一些Apache Airflow最佳实践

设计DAG重跑机制，DAG任务事务原子性

1）按照业务日期或Parttiton，取数逻辑一致条件下多次重跑取数结果一致

- sql例子，来源和目标处理逻辑条件是根据业务日期

```
-- 创建临时表存储中间结果
CREATE TEMP TABLE tmp_results AS
SELECT
    business_date,
    column1,
    column2,
    ...
FROM source_table
WHERE business_date = 'desired_date';

-- 删除目标表中旧数据
DELETE FROM target_table
WHERE business_date = 'desired_date';

-- 将中间结果插入到目标表
INSERT INTO target_table
SELECT * FROM tmp_results;
```

2）对目标表使用delete/insert分区增量数据或者upsert全量写入数据到目标，实现重跑，避免目标表数据重复

- 根据业务日期进行增量处理

```
-- 创建临时表存储新数据
CREATE TEMP TABLE tmp_new_data AS
SELECT
    business_date,
    column1,
    column2,
    ...
FROM source_table
WHERE business_date > (SELECT MAX(business_date) FROM target_table);

-- 插入新数据到目标表
```

```
INSERT INTO target_table
SELECT * FROM tmp_new_data;
```

- 根据业务日期+业务PK进行增量处理

```
-- 创建临时表存储新数据
CREATE TEMP TABLE tmp_new_data AS
SELECT
    source.pk_column,
    source.business_date,
    source.column1,
    source.column2,
    ...
FROM source_table AS source
LEFT JOIN target_table AS target
    ON source.pk_column = target.pk_column
WHERE source.business_date > (SELECT MAX(business_date) FROM target_table);

-- 更新已有数据
UPDATE target_table AS target
SET
    column1 = tmp.column1,
    column2 = tmp.column2,
    ...
FROM tmp_new_data AS tmp
WHERE target.pk_column = tmp.pk_column;

-- 插入新数据到目标表
INSERT INTO target_table
SELECT * FROM tmp_new_data
WHERE pk_column NOT IN (SELECT pk_column FROM target_table);
```

4）慎重删除task，因为被删除的task将无法找到对应的日志信息，通常建议新建一个DAG

5）下游作业利用Sensor检查上游的执行结果，例如检查S3上文件是否就绪

```
task = PushToS3(...)
check = S3KeySensor(
    task_id="check_parquet_exists",
    bucket_key="s3://bucket/key/foo.parquet",
    poke_interval=0,
    timeout=0,
)
task >> check
```

如何编写性能好的DAG

优化DAG Python顶级代码，提升DAG加载性能

1）编写DAG Pyton脚本只在顶部引用全局使用的包模块，建议在使用子模块里再引用。

2）避免在顶级代码里编写复杂的或者耗时的处理逻辑，例如数据库操作、网络处理等。

3）避免在顶级代码里使用Airflow应用变量，应用变量只在具体task任务使用时在获取。

4）Airflow应用变量也可使用Jinjia模板实现变量动态获取

```
{{ var.value.<variable_name> }}
或
{{ var.json.<variable_name> }}
```

更好的隔离环境

1）动态构建DAG，根据获取的系统环境变量动态调整配置参数，从而确定当前的运行环境（例如：区分生产环境和开发环境）

```
deployment = os.environ.get("DEPLOYMENT", "PROD")
if deployment == "PROD":
    task = Operator(param="prod-param")
elif deployment == "DEV":
    task = Operator(param="dev-param")
```

2）依赖包管理

Airflow 在DAG开发和配置中可使用自己的Python模块，当在Airflow部署时提前将通用代码、代码库或者可共享的DAG就可以使用，这些模块体现在PYTHONPATH上。添加方式有模块、额外文件夹、代码包等。

- 查看sys.path。使用sys.path获取当前系统中的加载Package/modules

```
>>> import sys
>>> from pprint import pprint
>>> pprint(sys.path)
['',
 '/home/arch/.pyenv/versions/3.7.4/lib/python37.zip',
 '/home/arch/.pyenv/versions/3.7.4/lib/python3.7',
 '/home/arch/.pyenv/versions/3.7.4/lib/python3.7/lib-dynload',
 '/home/arch/venvs/airflow/lib/python3.7/site-packages']
```

- 在Python代码上直接添加，添加包后，Python 将开始在较新的路径中搜索包

```
sys.path.append("/path/to/custom/package")
```

- 典型的DAG结构，可通过.ariflowignore让Airflow忽略加载到DAG中

```
<DIRECTORY ON PYTHONPATH>
| .airflowignore  -- only needed in ``dags`` folder, see below
| -- my_company
               | __init__.py
               | common_package
               |              |  __init__.py
               |              | common_module.py
               |              | subpackage
               |                          | __init__.py
               |                          | subpackaged_util_module.py
               |
               | my_custom_dags
                               | __init__.py
                               | my_dag_1.py
                               | my_dag_2.py
                               | base_dag.py
```

- Airflow已内置dags、config、plugins到PYTHONPATH
- 模块管理最佳实践
  a. 在DAGS目录下使用顶级包名称，并且是唯一的（例如：mycompany），避免子模块的命名跟系统有冲突。
  b. 在DAGS代码中引用包时采用完整路径引用。
  c. 在Python的Package的每个文件中添加 __init__.py
- 通过airflow info查看**PYTHONPATH明细信息**
- 模块/Package还可以以**pip 包**的方式安装到Airflow环境

3）利用环境变量动态切换环境，避免在代码里硬编码。例如：使用动态的环境变量确定S3目标路径

```
import os
dest = os.environ.get("MY_DAG_DEST_PATH", "s3://default-target/path/")
```

4）使用jinjia模板动态替换，使用template_fields动态定义参数，确定环境参数。例如：根据跑批日期确定{{ds}}参数值，从而动态确定要加载的文件路径。

```
class MyDataReader:
    template_fields = ["path"]

    def __init__(self, my_path):
        self.path = my_path

    # [additional code here...]


t = PythonOperator(
    task_id="transform_data",
    python_callable=transform_data,
    op_args=[MyDataReader("/tmp/{{ ds }}/my_file")],
    dag=dag,
)
```

降低DAG复杂性

1）测试python时间复杂性

```
time python airflow/example_dags/example_python_operator.py
```

2）降低DAG任务上下游之间的依赖复杂性

3）单个Python文件中保持一个DAG定义

更加安全的配置

1）Secrets Backend，通过airflow.providers.amazon.aws.secrets.secrets_manager使用AWS Secrts Mangaer。

例如：配置cfg参数

```
[secrets]
backend = airflow.providers.amazon.aws.secrets.secrets_manager.SecretsManagerBac
backend_kwargs = {"connections_prefix": "airflow/connections", "region_name": "e
```

参考代码

```python
from airflow import DAG
from airflow.providers.amazon.aws.hooks.secrets_manager import SecretsManagerHoo
from airflow.models.base import Base
from airflow.utils.dates import days_ago

# Define your DAG
dag = DAG(
    'secrets_example',
    schedule_interval=None,  # Set to None to disable automatic scheduling
    start_date=days_ago(1),
)

def my_task_function(**kwargs):
    # Initialize the SecretsManagerHook
    hook = SecretsManagerHook(aws_conn_id='aws_default', region_name='eu-west-1'

    # Retrieve the secret value
    secret_value = hook.get_secret(secret_id='my_secret_name')

    # Perform your task logic using the secret value
    print("Secret value:", secret_value)

# Define a task using the task function
my_task = PythonOperator(
    task_id='my_task',
    python_callable=my_task_function,
    provide_context=True,
    dag=dag,
)
```

2）在webserver上配置DAGS目录存在潜在的安全风险，例如：通过编写DAG在webserver上执行某些代码。

数据/信息如何更好在DAG/TASK之间进行传递

1. **XComs（交流对象）**：XComs 是 Airflow 中用于在任务之间传递数据的机制。每个任务可以在运行时生成一个 XCom，然后其他任务可以读取这些 XComs 来获取数据。您可以通过任务实例的 `xcom_push` 和 `xcom_pull` 方法来操作 XComs。

2. **任务参数传递**：您可以在任务定义中将参数传递给任务。这些参数可以是普通的 Python 变量，可以在任务中使用。这种方式适用于传递静态数据。

3. **外部数据存储**：如果需要在任务之间传递大量数据，您可以将数据存储在外部数据存储中，例如数据库、文件系统或消息队列，然后在任务中读取这些数据。

4. **PythonOperator 的 provide_context 参数**：当使用 PythonOperator 时，可以设置参数 `provide_context=True`，这会将上下文变量传递给您的 Python 可调用函数。通过这种方式，您可以访问任务实例、执行日期、DAG 上下文等信息。

5. **DAG 运行时变量**：在任务之间共享信息时，您还可以考虑使用 DAG 运行时变量。这些变量可以在 DAG 运行时存储和读取数据，供任务之间共享。可以使用 `Variable` 类来处理这些运行时变量。

6. **全局变量**：在某些情况下，您可以使用全局变量来传递信息。这可能不是最佳实践，但在某些情况下可能是有效的方式。

更好地管理多套依赖环境

1. **使用虚拟环境（Virtual Environments）**：为每个依赖环境创建独立的 Python 虚拟环境。这可以帮助您隔离不同环境的依赖包，避免版本冲突。您可以使用工具如 `virtualenv` 或 `conda` 来创建和管理虚拟环境。

2. **DAG 参数化**：将 DAG 的参数化，以便在每个环境中重用相同的 DAG 逻辑。您可以将需要根据环境变化的参数抽象出来，以便在不同的环境中进行配置。

3. **使用配置文件**：将环境相关的配置参数存储在配置文件中，然后在 DAG 中根据环境读取这些配置。这可以帮助您在不同环境中使用不同的设置。

4. **使用变量（Variables）**：在 Airflow 中，您可以使用变量来存储和管理环境相关的参数。这些变量可以在 DAG 中共享，并且可以在 Airflow Web UI 中进行配置。

5. **条件判断**：在 DAG 中使用条件判断，根据不同的环境选择不同的任务路径或参数。您可以使用 `BranchPythonOperator` 或 `ShortCircuitOperator` 等操作来实现条件分支。

6. **使用 SubDags**：如果有多个 DAG 共享相同的一些任务，可以将这些任务封装为 SubDags，然后在不同的 DAG 中引用这些 SubDags，以实现任务的重用和管理。

7. **使用版本控制**：使用版本控制系统（如 Git）来管理不同环境的 DAG 定义和配置文件。这样可以确保在不同环境中使用相同的代码和配置。

8. **自定义 Operator 和 Hook**：如果需要在不同环境中执行不同的操作，可以考虑创建自定义的 Operator 或 Hook，以实现环境特定的逻辑。

参考资源

Amazon MWAA docs https://docs.aws.amazon.com/mwaa

Airflow Slack Channel: https://apache-airflow.slack.com

MWAA workshop: https://amazon-mwaa-for-analytics.workshop.aws/en/

MWAA local development and testing: https://github.com/aws/aws-mwaa-local-runner

Environment checker: https://github.com/awslabs/aws-support-tools/tree/master/MWAA

MWAA alerts and dashboard: https://aws.amazon.com/blogs/compute/automating-amazon-cloudwatch-dashboards-and-alarms-for-amazon-managed-workflows-for-apache-airflow/

https://docs.aws.amazon.com/mwaa/latest/userguide/sample-code.html

https://github.com/aws-samples/amazon-mwaa-examples