
Cloudmesh Workflow Documentation

Release 0.1.2-1457030360-944db997

Badi' Abdul-Wahid

March 03, 2016

1	workflow package	3
1.1	Usage Summary	3
1.2	Description	3
1.3	Usage	4
1.4	Cloudmesh Workflow Example	4
1.5	Concepts	5
1.5.1	Deferring function evaluation	5
1.5.2	Composing Nodes for parallel/sequential semantics	5
1.5.3	Evaluation of a delayed function	5
1.6	API	5
2	Indices and tables	11
	Python Module Index	13

Contents:

workflow package

1.1 Usage Summary

1. Define delayed functions:

```
>>> @delayed()
... def A(foo):
...     time.sleep(0.05)
...     print foo
>>>
>>> @delayed()
... def B():
...     print 'Boo!'
>>>
>>> @delayed()
... def C(x, y):
...     return x ** y
```

2. Compose the functions using | and & for parallel and sequential evaluation:

```
>>> root_node = (A('hello world!') | B()) & C(4, 2)
>>> root_node
<cloudmesh_workflow.workflow.AndNode object at ...>
```

3. Evaluate the resulting graph

```
>>> evaluate(root_node.graph)
Boo!
hello world!
```

1.2 Description

This module provides an api for building a workflow graph of labeled functions which can then be evaluated. Nodes connected with a desired ordering or run sequentially, others can be run in parallel.

Syntax is inspired by the parallel (||) and sequential (;) operators. For example:

```
(A || B) ; (C || D)
```

means that A and B can be evaluated in parallel, and likewise C and D, but both A and B must be completed before C or D may begin.

The python implementation overrides the bitwise **OR** (`|`) and **AND** (`&`) operators to provide a similar syntactic feel. The example above should be defined as such:

```
(A() | B()) & (C() | D())
```

Note: The python operator precedence for `|` and `&` is unchanged: `&` has higher precedence than `|`.

1.3 Usage

The first part is to mark top-level functions as `delayed()`. The `@delayed()` decoration wraps the function so that calling the function inserts the `Node`, without applying the parameters, into the call `Graph`. You can access the `graph` property of any node to get the current call graph.

For example, define two delayed functions A and B:

```
>>> @delayed()
... def A(x):
...     return x*2
```

```
>>> @delayed()
... def B(x, y):
...     return x ** y
```

Compose A and B to run in parallel

```
>>> node = A(24) | B(40, 2)
```

Evaluate the graph:

```
>>> evaluate(node.graph)
```

Print the results:

```
>>> for _, data in node.graph.nodes(data=True):
...     n = data['node']
...     print n.name, n.result.result()
| None
A 48
B 1600
```

1.4 Cloudbase Workflow Example

Warning: This is a proposed usage example and hasn't been tested yet.

```
from cloudbase_base import Shell
from workflow import delayed, evaluate

@delayed()
def FutureSystems():
    "Start a VM on FutureSystems OpenStack Kilo"
    Shell.cm('boot', 'kilo')
```



```
@delayed()
def Cybera(x, y):
    "Start a VM on Cybera cloud"
    Shell.cm('boot', 'cybera')

@delayed()
def Rackspace():
    "Start a VM on Rackspace"
    Shell.cm('boot', 'rackspace')

def main():
    "Boot machines in parallel"
    node = FutureSystems() | Cybera() | Rackspace()
    evaluate(node.graph)
```

1.5 Concepts

1.5.1 Deferring function evaluation

A *delayed* is intended to be used as a decorator to lift arbitrary functions to have delayed semantics. Evaluation semantics of *delayed* objects is:

1. calling a delayed function stores the arguments and returns a *Node*.
2. *Nodes* are composed using bitwise & and | operators to denote sequential and parallel evaluation order, respectively.

1.5.2 Composing Nodes for parallel/sequential semantics

A *Node* captures the evaluation state of a *delayed* function. It provides several important attributes:

1. *graph*: the evaluation graph in which the function is located.
2. *f*: the function to evaluate.
3. *name*: the name of the node. Typically captured from *f*, but may be a shorthand representation of *OpNode*.
4. *result*: the status and result of the evaluation.

Nodes are created by calling *delayed* functions and then composed using & and |. Each composition returns a new *Node* in the graph.

1.5.3 Evaluation of a delayed function

Once *Nodes* have been composed to achieve the desired parallelism, evaluate the graph by calling *evaluate()* on the *graph* attribute of the composed node.

1.6 API

```
class cloudmesh_workflow.workflow.delayed(graph=None, **kws)
    Bases: object
```

A *delayed* is a decorator that delays evaluation of a function until explicitly called for using *evaluate()*.

Intended usage: decorate a function such that `__call__()` ing it returns a *Node* instance that can be combined with other *Node* instances using the bitwise `__and__()` (&) and `__or__()` (|) operators to create a workflow.

Example:

```
>>> @delayed()
... def foo(*args):
...     for a in args:
...         print a
>>> type(foo)
<type 'function'>
>>> node = foo(1, 2) & foo(3, 4)
>>> print node
<cloudmesh_workflow.workflow.AndNode object at ...>
>>> evaluate(node.graph)
1
2
3
4
```

kws will be passed to the *Node* constructor.

Parameters *graph* (*Graph* or None) – If *graph* not None, this explicitly specifies the *graph* into which the *Node* will be inserted.

`cloudmesh_workflow.workflow.evaluate(Graph) → None`
Graph -> IO ()

Starting from the root node, evaluate the branches. The *graph* nodes are updated in-place.

Example:

```
>>> @delayed()
... def foo(a):
...     return a
>>> node = foo(42) & foo(24)
>>> print evaluate(node.graph)
None
>>> for _, data in node.graph.nodes(data=True):
...     n = data['node']
...     print n.name, n.result.result()
& None
foo 42
foo 24
```

class `cloudmesh_workflow.workflow.Node(f_args_kws, graph=None, executor=None, time-out=None)`

Bases: `traits.has_traits.HasTraits`

A node in the *Graph* and associated state.

Nodes can be composed using bitwise `__and__()` and `__or__()` operators to denote sequential or parallel evaluation order, respectively.

For example, give A, B, and C functions that have been lifted to a *Node* type (eg through the *delayed* decorator `@delayed()`), to evaluate A and B in parallel, then C:

```
G = ( (A(argA0, argA1) | B()) & C(argC) ).graph
```

will create the call *Graph* G. In order to evaluate G:

evaluate(G)

Create a *Node* to evaluate a function *f* in some graph using a given executor

Parameters

- **func** – *f_args_kws* = (*f*, *args*, *kws*) a 3-tuple of the function to evaluate (any callable) along with positional and keyword arguments.
- **graph** – The *Graph* in which to insert the node upon composition with others. A value of *None* will create a new graph. When composed with another node in a different *Node.graph()* the two graphs will be merged.
- **executor** – a *futures.Executor* instance
- **timeout** – seconds (float or int) to wait.

children

[*Node*]

The children of this node. See *Node.children_iter()*

Return type list of *Node*

children_iter

Generator of *Nodes*

This *yield*'s all the children *Nodes* of this node.

Returns Child nodes of this node.

Return type generator of *Node*

compose (*other*, *callable*(*graph*=*Graph*)) → *OpNode*

Compose this *Node* with another *Node*.

Two *Nodes* are composed using a proxy *OpNode*. The *OpNode* defines the evaluation semantics of its child nodes (eg sequential or parallel).

Parameters

- **other** – a *Node*
- **MkOpNode** – a callable with keyword arg *graph* constructor for the proxy node

Returns A new *Node* with *self* and *other* and children.

Return type *Node*

eval () → *None*

Start and wait for a node.

start () → *None*

Start evaluating this node

Start evaluating this nodes function *self.f* if it hasn't already started.

wait () → *None*

Wait for this node to finish evaluating

This may timeout if *timeout* is specified.

class *cloudmesh_workflow.workflow.OpNode* (***kwargs*)

Bases: *cloudmesh_workflow.workflow.Node*

A proxy node defining the evaluation semantics of its children *Nodes*

Intended usage: this class is not intended to be instantiated directly. Rather, classes should inherit from *OpNode* to define the desired semantics.

class `cloudmesh_workflow.workflow.AndNode` (***kwargs*)

Bases: `cloudmesh_workflow.workflow.OpNode`

Sequential evaluation semantics.

Children of *AndNode* will be evaluated in the order in which they were added as children of this node.

Example:

```
>>> @delayed()
... def foo(a): return 42
>>> foo(42) & foo(24)
<cloudmesh_workflow.workflow.AndNode object at ...>
```

start ()

wait ()

class `cloudmesh_workflow.workflow.OrNode` (***kwargs*)

Bases: `cloudmesh_workflow.workflow.OpNode`

Parallel evaluation semantics

Children of *OrNode* will be evaluated in parallel, sparked in the order in which they were added as children of this node.

Example:

```
>>> @delayed()
... def foo(a): return 42
>>> foo(42) | foo(24)
<cloudmesh_workflow.workflow.OrNode object at ...>
```

start ()

wait ()

class `cloudmesh_workflow.workflow.Graph` (*data=None, **attr*)

Bases: `networkx.classes.digraph.DiGraph`

A NetworkX `networkx.DiGraph()` where the ordering of edges/nodes is preserved

Initialize a graph with edges, name, graph attributes.

data [input graph] Data to initialize graph. If *data=None* (default) an empty graph is created. The data can be an edge list, or any NetworkX graph object. If the corresponding optional Python packages are installed the data can also be a NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.

name [string, optional (default='')] An optional name for the graph.

attr [keyword arguments, optional (default= no attributes)] Attributes to add to graph as key=value pairs.

convert

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G = nx.Graph(name='my_graph')
>>> e = [(1,2), (2,3), (3,4)] # list of edges
>>> G = nx.Graph(e)
```

Arbitrary graph attribute pairs (key=value) may be assigned

```
>>> G=nx.Graph(e, day="Friday")
>>> G.graph
{'day': 'Friday'}
```

adjlist_dict_factory
alias of OrderedDict

node_dict_factory
alias of OrderedDict

cloudmesh_workflow.workflow.**find_root_node**(*Graph*) → Node
Graph → *Node*

Find the root node of a connected DAG

Return type *Node*

Example:

```
>>> @delayed()
... def foo(a):
...     return a
>>> node = foo(42) | foo(24)
>>> print node.name
|
>>> print find_root_node(node.graph).name
|
```

Indices and tables

- `genindex`
- `modindex`
- `search`

C

`cloudmesh_workflow.workflow`, 3

A

adjust_dict_factory (cloudmesh_workflow.workflow.Graph attribute), 9

AndNode (class in cloudmesh_workflow.workflow), 8

C

children (cloudmesh_workflow.workflow.Node attribute), 7

children_iter (cloudmesh_workflow.workflow.Node attribute), 7

cloudmesh_workflow.workflow (module), 3

compose() (cloudmesh_workflow.workflow.Node method), 7

D

delayed (class in cloudmesh_workflow.workflow), 5

E

eval() (cloudmesh_workflow.workflow.Node method), 7

evaluate() (in module cloudmesh_workflow.workflow), 6

F

find_root_node() (in module cloudmesh_workflow.workflow), 9

G

Graph (class in cloudmesh_workflow.workflow), 8

N

Node (class in cloudmesh_workflow.workflow), 6

node_dict_factory (cloudmesh_workflow.workflow.Graph attribute), 9

O

OpNode (class in cloudmesh_workflow.workflow), 7

OrNode (class in cloudmesh_workflow.workflow), 8

S

start() (cloudmesh_workflow.workflow.AndNode method), 8

start() (cloudmesh_workflow.workflow.Node method), 7

start() (cloudmesh_workflow.workflow.OrNode method), 8

W

wait() (cloudmesh_workflow.workflow.AndNode method), 8

wait() (cloudmesh_workflow.workflow.Node method), 7

wait() (cloudmesh_workflow.workflow.OrNode method), 8