Workflow Control and Functional Operators

Treasure Workflow is based on the Treasure Data created Digdag open source project. The original source of Digdag information can be found at docs.digdag.io. Most, but not all, Digdag operators can be used as part of Treasure Workflow.

Operator	Description
call>:	Calls another workflow
require>:	Depends on another workflow
loop>:	Repeat tasks
for_each>:	Repeat tasks for values
for_range>:	Repeat tasks for a range
if>:	Conditional execution
fail>:	Changes the workflow status to failed
wait>:	Waits for the duration specified

- Control Operators
- Functional Operators

call>:

call> operator calls another workflow.

This operator embeds another workflow as a subtask. This operator waits until all tasks of the workflow complete.

```
# workflow1.dig
+step1:
  call>: another_workflow.dig
+step2:
  call>: common/shared_workflow.dig
```

```
# another_workflow.dig
+another:
  sh>: ../scripts/my_script.sh
```

Options

- **call>**: FILE

  Path to a workflow definition file. Filename must end with `.dig`. If called, and the workflow is in a subdirectory, the workflow uses the subdirectory as the working directory. For example, a task has `call>: common/called_workflow.dig`, using `queries/data.sql` file in the called workflow should be `../queries/data.sql`.

  Examples:

  `call>: another_workflow.dig`

require>:

require> operator requires the completion of another workflow. This operator is similar to call> operator, but this operator doesn't start the other workflow if it's already running or has done for the same session time of this workflow. If the workflow is running or newly started, this operator waits until it completes.

+step1: require>: another_workflow ```

+step2: sh>: tasks/step2.sh ```

Options

- **require>**: NAME

Name of a workflow.

Examples:

```
require>: another_workflow
```

- **session_time**: ISO_TIME_WITH_ZONE

  Examples:

  ```
  require>: another_workflow session_time: 2017-01-01T00:00:00+00:00
  ```

  timezone: UTC

  schedule: monthly>: 1,09:00:00

  +depend_on_all_daily_workflow_in_month: loop>: ${moment(last_session_time).daysInMonth()} _do:

  ```
  require>: daily_workflow
  session_time: ${moment(last_session_time).add(i, 'day')}
  ```

- **ignore_failure**: BOOLEAN

  This operator fails when the dependent workflow finished with errors by default.

  But if `ignore_failure: true` is set, this operator succeeds even when the workflow finished with errors.

  ```
  require>: another_workflow ignore_failure: true
  ```

loop>:

loop> operator runs subtasks multiple times. It repeats tasks.

This operator exports `${i}` variable for the subtasks. Its value begins from 0. For example, if count is 3, a task runs with i=0, i=1, and i=2.

```
+repeat:
  loop>: 7
  _do:
    +step1:
      echo>: ${moment(session_time).add(i, 'days')} is ${i} days later than ${session_date}
    +step2:
      echo>: ${moment(session_time).add(i, 'hours')} is ${i} hours later than ${session_local_time}.
```

Options

- **loop>:** COUNT Number of times to run the tasks.

  Examples:

  ```
  loop>: 7
  ```

- **_parallel**: BOOLEAN Runs the repeating tasks in parallel.

  Examples:

  ```
  \_parallel: true
  ```

- **_do**: TASKS

  Tasks to run.

for_each>:

Repeat tasks for values.

for_each> operator runs subtasks multiple times using sets of variables.

(This operator is EXPERIMENTAL. Parameters are subject to change)

```
+repeat:
  for_each>:
    fruit: [apple, orange]
    verb: [eat, throw]
  _do:
    echo>: ${verb} ${fruit}
    # this will generate 4 tasks:
    #  +for-fruit=apple&verb=eat:
    #    echo>: eat apple
    #  +for-fruit=apple&verb=throw:
    #    echo>: throw apple
    #  +for-fruit=orange&verb=eat:
    #    echo>: eat orange
    #  +for-fruit=orange&verb=throw:
    #    echo>: throw orange
```

Options

- **for_each>**: VARIABLES

  Variables used for the loop in `key: [value, value, ...]` syntax. Variables can be an object or JSON string.

  Examples:

  ```
  for_each>: {i: [1, 2, 3]}
  ```

  Examples:

  ```
  for_each>: {i: '[1, 2, 3]'}
  ```
- **_parallel**: BOOLEAN

  Runs the repeating tasks in parallel.

  Examples:

  ```
  \_parallel: true
  ```
- **_do**: TASKS

  Tasks to run.

for_range>:

Repeat tasks for a range. This operator is EXPERIMENTAL. Parameters are subject to change.

for_range> operator runs subtasks multiple times using sets of variables.

This operator exports `${range.from}`, `${range.to}`, and `${range.index}` variables for the subtasks. Index begins from 0.

```
+repeat:
  for_range>:
    from: 10
    to: 50
    step: 10
  _do:
    echo>: processing from ${range.from} to ${range.to}.
    # this will generate 4 tasks:
    #  +range-from=10&verb=20:
    #    echo>: processing from 10 to 20.
    #  +range-from=20&verb=30:
    #    echo>: processing from 20 to 30.
    #  +range-from=30&verb=40:
    #    echo>: processing from 30 to 40.
    #  +range-from=40&verb=50:
    #    echo>: processing from 40 to 50.
```

- **for_each>**: object of from, to, and slices or step

This nested object is used to declare a range for **from** to **to**.

Then you divide the range into a fixed number of slices using **slices** option, or divide the range by width by **step** option. Setting both slices and step is an error.

Examples:

```
for_range>: from: 0 to: 10 step: 3
# this repeats tasks for 4 times (number of slices is computed automatically):
# * {range.from: 0, range.to: 3, range.index: 0}
# * {range.from: 3, range.to: 6, range.index: 1}
# * {range.from: 6, range.to: 9, range.index: 2}
# * {range.from: 9, range.to: 10, range.index: 3}
\_do: echo>: from ${range.from} to ${range.to}

for_range>: from: 0 to: 10 slices: 3
# this repeats tasks for 3 times (size of a slice is computed automatically):
# * {range.from: 0, range.to: 4, range.index: 0}
# * {range.from: 4, range.to: 8, range.index: 1}
# * {range.from: 8, range.to: 10, range.index: 2}
\_do: echo>: from ${range.from} to ${range.to}
```

- **_parallel**: BOOLEAN

  Runs the repeating tasks in parallel.

  Examples:

  `\_parallel: true`
- **_do**: TASKS

  Tasks to run.

if>:

Conditional execution. This operator is EXPERIMENTAL. Parameters are subject to change.

if> operator runs subtasks if `true` is given.

```
+run_if_param_is_true:
  if>: ${param}
  _do:
    echo>: ${param} == true
```

Options

- **if>**: BOOLEAN

  `true` or `false`.
- **_do**: TASKS

  Tasks to run if `true` is given.

fail>:

Changes the workflow status to failed.

fail> always fails and makes the workflow failed.

This operator is used with the if> operator to validate results of a previous task. Use it with `_check`directive so that a workflow fails when the validation doesn't pass.

```
+fail_if_too_few:
  if>: ${count < 10}
  _do:
    fail>: count is less than 10!
```

Options

- **fail>**: STRING

  Message so that `_error` task can refer the message using `${error.message}` syntax.

wait>:

Waits for a specific duration in the workflow.

This operator temporarily suspends the current execution and waits for the duration specified. The operation is resumed once the wait duration has passed.

```
+wait_10s:
  wait>: 10s
```

Options

- **wait>**: DURATION

  Duration to wait.