class MyError(Exception):
 pass
def handle(event, context):
 evt = json.loads(event)
 if "name" in evt:
  return {
   "hello": evt["name"]
  }
 else:
  raise MyError("My unhandled exception")
```

- 错误重试

下面的示例会重试`MyError`。如果流程没有指定输入或者输入不包含`name` key，Serverless工作流服务会根据重试策略多次重试失败任务调用。

- 在第一次发生错误后等待3秒，然后再次调用函数。
- 在第二次发生错误后等待6秒（`intervalSeconds x multiplier`），然后再次调用函数。
- 在第三次发生错误后等待12秒（`intervalSeconds x multiplier x multiplier`），然后再次调用函数。

- 经过三次重试后，函数仍然返回错误，重试次数超过 maxAttempts ，任务步骤失败，流程以失败结束。

```
version: v1
type: flow
steps:
 - type: task
  name: hello
  resourceArn: acs:fc:{region}:{accountID}:services/fnf_test/functions/hello
  retry:
   - errors:
    - MyError
    intervalSeconds: 3
    maxAttempts: 3
    multiplier: 2
```

- 错误捕获

下面的示例会对捕获 MyError ，然后跳转到 final 步骤。由于错误被成功捕获，最后流程以成功结束。

```
version: v1
type: flow
steps:
 - type: task
  name: hello
  resourceArn: acs:fc:{region}:{accountID}:services/fnf_test/functions/hello
  catch:
   - errors:
    - MyError
    goto: final
 - type: pass
  name: pass1
 - type: pass
  name: final
```

- 带错误映射的错误捕获

下面的示例会对 MyError 捕获，然后跳转到 final 步骤。由于进行了错误映射，可以在 final 步骤中获取错误信息并进行处理。最后流程以成功结束。当然，您也可以将该步骤的输入、常量等在 errorMappings 中进行输出映射。

```
version: v1
type: flow
steps:
 - type: task
  name: hello
  resourceArn: acs:fc:{region}:{accountID}:services/fnf_test/functions/hello
  errorMappings:
   - target: errMsg
    source: $local.cause #该值为系统预留，在本步骤发生错误时可直接使用
   - target: errCode
    source: $local.error #该值为系统预留，在本步骤发生错误时可直接使用
  catch:
   - errors:
    - MyError
    goto: final
 - type: pass
  name: pass1
 - type: pass
  name: final
```

映射后在进入final步骤的 `event` 中将看到 `EventDetail` 为如下内容：

```
"EventDetail": "{\"input\":{},\"local\":{\"errorCode\":\"MyError\",\"errorMsg\":\"some message\"}}",
```

4.等待步骤

本文介绍了等待步骤和其相关使用示例。

参数说明

等待（Wait）步骤可以将执行流程暂停一段时间，然后再继续执行。您可以选择一个相对时间，也可以以时间戳方式指定一个绝对结束时间。

等待步骤包含以下参数：

- （必需） `type` ：wait表示该步骤是等待步骤。
- （必需） `name` ：步骤名称。
- （可选） `duration` ：等待的相对时间以秒为单位，可以是常量也可以是输入中的某个参数。例如 `10` 表示等待10秒钟， `$.sleep` 表示等待时间从输入的 `sleep` Key获取。必须指定 `duration` 或者 `timestamp` 中的一个，且不能同时指定二者。
- （可选） `timestamp` ：等待的绝对时间，格式为RFC3339，可以是常量也可以是输入中的某个参数。例如 `2019-05-02T15:04:05Z` 表示等待到UTC时间的2019年5月2日15点04分05秒。如果该时间早于当前时间，则等待步骤直接结束。

  > 说明 最长等待时间限制为2天。

- （可选） `end` ：当前步骤结束后是否继续执行其后定义的步骤。
- （可选） `inputMappings` ：输入映射。
- （可选） `outputMappings` ：输出映射。本步骤不产生数据，其 `$local` 为空。

示例

- 等待20秒

```
version: v1
type: flow
steps:
  - type: wait
    name: wait20s
    duration: 20
```

- 等待时间由输入决定

```
version: v1
type: flow
steps:
  - type: wait
    name: custom_wait
    duration: $.wait
```

- 等待到绝对时间

```
version: v1
type: flow
steps:
  - type: wait
    name: wait20s
    timestamp: 2019-05-02T15:04:05Z
```

- 等待绝对时间由输入决定

```
version: v1
type: flow
steps:
  - type: wait
    name: custom_wait
    timestamp: $.wait_timestamp
```

5.传递步骤

本文介绍了传递步骤和其相关使用示例。

介绍

传递（pass）步骤可以用来输出常量或者将输入转换成期望的输出。例如在开始定义流程时，如果您还没有创建任务步骤对应的函数计算 Function，可以先使用控制步骤和传递步骤规划并调试流程逻辑，然后再逐步将传递步骤替换为任务步骤。

传递步骤包含以下属性：

- （必需）type：pass表示该步骤是传递步骤。
- （必需）name：步骤名称。
- （可选）end：当前步骤结束后是否继续执行其后定义的步骤。
- （可选）inputMappings：输入映射。
- （可选）outputMappings：输出映射。本步骤不产生数据，其 `$local` 为空。

示例

以下示例定义了一个传递步骤，执行结果输出一个大写字母数组。

```
version: v1
type: flow
steps:
  - type: pass
    name: toUpperCase
    outputMappings:
      - target: names
        source: ["A", "B", "C"]
```

6.选择步骤

本文介绍了选择步骤的基本知识和其相关使用示例，以及条件表达式的相关内容。

参数说明

选择（Choice）步骤让流程根据条件执行不同的步骤，类似于编程语言中的 `switch-case` ，它包含多个条件选项（Choice）和一个默认选项（Default），每个条件选项带有一个条件表达式（Condition），若干步骤（Steps）和跳转指令（Goto）。默认选项只带有若干步骤和跳转指令。当流程执行到选择步骤，系统会按照选项定义的顺序依次评估其条件表达式是否返回 `True` 。

- 如果返回 `True` 则执行选项对应的步骤（如果定义了步骤），然后执行跳转（如果定义跳转指令）。
- 如果没有任何选项返回 `True` ，则执行默认选项对应的步骤和跳转。
- 如果没有定义默认选项，则结束选择步骤。

选择步骤包含以下属性：

- （必需）type：choice表示该步骤是选择步骤。
- （必需）name：步骤名称。
- （必需）choices：多个选项，数组类型，每个元素对应一个选项。
  - （必需）condition：定义条件表达式。条件表达式可以通过JSON Path（ `$.key` ）引用步骤输入数据。
  - （可选）steps：定义了选项所对应的多个串行步骤。
  - （可选）goto：指定跳转目的步骤名称，该目的步骤只能是和选择步骤并列的一个步骤。
- （必需）default：默认选项。
  - （可选）steps：定义了选项所对应的多个串行步骤。
  - （可选）goto：指定跳转目的步骤名称，该目的步骤只能是和当前选择步骤并列的一个步骤。
- （可选）end：当前步骤结束后是否继续执行其后定义的步骤。
- （可选）inputMappings：输入映射。
- （可选）outputMappings：输出映射。本步骤 `$local` 为实际执行选择分支的执行结果。

> **说明** 如果未指定输出映射，本步骤默认将 `$local` 作为输出。

示例

下面的流程定义了一个选择步骤。

- 如果输入中的 `status` 的值是 `ready` ，则会执行第一个条件选项的步骤 `pass1` ，然后执行 `pass3` 和 `final` 。
- 如果输入中的 `status` 的值是 `failed` ，则会执行第二个条件选项的跳转，结束选择步骤执行 `final` 。
- 如果输入中的 `status` 的值不是 `ready` 和 `failed` ，则会执行默认选项逻辑，即 `pass2` 和 `final` 。

```yaml
version: v1
type: flow
steps:
 - type: choice
  name: mychoice
  choices:
   - condition: $.status == "ready"
    # choice with steps
    steps:
     - type: pass
      name: pass1
   - condition: $.status == "failed"
    # choice with goto
    goto: final
  default:
   # choice with both steps and goto
   steps:
    - type: pass
     name: pass2
   goto: final
 - type: pass
  name: pass3
 - type: pass
  name: final
```

条件表达式

条件表达式由以下操作和变量组成：

- 比较操作：`>` `>=` `<` `<=` `==` `!=` ，适用于字符串和数字类型。
- 逻辑操作：`||` `&&` 。
- 字符常量：以双引号（ “” ）或者反引号（``）开始和结束，例如"foobar"或者`foobar`。
- 数字常量：`1` `12.5` 。
- 布尔常量：`true` `false` 。
- 前缀：`!` `-` 。
- 包含：`in` ，用来判断数组是否包含某个值，或者对象是否包含某个键值。

以下示例对于下面的步骤输入，针对不同的条件表达式，显示了相应的执行结果。

```
{
 "a": 1,
 "b": {
  "b1": true,
  "b2": "ready"
 },
 "c": [1, 2, 3],
 "d": 1,
 "e": 1,
 "f": {
  "f1": false,
  "f2": "inprogress"
 }
}
```

条件表达式	执行结果
`$.a==1`	true
`$.a==2`	false
`$.a>0`	true
`0<$.a`	true
`$.a>=1`	true
`$.a!=2`	true
`$.b.b1`	true
`$.b.b1==true`	true
`$.b.b1==false`	false
`$.b.b2=="ready"`	true
`` $.b.b2==`ready` ``	true
`$.b.b2=="inprogress"`	false
`$.a==1 && $.b.b1`	true

条件表达式	执行结果
`$.a==1 || $.b.b1`	true
`$.a==2 && $.b.b1`	false
`$.a==2 || $.b.b1`	true
`$.c[0]==1`	true
`$.c[0]==$.a`	true

7.并行步骤

本文介绍了并行循环步骤和其相关使用示例。

介绍

并行（Parallel）步骤用来并行执行多个步骤。它定义了多个分支（Branches），每个分支包含一系列串行步骤。

并行步骤的每个分支都对应一个本地变量。执行并行步骤会并发执行所有分支包含的串行步骤。这些串行步骤会改变其分支对应的本地变量。当所有分支执行结束后，可以通过输出映射将分支本地变量数组转换为并行步骤的输出。

> **说明** 并行步骤最大分支数限制为100。

并行步骤包含以下属性：

- （必需）type：parallel表示该步骤是并行步骤。
- （必需）name：步骤名称。
- （必需）branches：多个分支，数组类型，每个元素对应一个分支。

  （必需）steps：定义了分支所对应的多个串行步骤。
- （可选）end：当前步骤结束后是否继续执行其后定义的步骤。
- （可选）inputMappings：输入映射。
- （可选）outputMappings：输出映射。本步骤的 `$local` 是数组类型，其中的每个元素是一个JSON对象，记录了每个分支步骤执行的结果。

  > **说明** 如果未指定输出映射，本步骤默认输出为空。

示例

下面的示例流程定义了一个并行步骤，这个并行步骤包含两个分支，每个分支又包含了一个传递步骤。

```yaml
version: v1
type: flow
steps:
 - type: parallel
  name: myparallel
  branches:
   - steps:
    - type: pass
     name: pass1
     outputMappings:
      - target: result
       source: pass1
   - steps:
    - type: pass
     name: pass2
     outputMappings:
      - target: result
       source: pass2
  outputMappings:
   - target: result
    source: $local[*].result
```

- `pass1` 的输出如下。

```json
{
 "result": "pass1"
}
```

- `pass2` 的输出如下。

```json
{
 "result": "pass2"
}
```

- `myparallel` 的输出如下。

```json
{
 "result": ["pass1", "pass2"]
}
```

8.并行循环步骤

本文介绍了并行循环步骤和其相关使用示例。

介绍

并行循环（Foreach）步骤遍历输入中的某个数组类型参数，对于数组中的每个元素并行执行其串行步骤。并行循环步骤类似于编程语言中的 `foreach` ，不同之处是这里的迭代是并行执行的。

并行循环步骤的每次迭代执行都对应一个本地变量。执行并行循环步骤会对输入参数里的每个元素并发执行串行步骤。这些串行步骤会改变其迭代对应的本地变量。当所有分支执行结束后，可以通过输出映射将迭代本地变量数组转换为并行步骤的输出。

> **说明** 并行循环步骤最大并发数限制为100。

并行步骤包含以下属性：

- （必需）type：foreach表示该步骤是并行循环步骤。
- （必需）name：步骤名称。
- （必需）iterationMapping：迭代映射。
  - （必需）collection：定义输入中的哪个参数作为循环的集合。
  - （必需）item：定义当前元素以什么名称并入迭代输入。
  - （可选）index：定义当前位置以什么名称并入迭代输入。
- （必需）steps：定义串行步骤。
- （可选）end：当前步骤结束后是否继续执行其后定义的步骤。
- （可选）inputMappings：输入映射。
- （可选）outputMappings：输出映射。本步骤的 `$local` 是数组类型，其中的每个元素是一个JSON对象，记录了每次迭代的结果。

  > **说明** 如果未指定输出映射，本步骤默认输出为空。

示例

下面的示例流程定义了一个并行循环步骤，这个并行步骤包含一个任务步骤。

```
version: v1
type: flow
steps:
 - type: foreach
  name: myforeach
  iterationMapping:
   collection: $.names
   item: name
  steps:
   - type: task
    name: toUpperCase
    resourceArn: acs:fc:{region}:{account}:services/fnf_test/functions/toUpperCase
  outputMappings:
   - target: names
    source: $local[*].name
```

- 流程的输入如下所示。由于 `myforeach` 步骤没有指定输入映射，其输入和流程输入一样。

```
{
 "names": ["a", "b", "c"]
}
```

- `toUpperCase` 没有定义输入映射，其输入继承父步骤输入，系统根据迭代映射（ `iterationMapping` ）在每次执行迭代时，将当期元素（依次是 `a` 、 `b` 、 `c` ）为值，以 `name` 为key并入输入。

```
{
 "name": "a",
 "names":["a","b","c"]
}
{
 "name": "b",
 "names":["a","b","c"]
}
{
 "name": "c",
 "names":["a","b","c"]
}
```

- `toUpperCase` 被执行了3次，输出依次如下所示。

```
{
 "name": "A"
}
{
 "name": "B"
}
{
 "name": "C"
}
```

- `myforeach` 的本地变量是一个数组类型，值如下所示。

```
[
 {
  "name": "A"
 },
 {
  "name": "B"
 },
 {
  "name": "C"
 }
]
```

- `myforeach` 的输出如下所示。由于流程没有定义输出和结果映射，这个输出也是最后流程执行的输出。

```
{
 "names": ["A", "B", "C"]
}
```

9.成功步骤

本文介绍了成功步骤和其相关使用示例。

介绍

成功（Succeed）步骤用来提前结束一系列串行步骤，类似于编程语言中的 `return` 。FDL `steps` 定义的步骤是串行的，通常一个步骤执行完成后会继续执行后续步骤，而成功步骤不会继续执行下一个步骤。成功步骤通常和选择步骤结合使用，在选择步骤条件满足的情况下跳转到一个成功步骤，从而不再执行其它步骤。

成功步骤包含以下属性：

- （必需）type：succeed表示该步骤是成功步骤。
- （必需）name：步骤名称。
- （可选）inputMappings：输入映射。
- （可选）outputMappings：输出映射。

示例

下面的流程定义使用成功步骤提前结束执行流程。

- 如果输入中的 `status` 的值是 `ready` ，则会执行第一个条件选项的步骤 `pass1` ，然后执行 `final` 。由于 `final` 是成功步骤，其结束后不会继续执行 `handle_failure` 步骤。
- 如果输入中的 `status` 的值是 `failed` ，则会执行第二个条件选项的跳转，结束选择步骤执行 `handle_failure` 。
- 如果输入中不存在 `status` 或者 `status` 的值不是 `ready` 和 `failed` ，则会执行默认选项逻辑，即 `pass2` 和 `handle_failure` 。

```yaml
version: v1
type: flow
steps:
  - type: choice
    name: mychoice
    choices:
      - condition: $.status == "ready"
        # choice with steps
        steps:
          - type: pass
            name: pass1
      - condition: $.status == "failed"
        # choice with goto
        goto: handle_failure
    default:
      # choice with both steps and goto
      steps:
        - type: pass
          name: pass2
      goto: handle_failure
  - type: succeed
    name: final
  - type: pass
    name: handle_failure
```

10.失败步骤

本文介绍了失败步骤和其相关使用示例。

介绍

失败（Fail）步骤用来提前结束一系列步骤，类似于编程语言中的 `raise` 、 `throw` 等操作。当流程执行完失败步骤后，定义在失败步骤之后的步骤不会被继续执行，并且导致失败步骤的父步骤失败，并一直传递，最后导致流程执行失败。

失败步骤包含以下属性：

- （必需）type：fail表示该步骤是失败步骤。
- （必需）name：步骤名称。
- （可选）error：错误类型。
- （可选）cause：错误原因。
- （可选）inputMappings：输入映射。
- （可选）outputMappings：输出映射。

示例

下面的流程定义使用失败步骤提前结束执行流程。

- 如果输入中的 `status` 的值是 `ready` ，则会执行第一个条件选项的步骤 `pass1` ，然后执行 `final` 。
- 如果输入中的 `status` 的值是 `failed` ，则会执行第二个条件选项的跳转，结束选择步骤，执行 `handle_failure` 。由于 `handle_failure` 是失败步骤，其结束后不会继续执行 `final` 步骤。
- 如果输入中不存在 `status` 或者 `status` 的值不是 `ready` 和 `failed` ，则会执行默认选项逻辑，即 `pass2` 和 `handle_failure` 。

```yaml
version: v1
type: flow
steps:
  - type: choice
    name: mychoice
    choices:
      - condition: $.status == "ready"
        # choice with steps
        steps:
          - type: pass
            name: pass1
        goto: final
      - condition: $.status == "failed"
        goto: handle_failure
    default:
      # no need to use goto
      steps:
        - type: pass
          name: pass2
  - type: fail
    name: handle_failure
    error: StatusIsNotReady
    cause: status is not ready
  - type: pass
    name: final
```