ALIBABA CLOUD

Alibaba Cloud

FunctionFlow

Flow Definition Language

Document Version: 20211230

Alibaba Cloud

Legal disclaimer

Alibaba Cloud reminds you to carefully read and fully understand the terms and conditions of this legal disclaimer before you read or use this document. If you have read or used this document, it shall be deemed as your total acceptance of this legal disclaimer.

1. You shall download and obtain this document from the Alibaba Cloud website or other Alibaba Cloud-authorized channels, and use this document for your own legal business activities only. The content of this document is considered confidential information of Alibaba Cloud. You shall strictly abide by the confidentiality obligations. No part of this document shall be disclosed or provided to any third party for use without the prior written consent of Alibaba Cloud.

2. No part of this document shall be excerpted, translated, reproduced, transmitted, or disseminated by any organization, company or individual in any form or by any means without the prior written consent of Alibaba Cloud.

3. The content of this document may be changed because of product version upgrade, adjustment, or other reasons. Alibaba Cloud reserves the right to modify the content of this document without notice and an updated version of this document will be released through Alibaba Cloud-authorized channels from time to time. You should pay attention to the version changes of this document as they occur and download and obtain the most up-to-date version of this document from Alibaba Cloud-authorized channels.

4. This document serves only as a reference guide for your use of Alibaba Cloud products and services. Alibaba Cloud provides this document based on the "status quo", "being defective", and "existing functions" of its products and services. Alibaba Cloud makes every effort to provide relevant operational guidance based on existing technologies. However, Alibaba Cloud hereby makes a clear statement that it in no way guarantees the accuracy, integrity, applicability, and reliability of the content of this document, either explicitly or implicitly. Alibaba Cloud shall not take legal responsibility for any errors or lost profits incurred by any organization, company, or individual arising from download, use, or trust in this document. Alibaba Cloud shall not, under any circumstances, take responsibility for any indirect, consequential, punitive, contingent, special, or punitive damages, including lost profits arising from the use or trust in this document (even if Alibaba Cloud has been notified of the possibility of such a loss).

5. By law, all the contents in Alibaba Cloud documents, including but not limited to pictures, architecture design, page layout, and text description, are intellectual property of Alibaba Cloud and/or its affiliates. This intellectual property includes, but is not limited to, trademark rights, patent rights, copyrights, and trade secrets. No part of this document shall be used, modified, reproduced, publicly transmitted, changed, disseminated, distributed, or published without the prior written consent of Alibaba Cloud and/or its affiliates. The names owned by Alibaba Cloud shall not be used, published, or reproduced for marketing, advertising, promotion, or other purposes without the prior written consent of Alibaba Cloud. The names owned by Alibaba Cloud include, but are not limited to, "Alibaba Cloud", "Aliyun", "HiChina", and other brands of Alibaba Cloud and/or its affiliates, which appear separately or in combination, as well as the auxiliary signs and patterns of the preceding brands, or anything similar to the company names, trade names, trademarks, product or service names, domain names, patterns, logos, marks, signs, or special descriptions that third parties identify as Alibaba Cloud and/or its affiliates.

6. Please directly contact Alibaba Cloud for any errors of this document.

Document conventions

Style	Description	Example
Danger	A danger notice indicates a situation that will cause major system changes, faults, physical injuries, and other adverse results.	**Danger:** Resetting will result in the loss of user configuration data.
Warning	A warning notice indicates a situation that may cause major system changes, faults, physical injuries, and other adverse results.	**Warning:** Restarting will cause business interruption. About 10 minutes are required to restart an instance.
Notice	A caution notice indicates warning information, supplementary instructions, and other content that the user must understand.	**Notice:** If the weight is set to 0, the server no longer receives new requests.
Note	A note indicates supplemental instructions, best practices, tips, and other content.	**Note:** You can use Ctrl + A to select all files.
>	Closing angle brackets are used to indicate a multi-level menu cascade.	Click **Settings**> **Network**> **Set network type**.
Bold	Bold formatting is used for buttons , menus, page names, and other UI elements.	Click **OK**.
`Courier font`	Courier font is used for commands	Run the `cd /d C:/window` command to enter the Windows system folder.
Italic	Italic formatting is used for parameters and variables.	`bae log list --instanceid` *Instance_ID*
[] or [a|b]	This format is used for an optional value, where only one item can be selected.	`ipconfig [-all|-t]`
{} or {a|b}	This format is used for a required value, where only one item can be selected.	`switch {active|stand}`

Table of Contents

1.Overview

This topic describes the basics of Flow Definition Language (FDL) and related examples.

Basics

FDL is used to describe and define business logic. When a flow is executed, the Serverless workflow service executes steps in sequence based on the flow definition. In FDL, a flow usually contains several steps. These steps can be simple atomic steps, such as `task`, `succeed`, `fail`, `wait`, and `pass` steps, or complex control steps, such as `choice`, `parallel`, and `foreach` steps. These steps can be combined to build complex business logic. For example, a branch of a parallel step may be a series of serial steps. Errors may occur in the execution of steps, but FDL provides the `retry` and `catch` capabilities.

Steps in FDL are similar to functions in programming languages, and a combination of steps is similar to function calls. Data is passed between steps through input and output. Local variables are used to store data of steps. If a step contains another step, the outer step is called a parent step, and the included step is called a child step.

When you define a flow, you can perform the following operations:

- Build the basic structure of the placeholder planning flow with `pass` steps.
- Call functions of the Function Compute service with `task` steps.
- Suspend the flow for a period of time with `wait` steps.
- Define different execution paths with `choice` steps.
- Terminate a flow in advance with `succeed` or `fail` steps.
- Execute multiple branches in parallel with `parallel` steps.
- Process array data in parallel with `foreach` steps.

A flow contains the following attributes:

- version: Required. The flow version. Only v1 is supported.
- type: Required. The flow type.
- steps: Required. Multiple serial steps in a flow. When a step is executed successfully, the next step starts. To stop a flow in advance, you can use the `end` attribute or execute a succeed or fail step.
- inputMappings: Optional. The input mappings. The `$input` referenced in the input mappings is the `Input` parameter in a `StartExecution` API request.
- outputMappings: Optional. The output mappings. The `$local` referenced in the output mappings is a JSON object that records the execution result of each serial step.

  > **Note** If no output mappings are specified, `$local` is used as the final flow output.

- timeoutSeconds: Optional. The timeout period of a flow. If the flow execution duration exceeds the specified timeout period, the flow times out.

Examples

- The following sample flow consists of a task step that calls a function of Function Compute:

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

- The following sample flow consists of two steps (`step1` and `step4`), in which `step1` contains two child steps (`step2` and `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
```

References

For more information about FDL features, see the following topics:

- Inputs and outputs
- Task steps
- Wait steps
- Pass steps
- Choice steps
- Parallel steps
- Foreach steps
- Succeed steps
- Fail steps

2.Inputs and outputs

This topic describes the basics of inputs and outputs.

Flows and steps

Typically, data needs to be passed between flows and steps, and between multiple steps of a flow. Steps of the Flow Definition Language (FDL) are similar to functions in functional programming languages. These steps accept inputs and produce outputs, and the outputs are stored in the local variable of the parent step (caller). The inputs and outputs must be JSON object structures, and the types of local variables vary with steps. For example, a task step uses the returned result of calling a function of Function Compute as the local variable, whereas a parallel step uses the outputs (arrays) of all branches as the local variable. The total size of inputs, outputs, and local variable in a step cannot exceed 32 KiB. Otherwise, the flow execution fails.

If a step contains another step, the outer step is called a parent step, and the included step is called a child step. The parent step of the outermost step is the flow. If the parent steps of two steps are the same, the two steps are of the same level.

Each flow and step contain inputs, outputs, and a local variable. Their mappings are listed in the following content:

- The `inputMappings` of a step maps the input and local variable of the parent step to the inputs of child steps.
- The `outputMappings` of a step maps the input and local variable of the current step to the output of the current step.
- The `inputMappings` of a flow maps the input of a flow execution to the input of the flow.
- The `outputMappings` of a flow maps the input and local variable of the flow to the output of the flow.

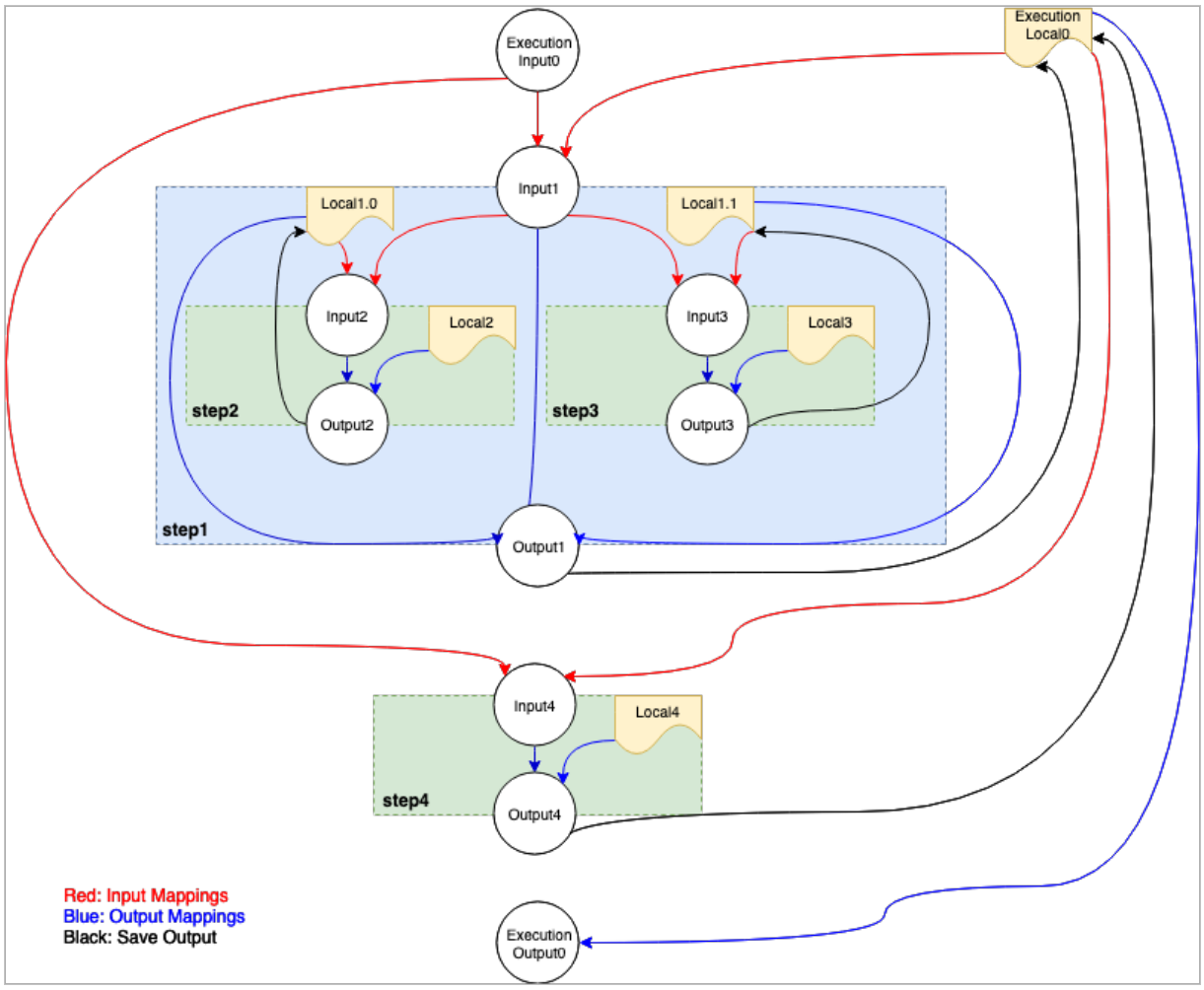
The local variable of a parent step contains a union set of the outputs of all its child steps. If the outputs contain repeated key values, the result of a later step overwrites that of an earlier step. In most cases, you can use the default mappings without specifying the input and output mappings.

- When no input mappings are specified, the input of a child step is the combination of the input and local variable of its parent step (if the local variable and the input have the same key value, the local variable will overwrite the input).
- When no output mappings are specified, the local variable is used as the output in all steps except parallel steps and foreach steps.

If you want to better control the input and output, you need to understand detailed mapping rules.

The following figure shows the input and output mappings of the example flow. In the flow, step1 is the parent step of step2 and step3, and step1 and step4 are the outermost steps.

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



The following code can be used to describe mappings can be described, so that they can be easier to understand:

```
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) {
}
```

In this example, the flow consists of two child steps: `step1` and `step4`. `step1` is a parallel step that contains `step2` and `step3`.

1. When the system starts to execute the flow, it converts the `StartExecution` input into the flow input (`input0`) based on the input mappings of the flow.
2. When the `flow` execution starts, the `local0` is empty.
3. The system prepares the `input1` input for step1 based on the input mappings (`step1InputMappings`) of `step1`. The mapping sources are the `input0` input and the `local0` local variable of the `flow`.
4. The system calls `step1` to load `input1`. `step1` returns `output1`.
   - When the system starts to execute `step1`, its `local10` is empty. `step1` is a parallel step, so each branch corresponds to a local variable, avoiding concurrent access.
   - The system prepares the `input2` input for step2 based on the input mappings of `step2` (`step2InputMappings`). The mapping sources are the `input1` input and the `local10` local variable of `step1`.
   - The system calls `step2` to load `input2`. `step2` returns `output2`.
   - The system saves the output of `step2` to the `local10` local variable of `step1`.
   - Similarly, the system calls `step3` and saves the result to the `local11` local variable of `step1`.

5. The system saves the output of `step1` to the `local0` local variable of the `flow`.
6. Similarly, the system prepares the `input4` input for step4 based on the input mappings of `step4`. The mapping sources are the `input0` input and the `local0` local variable of the `flow`.

   > **Note** At this point, the `local0` local variable may contain the output of `step1`. This achieves data transfer between `step1` and `step4`.

7. The system calls `step4` to load `input4`. `step4` returns `output4`.
8. The system saves the output of `step4` to the `local0` local variable of the `flow`.
9. Finally, the system converts `local0` into the flow output based on the output mappings of the flow.

Types

Both input and output mappings are arrays composed of `target` and `source`. The `source` defines the parameter source and is set to different values for different mappings. For example, `$input.key` indicates that the parameter source is the value of `$.key` in `input`. The `target` defines the names of target parameters. If the value of `source` starts with `$`, the value is specified in JSON path format (you can use JSONPath Online Evaluator to debug the JSON path), and the system parses the source into a specific value based on the path. Otherwise, the value is considered a constant.

- Source

  The `source` can be set to a constant, such as a value of the `number`, `string`, `boolean`, `array`, `object`, or `null` type.

  The source in the following example uses constants of different types. The information following the example shows the output.

```
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"
}
```

- Target

  The target can only be a constant of the string type.

Input mappings

Input mappings convert the input (`$input`) of a parent step, the local variable (`$local`) of a parent step, or constants into the input of child steps. If no input mappings are specified, the input and local variable of the parent step are combined and used as the input of child steps. If the input and local variable of the parent step have the same name, the new input uses the name and value in the local variable.

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

Input `$input`	Local variable `$local`	Input mapping	Child step input
`{ "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" }`	None	`{ "key1":"value1" "key2":"value2" }`
`{ "key1":"value1" }`	`{ "key1":"value2" }`	None	`{ "key1":"value2" }`

Output mappings

Output mappings convert the input (`$input`) of the current step, the local variable (`$local`) of the current step, or constants into the output of this step. If no output mappings are specified, choice steps and foreach steps use their local variables as outputs, whereas task steps uses task execution results as outputs. The local variables of parallel and foreach steps are arrays. Therefore, you must define output mappings to convert the arrays into JSON objects. By default, their local variables are not output. For more information, see the step description.

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

Input `$input`	Local variable `$local`	Output mapping	Step output
`{ "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" }`
`{ "key1":"value1" }`	`{ "key2":"value2" }`	None	`{ "key2":"value2" }`

Save outputs to local variables of the parent steps

Child step outputs (`$output`) will be saved to local variables of the parent steps. If they contain the same name, the name and value in the outputs will overwrite the corresponding name and value in the local variables.

Output `$output`	Local variable of the parent step `$local`	Local variable of the parent step after modification
`{ "key1":"value1" }`	`{ "key2":"value2" }`	`{ "key1":"value1" }, { "key2":"value2" }`
`{ "key1":"value11" }`	`{ "key1":"value1" }, { "key2":"value2" }`	`{ "key1":"value11" }, { "key2":"value2" }`

3.Task steps

This topic describes task steps and examples.

Attributes

A task step defines the function invocation information of Function Compute. When a task step is executed, the corresponding function is invoked.

A task step contains the following attributes:

- `type`: the step type. The value task indicates that the step is a task step.
- `name`: the name of the step.
- `resourceArn`: the resource identifier, which can be a function, MNS queue, or Serverless workflow flow. Example: `acs:fc:cn-shanghai:18807708****3420:services/fnf_test/functions/hello`. For more information, see Service Integration.
- (Optional)pattern: the execution mode of an integration service. Different integration services support different execution modes. Default value: `requestResponse`. Valid values:
  - `requestResponse`: The system waits until the task execution ends after a task is submitted and then continues the step.
  - `sync`: The system waits until the task execution ends after a task is asynchronously submitted, and then continues the step after the system receives the task execution result.
  - `waitForCallback`: The system suspends the step after a task is asynchronously submitted (such as invoking a function), and waits until the system receives a callback request or timeout notification of the task.
- (Optional) `timeoutSeconds`: the timeout period of the task. If the task execution duration exceeds the specified timeout period, the task step times out.
- (Optional) `end`: specifies whether to proceed with the subsequent steps after the current step ends.
- (Optional) `inputMappings`: the input mappings. The input of the task step will be used as the event of a function invocation. For more information, see InvokeFunction.
- (Optional) `outputMappings`: the output mappings. `$local` is the result of a function invocation and must be in JSON format.

  > **Note** If no output mappings are specified, `$local` is used as the output of this step by default.

- (Optional) `errorMappings`: the error mappings. This attribute parameter is valid only when an error occurs during step execution and the `catch` parameter is specified. You can use the `$local.cause` and `$local.error` values to map error information to the output and pass it to the next step.

  > **Note** The `$local.error` and `$local.code` values are reserved for the system. The `source` field in errorMappings must be set to these two values. For more information, see Examples. In addition, the `errorMappings` parameter is optional. If it is not specified, error information cannot be obtained in the next step after an error occurs.

- `retry`: the group of retry policies. Each retry policy has the following attributes:
  - `errors`: the one or more errors. For more information, see Error definitions.
  - `intervalSeconds`: the initial interval between retries. The maximum value is 86,400 seconds. Default value: 1 second.
  - `maxIntervalSeconds`: the maximum time interval for retries. Both the maximum value and default value are 86,400 seconds.
  - `maxAttempts`: the maximum number of retries. Default value: 3.
  - `multiplier`: the value by which a retry interval is multiplied to make the next retry interval. Default value: 2.
- `catch`: the group of catch policies. Each catch policy has the following attributes:
  - `errors`: the one or more errors. For more information, see the following table.
  - `goto`: the name of the destination step.

    Note The destination step must be a step parallel to the current task step.

Error definitions

Function execution status	HTTP status code of a Function Compute response	Function Compute response	Serverless workflow step failure (**for retry and catch**)	Retry
Not executed	429	`ResourceExhausted`	`FC.ResourceExhausted`	Yes
Not executed	4xx but not 429	`ServiceNotFound`, `FunctionNotFound`, or `InvalidArgument`	`FC.ServiceNotFound`, `FC.FunctionNotFound`, or `FC.InvalidArgument`	No
Uncertain	500	`InternalServerError`	`FC.InternalServerError`	Yes
Not executed	503	`ResourceThrottled`	`FC.ResourceThrottled`	Yes
Execution successful, with an error code returned	200	A custom error, including `errorType`	`errorType`	Determined based on business
Execution failed, with an error code returned	200	No `errorType`	`FC.Unknown`	Yes

Function execution status	HTTP status code of a Function Compute response	Function Compute response	Serverless workflow step failure (**for retry and catch**)	Retry
Execution successful, with a non-JSON object returned	200	No `errorType`	`FC.InvalidOutput`	No

Other errors:

- `FnF.ALL`: captures all failures for retrying or goto use cases.

Examples

- Simple task steps

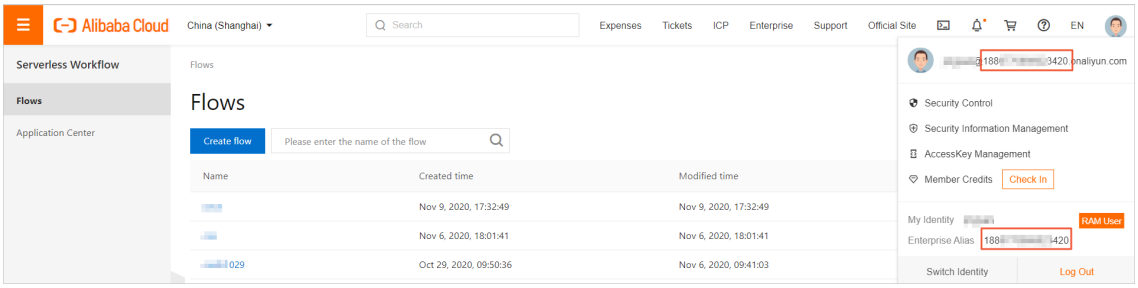
  The following sample flow contains a task step.

  - If the input is `{"name": "function flow"}`, the output is `{"hello": "function flow"}`.
  - If no input is specified for the flow or the flow input does not contain the `name` key, the task step execution fails, which causes a flow failure.
  - Define the flow.

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

    Parameters of `resourceArn`:

    - `{region}`: Replace it with the actual region, such as `cn-shanghai`.
    - `{accountID}`: Replace it with your account ID. You can view the account ID by clicking the profile picture on the **Flows** page of the Serverless Workflow console, as shown in the following figure.



- Define the function.

```
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")
```

- Retry

  The following example shows how to retry a task upon `MyError`. If no input is specified for the flow or the flow input does not contain the `name` key, Serverless workflow fails to retry tasks multiple times based on retry policies.

  - It waits 3 seconds after the first error occurs, and then invokes the function again.
  - It waits 6 seconds (`intervalSeconds x multiplier`) after the second error occurs, and then invokes the function again.
  - It waits 12 seconds (`intervalSeconds x multiplier x multiplier`) after the third error occurs, and then invokes the function again.
  - If an error still occurs after three retries, the number of retries exceeds `maxAttempts`. Therefore, the task step fails and the flow fails.

```
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
```

- Catch errors

  The following example shows how to catch `MyError` and then go to the `final` step. The error is caught, so the flow is successful.

```
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
```

- Catch errors with error mappings specified

  The following example shows how to catch `MyError` and then go to the `final` step. Error information can be obtained and processed in the `final` step because error mappings are specified. The flow is successful. You can also specify in the `errorMappings` to map the inputs and constants of this step to the outputs.

```
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 # This value is reserved for the system and can be used dire
ctly when an error occurs in this step.
      - target: errCode
        source: $local.error # This value is reserved for the system and can be used dire
ctly when an error occurs in this step.
    catch:
      - errors:
        - MyError
        goto: final
  - type: pass
    name: pass1
  - type: pass
    name: final
```

  In the `event` of the final step, you can see the following content in `EventDetail`:

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

4.Wait steps

This topic describes wait steps and related examples.

Overview

A wait step pauses a flow execution for a period of time before proceeding. You can select a relative time or use a timestamp to specify an absolute end time.

A wait step contains the following attributes:

- type: Required. The step type. The value wait indicates that the step is a wait step.
- name: Required. The step name.
- duration: Optional. The relative time to wait in seconds. It can be a constant or a parameter in the input. For example, `10` indicates waiting for 10 seconds, and `$.sleep` indicates that the wait time is obtained from the input `sleep` key. You must specify either `duration` or `timestamp`.
- timestamp: Optional. The absolute time to wait in RFC3339 format. It can be a constant or a parameter in the input. For example, `2019-05-02T15:04:05Z` indicates waiting until 15:04:05 on May 2, 2019 UTC. If the time is earlier than the current time, the wait step ends.
- end: Optional. Specifies whether to proceed with the subsequent steps after the current step ends.
- inputMappings: Optional. The input mappings.
- outputMappings: Optional. The output mappings. A wait step does not generate data, and its `$local` is empty.

> **Note** The maximum wait time is limited to two days.

Examples

- Wait time of 20 seconds

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

- Wait time determined by the input

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

- Absolute wait time

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

- Absolute wait time determined by the input

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

5.Pass steps

This topic describes pass steps and related examples.

Overview

A pass step can be used to output constants or convert inputs into the desired outputs. For example, when you define a flow, if you have not created functions of Function Compute for task steps, you can first plan and debug the flow logic by using control steps and pass steps, and then gradually replace the pass steps with task steps.

A pass step contains the following attributes:

- type: Required. The step type. The value pass indicates that the step is a pass step.
- name: Required. The step name.
- end: Optional. Specifies whether to proceed with the subsequent steps after the current step ends.
- inputMappings: Optional. The input mappings.
- outputMappings: Optional. The output mappings. This step does not generate data, and its `$local` is empty.

Examples

The following example defines a pass step that outputs an array of uppercase letters.

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

6.Choice steps

This topic describes the basics and examples of choice steps, and related conditional expressions.

Overview

Choice steps allow execution of different steps in a flow, similar to `switch-case` in programming languages. A choice step contains multiple choices and a default. Each choice contains a conditional expression, several steps, and goto instructions. The default contains only several steps and goto instructions. When the flow proceeds to a choice step, the system evaluates whether the conditional expressions return `True` in the defined sequence.

- If `True` is returned, the steps and then goto instructions defined in the corresponding choice are executed.
- If no choice returns `True`, the steps and goto instructions defined in the default are executed.
- If no default is defined, the choice step ends.

A choice step contains the following attributes:

- type: Required. The step type. The value choice indicates that the step is a choice step.
- name: Required. The step name.
- choices: Required. Multiple choices of the array type. Each element corresponds to a choice.
  - condition: Required. The conditional expression. Conditional expressions reference step inputs based on JSON paths (`$.key`).
  - steps: Optional. The multiple serial steps defined for a choice.
  - goto: Optional. The name of the target step, which must be parallel to the current choice step.
- default: Required. The default.
  - steps: Optional. The multiple serial steps defined for the default.
  - goto: Optional. The name of the target step, which must be parallel to the current choice step.
- end: Optional. Specifies whether to proceed with the subsequent steps after the current step ends.
- inputMappings: Optional. The input mappings.
- outputMappings: Optional. The output mappings. The `$local` of this step indicates the execution result of the choice branch.

> **Note** If no output mappings are specified, `$local` is used as the output of this step by default.

Examples

The following sample flow defines a choice step.

- If the value of `status` in the input is `ready`, the `pass1`, `pass3`, and `final` steps of the first choice are executed in sequence.
- If the value of `status` in the input is `failed`, the goto instructions of the second choice are executed, the choice step ends, and the `final` step is executed.
- If the value of `status` in the input is neither `ready` nor `failed`, the default is executed. In other words, the `pass2` and `final` steps are executed.

```
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
```

Conditional expressions

A conditional expression consists of the following operations and variables:

- Comparison operations: `>` `>=` `<` `<=` `==` `!=`. They are applicable to strings and numbers.
- Logical operations: `||` `&&`.
- String constants: A string constant is enclosed in double quotation marks (") or grave accents (`), for example, "foobar" or `foobar`.
- Numeric constants: `1` `12.5`.
- Boolean constants: `true` or `false`.
- Prefix: `!` `-`.
- Contain: `in`, which is used to determine whether an array contains a value or whether an object contains a key value.

The following example shows the execution results of steps for different conditional expressions.

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

Conditional expression	Result
`$.a==1`	true
`$.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 && $.b.b1`	true
`$.a==1 || $.b.b1`	true
`$.a==2 && $.b.b1`	false
`$.a==2 || $.b.b1`	true
`$.c[0]==1`	true
`$.c[0]==$.a`	true

7.Parallel steps

This topic describes parallel steps and related examples.

Overview

A parallel step is used to execute multiple child steps in parallel. A parallel step defines multiple branches, each of which contains a series of serial steps.

Each branch of a parallel step corresponds to a local variable. When a parallel step is executed, serial steps in all branches are executed concurrently. These serial steps change the local variables corresponding to their branches. After all branches are executed, output mappings can be used to convert the local variable arrays of branches into the output of the parallel step.

> **Note** The maximum number of branches in a parallel step is 100.

A parallel step contains the following attributes:

- type: Required. The step type. The value parallel indicates that the step is a parallel step.
- name: Required. The step name.
- branches: Required. Multiple branches of the array type. Each element corresponds to a branch.
  - steps: Required. The multiple serial steps defined for a branch.
- end: Optional. Specifies whether to proceed with the subsequent steps after the current step ends.
- inputMappings: Optional. The input mappings.
- outputMappings: Optional. The output mappings. In this step, the `$local` is an array. Each element in the array is a JSON object that records the execution result of each branch.

  > **Note** If no output mappings are specified, this step has no output by default.

Examples

The following sample flow defines a parallel step. This parallel step contains two branches, and each branch contains a pass step.

```
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
```

- The following information is the output of `pass1`:

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

- The following information is the output of `pass2`:

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

- The following information is the output of `myparallel`:

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

8.Foreach steps

This topic describes foreach steps and related examples.

Overview

A foreach step traverses parameters of an array type in the input, and executes the serial steps for each element in the array in parallel. Foreach steps are similar to `foreach` in programming languages. The difference is that iterations of foreach steps are executed in parallel.

Each iteration of a foreach step corresponds to a local variable. In a foreach step, serial steps of each element in the input parameters are executed in parallel. These serial steps change the local variables corresponding to their iterations. After all iterations are executed, output mappings can be used to convert the local variable arrays of iterations into the output of the foreach step.

> **Note** The maximum number of serial steps that can be concurrently executed in a foreach step is 100.

A foreach step contains the following attributes:

- type: Required. The step type. The value foreach indicates that the step is a foreach step.
- name: Required. The step name.
- iterationMapping: Required. The iterative mapping.
  - collection: Required. The input parameter that serves as a collection for a foreach step.
  - item: Required. The name of the current element that is incorporated into the iteration input.
  - index: Optional. The name of the current position that is incorporated into the iteration input.
- steps: Required. The serial steps.
- end: Optional. Specifies whether to proceed with the subsequent steps after the current step ends.
- inputMappings: Optional. The input mappings.
- outputMappings: Optional. The output mappings. In this step, the `$local` is an array. Each element in the array is a JSON object that records the result of each iteration.

  > **Note** If no output mappings are specified, this step has no output by default.

Examples

The following sample flow defines a foreach step that contains a task step.

```
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
```

- The following information is the flow input. No input mapping is specified for the `myforeach` step. Therefore, its input is the same as the flow input.

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

- No input mapping is defined for `toUpperCase`. Therefore, its input is the same as the input of the parent step. According to `iterationMapping`, the system inputs the current elements (`a`, `b`, and `c` in sequence) as values and the `name` as the key upon each iteration

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

- `toUpperCase` is executed three times, with the following outputs in sequence:

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

- The local variable of `myforeach` is an array, with the following values:

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

- The following information is the output of `myforeach`. No output or result mappings are defined for the flow. Therefore, the output is the final flow output.

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

9.Succeed steps

This topic describes succeed steps and related examples.

Overview

A succeed step ends a series of serial steps in advance, similar to `return` in programming languages. Flow Definition Language (FDL) `steps` are serial steps. In general, a next step is executed after a previous step is completed. However, after a succeed step is executed, subsequent steps are not executed. Succeed steps are usually used together with choice steps. When the conditions of a choice step are met, the flow goes to a succeed step, and no other steps are executed.

A succeed step contains the following attributes:

- type: Required. The step type. The value succeed indicates that the step is a succeed step.
- name: Required. The step name.
- inputMappings: Optional. The input mappings.
- outputMappings: Optional. The output mappings.

Examples

The following sample flow ends in advance by using a succeed step.

- If the value of `status` in the input is `ready`, the `pass1` and `final` steps of the first choice are executed in sequence. The `final` step is a succeed step. Therefore, after it is executed, the `handle_failure` step will not be executed.
- If the value of `status` in the input is `failed`, the goto instructions of the second choice are executed, the choice step ends, and the `handle_failure` step is executed.
- If the input does not contain `status` or the value of `status` is neither `ready` nor `failed`, the default choice is executed, that is, the `pass2` and `handle_failure` steps are executed.

```
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.Fail steps

This topic describes fail steps and related examples.

Overview

A fail step ends a series of steps in advance, similar to `raise` and `throw` in programming languages. After a fail step is executed in a flow, steps following the fail step will not be executed and the parent step of the fail step also fails. This continues until the flow execution fails.

A fail step contains the following attributes:

- type: Required. The step type. The value fail indicates that the step is a fail step.
- name: Required. The step name.
- error: Optional. The error type.
- cause: Optional. The cause of the error.
- inputMappings: Optional. The input mappings.
- outputMappings: Optional. The output mappings.

Examples

The following sample flow ends in advance by using a fail step.

- If the value of `status` in the input is `ready`, the `pass1` and `final` steps of the first choice are executed in sequence.
- If the value of `status` in the input is `failed`, the goto instructions of the second choice are executed, the choice step ends, and the `handle_failure` step is executed. The `handle_failure` step is a fail step. Therefore, after it is executed, the `final` step will not be executed.
- If the input does not contain `status` or the value of `status` is neither `ready` nor `failed`, the default choice is executed, that is, the `pass2` and `handle_failure` steps are executed.

```
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
```