Airflow使用指南

日期：2019.06.27

问题导读

1.如何安装和配置 Airflow?
2.如何通过 Airflow UI 监控 data pipeline（管道）并对其进行故障排除
3.什么是 Airflow Platform？
4. Airflow 是如何进行数据分析，追踪数据，调试数据流的？
5. Airflow 命令行接口的基本操作有哪些？
6.什么是集成以及我们如何进行配置？

快速开始

安装是快速而直接的。

```
# airflow 需要 home 目录，默认是~/airflow，
# 但是如果你需要，放在其它位置也是可以的
# (可选)
export AIRFLOW_HOME = ~/airflow

# 使用 pip 从 pypi 安装
pip install apache-airflow

# 初始化数据库
airflow initdb

# 启动 web 服务器，默认端口是 8080
airflow webserver -p 8080

# 启动定时器
airflow scheduler

# 在浏览器中浏览 localhost:8080，并在 home 页开启 example dag
```

运行这些命令后，Airflow 将创建$AIRFLOW_HOME 文件夹，并放置一个 airflow.cfg 文件，其默认值可以让您快速上手。您可以通过查看$AIRFLOW_HOME/airflow.cfg 文件或者在 UI 的 Admin->Configuration 菜单中检查相关配置。如果由 systemd 启动，则 webserver 的 PID 文件将存储在$AIRFLOW_HOME/airflow-webserver.pid 或/run/airflow/webserver.pid。

开箱即用，Airflow 使用 sqlite 数据库，由于使用此数据库后端无法进行并行化，因此您应该迅速替换它。它与 SequentialExecutor 一起使用，但仅能按顺序运行任务实例。虽然这是非常有限的，但它允许您快速启动和运行并浏览 UI 和命令行实用程序。

以下是一些将触发一些任务实例的命令。在运行以下命令时，您应该能够在 example_bash_operatorDAG 中看到任务的状态发生变化。

```
# 运行第一个任务实例
airflow run example_bash_operator runme_0 2015-01-01
# 运行两天的任务回填
airflow backfill example_bash_operator -s 2015-01-01 -e 2015-01-02
```

安装

1.获取 Airflow：

安装最新稳定版 Airflow 的最简单方法是使用 pip ：

```
pip install apache-airflow
```

您还可以安装 Airflow 的一些别的支持功能组件，例如 gcp_api 或者 postgres：

```
pip install apache-airflow[postgres,gcp_api]
```

2.额外的扩展包：

通过 PyPI 的 apache-airflow 命令下载的基本包只含有启动的基础部分内容。您可以根据您环境的需要下载您的扩展包。例如，如果您不需要连接 Postgres，那么您就不需要使用 yum 命令安装 postgres-devel，或者在您使用的系统上面安装 postgre 应用，并在安装中的经历一些痛苦过程。

除此之外，Airflow 可以按照需求导入这些扩展包来使用。

如下是列举出来的子包列表和他的功能：

包名	安装命令	说明
all	pip install apache-airflow[all]	所有 Airflow 功能
all_dbs	pip install apache-airflow[all_dbs]	所有集成的数据库
async	pip install apache-airflow[async]	Gunicorn 的异步 worker classes
celery	pip install apache-airflow[celery]	CeleryExecutor
cloudant	pip install apache-airflow[cloudant]	Cloudant hook
crypto	pip install apache-airflow[crypto]	加密元数据 db 中的连接密码

devel	`pip install apache-airflow[devel]`	最小开发工具要求
devel_hadoop	`pip install apache-airflow[devel_hadoop]`	Airflow + Hadoop stack 的依赖
druid	`pip install apache-airflow[druid]`	Druid.io 相关的 operators 和 hooks
gcp_api	`pip install apache-airflow[gcp_api]`	Google 云平台 hooks 和 operators（使用 `google-api-python-client`）
github_enterprise	`pip install apache-airflow[github_enterprise]`	Github 企业版身份认证
google_auth	`pip install apache-airflow[google_auth]`	Google 身份认证
hdfs	`pip install apache-airflow[hdfs]`	HDFS hooks 和 operators
hive	`pip install apache-airflow[hive]`	所有 Hive 相关的 operators
jdbc	`pip install apache-airflow[jdbc]`	JDBC hooks 和 operators
kerberos	`pip install apache-airflow[kerberos]`	Kerberos 集成 Kerberized Hadoop
kubernetes	`pip install apache-airflow[kubernetes]`	Kubernetes Executor 以及 operator
ldap	`pip install apache-airflow[ldap]`	用户的 LDAP 身份验证
mssql	`pip install apache-airflow[mssql]`	Microsoft SQL Server operators 和 hook，作为 Airflow 后端支持
mysql	`pip install apache-airflow[mysql]`	MySQL operators 和 hook，支持作为 Airflow 后端。 MySQL 服务器的版本必须是 5.6.4+。确切的版本上限取决于 `mysqlclient` 包的版本。例

		如， mysqlclient 1.3.12 只能与 MySQL 服务器 5.6.4 到 5.7 一起使用。
password	pip install apache-airflow[password]	用户密码验证
postgres	pip install apache-airflow[postgres]	Postgres operators 和 hook，作为 Airflow 后端支持
qds	pip install apache-airflow[qds]	启用 QDS（Qubole 数据服务）支持
rabbitmq	pip install apache-airflow[rabbitmq]	rabbitmq 作为 Celery 后端支持
redis	pip install apache-airflow[redis]	Redis hooks 和 sensors
s3	pip install apache-airflow[s3]	S3KeySensor, S3PrefixSensor
samba	pip install apache-airflow[samba]	Hive2SambaOperator
slack	pip install apache-airflow[slack]	SlackAPIPostOperator
ssh	pip install apache-airflow[ssh]	SSH hooks 及 Operator
vertica	pip install apache-airflow[vertica]	做为 Airflow 后端的 Vertica hook 支持

3.初始化 Airflow 数据库：

在您运行任务之前，Airflow 需要初始化数据库。 如果您只是在试验和学习 Airflow，您可以坚持使用默认的 SQLite 选项。 如果您不想使用 SQLite，请查看初始化数据库后端以设置其他数据库。

配置完成后，若您想要运行任务，需要先初始化数据库：
`airflow initdb`

操作指南

1. 设置配置选项：

第一次运行 Airflow 时，它会在$AIRFLOW_HOME 目录中创建一个名为 airflow.cfg 的文件（默认情况下为 ~/airflow）。此文件包含 Airflow 的配置，您可以对其进行编辑以更改任何设置。您还可以使用以下格式设置带有环境变量的选项：$AIRFLOW__{SECTION}__{KEY} （注意使用双下划线）。

例如，元数据库连接字符串可以在 airflow.cfg 设置，如下所示：

```
[core]
sql_alchemy_conn = my_conn_string
```

或者通过创建相应的环境变量：

```
AIRFLOW__CORE__SQL_ALCHEMY_CONN = my_conn_string
```

您还可以通过将_cmd 附加到键来在运行时派生连接字符串，如下所示：

```
[core]
sql_alchemy_conn_cmd = bash_command_to_run
```

下列配置选项支持这种_cmd 配置办法

- [core]部分中的 sql_alchemy_conn
- [core]部分中的 fernet_key
- [celery]部分中的 broker_url
- [celery]部分中的 result_backend
- [atlas]部分中的 password
- [smtp]部分中的 smtp_password
- [ldap]部分中的 bind_password
- [kubernetes]部分中的 git_password

这背后的想法是不将密码存储在纯文本文件的框中。

所有配置选项的优先顺序如下 -

1. 环境变量
2. airflow.cfg 中的配置
3. airflow.cfg 中的命令
4. 默认

2.初始化数据库后端：

如果您想对 Airflow 进行真正的试使用，您应该考虑设置一个真正的数据库后端并切换到 LocalExecutor。

由于 Airflow 是使用优秀的 SqlAlchemy 库与其元数据进行交互而构建的，因此您可以使用任何 SqlAlchemy 所支持的数据库作为后端数据库。我们推荐使用 MySQL 或 Postgres。

注意：我们依赖更严格的 MySQL SQL 设置来获得合理的默认值。确保在[mysqld]下的 my.cnf 中指定了 explicit_defaults_for_timestamp = 1;

注意：如果您决定使用 Postgres，我们建议您使用 psycopg2 驱动程序并在 SqlAlchemy 连接字符串中指定它。另请注意，由于 SqlAlchemy 没有公开在 Postgres 连接 URI 中定位特定模式的方法，因此您可能需要使用类似于 ALTER ROLE username SET search_path = airflow, foobar;的命令为您的角色设置默认模式

一旦您设置好管理 Airflow 的数据库以后，您需要更改配置文件$AIRFLOW_HOME/airflow.cfg 中的 SqlAlchemy 连接字符串。然后，您还应该将“executor”设置更改为使用“LocalExecutor”，这是一个可以在本地并行化任务实例的执行程序。

```
# 初始化数据库
airflow initdb
```

3.使用 Operators（执行器）：

operator（执行器）代表一个理想情况下是幂等的任务。operator（执行器）决定了 DAG 运行时实际执行的内容。

（1）BashOperator

使用 BashOperator 在 Bash shell 中执行命令。

```
run_this = BashOperator(
    task_id='run_after_loop',
    bash_command='echo 1',
    dag=dag)
```

模板

您可以使用 Jinja 模板来参数化 bash_command 参数。

```
also_run_this = BashOperator(
    task_id='also_run_this',
    bash_command='echo "run_id={{ run_id }} | dag_run={{ dag_run }}"',
    dag=dag,
)
```

故障排除

(a)找不到 Jinja 模板

在使用 bash_command 参数直接调用 Bash 脚本时，需要在脚本名称后添加空格。这是因为 Airflow 尝试将 Jinja 模板应用于一个失败的脚本。

```
t2 = BashOperator(
    task_id='bash_example',

    # 这将会出现`Jinja template not found`的错误
    # bash_command="/home/batcher/test.sh",

    # 在加了空格之后，这会正常工作
    bash_command="/home/batcher/test.sh ",
```

```
    dag=dag)
```

（2）PythonOperator

使用 PythonOperator 执行 Python 回调。

```
def print_context ( ds , ** kwargs ):
    pprint ( kwargs )
    print ( ds )
    return 'Whatever you return gets printed in the logs'

run_this = PythonOperator (
    task_id = 'print_the_context' ,
    provide_context = True ,
    python_callable = print_context ,
    dag = dag )
```

传递参数

使用 op_args 和 op_kwargs 参数将额外参数传递给 Python 的回调函数。

```
def my_sleeping_function(random_base):
    """这是一个将在 DAG 执行体中运行的函数"""
    time.sleep(random_base)


# Generate 10 sleeping tasks, sleeping from 0 to 4 seconds respectively
for i in range(5):
    task = PythonOperator(
        task_id='sleep_for_' + str(i),
        python_callable=my_sleeping_function,
        op_kwargs={'random_base': float(i) / 10},
        dag=dag,
    )

    run_this >> task
```

模板

当您将 provide_context 参数设置为 True，Airflow 会传入一组额外的关键字参数：一个用于每个 Jinja 模板变量和一个 templates_dict 参数。

templates_dict 参数是模板化的，因此字典中的每个值都被评估为 Jinja 模板。

（3）Google 云平台 Operators（执行器）

GoogleCloudStorageToBigQueryOperator

使用 GoogleCloudStorageToBigQueryOperator 执行 BigQuery 加载作业。

GceInstanceStartOperator

允许启动一个已存在的 Google Compute Engine 实例。

在此示例中，参数值从 Airflow 变量中提取。此外，default_args 字典用于将公共参数传递给单个 DAG 中的所有 operator（执行器）。

```
PROJECT_ID = models.Variable.get('PROJECT_ID', '')
LOCATION = models.Variable.get('LOCATION', '')
INSTANCE = models.Variable.get('INSTANCE', '')
SHORT_MACHINE_TYPE_NAME = models.Variable.get('SHORT_MACHINE_TYPE_NAME', '')
SET_MACHINE_TYPE_BODY = {
    'machineType': 'zones/{}/machineTypes/{}'.format(LOCATION, SHORT_MACHINE_TYPE_NAME)
}

default_args = {
    'start_date': airflow.utils.dates.days_ago(1)
}
```

通过将所需的参数传递给构造函数来定义 GceInstanceStartOperator。

```
gce_instance_start = GceInstanceStartOperator(
    project_id=PROJECT_ID,
    zone=LOCATION,
    resource_id=INSTANCE,
    task_id='gcp_compute_start_task'
)
```

GceInstanceStopOperator

允许停止一个已存在的 Google Compute Engine 实例。

参数定义请参阅上面的 GceInstanceStartOperator。

通过将所需的参数传递给构造函数来定义 GceInstanceStopOperator。

```
gce_instance_stop = GceInstanceStopOperator(
    project_id=PROJECT_ID,
```

```
    zone=LOCATION,
    resource_id=INSTANCE,
    task_id='gcp_compute_stop_task'
)
```

GceSetMachineTypeOperator

允许把一个已停止实例的机器类型改变至特定的类型。

参数定义请参阅上面的 GceInstanceStartOperator。

通过将所需的参数传递给构造函数来定义 GceSetMachineTypeOperator。

```
gce_set_machine_type = GceSetMachineTypeOperator(
    project_id=PROJECT_ID,
    zone=LOCATION,
    resource_id=INSTANCE,
    body=SET_MACHINE_TYPE_BODY,
    task_id='gcp_compute_set_machine_type'
)
```

GcfFunctionDeleteOperator

使用 default_args 字典来传递参数给 operator（执行器）。

```
PROJECT_ID = models.Variable.get('PROJECT_ID', '')
LOCATION = models.Variable.get('LOCATION', '')
ENTRYPOINT = models.Variable.get('ENTRYPOINT', '')
# A fully-qualified name of the function to delete

FUNCTION_NAME = 'projects/{}/locations/{}/functions/{}'.format(PROJECT_ID, LOCATION,
                                                               ENTRYPOINT)
default_args = {
    'start_date': airflow.utils.dates.days_ago(1)
}
```

使用 GcfFunctionDeleteOperator 来从 Google Cloud Functions 删除一个函数。

```
t1 = GcfFunctionDeleteOperator(
    task_id="gcf_delete_task",
    name=FUNCTION_NAME
)
```

(a)故障排除

如果你想要使用服务账号来运行或部署一个 operator（执行器），但得到了一个 403 禁止的错误，这意味着你的服务账号没有正确的 Cloud IAM 权限。

1. 指定该服务账号为 Cloud Functions Developer 角色。
2. 授权 Cloud Functions 的运行账户为 Cloud IAM Service Account User 角色。

使用 gcloud 分配 Cloud IAM 权限的典型方法如下所示。只需将您的 Google Cloud Platform 项目 ID 替换为 PROJECT_ID，将 SERVICE_ACCOUNT_EMAIL 替换为您的服务帐户的电子邮件 ID 即可。

```
gcloud iam service-accounts add-iam-policy-binding \
  PROJECT_ID@appspot.gserviceaccount.com \
  --member="serviceAccount:[SERVICE_ACCOUNT_EMAIL]" \
  --role="roles/iam.serviceAccountUser"
```

GcfFunctionDeployOperator

使用 GcfFunctionDeployOperator 来从 Google Cloud Functions 部署一个函数。

以下 Airflow 变量示例显示了您可以使用的 default_args 的各种变体和组合。变量定义如下：

```
PROJECT_ID = models.Variable.get('PROJECT_ID', '')
LOCATION = models.Variable.get('LOCATION', '')
SOURCE_ARCHIVE_URL = models.Variable.get('SOURCE_ARCHIVE_URL', '')
SOURCE_UPLOAD_URL = models.Variable.get('SOURCE_UPLOAD_URL', '')
SOURCE_REPOSITORY = models.Variable.get('SOURCE_REPOSITORY', '')
ZIP_PATH = models.Variable.get('ZIP_PATH', '')
ENTRYPOINT = models.Variable.get('ENTRYPOINT', '')
FUNCTION_NAME = 'projects/{}/locations/{}/functions/{}'.format(PROJECT_ID, LOCATION,
                                                               ENTRYPOINT)
RUNTIME = 'nodejs6'
VALIDATE_BODY = models.Variable.get('VALIDATE_BODY', True)
```

使用这些变量，您可以定义请求的主体：

```
body = {
    "name": FUNCTION_NAME,
    "entryPoint": ENTRYPOINT,
    "runtime": RUNTIME,
    "httpsTrigger": {}
}
```

创建 DAG 时，default_args 字典可用于传递正文和其他参数：

```
default_args = {
```

```
    'start_date': dates.days_ago(1),
    'project_id': PROJECT_ID,
    'location': LOCATION,
    'body': body,
    'validate_body': VALIDATE_BODY
}
```

请注意，在上面的示例中，body 和 default_args 都是不完整的。根据设置的变量，如何传递源代码相关字段可能有不同的变体。目前，您可以传递 sourceArchiveUrl，sourceRepository 或 sourceUploadUrl，CloudFunction API 规范中所述。此外，default_args 可能包含 zip_path 参数，以在部署源代码之前运行上载源代码的额外步骤。在最后一种情况下，您还需要在正文中提供一个空的 sourceUploadUrl 参数。

基于上面定义的变量，此处显示了设置源代码相关字段的示例逻辑：

```
if SOURCE_ARCHIVE_URL:
    body['sourceArchiveUrl'] = SOURCE_ARCHIVE_URL
elif SOURCE_REPOSITORY:
    body['sourceRepository'] = {
        'url': SOURCE_REPOSITORY
    }
elif ZIP_PATH:
    body['sourceUploadUrl'] = ''
    default_args['zip_path'] = ZIP_PATH
elif SOURCE_UPLOAD_URL:
    body['sourceUploadUrl'] = SOURCE_UPLOAD_URL
else:
    raise Exception("Please provide one of the source_code parameters")
```

创建 operator（执行器）的代码如下：

```
deploy_task = GcfFunctionDeployOperator(
    task_id="gcf_deploy_task",
    name=FUNCTION_NAME
)
```

Troubleshooting

如果你想要使用服务账号来运行或部署一个 operator（执行器），但得到了一个 403 禁止的错误，这意味着你的服务账号没有正确的 Cloud IAM 权限。

1. 指定该服务账号为 Cloud Functions Developer 角色。
2. 授权 Cloud Functions 的运行账户为 Cloud IAM Service Account User 角色。

使用 gcloud 分配 Cloud IAM 权限的典型方法如下所示。只需将您的 Google Cloud Platform 项目 ID 替

换为 PROJECT_ID，将 SERVICE_ACCOUNT_EMAIL 的替换为您的服务帐户的电子邮件 ID 即可。

```
gcloud iam service-accounts add-iam-policy-binding \
  PROJECT_ID@appspot.gserviceaccount.com \
  --member="serviceAccount:[SERVICE_ACCOUNT_EMAIL]" \
  --role="roles/iam.serviceAccountUser"
```

如果您的函数的源代码位于 Google Source Repository 中，请确保您的服务帐户具有 Source Repository Viewer 角色，以便在必要时可以下载源代码。

CloudSqlInstanceDatabaseCreateOperator

在 Cloud SQL 实例中创建新数据库。

有关参数定义，请参阅上面的 GceInstanceStartOperator。

通过将所需的参数传递给构造函数来定义 CloudSqlInstanceDatabaseCreateOperator。

(a)参数

示例 DAG 中的一些参数取自环境变量：

```
PROJECT_ID = os.environ.get('PROJECT_ID', 'example-project')
INSTANCE_NAME = os.environ.get('INSTANCE_NAME', 'testinstance')
DB_NAME = os.environ.get('DB_NAME', 'testdb')
```

(b)使用 operator（执行器）

```
sql_db_create_task = CloudSqlInstanceDatabaseCreateOperator(
    project_id=PROJECT_ID,
    body=db_create_body,
    instance=INSTANCE_NAME,
    task_id='sql_db_create_task'
)
```

示例请求体：

```
db_create_body = {
    "instance": INSTANCE_NAME,
    "name": DB_NAME,
    "project": PROJECT_ID
}
```

(c)模版

```
template_fields = ('project_id', 'instance', 'gcp_conn_id', 'api_version')
```

CloudSqlInstanceDatabaseDeleteOperator

在 Cloud SQL 实例中删除数据库。

有关参数定义，请参阅 CloudSqlInstanceDatabaseDeleteOperator。

(a)参数

示例 DAG 中的一些参数取自环境变量：

```
PROJECT_ID = os.environ.get('PROJECT_ID', 'example-project')
INSTANCE_NAME = os.environ.get('INSTANCE_NAME', 'testinstance')
DB_NAME = os.environ.get('DB_NAME', 'testdb')
```

(b)使用 operator（执行器）

```
sql_db_delete_task = CloudSqlInstanceDatabaseDeleteOperator(
    project_id=PROJECT_ID,
    instance=INSTANCE_NAME,
    database=DB_NAME,
    task_id='sql_db_delete_task'
)
```

(c)模版

```
template_fields = ('project_id', 'instance', 'database', 'gcp_conn_id',
                   'api_version')
```

CloudSqlInstanceDatabasePatchOperator

(a)参数

示例 DAG 中的一些参数取自环境变量：

```
PROJECT_ID = os.environ.get('PROJECT_ID', 'example-project')
INSTANCE_NAME = os.environ.get('INSTANCE_NAME', 'testinstance')
DB_NAME = os.environ.get('DB_NAME', 'testdb')
```

(b)使用 operator（执行器）

```
sql_db_patch_task = CloudSqlInstanceDatabasePatchOperator(
```

```
    project_id=PROJECT_ID,
    body=db_patch_body,
    instance=INSTANCE_NAME,
    database=DB_NAME,
    task_id='sql_db_patch_task'
)
```

示例请求体：

```
db_patch_body = {
    "charset": "utf16",
    "collation": "utf16_general_ci"
}
```

(c)模版

```
template_fields = ('project_id', 'instance', 'database', 'gcp_conn_id',
                   'api_version')
```

CloudSqlInstanceDeleteOperator

示例 DAG 中的一些参数取自环境变量：

```
PROJECT_ID = os.environ.get('PROJECT_ID', 'example-project')
INSTANCE_NAME = os.environ.get('INSTANCE_NAME', 'testinstance')
DB_NAME = os.environ.get('DB_NAME', 'testdb')
```

(a)使用 operator（执行器）

```
sql_instance_delete_task = CloudSqlInstanceDeleteOperator(
    project_id=PROJECT_ID,
    instance=INSTANCE_NAME,
    task_id='sql_instance_delete_task'
)
```

(b)模版

```
template_fields = ('project_id', 'instance', 'gcp_conn_id', 'api_version')
```

CloudSqlInstanceCreateOperator

在 Google Cloud Platform 中创建新的 Cloud SQL 实例。

如果存在具有相同名称的实例，则不会执行任何操作，并且 operator（执行器）将成功执行。

(a)参数

示例 DAG 中的一些参数取自环境变量：

```
PROJECT_ID = os.environ.get('PROJECT_ID', 'example-project')
INSTANCE_NAME = os.environ.get('INSTANCE_NAME', 'testinstance')
DB_NAME = os.environ.get('DB_NAME', 'testdb')
定义实例的示例：
body = {
    "name": INSTANCE_NAME,
    "settings": {
        "tier": "db-n1-standard-1",
        "backupConfiguration": {
            "binaryLogEnabled": True,
            "enabled": True,
            "startTime": "05:00"
        },
        "activationPolicy": "ALWAYS",
        "dataDiskSizeGb": 30,
        "dataDiskType": "PD_SSD",
        "databaseFlags": [],
        "ipConfiguration": {
            "ipv4Enabled": True,
            "requireSsl": True,
        },
        "locationPreference": {
            "zone": "europe-west4-a"
        },
        "maintenanceWindow": {
            "hour": 5,
            "day": 7,
            "updateTrack": "canary"
        },
        "pricingPlan": "PER_USE",
        "replicationType": "ASYNCHRONOUS",
        "storageAutoResize": False,
        "storageAutoResizeLimit": 0,
        "userLabels": {
            "my-key": "my-value"
        }
    },
    "databaseVersion": "MYSQL_5_7",
    "region": "europe-west4",
```

```
}
```

(b)使用 operator（执行器）

```
sql_instance_create_task = CloudSqlInstanceCreateOperator(
    project_id=PROJECT_ID,
    body=body,
    instance=INSTANCE_NAME,
    task_id='sql_instance_create_task'
)
```

(c)模版

```
template_fields = ('project_id', 'instance', 'gcp_conn_id', 'api_version')
```

CloudSqlInstancePatchOperator

(a)参数

示例 DAG 中的一些参数取自环境变量：

```
PROJECT_ID = os.environ.get('PROJECT_ID', 'example-project')
INSTANCE_NAME = os.environ.get('INSTANCE_NAME', 'testinstance')
DB_NAME = os.environ.get('DB_NAME', 'testdb')
定义实例的示例：
patch_body = {
    "name": INSTANCE_NAME,
    "settings": {
        "dataDiskSizeGb": 35,
        "maintenanceWindow": {
            "hour": 3,
            "day": 6,
            "updateTrack": "canary"
        },
        "userLabels": {
            "my-key-patch": "my-value-patch"
        }
    }
}
```

(b)使用 operator（执行器）

```
sql_instance_patch_task = CloudSqlInstancePatchOperator(
    project_id=PROJECT_ID,
```

```
    body=patch_body,
    instance=INSTANCE_NAME,
    task_id='sql_instance_patch_task'
)
```

(c)模版

```
template_fields = ('project_id', 'instance', 'gcp_conn_id', 'api_version')
```

4.管理连接

Airflow 需要知道如何连接到您的环境。其他系统和服务的主机名，端口，登录名和密码等信息在 UI 的 Admin->Connection 部分中处理。您编写的 pipeline(管道)代码将引用 Connection 对象的“conn_id”。

可以使用 UI 或环境变量创建和管理连接。

(1)使用 UI 创建连接

打开 UI 的 Admin->Connection 部分。 单击 Create 按钮以创建新连接。

1. 使用所需的连接 ID 填写 Conn Id 字段。建议您使用小写字符和单独的带下划线的单词。
2. 使用 Conn Type 字段选择连接类型。
3. 填写其余字段。有关属于不同连接类型的字段的说明，请参阅连接类型。
4. 单击 Save 按钮以创建连接。

(2)使用 UI 编辑连接

打开 UI 的 Admin->Connection 部分。 单击连接列表中要编辑的连接旁边的铅笔图标。

修改连接属性，然后单击 Save 按钮以保存更改。

(3)使用环境变量创建连接

可以使用环境变量创建 Airflow pipeline（管道）中的连接。环境变量需要具有 AIRFLOW_CONN_的前缀的 URI 格式才能正确使用连接。

在引用 Airflow pipeline(管道)中的连接时，conn_id 应该是没有前缀的变量的名称。例如，如果 conn_id 名为 postgres_master，则环境变量应命名为 AIRFLOW_CONN_POSTGRES_MASTER（请注意，环境变量必须全部为大写）。 Airflow 假定环境变量返回的值为 URI 格式（例如 postgres://user:password@localhost:5432/master 或 s3://accesskey:secretkey@S3）。

(4)连接类型

Google Cloud Platform

Google Cloud Platform 连接类型支持 GCP 集成。

对 GCP 进行身份验证

有两种方法可以使用 Airflow 连接到 GCP。

1. 使用应用程序默认凭据，例如在 Google Compute Engine 上运行时通过元数据服务器。
2. 在磁盘上使用服务帐户密钥文件（JSON 格式）。

默认连接 ID

默认情况下使用以下连接 ID。

`bigquery_default`

由 BigQueryHook 钩子使用。

`google_cloud_datastore_default`

由 DatastoreHook 钩子使用。

`google_cloud_default`

由 GoogleCloudBaseHook，DataFlowHook，DataProcHook，MLEngineHook 和 GoogleCloudStorageHook 挂钩使用。

配置连接

Project Id（必填）

要连接的 Google Cloud 项目 ID。

Keyfile Path

磁盘上**服务帐户**密钥文件（JSON 格式）的路径。

如果使用应用程序默认凭据则不需要

Keyfile JSON

磁盘上的**服务帐户**密钥文件（JSON 格式）的内容。 如果使用此方法进行身份验证，建议**保护您的连接** 。
如果使用应用程序默认凭据则不需要

Scopes（逗号分隔）

要通过身份验证的逗号分隔 **Google 云端范围**列表。

注意 使用应用程序默认凭据时，将忽略范围。 请参阅 AIRFLOW-2522 。

MySQL

MySQL 连接类型允许连接 MySQL 数据库。

配置连接

主机（必填）
要连接的主机。

模式（选填）
指定要在数据库中使用的模式名称。

登陆（必填）
指定要连接的用户名

密码（必填）
指定要连接的密码

额外参数（选填）
指定可以在 mysql 连接中使用的额外参数（作为 json 字典）。支持以下参数：

- charset：指定连接的 charset
- cursor：指定使用的 cursor，为“sscursor”、“dictcurssor”、“ssdictcursor”其中之一
- local_infile：控制 MySQL 的 LOCAL 功能（允许客户端加载本地数据）。有关详细信息，请参阅 **MySQLdb 文档**。
- unix_socket：使用 UNIX 套接字而不是默认套接字
- ssl:控制使用 SSL 连接的 SSL 参数字典(这些参数是特定于服务器的,应包含“ca”,“cert”,“key”,“capath”,“cipher”参数。有关详细信息，请参阅 **MySQLdb 文档**。请注意，以便在 URL 表示法中很有用，此参数也可以是 SSL 字典是字符串编码的 JSON 字典的字符串。

示例“extras”字段：

```
{
  "charset": "utf8",
  "cursorclass": "sscursor",
  "local_infile": true,
  "unix_socket": "/var/socket",
  "ssl": {
    "cert": "/tmp/client-cert.pem",
    "ca": "/tmp/server-ca.pem'",
    "key": "/tmp/client-key.pem"
  }
}
```

或

```
{
  "charset": "utf8",
  "cursorclass": "sscursor",
  "local_infile": true,
  "unix_socket": "/var/socket",
  "ssl":  "{\"cert\":  \"/tmp/client-cert.pem\",  \"ca\":  \"/tmp/server-ca.pem\",  \"key\":
```

```
\"/tmp/client-key.pem\"}"
}
```

将连接指定为 URI（在 AIRFLOW_CONN_ *变量中）时，应按照 DB 连接的标准语法指定它，其中附加项作为 URI 的参数传递（请注意，URI 的所有组件都应进行 URL 编码）。
举个例子：

```
mysql://mysql_user:XXXXXXXXXXXX@1.1.1.1:3306/mysqldb?ssl=%7B%22cert%22%3A+%22%2Ftmp%2Fclient-cert.pem%22%2C+%22ca%22%3A+%22%2Ftmp%2Fserver-ca.pem%22%2C+%22key%22%3A+%22%2Ftmp%2Fclient-key.pem%22%7D
```

注意 如果在使用 MySQL 连接时遇到 UnicodeDecodeError，请检查定义的字符集是否与数据库字符集匹配。

5.保护连接

默认情况下，Airflow 在元数据数据库中是以纯文本格式保存连接的密码。在安装过程中强烈建议使用 crypto 包。crypto 包要求您的操作系统安装了 libffi-dev。

如果最初未安装 crypto 软件包，您仍可以通过以下步骤为连接启用加密：
1.安装 crypto 包 pip install apache-airflow[crypto]
2.使用下面的代码片段生成 fernet_key。 fernet_key 必须是 base64 编码的 32 字节密钥。

```
from cryptography.fernet import Fernet
fernet_key= Fernet.generate_key()
print(fernet_key.decode()) # 你的 fernet_key，把它放在安全的地方
```

3.将 airflow.cfg 中的 fernet_key 值替换为步骤 2 中的值。或者，可以将 fernet_key 存储在 OS 环境变量中。在这种情况下，您不需要更改 airflow.cfg，因为 Airflow 将使用环境变量而不是 airflow.cfg 中的值：

```
# 注意双下划线
export AIRFLOW__CORE__FERNET_KEY=your_fernet_key
```

1. 重启 Airflow webserver。
2. 对于现有连接（在安装 airflow[crypto]和创建 Fernet 密钥之前已定义的连接），您需要在连接管理 UI 中打开每个连接，重新键入密码并保存。

6.写日志

（1）在本地写日志

用户可以使用在 airflow.cfg 中的 base_log_folder 指定日志文件夹。默认情况下，它位于 AIRFLOW_HOME 目录中。

此外，用户也可以提供远程位置，以便在云存储中存储日志和日志备份。

在 Airflow Web UI 中，本地日志优先于远程日志。 如果找不到或访问本地日志，将显示远程日志。 请注意，只有在任务完成（包括失败）后才会将日志发送到远程存储。 换句话说，运行任务的远程日志不可

用。日志以{dag_id}/{task_id}/{execution_date}/{try_number}.log 的路径格式存储在日志文件夹中。

（2）将日志写入 Amazon S3

在您开始之前

远程日志记录使用现有的 Airflow 连接来读取/写入日志。如果没有正确设置连接，则会失败。

启用远程日志记录

要启用此功能，必须按照此示例配置 airflow.cfg ：

```
[core]
# Airflow 可以在 AWS S3 中远程存储日志。用户必须提供一个远程地址的 URL（以's3://...'开始）
# 和一个提供对存储位置访问的 Airflow 连接 id
remote_base_log_folder = s3://my-bucket/path/to/logs
remote_log_conn_id = MyS3Conn
# 对存储在 S3 中的日志使用服务器端加密
encrypt_s3_logs = False
```

在上面的例子中，Airflow 将尝试使用 S3Hook('MyS3Conn') 。

（3）将日志写入 Azure Blob Storage

在 Azure Blob Storage 中 Airflow 可以为配置读取和写入任务日志。 可以按照以下步骤启用 Azure Blob Storage 日志记录。

1. Airflow 的日志记录系统需要将一个自定义的 .py 文件 放在 PYTHONPATH，以便可以从 Airflow 导入。首先创建一个存储配置文件的目录。建议使用$AIRFLOW_HOME/config。
2. 创建名为$AIRFLOW_HOME/config/log_config.py 和$AIRFLOW_HOME/config/__init__.py 的空文件。
3. 将 airflow/config_templates/airflow_local_settings.py 的内容复制到刚刚在上面的步骤中创建的 log_config.py 文件中。
4. 自定义模板的以下部分：

```
# wasb buckets 应该从“wasb”开始，以帮助 Airflow 选择正确的处理程序
REMOTE_BASE_LOG_FOLDER = 'wasb-<whatever you want here>'

# 重命名 DEFAULT_LOGGING_CONFIG 为 LOGGING CONFIG
LOGGING_CONFIG = ...
```

5. 确保已在 Airflow 中定义 Azure Blob Storage（Wasb）的连接 hook。hook 应具有在 REMOTE_BASE_LOG_FOLDER 中定义的 Azure Blob Storage bucket 的读写访问权限。
6. 更新$AIRFLOW_HOME/airflow.cfg 以包含：

```
remote_logging = True
logging_config_class = log_config.LOGGING_CONFIG
remote_log_conn_id = <name of the Azure Blob Storage connection>
```

7. 重新启动 Airflow webserver 和 scheduler，并触发（或等待）新任务执行。
8. 验证日志是否显示在您定义的 bucket 中新执行的任务中。

（4）将日志写入 Google Cloud Storage

请按照以下步骤启用 Google Cloud Storage 日志记录。

要启用此功能，必须按照此示例配置 airflow.cfg：

```
[core]
# Airflow 可以在 AWS S3, Google Cloud Storage 或者 Elastic Search 中远程存储日志.
# 用户必须提供可访问存储位置的 Airflow 连接 id。
# 如果 remote_logging 被设置为 true，请参见 UPDATING.md 以查看其他相关配置的要求。
remote_logging = True
remote_base_log_folder = gs://my-bucket/path/to/logs
remote_log_conn_id = MyGCSConn
```

1. 首先安装 gcp_api 安装包，pip install apache-airflow[gcp_api]。
2. 确保已在 Airflow 中定义了 Google Cloud Platform 连接 hook。该 hook 应具有对 remote_base_log_folder 中上面定义的 Google Cloud Storage bucket 的读写访问权限。
3. 重新启动 Airflow webserver 和 scheduler，并触发（或等待）新任务执行。
4. 验证日志是否在您定义的 bucket 中显示新执行的任务日志。
5. 确认 Google Cloud Storage 查看器在 UI 中正常运行。拉出新执行的任务，并验证您是否看到类似的内容

```
*** Reading remote log from gs://<bucket where logs should be persisted>/example_bash_operator/run_this_last/2017-10-03T00:00:00/16.log.
[2017-10-03 21:57:50,056] {cli.py:377} INFO - Running on host chrisr-00532
[2017-10-03 21:57:50,093] {base_task_runner.py:115} INFO - Running: ['bash', '-c', u'airflow run example_bash_operator run_this_last 2017-10-03T00:00:00 --job_id 47 --raw -sd DAGS_FOLDER/example_dags/example_bash_operator.py']
[2017-10-03 21:57:51,264] {base_task_runner.py:98} INFO - Subtask: [2017-10-03 21:57:51,263] {__init__.py:45} INFO - Using executor SequentialExecutor
[2017-10-03 21:57:51,306] {base_task_runner.py:98} INFO - Subtask: [2017-10-03 21:57:51,306] {models.py:186} INFO - Filling up the DagBag from /airflow/dags/example_dags/example_bash_operator.py
```

请注意项行说明了它是从远程日志文件中读取的。

7.使用 Celery 扩大规模

CeleryExecutor 是您扩展 worker 数量的方法之一。为此，您需要设置 Celery 后端（**RabbitMQ,Redis,...**）并更改 airflow.cfg 以将执行程序参数指向 CeleryExecutor 并提供相关的 Celery 设置。

以下是您的 workers 的一些必要要求：

- 需要安装 airflow，CLI 需要在路径中
- 整个集群中的 Airflow 配置设置应该是同构的
- 在 worker 上执行的 Operators（执行器）需要在该上下文中满足其依赖项。例如，如果您使用 HiveOperator，则需要在该框上安装 hive CLI，或者如果您使用 MySqlOperator，则必须以某种方式在 PYTHONPATH 提供所需的 Python 库
- workers 需要访问其 DAGS_FOLDER 的权限，您需要通过自己的方式同步文件系统。常见的设置是将 DAGS_FOLDER 存储在 Git 存储库中，并使用 Chef，Puppet，Ansible 或用于配置环境中的计算机的任何内容在计算机之间进行同步。如果您的所有盒子都有一个共同的挂载点，那么共享您的管道文件也应该可以正常工作

要启动 worker，您需要设置 Airflow 并启动 worker 子命令

```
airflow worker
```

您的 worker 一旦启动就应该开始接收任务。

请注意,您还可以运行“Celery Flower”,这是一个建立在 Celery 之上的 Web UI,用于监控您的 worker。您可以使用快捷命令 airflow flower 启动 Flower Web 服务器。

一些警告：

- 确保使用数据库来作为 result backend（Celery result_backend，celery 的后台存储数据库）的后台存储
- 确保在[celery_broker_transport_options]中设置超过最长运行任务的 ETA 的可见性超时
- 任务会消耗资源，请确保您的 worker 有足够的资源来运行 worker_concurrency 任务

8.使用 Dask 扩展

DaskExecutor 允许您在 Dask 分布式集群中运行 Airflow 任务。

Dask 集群可以在单个机器上运行，也可以在远程网络上运行。有关完整详细信息，请参阅**分布式文档** 。
要创建集群，首先启动一个 Scheduler：

```
# 一个本地集群的默认设置
DASK_HOST=127.0.0.1
DASK_PORT=8786

dask-scheduler --host $DASK_HOST --port $DASK_PORT
```

接下来，在任何可以连接到主机的计算机上启动至少一个 Worker：

```
dask-worker $DASK_HOST:$DASK_PORT
```

编辑 airflow.cfg 以将执行程序设置为 DaskExecutor 并在[dask]部分中提供 Dask Scheduler 地址。

请注意：

- 每个 Dask worker 必须能够导入 Airflow 和您需要的任何依赖项。
- Dask 不支持队列。如果使用队列创建了 Airflow 任务，则会引发警告，但该任务会被提交给集

群。

9.使用 Mesos 扩展（社区贡献）

有两种方法可以将 Airflow 作为 mesos 框架运行：

1. 想要直接在 mesos slaves 上运行 Airflow 任务，要求每个 mesos slave 安装和配置 Airflow。
2. 在安装了 Airflow 的 docker 容器内运行 Airflow 任务，该容器在 mesos slave 上运行。

任务直接在 mesos slave 上执行

MesosExecutor 允许您在 Mesos 群集上调度 Airflow 任务。为此，您需要一个正在运行的 mesos 集群，并且必须执行以下步骤 -

1. 在可以运行 web server 和 scheduler 的 mesos slave 上安装 Airflow，让我们将其称为“Airflow server”。
2. 在 Airflow server 上，从**mesos 下载**安装 mesos python eggs。
3. 在 Airflow server 上，使用可以被所有 mesos slave 访问的数据库（例如 mysql），并在 airflow.cfg 添加配置。
4. 将您的 airflow.cfg 里 executor 的参数指定为 *MesosExecutor*，并提供相关的 Mesos 设置。
5. 在所有 mesos slave 上，安装 Airflow。 从 Airflow 服务器复制 airflow.cfg （以便它使用相同的 sqlalchemy 连接）。
6. 在所有 mesos slave 上，运行以下服务日志：

```
airflow serve_logs
```

7. 在 Airflow server 上，如果想要开始在 mesos 上处理/调度 DAG，请运行：

```
airflow scheduler -p
```

注意：我们需要-p 参数来挑选 DAG。

您现在可以在 mesos UI 中查看 Airflow 框架和相应的任务。Airflow 任务的日志可以像往常一样在 Airflow UI 中查看。

在 mesos slave 上的容器中执行的任务

此 gist 包含实现以下所需的所有文件和配置更改：

1. 使用安装了 mesos python eggs 的软件环境创建一个 dockerized 版本的 Airflow。

我们建议利用 docker 的多阶段构建来实现这一目标。我们有一个 Dockerfile 定义从源（Dockerfile-mesos）构建特定版本的 mesos，以便创建 python eggs。在 Airflow Dockerfile（Dockerfile-airflow）中，我们从 mesos 镜像中复制 python eggs。

2. 在 airflow.cfg 创建一个 mesos 配置块。

配置块保持与默认 Airflow 配置(default_airflow.cfg)相同，但添加了一个选项 docker_image_slave。这应该设置为您希望 mesos 在运行 Airflow 任务时使用的镜像的名称。确保您具有适用于您的 mesos 主服务器的 DNS 记录的正确配置以及任何类型的授权（如果存在）。

3. 更改 airflow.cfg 以将执行程序参数指向 MesosExecutor（executor = SequentialExecutor）。

4. 确保您的 mesos slave 可以访问您 docker_image_slave 的 docker 存储库。

其余部分取决于您以及您希望如何使用 dockerized Airflow 配置。

10.使用 systemd 运行 Airflow

Airflow 可以与基于 systemd 的系统集成。这使得观察您的守护进程变得容易，因为 systemd 可以在失败时重新启动守护进程。在 scripts/systemd 目录中，您可以找到已在基于 Redhat 的系统上测试过的单元文件。您可以将它们复制到/usr/lib/systemd/system。这是基于假设 Airflow 将在 airflow:airflow 用户/组下运行。如果不是（或者如果您在非基于 Redhat 的系统上运行），则可能需要调整单元文件。

从/etc/sysconfig/airflow 获取环境配置。提供了一个示例文件。 运行调度程序时，请确保在此文件中指定 SCHEDULER_RUNS 变量。您也可以在此处定义，例如 AIRFLOW_HOME 或 AIRFLOW_CONFIG。

11.使用 upstart 运行 Airflow

Airflow 可以与基于 upstart 的系统集成。Upstart 会在系统启动时自动启动在/etc/init 具有相应*.conf 文件的所有 Airflow 服务。失败时，upstart 会自动重启进程（直到达到*.conf 文件中设置的重启限制）。

您可以在 scripts/upstart 目录中找到示例 upstart 作业文件。这些文件已在 Ubuntu 14.04 LTS 上测试过。您可能需要调整 start on 和 stop on 设置，以使其适用于其他 upstart 系统。scripts/upstart/README 中列出了一些可能的选项。

您可以根据需要修改*.conf 文件并将它们复制到/etc/init 目录。这是基于假设 Airflow 将在 airflow:airflow 用户/组下运行的情况。如果您使用其他用户/组，请在*.conf 文件中更改 setuid 和 setgid。

您可以使用 initctl 手动启动，停止，查看已与 upstart 集成的 Airflow 过程的状态

```
initctl airflow-webserver status
```

12.使用测试模式配置

Airflow 具有一组固定的“test mode”配置选项。您可以随时通过调用 airflow.configuration.load_test_config()来加载它们（注意此操作不可逆！）。但是，在您有机会调用 load_test_config()之前，会加载一些选项（如 DAG_FOLDER）。为了更快加载测试配置，请在 airflow.cfg 中设置 test_mode：

```
[tests]
unit_test_mode = True
```

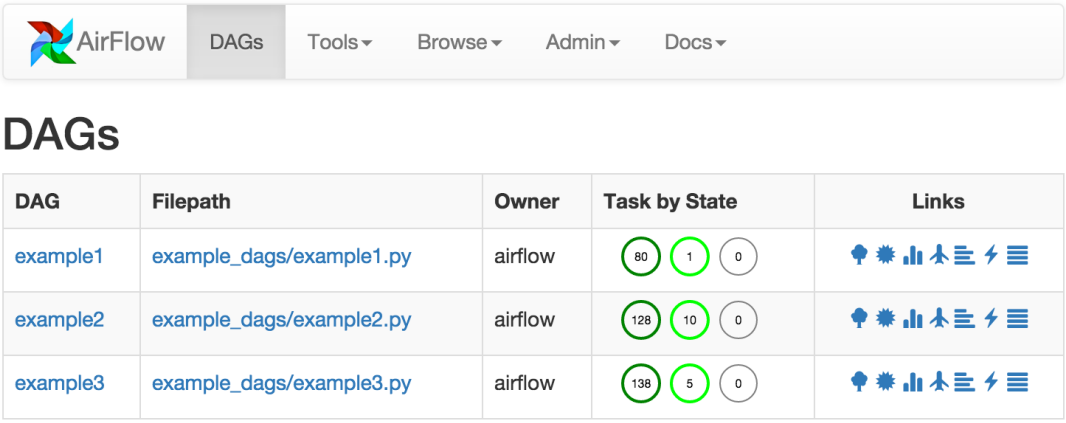
由于 Airflow 的自动环境变量扩展（请参阅设置配置选项），您还可以设置环境变量 AIRFLOW__CORE__UNIT_TEST_MODE 以临时覆盖 airflow.cfg。

UI/截图

通过 Airflow UI，您可以轻松监控 data pipeline（管道）并对其进行故障排除。如下是一些特性的简单预览和一些您可以在 Airflow 的 UI 中找到的可视化效果。

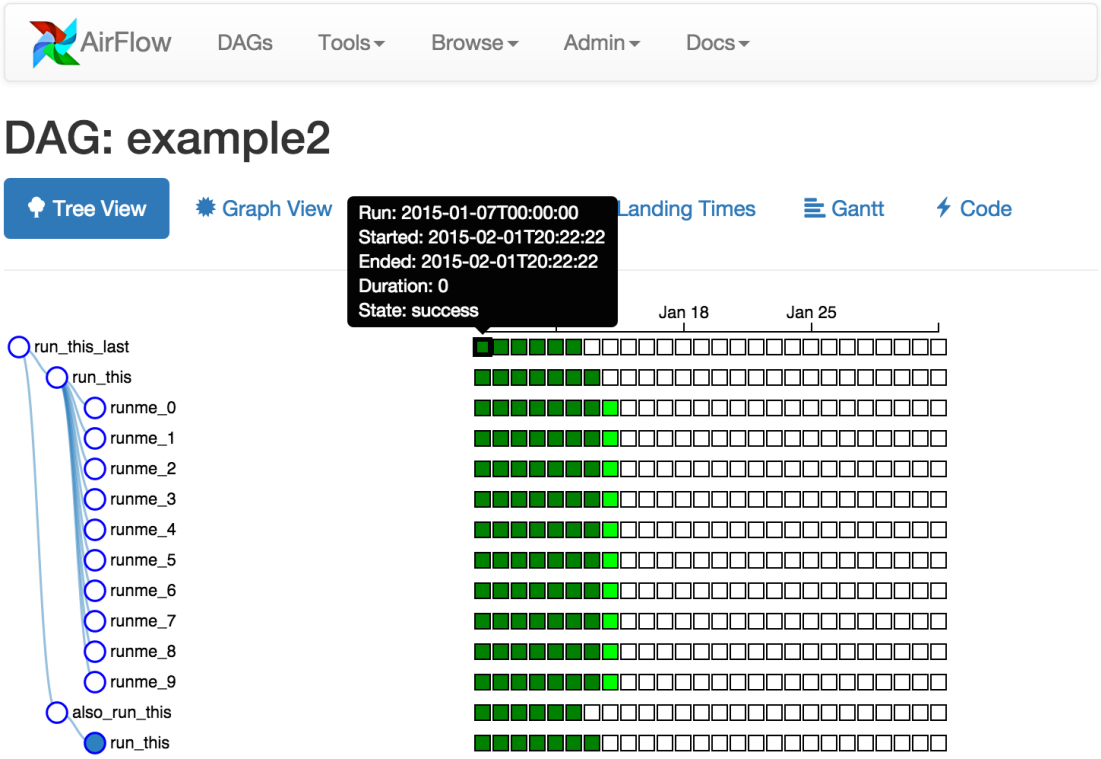
1.DAGs 查看

您环境中的 DAG 列表，以及一系列进入常用页面的快捷方式。您可以一目了然地查看成功、失败及当前正在运行的任务数量。



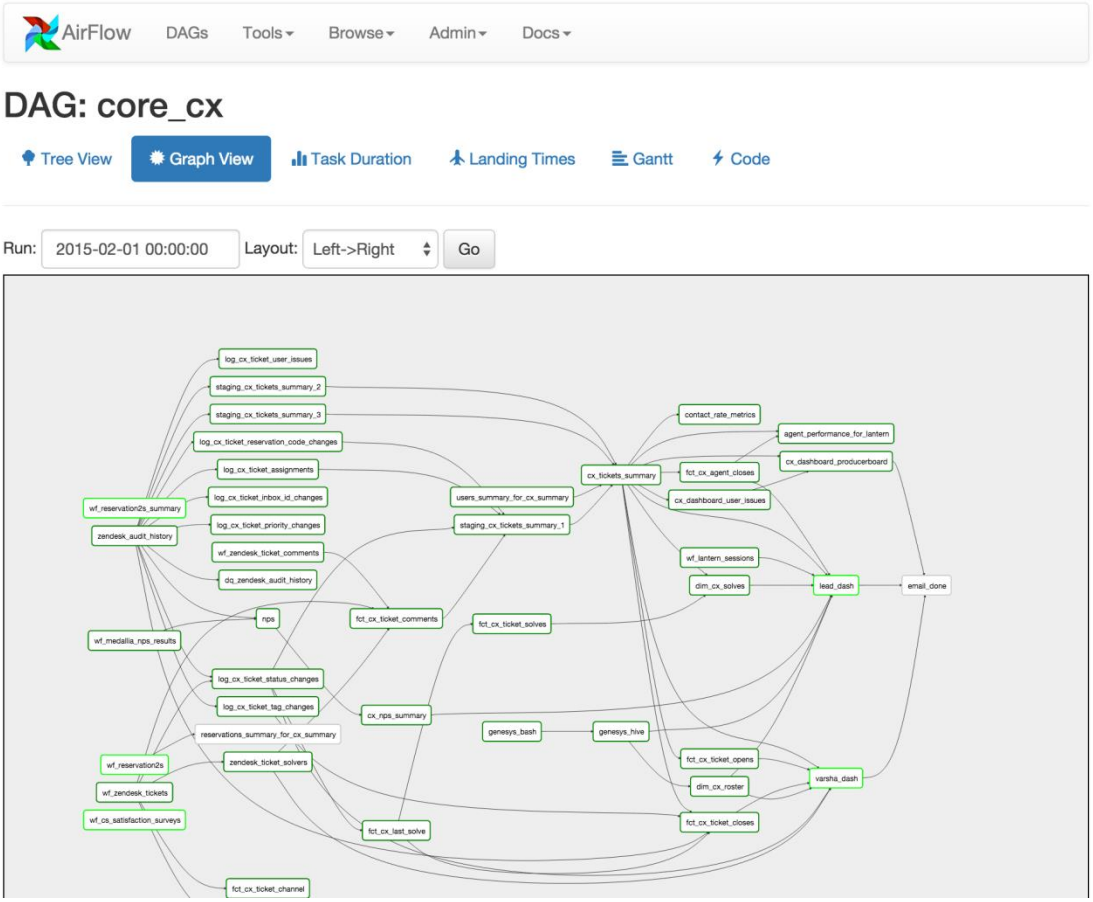
2.树视图

跨越时间的 DAG 的树表示。如果 pipeline（管道）延迟了，您可以很快地看到哪里出现了错误的步骤并且辨别出堵塞的进程。



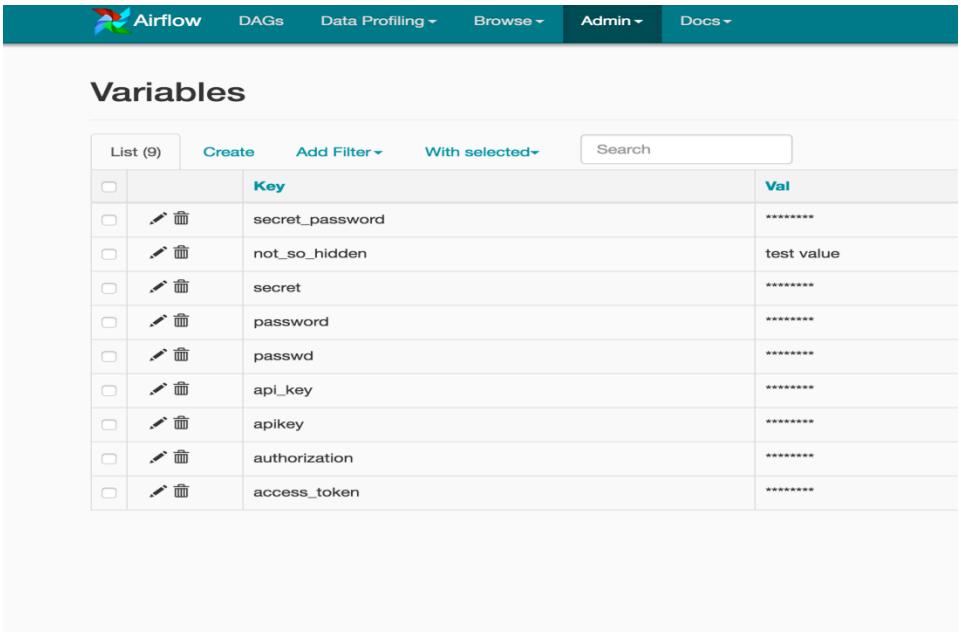
3. 图表视图

图形视图可能是最全面的一种表现形式了。它可以可视化您的 DAG 依赖以及某个运行实例的当前状态。



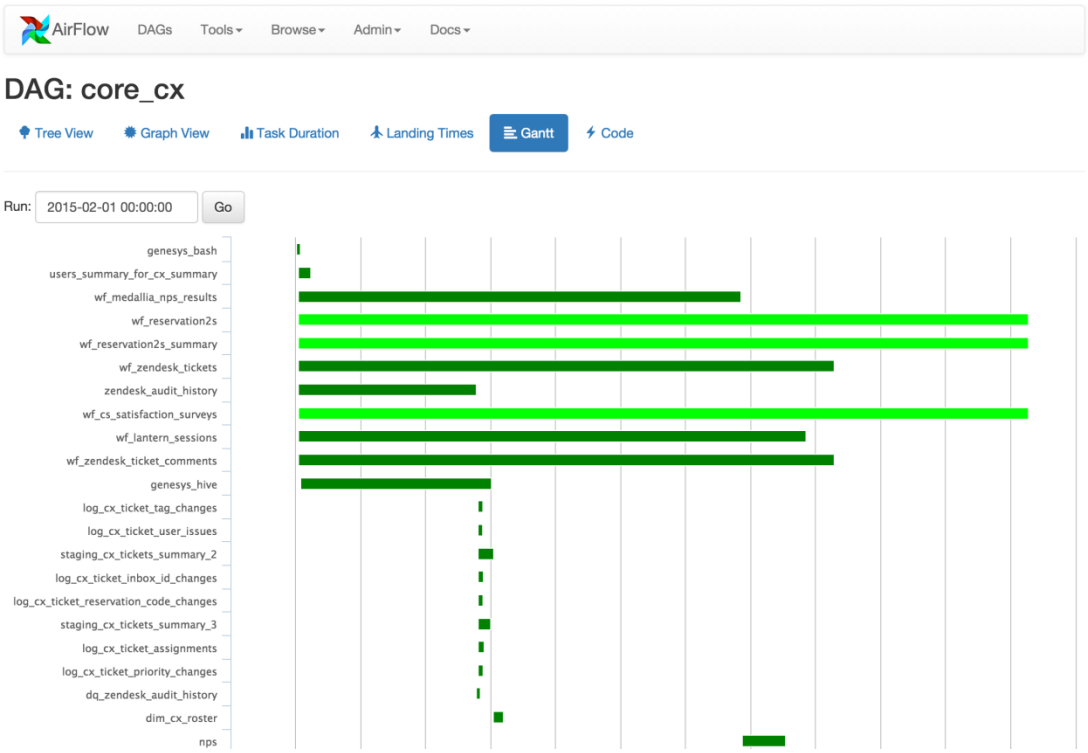
4. 变量视图

变量视图允许您列出、创建、编辑或删除作业期间使用的变量的键值对。如果密钥默认包含（'password'，'secret'，'passwd'，'authorization'，'api_key'，'apikey'，'access_token'）中的任何单词，则隐藏变量的值，但可以配置以明文显示。



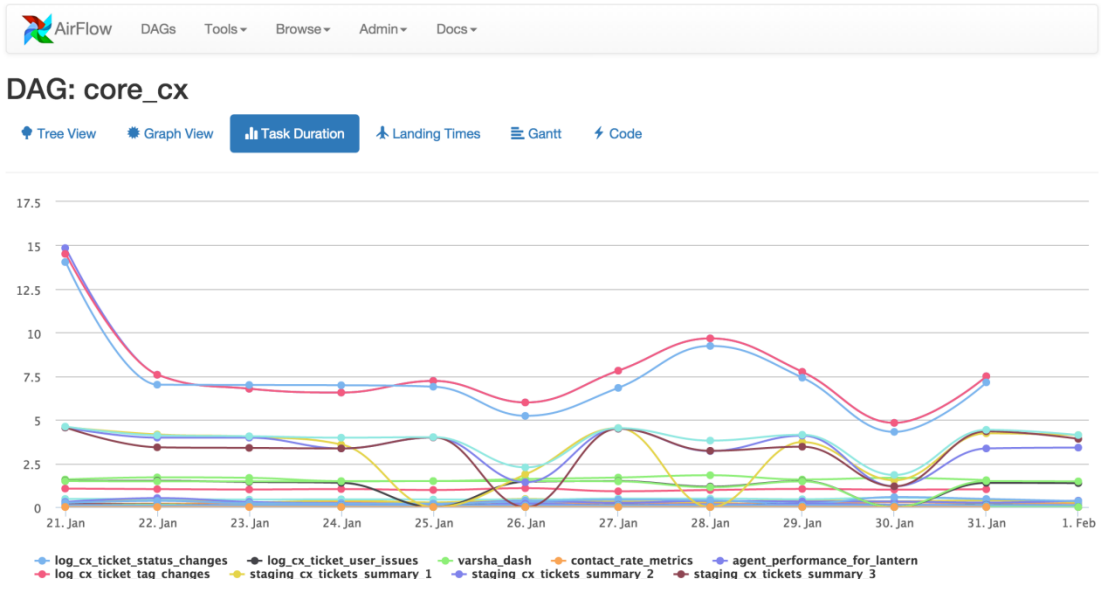
5. 甘特图

甘特图可让您分析任务持续时间和重叠情况。您可以快速识别系统瓶颈和哪些特定 DAG 在运行中花费了大量的时间。



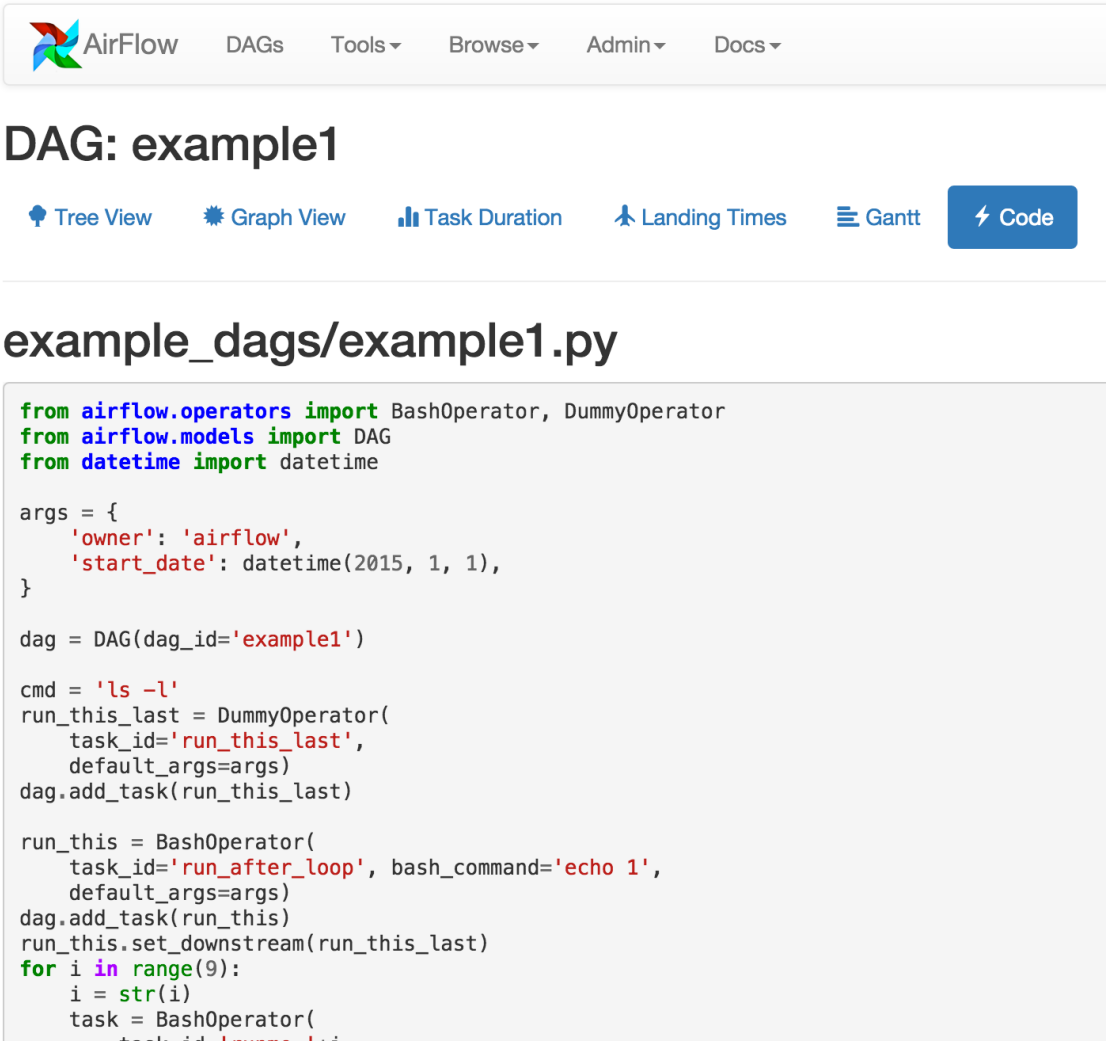
6. 任务持续时间

过去 N 次运行的不同任务的持续时间。通过此视图，您可以查找异常值并快速了解 DAG 在多次运行中花费的时间。



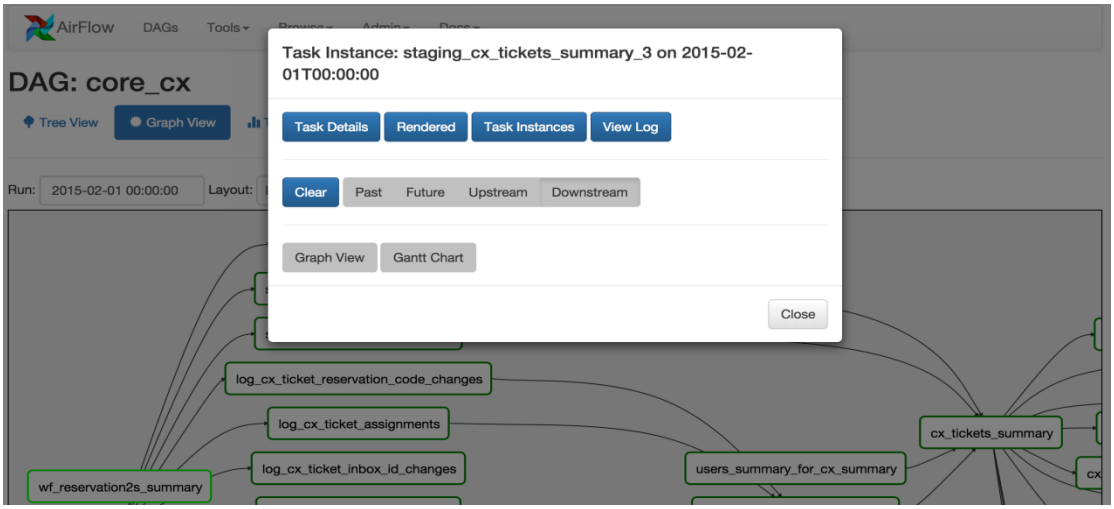
7. 代码视图

透明就是一切。虽然您的 pipeline（管道）代码在源代码管理中，但这是一种快速获取 DAG 代码并提供更多上下文的方法。



8. 任务实例上下文菜单

从上面的页面（树视图，图形视图，甘特图.....）中，始终可以单击任务实例，并进入此丰富的上下文菜单，该菜单可以将您带到更详细的元数据并执行某些操作。



概念

Airflow Platform 是用于描述，执行和监控工作流的工具。

1. 核心理念

DAGs

在 Airflow 中，DAG（或者叫有向无环图）是您要运行的所有任务的集合，以反映其关系和依赖关系的方式进行组织。

例如，一个简单的 DAG 可以包含三个任务：A，B 和 C.可以说 A 必须在 B 可以运行之前成功运行，但 C 可以随时运行。它可以说任务 A 在 5 分钟后超时，并且 B 可以重新启动最多 5 次以防它失败。它也可能会说工作流程将在每天晚上 10 点运行，但不应该在某个特定日期之前开始。

通过这种方式,DAG 描述了您希望如何执行工作流程; 但请注意,我们还没有说过我们真正想做的事情!A，B 和 C 可以是任何东西。当 C 发送电子邮件时，也许 A 准备 B 进行分析的数据。或者也许 A 监控你的位置，这样 B 可以打开你的车库门，而 C 打开你的房子灯。 重要的是，DAG 并不关心其组成任务的作用; 它的工作是确保无论他们做什么在正确的时间，或正确的顺序，或正确处理任何意外的问题。

DAG 是在标准 Python 文件中定义的，这些文件放在 Airflow 的 DAG_FOLDER 中。Airflow 将执行每个文件中的代码以动态构建 DAG 对象。 您可以拥有任意数量的 DAG，每个 DAG 都可以描述任意数量的任务。通常，每个应该对应于单个逻辑工作流。

注意:搜索 DAG 时，Airflow 将仅考虑字符串“airflow”和“DAG”都出现在.py 文件内容中的文件。

范围

Airflow 将加载它可以从 DAG 文件导入的任何 DAG 对象。重要的是,这意味着 DAG 必须出现在 globals()。考虑以下两个 DAG。只会加载 dag_1 ； 另一个只出现在本地范围内。

```
dag_1 = DAG('this_dag_will_be_discovered')

def my_function():
    dag_2 = DAG('but_this_dag_will_not')

my_function()
```

有时这可以很好地利用。例如，SubDagOperator 的常见模式是定义函数内的子标记，以便 Airflow 不会尝试将其作为独立的 DAG 加载。

默认参数

如果将 default_args 字典传递给 DAG，它将把它们应用于任何运算符。这使得很容易将公共参数应用于许多运算符而无需多次键入。

```
default_args = {
    'start_date': datetime(2016, 1, 1),
    'owner': 'Airflow'
}

dag = DAG('my_dag', default_args=default_args)
op = DummyOperator(task_id='dummy', dag=dag)
print(op.owner) # Airflow
```

上下文管理器

_ 在 Airflow 1.8 中添加 _

DAG 可用作上下文管理器，以自动将新 Operator 分配给该 DAG。

```
with DAG('my_dag', start_date=datetime(2016, 1, 1)) as dag:
    op = DummyOperator('op')

op.dag is dag # True
```

运营商

虽然 DAG 描述了如何运行工作流，但 Operators 确定实际完成的工作。

Operator 描述工作流中的单个任务。Operator 通常（但并非总是）是原子的，这意味着他们可以独立运营，而不需要与任何其他 Operator 共享资源。DAG 将确保 Operator 以正确的顺序运行；除了这些依赖项之外， Operator 通常独立运行。实际上，它们可以在两台完全不同的机器上运行。

这是一个微妙但非常重要的一点：通常，如果两个 Operator 需要共享信息，如文件名或少量数据，您应该考虑将它们组合到一个 Operator 中。如果绝对无法避免，Airflow 确实具有操作员交叉通信的功能，称为 XCom，本文档的其他部分对此进行了描述。

Airflow 为许多常见任务提供 Operator ，包括：

- BashOperator - 执行 bash 命令
- PythonOperator - 调用任意 Python 函数
- EmailOperator - 发送电子邮件
- SimpleHttpOperator - 发送 HTTP 请求
- MySqlOperator,SqliteOperator,PostgresOperator,MsSqlOperator,OracleOperator,JdbcOperator 等 - 执行 SQL 命令
- Sensor - 等待一定时间，文件，数据库行，S3 键等...

除了这些基本构建块之外，还有许多特定的 Operator ： DockerOperator，HiveOperator，S3FileTransformOperator，PrestoToMysqlOperator，SlackOperator.....你会明白的！

airflow/contrib/目录包含更多由社区构建的 Operator 。这些运算符并不总是像主发行版中那样完整或经过良好测试，但允许用户更轻松地向平台添加新功能。

如果将 Operator 分配给 DAG，则 Operator 仅由 Airflow 加载。

DAG 分配

_ 在 Airflow 1.8 中添加 _

Operator 不必立即分配给 DAG（之前的 dag 是必需参数）。但是，一旦将 Operator 分配给 DAG，就无法转移或取消分配。在创建运算符时，通过延迟赋值或甚至从其他运算符推断，可以显式地完成 DAG 分配。

```
dag = DAG('my_dag', start_date=datetime(2016, 1, 1))

# sets the DAG explicitly
explicit_op = DummyOperator(task_id='op1', dag=dag)

# deferred DAG assignment
deferred_op = DummyOperator(task_id='op2')
deferred_op.dag = dag

# inferred DAG assignment (linked operators must be in the same DAG)
inferred_op = DummyOperator(task_id='op3')
inferred_op.set_upstream(deferred_op)
```

位运算符

_ 在 Airflow 1.8 中添加 _

传统上，使用 set_upstream()和 set_downstream()方法设置运算符关系。在 Airflow 1.8 中，这可以通过 Python 位运算符>>和<<来完成。以下四个语句在功能上都是等效的：

```
op1 >> op2
op1.set_downstream(op2)

op2 << op1
op2.set_upstream(op1)
```

当使用位运算符时，关系设置在位运算符指向的方向上。例如，op1 >> op2 表示 op1 先运行，op2 第二运行。可以组成多个运算符 - 请记住，链从左到右执行，并且始终返回最右边的对象。 例如：

```
op1 >> op2 >> op3 << op4
```

相当于：

```
op1.set_downstream(op2)
op2.set_downstream(op3)
op3.set_upstream(op4)
```

为方便起见，位运算符也可以与 DAG 一起使用。 例如：

```
dag >> op1 >> op2
```

相当于：

```
op1.dag = dag
op1.set_downstream(op2)
```

我们可以把这一切放在一起构建一个简单的管道：

```
with DAG('my_dag', start_date=datetime(2016, 1, 1)) as dag:
    (
        DummyOperator(task_id='dummy_1')
        >> BashOperator(
            task_id='bash_1',
            bash_command='echo "HELLO!"')
        >> PythonOperator(
            task_id='python_1',
            python_callable=lambda: print("GOODBYE!"))
    )
```

任务

一旦 Operator 被实例化，它就被称为“任务”。实例化在调用抽象 Operator 时定义特定值，参数化任务成为 DAG 中的节点。

任务实例

任务实例表示任务的特定运行，其特征在于 dag，任务和时间点的组合。 任务实例也有一个指示状态，可以是“运行”，“成功”，“失败”，“跳过”，“重试”等。

工作流程

您现在熟悉 Airflow 的核心构建模块。 有些概念可能听起来非常相似，但词汇表可以概念化如下：

- DAG：描述工作应该发生的顺序
- Operator：作为执行某些工作的模板的类
- 任务： Operator 参数化实例
- 任务实例：1）已分配给 DAG 的任务，2）具有与 DAG 的特定运行相关联的状态

通过组合 DAGs 和 Operators 来创建 TaskInstances，您可以构建复杂的工作流。

附加功能

除了核心 Airflow 对象之外，还有许多更复杂的功能可以实现限制同时访问资源，交叉通信，条件执行等行为。

钩

钩子是外部平台和数据库的接口，如 Hive，S3，MySQL，Postgres，HDFS 和 Pig。Hooks 尽可能实现通用接口，并充当 Operator 的构建块。他们还使用 airflow.models.Connection 模型来检索主机名和身份验证信息。钩子将身份验证代码和信息保存在管道之外，集中在元数据数据库中。

钩子在 Python 脚本，Airflow airflow.operators.PythonOperator 以及 iPython 或 Jupyter Notebook 等交互式环境中使用它们也非常有用。

池

当有太多进程同时运行时，某些系统可能会被淹没。Airflow 池可用于限制任意任务集上的执行并行性 。通过为池命名并为其分配多个工作槽来在 UI（ Menu -> Admin -> Pools ）中管理池列表。然后，在创建任务时（即，实例化运算符），可以使用 pool 参数将任务与其中一个现有池相关联。

```
aggregate_db_message_job = BashOperator(
    task_id='aggregate_db_message_job',
    execution_timeout=timedelta(hours=3),
```

```
    pool='ep_data_pipeline_db_msg_agg',
    bash_command=aggregate_db_message_job_cmd,
    dag=dag)
aggregate_db_message_job.set_upstream(wait_for_empty_queue)
```

pool 参数可以与 priority_weight 结合使用，以定义队列中的优先级，以及在池中打开的槽时首先执行哪些任务。默认的 priority_weight 是 1，可以碰到任何数字。 在对队列进行排序以评估接下来应该执行哪个任务时，我们使用 priority_weight 与来自此任务下游任务的所有 priority_weight 值相加。您可以使用它来执行特定的重要任务，并相应地优先处理该任务的整个路径。

当插槽填满时，任务将照常安排。达到容量后，可运行的任务将排队，其状态将在 UI 中显示。当插槽空闲时，排队的任务将根据 priority_weight （任务及其后代）开始运行。

请注意，默认情况下，任务不会分配给任何池，并且它们的执行并行性仅限于执行程序的设置。

连接

外部系统的连接信息存储在 Airflow 元数据数据库中，并在 UI 中进行管理（Menu -> Admin -> Connections）。在那里定义了 conn_id ，并附加了主机名/登录/密码/结构信息。 Airflow 管道可以简单地引用集中管理的 conn_id 而无需在任何地方硬编码任何此类信息。

可以定义具有相同conn_id许多连接,并且在这种情况下,并且当挂钩使用来自BaseHook的get_connection方法时，Airflow 将随机选择一个连接，允许在与重试一起使用时进行一些基本的负载平衡和容错。

Airflow 还能够通过操作系统中的环境变量引用连接。但它只支持 URI 格式。如果您需要为连接指定extra信息，请使用 Web UI。

如果在 Airflow 元数据数据库和环境变量中都定义了具有相同 conn_id 连接，则 Airflow 将仅引用环境变量中的连接（例如，给定 conn_id postgres_master，在开始搜索元数据数据库之前，Airflow 将优先在环境变量中搜索 AIRFLOW_CONN_POSTGRES_MASTER 并直接引用它）。

许多钩子都有一个默认的conn_id,使用该挂钩的 Operator 不需要提供显式连接 ID。例如,PostgresHook的默认 conn_id 是 postgres_default 。

队列

使用 CeleryExecutor 时，可以指定发送任务到 Celery 队列。queue 是 BaseOperator 的一个属性，因此任何任务都可以分配给任何队列。 环境的默认队列配置在 airflow.cfg 的 celery -> default_queue。这定义了未指定任务时分配给的队列，以及 Airflow 工作程序在启动时侦听的队列。

Workers 可以收听一个或多个任务队列。当工作程序启动时（使用命令 airflow worker），可以指定一组逗号分隔的队列名称（例如， airflow worker -q spark）。然后，该 worker 将仅接收连接到指定队列的任务。

如果您需要特定的 workers，从资源角度来看（例如，一个工作人员可以毫无问题地执行数千个任务），或者从环境角度（您希望工作人员从 Spark 群集中运行），这可能非常有用本身，因为它需要一个非常具体的环境和安全的权利）。

XComs

XComs 允许任务交换消息，允许更细微的控制形式和共享状态。该名称是“交叉通信”的缩写。XComs 主要由键，值和时间戳定义，但也跟踪创建 XCom 的任务/DAG 以及何时应该可见的属性。任何可以被 pickle 的对象都可以用作 XCom 值，因此用户应该确保使用适当大小的对象。

可以“推”（发送）或“拉”（接收）XComs。当任务推送 XCom 时，它通常可用于其他任务。任务可以通过调用 xcom_push()方法随时推送 XComs。 此外，如果任务返回一个值（来自其 Operator 的 execute()方法，或者来自 PythonOperator 的 python_callable 函数），则会自动推送包含该值的 XCom。

任务调用 xcom_pull()来检索 XComs，可选地根据 key，source task_ids 和 source dag_id 等条件应用过滤器。默认情况下， xcom_pull()过滤掉从执行函数返回时被自动赋予 XCom 的键（与手动推送的 XCom 相反）。

如果为 task_ids 传递 xcom_pull 单个字符串,则返回该任务的最新 XCom 值; 如果传递了 task_ids 列表，则返回相应的 XCom 值列表。

```
# inside a PythonOperator called 'pushing_task'
def push_function():
    return value
```

```
# inside another PythonOperator where provide_context=True
def pull_function(**context):
    value = context['task_instance'].xcom_pull(task_ids='pushing_task')
```

也可以直接在模板中提取 XCom，这是一个示例：

```
SELECT * FROM {{ task_instance.xcom_pull(task_ids='foo', key='table_name') }}
```

请注意，XCom 与变量类似，但专门用于任务间通信而非全局设置。

变量

变量是将任意内容或设置存储和检索为 Airflow 中的简单键值存储的通用方法。可以从 UI（Admin -> Variables），代码或 CLI 列出，创建，更新和删除变量。此外，json 设置文件可以通过 UI 批量上传。虽然管道代码定义和大多数常量和变量应该在代码中定义并存储在源代码控制中，但是通过 UI 可以访问和修改某些变量或配置项会很有用。

```
from airflow.models import Variable
foo = Variable.get("foo")
bar = Variable.get("bar", deserialize_json=True)
```

第二个调用假设 json 内容，并将反序列化为 bar。请注意， Variable 是 sqlalchemy 模型，可以这样使用。

您可以使用 jinja 模板中的变量，其语法如下：
`echo {{ var.value.<variable_name> }}`

或者如果需要从变量反序列化 json 对象：
`echo {{ var.json.<variable_name> }}`

分枝

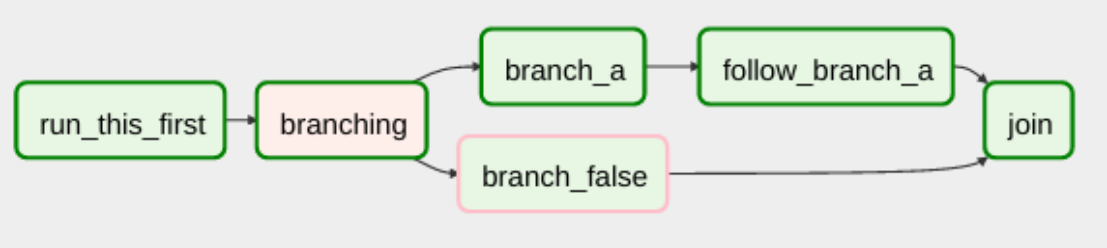
有时您需要一个工作流来分支，或者只根据任意条件走下某条路径，这通常与上游任务中发生的事情有关。一种方法是使用 BranchPythonOperator 。

BranchPythonOperator 与 PythonOperator 非常相似，只是它需要一个返回 task_id 的 python_callable。返回返回的 task_id，并跳过所有其他路径。Python 函数返回的 task_id 必须直接引用 BranchPythonOperator 任务下游的任务。

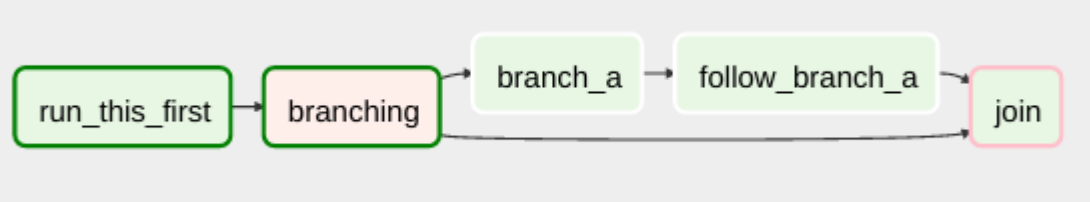
请注意，使用 depends_on_past=True 下游的任务在逻辑上是不合理的，因为 skipped 状态将总是导致依赖于过去成功的块任务。 skipped 状态在所有直接上游任务被 skipped 地方传播。

如果你想跳过一些任务，请记住你不能有一个空路径，如果是这样，那就做一个虚拟任务。

像这样，跳过虚拟任务“branch_false”



不喜欢这样，跳过连接任务



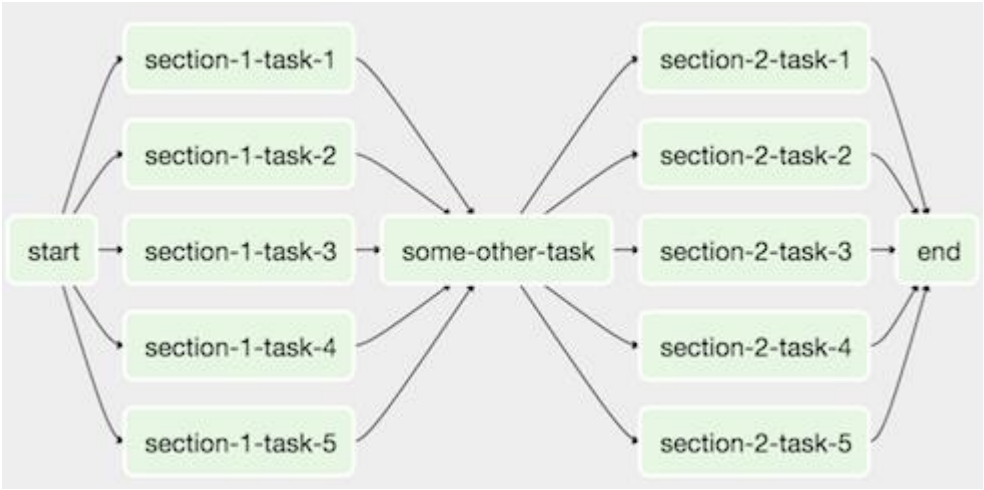
SubDAGs

SubDAG 非常适合重复模式。在使用 Airflow 时，定义一个返回 DAG 对象的函数是一个很好的设计模式。

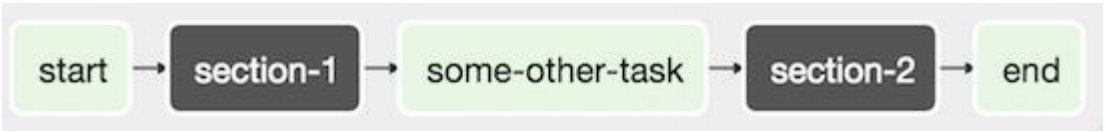
Airbnb 在加载数据时使用阶段检查交换模式。数据在临时表中暂存，然后对该表执行数据质量检查。 一

旦检查全部通过，分区就会移动到生产表中。

再举一个例子，考虑以下 DAG：



我们可以将所有并行 task-*运算符组合到一个 SubDAG 中，以便生成的 DAG 类似于以下内容：



请注意，SubDAG 运算符应包含返回 DAG 对象的工厂方法。 这将阻止 SubDAG 在主 UI 中被视为单独的 DAG。 例如：

```
#dags/subdag.py
from airflow.models import DAG
from airflow.operators.dummy_operator import DummyOperator

# Dag is returned by a factory method
def sub_dag(parent_dag_name, child_dag_name, start_date, schedule_interval):
  dag = DAG(
    '%s.%s' % (parent_dag_name, child_dag_name),
    schedule_interval=schedule_interval,
    start_date=start_date,
  )

  dummy_operator = DummyOperator(
    task_id='dummy_task',
    dag=dag,
  )
```

```
  return dag
```

然后可以在主 DAG 文件中引用此 SubDAG：

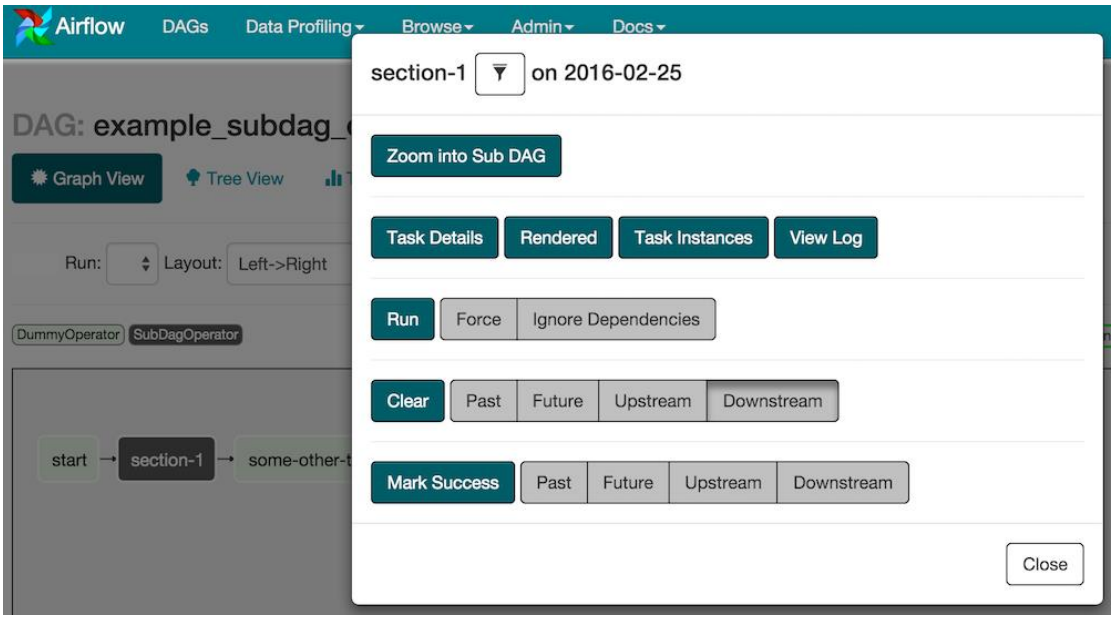
```
# main_dag.py
from datetime import datetime, timedelta
from airflow.models import DAG
from airflow.operators.subdag_operator import SubDagOperator
from dags.subdag import sub_dag


PARENT_DAG_NAME = 'parent_dag'
CHILD_DAG_NAME = 'child_dag'

main_dag = DAG(
  dag_id=PARENT_DAG_NAME,
  schedule_interval=timedelta(hours=1),
  start_date=datetime(2016, 1, 1)
)

sub_dag = SubDagOperator(
  subdag=sub_dag(PARENT_DAG_NAME, CHILD_DAG_NAME, main_dag.start_date,
                 main_dag.schedule_interval),
  task_id=CHILD_DAG_NAME,
  dag=main_dag,
)
```

您可以从主 DAG 的图形视图放大 SubDagOperator，以显示 SubDAG 中包含的任务：



使用 SubDAG 时的一些其他提示：

- 按照惯例，SubDAG 的 dag_id 应以其父级和点为前缀。 和在 parent.child
- 通过将参数传递给 SubDAG 运算符来共享主 DAG 和 SubDAG 之间的参数（如上所示）
- SubDAG 必须有一个计划并启用。如果 SubDAG 的时间表设置为 None 或@once ，SubDAG 将成功完成而不做任何事情
- 清除 SubDagOperator 也会清除其中的任务状态
- 在 SubDagOperator 上标记成功不会影响其中的任务状态
- 避免在任务中使用 depends_on_past=True，因为这可能会造成混淆
- 可以为 SubDAG 指定执行程序。 如果要在进程中运行 SubDAG 并有效地将其并行性限制为 1，则通常使用 SequentialExecutor。使用 LocalExecutor 可能会有问题，因为它可能会过度消耗您的 workers，在单个插槽中运行多个任务

SLAs

服务级别协议或者叫任务或 DAG 应该成功的时间，可以在任务级别设置为 timedelta。如果此时一个或多个实例未成功，则会发送警报电子邮件，详细说明错过其 SLA 的任务列表。该事件也记录在数据库中，并在 Browse->Missed SLAs 下的 Web UI 中显示，其中可以分析和记录事件。

触发规则

虽然正常的工作流行为是在所有直接上游任务都成功时触发任务，但 Airflow 允许更复杂的依赖项设置。

所有运算符都有一个 trigger_rule 参数，该参数定义生成的任务被触发的规则。trigger_rule 的默认值是 all_success，可以定义为“当所有直接上游任务都成功时触发此任务”。此处描述的所有其他规则都基于直接父任务，并且是在创建任务时可以传递给任何操作员的值：

- all_success :(默认）所有父任务都成功了
- all_failed ：所有父 all_failed 都处于 failed 或 upstream_failed 状态
- all_done ： 所有父任务都完成了他们的执行
- one_failed ：一旦至少一个父任务就会触发，它不会等待所有父任务完成
- one_success ：一旦至少一个父任务成功就触发，它不会等待所有父母完成
- dummy ：依赖项仅用于显示，随意触发

请注意，这些可以与 depends_on_past（boolean）结合使用，当设置为 True，如果任务的先前计划未成功，则不会触发任务。

只运行最新的

标准工作流行为涉及为特定日期/时间范围运行一系列任务。但是，某些工作流执行的任务与运行时无关，但需要按计划运行，就像标准的 cron 作业一样。在这些情况下，暂停期间错过的回填或运行作业会浪费 CPU 周期。

对于这种情况，您可以使用 LatestOnlyOperator 跳过在 DAG 的最近计划运行期间未运行的任务。如果现

在的时间不在其 execution_time 和下一个计划的 execution_time 之间，则 LatestOnlyOperator 跳过所有直接下游任务及其自身。

必须意识到跳过的任务和触发器规则之间的相互作用。跳过的任务将通过触发器规则 all_success 和 all_failed 级联，但不是 all_done ， one_failed ， one_success 和 dummy 。如果您希望将 LatestOnlyOperator 与不级联跳过的触发器规则一起使用，则需要确保 LatestOnlyOperator 直接位于您要跳过的任务的上游。

通过使用触发器规则来混合应该在典型的日期/时间依赖模式下运行的任务和使用 LatestOnlyOperator 任务是可能的。

例如，考虑以下 dag：

```
#dags/latest_only_with_trigger.py
import datetime as dt

from airflow.models import DAG
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators.latest_only_operator import LatestOnlyOperator
from airflow.utils.trigger_rule import TriggerRule


dag = DAG(
    dag_id='latest_only_with_trigger',
    schedule_interval=dt.timedelta(hours=4),
    start_date=dt.datetime(2016, 9, 20),
)

latest_only = LatestOnlyOperator(task_id='latest_only', dag=dag)

task1 = DummyOperator(task_id='task1', dag=dag)
task1.set_upstream(latest_only)

task2 = DummyOperator(task_id='task2', dag=dag)

task3 = DummyOperator(task_id='task3', dag=dag)
task3.set_upstream([task1, task2])

task4 = DummyOperator(task_id='task4', dag=dag,
                      trigger_rule=TriggerRule.ALL_DONE)
task4.set_upstream([task1, task2])
```

在这个 dag 的情况下，对于除最新运行之外的所有运行，latest_only 任务将显示为跳过。task1 直接位于 latest_only 下游，并且除了最新的之外还将跳过所有运行。task2 完全独立于 latest_only，将在所有

计划的时间段内运行。task3 是 task1 和 task2 下游，由于默认的 trigger_rule 是 all_success 将从 all_success 接收级联跳过。task4 是 task1 和 task2 下游，但由于其 trigger_rule 设置为 all_done 因此一旦跳过 all_done（有效的完成状态）并且 task2 成功，它将立即触发。

僵尸与不死

任务实例一直在死，通常是正常生命周期的一部分，但有时会出乎人到意料。

僵尸任务的特点是没有心跳（由工作定期发出）和数据库中的 running 状态。当工作节点无法访问数据库，Airflow 进程在外部被终止或者节点重新启动时，它们可能会发生。僵尸任务查杀是由调度程序的进程定期执行。

不死进程的特点是存在进程和匹配的心跳，但 Airflow 不知道此任务在数据库中 running。这种不匹配通常在数据库状态发生变化时发生，最有可能是通过删除 UI 中“任务实例”视图中的行。指示任务验证其作为心跳例程的一部分的状态，并在确定它们处于这种“不死”状态时终止自身。

集群策略

您的本地 Airflow 设置文件可以定义一个 policy 功能，该功能可以根据其他任务或 DAG 属性改变任务属性。它接收单个参数作为对任务对象的引用，并期望改变其属性。

例如，此函数可以在使用特定运算符时应用特定队列属性，或强制执行任务超时策略，确保没有任务运行超过 48 小时。 以下是 airflow_settings.py ：

```
def policy(task):
    if task.__class__.__name__ == 'HivePartitionSensor':
        task.queue = "sensor_queue"
    if task.timeout > timedelta(hours=48):
        task.timeout = timedelta(hours=48)
```

文档和注释

可以在 Web 界面中显示的 dag 和任务对象中添加文档或注释（dag 为“Graph View”，任务为“Task Details”）。如果定义了一组特殊任务属性，它们将被呈现为丰富内容：

属性	渲染到
doc	monospace
doc_json	JSON
doc_yaml	YAML
doc_md	markdown
doc_rst	reStructuredText

请注意，对于 dags，doc_md 是解释的唯一属性。

如果您的任务是从配置文件动态构建的，则此功能特别有用，它允许您公开 Airflow 中相关任务的配置。

```
"""
### My great DAG
"""

dag = DAG('my_dag', default_args=default_args)
dag.doc_md = __doc__

t = BashOperator("foo", dag=dag)
t.doc_md = """\
#Title"
Here's a [url](www.airbnb.com)
"""
```

此内容将分别在“图表视图”和“任务详细信息”页面中呈现为 markdown。

Jinja 模板

Airflow 充分利用了 Jinja Templating 的强大功能，这可以成为与宏结合使用的强大工具（参见宏部分）。

例如，假设您希望使用 BashOperator 将执行日期作为环境变量传递给 Bash 脚本。

```
# The execution date as YYYY-MM-DD
date = "{{ ds }}"
t = BashOperator(
    task_id='test_env',
    bash_command='/tmp/test.sh ',
    dag=dag,
    env={'EXECUTION_DATE': date})
```

这里， {{ ds }}是一个宏，并且由于 BashOperator 的 env 参数是使用 Jinja 模板化的，因此执行日期将作为 Bash 脚本中名为 EXECUTION_DATE 的环境变量提供。

您可以将 Jinja 模板与文档中标记为“模板化”的每个参数一起使用。模板替换发生在调用运算符的 pre_execute 函数之前。

打包的 dags

虽然通常会在单个.py 文件中指定 dags，但有时可能需要将 dag 及其依赖项组合在一起。 例如，您可能希望将多个 dag 组合在一起以将它们一起版本，或者您可能希望将它们一起管理，或者您可能需要一个额外的模块，默认情况下在您运行 airflow 的系统上不可用。 为此，您可以创建一个 zip 文件，其中包含 zip 文件根目录中的 dag，并在目录中解压缩额外的模块。

例如，您可以创建一个如下所示的 zip 文件：

```
my_dag1.py
my_dag2.py
package1/__init__.py
package1/functions.py
```

Airflow 将扫描 zip 文件并尝试加载 my_dag1.py 和 my_dag2.py 。 它不会进入子目录，因为它们被认为是潜在的包。

如果您想将模块依赖项添加到 DAG，您基本上也可以这样做，但是更多的是使用 virtualenv 和 pip。

```
virtualenv zip_dag
source zip_dag/bin/activate

mkdir zip_dag_contents
cd zip_dag_contents

pip install --install-option = "--install-lib= $PWD " my_useful_package
cp ~/my_dag.py .

zip -r zip_dag.zip *
```

注意:zip 文件将插入模块搜索列表（sys.path）的开头，因此它将可用于驻留在同一解释器中的任何其他代码。

注意:打包的 dags 不能与打开 pickling 时一起使用。

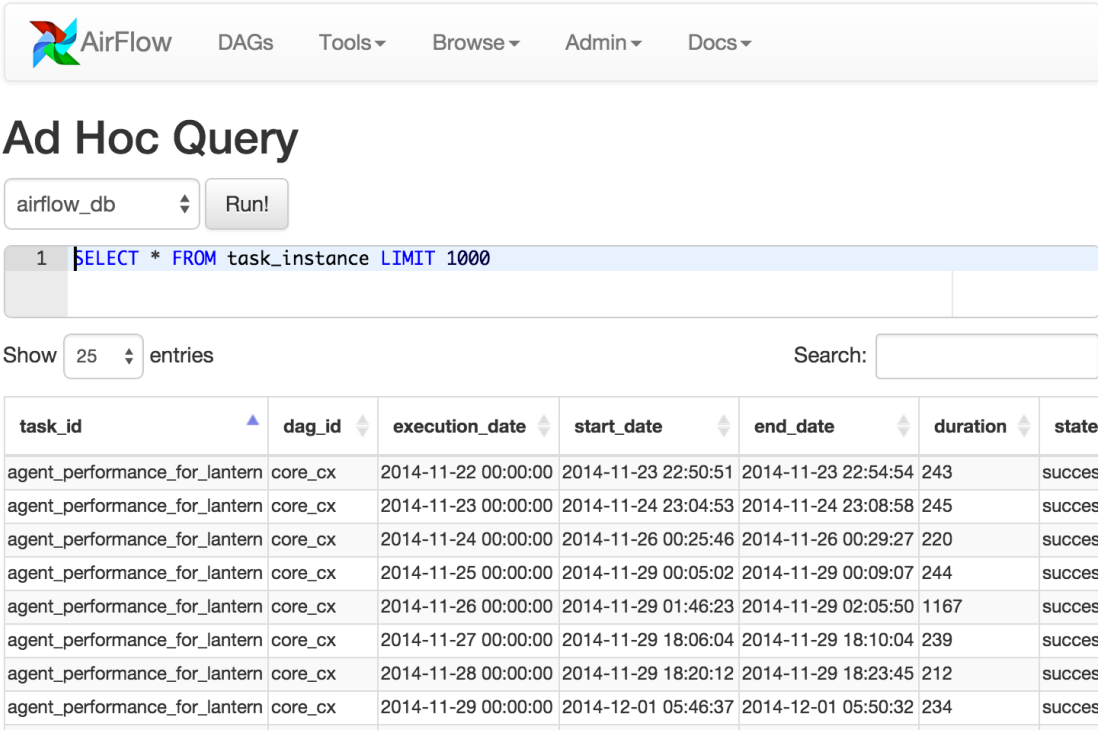
注意:打包的 dag 不能包含动态库（例如 libz.so），如果模块需要这些库，则需要在系统上使用这些库。换句话说，只能打包纯 python 模块。

数据分析

使用数据生产效率的一部分是拥有正确的武器来分析您正在使用的数据。Airflow 提供了一个简单的查询界面来编写 SQL 并快速获得结果，以及一个图表应用程序，可以让您可视化数据。

临时查询

adhoc 查询 UI 允许与 Airflow 中注册的数据库连接进行简单的 SQL 交互。



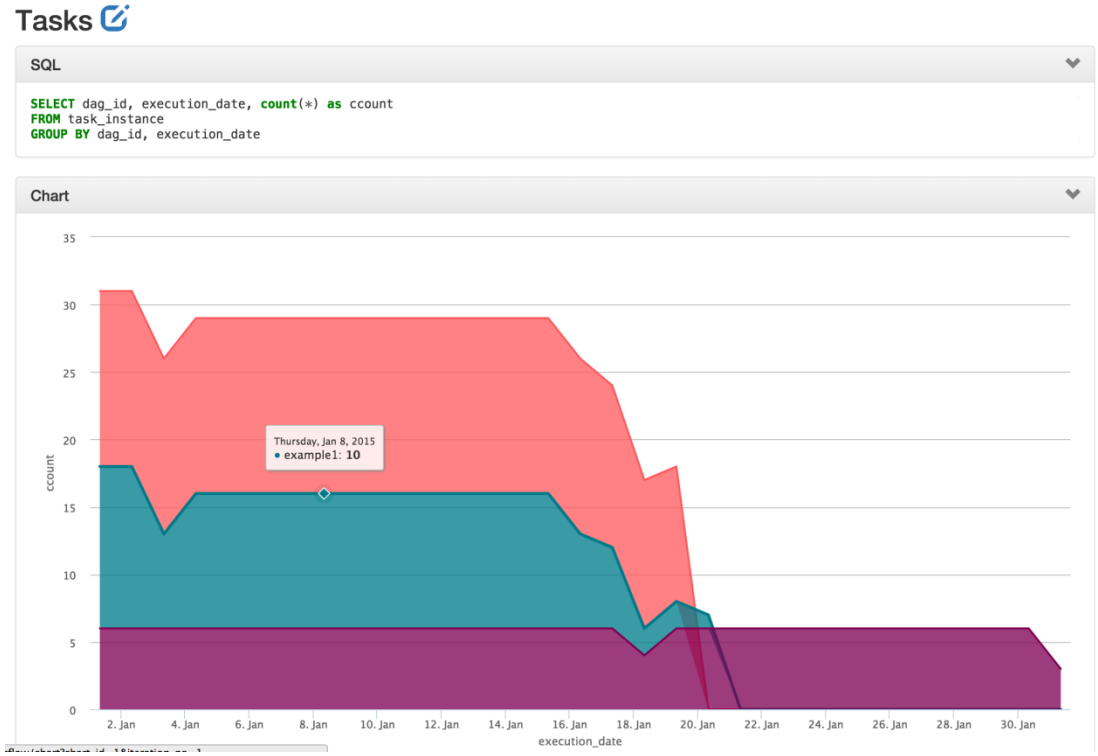
图表

基于 flask-admin 和 highcharts 构建的简单 UI 允许轻松构建数据可视化和图表。使用标签，SQL，图表类型填写表单，从环境的连接中选择源数据库，选择其他一些选项，然后保存以供以后使用。

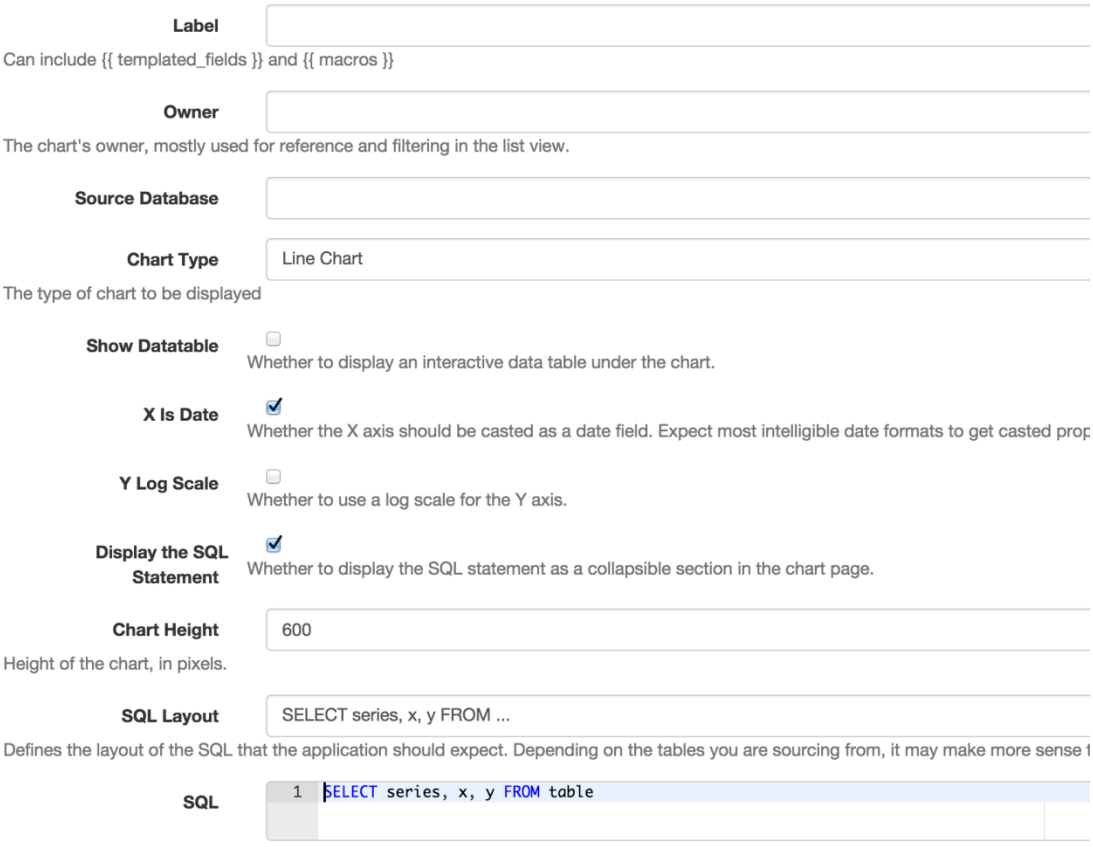
在编写 Airflow 管道，参数化查询和直接在 URL 中修改参数时，您甚至可以使用相同的模板和宏。

这些图表是基本的，但它们很容易创建，修改和共享。

图表截图


Tasks
SQL
SELECT dag_id, execution_date, count(*) as ccount
FROM task_instance
GROUP BY dag_id, execution_date
Chart
35
30
25
20
15
10
5
0
ccount
Thursday, Jan 8, 2015
example1: 10
2. Jan
4. Jan
6. Jan
8. Jan
10. Jan
12. Jan
14. Jan
16. Jan
18. Jan
20. Jan
22. Jan
24. Jan
26. Jan
28. Jan
30. Jan
execution_date

图表表格截图


Label
Can include {{ templated_fields }} and {{ macros }}
Owner
The chart's owner, mostly used for reference and filtering in the list view.
Source Database
Chart Type
Line Chart
The type of chart to be displayed
Show Datatable
Whether to display an interactive data table under the chart.
X Is Date
Whether the X axis should be casted as a date field. Expect most intelligible date formats to get casted prop
Y Log Scale
Whether to use a log scale for the Y axis.
Display the SQL Statement
Whether to display the SQL statement as a collapsible section in the chart page.
Chart Height
600
Height of the chart, in pixels.
SQL Layout
SELECT series, x, y FROM ...
Defines the layout of the SQL that the application should expect. Depending on the tables you are sourcing from, it may make more sense t
SQL
1
SELECT series, x, y FROM table

命令行接口

Airflow 具有非常丰富的命令行接口，允许在 DAG 上执行多种类型的操作，启动服务以及支持开发和测试。

```
usage: airflow [-h]
               {resetdb,render,variables,connections,create_user,pause,task_failed_deps,version,trigger_dag
,initdb,test,unpause,dag_state,run,list_tasks,backfill,list_dags,kerberos,worker,webserver,f
lower,scheduler,task_state,pool,serve_logs,clear,upgradedb,delete_dag}
               ...
```

必填参数

| 子命令 | 可能的选择：resetdb， render， variables， connections， create_user， pause，task_failed_deps,version,trigger_dag,initdb,test,unpause,dag_state,run,list_tasks,backfill,list_dags,kerberos,worker,webserver,flower,scheduler,task_state,pool ,serve_logs,clear,upgrab，delete_dag 子命令帮助 |

子命令：

Resetdb

删除并重建元数据数据库:

```
airflow resetdb [-h] [-y]
```

可选参数

| -y, --yes | 不要提示确认重置。请小心使用！默认值：False |

Render

渲染任务实例的模板

```
airflow render [-h] [-sd SUBDIR] dag_id task_id execution_date
```

必填参数

| dag_id | dag 的 id | | task_id | 任务的 id | | execution_date | DAG 的执行日期 |

可选参数

| -sd, --subdir | 从中查找 dag 的文件位置或目录 默认值：“[AIRFLOW_HOME]/dags” |

变量

对变量的 CRUD 操作

```
airflow variables [-h] [-s KEY VAL] [-g KEY] [-j] [-d VAL] [-i FILEPATH]
                  [-e FILEPATH] [-x KEY]
```

可选参数

| -s, --set | 设置变量 | | -g, --get | 获取变量的值 | | -j, --json | 反序列化 JSON 变量默认值：False | | -d, --default | 如果变量不存在，则返回默认值 | | -i, --import | 从 JSON 文件导入变量 | | -e, --export | 将变量导出到 JSON 文件 | | -x, --delete | 删除变量 |

Connections

列表/添加/删除连接

```
airflow connections [-h] [-l] [-a] [-d] [--conn_id CONN_ID]
                    [--conn_uri CONN_URI] [--conn_extra CONN_EXTRA]
                    [--conn_type CONN_TYPE] [--conn_host CONN_HOST]
                    [--conn_login CONN_LOGIN] [--conn_password CONN_PASSWORD]
                    [--conn_schema CONN_SCHEMA] [--conn_port CONN_PORT]
```

可选参数

| -l，--list | 列出所有连接，默认值：False | | -a，--add | 添加连接，默认值：False | | -d，--delete | 删除连接，默认值：False | | --conn_id | 连接 ID，添加/删除连接时必填 | | --conn_uri | 连接 URI，添加没有 conn_type 的连接时必填 | | --conn_extra | 连接的 Extra 字段，添加连接时可选 | | --conn_type | 连接类型，添加没有 conn_uri 的连接时时必填 | | --conn_host | 连接主机，添加连接时可选 | | --conn_login | 连接登录，添加连接时可选 | | --conn_password | 连接密码，添加连接时可选 | | --conn_schema | 连接架构，添加连接时可选 | | --conn_port | 连接端口，添加连接时可选 |

create_user

创建管理员帐户

```
airflow create_user [-h] [-r ROLE] [-u USERNAME] [-e EMAIL] [-f FIRSTNAME]
                    [-l LASTNAME] [-p PASSWORD] [--use_random_password]
```

可选参数

| -r，--role | 用户的角色。现有角色包括 Admin，User，Op，Viewer 和 Public | | -u，--username | 用户的用户名 | | -e，--电子邮件 | 用户的电子邮件 | | -f，--firstname | 用户的名字 | | -l，--lastname | 用户的姓氏 | | -p，--password | 用户密码 | | --use_random_password | 不提示输入密码。改为使用随机字符串默认值：False |

Pause

暂停 DAG

```
airflow pause [-h] [-sd SUBDIR] dag_id
```

必填参数

| dag_id | dag 的 id |

可选参数

| -sd，--subdir | 从中查找 dag 的文件位置或目录，默认值：“[AIRFLOW_HOME]/dags” |

task_failed_deps

从调度程序的角度返回任务实例的未满足的依赖项。 换句话说，为什么任务实例不会被调度程序调度然后排队，然后由执行程序运行。

```
airflow task_failed_deps [-h] [-sd SUBDIR] dag_id task_id execution_date
```

必填参数

| dag_id | dag 的 id | | task_id | 任务的 id | | execution_date | DAG 的执行日期 |

可选参数

| -sd，--subdir | 从中查找 dag 的文件位置或目录，默认值：“[AIRFLOW_HOME]/dags” |

Version

显示版本

```
airflow version [-h]
```

trigger_dag

触发 DAG 运行

```
airflow trigger_dag [-h] [-sd SUBDIR] [-r RUN_ID] [-c CONF] [-e EXEC_DATE]
                    dag_id
```

必填参数

| dag_id | dag 的 id |

可选参数

| -sd，--subdir | 从中查找 dag 的文件位置或目录，默认值:“[AIRFLOW_HOME]/dags” | | -r，--run_id | 帮助识别此次运行 | | -c,--conf | JSON 字符串被腌制到 DagRun 的 conf 属性中 | | -e,--exec_date | DAG 的执行日期 |

Initdb

初始化元数据数据库

```
airflow initdb [-h]
```

测试

测试任务实例。这将在不检查依赖关系或在数据库中记录其状态的情况下运行任务。

```
airflow test [-h] [-sd SUBDIR] [-dr] [-tp TASK_PARAMS]
             dag_id task_id execution_date
```

必填参数

| dag_id | dag 的 id | | task_id | 任务的 id | | execution_date | DAG 的执行日期 |

可选参数

| -sd，--subdir | 从中查找 dag 的文件位置或目录，默认值:“[AIRFLOW_HOME]/dags” | | -dr，--dr_run | 进行干运行默认值：False | | -tp，--task_params | 向任务发送 JSON params dict |

Unpause

恢复暂停的 DAG

```
airflow unpause [-h] [-sd SUBDIR] dag_id
```

必填参数

| dag_id | dag 的 id |

可选参数

| -sd，--subdir | 从中查找 dag 的文件位置或目录，默认值:“[AIRFLOW_HOME]/dags” |

dag_state

获取 dag run 的状态

```
airflow dag_state [-h] [-sd SUBDIR] dag_id execution_date
```

必填参数

| dag_id | dag 的 id | | execution_date | DAG 的执行日期 |

可选参数

| -sd，--subdir | 从中查找 dag 的文件位置或目录，默认值：“[AIRFLOW_HOME]/dags” |

Run

运行单个任务实例

```
airflow run [-h] [-sd SUBDIR] [-m] [-f] [--pool POOL] [--cfg_path CFG_PATH]
            [-l] [-A] [-i] [-I] [--ship_dag] [-p PICKLE] [-int]
            dag_id task_id execution_date
```

必填参数

| dag_id | dag 的 id | | task_id | 任务的 id | | execution_date | DAG 的执行日期 |

可选参数

| -sd，--subdir | 从中查找 dag 的文件位置或目录，默认值：“[AIRFLOW_HOME]/dags” | | -m，--mark_success | 将作业标记为成功而不运行它们默认值：False | | -f，--force | 忽略先前的任务实例状态，无论任务是否已成功/失败，都重新运行，默认值：False | | --pool | 要使用的资源池 | | --cfg_path | 要使用的配置文件的路径而不是 airflow.cfg | | -l，--local | 使用 LocalExecutor 运行任务，默认值：False | | -A，--ignore_all_dependencies | 忽略所有非关键依赖项，包括 ignore_ti_state 和 ignore_task_deps，默认值：False | | -i，--ignore_dependencies | 忽略特定于任务的依赖项，例如 upstream，depends_on_past 和重试延迟依赖项，默认值：False | | -I，--signore_depends_on_past | 忽略 depends_on_past 依赖项（但尊重上游依赖项），默认值：False | | --ship_dag | 泡菜（序列化）DAG 并将其运送给工人，默认值：False | | -p，--pickle | 整个 dag 的序列化 pickle 对象（内部使用） | | -int，--interactive | 不捕获标准输出和错误流（对交互式调试很有用），默认值：False |

list_tasks

列出 DAG 中的任务

```
airflow list_tasks [-h] [-t] [-sd SUBDIR] dag_id
```

必填参数

| dag_id | dag 的 id |

可选参数

| -t，--tree | 树视图，默认值：False | | -sd，--subdir | 从中查找 dag 的文件位置或目录，默认值：“[AIRFLOW_HOME]/dags” |

Backfill

在指定的日期范围内运行 DAG 的子部分 如果使用 reset_dag_run 选项，则回填将首先提示用户 Airflow 是否应清除回填日期范围内的所有先前 dag_run 和 task_instances。如果使用 rerun_failed_tasks，则回填将自动重新运行回填日期范围内的先前失败的任务实例。

```
airflow backfill [-h] [-t TASK_REGEX] [-s START_DATE] [-e END_DATE] [-m] [-l]
                 [-x] [-i] [-I] [-sd SUBDIR] [--pool POOL]
                 [--delay_on_limit DELAY_ON_LIMIT] [-dr] [-v] [-c CONF]
                 [--reset_dagruns] [--rerun_failed_tasks]
                 dag_id
```

必填参数

| dag_id | dag 的 id |

可选参数

| -t，--task_regex | | 用于过滤特定 task_ids 以回填的正则表达式（可选） | | -s，--start_date | | 覆盖 start_date YYYY-MM-DD | | -e，--end_date | 覆盖 end_date YYYY-MM-DD | | -m，--mark_success | | 将作业标记为成功而不运行它们，默认值：False | | -l，--local | 使用 LocalExecutor 运行任务，默认值：False | | -x，--donot_pickle | | 不要试图挑选 DAG 对象发送给工人，只要告诉工人运行他们的代码版本。默认值：False | | -i，--ignore_dependencies | | 跳过上游任务，仅运行与正则表达式匹配的任务。仅适用于 task_regex，默认值：False | | -I，--signore_first_depends_on_past | | 仅忽略第一组任务的 depends_on_past 依赖关系（回填 DO 中的后续执行依赖 depends_on_past）。默认值：False | | -sd，--subdir | 从中查找 dag 的文件位置或目录，默认值：“[AIRFLOW_HOME]/dags” | | --pool | 要使用的资源池 | | --delay_on_limit | | 在尝试再次执行 dag 运行之前达到最大活动 dag 运行限制(max_active_runs)时等待的时间(以秒为单位)。默认值：1.0 | | -dr，--dr_run | 进行干运行，默认值：False | | -v，--verbose | 使日志输出更详细，默认值：False | | -c，--conf | JSON 字符串被腌制到 DagRun 的 conf 属性中 | | --reset_dagruns | | | 如果设置，则回填将删除现有的与回填相关的 DAG 运行，并重新开始运行新的 DAG 运行，默认值：False | | --rerun_failed_tasks | | | 如果设置，则回填将自动重新运行回填日期范围的所有失败任务，而不是抛出异常，默认值：False |

list_dags

列出所有 DAG

```
airflow list_dags [-h] [-sd SUBDIR] [-r]
```

可选参数

| -sd，--subdir | 从中查找 dag 的文件位置或目录，默认值：“[AIRFLOW_HOME]/dags” | | -r，--report | 显示 DagBag 加载报告，默认值：False |

Kerberos

启动 kerberos 票证续订

```
airflow kerberos [-h] [-kt [KEYTAB]] [--pid [PID]] [-D] [--stdout STDOUT]
                 [--stderr STDERR] [-l LOG_FILE]
                 [principal]
```

必填参数

| principal | kerberos principal 默认值：airflow |

可选参数

| -kt，--keytab | 密钥表默认值：airflow.keytab | | --pid | PID 文件位置 | | -D，--daemon | 守护进程而不是在前台运行默认值：False | | --stdout | 将 stdout 重定向到此文件 | | --stderr | 将 stderr 重定向到此文件 | | -l，--log-file | 日志文件的位置 |

Worker

启动 Celery 工作节点

```
airflow worker [-h] [-p] [-q QUEUES] [-c CONCURRENCY] [-cn CELERY_HOSTNAME]
               [--pid [PID]] [-D] [--stdout STDOUT] [--stderr STDERR]
               [-l LOG_FILE]
```

可选参数

| -p，--do_pickle | | 尝试将 DAG 对象发送给工作人员，而不是让工作人员运行他们的代码版本。默认值：False | | -q，--queue | 以逗号分隔的队列列表，默认值：default | | -c，--concurrency | | 工作进程的数量，默认值：16 | | -cn，--slowry_hostname | | 如果一台计算机上有多个 worker，请设置 celery worker 的主机名。 | | --pid | PID 文件位置 | | -D，--daemon | 守护进程而不是在前台运行，默认值：False | | --stdout | 将 stdout 重定向到此文件 | | --stderr | 将 stderr 重定向到此文件 | | -l，--log-file | 日志文件的位置 |

webserver

启动 Airflow 网络服务器实例

```
airflow webserver [-h] [-p PORT] [-w WORKERS]
                  [-k {sync,eventlet,gevent,tornado}] [-t WORKER_TIMEOUT]
                  [-hn HOSTNAME] [--pid [PID]] [-D] [--stdout STDOUT]
                  [--stderr STDERR] [-A ACCESS_LOGFILE] [-E ERROR_LOGFILE]
                  [-l LOG_FILE] [--ssl_cert SSL_CERT] [--ssl_key SSL_KEY] [-d]
```

可选参数

| -p，--port | 运行服务器的端口，默认值：8080 | | -w，--workers | 运行 Web 服务器的工作者数量，默认值：4 | | -k，--workerclass | | | 可能的选择：sync，eventlet，gevent，tornado 用于 Gunicorn 的 worker class，默认值：sync | | -t，--worker_timeout | | | 等待 Web 服务器工作者的超时时间，默认值：120 | | -hn，--hostname | | | 设置运行 Web 服务器的主机名，默认值：0.0.0.0 | | --pid | PID 文件位置 | | -D，--daemon | 守护进程而不是在前台运行，默认值：False | | --stdout | 将 stdout 重定向到此文件 | | --stderr | 将 stderr 重定向到此文件 | | -A，--access_logfile | | | 用于存储 Web 服务器访问日志的日志文件。 使用'-'打印到 stderr。默认值：- | | -E，--error_logfile | | | 用于存储 Web 服务器错误日志的日志文件。 使用'-'打印到 stderr。默认值：- | | -l，--log-file | 日志文件的位置 | | --ssl_cert | Web 服务器的 SSL 证书的路径 | | --ssl_key | 用于 SSL 证书的密钥的路径 | | -d，--debug | 在调试模式下使用 Flask 附带的服务器，默认值：False |

Flower

运行 Celery Flower

```
airflow flower [-h] [-hn HOSTNAME] [-p PORT] [-fc FLOWER_CONF] [-u URL_PREFIX]
               [-a BROKER_API] [--pid [PID]] [-D] [--stdout STDOUT]
               [--stderr STDERR] [-l LOG_FILE]
```

可选参数

| -hn，--hostname | | | 设置运行服务器的主机名，默认值：0.0.0.0 | | -p，--port | 运行服务器的端口，默认值：5555 | | -fc，--flowers_conf | | | celery 的配置文件 | | -u，--url_prefix | | | Flower 的 URL 前缀 | | -a，--broker_api | | | Broker api | | --pid | PID 文件位置 | | -D，--daemon | 守护进程而不是在前台运行，默认值：False | | --stdout | 将 stdout 重定向到此文件 | | --stderr | 将 stderr 重定向到此文件 | | -l，--log-file | 日志文件的位置 |

Scheduler

启动调度程序实例

```
airflow scheduler [-h] [-d DAG_ID] [-sd SUBDIR] [-r RUN_DURATION]
                  [-n NUM_RUNS] [-p] [--pid [PID]] [-D] [--stdout STDOUT]
                  [--stderr STDERR] [-l LOG_FILE]
```

可选参数

| -d，--dag_id | 要运行的 dag 的 id | | -sd，--subdir | 从中查找 dag 的文件位置或目录，默认值：“[AIRFLOW_HOME]/dags” | | -r，--run-duration | | | 设置退出前执行的秒数 | | -n，--num_runs | 设置退出前要执行的运行次数，默认值：-1 | | -p，--do_pickle | | | 尝试将 DAG 对象发送给工作人员，而不是让工作人员运行他们的代码版本。默认值：False | | --pid | PID 文件位置 | | -D，--daemon | 守护进程而不是在前台运行默认值：False | | --stdout | 将 stdout 重定向到此文件 | | --stderr | 将 stderr 重定向到此文件 | | -l，--log-file | 日志文件的位置 |

task_state

获取任务实例的状态

```
airflow task_state [-h] [-sd SUBDIR] dag_id task_id execution_date
```

必填参数

| dag_id | dag 的 id | | task_id | 任务的 id | | execution_date | DAG 的执行日期 |

可选参数

| -sd，--subdir | 从中查找 dag 的文件位置或目录，默认值：“[AIRFLOW_HOME]/dags” |

Pool

pool 的 CRUD 操作

```
airflow pool [-h] [-s NAME SLOT_COUNT POOL_DESCRIPTION] [-g NAME] [-x NAME]
```

可选参数

| -s，--set | 分别设置池槽数和描述 | | -g，--get | 获取池信息 | | -x，--delete | 删除池 |

serve_logs

由 worker 生成的服务日志

```
airflow serve_logs [-h]
```

clear

清除一组任务实例，就好像它们从未运行过一样

```
airflow clear [-h] [-t TASK_REGEX] [-s START_DATE] [-e END_DATE] [-sd SUBDIR]
```

```
              [-u] [-d] [-c] [-f] [-r] [-x] [-xp] [-dx]
              dag_id
```

必填参数

| dag_id | dag 的 id |

可选参数

| -t，--task_regex | | 用于过滤特定 task_ids 以回填的正则表达式（可选） | | -s，--start_date | | 覆盖 start_date YYYY-MM-DD | | -e，--end_date | 覆盖 end_date YYYY-MM-DD | | -sd，--subdir | 从中查找 dag 的文件位置或目录，默认值：“[AIRFLOW_HOME]/dags” | | -u，--upstream | 包括上游任务，默认值：False | | -d，--downstream | | 包括下游任务，默认值：False | | -c，--no_confirm | | 请勿要求确认，默认值：False | | -f，--only_failed | | 只有失败的工作，默认值：False | | -r，--only_running | | 只运行工作，默认值：False | | -x，--exclude_subdags | | 排除子标记，默认值：False | | -dx，--dag_regex | | 将 dag_id 搜索为正则表达式而不是精确字符串，默认值：False |

Upgradedb

将元数据数据库升级到最新版本

```
airflow upgradedb [-h]
delete_dag
```

删除与指定 DAG 相关的所有 DB 记录

```
airflow delete_dag [-h] [-y] dag_id
```

必填参数

| dag_id | dag 的 id |

可选参数

| -y，--是的 | 不要提示确认重置。 小心使用！默认值：False |

调度和触发器

Airflow 调度程序监视所有任务和所有 DAG，并触发已满足其依赖关系的任务实例。 在幕后，它监视并与其可能包含的所有 DAG 对象的文件夹保持同步，并定期（每分钟左右）检查活动任务以查看是否可以触发它们。

Airflow 调度程序旨在作为 Airflow 生产环境中的持久服务运行。 要开始，您需要做的就是执行 airflow

scheduler 。 它将使用 airflow.cfg 指定的配置。

请注意，如果您在一天的 schedule_interval 上运行 DAG，则会在 2016-01-01T23:59 之后不久触发标记为 2016-01-01 的运行。 换句话说，作业实例在其覆盖的时间段结束后启动。 请注意，如果您运行一个 schedule_interval 为 1 天的的 DAG，run 标记为 2016-01-01，那么它会在 2016-01-01T23:59 之后马上触发。换句话说，一旦设定的时间周期结束后，工作实例将立马开始。

让我们重复一遍调度 schedule_interval 在开始日期之后，在句点结束时运行您的作业一个 schedule_interval 。

调度程序启动 airflow.cfg 指定的执行程序的实例。 如果碰巧是 LocalExecutor ，任务将作为子 LocalExecutor 执行; 在 CeleryExecutor 和 MesosExecutor 的情况下，任务是远程执行的。

要启动调度程序，只需运行以下命令：

```
airflow scheduler
```

DAG 运行

DAG Run 是一个表示 DAG 实例化的对象。

每个 DAG 可能有也可能没有时间表，通知如何创建 DAG Runs 。 schedule_interval 被定义为 DAG 参数，并且优选地接收作为 str 的 cron 表达式或 datetime.timedelta 对象。或者,您也可以使用其中一个 cron “预设”:

| 预置 | 含义 | cron 的 | | --- | --- | --- | | `None` | 不要安排，专门用于“外部触发”的 DAG | | | `@once` | 安排一次，只安排一次 | | | `@hourly` | 在小时开始时每小时运行一次 | `0 * * * *` | | `@daily` | 午夜一天运行一次 | `0 0 * * *` | | `@weekly` | 周日早上每周午夜运行一次 | `0 0 * * 0` | | `@monthly` | 每个月的第一天午夜运行一次 | `0 0 1 * *` | | `@yearly` | 每年 1 月 1 日午夜运行一次 | `0 0 1 1 *` |
您的 DAG 将针对每个计划进行实例化，同时为每个计划创建 DAG Run 条目。

DAG 运行具有与它们相关联的状态(运行，失败，成功)，并通知调度程序应该针对任务提交评估哪组调度。如果没有 DAG 运行级别的元数据，Airflow 调度程序将需要做更多的工作才能确定应该触发哪些任务并进行爬行。 在更改 DAG 的形状时，也可能会添加新任务，从而创建不需要的处理。

回填和追赶

具有 start_date （可能是 end_date ）和 schedule_interval 的 Airflow DAG 定义了一系列间隔，调度程序将这些间隔转换为单独的 Dag 运行并执行。Airflow 的一个关键功能是这些 DAG 运行是原子的幂等项，默认情况下，调度程序将检查 DAG 的生命周期（从开始到结束/现在，一次一个间隔）并启动 DAG 运行对于尚未运行（或已被清除）的任何间隔。 这个概念叫做 Catchup。

如果你的 DAG 被编写来处理它自己的追赶（IE 不仅限于间隔，而是改为“现在”。），那么你将需要关闭追

赶（在 DAG 本身上使用 dag.catchup = False ）或者默认情况下在配置文件级别使用 catchup_by_default = False 。 这样做，是指示调度程序仅为 DAG 间隔序列的最新实例创建 DAG 运行。

```
"""
Code that goes along with the Airflow tutorial located at:
https://github.com/airbnb/airflow/blob/master/airflow/example_dags/tutorial.py
"""
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime, timedelta


default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2015, 12, 1),
    'email': ['airflow@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
    'schedule_interval': '@hourly',
}

dag = DAG('tutorial', catchup=False, default_args=default_args)
```

在上面的示例中，如果调度程序守护程序在 2016-01-02 上午 6 点（或从命令行）拾取 DAG，则将创建单个 DAG 运行，其 execution_date 为 2016-01-01 ，下一个将在 2016-01-03 上午午夜后创建，执行日期为 2016-01-02。

如果 dag.catchup 值为 True，则调度程序将为 2015-12-01 和 2016-01-02 之间的每个完成时间间隔创建一个 DAG Run（但不是 2016-01-02 中的一个，因为该时间间隔）尚未完成）并且调度程序将按顺序执行它们。 对于可以轻松拆分为句点的原子数据集，此行为非常有用。 如果您的 DAG 运行在内部执行回填，则关闭追赶是很好的。

外部触发器

请注意，在运行 airflow trigger_dag 命令时，也可以通过 CLI 手动创建 DAG Runs ，您可以在其中定义特定的 run_id 。 在调度程序外部创建的 DAG Runs 与触发器的时间戳相关联，并将与预定的 DAG runs 一起显示在 UI 中。

此外，您还可以使用 Web UI 手动触发 DAG Run （选项卡“DAG” ->列“链接” ->按钮“触发器 Dag”）。

要牢记

- 第一个 DAG Run 是基于 DAG 中任务的最小 start_date 创建的。
- 后续 DAG Runs 由调度程序进程根据您的 DAG 的 schedule_interval 顺序创建。
- 当清除一组任务的状态以期让它们重新运行时，重要的是要记住 DAG Run 的状态，因为它定义了调度程序是否应该查看该运行的触发任务。

以下是一些可以取消阻止任务的方法 ：

- 在 UI 中，您可以从任务实例对话框中清除 （如删除状态）各个任务实例，同时定义是否要包括过去/未来和上游/下游依赖项。 请注意，接下来会出现一个确认窗口，您可以看到要清除的设置。 您还可以清除与 dag 关联的所有任务实例。
- CLI 命令 airflow clear -h 在清除任务实例状态时有很多选项，包括指定日期范围，通过指定正则表达式定位 task_ids，包含上游和下游亲属的标志，以及特定状态下的目标任务实例（ failed 或 success ）
- 清除任务实例将不再删除任务实例记录。 相反，它更新 max_tries 并将当前任务实例状态设置为 None。
- 将任务实例标记为失败可以通过 UI 完成。 这可用于停止运行任务实例。
- 将任务实例标记为成功可以通过 UI 完成。 这主要是为了修复漏报，或者例如在 Airflow 之外应用修复时。
- airflow backfill CLI 子命令具有--mark_success 标志，允许选择 DAG 的子部分以及指定日期范围。

插件

Airflow 内置了一个简单的插件管理器，可以通过简单地删除$AIRFLOW_HOME/plugins 文件夹中的文件，将外部功能集成到其核心。

plugins 文件夹中的 python 模块将被导入，钩子，操作符，传感器，宏，执行器和 Web 视图将集成到 Airflow 的主要集合中，并可供使用。

做什么的？

Airflow 提供了一个用于处理数据的通用工具箱。不同的组织有不同的堆栈和不同的需求。 使用 Airflow 插件可以让公司定制他们的 Airflow 安装以反映他们的生态系统。

插件可以简便地用作编写，共享和激活新功能集。

当然还需要一组更复杂的应用程序来与不同风格的数据和元数据进行交互。

例子：

- 一组用于解析 Hive 日志和公开 Hive 元数据（CPU/IO/阶段/倾斜/...）的工具
- 异常检测框架，允许人们收集指标，设置阈值和警报
- 审计工具，帮助了解谁访问了什么
- 配置驱动的 SLA 监控工具，允许您设置受监控的表以及应该在何时着陆，提醒人员并公开停机的可视化

- ...

为什么要建立在 Airflow 之上?

Airflow 有许多组件可以在构建应用程序时重用:

- 可用于呈现视图的 Web 服务器
- 用于存储模型的元数据数据库
- 访问您的数据库,以及如何连接到它们
- 应用程序可以将工作负载推送到的一组 Workers
- 部署了 Airflow,您可以专注于后面的工作
- 基本的图表功能,底层库和抽象

接口

要创建插件,您需要派生 airflow.plugins_manager.AirflowPlugin 类并引用要插入 Airflow 的对象。以下是类似您需要派生的类:

```
class AirflowPlugin(object):
    # The name of your plugin (str)
    name = None
    # A list of class(es) derived from BaseOperator
    operators = []
    # A list of class(es) derived from BaseSensorOperator
    sensors = []
    # A list of class(es) derived from BaseHook
    hooks = []
    # A list of class(es) derived from BaseExecutor
    executors = []
    # A list of references to inject into the macros namespace
    macros = []
    # A list of objects created from a class derived
    # from flask_admin.BaseView
    admin_views = []
    # A list of Blueprint object created from flask.Blueprint. For use with the flask_admin based GUI
    flask_blueprints = []
    # A list of menu links (flask_admin.base.MenuLink). For use with the flask_admin based GUI
    menu_links = []
    # A list of dictionaries containing FlaskAppBuilder BaseView object and some metadata. See example below
    appbuilder_views = []
    # A list of dictionaries containing FlaskAppBuilder BaseView object and some metadata. See example below
```

```
    appbuilder_menu_items = []
```

示例

下面的代码定义了一个插件，它在 Airflow 中注入一组虚拟对象定义。

```
# This is the class you derive to create a plugin
from airflow.plugins_manager import AirflowPlugin

from flask import Blueprint
from flask_admin import BaseView, expose
from flask_admin.base import MenuLink

# Importing base classes that we need to derive
from airflow.hooks.base_hook import BaseHook
from airflow.models import BaseOperator
from airflow.sensors.base_sensor_operator import BaseSensorOperator
from airflow.executors.base_executor import BaseExecutor

# Will show up under airflow.hooks.test_plugin.PluginHook
class PluginHook(BaseHook):
    pass

# Will show up under airflow.operators.test_plugin.PluginOperator
class PluginOperator(BaseOperator):
    pass

# Will show up under airflow.sensors.test_plugin.PluginSensorOperator
class PluginSensorOperator(BaseSensorOperator):
    pass

# Will show up under airflow.executors.test_plugin.PluginExecutor
class PluginExecutor(BaseExecutor):
    pass

# Will show up under airflow.macros.test_plugin.plugin_macro
def plugin_macro():
    pass

# Creating a flask admin BaseView
class TestView(BaseView):
    @expose('/')
    def test(self):
        # in this example, put your test_plugin/test.html template at
airflow/plugins/templates/test_plugin/test.html
```

```
        return self.render("test_plugin/test.html", content="Hello galaxy!")
v = TestView(category="Test Plugin", name="Test View")

# Creating a flask blueprint to integrate the templates and static folder
bp = Blueprint(
    "test_plugin", __name__,
    template_folder='templates', # registers airflow/plugins/templates as a Jinja template
folder
    static_folder='static',
    static_url_path='/static/test_plugin')

ml = MenuLink(
    category='Test Plugin',
    name='Test Menu Link',
    url='https://airflow.incubator.apache.org/')

# Creating a flask appbuilder BaseView
class TestAppBuilderBaseView(AppBuilderBaseView):
    @expose("/")
    def test(self):
        return self.render("test_plugin/test.html", content="Hello galaxy!")
v_appbuilder_view = TestAppBuilderBaseView()
v_appbuilder_package = {"name": "Test View",
                        "category": "Test Plugin",
                        "view": v_appbuilder_view}

# Creating a flask appbuilder Menu Item
appbuilder_mitem = {"name": "Google",
                    "category": "Search",
                    "category_icon": "fa-th",
                    "href": "https://www.google.com"}

# Defining the plugin class
class AirflowTestPlugin(AirflowPlugin):
    name = "test_plugin"
    operators = [PluginOperator]
    sensors = [PluginSensorOperator]
    hooks = [PluginHook]
    executors = [PluginExecutor]
    macros = [plugin_macro]
    admin_views = [v]
    flask_blueprints = [bp]
    menu_links = [ml]
    appbuilder_views = [v_appbuilder_package]
```

```
appbuilder_menu_items = [appbuilder_mitem]
```

注意基于角色的视图

Airflow 1.10 使用 FlaskAppBuilder 引入了基于角色的视图。您可以通过设置 rbac = True 来配置使用的 UI。为了支持两个版本的 UI 的插件视图和链接以保持向后兼容性，请将字段 appbuilder_views 和 appbuilder_menu_items 添加到 AirflowTestPlugin 类中。

安全

默认情况下，所有门都是打开的。限制对 Web 应用程序的访问的一种简单方法是在网络级别执行此操作，比如使用 SSH 隧道。

但也可以通过使用其中一个已提供的认证后端或创建自己的认证后端来打开身份验证。

请务必查看 Experimental Rest API 以保护 API。

Web 身份验证

密码

最简单的身份验证机制之一是要求用户在登录前输入密码。密码身份验证需要在 requirements 文件中使用 password 子扩展包。它在存储密码之前使用了 bcrypt 扩展包来对密码进行哈希。

```
[webserver]
authenticate = True
auth_backend = airflow.contrib.auth.backends.password_auth
```

启用密码身份验证后，需要先创建初始用户凭据，然后才能登录其他人。未在此身份验证后端的迁移中创建初始用户，以防止默认 Airflow 安装受到攻击。必须通过安装 Airflow 的同一台机器上的 Python REPL 来创建新用户。

```
# navigate to the airflow installation directory
$ cd ~/airflow
$ python
Python 2.7.9 (default, Feb 10 2015, 03:28:08)
Type "help", "copyright", "credits" or "license" for more information.
>>> import airflow
>>> from airflow import models, settings
>>> from airflow.contrib.auth.backends.password_auth import PasswordUser
>>> user = PasswordUser(models.User())
>>> user.username = 'new_user_name'
>>> user.email = 'new_user_email@example.com'
>>> user.password = 'set_the_password'
```

```
>>> session = settings.Session()
>>> session.add(user)
>>> session.commit()
>>> session.close()
>>> exit()
```

LDAP

要打开 LDAP 身份验证，请按如下方式配置 airflow.cfg。请注意，该示例使用与 ldap 服务器的加密连接，因为您可能不希望密码在网络级别上可读。 但是，如果您真的想要，可以在不加密的情况下进行配置。

此外，如果您使用的是 Active Directory，并且没有明确指定用户所在的 OU，则需要将 search_scope 更改为“SUBTREE”。

有效的 search_scope 选项可以在 ldap3 文档中找到

```
[webserver]
authenticate = True
auth_backend = airflow.contrib.auth.backends.ldap_auth

[ldap]
# set a connection without encryption: uri = ldap://<your.ldap.server>:<port>
uri = ldaps://<your.ldap.server>:<port>
user_filter = objectClass=*
# in case of Active Directory you would use: user_name_attr = sAMAccountName
user_name_attr = uid
# group_member_attr should be set accordingly with *_filter
# eg :
#     group_member_attr = groupMembership
#     superuser_filter = groupMembership=CN=airflow-super-users...
group_member_attr = memberOf
superuser_filter                                                                =
memberOf=CN=airflow-super-users,OU=Groups,OU=RWC,OU=US,OU=NORAM,DC=example,DC=com
data_profiler_filter                                                            =
memberOf=CN=airflow-data-profilers,OU=Groups,OU=RWC,OU=US,OU=NORAM,DC=example,DC=com
bind_user = cn=Manager,dc=example,dc=com
bind_password = insecure
basedn = dc=example,dc=com
cacert = /etc/ca/ldap_ca.crt
# Set search_scope to one of them: BASE, LEVEL, SUBTREE
# Set search_scope to SUBTREE if using Active Directory, and not specifying an Organizational
Unit
search_scope = LEVEL
```

superuser_filter 和 data_profiler_filter 是可选的。如果已定义，则这些配置允许您指定用户必须属于的 LDAP 组，以便拥有超级用户（admin）和数据分析权限。如果未定义，则所有用户都将成为超级用户和拥有数据分析的权限。

创建自定义

Airflow 使用了 flask_login 扩展包并在 airflow.default_login 模块中公开了一组钩子。您可以更改内容并使其成为 PYTHONPATH 一部分，并将其配置为 airflow.cfg 的认证后端。

```
[webserver]
authenticate = True
auth_backend = mypackage.auth
```

多租户

通过在配置中设置 webserver:filter_by_owner，可以在启用身份验证时按所有者名称筛选 webserver:filter_by_owner。有了这个，用户将只看到他所有者的 dags，除非他是超级用户。

```
[webserver]
filter_by_owner = True
```

Kerberos

Airflow 天然支持 Kerberos。这意味着 Airflow 可以为自己更新 kerberos 票证并将其存储在票证缓存中。钩子和 dags 可以使用票证来验证 kerberized 服务。

限制

请注意，此时并未调整所有钩子以使用此功能。此外，它没有将 kerberos 集成到 Web 界面中，您现在必须依赖网络级安全性来确保您的服务保持安全。

Celery 集成尚未经过试用和测试。 但是，如果您为每个主机生成一个 key tab，并在每个 Worker 旁边启动一个 ticket renewer，那么它很可能会起作用。

启用 Kerberos

Airflow

要启用 kerberos，您需要生成（服务）key tab。

```
# in the kadmin.local or kadmin shell, create the airflow principal
kadmin:  addprinc -randkey airflow/fully.qualified.domain.name@YOUR-REALM.COM

# Create the airflow keytab file that will contain the airflow principal
kadmin:  xst -norandkey -k airflow.keytab airflow/fully.qualified.domain.name
```

现在将此文件存储在 airflow 用户可以读取的位置（chmod 600）。然后将以下内容添加到 airflow.cfg

```
[core]
security = kerberos

[kerberos]
keytab = /etc/airflow/airflow.keytab
reinit_frequency = 3600
principal = airflow
```

启动 ticket renewer

```
# run ticket renewer
airflow Kerberos
```

Hadoop

如果要使用模拟，则需要在 hadoop 配置的 core-site.xml 中启用。

```
<property>
  <name>hadoop.proxyuser.airflow.groups</name>
  <value>*</value>
</property>

<property>
  <name>hadoop.proxyuser.airflow.users</name>
  <value>*</value>
</property>

<property>
  <name>hadoop.proxyuser.airflow.hosts</name>
  <value>*</value>
</property>
```

当然，如果您需要加强安全性，请用更合适的东西替换星号。

使用 kerberos 身份验证

已更新 Hive 钩子以利用 kerberos 身份验证。要允许 DAG 使用它，只需更新连接详细信息，例如：

```
{"use_beeline": true, "principal": "hive/_HOST@EXAMPLE.COM"}
```

请根据您的设置进行调整。_HOST 部分将替换为服务器的完全限定域名。

您可以指定是否要将 dag 所有者用作连接的用户或连接的登录部分中指定的用户。对于登录用户，请将以下内容指定为额外：

```
{"use_beeline": true, "principal": "hive/_HOST@EXAMPLE.COM", "proxy_user": "login"}
```

对于 DAG 所有者使用：

```
{"use_beeline": true, "principal": "hive/_HOST@EXAMPLE.COM", "proxy_user": "owner"}
```

在 DAG 中，初始化 HiveOperator 时，请指定：

```
run_as_owner = True
```

为了使用 kerberos 认证，请务必在安装 Airflow 时添加 kerberos 子扩展包：

```
pip install airflow[kerberos]
```

OAuth 认证

GitHub Enterprise（GHE）认证

GitHub Enterprise 认证后端可用于对使用 OAuth2 安装 GitHub Enterprise 的用户进行身份认证。您可以选择指定团队白名单（由 slug cased 团队名称组成）以限制仅允许这些团队的成员登录。

```
[webserver]
authenticate = True
auth_backend = airflow.contrib.auth.backends.github_enterprise_auth

[github_enterprise]
host = github.example.com
client_id = oauth_key_from_github_enterprise
client_secret = oauth_secret_from_github_enterprise
oauth_callback_route = /example/ghe_oauth/callback
allowed_teams = 1, 345, 23
```

注意:如果您未指定团队白名单，那么在 GHE 安装中拥有有效帐户的任何人都可以登录 Airflow。

设置 GHE 身份验证

必须先在 GHE 中设置应用程序，然后才能使用 GHE 身份验证后端。 要设置应用程序：

1. 导航到您的 GHE 配置文件
2. 从左侧导航栏中选择“应用程序”

3. 选择“开发者应用程序”选项卡
4. 点击“注册新申请”
5. 填写所需信息（“授权回调 URL”必须完全合格，例如 http://airflow.example.com/example/ghe_oauth/callback ）
6. 点击“注册申请”
7. 根据上面的示例，将“客户端 ID”,“客户端密钥”和回调路由复制到 airflow.cfg

在 github.com 上使用 GHE 身份验证

可以在 github.com 上使用 GHE 身份验证：

1. 创建一个 Oauth 应用程序
2. 根据上面的示例，将“客户端 ID”,“客户端密钥”复制到 airflow.cfg
3. 在 airflow.cfg 设置 host = github.com 和 oauth_callback_route = /oauth/callback

Google 身份验证

Google 身份验证后端可用于使用 OAuth2 对 Google 用户进行身份验证。您必须指定电子邮件域以限制登录（以逗号分隔），仅限于这些域的成员。

```
[webserver]
authenticate = True
auth_backend = airflow.contrib.auth.backends.google_auth

[google]
client_id = google_client_id
client_secret = google_client_secret
oauth_callback_route = /oauth2callback
domain = "example1.com,example2.com"
```

设置 Google 身份验证

必须先在 Google API 控制台中设置应用程序，然后才能使用 Google 身份验证后端。 要设置应用程序：

1. 导航到 https://console.developers.google.com/apis/
2. 从左侧导航栏中选择“凭据”
3. 点击“创建凭据”，然后选择“OAuth 客户端 ID”
4. 选择“Web 应用程序”
5. 填写所需信息（’授权重定向 URI’必须完全合格，例如 http://airflow.example.com/oauth2callback ）
6. 点击“创建”
7. 根据上面的示例，将“客户端 ID”,“客户端密钥”和重定向 URI 复制到 airflow.cfg

SSL

可以通过提供证书和密钥来启用 SSL。启用后，请务必在浏览器中使用“https//”。

```
[webserver]
web_server_ssl_cert = <path to cert>
web_server_ssl_key = <path to key>
```

启用 SSL 不会自动更改 Web 服务器端口。如果要使用标准端口 443，则还需要配置它。请注意，侦听端口 443 需要超级用户权限（或 Linux 上的 cap_net_bind_service）。

```
# Optionally, set the server to listen on the standard SSL port.
web_server_port = 443
base_url = http://<hostname or IP>:443
```

使用 SSL 启用 CeleryExecutor。 确保正确生成客户端和服务器证书和密钥。

```
[celery]
CELERY_SSL_ACTIVE = True
CELERY_SSL_KEY = <path to key>
CELERY_SSL_CERT = <path to cert>
CELERY_SSL_CACERT = <path to cacert>
```

模拟

Airflow 能够在运行任务实例时模拟 unix 用户，该任务实例基于任务的 run_as_user 参数，该参数采用用户的名称。

注意： 要模拟工作，必须使用 sudo 运行 Airflow，因为使用 sudo -u 运行子任务并更改文件的权限。此外，unix 用户需要存在于 worker 中。假设 airflow 是 airflow 用户运行，一个简单的 sudoers 文件可能看起来像这样。请注意，这意味着必须以与 root 用户相同的方式信任和处理 airflow 用户。

```
airflow ALL=(ALL) NOPASSWD: ALL
```

带模拟的子任务仍将记录到同一文件夹，但他们登录的文件将更改权限，只有 unix 用户才能写入。

默认模拟

要防止不使用模拟的任务以 sudo 权限运行，如果未设置 run_as_user，可以在 core:default_impersonation 配置中来设置默认的模拟用户。

```
[core]
default_impersonation = airflow
```

Flower 认证

Celery Flower 的基础认证是支持的。

可以在 Flower 进程启动命令中指定可选参数，或者在 airflow.cfg 指定配置项。对于这两种情况，请提供用逗号分隔的 user:password 对。

```
airflow flower --basic_auth=user1:password1,user2:password2
[celery]
flower_basic_auth = user1:password1,user2:password2
```

时区

默认情况下启用对时区的支持。 Airflow 在内部处理和数据库中以 UTC 格式存储日期时间信息。 这样您就可以在调度中执行带有时区的 DAG。 目前，Airflow 不支持在 Web UI 中显示最终用户时区，它始终以 UTC 显示。 此外，Operator 中使用的模板也不会被转换。 时区信息取决于 DAG 作者如何使用它。

如果您的用户居住在多个时区，并且您希望根据每个用户的挂钟(wall clock)显示日期时间信息，这将非常方便。

即使您只在一个时区运行 Airflow，在数据库中以 UTC 格式存储数据仍然是一种很好的做法（在 Airflow 关心时区问题之前也是如此，这也是推荐的甚至是必需的设置）。 主要原因是夏令时（DST）。 许多国家都有 DST 系统，其中时间在春季向前移动，在秋季向后移动。 如果您在当地工作，那么当发生转换时，您可能每年会遇到两次错误。 （pendulum 和 pytz 文档更详细地讨论了这些问题。）这可能并不影响简单的 DAG，但如果您涉及金融服务中，这可能就变成灾难。

时区在 airflow.cfg 中设置。默认情况下，它设置为 utc，您可以将其更改为使用系统设置或任意 IANA 时区，例如 Europe/Amsterdam (阿姆斯特丹)。 安装 Airflow 时会安装 Pendulum，它比 pytz 更精确，也会让您更容易处理时区问题。

请注意，Web UI 目前仅显示 UTC 时间。

概念

了解 datetime 时区敏感对象

tzinfo 是 datetime.datetime 对象中表示时区信息的属性，它是 datetime.tzinfo 类的对象。 设置 tzinfo 会使 datetime 对象变为时区敏感，否则就是不敏感。

您可以使用 timezone.is_aware()和 timezone.is_naive()来确定 datetime 是否时区敏感。

因为 Airflow 使用时区敏感 datetime 对象，所以您创建 datetime 对象时也应该考虑时区问题。

```
from airflow.utils import timezone

now = timezone.utcnow()
```

a_date = timezone.datetime(2017,1,1)

理解原生 datetime 对象

尽管 Airflow 现在是时区敏感的，但为了向后兼容，它仍然允许在 DAG 定义中使用原生 datetime 为 start_dates 和 end_dates 赋值。 如果遇到使用原生 datetime 的 start_date 或 end_date 则使用默认时区。换句话说，如果您设置了默认时区为 Europe/Amsterdam，那么对于 start_date = datetime(2017,1,1)，则认定它是 2017 年 1 月 1 日的 Amesterdam 时间。

```
default_args = dict(
    start_date = datetime(2016, 1, 1),
    owner = 'Airflow'
)

dag = DAG ('my_dag', default_args=default_args)
op = DummyOperator(task_id='dummy', dag=dag)
print(op.owner) # Airflow
```

不幸的是，在 DST 转换期间，某些时间不存在或有二义性。 在这种情况下，pendulum 会抛出异常。 这就是为什么在启用时区支持时应始终创建时区敏感的 datetime 对象。

实际上，这个问题几乎不会发生。 在 model 或 DAG 中，Airflow 总是返回给您的是时区敏感的 datetime 对象，通常新的 datetime 对象是通过现有对象和 timedelta 计算而来的。 唯一经常创建并且可能出问题的是当前时间，用 timezone.utcnow()会避免这些麻烦。

默认时区

默认时区是在 airflow.cfg 中的[core]下的 default_timezone 定义的。 如果您刚刚安装了 Airflow，它推荐您设置为 utc。 您还可以将其设置为系统或 IANA 时区（例如 Europe/Amsterdam）。 DAG 也会在 worker 上进行计算，因此确保所有 worker 节点上的此设置相同。

```
[core]
default_timezone = utc
```

时区敏感 DAG

只需设置 start_date 为时区敏感的 datetime，就创建了时区敏感的 DAG。 建议使用 pendulum，但也可以使用 pytz（手动安装）。

```
import pendulum

local_tz = pendulum.timezone("Europe/Amsterdam")

default_args = dict (
```

```
    start_date = datetime(2016, 1, 1, tzinfo=local_tz),
    owner = 'Airflow'
)

dag = DAG('my_tz_dag', default_args=default_args)
op = DummyOperator(task_id='dummy', dag=dag)
print(dag.timezone) # <Timezone [Europe/Amsterdam]>
```

模板

Airflow 在模板中返回时区敏感的 datetime,但不会将它们转换为本地时间,因此它们仍然是 UTC 时区。由 DAG 来处理转换。

```
import pendulum

local_tz = pendulum.timezone("Europe/Amsterdam")
local_tz.convert(execution_date)
```

Cron 安排

如果您设置了 crontab，Airflow 会假定您始终希望在同一时间运行。 然后它将忽略 DST。 比如，您有一个调度要在每天格林威治标准时间 +1 的 08:00 运行,它将始终在这个时间运行,无论 DST 是否发生。

时间增量调度

对于具有时间增量的调度,Airflow 假定您始终希望以指定的间隔运行。比如,您指定 timedelta(hours=2)运行，它将始终每两小时运行一次。 在这种情况下，将考虑 DST。

实验性 Rest API

Airflow 公开了一个实验性的 Rest API。它可以通过 Web 服务器获得来请求。 API 端点以/api/experimental/开头，同时请注意，我们也希望端点定义发生变化。

端点

这是占位符，直到 swagger 定义处于活动状态

- /api/experimental/dags//tasks/ 返回任务信息（GET）。
- /api/experimental/dags//dag_runs 为给定的 dag id 创建一个 dag_run（POST）。

CLI

对于某些功能，cli 可以使用 API。 要配置 CLI 选项使得在可用时能够使用 API，请按如下方式配置：

```
[cli]
api_client = airflow.api.client.json_client
endpoint_url = http://<WEBSERVER>:<PORT>
```

认证

API 的身份验证与 Web 身份验证分开处理。 默认情况下，不需要对 API 进行任何身份验证 - 即默认情况下全开。 如果您的 Airflow 网络服务器可公开访问，那么不建议这样做，您应该使用拒绝所有后端请求：

```
[api]
auth_backend = airflow.api.auth.backend.deny_all
```

API 目前支持两种“真实”的身份验证方法。

要启用密码身份验证，请在配置中进行以下设置：

```
[api]
auth_backend = airflow.contrib.auth.backends.password_auth
```

它的用法类似于用于 Web 界面的密码验证。

要启用 Kerberos 身份验证，请在配置中设置以下内容：

```
[api]
auth_backend = airflow.api.auth.backend.kerberos_auth

[kerberos]
keytab = <KEYTAB>
```

Kerberos 服务配置为 airflow/fully.qualified.domainname@REALM。确保密钥表文件中存在此配置。

集成

1.反向代理

Airflow 可以通过设置反向代理，使其可以灵活设置访问地址。

例如，您可以这样配置反向代理：

```
https://lab.mycompany.com/myorg/airflow/
```

为此，您需要在 airflow.cfg 中设置：

```
base_url = http://my_host/myorg/airflow
```

此外，如果您使用 Celery Executor，您可以配置 myorg/flower 的地址：

```
flower_url_prefix = /myorg/flower
```

您的反向代理（例如：nginx）应配置如下：

- 传递 url 和 http 头给 Airflow 服务器，不需要重写，例如：

```
server {
  listen 80;
  server_name lab.mycompany.com;

  location /myorg/airflow/ {
      proxy_pass http://localhost:8080;
      proxy_set_header Host $host;
      proxy_redirect off;
      proxy_http_version 1.1;
      proxy_set_header Upgrade $http_upgrade;
      proxy_set_header Connection "upgrade";
  }
}
```

- 重写 flower 的地址：

```
server {
    listen 80;
    server_name lab.mycompany.com;

    location /myorg/flower/ {
        rewrite ^/myorg/flower/(.*)$ /$1 break;  # remove prefix from http header
        proxy_pass http://localhost:5555;
        proxy_set_header Host $host;
        proxy_redirect off;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
    }
}
```

Azure：Microsoft Azure

Airflow 对 Microsoft Azure 的支持有限：仅支持 Azure Blob Storage 和 Azure Data Lake 的接口。 对

Blob Storage 的 Hook, Sensor 和 Operator 以及 Azure Data Lake 的 Hook 都在 contrib 部分。

Azure Blob Storage

所有类都通过 Window Azure Storage Blob 协议进行通信。 确保 Airflow 的 wasb 连接存在。 可以通过在额外字段中指定登录用户名（=Storage account name）和密码（=KEY），或用 SAS 令牌来完成授权（参看 wasb_default 连接例子）。

- WasbBlobSensor ： 检查一个 blob 是否在 Azure Blob Storage 上。
- WasbPrefixSensor ： 检查满足前缀匹配的 blob 是否在 Azure Blob Storage 上。
- FileToWasbOperator ： 将本地文件作为 blob 上传到容器。
- WasbHook ： 与 Azure Blob Storage 的接口。

WasbBlobSensor

```
class airflow.contrib.sensors.wasb_sensor.WasbBlobSensor(container_name,blob_name, wasb_conn_id='wasb_default', check_options=None, *args, **kwargs)
```

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

等待 blob 到达 Azure Blob Storage。

参数：

- container_name(str) - 容器的名称。
- blob_name(str) - blob 的名称。
- wasb_conn_id(str) - 对 wasb 连接的引用。
- check_options(dict) - 传给 WasbHook.check_for_blob()的可选关键字参数。

```
poke(context)
```

Operator 在继承此类时应该覆盖以上函数。

WasbPrefixSensor

```
class airflow.contrib.sensors.wasb_sensor.WasbPrefixSensor(container_name, prefix, wasb_conn_id='wasb_default', check_options=None, *args, **kwargs)
```

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

等待前缀匹配的 blob 到达 Azure Blob Storage。

参数：

- container_name(str) - 容器的名称。
- prefix(str) - blob 的前缀。
- wasb_conn_id(str) - 对 wasb 连接的引用。
- check_options(dict) - 传给 WasbHook.check_for_prefix()的可选关键字参数。

poke(context)

Operator 在继承此类时应该覆盖以上函数。

FileToWasbOperator

class airflow.contrib.operators.file_to_wasb.FileToWasbOperator(file_path, container_name, blob_name, wasb_conn_id='wasb_default', load_options=None, *args, **kwargs)

基类： airflow.models.BaseOperator

将文件上传到 Azure Blob Storage。

参数：

- file_path(str) - 要加载的文件的路径。（模板渲染后）
- container_name(str) - 容器的名称。（模板渲染后）
- blob_name(str) - blob 的名称。（模板渲染后）
- wasb_conn_id(str) - 对 wasb 连接的引用。
- load_options(dict) - 传给 WasbHook.load_file()的可选关键字参数。

execute(context)

将文件上传到 Azure Blob Storage。

WasbHook

class airflow.contrib.hooks.wasb_hook.WasbHook(wasb_conn_id='wasb_default')

基类： airflow.hooks.base_hook.BaseHook

通过 wasb:// 协议与 Azure Blob Storage 进行交互。

在连接的"extra"字段中传递的其他参数将传递给 BlockBlockService()构造函数。 例如，通过添加{"sas_token": "YOUR_TOKEN"}使用 SAS 令牌进行身份验证。

参数：wasb_conn_id(str) - 对 wasb 连接的引用。

check_for_blob(container_name, blob_name, **kwargs)

检查 Azure Blob Storage 上是否存在 Blob。

参数：

- container_name(str) - 容器的名称。
- blob_name(str) - blob 的名称。
- kwargs(object) - 传给 BlockBlobService.exists()的可选关键字参数。

返回：如果 blob 存在则为 True，否则为 False。

返回类型：bool

`check_for_prefix(container_name, prefix, **kwargs)`

检查 Azure Blob Storage 上是否存在前缀匹配的 blob。

参数：

- container_name(str) - 容器的名称。
- prefix(str) - blob 的前缀。
- kwargs(object) - 传给 BlockBlobService.list_blobs()的可选关键字参数。

返回：如果存在与前缀匹配的 blob，则为 True，否则为 False。

返回类型：bool

`get_conn()`

返回 BlockBlobService 对象。

`get_file(file_path, container_name, blob_name, **kwargs)`

从 Azure Blob Storage 下载文件。

参数：

- file_path(str) - 要下载的文件的路径。
- container_name(str) - 容器的名称。
- blob_name(str) - blob 的名称。
- kwargs(object) - 传给 BlockBlobService.create_blob_from_path()的可选关键字参数。

load_file(file_path, container_name, blob_name, **kwargs)

将文件上传到 Azure Blob Storage。

参数：

- file_path(str) – 要加载的文件的路径。
- container_name(str) – 容器的名称。
- blob_name(str) – blob 的名称。
- kwargs(object) – 传给 BlockBlobService.create_blob_from_path()的可选关键字参数。

load_string(string_data, container_name, blob_name, **kwargs)

将字符串上传到 Azure Blob Storage。

参数：

- string_data(str) – 要上传的字符串。
- container_name(str) – 容器的名称。
- blob_name(str) – blob 的名称。
- kwargs(object) – 传给 BlockBlobService.create_blob_from_text()的可选关键字参数。

read_file(container_name, blob_name, **kwargs)

从 Azure Blob Storage 读取文件并以字符串形式返回。

参数：

- container_name(str) – 容器的名称。
- blob_name(str) – blob 的名称。
- kwargs(object) – 传给 BlockBlobService.create_blob_from_path()的可选关键字参数。

Azure File Share

SMB 文件共享的云变体。 确保 Airflow 存在类型为 wasb 的连接。 可以通过在额外字段中提供登录（=Storage 帐户名称）和密码（=Storage 帐户密钥）或通过 SAS 令牌来完成授权（请参阅连接 wasb_default 示例）。

AzureFileShareHook

```
class
airflow.contrib.hooks.azure_fileshare_hook.AzureFileShareHook(wasb_conn_id='wasb_default')
```

基类： airflow.hooks.base_hook.BaseHook

与 Azure FileShare Storage 交互。

在连接的"extra"字段中传递的参数将传递给 FileService()构造函数。

参数：wasb_conn_id(str) - 对 wasb 连接的引用。

check_for_directory(share_name, directory_name, **kwargs)

检查 Azure File Share 上是否存在目录。

参数：

- share_name(str) - 共享的名称。
- directory_name(str) - 目录的名称。
- kwargs(object) - 传给 FileService.exists()的可选关键字参数。

返回：如果文件存在则为 True，否则为 False。

返回类型：bool

check_for_file(share_name, directory_name, file_name, **kwargs)

检查 Azure File Share 上是否存在文件。

参数：

- share_name(str) - 共享的名称。
- directory_name(str) - 目录的名称。
- file_name(str) - 文件名。
- kwargs(object) - 传给 FileService.exists()的可选关键字参数。

返回：如果文件存在则为 True，否则为 False。

返回类型：bool

create_directory(share_name, directory_name, **kwargs)

在 Azure File Share 上创建目录。

参数：

share_name(str) - 共享的名称。
directory_name(str) - 目录的名称。
kwargs(object) - 传给 FileService.create_directory()的可选关键字参数。

返回：文件和目录列表

返回类型：list

`get_conn()`

返回 FileService 对象。

`get_file(file_path, share_name, directory_name, file_name, **kwargs)`

从 Azure File Share 下载文件。

参数：

- file_path(str) – 存储文件的位置。
- share_name(str) – 共享的名称。
- directory_name(str) – 目录的名称。
- file_name(str) – 文件名。
- kwargs(object) – 传给 FileService.get_file_to_path()的可选关键字参数。

`get_file_to_stream(stream, share_name, directory_name, file_name, **kwargs)`

以流的形式从 Azure File Share 下载文件。

参数：

- stream(类文件对象) – 用于存储文件的文件句柄。
- share_name(str) – 共享的名称。
- directory_name(str) – 目录的名称。
- file_name(str) – 文件名。
- kwargs(object) – 传给 FileService.get_file_to_stream()的可选关键字参数。

list_directories_and_files(share_name, directory_name=None, **kwargs)

返回存储在 Azure File Share 中的目录和文件列表。

参数：

- share_name(str) – 共享的名称。
- directory_name(str) – 目录的名称。
- kwargs(object) – 传给 FileService.list_directories_and_files()的可选关键字参数。

返回：文件和目录列表

返回类型：list

load_file(file_path, share_name, directory_name, file_name, **kwargs)

将文件上传到 Azure File Share。

参数：

- file_path(str) - 要上传的文件路径。
- share_name(str) - 共享的名称。
- directory_name(str) - 目录的名称。
- file_name(str) - 文件名。
- kwargs(object) - 传给 FileService.create_file_from_path()的可选关键字参数。

load_stream(stream, share_name, directory_name, file_name, count, **kwargs)

将流上传到 Azure File Share。

参数：

- stream(类文件对象) - 打开的文件/流作为文件内容上传。
- share_name(str) - 共享的名称。
- directory_name(str) - 目录的名称。
- file_name(str) - 文件名。
- count(int) - 流的大小（以字节为单位）
- kwargs(object) - 传给 FileService.create_file_from_stream()的可选关键字参数。

load_string(string_data, share_name, directory_name, file_name, **kwargs)

将字符串上传到 Azure File Share。

参数：

- string_data(str) - 要加载的字符串。
- share_name(str) - 共享的名称。
- directory_name(str) - 目录的名称。
- file_name(str) - 文件名。
- kwargs(object) - 传给 FileService.create_file_from_text()的可选关键字参数。

Logging

可以将 Airflow 配置为在 Azure Blob Storage 中读取和写入任务日志。 请参阅将日志写入 Azure Blob Storage 。

Azure Data Lake

AzureDataLakeHook 通过与 WebHDFS 兼容的 REST API 进行通信。 确保 Airflow 存在 azure_data_lake 类型的连接。 可以通过提供用户名（=客户端 ID），密码（=客户端密钥），额外字段可以提供租户和帐户名称来完成授权。

- AzureDataLakeHook：与 Azure Data Lake 的接口。

AzureDataLakeHook

```
class airflow.contrib.hooks.azure_data_lake_hook.AzureDataLakeHook(azure_data_lake_conn_id='azure_data_lake_default')
```

基类： airflow.hooks.base_hook.BaseHook

与 Azure Data Lake 进行交互。

客户端 ID 和客户端密钥应该在用户和密码参数中。 租户和帐户名称应为额外字段，如 {"tenant": "", "account_name": "ACCOUNT_NAME"}

参数：azure_data_lake_conn_id(str) - 对 Azure Data Lake 连接的引用。

```
check_for_file(file_path)
```

检查 Azure Data Lake 上是否存在文件。

参数：file_path(str) - 文件的路径和名称。

返回：如果文件存在则为 True，否则为 False。

返回类型：bool

```
download_file(local_path, remote_path, nthreads=64, overwrite=True, buffersize=4194304, blocksize=4194304)
```

从 Azure Blob Storage 下载文件。

参数：

- local_path(str) - 本地路径。 如果下载单个文件，将写入此文件，如果它是现有目录，将在其中创建文件。 如果下载多个文件，这是要写入的根目录。将根据需要创建目录。
- remote_path(str) - 用于查找远程文件的远程路径，可以使用通配符。 不支持使用**的递归 glob 模式。

- nthreads(int) - 要使用的线程数。 如果为 None，则使用核心数。
- overwrite(bool) - 是否强制覆盖现有文件/目录。 如果 False 并且远程路径是目录，则无论是否覆盖任何文件都将退出。 如果为 True，则实际仅覆盖匹配的文件名。
- buffersize(int) - 支持最大 2 ** 22 内部缓冲区的字节数。 block 不能大于 trunk。
- blocksize(int) - 一个 block 支持最大 2 ** 22 字节数。 在每个 trunk 中，我们为每个 API 调用写一个较小的 block。 这个 block 不能大于 trunk。

get_conn()

返回 AzureDLFileSystem 对象。

upload_file(local_path, remote_path, nthreads=64, overwrite=True, buffersize=4194304, blocksize=4194304)

将文件上传到 Azure Data Lake。

参数：

- local_path(str) - 本地路径。 可以是单个文件，目录（在这种情况下，递归上传）或 glob 模式。不支持使用**的递归 glob 模式。
- remote_path(str) - 要上传的远程路径; 如果有多个文件，这就是要写入的根目录。
- nthreads(int) - 要使用的线程数。 如果为 None，则使用核心数。
- overwrite(bool) - 是否强制覆盖现有文件/目录。 如果 False 并且远程路径是目录，则无论是否覆盖任何文件都将退出。 如果为 True，则实际仅覆盖匹配的文件名。
- buffersize(int) - 支持最大 2 ** 22 内部缓冲区的字节数。 block 不能大于 trunk。
- blocksize(int) - 一个 block 支持最大 2 ** 22 字节数。 在每个 trunk 中，我们为每个 API 调用写一个较小的 block。 这个 block 不能大于 trunk。

AWS: Amazon Web Services

Airflow 大量支持 Amazon Web Services。 但请注意，Hook，Sensors 和 Operators 都在 contrib 部分。

AWS EMR

- EmrAddStepsOperator ： 向现有 EMR JobFlow 添加步骤。
- EmrCreateJobFlowOperator ： 创建 EMR JobFlow，从 EMR 连接读取配置。
- EmrTerminateJobFlowOperator ： 终止 EMR JobFlow。
- EmrHook ： 与 AWS EMR 互动。

EmrAddStepsOperator

class airflow.contrib.operators.emr_add_steps_operator.EmrAddStepsOperator(job_flow_id, aws_conn_id='s3_default', steps=None, *args, **kwargs)

基类： airflow.models.BaseOperator

向现有 EMR job_flow 添加步骤的 operator。

参数：

job_flow_id - 要添加步骤的 JobFlow 的 ID。（模板渲染后）
aws_conn_id(str) - 与使用的 aws 连接
steps(list) - 要添加到作业流的 boto3 步骤。 （模板渲染后）

EmrCreateJobFlowOperator

class airflow.contrib.operators.emr_create_job_flow_operator.EmrCreateJobFlowOperator(aws_conn_id='s3_default', emr_conn_id='emr_default', job_flow_overrides=None, *args, **kwargs)

基类： airflow.models.BaseOperator

创建 EMR JobFlow，从 EMR 连接读取配置。 可以向 JobFlow 传递参数以覆盖连接中的配置。

参数：

- aws_conn_id(str) - 要使用的 aws 连接
- emr_conn_id(str) - 要使用的 emr 连接
- job_flow_overrides - 用于覆盖 emr_connection extra 的 boto3 式参数。 （模板渲染后）

EmrTerminateJobFlowOperator

class airflow.contrib.operators.emr_terminate_job_flow_operator.EmrTerminateJobFlowOperator(job_flow_id, aws_conn_id='s3_default', *args, **kwargs)

基类： airflow.models.BaseOperator

终止 EMR JobFlows 的 operator。

参数：

- job_flow_id - 要终止的 JobFlow 的 id。（模板渲染后）
- aws_conn_id(str) - 要使用的 aws 连接

EmrHook

class airflow.contrib.hooks.emr_hook.EmrHook(emr_conn_id=None, *args, **kwargs)

基类： airflow.contrib.hooks.aws_hook.AwsHook

与 AWS EMR 交互。 emr_conn_id 是使用 create_job_flow 方法唯一必需的。

```
create_job_flow(job_flow_overrides)
```

使用 EMR 连接中的配置创建作业流。 json_flow_overrrides 是传给 run_job_flow 方法的参数。

AWS S3

- S3Hook ：与 AWS S3 交互。
- S3FileTransformOperator ：将数据从 S3 源位置复制到本地文件系统上的临时位置。
- S3ListOperator ：列出与 S3 位置的键前缀匹配的文件。
- S3ToGoogleCloudStorageOperator ：将 S3 位置与 Google 云端存储分区同步。
- S3ToHiveTransfer ：将数据从 S3 移动到 Hive。 operator 从 S3 下载文件，在将文件加载到 Hive 表之前将其存储在本地。

S3Hook

```
class airflow.hooks.S3_hook.S3Hook(aws_conn_id='aws_default')
```

基类： airflow.contrib.hooks.aws_hook.AwsHook

使用 boto3 库与 AWS S3 交互。

```
check_for_bucket(bucket_name)
```

检查 bucket_name 是否存在。

参数：bucket_name(str) - 存储桶的名称

```
check_for_key(key, bucket_name=None)
```

检查存储桶中是否存在密钥

参数：

- key(str) - 指向文件的 S3 的 key
- bucket_name(str) - 存储桶的名称

```
check_for_prefix(bucket_name, prefix, delimiter)
```

检查存储桶中是否存在前缀

```
check_for_wildcard_key(wildcard_key, bucket_name=None, delimiter='')
```

检查桶中是否存在与通配符表达式匹配的密钥

```
get_bucket(bucket_name)
```

返回 boto3.S3.Bucket 对象

参数：bucket_name(str) - 存储桶的名称

```
get_key(key, bucket_name=None)
```

返回 boto3.s3.Object

参数：

- key(str) - 密钥的路径
- bucket_name(str) - 存储桶的名称

```
get_wildcard_key(wildcard_key, bucket_name=None, delimiter='')
```

返回与通配符表达式匹配的 boto3.s3.Object 对象

参数：

- wildcard_key(str) - 密钥的路径
- bucket_name(str) - 存储桶的名称

```
list_keys(bucket_name, prefix='', delimiter='', page_size=None, max_items=None)
```

列出前缀下的存储桶中的密钥，但不包含分隔符

参数：

- bucket_name(str) - 存储桶的名称
- prefix(str) - 一个密钥前缀
- delimiter(str) - 分隔符标记键层次结构。
- page_size(int) - 分页大小
- max_items(int) - 要返回的最大项目数

```
list_prefixes(bucket_name, prefix='', delimiter='', page_size=None, max_items=None)
```

列出前缀下的存储桶中的前缀

参数：

- bucket_name(str) - 存储桶的名称
- prefix(str) - 一个密钥前缀
- delimiter(str) - 分隔符标记键层次结构。
- page_size(int) - 分页大小
- max_items(int) - 要返回的最大项目数

load_bytes(bytes_data, key, bucket_name=None, replace=False, encrypt=False)

将字节加载到 S3

这是为了方便在 S3 中删除字符串。 它使用 boto 基础结构将文件发送到 s3。

参数：

- bytes_data(bytes) - 设置为密钥内容的字节。
- key(str) - 指向文件的 S3 键
- bucket_name(str) - 存储桶的名称
- replace(bool) - 一个标志，用于决定是否覆盖密钥（如果已存在）
- encrypt(bool) - 如果为 True，则文件将在服务器端由 S3 加密，并在 S3 中静止时以加密形式存储。

load_file(filename, key, bucket_name=None, replace=False, encrypt=False)

将本地文件加载到 S3

参数：

- filename(str) - 要加载的文件的名称。
- key(str) - 指向文件的 S3 键
- bucket_name(str) - 存储桶的名称
- replace(bool)- 一个标志，用于决定是否覆盖密钥（如果已存在）。 如果 replace 为 False 且密钥存在，则会引发错误。
- encrypt(bool) - 如果为 True，则文件将在服务器端由 S3 加密，并在 S3 中静止时以加密形式存储。

load_string(string_data, key, bucket_name=None, replace=False, encrypt=False, encoding='utf-8')

将字符串加载到 S3

这是为了方便在 S3 中删除字符串。 它使用 boto 基础结构将文件发送到 s3。

参数：

- string_data(str) – 要设置为键的内容的字符串。
- key(str) – 指向文件的 S3 键
- bucket_name(str) – 存储桶的名称
- replace(bool) – 一个标志，用于决定是否覆盖密钥（如果已存在）
- encrypt(bool) – 如果为 True，则文件将在服务器端由 S3 加密，并在 S3 中静止时以加密形式存储。

```
read_key(key, bucket_name=None)
```

从 S3 读取密钥

参数：

- key(str) – 指向文件的 S3 键
- bucket_name(str) – 存储桶的名称

```
select_key(key, bucket_name=None, expression='SELECT * FROM S3Object', expression_type='SQL', input_serialization={'CSV': {}}, output_serialization={'CSV': {}})
```

使用 S3 Select 读取密钥。

参数：

- key(str) – 指向文件的 S3 键
- bucket_name(str) – 存储桶的名称
- expression(str) – S3 选择表达式
- expression_type(str) – S3 选择表达式类型
- input_serialization(dict) – S3 选择输入数据序列化格式
- output_serialization(dict) – S3 选择输出数据序列化格式

返回：通过 S3 Select 检索原始数据的子集

返回类型：str

S3FileTransformOperator

```
class    airflow.operators.s3_file_transform_operator.S3FileTransformOperator(source_s3_key, dest_s3_key, transform_script=None, select_expression=None, source_aws_conn_id='aws_default', dest_aws_conn_id='aws_default', replace=False, *args, **kwargs)
```

基类： airflow.models.BaseOperator

将数据从源 S3 位置复制到本地文件系统上的临时位置。 用转换脚本对此文件运行转换，并将输出上传到目标 S3 位置。

本地文件系统中的源文件和目标文件的位置作为转换脚本的第一个和第二个参数提供。 转换脚本应该从源读取数据，转换它并将输出写入本地目标文件。 然后，operator 将本地目标文件上传到 S3。

S3 Select 也可用于过滤源内容。 如果指定了 S3 Select 表达式，则用户可以省略转换脚本。

参数：

- source_s3_key(str) - 源 S3 的密钥。（模板渲染后）
- source_aws_conn_id(str) - 源 s3 连接
- dest_s3_key(str) - 目的 S3 的密钥。（模板渲染后）
- dest_aws_conn_id(str) - 目标 s3 连接
- replace(bool) - 替换 dest S3 密钥（如果已存在）
- transform_script(str) - 可执行转换脚本的位置
- select_expression(str) - S3 选择表达式

S3ListOperator

```
class airflow.contrib.operators.s3listoperator.S3ListOperator(bucket, prefix='', delimiter='', aws_conn_id='aws_default', *args, **kwargs)
```

基类： airflow.models.BaseOperator

列出桶中具有给定前缀的所有对象。

此 operator 返回一个 python 列表，其中包含可由 xcom 在下游任务中使用的对象名称。

参数：

- bucket(str) - S3 存储桶在哪里找到对象。（模板渲染后）
- prefix(str) - 用于过滤名称以此字符串为前缀的对象。（模板渲染后）
- delimiter(str) - 分隔符标记键层次结构。（模板渲染后）
- aws_conn_id(str) - 连接到 S3 存储时使用的连接 ID。

以下 operator 将列出 data 存储区中 S3 customers/2018/04/ key 的所有文件（不包括子文件夹）。

```
s3_file = S3ListOperator (
    task_id = 'list_3s_files' ,
    bucket = 'data' ,
    prefix = 'customers/2018/04/' ,
    delimiter = '/' ,
    aws_conn_id = 'aws_customers_conn'
```

)

S3ToGoogleCloudStorageOperator

class airflow.contrib.operators.s3_to_gcs_operator.S3ToGoogleCloudStorageOperator(bucket, prefix='', delimiter='', aws_conn_id='aws_default', dest_gcs_conn_id=None, dest_gcs=None, delegate_to=None, replace=False, *args, **kwargs)

基类： airflow.contrib.operators.s3listoperator.S3ListOperator

将 S3 密钥（可能是前缀）与 Google 云端存储目标路径同步。

参数：

- bucket(str) - S3 存储桶在哪里找到对象。（模板渲染后）
- prefix(str) - 用于过滤名称以此字符串为前缀的对象。（模板渲染后）
- delimiter(str) - 分隔符标记键层次结构。（模板渲染后）
- aws_conn_id(str) - 源 S3 连接
- dest_gcs_conn_id(str) - 连接到 Google 云端存储时要使用的目标连接 ID。
- dest_gcs(str) - 要存储文件的目标 Google 云端存储分区和前缀。（模板渲染后）
- delegate_to(str) - 代理的帐户（如果有）。 为此，发出请求的服务帐户必须启用域范围委派。
- replace(bool) - 是否要替换现有目标文件。

例子

```
s3_to_gcs_op = S3ToGoogleCloudStorageOperator(
task_id ='s3_to_gcs_example', bucket ='my-s3-bucket', prefix ='data / customers-201804',
dest_gcs_conn_id ='google_cloud_default', dest_gcs ='gs: //my.gcs.bucket/some/customers/' ,
replace = False, dag = my-dag)
```

请注意， bucket ， prefix ， delimiter 和 dest_gcs 是模板化的，因此如果您愿意，可以在其中使用变量。

S3ToHiveTransfer

class airflow.operators.s3_to_hive_operator.S3ToHiveTransfer(s3_key, field_dict, hive_table, delimiter=',', create=True, recreate=False, partition=None, headers=False, check_headers=False, wildcard_match=False, aws_conn_id='aws_default', hive_cli_conn_id='hive_cli_default', input_compressed=False, tblproperties=None, select_expression=None, *args, **kwargs)

基类： airflow.models.BaseOperator

将数据从 S3 移动到 Hive。 operator 从 S3 下载文件，在将文件加载到 Hive 表之前将其存储在本地。如果 create 或 recreate 参数设置为 True ，则会生成 CREATE TABLE 和 DROP TABLE 语句。 Hive 数据类

型是从游标的元数据中推断出来的。

请注意，Hive 中生成的表使用 STORED AS textfile ，这不是最有效的序列化格式。 如果加载了大量数据和/或表格被大量查询，您可能只想使用此 operator 将数据暂存到临时表中，然后使用 HiveOperator 将其加载到最终目标表中。

参数：

- s3_key(str) - 从 S3 检索的密钥。（模板渲染后）
- field_dict(dict) - 字段的字典在文件中命名为键，其 Hive 类型为值
- hive_table(str) - 目标 Hive 表，使用点表示法来定位特定数据库。（模板渲染后）
- create(bool) - 是否创建表，如果它不存在
- recreate(bool) - 是否在每次执行时删除并重新创建表
- partition(dict) - 将目标分区作为分区列和值的字典。（模板渲染后）
- headers(bool) - 文件是否包含第一行的列名
- check_headers(bool) - 是否应该根据 field_dict 的键检查第一行的列名
- wildcard_match(bool) - 是否应将 s3_key 解释为 Unix 通配符模式
- delimiter(str) - 文件中的字段分隔符
- aws_conn_id(str) - 源 s3 连接
- hive_cli_conn_id(str) - 目标配置单元连接
- input_compressed(bool) - 布尔值，用于确定是否需要文件解压缩来处理标头
- tblproperties(dict) - 正在创建的 hive 表的 TBLPROPERTIES
- select_expression(str) - S3 选择表达式

AWS EC2 容器服务

- ECSOperator ：在 AWS EC2 容器服务上执行任务。

ECSOperator

```
class airflow.contrib.operators.ecs_operator.ECSOperator(task_definition, cluster, overrides, aws_conn_id=None, region_name=None, launch_type='EC2', **kwargs)
```

基类： airflow.models.BaseOperator

在 AWS EC2 Container Service 上执行任务

参数：

- task_definition(str) - EC2 容器服务上的任务定义名称
- cluster(str) - EC2 Container Service 上的集群名称
- aws_conn_id(str) - AWS 的连接 ID。 如果为 None ，将使用 boto3 凭证（ http://boto3.readthedocs.io/en/latest/guide/configuration.html ）。
- region_name - 要在 AWS Hook 中使用的 region 名称。 覆盖连接中的 region_name（如果提供）

- launch_type - 运行任务的启动类型（'EC2'或'FARGATE'）

参数：boto3 将接收的相同参数（模板化）：http://boto3.readthedocs.org/en/latest/reference/services/ecs.html#ECS.Client.run_task

类型：dict

类型：launch_type：str

AWS Batch Service

- AWSBatchOperator ：在 AWS Batch Service 上执行任务。

AWSBatchOperator

```
class airflow.contrib.operators.awsbatch_operator.AWSBatchOperator(job_name, job_definition, job_queue, overrides, max_retries=4200, aws_conn_id=None, region_name=None, **kwargs)
```

基类： airflow.models.BaseOperator

在 AWS Batch Service 上执行作业

参数：

- job_name(str) - 将在 AWS Batch 上运行的作业的名称
- job_definition(str) - AWS Batch 上的作业定义名称
- job_queue(str) - AWS Batch 上的队列名称
- max_retries(int) - 服务器未合并时的指数退避重试，4200 = 48 小时
- aws_conn_id(str) - AWS 的连接 ID。 如果为 None，将使用凭证 boto3 策略（ http://boto3.readthedocs.io/en/latest/guide/configuration.html ）。
- region_name - 要在 AWS Hook 中使用的区域名称。 覆盖连接中的 region_name（如果提供）

参数：boto3 将在 containerOverrides 上接收的相同参数（模板化）：http://boto3.readthedocs.io/en/latest/reference/services/batch.html#submit_job

类型：dict

AWS RedShift

- AwsRedshiftClusterSensor ：等待 Redshift 集群达到特定状态。
- RedshiftHook ：使用 boto3 库与 AWS Redshift 交互。
- RedshiftToS3Transfer ：对带有或不带标头的 CSV 执行卸载命令。
- S3ToRedshiftTransfer ：从 S3 执行复制命令为 CSV，带或不带标题。

AwsRedshiftClusterSensor

class airflow.contrib.sensors.aws_redshift_cluster_sensor.AwsRedshiftClusterSensor(cluster_identifier, target_status='available', aws_conn_id='aws_default', *args, **kwargs)

基类： [airflow.sensors.base_sensor_operator.BaseSensorOperator]

等待 Redshift 集群达到特定状态。

参数：

- cluster_identifier(str) - 要 ping 的集群的标识符。
- target_status(str) - 所需的集群状态。

poke(context)

Operator 在继承此类时应该覆盖以上函数。

RedshiftHook

class airflow.contrib.hooks.redshift_hook.RedshiftHook(aws_conn_id='aws_default')

基类： [airflow.contrib.hooks.aws_hook.AwsHook]

使用 boto3 库与 AWS Redshift 交互

cluster_status(cluster_identifier)

返回集群的状态

参数：cluster_identifier(str) - 集群的唯一标识符

create_cluster_snapshot(snapshot_identifier, cluster_identifier)

创建集群的快照

参数：

- snapshot_identifier(str) - 集群快照的唯一标识符
- cluster_identifier(str) - 集群的唯一标识符

delete_cluster(cluster_identifier, skip_final_cluster_snapshot=True, final_cluster_snapshot_identifier='')

删除集群并可选择创建快照

参数：

- cluster_identifier(str) - 集群的唯一标识符
- skip_final_cluster_snapshot(bool) - 确定集群快照创建
- final_cluster_snapshot_identifier(str) - 最终集群快照的名称

```
describe_cluster_snapshots(cluster_identifier)
```

获取集群的快照列表

参数：cluster_identifier(str) - 集群的唯一标识符

```
restore_from_cluster_snapshot(cluster_identifier, snapshot_identifier)
```

从其快照还原集群

参数：

- cluster_identifier(str) - 集群的唯一标识符
- snapshot_identifier(str) - 集群快照的唯一标识符

RedshiftToS3Transfer

```
class airflow.operators.redshift_to_s3_operator.RedshiftToS3Transfer(schema，table，s3_bucket，s3_key，redshift_conn_id='redshift_default'，aws_conn_id='aws_default'，unload_options=()，autocommit = False，parameters = None，include_header = False，*args，**kwargs)
```

基类： airflow.models.BaseOperator

执行 UNLOAD 命令，将 s3 作为带标题的 CSV

参数：

- schema(str) - 对 redshift 数据库中特定模式的引用
- table(str) - 对 redshift 数据库中特定表的引用
- s3_bucket(str) - 对特定 S3 存储桶的引用
- s3_key(str) - 对特定 S3 密钥的引用
- redshift_conn_id(str) - 对特定 redshift 数据库的引用
- aws_conn_id(str) - 对特定 S3 连接的引用
- unload_options(list) - 对 UNLOAD 选项列表的引用

S3ToRedshiftTransfer

```
class airflow.operators.s3_to_redshift_operator.S3ToRedshiftTransfer(schema，table，s3_bucket，s3_key，redshift_conn_id ='redshift_default'，aws_conn_id ='aws_default'，copy_options =()，autocommit = False，parameters = None，*args，**kwargs)
```

基类： airflow.models.BaseOperator

执行 COPY 命令将文件从 s3 加载到 Redshift

参数：

- schema(str) - 对 redshift 数据库中特定模式的引用
- table(str) - 对 redshift 数据库中特定表的引用
- s3_bucket(str) - 对特定 S3 存储桶的引用
- s3_key(str) - 对特定 S3 密钥的引用
- redshift_conn_id(str) - 对特定 redshift 数据库的引用
- aws_conn_id(str) - 对特定 S3 连接的引用
- copy_options(list) - 对 COPY 选项列表的引用

Databricks

Databricks 贡献了一个 Airflow operator，可以将运行提交到 Databricks 平台。在运营商内部与 api/2.0/jobs/runs/submit 端点进行通信。

DatabricksSubmitRunOperator

```
class airflow.contrib.operators.databricks_operator.DatabricksSubmitRunOperator(json = None，spark_jar_task = None，notebook_task = None，new_cluster = None，existing_cluster_id = None，libraries = None ， run_name = None ， timeout_seconds = None ， databricks_conn_id ='databricks_default'，polling_period_seconds = 30，databricks_retry_limit = 3，do_xcom_push = False，**kwargs)
```

基类： airflow.models.BaseOperator

使用 api / 2.0 / jobs / runs / submit API 端点向 Databricks 提交 Spark 作业运行。

有两种方法可以实例化此 operator。

在第一种方式，你可以把你通常用它来调用的 JSON 有效载荷 api/2.0/jobs/runs/submit 端点并将其直接传递到我们 DatabricksSubmitRunOperator 通过 json 参数。例如

```
json = {
  'new_cluster' : {
```

```
    'spark_version': '2.1.0-db3-scala2.11',
    'num_workers': 2
  },
  'notebook_task': {
    'notebook_path': '/Users/airflow@example.com/PrepareData',
  },
}
notebook_run = DatabricksSubmitRunOperator ( task_id = 'notebook_run', json = json )
```

另一种完成同样事情的方法是直接使用命名参数 DatabricksSubmitRunOperator。请注意，runs/submit 端点中的每个顶级参数都只有一个命名参数。在此方法中，您的代码如下所示：

```
 new_cluster = {
  'spark_version': '2.1.0-db3-scala2.11',
  'num_workers': 2
}
notebook_task = {
  'notebook_path': '/Users/airflow@example.com/PrepareData',
}
notebook_run = DatabricksSubmitRunOperator (
    task_id = 'notebook_run',
    new_cluster = new_cluster ,
    notebook_task = notebook_task )
```

在提供 json 参数和命名参数的情况下，它们将合并在一起。如果在合并期间存在冲突，则命名参数将优先并覆盖顶级 json 键。

目前 DatabricksSubmitRunOperator 支持的命名参数是

- spark_jar_task
- notebook_task
- new_cluster
- existing_cluster_id
- libraries
- run_name
- timeout_seconds

参数：

- json(dict) -

包含 API 参数的 JSON 对象，将直接传递给 api/2.0/jobs/runs/submit 端点。其他命名参数（即 spark_jar_task，notebook_task..）到该运营商将与此 JSON 字典合并如果提供他们。如果在合并期间存在冲突，则命名参数将优先并覆盖顶级 json 键。（模板渲染后）

- spark_jar_task(dict) -

JAR 任务的主要类和参数。请注意，实际的 JAR 在 libraries。中指定。_ 无论是 _spark_jar_task_ 或 _ notebook_task 应符合规定。该字段将被模板化。

- notebook_task(dict) -

笔记本任务的笔记本路径和参数。_ 无论是 _ spark_jar_task _ 或 _ notebook_task 应符合规定。该字段将被模板化。

- new_cluster(dict) -

将在其上运行此任务的新集群的规范。_ 无论是 _ new_cluster _ 或 _ existing_cluster_id 应符合规定。该字段将被模板化。

- existing_cluster_id(str) - 要运行此任务的现有集群的 ID。_ 无论是 _ new_cluster _ 或 _ existing_cluster_id 应符合规定。该字段将被模板化。

- libraries(list 或 dict) -

这个运行的库将使用。该字段将被模板化。

- run_name(str) - 用于此任务的运行名称。默认情况下，这将设置为 Airflow task_id。这 task_id 是超类 BaseOperator 的必需参数。该字段将被模板化。

- timeout_seconds(int32) - 此次运行的超时。默认情况下，使用值 0 表示没有超时。该字段将被模板化。

- databricks_conn_id(str) - 要使用的 Airflow 连接的名称。默认情况下，在常见情况下，这将是 databricks_default。要使用基于令牌的身份验证，请在连接的额外字段 token 中提供密钥。

- polling_period_seconds(int) - 控制我们轮询此运行结果的速率。默认情况下，operator 每 30 秒轮询一次。

- databricks_retry_limit(int) - 如果 Databricks 后端无法访问，则重试的次数。其值必须大于或等于 1。

- do_xcom_push(bool) - 我们是否应该将 run_id 和 run_page_url 推送到 xcom。

GCP：Google 云端平台

Airflow 广泛支持 Google Cloud Platform。但请注意，大多数 Hooks 和 Operators 都在 contrib 部分。这意味着他们具有 beta 状态，这意味着他们可以在次要版本之间进行重大更改。

请参阅 GCP 连接类型文档以配置与 GCP 的连接。

记录

可以将 Airflow 配置为在 Google 云端存储中读取和写入任务日志。请参阅将日志写入 Google 云端存储。

BigQuery 的

BigQuery Operator

- BigQueryCheckOperator：对 SQL 查询执行检查，该查询将返回具有不同值的单行。
- BigQueryValueCheckOperator：使用 SQL 代码执行简单的值检查。
- BigQueryIntervalCheckOperator：检查作为 SQL 表达式给出的度量值是否在 days_back 之前的某个容差范围内。
- BigQueryCreateEmptyTableOperator：在指定的 BigQuery 数据集中创建一个新的空表，可选择使用模式。
- BigQueryCreateExternalTableOperator：使用 Google Cloud Storage 中的数据在数据集中创建新的外部表。
- BigQueryDeleteDatasetOperator：删除现有的 BigQuery 数据集。
- BigQueryOperator：在特定的 BigQuery 数据库中执行 BigQuery SQL 查询。
- BigQueryToBigQueryOperator：将 BigQuery 表复制到另一个 BigQuery 表。
- BigQueryToCloudStorageOperator：将 BigQuery 表传输到 Google Cloud Storage 存储桶

BigQueryCheckOperator

```
class      airflow.contrib.operators.bigquery_check_operator.BigQueryCheckOperator(sql     , bigquery_conn_id ='bigquery_default', *args, **kwargs)
```

基类：[airflow.operators.check_operator.CheckOperator]

对 BigQuery 执行检查。该 BigQueryCheckOperator 预期的 SQL 查询将返回一行。使用 python bool 强制转换第一行的每个值。如果任何值返回，False 则检查失败并输出错误。

请注意，Python bool 强制转换如下 False：

- False
- 0
- 空字符串（""）
- 空列表（[]）
- 空字典或集（{}）

给定一个查询，SELECT COUNT(*) FROM foo 当且仅当== 0 时会失败。您可以制作更复杂的查询，例如，可以检查表与上游源表的行数相同，或者今天的分区计数大于昨天的分区，或者一组指标是否更少 7 天平均值超过 3 个标准差。

此 Operator 可用作管道中的数据质量检查，并且根据您在 DAG 中的位置，您可以选择在关键路径停止，防止发布可疑数据，或者接收电子邮件报警而不阻止 DAG 的继续。

参数：

- sql(str) - 要执行的 sql
- bigquery_conn_id(str) - 对 BigQuery 数据库的引用

BigQueryValueCheckOperator

```
class airflow.contrib.operators.bigquery_check_operator.BigQueryValueCheckOperator(sql , pass_value, tolerance = None, bigquery_conn_id ='bigquery_default', *args, **kwargs)
```

基类：[airflow.operators.check_operator.ValueCheckOperator]

使用 sql 代码执行简单的值检查。

参数：sql(str) - 要执行的 sql

BigQueryIntervalCheckOperator

```
class airflow.contrib.operators.bigquery_check_operator.BigQueryIntervalCheckOperator(table, metrics_thresholds , date_filter_column ='ds' , days_back = -7 , bigquery_conn_id ='bigquery_default', *args, **kwargs)
```

基类：[airflow.operators.check_operator.IntervalCheckOperator]

检查作为 SQL 表达式给出的度量值是否在 days_back 之前的某个容差范围内。

此方法构造一个类似的查询

```
SELECT { metrics_thresholddictkey } FROM { table }
    WHERE { date_filter_column } =< date >
```

参数：

- table(str) - 表名
- days_back(int) - ds 与我们要检查的 ds 之间的天数。默认为 7 天
- metrics_threshold(dict) - 由指标索引的比率字典，例如'COUNT(*)'：1.5 将需要当前日和之前的 days_back 之间 50%或更小的差异。

BigQueryGetDataOperator

```
class airflow.contrib.operators.bigquery_get_data.BigQueryGetDataOperator(dataset_id ,
```

```
table_id, max_results ='100', selected_fields = None, bigquery_conn_id ='bigquery_default', delegate_to = None, *args, **kwargs)
```

基类： airflow.models.BaseOperator

从 BigQuery 表中获取数据（或者为所选列获取数据）并在 python 列表中返回数据。返回列表中的元素数将等于获取的行数。列表中的每个元素将是一个列表，其中元素将表示该行的列值。

结果示例：[['Tony', '10'], ['Mike', '20'], ['Steve', '15']]

注意:如果传递的字段 selected_fields 的顺序与 BQ 表中已有的列的顺序不同，则数据仍将按 BQ 表的顺序排列。例如，如果 BQ 表有 3 列，[A,B,C]并且您传递'B, A', selected_fields 仍然是表格'A,B'。

示例 ：

```
get_data = BigQueryGetDataOperator (
    task_id = 'get_data_from_bq' ,
    dataset_id = 'test_dataset' ,
    table_id = 'Transaction_partitions' ,
    max_results = '100' ,
    selected_fields = 'DATE' ,
    bigquery_conn_id = 'airflow-service-account'
)
```

参数：

- dataset_id - 请求的表的数据集 ID。（模板渲染后）
- table_id(str) - 请求表的表 ID。（模板渲染后）
- max_results(str) - 从表中获取的最大记录数（行数）。（模板渲染后）
- selected_fields(str) - 要返回的字段列表（逗号分隔）。如果未指定，则返回所有字段。
- bigquery_conn_id(str) - 对特定 BigQuery 的引用。
- delegate_to(str) - 代理的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。

BigQueryCreateEmptyTableOperator

```
class airflow.contrib.operators.bigquery_operator.BigQueryCreateEmptyTableOperator(dataset_id , table_id, project_id = None, schema_fields = None, gcs_schema_object = None, time_partitioning = {} , bigquery_conn_id ='bigquery_default' , google_cloud_storage_conn_id ='google_cloud_default', delegate_to = None, *args , **kwargs)
```

基类： airflow.models.BaseOperator

在指定的 BigQuery 数据集中创建一个新的空表，可选择使用模式。

可以用两种方法之一指定用于 BigQuery 表的模式。您可以直接传递架构字段，也可以将运营商指向 Google 云存储对象名称。Google 云存储中的对象必须是包含架构字段的 JSON 文件。您还可以创建没有架构的表。

参数：

- project_id(str) - 将表创建的项目。（模板渲染后）
- dataset_id(str) - 用于创建表的数据集。（模板渲染后）
- table_id(str) - 要创建的表的名称。（模板渲染后）
- schema_fields(list) -

如果设置，则此处定义的架构字段列表：https://cloud.google.com/bigquery/docs/reference/rest/v2/jobs#configuration.load.schema

示例 ：

```
schema_fields = [{ "name" : "emp_name" , "type" : "STRING" , "mode" : "REQUIRED" },
                 { "name" : "salary" , "type" : "INTEGER" , "mode" : "NULLABLE" }]
```

- gcs_schema_object(str) - 包含模式（模板化）的 JSON 文件的完整路径。例如：gs://test-bucket/dir1/dir2/employee_schema.json
- time_partitioning(dict) -

配置可选的时间分区字段，即按 API 规范按字段，类型和到期分区。

- bigquery_conn_id(str) - 对特定 BigQuery 的引用。
- google_cloud_storage_conn_id(str) - 对特定 Google 云存储的引用。
- delegate_to(str) - 代理的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。

示例（在 GCS 中使用 JSON schema）：

```
CreateTable = BigQueryCreateEmptyTableOperator (
    task_id = 'BigQueryCreateEmptyTableOperator_task',
    dataset_id = 'ODS',
    table_id = 'Employees',
    project_id = 'internal-gcp-project',
    gcs_schema_object = 'gs://schema-bucket/employee_schema.json',
    bigquery_conn_id = 'airflow-service-account',
```

```
    google_cloud_storage_conn_id = 'airflow-service-account'
)
```

对应的 Schema 文件（employee_schema.json）：

```
 [
  {
    "mode" : "NULLABLE" ,
    "name" : "emp_name" ,
    "type" : "STRING"
  },
  {
    "mode" : "REQUIRED" ,
    "name" : "salary" ,
    "type" : "INTEGER"
  }
]
```

示例（在 DAG 中使用 schema）：

```
 CreateTable = BigQueryCreateEmptyTableOperator (
    task_id = 'BigQueryCreateEmptyTableOperator_task' ,
    dataset_id = 'ODS' ,
    table_id = 'Employees' ,
    project_id = 'internal-gcp-project' ,
    schema_fields = [{ "name" : "emp_name" , "type" : "STRING" , "mode" : "REQUIRED" },
                     { "name" : "salary" , "type" : "INTEGER" , "mode" : "NULLABLE" }],
    bigquery_conn_id = 'airflow-service-account' ,
    google_cloud_storage_conn_id = 'airflow-service-account'
)
```

BigQueryCreateExternalTableOperator

```
class airflow.contrib.operators.bigquery_operator.BigQueryCreateExternalTableOperator(bucket, source_objects, destination_project_dataset_table, schema_fields=None, schema_object=None, source_format='CSV', compression='NONE', skip_leading_rows=0, field_delimiter=',', max_bad_records=0, quote_character=None, allow_quoted_newlines=False, allow_jagged_rows=False, bigquery_conn_id='bigquery_default', google_cloud_storage_conn_id='google_cloud_default', delegate_to=None, src_fmt_configs={}, *args, **kwargs)
```

基类： airflow.models.BaseOperator

使用 Google 云端存储中的数据在数据集中创建新的外部表。

可以用两种方法之一指定用于 BigQuery 表的模式。您可以直接传递架构字段，也可以将运营商指向 Google 云存储对象名称。Google 云存储中的对象必须是包含架构字段的 JSON 文件。

参数：

- bucket(str) - 指向外部表的存储桶。（模板渲染后）
- source_objects - 指向表格的 Google 云存储 URI 列表。（模板化）如果 source_format 是 'DATASTORE_BACKUP'，则列表必须只包含一个 URI。
- destination_project_dataset_table(str) - 用于将数据加载到（模板化）的表\1。
- BigQuery 表。如果未包\1
- schema_fields(list) -

如果设置，则此处定义的架构字段列表：https://cloud.google.com/bigquery/docs/reference/rest/v2/jobs#configuration.load.schema

示例 ：

```
schema_fields = [{ "name" : "emp_name" , "type" : "STRING" , "mode" : "REQUIRED" },
                 { "name" : "salary" , "type" : "INTEGER" , "mode" : "NULLABLE" }]
```

当 source_format 为'DATASTORE_BACKUP'时，不应设置。

- schema_object - 如果设置，则指向包含表的架构的.json 文件的 GCS 对象路径。（模板渲染后）

- schema_object - 字符串

- source_format(str) - 数据的文件格式。
- compression(str) -[可选]数据源的压缩类型。可能的值包括 GZIP 和 NONE。默认值为 NONE。Google Cloud Bigtable，Google Cloud Datastore 备份和 Avro 格式会忽略此设置。
- skip_leading_rows(int) - 从 CSV 加载时要跳过的行数。
- field_delimiter(str) - 用于 CSV 的分隔符。
- max_bad_records(int) - BigQuery 在运行作业时可以忽略的最大错误记录数。
- quote_character(str) - 用于引用 CSV 文件中数据部分的值。
- allow_quoted_newlines(bool) - 是否允许引用的换行符（true）或不允许（false）。
- allow_jagged_rows(bool) - 接受缺少尾随可选列的行。缺失值被视为空值。如果为 false，则缺少尾随列的记录将被视为错误记录，如果错误记录太多，则会在作业结果中返回无效错误。仅适用于 CSV，忽略其他格式。
- bigquery_conn_id(str) - 对特定 BigQuery 挂钩的引用。
- google_cloud_storage_conn_id(str) - 对特定 Google 云存储挂钩的引用。
- delegate_to(str) - 代理的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。
- src_fmt_configs(dict) - 配置特定于源格式的可选字段

BigQueryDeleteDatasetOperator

BigQueryOperator

```
class airflow.contrib.operators.bigquery_operator.BigQueryOperator(bql = None, sql = None, destination_dataset_table = False, write_disposition = 'WRITE_EMPTY', allow_large_results = False, flatten_results = False, bigquery_conn_id = 'bigquery_default', delegate_to = None, udf_config = False , use_legacy_sql = True, maximum_billing_tier = None, maximumbytesbilled = None, create_disposition = 'CREATE_IF_NEEDED', schema_update_options = (), query_params = None, priority = 'INTERACTIVE', time_partitioning = {}, *args, **kwargs)
```

基类： airflow.models.BaseOperator

在特定的 BigQuery 数据库中执行 BigQuery SQL 查询

参数：

- BQL(可接收表示 SQL 语句中的海峡，海峡列表（SQL 语句），或参照模板文件模板引用在“.SQL”结束海峡认可。）- （不推荐使用。SQL 参数代替）要执行的 sql 代码（模板化）
- SQL(可接收表示 SQL 语句中的海峡，海峡列表（SQL 语句），或参照模板文件模板引用在“.SQL”结束海峡认可。）- SQL 代码被执行（模板渲染后）
- destination_dataset_table(str) - 目的数据集表（
- write_disposition(str) - 指定目标表已存在时发生的操作。（默认：'WRITE_EMPTY'）
- create_disposition(str) - 指定是否允许作业创建新表。（默认值：'CREATE_IF_NEEDED'）
- allow_large_results(bool) - 是否允许大结果。
- flatten_results(bool) - 如果为 true 且查询使用旧版 SQL 方言，则展平查询结果中的所有嵌套和重复字段。allow_large_results 必须是 true 如果设置为 false。对于标准 SQL 查询，将忽略此标志，并且结果永远不会展平。
- bigquery_conn_id(str) - 对特定 BigQuery 钩子的引用。
- delegate_to(str) - 代理的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。
- udf_config(list) - 查询的用户定义函数配置。有关详细信息，请参阅 https://cloud.google.com/bigquery/user-defined-functions。
- use_legacy_sql(bool) - 是使用旧 SQL（true）还是标准 SQL（false）。
- maximum_billing_tier(int) - 用作基本价格乘数的正整数。默认为 None，在这种情况下，它使用项目中设置的值。
- maximumbytesbilled(float) - 限制为此作业计费的字节数。超出此限制的字节数的查询将失败（不会产生费用）。如果未指定，则将其设置为项目默认值。
- schema_update_options(tuple) - 允许更新目标表的模式作为加载作业的副作用。
- query_params(dict) - 包含查询参数类型和值的字典，传递给 BigQuery。
- priority(str) - 指定查询的优先级。可能的值包括 INTERACTIVE 和 BATCH。默认值为 INTERACTIVE。
- time_partitioning(dict) - 配置可选的时间分区字段，即按 API 规范按字段，类型和到期分区。请注意，'field'不能与 dataset.table $ partition 一起使用。

BigQueryTableDeleteOperator

```
class
```

airflow.contrib.operators.bigquery_table_delete_operator.BigQueryTableDeleteOperator(deletion_dataset_table, bigquery_conn_id='bigquery_default', delegate_to=None, ignore_if_missing=False, *args, **kwargs)

基类： airflow.models.BaseOperator

删除 BigQuery 表

参数：

- deletion_dataset_table(str) - 删除的数据集表（
- bigquery_conn_id(str) - 对特定 BigQuery 钩子的引用。
- delegate_to(str) - 代理的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。
- ignore_if_missing(bool) - 如果为 True，则即使请求的表不存在也返回成功。

BigQueryToBigQueryOperator

class airflow.contrib.operators.bigquery_to_bigquery.BigQueryToBigQueryOperator(source_project_dataset_tables , destination_project_dataset_table , write_disposition ='WRITE_EMPTY' , create_disposition='CREATE_IF_NEEDED',bigquery_conn_id='bigquery_default',delegate_to=None, *args, **kwargs)

基类： airflow.models.BaseOperator

将数据从一个 BigQuery 表复制到另一个。

参数：

- source_project_dataset_tables(list|str) - 一个或多个点(project:|project.)
- destination_project_dataset_table(str) - 目标 BigQuery 表。格式为：（project:|project）.
- （模板化）
- write_disposition(str) - 表已存在时的处理。
- create_disposition(str) - 如果表不存在，则创建处理。
- bigquery_conn_id(str) - 对特定 BigQuery 的引用。
- delegate_to(str) - 代理的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。

BigQueryToCloudStorageOperator

class airflow.contrib.operators.bigquery_to_gcs.BigQueryToCloudStorageOperator(source_project_dataset_table , destination_cloud_storage_uris , compression ='NONE' , export_format ='CSV' , field_delimiter=',', print_header=True, bigquery_conn_id='bigquery_default', delegate_to=

None，*args， **kwargs)

基类： airflow.models.BaseOperator

将 BigQuery 表传输到 Google Cloud Storage 存储桶。

参数：

- source_project_dataset_table(str) - 用作源数据的(:)
- destination_cloud_storage_uris(list) - 目标 Google 云端存储 URI（例如 gs://some-bucket/some-file.txt ）。（模板化）遵循此处定义的惯例： https：//cloud.google.com/bigquery/exporting-data-from-bigquery#exportingmultiple
- compression(str) - 要使用的压缩类型。
- export_format - 要导出的文件格式。
- field_delimiter(str) - 提取到 CSV 时使用的分隔符。
- print_header(bool) - 是否打印 CSV 文件头。
- bigquery_conn_id(str) - 对特定 BigQuery 的引用。
- delegate_to(str) - 代理的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。

BigQueryHook

class airflow.contrib.hooks.bigquery_hook.BigQueryHook(bigquery_conn_id ='bigquery_default', delegate_to = None, use_legacy_sql = True)

基类： [airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook] , [airflow.hooks.dbapi_hook.DbApiHook], airflow.utils.log.logging_mixin.LoggingMixin

与 BigQuery 交互。此挂钩使用 Google Cloud Platform 连接。

get_conn()

返回 BigQuery PEP 249 连接对象。

get_pandas_df(sql, parameters = None, dialect = None)

返回 BigQuery 查询生成的结果的 Pandas DataFrame。必须重写 DbApiHook 方法，因为 Pandas 不支持 PEP 249 连接，但 SQLite 除外。参看：

https://github.com/pydata/pandas/blob/master/pandas/io/sql.py#L447
https://github.com/pydata/pandas/issues/6900

参数：

- sql(str) - 要执行的 BigQuery SQL。

- parameters(map 或 iterable) - 用于呈现 SQL 查询的参数（未使用，请保留覆盖超类方法）
- dialect({'legacy', 'standard'}) - BigQuery SQL 的方言 - 遗留 SQL 或标准 SQL 默认使用 self.use_legacy_sql（如果未指定）

```
get_service()
```

返回一个 BigQuery 服务对象。

```
insert_rows(table, rows, target_fields = None, commit_every = 1000)
```

目前不支持插入。从理论上讲，您可以使用 BigQuery 的流 API 将行插入表中，但这尚未实现。

```
table_exists(project_id, dataset_id, table_id)
```

检查 Google BigQuery 中是否存在表格。

参数：

- project_id(str) - 要在其中查找表的 Google 云项目。连接必须有对指定项目的访问权限。
- dataset_id(str) - 要在其中查找表的数据集的名称。
- table_id(str) - 要检查的表的名称。

云 DataFlow

DataFlow Operator

- DataFlowJavaOperator：启动用 Java 编写的 Cloud Dataflow 作业。
- DataflowTemplateOperator：启动模板化的 Cloud DataFlow 批处理作业。
- DataFlowPythonOperator：启动用 python 编写的 Cloud Dataflow 作业。

DataFlowJavaOperator

```
class airflow.contrib.operators.dataflow_operator.DataFlowJavaOperator(jar , dataflow_default_options = None,options = None,gcp_conn_id ='google_cloud_default',delegate_to = None, poll_sleep = 10, job_class = None, *args, **kwargs)
```

基类： airflow.models.BaseOperator

启动 Java Cloud DataFlow 批处理作业。操作的参数将传递给作业。

在 dag 的 default_args 中定义 dataflow_*参数是一个很好的做法，例如项目，区域和分段位置。

```
 default_args = {
    'dataflow_default_options' : {
        'project' : 'my-gcp-project' ,
```

```
        'zone' : 'europe-west1-d',
        'stagingLocation' : 'gs://my-staging-bucket/staging/'
    }
}
```

您需要使用 jar 参数将路径作为文件引用传递给数据流，jar 需要是一个自动执行的 jar（请参阅以下文档：https://beam.apache.org/documentation/runners/dataflow/。使用 options 传递你的参数。

```
t1 = DataFlowOperation (
    task_id = 'datapflow_example',
    jar = '{{var.value.gcp_dataflow_base}}pipeline/build/libs/pipeline-example-1.0.jar',
    options = {
        'autoscalingAlgorithm' : 'BASIC',
        'maxNumWorkers' : '50',
        'start' : '{{ds}}',
        'partitionType' : 'DAY',
        'labels' : { 'foo' : 'bar' }
    },
    gcp_conn_id = 'gcp-airflow-service-account',
    dag = my - dag )
```

这两个 jar 和 options 模板化，所以你可以在其中使用变量。

```
default_args = {
    'owner' : 'airflow',
    'depends_on_past' : False ,
    'start_date' :
        ( 2016 , 8 , 1 ),
    'email' : [ 'alex@vanboxel.be' ],
    'email_on_failure' : False ,
    'email_on_retry' : False ,
    'retries' : 1 ,
    'retry_delay' : timedelta ( minutes = 30 ),
    'dataflow_default_options' : {
        'project' : 'my-gcp-project',
        'zone' : 'us-central1-f',
        'stagingLocation' : 'gs://bucket/tmp/dataflow/staging/',
    }
}

dag = DAG ( 'test-dag', default_args = default_args )

task = DataFlowJavaOperator (
    gcp_conn_id = 'gcp_default',
```

```
    task_id = 'normalize-cal',
    jar = '{{var.value.gcp_dataflow_base}}pipeline-ingress-cal-normalize-1.0.jar',
    options = {
        'autoscalingAlgorithm' : 'BASIC',
        'maxNumWorkers' : '50',
        'start' : '{{ds}}',
        'partitionType' : 'DAY'

    },
    dag = dag )
```

DataflowTemplateOperator

```
class     airflow.contrib.operators.dataflow_operator.DataflowTemplateOperator(template     ,
dataflow_default_options = None, parameters = None, gcp_conn_id ='google_cloud_default',
delegate_to = None, poll_sleep = 10, *args, **kwargs)
```

基类： airflow.models.BaseOperator

启动模板化云 DataFlow 批处理作业。操作的参数将传递给作业。在 dag 的 default_args 中定义 dataflow_ *参数是一个很好的做法，例如项目，区域和分段位置。

也可以看看

https://cloud.google.com/dataflow/docs/reference/rest/v1b3/LaunchTemplateParameters
https://cloud.google.com/dataflow/docs/reference/rest/v1b3/RuntimeEnvironment

```
default_args = {
    'dataflow_default_options' : {
        'project' : 'my-gcp-project'
        'zone' : 'europe-west1-d',
        'tempLocation' : 'gs://my-staging-bucket/staging/'
        }
    }
}
```

您需要将路径作为带 template 参数的文件引用传递给数据流模板。使用 parameters 来传递参数给你的工作。使用 environment 对运行环境变量传递给你的工作。

```
t1 = DataflowTemplateOperator (
    task_id = 'datapflow_example',
    template = '{{var.value.gcp_dataflow_base}}',
    parameters = {
        'inputFile' : "gs://bucket/input/my_input.txt",
```

```
        'outputFile' : "gs://bucket/output/my_output.txt"
    },
    gcp_conn_id = 'gcp-airflow-service-account' ,
    dag = my - dag )
```

template，dataflow_default_options 并且 parameters 是模板化的，因此您可以在其中使用变量。

DataFlowPythonOperator

```
class airflow.contrib.operators.dataflow_operator.DataFlowPythonOperator(py_file,py_options = None, dataflow_default_options = None, options = None, gcp_conn_id ='google_cloud_default', delegate_to = None, poll_sleep = 10, *args, **kwargs)
```

基类： airflow.models.BaseOperator

```
execute(context)
```

执行 python 数据流作业。

DataFlowHook

```
class airflow.contrib.hooks.gcp_dataflow_hook.DataFlowHook(gcp_conn_id ='google_cloud_default', delegate_to = None, poll_sleep = 10)
```

基类： [airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook]

```
get_conn()
```

返回 Google 云端存储服务对象。

Cloud DataProc

DataProc Operator

- DataprocClusterCreateOperator：在 Google Cloud Dataproc 上创建新集群。
- DataprocClusterDeleteOperator：删除 Google Cloud Dataproc 上的集群。
- DataprocClusterScaleOperator：在 Google Cloud Dataproc 上向上或向下扩展集群。
- DataProcPigOperator：在 Cloud DataProc 集群上启动 Pig 查询作业。
- DataProcHiveOperator：在 Cloud DataProc 集群上启动 Hive 查询作业。
- DataProcSparkSqlOperator：在 Cloud DataProc 集群上启动 Spark SQL 查询作业。
- DataProcSparkOperator：在 Cloud DataProc 集群上启动 Spark 作业。
- DataProcHadoopOperator：在 Cloud DataProc 集群上启动 Hadoop 作业。
- DataProcPySparkOperator：在 Cloud DataProc 集群上启动 PySpark 作业。
- DataprocWorkflowTemplateInstantiateOperator ： 在 Google Cloud Dataproc 上实例化

WorkflowTemplate。

- DataprocWorkflowTemplateInstantiateInlineOperator：在 Google Cloud Dataproc 上实例化 WorkflowTemplate 内联。

DataprocClusterCreateOperator

class airflow.contrib.operators.dataproc_operator.DataprocClusterCreateOperator(cluster_name, project_id,num_workers,zone,network_uri=None,subnetwork_uri=None,internal_ip_only=None, tags=None,storage_bucket=None,init_actions_uris=None,init_action_timeout='10m',metadata =None, image_version=None, 属性=None, master_machine_type='n1-standard-4', master_disk_size =500, worker_machine_type='n1-standard-4', worker_disk_size=500, num_preemptible_workers = 0, labels=None, region=' global', gcp_conn_id='google_cloud_default', delegate_to=None, service_account=None, service_account_scopes=None, idle_delete_ttl=None, auto_delete_time = None, auto_delete_ttl = None, *args, **kwargs)

基类： airflow.models.BaseOperator

在 Google Cloud Dataproc 上创建新集群。operator 将等待创建成功或创建过程中发生错误。

参数允许配置集群。请参阅

https://cloud.google.com/dataproc/docs/reference/rest/v1/projects.regions.clusters

有关不同参数的详细说明。链接中详述的大多数配置参数都可作为此 operator 的参数。

参数：

- cluster_name(str) - 要创建的 DataProc 集群的名称。（模板渲染后）
- project_id(str) - 用于创建集群的 Google 云项目的 ID。（模板渲染后）
- num_workers(int) - 工人数量
- storage_bucket(str) - 要使用的存储桶，设置为 None 允许 dataproc 为您生成自定义存储桶
- init_actions_uris(list[str]) - 包含数据空间初始化脚本的 GCS uri 列表
- init_action_timeout(str) - init_actions_uris 中可执行脚本限定的完成时间
- metadata(dict) - 要添加到所有实例的键值 google 计算引擎元数据条目的字典
- image_version(str) - Dataproc 集群内的软件版本
- properties(dict) - 配置的属性字典（如 spark-defaults.conf），见 https://cloud.google.com/dataproc/docs/reference/rest/v1/projects.regions.clusters#SoftwareConfig
- master_machine_type(str) - 计算要用于主节点的引擎机器类型
- master_disk_size(int) - 主节点的磁盘大小
- worker_machine_type(str) - 计算要用于工作节点的引擎计算机类型
- worker_disk_size(int) - 工作节点的磁盘大小
- num_preemptible_workers(int) - 可抢占的工作节点数
- labels(dict) - 要添加到集群的标签的字典

- zone(str) – 集群所在的区域。(模板渲染后)
- network_uri(str) – 用于机器通信的网络 uri，不能用 subnetwork_uri 指定
- subnetwork_uri(str) – 无法使用 network_uri 指定要用于机器通信的子网 uri
- internal_ip_only(bool) – 如果为 true，则集群中的所有实例将只具有内部 IP 地址。这只能为启用子网的网络启用
- tags(list[str]) – 要添加到所有实例的 GCE 标记
- region – 默认为'global'，可能在未来变得相关。(模板渲染后)
- gcp_conn_id(str) – 用于连接到 Google Cloud Platform 的连接 ID。
- delegate_to(str) – 代理的帐户(如果有)。为此，发出请求的服务帐户必须启用域范围委派。
- service_account(str) – dataproc 实例的服务帐户。
- service_account_scopes(list[str]) – 要包含的服务帐户范围的 URI。
- idle_delete_ttl(int) – 集群在保持空闲状态时保持活动状态的最长持续时间。通过此阈值将导致集群被自动删除。持续时间(秒)。
- auto_delete_time(datetime.datetime) – 自动删除集群的时间。
- auto_delete_ttl(int) – 集群的生命周期，集群将在此持续时间结束时自动删除，持续时间(秒)。(如果设置了 auto_delete_time，则将忽略此参数)

DataprocClusterScaleOperator

```
class airflow.contrib.operators.dataproc_operator.DataprocClusterScaleOperator(cluster_name, project_id,region='global',gcp_conn_id='google_cloud_default',delegate_to=None,num_workers=2, num_preemptible_workers=0, graceful_decommission_timeout=None, *args, **kwargs)
```

基类： airflow.models.BaseOperator

在 Google Cloud Dataproc 上进行扩展，向上或向下扩展。operator 将等待，直到重新调整集群。

示例 ：

```
t1 = DataprocClusterScaleOperator(
task_id='dataproc_scale', project_id='my-project', cluster_name='cluster-1', num_workers=10, num_preemptible_workers=10, graceful_decommission_timeout='1h'dag=dag)
```

也可以看看有关扩展集群的更多详细信息，请参阅以下参考：https://cloud.google.com/dataproc/docs/concepts/configuring-clusters/scaling-clusters

参数：

- cluster_name(str) – 要扩展的集群的名称。(模板渲染后)
- project_id(str) – 集群运行的 Google 云项目的 ID。(模板渲染后)
- region(str) – 数据通路簇的区域。(模板渲染后)
- gcp_conn_id(str) – 用于连接到 Google Cloud Platform 的连接 ID。
- num_workers(int) – 工人数量
- num_preemptible_workers(int) – 新的可抢占工人数量
- graceful_decommission_timeout(str) – 优雅的 YARN decomissioning 超时。最大值为 1d

- delegate_to(str) - 代理的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。

DataprocClusterDeleteOperator

```
class airflow.contrib.operators.dataproc_operator.DataprocClusterDeleteOperator(cluster_name, project_id, region='global', gcp_conn_id='google_cloud_default', delegate_to=None, *args, **kwargs)
```

基类： airflow.models.BaseOperator

删除 Google Cloud Dataproc 上的集群。operator 将等待，直到集群被销毁。

参数：

- cluster_name(str) - 要创建的集群的名称。（模板渲染后）
- project_id(str) - 集群运行的 Google 云项目的 ID。（模板渲染后）
- region(str) - 保留为“全局”，将来可能会变得相关。（模板渲染后）
- gcp_conn_id(str) - 用于连接到 Google Cloud Platform 的连接 ID。
- delegate_to(str) - 代理的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。

DataProcPigOperator

```
class airflow.contrib.operators.dataproc_operator.DataProcPigOperator(query=None, query_uri=None,variables=None,job_name='{{task.task_id}}_{{ds_nodash}}',cluster_name='cluster-1', dataproc_pig_properties=None, dataproc_pig_jars=None, gcp_conn_id='google_cloud_default', delegate_to=None, region='global', *args, **kwargs)
```

基类： airflow.models.BaseOperator

在 Cloud DataProc 集群上启动 Pig 查询作业。操作的参数将传递给集群。

在 dag 的 default_args 中定义 dataproc_*参数是一种很好的做法，比如集群名称和 UDF。

```
default_args = {
    'cluster_name': 'cluster-1',
    'dataproc_pig_jars': [
        'gs://example/udf/jar/datafu/1.2.0/datafu.jar',
        'gs://example/udf/jar/gpig/1.2/gpig.jar'
    ]
}
```

您可以将 pig 脚本作为字符串或文件引用传递。使用变量传递要在集群上解析的 pig 脚本的变量，或者使用要在脚本中解析的参数作为模板参数。

示例 ：

```
t1 = DataProcPigOperator (
        task_id = 'dataproc_pig',
        query = 'a_pig_script.pig',
        variables = { 'out': 'gs://example/output/{{ds}}' },
        dag = dag )
```

也可以看看有关工作提交的更多详细信息，请参阅以下参考：https://cloud.google.com/dataproc/reference/rest/v1/projects.regions.jobs

参数：

- query(str) - 对查询文件的查询或引用(pg 或 pig 扩展)。(模板渲染后)
- query_uri(str) - 云存储上的猪脚本的 uri。
- variables(dict) - 查询的命名参数的映射。(模板渲染后)
- job_name(str) - DataProc 集群中使用的作业名称。默认情况下，此名称是附加执行数据的 task_id，但可以进行模板化。该名称将始终附加一个随机数，以避免名称冲突。(模板渲染后)
- cluster_name(str) - DataProc 集群的名称。(模板渲染后)
- dataproc_pig_properties(dict) - Pig 属性的映射。非常适合放入默认参数
- dataproc_pig_jars(list) - 在云存储中配置的 jars 的 URI (例如：用于 UDF 和 lib)，非常适合放入默认参数。
- gcp_conn_id(str) - 用于连接到 Google Cloud Platform 的连接 ID。
- delegate_to(str) - 代理的帐户 (如果有)。为此，发出请求的服务帐户必须启用域范围委派。
- region(str) - 创建数据加载集群的指定区域。

DataProcHiveOperator

```
class airflow.contrib.operators.dataproc_operator.DataProcHiveOperator(query=None, query_uri=None, variables=None, job_name='{{task.task_id}}_{{ds_nodash}}', cluster_name='cluster-1', dataproc_hive_properties=None, dataproc_hive_jars=None, gcp_conn_id='google_cloud_default', delegate_to=None, region='global', *args, **kwargs)
```

基类： airflow.models.BaseOperator

在 Cloud DataProc 集群上启动 Hive 查询作业。

参数：

- query(str) - 查询或对查询文件的引用 (q 扩展名)。
- query_uri(str) - 云存储上的 hive 脚本的 uri。
- variables(dict) - 查询的命名参数的映射。
- job_name(str) - DataProc 集群中使用的作业名称。默认情况下，此名称是附加执行数据的 task_id，但可以进行模板化。该名称将始终附加一个随机数，以避免名称冲突。

- cluster_name(str) - DataProc 集群的名称。
- dataproc_hive_properties(dict) - Pig 属性的映射。非常适合放入默认参数
- dataproc_hive_jars(list) - 在云存储中配置的 jars 的 URI（例如：用于 UDF 和 lib），非常适合放入默认参数。
- gcp_conn_id(str) - 用于连接到 Google Cloud Platform 的连接 ID。
- delegate_to(str) - 代理的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。
- region(str) - 创建数据加载集群的指定区域。

DataProcSparkSqlOperator

```
class airflow.contrib.operators.dataproc_operator.DataProcSparkSqlOperator(query = None , query_uri = None, variables = None, job_name ='{{task.task_id}}_{{ds_nodash}}', cluster_name ='cluster-1', dataproc_spark_properties = None , dataproc_spark_jars = None , gcp_conn_id ='google_cloud_default', delegate_to = None, region ='global', *args, **kwargs)
```

基类： airflow.models.BaseOperator

在 Cloud DataProc 集群上启动 Spark SQL 查询作业。

参数：

- query(str) - 查询或对查询文件的引用（q 扩展名）。（模板渲染后）
- query_uri(str) - 云存储上的一个 spark sql 脚本的 uri。
- variables(dict) - 查询的命名参数的映射。（模板渲染后）
- job_name(str) - DataProc 集群中使用的作业名称。默认情况下，此名称是附加执行数据的 task_id，但可以进行模板化。该名称将始终附加一个随机数，以避免名称冲突。（模板渲染后）
- cluster_name(str) - DataProc 集群的名称。（模板渲染后）
- dataproc_spark_properties(dict) - Pig 属性的映射。非常适合放入默认参数
- dataproc_spark_jars(list) - 在云存储中配置的 jars 的 URI（例如：用于 UDF 和 lib），非常适合放入默认参数。
- gcp_conn_id(str) - 用于连接到 Google Cloud Platform 的连接 ID。
- delegate_to(str) - 代理的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。
- region(str) - 创建数据加载集群的指定区域。

DataProcSparkOperator

```
class airflow.contrib.operators.dataproc_operator.DataProcSparkOperator(main_jar = None , main_class = None , arguments = None , archives = None , files = None , job_name ='{{task.task_id}}_{{ds_nodash}}', cluster_name ='cluster-1', dataproc_spark_properties =None, dataproc_spark_jars = None, gcp_conn_id ='google_cloud_default', delegate_to = None, region ='global', *args, **kwargs)
```

基类： airflow.models.BaseOperator

在 Cloud DataProc 集群上启动 Spark 作业。

参数：

- main_jar(str) - 在云存储上配置的作业 jar 的 URI。(使用 this 或 main_class，而不是两者一起)。
- main_class(str) - 作业类的名称。(使用 this 或 main_jar，而不是两者一起)。
- arguments(list) - 作业的参数。(模板渲染后)
- archives(list) - 将在工作目录中解压缩的已归档文件列表。应存储在云存储中。
- files(list) - 要复制到工作目录的文件列表
- job_name(str) - DataProc 集群中使用的作业名称。默认情况下，此名称是附加执行数据的 task_id，但可以进行模板化。该名称将始终附加一个随机数，以避免名称冲突。(模板渲染后)
- cluster_name(str) - DataProc 集群的名称。(模板渲染后)
- dataproc_spark_properties(dict) - Pig 属性的映射。非常适合放入默认参数
- dataproc_spark_jars(list) - 在云存储中配置的 jars 的 URI (例如：用于 UDF 和 lib)，非常适合放入默认参数。
- gcp_conn_id(str) - 用于连接到 Google Cloud Platform 的连接 ID。
- delegate_to(str) - 代理的帐户 (如果有)。为此，发出请求的服务帐户必须启用域范围委派。
- region(str) - 创建数据加载集群的指定区域。

DataProcHadoopOperator

```
class airflow.contrib.operators.dataproc_operator.DataProcHadoopOperator(main_jar = None , main_class = None , arguments = None , archives = None , files = None , job_name ='{{task.task_id}}_{{ds_nodash}}',cluster_name ='cluster-1',dataproc_hadoop_properties = None, dataproc_hadoop_jars = None, gcp_conn_id ='google_cloud_default', delegate_to = None, region ='global', *args, **kwargs)
```

基类： airflow.models.BaseOperator

在 Cloud DataProc 集群上启动 Hadoop 作业。

参数：

- main_jar(str) - 在云存储上配置的作业 jar 的 URI。(使用 this 或 main_class，而不是两者一起)。
- main_class(str) - 作业类的名称。(使用 this 或 main_jar，而不是两者一起)。
- arguments(list) - 作业的参数。(模板渲染后)
- archives(list) - 将在工作目录中解压缩的已归档文件列表。应存储在云存储中。
- files(list) - 要复制到工作目录的文件列表
- job_name(str) - DataProc 集群中使用的作业名称。默认情况下，此名称是附加执行数据的 task_id，但可以进行模板化。该名称将始终附加一个随机数，以避免名称冲突。(模板渲染后)
- cluster_name(str) - DataProc 集群的名称。(模板渲染后)
- dataproc_hadoop_properties(dict) - Pig 属性的映射。非常适合放入默认参数

- dataproc_hadoop_jars(list) - 在云存储中配置的 jars 的 URI（例如：用于 UDF 和 lib），非常适合放入默认参数。
- gcp_conn_id(str) - 用于连接到 Google Cloud Platform 的连接 ID。
- delegate_to(str) - 代理的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。
- region(str) - 创建数据加载集群的指定区域。

DataProcPySparkOperator

```
class airflow.contrib.operators.dataproc_operator.DataProcPySparkOperator(main, arguments = None, archives = None, pyfiles = None, files = None, job_name ='{{task.task_id}}_{{ds_nodash}}', cluster_name =' cluster-1', dataproc_pyspark_properties = None, dataproc_pyspark_jars = None, gcp_conn_id ='google_cloud_default', delegate_to = None, region ='global', *args, **kwargs)
```

基类： airflow.models.BaseOperator

在 Cloud DataProc 集群上启动 PySpark 作业。

参数：

- main(str) -[必需]用作驱动程序的主 Python 文件的 Hadoop 兼容文件系统（HCFS）URI。必须是.py 文件。
- arguments(list) - 作业的参数。（模板渲染后）
- archives(list) - 将在工作目录中解压缩的已归档文件列表。应存储在云存储中。
- files(list) - 要复制到工作目录的文件列表
- pyfiles(list) - 要传递给 PySpark 框架的 Python 文件列表。支持的文件类型：.py，.egg 和.zip
- job_name(str) -DataProc 集群中使用的作业名称。默认情况下，此名称是附加执行数据的 task_id，但可以进行模板化。该名称将始终附加一个随机数，以避免名称冲突。（模板渲染后）
- cluster_name(str) - DataProc 集群的名称。
- dataproc_pyspark_properties(dict) - Pig 属性的映射。非常适合放入默认参数
- dataproc_pyspark_jars(list) - 在云存储中配置的 jars 的 URI（例如：用于 UDF 和 lib），非常适合放入默认参数。
- gcp_conn_id(str) - 用于连接到 Google Cloud Platform 的连接 ID。
- delegate_to(str) - 代理的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。
- region(str) - 创建数据加载集群的指定区域。

DataprocWorkflowTemplateInstantiateOperator

```
class airflow.contrib.operators.dataproc_operator.DataprocWorkflowTemplateInstantiateOperator(template_id, *args, **kwargs)
```

基类： [airflow.contrib.operators.dataproc_operator.DataprocWorkflowTemplateBaseOperator]

在 Google Cloud Dataproc 上实例化 WorkflowTemplate。operator 将等待 WorkflowTemplate 完成执行。

参数：

template_id(str) - 模板的 id。（模板渲染后）
project_id(str) - 模板运行所在的 Google 云项目的 ID
region(str) - 保留为“全局”，将来可能会变得相关
gcp_conn_id(str) - 用于连接到 Google Cloud Platform 的连接 ID。
delegate_to(str) - 代理的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。

DataprocWorkflowTemplateInstantiateInlineOperator

class airflow.contrib.operators.dataproc_operator.DataprocWorkflowTemplateInstantiateInlineOperator(template, *args, **kwargs)

基类： [airflow.contrib.operators.dataproc_operator.DataprocWorkflowTemplateBaseOperator]

在 Google Cloud Dataproc 上实例化 WorkflowTemplate 内联。operator 将等待 WorkflowTemplate 完成执行。

参数：

- template(map) - 模板内容。（模板渲染后）
- project_id(str) - 模板运行所在的 Google 云项目的 ID
- region(str) - 保留为“全局”，将来可能会变得相关
- gcp_conn_id(str) - 用于连接到 Google Cloud Platform 的连接 ID。
- delegate_to(str) - 代理的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。

云数据存储区

数据存储区运营商

- DatastoreExportOperator：将实体从 Google Cloud Datastore 导出到云存储。
- DatastoreImportOperator：将实体从云存储导入 Google Cloud Datastore。

DatastoreExportOperator

class airflow.contrib.operators.datastore_export_operator.DatastoreExportOperator(bucket, namespace = None, datastore_conn_id ='google_cloud_default', cloud_storage_conn_id ='google_cloud_default', delegate_to = None, entity_filter = None, labels = None, polling_interval_in_seconds=10, overwrite_existing=False, xcom_push=False, *args, **kwargs)

基类： airflow.models.BaseOperator

将实体从 Google Cloud Datastore 导出到云存储

参数：

- bucket(str) - 要备份数据的云存储桶的名称
- namespace(str) - 指定云存储桶中用于备份数据的可选命名空间路径。如果 GCS 中不存在此命名空间，则将创建该命名空间。
- datastore_conn_id(str) - 要使用的数据存储区连接 ID 的名称
- cloud_storage_conn_id(str) - 强制写入备份的云存储连接 ID 的名称
- delegate_to(str) - 代理的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。
- entity_filter(dict) - 导出中包含项目中哪些数据的说明，请参阅 https://cloud.google.com/datastore/docs/reference/rest/Shared.Types/EntityFilter
- labels(dict) - 客户端分配的云存储标签
- polling_interval_in_seconds(int) - 再次轮询执行状态之前等待的秒数
- overwrite_existing(bool) - 如果存储桶+命名空间不为空，则在导出之前将清空它。这样可以覆盖现有备份。
- xcom_push(bool) - 将操作名称推送到 xcom 以供参考

DatastoreImportOperator

```
class airflow.contrib.operators.datastore_import_operator.DatastoreImportOperator(bucket , file , namespace = None , entity_filter = None , labels = None , datastore_conn_id ='google_cloud_default',delegate_to=None,polling_interval_in_seconds=10,xcom_push=False, *args, **kwargs)
```

基类： airflow.models.BaseOperator

将实体从云存储导入 Google Cloud Datastore

参数：

- bucket(str) - 云存储中用于存储数据的容器
- file(str) - 指定云存储桶中备份元数据文件的路径。它应该具有扩展名.overall_export_metadata
- namespace(str) - 指定云存储桶中备份元数据文件的可选命名空间。
- entity_filter(dict) - 导出中包含项目中哪些数据的说明，请参阅 https://cloud.google.com/datastore/docs/reference/rest/Shared.Types/EntityFilter
- labels(dict) - 客户端分配的云存储标签
- datastore_conn_id(str) - 要使用的连接 ID 的名称
- delegate_to(str) - 代理的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。
- polling_interval_in_seconds(int) - 再次轮询执行状态之前等待的秒数
- xcom_push(bool) - 将操作名称推送到 xcom 以供参考

DatastoreHook

class airflow.contrib.hooks.datastore_hook.DatastoreHook(datastore_conn_id='google_cloud_datastore_default', delegate_to = None)

基类： [airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook]

与 Google Cloud Datastore 互动。此挂钩使用 Google Cloud Platform 连接。

此对象不是线程安全的。如果要同时发出多个请求，则需要为每个线程创建一个钩子。

allocate_ids(partialKeys)

参数：partialKeys - 部分键列表

返回：完整密钥列表。

begin_transaction()

获取新的事务处理

也可以看看

https://cloud.google.com/datastore/docs/reference/rest/v1/projects/beginTransaction

返回：交易句柄

commit(body)

提交事务，可选地创建，删除或修改某些实体。

也可以看看

https://cloud.google.com/datastore/docs/reference/rest/v1/projects/commit

参数：body - 提交请求的主体

返回：提交请求的响应主体

delete_operation（名称）

删除长时间运行的操作

参数：name - 操作资源的名称

export_to_storage_bucket(bucket, namespace = None, entity_filter = None, labels = None)

将实体从 Cloud Datastore 导出到 Cloud Storage 进行备份

```
get_conn（version='V1'）
```

返回 Google 云端存储服务对象。

```
GET_OPERATION（name）
```

获取长时间运行的最新状态

参数：name - 操作资源的名称

```
import_from_storage_bucket(bucket, file, namespace=None, entity_filter=None, labels=None)
```

将备份从云存储导入云数据存储

```
lookup(keys, read_consistency = None, transaction = None)
```

按键查找一些实体

也可以看看

https://cloud.google.com/datastore/docs/reference/rest/v1/projects/lookup

参数：

- keys - 要查找的键
- read_consistency - 要使用的读取一致性。默认，强或最终。不能与事务一起使用。
- transaction - 要使用的事务，如果有的话。

返回：查找请求的响应主体。

```
poll_operation_until_done(name, polling_interval_in_seconds)
```

轮询备份操作状态直到完成

```
rollback(transaction)
```

回滚事务

也可以看看 https://cloud.google.com/datastore/docs/reference/rest/v1/projects/rollback

参数：transaction - 要回滚的事务

run_query（体）

运行实体查询。

也可以看看 https://cloud.google.com/datastore/docs/reference/rest/v1/projects/runQuery

参数：body - 查询请求的主体

返回：批量查询结果。

云 ML 引擎

云 ML 引擎运营商

- MLEngineBatchPredictionOperator：启动 Cloud ML Engine 批量预测作业。
- MLEngineModelOperator：管理 Cloud ML Engine 模型。
- MLEngineTrainingOperator：启动 Cloud ML Engine 培训工作。
- MLEngineVersionOperator：管理 Cloud ML Engine 模型版本。

MLEngineBatchPredictionOperator

```
class airflow.contrib.operators.mlengine_operator.MLEngineBatchPredictionOperator(project_id, job_id, region, data_format, input_paths, output_path, model_name = None, version_name = None, uri = None, max_worker_count = None, runtime_version = None, gcp_conn_id = 'google_cloud_default', delegate_to = None, *args, **kwargs)
```

基类： airflow.models.BaseOperator

启动 Google Cloud ML Engine 预测作业。

注意：对于模型原点，用户应该考虑以下三个选项中的一个：1。仅填充“uri”字段，该字段应该是指向 tensorflow savedModel 目录的 GCS 位置。2.仅填充'model_name'字段，该字段引用现有模型，并将使用模型的默认版本。3.填充“model_name”和“version_name”字段，这些字段指特定模型的特定版本。

在选项 2 和 3 中，模型和版本名称都应包含最小标识符。例如，打电话

```
MLEngineBatchPredictionOperator (
    ... ,
    model_name = 'my_model' ,
    version_name = 'my_version' ,
    ... )
```

如果所需的型号版本是“projects / my_project / models / my_model / versions / my_version”。

有关参数的更多文档,请参阅https://cloud.google.com/ml-engine/reference/rest/v1/projects.jobs。

参数：

- project_id(str) - 提交预测作业的 Google Cloud 项目名称。（模板渲染后）
- job_id(str) - Google Cloud ML Engine 上预测作业的唯一 ID。（模板渲染后）
- data_format(str) - 输入数据的格式。如果未提供或者不是[“TEXT”,“TF_RECORD”,“TF_RECORD_GZIP”]之一，它将默认为“DATA_FORMAT_UNSPECIFIED”。
- input_paths(list[str]) - 批量预测的输入数据的 GCS 路径列表。接受通配符 operator *，但仅限于结尾处。（模板渲染后）
- output_path(str) - 写入预测结果的 GCS 路径。（模板渲染后）
- region(str) - 用于运行预测作业的 Google Compute Engine 区域。（模板化）
- model_name(str) - 用于预测的 Google Cloud ML Engine 模型。如果未提供 version_name，则将使用此模型的默认版本。如果提供了 version_name，则不应为 None。如果提供 uri，则应为 None。（模板渲染后）
- version_name(str) - 用于预测的 Google Cloud ML Engine 模型版本。如果提供 uri，则应为 None。（模板渲染后）
- uri(str) - 用于预测的已保存模型的 GCS 路径。如果提供了 model_name，则应为 None。它应该是指向张量流 SavedModel 的 GCS 路径。（模板渲染后）
- max_worker_count(int) - 用于并行处理的最大 worker 数。如果未指定，则默认为 10。
- runtime_version(str) - 用于批量预测的 Google Cloud ML Engine 运行时版本。
- gcp_conn_id(str) - 用于连接到 Google Cloud Platform 的连接 ID。
- delegate_to(str) - 代理的帐户（如果有）。为此，发出请求的服务帐户必须启用 doamin 范围的委派。

Raises:
`ValueError` ：如果无法确定唯一的模型/版本来源。

MLEngineModelOperator

class airflow.contrib.operators.mlengine_operator.MLEngineModelOperator(project_id, model, operation='create', gcp_conn_id='google_cloud_default', delegate_to=None, *args, **kwargs)

基类： airflow.models.BaseOperator

管理 Google Cloud ML Engine 模型的运营商。

参数：

- project_id(str) - MLEngine 模型所属的 Google Cloud 项目名称。（模板渲染后）
- model(dict) -

包含有关模型信息的字典。如果操作是 create，则 model 参数应包含有关此模型的所有信息，例如 name。如果`操作`是`get`，则`model`参数应包含`模型`的`名称`。

- 操作 -

执行的操作。可用的操作是：

* `create`：创建`model`参数提供的新模型。
* `get`：获取在模型中指定名称的特定`模型`。

- gcp_conn_id(str) - 获取连接信息时使用的连接 ID。
- delegate_to(str) - 代理的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。

MLEngineTrainingOperator

```
class airflow.contrib.operators.mlengine_operator.MLEngineTrainingOperator(project_id, job_id, package_uris, training_python_module, training_args, region, scale_tier=None, runtime_version=None, python_version=None, job_dir=None, gcp_conn_id='google_cloud_default', delegate_to=None, mode='PRODUCTION', *args, **kwargs)
```

基类： airflow.models.BaseOperator

启动 MLEngine 培训工作的 operator 。

参数：

- project_id(str) - 应在其中运行 MLEngine 培训作业的 Google Cloud 项目名称（模板化）。
- job_id(str) - 提交的 Google MLEngine 培训作业的唯一模板化 ID。（模板渲染后）
- package_uris(str) - MLEngine 培训作业的包位置列表，其中应包括主要培训计划+任何其他依赖项。（模板渲染后）
- training_python_module(str) - 安装'package_uris'软件包后，在 MLEngine 培训作业中运行的 Python 模块名称。（模板渲染后）
- training_args(str) - 传递给 MLEngine 训练程序的模板化命令行参数列表。（模板渲染后）
- region(str) - 用于运行 MLEngine 培训作业的 Google Compute Engine 区域（模板化）。
- scale_tier(str) - MLEngine 培训作业的资源层。（模板渲染后）
- runtime_version(str) - 用于培训的 Google Cloud ML 运行时版本。（模板渲染后）
- python_version(str) - 训练中使用的 Python 版本。（模板渲染后）
- job_dir(str) - 用于存储培训输出和培训所需的其他数据的 Google 云端存储路径。（模板渲染后）
- gcp_conn_id(str) - 获取连接信息时使用的连接 ID。
- delegate_to(str) - 代理的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。
- mode(str) - 可以是'DRY_RUN'/'CLOUD'之一。在“DRY_RUN”模式下，不会启动真正的培训作业，但会打印出 MLEngine 培训作业请求。在“CLOUD”模式下，将发出真正的 MLEngine 培训作业创建请求。

MLEngineVersionOperator

```
class airflow.contrib.operators.mlengine_operator.MLEngineVersionOperator(project_id, model_name, version_name=None, version=None, operation='create', gcp_conn_id='google_cloud_default', delegate_to=None, *args, **kwargs)
```

基类： airflow.models.BaseOperator

管理 Google Cloud ML Engine 版本的运营商。

参数：

- project_id(str) - MLEngine 模型所属的 Google Cloud 项目名称。
- model_name(str) - 版本所属的 Google Cloud ML Engine 模型的名称。（模板渲染后）
- version_name(str) - 用于正在操作的版本的名称。如果没有人及版本的说法是没有或不具备的值名称键，那么这将是有效载荷中用于填充名称键。（模板渲染后）

- version(dict) - 包含版本信息的字典。如果操作是 create，则 version 应包含有关此版本的所有信息，例如 name 和 deploymentUrl。如果操作是 get 或 delete，则 version 参数应包含版本的名称。如果是 None，则唯一可能的操作是 list。（模板渲染后）
- operation(str) -
  执行的操作。可用的操作是：
  create：在 model_name 指定的模型中创建新版本，在这种情况下，version 参数应包含创建该版本的所有信息（例如 name，deploymentUrl）。
  get：获取 model_name 指定的模型中特定版本的完整信息。应在 version 参数中指定版本的名称。
  list：列出 model_name 指定的模型的所有可用版本。
  delete：从 model_name 指定的模型中删除 version 参数中指定的版本。应在 version 参数中指定版本的名称。
- gcp_conn_id(str) - 获取连接信息时使用的连接 ID。
- delegate_to(str) - 代理的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。

Cloud ML Engine Hook

MLEngineHook

```
class airflow.contrib.hooks.gcp_mlengine_hook.MLEngineHook(gcp_conn_id='google_cloud_default', delegate_to = None)
```

基类： [airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook]

```
create_job(project_id, job, use_existing_job_fn = None)
```

启动 MLEngine 作业并等待它达到终端状态。

参数：

- project_id(str) - 将在其中启动 MLEngine 作业的 Google Cloud 项目 ID。
- job(dict) -

  应该提供给 MLEngine API 的 MLEngine Job 对象，例如：

```
{
  'jobId': 'my_job_id',
  'trainingInput': {
    'scaleTier': 'STANDARD_1',
    ...
  }
}
```

- use_existing_job_fn(function) - 如果已存在具有相同 job_id 的 MLEngine 作业，则此方法（如果提供）将决定是否应使用此现有作业，继续等待它完成并返回作业对象。它应该接受 MLEngine 作业对象，并返回一个布尔值，指示是否可以重用现有作业。如果未提供“use_existing_job_fn”，我们默认重用现有的 MLEngine 作业。

返回：如果作业成功到达终端状态（可能是 FAILED 或 CANCELED 状态），则为 MLEngine 作业对象。

返回类型：字典

```
create_model(project_id, model)
```

创建一个模型。阻止直到完成。

```
create_version(project_id, model_name, version_spec)
```

在 Google Cloud ML Engine 上创建版本。

如果版本创建成功则返回操作，否则引发错误。

```
delete_version(project_id, model_name, version_name)
```

删除给定版本的模型。阻止直到完成。

```
get_conn()
```

返回 Google MLEngine 服务对象。

```
get_model(project_id, model_name)
```

获取一个模型。阻止直到完成。

```
list_versions(project_id, model_name)
```

列出模型的所有可用版本。阻止直到完成。

```
set_default_version(project_id, model_name, version_name)
```

将版本设置为默认值。阻止直到完成。

云储存

存储运营商

- FileToGoogleCloudStorageOperator：将文件上传到 Google 云端存储。
- GoogleCloudStorageCreateBucketOperator：创建新的云存储桶。
- GoogleCloudStorageListOperator：列出存储桶中的所有对象，并在名称中添加字符串前缀和分隔符。
- GoogleCloudStorageDownloadOperator：从 Google 云端存储中下载文件。
- GoogleCloudStorageToBigQueryOperator：将 Google 云存储中的文件加载到 BigQuery 中。
- GoogleCloudStorageToGoogleCloudStorageOperator：将对象从存储桶复制到另一个存储桶，并在需要时重命名。

FileToGoogleCloudStorageOperator

```
class airflow.contrib.operators.file_to_gcs.FileToGoogleCloudStorageOperator(src，dst，bucket，google_cloud_storage_conn_id ='google_cloud_default'，mime_type ='application / octet-stream'，delegate_to = None，*args，**kwargs)
```

基类： airflow.models.BaseOperator

将文件上传到 Google 云端存储

参数：

- src(str) - 本地文件的路径。（模板渲染后）
- dst(str) - 指定存储桶中的目标路径。（模板渲染后）
- bucket(str) - 要上传的存储桶。（模板渲染后）
- google_cloud_storage_conn_id(str) - 要上传的 Airflow 连接 ID
- mime_type(str) - mime 类型字符串
- delegate_to(str) - 代理的帐户（如果有）

```
execute(context)
```

将文件上传到 Google 云端存储

GoogleCloudStorageCreateBucketOperator

```
class airflow.contrib.operators.gcs_operator.GoogleCloudStorageCreateBucketOperator(bucket_name ，storage_class ='MULTI_REGIONAL'， location ='US'， project_id = None， labels = None，google_cloud_storage_conn_id ='google_cloud_default'，delegate_to = None，*args，**kwargs)
```

基类： airflow.models.BaseOperator

创建一个新存储桶。Google 云端存储使用平面命名空间，因此您无法创建名称已在使用中的存储桶。

参数：

- bucket_name(str) - 存储桶的名称。（模板渲染后）
- storage_class(str) -

  这定义了存储桶中对象的存储方式，并确定了 SLA 和存储成本（模板化）。价值包括
  - MULTI_REGIONAL
  - REGIONAL
  - STANDARD
  - NEARLINE
  - COLDLINE

  如果在创建存储桶时未指定此值，则默认为 STANDARD。

- location(str) -
  水桶的位置。（模板化）存储桶中对象的对象数据驻留在此区域内的物理存储中。默认为美国。
- project_id(str) - GCP 项目的 ID。（模板渲染后）
- labels(dict) - 用户提供的键/值对标签。
- google_cloud_storage_conn_id(str) - 连接到 Google 云端存储时使用的连接 ID。
- delegate_to(str) - 代理的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。

示例:

以下 operator 将在区域中创建 test-bucket 具有 MULTI_REGIONAL 存储类的新存储桶 EU

```
CreateBucket = GoogleCloudStorageCreateBucketOperator (
    task_id = 'CreateNewBucket',
    bucket_name = 'test-bucket',
    storage_class = 'MULTI_REGIONAL',
    location = 'EU',
    labels = { 'env' : 'dev' , 'team' : 'airflow' },
    google_cloud_storage_conn_id = 'airflow-service-account'
)
```

GoogleCloudStorageDownloadOperator

```
class airflow.contrib.operators.gcs_download_operator.GoogleCloudStorageDownloadOperator(bucket , object , filename = None , store_to_xcom_key = None , google_cloud_storage_conn_id ='google_cloud_default', delegate_to = None, *args, **kwargs)
```

基类： airflow.models.BaseOperator

从 Google 云端存储下载文件。

参数：

- bucket(str) - 对象所在的 Google 云存储桶。（模板渲染后）
- object(str) - 要在 Google 云存储桶中下载的对象的名称。（模板渲染后）
- filename(str) - 应将文件下载到的本地文件系统（正在执行操作符的位置）上的文件路径。（模板化）如果未传递文件名，则下载的数据将不会存储在本地文件系统中。
- store_to_xcom_key(str) - 如果设置了此参数，operator 将使用此参数中设置的键将下载文件的内容推送到 XCom。如果未设置，则下载的数据不会被推送到 XCom。（模板渲染后）
- google_cloud_storage_conn_id(str) - 连接到 Google 云端存储时使用的连接 ID。
- delegate_to(str) - 代理的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。

GoogleCloudStorageListOperator

```
class airflow.contrib.operators.gcslistoperator.GoogleCloudStorageListOperator(bucket, prefix = None, delimiter = None, google_cloud_storage_conn_id ='google_cloud_default', delegate_to = None, *args, **kwargs)
```

基类： airflow.models.BaseOperator

使用名称中的给定字符串前缀和分隔符列出存储桶中的所有对象。

此 operator 返回一个 python 列表，其中包含可供其使用的对象的名称 xcom 在下游任务中。

参数：

- bucket(str) - 用于查找对象的 Google 云存储桶。（模板渲染后）
- prefix(str) - 前缀字符串，用于过滤名称以此前缀开头的对象。（模板渲染后）
- delimiter(str) - 要过滤对象的分隔符。（模板化）例如，要列出 GCS 目录中的 CSV 文件，您可以使用 delimiter ='。csv'。
- google_cloud_storage_conn_id(str) - 连接到 Google 云端存储时使用的连接 ID。
- delegate_to(str) - 代理的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。

示例：

以下 operator 将列出存储桶中文件 sales/sales-2017 夹中的所有 Avro 文件 data。

```
GCS_Files = GoogleCloudStorageListOperator (
    task_id = 'GCS_Files',
    bucket = 'data',
    prefix = 'sales/sales-2017/',
    delimiter = '.avro',
```

```
    google_cloud_storage_conn_id = google_cloud_conn_id
)
```

GoogleCloudStorageToBigQueryOperator

```
class    airflow.contrib.operators.gcs_to_bq.GoogleCloudStorageToBigQueryOperator(bucket    ,
source_objects, destination_project_dataset_table, schema_fields=None, schema_object=None,
source_format  ='CSV',  compression  ='NONE',  create_disposition  ='CREATE_IF_NEEDED',
skip_leading_rows=0, write_disposition='WRITE_EMPTY', field_delimiter=',', max_bad_records
= 0, quote_character = None, ignore_unknown_values = False, allow_quoted_newlines = False,
allow_jagged_rows  =  False ,  max_id_key  =  None ,  bigquery_conn_id  ='bigquery_default',
google_cloud_storage_conn_id='google_cloud_default',delegate_to=None,schema_update_options
=(), src_fmt_configs = {}, external_table = False, time_partitioning = {}, *args, **kwargs)
```

基类： airflow.models.BaseOperator

将文件从 Google 云存储加载到 BigQuery 中。

可以用两种方法之一指定用于 BigQuery 表的模式。您可以直接传递架构字段，也可以将运营商指向 Google 云存储对象名称。Google 云存储中的对象必须是包含架构字段的 JSON 文件。

参数：

- bucket(str) - 要加载的桶。（模板渲染后）
- source_objects - 要加载的 Google 云存储 URI 列表（模板化）。如果 source_format 是 'DATASTORE_BACKUP'，则列表必须只包含一个 URI。
- destination_project_dataset_table(str) - 用于加载数据的表(
- schema_fields(list) - 如果设置，则此处定义的架构字段列表：https://cloud.google.com/bigquery/docs/reference/v2/jobs#configuration.load 当 source_format 为'DATASTORE_BACKUP'时，不应设置。
- schema_object - 如果设置，则指向包含表的架构的.json 文件的 GCS 对象路径。（模板渲染后）
- schema_object - 字符串
- source_format(str) - 要导出的文件格式。
- compression(str)-[可选]数据源的压缩类型。可能的值包括 GZIP 和 NONE。默认值为 NONE。Google Cloud Bigtable，Google Cloud Datastore 备份和 Avro 格式会忽略此设置。
- create_disposition(str) - 如果表不存在，则创建处置。
- skip_leading_rows(int) - 从 CSV 加载时要跳过的行数。
- write_disposition(str) - 表已存在时的写处置。
- field_delimiter(str) - 从 CSV 加载时使用的分隔符。
- max_bad_records(int) - BigQuery 在运行作业时可以忽略的最大错误记录数。
- quote_character(str) - 用于引用 CSV 文件中数据部分的值。
- ignore_unknown_values(bool) -[可选]指示 BigQuery 是否应允许表模式中未表示的额外值。如果为 true，则忽略额外值。如果为 false，则将具有额外列的记录视为错误记录，如果错误记录太多，则在作业结果中返回无效错误。

- allow_quoted_newlines(bool) - 是否允许引用的换行符（true）或不允许（false）。
- allow_jagged_rows(bool) - 接受缺少尾随可选列的行。缺失值被视为空值。如果为 false，则缺少尾随列的记录将被视为错误记录，如果错误记录太多，则会在作业结果中返回无效错误。仅适用于 CSV，忽略其他格式。
- max_id_key(str) - 如果设置，则是 BigQuery 表中要加载的列的名称。在加载发生后，Thsi 将用于从 BigQuery 中选择 MAX 值。结果将由 execute()命令返回，该命令又存储在 XCom 中供将来的 operator 使用。这对增量加载很有帮助 - 在将来的执行过程中，您可以从最大 ID 中获取。
- bigquery_conn_id(str) - 对特定 BigQuery 的引用。
- google_cloud_storage_conn_id(str) - 对特定 Google 云存储挂钩的引用。
- delegate_to(str) - 代理的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。
- schema_update_options(list) - 允许更新目标表的模式作为加载作业的副作用。
- src_fmt_configs(dict) - 配置特定于源格式的可选字段
- external_table(bool) - 用于指定目标表是否应为 BigQuery 外部表的标志。默认值为 False。
- time_partitioning(dict) - 配置可选的时间分区字段，即按 API 规范按字段，类型和到期分区。请注意，“field”在 dataset.table $ partition 的并发中不可用。

GoogleCloudStorageToGoogleCloudStorageOperator

class airflow.contrib.operators.gcs_to_gcs.GoogleCloudStorageToGoogleCloudStorageOperator(source_bucket, source_object, destination_bucket = None, destination_object = None, move_object = False, google_cloud_storage_conn_id ='google_cloud_default', delegate_to = None, *args, **kwargs)

基类： airflow.models.BaseOperator

将对象从存储桶复制到另一个存储桶，并在需要时重命名。

参数：

- source_bucket(str) - 对象所在的源 Google 云存储桶。（模板渲染后）
- source_object(str) -

  要在 Google 云存储分区中复制的对象的源名称。（模板化）如果在此参数中使用通配符：
  您只能在存储桶中使用一个通配符作为对象（文件名）。通配符可以出现在对象名称内或对象名称的末尾。不支持在存储桶名称中附加通配符。

- destination_bucket - 目标 Google 云端存储分区

对象应该在哪里。（模板化）：type destination_bucket：string：param destination_object：对象的目标名称

目标 Google 云存储桶。（模板化）如果在 source_object 参数中提供了通配符，则这是将添加到最终目标对象路径的前缀。请注意，将删除通配符之前的源路径部分; 如果需要保留，则应将其附加到 destination_object。例如，使用 prefix foo/*和 destination_object'blah /``，文件 foo/baz 将被复

制到 blah/baz; 保留前缀写入 destination_object，例如 blah/foo，在这种情况下，复制的文件将被命名 blah/foo/baz`。

参数：move_object - 当移动对象为 True 时，移动对象

复制到新位置。

这相当于 mv 命令而不是 cp 命令。

参数：

- google_cloud_storage_conn_id(str) - 连接到 Google 云端存储时使用的连接 ID。
- delegate_to(str) - 代理的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。

Examples:
下面的操作将命名一个文件复制 sales/sales-2017/january.avro 在 data 桶的文件和名为斗 copied_sales/2017/january-backup.avro in the ``data_backup`

```
copy_single_file = GoogleCloudStorageToGoogleCloudStorageOperator (
    task_id = 'copy_single_file',
    source_bucket = 'data',
    source_object = 'sales/sales-2017/january.avro',
    destination_bucket = 'data_backup',
    destination_object = 'copied_sales/2017/january-backup.avro',
    google_cloud_storage_conn_id = google_cloud_conn_id
)
```

以下 operator 会将文件 sales/sales-2017 夹中的所有 Avro 文件（即名称以该前缀开头）复制到存储 data 桶中的 copied_sales/2017 文件夹中 data_backup。

```
copy_files = GoogleCloudStorageToGoogleCloudStorageOperator (
    task_id = 'copy_files',
    source_bucket = 'data',
    source_object = 'sales/sales-2017/*.avro',
    destination_bucket = 'data_backup',
    destination_object = 'copied_sales/2017/',
    google_cloud_storage_conn_id = google_cloud_conn_id
)
```

以下 operator 会将文件 sales/sales-2017 夹中的所有 Avro 文件（即名称以该前缀开头）移动到 data 存储桶中的同一文件夹 data_backup，删除过程中的原始文件。

```
move_files = GoogleCloudStorageToGoogleCloudStorageOperator (
    task_id = 'move_files',
```

```
    source_bucket = 'data',
    source_object = 'sales/sales-2017/*.avro',
    destination_bucket = 'data_backup',
    move_object = True,
    google_cloud_storage_conn_id = google_cloud_conn_id
)
```

GoogleCloudStorageHook

```
class     airflow.contrib.hooks.gcs_hook.GoogleCloudStorageHook(google_cloud_storage_conn_id
='google_cloud_default', delegate_to = None)
```

基类： [airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook]

与 Google 云端存储互动。此挂钩使用 Google Cloud Platform 连接。

```
copy(source_bucket, source_object, destination_bucket = None, destination_object = None)
```

将对象从存储桶复制到另一个存储桶，并在需要时重命名。

destination_bucket 或 destination_object 可以省略，在这种情况下使用源桶/对象，但不能同时使用两者。

参数：

- source_bucket(str) - 要从中复制的对象的存储桶。
- source_object(str) - 要复制的对象。
- destination_bucket(str) - 要复制到的对象的目标。可以省略; 然后使用相同的桶。
- destination_object - 给定对象的（重命名）路径。可以省略; 然后使用相同的名称。
- create_bucket(bucket_name, storage_class ='MULTI_REGIONAL', location ='US', project_id = None, labels = None)

创建一个新存储桶。Google 云端存储使用平面命名空间，因此您无法创建名称已在使用中的存储桶。

参数：

- bucket_name(str) - 存储桶的名称。
- storage_class(str) -
  这定义了存储桶中对象的存储方式，并确定了 SLA 和存储成本。价值包括
  - MULTI_REGIONAL
  - REGIONAL
  - STANDARD
  - NEARLINE
  - COLDLINE 。

如果在创建存储桶时未指定此值，则默认为 STANDARD。

- location(str) -

桶的位置。存储桶中对象的对象数据驻留在此区域内的物理存储中。默认为美国。

- project_id(str) - GCP 项目的 ID。

- labels(dict) - 用户提供的键/值对标签。

返回：如果成功，则返回 id 桶的内容。

`delete(bucket, object, generation=None)`

如果未对存储桶启用版本控制，或者使用了生成参数，则删除对象。

参数：

- bucket(str) - 对象所在的存储桶的名称
- object(str) - 要删除的对象的名称
- generation(str) - 如果存在，则永久删除该代的对象

返回：如果成功则为真

下载（bucket, object, filename = None）

从 Google 云端存储中获取文件。

参数：

- bucket(str) - 要获取的存储桶。
- object(str) - 要获取的对象。
- filename(str) - 如果设置，则应写入文件的本地文件路径。

`exists(budket, object)`

检查 Google 云端存储中是否存在文件。

参数：

- bucket(str) - 对象所在的 Google 云存储桶。
- object(str) - 要在 Google 云存储分区中检查的对象的名称。

`get_conn()`

返回 Google 云端存储服务对象。

`get_crc32c(bucket, object)`

获取 Google Cloud Storage 中对象的 CRC32c 校验和。

参数：

- bucket(str) - 对象所在的 Google 云存储桶。
- object(str) - 要在 Google 云存储分区中检查的对象的名称。

`get_md5hash(bucket, object)`

获取 Google 云端存储中对象的 MD5 哈希值。

参数：

- bucket(str) - 对象所在的 Google 云存储桶。
- object(str) - 要在 Google 云存储分区中检查的对象的名称。

`get_size(bucket, object)`

获取 Google 云端存储中文件的大小。

参数：

- bucket(str) - 对象所在的 Google 云存储桶。
- object(str) - 要在 Google 云存储分区中检查的对象的名称。

`is_updated_after(bucket, object, ts)`

检查 Google Cloud Storage 中是否更新了对象。

参数：

- bucket(str) - 对象所在的 Google 云存储桶。
- object(str) - 要在 Google 云存储分区中检查的对象的名称。
- ts(datetime) - 要检查的时间戳。

`list(bucket, versions = None, maxResults = None, prefix = None, delimiter = None)`

使用名称中的给定字符串前缀列出存储桶中的所有对象

参数：

- bucket(str) - 存储桶名称
- versions(bool) - 如果为 true，则列出对象的所有版本
- maxResults(int) - 在单个响应页面中返回的最大项目数
- prefix(str) - 前缀字符串，用于过滤名称以此前缀开头的对象
- delimiter(str) - 根据分隔符过滤对象（例如'.csv'）

返回：与过滤条件匹配的对象名称流

```
rewrite(source_bucket, source_object, destination_bucket, destination_object = None)
```

具有与复制相同的功能，除了可以处理超过 5 TB 的文件，以及在位置和/或存储类之间复制时。

destination_object 可以省略，在这种情况下使用 source_object。

参数：

- source_bucket(str) - 要从中复制的对象的存储桶。
- source_object(str) - 要复制的对象。
- destination_bucket(str) - 要复制到的对象的目标。
- destination_object - 给定对象的（重命名）路径。可以省略; 然后使用相同的名称。

```
upload(bucket, object, filename, mime_type ='application / octet-stream')
```

将本地文件上传到 Google 云端存储。

参数：

- bucket(str) - 要上传的存储桶。
- object(str) - 上传本地文件时要设置的对象名称。
- filename(str) - 要上传的文件的本地文件路径。
- mime_type(str) - 上传文件时要设置的 MIME 类型。

Google Kubernetes 引擎

Google Kubernetes 引擎集群 Operators

- GKEClusterDeleteOperator：在 Google Cloud Platform 中创建 Kubernetes 集群
- GKEPodOperator：删除 Google Cloud Platform 中的 Kubernetes 集群

GKEClusterCreateOperator

```
class airflow.contrib.operators.gcp_container_operator.GKEClusterCreateOperator(project_id,
location, body = {}, gcp_conn_id ='google_cloud_default', api_version ='v2', *args, **kwargs)
```

基类： airflow.models.BaseOperator

GKEClusterDeleteOperator

class airflow.contrib.operators.gcp_container_operator.GKEClusterDeleteOperator(project_id, name, location, gcp_conn_id ='google_cloud_default', api_version ='v2', *args, **kwargs)
基类： airflow.models.BaseOperator

GKEPodOperator

Google Kubernetes Engine Hook

class airflow.contrib.hooks.gcp_container_hook.GKEClusterHook(project_id, location)

基类： airflow.hooks.base_hook.BaseHook

create_cluster(cluster, retry = <object object>, timeout = <object object>)

创建一个集群，由指定数量和类型的 Google Compute Engine 实例组成。

参数：

- cluster(dict 或 google.cloud.container_v1.types.Cluster) - 集群 protobuf 或 dict。如果提供了 dict，它必须与 protobuf 消息的格式相同 google.cloud.container_v1.types.Cluster
- 重试(google.api_core.retry.Retry) - 用于重试请求的重试对象(google.api_core.retry.Retry)。如果指定 None，则不会重试请求。
- timeout(float) - 等待请求完成的时间（以秒为单位）。请注意，如果指定了重试，则超时适用于每次单独尝试。

返回：新集群或现有集群的完整 URL

raise

ParseError：在尝试转换 dict 时出现 JSON 解析问题 AirflowException：cluster 不是 dict 类型也不是 Cluster proto 类型

delete_cluster(name, retry = <object object>, timeout = <object object>)

删除集群，包括 Kubernetes 端点和所有工作节点。在集群创建期间配置的防火墙和路由也将被删除。集群可能正在使用的其他 Google Compute Engine 资源（例如，负载均衡器资源）如果在初始创建时不存在，则不会被删除。

参数：

- name(str) - 要删除的集群的名称
- retry(google.api_core.retry.Retry) - 重试用于确定何时/是否重试请求的对象。如果指定 None，则不会重试请求。
- timeout(float) - 等待请求完成的时间（以秒为单位）。请注意，如果指定了重试，则超时适用于每次单独尝试。

返回：如果成功则删除操作的完整 URL，否则为 None

```
get_cluster(name, retry = <object object>, timeout = <object object>)
```

获取指定集群的详细信息：param name：要检索的集群的名称：type name：str：param retry：用于重试请求的重试对象。如果指定了 None，

请求不会被重试。

参数：timeout(float) - 等待请求完成的时间（以秒为单位）。请注意，如果指定了重试，则超时适用于每次单独尝试。

返回：一个 google.cloud.container_v1.types.Cluster 实例

get_operation(operation_name)

从 Google Cloud 获取操作：param operation_name：要获取的操作的名称：type operation_name：str：return：来自 Google Cloud 的新的更新操作

wait_for_operation (operation)

给定操作，持续从 Google Cloud 获取状态，直到完成或发生错误：param 操作：等待的操作：键入操作：google.cloud.container_V1.gapic.enums.Operator：return：a new，updated 从 Google Cloud 获取的操作

数据血缘(Lineage)

注意： Lineage 支持是实验性的，可能随时会发生变化。

Airflow 可以帮助跟踪数据的来源，以及数据发生了什么变化。 这有助于实现审计跟踪和数据治理，还可以调试数据流。

Airflow 通过任务的 inlets 和 outlets 跟踪数据。 让我们通过一个例子看看它是如何工作的。

```
from airflow.operators.bash_operator import BashOperator
from airflow.operators.dummy_operator import DummyOperator
from airflow.lineage.datasets import File
from airflow.models import DAG
from datetime import timedelta
```

```
FILE_CATEGORIES = ["CAT1", "CAT2", "CAT3"]

args = {
    'owner': 'airflow' ,
    'start_date': airflow.utils.dates.days_ago(2)
}

dag = DAG(
    dag_id='example_lineage', default_args=args,
    schedule_interval='0 0 * * *',
    dagrun_timeout=timedelta(minutes=60))

f_final = File("/tmp/final")
run_this_last = DummyOperator(task_id='run_this_last', dag=dag,
    inlets={"auto": True},
    outlets={"datasets": [f_final,]})

f_in = File("/tmp/whole_directory/")
outlets = []
for file in FILE_CATEGORIES:
    f_out = File("/tmp/{}/{{{{ execution_date }}}}".format(file))
    outlets.append(f_out)
run_this = BashOperator(
    task_id='run_me_first', bash_command='echo 1', dag=dag,
    inlets={"datasets": [f_in,]},
    outlets={"datasets": outlets}
    )
run_this.set_downstream(run_this_last)
```

任务定义了参数 inlets 和 outlets。inlets 可以是一个数据集列表{"datesets":[dataset1,dataset2]}，也可以是指定的上游任务 outlets 像这样{"task_ids": ["task_id1", "task_id2"]}，或者不想指定直接用{"auto":True}也可以，甚至是前面几种的组合。outlets 也是一个数据集列表{"datesets":[dataset1, dataset2]}。 在运行任务时，数据集的字段会被模板渲染。

注意： 只要 Operator 支持，它会自动地加上 inlets 和 outlets。

在示例 DAG 任务中， run_me_first 是一个 BashOperator，它接收 CAT1, CAT2, CAT3 作为 inlets（译注：根据代码，应为“输出 outlets”)。 其中的 execution_date 会在任务运行时被渲染成执行时间。

注意： 在底层，Airflow 会在 pre_execute 方法中准备 lineage 元数据。 当任务运行结束时，会调用 post_execute 将 lineage 元数据推送到 XCOM 中。 因此，如果您要创建自己的 Operator，并且需要覆写这些方法，确保分别用 prepare_lineage 和 apply_lineage 装饰这些方法。

Apache Atlas

Airflow 可以将 lineage 元数据发送到 Apache Atlas。 您需要在 airflow.cfg 中配置 atlas：

```
[lineage]
backend = airflow.lineage.backend.atlas

[atlas]
username = my_username
password = my_password
host = host
port = 21000
```

请确保已经安装了 atlasclient。

常见问题

为什么我的任务没有被调度？

您的任务没有被调度的原因有很多。以下是一些常见原因：

- 您的脚本是否“编译”，Airflow 引擎是否可以解析它并找到您的 DAG 对象。如果要对此进行测试，您可以运行 airflow list_dags 并确认您的 DAG 显示在列表中。 您还可以运行 airflow list_tasks foo_dag_id --tree 并确认您的任务按预期显示在列表中。 如果您使用 CeleryExecutor，您需要确认上面的命令在 scheduler 和 worker 中能如期工作。
- DAG 的文件是否在内容的某处有字符串 “airflow” 和 “DAG”？ 在搜索 DAG 目录时，Airflow 忽略不包含“airflow”和“DAG”的文件，以防止 DagBag 解析导入与用户的 DAG 并置的所有 python 文件。
- 你的 start_date 设置正确吗？ Airflow 会在 start_date + scheduler_interval 时间之后触发任务。
- 您的 schedule_interval 设置正确吗？ 默认 schedule_interval 是一天(datetime.timedelta(1))。您必须在实例化的 DAG 对象的时候直接指定不同的 schedule_interval ，而不是通过 default_param 传递参数，因为 task instances 不会覆盖其父 DAG 的 schedule_interval。
- 您的 start_date 是否超出了在 UI 中可以看到的范围？ 如果将 start_date 设置为 3 个月之前的某个时间，您将无法在 UI 的主视图中看到它，但您应该能够在 Menu -> Browse -> Task Instances 看到它。
- 是否满足 task 的依赖性。 直接位于 task 上游的任务实例需要处于 success 状态。 此外，如果设置 depends_on_past=True ，除了满足上游成功之外，前一个调度周期的 task instance 也需要成功（除非该 task 是第一次运行）。 此外，如果设置了 wait_for_downstream=True，请确保您了解其含义。 您可以从 Task Instance Details 页面查看如何设置这些属性。
- 您需要创建并激活 DagRuns 吗？ DagRun 表示整个 DAG 的特定执行，并具有状态（运行，成功，失败，.....）。 scheduler 在向前移动时创建新的 DagRun，但永远不会及时创建新的 DagRun。scheduler 仅评估 running 状态的 DagRuns 以查看它可以触发的 task instances。 请注意，清除任务实例（从 UI 或 CLI）会将 DagRun 的状态设置为恢复 running。 您可以通过单击 DAG 的计划标记来批量查看 DagRuns 列表并更改状态。

- 是否达到了 DAG 的concurrency参数的上限？ concurrency定义了允许 DAG 在running任务实例的数量，超过这个值，别的就会进入排队队列。
- 是否达到了 DAG 的max_active_runs参数的上限？ max_active_runs定义允许的 DAG running并发实例的数量。

您可能还想阅读文档的 scheduler 部分，并确保完全了解其运行机制。

如何根据其他任务的失败触发任务？

查看文档“概念”中的Trigger Rule部分

安装 airflow[crypto]后，为什么密码在元数据 db 中仍未加密？

查看文档“配置”中的Connections部分

怎么处理start_date？

start_date是前 DagRun 时代的部分遗留问题，但它在很多方面仍然具有相关性。 创建新 DAG 时，您可能希望使用default_args为任务设置全局start_date 。将基于所有任务的min(start_date)创建第一个DagRun 。 从那时起，scheduler 根据您的 schedule_interval 创建新的 DagRuns，并在满足您的依赖项时运行相应的任务实例。 在向 DAG 引入新任务时，您需要特别注意 start_date ，并且可能希望重新激活非活动的 DagRuns 以正确启用新任务。

我们建议不要使用动态值作为start_date ，尤其是datetime.now()因为它非常令人疑惑。 周期结束，任务就会被触发，理论上@hourly DAG 永远不会达到一小时后，因为now()会一直在变化。

以前我们还建议使用与schedule_interval相关的start_date 。这意味着@hourly将在00:00分钟:秒，@daily 会午夜工作，@monthly 在每个月的第一天工作。 这不再是必需的。 现在 Airflow 将自动对齐start_date和schedule_interval ，通过使用start_date作为开始查看的时刻。

您可以使用任何传感器或TimeDeltaSensor来延迟计划间隔内的任务执行。虽然schedule_interval允许指定datetime.timedelta对象，但我们建议使用宏或 cron 表达式作为他的值，因为它强制执行舍入计划的这种想法。

使用 depends_on_past=True 时，必须特别注意 start_date，因为过去的依赖关系不会仅针对为任务指定的 start_date 的特定计划强制执行。 除非您计划为新任务运行 backfill，否则在引入新的depends_on_past=True时及时观察 DagRun 活动状态。

另外需要注意的是，在 backfill CLI 命令的上下文中，任务start_date会被 backfill 命令start_date覆盖。 这允许对具有depends_on_past=True属性任务进行 backfill 操作，如果不是这样涉设计的话，backfill 将无法启动。

如何动态创建 DAG？

Airflow 在DAGS_FOLDER查找全局命名空间中包含DAG对象的模块，并将其添加到DagBag中。 在知道这个原理的情况下，我们需要一种方法分配变量到全局命名空间，这可以在 python 中使用标准库中的

globals()函数轻松完成，就像一个简单的字典。

```
for i in range (10):
    dag_id = 'foo_{} '.format(i)
    globals()[dag_id] = DAG(dag_id)
    # 调用一个函数返回 DAG 对象会更好
```

airflow run 的所有子命令代表什么？

airflow run 命令有很多层，这意味着它可以调用自身。

- 基本的 airflow run：启动 executor，并告诉它运行 airflow run --local 命令。 如果使用 Celery，这意味着它会在队列中放置一个命令，并调用 worker 远程运行。 如果使用 LocalExecutor，则会在子进程池中运行。
- 本地的 airflow run --local：启动 airflow run --raw 命令（如下所述）作为子进程，负责发出心跳，监听外部杀死进程信号，并确保在子进程失败时进行一些清理工作
- 原始的 airflow run --raw 运行实际 operator 的 execute 方法并执行实际工作

怎么使 Airflow dag 运行得更快？

我们可以控制三个变量来改善气流 dag 性能：

- parallelism： 此变量控制 Airflow worker 可以同时运行的任务实例的数量。 用户可以通过改变 airflow.cfg 中的 parallelism 调整 并行度变量。
- concurrency： Airflow scheduler 在任何时间不会运行超过 concurrency 数量的 DAG 实例。 concurrency 在 Airflow DAG 中定义。 如果在 DAG 中没有设置 concurrency，则 scheduler 将使用 airflow.cfg 文件中定义的 dag_concurrency 作为默认值。
- max_active_runs： Airflow scheduler 在任何时间不会运行超过 max_active_runs DagRuns 数量。 如果在 DAG 中没有设置 max_active_runs ，则 scheduler 将使用 airflow.cfg 文件中定义的 max_active_runs_per_dag 作为默认值。

如何减少 Airflow UI 页面加载时间？

如果你的 dag 需要很长时间才能加载，你可以减小 airflow.cfg 中的 default_dag_run_display_number 的值。 此可配置控制在 UI 中显示的 dag run 的数量，默认值为 25。

如何修复异常：Global variable explicit_defaults_for_timestamp needs to be on (1)？

这意味着在 mysql 服务器中禁用了 explicit_defaults_for_timestamp，您需要通过以下方式启用它：

- 在 my.cnf 文件的 mysqld 部分下设置 explicit_defaults_for_timestamp = 1 。
- 重启 Mysql 服务器。

如何减少生产环境中的 Airflow dag 调度延迟？

- max_threads： scheduler 将并行生成多个线程来调度 dags。 这数量是由 max_threads 参数控制，默认值为 2.用户应在生产中将此值增加到更大的值(例如,scheduler 运行机器的 cpus 数量 - 1)。
- scheduler_heartbeat_sec：用户应考虑将 scheduler_heartbeat_sec 配置增加到更高的值(例如 60 秒)，该值控制 airflow scheduler 获取心跳和更新作业到数据库中的频率。

API 参考

运营商

运算符允许生成某些类型的任务，这些任务在实例化时成为 DAG 中的节点。 所有运算符都派生自 BaseOperator ，并以这种方式继承许多属性和方法。 有关更多详细信息，请参阅 BaseOperator 文档。

有三种主要类型的运营商：

- 执行操作的操作员，或告诉其他系统执行操作的操作员
- 传输操作员将数据从一个系统移动到另一个系
- 传感器是某种类型的运算符，它将一直运行直到满足某个标准。 示例包括在 HDFS 或 S3 中登陆的特定文件，在 Hive 中显示的分区或当天的特定时间。 传感器派生自 BaseSensorOperator 并在指定的 poke_interval 运行 poke 方法，直到它返回 True 。

BaseOperator

所有运算符都派生自 BaseOperator 并通过继承获得许多功能。 由于这是引擎的核心，因此值得花时间了解 BaseOperator 的参数，以了解可在 DAG 中使用的原始功能。

```
class airflow.models.BaseOperator(task_id, owner='Airflow', email=None, email_on_retry=True, email_on_failure=True, retries=0, retry_delay=datetime.timedelta(0, 300), retry_exponential_backoff=False, max_retry_delay=None, start_date=None, end_date=None, schedule_interval=None, depends_on_past=False, wait_for_downstream=False, dag=None, params=None, default_args=None, adhoc=False, priority_weight=1, weight_rule=u'downstream', queue='default', pool=None, sla=None, execution_timeout=None, on_failure_callback=None, on_success_callback=None, on_retry_callback=None, trigger_rule=u'all_success', resources=None, run_as_user=None, task_concurrency=None, executor_config=None, inlets=None, outlets=None, *args, **kwargs)
```

基类： airflow.utils.log.logging_mixin.LoggingMixin

所有运营商的抽象基类。 由于运算符创建的对象成为 dag 中的节点，因此 BaseOperator 包含许多用于 dag 爬行行为的递归方法。 要派生此类，您需要覆盖构造函数以及“execute”方法。

从此类派生的运算符应同步执行或触发某些任务（等待完成）。 运算符的示例可以是运行 Pig 作业（PigOperator）的运算符，等待分区在 Hive（HiveSensorOperator）中着陆的传感器运算符，或者是将数据从 Hive 移动到 MySQL（Hive2MySqlOperator）的运算符。 这些运算符（任务）的实例针对特定操作，

运行特定脚本，函数或数据传输。

此类是抽象的，不应实例化。 实例化从这个派生的类导致创建任务对象，该对象最终成为 DAG 对象中的节点。 应使用 set_upstream 和/或 set_downstream 方法设置任务依赖性。

参数：

- task_id(string) - 任务的唯一，有意义的 id

- owner(string) - 使用 unix 用户名的任务的所有者
- retries(int) - 在失败任务之前应执行的重试次数
- retry_delay(timedelta) - 重试之间的延迟
- retry_exponential_backoff(bool) - 允许在重试之间使用指数退避算法在重试之间进行更长时间的等待（延迟将转换为秒）
- max_retry_delay(timedelta) - 重试之间的最大延迟间隔
- start_date(datetime) - 任务的 start_date ，确定第一个任务实例的 execution_date 。 最佳做法是将 start_date 四舍五入到 DAG 的 schedule_interval 。 每天的工作在 000000 有一天的 start_date，每小时的工作在特定时间的 00:00 有 start_date。 请注意，Airflow 只查看最新的 execution_date 并添加 schedule_interval 以确定下一个 execution_date 。 同样非常重要的是要注意不同任务的依赖关系需要及时排列。 如果任务 A 依赖于任务 B 并且它们的 start_date 以其 execution_date 不排列的方式偏移，则永远不会满足 A 的依赖性。 如果您希望延迟任务，例如在凌晨 2 点运行每日任务，请查看 TimeSensor 和 TimeDeltaSensor 。 我们建议不要使用动态 start_date 并建议使用固定的。 有关更多信息，请阅读有关 start_date 的 FAQ 条目。
- end_date(datetime) - 如果指定，则调度程序不会超出此日期
- depends_on_past(bool) - 当设置为 true 时，任务实例将依次运行，同时依赖上一个任务的计划成功。 允许 start_date 的任务实例运行。
- wait_for_downstream(bool) - 当设置为 true 时，任务 X 的实例将等待紧接上一个任务 X 实例下游的任务在运行之前成功完成。 如果任务 X 的不同实例更改同一资产，并且任务 X 的下游任务使用此资产，则此操作非常有用。请注意，在使用 wait_for_downstream 的任何地方，depends_on_past 都将强制为 True。
- queue(str) - 运行此作业时要定位到哪个队列。 并非所有执行程序都实现队列管理，CeleryExecutor 确实支持定位特定队列。
- dag(DAG) - 对任务所附的 dag 的引用（如果有的话）
- priority_weight(int) - 此任务相对于其他任务的优先级权重。 这允许执行程序在事情得到备份时在其他任务之前触发更高优先级的任务。
- weight_rule(str) - 用于任务的有效总优先级权重的加权方法。 选项包括： { downstream | upstream | absolute } { downstream | upstream | absolute } { downstream | upstream | absolute }默认为 downstream 当设置为 downstream 时，任务的有效权重是所有下游后代的总和。 因此，上游任务将具有更高的权重，并且在使用正权重值时将更积极地安排。 当您有多个 dag 运行实例并希望在每个 dag 可以继续处理下游任务之前完成所有运行的所有上游任务时，这非常有用。 设置为 upstream ，有效权重是所有上游祖先的总和。 这与 downtream 任务具有更高权重并且在使用正权重值时将更积极地安排相反。 当你有多个 dag 运行实例并希望在启动其他 dag 的上游任务之前完成每个 dag 时，这非常有用。 设置为 absolute ，有效权重是指定的确切 priority_weight 无需额外加权。当您确切知道每个任务应具有的优先级权重时，您可能希望这样做。

此外，当设置为 absolute ，对于非常大的 DAGS，存在显着加速任务创建过程的额外效果。 可以将选项设置为字符串或使用静态类 airflow.utils.WeightRule 定义的常量

- pool(str) - 此任务应运行的插槽池，插槽池是限制某些任务的并发性的一种方法
- sla(datetime.timedelta) - 作业预期成功的时间。 请注意，这表示期间结束后的 timedelta 。 例如，如果您将 SLA 设置为 1 小时，则如果 2016-01-01 实例尚未成功，则调度程序将在 2016-01-02 凌晨 1:00 之后发送电子邮件。 调度程序特别关注具有 SLA 的作业，并发送警报电子邮件以防止未命中。 SLA 未命中也记录在数据库中以供将来参考。 共享相同 SLA 时间的所有任务都捆绑在一封电子邮件中，在此之后很快就会发送。 SLA 通知仅为每个任务实例发送一次且仅一次。
- execution_timeout(datetime.timedelta) - 执行此任务实例所允许的最长时间，如果超出它将引发并失败。
- on_failure_callback(callable) - 当此任务的任务实例失败时要调用的函数。 上下文字典作为单个参数传递给此函数。 Context 包含对任务实例的相关对象的引用，并记录在 API 的宏部分下。
- on_retry_callback - 与 on_failure_callback 非常相似，只是在重试发生时执行。
- on_success_callback(callable) - 与 on_failure_callback 非常相似，只是在任务成功时执行。
- trigger_rule(str) - 定义应用依赖项以触发任务的规则。 选项包括： { all_success | all_failed | all_done | one_success | one_failed | dummy} { all_success | all_failed | all_done | one_success | one_failed | dummy} { all_success | all_failed | all_done | one_success | one_failed | dummy} 默认为 all_success 。 可以将选项设置为字符串，也可以使用静态类 airflow.utils.TriggerRule 定义的常量
- resources(dict) - 资源参数名称（Resources 构造函数的参数名称）与其值的映射。
- run_as_user(str) - 在运行任务时使用 unix 用户名进行模拟
- task_concurrency(int) - 设置后，任务将能够限制 execution_dates 之间的并发运行
- executor_config(dict) -

由特定执行程序解释的其他任务级配置参数。 参数由 executor 的名称命名。 ``示例：通过 KubernetesExecutor MyOperator（...，在特定的 docker 容器中运行此任务）

> executor_config = {“KubernetesExecutor”: > > > {“image”:“myCustomDockerImage”}}

)``

clear(**kwargs)

根据指定的参数清除与任务关联的任务实例的状态。

dag

如果设置则返回运算符的 DAG，否则引发错误

deps

返回运算符的依赖项列表。 这些与执行上下文依赖性的不同之处在于它们特定于任务，并且可以由子类扩展/覆盖。

`downstream_list`

@property：直接下游的任务列表

`execute(context)`

这是在创建运算符时派生的主要方法。 Context 与渲染 jinja 模板时使用的字典相同。

有关更多上下文，请参阅 get_template_context。

`get_direct_relative_ids(upstream=False)`

获取当前任务（上游或下游）的直接相对 ID。

`get_direct_relatives(upstream=False)`

获取当前任务的直接亲属，上游或下游。

`get_flat_relative_ids(upstream=False, found_descendants=None)`

获取上游或下游的亲属 ID 列表。

`get_flat_relatives(upstream=False)`

获取上游或下游亲属的详细列表。

`get_task_instances(session, start_date=None, end_date=None)`

获取与特定日期范围的此任务相关的一组任务实例。

`has_dag()`

如果已将运算符分配给 DAG，则返回 True。

`on_kill()`

当任务实例被杀死时，重写此方法以清除子进程。 在操作员中使用线程，子过程或多处理模块的任何使用都需要清理，否则会留下重影过程。

`post_execute(context, *args, **kwargs)`

调用 self.execute（）后立即触发此挂接。 它传递执行上下文和运算符返回的任何结果。

pre_execute(context, *args, **kwargs)

在调用 self.execute（）之前触发此挂钩。

prepare_template()

模板化字段被其内容替换后触发的挂钩。 如果您需要操作员在呈现模板之前更改文件的内容，则应覆盖此方法以执行此操作。

render_template(attr, content, context)

从文件或直接在字段中呈现模板，并返回呈现的结果。

render_template_from_field(attr, content, context, jinja_env)

从字段中呈现模板。 如果字段是字符串，它将只是呈现字符串并返回结果。 如果它是集合或嵌套的集合集，它将遍历结构并呈现其中的所有字符串。

run(start_date=None, end_date=None, ignore_first_depends_on_past=False, ignore_ti_state=False, mark_success=False)

为日期范围运行一组任务实例。

schedule_interval

DAG 的计划间隔始终胜过单个任务,因此 DAG 中的任务始终排列。该任务仍然需要 schedule_interval，因为它可能未附加到 DAG。

set_downstream(task_or_task_list)

将任务或任务列表设置为直接位于当前任务的下游。

set_upstream(task_or_task_list)

将任务或任务列表设置为直接位于当前任务的上游。

upstream_list

@property：直接上游的任务列表

xcom_pull(context, task_ids=None, dag_id=None, key=u'return_value', include_prior_dates=None)

请参见 TaskInstance.xcom_pull（）

```
xcom_push(context, key, value, execution_date=None)
```

请参见 TaskInstance.xcom_push（）

BaseSensorOperator

所有传感器均来自 BaseSensorOperator 。 所有传感器都在 BaseOperator 属性之上继承 timeout 和 poke_interval 。

```
class airflow.sensors.base_sensor_operator.BaseSensorOperator(poke_interval=60, timeout=604800, soft_fail=False, *args, **kwargs)
```

基类： airflow.models.BaseOperator ， airflow.models.SkipMixin

传感器操作符派生自此类继承这些属性。

```
Sensor operators keep executing at a time interval and succeed when
```
如果标准超时，则会达到并且失败。

参数：

- soft_fail(bool) - 设置为 true 以在失败时将任务标记为 SKIPPED
- poke_interval(int) - 作业在每次尝试之间应等待的时间（以秒为单位）
- timeout(int) - 任务超时和失败前的时间（以秒为单位）。

```
poke(context)
```

传感器在派生此类时定义的功能应该覆盖。

核心运营商

运营商

```
class airflow.operators.bash_operator.BashOperator(bash_command, xcom_push=False, env=None, output_encoding='utf-8', *args, **kwargs)
```

基类： airflow.models.BaseOperator

执行 Bash 脚本，命令或命令集。

参数：

- bash_command(string) - 要执行的命令，命令集或对 bash 脚本（必须为'.sh'）的引用。 （模板）

- xcom_push(bool) – 如果 xcom_push 为 True，写入 stdout 的最后一行也将在 bash 命令完成时被推送到 XCom。
- env(dict) – 如果 env 不是 None，它必须是一个定义新进程的环境变量的映射; 这些用于代替继承当前进程环境，这是默认行为。 （模板）

```
execute(context)
```

在临时目录中执行 bash 命令，之后将对其进行清理

```
class airflow.operators.python_operator.BranchPythonOperator(python_callable, op_args=None, op_kwargs=None, provide_context=False, templates_dict=None, templates_exts=None, *args, **kwargs)
```

基类： airflow.operators.python_operator.PythonOperator ， airflow.models.SkipMixin

允许工作流在执行此任务后“分支”或遵循单个路径。

它派生 PythonOperator 并期望一个返回 task_id 的 Python 函数。返回的 task_id 应指向{self}下游的任务。 所有其他“分支”或直接下游任务都标记为 skipped 状态，以便这些路径不能向前移动。 skipped 状态在下游被提出以允许 DAG 状态填满并且推断 DAG 运行的状态。

请注意，在 depends_on_past=True 中使用 depends_on_past=True 下游的任务在逻辑上是不合理的，因为 skipped 状态将总是导致依赖于过去成功的块任务。 skipped 状态在所有直接上游任务被 skipped 地方传播。

```
class airflow.operators.check_operator.CheckOperator(sql, conn_id=None, *args, **kwargs)
```

基类： airflow.models.BaseOperator

对 db 执行检查。 CheckOperator 需要一个返回单行的 SQL 查询。 第一行的每个值都使用 python bool cast 进行计算。 如果任何值返回 False 则检查失败并输出错误。

请注意，Python bool cast 会将以下内容视为 False ：

- False
- 0
- 空字符串（ "" ）
- 空列表（ [] ）
- 空字典或集（ {} ）

给定 SELECT COUNT(*) FROM foo 类的查询，只有当 count == 0 它才会失败。 您可以制作更复杂的查询，例如，可以检查表与上游源表的行数相同，或者今天的分区计数大于昨天的分区，或者一组指标是否更少 7 天平均值超过 3 个标准差。

此运算符可用作管道中的数据质量检查，并且根据您在 DAG 中的位置，您可以选择停止关键路径，防止发

布可疑数据，或者侧面接收电子邮件警报阻止 DAG 的进展。

请注意，这是一个抽象类，需要定义 get_db_hook。 而 get_db_hook 是钩子，它从外部源获取单个记录。

参数：sql(string) - 要执行的 sql。 （模板）

```
class airflow.operators.docker_operator.DockerOperator(image, api_version=None, command=None, cpus=1.0, docker_url='unix://var/run/docker.sock', environment=None, force_pull=False, mem_limit=None, network_mode=None, tls_ca_cert=None, tls_client_cert=None, tls_client_key=None, tls_hostname=None, tls_ssl_version=None, tmp_dir='/tmp/airflow', user=None, volumes=None, working_dir=None, xcom_push=False, xcom_all=False, docker_conn_id=None, *args, **kwargs)
```

基类： airflow.models.BaseOperator

在 docker 容器中执行命令。

在主机上创建临时目录并将其装入容器，以允许在容器中存储超过默认磁盘大小 10GB 的文件。 可以通过环境变量 AIRFLOW_TMP_DIR 访问已安装目录的路径。

如果在提取映像之前需要登录私有注册表，则需要在 Airflow 中配置 Docker 连接，并使用参数 docker_conn_id 提供连接 ID。

参数：

- image(str) - 用于创建容器的 Docker 镜像。
- api_version(str) - 远程 API 版本。 设置为 auto 以自动检测服务器的版本。
- command(str 或 list) - 要在容器中运行的命令。 （模板）
- cpus(float) - 分配给容器的 CPU 数。 此值乘以 1024. 请参阅 https://docs.docker.com/engine/reference/run/#cpu-share-constraint
- docker_url(str) - 运行 docker 守护程序的主机的 URL。 默认为 unix：//var/run/docker.sock
- environment(dict) - 要在容器中设置的环境变量。 （模板）
- force_pull(bool) - 每次运行时拉动泊坞窗图像。 默认值为 false。
- mem_limit(float 或 str) - 容器可以使用的最大内存量。 浮点值（表示以字节为单位的限制）或字符串（如 128m 或 1g 。
- network_mode(str) - 容器的网络模式。
- tls_ca_cert(str) - 用于保护 docker 连接的 PEM 编码证书颁发机构的路径。
- tls_client_cert(str) - 用于验证 docker 客户端的 PEM 编码证书的路径。
- tls_client_key(str) - 用于验证 docker 客户端的 PEM 编码密钥的路径。
- tls_hostname(str 或 bool) - 与 docker 服务器证书匹配的主机名或 False 以禁用检查。
- tls_ssl_version(str) - 与 docker 守护程序通信时使用的 SSL 版本。
- tmp_dir(str) - 将容器内的挂载点挂载到操作员在主机上创建的临时目录。 该路径也可通过容器内的环境变量 AIRFLOW_TMP_DIR 获得。
- user(int 或 str) - docker 容器内的默认用户。
- 卷 - 要装入容器的卷列表， 例如 ['/host/path:/container/path',

'/host/path2:/container/path2:ro'] 。

- working_dir(str) – 在容器上设置的工作目录（相当于-w 切换 docker 客户端）
- xcom_push(bool) – 是否会使用 XCom 将 stdout 推送到下一步。 默认值为 False。
- xcom_all(bool) – 推送所有标准输出或最后一行。 默认值为 False（最后一行）。
- docker_conn_id(str) – 要使用的 Airflow 连接的 ID
- class airflow.operators.dummy_operator.DummyOperator(*args, **kwargs)

基类： airflow.models.BaseOperator

操作员确实没什么。 它可用于对 DAG 中的任务进行分组。

class airflow.operators.druid_check_operator.DruidCheckOperator(sql, druid_broker_conn_id='druid_broker_default', *args, **kwargs)

基类： airflow.operators.check_operator.CheckOperator

对德鲁伊进行检查。DruidCheckOperator需要一个返回单行的 SQL 查询。第一行的每个值都使用 python bool cast 进行计算。 如果任何值返回False则检查失败并输出错误。

请注意，Python bool cast 会将以下内容视为False ：

- False
- 0
- 空字符串（ "" ）
- 空列表（ [] ）
- 空字典或集（ {} ）

给定 SELECT COUNT(*) FROM foo 类的查询，只有当 count == 0 它才会失败。 您可以制作更复杂的查询，例如，可以检查表与上游源表的行数相同，或者今天的分区计数大于昨天的分区，或者一组指标是否更少 7 天平均值超过 3 个标准差。 此运算符可用作管道中的数据质量检查，并且根据您在 DAG 中的位置，您可以选择停止关键路径，防止发布可疑数据，或者在旁边接收电子邮件替代品阻止 DAG 的进展。

参数：

- sql(string) – 要执行的 sql
- druid_broker_conn_id(string) – 对德鲁伊经纪人的提及

get_db_hook()

返回德鲁伊 db api 钩子。

get_first(sql)

执行德鲁伊 sql 到德鲁伊经纪人并返回第一个结果行。

参数：sql(str) – 要执行的 sql 语句（str）

```
class airflow.operators.email_operator.EmailOperator(to, subject, html_content, files=None, cc=None, bcc=None, mime_subtype='mixed', mime_charset='us_ascii', *args, **kwargs)
```

基类： airflow.models.BaseOperator

发送电子邮件。

参数：

- to(list 或 string （ 逗号 或 分号分隔 ）) - 要发送电子邮件的电子邮件列表。 （模板）
- subject(string) - 电子邮件的主题行。 （模板）
- html_content(string) - 电子邮件的内容，允许使用 html 标记。 （模板）
- files(list) - 要在电子邮件中附加的文件名
- cc(list 或 string _（ 逗号 或 分号分隔 ）) - 要在 CC 字段中添加的收件人列表
- 密件抄送(list 或 string （ 逗号 或 分号分隔 ）) - 要在 BCC 字段中添加的收件人列表
- mime_subtype(string) - MIME 子内容类型
- mime_charset(string) - 添加到 Content-Type 标头的字符集参数。

```
class airflow.operators.generic_transfer.GenericTransfer(sql, destination_table, source_conn_id, destination_conn_id, preoperator=None, *args, **kwargs)
```

基类： airflow.models.BaseOperator

将数据从连接移动到另一个连接，假设它们都在各自的钩子中提供所需的方法。 源钩子需要公开 get_records 方法，目标是 insert_rows 方法。

这适用于适合内存的小型数据集。

参数：

- sql(str) - 针对源数据库执行的 SQL 查询。 （模板）
- destination_table(str) - 目标表。 （模板）
- source_conn_id(str) - 源连接
- destination_conn_id(str) - 源连接
- preoperator(str 或 list[str]) - sql 语句或在加载数据之前要执行的语句列表。 （模板）

```
class airflow.operators.hive_to_druid.HiveToDruidTransfer(sql, druid_datasource, ts_dim, metric_spec=None, hive_cli_conn_id='hive_cli_default', druid_ingest_conn_id='druid_ingest_default', metastore_conn_id='metastore_default', hadoop_dependency_coordinates=None, intervals=None, num_shards=-1, target_partition_size=-1, query_granularity='NONE', segment_granularity='DAY', hive_tblproperties=None, *args, **kwargs)
```

基类： airflow.models.BaseOperator

将数据从 Hive 移动到 Druid，[del]注意到现在数据在被推送到 Druid 之前被加载到内存中，因此该运算符应该用于少量数据。[/ del]

参数：

- sql(str) - 针对 Druid 数据库执行的 SQL 查询。 （模板）
- druid_datasource(str) - 您想要在德鲁伊中摄取的数据源
- ts_dim(str) - 时间戳维度
- metric_spec(list) - 您要为数据定义的指标
- hive_cli_conn_id(str) - 配置单元连接 ID
- druid_ingest_conn_id(str) - 德鲁伊摄取连接 ID
- metastore_conn_id(str) - Metastore 连接 ID
- hadoop_dependency_coordinates(list[str]) - 用于挤压摄取 json 的坐标列表
- interval(list) - 定义段的时间间隔列表，按原样传递给 json 对象。 （模板）
- hive_tblproperties(dict) - 用于登台表的 hive 中 tblproperties 的其他属性

```
construct_ingest_query(static_path, columns)
```

构建 HDFS TSV 负载的接收查询。

参数：

- static_path(str) - 数据所在的 hdfs 上的路径
- columns(list) - 所有可用列的列表

```
class          airflow.operators.hive_to_mysql.HiveToMySqlTransfer(sql,          mysql_table,
hiveserver2_conn_id='hiveserver2_default',                    mysql_conn_id='mysql_default',
mysql_preoperator=None, mysql_postoperator=None, bulk_load=False, *args, **kwargs)
```

基类： airflow.models.BaseOperator

将数据从 Hive 移动到 MySQL，请注意，目前数据在被推送到 MySQL 之前已加载到内存中，因此该运算符应该用于少量数据。

参数：

- sql(str) - 对 Hive 服务器执行的 SQL 查询。 （模板）
- mysql_table(str) - 目标 MySQL 表，使用点表示法来定位特定数据库。 （模板）
- mysql_conn_id(str) - 源码 mysql 连接
- hiveserver2_conn_id(str) - 目标配置单元连接
- mysql_preoperator(str) - 在导入之前对 mysql 运行的 sql 语句，通常用于截断删除代替进入的数据，允许任务是幂等的（运行任务两次不会加倍数据）。 （模板）
- mysql_postoperator(str) - 导入后针对 mysql 运行的 sql 语句，通常用于将数据从登台移动到

生产并发出清理命令。 （模板）

- bulk_load(bool) - 使用 bulk_load 选项的标志。 这使用 LOAD DATA LOCAL INFILE 命令直接从制表符分隔的文本文件加载 mysql。 此选项需要为目标 MySQL 连接提供额外的连接参数：{'local_infile'：true}。

class airflow.operators.hive_to_samba_operator.Hive2SambaOperator(hql, destination_filepath, samba_conn_id='samba_default', hiveserver2_conn_id='hiveserver2_default', *args, **kwargs)

基类： airflow.models.BaseOperator

在特定的 Hive 数据库中执行 hql 代码，并将查询结果作为 csv 加载到 Samba 位置。

参数：

- hql(string) - 要导出的 hql。 （模板）
- hiveserver2_conn_id(string) - 对 hiveserver2 服务的引用
- samba_conn_id(string) - 对 samba 目标的引用

class airflow.operators.hive_operator.HiveOperator(hql, hive_cli_conn_id=u'hive_cli_default', schema=u'default', hiveconfs=None, hiveconf_jinja_translate=False, script_begin_tag=None, run_as_owner=False, mapred_queue=None, mapred_queue_priority=None, mapred_job_name=None, *args, **kwargs)

基类： airflow.models.BaseOperator

在特定的 Hive 数据库中执行 hql 代码或 hive 脚本。

参数：

- hql(string) - 要执行的 hql。 请注意，您还可以使用（模板）配置单元脚本的 dag 文件中的相对路径。 （模板）
- hive_cli_conn_id(string) - 对 Hive 数据库的引用。 （模板）
- hiveconfs(dict) - 如果已定义，这些键值对将作为-hiveconf "key"="value"传递给 hive
- hiveconf_jinja_translate(_boolean_) - 当为 True 时，hiveconf-type 模板$ {var}被翻译成 jinja-type templating {{var}}，$ {hiveconf：var}被翻译成 jinja-type templating {{var}}。请注意，您可能希望将此选项与 DAG(user_defined_macros=myargs)参数一起使用。 查看 DAG 对象文档以获取更多详细信息。
- script_begin_tag(str) - 如果已定义，运算符将在第一次出现 script_begin_tag 之前删除脚本的一部分
- mapred_queue(string) - Hadoop CapacityScheduler 使用的队列。 （模板）
- mapred_queue_priority(string) - CapacityScheduler 队列中的优先级。 可能的设置包括：VERY_HIGH，HIGH，NORMAL，LOW，VERY_LOW
- mapred_job_name(string) - 此名称将出现在 jobtracker 中。 这可以使监控更容易。

```
class airflow.operators.hive_stats_operator.HiveStatsCollectionOperator(table, partition, extra_exprs=None, col_blacklist=None, assignment_func=None, metastore_conn_id='metastore_default', presto_conn_id='presto_default', mysql_conn_id='airflow_db', *args, **kwargs)
```

基类： airflow.models.BaseOperator

使用动态生成的 Presto 查询收集分区统计信息，将统计信息插入到具有此格式的 MySql 表中。 如果重新运行相同的日期/分区，统计信息将覆盖自己。

```
CREATE TABLE hive_stats (
    ds VARCHAR ( 16 ),
    table_name VARCHAR ( 500 ),
    metric VARCHAR ( 200 ),
    value BIGINT
);
```

参数：

- table(str) – 源表，格式为 database.table_name 。 （模板）
- partition(dict{col:value}) – 源分区。 （模板）
- extra_exprs(dict) – 针对表运行的表达式，其中键是度量标准名称，值是 Presto 兼容表达式
- col_blacklist(list) – 列入黑名单的列表，考虑列入黑名单，大型 json 列，.....
- assignment_func(function) – 接收列名和类型的函数，返回度量标准名称和 Presto 表达式的 dict。 如果返回 None，则应用全局默认值。 如果返回空字典，则不会为该列计算统计信息。

```
class airflow.operators.check_operator.IntervalCheckOperator(table, metrics_thresholds, date_filter_column='ds', days_back=-7, conn_id=None, *args, **kwargs)
```

基类： airflow.models.BaseOperator

检查作为 SQL 表达式给出的度量值是否在 days_back 之前的某个容差范围内。

请注意，这是一个抽象类，需要定义 get_db_hook。 而 get_db_hook 是钩子，它从外部源获取单个记录。

参数：

- table(str) – 表名
- days_back(int) – ds 与我们要检查的 ds 之间的天数。 默认为 7 天
- metrics_threshold(dict) – 由指标索引的比率字典

```
class airflow.operators.jdbc_operator.JdbcOperator(sql, jdbc_conn_id='jdbc_default', autocommit=False, parameters=None, *args, **kwargs)
```

基类： airflow.models.BaseOperator

使用 jdbc 驱动程序在数据库中执行 sql 代码。

需要 jaydebeapi。

参数：

- jdbc_conn_id(string) - 对预定义数据库的引用
- sql(可以接收表示 sql 语句的 str,list[str](sql 语句)或模板文件的引用。模板引用由以 '.sql' 结尾的 str 识别) - 要执行的 sql 代码。 （模板）

```
class airflow.operators.latest_only_operator.LatestOnlyOperator(task_id, owner='Airflow', email=None, email_on_retry=True, email_on_failure=True, retries=0, retry_delay=datetime.timedelta(0, 300), retry_exponential_backoff=False, max_retry_delay=None, start_date=None, end_date=None, schedule_interval=None, depends_on_past=False, wait_for_downstream=False, dag=None, params=None, default_args=None, adhoc=False, priority_weight=1, weight_rule=u'downstream', queue='default', pool=None, sla=None, execution_timeout=None, on_failure_callback=None, on_success_callback=None, on_retry_callback=None, trigger_rule=u'all_success', resources=None, run_as_user=None, task_concurrency=None, executor_config=None, inlets=None, outlets=None, *args, **kwargs)
```

基类：airflow.models.BaseOperator，airflow.models.SkipMixin

允许工作流跳过在最近的计划间隔期间未运行的任务。

如果任务在最近的计划间隔之外运行，则将跳过所有直接下游任务。

```
class airflow.operators.mssql_operator.MsSqlOperator（sql，mssql_conn_id ='mssql_default'，parameters = None，autocommit = False，database = None，* args，** kwargs）
```

基类： airflow.models.BaseOperator

在特定的 Microsoft SQL 数据库中执行 sql 代码

参数：

- mssql_conn_id(string) - 对特定 mssql 数据库的引用
- sql(_ 指向扩展名为.sql 的模板文件的 _string 或 _ 字符串。_ _（_ _ 模板化 _ _）_) - 要执行的 sql 代码
- database(string) - 覆盖连接中定义的数据库的数据库名称

```
class airflow.operators.mssql_to_hive.MsSqlToHiveTransfer（sql，hive_table，create = True，recreate = False，partition = None，delimiter = u'x01'，mssql_conn_id ='mssql_default'，
```

hive_cli_conn_id ='hive_cli_default', tblproperties = None, * args , ** kwargs)

基类： airflow.models.BaseOperator

将数据从 Microsoft SQL Server 移动到 Hive。操作员针对 Microsoft SQL Server 运行查询，在将文件加载到 Hive 表之前将其存储在本地。如果将 create 或 recreate 参数设置为 True，则生成 a 和语句。从游标的元数据推断出 Hive 数据类型。请注意，在 Hive 中生成的表使用的不是最有效的序列化格式。如果加载了大量数据和/或表格被大量查询，您可能只想使用此运算符将数据暂存到临时表中，然后使用 a 将其加载到最终目标中。CREATE TABLE``DROP TABLE``STORED AS textfile``HiveOperator

参数：

- sql(str) - 针对 Microsoft SQL Server 数据库执行的 SQL 查询。（模板）
- hive_table(str) - 目标 Hive 表，使用点表示法来定位特定数据库。（模板）
- create(bool) - 是否创建表，如果它不存在
- recreate(bool) - 是否在每次执行时删除并重新创建表
- partition(dict) - 将目标分区作为分区列和值的字典。（模板）
- delimiter(str) - 文件中的字段分隔符
- mssql_conn_id(str) - 源 Microsoft SQL Server 连接
- hive_conn_id(str) - 目标配置单元连接
- tblproperties(dict) - 正在创建的 hive 表的 TBLPROPERTIES

class airflow.operators.mysql_operator.MySqlOperator（sql, mysql_conn_id ='mysql_default', parameters = None, autocommit = False, database = None, * args, ** kwargs)

基类： airflow.models.BaseOperator

在特定的 MySQL 数据库中执行 sql 代码

参数：

- mysql_conn_id(string) - 对特定 mysql 数据库的引用
- SQL(_ 可接收表示 SQL 语句中的海峡 _ _，_ _ 海峡列表 _ _（_ _SQL 语句 _ _）_ _，或 _ _ 参照模板文件模板引用在“.SQL”结束海峡认可。_) -要执行的 SQL 代码。（模板）
- database(string) - 覆盖连接中定义的数据库的数据库名称

class airflow.operators.mysql_to_hive.MySqlToHiveTransfer（sql, hive_table, create = True, recreate = False, partition = None, delimiter = u'x01', mysql_conn_id ='mysql_default', hive_cli_conn_id ='hive_cli_default', tblproperties = None, * args , ** kwargs)

基类： airflow.models.BaseOperator

将数据从 MySql 移动到 Hive。操作员针对 MySQL 运行查询，在将文件加载到 Hive 表之前将文件存储在本地。如果将 create 或 recreate 参数设置为 True，则生成 a 和语句。从游标的元数据推断出 Hive 数

据类型。请注意，在 Hive 中生成的表使用的不是最有效的序列化格式。如果加载了大量数据和/或表格被大量查询，您可能只想使用此运算符将数据暂存到临时表中，然后使用 a 将其加载到最终目标中。CREATE TABLE``DROP TABLE``STORED AS textfile``HiveOperator

参数：

- sql(str) - 针对 MySQL 数据库执行的 SQL 查询。（模板）
- hive_table(str) - 目标 Hive 表，使用点表示法来定位特定数据库。（模板）
- create(bool) - 是否创建表，如果它不存在
- recreate(bool) - 是否在每次执行时删除并重新创建表
- partition(dict) - 将目标分区作为分区列和值的字典。（模板）
- delimiter(str) - 文件中的字段分隔符
- mysql_conn_id(str) - 源码 mysql 连接
- hive_conn_id(str) - 目标配置单元连接
- tblproperties(dict) - 正在创建的 hive 表的 TBLPROPERTIES

class airflow.operators.oracle_operator.OracleOperator（sql，oracle_conn_id ='oracle_default'，parameters = None，autocommit = False，* args，** kwargs）

基类： airflow.models.BaseOperator

在特定的 Oracle 数据库中执行 sql 代码：param oracle_conn_id：对特定 Oracle 数据库的引用：type oracle_conn_id：string：param sql：要执行的 sql 代码。（模板化）：type sql：可以接收表示 sql 语句的 str，

str（sql 语句）列表或对模板文件的引用。模板引用由以'.sql'结尾的 str 识别

class airflow.operators.pig_operator.PigOperator（pig，pig_cli_conn_id ='pig_cli_default'，pigparams_jinja_translate = False，* args，** kwargs）

基类： airflow.models.BaseOperator

执行猪脚本。

参数：

- pig(string) - 要执行的猪拉丁文字。（模板）
- pig_cli_conn_id(string) - 对 Hive 数据库的引用
- pigparams_jinja_translate(_boolean_) - 当为 True 时，猪 params 类型的模板$ {var}被转换为 jinja-type templating {{var}}。请注意，您可能希望将此 DAG(user_defined_macros=myargs)参数与参数一起使用。查看 DAG 对象文档以获取更多详细信息。

class airflow.operators.postgres_operator.PostgresOperator（sql，postgres_conn_id ='postgres_default'，autocommit = False，parameters = None，database = None，* args，** kwargs）

基类： airflow.models.BaseOperator

在特定的 Postgres 数据库中执行 sql 代码

参数：

- postgres_conn_id(string) - 对特定 postgres 数据库的引用
- SQL(_ 可接收表示 SQL 语句中的海峡 _ _，_ _ 海峡列表 _ _（_ _SQL 语句 _ _）_ _，或 _ _ 参照模板文件模板引用在“.SQL”结束海峡认可。_) -要执行的 SQL 代码。（模板）
- database(string) - 覆盖连接中定义的数据库的数据库名称

```
class airflow.operators.presto_check_operator.PrestoCheckOperator （ sql ， presto_conn_id ='presto_default'，* args，** kwargs）
```

基类： airflow.operators.check_operator.CheckOperator

对 Presto 执行检查。该 PrestoCheckOperator 预期的 SQL 查询将返回一行。使用 python bool 强制转换评估第一行的每个值。如果任何值返回，False 则检查失败并输出错误。

请注意，Python bool 强制转换如下 False：

- False
- 0
- 空字符串（""）
- 空列表（[]）
- 空字典或集（{}）

给定一个查询，它只会在计数时失败。您可以制作更复杂的查询，例如，可以检查表与上游源表的行数相同，或者今天的分区计数大于昨天的分区，或者一组指标是否更少 7 天平均值超过 3 个标准差。SELECT COUNT(*) FROM foo``== 0

此运算符可用作管道中的数据质量检查，并且根据您在 DAG 中的位置，您可以选择停止关键路径，防止发布可疑数据，或者在旁边接收电子邮件替代品阻止 DAG 的进展。

参数：

- sql(string) - 要执行的 sql
- presto_conn_id(string) - 对 Presto 数据库的引用

```
class airflow.operators.presto_check_operator.PrestoIntervalCheckOperator （ table ， metrics_thresholds，date_filter_column ='ds'，days_back = -7，presto_conn_id ='presto_default'，* args，** kwargs）
```

基类： airflow.operators.check_operator.IntervalCheckOperator

检查作为 SQL 表达式给出的度量值是否在 days_back 之前的某个容差范围内。

参数：

- table(str) - 表名
- days_back(int) - ds 与我们要检查的 ds 之间的天数。默认为 7 天
- metrics_threshold(dict) - 由指标索引的比率字典
- presto_conn_id(string) - 对 Presto 数据库的引用

```
class airflow.operators.presto_to_mysql.PrestoToMySqlTransfer(sql,mysql_table,presto_conn_id='presto_default', mysql_conn_id='mysql_default', mysql_preoperator=None, *args, **kwargs)
```

基类： airflow.models.BaseOperator

将数据从 Presto 移动到 MySQL，注意到现在数据在被推送到 MySQL 之前被加载到内存中，因此该运算符应该用于少量数据。

参数：

- sql(str) - 对 Presto 执行的 SQL 查询。(模板)
- mysql_table(str) - 目标 MySQL 表，使用点表示法来定位特定数据库。(模板)
- mysql_conn_id(str) - 源码 mysql 连接
- presto_conn_id(str) - 源 presto 连接
- mysql_preoperator(str) - 在导入之前对 mysql 运行的 sql 语句，通常用于截断删除代替进入的数据，允许任务是幂等的（运行任务两次不会加倍数据）。(模板)

```
class airflow.operators.presto_check_operator.PrestoValueCheckOperator (sql, pass_value, tolerance=None, presto_conn_id='presto_default', *args, **kwargs)
```

基类： airflow.operators.check_operator.ValueCheckOperator

使用 sql 代码执行简单的值检查。

参数：

- sql(string) - 要执行的 sql
- presto_conn_id(string) - 对 Presto 数据库的引用

```
class airflow.operators.python_operator.PythonOperator (python_callable, op_args=None, op_kwargs=None, provide_context=False, templates_dict=None, templates_exts=None, *args, **kwargs)
```

基类： airflow.models.BaseOperator

执行 Python 可调用

参数：

- python_callable(_python callable_) - 对可调用对象的引用
- op_kwargs(dict) - 一个关键字参数的字典，将在你的函数中解压缩
- op_args(list) - 调用 callable 时将解压缩的位置参数列表
- provide_context(bool) - 如果设置为 true，Airflow 将传递一组可在函数中使用的关键字参数。这组 kwargs 完全对应于你在 jinja 模板中可以使用的内容。为此,您需要在函数头中定义** kwargs。
- templates_dict(_STR 的字典 _) --一本字典其中的值将由发动机气流之间的某个时候得到模板模板__init__和 execute 发生和已应用模板后，您可调用的上下文中提供。(模板)
- templates_exts(_list __(_ _str __) _) - 例如，在处理模板化字段时要解析的文件扩展名列表['.sql', '.hql']

class airflow.operators.python_operator.PythonVirtualenvOperator (python_callable , requirements = None，python_version = None，use_dill = False，system_site_packages = True，op_args = None，op_kwargs = None，string_args = None，templates_dict = None，templates_exts = None，* args， ** kwargs）

基类： airflow.operators.python_operator.PythonOperator

允许一个人在自动创建和销毁的 virtualenv 中运行一个函数（有一些警告)。

该函数必须使用 def 定义，而不是类的一部分。所有导入必须在函数内部进行，并且不能引用范围之外的变量。名为 virtualenvstringargs 的全局范围变量将可用（由 string_args 填充)。另外，可以通过 op_args 和 op_kwargs 传递内容，并且可以使用返回值。

请注意，如果您的 virtualenv 运行在与 Airflow 不同的 Python 主要版本中，则不能使用返回值，op_args 或 op_kwargs。你可以使用 string_args。

参数：

- python_callable(function) - 一个没有引用外部变量的 python 函数，用 def 定义，将在 virtualenv 中运行
- requirements(_list __(_ _str __) _) - pip install 命令中指定的要求列表
- python_version(str) - 用于运行 virtualenv 的 Python 版本。请注意，2 和 2.7 都是可接受的形式。
- use_dill(bool) - 是否使用 dill 序列化 args 和结果(pickle 是默认值)。这允许更复杂的类型，但要求您在您的要求中包含莳萝。
- system_site_packages(bool) - 是否在 virtualenv 中包含 system_site_packages。有关更多信息，请参阅 virtualenv 文档。
- op_args - 要传递给 python_callable 的位置参数列表。

- op_kwargs(dict) - 传递给 python_callable 的关键字参数的字典。
- string_args(_list __（_ _str __）_) - 全局 var virtualenvstringargs 中存在的字符串，在运行时可用作列表（str）的 python_callable。请注意，args 按换行符分割。
- templates_dict(_STR 的字典 _)--一本字典,其中值是将某个之间的气流引擎获得模板模板__init__和 execute 发生，并取得了您的可调用的上下文提供的模板已被应用后，
- templates_exts(_list __（_ _str __）_) - 例如，在处理模板化字段时要解析的文件扩展名列表['.sql', '.hql']

class airflow.operators.s3_file_transform_operator.S3FileTransformOperator（source_s3_key，dest_s3_key,transform_script=None,select_expression=None,source_aws_conn_id='aws_default',dest_aws_conn_id='aws_default', replace = False, * args, ** kwargs）

基类： airflow.models.BaseOperator

将数据从源 S3 位置复制到本地文件系统上的临时位置。根据转换脚本的指定对此文件运行转换，并将输出上载到目标 S3 位置。

本地文件系统中的源文件和目标文件的位置作为转换脚本的第一个和第二个参数提供。转换脚本应该从源读取数据，转换它并将输出写入本地目标文件。然后，操作员接管控制并将本地目标文件上载到 S3。

S3 Select 也可用于过滤源内容。如果指定了 S3 Select 表达式，则用户可以省略转换脚本。

参数：

- source_s3_key(str) - 从 S3 检索的密钥。（模板）
- source_aws_conn_id(str) - 源 s3 连接
- dest_s3_key(str) - 从 S3 写入的密钥。（模板）
- dest_aws_conn_id(str) - 目标 s3 连接
- replace(bool) - 替换 dest S3 密钥（如果已存在）
- transform_script(str) - 可执行转换脚本的位置
- select_expression(str) - S3 选择表达式

class airflow.operators.s3_to_hive_operator.S3ToHiveTransfer（s3_key，field_dict，hive_table，delimiter=',',create=True，recreate=False，partition=None，headers=False，check_headers=False，wildcard_match=False，aws_conn_id='aws_default'，hive_cli_conn_id='hive_cli_default'，input_compressed=False，tblproperties=None，select_expression=None，* args，** kwargs）

基类： airflow.models.BaseOperator

将数据从 S3 移动到 Hive。操作员从 S3 下载文件，在将文件加载到 Hive 表之前将其存储在本地。如果将 create 或 recreate 参数设置为 True，则生成 a 和语句。Hive 数据类型是从游标的元数据中推断出来的。CREATE TABLE``DROP TABLE

请注意，在 Hive 中生成的表使用的不是最有效的序列化格式。如果加载了大量数据和/或表格被大量查询，您可能只想使用此运算符将数据暂存到临时表中，然后使用 a 将其加载到最终目标中。STORED AS textfile``HiveOperator

参数：

- s3_key(str) - 从 S3 检索的密钥。（模板）
- field_dict(dict) - 字段的字典在文件中命名为键，其 Hive 类型为值
- hive_table(str) - 目标 Hive 表，使用点表示法来定位特定数据库。（模板）
- create(bool) - 是否创建表，如果它不存在
- recreate(bool) - 是否在每次执行时删除并重新创建表
- partition(dict) - 将目标分区作为分区列和值的字典。（模板）
- headers(bool) - 文件是否包含第一行的列名
- check_headers(bool) - 是否应该根据 field_dict 的键检查第一行的列名
- wildcard_match(bool) - 是否应将 s3_key 解释为 Unix 通配符模式
- delimiter(str) - 文件中的字段分隔符
- aws_conn_id(str) - 源 s3 连接
- hive_cli_conn_id(str) - 目标配置单元连接
- input_compressed(bool) - 布尔值，用于确定是否需要文件解压缩来处理标头
- tblproperties(dict) - 正在创建的 hive 表的 TBLPROPERTIES
- select_expression(str) - S3 选择表达式

class airflow.operators.s3_to_redshift_operator.S3ToRedshiftTransfer(schema,table,s3_bucket, s3_key, redshift_conn_id ='redshift_default', aws_conn_id ='aws_default', copy_options = (), autocommit = False, parameters = None, * args, ** kwargs)

基类： airflow.models.BaseOperator

执行 COPY 命令将文件从 s3 加载到 Redshift

参数：

- schema(string) - 对 redshift 数据库中特定模式的引用
- table(string) - 对 redshift 数据库中特定表的引用
- s3_bucket(string) - 对特定 S3 存储桶的引用
- s3_key(string) - 对特定 S3 密钥的引用
- redshift_conn_id(string) - 对特定 redshift 数据库的引用
- aws_conn_id(string) - 对特定 S3 连接的引用
- copy_options(list) - 对 COPY 选项列表的引用

class airflow.operators.python_operator.ShortCircuitOperator(python_callable,op_args=None, op_kwargs=None, provide_context=False, templates_dict=None, templates_exts=None, *args, ** kwargs)

基类：airflow.operators.python_operator.PythonOperator，airflow.models.SkipMixin

仅在满足条件时才允许工作流继续。否则，将跳过工作流程“短路”和下游任务。

ShortCircuitOperator 派生自 PythonOperator。如果条件为 False，它会评估条件并使工作流程短路。任何下游任务都标记为“已跳过”状态。如果条件为 True，则下游任务正常进行。

条件由 python_callable 的结果决定。

class airflow.operators.http_operator.SimpleHttpOperator(endpoint,method='POST',data=None, headers = None，response_check = None，extra_options = None，xcom_push = False，http_conn_id='http_default'，* args，** kwargs）

基类： airflow.models.BaseOperator

在 HTTP 系统上调用端点以执行操作

参数：

- http_conn_id(string) - 运行传感器的连接
- endpoint(string) - 完整 URL 的相对部分。（模板）
- method(string) - 要使用的 HTTP 方法，default =“POST”
- data(_ 对于 POST / PUT __，_ _ 取决于 content-type 参数 _ _，_ _ 用于 GET 键/值字符串对的字典 _) - 要传递的数据。POST / PUT 中的 POST 数据和 GET 请求的 URL 中的 params。（模板）
- headers(_ 字符串键/值对的字典 _) - 要添加到 GET 请求的 HTTP 头
- response_check(_lambda _ 或 _ 定义的函数。_) - 检查'requests'响应对象。对于'pass'返回 True，否则返回 False。
- extra_options(_ 选项字典 _ _，_ _ 其中键是字符串，值取决于正在修改的选项。_) - 'requests'库的额外选项，请参阅'requests'文档（修改超时，ssl 等选项）

class airflow.operators.slack_operator.SlackAPIOperator（slack_conn_id = None，token = None，method = None，api_params = None，* args，** kwargs）

基类： airflow.models.BaseOperator

Base Slack 运算符 SlackAPIPostOperator 派生自此运算符。在未来，还将从此类派生其他 Slack API 操作符

参数：

- slack_conn_id(string) - Slack 连接 ID，其密码为 Slack API 令牌
- token(string) - Slack API 令牌（https://api.slack.com/web）
- method(string) - 要调用的 Slack API 方法（https://api.slack.com/methods）
- api_params(dict) - API 方法调用参数（https://api.slack.com/methods）

construct_api_call_params（）

由 execute 函数使用。允许在构造之前对 api_call_params dict 的源字段进行模板化

覆盖子类。每个 SlackAPIOperator 子类都负责使用 construct_api_call_params 函数，该函数使用 API 调用参数的字典设置 self.api_call_params（https://api.slack.com/methods）

执行（** kwargs）
即使调用不成功，SlackAPIOperator 调用也不会失败。它不应该阻止 DAG 成功完成

class airflow.operators.slack_operator.SlackAPIPostOperator（channel ='＃general', username ='Airflow', text ='

没有设置任何消息。这是一个猫视频而不是 http://www.youtube.com/watch？v = J --- aiyznGQ', icon_url ='https ：//raw.githubusercontent.com/airbnb/airflow/master/airflow/www/static/pin_100.png'，附件=无，* args，** kwargs）

基类： airflow.operators.slack_operator.SlackAPIOperator

将消息发布到松弛通道

参数：

- channel(string) - 在松弛名称（#general）或 ID（C12318391）上发布消息的通道。（模板）
- username(string) - 气流将发布到 Slack 的用户名。（模板）
- text(string) - 要发送到 slack 的消息。（模板）
- icon_url(string) - 用于此消息的图标的 url
- 附件（哈希数组） - 额外的格式详细信息。（模板化） - 请参阅 https://api.slack.com/docs/attachments。

construct_api_call_params（）

由 execute 函数使用。允许在构造之前对 api_call_params dict 的源字段进行模板化

覆盖子类。每个 SlackAPIOperator 子类都负责使用 construct_api_call_params 函数，该函数使用 API 调用参数的字典设置 self.api_call_params（https://api.slack.com/methods）

class airflow.operators.sqlite_operator.SqliteOperator(sql,sqlite_conn_id ='sqlite_default', parameters = None，* args，** kwargs）

基类： airflow.models.BaseOperator

在特定的 Sqlite 数据库中执行 sql 代码

参数：

- sqlite_conn_id(string) - 对特定 sqlite 数据库的引用
- sql(_ 指向模板文件的 _string 或 _ 字符串。文件必须具有'.sql'扩展名。_) - 要执行的 sql 代码。（模板）

class airflow.operators.subdag_operator.SubDagOperator（** kwargs）

基类： airflow.models.BaseOperator

class airflow.operators.dagrun_operator.TriggerDagRunOperator(trigger_dag_id,python_callable = None, execution_date = None, * args, ** kwargs)

基类： airflow.models.BaseOperator

触发指定的 DAG 运行 dag_id

参数：

- trigger_dag_id(str) - 要触发的 dag_id
- python_callable(_python callable_) - 对 python 函数的引用，在传递 context 对象时将调用该对象,obj 并且如果要创建 DagRun,则可调用的占位符对象可以填充并返回。这个 obj 对象包含 run_id 和 payload 属性，你可以在你的函数修改。本 run_id 应为 DAG 运行的唯一标识符，有效载荷必须是在执行该 DAG 运行，这将提供给你的任务 picklable 对象。你的函数头应该是这样的 def foo(context, dag_run_obj):
- execution_date(_datetime.datetime_) - dag 的执行日期

class airflow.operators.check_operator.ValueCheckOperator（sql，pass_value，tolerance = None，conn_id = None，* args，** kwargs）

基类： airflow.models.BaseOperator

使用 sql 代码执行简单的值检查。

请注意，这是一个抽象类，需要定义 get_db_hook。而 get_db_hook 是钩子，它从外部源获取单个记录。

参数：sql(string) - 要执行的 sql。（模板）

class airflow.operators.redshift_to_s3_operator.RedshiftToS3Transfer(schema,table,s3_bucket,s3_key，redshift_conn_id ='redshift_default'，aws_conn_id ='aws_default'，unload_options = ()，autocommit = False，parameters = None，include_header = False，* args，* * kwargs)

基类： airflow.models.BaseOperator

执行 UNLOAD 命令，将 s3 作为带标题的 CSV

参数：

- schema(string) - 对 redshift 数据库中特定模式的引用
- table(string) - 对 redshift 数据库中特定表的引用
- s3_bucket(string) - 对特定 S3 存储桶的引用
- s3_key(string) - 对特定 S3 密钥的引用
- redshift_conn_id(string) - 对特定 redshift 数据库的引用
- aws_conn_id(string) - 对特定 S3 连接的引用
- unload_options(list) - 对 UNLOAD 选项列表的引用

传感器

class airflow.sensors.external_task_sensor.ExternalTaskSensor（external_dag_id，external_task_id,allowed_states=None,execution_delta=None,execution_date_fn=None,*args,** kwargs）

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

等待任务在不同的 DAG 中完成

参数：

- external_dag_id(string) - 包含您要等待的任务的 dag_id
- external_task_id(string) - 包含您要等待的任务的 task_id
- allowed_states(list) - 允许的状态列表，默认为['success']
- execution_delta(datetime.timedelta) - 与上一次执行的时间差异来看，默认是与当前任务相同的 execution_date。对于昨天，使用[positive！]datetime.timedelta（days=1）。execution_delta 或 execution_date_fn 可以传递给 ExternalTaskSensor，但不能同时传递给两者。
- execution_date_fn(callable) - 接收当前执行日期并返回所需执行日期以进行查询的函数。execution_delta 或 execution_date_fn 可以传递给 ExternalTaskSensor，但不能同时传递给两者。

戳（** kwargs）

传感器在派生此类时定义的功能应该覆盖。

class airflow.sensors.hdfs_sensor.HdfsSensor（filepath，hdfs_conn_id='hdfs_default'，ignored_ext=['_COPYING_']，ignore_copying=True，file_size=None，hook=<class'airflow.hooks.hdfs_hook.HDFSHook'>，* args，** kwargs）

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

等待文件或文件夹降落到 HDFS 中

static filter_for_filesize（result，size = None）

将测试文件路径结果并测试其大小是否至少为 self.filesize

参数：

- 结果 - Snakebite ls 返回的 dicts 列表
- size - 文件大小（MB）文件应该至少触发 True

返回值： （bool）取决于匹配标准

static filter_for_ignored_ext（result，ignored_ext，ignore_copying）

如果指示过滤将删除匹配条件的结果

参数：

- 结果 - Snakebite ls 返回的 dicts（列表）
- ignored_ext - 忽略扩展名的（列表）
- ignore_copying - （bool）我们会忽略吗？

返回值： （清单）未删除的词典

戳（上下文）

传感器在派生此类时定义的功能应该覆盖。

class airflow.sensors.hive_partition_sensor.HivePartitionSensor（table，partition =“ds='{{ds}}'”，metastore_conn_id ='metastore_default'，schema ='default'，poke_interval = 180，*args，** kwargs）

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

等待分区显示在 Hive 中。

注意：因为 partition 支持通用逻辑运算符，所以效率低下。如果您不需要 HivePartitionSensor 的完全灵活性，请考虑使用 NamedHivePartitionSensor。

参数：

- table(string) - 要等待的表的名称，支持点表示法（my_database.my_table）
- partition(string) - 要等待的分区子句。这是原样传递给 Metastore Thrift 客户端 get_partitions_by_filter 方法，显然支持 SQL 作为符号和比较运算符，如 ds='2015-01-01' AND

type='value'``"ds>=2015-01-01"

- metastore_conn_id(str) - 对 Metastore thrift 服务连接 id 的引用

戳（上下文）

传感器在派生此类时定义的功能应该覆盖。

class airflow.sensors.http_sensor.HttpSensor（endpoint，http_conn_id ='http_default'，method ='GET'，request_params = None，headers = None，response_check = None，extra_options = None，* args，** kwargs）

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

执行 HTTP get 语句并在失败时返回 False：

找不到 404 或者 response_check 函数返回 False

参数：

- http_conn_id(string) - 运行传感器的连接
- method(string) - 要使用的 HTTP 请求方法
- endpoint(string) - 完整 URL 的相对部分
- request_params(_ 字符串键/值对的字典 _) - 要添加到 GET URL 的参数
- headers(_ 字符串键/值对的字典 _) - 要添加到 GET 请求的 HTTP 头
- response_check(_lambda _ 或 _ 定义的函数。_) - 检查'requests'响应对象。对于'pass'返回 True，否则返回 False。
- extra_options(_ 选项字典 _ _，_ _ 其中键是字符串，值取决于正在修改的选项。_) - 'requests'库的额外选项，请参阅'requests'文档（修改超时，ssl 等选项）

戳（上下文）

传感器在派生此类时定义的功能应该覆盖。

class airflow.sensors.metastore_partition_sensor.MetastorePartitionSensor（table，partition_name，schema ='default'，mysql_conn_id ='metastore_mysql'，* args，** kwargs）

基类： airflow.sensors.sql_sensor.SqlSensor

HivePartitionSensor 的替代方案，直接与 MySQL 数据库对话。这是在观察子分区表时 Metastore thrift 服务生成的子最优查询的结果。Thrift 服务的查询是以不利用索引的方式编写的。

参数：

- schema(str) - 架构

- table(str) - 表格
- partition_name(str) - 分区名称，在 Metastore 的 PARTITIONS 表中定义。字段顺序很重要。示例：ds=2016-01-01 或 ds=2016-01-01/sub=foo 用于子分区表
- mysql_conn_id(str) - 对 Metastore 的 MySQL conn_id 的引用

戳（上下文）

传感器在派生此类时定义的功能应该覆盖。

class airflow.sensors.named_hive_partition_sensor.NamedHivePartitionSensor（partition_names，metastore_conn_id ='metastore_default'，poke_interval = 180，hook = None，* args，** kwargs）

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

等待一组分区显示在 Hive 中。

参数：

- partition_names(_ 字符串列表 _) - 要等待的分区的完全限定名称列表。完全限定名称的格式 schema.table/pk1=pv1/pk2=pv2 为，例如，default.users / ds = 2016-01-01。这将原样传递给 Metastore Thrift 客户端 get_partitions_by_name 方法。请注意，您不能像在 HivePartitionSensor 中那样使用逻辑或比较运算符。
- metastore_conn_id(str) - 对 Metastore thrift 服务连接 id 的引用

戳（上下文）

传感器在派生此类时定义的功能应该覆盖。

class airflow.sensors.s3_key_sensor.S3KeySensor（bucket_key，bucket_name = None，wildcard_match = False，aws_conn_id ='aws_default'，* args，** kwargs）

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

等待 S3 存储桶中存在密钥（S3 上的文件类实例）。S3 是键/值，它不支持文件夹。路径只是资源的关键。

参数：

- bucket_key(str) - 正在等待的密钥。支持完整的 s3：//样式 URL 或从根级别的相对路径。
- bucket_name(str) - S3 存储桶的名称
- wildcard_match(bool) - 是否应将 bucket_key 解释为 Unix 通配符模式
- aws_conn_id(str) - 对 s3 连接的引用

戳（上下文）

传感器在派生此类时定义的功能应该覆盖。

class airflow.sensors.s3_prefix_sensor.S3PrefixSensor（bucket_name，prefix，delimiter ='/'，aws_conn_id ='aws_default'，* args，** kwargs）

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

等待前缀存在。前缀是键的第一部分，因此可以检查类似于 glob airfl *或 SQL LIKE'airfl％'的构造。有可能精确定界符来指示层次结构或键，这意味着匹配将停止在该定界符处。当前代码接受合理的分隔符，即 Python 正则表达式引擎中不是特殊字符的字符。

参数：

- bucket_name(str) - S3 存储桶的名称
- prefix(str) - 等待的前缀。桶根级别的相对路径。
- delimiter(str) - 用于显示层次结构的分隔符。默认为'/'。

戳（上下文）

传感器在派生此类时定义的功能应该覆盖。

class airflow.sensors.sql_sensor.SqlSensor（conn_id，sql，* args，** kwargs）

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

运行 sql 语句，直到满足条件。它将继续尝试，而 sql 不返回任何行，或者如果第一个单元格返回（0，'0'，''）。

参数：

- conn_id(string) - 运行传感器的连接
- sql - 要运行的 sql。要传递，它需要返回至少一个包含非零/空字符串值的单元格。

戳（上下文）

传感器在派生此类时定义的功能应该覆盖。

class airflow.sensors.time_sensor.TimeSensor（target_time，* args，** kwargs）

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

等到当天的指定时间。

参数：target_time(_datetime.time_) - 作业成功的时间

戳（上下文）

传感器在派生此类时定义的功能应该覆盖。

class airflow.sensors.time_delta_sensor.TimeDeltaSensor（delta，* args，** kwargs）

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

在任务的 execution_date + schedule_interval 之后等待 timedelta。在 Airflow 中，标有 execution_date2016-01-01 的每日任务只能在 2016-01-02 开始运行。timedelta 在此表示执行期间结束后的时间。

参数：delta(datetime.timedelta) - 成功之前在 execution_date 之后等待的时间长度

戳（上下文）

传感器在派生此类时定义的功能应该覆盖。

class airflow.sensors.web_hdfs_sensor.WebHdfsSensor（filepath，webhdfs_conn_id='webhdfs_default'，* args，** kwargs）

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

等待文件或文件夹降落到 HDFS 中

戳（上下文）

传感器在派生此类时定义的功能应该覆盖。

社区贡献的运营商

运营商

class airflow.contrib.operators.awsbatch_operator.AWSBatchOperator（job_name，job_definition，job_queue，overrides，max_retries = 4200，aws_conn_id = None，region_name = None，** kwargs）
基类： airflow.models.BaseOperator

在 AWS Batch Service 上执行作业

参数：

- job_name(str) - 将在 AWS Batch 上运行的作业的名称
- job_definition(str) - AWS Batch 上的作业定义名称

- job_queue(str) - AWS Batch 上的队列名称
- max_retries(int) - 服务器未合并时的指数退避重试，4200 = 48 小时
- aws_conn_id(str) - AWS 凭证/区域名称的连接 ID。如果为 None，将使用凭证 boto3 策略（http://boto3.readthedocs.io/en/latest/guide/configuration.html）。
- region_name - 要在 AWS Hook 中使用的区域名称。覆盖连接中的 region_name（如果提供）

| 帕拉姆： | 覆盖：boto3 将在 containerOverrides 上接收的相同参数（模板化）：http://boto3.readthedocs.io/en/latest/reference/services/batch.html#submit_job 类型： 覆盖：dict

class airflow.contrib.operators.bigquery_check_operator.BigQueryCheckOperator （ sql ， bigquery_conn_id ='bigquery_default'， * args， ** kwargs）

基类： airflow.operators.check_operator.CheckOperator

对 BigQuery 执行检查。该 BigQueryCheckOperator 预期的 SQL 查询将返回一行。使用 python bool 强制转换评估第一行的每个值。如果任何值返回，False 则检查失败并输出错误。

请注意，Python bool 强制转换如下 False：

- False
- 0
- 空字符串（""）
- 空列表（[]）
- 空字典或集（{}）

给定一个查询，它只会在计数时失败。您可以制作更复杂的查询，例如，可以检查表与上游源表的行数相同，或者今天的分区计数大于昨天的分区，或者一组指标是否更少 7 天平均值超过 3 个标准差。SELECT COUNT(*) FROM foo``== 0

此运算符可用作管道中的数据质量检查，并且根据您在 DAG 中的位置，您可以选择停止关键路径，防止发布可疑数据，或者在旁边接收电子邮件替代品阻止 DAG 的进展。

参数：

- sql(string) - 要执行的 sql
- bigquery_conn_id(string) - 对 BigQuery 数据库的引用

class airflow.contrib.operators.bigquery_check_operator.BigQueryValueCheckOperator （ sql ， pass_value， tolerance = None， bigquery_conn_id ='bigquery_default'， * args， ** kwargs）

基类： airflow.operators.check_operator.ValueCheckOperator

使用 sql 代码执行简单的值检查。

参数：sql(string) - 要执行的 sql

class airflow.contrib.operators.bigquery_check_operator.BigQueryIntervalCheckOperator（table, metrics_thresholds , date_filter_column ='ds' , days_back = -7 , bigquery_conn_id ='bigquery_default', * args, ** kwargs）

基类： airflow.operators.check_operator.IntervalCheckOperator

检查作为 SQL 表达式给出的度量值是否在 days_back 之前的某个容差范围内。

此方法构造一个类似的查询

```
SELECT { metrics_thresholddictkey } FROM { table }
    WHERE { date_filter_column } =< date >
```

参数：

- table(str) - 表名
- days_back(int) - ds 与我们要检查的 ds 之间的天数。默认为 7 天
- metrics_threshold(dict) - 由指标索引的比率字典，例如'COUNT（*）'：1.5 将需要当前日和之前的 days_back 之间 50％或更小的差异。

class airflow.contrib.operators.bigquery_get_data.BigQueryGetDataOperator （ dataset_id , table_id, max_results ='100', selected_fields = None, bigquery_conn_id ='bigquery_default', delegate_to = None, * args, ** kwargs）

基类： airflow.models.BaseOperator

从 BigQuery 表中获取数据（或者为所选列获取数据）并在 python 列表中返回数据。返回列表中的元素数将等于获取的行数。列表中的每个元素将再次是一个列表，其中元素将表示该行的列值。

结果示例：[['Tony', '10'], ['Mike', '20'], ['Steve', '15']]

注意

如果传递的字段 selected_fields 的顺序与 BQ 表中已有的列的顺序不同，则数据仍将按 BQ 表的顺序排列。例如，如果 BQ 表有 3 列，[A,B,C]并且您传递'B，selected_fields 那么数据中的 A' 仍然是表格'A,B'。

示例 ：

```
get_data = BigQueryGetDataOperator (
    task_id = 'get_data_from_bq' ,
    dataset_id = 'test_dataset' ,
    table_id = 'Transaction_partitions' ,
```

```
    max_results = '100',
    selected_fields = 'DATE',
    bigquery_conn_id = 'airflow-service-account'
)
```

参数：

- dataset_id - 请求的表的数据集 ID。（模板）
- table_id(string) - 请求表的表 ID。（模板）
- max_results(string) - 从表中获取的最大记录数（行数）。（模板）
- selected_fields(string) - 要返回的字段列表（逗号分隔）。如果未指定，则返回所有字段。
- bigquery_conn_id(string) - 对特定 BigQuery 钩子的引用。
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。

```
class           airflow.contrib.operators.bigquery_operator.BigQueryCreateEmptyTableOperator
（dataset_id, table_id, project_id = None, schema_fields = None, gcs_schema_object = None,
time_partitioning = {}, bigquery_conn_id ='bigquery_default', google_cloud_storage_conn_id
='google_cloud_default', delegate_to = None, * args , ** kwargs）
```

基类： airflow.models.BaseOperator

在指定的 BigQuery 数据集中创建一个新的空表，可选择使用模式。

可以用两种方法之一指定用于 BigQuery 表的模式。您可以直接传递架构字段，也可以将运营商指向 Google 云存储对象名称。Google 云存储中的对象必须是包含架构字段的 JSON 文件。您还可以创建没有架构的表。

参数：

- project_id(string) - 将表创建的项目。（模板）
- dataset_id(string) - 用于创建表的数据集。（模板）
- table_id(string) - 要创建的表的名称。（模板）
- schema_fields(list) -

如果设置，则此处定义的架构字段列表：https://cloud.google.com/bigquery/docs/reference/rest/v2/jobs#configuration.load.schema

示例：

```
schema_fields = [{ "name" : "emp_name" , "type" : "STRING" , "mode" : "REQUIRED" },
              { "name" : "salary" , "type" : "INTEGER" , "mode" : "NULLABLE" }]
```

- gcs_schema_object(string) - 包含模式（模板化）的 JSON 文件的完整路径。例如：gs://test-bucket/dir1/dir2/employee_schema.json

- time_partitioning(dict) -

配置可选的时间分区字段，即按 API 规范按字段，类型和到期分区。

- bigquery_conn_id(string) - 对特定 BigQuery 挂钩的引用。
- google_cloud_storage_conn_id(string) - 对特定 Google 云存储挂钩的引用。
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。

示例（在 GCS 中使用模式 JSON）：

```
CreateTable = BigQueryCreateEmptyTableOperator (
    task_id = 'BigQueryCreateEmptyTableOperator_task',
    dataset_id = 'ODS',
    table_id = 'Employees',
    project_id = 'internal-gcp-project',
    gcs_schema_object = 'gs://schema-bucket/employee_schema.json',
    bigquery_conn_id = 'airflow-service-account',
    google_cloud_storage_conn_id = 'airflow-service-account'
)
```

对应的 Schema 文件（employee_schema.json）：

```
[
  {
    "mode" : "NULLABLE",
    "name" : "emp_name",
    "type" : "STRING"
  },
  {
    "mode" : "REQUIRED",
    "name" : "salary",
    "type" : "INTEGER"
  }
]
```

示例（在 DAG 中使用模式）：

```
CreateTable = BigQueryCreateEmptyTableOperator (
    task_id = 'BigQueryCreateEmptyTableOperator_task',
    dataset_id = 'ODS',
    table_id = 'Employees',
    project_id = 'internal-gcp-project',
    schema_fields = [{ "name" : "emp_name" , "type" : "STRING" , "mode" : "REQUIRED" },
                     { "name" : "salary" , "type" : "INTEGER" , "mode" : "NULLABLE" }],
```

```
    bigquery_conn_id = 'airflow-service-account',
    google_cloud_storage_conn_id = 'airflow-service-account'
)
```

```
class airflow.contrib.operators.bigquery_operator.BigQueryCreateExternalTableOperator(bucket, source_objects, destination_project_dataset_table, schema_fields=None, schema_object=None, source_format='CSV', compression='NONE', skip_leading_rows=0, field_delimiter=',', max_bad_records=0, quote_character=None, allow_quoted_newlines=False, allow_jagged_rows=False, bigquery_conn_id='bigquery_default', google_cloud_storage_conn_id='google_cloud_default', delegate_to=None, src_fmt_configs={}, *args, **kwargs)
```

基类： airflow.models.BaseOperator

使用 Google 云端存储中的数据在数据集中创建新的外部表。

可以用两种方法之一指定用于 BigQuery 表的模式。您可以直接传递架构字段，也可以将运营商指向 Google 云存储对象名称。Google 云存储中的对象必须是包含架构字段的 JSON 文件。

参数：

- bucket(string) - 指向外部表的存储桶。（模板）
- source_objects - 指向表格的 Google 云存储 URI 列表。（模板化）如果 source_format 是 'DATASTORE_BACKUP'，则列表必须只包含一个 URI。
- destination_project_dataset_table(string) - 用于将数据加载到（模板化）的虚线（<project>。）<dataset>。<table> BigQuery 表。如果未包含<project>，则项目将是连接 json 中定义的项目。
- schema_fields(list) -

如果设置，则此处定义的架构字段列表：https://cloud.google.com/bigquery/docs/reference/rest/v2/jobs#configuration.load.schema

示例：

```
schema_fields=[{"name":"emp_name","type":"STRING","mode":"REQUIRED"},
               {"name":"salary","type":"INTEGER","mode":"NULLABLE"}]
```

当 source_format 为'DATASTORE_BACKUP'时，不应设置。

- schema_object - 如果设置，则指向包含表的架构的.json 文件的 GCS 对象路径。（模板）
- schema_object - 字符串
- source_format(string) - 数据的文件格式。
- compression(string) - [可选]数据源的压缩类型。可能的值包括 GZIP 和 NONE。默认值为 NONE。Google Cloud Bigtable，Google Cloud Datastore 备份和 Avro 格式会忽略此设置。

- skip_leading_rows(int) - 从 CSV 加载时要跳过的行数。
- field_delimiter(string) - 用于 CSV 的分隔符。
- max_bad_records(int) - BigQuery 在运行作业时可以忽略的最大错误记录数。
- quote_character(string) - 用于引用 CSV 文件中数据部分的值。
- allow_quoted_newlines(_boolean_) - 是否允许引用的换行符（true）或不允许（false）。
- allow_jagged_rows(bool) - 接受缺少尾随可选列的行。缺失值被视为空值。如果为 false，则缺少尾随列的记录将被视为错误记录，如果错误记录太多，则会在作业结果中返回无效错误。仅适用于 CSV，忽略其他格式。
- bigquery_conn_id(string) - 对特定 BigQuery 挂钩的引用。
- google_cloud_storage_conn_id(string) - 对特定 Google 云存储挂钩的引用。
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。
- src_fmt_configs(dict) - 配置特定于源格式的可选字段

```
class airflow.contrib.operators.bigquery_operator.BigQueryOperator（bql = None，sql = None，destination_dataset_table = False，write_disposition ='WRITE_EMPTY'，allow_large_results = False，flatten_results = False，bigquery_conn_id ='bigquery_default'，delegate_to = None，udf_config = False ，use_legacy_sql = True，maximum_billing_tier = None，maximum_bytes_billed = None，create_disposition ='CREATE_IF_NEEDED'，schema_update_options =()，query_params = None，priority ='INTERACTIVE'，time_partitioning = {}，* args，** kwargs）
```

基类： airflow.models.BaseOperator

在特定的 BigQuery 数据库中执行 BigQuery SQL 查询

参数：

- BQL(_ 可接收表示 SQL 语句中的海峡 _ _，_ _ 海峡列表 _ _（_ _SQL 语句 _ _）_ _，或 _ _ 参照模板文件模板引用在“.SQL”结束海峡认可。_) - （不推荐使用。SQL 参数代替）要执行的 sql 代码（模板化）
- SQL(_ 可接收表示 SQL 语句中的海峡 _ _，_ _ 海峡列表 _ _（_ _SQL 语句 _ _）_ _，或 _ _ 参照模板文件模板引用在“.SQL”结束海峡认可。_) - SQL 代码被执行（模板）
- destination_dataset_table(string) - 一个虚线（<project>。| <project>：）<dataset>。<table>，如果设置，将存储查询结果。（模板）
- write_disposition(string) - 指定目标表已存在时发生的操作。（默认：'WRITE_EMPTY'）
- create_disposition(string) - 指定是否允许作业创建新表。（默认值：'CREATE_IF_NEEDED'）
- allow_large_results(_boolean_) - 是否允许大结果。
- flatten_results(_boolean_) - 如果为 true 且查询使用旧版 SQL 方言，则展平查询结果中的所有嵌套和重复字段。allow_large_results 必须是 true 如果设置为 false。对于标准 SQL 查询，将忽略此标志，并且结果永远不会展平。
- bigquery_conn_id(string) - 对特定 BigQuery 钩子的引用。
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。
- udf_config(list) - 查询的用户定义函数配置。有关详细信息，请参阅 https://cloud.google.com/bigquery/user-defined-functions。
- use_legacy_sql(_boolean_) - 是使用旧 SQL（true）还是标准 SQL（false）。

- maximum_billing_tier(_ 整数 _) - 用作基本价格乘数的正整数。默认为 None，在这种情况下，它使用项目中设置的值。
- maximum_bytes_billed(float) - 限制为此作业计费的字节数。超出此限制的字节数的查询将失败(不会产生费用)。如果未指定，则将其设置为项目默认值。
- schema_update_options(_tuple_) - 允许更新目标表的模式作为加载作业的副作用。
- query_params(dict) - 包含查询参数类型和值的字典，传递给 BigQuery。
- priority(string) - 指定查询的优先级。可能的值包括 INTERACTIVE 和 BATCH。默认值为 INTERACTIVE。
- time_partitioning(dict) - 配置可选的时间分区字段，即按 API 规范按字段，类型和到期分区。请注意，'field'不能与 dataset.table $ partition 一起使用。

```
class airflow.contrib.operators.bigquery_table_delete_operator.BigQueryTableDeleteOperator(deletion_dataset_table, bigquery_conn_id='bigquery_default', delegate_to=None, ignore_if_missing=False, *args, **kwargs)
```

基类： airflow.models.BaseOperator

删除 BigQuery 表

参数：

- deletion_dataset_table(string) - 一个虚线(<project>。| <project>:)<dataset>。<table>，指示将删除哪个表。(模板)
- bigquery_conn_id(string) - 对特定 BigQuery 钩子的引用。
- delegate_to(string) - 模拟的帐户(如果有)。为此，发出请求的服务帐户必须启用域范围委派。
- ignore_if_missing(_boolean_) - 如果为 True，则即使请求的表不存在也返回成功。

```
class airflow.contrib.operators.bigquery_to_bigquery.BigQueryToBigQueryOperator(source_project_dataset_tables, destination_project_dataset_table, write_disposition='WRITE_EMPTY', create_disposition='CREATE_IF_NEEDED', bigquery_conn_id='bigquery_default', delegate_to=None, *args, **kwargs)
```

基类： airflow.models.BaseOperator

将数据从一个 BigQuery 表复制到另一个。

也可以看看有关这些参数的详细信息，请访问：https//cloud.google.com/bigquery/docs/reference/v2/jobs#configuration.copy

参数：

- source_project_dataset_tables(_list | string_) - 一个或多个点(项目:|项目。)<dataset>。<table>用作源数据的 BigQuery 表。如果未包含<project>，则项目将是连接 json 中定义的项目。如果有多个源表，请使用列表。(模板)

- destination_project_dataset_table(string) - 目标 BigQuery 表。格式为:(project:|project。)<dataset>。<table>(模板化)
- write_disposition(string) - 表已存在时的写处置。
- create_disposition(string) - 如果表不存在，则创建处置。
- bigquery_conn_id(string) - 对特定 BigQuery 钩子的引用。
- delegate_to(string) - 模拟的帐户(如果有)。为此，发出请求的服务帐户必须启用域范围委派。

```
class airflow.contrib.operators.bigquery_to_gcs.BigQueryToCloudStorageOperator(source_project_dataset_table, destination_cloud_storage_uris, compression='NONE', export_format='CSV', field_delimiter=',', print_header=True, bigquery_conn_id='bigquery_default', delegate_to=None, *args, **kwargs)
```

基类: airflow.models.BaseOperator

将 BigQuery 表传输到 Google Cloud Storage 存储桶。

也可以看看有关这些参数的详细信息，请访问：https://cloud.google.com/bigquery/docs/reference/v2/jobs

参数:

- source_project_dataset_table(string) - 用作源数据的虚线(<project>。|<project>:)<dataset>。<table> BigQuery 表。如果未包含<project>，则项目将是连接 json 中定义的项目。(模板)
- destination_cloud_storage_uris(list) - 目标 Google 云端存储 URI(例如 gs://some-bucket/some-file.txt)。(模板化)遵循此处定义的惯例：https://cloud.google.com/bigquery/exporting-data-from-bigquery#exportingmultiple
- compression(string) - 要使用的压缩类型。
- export_format - 要导出的文件格式。
- field_delimiter(string) - 提取到 CSV 时使用的分隔符。
- print_header(_boolean_) - 是否打印 CSV 文件提取的标头。
- bigquery_conn_id(string) - 对特定 BigQuery 钩子的引用。
- delegate_to(string) - 模拟的帐户(如果有)。为此，发出请求的服务帐户必须启用域范围委派。

```
class airflow.contrib.operators.cassandra_to_gcs.CassandraToGoogleCloudStorageOperator(cql, bucket, filename, schema_filename=None, approx_max_file_size_bytes=1900000000, cassandra_conn_id=u'cassandra_default', google_cloud_storage_conn_id=u'google_cloud_default', delegate_to=None, *args, **kwargs)
```

基类: airflow.models.BaseOperator

将数据从 Cassandra 复制到 JSON 格式的 Google 云端存储

注意：不支持数组数组。

classmethod convert_map_type（name，value）

将映射转换为包含两个字段的重复 RECORD：'key'和'value'，每个字段将转换为 BQ 中的相应数据类型。

classmethod convert_tuple_type（name，value）

将元组转换为包含 n 个字段的 RECORD，每个字段将转换为 bq 中相应的数据类型，并将命名为“field_<index>”，其中 index 由 cassandra 中定义的元组元素的顺序确定。

classmethod convert_user_type（name，value）

将用户类型转换为包含 n 个字段的 RECORD，其中 n 是属性的数量。用户类型类中的每个元素将在 BQ 中转换为其对应的数据类型。

class airflow.contrib.operators.databricks_operator.DatabricksSubmitRunOperator（json = None，spark_jar_task = None，notebook_task = None，new_cluster = None，existing_cluster_id = None，libraries = None，run_name = None，timeout_seconds = None，databricks_conn_id ='databricks_default'，polling_period_seconds = 30，databricks_retry_limit = 3，do_xcom_push = False，** kwargs）

基类： airflow.models.BaseOperator

使用 api / 2.0 / jobs / runs / submit API 端点向 Databricks 提交 Spark 作业运行。

有两种方法可以实例化此运算符。

在第一种方式，你可以把你通常用它来调用的 JSON 有效载荷 api/2.0/jobs/runs/submit 端点并将其直接传递到我们 DatabricksSubmitRunOperator 通过 json 参数。例如

```
json = {
  'new_cluster' : {
    'spark_version' : '2.1.0-db3-scala2.11',
    'num_workers' : 2
  },
  'notebook_task' : {
    'notebook_path' : '/Users/airflow@example.com/PrepareData',
  },
}
notebook_run = DatabricksSubmitRunOperator ( task_id = 'notebook_run', json = json )
```

另一种完成同样事情的方法是直接使用命名参数 DatabricksSubmitRunOperator。请注意，runs/submit 端点中的每个顶级参数都只有一个命名参数。在此方法中，您的代码如下所示：

```
new_cluster = {
```

```
  'spark_version': '2.1.0-db3-scala2.11',
  'num_workers': 2
}
notebook_task = {
  'notebook_path': '/Users/airflow@example.com/PrepareData',
}
notebook_run = DatabricksSubmitRunOperator (
    task_id = 'notebook_run',
    new_cluster = new_cluster ,
    notebook_task = notebook_task )
```

在提供 json 参数和命名参数的情况下，它们将合并在一起。如果在合并期间存在冲突，则命名参数将优先并覆盖顶级 json 键。

目前 DatabricksSubmitRunOperator 支持的命名参数是

- spark_jar_task
- notebook_task
- new_cluster
- existing_cluster_id
- libraries
- run_name
- timeout_seconds

参数：

- json(dict) -

包含 API 参数的 JSON 对象，将直接传递给 api/2.0/jobs/runs/submit 端点。其他命名参数（即 spark_jar_task，notebook_task..）到该运营商将与此 JSON 字典合并如果提供他们。如果在合并期间存在冲突，则命名参数将优先并覆盖顶级 json 键。（模板）

也可以看看有关模板的更多信息，请参阅 Jinja 模板。 https://docs.databricks.com/api/latest/jobs.html#runs-submit

- spark_jar_task(dict) -

JAR 任务的主要类和参数。请注意，实际的 JAR 在 libraries。中指定。_ 无论是 _ spark_jar_task 或 notebook_task 应符合规定。该字段将被模板化。

也可以看看 https://docs.databricks.com/api/latest/jobs.html#jobssparkjartask

- notebook_task(dict) -

笔记本任务的笔记本路径和参数。_ 无论是 _ spark_jar_task 或 notebook_task 应符合规定。该字段将被模板化。

也可以看看 https://docs.databricks.com/api/latest/jobs.html#jobsnotebooktask

- new_cluster(dict) -

将在其上运行此任务的新群集的规范。_ 无论是 _ new_cluster 或 existing_cluster_id 应符合规定。该字段将被模板化。

也可以看看 https://docs.databricks.com/api/latest/jobs.html#jobsclusterspecnewcluster

- existing_cluster_id(string) - 要运行此任务的现有集群的 ID。_ 无论是 _ new_cluster 或 existing_cluster_id 应符合规定。该字段将被模板化。

- 图书馆(_dicts 列表 _) -

这个运行的库将使用。该字段将被模板化。

也可以看看 https://docs.databricks.com/api/latest/libraries.html#managedlibrarieslibrary

- run_name(string) - 用于此任务的运行名称。默认情况下，这将设置为 Airflow task_id。这 task_id 是超类的必需参数 BaseOperator。该字段将被模板化。
- timeout_seconds(_int32_) - 此次运行的超时。默认情况下，使用值 0 表示没有超时。该字段将被模板化。
- databricks_conn_id(string) - 要使用的 Airflow 连接的名称。默认情况下，在常见情况下，这将是 databricks_default。要使用基于令牌的身份验证，请 token 在连接的额外字段中提供密钥。
- polling_period_seconds(int) - 控制我们轮询此运行结果的速率。默认情况下，操作员每 30 秒轮询一次。
- databricks_retry_limit(int) - 如果 Databricks 后端无法访问，则重试的次数。其值必须大于或等于 1。
- do_xcom_push(_boolean_) - 我们是否应该将 run_id 和 run_page_url 推送到 xcom。

```
class airflow.contrib.operators.dataflow_operator.DataFlowJavaOperator ( jar , dataflow_default_options=None,options=None,gcp_conn_id='google_cloud_default',delegate_to= None, poll_sleep = 10, job_class = None, * args, ** kwargs)
```

基类： airflow.models.BaseOperator

启动 Java Cloud DataFlow 批处理作业。操作的参数将传递给作业。

在 dag 的 default_args 中定义 dataflow_ *参数是一个很好的做法，例如项目，区域和分段位置。

```
default_args = {
    'dataflow_default_options' : {
        'project' : 'my-gcp-project' ,
```

```
        'zone' : 'europe-west1-d',
        'stagingLocation' : 'gs://my-staging-bucket/staging/'
    }
}
```

您需要使用 jar 参数将路径作为文件引用传递给数据流，jar 需要是一个自动执行的 jar（请参阅以下文档：https：//beam.apache.org/documentation/runners/dataflow/#self-执行 jar。使用 options 转嫁选项你的工作。

```
t1 = DataFlowOperation (
    task_id = 'datapflow_example',
    jar = '{{var.value.gcp_dataflow_base}}pipeline/build/libs/pipeline-example-1.0.jar',
    options = {
        'autoscalingAlgorithm' : 'BASIC',
        'maxNumWorkers' : '50',
        'start' : '{{ds}}',
        'partitionType' : 'DAY',
        'labels' : { 'foo' : 'bar' }
    },
    gcp_conn_id = 'gcp-airflow-service-account',
    dag = my - dag )
```

这两个 jar 和 options 模板化，所以你可以在其中使用变量。

```
class  airflow.contrib.operators.dataflow_operator.DataflowTemplateOperator  (  template  ,
dataflow_default_options  =  None，  parameters  =  None，  gcp_conn_id  ='google_cloud_default',
delegate_to = None，poll_sleep = 10，* args，** kwargs）
```

基类： airflow.models.BaseOperator

启动模板化云 DataFlow 批处理作业。操作的参数将传递给作业。在 dag 的 default_args 中定义 dataflow_ *参数是一个很好的做法，例如项目，区域和分段位置。

也可以看看
https://cloud.google.com/dataflow/docs/reference/rest/v1b3/LaunchTemplateParameters
https://cloud.google.com/dataflow/docs/reference/rest/v1b3/RuntimeEnvironment

```
default_args = {
    'dataflow_default_options' : {
        'project' : 'my-gcp-project'
        'zone' : 'europe-west1-d',
        'tempLocation' : 'gs://my-staging-bucket/staging/'
        }
    }
```

```
}
```

您需要将路径作为带 template 参数的文件引用传递给数据流模板。使用 parameters 来传递参数给你的工作。使用 environment 对运行环境变量传递给你的工作。

```
t1 = DataflowTemplateOperator (
    task_id = 'datapflow_example',
    template = '{{var.value.gcp_dataflow_base}}',
    parameters = {
        'inputFile' : "gs://bucket/input/my_input.txt",
        'outputFile' : "gs://bucket/output/my_output.txt"
    },
    gcp_conn_id = 'gcp-airflow-service-account',
    dag = my - dag )
```

template, dataflow_default_options 并且 parameters 是模板化的，因此您可以在其中使用变量。

```
class airflow.contrib.operators.dataflow_operator.DataFlowPythonOperator（py_file，py_options = None，dataflow_default_options = None，options = None，gcp_conn_id ='google_cloud_default'，delegate_to = None，poll_sleep = 10，* args，** kwargs）
```

基类： airflow.models.BaseOperator

```
执行（上下文）
```

执行 python 数据流作业。

```
class airflow.contrib.operators.dataproc_operator.DataprocClusterCreateOperator（cluster_name，project_id，num_workers，zone，network_uri = None，subnetwork_uri = None，internal_ip_only = None，tags = None，storage_bucket = None，init_actions_uris = None，init_action_timeout ='10m'，metadata =无，image_version =无，属性=无，master_machine_type ='n1-standard-4'，master_disk_size = 500，worker_machine_type ='n1-standard-4'，worker_disk_size = 500，num_preemptible_workers = 0，labels = None，region ='global'，gcp_conn_id ='google_cloud_default'，delegate_to = None，service_account = None，service_account_scopes = None，idle_delete_ttl = None，auto_delete_time = None，auto_delete_ttl = None，* args，** kwargs）
```

基类： airflow.models.BaseOperator

在 Google Cloud Dataproc 上创建新群集。操作员将等待创建成功或创建过程中发生错误。

参数允许配置群集。请参阅
https://cloud.google.com/dataproc/docs/reference/rest/v1/projects.regions.clusters

有关不同参数的详细说明。链接中详述的大多数配置参数都可作为此运算符的参数。

参数：

- cluster_name(string) - 要创建的 DataProc 集群的名称。（模板）
- project_id(string) - 用于创建集群的 Google 云项目的 ID。（模板）
- num_workers(int) - 旋转的工人数量
- storage_bucket(string) - 要使用的存储桶，设置为 None 允许 dataproc 为您生成自定义存储桶
- init_actions_uris(_list __[__string __]_) - 包含数据空间初始化脚本的 GCS uri 列表
- init_action_timeout(string) - init_actions_uris 中可执行脚本必须完成的时间
- 元数据(_ 字典 _) - 要添加到所有实例的键值 google 计算引擎元数据条目的字典
- image_version(string) - Dataproc 集群内的软件版本
- 属性(_ 字典 _) -性能上的配置文件设置的字典（如火花 defaults.conf），见 https://cloud.google.com/dataproc/docs/reference/rest/v1/ projects.regions.clusters #SoftwareConfig
- master_machine_type(string) - 计算要用于主节点的引擎机器类型
- master_disk_size(int) - 主节点的磁盘大小
- worker_machine_type(string) - 计算要用于工作节点的引擎计算机类型
- worker_disk_size(int) - 工作节点的磁盘大小
- num_preemptible_workers(int) - 要旋转的可抢占工作节点数
- labels(dict) - 要添加到集群的标签的字典
- zone(string) - 群集所在的区域。（模板）
- network_uri(string) - 用于机器通信的网络 uri，不能用 subnetwork_uri 指定
- subnetwork_uri(string) - 无法使用 network_uri 指定要用于机器通信的子网 uri
- internal_ip_only(bool) - 如果为 true，则群集中的所有实例将只具有内部 IP 地址。这只能为启用子网的网络启用
- tags(_list __[__string __]_) - 要添加到所有实例的 GCE 标记
- 地区 - 作为'全球'留下，可能在未来变得相关。（模板）
- gcp_conn_id(string) - 用于连接到 Google Cloud Platform 的连接 ID。
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。
- service_account(string) - dataproc 实例的服务帐户。
- service_account_scopes(_list __[__string __]_) - 要包含的服务帐户范围的 URI。
- idle_delete_ttl(int) - 群集在保持空闲状态时保持活动状态的最长持续时间。通过此阈值将导致群集被自动删除。持续时间（秒）。
- auto_delete_time(_datetime.datetime_) - 自动删除群集的时间。
- auto_delete_ttl(int) - 群集的生命周期，群集将在此持续时间结束时自动删除。持续时间（秒）。（如果设置了 auto_delete_time，则将忽略此参数）

```
class airflow.contrib.operators.dataproc_operator.DataprocClusterScaleOperator（cluster_name，project_id,region ='global',gcp_conn_id ='google_cloud_default',delegate_to = None,num_workers = 2，num_preemptible_workers = 0，graceful_decommission_timeout = None，* args，** kwargs）
```

基类： airflow.models.BaseOperator

在 Google Cloud Dataproc 上进行扩展，向上或向下扩展。操作员将等待，直到重新调整群集。

示例 ：

```
t1 = DataprocClusterScaleOperator（
task_id ='dataproc_scale', project_id ='my-project', cluster_name ='cluster-1', num_workers =
10, num_preemptible_workers = 10, graceful_decommission_timeout ='1h'dag = dag）
```

也可以看看有关扩展群集的更多详细信息，请参阅以下参考：https：//cloud.google.com/dataproc/docs/concepts/configuring-clusters/scaling-clusters

参数：

- cluster_name(string) - 要扩展的集群的名称。（模板）
- project_id(string) - 群集运行的 Google 云项目的 ID。（模板）
- region(string) - 数据通路簇的区域。（模板）
- gcp_conn_id(string) - 用于连接到 Google Cloud Platform 的连接 ID。
- num_workers(int) - 新的工人数量
- num_preemptible_workers(int) - 新的可抢占工人数量
- graceful_decommission_timeout(string) - 优雅的 YARN decomissioning 超时。最大值为 1d
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。

```
class airflow.contrib.operators.dataproc_operator.DataprocClusterDeleteOperator（cluster_name,
project_id, region ='global', gcp_conn_id ='google_cloud_default', delegate_to = None, * args,
** kwargs）
```

基类： airflow.models.BaseOperator

删除 Google Cloud Dataproc 上的群集。操作员将等待，直到群集被销毁。

参数：

- cluster_name(string) - 要创建的集群的名称。（模板）
- project_id(string) - 群集运行的 Google 云项目的 ID。（模板）
- region(string) - 保留为“全局”，将来可能会变得相关。（模板）
- gcp_conn_id(string) - 用于连接到 Google Cloud Platform 的连接 ID。
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。

```
class airflow.contrib.operators.dataproc_operator.DataProcPigOperator（query = None, query_uri
= None,variables = None,job_name ='{{task.task_id}}_{{ds_nodash}}',cluster_name ='cluster-1',
dataproc_pig_properties =无， dataproc_pig_jars =无， gcp_conn_id ='google_cloud_default',
delegate_to =无，region ='全局', * args, ** kwargs）
```

基类： airflow.models.BaseOperator

在 Cloud DataProc 群集上启动 Pig 查询作业。操作的参数将传递给集群。

在 dag 的 default_args 中定义 dataproc_ *参数是一种很好的做法，比如集群名称和 UDF。

```
default_args = {
    'cluster_name' : 'cluster-1',
    'dataproc_pig_jars' : [
        'gs://example/udf/jar/datafu/1.2.0/datafu.jar',
        'gs://example/udf/jar/gpig/1.2/gpig.jar'
    ]
}
```

您可以将 pig 脚本作为字符串或文件引用传递。使用变量传递要在群集上解析的 pig 脚本的变量，或者使用要在脚本中解析的参数作为模板参数。

示例 ：

```
t1 = DataProcPigOperator (
        task_id = 'dataproc_pig',
        query = 'a_pig_script.pig',
        variables = { 'out' : 'gs://example/output/{{ds}}' },
        dag = dag )
```

也可以看看有关工作提交的更多详细信息，请参阅以下参考： https ：//cloud.google.com/dataproc/reference/rest/v1/projects.regions.jobs

参数：

- query(string) - 对查询文件的查询或引用（pg 或 pig 扩展）。（模板）
- query_uri(string) - 云存储上的猪脚本的 uri。
- variables(dict) - 查询的命名参数的映射。（模板）
- job_name(string) - DataProc 集群中使用的作业名称。默认情况下，此名称是附加执行数据的 task_id，但可以进行模板化。该名称将始终附加一个随机数，以避免名称冲突。（模板）
- cluster_name(string) - DataProc 集群的名称。（模板）
- dataproc_pig_properties(dict) - Pig 属性的映射。非常适合放入默认参数
- dataproc_pig_jars(list) - 在云存储中配置的 jars 的 URI（例如：用于 UDF 和 lib），非常适合放入默认参数。
- gcp_conn_id(string) - 用于连接到 Google Cloud Platform 的连接 ID。
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。
- region(string) - 创建数据加载集群的指定区域。

```
class airflow.contrib.operators.dataproc_operator.DataProcHiveOperator(query=None,query_uri=None,variables=None,job_name='{{task.task_id}}_{{ds_nodash}}',cluster_name='cluster-1',dataproc_hive_properties =无， dataproc_hive_jars =无， gcp_conn_id ='google_cloud_default',
```

delegate_to =无，region ='全局'，* args，** kwargs）

基类： airflow.models.BaseOperator

在 Cloud DataProc 群集上启动 Hive 查询作业。

参数：

- query(string) - 查询或对查询文件的引用（q 扩展名）。
- query_uri(string) - 云存储上的 hive 脚本的 uri。
- variables(dict) - 查询的命名参数的映射。
- job_name(string) - DataProc 集群中使用的作业名称。默认情况下，此名称是附加执行数据的 task_id，但可以进行模板化。该名称将始终附加一个随机数，以避免名称冲突。
- cluster_name(string) - DataProc 集群的名称。
- dataproc_hive_properties(dict) - Pig 属性的映射。非常适合放入默认参数
- dataproc_hive_jars(list) - 在云存储中配置的 jars 的 URI（例如：用于 UDF 和 lib），非常适合放入默认参数。
- gcp_conn_id(string) - 用于连接到 Google Cloud Platform 的连接 ID。
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。
- region(string) - 创建数据加载集群的指定区域。

class airflow.contrib.operators.dataproc_operator.DataProcSparkSqlOperator（query = None，query_uri = None，variables = None，job_name ='{{task.task_id}} _ {{ds_nodash}}'，cluster_name ='cluster-1'， dataproc_spark_properties = 无， dataproc_spark_jars = 无， gcp_conn_id ='google_cloud_default'，delegate_to =无，region ='全局'，* args，** kwargs）

基类： airflow.models.BaseOperator

在 Cloud DataProc 集群上启动 Spark SQL 查询作业。

参数：

- query(string) - 查询或对查询文件的引用（q 扩展名）。（模板）
- query_uri(string) - 云存储上的一个 spark sql 脚本的 uri。
- variables(dict) - 查询的命名参数的映射。（模板）
- job_name(string) - DataProc 集群中使用的作业名称。默认情况下，此名称是附加执行数据的 task_id，但可以进行模板化。该名称将始终附加一个随机数，以避免名称冲突。（模板）
- cluster_name(string) - DataProc 集群的名称。（模板）
- dataproc_spark_properties(dict) - Pig 属性的映射。非常适合放入默认参数
- dataproc_spark_jars(list) - 在云存储中配置的 jars 的 URI（例如：用于 UDF 和 lib），非常适合放入默认参数。
- gcp_conn_id(string) - 用于连接到 Google Cloud Platform 的连接 ID。
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。
- region(string) - 创建数据加载集群的指定区域。

class airflow.contrib.operators.dataproc_operator.DataProcSparkOperator（main_jar = None，main_class = None，arguments = None，archives = None，files = None，job_name ='{{task.task_id}}_{{ds_nodash}}'，cluster_name ='cluster-1'，dataproc_spark_properties =无，dataproc_spark_jars =无，gcp_conn_id ='google_cloud_default'，delegate_to =无，region ='全局'，* args，** kwargs）

基类： airflow.models.BaseOperator

在 Cloud DataProc 群集上启动 Spark 作业。

参数：

- main_jar(string) - 在云存储上配置的作业 jar 的 URI。（使用 this 或 main_class，而不是两者一起）。
- main_class(string) - 作业类的名称。（使用 this 或 main_jar，而不是两者一起）。
- arguments(list) - 作业的参数。（模板）
- archives(list) - 将在工作目录中解压缩的已归档文件列表。应存储在云存储中。
- files(list) - 要复制到工作目录的文件列表
- job_name(string) - DataProc 集群中使用的作业名称。默认情况下，此名称是附加执行数据的 task_id，但可以进行模板化。该名称将始终附加一个随机数，以避免名称冲突。（模板）
- cluster_name(string) - DataProc 集群的名称。（模板）
- dataproc_spark_properties(dict) - Pig 属性的映射。非常适合放入默认参数
- dataproc_spark_jars(list) - 在云存储中配置的 jars 的 URI（例如：用于 UDF 和 lib），非常适合放入默认参数。
- gcp_conn_id(string) - 用于连接到 Google Cloud Platform 的连接 ID。
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。
- region(string) - 创建数据加载集群的指定区域。

class airflow.contrib.operators.dataproc_operator.DataProcHadoopOperator（main_jar = None，main_class = None，arguments = None，archives = None，files = None，job_name ='{{task.task_id}}_{{ds_nodash}}'，cluster_name ='cluster-1'，dataproc_hadoop_properties = 无，dataproc_hadoop_jars =无，gcp_conn_id ='google_cloud_default'，delegate_to =无，region ='全局'，* args，** kwargs）

基类： airflow.models.BaseOperator

在 Cloud DataProc 群集上启动 Hadoop 作业。

参数：

- main_jar(string) - 在云存储上配置的作业 jar 的 URI。（使用 this 或 main_class，而不是两者一起）。
- main_class(string) - 作业类的名称。（使用 this 或 main_jar，而不是两者一起）。
- arguments(list) - 作业的参数。（模板）
- archives(list) - 将在工作目录中解压缩的已归档文件列表。应存储在云存储中。

- files(list) - 要复制到工作目录的文件列表
- job_name(string) - DataProc 集群中使用的作业名称。默认情况下，此名称是附加执行数据的 task_id，但可以进行模板化。该名称将始终附加一个随机数，以避免名称冲突。（模板）
- cluster_name(string) - DataProc 集群的名称。（模板）
- dataproc_hadoop_properties(dict) - Pig 属性的映射。非常适合放入默认参数
- dataproc_hadoop_jars(list) - 在云存储中配置的 jars 的 URI（例如：用于 UDF 和 lib），非常适合放入默认参数。
- gcp_conn_id(string) - 用于连接到 Google Cloud Platform 的连接 ID。
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。
- region(string) - 创建数据加载集群的指定区域。

```
class airflow.contrib.operators.dataproc_operator.DataProcPySparkOperator（main，arguments = None，archives = None，pyfiles = None，files = None，job_name ='{{task.task_id}}_{{ds_nodash}}'，cluster_name =' cluster-1'，dataproc_pyspark_properties = None，dataproc_pyspark_jars = None，gcp_conn_id ='google_cloud_default'，delegate_to = None，region ='global'，* args，** kwargs）
```

基类： airflow.models.BaseOperator

在 Cloud DataProc 群集上启动 PySpark 作业。

参数：

- main(string) - [必需]用作驱动程序的主 Python 文件的 Hadoop 兼容文件系统（HCFS）URI。必须是.py 文件。
- arguments(list) - 作业的参数。（模板）
- archives(list) - 将在工作目录中解压缩的已归档文件列表。应存储在云存储中。
- files(list) - 要复制到工作目录的文件列表
- pyfiles(list) - 要传递给 PySpark 框架的 Python 文件列表。支持的文件类型：.py，.egg 和.zip
- job_name(string) - DataProc 集群中使用的作业名称。默认情况下，此名称是附加执行数据的 task_id，但可以进行模板化。该名称将始终附加一个随机数，以避免名称冲突。（模板）
- cluster_name(string) - DataProc 集群的名称。
- dataproc_pyspark_properties(dict) - Pig 属性的映射。非常适合放入默认参数
- dataproc_pyspark_jars(list) - 在云存储中配置的 jars 的 URI（例如：用于 UDF 和 lib），非常适合放入默认参数。
- gcp_conn_id(string) - 用于连接到 Google Cloud Platform 的连接 ID。
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。
- region(string) - 创建数据加载集群的指定区域。

```
class airflow.contrib.operators.dataproc_operator.DataprocWorkflowTemplateBaseOperator（project_id，region ='global'，gcp_conn_id ='google_cloud_default'，delegate_to = None，* args，** kwargs）
```

基类： airflow.models.BaseOperator

class airflow.contrib.operators.dataproc_operator.DataprocWorkflowTemplateInstantiateOperator（template_id，* args，** kwargs）

基类： airflow.contrib.operators.dataproc_operator.DataprocWorkflowTemplateBaseOperator

在 Google Cloud Dataproc 上实例化 WorkflowTemplate。操作员将等待 WorkflowTemplate 完成执行。

也可以看看请参阅：https://cloud.google.com/dataproc/docs/reference/rest/v1beta2/projects.regions.workflowTemplates/instantiate

参数：

- template_id(string) - 模板的 id。（模板）
- project_id(string) - 模板运行所在的 Google 云项目的 ID
- region(string) - 保留为“全局”，将来可能会变得相关
- gcp_conn_id(string) - 用于连接到 Google Cloud Platform 的连接 ID。
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。

class airflow.contrib.operators.dataproc_operator.DataprocWorkflowTemplateInstantiateInlineOperator（template，* args，** kwargs）

基类： airflow.contrib.operators.dataproc_operator.DataprocWorkflowTemplateBaseOperator

在 Google Cloud Dataproc 上实例化 WorkflowTemplate 内联。操作员将等待 WorkflowTemplate 完成执行。

也可以看看请参阅：https://cloud.google.com/dataproc/docs/reference/rest/v1beta2/projects.regions.workflowTemplates/instantiateInline

参数：

- template(_map_) - 模板内容。（模板）
- project_id(string) - 模板运行所在的 Google 云项目的 ID
- region(string) - 保留为“全局”，将来可能会变得相关
- gcp_conn_id(string) - 用于连接到 Google Cloud Platform 的连接 ID。
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。

class airflow.contrib.operators.datastore_export_operator.DatastoreExportOperator（bucket，namespace = None，datastore_conn_id ='google_cloud_default'，cloud_storage_conn_id ='google_cloud_default'，delegate_to = None，entity_filter = None，labels = None，

polling_interval_in_seconds = 10, overwrite_existing = False, xcom_push =假, * args, ** kwargs）

基类： airflow.models.BaseOperator

将实体从 Google Cloud Datastore 导出到云存储

参数：

- bucket(string) - 要备份数据的云存储桶的名称
- namespace(str) - 指定云存储桶中用于备份数据的可选命名空间路径。如果 GCS 中不存在此命名空间，则将创建该命名空间。
- datastore_conn_id(string) - 要使用的数据存储区连接 ID 的名称
- cloud_storage_conn_id(string) - 强制写入备份的云存储连接 ID 的名称
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。
- entity_filter(dict) - 导出中包含项目中哪些数据的说明，请参阅 https://cloud.google.com/datastore/docs/reference/rest/Shared.Types/EntityFilter
- labels(dict) - 客户端分配的云存储标签
- polling_interval_in_seconds(int) - 再次轮询执行状态之前等待的秒数
- overwrite_existing(bool) - 如果存储桶+命名空间不为空，则在导出之前将清空它。这样可以覆盖现有备份。
- xcom_push(bool) - 将操作名称推送到 xcom 以供参考

class airflow.contrib.operators.datastore_import_operator.DatastoreImportOperator（bucket，file，namespace = None，entity_filter = None，labels = None，datastore_conn_id ='google_cloud_default',delegate_to = None,polling_interval_in_seconds = 10,xcom_push = False，* args，** kwargs）

基类： airflow.models.BaseOperator

将实体从云存储导入 Google Cloud Datastore

参数：

- bucket(string) - 云存储中用于存储数据的容器
- file(string) - 指定云存储桶中备份元数据文件的路径。它应该具有扩展名.overall_export_metadata
- namespace(str) - 指定云存储桶中备份元数据文件的可选命名空间。
- entity_filter(dict) - 导出中包含项目中哪些数据的说明，请参阅 https://cloud.google.com/datastore/docs/reference/rest/Shared.Types/EntityFilter
- labels(dict) - 客户端分配的云存储标签
- datastore_conn_id(string) - 要使用的连接 ID 的名称
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。
- polling_interval_in_seconds(int) - 再次轮询执行状态之前等待的秒数
- xcom_push(bool) - 将操作名称推送到 xcom 以供参考

class airflow.contrib.operators.discord_webhook_operator.DiscordWebhookOperator（http_conn_id = None，webhook_endpoint = None，message =''，username = None，avatar_url = None，tts = False，proxy = None，* args，** kwargs）

基类： airflow.operators.http_operator.SimpleHttpOperator

此运算符允许您使用传入的 webhook 将消息发布到 Discord。使用默认相对 webhook 端点获取 Discord 连接 ID。可以使用 webhook_endpoint 参数（https://discordapp.com/developers/docs/resources/webhook）覆盖默认端点。

每个 Discord webhook 都可以预先配置为使用特定的用户名和 avatar_url。您可以在此运算符中覆盖这些默认值。

参数：

- http_conn_id(str) - Http 连接 ID，主机为“ https://discord.com/api/ ”，默认 webhook 端点在额外字段中，格式为{“webhook_endpoint”:“webhooks / {webhook.id} / { webhook.token}”
- webhook_endpoint(str) - 以“webhooks / {webhook.id} / {webhook.token}”的形式 Discord webhook 端点
- message(str) - 要发送到 Discord 频道的消息（最多 2000 个字符）。（模板）
- username(str) - 覆盖 webhook 的默认用户名。（模板）
- avatar_url(str) - 覆盖 webhook 的默认头像
- tts(bool) - 是一个文本到语音的消息
- proxy(str) - 用于进行 Discord webhook 调用的代理

执行（上下文）

调用 DiscordWebhookHook 发布消息

class airflow.contrib.operators.druid_operator.DruidOperator（json_index_file，druid_ingest_conn_id ='druid_ingest_default'，max_ingestion_time = None，* args，** kwargs）

基类： airflow.models.BaseOperator

允许直接向德鲁伊提交任务

参数：

- json_index_file(str) - 德鲁伊索引规范的文件路径
- druid_ingest_conn_id(str) - 接受索引作业的德鲁伊霸主的连接 ID

class airflow.contrib.operators.ecs_operator.ECSOperator(task_definition,cluster,overrides，aws_conn_id = None，region_name = None，launch_type ='EC2'，** kwargs）

基类： airflow.models.BaseOperator

在 AWS EC2 Container Service 上执行任务

参数：

- task_definition(str) - EC2 容器服务上的任务定义名称
- cluster(str) - EC2 Container Service 上的群集名称
- aws_conn_id(str) - AWS 凭证/区域名称的连接 ID。如果为 None，将使用凭证 boto3 策略（http://boto3.readthedocs.io/en/latest/guide/configuration.html）。
- region_name - 要在 AWS Hook 中使用的区域名称。覆盖连接中的 region_name（如果提供）
- launch_type - 运行任务的启动类型（'EC2'或'FARGATE'）

| 帕拉姆： | 覆盖：boto3 将接收的相同参数（模板化）：http：//boto3.readthedocs.org/en/latest/reference/services/ecs.html#ECS.Client.run_task类型：覆盖：dict 类型： launch_type：str

class airflow.contrib.operators.emr_add_steps_operator.EmrAddStepsOperator（ job_flow_id，aws_conn_id ='s3_default'，steps = None，* args，** kwargs）

基类： airflow.models.BaseOperator

向现有 EMR job_flow 添加步骤的运算符。

参数：

- job_flow_id - 要添加步骤的 JobFlow 的 ID。（模板）
- aws_conn_id(str) - 与使用的 aws 连接
- 步骤(list) - 要添加到作业流的 boto3 样式步骤。（模板）

class airflow.contrib.operators.emr_create_job_flow_operator.EmrCreateJobFlowOperator（aws_conn_id ='s3_default'，emr_conn_id ='emr_default'，job_flow_overrides = None，* args，** kwargs）

基类： airflow.models.BaseOperator

创建 EMR JobFlow，从 EMR 连接读取配置。可以传递 JobFlow 覆盖的字典，覆盖连接中的配置。

参数：

- aws_conn_id(str) - 与使用的 aws 连接
- emr_conn_id(str) - 要使用的 emr 连接
- job_flow_overrides - 用于覆盖 emr_connection extra 的 boto3 样式参数。（模板）

class airflow.contrib.operators.emr_terminate_job_flow_operator.EmrTerminateJobFlowOperator（job_flow_id, aws_conn_id ='s3_default', * args, ** kwargs）

基类： airflow.models.BaseOperator

运营商终止 EMR JobFlows。

参数：

- job_flow_id - 要终止的 JobFlow 的 id。（模板）
- aws_conn_id(str) - 与使用的 aws 连接

class airflow.contrib.operators.file_to_gcs.FileToGoogleCloudStorageOperator（src,dst,bucket,google_cloud_storage_conn_id ='google_cloud_default',mime_type ='application / octet-stream',delegate_to = None, * args, ** kwargs）

基类： airflow.models.BaseOperator

将文件上传到 Google 云端存储

参数：

- src(string) - 本地文件的路径。（模板）
- dst(string) - 指定存储桶中的目标路径。（模板）
- bucket(string) - 要上传的存储桶。（模板）
- google_cloud_storage_conn_id(string) - 要上传的 Airflow 连接 ID
- mime_type(string) - mime 类型字符串
- delegate_to(string) - 模拟的帐户（如果有）

执行（上下文）

将文件上传到 Google 云端存储

class airflow.contrib.operators.file_to_wasb.FileToWasbOperator（file_path, container_name, blob_name, wasb_conn_id ='wasb_default', load_options = None, * args, ** kwargs）

基类： airflow.models.BaseOperator

将文件上载到 Azure Blob 存储。

参数：

- file_path(str) - 要加载的文件的路径。（模板）

- container_name(str) - 容器的名称。（模板）
- blob_name(str) - blob 的名称。（模板）
- wasb_conn_id(str) - 对 wasb 连接的引用。
- load_options(dict) - WasbHook.load_file（）采用的可选关键字参数。

执行（上下文）

将文件上载到 Azure Blob 存储。

class airflow.contrib.operators.gcp_container_operator.GKEClusterCreateOperator（project_id，location，body = {}，gcp_conn_id ='google_cloud_default'，api_version ='v2'，* args，** kwargs）

基类： airflow.models.BaseOperator

class airflow.contrib.operators.gcp_container_operator.GKEClusterDeleteOperator（project_id，name，location，gcp_conn_id ='google_cloud_default'，api_version ='v2'，* args，** kwargs）

基类： airflow.models.BaseOperator

class airflow.contrib.operators.gcs_download_operator.GoogleCloudStorageDownloadOperator（bucket，object，filename = None，store_to_xcom_key = None，google_cloud_storage_conn_id ='google_cloud_default'，delegate_to = None，* args，** kwargs）

基类： airflow.models.BaseOperator

从 Google 云端存储下载文件。

参数：

- bucket(string) - 对象所在的 Google 云存储桶。（模板）
- object(string) - 要在 Google 云存储桶中下载的对象的名称。（模板）
- filename(string) - 应将文件下载到的本地文件系统（正在执行操作符的位置）上的文件路径。（模板化）如果未传递文件名，则下载的数据将不会存储在本地文件系统中。
- store_to_xcom_key(string) - 如果设置了此参数，操作员将使用此参数中设置的键将下载文件的内容推送到 XCom。如果未设置，则下载的数据不会被推送到 XCom。（模板）
- google_cloud_storage_conn_id(string) - 连接到 Google 云端存储时使用的连接 ID。
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。

class airflow.contrib.operators.gcslistoperator.GoogleCloudStorageListOperator（bucket，prefix = None，delimiter = None，google_cloud_storage_conn_id ='google_cloud_default'，delegate_to = None，* args，** kwargs）

基类： airflow.models.BaseOperator

使用名称中的给定字符串前缀和分隔符列出存储桶中的所有对象。

此运算符返回一个 python 列表，其中包含可供其使用的对象的名称
xcom 在下游任务中。

参数：

- bucket(string) - 用于查找对象的 Google 云存储桶。（模板）
- prefix(string) - 前缀字符串，用于过滤名称以此前缀开头的对象。（模板）
- delimiter(string) - 要过滤对象的分隔符。（模板化）例如，要列出 GCS 目录中的 CSV 文件，您可以使用 delimiter ='。csv'。
- google_cloud_storage_conn_id(string) - 连接到 Google 云端存储时使用的连接 ID。
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。

Example:
以下运算符将列出存储桶中文件 sales/sales-2017 夹中的所有 Avro 文件 data。

```
GCS_Files = GoogleCloudStorageListOperator (
    task_id = 'GCS_Files',
    bucket = 'data',
    prefix = 'sales/sales-2017/',
    delimiter = '.avro',
    google_cloud_storage_conn_id = google_cloud_conn_id
)
class          airflow.contrib.operators.gcs_operator.GoogleCloudStorageCreateBucketOperator
(bucket_name,storage_class ='MULTI_REGIONAL',location ='US',project_id = None,labels = None,
google_cloud_storage_conn_id ='google_cloud_default', delegate_to = None, * args, ** kwargs)
```

基类： airflow.models.BaseOperator

创建一个新存储桶。Google 云端存储使用平面命名空间，因此您无法创建名称已在使用中的存储桶。

也可以看看有关详细信息，请参阅存储桶命名指南：https://cloud.google.com/storage/docs/bucketnaming.html#requirements

参数：

- bucket_name(string) - 存储桶的名称。（模板）
- storage_class(string) -

这定义了存储桶中对象的存储方式，并确定了 SLA 和存储成本（模板化）。价值包括

  - MULTI_REGIONAL
  - REGIONAL

- STANDARD
- NEARLINE
- COLDLINE 。

如果在创建存储桶时未指定此值，则默认为 STANDARD。

- 位置(string) -

水桶的位置。（模板化）存储桶中对象的对象数据驻留在此区域内的物理存储中。默认为美国。

也可以看看 https://developers.google.com/storage/docs/bucket-locations

- project_id(string) - GCP 项目的 ID。（模板）

- labels(dict) - 用户提供的键/值对标签。
- google_cloud_storage_conn_id(string) - 连接到 Google 云端存储时使用的连接 ID。
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。

Example:
以下运算符将在区域中创建 test-bucket 具有 MULTI_REGIONAL 存储类的新存储桶 EU

```
CreateBucket = GoogleCloudStorageCreateBucketOperator (
    task_id = 'CreateNewBucket',
    bucket_name = 'test-bucket',
    storage_class = 'MULTI_REGIONAL',
    location = 'EU',
    labels = { 'env' : 'dev' , 'team' : 'airflow' },
    google_cloud_storage_conn_id = 'airflow-service-account'
)
```

```
class airflow.contrib.operators.gcs_to_bq.GoogleCloudStorageToBigQueryOperator ( bucket , source_objects, destination_project_dataset_table, schema_fields = None, schema_object = None, source_format ='CSV', compression ='NONE', create_disposition ='CREATE_IF_NEEDED', skip_leading_rows = 0, write_disposition = 'WRITE_EMPTY', field_delimiter =',', max_bad_records = 0, quote_character = None, ignore_unknown_values = False, allow_quoted_newlines = False, allow_jagged_rows = False , max_id_key = None , bigquery_conn_id ='bigquery_default', google_cloud_storage_conn_id ='google_cloud_default', delegate_to = None, schema_update_options = (), src_fmt_configs = {}, external_table = False, time_partitioning = {}, * args, ** kwargs )
```

基类： airflow.models.BaseOperator

将文件从 Google 云存储加载到 BigQuery 中。

可以用两种方法之一指定用于 BigQuery 表的模式。您可以直接传递架构字段，也可以将运营商指向 Google 云存储对象名称。Google 云存储中的对象必须是包含架构字段的 JSON 文件。

参数：

- bucket(string) - 要加载的桶。（模板）
- source_objects - 要加载的 Google 云存储 URI 列表。（模板化）如果 source_format 是'DATASTORE_BACKUP'，则列表必须只包含一个 URI。
- destination_project_dataset_table(string) - 用于加载数据的虚线（<project>。）<dataset>。<table> BigQuery 表。如果未包含<project>，则项目将是连接 json 中定义的项目。（模板）
- schema_fields(list) - 如果设置，则此处定义的架构字段列表：https://cloud.google.com/bigquery/docs/reference/v2/jobs#configuration.load 当 source_format 为'DATASTORE_BACKUP'时，不应设置。
- schema_object - 如果设置，则指向包含表的架构的.json 文件的 GCS 对象路径。（模板）
- schema_object - 字符串
- source_format(string) - 要导出的文件格式。
- compression(string) - [可选]数据源的压缩类型。可能的值包括 GZIP 和 NONE。默认值为 NONE。Google Cloud Bigtable，Google Cloud Datastore 备份和 Avro 格式会忽略此设置。
- create_disposition(string) - 如果表不存在，则创建处置。
- skip_leading_rows(int) - 从 CSV 加载时要跳过的行数。
- write_disposition(string) - 表已存在时的写处置。
- field_delimiter(string) - 从 CSV 加载时使用的分隔符。
- max_bad_records(int) - BigQuery 在运行作业时可以忽略的最大错误记录数。
- quote_character(string) - 用于引用 CSV 文件中数据部分的值。
- ignore_unknown_values(bool) - [可选]指示 BigQuery 是否应允许表模式中未表示的额外值。如果为 true，则忽略额外值。如果为 false，则将具有额外列的记录视为错误记录，如果错误记录太多，则在作业结果中返回无效错误。
- allow_quoted_newlines(_boolean_) - 是否允许引用的换行符（true）或不允许（false）。
- allow_jagged_rows(bool) - 接受缺少尾随可选列的行。缺失值被视为空值。如果为 false，则缺少尾随列的记录将被视为错误记录，如果错误记录太多，则会在作业结果中返回无效错误。仅适用于 CSV，忽略其他格式。
- max_id_key(string) - 如果设置，则是 BigQuery 表中要加载的列的名称。在加载发生后，Thsi 将用于从 BigQuery 中选择 MAX 值。结果将由 execute（）命令返回，该命令又存储在 XCom 中供将来的操作员使用。这对增量加载很有帮助 - 在将来的执行过程中，您可以从最大 ID 中获取。
- bigquery_conn_id(string) - 对特定 BigQuery 挂钩的引用。
- google_cloud_storage_conn_id(string) - 对特定 Google 云存储挂钩的引用。
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。
- schema_update_options(list) - 允许更新目标表的模式作为加载作业的副作用。
- src_fmt_configs(dict) - 配置特定于源格式的可选字段
- external_table(bool) - 用于指定目标表是否应为 BigQuery 外部表的标志。默认值为 False。
- time_partitioning(dict) - 配置可选的时间分区字段，即按 API 规范按字段，类型和到期分区。请注意，“field”在 dataset.table $ partition 的并发中不可用。

```
class    airflow.contrib.operators.gcs_to_gcs.GoogleCloudStorageToGoogleCloudStorageOperator
（source_bucket,source_object,destination_bucket=None,destination_object=None,move_object
= False, google_cloud_storage_conn_id ='google_cloud_default', delegate_to = None, * args, **
```

kwargs）

基类： airflow.models.BaseOperator

将对象从存储桶复制到另一个存储桶，并在需要时重命名。

参数：

- source_bucket(string) - 对象所在的源 Google 云存储桶。（模板）
- source_object(string) -

要在 Google 云存储分区中复制的对象的源名称。（模板化）如果在此参数中使用通配符：

> 您只能在存储桶中使用一个通配符作为对象（文件名）。通配符可以出现在对象名称内或对象名称的末尾。不支持在存储桶名称中附加通配符。

- destination_bucket - 目标 Google 云端存储分区

对象应该在哪里。（模板化）：type destination_bucket：string：param destination_object：对象的目标名称

目标 Google 云存储桶。（模板化）如果在 source_object 参数中提供了通配符，则这是将添加到最终目标对象路径的前缀。请注意，将删除通配符之前的源路径部分；如果需要保留，则应将其附加到 destination_object。例如，使用 prefix foo/*和 destination_object'blah /``，文件 foo/baz 将被复制到 blah/baz；保留前缀写入 destination_object，例如 blah/foo，在这种情况下，复制的文件将被命名 blah/foo/baz`。

参数：move_object - 当移动对象为 True 时，移动对象

复制到新位置。

这相当于 mv 命令而不是 cp 命令。

参数：

- google_cloud_storage_conn_id(string) - 连接到 Google 云端存储时使用的连接 ID。
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。

Examples:
下面的操作将命名一个文件复制 sales/sales-2017/january.avro 在 data 桶的文件和名为斗 copied_sales/2017/january-backup.avro in the ``data_backup`

```
copy_single_file = GoogleCloudStorageToGoogleCloudStorageOperator (
    task_id = 'copy_single_file',
```

```
    source_bucket = 'data',
    source_object = 'sales/sales-2017/january.avro',
    destination_bucket = 'data_backup',
    destination_object = 'copied_sales/2017/january-backup.avro',
    google_cloud_storage_conn_id = google_cloud_conn_id
)
```

以下运算符会将文件 sales/sales-2017 夹中的所有 Avro 文件（即名称以该前缀开头）复制到存储 data 桶中的 copied_sales/2017 文件夹中 data_backup。

```
copy_files = GoogleCloudStorageToGoogleCloudStorageOperator (
    task_id = 'copy_files',
    source_bucket = 'data',
    source_object = 'sales/sales-2017/*.avro',
    destination_bucket = 'data_backup',
    destination_object = 'copied_sales/2017/',
    google_cloud_storage_conn_id = google_cloud_conn_id
)
```

以下运算符会将文件 sales/sales-2017 夹中的所有 Avro 文件（即名称以该前缀开头）移动到 data 存储桶中的同一文件夹 data_backup，删除过程中的原始文件。

```
move_files = GoogleCloudStorageToGoogleCloudStorageOperator (
    task_id = 'move_files',
    source_bucket = 'data',
    source_object = 'sales/sales-2017/*.avro',
    destination_bucket = 'data_backup',
    move_object = True ,
    google_cloud_storage_conn_id = google_cloud_conn_id
)
```

```
class airflow.contrib.operators.gcs_to_s3.GoogleCloudStorageToS3Operator(bucket, prefix = None, delimiter = None, google_cloud_storage_conn_id ='google_cloud_storage_default', delegate_to = None, dest_aws_conn_id = None, dest_s3_key = None, replace = False, * args, ** kwargs)
```

基类： airflow.contrib.operators.gcslistoperator.GoogleCloudStorageListOperator

将 Google 云端存储分区与 S3 分区同步。

参数：

- bucket(string) - 用于查找对象的 Google Cloud Storage 存储桶。（模板）
- prefix(string) - 前缀字符串，用于过滤名称以此前缀开头的对象。（模板）
- delimiter(string) - 要过滤对象的分隔符。（模板化）例如，要列出 GCS 目录中的 CSV 文件，您

可以使用 delimiter ='。csv'。

- google_cloud_storage_conn_id(string) - 连接到 Google 云端存储时使用的连接 ID。
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。
- dest_aws_conn_id(str) - 目标 S3 连接
- dest_s3_key(str) - 用于存储文件的基本 S3 密钥。（模板）

class airflow.contrib.operators.hipchat_operator.HipChatAPIOperator（token，base_url ='https：//api.hipchat.com/v2'，* args，** kwargs）

基类： airflow.models.BaseOperator

基础 HipChat 运算符。所有派生的 HipChat 运营商都参考了 HipChat 的官方 REST API 文档，网址为 https://www.hipchat.com/docs/apiv2 。在使用任何 HipChat API 运算符之前，您需要在 https://www.hipchat.com/docs/apiv2/auth 获取身份验证令牌。在未来，其他 HipChat 运算符也将从此类派生。

参数：

- token(str) - HipChat REST API 身份验证令牌
- base_url(str) - HipChat REST API 基本 URL。
- prepare_request（）
- 由 execute 函数使用。设置 HipChat 的 REST API 调用的请求方法，URL 和正文。在子类中重写。每个 HipChatAPI 子运算符负责具有设置 self.method， self.url 和 self.body 的 prepare_request 方法调用。

class airflow.contrib.operators.hipchat_operator.HipChatAPISendRoomNotificationOperator（room_id，message，* args，** kwargs）

基类： airflow.contrib.operators.hipchat_operator.HipChatAPIOperator

将通知发送到特定的 HipChat 会议室。更多信息： https：//www.hipchat.com/docs/apiv2/method/send_room_notification

参数：

- room_id(str) - 在 HipChat 上发送通知的房间。（模板）
- message(str) - 邮件正文。（模板）
- frm(str) - 除发件人姓名外还要显示的标签
- message_format(str) - 如何呈现通知：html 或文本
- color(str) - msg 的背景颜色：黄色，绿色，红色，紫色，灰色或随机
- attach_to(str) - 将此通知附加到的消息 ID
- notify(bool) - 此消息是否应触发用户通知
- card(dict) - HipChat 定义的卡片对象
- prepare_request（）

- 由 execute 函数使用。设置 HipChat 的 REST API 调用的请求方法，URL 和正文。在子类中重写。每个 HipChatAPI 子运算符负责具有设置 self.method，self.url 和 self.body 的 prepare_request 方法调用。

class airflow.contrib.operators.hive_to_dynamodb.HiveToDynamoDBTransferOperator（ sql ，table_name，table_keys，pre_process = None，pre_process_args = None，pre_process_kwargs = None，region_name = None，schema ='default'，hiveserver2_conn_id ='hiveserver2_default'，aws_conn_id ='aws_default' ，* args，** kwargs）

基类： airflow.models.BaseOperator

将数据从 Hive 移动到 DynamoDB，请注意，现在数据在被推送到 DynamoDB 之前已加载到内存中，因此该运算符应该用于少量数据。

参数：

- sql(str) - 针对 hive 数据库执行的 SQL 查询。（模板）
- table_name(str) - 目标 DynamoDB 表
- table_keys(list) - 分区键和排序键
- pre_process(function) - 实现源数据的预处理
- pre_process_args(list) - pre_process 函数参数列表
- pre_process_kwargs(dict) - pre_process 函数参数的字典
- region_name(str) - aws 区域名称（例如：us-east-1）
- schema(str) - hive 数据库模式
- hiveserver2_conn_id(str) - 源配置单元连接
- aws_conn_id(str) - aws 连接

class airflow.contrib.operators.jenkins_job_trigger_operator.JenkinsJobTriggerOperator（jenkins_connection_id，job_name，parameters =''，sleep_time = 10，max_try_before_job_appears = 10，* args，** kwargs）

基类： airflow.models.BaseOperator

触发 Jenkins 作业并监视它的执行情况。此运算符依赖于 python-jenkins 库，版本> = 0.4.15 与 jenkins 服务器通信。您还需要在连接屏幕中配置 Jenkins 连接。：param jenkins_connection_id：用于此作业的 jenkins 连接：type jenkins_connection_id：string：param job_name：要触发的作业的名称：type job_name：string：param parameters：要提供给 jenkins 的参数块。（模板化）：类型参数：string：param sleep_time：操作员在作业的每个状态请求之间休眠多长时间（min 1，默认值为 10）：type sleep_time：int：param max_try_before_job_appears：要发出的最大请求数

等待作业出现在 jenkins 服务器上（默认为 10）

build_job（jenkins_server）

此函数对 Jenkins 进行 API 调用以触发'job_name'的构建。它返回了一个带有 2 个键的字典：body 和 headers。headers 还包含一个类似 dict 的对象，可以查询该对象以获取队列中轮询的位置。：param jenkins_server：应该触发作业的 jenkins 服务器：return：包含响应正文（密钥正文）和标题出现的标题（标题）

poll_job_in_queue（location，jenkins_server）

此方法轮询 jenkins 队列，直到执行作业。当我们通过 API 调用触发作业时，首先将作业放入队列而不分配内部版本号。因此，我们必须等待作业退出队列才能知道其内部版本号。为此，我们必须将/ api / json（或/ api / xml）添加到 build_job 调用返回的位置并轮询该文件。当 json 中出现'可执行'块时，表示作业执行已开始，字段'number'则包含内部版本号。 ：param location：轮询的位置，在 build_job 调用的标头中返回：param jenkins_server：要轮询的 jenkins 服务器：return：与触发的作业对应的 build_number

class airflow.contrib.operators.jira_operator.JiraOperator（jira_conn_id ='jira_default'，jira_method = None，jira_method_args = None，result_processor = None，get_jira_resource_method = None，* args，** kwargs）

基类： airflow.models.BaseOperator

JiraOperator 在 Jira 问题跟踪系统上进行交互并执行操作。此运算符旨在使用 Jira Python SDK：http：//jira.readthedocs.io

参数：

- jira_conn_id(str) - 对预定义的 Jira 连接的引用
- jira_method(str) - 要调用的 Jira Python SDK 中的方法名称
- jira_method_args(dict) - jira_method 所需的方法参数。（模板）
- result_processor(function) - 用于进一步处理 Jira 响应的函数
- get_jira_resource_method(function) - 用于获取将在其上执行提供的 jira_method 的 jira 资源的函数或运算符

class airflow.contrib.operators.kubernetes_pod_operator.KubernetesPodOperator（namespace，image，name，cmds = None，arguments = None，volume_mounts = None，volumes = None，env_vars = None，secrets = None，in_cluster = False，cluster_context = None，labels = None，startup_timeout_seconds = 120，get_logs = True，image_pull_policy ='IfNotPresent'，annotations = None，resources = None，affinity = None，config_file = None，xcom_push = False，* args，** kwargs）

基类： airflow.models.BaseOperator

在 Kubernetes Pod 中执行任务

参数：

- image(str) - 您希望启动的 Docker 镜像。默认为 dockerhub.io，但完全限定的 URLS 将指向自定义存储库
- cmds(list[str]) - 容器的入口点。（模板化）如果未提供，则使用泊坞窗图像的入口点。
- arguments(list[str]) - 入口点的参数。（模板化）如果未提供，则使用泊坞窗图像的 CMD。
- volume_mounts(_VolumeMount 列表 _) - 已启动 pod 的 volumeMounts
- 卷(**卷**list) - 已启动 pod 的卷。包括 ConfigMaps 和 PersistentVolumes
- labels(dict) - 要应用于 Pod 的标签
- startup_timeout_seconds(int) - 启动 pod 的超时时间（秒）
- name(str) - 要运行的任务的名称，将用于生成 pod id
- env_vars(dict) - 在容器中初始化的环境变量。（模板）
- 秘密(**秘密**list) - Kubernetes 秘密注入容器中，它们可以作为环境变量或卷中的文件公开。
- in_cluster(bool) - 使用 in_cluster 配置运行 kubernetes 客户端
- cluster_context(string) - 指向 kubernetes 集群的上下文。当 in_cluster 为 True 时忽略。如果为 None，则使用 current-context。
- get_logs(bool) - 获取容器的标准输出作为任务的日志
- affinity(dict) - 包含一组关联性调度规则的 dict
- config_file(str) - Kubernetes 配置文件的路径
- xcom_push(bool) - 如果 xcom_push 为 True，容器中的文件/airflow/xcom/return.json 的内容也将在容器完成时被推送到 XCom。

| 帕拉姆： | namespace：在 kubernetes 中运行的命名空间类型： namespace：str

```
class airflow.contrib.operators.mlengine_operator.MLEngineBatchPredictionOperator（project_id，job_id，region，data_format，input_paths，output_path，model_name = None，version_name = None，uri = None，max_worker_count = None，runtime_version = None，gcp_conn_id ='google_cloud_default'，delegate_to =无，* args，** kwargs）
```

基类： airflow.models.BaseOperator

启动 Google Cloud ML Engine 预测作业。

注意：对于模型原点，用户应该考虑以下三个选项中的一个：1。仅填充“uri”字段，该字段应该是指向 tensorflow savedModel 目录的 GCS 位置。2.仅填充'model_name'字段，该字段引用现有模型，并将使用模型的默认版本。3.填充“model_name”和“version_name”字段，这些字段指特定模型的特定版本。

在选项 2 和 3 中，模型和版本名称都应包含最小标识符。例如，打电话

```
MLEngineBatchPredictionOperator（
    ...，
    model_name = 'my_model'，
    version_name = 'my_version'，
    ...）
```

如果所需的型号版本是“projects / my_project / models / my_model / versions / my_version”。

有关参数的更多文档,请参阅https://cloud.google.com/ml-engine/reference/rest/v1/projects.jobs。

参数:

- project_id(string) - 提交预测作业的 Google Cloud 项目名称。(模板)
- job_id(string) - Google Cloud ML Engine 上预测作业的唯一 ID。(模板)
- data_format(string) - 输入数据的格式。如果未提供或者不是[“TEXT”,“TF_RECORD”,“TF_RECORD_GZIP”]之一,它将默认为“DATA_FORMAT_UNSPECIFIED”。
- input_paths(_ 字符串列表 _) - 批量预测的输入数据的 GCS 路径列表。接受通配符运算符*,但仅限于结尾处。(模板)
- output_path(string) - 写入预测结果的 GCS 路径。(模板)
- region(string) - 用于运行预测作业的 Google Compute Engine 区域。(模板化)
- model_name(string) - 用于预测的 Google Cloud ML Engine 模型。如果未提供 version_name,则将使用此模型的默认版本。如果提供了 version_name,则不应为 None。如果提供 uri,则应为 None。(模板)
- version_name(string) - 用于预测的 Google Cloud ML Engine 模型版本。如果提供 uri,则应为 None。(模板)
- uri(string) - 用于预测的已保存模型的 GCS 路径。如果提供了 model_name,则应为 None。它应该是指向张量流 SavedModel 的 GCS 路径。(模板)
- max_worker_count(int) - 用于并行处理的最大 worker 数。如果未指定,则默认为 10。
- runtime_version(string) - 用于批量预测的 Google Cloud ML Engine 运行时版本。
- gcp_conn_id(string) - 用于连接到 Google Cloud Platform 的连接 ID。
- delegate_to(string) - 模拟的帐户(如果有)。为此,发出请求的服务帐户必须启用 doamin 范围的委派。
- Raises:
- ValueError : 如果无法确定唯一的模型/版本来源。

```
class airflow.contrib.operators.mlengine_operator.MLEngineModelOperator(project_id, model, operation='create', gcp_conn_id='google_cloud_default', delegate_to=None, *args, **kwargs)
```

基类: airflow.models.BaseOperator

管理 Google Cloud ML Engine 模型的运营商。

参数:

- project_id(string) - MLEngine 模型所属的 Google Cloud 项目名称。(模板)
- 型号(_ 字典 _) -
  包含有关模型信息的字典。如果操作是 create,则 model 参数应包含有关此模型的所有信息,例如 name。
  如果操作是 get,则 model 参数应包含模型的名称。
- 操作 -
  执行的操作。可用的操作是:

- create：创建 model 参数提供的新模型。
- get：获取在模型中指定名称的特定模型。

- gcp_conn_id(string) - 获取连接信息时使用的连接 ID。
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。

class airflow.contrib.operators.mlengine_operator.MLEngineVersionOperator（project_id，model_name，version_name = None，version = None，operation ='create'，gcp_conn_id ='google_cloud_default'，delegate_to = None，* args，** kwargs）

基类： airflow.models.BaseOperator

管理 Google Cloud ML Engine 版本的运营商。

参数：

- project_id(string) - MLEngine 模型所属的 Google Cloud 项目名称。
- model_name(string) - 版本所属的 Google Cloud ML Engine 模型的名称。（模板）
- version_name(string) - 用于正在操作的版本的名称。如果没有人及版本的说法是没有或不具备的值名称键，那么这将是有效载荷中用于填充名称键。 （模板）
- version(dict) - 包含版本信息的字典。如果操作是 create，则 version 应包含有关此版本的所有信息，例如 name 和 deploymentUrl。如果操作是 get 或 delete，则 version 参数应包含版本的名称。如果是 None，则唯一可能的操作是 list。（模板）
- 操作(string) -
  执行的操作。可用的操作是：
  - create：在 model_name 指定的模型中创建新版本，在这种情况下，version 参数应包含创建该版本的所有信息（例如 name，deploymentUrl）。
  - get：获取 model_name 指定的模型中特定版本的完整信息。应在 version 参数中指定版本的名称。
  - list：列出 model_name 指定的模型的所有可用版本。
  - delete：从 model_name 指定的模型中删除 version 参数中指定的版本。应在 version 参数中指定版本的名称。
- gcp_conn_id(string) - 获取连接信息时使用的连接 ID。
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。

class airflow.contrib.operators.mlengine_operator.MLEngineTrainingOperator（project_id，job_id，package_uris，training_python_module，training_args，region，scale_tier = None，runtime_version = None，python_version = None，job_dir = None，gcp_conn_id ='google_cloud_default'，delegate_to = None，mode ='生产'，* args，** kwargs）

基类： airflow.models.BaseOperator

启动 MLEngine 培训工作的操作员。

参数：

- project_id(string) - 应在其中运行 MLEngine 培训作业的 Google Cloud 项目名称（模板化）。
- job_id(string) - 提交的 Google MLEngine 培训作业的唯一模板化 ID。（模板）
- package_uris(string) - MLEngine 培训作业的包位置列表，其中应包括主要培训计划+任何其他依赖项。（模板）
- training_python_module(string) - 安装'package_uris'软件包后，在 MLEngine 培训作业中运行的 Python 模块名称。（模板）
- training_args(string) - 传递给 MLEngine 训练程序的模板化命令行参数列表。（模板）
- region(string) - 用于运行 MLEngine 培训作业的 Google Compute Engine 区域（模板化）。
- scale_tier(string) - MLEngine 培训作业的资源层。（模板）
- runtime_version(string) - 用于培训的 Google Cloud ML 运行时版本。（模板）
- python_version(string) - 训练中使用的 Python 版本。（模板）
- job_dir(string) - 用于存储培训输出和培训所需的其他数据的 Google 云端存储路径。（模板）
- gcp_conn_id(string) - 获取连接信息时使用的连接 ID。
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。
- mode(string) - 可以是'DRY_RUN'/'CLOUD'之一。在“DRY_RUN”模式下，不会启动真正的培训作业，但会打印出 MLEngine 培训作业请求。在“CLOUD”模式下，将发出真正的 MLEngine 培训作业创建请求。

class airflow.contrib.operators.mongo_to_s3.MongoToS3Operator（mongo_conn_id，s3_conn_id，mongo_collection，mongo_query，s3_bucket，s3_key，mongo_db = None，* args，** kwargs）

基类： airflow.models.BaseOperator

Mongo - > S3

更具体的 baseOperator 意味着将数据从 mongo 通过 pymongo 移动到 s3 通过 boto

需要注意的事项

.execute（）编写依赖于.transform（）。transsnsform（）旨在由子类扩展，以执行特定于运算符需求的转换

执行（上下文）

由 task_instance 在运行时执行

变换（文档）

处理 pyMongo 游标并返回每个元素都在的 iterable

一个 JSON 可序列化的字典

Base transform（）假设不需要处理，即。docs 是文档的 pyMongo 游标，只需要传递游标

重写此方法以进行自定义转换

class airflow.contrib.operators.mysql_to_gcs.MySqlToGoogleCloudStorageOperator（sql，bucket，filename，schema_filename = None，approx_max_file_size_bytes = 1900000000，mysql_conn_id ='mysql_default'，google_cloud_storage_conn_id ='google_cloud_default'，schema = None，delegate_to = None，* args，** kwargs）

基类： airflow.models.BaseOperator

将数据从 MySQL 复制到 JSON 格式的 Google 云存储。

classmethod type_map（mysql_type）

从 MySQL 字段映射到 BigQuery 字段的辅助函数。在设置 schema_filename 时使用。

class airflow.contrib.operators.postgres_to_gcs_operator.PostgresToGoogleCloudStorageOperator（sql，bucket，filename，schema_filename = None，approx_max_file_size_bytes = 1900000000，postgres_conn_id ='postgres_default'，google_cloud_storage_conn_id ='google_cloud_default'，delegate_to = None，parameters = None，* args，** kwargs）

基类： airflow.models.BaseOperator

将数据从 Postgres 复制到 JSON 格式的 Google 云端存储。

classmethod convert_types（value）

从 Postgres 获取值，并将其转换为对 JSON / Google Cloud Storage / BigQuery 安全的值。日期转换为 UTC 秒。小数转换为浮点数。时间转换为秒。

classmethod type_map（postgres_type）

从 Postgres 字段映射到 BigQuery 字段的 Helper 函数。在设置 schema_filename 时使用。

class airflow.contrib.operators.pubsub_operator.PubSubTopicCreateOperator（project，topic，fail_if_exists = False，gcp_conn_id ='google_cloud_default'，delegate_to = None，* args，** kwargs）

基类： airflow.models.BaseOperator

创建 PubSub 主题。

默认情况下，如果主题已存在，则此运算符不会导致 DAG 失败。

```
with DAG ( 'successful DAG' ) as dag :
    (
        dag
        >> PubSubTopicCreateOperator ( project = 'my-project' ,
                                       topic = 'my_new_topic' )
        >> PubSubTopicCreateOperator ( project = 'my-project' ,
                                       topic = 'my_new_topic' )
    )
```

如果主题已存在，则可以将操作员配置为失败。

```
with DAG ( 'failing DAG' ) as dag :
    (
        dag
        >> PubSubTopicCreateOperator ( project = 'my-project' ,
                                       topic = 'my_new_topic' )
        >> PubSubTopicCreateOperator ( project = 'my-project' ,
                                       topic = 'my_new_topic' ,
                                       fail_if_exists = True )
    )
```

这两个 project 和 topic 模板化，所以你可以在其中使用变量。

```
class airflow.contrib.operators.pubsub_operator.PubSubTopicDeleteOperator（project, topic,
fail_if_not_exists = False, gcp_conn_id ='google_cloud_default', delegate_to = None, * args,
** kwargs）
```

基类： airflow.models.BaseOperator

删除 PubSub 主题。

默认情况下，如果主题不存在，则此运算符不会导致 DAG 失败。

```
with DAG ( 'successful DAG' ) as dag :
    (
        dag
        >> PubSubTopicDeleteOperator ( project = 'my-project' ,
                                       topic = 'non_existing_topic' )
    )
```

如果主题不存在，则可以将运算符配置为失败。

```
with DAG ( 'failing DAG' ) as dag :
    (
```

```
        dag
        >> PubSubTopicCreateOperator ( project = 'my-project' ,
                                       topic = 'non_existing_topic' ,
                                       fail_if_not_exists = True )
    )
```

这两个 project 和 topic 模板化，所以你可以在其中使用变量。

```
class              airflow.contrib.operators.pubsub_operator.PubSubSubscriptionCreateOperator
（topic_project, topic, subscription = None, subscription_project = None, ack_deadline_secs =
10, fail_if_exists = False, gcp_conn_id ='google_cloud_default', delegate_to = None, * args,
** kwargs）
```

基类： airflow.models.BaseOperator

创建 PubSub 订阅。

默认情况下，将在中创建订阅 topic_project。如果 subscription_project 已指定且 GCP 凭据允许，则可以在与其主题不同的项目中创建订阅。

默认情况下，如果订阅已存在，则此运算符不会导致 DAG 失败。但是，该主题必须存在于项目中。

```
 with DAG ( 'successful DAG' ) as dag :
    (
        dag
        >> PubSubSubscriptionCreateOperator (
            topic_project = 'my-project' , topic = 'my-topic' ,
            subscription = 'my-subscription' )
        >> PubSubSubscriptionCreateOperator (
            topic_project = 'my-project' , topic = 'my-topic' ,
            subscription = 'my-subscription' )
    )
```

如果订阅已存在，则可以将运算符配置为失败。

```
 with DAG ( 'failing DAG' ) as dag :
    (
        dag
        >> PubSubSubscriptionCreateOperator (
            topic_project = 'my-project' , topic = 'my-topic' ,
            subscription = 'my-subscription' )
        >> PubSubSubscriptionCreateOperator (
            topic_project = 'my-project' , topic = 'my-topic' ,
            subscription = 'my-subscription' , fail_if_exists = True )
    )
```

最后，不需要订阅。如果未通过，则运营商将为订阅的名称生成通用唯一标识符。

```
with DAG ( 'DAG' ) as dag :
    (
        dag >> PubSubSubscriptionCreateOperator (
            topic_project = 'my-project' , topic = 'my-topic' )
    )
```

topic_project,, topic 和 subscription，并且 subscription 是模板化的，因此您可以在其中使用变量。

```
class airflow.contrib.operators.pubsub_operator.PubSubSubscriptionDeleteOperator （project，
subscription,fail_if_not_exists = False,gcp_conn_id ='google_cloud_default',delegate_to = None,
* args，** kwargs）
```

基类： airflow.models.BaseOperator

删除 PubSub 订阅。

默认情况下，如果订阅不存在，则此运算符不会导致 DAG 失败。

```
with DAG ( 'successful DAG' ) as dag :
    (
        dag
        >> PubSubSubscriptionDeleteOperator ( project = 'my-project' ,
                                              subscription = 'non-existing' )
    )
```

如果订阅已存在，则可以将运算符配置为失败。

```
with DAG ( 'failing DAG' ) as dag :
    (
        dag
        >> PubSubSubscriptionDeleteOperator (
             project = 'my-project' , subscription = 'non-existing' ,
             fail_if_not_exists = True )
    )
```

project，并且 subscription 是模板化的，因此您可以在其中使用变量。

```
class airflow.contrib.operators.pubsub_operator.PubSubPublishOperator(project,topic,messages,
gcp_conn_id ='google_cloud_default', delegate_to = None，* args，** kwargs）
```

基类： airflow.models.BaseOperator

将消息发布到 PubSub 主题。

每个任务都将所有提供的消息发布到单个 GCP 项目中的同一主题。如果主题不存在，则此任务将失败。

```
  from base64 import b64encode as b64e

  m1 = {'data': b64e('Hello, World!'),
        'attributes': {'type': 'greeting'}
       }
  m2 = {'data': b64e('Knock, knock')}
  m3 = {'attributes': {'foo': ''}}

  t1 = PubSubPublishOperator(
       project='my-project',topic='my_topic',
       messages=[m1, m2, m3],
       create_topic=True,
       dag=dag)

``project`` , ``topic``, and ``messages`` are templated so you can use
```

其中的变量。

```
class  airflow.contrib.operators.qubole_check_operator.QuboleCheckOperator （ qubole_conn_id ='qubole_default'，* args，** kwargs）
```

基 类 :airflow.operators.check_operator.CheckOperator ， airflow.contrib.operators.qubole_operator.QuboleOperator

对 Qubole 命令执行检查。QuboleCheckOperator 期望一个将在 QDS 上执行的命令。默认情况下，使用 python bool 强制计算此 Qubole Commmand 结果第一行的每个值。如果返回任何值 False，则检查失败并输出错误。

请注意，Python bool 强制转换如下 False：

- False
- 0
- 空字符串（""）
- 空列表（[]）
- 空字典或集（{}）

给定一个查询，它只会在计数时失败。您可以制作更复杂的查询，例如，可以检查表与上游源表的行数相同，或者今天的分区计数大于昨天的分区，或者一组指标是否更少 7 天平均值超过 3 个标准差。SELECT COUNT(*) FROM foo``== 0

此运算符可用作管道中的数据质量检查，并且根据您在 DAG 中的位置，您可以选择停止关键路径，防止发布可疑数据，或者侧面接收电子邮件警报阻止 DAG 的进展。

参数：qubole_conn_id(str) - 由 qds auth_token 组成的连接 ID kwargs：

可以从 QuboleOperator 文档中引用特定于 Qubole 命令的参数。

<colgroup><col class="field-name"><col class="field-body"></colgroup> | results_parser_callable： | | --- | | | 这是一个可选参数，用于扩展将 Qubole 命令的结果解析给用户的灵活性。这是一个 python 可调用的，它可以保存逻辑来解析 Qubole 命令返回的行列表。默认情况下，仅第一行的值用于执行检查。此可调用对象应返回必须在其上执行检查的记录列表。 |

注意:与 QuboleOperator 和 CheckOperator 的模板字段相同的所有字段都是模板支持的。

class airflow.contrib.operators.qubole_check_operator.QuboleValueCheckOperator（pass_value，tolerance = None，qubole_conn_id ='qubole_default'，* args，** kwargs）

基 类 ： airflow.operators.check_operator.ValueCheckOperator ， airflow.contrib.operators.qubole_operator.QuboleOperator

使用 Qubole 命令执行简单的值检查。默认情况下，此 Qubole 命令的第一行上的每个值都与预定义的值进行比较。如果命令的输出不在预期值的允许限度内，则检查失败并输出错误。

参数：

- qubole_conn_id(str) - 由 qds auth_token 组成的连接 ID
- pass_value(_str / int / float_) - 查询结果的预期值。
- tolerance(_int / float_) - 定义允许的 pass_value 范围，例如，如果 tolerance 为 2，则 Qubole 命令输出可以是-2 * pass_value 和 2 * pass_value 之间的任何值，而不会导致操作员错误输出。

kwargs：

可以从 QuboleOperator 文档中引用特定于 Qubole 命令的参数。

<colgroup><col class="field-name"><col class="field-body"></colgroup> | results_parser_callable： | | --- | | | 这是一个可选参数，用于扩展将 Qubole 命令的结果解析给用户的灵活性。这是一个 python 可调用的，它可以保存逻辑来解析 Qubole 命令返回的行列表。默认情况下，仅第一行的值用于执行检查。此可调用对象应返回必须在其上执行检查的记录列表。 |

注意:与 QuboleOperator 和 ValueCheckOperator 的模板字段相同的所有字段都是模板支持的。

class airflow.contrib.operators.qubole_operator.QuboleOperator（qubole_conn_id ='qubole_default'，* args，** kwargs）

基类： airflow.models.BaseOperator

在 QDS 上执行任务（命令）(https://qubole.com)。

参数：qubole_conn_id(str) - 由 qds auth_token 组成的连接 ID

kwargs：

| COMMAND_TYPE： | 要执行的命令类型，例如 hivecmd，shellcmd，hadoopcmd| 标签： | 要使用该命令分配的标记数组| cluster_label： | 将在其上执行命令的集群标签| 名称： | 要命令的名称| 通知： | 是否在命令完成时发送电子邮件（默认为 False） 特定于命令类型的参数

hivecmd：

| 查询： | 内联查询语句| script_location： ||---||| 包含查询语句的 s3 位置 ||SAMPLE_SIZE： | 要运行查询的样本大小（以字节为单位）| 宏： | 在查询中使用的宏值

prestocmd：

| 查询： | 内联查询语句| script_location： ||---||| 包含查询语句的 s3 位置 || 宏： | 在查询中使用的宏值

hadoopcmd：

| sub_commnad： | 必须是这些[“jar”,“s3distcp”,“流媒体”]后跟一个或多个 args

shellcmd：

| 脚本： | 带有 args 的内联命令| script_location： ||---||| 包含查询语句的 s3 位置 || 文件： | s3 存储桶中的文件列表为 file1，file2 格式。这些文件将被复制到正在执行 qubole 命令的工作目录中。| 档案： | s3 存储桶中的存档列表为 archive1，archive2 格式。这些将被解压缩到正在执行 qubole 命令的工作目录中参数:需要传递给脚本的任何额外的 args(仅当提供了 script_location 时）

pigcmd：

| 脚本： | 内联查询语句（latin_statements）| script_location： ||---||| 包含 pig 查询的 s3 位置 | 参数：需要传递给脚本的任何额外的 args（仅当提供了 script_location 时）

sparkcmd：

| 程序： | Scala，SQL，Command，R 或 Python 中的完整 Spark 程序| CMDLINE： | spark-submit 命令行，必须在 cmdline 本身中指定所有必需的信息。| SQL： | 内联 sql 查询| script_location： ||---||| 包含查询语句的 s3 位置 || 语言：| 程序的语言,Scala,SQL,Command,R 或 Python| APP_ID： | Spark 作业服务器应用程序的 ID 参数：spark-submit 命令行参数| user_program_arguments： ||---

| | | 用户程序所接受的参数 | | 宏： | 在查询中使用的宏值

dbtapquerycmd：

| db_tap_id： | Qubole 中目标数据库的数据存储 ID。| 查询： | 内联查询语句| 宏： | 在查询中使用的宏值

dbexportcmd：

| 模式： | 1（简单），2（提前）| hive_table： | 配置单元表的名称| partition_spec： | Hive 表的分区规范。| dbtap_id： | Qubole 中目标数据库的数据存储 ID。| db_table： | db 表的名称| db_update_mode： | allowinsert 或 updateonly| db_update_keys： | 用于确定行唯一性的列| export_dir： | HDFS / S3 将从中导出数据的位置。| fields_terminated_by： | | --- | | | 用作数据集中列分隔符的 char 的十六进制 |

dbimportcmd：

| 模式： | 1（简单），2（提前）| hive_table： | 配置单元表的名称| dbtap_id： | Qubole 中目标数据库的数据存储 ID。| db_table： | db 表的名称| where_clause： | where 子句，如果有的话| 并行： | 用于提取数据的并行数据库连接数| extract_query： | SQL 查询从 db 中提取数据。$ CONDITIONS 必须是 where 子句的一部分。| boundary_query：| 要使用的查询获取要提取的行 ID 范围| split_column：| 用作行 ID 的列将数据拆分为范围（模式 2） 注意

以下字段模板支持：query，script_location，sub_command，script，files，archives，program，cmdline，sql，where_clause，extract_query，boundary_query，macros，tags，name，parameters，dbtap_id，hive_table，db_table，split_column，note_id，db_update_keys，export_dir，partition_spec，qubole_conn_id，arguments，user_program_arguments。

您还可以将.txt 文件用于模板驱动的用例。

注意:在 QuboleOperator 中，有一个用于任务失败和重试的默认处理程序，它通常会杀死在 QDS 上运行的相应任务实例的命令。您可以通过在任务定义中提供自己的失败和重试处理程序来覆盖此行为。

class airflow.contrib.operators.s3listoperator.S3ListOperator(bucket,prefix='',delimiter='',aws_conn_id='aws_default', * args, ** kwargs)

基类： airflow.models.BaseOperator

列出桶中具有名称中给定字符串前缀的所有对象。

此运算符返回一个 python 列表，其中包含可由 xcom 在下游任务中使用的对象名称。

参数：

- bucket(string) - S3 存储桶在哪里找到对象。（模板）
- prefix(string) - 用于过滤名称以此前缀开头的对象的前缀字符串。（模板）
- delimiter(string) - 分隔符标记键层次结构。（模板）
- aws_conn_id(string) - 连接到 S3 存储时使用的连接 ID。

Example:
以下运算符将列出存储桶中 S3 customers/2018/04/键的所有文件（不包括子文件夹）data。

```
s3_file = S3ListOperator (
    task_id = 'list_3s_files' ,
    bucket = 'data' ,
    prefix = 'customers/2018/04/' ,
    delimiter = '/' ,
    aws_conn_id = 'aws_customers_conn'
)
```

```
class airflow.contrib.operators.s3_to_gcs_operator.S3ToGoogleCloudStorageOperator （bucket，prefix =''，delimiter =''，aws_conn_id ='aws_default'，dest_gcs_conn_id = None，dest_gcs = None，delegate_to = None，replace = False，* args，** kwargs）
```

基类： airflow.contrib.operators.s3listoperator.S3ListOperator

将 S3 密钥（可能是前缀）与 Google 云端存储目标路径同步。

参数：

- bucket(string) - S3 存储桶在哪里找到对象。（模板）
- prefix(string) - 前缀字符串，用于过滤名称以此前缀开头的对象。（模板）
- delimiter(string) - 分隔符标记键层次结构。（模板）
- aws_conn_id(string) - 源 S3 连接
- dest_gcs_conn_id(string) - 连接到 Google 云端存储时要使用的目标连接 ID。
- dest_gcs(string) - 要存储文件的目标 Google 云端存储分区和前缀。（模板）
- delegate_to(string) - 模拟的帐户（如果有）。为此，发出请求的服务帐户必须启用域范围委派。
- replace(bool) - 是否要替换现有目标文件。

示例：.. code-block :: python

```
>  s3_to_gcs_op = S3ToGoogleCloudStorageOperator (
>
```

task_id ='s3_to_gcs_example'， bucket ='my-s3-bucket'， prefix ='data / customers-201804'，dest_gcs_conn_id ='google_cloud_default'，dest_gcs ='gs：//my.gcs.bucket/some/customers/' ，replace = False，dag = my-dag）

需要注意的是 bucket，prefix，delimiter 和 dest_gcs 模板化，所以你可以，如果你想使用变量在其中。

class airflow.contrib.operators.segment_track_event_operator.SegmentTrackEventOperator（user_id，event，properties = None，segment_conn_id ='segment_default'，segment_debug_mode = False，* args，** kwargs）

基类： airflow.models.BaseOperator

将跟踪事件发送到 Segment 以获取指定的 user_id 和事件

参数：

- user_id(string) - 数据库中此用户的 ID。（模板）
- event(string) - 您要跟踪的事件的名称。（模板）
- properties(dict) - 事件属性的字典。（模板）
- segment_conn_id(string) - 连接到 Segment 时使用的连接 ID。
- segment_debug_mode(_boolean_) - 确定 Segment 是否应在调试模式下运行。默认为 False

class airflow.contrib.operators.sftp_operator.SFTPOperator（ssh_hook = None，ssh_conn_id = None，remote_host = None，local_filepath = None，remote_filepath = None，operation ='put'，* args，** kwargs）

基类： airflow.models.BaseOperator

SFTPOperator 用于将文件从远程主机传输到本地或反之亦然。此运算符使用 ssh_hook 打开 sftp trasport 通道，该通道用作文件传输的基础。

参数：

- ssh_hook(SSHHook) - 用于远程执行的预定义 ssh_hook
- ssh_conn_id(str) - 来自 airflow Connections 的连接 ID
- remote_host(str) - 要连接的远程主机
- local_filepath(str) - 要获取或放置的本地文件路径。（模板）
- remote_filepath(str) - 要获取或放置的远程文件路径。（模板）
- 操作 - 指定操作'get'或'put'，默认为 get

class airflow.contrib.operators.slack_webhook_operator.SlackWebhookOperator （http_conn_id = None，webhook_token = None，message =''，channel = None，username = None，icon_emoji = None，link_names = False，proxy = None，* args，** kwargs ）

基类： airflow.operators.http_operator.SimpleHttpOperator

此运算符允许您使用传入的 webhook 将消息发布到 Slack。直接使用 Slack webhook 令牌和具有 Slack webhook 令牌的连接。如果两者都提供，将使用 Slack webhook 令牌。

每个 Slack webhook 令牌都可以预先配置为使用特定频道，用户名和图标。您可以在此挂钩中覆盖这些默认值。

参数：

- conn_id(str) - 在额外字段中具有 Slack webhook 标记的连接
- webhook_token(str) - Slack webhook 令牌
- message(str) - 要在 Slack 上发送的消息
- channel(str) - 邮件应发布到的频道
- username(str) - 要发布的用户名
- icon_emoji(str) - 用作发布给 Slack 的用户图标的表情符号
- link_names(bool) - 是否在邮件中查找和链接频道和用户名
- proxy(str) - 用于进行 Slack webhook 调用的代理

执行（上下文）

调用 SparkSqlHook 来运行提供的 sql 查询

class airflow.contrib.operators.snowflake_operator.SnowflakeOperator（sql，snowflake_conn_id='snowflake_default'，parameters = None，autocommit = True，warehouse = None，database = None，* args，** kwargs）

基类： airflow.models.BaseOperator

在 Snowflake 数据库中执行 sql 代码

参数：

- snowflake_conn_id(string) - 对特定雪花连接 ID 的引用
- SQL(_ 可接收表示 SQL 语句中的海峡 _ _，_ _ 海峡列表 _ _（_ _SQL 语句 _ _）_ _，或 _ _ 参照模板文件模板引用在“.SQL”结束海峡认可。_) -要执行的 SQL 代码。（模板）
- warehouse(string) - 覆盖连接中已定义的仓库的仓库名称
- database(string) - 覆盖连接中定义的数据库的数据库名称

class airflow.contrib.operators.spark_jdbc_operator.SparkJDBCOperator（spark_app_name='airflow-spark-jdbc'，spark_conn_id='spark-default'，spark_conf = None，spark_py_files = None，spark_files = None，spark_jars = None，num_executors = None，executor_cores = None，executor_memory = None，driver_memory = None，verbose = False，keytab = None，principal = None，cmd_type='spark_to_jdbc'，jdbc_table = None，jdbc_conn_id ='jdbc-default'，jdbc_driver = None，metastore_table = None，jdbc_truncate = False，save_mode = None，save_format = None，batch_size = None，fetch_size = None，num_partitions = None，partition_column = None，lower_bound = None，upper_bound = None，create_table_column_types = None，* args，** kwargs）

基类： airflow.contrib.operators.spark_submit_operator.SparkSubmitOperator

此运算符专门用于扩展 SparkSubmitOperator，以便使用 Apache Spark 执行与基于 JDBC 的数据库之间的数据传输。与 SparkSubmitOperator 一样，它假定 PATH 上有“spark-submit”二进制文件。

参数：

- spark_app_name**（str） - 作业名称（默认为**airflow -spark-jdbc）
- spark_conn_id(str) - 在 Airflow 管理中配置的连接 ID
- spark_conf(dict) - 任何其他 Spark 配置属性
- spark_py_files(str) - 使用的其他 python 文件（.zip，.egg 或.py）
- spark_files(str) - 要上载到运行作业的容器的其他文件
- spark_jars(str) - 要上传并添加到驱动程序和执行程序类路径的其他 jar
- num_executors(int) - 要运行的执行程序的数量。应设置此项以便管理使用 JDBC 数据库建立的连接数
- executor_cores(int) - 每个执行程序的核心数
- executor_memory(str) - 每个执行者的内存（例如 1000M，2G）
- driver_memory(str) - 分配给驱动程序的内存（例如 1000M，2G）
- verbose(bool) - 是否将详细标志传递给 spark-submit 进行调试
- keytab(str) - 包含 keytab 的文件的完整路径
- principal(str) - 用于 keytab 的 kerberos 主体的名称
- cmd_type(str) - 数据应该以哪种方式流动。2 个可能的值：spark_to_jdbc：从 Metastore 到 jdbc jdbc_to_spark 的 spark 写入的数据：来自 jdbc 到 Metastore 的 spark 写入的数据
- jdbc_table(str) - JDBC 表的名称
- jdbc_conn_id - 用于连接 JDBC 数据库的连接 ID
- jdbc_driver(str) - 用于 JDBC 连接的 JDBC 驱动程序的名称。这个驱动程序（通常是一个 jar）应该在'jars'参数中传递
- metastore_table(str) - Metastore 表的名称，
- jdbc_truncate(bool) - （仅限 spark_to_jdbc）Spark 是否应截断或删除并重新创建 JDBC 表。这仅在'save_mode'设置为 Overwrite 时生效。此外，如果架构不同，Spark 无法截断，并将丢弃并重新创建
- save_mode(str) - 要使用的 Spark 保存模式（例如覆盖，追加等）
- save_format(str) - （jdbc_to_spark-only）要使用的 Spark 保存格式（例如镶木地板）
- batch_size(int) - （仅限 spark_to_jdbc）每次往返 JDBC 数据库时要插入的批处理的大小。默认为 1000
- fetch_size(int) - （仅限 jdbc_to_spark）从 JDBC 数据库中每次往返获取的批处理的大小。默认值取决于 JDBC 驱动程序
- num_partitions(int) - Spark 同时可以使用的最大分区数，包括 spark_to_jdbc 和 jdbc_to_spark 操作。这也将限制可以打开的 JDBC 连接数
- partition_column(str) - （jdbc_to_spark-only）用于对 Metastore 表进行分区的数字列。如果已指定，则还必须指定：num_partitions，lower_bound，upper_bound
- lower_bound(int) - （jdbc_to_spark-only）要获取的数字分区列范围的下限。如果已指定，则还必须指定：num_partitions，partition_column，upper_bound
- upper_bound(int) - （jdbc_to_spark-only）要获取的数字分区列范围的上限。如果已指定，则还必须指定：num_partitions，partition_column，lower_bound

- create_table_column_types - （spark_to_jdbc-only）创建表时要使用的数据库列数据类型而不是默认值。应以与 CREATE TABLE 列语法相同的格式指定数据类型信息（例如：“name CHAR(64),comments VARCHAR（1024）”）。指定的类型应该是有效的 spark sql 数据类型。
- 类型： jdbc_conn_id：str

执行（上下文）

调用 SparkSubmitHook 来运行提供的 spark 作业

class airflow.contrib.operators.spark_sql_operator.SparkSqlOperator（sql，conf = None，conn_id ='spark_sql_default',total_executor_cores = None,executor_cores = None,executor_memory = None，keytab = None，principal = None，master ='yarn'，name ='default-name'，num_executors = None，yarn_queue ='default'，* args，** kwargs）

基类： airflow.models.BaseOperator

执行 Spark SQL 查询

参数：

- sql(str) - 要执行的 SQL 查询。（模板）
- conf(_str __（_ _ 格式：PROP = VALUE __）_) - 任意 Spark 配置属性
- conn_id(str) - connection_id 字符串
- total_executor_cores(int) - （仅限 Standalone 和 Mesos）所有执行程序的总核心数（默认值：工作程序上的所有可用核心）
- executor_cores(int) - （仅限 Standalone 和 YARN）每个执行程序的核心数（默认值：2）
- executor_memory(str) - 每个执行程序的内存（例如 1000M，2G）（默认值：1G）
- keytab(str) - 包含 keytab 的文件的完整路径
- master(str) - spark：// host：port，mesos：// host：port，yarn 或 local
- name(str) - 作业名称
- num_executors(int) - 要启动的执行程序数
- verbose(bool) - 是否将详细标志传递给 spark-sql
- yarn_queue(str) - 要提交的 YARN 队列（默认值：“default”）

执行（上下文）

调用 SparkSqlHook 来运行提供的 sql 查询

class airflow.contrib.operators.spark_submit_operator.SparkSubmitOperator（application =''，conf = None，conn_id ='spark_default'，files = None，py_files = None，driver_classpath = None，jars = None，java_class = None，packages = None， exclude_packages = None，repositories = None，total_executor_cores = None,executor_cores = None,executor_memory = None,driver_memory = None，keytab = None，principal = None，name ='airflow-spark'，num_executors = None，application_args = None，env_vars =无，详细=假，* args，** kwargs）

基类： airflow.models.BaseOperator

这个钩子是一个围绕 spark-submit 二进制文件的包装器来启动一个 spark-submit 作业。它要求“spark-submit”二进制文件在 PATH 中，或者 spark-home 在连接中的额外设置。

参数：

- application(str) - 作为作业提交的应用程序，jar 或 py 文件。(模板)
- conf(dict) - 任意 Spark 配置属性
- conn_id(str) - Airflow 管理中配置的连接 ID。当提供无效的 connection_id 时，它将默认为 yarn。
- files(str) - 将其他文件上载到运行作业的执行程序，以逗号分隔。文件将放在每个执行程序的工作目录中。例如，序列化对象。
- py_files(str) - 作业使用的其他 python 文件可以是.zip，.egg 或.py。
- jars(str) - 提交其他 jar 以上传并将它们放在执行程序类路径中。
- driver_classpath(str) - 其他特定于驱动程序的类路径设置。
- java_class(str) - Java 应用程序的主要类
- packages(str) - 包含在驱动程序和执行程序类路径上的 jar 的 maven 坐标的逗号分隔列表。(模板)
- exclude_packages(str) - 解析“包”中提供的依赖项时要排除的 jar 的 maven 坐标的逗号分隔列表
- repositories(str) - 以逗号分隔的其他远程存储库列表，用于搜索“packages”给出的 maven 坐标
- total_executor_cores(int) - （仅限 Standalone 和 Mesos）所有执行程序的总核心数（默认值：工作程序上的所有可用核心）
- executor_cores(int) - （仅限 Standalone 和 YARN）每个执行程序的核心数（默认值：2）
- executor_memory(str) - 每个执行程序的内存（例如 1000M，2G）(默认值：1G)
- driver_memory(str) - 分配给驱动程序的内存（例如 1000M，2G）(默认值：1G)
- keytab(str) - 包含 keytab 的文件的完整路径
- principal(str) - 用于 keytab 的 kerberos 主体的名称
- name(str) - 作业名称（默认气流 - 火花)。(模板)
- num_executors(int) - 要启动的执行程序数
- application_args(list) - 正在提交的应用程序的参数
- env_vars(dict) - spark-submit 的环境变量。它也支持纱线和 k8s 模式。
- verbose(bool) - 是否将详细标志传递给 spark-submit 进程进行调试

执行（上下文）

调用 SparkSubmitHook 来运行提供的 spark 作业

```
class airflow.contrib.operators.sqoop_operator.SqoopOperator ( conn_id ='sqoop_default', cmd_type ='import',table = None,query = None,target_dir = None,append = None,file_type ='text', columns = None,num_mappers = None,split_by = None,其中= None,export_dir = None,input_null_string = None, input_null_non_string = None, staging_table = None, clear_staging_table = False,
```

enclosed_by = None , escaped_by = None , input_fields_terminated_by = None , input_lines_terminated_by = None, input_optionally_enclosed_by = None , batch = False, direct = False, driver = None, verbose = False, relaxed_isolation = False, properties = None, hcatalog_database = None , hcatalog_table = None , create_hcatalog_table = False , extra_import_options = None, extra_export_options = None, * args, ** kwargs)

基类： airflow.models.BaseOperator

执行 Sqoop 作业。Apache Sqoop 的文档可以在这里找到：
https://sqoop.apache.org/docs/1.4.2/SqoopUserGuide.html。

执行（上下文）

执行 sqoop 作业

class airflow.contrib.operators.ssh_operator.SSHOperator（ssh_hook = None, ssh_conn_id = None, remote_host = None, command = None, timeout = 10, do_xcom_push = False, * args, ** kwargs）

基类： airflow.models.BaseOperator

SSHOperator 使用 ssh_hook 在给定的远程主机上执行命令。

参数：

- ssh_hook(SSHHook) - 用于远程执行的预定义 ssh_hook
- ssh_conn_id(str) - 来自 airflow Connections 的连接 ID
- remote_host(str) - 要连接的远程主机
- command(str) - 在远程主机上执行的命令。（模板）
- timeout(int) - 执行命令的超时（以秒为单位）。
- do_xcom_push(bool) - 返回由气流平台在 xcom 中设置的标准输出

class airflow.contrib.operators.vertica_operator.VerticaOperator（ sql , vertica_conn_id ='vertica_default', * args, ** kwargs）

基类： airflow.models.BaseOperator

在特定的 Vertica 数据库中执行 sql 代码

参数：

- vertica_conn_id(string) - 对特定 Vertica 数据库的引用
- SQL(_ 可接收表示 SQL 语句中的海峡 _ _, _ _ 海峡列表 _ _（_ _SQL 语句 _ _）_ _, 或 _ _ 参照模板文件模板引用在“.SQL”结束海峡认可。_) -要执行的 SQL 代码。（模板）

class airflow.contrib.operators.vertica_to_hive.VerticaToHiveTransfer（sql，hive_table，create = True,recreate = False,partition = None,delimiter = u'x01',vertica_conn_id ='vertica_default', hive_cli_conn_id ='hive_cli_default', * args, * * kwargs）
基类： airflow.models.BaseOperator

将数据从 Vertia 移动到 Hive。操作员针对 Vertia 运行查询，在将文件加载到 Hive 表之前将文件存储在本地。如果将 create 或 recreate 参数设置为 True，则生成 a 和语句。从游标的元数据推断出 Hive 数据类型。请注意，在 Hive 中生成的表使用的不是最有效的序列化格式。如果加载了大量数据和/或表格被大量查询，您可能只想使用此运算符将数据暂存到临时表中，然后使用 a 将其加载到最终目标中。CREATE TABLE``DROP TABLE``STORED AS textfile``HiveOperator

参数：

- sql(str) - 针对 Vertia 数据库执行的 SQL 查询。（模板）
- hive_table(str) - 目标 Hive 表，使用点表示法来定位特定数据库。（模板）
- create(bool) - 是否创建表，如果它不存在
- recreate(bool) - 是否在每次执行时删除并重新创建表
- partition(dict) - 将目标分区作为分区列和值的字典。（模板）
- delimiter(str) - 文件中的字段分隔符
- vertica_conn_id(str) - 源 Vertica 连接
- hive_conn_id(str) - 目标配置单元连接

class airflow.contrib.operators.winrm_operator.WinRMOperator（winrm_hook = None， ssh_conn_id = None， remote_host = None， command = None， timeout = 10， do_xcom_push = False， * args， ** kwargs）

基类： airflow.models.BaseOperator

WinRMOperator 使用 winrm_hook 在给定的远程主机上执行命令。

参数：

winrm_hook(WinRMHook) - 用于远程执行的预定义 ssh_hook
ssh_conn_id(str) - 来自 airflow Connections 的连接 ID
remote_host(str) - 要连接的远程主机
command(str) - 在远程主机上执行的命令。（模板）
timeout(int) - 执行命令的超时。
do_xcom_push(bool) - 返回由气流平台在 xcom 中设置的标准输出

传感器

class airflow.contrib.sensors.aws_redshift_cluster_sensor.AwsRedshiftClusterSensor（cluster_identifier,target_status ='available',aws_conn_id ='aws_default',* args,** kwargs）

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

等待 Redshift 群集达到特定状态。

参数：

- cluster_identifier(str) - 要 ping 的集群的标识符。
- target_status(str) - 所需的集群状态。

戳（上下文）

传感器在派生此类时定义的功能应该覆盖。

class airflow.contrib.sensors.bash_sensor.BashSensor（bash_command，env = None，output_encoding ='utf-8'，* args，** kwargs）

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

当且仅当返回码为 0 时，执行 bash 命令/脚本并返回 True。

参数：

- bash_command(string) - 要执行的命令，命令集或对 bash 脚本（必须为'.sh'）的引用。
- env(dict) - 如果 env 不是 None，它必须是一个定义新进程的环境变量的映射；这些用于代替继承当前进程环境，这是默认行为。（模板）
- output_encoding(string) - bash 命令的输出编码。

戳（上下文）

在临时目录中执行 bash 命令，之后将对其进行清理

class airflow.contrib.sensors.bigquery_sensor.BigQueryTableSensor（project_id，dataset_id，table_id，bigquery_conn_id ='bigquery_default_conn'，delegate_to = None，* args，** kwargs）

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

检查 Google Bigquery 中是否存在表格。

<colgroup><col class="field-name"><col class="field-body"></colgroup> | param project_id： | | --- | | | 要在其中查找表的 Google 云项目。提供给钩子的连接必须提供对指定项目的访问。 | | 类型 project_id： | | --- | | | 串 | | param dataset_id： | | --- | | | 要在其中查找表的数据集的名称。存储桶。 | | 类型 dataset_id： | | --- | | | 串 | | param table_id： | 要检查表的名称是否存在。 | | --- | --- | | type table_id： | 串 | | --- | --- | | param bigquery_conn_id： | | --- | | | 连接到 Google BigQuery 时使用的连接 ID。 | | 输入 bigquery_conn_id： | | --- | | | 串 | | param delegate_to： | | --- | | | 假冒的帐户，如果有的话。为此，发出请求的服务帐户必须启用

域范围委派。 | | type delegate_to: | | --- | | | 串 |

戳（上下文）
传感器在派生此类时定义的功能应该覆盖。

class airflow.contrib.sensors.datadog_sensor.DatadogSensor（datadog_conn_id ='datadog_default'，from_seconds_ago = 3600，up_to_seconds_from_now = 0，priority = None，sources = None，tags = None，response_check = None，* args，** kwargs）

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

一个传感器，通过过滤器监听数据对象事件流并确定是否发出了某些事件。

取决于 datadog API，该 API 必须部署在 Airflow 运行的同一服务器上。

参数：

- datadog_conn_id - 与 datadog 的连接，包含 api 密钥的元数据。
- datadog_conn_id - 字符串

戳（上下文）

传感器在派生此类时定义的功能应该覆盖。

class airflow.contrib.sensors.emr_base_sensor.EmrBaseSensor（aws_conn_id ='aws_default'，* args，** kwargs）

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

包含 EMR 的一般传感器行为。子类应该实现 get_emr_response（）和 state_from_response（）方法。子类还应实现 NON_TERMINAL_STATES 和 FAILED_STATE 常量。

戳（上下文）

传感器在派生此类时定义的功能应该覆盖。

class airflow.contrib.sensors.emr_job_flow_sensor.EmrJobFlowSensor（job_flow_id，* args，** kwargs）

基类： airflow.contrib.sensors.emr_base_sensor.EmrBaseSensor

询问 JobFlow 的状态，直到它达到终端状态。如果传感器错误失败，则任务失败。

参数：job_flow_id(string) - job_flow_id 来检查状态

class airflow.contrib.sensors.emr_step_sensor.EmrStepSensor（job_flow_id，step_id，*args，**kwargs）

基类： airflow.contrib.sensors.emr_base_sensor.EmrBaseSensor

询问步骤的状态，直到达到终端状态。如果传感器错误失败，则任务失败。

参数：

- job_flow_id(string) - job_flow_id，其中包含检查状态的步骤
- step_id(string) - 检查状态的步骤

class airflow.contrib.sensors.file_sensor.FileSensor（filepath，fs_conn_id ='fs_default2'，*args，** kwargs）

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

等待文件或文件夹降落到文件系统中。

如果给定的路径是目录，则该传感器仅在其中存在任何文件时才返回 true（直接或在子目录中）

参数：

- fs_conn_id(string) - 对文件（路径）连接 ID 的引用
- filepath - 文件或文件夹名称（相对于连接中设置的基本路径）

戳（上下文）

传感器在派生此类时定义的功能应该覆盖。

class airflow.contrib.sensors.ftp_sensor.FTPSensor（path，ftp_conn_id ='ftp_default'，*args，** kwargs）

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

等待 FTP 上存在文件或目录。

参数：

- path(str) - 远程文件或目录路径
- ftp_conn_id(str) - 运行传感器的连接

戳（上下文）

传感器在派生此类时定义的功能应该覆盖。

class airflow.contrib.sensors.ftp_sensor.FTPSSensor（path，ftp_conn_id ='ftp_default'，* args，** kwargs）

基类： airflow.contrib.sensors.ftp_sensor.FTPSensor

等待通过 SSL 在 FTP 上存在文件或目录。

class airflow.contrib.sensors.gcs_sensor.GoogleCloudStorageObjectSensor （ bucket， object，google_cloud_conn_id ='google_cloud_default'，delegate_to = None，* args，** kwargs）

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

检查 Google 云端存储中是否存在文件。创建一个新的 GoogleCloudStorageObjectSensor。

<colgroup><col class="field-name"><col class="field-body"></colgroup> | param bucket： | 对象所在的 Google 云存储桶。 | | --- | --- | | 型桶： | 串 | | --- | --- | | 参数对象： | 要在 Google 云存储分区中检查的对象的名称。 | | --- | --- | | 类型对象： | 串 | | --- | --- | | param google_cloud_storage_conn_id： | | --- | | | 连接到 Google 云端存储时使用的连接 ID。 | | 输入 google_cloud_storage_conn_id： | | --- | | | 串 | | param delegate_to： | | --- | | | 假冒的帐户，如果有的话。为此，发出请求的服务帐户必须启用域范围委派。 | | type delegate_to： | | --- | | | 串 |

戳（上下文）

传感器在派生此类时定义的功能应该覆盖。

class airflow.contrib.sensors.gcs_sensor.GoogleCloudStorageObjectUpdatedSensor（bucket，object，ts_func = <function ts_function>，google_cloud_conn_id ='google_cloud_default'，delegate_to = None，* args，** kwargs）

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

检查 Google Cloud Storage 中是否更新了对象。创建一个新的 GoogleCloudStorageObjectUpdatedSensor。

<colgroup><col class="field-name"><col class="field-body"></colgroup> | param bucket： | 对象所在的 Google 云存储桶。 | | --- | --- | | 型桶： | 串 | | --- | --- | | 参数对象： | 要在 Google 云存储分区中下载的对象的名称。 | | --- | --- | | 类型对象： | 串 | | --- | --- | | param ts_func： | 用于定义更新条件的回调。默认回调返回 execution_date + schedule_interval。回调将上下文作为参数。 | | --- | --- | | 输入 ts_func： | 功能 | | --- | --- | | param google_cloud_storage_conn_id： | | --- | | | 连接到 Google 云端存储时使用的连接 ID。 | | 输入 google_cloud_storage_conn_id： | | --- | | | 串 | | param delegate_to： | | --- | | | 假冒的帐户，如果有的话。为此，发出请求

的服务帐户必须启用域范围委派。 | | type delegate_to： | | --- | | | 串 |

戳（上下文）

传感器在派生此类时定义的功能应该覆盖。

class airflow.contrib.sensors.gcs_sensor.GoogleCloudStoragePrefixSensor （ bucket， prefix， google_cloud_conn_id ='google_cloud_default'， delegate_to = None， * args， ** kwargs）

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

检查 Google 云端存储分区中是否存在前缀文件。创建一个新的 GoogleCloudStorageObjectSensor。

<colgroup><col class="field-name"><col class="field-body"></colgroup> | param bucket： | 对象所在的 Google 云存储桶。 | | --- | --- | | 型桶： | 串 | | --- | --- | | 参数前缀： | 要在 Google 云存储分区中检查的前缀的名称。 | | --- | --- | | 类型前缀： | 串 | | --- | --- | | param google_cloud_storage_conn_id： | | --- | | | 连接到 Google 云端存储时使用的连接 ID。 | | 输入 google_cloud_storage_conn_id： | | --- | | | 串 | | param delegate_to： | | --- | | | 假冒的帐户，如果有的话。为此，发出请求的服务帐户必须启用域范围委派。 | | type delegate_to： | | --- | | | 串 |

戳（上下文）

传感器在派生此类时定义的功能应该覆盖。

class airflow.contrib.sensors.hdfs_sensor.HdfsSensorFolder(be_empty = False，* args，** kwargs）

基类： airflow.sensors.hdfs_sensor.HdfsSensor

戳（上下文）

戳一个非空的目录

返回值： Bool 取决于搜索条件

class airflow.contrib.sensors.hdfs_sensor.HdfsSensorRegex（regex， * args， ** kwargs）

基类： airflow.sensors.hdfs_sensor.HdfsSensor

戳（上下文）

用 self.regex 戳一个目录中的文件

返回值： Bool 取决于搜索条件

class airflow.contrib.sensors.jira_sensor.JiraSensor（jira_conn_id='jira_default',method_name = None，method_params = None，result_processor = None，* args，** kwargs）

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

如有任何变更，请监控 jira 票。

参数：

- jira_conn_id(str) - 对预定义的 Jira 连接的引用
- method_name(str) - 要执行的 jira-python-sdk 的方法名称
- method_params(dict) - 方法 method_name 的参数
- result_processor(function) - 返回布尔值并充当传感器响应的函数

戳（上下文）

传感器在派生此类时定义的功能应该覆盖。

class airflow.contrib.sensors.pubsub_sensor.PubSubPullSensor（ project， subscription， max_messages = 5， return_immediately = False， ack_messages = False， gcp_conn_id ='google_cloud_default'，delegate_to = None，* args，** kwargs）

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

从 PubSub 订阅中提取消息并通过 XCom 传递它们。

此传感器操作员将从 max_messages 指定的 PubSub 订阅中提取消息。当订阅返回消息时，将完成 poke 方法的标准，并且将从操作员返回消息并通过 XCom 传递下游任务。

如果 ack_messages 设置为 True，则在返回之前将立即确认消息，否则，下游任务将负责确认消息。

project 并且 subscription 是模板化的，因此您可以在其中使用变量。

执行（上下文）

重写以允许传递消息

戳（上下文）
传感器在派生此类时定义的功能应该覆盖。

class airflow.contrib.sensors.qubole_sensor.QuboleSensor（ data， qubole_conn_id ='qubole_default'，* args，** kwargs）

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

所有 Qubole 传感器的基类

参数：

- qubole_conn_id(string) - 用于运行传感器的 qubole 连接
- data(_JSON 对象 _) - 包含有效负载的 JSON 对象，需要检查其存在

注意:两个 data 和 qubole_conn_id 字段都是模板支持的。您可以

还使用.txt 文件进行模板驱动的用例。

戳（上下文）

传感器在派生此类时定义的功能应该覆盖。

class airflow.contrib.sensors.redis_key_sensor.RedisKeySensor（key，redis_conn_id，*args，**kwargs）

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

检查 Redis 数据库中是否存在密钥

戳（上下文）

传感器在派生此类时定义的功能应该覆盖。

class airflow.contrib.sensors.sftp_sensor.SFTPSensor(path,sftp_conn_id='sftp_default',*args,** kwargs）

基类： airflow.operators.sensors.BaseSensorOperator

等待 SFTP 上存在文件或目录。:param path：远程文件或目录路径：type path：str：param sftp_conn_id：运行传感器的连接：键入 sftp_conn_id：str

戳（上下文）

传感器在派生此类时定义的功能应该覆盖。

class airflow.contrib.sensors.wasb_sensor.WasbBlobSensor （ container_name ， blob_name ，wasb_conn_id ='wasb_default'，check_options = None，* args，** kwargs）

基类： airflow.sensors.base_sensor_operator.BaseSensorOperator

等待 blob 到达 Azure Blob 存储。

参数：

- container_name(str) - 容器的名称。
- blob_name(str) - blob 的名称。
- wasb_conn_id(str) - 对 wasb 连接的引用。
- check_options(dict) - WasbHook.check_for_blob（）采用的可选关键字参数。

戳（上下文）

传感器在派生此类时定义的功能应该覆盖。

宏

这是可以在模板中使用的变量和宏的列表

默认变量

默认情况下，Airflow 引擎会传递一些可在所有模板中访问的变量

| 变量 | 描述| `{{ ds }}` | 执行日期为 `YYYY-MM-DD` | | `{{ ds_nodash }}` | 执行日期为 `YYYYMMDD` | | `{{ prev_ds }}` | 上一个执行日期为 `YYYY-MM-DD` 。如果是和是，将是。`{{ ds }}``2016-01-08``schedule_interval``@weekly``{{ prev_ds }}``2016-01-01` | | `{{ next_ds }}` | 下一个执行日期为 `YYYY-MM-DD` 。如果是和是，将是。`{{ ds }}``2016-01-01``schedule_interval``@weekly``{{ prev_ds }}``2016-01-08` | | `{{ yesterday_ds }}` | 昨天的日期是 `YYYY-MM-DD` | | `{{ yesterday_ds_nodash }}` | 昨天的日期是 `YYYYMMDD` | | `{{ tomorrow_ds }}` | 明天的日期是 `YYYY-MM-DD` | | `{{ tomorrow_ds_nodash }}` | 明天的日期是 `YYYYMMDD` | | `{{ ts }}` | 与...一样 `execution_date.isoformat()` | | `{{ ts_nodash }}` | 和`ts`没有`-`和`:` | | `{{ execution_date }}` | execution_date，（datetime.datetime） | | `{{ prev_execution_date }}` | 上一个执行日期（如果可用）（datetime.datetime） | | `{{ next_execution_date }}` | 下一个执行日期（datetime.datetime） | | `{{ dag }}` | DAG 对象 | | `{{ task }}` | Task 对象 | | `{{ macros }}` | 对宏包的引用，如下所述 | | `{{ task_instance }}` | task_instance 对象 | | `{{ end_date }}` | 与...一样 `{{ ds }}` | | `{{ latest_date }}` | 与...一样 `{{ ds }}` | | `{{ ti }}` | 与...一样 `{{ task_instance }}` | | `{{ params }}` | 对用户定义的 params 字典的引用，如果启用，则可以通过字典覆盖该字典 `trigger_dag -c``dag_run_conf_overrides_params` in ``airflow.cfg` | | `{{ var.value.my_var }}` | 全局定义的变量表示为字典 | | `{{ var.json.my_var.path }}` | 全局定义的变量表示为具有反序列化 JSON 对象的字典，将路径追加到 JSON 对象中的键 | | `{{ task_instance_key_str }}` | 格式化任务实例的唯一，人类可读的键 `{dag_id}_{task_id}_{ds}` | | `{{ conf }}` | 完整的配置对象位于`airflow.configuration.conf`其中，代表您的内容`airflow.cfg` | | `{{ run_id }}` | 在`run_id`当前 DAG 运行 | | `{{ dag_run }}` | 对 DagRun 对象的引用 | | `{{ test_mode }}` | 是否使用 CLI 的 test 子命令调用任务实例 |

请注意，您可以使用简单的点表示法访问对象的属性和方法。这里有什么是可能的一些例子:,,, ...参考模型文档上对象的属性和方法的详细信息。{{ task.owner }}``{{ task.task_id }}``{{ ti.hostname }}

该 var 模板变量允许您访问气流的 UI 定义的变量。您可以以纯文本或 JSON 格式访问它们。如果使用 JSON，您还可以遍历嵌套结构，例如：{{ var.json.mydictvar.key1 }}

宏

宏是一种向模板公开对象并在模板中位于 macros 命名空间下的方法。

提供了一些常用的库和方法。

| 变量 | 描述| `macros.datetime` | 标准的 lib `datetime.datetime` | | `macros.timedelta` | 标准的 lib `datetime.timedelta` | | `macros.dateutil` | 对`dateutil`包的引用 | | `macros.time` | 标准的 lib `time` | | `macros.uuid` | 标准的 lib `uuid` | | `macros.random` | 标准的 lib `random` |
还定义了一些特定于气流的宏：

```
airflow.macros.ds_add（ds，days）
```

从 YYYY-MM-DD 添加或减去天数

参数：

- ds(str) - YYYY-MM-DD 要添加的格式的锚定日期
- days(int) - 要添加到 ds 的天数，可以使用负值

```
>>> ds_add ('2015-01-01', 5)
'2015-01-06'
>>> ds_add ('2015-01-06', -5)
'2015-01-01'
```

```
airflow.macros.ds_format（ds，input_format，output_format）
```

获取输入字符串并输出输出格式中指定的另一个字符串

参数：

- ds(str) - 包含日期的输入字符串
- input_format(str) - 输入字符串格式。例如%Y-%m-%d
- output_format(str) - 输出字符串格式例如%Y-%m-%d

```
>>> ds_format ('2015-01-01', "%Y-%m- %d ", "%m- %d -%y")
'01-01-15'
```

```
>>> ds_format ('1/5/2015', "%m/ %d /%Y", "%Y-%m- %d ")
'2015-01-05'
```

airflow.macros.random（）→x 在区间[0,1）中。

airflow.macros.hive.closest_ds_partition（table， ds， before = True， schema ='default'，metastore_conn_id ='metastore_default'）

此函数在最接近目标日期的列表中查找日期。可以给出可选参数以获得最接近的之前或之后。

参数：

- table(str) - 配置单元表名称
- ds(_datetime.date list_) - _ 日期 _ 戳，%Y-%m-%d 例如 yyyy-mm-dd
- 之前(_bool _ 或 _None_) - 最接近之前（True），之后（False）或 ds 的任一侧

返回值： 最近的日期| 返回类型： | str 或 None

```
>>> tbl = 'airflow.static_babynames_partitioned'
>>> closest_ds_partition ( tbl , '2015-01-02' )
'2015-01-01'
```

airflow.macros.hive.max_partition（table，schema ='default'，field = None，filter_map = None，metastore_conn_id ='metastore_default'）

获取表的最大分区。

参数：

- schema(string) - 表所在的 hive 模式
- table(string) - 您感兴趣的 hive 表，支持点符号，如“my_database.my_table”，如果找到一个点，则模式参数被忽略
- metastore_conn_id(string) - 您感兴趣的配置单元连接。如果设置了默认值，则不需要使用此参数。
- filter_map(_map_) - partition_key：用于分区过滤的 partition_value 映射，例如{'key1'：'value1'，'key2'：'value2'}。只有匹配所有 partition_key：partition_value 对的分区才会被视为最大分区的候选者。
- field(str) - 从中获取最大值的字段。如果只有一个分区字段，则会推断出这一点

```
>>> max_partition ('airflow.static_babynames_partitioned')
'2015-01-01'
```

楷模

模型构建在 SQLAlchemy ORM Base 类之上，实例保留在数据库中。

class airflow.models.BaseOperator(task_id,owner='Airflow',email=None,email_on_retry=True, email_on_failure = True , retries = 0 , retry_delay = datetime.timedelta (0,300), retry_exponential_backoff=False, max_retry_delay=None, start_date=None, end_date=None, schedule_interval = None, depends_on_past = False, wait_for_downstream = False, dag = None, params=None,default_args=None,adhoc=False,priority_weight=1,weight_rule=u'downstream', queue='default', pool=None, sla=None, execution_timeout=None, on_failure_callback=None, on_success_callback=None, on_retry_callback=None, trigger_rule=u'all_success', resources = None, run_as_user = None, task_concurrency = None, executor_config = None, inlets = None, outlets = None, * args, ** kwargs)

基类： airflow.utils.log.logging_mixin.LoggingMixin

所有运营商的抽象基类。由于运算符创建的对象成为 dag 中的节点，因此 BaseOperator 包含许多用于 dag 爬行行为的递归方法。要派生此类，您需要覆盖构造函数以及“execute”方法。

从此类派生的运算符应同步执行或触发某些任务（等待完成）。运算符的示例可以是运行 Pig 作业（PigOperator）的运算符，等待分区在 Hive（HiveSensorOperator）中着陆的传感器运算符，或者是将数据从 Hive 移动到 MySQL（Hive2MySqlOperator）的运算符。这些运算符（任务）的实例针对特定操作，运行特定脚本，函数或数据传输。

此类是抽象的，不应实例化。实例化从这个派生的类导致创建任务对象，该对象最终成为 DAG 对象中的节点。应使用 set_upstream 和/或 set_downstream 方法设置任务依赖性。

参数：

- task_id(string) - 任务的唯一，有意义的 id
- owner(string) - 使用 unix 用户名的任务的所有者
- retries(int) - 在失败任务之前应执行的重试次数
- retry_delay(timedelta) - 重试之间的延迟
- retry_exponential_backoff(bool) - 允许在重试之间使用指数退避算法在重试之间进行更长时间的等待（延迟将转换为秒）
- max_retry_delay(timedelta) - 重试之间的最大延迟间隔
- start_date(datetime) - start_date 用于任务，确定 execution_date 第一个任务实例。最佳做法是将 start_date 四舍五入到 DAG 中 schedule_interval。每天的工作在 000000 有一天的 start_date，每小时的工作在特定时间的 00:00 有 start_date。请注意，Airflow 只是查看最新的 execution_date 并添加 schedule_interval 以确定下一个 execution_date。同样非常重要的是要注意不同任务的依赖关系需要及时排列。如果任务 A 依赖于任务 B 并且它们的 start_date 以其 execution_date 不排列的方式偏移，则永远不会满足 A 的依赖性。如果您希望延迟任务，例如在凌晨 2 点运行每日任务,请查看 TimeSensor 和 TimeDeltaSensor。我们建议不要使用动态 start_date，建议使用固定的。有关更多信息，请阅读有关 start_date 的 FAQ 条目。
- end_date(datetime) - 如果指定，则调度程序不会超出此日期
- depends_on_past(bool) - 当设置为 true 时，任务实例将依次运行，同时依赖上一个任务的计划成功。允许 start_date 的任务实例运行。
- wait_for_downstream(bool) - 当设置为 true 时，任务 X 的实例将等待紧接上一个任务 X 实例下

游的任务在运行之前成功完成。如果任务 X 的不同实例更改同一资产，并且任务 X 的下游任务使用此资产，则此操作非常有用。请注意，在使用 wait_for_downstream 的任何地方，depends_on_past 都将强制为 True。

- queue(str) - 运行此作业时要定位到哪个队列。并非所有执行程序都实现队列管理，CeleryExecutor 确实支持定位特定队列。
- dag([_DAG_](31 "airflow.models.DAG")) - 对任务所附的 dag 的引用（如果有的话）
- priority_weight(int) - 此任务相对于其他任务的优先级权重。这允许执行程序在事情得到备份时在其他任务之前触发更高优先级的任务。
- weight_rule(str) - 用于任务的有效总优先级权重的加权方法。选项包括：default 是设置为任务的有效权重时是所有下游后代的总和。因此，上游任务将具有更高的权重，并且在使用正权重值时将更积极地安排。当您有多个 dag 运行实例并希望在每个 dag 可以继续处理下游任务之前完成所有运行的所有上游任务时，这非常有用。设置为时{ downstream | upstream | absolute}``downstream``downstream``upstream 有效权重是所有上游祖先的总和。这与 downtream 任务具有更高权重并且在使用正权重值时将更积极地安排相反。当你有多个 dag 运行实例并希望在启动其他 dag 的上游任务之前完成每个 dag 时，这非常有用。设置为时 absolute，有效重量是精确 priority_weight 指定的，无需额外加权。当您确切知道每个任务应具有的优先级权重时，您可能希望这样做。另外，当设置 absolute 为时，对于非常大的 DAGS，存在显着加速任务创建过程的额外效果。可以将选项设置为字符串或使用静态类中定义的常量 airflow.utils.WeightRule
- pool(str) - 此任务应运行的插槽池，插槽池是限制某些任务的并发性的一种方法
- sla(datetime.timedelta) - 作业预期成功的时间。请注意，这表示 timedelta 期间结束后。例如，如果您将 SLA 设置为 1 小时，则调度程序将在凌晨 1:00 之后发送电子邮件，2016-01-02 如果 2016-01-01 实例尚未成功的话。调度程序特别关注具有 SLA 的作业，并发送警报电子邮件以防止未命中。SLA 未命中也记录在数据库中以供将来参考。共享相同 SLA 时间的所有任务都捆绑在一封电子邮件中，在此之后很快就会发送。SLA 通知仅为每个任务实例发送一次且仅一次。
- execution_timeout(datetime.timedelta) - 执行此任务实例所允许的最长时间，如果超出它将引发并失败。
- on_failure_callback(callable) - 当此任务的任务实例失败时要调用的函数。上下文字典作为单个参数传递给此函数。Context 包含对任务实例的相关对象的引用，并记录在 API 的宏部分下。
- on_retry_callback - 非常类似于 on_failure_callback 重试发生时执行的。
- on_success_callback(callable) - 很像 on_failure_callback 是在任务成功时执行它。
- trigger_rule(str) - 定义应用依赖项以触发任务的规则。选项包括：默认为。可以将选项设置为字符串或使用静态类中定义的常量{ all_success | all_failed | all_done | one_success | one_failed | dummy}``all_success``airflow.utils.TriggerRule
- resources(dict) - 资源参数名称（Resources 构造函数的参数名称）与其值的映射。
- run_as_user(str) - 在运行任务时使用 unix 用户名进行模拟
- task_concurrency(int) - 设置后，任务将能够限制 execution_dates 之间的并发运行
- executor_config(dict) -
  由特定执行程序解释的其他任务级配置参数。参数由 executor 的名称命名。``示例：通过 KubernetesExecutor MyOperator（...，在特定的 docker 容器中运行此任务）

  > executor_config = {“KubernetesExecutor”： > > > {“image”:“myCustomDockerImage”}}

) ``

清晰（** kwargs）

根据指定的参数清除与任务关联的任务实例的状态。

DAG

如果设置则返回运算符的 DAG，否则引发错误

deps

返回运算符的依赖项列表。这些与执行上下文依赖性的不同之处在于它们特定于任务，并且可以由子类扩展/覆盖。

downstream_list

@property：直接下游的任务列表

执行（上下文）

这是在创建运算符时派生的主要方法。Context 与渲染 jinja 模板时使用的字典相同。

有关更多上下文，请参阅 get_template_context。

get_direct_relative_ids（上游=假）

获取当前任务（上游或下游）的直接相对 ID。

get_direct_relatives（上游=假）

获取当前任务的直接亲属，上游或下游。

get_flat_relative_ids（upstream = False，found_descendants = None）

获取上游或下游的亲属 ID 列表。

get_flat_relatives（上游=假）

获取上游或下游亲属的详细列表。

get_task_instances（session，start_date = None，end_date = None）

获取与特定日期范围的此任务相关的一组任务实例。

has_dag（）

如果已将运算符分配给 DAG，则返回 True。

on_kill（）

当任务实例被杀死时，重写此方法以清除子进程。在操作员中使用线程，子过程或多处理模块的任何使用都需要清理，否则会留下重影过程。

post_execute（context，* args，** kwargs）

调用 self.execute（）后立即触发此挂接。它传递执行上下文和运算符返回的任何结果。

pre_execute（context，* args，** kwargs）

在调用 self.execute（）之前触发此挂钩。

prepare_template（）

模板化字段被其内容替换后触发的挂钩。如果您需要操作员在呈现模板之前更改文件的内容，则应覆盖此方法以执行此操作。

render_template（attr，content，context）

从文件或直接在字段中呈现模板，并返回呈现的结果。

render_template_from_field（attr，content，context，jinja_env）

从字段中呈现模板。如果字段是字符串，它将只是呈现字符串并返回结果。如果它是集合或嵌套的集合集，它将遍历结构并呈现其中的所有字符串。

run（start_date = None，end_date = None，ignore_first_depends_on_past = False，ignore_ti_state = False，mark_success = False）

为日期范围运行一组任务实例。

schedule_interval

DAG 的计划间隔始终胜过单个任务，因此 DAG 中的任务始终排列。该任务仍然需要 schedule_interval，因为它可能未附加到 DAG。

set_downstream（task_or_task_list）

将任务或任务列表设置为直接位于当前任务的下游。

set_upstream（task_or_task_list）

将任务或任务列表设置为直接位于当前任务的上游。

upstream_list

@property：直接上游的任务列表

xcom_pull（context，task_ids = None，dag_id = None，key = u'return_value'，include_prior_dates = None）

请参见 TaskInstance.xcom_pull（）

xcom_push（context，key，value，execution_date = None）

请参见 TaskInstance.xcom_push（）

class airflow.models.Chart（** kwargs）

基类： sqlalchemy.ext.declarative.api.Base

class airflow.models.Connection（conn_id = None，conn_type = None，host = None，login = None，password = None，schema = None，port = None，extra = None，uri = None）

基类：sqlalchemy.ext.declarative.api.Base，airflow.utils.log.logging_mixin.LoggingMixin

占位符用于存储有关不同数据库实例连接信息的信息。这里的想法是脚本使用对数据库实例（conn_id）的引用，而不是在使用运算符或钩子时硬编码主机名，登录名和密码。

extra_dejson

通过反序列化 json 返回额外的属性。

class airflow.models.DAG（dag_id，description = u''，schedule_interval = datetime.timedelta（1），start_date = None，end_date = None，full_filepath = None，template_searchpath = None，user_defined_macros = None，user_defined_filters = None，default_args = None，concurrency = 16，max_active_runs = 16，dagrun_timeout = None，sla_miss_callback = None，default_view = u'tree'，orientation ='LR'，catchup = True，on_success_callback = None，on_failure_callback = None，params = None）

基类：airflow.dag.base_dag.BaseDag，airflow.utils.log.logging_mixin.LoggingMixin

dag（有向无环图）是具有方向依赖性的任务集合。dag 也有一个时间表，一个开始结束日期（可选）。对于每个计划（例如每天或每小时），DAG 需要在满足其依赖性时运行每个单独的任务。某些任务具有依赖于

其自身过去的属性，这意味着它们在完成之前的计划（和上游任务）之前无法运行。

DAG 本质上充当任务的命名空间。task_id 只能添加一次到 DAG。

参数：

- dag_id(string) - DAG 的 ID
- description(string) - 例如，在 Web 服务器上显示 DAG 的描述
- schedule_interval(_datetime.timedelta _ 或 _dateutil.relativedelta.relativedelta _ 或 _ 作为 cron 表达式的 str_) - 定义 DAG 运行的频率，此 timedelta 对象被添加到最新任务实例的 execution_date 以确定下一个计划
- start_date(_datetime.datetime_) - 调度程序将尝试回填的时间戳
- end_date(_datetime.datetime_) - DAG 无法运行的日期，对于开放式调度保留为 None
- template_searchpath(string 或 _stings 列表 _) - 此文件夹列表（非相对）定义 jinja 将在哪里查找模板。订单很重要。请注意，jinja / airflow 默认包含 DAG 文件的路径
- user_defined_macros(dict) - 将在您的 jinja 模板中公开的宏字典。例如，传递 dict(foo='bar') 给此参数允许您在与此 DAG 相关的所有 jinja 模板中。请注意，您可以在此处传递任何类型的对象。{{ foo }}
- user_defined_filters(dict) - 将在 jinja 模板中公开的过滤器字典。例如，传递给此参数允许您在与此 DAG 相关的所有 jinja 模板中。dict(hello=lambda name: 'Hello %s' % name)``{{'world' | hello }}
- default_args(dict) - 初始化运算符时用作构造函数关键字参数的默认参数的字典。请注意，运算符具有相同的钩子，并且在此处定义的钩子之前，这意味着如果您的 dict 包含'depends_on_past'：这里为True并且'depends_on_past':在运算符的调用default_args 中为False,实际值将为False。
- params(dict) - DAG 级别参数的字典，可在模板中访问，在 params 下命名。这些参数可以在任务级别覆盖。
- concurrency(int) - 允许并发运行的任务实例数
- max_active_runs(int) - 活动 DAG 运行的最大数量，超过运行状态下运行的 DAG 数量，调度程序将不会创建新的活动 DAG 运行
- dagrun_timeout(datetime.timedelta) - 指定在超时/失败之前 DagRun 应该运行多长时间，以便可以创建新的 DagRuns
- sla_miss_callback(_types.FunctionType_) - 指定报告 SLA 超时时要调用的函数。
- default_view(string) - 指定 DAG 默认视图（树，图，持续时间，甘特图，landing_times）
- orientation(string) - 在图表视图中指定 DAG 方向（LR，TB，RL，BT）
- catchup(bool) - 执行调度程序追赶（或只运行最新）？默认为 True
- on_failure_callback(callable) - 当这个 dag 的 DagRun 失败时要调用的函数。上下文字典作为单个参数传递给此函数。
- on_success_callback(callable) - 很像 on_failure_callback 是在 dag 成功时执行的。

add_task（任务）

将任务添加到 DAG

参数：任务(_task_) - 要添加的任务

add_tasks（任务）

将任务列表添加到 DAG

参数：任务(**任务**list) - 您要添加的任务

清晰（** kwargs）

清除与指定日期范围的当前 dag 关联的一组任务实例。

CLI（）

公开特定于此 DAG 的 CLI

concurrency_reached

返回一个布尔值，指示是否已达到此 DAG 的并发限制

create_dagrun（** kwargs）

从这个 dag 创建一个 dag run，包括与这个 dag 相关的任务。返回 dag run。

参数：

- run_id(string) - 定义此 dag 运行的运行 ID
- execution_date(datetime) - 此 dag 运行的执行日期
- 州(_ 州 _) - dag run 的状态
- start_date(datetime) - 应评估此 dag 运行的日期
- external_trigger(bool) - 这个 dag run 是否是外部触发的
- session(_Session_) - 数据库会话

static deactivate_stale_dags（* args，** kwargs）

在到期日期之前，停用调度程序最后触及的所有 DAG。这些 DAG 可能已被删除。

参数：expiration_date(datetime) - 设置在此时间之前触摸的非活动 DAG 返回值： 没有

static deactivate_unknown_dags（* args，** kwargs）

给定已知 DAG 列表，停用在 ORM 中标记为活动的任何其他 DAG

参数：active_dag_ids(_list __[__unicode __]_) - 活动的 DAG ID 列表返回值： 没有

文件路径

dag 对象实例化的文件位置

夹

dag 对象实例化的文件夹位置

following_schedule（DTTM）

在当地时间计算此 dag 的以下计划

参数：dttm - utc datetime 返回值： utc datetime

get_active_runs（** kwargs）

返回当前运行的 dag 运行执行日期列表

参数：会议 -返回值： 执行日期清单

get_dagrun（** kwargs）

返回给定执行日期的 dag 运行（如果存在），否则为 none。

参数：

- execution_date - 要查找的 DagRun 的执行日期。
- 会议 -

返回值： 如果找到 DagRun，否则为 None。

get_last_dagrun（** kwargs）

返回此 dag 的最后一个 dag 运行，如果没有则返回 None。最后的 dag run 可以是任何类型的运行，例如。预定或回填。重写的 DagRuns 被忽略

get_num_active_runs（** kwargs）

返回活动“正在运行”的 dag 运行的数量

参数：

- external_trigger(bool) - 对于外部触发的活动 dag 运行为 True
- 会议 -

返回值： 活动 dag 运行的数字大于 0

static get_num_task_instances（* args，** kwargs）

返回给定 DAG 中的任务实例数。

参数：

- 会话 - ORM 会话
- dag_id(_unicode_) - 获取任务并发性的 DAG 的 ID
- task_ids(_list __[__unicode __]_) - 给定 DAG 的有效任务 ID 列表
- states(_list __[__state __]_) - 要提供的过滤状态列表

返回值： 正在运行的任务数量| 返回类型： | INT

get_run_dates（start_date，end_date = None）

使用此 dag 的计划间隔返回作为参数接收的间隔之间的日期列表。返回日期可用于执行日期。

参数：

- start_date(datetime) - 间隔的开始日期
- end_date(datetime) - 间隔的结束日期，默认为 timezone.utcnow（）

返回值： dag 计划之后的时间间隔内的日期列表| 返回类型： | 名单

get_template_env（）

返回 jinja2 环境，同时考虑 DAGs template_searchpath，user_defined_macros 和 user_defined_filters

handle_callback（** kwargs）

根据成功的值触发相应的回调，即 on_failure_callback 或 on_success_callback。此方法获取此 DagRun 的单个 TaskInstance 部分的上下文，并将其与“原因”一起传递给 callable，主要用于区分 DagRun 失败。.. 注意：

The logs end up in $AIRFLOW_HOME/logs/scheduler/latest/PROJECT/DAG_FILE.py.log

参数：

- dagrun - DagRun 对象
- success - 指定是否应调用失败或成功回调的标志

- 理由 - 完成原因
- session - 数据库会话

`is_paused`

返回一个布尔值，指示此 DAG 是否已暂停

`latest_execution_date`

返回至少存在一个 dag 运行的最新日期

`normalize_schedule（DTTM）`

返回 dttm + interval，除非 dttm 是第一个区间，然后它返回 dttm

`previous_schedule（DTTM）`

在当地时间计算此 dag 的先前计划

参数：dttm - utc datetime 返回值： utc datetime

`run（start_date = None，end_date = None，mark_success = False，local = False，executor = None，donot_pickle = False,ignore_task_deps = False,ignore_first_depends_on_past = False,pool = None，delay_on_limit_secs = 1.0，verbose = False，conf = None， rerun_failed_tasks = FALSE）`

运行 DAG。

参数：

- start_date(datetime) - 要运行的范围的开始日期
- end_date(datetime) - 要运行的范围的结束日期
- mark_success(bool) - 如果成功，则将作业标记为成功而不运行它们
- local(bool) - 如果使用 LocalExecutor 运行任务，则为 True
- executor(_BaseExecutor_) - 运行任务的执行程序实例
- donot_pickle(bool) - 正确以避免腌制 DAG 对象并发送给工人
- ignore_task_deps(bool) - 如果为 True 则跳过上游任务
- ignore_first_depends_on_past(bool) - 如果为 True，则忽略第一组任务的 depends_on_past 依赖项
- pool(string) - 要使用的资源池
- delay_on_limit_secs(float) - 达到 max_active_runs 限制时，下次尝试运行 dag run 之前等待的时间（以秒为单位）
- verbose(_boolean_) - 使日志输出更详细
- conf(dict) - 从 CLI 传递的用户定义字典

set_dependency（upstream_task_id，downstream_task_id）

使用 add_task（）在已添加到 DAG 的两个任务之间设置依赖关系的简单实用工具方法

sub_dag（task_regex，include_downstream = False，include_upstream = True）

基于应该与一个或多个任务匹配的正则表达式返回当前 dag 的子集作为当前 dag 的深层副本，并且包括基于传递的标志的上游和下游邻居。

subdags

返回与此 DAG 关联的子标记对象的列表

sync_to_db（** kwargs）

将有关此 DAG 的属性保存到数据库。请注意，可以为 DAG 和 SubDAG 调用此方法。SubDag 实际上是 SubDagOperator。

参数：

- dag([_DAG_](31 "airflow.models.DAG")) - 要保存到 DB 的 DAG 对象
- sync_time(datetime) - 应将 DAG 标记为 sync'ed 的时间

返回值： 没有

test_cycle（）

检查 DAG 中是否有任何循环。如果没有找到循环，则返回 False，否则引发异常。

topological_sort（）

按地形顺序对任务进行排序，以便任务在其任何上游依赖项之后。

深受启发：http：//blog.jupo.org/2012/04/06/topological-sorting-acyclic-directed-graphs/

返回值： 拓扑顺序中的任务列表

树视图（）

显示 DAG 的 ascii 树表示

class airflow.models.DagBag（dag_folder = None，executor = None，include_examples = False）

基类：airflow.dag.base_dag.BaseDagBag，airflow.utils.log.logging_mixin.LoggingMixin

dagbag 是 dags 的集合，从文件夹树中解析出来并具有高级配置设置，例如用作后端的数据库以及用于触发任务的执行器。这样可以更轻松地为生产和开发，测试或不同的团队或安全配置文件运行不同的环境。系统级别设置现在是 dagbag 级别，以便一个系统可以运行多个独立的设置集。

参数：

- dag_folder(_unicode_) - 要扫描以查找 DAG 的文件夹
- executor - 在此 DagBag 中执行任务实例时使用的执行程序
- include_examples(bool) - 是否包含带有气流的示例
- has_logged - 在跳过文件后从 False 翻转为 True 的实例布尔值。这是为了防止用户记录有关跳过文件的消息而使用户过载。因此，每个 DagBag 只有一次被记录的文件被跳过。

bag_dag（dag，parent_dag，root_dag）

将 DAG 添加到包中，递归到子 d .. 如果在此 dag 或其子标记中检测到循环，则抛出 AirflowDagCycleException

collect_dags（dag_folder = None，only_if_updated = True）

给定文件路径或文件夹，此方法查找 python 模块，导入它们并将它们添加到 dagbag 集合中。

请注意，如果在处理目录时发现.airflowignore 文件，它的行为与.gitignore 非常相似，忽略了与文件中指定的任何正则表达式模式匹配的文件。注意：.airflowignore 中的模式被视为未锚定的正则表达式，而不是类似 shell 的 glob 模式。

dagbag_report（）

打印有关 DagBag loading stats 的报告

get_dag（dag_id）

从字典中获取 DAG，并在过期时刷新它

kill_zombies（** kwargs）

失败的任务太长时间没有心跳

process_file（filepath，only_if_updated = True，safe_mode = True）

给定 python 模块或 zip 文件的路径，此方法导入模块并在其中查找 dag 对象。

尺寸（）

返回值： 这个 dagbag 中包含的 dags 数量

class airflow.models.DagModel（** kwargs）

基类： sqlalchemy.ext.declarative.api.Base

class airflow.models.DagPickle（dag）

基类： sqlalchemy.ext.declarative.api.Base

Dags 可以来自不同的地方（用户回购，主回购，.....），也可以在不同的地方（不同的执行者）执行。此对象表示 DAG 的一个版本，并成为 BackfillJob 执行的真实来源。pickle 是本机 python 序列化对象，在这种情况下，在作业持续时间内存储在数据库中。

执行程序获取 DagPickle id 并从数据库中读取 dag 定义。

class airflow.models.DagRun（** kwargs）

基类：sqlalchemy.ext.declarative.api.Base，airflow.utils.log.logging_mixin.LoggingMixin

DagRun 描述了 Dag 的一个实例。它可以由调度程序（用于常规运行）或外部触发器创建

静态查找（* args，** kwargs）

返回给定搜索条件的一组 dag 运行。

参数：

- dag_id(_integer __，_ list) - 用于查找 dag 的 dag_id
- run_id(string) - 定义此 dag 运行的运行 ID
- execution_date(datetime) - 执行日期
- 州(_ 州 _) - dag run 的状态
- external_trigger(bool) - 这个 dag run 是否是外部触发的
- no_backfills - 返回无回填（True），全部返回（False）。

默认为 False：键入 no_backfills：bool：param session：数据库会话：类型会话：会话

get_dag（）

返回与此 DagRun 关联的 Dag。

返回值： DAG

classmethod get_latest_runs（** kwargs）

返回每个 DAG 的最新 DagRun。

get_previous_dagrun（** kwargs）

以前的 DagRun，如果有的话

get_previous_scheduled_dagrun（** kwargs）

如果有的话，以前的 SCHEDULED DagRun

static get_run（session，dag_id，execution_date）

参数：

- dag_id(_unicode_) - DAG ID
- execution_date(datetime) - 执行日期

返回值： DagRun 对应于给定的 dag_id 和执行日期 如果存在的话。否则没有。：rtype：DagRun

get_task_instance（** kwargs）

返回此 dag 运行的 task_id 指定的任务实例

参数：task_id - 任务 ID

get_task_instances（** kwargs）

返回此 dag 运行的任务实例

refresh_from_db（** kwargs）

从数据库重新加载当前 dagrun：param session：数据库会话

update_state（** kwargs）

根据 TaskInstances 的状态确定 DagRun 的整体状态。

返回值： 州

verify_integrity（** kwargs）

通过检查已删除的任务或尚未在数据库中的任务来验证 DagRun。如果需要，它会将状态设置为已删除或添加任务。

class airflow.models.DagStat（dag_id，state，count = 0，dirty = False）

基类： sqlalchemy.ext.declarative.api.Base

静态创建（* args，** kwargs）

为指定的 dag 创建缺少状态 stats 表

参数：

- dag_id - 用于创建统计数据的 dag 的 dag id
- 会话 - 数据库会话

返回值：

static set_dirty（* args，** kwargs）

参数：

- dag_id - 用于标记脏的 dag_id
- 会话 - 数据库会话

返回值：

静态更新（* args，** kwargs）

更新脏/不同步 dag 的统计信息

参数：

- dag_ids(list) - 要更新的 dag_ids
- dirty_only(bool) - 仅针对标记的脏更新，默认为 True
- session(_Session_) - 要使用的 db 会话

class airflow.models.ImportError（** kwargs）

基类： sqlalchemy.ext.declarative.api.Base

exception airflow.models.InvalidFernetToken

基类： exceptions.Exception

class airflow.models.KnownEvent（** kwargs）

基类： sqlalchemy.ext.declarative.api.Base

class airflow.models.KnownEventType（** kwargs）

基类： sqlalchemy.ext.declarative.api.Base

class airflow.models.KubeResourceVersion（** kwargs）

基类： sqlalchemy.ext.declarative.api.Base

class airflow.models.KubeWorkerIdentifier（** kwargs）

基类： sqlalchemy.ext.declarative.api.Base

class airflow.models.Log（event，task_instance，owner = None，extra = None，** kwargs）

基类： sqlalchemy.ext.declarative.api.Base

用于主动将事件记录到数据库

class airflow.models.NullFernet

基类： future.types.newobject.newobject

“Null”加密器类，不加密或解密，但提供与 Fernet 类似的接口。

这样做的目的是使其余代码不必知道差异,并且只显示消息一次,而不是在运行气流 initdb时显示 20 次。

class airflow.models.Pool（** kwargs）

基类： sqlalchemy.ext.declarative.api.Base

open_slots（** kwargs）

返回此刻打开的插槽数

queued_slots（** kwargs）

返回此刻使用的插槽数

used_slots（** kwargs）

返回此刻使用的插槽数

class airflow.models.SlaMiss（** kwargs）

基类： sqlalchemy.ext.declarative.api.Base

存储已错过的 SLA 历史的模型。它用于跟踪 SLA 故障，并避免双重触发警报电子邮件。

class airflow.models.TaskFail（task，execution_date，start_date，end_date）

基类： sqlalchemy.ext.declarative.api.Base

TaskFail 跟踪每个任务实例的失败运行持续时间。

class airflow.models.TaskInstance（task，execution_date，state = None）

基类：sqlalchemy.ext.declarative.api.Base，airflow.utils.log.logging_mixin.LoggingMixin

任务实例存储任务实例的状态。该表是围绕已运行任务及其所处状态的权威和单一事实来源。

SqlAlchemy 模型没有任务或 dag 模型的 SqlAlchemy 外键故意对事务进行更多控制。

此表上的数据库事务应该确保双触发器以及围绕哪些任务实例准备好运行的任何混淆，即使多个调度程序可能正在触发任务实例。

are_dependencies_met（** kwargs）

返回在给定依赖关系的上下文的情况下是否满足所有条件以运行此任务实例（例如，从 UI 强制运行的任务实例将忽略某些依赖关系）。

参数：

- dep_context(_DepContext_) - 确定应评估的依赖项的执行上下文。
- session(_Session_) - 数据库会话
- verbose(_boolean_) - 是否在信息或调试日志级别上的失败依赖项的日志详细信息

are_dependents_done（** kwargs）

检查此任务实例的依赖项是否都已成功。这应该由 wait_for_downstream 使用。

当您不希望在完成依赖项之前开始处理任务的下一个计划时，这非常有用。例如，如果任务 DROPs 并重新创建表。

clear_xcom_data（** kwargs）

从数据库中清除任务实例的所有 XCom 数据

command（mark_success = False，ignore_all_deps = False，ignore_depends_on_past = False，ignore_task_deps = False，ignore_ti_state = False，local = False，pickle_id = None，raw = False，job_id = None，pool = None，cfg_path = None）

返回可在安装气流的任何位置执行的命令。此命令是 orchestrator 发送给执行程序的消息的一部分。

command_as_list（mark_success = False，ignore_all_deps = False，ignore_task_deps = False，ignore_depends_on_past = False，ignore_ti_state = False，local = False，pickle_id = None，raw = False，job_id = None，pool = None，cfg_path = None）

返回可在安装气流的任何位置执行的命令。此命令是 orchestrator 发送给执行程序的消息的一部分。

current_state（** kwargs）

从数据库中获取最新状态，如果会话通过，我们使用并查找状态将成为会话的一部分，否则将使用新会话。

误差（** kwargs）

在数据库中强制将任务实例的状态设置为 FAILED。

static generate_command（dag_id，task_id，execution_date，mark_success = False，ignore_all_deps = False，ignore_depends_on_past = False，ignore_task_deps = False，ignore_ti_state = False，local = False，pickle_id = None，file_path = None，raw = False，job_id = None，pool =无，cfg_path =无）

生成执行此任务实例所需的 shell 命令。

参数：

- dag_id(_unicode_) - DAG ID
- task_id(_unicode_) - 任务 ID
- execution_date(datetime) - 任务的执行日期
- mark_success(bool) - 是否将任务标记为成功
- ignore_all_deps(_boolean_) - 忽略所有可忽略的依赖项。覆盖其他 ignore_ *参数。
- ignore_depends_on_past(_boolean_) - 忽略 DAG 的 depends_on_past 参数（例如，对于回填）
- ignore_task_deps(_boolean_) - 忽略任务特定的依赖项，例如 depends_on_past 和触发器规则
- ignore_ti_state(_boolean_) - 忽略任务实例之前的失败/成功
- local(bool) - 是否在本地运行任务
- pickle_id(_unicode_) - 如果 DAG 序列化到 DB，则与酸洗 DAG 关联的 ID
- file_path - 包含 DAG 定义的文件的路径
- 原始 - 原始模式（需要更多细节）
- job_id - 工作 ID（需要更多细节）

- pool(_unicode_) - 任务应在其中运行的 Airflow 池
- cfg_path(_basestring_) - 配置文件的路径

返回值： shell 命令，可用于运行任务实例

`get_dagrun（** kwargs）`

返回此 TaskInstance 的 DagRun

参数：会议 -返回值： DagRun

`init_on_load（）`

初始化未存储在 DB 中的属性。

`init_run_context（原始=假）`

设置日志上下文。

`is_eligible_to_retry（）`

任务实例是否有资格重试

`is_premature`

返回任务是否处于 UP_FOR_RETRY 状态且其重试间隔是否已过去。

`key`
返回唯一标识任务实例的元组

`next_retry_datetime（）`

如果任务实例失败，则获取下次重试的日期时间。对于指数退避，retry_delay 用作基数并将转换为秒。

`pool_full（** kwargs）`

返回一个布尔值，指示槽池是否有空间运行此任务

`previous_ti`

在此任务实例之前运行的任务的任务实例

`ready_for_retry（）`

检查任务实例是否处于正确状态和重试时间范围。

refresh_from_db（** kwargs）

根据主键刷新数据库中的任务实例

参数：lock_for_update - 如果为 True，则表示数据库应锁定 TaskInstance（发出 FOR UPDATE 子句），直到提交会话为止。

try_number

返回该任务编号在实际运行时的尝试编号。

如果 TI 当前正在运行，这将匹配数据库中的列，在所有其他情况下，这将是增量的

xcom_pull（task_ids = None，dag_id = None，key = u'return_value'，include_prior_dates = False）

拉 XComs 可选择满足特定条件。

key 的默认值将搜索限制为由其他任务返回的 XComs（而不是手动推送的那些）。要删除此过滤器，请传递 key = None（或任何所需的值）。

如果提供了单个 task_id 字符串，则结果是该 task_id 中最新匹配的 XCom 的值。如果提供了多个 task_id，则返回匹配值的元组。无法找到匹配项时返回 None。

参数：

- key(string) - XCom 的密钥。如果提供，将仅返回具有匹配键的 XCom。默认键是'return_value'，也可以作为常量 XCOM_RETURN_KEY 使用。此键自动提供给任务返回的 XComs（而不是手动推送）。要删除过滤器，请传递 key = None。
- task_ids(string 或 _ 可迭代的字符串 _ _（_ _ 表示 task_ids __）_) - 仅提取具有匹配 ID 的任务的 XCom。可以通过 None 删除过滤器。
- dag_id(string) - 如果提供，则仅从此 DAG 中提取 XComs。如果为 None（默认值），则使用调用任务的 DAG。
- include_prior_dates(bool) - 如果为 False，则仅返回当前 execution_date 中的 XComs。如果为 True，则返回之前日期的 XComs。

xcom_push（key，value，execution_date = None）

使 XCom 可用于执行任务。

参数：

- key(string) - XCom 的密钥

- value(_ 任何 pickleable 对象 _) - XCom 的值。该值被 pickle 并存储在数据库中。
- execution_date(datetime) - 如果提供，XCom 将在此日期之前不可见。例如，这可以用于在将来的日期将消息发送到任务，而不会立即显示。

class airflow.models.User（** kwargs）

基类： sqlalchemy.ext.declarative.api.Base

class airflow.models.Variable（** kwargs）

基类：sqlalchemy.ext.declarative.api.Base，airflow.utils.log.logging_mixin.LoggingMixin

classmethod setdefault（key，default，deserialize_json = False）

与 Python 内置的 dict 对象一样，setdefault 返回键的当前值，如果不存在，则存储默认值并返回它。

参数：

- key(string) - 此变量的 Dict 键
- default - 要设置的默认值，如果变量则返回

不在 DB 中：type default：Mixed：param deserialize_json：将其存储为 DB 中的 JSON 编码值

并在检索值时取消编码

返回值： 杂

class airflow.models.XCom（** kwargs）

基类：sqlalchemy.ext.declarative.api.Base，airflow.utils.log.logging_mixin.LoggingMixin

XCom 对象的基类。

classmethod get_many（** kwargs）

检索 XCom 值，可选地满足某些条件 TODO：“pickling”已被弃用，JSON 是首选。

Airflow 2.0 中将删除“酸洗”。

classmethod get_one（** kwargs）

检索 XCom 值，可选择满足特定条件。TODO：“pickling”已被弃用，JSON 是首选。

Airflow 2.0 中将删除“酸洗”。

返回值： XCom 值

classmethod set（** kwargs）

存储 XCom 值。TODO:“pickling”已被弃用，JSON 是首选。

Airflow 2.0 中将删除“酸洗”。

返回值： 没有

airflow.models.clear_task_instances（tis，session，activate_dag_runs = True，dag = None）

清除一组任务实例，但确保正在运行的任务实例被杀死。

参数：

- tis - 任务实例列表
- 会话 - 当前会话
- activate_dag_runs - 用于检查活动 dag 运行的标志
- dag - DAG 对象

airflow.models.get_fernet（）

Fernet 密钥的延期负载。

此功能可能因为未安装加密或因为 Fernet 密钥无效而失败。

返回值： Fernet 对象| 举： | 如果尝试加载 Fernet 时出现问题，则会出现 AirflowException

钩

钩子是外部平台和数据库的接口，在可能的情况下实现通用接口，并充当操作员的构建块。

class airflow.hooks.dbapi_hook.DbApiHook（* args，** kwargs）

基类： airflow.hooks.base_hook.BaseHook

sql hooks 的抽象基类。

bulk_dump（table，tmp_file）

将数据库表转储到制表符分隔的文件中

参数：

- table(str) - 源表的名称
- tmp_file(str) - 目标文件的路径

bulk_load（table，tmp_file）

将制表符分隔的文件加载到数据库表中

参数：

- table(str) - 目标表的名称
- tmp_file(str) - 要加载到表中的文件的路径

get_autocommit（康涅狄格州）

获取提供的连接的自动提交设置。如果 conn.autocommit 设置为 True，则返回 True。如果未设置 conn.autocommit 或将其设置为 False 或 conn 不支持自动提交，则返回 False。：param conn：从中获取自动提交设置的连接。：type conn：连接对象。：return：连接自动提交设置。：rtype 布尔。

get_conn（）

返回一个连接对象

get_cursor（）

返回一个游标

get_first（sql，parameters = None）

执行 sql 并返回第一个结果行。

参数：

- sql(_str _ 或 list) - 要执行的 sql 语句（str）或要执行的 sql 语句列表
- 参数(_mapping _ 或 _iterable_) - 用于呈现 SQL 查询的参数。

get_pandas_df（sql，parameters = None）

执行 sql 并返回一个 pandas 数据帧

参数：

- sql(_str _ 或 list) - 要执行的 sql 语句（str）或要执行的 sql 语句列表

- 参数(_mapping _ 或 _iterable_) - 用于呈现 SQL 查询的参数。

`get_records（sql，parameters = None）`

执行 sql 并返回一组记录。

参数：

- sql(_str _ 或 list) - 要执行的 sql 语句（str）或要执行的 sql 语句列表
- 参数(_mapping _ 或 _iterable_) - 用于呈现 SQL 查询的参数。

`insert_rows（table，rows，target_fields = None，commit_every = 1000，replace = False）`

将一组元组插入表中的通用方法是，每个 commit_every 行都会创建一个新事务

参数：

- table(str) - 目标表的名称
- rows(_ 可迭代的元组 _) - 要插入表中的行
- target_fields(_ 可迭代的字符串 _) - 要填充表的列的名称
- commit_every(int) - 要在一个事务中插入的最大行数。设置为 0 以在一个事务中插入所有行。
- replace(bool) - 是否替换而不是插入

`run（sql，autocommit = False，parameters = None）`

运行命令或命令列表。将 sql 语句列表传递给 sql 参数，以使它们按顺序执行

参数：

- sql(_str _ 或 list) - 要执行的 sql 语句（str）或要执行的 sql 语句列表
- autocommit(bool) - 在执行查询之前将连接的自动提交设置设置为什么。
- 参数(_mapping _ 或 _iterable_) - 用于呈现 SQL 查询的参数。

`set_autocommit（conn，autocommit）`

设置连接上的自动提交标志

`class airflow.hooks.docker_hook.DockerHook（docker_conn_id='docker_default'，base_url = None，version = None，tls = None）`

基类：airflow.hooks.base_hook.BaseHook，airflow.utils.log.logging_mixin.LoggingMixin

与私有 Docker 注册表交互。

参数：docker_conn_id(str) - 存储凭证和额外配置的 Airflow 连接的 ID

class airflow.hooks.hive_hooks.HiveCliHook（hive_cli_conn_id = u'hive_cli_default'，run_as = None，mapred_queue = None，mapred_queue_priority = None，mapred_job_name = None）

基类： airflow.hooks.base_hook.BaseHook

围绕 hive CLI 的简单包装。

它还支持 beeline 运行 JDBC 的轻量级 CLI，并取代较重的传统 CLI。要启用 beeline，请在连接的额外字段中设置 use_beeline 参数，如下所示{ "use_beeline": true }

请注意，您还 hive_cli_params 可以使用要在连接中使用的默认配置单元 CLI 参数，因为在此处传递的参数可以被 run_cli 的 hive_conf 参数覆盖。{"hive_cli_params": "-hiveconf mapred.job.tracker=some.jobtracker:444"}

额外的连接参数 auth 将按原样在 jdbc 连接字符串中传递。

参数：

- mapred_queue(string) - Hadoop 调度程序使用的队列（容量或公平）
- mapred_queue_priority(string) - 作业队列中的优先级。可能的设置包括:VERY_HIGH,HIGH,NORMAL,LOW, VERY_LOW
- mapred_job_name(string) - 此名称将出现在 jobtracker 中。这可以使监控更容易。

load_df（df，table，field_dict = None，delimiter = u'，'，encoding = u'utf8'，pandas_kwargs = None，** kwargs）
将 pandas DataFrame 加载到配置单元中。

如果未传递 Hive 数据类型，则会推断 Hive 数据类型，但不会清理列名称。

参数：

- df(_DataFrame_) - 要加载到 Hive 表中的 DataFrame
- table(str) - 目标 Hive 表，使用点表示法来定位特定数据库
- field_dict(_OrderedDict_) - 从列名映射到 hive 数据类型。请注意，它必须是 OrderedDict 才能保持列的顺序。
- delimiter(str) - 文件中的字段分隔符
- encoding(str) - 将 DataFrame 写入文件时使用的字符串编码
- pandas_kwargs(dict) - 传递给 DataFrame.to_csv
- kwargs - 传递给 self.load_file

load_file（filepath，table，delimiter = u'，'，field_dict = None，create = True，overwrite = True，partition = None，recreate = False，tblproperties = None）

将本地文件加载到 Hive 中

请注意,在 Hive 中生成的表使用的不是最有效的序列化格式。如果加载了大量数据和/或表格被大量查询,您可能只想使用此运算符将数据暂存到临时表中，然后使用 a 将其加载到最终目标中。STORED AS textfile``HiveOperator

参数：

- filepath(str) - 要加载的文件的本地文件路径
- table(str) - 目标 Hive 表，使用点表示法来定位特定数据库
- delimiter(str) - 文件中的字段分隔符
- field_dict(_OrderedDict_) - 文件中字段的字典名称作为键，其 Hive 类型作为值。请注意，它必须是 OrderedDict 才能保持列的顺序。
- create(bool) - 是否创建表，如果它不存在
- overwrite(bool) - 是否覆盖表或分区中的数据
- partition(dict) - 将目标分区作为分区列和值的字典
- recreate(bool) - 是否在每次执行时删除并重新创建表
- tblproperties(dict) - 正在创建的 hive 表的 TBLPROPERTIES

```
run_cli（hql，schema = None，verbose = True，hive_conf = None）
```

使用 hive cli 运行 hql 语句。如果指定了 hive_conf，它应该是一个 dict，并且条目将在 HiveConf 中设置为键/值对

参数：hive_conf(dict) - 如果指定，这些键值对将作为传递给 hive 。请注意，它们将在之后传递，因此将覆盖数据库中指定的任何值。-hiveconf "key"="value"``hive_cli_params

```
 >>>  hh = HiveCliHook ()
>>>  result = hh . run_cli ( "USE airflow;" )
>>>  ( "OK" in result )
True
```

```
test_hql（HQL）
```

使用 hive cli 和 EXPLAIN 测试 hql 语句

```
class airflow.hooks.hive_hooks.HiveMetastoreHook（metastore_conn_id = u'metastore_default'）
```

基类： airflow.hooks.base_hook.BaseHook

包装器与 Hive Metastore 交互

```
check_for_named_partition（schema，table，partition_name）
```

检查是否存在具有给定名称的分区

参数：

- schema(string) - @table 所属的 hive 模式（数据库）的名称
- table - @partition 所属的配置单元表的名称

| 划分： | 要检查的分区的名称（例如 a = b / c = d）| 返回类型： | 布尔

```
>>> hh = HiveMetastoreHook()
>>> t = 'static_babynames_partitioned'
>>> hh.check_for_named_partition('airflow', t, "ds=2015-01-01")
True
>>> hh.check_for_named_partition('airflow', t, "ds=xxx")
False
```

check_for_partition（架构，表，分区）

检查分区是否存在

参数：

- schema(string) - @table 所属的 hive 模式（数据库）的名称
- table - @partition 所属的配置单元表的名称

| 划分： | 与要检查的分区匹配的表达式（例如 a ='b'和 c ='d'）| 返回类型： | 布尔

```
>>> hh = HiveMetastoreHook()
>>> t = 'static_babynames_partitioned'
>>> hh.check_for_partition('airflow', t, "ds='2015-01-01'")
True
```

get_databases（图案= U '*'）

获取 Metastore 表对象

get_metastore_client（）

返回一个 Hive thrift 客户端。

get_partitions（schema，table_name，filter = None）

返回表中所有分区的列表。仅适用于小于 32767（java short max val）的表。对于子分区表，该数字可

能很容易超过此值。

```
>>> hh = HiveMetastoreHook()
>>> t = 'static_babynames_partitioned'
>>> parts = hh.get_partitions(schema='airflow', table_name=t)
>>> len(parts)
1
>>> parts
[{'ds': '2015-01-01'}]
```

```
get_table(table_name, db='default')
```

获取 Metastore 表对象

```
>>> hh = HiveMetastoreHook()
>>> t = hh.get_table(db='airflow', table_name='static_babynames')
>>> t.tableName
'static_babynames'
>>> [col.name for col in t.sd.cols]
['state', 'year', 'name', 'gender', 'num']
```

```
get_tables(db, pattern='*')
```

获取 Metastore 表对象

```
max_partition(schema, table_name, field=None, filter_map=None)
```

返回表中给定字段的所有分区的最大值。如果表中只存在一个分区键，则该键将用作字段。filter_map 应该是 partition_key：partition_value map，将用于过滤掉分区。

参数：

- schema(string) - 模式名称。
- table_name(string) - 表名。
- field(string) - 从中获取最大分区的分区键。
- filter_map(_map_) - partition_key：用于分区过滤的 partition_value 映射。

```
>>> hh = HiveMetastoreHook()
>>> filter_map = {'ds': '2015-01-01', 'ds': '2014-01-01'}
>>> t = 'static_babynames_partitioned'
>>> hh.max_partition(schema='airflow',
...     table_name=t, field='ds', filter_map=filter_map)
'2015-01-01'
```

```
table_exists（table_name，db = u'default'）
```

检查表是否存在

```
 >>>  hh = HiveMetastoreHook ()
>>>  hh . table_exists ( db = 'airflow' , table_name = 'static_babynames' )
True
>>>  hh . table_exists ( db = 'airflow' , table_name = 'does_not_exist' )
False
```

```
class airflow.hooks.hive_hooks.HiveServer2Hook（hiveserver2_conn_id = u'hiveserver2_default'）
```

基类： airflow.hooks.base_hook.BaseHook

pyhive 库周围的包装

请注意，默认的 authMechanism 是 PLAIN，要覆盖它，您可以 extra 在 UI 中的连接中指定它，如在

```
get_pandas_df（hql，schema = u'default'）
```

从 Hive 查询中获取 pandas 数据帧

```
 >>>  hh = HiveServer2Hook ()
>>>  sql = "SELECT * FROM airflow.static_babynames LIMIT 100"
>>>  df = hh . get_pandas_df ( sql )
>>>  len ( df . index )
100
```

```
get_records（hql，schema = u'default'）
```

从 Hive 查询中获取一组记录。

```
 >>>  hh = HiveServer2Hook ()
>>>  sql = "SELECT * FROM airflow.static_babynames LIMIT 100"
>>>  len ( hh . get_records ( sql ))
100
```

```
class airflow.hooks.http_hook.HttpHook（method ='POST'，http_conn_id ='http_default'）
```

基类： airflow.hooks.base_hook.BaseHook

与 HTTP 服务器交互。：param http_conn_id：具有基本 API 网址的连接，即 https://www.google.com/

和可选的身份验证凭据 也可以在 Json 格式的 Extra 字段中指定默认标头。

参数：method(str) - 要调用的 API 方法

check_response（响应）

检查状态代码并在非 2XX 或 3XX 状态代码上引发 AirflowException 异常：param response：请求响应对象：type response：requests.response

get_conn（头=无）

返回用于请求的 http 会话：param headers：要作为字典传递的其他标头：type headers：dict

run（endpoint，data = None，headers = None，extra_options = None）

执行请求：param endpoint：要调用的端点，即 resource / v1 / query？：type endpoint：str：param data：要上载的有效负载或请求参数：type data：dict：param headers：要作为字典传递的其他头：type headers：dict：param extra_options：执行时要使用的其他选项请求

ie {'check_response'：False}以避免在非 2XX 或 3XX 状态代码上检查引发异常

run_and_check（session，prepped_request，extra_options）

获取超时等额外选项并实际运行请求，检查结果：param session：用于执行请求的会话：type session：requests.Session：param prepped_request：在 run（）中生成的准备好的请求：type prepped_request ：session.prepare_request：param extra_options：执行请求时要使用的其他选项

ie {'check_response'：False}以避免在非 2XX 或 3XX 状态代码上检查引发异常

run_with_advanced_retry（_retry_args，* args，** kwargs）

运行 Hook.run（）并附加一个 Tenacity 装饰器。这对于可能受间歇性问题干扰并且不应立即失效的连接器非常有用。：param retryargs：定义重试行为的参数。

请参阅 https://github.com/jd/tenacity 上的 Tenacity 文档

示例：::

hook = HttpHook（http_conn_id ='my_conn'，method ='GET'）retry_args = dict（

> wait = tenacity.wait_exponential（），stop = tenacity.stop_after_attempt（10），retry = requests.exceptions.ConnectionError

）hook.run_with_advanced_retry（

> > endpoint ='v1 / test'，_ retry_args = retry_args > > ）

class airflow.hooks.druid_hook.DruidDbApiHook（* args，** kwargs）

基类： airflow.hooks.dbapi_hook.DbApiHook

与德鲁伊经纪人互动

这个钩子纯粹是供用户查询德鲁伊经纪人。摄取，请使用 druidHook。

get_conn（）

建立与德鲁伊经纪人的联系。

get_pandas_df（sql，parameters = None）

执行 sql 并返回一个 pandas 数据帧

参数：

- sql(_str _ 或 list) - 要执行的 sql 语句（str）或要执行的 sql 语句列表
- 参数(_mapping _ 或 _iterable_) - 用于呈现 SQL 查询的参数。

get_uri（）

获取德鲁伊经纪人的连接 uri。

例如：druid：// localhost：8082 / druid / v2 / sql /

insert_rows（table，rows，target_fields = None，commit_every = 1000）

将一组元组插入表中的通用方法是，每个 commit_every 行都会创建一个新事务

参数：

- table(str) - 目标表的名称
- rows(_ 可迭代的元组 _) - 要插入表中的行
- target_fields(_ 可迭代的字符串 _) - 要填充表的列的名称
- commit_every(int) - 要在一个事务中插入的最大行数。设置为 0 以在一个事务中插入所有行。
- replace(bool) - 是否替换而不是插入

set_autocommit（conn，autocommit）

设置连接上的自动提交标志

class airflow.hooks.druid_hook.DruidHook(druid_ingest_conn_id='druid_ingest_default',timeout = 1, max_ingestion_time = None)

基类： airflow.hooks.base_hook.BaseHook

与德鲁伊霸主的联系以供摄取

参数：

- druid_ingest_conn_id(string) - 接受索引作业的德鲁伊霸王机器的连接 ID
- timeout(int) - 轮询 Druid 作业以获取摄取作业状态之间的间隔
- max_ingestion_time(int) - 假定作业失败前的最长摄取时间

class airflow.hooks.hdfs_hook.HDFSHook（hdfs_conn_id ='hdfs_default'， proxy_user = None，autoconfig = False）

基类： airflow.hooks.base_hook.BaseHook

与 HDFS 互动。这个类是 snakebite 库的包装器。

参数：

- hdfs_conn_id - 用于获取连接信息的连接 ID
- proxy_user(string) - HDFS 操作的有效用户
- autoconfig(bool) - 使用 snakebite 自动配置的客户端

get_conn（）

返回一个 snakebite HDFSClient 对象。

class airflow.hooks.jdbc_hook.JdbcHook（* args，** kwargs）
基类： airflow.hooks.dbapi_hook.DbApiHook

jdbc db 访问的常规挂钩。

JDBC URL，用户名和密码将取自预定义的连接。请注意，必须在 DB 中的“host”字段中指定整个 JDBC URL。如果给定的连接 ID 不存在，则引发气流错误。

get_conn（）

返回一个连接对象

set_autocommit（conn，autocommit）

启用或禁用给定连接的自动提交。

参数：conn - 连接返回值：

class airflow.hooks.mssql_hook.MsSqlHook（* args，** kwargs）

基类： airflow.hooks.dbapi_hook.DbApiHook

与 Microsoft SQL Server 交互。

get_conn（）

返回一个 mssql 连接对象

set_autocommit（conn，autocommit）

设置连接上的自动提交标志

class airflow.hooks.mysql_hook.MySqlHook（* args，** kwargs）

基类： airflow.hooks.dbapi_hook.DbApiHook

与 MySQL 交互。

您可以在连接的额外字段中指定 charset 。您也可以选择光标。有关更多详细信息，请参阅 MySQLdb.cursors。{"charset": "utf8"}``{"cursor": "SSCursor"}

bulk_dump（table，tmp_file）

将数据库表转储到制表符分隔的文件中

bulk_load（table，tmp_file）

将制表符分隔的文件加载到数据库表中

get_autocommit（康涅狄格州）

MySql 连接以不同的方式进行自动提交。：param conn：连接以获取自动提交设置。：type conn：连接对象。：return：connection autocommit setting：rtype bool

get_conn（）

返回一个 mysql 连接对象

set_autocommit（conn，autocommit）

MySql 连接以不同的方式设置自动提交。

class airflow.hooks.oracle_hook.OracleHook（* args，** kwargs）

基类： airflow.hooks.dbapi_hook.DbApiHook

与 Oracle SQL 交互。

bulk_insert_rows（table，rows，target_fields = None，commit_every = 5000）

cx_Oracle 的高性能批量插入，它通过 executemany（）使用预处理语句。为获得最佳性能，请将行作为迭代器传入。

get_conn（）

返回 oracle 连接对象使用自定义 DSN 连接的可选参数（而不是使用来自 tnsnames.ora 的服务器别名）dsn（数据源名称）是 TNS 条目（来自 Oracle 名称服务器或 tnsnames.ora 文件）或者是一个像 makedsn（）返回的字符串。

参数：

- dsn - Oracle 服务器的主机地址
- service_name - 要连接的数据库的 db_unique_name（TNS 的 CONNECT_DATA 部分）

您可以在连接的额外字段中设置这些参数，如 { "dsn":"some.host.address" ，"service_name":"some.service.name" }

insert_rows（table，rows，target_fields = None，commit_every = 1000）

将一组元组插入表中的通用方法，整个插入集被视为一个事务从标准 DbApiHook 实现更改： - cx_Oracle 中的 Oracle SQL 查询不能以分号（';'）终止 - 替换 NaN 使用 numpy.nan_to_num 的 NULL 值（不使用 is_nan（）因为字符串的输入类型错误）

在插入期间将日期时间单元格强制转换为 Oracle DATETIME 格式

class airflow.hooks.pig_hook.PigCliHook（pig_cli_conn_id ='pig_cli_default'）

基类： airflow.hooks.base_hook.BaseHook

猪 CLI 的简单包装。

请注意，您还 pig_properties 可以使用要在连接中使用的默认 pig CLI 属性来设置{"pig_properties":

"-Dpig.tmpfilecompression=true"}

```
run_cli（pig，verbose = True）
```

使用猪 cli 运行猪脚本

```
 >>>  ph = PigCliHook ()
>>>  result = ph . run_cli ( "ls /;" )
>>>  ( "hdfs://" in result )
True
```

```
class airflow.hooks.postgres_hook.PostgresHook（* args，** kwargs）
```

基类： airflow.hooks.dbapi_hook.DbApiHook

与 Postgres 互动。您可以在连接的额外字段中指定 ssl 参数。{"sslmode": "require", "sslcert": "/path/to/cert.pem", etc}

注意：对于 Redshift，请在额外的连接参数中使用 keepalives_idle 并将其设置为小于 300 秒。

```
bulk_dump（table，tmp_file）
```

将数据库表转储到制表符分隔的文件中

```
bulk_load（table，tmp_file）
```

将制表符分隔的文件加载到数据库表中

```
copy_expert（sql，filename，open = <内置函数打开>）
```

使用 psycopg2 copy_expert 方法执行 SQL。必须在不访问超级用户的情况下执行 COPY 命令。

注意：如果使用“COPY FROM”语句调用此方法并且指定的输入文件不存在，则会创建一个空文件并且不会加载任何数据，但操作会成功。因此，如果用户想要知道输入文件何时不存在，他们必须自己检查它的存在。

```
get_conn（）
```

返回一个连接对象

```
class airflow.hooks.presto_hook.PrestoHook（* args，** kwargs）
```

基类： airflow.hooks.dbapi_hook.DbApiHook

通过 PyHive 与 Presto 互动!

```
 >>>  ph = PrestoHook ()
>>>  sql = "SELECT count(1) AS num FROM airflow.static_babynames"
>>>  ph . get_records ( sql )
[[340698]]
```

```
get_conn ()
```

返回一个连接对象

```
get_first (hql, parameters = None)
```

无论查询返回多少行，都只返回第一行。

```
get_pandas_df (hql, parameters = None)
```

从 sql 查询中获取 pandas 数据帧。

```
get_records (hql, parameters = None)
```

从 Presto 获取一组记录

```
insert_rows (table, rows, target_fields = None)
```

将一组元组插入表中的通用方法。

参数：

- table(str) - 目标表的名称
- rows(_ 可迭代的元组 _) - 要插入表中的行
- target_fields(_ 可迭代的字符串 _) - 要填充表的列的名称

```
run (hql, parameters = None)
```

执行针对 Presto 的声明。可用于创建视图。

```
class airflow.hooks.S3_hook.S3Hook (aws_conn_id ='aws_default')
```

基类： airflow.contrib.hooks.aws_hook.AwsHook

使用 boto3 库与 AWS S3 交互。

```
check_for_bucket (BUCKET_NAME)
```

检查 bucket_name 是否存在。

参数：bucket_name(str) - 存储桶的名称

```
check_for_key（key，bucket_name = None）
```

检查存储桶中是否存在密钥

参数：

- key(str) - 指向文件的 S3 键
- bucket_name(str) - 存储文件的存储桶的名称

```
check_for_prefix（bucket_name，prefix，delimiter）
```

检查存储桶中是否存在前缀

```
check_for_wildcard_key（wildcard_key，bucket_name = None，delimiter =''）
```

检查桶中是否存在与通配符表达式匹配的密钥

```
get_bucket（BUCKET_NAME）
```

返回 boto3.S3.Bucket 对象

参数：bucket_name(str) - 存储桶的名称

```
get_key（key，bucket_name = None）
```

返回 boto3.s3.Object

参数：

- key(str) - 密钥的路径
- bucket_name(str) - 存储桶的名称

```
get_wildcard_key（wildcard_key，bucket_name = None，delimiter =''）
```

返回与通配符表达式匹配的 boto3.s3.Object 对象

参数：

- wildcard_key(str) - 密钥的路径

- bucket_name(str) - 存储桶的名称

```
list_keys（bucket_name，prefix =''，delimiter =''，page_size = None，max_items = None）
```

列出前缀下的存储桶中的密钥，但不包含分隔符

参数：

- bucket_name(str) - 存储桶的名称
- prefix(str) - 一个密钥前缀
- delimiter(str) - 分隔符标记键层次结构。
- page_size(int) - 分页大小
- max_items(int) - 要返回的最大项目数

```
list_prefixes（bucket_name，prefix =''，delimiter =''，page_size = None，max_items = None）
```

列出前缀下的存储桶中的前缀

参数：

- bucket_name(str) - 存储桶的名称
- prefix(str) - 一个密钥前缀
- delimiter(str) - 分隔符标记键层次结构。
- page_size(int) - 分页大小
- max_items(int) - 要返回的最大项目数

```
load_bytes（bytes_data，key，bucket_name = None，replace = False，encrypt = False）
```

将字节加载到 S3

这是为了方便在 S3 中删除字符串。它使用 boto 基础结构将文件发送到 s3。

参数：

- bytes_data(_bytes_) - 设置为密钥内容的字节。
- key(str) - 指向文件的 S3 键
- bucket_name(str) - 存储文件的存储桶的名称
- replace(bool) - 一个标志，用于决定是否覆盖密钥（如果已存在）
- encrypt(bool) - 如果为 True，则文件将在服务器端由 S3 加密，并在 S3 中静止时以加密形式存储。

```
load_file（filename，key，bucket_name = None，replace = False，encrypt = False）
```

将本地文件加载到 S3

参数：

- filename(str) - 要加载的文件的名称。
- key(str) - 指向文件的 S3 键
- bucket_name(str) - 存储文件的存储桶的名称
- replace(bool) - 一个标志，用于决定是否覆盖密钥（如果已存在）。如果 replace 为 False 且密钥存在，则会引发错误。
- encrypt(bool) - 如果为 True，则文件将在服务器端由 S3 加密，并在 S3 中静止时以加密形式存储。

```
load_string（string_data，key，bucket_name = None，replace = False，encrypt = False，encoding ='utf-8'）
```

将字符串加载到 S3

这是为了方便在 S3 中删除字符串。它使用 boto 基础结构将文件发送到 s3。

参数：

- string_data(str) - 要设置为键的内容的字符串。
- key(str) - 指向文件的 S3 键
- bucket_name(str) - 存储文件的存储桶的名称
- replace(bool) - 一个标志，用于决定是否覆盖密钥（如果已存在）
- encrypt(bool) - 如果为 True，则文件将在服务器端由 S3 加密，并在 S3 中静止时以加密形式存储。

```
read_key（key，bucket_name = None）
```

从 S3 读取密钥

参数：

- key(str) - 指向文件的 S3 键
- bucket_name(str) - 存储文件的存储桶的名称

```
select_key(key,bucket_name = None,expression ='SELECT * FROM S3Object',expression_type ='SQL',input_serialization = {'CSV'：{}}，output_serialization = {'CSV'：{}}）
```

使用 S3 Select 读取密钥。

参数：

- key(str) - 指向文件的 S3 键

- bucket_name(str) - 存储文件的存储桶的名称
- expression(str) - S3 选择表达式
- expression_type(str) - S3 选择表达式类型
- input_serialization(dict) - S3 选择输入数据序列化格式
- output_serialization(dict) - S3 选择输出数据序列化格式

返回值： 通过 S3 Select 检索原始数据的子集| 返回类型： | 海峡 也可以看看

有关 S3 Select 参数的更多详细信息：
http：
//boto3.readthedocs.io/en/latest/reference/services/s3.html#S3.Client.selectobjectcontent

class airflow.hooks.samba_hook.SambaHook（samba_conn_id）

基类： airflow.hooks.base_hook.BaseHook

允许与 samba 服务器交互。

class airflow.hooks.slack_hook.SlackHook（token = None，slack_conn_id = None）

基类： airflow.hooks.base_hook.BaseHook

使用 slackclient 库与 Slack 交互。

class airflow.hooks.sqlite_hook.SqliteHook（* args，** kwargs）

基类： airflow.hooks.dbapi_hook.DbApiHook

与 SQLite 交互。

get_conn（）

返回一个 sqlite 连接对象

class airflow.hooks.webhdfs_hook.WebHDFSHook（webhdfs_conn_id ='webhdfs_default'，proxy_user = None）

基类： airflow.hooks.base_hook.BaseHook

与 HDFS 互动。这个类是 hdfscli 库的包装器。

check_for_path（hdfs_path）

通过查询 FileStatus 检查 HDFS 中是否存在路径。

get_conn（）

返回 hdfscli InsecureClient 对象。

load_file（source，destination，overwrite = True，parallelism = 1，** kwargs）

将文件上传到 HDFS

参数：

- source(str) - 文件或文件夹的本地路径。如果是文件夹，则会上传其中的所有文件（请注意，这意味着不会远程创建文件夹空文件夹）。
- destination(str) - PTarget HDFS 路径。如果它已经存在并且是目录，则文件将被上传到内部。
- overwrite(bool) - 覆盖任何现有文件或目录。
- parallelism(int) - 用于并行化的线程数。值 0（或负数）使用与文件一样多的线程。
- ** kwargs - 转发给的关键字参数 upload()。

class airflow.hooks.zendesk_hook.ZendeskHook（zendesk_conn_id）

基类： airflow.hooks.base_hook.BaseHook

与 Zendesk 交谈的钩子

call（path，query = None，get_all_pages = True，side_loading = False）

调用 Zendesk API 并返回结果

参数：

- path - 要调用的 Zendesk API
- query - 查询参数
- get_all_pages - 在返回之前累积所有页面的结果。由于严格的速率限制，这通常会超时。等待超时后尝试之间的建议时间段。
- side_loading - 作为单个请求的一部分检索相关记录。为了启用侧载，请添加一个'include'查询参数，其中包含要加载的以逗号分隔的资源列表。有关侧载的更多信息，请参阅 https://developer.zendesk.com/rest_api/docs/core/side_loading

社区贡献了钩子

class airflow.contrib.hooks.aws_dynamodb_hook.AwsDynamoDBHook（table_keys = None，table_name = None，region_name = None，* args，** kwargs）

基类： airflow.contrib.hooks.aws_hook.AwsHook

与 AWS DynamoDB 交互。

参数：

- table_keys(list) - 分区键和排序键
- table_name(str) - 目标 DynamoDB 表
- region_name(str) - aws 区域名称（例如：us-east-1）

write_batch_data（项目）

将批处理项目写入 dynamodb 表，并提供整个容量。

class airflow.contrib.hooks.aws_hook.AwsHook（aws_conn_id ='aws_default'）

基类： airflow.hooks.base_hook.BaseHook

与 AWS 互动。这个类是 boto3 python 库的一个瘦包装器。

get_credentials（REGION_NAME =无）

获取底层的 botocore.Credentials 对象。它包含以下属性：access_key，secret_key 和 token。

get_session（REGION_NAME =无）

获取底层的 boto3.session。

class airflow.contrib.hooks.aws_lambda_hook.AwsLambdaHook（function_name，region_name = None，log_type ='None'，qualifier ='$ LATEST'，invocation_type ='RequestResponse'，* args，** kwargs）

基类： airflow.contrib.hooks.aws_hook.AwsHook

与 AWS Lambda 互动

参数：

- function_name(str) - AWS Lambda 函数名称
- region_name(str) - AWS 区域名称（例如：us-west-2）
- log_type(str) - 尾调用请求
- qualifier(str) - AWS Lambda 函数版本或别名
- invocation_type(str) - AWS Lambda 调用类型（RequestResponse，Event 等）

invoke_lambda（有效载荷）

调用 Lambda 函数

class airflow.contrib.hooks.azure_data_lake_hook.AzureDataLakeHook（azure_data_lake_conn_id='azure_data_lake_default'）

基类： airflow.hooks.base_hook.BaseHook

与 Azure Data Lake 进行交互。

客户端 ID 和客户端密钥应该在用户和密码参数中。租户和帐户名称应为{“租户”:“<TENANT>”,“account_name”:“ACCOUNT_NAME”}的额外字段。

参数：azure_data_lake_conn_id(str) - 对 Azure Data Lake 连接的引用。

check_for_file（FILE_PATH）

检查 Azure Data Lake 上是否存在文件。

参数：file_path(str) - 文件的路径和名称。返回值： 如果文件存在则为 True，否则为 False。 ：rtype 布尔

download_file（local_path，remote_path，nthreads = 64，overwrite = True，buffersize = 4194304，blocksize = 4194304）

从 Azure Blob 存储下载文件。

参数：

- local_path(str) - 本地路径。如果下载单个文件，将写入此特定文件，除非它是现有目录，在这种情况下，将在其中创建文件。如果下载多个文件，这是要写入的根目录。将根据需要创建目录。
- remote_path(str) - 用于查找远程文件的远程路径/ globstring。不支持使用**的递归 glob 模式。
- nthreads(int) - 要使用的线程数。如果为 None，则使用核心数。
- overwrite(bool) - 是否强制覆盖现有文件/目录。如果 False 和远程路径是目录，则无论是否覆盖任何文件都将退出。如果为 True，则实际仅覆盖匹配的文件名。
- buffersize(int) - int [2 ** 22]内部缓冲区的字节数。此块不能大于块，并且不能小于块。
- blocksize(int) - int [2 ** 22]块的字节数。在每个块中，我们为每个 API 调用编写一个较小的块。这个块不能大于块。

get_conn（）

返回 AzureDLFileSystem 对象。

upload_file（local_path，remote_path，nthreads = 64，overwrite = True，buffersize = 4194304，blocksize = 4194304）

将文件上载到 Azure Data Lake。

参数：

- local_path(str) - 本地路径。可以是单个文件，目录（在这种情况下，递归上传）或 glob 模式。不支持使用**的递归 glob 模式。
- remote_path(str) - 要上传的远程路径; 如果有多个文件，这就是要写入的 dircetory 根目录。
- nthreads(int) - 要使用的线程数。如果为 None，则使用核心数。
- overwrite(bool) - 是否强制覆盖现有文件/目录。如果 False 和远程路径是目录，则无论是否覆盖任何文件都将退出。如果为 True，则实际仅覆盖匹配的文件名。
- buffersize(int) - int [2 ** 22]内部缓冲区的字节数。此块不能大于块，并且不能小于块。
- blocksize(int) - int [2 ** 22]块的字节数。在每个块中，我们为每个 API 调用编写一个较小的块。这个块不能大于块。

```
class airflow.contrib.hooks.azure_fileshare_hook.AzureFileShareHook ( wasb_conn_id ='wasb_default')
```

基类： airflow.hooks.base_hook.BaseHook

与 Azure FileShare 存储交互。

在连接的“额外”字段中传递的其他选项将传递给 FileService（）构造函数。

参数：wasb_conn_id(str) - 对 wasb 连接的引用。

```
check_for_directory（share_name，directory_name，** kwargs）
```

检查 Azure 文件共享上是否存在目录。

参数：

- share_name(str) - 共享的名称。
- directory_name(str) - 目录的名称。
- kwargs(object) - FileService.exists（）采用的可选关键字参数。

返回值： 如果文件存在则为 True，否则为 False。 ：rtype 布尔

```
check_for_file（share_name，directory_name，file_name，** kwargs）
```

检查 Azure 文件共享上是否存在文件。

参数：

- share_name(str) - 共享的名称。

- directory_name(str) - 目录的名称。
- file_name(str) - 文件名。
- kwargs(object) - FileService.exists（）采用的可选关键字参数。

返回值： 如果文件存在则为 True，否则为 False。 ：rtype 布尔

create_directory（share_name，directory_name，** kwargs）

在 Azure 文件共享上创建新的目标。

参数：

- share_name(str) - 共享的名称。
- directory_name(str) - 目录的名称。
- kwargs(object) - FileService.create_directory（）采用的可选关键字参数。

返回值： 文件和目录列表 ：rtype 列表

get_conn（）

返回 FileService 对象。

get_file（file_path，share_name，directory_name，file_name，** kwargs）

从 Azure 文件共享下载文件。

参数：

- file_path(str) - 存储文件的位置。
- share_name(str) - 共享的名称。
- directory_name(str) - 目录的名称。
- file_name(str) - 文件名。
- kwargs(object) - FileService.get_file_to_path（）采用的可选关键字参数。

get_file_to_stream（stream，share_name，directory_name，file_name，** kwargs）

从 Azure 文件共享下载文件。

参数：

- stream(类 _ 文件对象 _) - 用于存储文件的文件句柄。
- share_name(str) - 共享的名称。
- directory_name(str) - 目录的名称。
- file_name(str) - 文件名。

- kwargs(object) - FileService.get_file_to_stream（）采用的可选关键字参数。

list_directories_and_files（share_name，directory_name = None，** kwargs）

返回存储在 Azure 文件共享中的目录和文件列表。

参数：

- share_name(str) - 共享的名称。
- directory_name(str) - 目录的名称。
- kwargs(object) - FileService.list_directories_and_files（）采用的可选关键字参数。

返回值： 文件和目录列表 ：rtype 列表

load_file（file_path，share_name，directory_name，file_name，** kwargs）

将文件上载到 Azure 文件共享。

参数：

- file_path(str) - 要加载的文件的路径。
- share_name(str) - 共享的名称。
- directory_name(str) - 目录的名称。
- file_name(str) - 文件名。
- kwargs(object) - FileService.create_file_from_path（）采用的可选关键字参数。

load_stream（stream，share_name，directory_name，file_name，count，** kwargs）

将流上载到 Azure 文件共享。

参数：

- stream(类 _ 文件 _) - 打开的文件/流作为文件内容上传。
- share_name(str) - 共享的名称。
- directory_name(str) - 目录的名称。
- file_name(str) - 文件名。
- count(int) - 流的大小（以字节为单位）
- kwargs(object) - FileService.create_file_from_stream（）采用的可选关键字参数。

load_string（string_data，share_name，directory_name，file_name，** kwargs）

将字符串上载到 Azure 文件共享。

参数：

- string_data(str) - 要加载的字符串。
- share_name(str) - 共享的名称。
- directory_name(str) - 目录的名称。
- file_name(str) - 文件名。
- kwargs(object) - FileService.create_file_from_text（）采用的可选关键字参数。

class airflow.contrib.hooks.bigquery_hook.BigQueryHook(bigquery_conn_id='bigquery_default', delegate_to = None, use_legacy_sql = True)

基类：airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook，airflow.hooks.dbapi_hook.DbApiHook，airflow.utils.log.logging_mixin.LoggingMixin

与 BigQuery 交互。此挂钩使用 Google Cloud Platform 连接。

get_conn()

返回 BigQuery PEP 249 连接对象。

get_pandas_df(sql, parameters = None, dialect = None)

返回 BigQuery 查询生成的结果的 Pandas DataFrame。必须重写 DbApiHook 方法，因为 Pandas 不支持 PEP 249 连接，但 SQLite 除外。看到：

https://github.com/pydata/pandas/blob/master/pandas/io/sql.py#L447
https://github.com/pydata/pandas/issues/6900

参数：

- sql(string) - 要执行的 BigQuery SQL。
- 参数(_ 映射 _ 或 _ 可迭代 _) - 用于呈现 SQL 查询的参数（未使用，请保留覆盖超类方法）
- dialect(_{'legacy' __, _ _'standard'}中的 _string) - BigQuery SQL 的方言 - 遗留 SQL 或标准 SQL 默认使用 self.use_legacy_sql（如果未指定）

get_service()

返回一个 BigQuery 服务对象。

insert_rows(table, rows, target_fields = None, commit_every = 1000)

目前不支持插入。从理论上讲，您可以使用 BigQuery 的流 API 将行插入表中，但这尚未实现。

table_exists(project_id, dataset_id, table_id)

检查 Google BigQuery 中是否存在表格。

参数：

- project_id(string) - 要在其中查找表的 Google 云项目。提供给钩子的连接必须提供对指定项目的访问。
- dataset_id(string) - 要在其中查找表的数据集的名称。
- table_id(string) - 要检查的表的名称。

class airflow.contrib.hooks.cassandra_hook.CassandraHook (cassandra_conn_id ='cassandra_default')

基类：airflow.hooks.base_hook.BaseHook，airflow.utils.log.logging_mixin.LoggingMixin

胡克曾经与卡桑德拉互动

可以在连接的“hosts”字段中将联系点指定为逗号分隔的字符串。

可以在连接的端口字段中指定端口。

如果在 Cassandra 中启用了 SSL,则将额外字段中的 dict 作为 kwargs 传入ssl.wrap_socket()。例如：

> {
>

> 'ssl_options': {
>

'ca_certs': PATH_TO_CA_CERTS

}

}

默认负载平衡策略是 RoundRobinPolicy。要指定不同的 LB 策略：

*

DCAwareRoundRobinPolicy
{
> 'load_balancing_policy': 'DCAwareRoundRobinPolicy', 'load_balancing_policy_args': {
>
> > 'local_dc': LOCAL_DC_NAME, //可选'used_hosts_per_remote_dc': SOMEintVALUE, //可选
>

> }

}

*

WhiteListRoundRobinPolicy

{

'load_balancing_policy': 'WhiteListRoundRobinPolicy', 'load_balancing_policy_args': {

> '主持人': ['HOST1', 'HOST2', 'HOST3']

}

}

*

TokenAwarePolicy

{

'load_balancing_policy': 'TokenAwarePolicy', 'load_balancing_policy_args': {

> 'child_load_balancing_policy' : CHILD_POLICY_NAME , // 可 选
'child_load_balancing_policy_args': {...} //可选

}

}

有关群集配置的详细信息，请参阅 cassandra.cluster。

get_conn（）

返回一个 cassandra Session 对象

record_exists（表，键）

检查 Cassandra 中是否存在记录

参数：

- table(string) - 目标 Cassandra 表。使用点表示法来定位特定键空间。
- keys(dict) - 用于检查存在的键及其值。

shutdown_cluster（）

关闭与此群集关联的所有会话和连接。

class airflow.contrib.hooks.cloudant_hook.CloudantHook（cloudant_conn_id ='cloudant_default'）

基类： airflow.hooks.base_hook.BaseHook

与 Cloudant 互动。

这个类是 cloudant python 库的一个薄包装器。请参阅此处的文档。

D b（）

返回此挂接的 Database 对象。

请参阅 cloudant-python 的文档 https://github.com/cloudant-labs/cloudant-python。

class airflow.contrib.hooks.databricks_hook.DatabricksHook（ databricks_conn_id ='databricks_default', timeout_seconds = 180, retry_limit = 3）

基类：airflow.hooks.base_hook.BaseHook, airflow.utils.log.logging_mixin.LoggingMixin

与 Databricks 互动。

submit_run（JSON）

用于调用 api/2.0/jobs/runs/submit 端点的实用程序功能。

参数：json(dict)- 在 submit 端点请求体中使用的数据。返回值： run_id 为字符串| 返回类型： | 串

class airflow.contrib.hooks.datadog_hook.DatadogHook（datadog_conn_id ='datadog_default'）

基类：airflow.hooks.base_hook.BaseHook, airflow.utils.log.logging_mixin.LoggingMixin

使用 datadog API 发送几乎任何可测量的度量标准，因此可以跟踪插入/删除的 db 记录数，从文件读取的记录以及许多其他有用的度量标准。

取决于 datadog API，该 API 必须部署在 Airflow 运行的同一服务器上。

参数：

- datadog_conn_id - 与 datadog 的连接，包含 api 密钥的元数据。
- datadog_conn_id - 字符串

- post_event（title，text，tags = None，alert_type = None，aggregation_key = None）

将事件发布到 datadog（处理完成，潜在警报，其他问题）将此视为维持警报持久性的一种方法，而不是提醒自己。

参数：

- title(string) - 事件的标题
- text(string) - 事件正文（更多信息）
- tags(list) - 要应用于事件的字符串标记列表
- alert_type(string) - 事件的警报类型，[“错误”，“警告”，“信息”，“成功”之一]
- aggregation_key(string) - 可用于在流中聚合此事件的键

query_metric（query，from_seconds_ago，to_seconds_ago）

查询特定度量标准的数据路径，可能会应用某些功能并返回结果。

参数：

- query(string) - 要执行的 datadog 查询（请参阅 datadog docs）
- from_seconds_ago(int) - 开始查询的秒数。
- to_seconds_ago(int) - 最多查询前几秒。

send_metric（metric_name，datapoint，tags = None）

将单个数据点度量标准发送到 DataDog

参数：

- metric_name(string) - 度量标准的名称
- datapoint(_ 整数 _ 或 _ 浮点数 _) - 与度量标准相关的单个整数或浮点数
- tags(list) - 与度量标准关联的标记列表

class airflow.contrib.hooks.datastore_hook.DatastoreHook（datastore_conn_id='google_cloud_datastore_default'，delegate_to = None）

基类： airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook

与 Google Cloud Datastore 互动。此挂钩使用 Google Cloud Platform 连接。

此对象不是线程安全的。如果要同时发出多个请求，则需要为每个线程创建一个钩子。

allocate_ids（partialKeys）

为不完整的密钥分配 ID。请参阅
https://cloud.google.com/datastore/docs/reference/rest/v1/projects/allocateIds

参数：partialKeys - 部分键列表返回值： 完整密钥列表。

begin_transaction（）

获取新的事务处理

也可以看看
https://cloud.google.com/datastore/docs/reference/rest/v1/projects/beginTransaction

返回值： 交易句柄

提交（体）

提交事务，可选地创建，删除或修改某些实体。

也可以看看 https://cloud.google.com/datastore/docs/reference/rest/v1/projects/commit

参数：body - 提交请求的主体返回值： 提交请求的响应主体

delete_operation（名称）

删除长时间运行的操作

参数：name - 操作资源的名称

export_to_storage_bucket（bucket，namespace = None，entity_filter = None，labels = None）

将实体从 Cloud Datastore 导出到 Cloud Storage 进行备份

get_conn（版本='V1'）

返回 Google 云端存储服务对象。

GET_OPERATION（名称）

获取长时间运行的最新状态

参数：name - 操作资源的名称

import_from_storage_bucket（bucket，file，namespace = None，entity_filter = None，labels = None）

将备份从云存储导入云数据存储

lookup（keys，read_consistency = None，transaction = None）

按键查找一些实体

也可以看看 https://cloud.google.com/datastore/docs/reference/rest/v1/projects/lookup

参数：

- keys - 要查找的键
- read_consistency - 要使用的读取一致性。默认，强或最终。不能与事务一起使用。
- transaction - 要使用的事务，如果有的话。

返回值： 查找请求的响应主体。

poll_operation_until_done（name，polling_interval_in_seconds）

轮询备份操作状态直到完成

回滚（事务）

回滚交易

也可以看看

https://cloud.google.com/datastore/docs/reference/rest/v1/projects/rollback

参数：transaction - 要回滚的事务

run_query（体）

运行实体查询。

也可以看看 https://cloud.google.com/datastore/docs/reference/rest/v1/projects/runQuery

参数：body - 查询请求的主体返回值： 批量查询结果。

class airflow.contrib.hooks.discord_webhook_hook.DiscordWebhookHook（http_conn_id = None，webhook_endpoint = None，message =''，username = None，avatar_url = None，tts = False，proxy = None，* args，** kwargs）

基类： airflow.hooks.http_hook.HttpHook

此挂钩允许您使用传入的 webhooks 将消息发布到 Discord。使用默认相对 webhook 端点获取 Discord 连接 ID。可以使用 webhook_endpoint 参数（https://discordapp.com/developers/docs/resources/webhook）覆盖默认端点。

每个 Discord webhook 都可以预先配置为使用特定的用户名和 avatar_url。您可以在此挂钩中覆盖这些默认值。

参数：

- http_conn_id(str) - Http 连接 ID，主机为“ https://discord.com/api/ ”，默认 webhook 端点在额外字段中，格式为{“webhook_endpoint”:“webhooks / {webhook.id} / { webhook.token}”
- webhook_endpoint(str) - 以“webhooks / {webhook.id} / {webhook.token}”的形式 Discord webhook 端点
- message(str) - 要发送到 Discord 频道的消息（最多 2000 个字符）
- username(str) - 覆盖 webhook 的默认用户名
- avatar_url(str) - 覆盖 webhook 的默认头像
- tts(bool) - 是一个文本到语音的消息
- proxy(str) - 用于进行 Discord webhook 调用的代理

执行（）

执行 Discord webhook 调用

class airflow.contrib.hooks.emr_hook.EmrHook（emr_conn_id = None，* args，** kwargs）

基类： airflow.contrib.hooks.aws_hook.AwsHook

与 AWS EMR 交互。emr_conn_id 只是使用 create_job_flow 方法所必需的。

create_job_flow（job_flow_overrides）

使用 EMR 连接中的配置创建作业流。json 额外哈希的键可以具有 boto3 run_job_flow 方法的参数。此配置的覆盖可以作为 job_flow_overrides 传递。

class airflow.contrib.hooks.fs_hook.FSHook（conn_id ='fs_default'）

基类： airflow.hooks.base_hook.BaseHook

允许与文件服务器交互。

连接应具有名称和额外指定的路径：

示例：Conn Id：fs_test Conn 类型：文件（路径）主机，Shchema，登录，密码，端口：空额外：{“path”：“/ tmp”}

class airflow.contrib.hooks.ftp_hook.FTPHook（ftp_conn_id ='ftp_default'）

基类：airflow.hooks.base_hook.BaseHook，airflow.utils.log.logging_mixin.LoggingMixin

与 FTP 交互。

可能在整个过程中发生的错误应该在下游处理。

close_conn（）

关闭连接。如果未打开连接，则会发生错误。

create_directory（路径）

在远程系统上创建目录。

参数：path(str) - 要创建的远程目录的完整路径

delete_directory（路径）

删除远程系统上的目录。

参数：path(str) - 要删除的远程目录的完整路径

DELETE_FILE（路径）

删除 FTP 服务器上的文件。

参数：path(str) - 远程文件的完整路径

describe_directory（路径）

返回远程系统上所有文件的{filename：{attributes}}字典（支持 MLSD 命令）。

参数：path(str) - 远程目录的完整路径

get_conn（）

返回 FTP 连接对象

list_directory（path，nlst = False）
返回远程系统上的文件列表。

参数：path(str) - 要列出的远程目录的完整路径

重命名（from_name，to_name）

重命名文件。

参数：

- from_name - 从名称重命名文件
- to_name - 将文件重命名为 name

retrieve_file（remote_full_path，local_full_path_or_buffer）

将远程文件传输到本地位置。

如果 local_full_path_or_buffer 是字符串路径，则该文件将放在该位置；如果它是类似文件的缓冲区，则该文件将被写入缓冲区但不会被关闭。

参数：

- remote_full_path(str) - 远程文件的完整路径
- local_full_path_or_buffer(_str __ 或类 _ _ 文件缓冲区 _) - 本地文件的完整路径或类文件缓冲区

store_file（remote_full_path，local_full_path_or_buffer）

将本地文件传输到远程位置。

如果 local_full_path_or_buffer 是字符串路径，则将从该位置读取该文件；如果它是类似文件的缓冲区，则从缓冲区读取文件但不关闭。

参数：

- remote_full_path(str) - 远程文件的完整路径
- local_full_path_or_buffer(_str __ 或类 _ _ 文件缓冲区 _) - 本地文件的完整路径或类文件缓冲区

class airflow.contrib.hooks.ftp_hook.FTPSHook（ftp_conn_id ='ftp_default'）

基类： airflow.contrib.hooks.ftp_hook.FTPHook

get_conn（）

返回 FTPS 连接对象。

class airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook (gcp_conn_id ='google_cloud_default', delegate_to = None)

基类：airflow.hooks.base_hook.BaseHook，airflow.utils.log.logging_mixin.LoggingMixin

谷歌云相关钩子的基础钩子。Google 云有一个共享的 REST API 客户端，无论您使用哪种服务，都以相同的方式构建。此类有助于构建和授权所需的凭据，然后调用 apiclient.discovery.build（）来实际发现和构建 Google 云服务的客户端。

该类还包含一些其他辅助函数。

从此基本挂钩派生的所有挂钩都使用“Google Cloud Platform”连接类型。支持两种身份验证方式：

默认凭据：只需要“项目 ID”。您需要设置默认凭据，例如 GOOGLE_APPLICATION_DEFAULT 环境变量或 Google Compute Engine 上的元数据服务器。

JSON 密钥文件：指定“项目 ID”，“密钥路径”和“范围”。

不支持旧版 P12 密钥文件。

class airflow.contrib.hooks.gcp_container_hook.GKEClusterHook（project_id, location）

基类： airflow.hooks.base_hook.BaseHook

create_cluster（cluster, retry = <object object>, timeout = <object object>）

创建一个群集，由指定数量和类型的 Google Compute Engine 实例组成。

参数：

- cluster(_dict _ 或 _google.cloud.container_v1.types.Cluster_) - 群集 protobuf 或 dict。如果提供了 dict，它必须与 protobuf 消息的格式相同 google.cloud.container_v1.types.Cluster
- 重试（_google.api_core.retry.Retry_） - 用于重试请求的重试对象（google.api_core.retry.Retry）。如果指定 None，则不会重试请求。
- timeout(float) - 等待请求完成的时间（以秒为单位）。请注意，如果指定了重试，则超时适用于每次单独尝试。

返回值： 新集群或现有集群的完整 URL

：加薪

ParseError：在尝试转换 dict 时出现 JSON 解析问题 AirflowException：cluster 不是 dict 类型也不

是 Cluster proto 类型

delete_cluster（name，retry = <object object>，timeout = <object object>）

删除集群，包括 Kubernetes 端点和所有工作节点。在群集创建期间配置的防火墙和路由也将被删除。群集可能正在使用的其他 Google Compute Engine 资源（例如，负载均衡器资源）如果在初始创建时不存在，则不会被删除。

参数：

- name(str) - 要删除的集群的名称
- 重试(_google.api_core.retry.Retry_) - 重 _ 试用 _ 于确定何时/是否重试请求的对象。如果指定 None，则不会重试请求。
- timeout(float) - 等待请求完成的时间（以秒为单位）。请注意，如果指定了重试，则超时适用于每次单独尝试。

返回值： 如果成功则删除操作的完整 URL，否则为 None

get_cluster（name，retry = <object object>，timeout = <object object>）

获取指定集群的详细信息：param name：要检索的集群的名称：type name：str：param retry：用于重试请求的重试对象。如果指定了 None，

请求不会被重试。

参数：timeout(float) - 等待请求完成的时间（以秒为单位）。请注意，如果指定了重试，则超时适用于每次单独尝试。返回值： 一个 google.cloud.container_v1.types.Cluster 实例

GET_OPERATION（OPERATION_NAME）

从 Google Cloud 获取操作：param operation_name：要获取的操作的名称：type operation_name：str：return：来自 Google Cloud 的新的更新操作

wait_for_operation（操作）

给定操作，持续从 Google Cloud 获取状态，直到完成或发生错误：param 操作：等待的操作：键入操作：google.cloud.container_V1.gapic.enums.Operator：return：a new，updated 从 Google Cloud 获取的操作

class airflow.contrib.hooks.gcp_dataflow_hook.DataFlowHook（gcp_conn_id='google_cloud_default'，delegate_to = None，poll_sleep = 10）

基类： airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook

get_conn（）

返回 Google 云端存储服务对象。

class airflow.contrib.hooks.gcp_dataproc_hook.DataProcHook（gcp_conn_id='google_cloud_default'，delegate_to = None，api_version ='v1beta2'）

基类： airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook

Hook for Google Cloud Dataproc API。

等待（操作）

等待 Google Cloud Dataproc Operation 完成。

get_conn（）

返回 Google Cloud Dataproc 服务对象。

等待（操作）

等待 Google Cloud Dataproc Operation 完成。

class airflow.contrib.hooks.gcp_mlengine_hook.MLEngineHook（gcp_conn_id='google_cloud_default'，delegate_to = None）

基类： airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook

create_job（project_id，job，use_existing_job_fn = None）

启动 MLEngine 作业并等待它达到终端状态。

参数：

- project_id(string) - 将在其中启动 MLEngine 作业的 Google Cloud 项目 ID。
- 工作(_ 字典 _) -
  应该提供给 MLEngine API 的 MLEngine Job 对象，例如：

```
{
'jobId'：'my_job_id'，
'trainingInput'：{
  'scaleTier'：'STANDARD_1'，
  ...
}
```

}

- use_existing_job_fn(function) - 如果已存在具有相同 job_id 的 MLEngine 作业，则此方法（如果提供）将决定是否应使用此现有作业，继续等待它完成并返回作业对象。它应该接受 MLEngine 作业对象，并返回一个布尔值，指示是否可以重用现有作业。如果未提供“use_existing_job_fn”，我们默认重用现有的 MLEngine 作业。

返回值： 如果作业成功到达终端状态（可能是 FAILED 或 CANCELED 状态），则为 MLEngine 作业对象。| 返回类型： | 字典

create_model（project_id，model）

创建一个模型。阻止直到完成。

create_version（project_id，model_name，version_spec）

在 Google Cloud ML Engine 上创建版本。

如果版本创建成功则返回操作，否则引发错误。

delete_version（project_id，model_name，version_name）

删除给定版本的模型。阻止直到完成。

get_conn（）

返回 Google MLEngine 服务对象。

get_model（project_id，model_name）

获取一个模型。阻止直到完成。

list_versions（project_id，model_name）

列出模型的所有可用版本。阻止直到完成。

set_default_version（project_id，model_name，version_name）

将版本设置为默认值。阻止直到完成。

class airflow.contrib.hooks.gcp_pubsub_hook.PubSubHook（gcp_conn_id ='google_cloud_default'，delegate_to = None）

基类： airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook

用于访问 Google Pub / Sub 的 Hook。

应用操作的 GCP 项目由嵌入在 gcp_conn_id 引用的 Connection 中的项目确定。

确认（项目，订阅，ack_ids）

max_messages 从 Pub / Sub 订阅中提取消息。

参数：

- project(string) - 用于创建主题的 GCP 项目名称或 ID
- subscription(string) - 要删除的 Pub / Sub 订阅名称; 不包括'projects / {project} / topics /'前缀。
- ack_ids(list) - 来自先前拉取响应的 ReceivedMessage 确认列表

create_subscription（topic_project，topic，subscription = None，subscription_project = None，ack_deadline_secs = 10，fail_if_exists = False）

如果 Pub / Sub 订阅尚不存在，则创建它。

参数：

- topic_project(string) - 订阅将绑定到的主题的 GCP 项目 ID。
- topic(string) - 订阅将绑定创建的发布/订阅主题名称；不包括 projects/{project}/subscriptions/前缀。
- subscription(string) - 发布/订阅订阅名称。如果为空，将使用 uuid 模块生成随机名称
- subscription_project(string) - 将在其中创建订阅的 GCP 项目 ID。如果未指定，topic_project 将使用。
- ack_deadline_secs(int) - 订户必须确认从订阅中提取的每条消息的秒数
- fail_if_exists(bool) - 如果设置，则在主题已存在时引发异常

返回值： 订阅名称，如果 subscription 未提供参数，它将是系统生成的值| 返回类型： | 串

create_topic（project，topic，fail_if_exists = False）

如果 Pub / Sub 主题尚不存在，则创建它。

参数：

- project(string) - 用于创建主题的 GCP 项目 ID
- topic(string) - 要创建的发布/订阅主题名称; 不包括 projects/{project}/topics/前缀。
- fail_if_exists(bool) - 如果设置，则在主题已存在时引发异常

delete_subscription（project，subscription，fail_if_not_exists = False）

删除发布/订阅订阅（如果存在）。

参数：

- project(string) - 订阅所在的 GCP 项目 ID
- subscription(string) - 要删除的 Pub / Sub 订阅名称；不包括 projects/{project}/subscriptions/前缀。
- fail_if_not_exists(bool) - 如果设置，则在主题不存在时引发异常

delete_topic（项目，主题，fail_if_not_exists = False）

删除 Pub / Sub 主题（如果存在）。

参数：

- project(string) - 要删除主题的 GCP 项目 ID
- topic(string) - 要删除的发布/订阅主题名称；不包括 projects/{project}/topics/前缀。
- fail_if_not_exists(bool) - 如果设置，则在主题不存在时引发异常

get_conn（）

返回 Pub / Sub 服务对象。

| 返回类型： | apiclient.discovery.Resource

发布（项目，主题，消息）

将消息发布到 Pub / Sub 主题。

参数：

- project(string) - 要发布的 GCP 项目 ID
- topic(string) - 要发布的发布/订阅主题；不包括 projects/{project}/topics/前缀。
- 消息(PubSub 消息列表;请参阅 http://cloud.google.com/pubsub/docs/reference/rest/v1/PubsubMessage) - 要发布的消息；如果设置了消息中的数据字段，则它应该已经是 base64 编码的。

pull（项目，订阅，max_messages，return_immediately = False）

max_messages 从 Pub / Sub 订阅中提取消息。

参数：

- project(string) - 订阅所在的 GCP 项目 ID
- subscription(string) - 要从中提取的 Pub / Sub 订阅名称; 不包括'projects / {project} / topics /'前缀。
- max_messages(int) - 从 Pub / Sub API 返回的最大消息数。
- return_immediately(bool) - 如果设置，如果没有可用的消息，Pub / Sub API 将立即返回。否则，请求将阻止未公开但有限的时间段

: return 每个包含的 Pub / Sub ReceivedMessage 对象列表

的 ackId 属性和 message 属性，其中包括 base64 编码消息内容。请参阅 https://cloud.google.com/pubsub/docs/reference/rest/v1/ projects.subscriptions / pull #ReceivedMessage

class airflow.contrib.hooks.gcs_hook.GoogleCloudStorageHook （ google_cloud_storage_conn_id='google_cloud_default', delegate_to = None）

基类： airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook

与 Google 云端存储互动。此挂钩使用 Google Cloud Platform 连接。

copy（source_bucket, source_object, destination_bucket = None, destination_object = None）

将对象从存储桶复制到另一个存储桶，并在需要时重命名。

destination_bucket 或 destination_object 可以省略，在这种情况下使用源桶/对象，但不能同时使用两者。

参数：

- source_bucket(string) - 要从中复制的对象的存储桶。
- source_object(string) - 要复制的对象。
- destination_bucket(string) - 要复制到的对象的目标。可以省略; 然后使用相同的桶。
- destination_object - 给定对象的（重命名）路径。可以省略; 然后使用相同的名称。

create_bucket（bucket_name，storage_class ='MULTI_REGIONAL'，location ='US'，project_id = None，labels = None）

创建一个新存储桶。Google 云端存储使用平面命名空间，因此您无法创建名称已在使用中的存储桶。

也可以看看有关详细信息，请参阅存储桶命名指南：
https：//cloud.google.com/storage/docs/bucketnaming.html#requirements

参数：

- bucket_name(string) - 存储桶的名称。
- storage_class(string) -
  这定义了存储桶中对象的存储方式，并确定了 SLA 和存储成本。价值包括
  - MULTI_REGIONAL
  - REGIONAL
  - STANDARD
  - NEARLINE
  - COLDLINE
  如果在创建存储桶时未指定此值，则默认为 STANDARD。

- 位置(string) -
  水桶的位置。存储桶中对象的对象数据驻留在此区域内的物理存储中。默认为美国。

也可以看看 https://developers.google.com/storage/docs/bucket-locations

- project_id(string) - GCP 项目的 ID。
- labels(dict) - 用户提供的键/值对标签。

返回值： 如果成功，则返回 id 桶的内容。

删除（桶，对象，生成=无）

如果未对存储桶启用版本控制，或者使用了生成参数，则删除对象。

参数：

- bucket(string) - 对象所在的存储桶的名称
- object(string) - 要删除的对象的名称
- generation(string) - 如果存在，则永久删除该代的对象

返回值： 如果成功则为真

下载（bucket，object，filename = None）

从 Google 云端存储中获取文件。

参数：

- bucket(string) - 要获取的存储桶。
- object(string) - 要获取的对象。
- filename(string) - 如果设置，则应写入文件的本地文件路径。

存在（桶，对象）

检查 Google 云端存储中是否存在文件。

参数：

- bucket(string) - 对象所在的 Google 云存储桶。
- object(string) - 要在 Google 云存储分区中检查的对象的名称。

`get_conn（）`

返回 Google 云端存储服务对象。

`get_crc32c（bucket, object）`

获取 Google Cloud Storage 中对象的 CRC32c 校验和。

参数：

- bucket(string) - 对象所在的 Google 云存储桶。
- object(string) - 要在 Google 云存储分区中检查的对象的名称。

`get_md5hash（bucket, object）`
获取 Google 云端存储中对象的 MD5 哈希值。

参数：

- bucket(string) - 对象所在的 Google 云存储桶。
- object(string) - 要在 Google 云存储分区中检查的对象的名称。

`get_size（bucket, object）`

获取 Google 云端存储中文件的大小。

参数：

- bucket(string) - 对象所在的 Google 云存储桶。
- object(string) - 要在 Google 云存储分区中检查的对象的名称。

`is_updated_after（bucket, object, ts）`

检查 Google Cloud Storage 中是否更新了对象。

参数：

- bucket(string) - 对象所在的 Google 云存储桶。

- object(string) - 要在 Google 云存储分区中检查的对象的名称。
- ts(datetime) - 要检查的时间戳。

```
list（bucket，versions = None，maxResults = None，prefix = None，delimiter = None）
```

使用名称中的给定字符串前缀列出存储桶中的所有对象

参数：

- bucket(string) - 存储桶名称
- versions(_boolean_) - 如果为 true，则列出对象的所有版本
- maxResults(_ 整数 _) - 在单个响应页面中返回的最大项目数
- prefix(string) - 前缀字符串，用于过滤名称以此前缀开头的对象
- delimiter(string) - 根据分隔符过滤对象（例如'.csv'）

返回值： 与过滤条件匹配的对象名称流

```
重写（source_bucket，source_object，destination_bucket，destination_object = None）
```

具有与复制相同的功能，除了可以处理超过 5 TB 的文件，以及在位置和/或存储类之间复制时。

destination_object 可以省略，在这种情况下使用 source_object。

参数：

- source_bucket(string) - 要从中复制的对象的存储桶。
- source_object(string) - 要复制的对象。
- destination_bucket(string) - 要复制到的对象的目标。
- destination_object - 给定对象的（重命名）路径。可以省略; 然后使用相同的名称。

```
upload（bucket，object，filename，mime_type ='application / octet-stream'）
```

将本地文件上传到 Google 云端存储。

参数：

- bucket(string) - 要上传的存储桶。
- object(string) - 上载本地文件时要设置的对象名称。
- filename(string) - 要上载的文件的本地文件路径。
- mime_type(string) - 上载文件时要设置的 MIME 类型。

```
class airflow.contrib.hooks.jenkins_hook.JenkinsHook（conn_id ='jenkins_default'）
```

基类： airflow.hooks.base_hook.BaseHook

挂钩管理与 jenkins 服务器的连接

```
class airflow.contrib.hooks.jira_hook.JiraHook（jira_conn_id='jira_default', proxies=None）
```

基类：airflow.hooks.base_hook.BaseHook，airflow.utils.log.logging_mixin.LoggingMixin

Jira 交互钩，JIRA Python SDK 的 Wrapper。

参数：jira_conn_id(string) - 对预定义的 Jira 连接的引用

```
class airflow.contrib.hooks.mongo_hook.MongoHook（conn_id='mongo_default', *args, **kwargs）
```

基类： airflow.hooks.base_hook.BaseHook

PyMongo Wrapper 与 Mongo 数据库进行交互 Mongo 连接文档
https://docs.mongodb.com/manual/reference/connection-string/index.html
您可以在连接的额外字段中指定连接字符串选项
https : //docs.mongodb .com / manual / reference / connection-string / index.html # connection-string-options ex。

{replicaSet：test，ssl：True，connectTimeoutMS：30000}

```
aggregate（mongo_collection, aggregate_query, mongo_db = None, ** kwargs）
```

运行聚合管道并返回结果
https://api.mongodb.com/python/current/api/pymongo/collection.html#pymongo.collection.Collection.aggregate
https://api.mongodb.com/python/current /examples/aggregation.html

```
find（mongo_collection, query, find_one = False, mongo_db = None, ** kwargs）
```

运行 mongo 查找查询并返回结果
https://api.mongodb.com/python/current/api/pymongo/collection.html#pymongo.collection.Collection.find

```
get_collection（mongo_collection, mongo_db = None）
```

获取用于查询的 mongo 集合对象。

除非指定，否则将连接模式用作 DB。

```
get_conn（）
```

获取 PyMongo 客户端

insert_many（mongo_collection，docs，mongo_db = None，** kwargs）

将许多文档插入到 mongo 集合中。
https://api.mongodb.com/python/current/api/pymongo/collection.html#pymongo.collection.Collection.insert_many

insert_one（mongo_collection，doc，mongo_db = None，** kwargs）

将单个文档插入到 mongo 集合中
https://api.mongodb.com/python/current/api/pymongo/collection.html#pymongo.collection.Collection.insert_one

class airflow.contrib.hooks.pinot_hook.PinotDbApiHook（* args，** kwargs）

基类： airflow.hooks.dbapi_hook.DbApiHook

连接到 pinot db（https://github.com/linkedin/pinot）以发出 pql

get_conn（）

通过 pinot dbqpi 建立与 pinot 代理的连接。

get_first（SQL）

执行 sql 并返回第一个结果行。

参数：sql(_str _ 或 list) - 要执行的 sql 语句（str）或要执行的 sql 语句列表

get_pandas_df（sql，parameters = None）

执行 sql 并返回一个 pandas 数据帧

参数：

- sql(_str _ 或 list) - 要执行的 sql 语句（str）或要执行的 sql 语句列表
- 参数(_mapping _ 或 _iterable_) - 用于呈现 SQL 查询的参数。

get_records（SQL）

执行 sql 并返回一组记录。

参数：sql(str) - 要执行的 sql 语句（str）或要执行的 sql 语句列表

get_uri（）

获取 pinot 经纪人的连接 uri。

例如：http：// localhost：9000 / pql

insert_rows（table，rows，target_fields = None，commit_every = 1000）

将一组元组插入表中的通用方法是，每个 commit_every 行都会创建一个新事务

参数：

- table(str) - 目标表的名称
- rows(_ 可迭代的元组 _) - 要插入表中的行
- target_fields(_ 可迭代的字符串 _) - 要填充表的列的名称
- commit_every(int) - 要在一个事务中插入的最大行数。设置为 0 以在一个事务中插入所有行。
- replace(bool) - 是否替换而不是插入

set_autocommit（conn，autocommit）

设置连接上的自动提交标志

class airflow.contrib.hooks.qubole_hook.QuboleHook（* args，** kwargs）

基类：airflow.hooks.base_hook.BaseHook，airflow.utils.log.logging_mixin.LoggingMixin

get_jobs_id（TI）

获取与 Qubole 命令相关的作业：param ti：任务 dag 的实例，用于确定 Quboles 命令 id：return：与命令关联的作业信息

get_log（TI）

从 Qubole 获取命令的日志：param ti：dag 的任务实例，用于确定 Quboles 命令 id：return：命令日志作为文本

get_results（ti = None，fp = None，inline = True，delim = None，fetch = True）

从 Qubole 获取命令的结果（或仅 s3 位置）并保存到文件中：param ti：任务 dag 的实例，用于确定 Quboles 命令 id：param fp：可选文件指针，将创建一个并返回如果 None 传递：param inline：True 表示只下载实际结果，False 表示仅获取 s3 位置：param delim：用给定的 delim 替换 CTL-A 字符，默认为','：param fetch：当 inline 为 True 时，直接从中获取结果 s3（如果大）：return：包含实际结果的文件位置或结果的 s3 位置

杀（TI）

杀死（取消）一个 Qubole 委托：param ti：任务 dag 的实例，用于确定 Quboles 命令 id：return：来自 Qubole 的响应

class airflow.contrib.hooks.redis_hook.RedisHook（redis_conn_id ='redis_default'）

基类：airflow.hooks.base_hook.BaseHook，airflow.utils.log.logging_mixin.LoggingMixin

挂钩与 Redis 数据库交互

get_conn（）

返回 Redis 连接。

key_exists（钥匙）

检查 Redis 数据库中是否存在密钥

参数：key(string) - 检查存在的关键。

class airflow.contrib.hooks.redshift_hook.RedshiftHook（aws_conn_id ='aws_default'）

基类： airflow.contrib.hooks.aws_hook.AwsHook

使用 boto3 库与 AWS Redshift 交互

cluster_status（cluster_identifier）

返回群集的状态

参数：cluster_identifier(str) - 集群的唯一标识符

create_cluster_snapshot（snapshot_identifier，cluster_identifier）

创建群集的快照

参数：

- snapshot_identifier(str) - 群集快照的唯一标识符
- cluster_identifier(str) - 集群的唯一标识符

delete_cluster（cluster_identifier，skip_final_cluster_snapshot = True，

final_cluster_snapshot_identifier =''）

删除群集并可选择创建快照

参数：

- cluster_identifier(str) - 集群的唯一标识符
- skip_final_cluster_snapshot(bool) - 确定群集快照创建
- final_cluster_snapshot_identifier(str) - 最终集群快照的名称

describe_cluster_snapshots（cluster_identifier）

获取群集的快照列表

参数：cluster_identifier(str) - 集群的唯一标识符

restore_from_cluster_snapshot（cluster_identifier，snapshot_identifier）

从其快照还原群集

参数：

- cluster_identifier(str) - 集群的唯一标识符
- snapshot_identifier(str) - 群集快照的唯一标识符

class airflow.contrib.hooks.segment_hook.SegmentHook（segment_conn_id ='segment_default'，segment_debug_mode = False，* args，** kwargs）

基类：airflow.hooks.base_hook.BaseHook，airflow.utils.log.logging_mixin.LoggingMixin

on_error（错误，项目）

在将 segment_debug_mode 设置为 True 的情况下使用 Segment 时处理错误回调

class airflow.contrib.hooks.sftp_hook.SFTPHook（ftp_conn_id ='sftp_default'）

基类： airflow.hooks.base_hook.BaseHook

与 SFTP 互动。旨在与 FTPHook 互换。

陷阱： - 与 FTPHook 相比，describe_directory 只返回大小，类型和 修改。它不返回 unix.owner，unix.mode，perm，unix.group 和 unique。

- retrieve_file 和 store_file 只接受本地完整路径而不是缓冲区。

- 如果没有模式传递给 create_directory，它将以 777 权限创建。

可能在整个过程中发生的错误应该在下游处理。

close_conn（）

关闭连接。如果连接未被打开，则会发生错误。

create_directory（path，mode = 777）

在远程系统上创建目录。：param path：要创建的远程目录的完整路径：type path：str：param mode：int 表示目录的八进制模式

delete_directory（路径）

删除远程系统上的目录。：param path：要删除的远程目录的完整路径：type path：str

DELETE_FILE（路径）

删除 FTP 服务器上的文件：param path：远程文件的完整路径：type path：str

describe_directory（路径）

返回远程系统上所有文件的{filename：{attributes}}字典（支持 MLSD 命令）。：param path：远程目录的完整路径：type path：str

get_conn（）

返回 SFTP 连接对象

list_directory（路径）

返回远程系统上的文件列表。：param path：要列出的远程目录的完整路径：type path：str

retrieve_file（remote_full_path，local_full_path）

将远程文件传输到本地位置。如果 local_full_path 是字符串路径，则文件将放在该位置：param remote_full_path：远程文件的完整路径：type remote_full_path：str：param local_full_path：本地文件的完整路径：type local_full_path：str

store_file（remote_full_path，local_full_path）

将本地文件传输到远程位置。如果 local_full_path_or_buffer 是字符串路径，则将从该位置读取文件：param remote_full_path：远程文件的完整路径：type remote_full_path：str：param local_full_path：

本地文件的完整路径：type local_full_path：str

class airflow.contrib.hooks.slack_webhook_hook.SlackWebhookHook （ http_conn_id = None , webhook_token = None，message =''，channel = None，username = None，icon_emoji = None，link_names = False，proxy = None，* args，** kwargs ）

基类： airflow.hooks.http_hook.HttpHook

此挂钩允许您使用传入的 webhook 将消息发布到 Slack。直接使用 Slack webhook 令牌和具有 Slack webhook 令牌的连接。如果两者都提供，将使用 Slack webhook 令牌。

每个 Slack webhook 令牌都可以预先配置为使用特定频道，用户名和图标。您可以在此挂钩中覆盖这些默认值。

参数：

- http_conn_id(str) - 在额外字段中具有 Slack webhook 标记的连接
- webhook_token(str) - Slack webhook 令牌
- message(str) - 要在 Slack 上发送的消息
- channel(str) - 邮件应发布到的频道
- username(str) - 要发布的用户名
- icon_emoji(str) - 用作发布给 Slack 的用户图标的表情符号
- link_names(bool) - 是否在邮件中查找和链接频道和用户名
- proxy(str) - 用于进行 Slack webhook 调用的代理

执行（）

远程 Popen（实际上执行松弛的 webhook 调用）

参数：

- cmd - 远程执行的命令
- kwargs - Popen 的额外参数（参见 subprocess.Popen）

class airflow.contrib.hooks.snowflake_hook.SnowflakeHook（* args，** kwargs）

基类： airflow.hooks.dbapi_hook.DbApiHook

与 Snowflake 互动。

get_sqlalchemy_engine（）依赖于 snowflake-sqlalchemy

get_conn（）

返回一个 snowflake.connection 对象

get_uri（）

覆盖 get_sqlalchemy_engine（）的 DbApiHook get_uri 方法

set_autocommit（conn，autocommit）

设置连接上的自动提交标志

class airflow.contrib.hooks.spark_jdbc_hook.SparkJDBCHook（ spark_app_name ='airflow-spark-jdbc'，spark_conn_id ='spark-default'，spark_conf = None，spark_py_files = None，spark_files = None，spark_jars = None，num_executors = None，executor_cores = None，executor_memory = None， driver_memory = None， verbose = False， principal = None， keytab = None， cmd_type ='spark_to_jdbc'， jdbc_table = None， jdbc_conn_id ='jdbc-default'， jdbc_driver = None， metastore_table = None， jdbc_truncate = False， save_mode = None， save_format = None， batch_size = None， fetch_size = None， num_partitions = None， partition_column = None， lower_bound = None， upper_bound = None， create_table_column_types = None， * args， ** kwargs）

基类： airflow.contrib.hooks.spark_submit_hook.SparkSubmitHook

此钩子扩展了 SparkSubmitHook，专门用于使用 Apache Spark 执行与基于 JDBC 的数据库之间的数据传输。

参数：

- spark_app_name**（str） - 作业名称（默认为**airflow -spark-jdbc）
- spark_conn_id(str) - 在 Airflow 管理中配置的连接 ID
- spark_conf(dict) - 任何其他 Spark 配置属性
- spark_py_files(str) - 使用的其他 python 文件（.zip，.egg 或.py）
- spark_files(str) - 要上载到运行作业的容器的其他文件
- spark_jars(str) - 要上传并添加到驱动程序和执行程序类路径的其他 jar
- num_executors(int) - 要运行的执行程序的数量。应设置此项以便管理使用 JDBC 数据库建立的连接数
- executor_cores(int) - 每个执行程序的核心数
- executor_memory(str) - 每个执行者的内存（例如 1000M，2G）
- driver_memory(str) - 分配给驱动程序的内存（例如 1000M，2G）
- verbose(bool) - 是否将详细标志传递给 spark-submit 进行调试
- keytab(str) - 包含 keytab 的文件的完整路径
- principal(str) - 用于 keytab 的 kerberos 主体的名称
- cmd_type(str) - 数据应该以哪种方式流动。2 个可能的值：spark_to_jdbc：从 Metastore 到 jdbc jdbc_to_spark 的 spark 写入的数据：来自 jdbc 到 Metastore 的 spark 写入的数据
- jdbc_table(str) - JDBC 表的名称
- jdbc_conn_id - 用于连接 JDBC 数据库的连接 ID

- jdbc_driver(str) - 用于 JDBC 连接的 JDBC 驱动程序的名称。这个驱动程序（通常是一个 jar）应该在'jars'参数中传递
- metastore_table(str) - Metastore 表的名称，
- jdbc_truncate(bool) - （仅限 spark_to_jdbc）Spark 是否应截断或删除并重新创建 JDBC 表。这仅在'save_mode'设置为 Overwrite 时生效。此外，如果架构不同，Spark 无法截断，并将丢弃并重新创建
- save_mode(str) - 要使用的 Spark 保存模式（例如覆盖，追加等）
- save_format(str) - （jdbc_to_spark-only）要使用的 Spark 保存格式（例如镶木地板）
- batch_size(int) - （仅限 spark_to_jdbc）每次往返 JDBC 数据库时要插入的批处理的大小。默认为 1000
- fetch_size(int) - （仅限 jdbc_to_spark）从 JDBC 数据库中每次往返获取的批处理的大小。默认值取决于 JDBC 驱动程序
- num_partitions(int) - Spark 同时可以使用的最大分区数，包括 spark_to_jdbc 和 jdbc_to_spark 操作。这也将限制可以打开的 JDBC 连接数
- partition_column(str) - （jdbc_to_spark-only）用于对 Metastore 表进行分区的数字列。如果已指定，则还必须指定：num_partitions，lower_bound，upper_bound
- lower_bound(int) - （jdbc_to_spark-only）要获取的数字分区列范围的下限。如果已指定，则还必须指定：num_partitions，partition_column，upper_bound
- upper_bound(int) - （jdbc_to_spark-only）要获取的数字分区列范围的上限。如果已指定，则还必须指定：num_partitions，partition_column，lower_bound
- create_table_column_types - （spark_to_jdbc-only）创建表时要使用的数据库列数据类型而不是默认值。应以与 CREATE TABLE 列语法相同的格式指定数据类型信息（例如:“name CHAR(64),comments VARCHAR（1024)”)。指定的类型应该是有效的 spark sql 数据类型。

类型： jdbc_conn_id：str

class airflow.contrib.hooks.spark_sql_hook.SparkSqlHook （ sql ， conf = None ， conn_id ='spark_sql_default',total_executor_cores = None,executor_cores = None,executor_memory = None， keytab = None，principal = None，master ='yarn'，name ='default-name'，num_executors =无，verbose = True，yarn_queue ='default'）

基类： airflow.hooks.base_hook.BaseHook

这个钩子是 spark-sql 二进制文件的包装器。它要求“spark-sql”二进制文件在 PATH 中。：param sql：要执行的 SQL 查询：type sql：str：param conf：arbitrary Spark 配置属性：type conf：str（format：PROP = VALUE ）： param conn_id ： connection_id string ： type conn_id ： str ： param total_executor_cores :(仅限 Standalone 和 Mesos）所有执行程序的总核心数

（默认值：工作者的所有可用核心）

参数：

- executor_cores(int) - （仅限 Standalone 和 YARN）每个执行程序的核心数（默认值：2）
- executor_memory(str) - 每个执行程序的内存（例如 1000M，2G）（默认值：1G）

- keytab(str) - 包含 keytab 的文件的完整路径
- master(str) - spark：// host：port，mesos：// host：port，yarn 或 local
- name(str) - 作业名称。
- num_executors(int) - 要启动的执行程序数
- verbose(bool) - 是否将详细标志传递给 spark-sql
- yarn_queue(str) - 要提交的 YARN 队列（默认值：“default”）

run_query（cmd =''，** kwargs）

Remote Popen（实际执行 Spark-sql 查询）

参数：

- cmd - 远程执行的命令
- kwargs - Popen 的额外参数（参见 subprocess.Popen）

class airflow.contrib.hooks.spark_submit_hook.SparkSubmitHook （ conf = None ， conn_id ='spark_default'，files = None，py_files = None，driver_classpath = None，jars = None，java_class = None，packages = None，exclude_packages = None，repositories = None，total_executor_cores = None，executor_cores = None，executor_memory = None，driver_memory = None，keytab = None，principal = None，name ='default-name'，num_executors = None，application_args = None，env_vars = None，verbose = False ）

基类：airflow.hooks.base_hook.BaseHook，airflow.utils.log.logging_mixin.LoggingMixin

这个钩子是一个围绕 spark-submit 二进制文件的包装器来启动一个 spark-submit 作业。它要求“spark-submit”二进制文件在 PATH 或 spark_home 中提供。：param conf：任意 Spark 配置属性：类型 conf：dict：param conn_id：Airflow 管理中配置的连接 ID。当一个

提供的 connection_id 无效，默认为 yarn。

参数：

- files(str) - 将其他文件上载到运行作业的执行程序，以逗号分隔。文件将放在每个执行程序的工作目录中。例如，序列化对象。
- py_files(str) - 作业使用的其他 python 文件可以是.zip，.egg 或.py。
- driver_classpath(str) - 其他特定于驱动程序的类路径设置。
- jars(str) - 提交其他 jar 以上传并将它们放在执行程序类路径中。
- java_class(str) - Java 应用程序的主要类
- packages - 包含在的包的逗号分隔的 maven 坐标列表

驱动程序和执行程序类路径：类型包：str：param exclude_packages：解析“packages”中提供的依赖项时要排除的 jar 的 maven 坐标的逗号分隔列表：type exclude_packages：str：param repositories：以逗号分隔的其他远程存储库列表搜索“packages”给出的 maven 坐标：type repositories：str：param

total_executor_cores :(仅限 Standalone 和 Mesos）所有执行程序的总核心数（默认值：worker 上的所有可用核心）：type total_executor_cores：int：param executor_cores :(仅限 Standalone，YARN 和 Kubernetes）每个执行程序的核心数（默认值：2）：类型 executor_cores：int：param executor_memory：每个执行程序的内存(例如 1000M,2G)(默认值:1G):类型 executor_memory:str:param driver_memory ：分配给驱动程序的内存（例如 1000M，2G）（默认值：1G）：类型 driver_memory：str：param keytab：包含 keytab 的文件的完整路径:type keytab:str:param principal:用于 keytab 的 curb eros principal 的名称：type principal：str： param name：作业名称（默认为 airflow-spark）：类型名称：str：param num_executors：要启动的执行程序数：类型 num_executors：int：param application_args：正在提交的应用程序的参数：type application_args：list：param env_vars ：spark-submit 的环境变量。type num_executors：int：param application_args：正在提交的应用程序的参数：type application_args：list:param env_vars:spark-submit 的环境变量。type num_executors:int:param application_args:正在提交的应用程序的参数：type application_args：list：param env_vars：spark-submit 的环境变量。它

也支持纱线和 k8s 模式。

参数：verbose(bool) - 是否将详细标志传递给 spark-submit 进程进行调试

提交（application ='', ** kwargs）

Remote Popen 执行 spark-submit 作业

参数：

- application(str) - 提交的应用程序，jar 或 py 文件
- kwargs - Popen 的额外参数（参见 subprocess.Popen）

class airflow.contrib.hooks.sqoop_hook.SqoopHook（conn_id ='sqoop_default'，verbose = False，num_mappers = None，hcatalog_database = None，hcatalog_table = None，properties = None）

基类：airflow.hooks.base_hook.BaseHook，airflow.utils.log.logging_mixin.LoggingMixin

这个钩子是 sqoop 1 二进制文件的包装器。为了能够使用钩子，需要“sqoop”在 PATH 中。

可以通过 sqoop 连接的“额外”JSON 字段传递的其他参数：

- job_tracker：Job tracker local | jobtracker：port。
- namenode：Namenode。
- lib_jars：逗号分隔的 jar 文件，包含在类路径中。
- files：要复制到地图缩减群集的逗号分隔文件。
- archives：要在计算机上取消归档的逗号分隔的归档

机器。

password_file：包含密码的文件路径。

参数：

- conn_id(str) - 对 sqoop 连接的引用。
- verbose(bool) - 将 sqoop 设置为 verbose。
- num_mappers(int) - 要并行导入的地图任务数。
- properties(dict) - 通过-D 参数设置的属性

```
Popen（cmd，** kwargs）
```

远程 Popen

参数：

- cmd - 远程执行的命令
- kwargs - Popen 的额外参数（参见 subprocess.Popen）

返回值： 处理子进程

```
export_table（table，export_dir，input_null_string，input_null_non_string，staging_table，clear_staging_table ， enclosed_by ， escaped_by ， input_fields_terminated_by ，input_lines_terminated_by ， input_optionally_enclosed_by ， batch ， relaxed_isolation ，extra_export_options = None）
```

将 Hive 表导出到远程位置。参数是直接 sqoop 命令行的副本参数：param table：表远程目标：param export_dir：要导出的 Hive 表：param input_null_string：要解释为 null 的字符串

字符串列

参数：

- input_null_non_string - 非字符串列的解释为 null 的字符串
- staging_table - 在插入目标表之前将暂存数据的表
- clear_staging_table - 指示可以删除登台表中存在的任何数据
- enclosed_by - 设置包含字符的必填字段
- escaped_by - 设置转义字符
- input_fields_terminated_by - 设置字段分隔符
- input_lines_terminated_by - 设置行尾字符
- input_optionally_enclosed_by - 设置包含字符的字段
- batch - 使用批处理模式执行基础语句
- relaxed_isolation - 为映射器读取未提交的事务隔离
- extra_export_options - 作为 dict 传递的额外导出选项。如果某个键没有值，只需将空字符串传递给它即可。对于 sqoop 选项，不要包含 - 的前缀。

import_query（query，target_dir，append = False，file_type ='text'，split_by = None，direct = None，driver = None，extra_import_options = None）

从 rdbms 导入特定查询到 hdfs：param 查询：要运行的自由格式查询：param target_dir：HDFS 目标 dir：param append：将数据附加到 HDFS 中的现有数据集：param file_type：“avro”，“sequence”，“文字“或”镶木地板“

将数据导入 hdfs 为指定格式。默认为文字。

参数：

- split_by - 用于拆分工作单元的表的列
- 直接 - 使用直接导入快速路径
- driver - 手动指定要使用的 JDBC 驱动程序类
- extra_import_options - 作为 dict 传递的额外导入选项。如果某个键没有值，只需将空字符串传递给它即可。对于 sqoop 选项，不要包含 - 的前缀。

import_table（table，target_dir = None，append = False，file_type ='text'，columns = None，split_by = None，where = None，direct = False，driver = None，extra_import_options = None）

将表从远程位置导入到目标目录。参数是直接 sqoop 命令行参数的副本：param table：要读取的表：param target_dir：HDFS 目标 dir：param append：将数据附加到 HDFS 中的现有数据集：param file_type：“avro”，“sequence”，“text”或“镶木地板”。

将数据导入指定的格式。默认为文字。

参数：

- columns - <col，col，col ...>要从表导入的列
- split_by - 用于拆分工作单元的表的列
- where - 导入期间要使用的 WHERE 子句
- direct - 如果存在数据库，则使用直接连接器
- driver - 手动指定要使用的 JDBC 驱动程序类
- extra_import_options - 作为 dict 传递的额外导入选项。如果某个键没有值，只需将空字符串传递给它即可。对于 sqoop 选项，不要包含 - 的前缀。

class airflow.contrib.hooks.ssh_hook.SSHHook（ssh_conn_id = None，remote_host = None，username = None，password = None，key_file = None，port = 22，timeout = 10，keepalive_interval = 30）

基类：airflow.hooks.base_hook.BaseHook，airflow.utils.log.logging_mixin.LoggingMixin

使用 Paramiko 进行 ssh 远程执行的钩子。ref：https：//github.com/paramiko/paramiko 这个钩子还允许你创建 ssh 隧道并作为 SFTP 文件传输的基础

参数：

- ssh_conn_id(str) - 来自气流的连接 ID 从中可以获取所有必需参数的连接，如用户名，密码或 key_file。认为优先权是在 init 期间传递的参数
- remote_host(str) - 要连接的远程主机
- username(str) - 用于连接 remote_host 的用户名
- password(str) - 连接到 remote_host 的用户名的密码
- key_file(str) - 用于连接 remote_host 的密钥文件。
- port(int) - 要连接的远程主机的端口（默认为 paramiko SSH_PORT）
- timeout(int) - 尝试连接到 remote_host 的超时。
- keepalive_interval(int) - 每隔 keepalive_interval 秒向远程主机发送一个 keepalive 数据包

create_tunnel（** kwds）

在两台主机之间创建隧道。与 ssh -L <LOCAL_PORT>类似：host：<REMOTE_PORT>。记得关闭（）返回的“隧道”对象，以便在完成隧道后自行清理。

参数：

- local_port(int) -
- remote_port(int) -
- remote_host(str) -

返回值：

class airflow.contrib.hooks.vertica_hook.VerticaHook（* args，** kwargs）

基类： airflow.hooks.dbapi_hook.DbApiHook

与 Vertica 互动。

get_conn（）

返回 verticaql 连接对象

class airflow.contrib.hooks.wasb_hook.WasbHook（wasb_conn_id ='wasb_default'）

基类： airflow.hooks.base_hook.BaseHook

通过 wasb：//协议与 Azure Blob 存储进行交互。

在连接的“额外”字段中传递的其他选项将传递给 BlockBlockService（）构造函数。例如，通过添加 {“sas_token”：“YOUR_TOKEN”}使用 SAS 令牌进行身份验证。

参数：wasb_conn_id(str) - 对 wasb 连接的引用。

check_for_blob（container_name，blob_name，** kwargs）

检查 Azure Blob 存储上是否存在 Blob。

参数：

container_name(str) - 容器的名称。
blob_name(str) - blob 的名称。
kwargs(object) - BlockBlobService.exists（）采用的可选关键字参数。

返回值： 如果 blob 存在则为 True，否则为 False。 ：rtype 布尔

check_for_prefix（container_name，prefix，** kwargs）

检查 Azure Blob 存储上是否存在前缀。

参数：

- container_name(str) - 容器的名称。
- prefix(str) - blob 的前缀。
- kwargs(object) - BlockBlobService.list_blobs（）采用的可选关键字参数。

返回值： 如果存在与前缀匹配的 blob，则为 True，否则为 False。 ：rtype 布尔

get_conn（）

返回 BlockBlobService 对象。

get_file（file_path，container_name，blob_name，** kwargs）

从 Azure Blob 存储下载文件。

参数：

- file_path(str) - 要下载的文件的路径。
- container_name(str) - 容器的名称。
- blob_name(str) - blob 的名称。
- kwargs(object) - BlockBlobService.create_blob_from_path（）采用的可选关键字参数。

load_file（file_path，container_name，blob_name，** kwargs）

将文件上载到 Azure Blob 存储。

参数：

- file_path(str) - 要加载的文件的路径。
- container_name(str) - 容器的名称。
- blob_name(str) - blob 的名称。
- kwargs(object) - BlockBlobService.create_blob_from_path（）采用的可选关键字参数。

load_string（string_data，container_name，blob_name，** kwargs）

将字符串上载到 Azure Blob 存储。

参数：

- string_data(str) - 要加载的字符串。
- container_name(str) - 容器的名称。
- blob_name(str) - blob 的名称。
- kwargs(object) - BlockBlobService.create_blob_from_text（）采用的可选关键字参数。

read_file（container_name，blob_name，** kwargs）

从 Azure Blob Storage 读取文件并以字符串形式返回。

参数：

- container_name(str) - 容器的名称。
- blob_name(str) - blob 的名称。
- kwargs(object) - BlockBlobService.create_blob_from_path（）采用的可选关键字参数。

class airflow.contrib.hooks.winrm_hook.WinRMHook（ssh_conn_id = None，remote_host = None，username = None，password = None，key_file = None，timeout = 10，keepalive_interval = 30）

基类：airflow.hooks.base_hook.BaseHook，airflow.utils.log.logging_mixin.LoggingMixin

使用 pywinrm 进行 winrm 远程执行的钩子。

参数：

- ssh_conn_id(str) - 来自气流的连接 ID 从中可以获取所有必需参数的连接，如用户名，密码或 key_file。认为优先权是在 init 期间传递的参数
- remote_host(str) - 要连接的远程主机
- username(str) - 用于连接 remote_host 的用户名
- password(str) - 连接到 remote_host 的用户名的密码
- key_file(str) - 用于连接 remote_host 的密钥文件。
- timeout(int) - 尝试连接到 remote_host 的超时。
- keepalive_interval(int) - 每隔 keepalive_interval 秒向远程主机发送一个 keepalive 数据包

执行人

执行程序是运行任务实例的机制。

class airflow.executors.local_executor.LocalExecutor（parallelism = 32）

基类： airflow.executors.base_executor.BaseExecutor

LocalExecutor 并行本地执行任务。它使用多处理 Python 库和队列来并行执行任务。

结束（）

当调用者完成提交作业并且想要同步等待先前提交的作业全部完成时，调用此方法。

execute_async（key，command，queue = None，executor_config = None）

此方法将异步执行该命令。

开始（）

执行人员可能需要开始工作。例如，LocalExecutor 启动 N 个工作者。

同步（）

将通过心跳方法定期调用同步。执行者应该重写此操作以执行收集状态。

class airflow.executors.celery_executor.CeleryExecutor（parallelism = 32）

基类： airflow.executors.base_executor.BaseExecutor

CeleryExecutor 推荐用于 Airflow 的生产使用。它允许将任务实例的执行分配给多个工作节点。

Celery 是一个简单，灵活和可靠的分布式系统，用于处理大量消息，同时为操作提供维护此类系统所需的工具。

端（同步=假）

当调用者完成提交作业并且想要同步等待先前提交的作业全部完成时，调用此方法。

execute_async（key，command，queue ='default'，executor_config = None）

此方法将异步执行该命令。

开始（）

执行人员可能需要开始工作。例如，LocalExecutor 启动 N 个工作者。

同步（）

将通过心跳方法定期调用同步。执行者应该重写此操作以执行收集状态。

class airflow.executors.sequential_executor.SequentialExecutor

基类： airflow.executors.base_executor.BaseExecutor

此执行程序一次只运行一个任务实例，可用于调试。它也是唯一可以与 sqlite 一起使用的执行器，因为 sqlite 不支持多个连接。

由于我们希望气流能够开箱即用,因此在您首次安装时,它会默认使用此 SequentialExecutor 和 sqlite。

结束（）

当调用者完成提交作业并且想要同步等待先前提交的作业全部完成时，调用此方法。

execute_async（key，command，queue = None，executor_config = None）
此方法将异步执行该命令。

同步（）

将通过心跳方法定期调用同步。执行者应该重写此操作以执行收集状态。

社区贡献的遗嘱执行人

class airflow.contrib.executors.mesos_executor.MesosExecutor（parallelism = 32）

基类：airflow.executors.base_executor.BaseExecutor，airflow.www.utils.LoginMixin

MesosExecutor 允许将任务实例的执行分配给多个 mesos worker。

Apache Mesos 是一个分布式系统内核，可以将 CPU，内存，存储和其他计算资源从机器（物理或虚拟）中抽象出来，从而可以轻松构建和运行容错和弹性分布式系统。见 http://mesos.apache.org/

结束（）

当调用者完成提交作业并且想要同步等待先前提交的作业全部完成时，调用此方法。

execute_async（key，command，queue = None，executor_config = None）

此方法将异步执行该命令。

开始（）

执行人员可能需要开始工作。例如，LocalExecutor 启动 N 个工作者。

同步（）

将通过心跳方法定期调用同步。执行者应该重写此操作以执行收集状态。

以上内容来自 https://airflow.apachecn.org/#/，由 About 云整理