Couler

Unified Interface for Constructing and Managing Workflows

Yuan Tang from Ant Group
@terrytangyuan

Existing Workflow Solutions

Apache Airflow

```
def create_dag(dag_id,
               schedule,
               dag_number,
               default_args):

    def hello_world_py(*args):
        print('Hello World')
        print('This is one of Python function on DAG: {}'.format(str(dag_number)))

    dag = DAG(dag_id,
              schedule_interval=schedule,
              default_args=default_args)

    with dag:
        t1 = PythonOperator(
            task_id='hello_world',
            python_callable=hello_world_py,
            dag_number=dag_number)

    return dag


# build a dag for each number in range(10)
for n in range(1, 10):
    dag_id = 'hello_world_{}'.format(str(n))

    default_args = {'owner': 'airflow',
                    'start_date': datetime(2018, 1, 1)
                    }

    schedule = '@daily'

    dag_number = n

    globals()[dag_id] = create_dag(
        dag_id,
        schedule,
        dag_number,
        default_args)
```

Kubeflow Pipelines

```
class FlipCoinOp(dsl.ContainerOp):
    """Flip a coin and output heads or tails randomly."""

    def __init__(self):
        super(FlipCoinOp, self).__init__(
            name='Flip',
            image='python:alpine3.6',
            command=['sh', '-c'],
            arguments=['python -c "import random; result = \'heads\' if random.randint(0,1) == 0 '
                       'else \'tails\'; print(result)" | tee /tmp/output'],
            file_outputs={'output': '/tmp/output'})


class PrintOp(dsl.ContainerOp):
    """Print a message."""
    def __init__(self, msg):
        super(PrintOp, self).__init__(
            name='Print',
            image='alpine:3.6',
            command=['echo', msg],
        )


# define the recursive operation
@graph_component
def flip_component(flip_result):
    print_flip = PrintOp(flip_result)
    flipA = FlipCoinOp().after(print_flip)
    with dsl.Condition(flipA.output == 'heads'):
        flip_component(flipA.output)


@dsl.pipeline(
    name='pipeline flip coin',
    description='shows how to use graph_component.'
)
def recursive():
    flipA = FlipCoinOp()
    flipB = FlipCoinOp()
    flip_loop = flip_component(flipA.output)
    flip_loop.after(flipB)
    PrintOp('cool, it is over. %s' % flipA.output).after(flip_loop)
```

Argo Python DSL

```python
class DagDiamond(Workflow):

    @task
    @parameter(name="message", value="A")
    def A(self, message: V1alpha1Parameter) -> V1alpha1Template:
        return self.echo(message=message)

    @task
    @parameter(name="message", value="B")
    @dependencies(["A"])
    def B(self, message: V1alpha1Parameter) -> V1alpha1Template:
        return self.echo(message=message)

    @task
    @parameter(name="message", value="C")
    @dependencies(["A"])
    def C(self, message: V1alpha1Parameter) -> V1alpha1Template:
        return self.echo(message=message)

    @task
    @parameter(name="message", value="D")
    @dependencies(["B", "C"])
    def D(self, message: V1alpha1Parameter) -> V1alpha1Template:
        return self.echo(message=message)

    @template
    @inputs.parameter(name="message")
    def echo(self, message: V1alpha1Parameter) -> V1Container:
        container = V1Container(
            image="alpine:3.7",
            name="echo",
            command=["echo", "{{inputs.parameters.message}}"],
        )

        return container
```

What is Couler?

Unified Interface for Constructing and Managing Workflows

Couler Core APIs

- Basic Operation
  - couler.run_step(step_def)
- Control flow
  - map(func, *args, **kwargs)
  - when(cond, if_op, else_op)
  - while_loop(cond, func, *args, **kwargs)
- DAG
  - couler.dag([A, B, ...])
  - couler.set_dependencies(A, B)
- Utilities
  - submit(config=workflow_config(schedule="* * * * 1"))
  - get_status(workflow_name)

Why use Couler?

Why use Couler?

- Using Python for Workflow construction
  - Define workflow programmatically -> translated to Argo YAML specification.
  - Reuse Argo Python client for schema validation over Argo Workflows.
- Simplicity
  - Unified interface and imperative programming style for defining workflows.
- Extensibility
  - Extensible to support various workflow engines.
- Reusability
  - Reusable steps for tasks such as distributed training of machine learning models.

Couler Example #1 - Coin Flip

```python
def random_code():
    import random

    result = "heads" if random.randint(0, 1) == 0 else "tails"
    print(result)


def flip_coin():
    return couler.run_script(
        image="couler/python:3.6",
        source=random_code,
    )


def heads():
    return couler.run_container(
        image="couler/python:3.6",
        command=["bash", "-c", 'echo "it was heads"'],
    )


def tails():
    return couler.run_container(
        image="couler/python:3.6",
        command=["bash", "-c", 'echo "it was tails"'],
    )


result = flip_coin()
couler.when(couler.equal(result, "heads"), lambda: heads())
couler.when(couler.equal(result, "tails"), lambda: tails())
```

Couler Example #2 - DAG

```python
def job_a(message):
    couler.run_container(
        image="docker/whalesay:latest",
        command=["cowsay"],
        args=[message],
        step_name="A",
    )


def job_b(message):
    couler.run_container(
        image="docker/whalesay:latest",
        command=["cowsay"],
        args=[message],
        step_name="B",
    )


def job_c(message):
    couler.run_container(
        image="docker/whalesay:latest",
        command=["cowsay"],
        args=[message],
        step_name="C",
    )


def job_d(message):
    couler.run_container(
        image="docker/whalesay:latest",
        command=["cowsay"],
        args=[message],
        step_name="D",
    )
```

```python
#  A
# / \
# B  C
# /
# D
def linear_option1():
    couler.dag(
        [
            [lambda: job_a(message="A")],
            [lambda: job_a(message="A"), lambda: job_b(message="B")],  # A -> B
            [lambda: job_a(message="A"), lambda: job_c(message="C")],  # A -> C
            [lambda: job_b(message="B"), lambda: job_d(message="D")],  # B -> D
        ]
    )


def linear_option2():
    couler.set_dependencies(lambda: job_a(message="A"), dependencies=None)
    couler.set_dependencies(lambda: job_b(message="B"), dependencies=["A"])
    couler.set_dependencies(lambda: job_c(message="C"), dependencies=["A"])
    couler.set_dependencies(lambda: job_d(message="D"), dependencies=["B"])


#  A
# / \
# B  C
# \  /
#  D
def diamond():
    couler.dag(
        [
            [lambda: job_a(message="A")],
            [lambda: job_a(message="A"), lambda: job_b(message="B")],  # A -> B
            [lambda: job_a(message="A"), lambda: job_c(message="C")],  # A -> C
            [lambda: job_b(message="B"), lambda: job_d(message="D")],  # B -> D
            [lambda: job_b(message="C"), lambda: job_d(message="D")],  # C -> D
        ]
    )
```

Reusable Steps

- Kubeflow operators for distributed ML jobs
- Integration with third-party data sources and storage options
- ...

Project Status and Next Steps

- Initially developed and used at Ant Group with support for Argo Workflows
- Open sourced at: https://github.com/couler-proj/couler
- Better integration with argoproj-labs/argo-client-python
- Collaboration with open source communities and organizations
  - Other backends
  - Reusable steps
- Communications
  - dedicated Couler Slack workspace (link can be found in the repo's README.md)
  - #argo-sdks on Argo Slack workspace
  - @CoulerProject on Twitter