Apache Airfolw

1.1 什么是 Airflow

Apache Airflow 是一个提供基于 DAG 有向无环图来编排工作流的、可视化的分布式任务调度平台，与 Oozie、Azkaban 等任务流调度平台类似。Airflow 在 2014 年由 Airbnb 发起，2016 年 3 月进入 Apache 基金会，在 2019 年 1 月成为顶级项目。Airflow 采用 Python 语言编写，提供可编程方式定义 DAG 工作流，可以定义一组有依赖的任务，按照依赖依次执行， 实现任务管理、调度、监控功能。

另外，Airflow 提供了 WebUI 可视化界面，提供了工作流节点的运行监控，可以查看每个节点的运行状态、运行耗时、执行日志等。也可以在界面上对节点的状态进行操作，如：标记为成功、标记为失败以及重新运行等。在 Airflow 中工作流上每个 task 都是原子可重试的，一个工作流某个环节的 task 失败可自动或手动进行重试，不必从头开始跑。

Airflow 官网：http://airflow.apache.org/，Airflow 支持的任务调度类型如下：

- Airbyte
- Amazon
- Apache Beam
- Apache Cassandra
- Apache Drill
- Apache Druid
- Apache HDFS
- Apache Hive
- Apache Kylin
- Apache Livy
- Apache Pig
- Apache Pinot
- Apache Spark
- Apache Sqoop
- Asana
- Celery
- IBM Cloudant
- Kubernetes
- Databricks
- Datadog
- Dingding
- Discord
- Docker
- Elasticsearch
- Exasol
- Facebook
- File Transfer Protocol (FTP)
- Google
- gRPC
- Hashicorp
- Hypertext Transfer Protocol (HTTP)
- Internet Message Access Protocol (IMAP)
- Java Database Connectivity (JDBC)
- Jenkins
- Jira
- Microsoft Azure
- Microsoft SQL Server (MSSQL)
- Windows Remote Management (WinRM)
- MongoDB
- MySQL
- Neo4J
- ODBC
- OpenFaaS
- Opsgenie
- Oracle
- Pagerduty
- Papermill
- Plexus
- PostgreSQL
- Presto
- Qubole
- Redis
- Salesforce
- Samba
- Segment
- Sendgrid
- SFTP
- Singularity
- Slack
- Snowflake
- SQLite
- SSH
- Tableau
- Telegram
- Trino
- Vertica
- Yandex
- Zendesk

1.2 Airflow 架构及原理

1.2.1 Airflow 架构

Airflow 我们可以构建 Workflow 工作流，工作流使用 DAG 有向无环图来表示，DAG 指定了任务之间的关系，如下图：

Airflow 架构图如下：

Airflow 在运行时有很多守护进程，这些进程提供了 airflow 全部功能，守护进程包括如下：

- webserver：

  WebServer 服务器可以接收 HTTP 请求，用于提供用户界面的操作窗口，主要负责中止、恢复、触发任务；监控任务；断点续跑任务；查询任务状态、详细日志等。

- Scheduler:

  调度器，负责周期性调度处理工作流，并将工作流中的任务提交给 Executor 执行。

- Executor:

执行器，负责运行 task 任务，在默认本地模式下（单机 airflow）会运行在调度器 Scheduler 中并负责所有任务的处理。但是在 airflow 集群模式下的执行器 Executor 有很多类型，负责将任务 task 实例推送给 Workers 节点执行。

在 Airflow 中执行器有很多种选择，最关键的执行器有以下几种：

① SequentialExecutor：默认执行器，单进程顺序执行任务，通常只用于测试。

② LocalExecutor:多进程本地执行任务。

③ CeleryExecutor：分布式执行任务，多用于生产场景，使用时需要配置消息队列。

④ DaskExecutor:动态任务调度，支持远程集群执行 airflow 任务。

生产环境中建议使用 CeleryExecutor 作为执行器，Celery 是一个分布式调度框架，本身无队列功能，需要使用第三方插件，例如：RabbitMQ 或者 Redis。

关于不同 Executor 类型可以参考官网：https://airflow.apache.org/docs/apache-airflow/stable/executor/index.html

- work:

Worker 负责执行具体的 DAG 任务，会启动 1 个或者多个 Celery 任务队列，当 ariflow 的 Executor 设置为 CeleryExecutor 时才需要开启 Worker 进程。

- DAG Directory:

存放定义 DAG 任务的 Python 代码目录，代表一个 Airflow 的处理流程。需要保证 Scheduler 和 Executor 都能访问到。

- metadata database:

Airflow 的元数据库，用于 Webserver、Executor 及 Scheduler 存储各种状态数据，通常是 MySQL 或 PostgreSQL。

1.2.2 Airflow 术语

- **DAG**

DAG 是 Directed Acyclic Graph 有向无环图的简称，描述其描述数据流的计算过程。

- **Operators**

描述 DAG 中一个具体 task 要执行的任务，可以理解为 Airflow 中的一系列“算子”，底层对应 python class。不同的 Operator 实现了不同的功能，如：BashOperator 为执行一条 bash 命令，EmailOperator 用户发送邮件，HttpOperators 用户发送 HTTP 请求，PythonOperator 用于调用任意的 Python 函数。

- **Task**

Task 是 Operator 的一个实例，也就是 DAG 中的一个节点，在某个 Operator 的基础上指定具体的参数或者内容就形成一个 Task，DAG 中包含一个或者多个 Task。

- **Task Instance**

task 每一次运行对应一个 Task Instance，Task Instance 有自己的状态，例如：running,success,failed,skipped 等。

- **Task Relationships：**

一个 DAG 中可以有很多 task，这些 task 执行可以有依赖关系，例如：task1 执行后再执行 task2，表明 task2 依赖于 task1，这就是 task 之间的依赖关系。

1.2.3 Airflow 工作原理

airflow 中各个进程彼此之间是独立不互相依赖，也不互相感知，每个进程在运行时只处理分配到自身的任务，各个进程在一起运行，提供了 Airflow 全部功能，其工作原理如下：

1) 调度器 Scheduler 会间隔性轮询元数据库（Metastore）已注册的 DAG 有向无环图作业流，决定是否执行 DAG，如果一个 DAG 根据其调度计划需要执行，Scheduler 会调度当前 DAG 并触发 DAG 内部 task，这里的触发其实并不是真正的去执行任务，而是推送 task 消息到消息队列中，每一个 task 消息都包含此 task 的 DAG ID，Task ID 以及具体需要执行的函数，如果 task 执行的是 bash 脚本，那么 task 消息还会包含 bash 脚本代码。
2) Worker 进程将会监听消息队列，如果有消息就从消息队列中获取消息并执行 DAG 中的 task，如果成功将状态更新为成功，否则更新成失败。
3) 用户可以通过 webserver webui 来控制 DAG，比如手动触发一个 DAG 去执行，手动触发 DAG 与自动触发 DAG 执行过程都一样。

1.3 Airflow 单机搭建

Airflow 是基于 Python 的，就是 Python 中的一个包。安装要求 Python3.6 版本之上，Metadata DataBase 支持 PostgreSQL9.6+，MySQL5.7+，SQLLite3.15.0+。

1.3.1 安装 Anconda 及 python3.7

1. 官网下载 Anconda ，选择 linux 版本，并安装

下载官网地址：https://www.anaconda.com/products/individual#macos

2． 将下载好的 anconda 安装包上传至 mynode4 节点，进行安装

```
sh Anaconda3-2020.02-Linux-x86_64.sh  【一路回车即可】
Do you accept the license terms? [yes|no]
Yes【继续回车】
... ...
Anaconda3 will now be installed into this location:
/root/anaconda3

  - Press ENTER to confirm the location
  - Press CTRL-C to abort the installation
  - Or specify a different location below

[/root/anaconda3] >>> 【回车即可，安装到/root/anaconda3 路径下】
... ...
Do you wish the installer to initialize Anaconda3
by running conda init? [yes|no]
[no] >>>yes【输入 yes,回车即可】
... ...
【安装完成】
```

3． 配置 Anconda 的环境变量

```
在 /etc/profile 中加入以下语句:
export PATH=$PATH:/root/anaconda3/bin
#使环境变量生效
source /etc/profile
```

4. 安装 python3.7 python 环境

```
conda create -n python37 python=3.7
```

5. 激活使用 python37 python 环境

```
conda activate python37【激活使用 python37 环境，需要先执行下 source activate】
```

相关命令如下：

```
source activate 【初始化 conda,必须执行，执行之后可以使用 conda 命令激活环境】
conda deactivate 【退出当前 base 环境】
conda activate python37【激活使用 python37 环境】
conda deactivate 【退出当前使用 python37 环境】
conda remove -n python37 --all 【删除 python37 环境】
```

1.3.2 单机安装 Airflow

单节点部署 airflow 时，所有 airflow 进程都运行在一台机器上，架构图如下：

1. 安装 Airflow 必须需要的系统依赖

Airflow 正常使用必须需要一些系统依赖，在 mynode4 节点上安装以下依赖：

```
yum -y install mysql-devel gcc gcc-devel python-devel gcc-c++ cyrus-sasl
cyrus-sasl-devel cyrus-sasl-lib
```

2. 在 MySQL 中创建对应的库并设置参数

aiflow 使用的 Metadata database 我们这里使用 mysql，在 node2 节点的 mysql 中创建 airflow 使用的库及表信息。

```
CREATE DATABASE airflow CHARACTER SET utf8;
create user 'airflow'@'%' identified by '123456';
grant all privileges on airflow.* to 'airflow'@'%';
flush privileges;
```

在 mysql 安装节点 node2 上修改"/etc/my.cnf"，在[mysqld]下添加如下内容：

```
[mysqld]
explicit_defaults_for_timestamp=1
```

注意：以上配置 explicit_defaults_for_timestamp 系统变量决定 MySQL 服务端对 timestamp 列中的默认值和 NULL 值的不同处理方法。此变量自 MySQL 5.6.6 版本引入，默认值为 0，在默认情况下，如果 timestamp 列没有显式的指明 null 属性，那么该列会被自动加上 not null 属性，如果往这个列中插入 null 值，会自动的设置该列的值为 current timestamp 值。当这个值被设置为 1 时，如果 timestamp 列没有显式的指定 not null 属性，那么默认的该列可以为 null，此时向该列中插入 null 值时，会直接记录 null，而不是 current timestamp，如果指定 not null 就会报错。

在 Airflow 中需要对应 mysql 这个参数设置为 1。以上修改完成"my.cnf"值后，重启 Mysql 即可，重启之后，可以查询对应的参数是否生效：

```
#重启 mysql
[root@node2 ~]# service mysqld restart

#重新登录 mysql 查询
mysql> show variables like 'explicit_defaults_for_timestamp';
```

```
mysql> show variables like 'explicit_defaults_for_timestamp';
+--------------------------------+-------+
| Variable_name                  | Value |
+--------------------------------+-------+
| explicit_defaults_for_timestamp | ON    |
+--------------------------------+-------+
1 row in set (0.02 sec)
```

3. 安装 Airflow

在 node4 上切换 python37 环境，安装 airflow，指定版本为 2.1.3

```
(python37) [root@node4 ~]# conda activate python37
(python37) [root@node4 ~]# pip install apache-airflow==2.1.3 -i https://pypi.tuna.tsinghua.edu.cn/simple
```

默认 Airflow 安装在 $ANCONDA_HOME/envs/python37/lib/python3.7/site-packages/airflow 目录下。Airflow 文件存储目录默认在/root/airflow 目录下，但是这个目录需要执行下“airflow version”后自动创建，查看安装 Airflow 版本信息：

```
(python37) [root@node4 ~]# airflow version
2.1.3
```

注意：如果不想使用默认的“/root/airflow”目录当做文件存储目录，也可以在安装 airflow 之前设置环境变量：

```
(python37) [root@node4 ~]# vim /etc/profile
export AIRFLOW_HOME=/software/airflow

#使配置的环境变量生效
source /etc/profile
```

这样安装完成的 airflow 后，查看对应的版本会将“AIRFLOW_HOME”配置的目录当做 airflow 的文件存储目录。

4. 配置 Airflow 使用的数据库为 MySQL

打开配置的 airflow 文件存储目录，默认在 $AIRFLOW_HOME 目录“/root/airflow”中，会有“airflow.cfg”配置文件，修改配置如下：

```
[core]
dags_folder = /root/airflow/dags

#修改时区
default_timezone = Asia/Shanghai

# 配置数据库
sql_alchemy_conn=mysql+mysqldb://airflow:123456@node2:3306/airflow?use_unicode=true&charset=utf8

[webserver]
#设置时区
```

```
default_ui_timezone = Asia/Shanghai

#设置 DAG 显示方式
# Default DAG view. Valid values are: ``tree``, ``graph``, ``duration``, ``gantt``, ``landing_times``
dag_default_view = graph

[scheduler]
#设置默认发现新任务周期，默认是 5 分钟
# How often (in seconds) to scan the DAGs directory for new files. Default to 5 minutes.
dag_dir_list_interval = 30
```

5. 安装需要的 python 依赖包

初始化 Airflow 数据库时需要使用到连接 mysql 的包，执行如下命令来安装 mysql 对应的 python 包。

```
(python37) [root@node4 ~]# pip install mysqlclient -i https://pypi.tuna.tsinghua.edu.cn/simple
```

6. 初始化 Airflow 数据库

```
(python37) [root@node4 airflow]# airflow db init
```

初始化之后在 MySQL airflow 库下会生成对应的表。

7. 创建管理员用户信息

在 node4 节点上执行如下命令，创建操作 Airflow 的用户信息：

```
airflow users create \
    --username airflow \
    --firstname airflow \
    --lastname airflow \
    --role Admin \
    --email xx@qq.com
```

执行完成之后，设置密码为“123456”并确认，完成 Airflow 管理员信息创建。

1.3.3 启动 Airflow

1. 启动 webserver

```
#前台方式启动 webserver
(python37) [root@node4 airflow]# airflow webserver --port 8080

#以守护进程方式运行 webserver,端口默认 8080。 ps aux|grep webserver 查看后台进程
airflow webserver --port 8080 -D
```

2. 启动 scheduler

新开窗口，切换 python37 环境，启动 Schduler：

```
#前台方式启动 scheduler
(python37) [root@node4 ~]# airflow scheduler

#以守护进程方式运行 Scheduler,ps aux|grep scheduler 查看后台进程
 airflow scheduler -D
```

3. 访问 Airflow webui

浏览器访问：http://node4:8080

输入前面创建的用户名：airflow 密码：123456

1.4 Airflow WebUI 操作介绍

1.4.1 DAG

DAG 有对应的 id,其 id 全局唯一 ,DAG 是 airflow 的核心概念 ,任务装载到 DAG 中，封装成任务依赖链条，DAG 决定这些任务的执行规则。

点击以上每个 DAG 对应的 id 可以直接进入对应“Graph View”视图，可以查看当前 DAG 任务执行顺序图。

以上“Runs”列与“Recent Tasks”列下的“圆圈”代表当前 DAG 执行的某种状态，鼠标放到对应的“圆圈”上可以查看对应的提示说明。点击以上“Links”之后，出现以下选项：

- **Tree View**

将 DAG 以树的形式表示，如果执行过程中有延迟也可以通过这个界面查看问题出现在哪个步骤，在生产环境下，经常通过这个页面查看每个任务执行情况。

点击以上每个有颜色的“小块”都可以看到 task 详情：

- **Graph View**

此页面以图形方式呈现 DAG 有向无环图，对于理解 DAG 执行非常有帮助，不同颜色代表 task 执行的不同状态。

点击任意一个 task，都可以看到当前 task 执行情况：

- **Calendar View**

  日期视图，显示当前年每月每天任务执行情况。

- **Task Duration**

  此视图表示不同的 task 在过去每天执行的时长，可以通过每日执行时长对比，发现同一个 task 执行耗时情况。

- **Task Tries**

  此视图显示每个 task 重试次数情况。

- **Landing Times**

Landing Times 显示每个任务实际执行完成时间减去该 task 定时设置调度的时间，得到的小时数，可以通过这个图看出任务每天执行耗时、延迟情况。

以上得到的“Landing Times”如下：

- **Gantt**

甘特图，可以通过甘特图来分析 task 执行持续时间和重叠情况，可以直观看出哪些 task 执行时间长。

- **Details**

  可以通过“Details”发现任务详细情况。

- **Code**

  Code 页面主要显示当前 DAG python 代码编码，当前 DAG 如何运行以及任务依赖关系、执行成功失败做什么，都可以在代码中进行定义。

1.4.2 Security

“Security”涉及到 Airflow 中用户、用户角色、用户状态、权限等配置。

1.4.3 Browse

- **DAG Runs**

  显示所有 DAG 状态

- **Jobs**

显示 Airflow 中运行的 DAG 任务

- **Audit Logs**

  审计日志，查看所有 DAG 下面对应的 task 的日志，并且包含检索。

- **Task Instances**

  查看每个 task 实例执行情况。

- **Task Reschedules**

  Task 重新调度的实例情况。

- **SLA Misses**

  如果有一个或者多个实例未成功，则会发送报警电子邮件，此选项页面记录这些事件。

- **DAG Dependencies**

  查看 DAG 任务对应依赖关系。

1.4.4 Admin

在 Admin 标签下可以定义 Airflow 变量、配置 Airflow、配置外部连接等。

1.4.5 Docs

Docs 中是关于用户使用 Airflow 的一些官方使用说明文档连接。

1.5 Airflow 使用

上文说到使用 Airflow 进行任务调度大体步骤如下：

1） 创建 python 文件，根据实际需要，使用不同的 Operator

2） 在 python 文件不同的 Operator 中传入具体参数，定义一系列 task

3） 在 python 文件中定义 Task 之间的关系，形成 DAG

4） 将 python 文件上传执行，调度 DAG，每个 task 会形成一个 Instance

5） 使用命令行或者 WEBUI 进行查看和管理

以上 python 文件就是 Airflow python 脚本，使用代码方式指定 DAG 的结构。

1.5.1 Airflow 调度 Shell 命令

下面我们以调度执行 shell 命令为例，来讲解 Airflow 使用。

1） 首先我们需要创建一个 python 文件，导入需要的类库

```
# 导入 DAG 对象，后面需要实例化 DAG 对象

from airflow import DAG


# 导入 BashOperator Operators,我们需要利用这个对象去执行流程

from airflow.operators.bash import BashOperator
```

注意：以上代码可以在开发工具中创建，但是需要在使用的 python3.7 环境中导入安装 Airflow 包。

```
D:\ProgramData\Anaconda3\envs\python37\Scripts>pip install apache-airflow==2.1.3 -i https://pypi.tuna.tsinghua.edu.cn/simple
```

2） 实例化 DAG

```
from datetime import datetime, timedelta


# default_args 中定义一些参数，在实例化 DAG 时可以使用，使用 python dic 格式定义

default_args = {

    'owner': 'airflow', # 拥有者名称

    'start_date': datetime(2021, 9, 4),  # 第一次开始执行的时间，为 UTC 时间

    'retries': 1,  # 失败重试次数
```

```
    'retry_delay': timedelta(minutes=5),  # 失败重试间隔
}


dag = DAG(
    dag_id = 'myairflow_execute_bash', #DAG id ,必须完全由字母、数字、下划线组成
    default_args = default_args, #外部定义的 dic 格式的参数
    schedule_interval = timedelta(days=1) # 定义 DAG 运行的频率,可以配置天、周、小时、分钟、秒、
毫秒
)
```

注意：

◆ 实例化 DAG 有三种方式

第一种方式：

```
with DAG("my_dag_name") as dag:
    op=XXOperator(task_id="task")
```

第二种方式（以上采用这种方式）：

```
my_dag = DAG("my_dag_name")
op = XXOperator(task_id="task", dag=my_dag)
```

第三种方式：

```
@dag(start_date=days_ago(2))
def generate_dag():
    op = XXOperator(task_id="task")
dag = generate_dag()
```

◆ baseoperator 基础参数说明：

可以参照：

http://airflow.apache.org/docs/apache-airflow/stable/_api/airflow/models/baseoperator/index.html#module-airflow.models.baseoperator 查看 baseopartor 中更多参数。

◆ DAG 参数说明

可以参照：

http://airflow.apache.org/docs/apache-airflow/stable/_api/airflow/models/dag/index.html 查看 DAG 参数说明，也可以直接在开发工具点击 DAG 进入源码看下对应参数有哪些。

3） 定义 Task

当实例化 Operator 时会生成 Task 任务，从一个 Operator 中实例化出来对象的过程被称为一个构造方法，每个构造方法中都有“task_id”充当任务的唯一标识符。

下面我们定义三个 Operator，也就是三个 Task，每个 task_id 不能重复。

```
# operator 支持多种类型， 这里使用 BashOperator
first = BashOperator(
    task_id='first',
    bash_command='echo "run first task"',
    dag=dag
```

```
)


middle = BashOperator(
    task_id='middle',
    bash_command='echo "run middle task"',
    dag=dag
)


last = BashOperator(
    task_id='last',
    bash_command='echo "run last task"',
    dag=dag,
    retries=3
)
```

注意：

- 每个 operator 中可以传入对应的参数，覆盖 DAG 默认的参数，例如：last task 中“retries” =3 就替代了默认的 1。任务参数的优先规则如下：①.显示传递的参数 ②.default_args 字典中存在的值③.operator 的默认值（如果存在）。
- BashOperator 使用方式参照：http://airflow.apache.org/docs/apache-airflow/stable/howto/operator/bash.html#howto-operator-bashoperator

4） 设置 task 依赖关系

```
#使用 set_upstream、set_downstream 设置依赖关系,不能出现环形链路，否则报错
# middle.set_upstream(first) # middle 会在 first 执行完成之后执行
# last.set_upstream(middle) # last 会在 middle 执行完成之后执行


#也可以使用位移符来设置依赖关系
```

```
first >> middle >>last # first 首先执行，middle 次之，last 最后

# first >> [middle,last] # first 首先执行，middle ,last 并行执行
```

注意：当执行脚本时，如果在 DAG 中找到一条环形链路（例如：A->B->C-A）会引发异常。更多 DAG task 依赖关系可参照官网：http://airflow.apache.org/docs/apache-airflow/stable/concepts/dags.html#task-dependencies

5） 上传 python 配置脚本

到目前为止，python 配置如下：

```
# 导入 DAG 对象，后面需要实例化 DAG 对象

from airflow import DAG


# 导入 BashOperator Operators,我们需要利用这个对象去执行流程

from airflow.example_dags.example_bash_operator import dag


from airflow.operators.bash import BashOperator


from datetime import datetime, timedelta


# default_args 中定义一些参数，在实例化 DAG 时可以使用，使用 python dic 格式定义

default_args = {

    'owner': 'airflow', # 拥有者名称

    'start_date': datetime(2021, 9, 4),  # 第一次开始执行的时间，为 UTC 时间

    'retries': 1,  # 失败重试次数

    'retry_delay': timedelta(minutes=5),  # 失败重试间隔

}
```

```
dag = DAG(
    dag_id = 'myairflow_execute_bash', #DAG id ,必须完全由字母、数字、下划线组成
    default_args = default_args, #外部定义的 dic 格式的参数
    schedule_interval = timedelta(days=1) # 定义 DAG 运行的频率,可以配置天、周、小时、分钟、秒、毫秒
)

# operator 支持多种类型， 这里使用 BashOperator
first = BashOperator(
    task_id='first',
    bash_command='echo "run first task"',
    dag=dag
)

middle = BashOperator(
    task_id='middle',
    bash_command='echo "run middle task"',
    dag=dag
)

last = BashOperator(
    task_id='last',
    bash_command='echo "run last task"',
    dag=dag,
    retries=3
)

#使用 set_upstream、set_downstream 设置依赖关系,不能出现环形链路，否则报错
```

```
# middle.set_upstream(first) # middle 会在 first 执行完成之后执行

# last.set_upstream(middle) # last 会在 middle 执行完成之后执行


#也可以使用位移符来设置依赖关系

first >> middle >>last # first 首先执行，middle 次之，last 最后

# first >> [middle,last] # first 首先执行，middle ，last 并行执行
```

将以上 python 配置文件上传到 $AIRFLOW_HOME/dags 目录下，默认 $AIRFLOW_HOME 为安装节点的“/root/airflow”目录，当前目录下的 dags 目录需要手动创建。

6） 重启 Airflow

“ps aux|grep webserver”和“ps aux|grep scheduler”找到对应的 airflow 进程杀掉，重新启动 Airflow。重启之后，可以在 airflow webui 看到对应的 DAG ID ”myairflow_execute_bash”。

7） 执行 airflow

按照如下步骤执行 DAG，首先打开工作流，然后“Trigger DAG”执行，随后可以看到任务执行成功。

查看 task 执行日志：

1.5.2 DAG 调度触发时间

在 Airflow 中，调度程序会根据 DAG 文件中指定的“start_date”和“schedule_interval”来运行 DAG。特别需要注意的是 Airflow 计划程序在计划时间段的末尾触发执行 DAG，而不是在开始时刻触发 DAG，例如：

```
default_args = {
    'owner': 'airflow', # 拥有者名称
    'start_date': datetime(2021, 9, 4),  # 第一次开始执行的时间，为 UTC 时间
    'retries': 1,  # 失败重试次数
    'retry_delay': timedelta(minutes=5),  # 失败重试间隔
}

dag = DAG(
    dag_id = 'myairflow_execute_bash', #DAG id ,必须完全由字母、数字、下划线组成
    default_args = default_args, #外部定义的 dic 格式的参数
    schedule_interval = timedelta(days=1) # 定义 DAG 运行的频率,可以配置天、周、小时、分钟、秒、毫秒
)
```

以上配置的 DAG 是从世界标准时间 2021 年 9 月 4 号开始调度，每隔 1 天执行一次，这个 DAG 的具体运行时间如下图：

自动调度 DAG 执行日期	自动调度 DAG 实际执行触发时间
2021-09-04，00:00:00+00:00	2021-09-05，00:00:00+00:00
2021-09-05，00:00:00+00:00	2021-09-06，00:00:00+00:00
2021-09-06，00:00:00+00:00	2021-09-07，00:00:00+00:00
2021-09-07，00:00:00+00:00	2021-09-08，00:00:00+00:00
2021-09-08，00:00:00+00:00	2021-09-09，00:00:00+00:00
……	……

以上表格中以第一条数据为例解释，Airflow 正常调度是每天 00:00:00 ，假设当天日期为 2021-09-04，正常我们认为只要时间到了 2021-09-04 00:00:00 就会执行，改调度时间所处于的调度周期为 2021-09-04 00:00:00 ~ 2021-09-05 00:00:00 ，在 Airflow 中实际上是在调度周期末端触发执行，也就是说 2021-09-04 00:00:00 自动触发执行时刻为 2021-09-05 00:00:00。

如下图，在 airflow 中，“execution_date”不是实际运行时间，而是其计划周期的开始时间戳。例如：execution_date 是 2021-09-04 00:00:00 的 DAG 自动调度运行的实际时间为 2021-09-05 00:00:00。当然除了自动调度外，我们还可以手动触发执行 DAG 执行，要判断 DAG 运行时计划调度（自动调度）还是手动触发，可以查看“Run Type”。

1.5.3 DAG catchup 参数设置

在 Airflow 的工作计划中，一个重要的概念就是 catchup（追赶），在实现 DAG 具体逻辑后，如果将 catchup 设置为 True（默认就为 True），Airflow 将“回填”所有过去的 DAG run，如果将 catchup 设置为 False，Airflow 将从最新的 DAG run 时刻前一时刻开始执行 DAG run，忽略之前所有的记录。

例如：现在某个 DAG 每隔 1 分钟执行一次，调度开始时间为 2001-01-01 ，当前日期为 2021-10-01 15:23:21 ,如果 catchup 设置为 True ,那么 DAG 将从 2001-01-01 00:00:00 开始每分钟都会运行当前 DAG。如果 catchup 设置为 False，那么 DAG 将从 2021-10-01 15:22:20（当前 2021-10-01 15:23:21 前一时刻）开始执行 DAG run。

举例：有 first ，second，third 三个 shell 命令任务，按照顺序调度，每隔 1 分钟执行一次，首次执行时间为 2000-01-01。

设置 catchup 为 True（默认），DAG python 配置如下：

```
from airflow import DAG

from airflow.operators.bash import BashOperator

from datetime import datetime, timedelta
```

```
default_args = {
    'owner': 'airflow', # 拥有者名称
    'start_date': datetime(2001, 1, 1),  # 第一次开始执行的时间，为 UTC 时间
    'retries': 1,  # 失败重试次数
    'retry_delay': timedelta(minutes=5),  # 失败重试间隔
}
dag = DAG(
    dag_id = 'catchup_test1 ', #DAG id ,必须完全由字母、数字、下划线组成
    default_args = default_args, #外部定义的 dic 格式的参数
    schedule_interval = timedelta(minutes=1), # 定义 DAG 运行的频率,可以配置天、周、小时、分钟、秒、毫秒
    catchup=True # 执行 DAG 时，将开始时间到目前所有该执行的任务都执行，默认为 True
)

first = BashOperator(
    task_id='first',
    bash_command='echo "run first task"',
    dag=dag
)
middle = BashOperator(
    task_id='second',
    bash_command='echo "run second task"',
    dag=dag
)
last = BashOperator(
    task_id='third',
    bash_command='echo "run third task"',
    dag=dag,
    retries=3
)
first >> middle >>last
```

上传 python 配置文件到$AIRFLOW_HOME/dags 下，重启 airflow,DAG 执行调度如下：

State	Dag Id	Execution Date	Run Id	Run Type	Queued At	Start Date	End Date	External Trigger	Conf
success	catchup_test1	2001-01-01, 00:18:00	scheduled__2000-12-31T16:18:00+00:00	scheduled	2021-09-22 12:19:27+00:00	2021-09-22, 20:19:27	2021-09-22, 20:19:41	False	{}
success	catchup_test1	2001-01-01, 00:17:00	scheduled__2000-12-31T16:17:00+00:00	scheduled	2021-09-22 12:19:22+00:00	2021-09-22, 20:19:22	2021-09-22, 20:19:36	False	{}
success	catchup_test1	2001-01-01, 00:16:00	scheduled__2000-12-31T16:16:00+00:00	scheduled	2021-09-22 12:19:18+00:00	2021-09-22, 20:19:17	2021-09-22, 20:19:32	False	{}
success	catchup_test1	2001-01-01, 00:15:00	scheduled__2000-12-31T16:15:00+00:00	scheduled	2021-09-22 12:19:12+00:00	2021-09-22, 20:19:12	2021-09-22, 20:19:27	False	{}
success	catchup_test1	2001-01-01, 00:14:00	scheduled__2000-12-31T16:14:00+00:00	scheduled	2021-09-22 12:19:07+00:00	2021-09-22, 20:19:07	2021-09-22, 20:19:22	False	{}
success	catchup_test1	2001-01-01, 00:13:00	scheduled__2000-12-31T16:13:00+00:00	scheduled	2021-09-22 12:19:03+00:00	2021-09-22, 20:19:02	2021-09-22, 20:19:17	False	{}
success	catchup_test1	2001-01-01, 00:12:00	scheduled__2000-12-31T16:12:00+00:00	scheduled	2021-09-22 12:18:58+00:00	2021-09-22, 20:18:58	2021-09-22, 20:19:12	False	{}
success	catchup_test1	2001-01-01, 00:11:00	scheduled__2000-12-31T16:11:00+00:00	scheduled	2021-09-22 12:18:54+00:00	2021-09-22, 20:18:53	2021-09-22, 20:19:07	False	{}
success	catchup_test1	2001-01-01, 00:10:00	scheduled__2000-12-31T16:10:00+00:00	scheduled	2021-09-22 12:18:49+00:00	2021-09-22, 20:18:48	2021-09-22, 20:19:02	False	{}
success	catchup_test1	2001-01-01, 00:09:00	scheduled__2000-12-31T16:09:00+00:00	scheduled	2021-09-22 12:18:44+00:00	2021-09-22, 20:18:43	2021-09-22, 20:18:58	False	{}
success	catchup_test1	2001-01-01, 00:08:00	scheduled__2000-12-31T16:08:00+00:00	scheduled	2021-09-22 12:18:38+00:00	2021-09-22, 20:18:38	2021-09-22, 20:18:53	False	{}
success	catchup_test1	2001-01-01, 00:07:00	scheduled__2000-12-31T16:07:00+00:00	scheduled	2021-09-22 12:18:33+00:00	2021-09-22, 20:18:32	2021-09-22, 20:18:48	False	{}
success	catchup_test1	2001-01-01, 00:06:00	scheduled__2000-12-31T16:06:00+00:00	scheduled	2021-09-22 12:18:27+00:00	2021-09-22, 20:18:27	2021-09-22, 20:18:43	False	{}
success	catchup_test1	2001-01-01, 00:05:00	scheduled__2000-12-31T16:05:00+00:00	scheduled	2021-09-22 12:18:23+00:00	2021-09-22, 20:18:22	2021-09-22, 20:18:38	False	{}
success	catchup_test1	2001-01-01, 00:04:00	scheduled__2000-12-31T16:04:00+00:00	scheduled	2021-09-22 12:18:18+00:00	2021-09-22, 20:18:17	2021-09-22, 20:18:32	False	{}
success	catchup_test1	2001-01-01, 00:03:00	scheduled__2000-12-31T16:03:00+00:00	scheduled	2021-09-22 12:18:13+00:00	2021-09-22, 20:18:12	2021-09-22, 20:18:27	False	{}
success	catchup_test1	2001-01-01, 00:02:00	scheduled__2000-12-31T16:02:00+00:00	scheduled	2021-09-22 12:18:07+00:00	2021-09-22, 20:18:07	2021-09-22, 20:18:22	False	{}
success	catchup_test1	2001-01-01, 00:01:00	scheduled__2000-12-31T16:01:00+00:00	scheduled	2021-09-22 12:18:04+00:00	2021-09-22, 20:18:03	2021-09-22, 20:18:17	False	{}
success	catchup_test1	2001-01-01, 00:00:00	scheduled__2000-12-31T16:00:00+00:00	scheduled	2021-09-22 12:18:02+00:00	2021-09-22, 20:18:01	2021-09-22, 20:18:12	False	{}

设置 catchup 为 False，DAG python 配置如下：

```
from airflow import DAG

from airflow.operators.bash import BashOperator

from datetime import datetime, timedelta

default_args = {
    'owner': 'airflow', # 拥有者名称
    'start_date': datetime(2001, 1, 1),  # 第一次开始执行的时间，为 UTC 时间
    'retries': 1,  # 失败重试次数
    'retry_delay': timedelta(minutes=5),  # 失败重试间隔
}
dag = DAG(
    dag_id = 'catchup_test2', #DAG id ，必须完全由字母、数字、下划线组成
    default_args = default_args, #外部定义的 dic 格式的参数
    schedule_interval = timedelta(minutes=1), # 定义 DAG 运行的频率，可以配置天、周、小时、分钟、秒、毫秒
    catchup=False # 执行 DAG 时，将开始时间到目前所有该执行的任务都执行，默认为 True
)

first = BashOperator(
    task_id='first',
    bash_command='echo "run first task"',
    dag=dag
)
middle = BashOperator(
    task_id='second',
    bash_command='echo "run second task"',
```

```
    dag=dag
)
last = BashOperator(
    task_id='third',
    bash_command='echo "run third task"',
    dag=dag,
    retries=3
)
first >> middle >>last
```

上传 python 配置文件到$AIRFLOW_HOME/dags 下，重启 airflow，DAG 执行调度如下：

有两种方式在 Airflow 中配置 catchup：

- **全局配置**

在 airflow 配置文件 airflow.cfg 的 scheduler 部分下，设置 catchup_by_default=True（默认）或 False，这个设置是全局性的设置。

- **DAG 文件配置**

在 python 代码配置中设置 DAG 对象的参数：dag.catchup=True 或 False。

```
dag = DAG(
    dag_id = 'myairflow_execute_bash',
    default_args = default_args,
    catchup=False,
    schedule_interval = timedelta(days=1))
```

1.5.4 DAG 调度周期设置

每个 DAG 可以有或者没有调度执行周期，如果有调度周期，我们可以在 python 代码 DAG 配置中设置“schedule_interval”参数来指定调度 DAG 周期，可以通过以下三种方式来设置。

- **预置的 Cron 调度**

Airflow 预置了一些 Cron 调度周期，可以参照：

http://airflow.apache.org/docs/apache-airflow/stable/dag-run.html#cron-presets，如下图：

preset	meaning	cron
None	Don't schedule, use for exclusively "externally triggered" DAGs	
@once	Schedule once and only once	
@hourly	Run once an hour at the beginning of the hour	0 * * * *
@daily	Run once a day at midnight	0 0 * * *
@weekly	Run once a week at midnight on Sunday morning	0 0 * * 0
@monthly	Run once a month at midnight of the first day of the month	0 0 1 * *
@quarterly	Run once a quarter at midnight on the first day	0 0 1 */3 *
@yearly	Run once a year at midnight of January 1	0 0 1 1 *

在 python 配置文件中使用如下：

```
default_args = {
    'owner': 'airflow', # 拥有者名称
    'start_date': datetime(2021, 9, 4),  # 第一次开始执行的时间，为 UTC 时间
    'retries': 1,  # 失败重试次数
    'retry_delay': timedelta(minutes=5),  # 失败重试间隔
}

dag = DAG(
    dag_id = 'cron_test', #DAG id，必须完全由字母、数字、下划线组成
    default_args = default_args, #外部定义的 dic 格式的参数
    schedule_interval = '@daily' # 使用预置的 Cron 调度，每天 0 点 0 分调度
)
```

- **Cron**

这种方式就是写 Linux 系统的 crontab 定时任务命令，可以在 https://crontab.guru/网站先生成对应的定时调度命令，其格式如下：

minute hour day month week
minute:表示分钟，可以从 0~59 之间的任意整数。
hour:表示小时，可以是从 0 到 23 之间的任意整数。
day:表示日期，可以是 1 到 31 之间的任何整数。
month:表示月份，可以是从 1 到 12 之间的任何整数。
week:表示星期几，可以是从 0 到 7 之间的任何整数，这里的 0 或 7 代表星期日。

以上各个字段中还可以使用特殊符号代表不同意思：

星号（*）：代表所有可能的值，例如 month 字段如果是星号，则表示在满足其它字段的制约条件后每月都执行该命令操作。
逗号（，）：可以用逗号隔开的值指定一个列表范围，例如，"1,2,5,7,8,9"
中杠（-）：可以用整数之间的中杠表示一个整数范围，例如"2-6"表示"2,3,4,5,6"
正斜线（/）：可以用正斜线指定时间的间隔频率，步长，例如"0-23/2"表示每两小时执行一次。

在 python 配置文件中使用如下：

```
default_args = {
    'owner': 'airflow', # 拥有者名称
    'start_date': datetime(2021, 9, 4),  # 第一次开始执行的时间，为 UTC 时间
    'retries': 1,  # 失败重试次数
    'retry_delay': timedelta(minutes=5),  # 失败重试间隔
}

dag = DAG(
    dag_id = 'cron_test', #DAG id，必须完全由字母、数字、下划线组成
    default_args = default_args, #外部定义的 dic 格式的参数
```

```
    schedule_interval = '* * * * *' # 使用 Crontab 定时任务命令，每分钟运行一次
)
```

- **datetime.timedelta**

timedelta 是使用 python timedelta 设置调度周期，可以配置天、周、小时、分钟、秒、毫秒。在 python 配置文件中使用如下：

```
default_args = {
    'owner': 'airflow', # 拥有者名称
    'start_date': datetime(2021, 9, 4),  # 第一次开始执行的时间，为 UTC 时间
    'retries': 1,  # 失败重试次数
    'retry_delay': timedelta(minutes=5),  # 失败重试间隔
}

dag = DAG(
    dag_id = 'cron_test', #DAG id ,必须完全由字母、数字、下划线组成
    default_args = default_args, #外部定义的 dic 格式的参数
    schedule_interval = timedelta(minutes=5) # 使用 python timedelta 设置调度周期，可以配置天、周、小时、分钟、秒、毫秒
)
```

1.5.5 DAG 任务依赖设置

1.5.5.1 DAG 任务依赖设置一

- **DAG 调度流程图**

- **task 执行依赖**

```
A >> B >>C
```

- **完整代码**

```
'''
airflow 任务依赖关系设置一

'''
from airflow import DAG

from airflow.operators.bash import BashOperator

from datetime import datetime, timedelta
default_args = {
    'owner': 'airflow', # 拥有者名称
    'start_date': datetime(2021, 9, 22),  # 第一次开始执行的时间，为 UTC 时间
    'retries': 1,  # 失败重试次数
    'retry_delay': timedelta(minutes=5),  # 失败重试间隔
}

dag = DAG(
    dag_id = 'dag_relation_1', #DAG id ,必须完全由字母、数字、下划线组成
    default_args = default_args, #外部定义的 dic 格式的参数
    schedule_interval = timedelta(minutes=1) # 定义 DAG 运行的频率,可以配置天、周、小时、分钟、
秒、毫秒
)


A = BashOperator(
    task_id='A',
    bash_command='echo "run A task"',
    dag=dag
)
```

```
B = BashOperator(
    task_id='B',
    bash_command='echo "run B task"',
    dag=dag
)

C = BashOperator(
    task_id='C',
    bash_command='echo "run C task"',
    dag=dag,
    retries=3
)

A >> B >>C
```

1.5.5.2 DAG 任务依赖设置二

- **DAG 调度流程图**

- **task 执行依赖**

```
[A,B] >>C >>D
```

- **完整代码**

```
'''
airflow 任务依赖关系设置二

'''
from airflow import DAG

from airflow.operators.bash import BashOperator
```

```
from datetime import datetime, timedelta
default_args = {
    'owner': 'airflow', # 拥有者名称
    'start_date': datetime(2021, 9, 22),  # 第一次开始执行的时间，为 UTC 时间
    'retries': 1,  # 失败重试次数
    'retry_delay': timedelta(minutes=5),  # 失败重试间隔
}

dag = DAG(
    dag_id = 'dag_relation_2', #DAG id ,必须完全由字母、数字、下划线组成
    default_args = default_args, #外部定义的 dic 格式的参数
    schedule_interval = timedelta(minutes=1) # 定义 DAG 运行的频率,可以配置天、周、小时、分钟、秒、毫秒
)


A = BashOperator(
    task_id='A',
    bash_command='echo "run A task"',
    dag=dag
)

B = BashOperator(
    task_id='B',
    bash_command='echo "run B task"',
    dag=dag
)

C = BashOperator(
    task_id='C',
    bash_command='echo "run C task"',
    dag=dag,
    retries=3
)

D = BashOperator(
    task_id='D',
    bash_command='echo "run D task"',
    dag=dag
)

[A,B] >>C >>D
```

1.5.5.3 DAG 任务依赖设置三

- **DAG 调度流程图**

- **task 执行依赖**

```
[A,B,C] >>D >>[E,F]
```

- **完整代码**

```
'''
airflow 任务依赖关系设置三

'''
from airflow import DAG

from airflow.operators.bash import BashOperator

from datetime import datetime, timedelta
default_args = {
    'owner': 'airflow', # 拥有者名称
    'start_date': datetime(2021, 9, 22),  # 第一次开始执行的时间，为 UTC 时间
    'retries': 1,  # 失败重试次数
    'retry_delay': timedelta(minutes=5),  # 失败重试间隔
}

dag = DAG(
    dag_id = 'dag_relation_3', #DAG id ,必须完全由字母、数字、下划线组成
    default_args = default_args, #外部定义的 dic 格式的参数
    schedule_interval = timedelta(minutes=1) # 定义 DAG 运行的频率,可以配置天、周、小时、分钟、
```

```
秒、毫秒
)


A = BashOperator(
    task_id='A',
    bash_command='echo "run A task"',
    dag=dag
)

B = BashOperator(
    task_id='B',
    bash_command='echo "run B task"',
    dag=dag
)

C = BashOperator(
    task_id='C',
    bash_command='echo "run C task"',
    dag=dag,
    retries=3
)

D = BashOperator(
    task_id='D',
    bash_command='echo "run D task"',
    dag=dag
)

E = BashOperator(
    task_id='E',
    bash_command='echo "run E task"',
    dag=dag
)

F = BashOperator(
    task_id='F',
    bash_command='echo "run F task"',
    dag=dag
)

[A,B,C] >>D >>[E,F]
```

1.5.5.4 DAG 任务依赖设置四

- **DAG 调度流程图**

- **task 执行依赖**

```
A >>B>>C>>D

A >>E>>F
```

- **完整代码**

```
'''
airflow 任务依赖关系设置四

'''
from airflow import DAG

from airflow.operators.bash import BashOperator

from datetime import datetime, timedelta
default_args = {
    'owner': 'airflow', # 拥有者名称
    'start_date': datetime(2021, 9, 22),  # 第一次开始执行的时间，为 UTC 时间
    'retries': 1,  # 失败重试次数
    'retry_delay': timedelta(minutes=5),  # 失败重试间隔
}

dag = DAG(
    dag_id = 'dag_relation_4', #DAG id ,必须完全由字母、数字、下划线组成
    default_args = default_args, #外部定义的 dic 格式的参数
    schedule_interval = timedelta(minutes=1) # 定义 DAG 运行的频率,可以配置天、周、小时、分钟、秒、毫秒
)
```

```
A = BashOperator(
    task_id='A',
    bash_command='echo "run A task"',
    dag=dag
)

B = BashOperator(
    task_id='B',
    bash_command='echo "run B task"',
    dag=dag
)

C = BashOperator(
    task_id='C',
    bash_command='echo "run C task"',
    dag=dag,
    retries=3
)

D = BashOperator(
    task_id='D',
    bash_command='echo "run D task"',
    dag=dag
)

E = BashOperator(
    task_id='E',
    bash_command='echo "run E task"',
    dag=dag
)

F = BashOperator(
    task_id='F',
    bash_command='echo "run F task"',
    dag=dag
)

A >>[B,C,D]
A >>[E,F]
```

1.5.5.5 DAG 任务依赖设置五

- **DAG 调度流程图**

- **task 执行依赖**

```
A >>B>>E
C >>D>>E
```

- **完整代码**

```
'''
airflow 任务依赖关系设置五

'''
from airflow import DAG

from airflow.operators.bash import BashOperator

from datetime import datetime, timedelta
default_args = {
    'owner': 'airflow', # 拥有者名称
    'start_date': datetime(2021, 9, 22),  # 第一次开始执行的时间，为 UTC 时间
    'retries': 1,  # 失败重试次数
    'retry_delay': timedelta(minutes=5),  # 失败重试间隔
}

dag = DAG(
    dag_id = 'dag_relation_5', #DAG id ,必须完全由字母、数字、下划线组成
    default_args = default_args, #外部定义的 dic 格式的参数
    schedule_interval = timedelta(minutes=1) # 定义 DAG 运行的频率,可以配置天、周、小时、分钟、
秒、毫秒
)


A = BashOperator(
```

```
    task_id='A',
    bash_command='echo "run A task"',
    dag=dag
)

B = BashOperator(
    task_id='B',
    bash_command='echo "run B task"',
    dag=dag
)

C = BashOperator(
    task_id='C',
    bash_command='echo "run C task"',
    dag=dag,
    retries=3
)

D = BashOperator(
    task_id='D',
    bash_command='echo "run D task"',
    dag=dag
)

E = BashOperator(
    task_id='E',
    bash_command='echo "run E task"',
    dag=dag
)

A >>B>>E
C >>D>>E
```

1.6 Airflow Operators 及案例

Airflow 中最重要的还是各种 Operator，其允许生成特定类型的任务，这个任务在实例化时称为 DAG 中的任务节点，所有的 Operator 均派生自 BaseOparator，并且继承了许多属性和方法。关于 BaseOperator 的参数可以参照：

http://airflow.apache.org/docs/apache-airflow/stable/_api/airflow/models/baseoperator/index.html#module-airflow.models.baseoperator

BaseOperator 中常用参数如下：

task_id(str) ： 唯一 task_id 标记

owner(str):任务的所有者，建议使用 linux 用户名

email(str or list[str]):出问题时，发送报警 Email 的地址，可以填写多个，用逗号隔开。

email_on_retry(bool):当任务重试时是否发送电子邮件

email_on_failure(bool):当任务执行失败时是否发送电子邮件

retries(int):在任务失败之前应该重试的次数

retry_delay(datetime.timedelta):重试间隔，必须是 timedelta 对象

start_date(datetime.datetime):DAG 开始执行时间，这个参数必须是 datetime 对象，不可以使用字符串。

end_date(datetime.datetime): DAG 运行结束时间，任务启动后一般都会一直执行下去，一般不设置此参数。

depends_on_past(bool,默认 False):是否依赖于过去,如果为 True,那么必须之前的 DAG 调度成功了，现在的 DAG 调度才能执行。

dag(airflow.models.DAG):指定的 dag。

execution_timeout(datetime.timedelta): 执行此任务实例允许的最长时间，超过最长时间则任务失败。

trigger_rule(str):定义依赖的触发规则，包括选项如下：{ all_success | all_failed | all_done | one_success | one_failed | none_failed | none_failed_or_skipped | none_skipped | dummy(无条件执行)} default is all_success。

1.6.1 BashOperator 及调度 Shell 命令及脚本

BashOperator 主要执行 bash 脚本或命令，BashOperator 参数如下：

bash_command(str):要执行的命令或脚本(脚本必须是.sh 结尾)

- **BashOperator 调度 Shell 命令案例**

```
from datetime import datetime, timedelta


from airflow import DAG
from airflow.operators.bash import BashOperator


default_args = {
    'owner':'zhangsan',
    'start_date':datetime(2021, 9, 23),
    'email':'kettle_test1@163.com', #pwd:kettle123456
    'retries': 1,  # 失败重试次数
    'retry_delay': timedelta(minutes=5) # 失败重试间隔
```

```
}

dag = DAG(
    dag_id = 'execute_shell_cmd',
    default_args=default_args,
    schedule_interval=timedelta(minutes=1)
)

t1=BashOperator(
    task_id='print_date',
    bash_command='date',
    dag = dag
)

t2=BashOperator(
    task_id='print_helloworld',
    bash_command='echo "hello world!"',
    dag=dag
)

t3=BashOperator(
    task_id='tempplated',
    bash_command="""
    {% for i in range(5) %}
        echo "{{ ds }}"
        echo "{{ params.name}}"
        echo "{{ params.age}}"
    {% endfor %}
    """,
    params={'name':'wangwu','age':10},
    dag=dag
)

t1 >> t2 >> t3
```

注意在 t3 中使用了 Jinja 模板，“{% %}”内部是 for 标签，用于循环操作，但是必须以{% endfor %}结束。“{{}}”内部是变量，其中 ds 是执行日期，是 airflow 的宏变量，params.name 和 params.age 是自定义变量。

在 default_args 中的 email 是指当 DAG 执行失败时，发送邮件到指定邮箱，想要使用 airflow 发送邮件，需要在$AIRFLOW_HOME/airflow.cfg 中配置如下内容：

```
[smtp]
# If you want airflow to send emails on retries, failure, and you want to use
# the airflow.utils.email.send_email_smtp function, you have to configure an
# smtp server here
smtp_host = smtp.163.com
smtp_starttls = True
smtp_ssl = False
# Example: smtp_user = airflow
smtp_user =kettle_test2
# Example: smtp_password = airflow
smtp_password =VIOFSYMFDIKKIUEA
smtp_port = 25
smtp_mail_from =kettle_test2@163.com
smtp_timeout = 30
smtp_retry_limit = 5
```

此外，关于邮箱的信息如下：

邮箱 1：kettle_test1@163.com password:kettle123456

邮箱 2：kettle_test2@163.com password:kettle123456

163 邮箱 SMTP 服务器地址：smtp.163.com 端口：25

配置 163 邮箱时需要开启“POP3/SMTP/IMAP 服务”服务，设置如下：

kettle_test1@163.com FECJJVEPGPTZJYMQ

kettle_test2@163.com VIOFSYMFDIKKIUEA

- **BashOperator 调度 Shell 脚本案例**

准备如下两个 shell 脚本，将以下两个脚本放在$AIRFLOW_HOME/dags 目录下，

BashOperator 默认执行脚本时，默认从/tmp/airflow**临时目录查找对应脚本，由于临时目录名称不定，这里建议执行脚本时，在“bash_command”中写上绝对路径。如果要写相对路径，可以将脚本放在/tmp 目录下，在“bash_command”中执行命令写上“sh ../xxx.sh”也可以。

first_shell.sh

```
#!/bin/bash
dt=$1
echo "==== execute first shell ====="
echo "---- first : time is ${dt}"
```

second_shell.sh

```
#!/bin/bash
dt=$1
echo "==== execute second shell ====="
echo "---- second : time is ${dt}"
```

编写 airflow python 配置：

```
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.bash import BashOperator

default_args = {
    'owner':'zhangsan',
    'start_date':datetime(2021, 9, 23),
    'retries': 1,  # 失败重试次数
    'retry_delay': timedelta(minutes=5) # 失败重试间隔
}

dag = DAG(
    dag_id = 'execute_shell_sh',
    default_args=default_args,
    schedule_interval=timedelta(minutes=1)
)

first=BashOperator(
    task_id='first',
    #脚本路径建议写绝对路径
    bash_command='sh
/root/airflow/dags/first_shell.sh %s'%datetime.now().strftime("%Y-%m-%d"),
    dag = dag
)
```

```
second=BashOperator(
    task_id='second',
    #脚本路径建议写绝对路径
    bash_command='sh
/root/airflow/dags/second_shell.sh %s'%datetime.now().strftime("%Y-%m-%d"),
    dag=dag
)

first >> second
```

执行结果：

```
*** Reading local file: /root/airflow/logs/execute_shell_sh/first/2021-09-22T16:26:00+00:00/1.log
[2021-09-24 14:33:41,042] {taskinstance.py:903} INFO - Dependencies all met for <TaskInstance: execute_shell_sh.first 2021-09-22T16:26:00+00:00 [queued]>
[2021-09-24 14:33:41,057] {taskinstance.py:903} INFO - Dependencies all met for <TaskInstance: execute_shell_sh.first 2021-09-22T16:26:00+00:00 [queued]>
[2021-09-24 14:33:41,057] {taskinstance.py:1094} INFO -
--------------------------------------------------------------------------------
[2021-09-24 14:33:41,057] {taskinstance.py:1095} INFO - Starting attempt 1 of 2
[2021-09-24 14:33:41,057] {taskinstance.py:1096} INFO -
--------------------------------------------------------------------------------
[2021-09-24 14:33:41,066] {taskinstance.py:1114} INFO - Executing <Task(BashOperator): first> on 2021-09-22T16:26:00+00:00
[2021-09-24 14:33:41,068] {standard_task_runner.py:52} INFO - Started process 26900 to run task
[2021-09-24 14:33:41,072] {standard_task_runner.py:76} INFO - Running: ['airflow', 'tasks', 'run', 'execute_shell_sh', 'first', '2021-09-22T16:26:00+00:00', '--job-id', '352', '--pool', 'default_pool', '--raw', '--
[2021-09-24 14:33:41,072] {standard_task_runner.py:77} INFO - Job 352: Subtask first
[2021-09-24 14:33:41,121] {logging_mixin.py:109} INFO - Running <TaskInstance: execute_shell_sh.first 2021-09-22T16:26:00+00:00 [running]> on host node4
[2021-09-24 14:33:41,173] {taskinstance.py:1253} INFO - Exporting the following env vars:
AIRFLOW_CTX_DAG_OWNER=zhangsan
AIRFLOW_CTX_DAG_ID=execute_shell_sh
AIRFLOW_CTX_TASK_ID=first
AIRFLOW_CTX_EXECUTION_DATE=2021-09-22T16:26:00+00:00
AIRFLOW_CTX_DAG_RUN_ID=scheduled__2021-09-22T16:26:00+00:00
[2021-09-24 14:33:41,174] {subprocess.py:52} INFO - Tmp dir root location:
 /tmp
[2021-09-24 14:33:41,174] {subprocess.py:63} INFO - Running command: ['bash', '-c', 'sh /root/airflow/dags/first_shell.sh 2021-09-24']
[2021-09-24 14:33:41,181] {subprocess.py:74} INFO - Output:
[2021-09-24 14:33:41,183] {subprocess.py:78} INFO - ==== execute first shell ====
[2021-09-24 14:33:41,184] {subprocess.py:78} INFO - ----first : time is 2021-09-24
[2021-09-24 14:33:41,184] {subprocess.py:82} INFO - Command exited with return code 0
[2021-09-24 14:33:41,208] {taskinstance.py:1218} INFO - Marking task as SUCCESS. dag_id=execute_shell_sh, task_id=first, execution_date=20210922T162600, start_date=20210924T063341, end_date=20210924T063341
[2021-09-24 14:33:41,246] {local_task_job.py:151} INFO - Task exited with return code 0
[2021-09-24 14:33:41,275] {local_task_job.py:261} INFO - 1 downstream tasks scheduled from follow-on schedule check
```

```
*** Reading local file: /root/airflow/logs/execute_shell_sh/second/2021-09-22T16:26:00+00:00/1.log
[2021-09-24 14:33:46,119] {taskinstance.py:903} INFO - Dependencies all met for <TaskInstance: execute_shell_sh.second 2021-09-22T16:26:00+00:00 [queued]>
[2021-09-24 14:33:46,135] {taskinstance.py:903} INFO - Dependencies all met for <TaskInstance: execute_shell_sh.second 2021-09-22T16:26:00+00:00 [queued]>
[2021-09-24 14:33:46,135] {taskinstance.py:1094} INFO -
--------------------------------------------------------------------------------
[2021-09-24 14:33:46,135] {taskinstance.py:1095} INFO - Starting attempt 1 of 2
[2021-09-24 14:33:46,135] {taskinstance.py:1096} INFO -
--------------------------------------------------------------------------------
[2021-09-24 14:33:46,151] {taskinstance.py:1114} INFO - Executing <Task(BashOperator): second> on 2021-09-22T16:26:00+00:00
[2021-09-24 14:33:46,153] {standard_task_runner.py:52} INFO - Started process 26922 to run task
[2021-09-24 14:33:46,159] {standard_task_runner.py:76} INFO - Running: ['airflow', 'tasks', 'run', 'execute_shell_sh', 'second', '2021-09-22T16:26:00+00:00', '--job-id', '355', '--pool', 'default_pool', '--raw',
[2021-09-24 14:33:46,160] {standard_task_runner.py:77} INFO - Job 355: Subtask second
[2021-09-24 14:33:46,209] {logging_mixin.py:109} INFO - Running <TaskInstance: execute_shell_sh.second 2021-09-22T16:26:00+00:00 [running]> on host node4
[2021-09-24 14:33:46,249] {taskinstance.py:1253} INFO - Exporting the following env vars:
AIRFLOW_CTX_DAG_OWNER=zhangsan
AIRFLOW_CTX_DAG_ID=execute_shell_sh
AIRFLOW_CTX_TASK_ID=second
AIRFLOW_CTX_EXECUTION_DATE=2021-09-22T16:26:00+00:00
AIRFLOW_CTX_DAG_RUN_ID=scheduled__2021-09-22T16:26:00+00:00
[2021-09-24 14:33:46,250] {subprocess.py:52} INFO - Tmp dir root location:
 /tmp
[2021-09-24 14:33:46,251] {subprocess.py:63} INFO - Running command: ['bash', '-c', 'sh /root/airflow/dags/second_shell.sh 2021-09-24']
[2021-09-24 14:33:46,257] {subprocess.py:74} INFO - Output:
[2021-09-24 14:33:46,259] {subprocess.py:78} INFO - ==== execute second shell ====
[2021-09-24 14:33:46,259] {subprocess.py:78} INFO - ----second : time is 2021-09-24
[2021-09-24 14:33:46,259] {subprocess.py:82} INFO - Command exited with return code 0
[2021-09-24 14:33:46,276] {taskinstance.py:1218} INFO - Marking task as SUCCESS. dag_id=execute_shell_sh, task_id=second, execution_date=20210922T162600, start_date=20210924T063346, end_date=20210924T063346
[2021-09-24 14:33:46,293] {local_task_job.py:151} INFO - Task exited with return code 0
[2021-09-24 14:33:46,309] {local_task_job.py:261} INFO - 0 downstream tasks scheduled from follow-on schedule check
```

特别注意：在“bash_command”中写执行脚本时，一定要在脚本后跟上空格，有没有参数都要跟上空格，否则会找不到对应的脚本。如下：

```
first=BashOperator(
    task_id='first',
    #建议脚本路径写绝对路径
    bash_command='sh /root/airflow/dags/first_shell.sh %s'%datetime.now().strftime("%Y-%m-%d"),
    dag = dag
)
```

脚本有参数是，这里需要有空格

```
second=BashOperator(
    task_id='second',
    bash_command='sh /root/airflow/dags/second_shell.sh %s'%datetime.now().strftime("%Y-%m-%d"),
    dag=dag
)
```

```
first=BashOperator(
    task_id='first',
    #脚本路径建议写绝对路径
    bash_command='sh /root/airflow/dags/first_shell.sh ',
    dag = dag
)
```

脚本没有参数时，也需要跟上一个空格，否则报错找不到脚本

```
second=BashOperator(
    task_id='second',
    #脚本路径建议写绝对路径
    bash_command='sh /root/airflow/dags/second_shell.sh ',
    dag=dag
)
```

1.6.2 SSHOperator 及调度远程 Shell 脚本

在实际的调度任务中，任务脚本大多分布在不同的机器上，我们可以使用 SSHOperator 来调用远程机器上的脚本任务。SSHOperator 使用 ssh 协议与远程主机通信，需要注意的是 SSHOperator 调用脚本时并不会读取用户的配置文件，最好在脚本中加入以下代码以便脚本被调用时会自动读取当前用户的配置信息：

```
#Ubunto 系统
. ~/.profile

#CentoOS 或者 RedHat 系统
. ~/.bashrc
```

关于 SSHOperator 参数详解可以参照：

http://airflow.apache.org/docs/apache-airflow-providers-ssh/stable/_api/airflow/providers/ssh/operators/ssh/index.html#module-airflow.providers.ssh.operators.ssh

SSHOperator 的常用参数如下：

ssh_conn_id(str):ssh 连接 id，名称自取，需要在 airflow webserver 界面配置，具体配置参照案例。 **remote_host(str)**:远程连接节点 host，如果配置，可替换 ssh_conn_id 中配置的远程 host，可选。 **command(str):**在远程主机上执行的命令或脚本。

- **SSHOperator 调度远程节点脚本案例**

按照如下步骤来使用 SSHOperator 调度远程节点脚本：

1） 安装“apache-airflow-providers-ssh ” provider package

首先停止 airflow webserver 与 scheduler,在 node4 节点切换到 python37 环境，安装 ssh Connection 包。另外，关于 Providers package 安装方式可以参照如下官网地址：

https://airflow.apache.org/docs/apache-airflow-providers/packages-ref.html#apache-airflow-providers-ssh

```
#切换 Python37 环境
[root@node4 ~]# conda activate python37

#安装 ssh provider package
(python37) [root@node4 ~]# pip install apache-airflow-providers-ssh==2.1.1

#启动 airflow
(python37) [root@node4 ~]# airflow webserver --port 8080
(python37) [root@node4 ~]# airflow scheduler
```

2） 配置 SSH Connection 连接

登录 airflow webui ，选择“Admin” -> “Connections”:

点击“+”添加连接，这里 host 连接的是 node5 节点：

3） 准备远程执行脚本

在 node5 节点/root 路径下创建 first_shell.sh，内容如下：

```
#!/bin/bash
echo "==== execute first shell ====="
```

在 node3 节点/root 路径下创建 second_shell.sh，内容如下：

```
#!/bin/bash
echo "==== execute second shell ====="
```

4） 编写 DAG python 配置文件

注意在本地开发工具编写 python 配置时，需要用到 SSHOperator，需要在本地对应的 python 环境中安装对应的 provider package。

```
C:\Users\wubai>d:
D:\>cd d:\ProgramData\Anaconda3\envs\python37\Scripts
d:\ProgramData\Anaconda3\envs\python37\Scripts>pip install
apache-airflow-providers-ssh==2.1.1
```

python 配置文件：

```
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.bash import BashOperator
from airflow.providers.ssh.operators.ssh import SSHOperator

default_args = {
    'owner':'lisi',
    'start_date':datetime(2021, 9, 23),
    'retries': 1,  # 失败重试次数
    'retry_delay': timedelta(minutes=5) # 失败重试间隔
}

dag = DAG(
    dag_id = 'execute_remote_shell',
    default_args=default_args,
    schedule_interval=timedelta(minutes=1)
)

first=SSHOperator(
    task_id='first',
    ssh_conn_id='ssh-node5',# 配置在 Airflow webui Connection 中配置的 SSH Conn id
    command='sh /root/first_shell.sh ',
    dag = dag
)

second=SSHOperator(
    task_id='second',
    ssh_conn_id='ssh-node5',# 配置在 Airflow webui Connection 中配置的 SSH Conn id
    command='sh /root/second_shell.sh ',
    remote_host="192.168.179.6",#如果配置 remote_host ，将会替换 Connection 中的 SSH 配置的
host
    dag=dag
)

first >> second
```

5） 调度 python 配置脚本

将以上配置好的 python 文件上传至 node4 节点$AIRFLOW_HOME/dags 下，重启

Airflow websever 与 scheduler，登录 webui，开启调度：

调度结果如下：

```
*** Reading local file: /root/airflow/logs/execute_remote_shell/first/2021-09-22T16:11:00+00:00/1.log
[2021-09-24 17:33:48,049] {taskinstance.py:903} INFO - Dependencies all met for <TaskInstance: execute_remote_shell.first 2021-09-22T16:11:00+00:00 [queued]>
[2021-09-24 17:33:48,060] {taskinstance.py:903} INFO - Dependencies all met for <TaskInstance: execute_remote_shell.first 2021-09-22T16:11:00+00:00 [queued]>
[2021-09-24 17:33:48,060] {taskinstance.py:1094} INFO -
--------------------------------------------------------------------------------
[2021-09-24 17:33:48,060] {taskinstance.py:1095} INFO - Starting attempt 1 of 2
[2021-09-24 17:33:48,060] {taskinstance.py:1096} INFO -
--------------------------------------------------------------------------------
[2021-09-24 17:33:48,067] {taskinstance.py:1114} INFO - Executing <Task(SSHOperator): first> on 2021-09-22T16:11:00+00:00
[2021-09-24 17:33:48,068] {standard_task_runner.py:52} INFO - Started process 47993 to run task
[2021-09-24 17:33:48,073] {standard_task_runner.py:76} INFO - Running: ['airflow', 'tasks', 'run', 'execute_remote_shell', 'first', '2021-09-22T16:11:00+00:00', '--job-id', '473', '--pool', 'default_pool', '--raw'
[2021-09-24 17:33:48,073] {standard_task_runner.py:77} INFO - Job 473: Subtask first
[2021-09-24 17:33:48,114] {logging_mixin.py:109} INFO - Running <TaskInstance: execute_remote_shell.first 2021-09-22T16:11:00+00:00 [running]> on host node4
[2021-09-24 17:33:48,194] {taskinstance.py:1253} INFO - Exporting the following env vars:
AIRFLOW_CTX_DAG_OWNER=lisi
AIRFLOW_CTX_DAG_ID=execute_remote_shell
AIRFLOW_CTX_TASK_ID=first
AIRFLOW_CTX_EXECUTION_DATE=2021-09-22T16:11:00+00:00
AIRFLOW_CTX_DAG_RUN_ID=scheduled__2021-09-22T16:11:00+00:00
[2021-09-24 17:33:48,194] {ssh.py:90} INFO - ssh_hook is not provided or invalid. Trying ssh_conn_id to create SSHHook.
[2021-09-24 17:33:48,201] {base.py:79} INFO - Using connection to: id: ssh-node5. Host: 192.168.179.8, Port: 22, Schema: , Login: root, Password: ***, extra: {}
[2021-09-24 17:33:48,201] {ssh.py:232} WARNING - Remote Identification Change is not verified. This wont protect against Man-In-The-Middle attacks
[2021-09-24 17:33:48,204] {ssh.py:238} WARNING - No Host Key Verification. This wont protect against Man-In-The-Middle attacks
[2021-09-24 17:33:48,214] {transport.py:1819} INFO - Connected (version 2.0, client OpenSSH_7.4)
[2021-09-24 17:33:48,271] {transport.py:1819} INFO - Authentication (publickey) successful!
[2021-09-24 17:33:48,272] {ssh.py:109} INFO - Running command: sh /root/first_shell.sh
[2021-09-24 17:33:48,757] {ssh.py:141} INFO - ==== execute first shell ====
[2021-09-24 17:33:48,778] {taskinstance.py:1218} INFO - Marking task as SUCCESS. dag_id=execute_remote_shell, task_id=first, execution_date=20210922T161100, start_date=20210924T093348, end_date=20210924T093348
[2021-09-24 17:33:48,815] {local_task_job.py:151} INFO - Task exited with return code 0
[2021-09-24 17:33:48,888] {local_task_job.py:261} INFO - 1 downstream tasks scheduled from follow-on schedule check
```

```
*** Reading local file: /root/airflow/logs/execute_remote_shell/second/2021-09-22T16:11:00+00:00/1.log
[2021-09-24 17:33:55,166] {taskinstance.py:903} INFO - Dependencies all met for <TaskInstance: execute_remote_shell.second 2021-09-22T16:11:00+00:00 [queued]>
[2021-09-24 17:33:55,177] {taskinstance.py:903} INFO - Dependencies all met for <TaskInstance: execute_remote_shell.second 2021-09-22T16:11:00+00:00 [queued]>
[2021-09-24 17:33:55,178] {taskinstance.py:1094} INFO -
--------------------------------------------------------------------------------
[2021-09-24 17:33:55,178] {taskinstance.py:1095} INFO - Starting attempt 1 of 2
[2021-09-24 17:33:55,178] {taskinstance.py:1096} INFO -
--------------------------------------------------------------------------------
[2021-09-24 17:33:55,189] {taskinstance.py:1114} INFO - Executing <Task(SSHOperator): second> on 2021-09-22T16:11:00+00:00
[2021-09-24 17:33:55,191] {standard_task_runner.py:52} INFO - Started process 48020 to run task
[2021-09-24 17:33:55,196] {standard_task_runner.py:76} INFO - Running: ['airflow', 'tasks', 'run', 'execute_remote_shell', 'second', '2021-09-22T16:11:00+00:00', '--job-id', '476', '--pool', 'default_p
[2021-09-24 17:33:55,197] {standard_task_runner.py:77} INFO - Job 476: Subtask second
[2021-09-24 17:33:55,247] {logging_mixin.py:109} INFO - Running <TaskInstance: execute_remote_shell.second 2021-09-22T16:11:00+00:00 [running]> on host node4
[2021-09-24 17:33:55,295] {taskinstance.py:1253} INFO - Exporting the following env vars:
AIRFLOW_CTX_DAG_OWNER=lisi
AIRFLOW_CTX_DAG_ID=execute_remote_shell
AIRFLOW_CTX_TASK_ID=second
AIRFLOW_CTX_EXECUTION_DATE=2021-09-22T16:11:00+00:00
AIRFLOW_CTX_DAG_RUN_ID=scheduled__2021-09-22T16:11:00+00:00
[2021-09-24 17:33:55,296] {ssh.py:90} INFO - ssh_hook is not provided or invalid. Trying ssh_conn_id to create SSHHook.
[2021-09-24 17:33:55,303] {base.py:79} INFO - Using connection to: id: ssh-node5. Host: 192.168.179.8, Port: 22, Schema: , Login: root, Password: ***, extra: {}
[2021-09-24 17:33:55,303] {ssh.py:99} INFO - remote_host is provided explicitly. It will replace the remote_host which was defined in ssh_hook or predefined in connection of ssh_conn_id.
[2021-09-24 17:33:55,304] {ssh.py:232} WARNING - Remote Identification Change is not verified. This wont protect against Man-In-The-Middle attacks
[2021-09-24 17:33:55,306] {ssh.py:238} WARNING - No Host Key Verification. This wont protect against Man-In-The-Middle attacks
[2021-09-24 17:33:55,315] {transport.py:1819} INFO - Connected (version 2.0, client OpenSSH_7.4)
[2021-09-24 17:33:55,412] {transport.py:1819} INFO - Authentication (publickey) successful!
[2021-09-24 17:33:55,412] {ssh.py:109} INFO - Running command: sh /root/second_shell.sh
[2021-09-24 17:33:55,505] {ssh.py:141} INFO - ==== execute second shell ====
[2021-09-24 17:33:55,532] {taskinstance.py:1218} INFO - Marking task as SUCCESS. dag_id=execute_remote_shell, task_id=second, execution_date=20210922T161100, start_date=20210924T093355, end_date=202109
[2021-09-24 17:33:55,572] {local_task_job.py:151} INFO - Task exited with return code 0
[2021-09-24 17:33:55,622] {local_task_job.py:261} INFO - 0 downstream tasks scheduled from follow-on schedule check
[2021-09-24 17:33:55,674] {dagrun.py:434} INFO - Marking run <DagRun execute_remote_shell @ 2021-09-22 16:11:00+00:00: scheduled__2021-09-22T16:11:00+00:00, externally triggered: False> successful
```

替换host，连接配置的节点

1.6.3 HiveOperator 及调度 HQL

可以通过 HiveOperator 直接操作 Hive SQL ，HiveOperator 的参数如下：

hql(str)：需要执行的 Hive SQL。

hive_cli_conn_id(str)：连接 Hive 的 conn_id，在 airflow webui connection 中配置的。

想要在 airflow 中使用 HiveOperator 调用 Hive 任务，首先需要安装以下依赖并配置 Hive Metastore：

```
#切换 Python37 环境
[root@node4 ~]# conda activate python37

#安装 hive provider package
(python37) [root@node4 ~]# pip install apache-airflow-providers-apache-hive==2.0.2

#启动 airflow
(python37) [root@node4 ~]# airflow webserver --port 8080
(python37) [root@node4 ~]# airflow scheduler
```

登录 Airflow webui 并设置 Hive Metastore，登录后找到"Admin"->"Connections"，点击"+"新增配置：

- **HiveOperator 调度 HQL 案例**

1） 启动 Hive，准备表

启动 HDFS、Hive Metastore，在 Hive 中创建以下三张表：

```
create table person_info(id int,name string,age int) row format
```

```
delimited fields terminated by '\t';

create table score_info(id int,name string,score int) row format
delimited fields terminated by '\t';
```

向表 person_info 加载如下数据：

```
1   zs  18
2   ls  19
3   ww  20
```

向表 score_info 加载如下数据：

```
1   zs  100
2   ls  200
3   ww  300
```

2） 在 node4 节点配置 Hive 客户端

由于 Airflow 使用 HiveOperator 时需要在 Airflow 安装节点上有 Hive 客户端，所以需要在 node4 节点上配置 Hive 客户端。

将 Hive 安装包上传至 node4 “/software”下解压，并配置 Hive 环境变量

```
#在/etc/profile 文件最后配置 Hive 环境变量
export HIVE_HOME=/software/hive-1.2.1
export PATH=$PATH:$HIVE_HOME/bin

#使环境变量生效
source /etc/profile
```

修改 HIVE_HOME/conf/hive-site.xml ，写入如下内容：

```
<configuration>
 <property>
  <name>hive.metastore.warehouse.dir</name>
  <value>/user/hive/warehouse</value>
 </property>
 <property>
  <name>hive.metastore.local</name>
  <value>false</value>
 </property>
 <property>
  <name>hive.metastore.uris</name>
```

```
  <value>thrift://node1:9083</value>
 </property>
</configuration>
```

3） 编写 DAG python 配置文件

注意在本地开发工具编写 python 配置时，需要用到 HiveOperator，需要在本地对应的 python 环境中安装对应的 provider package。

```
C:\Users\wubai>d:
D:\>cd d:\ProgramData\Anaconda3\envs\python37\Scripts
d:\ProgramData\Anaconda3\envs\python37\Scripts>pip install apache-airflow-providers-apache-hive==2.0.2
注意：这里本地安装也有可能缺少对应的 C++环境，我们也可以不安装，直接跳过也可以。
```

Python 配置文件：

```
from datetime import datetime, timedelta
from airflow import DAG
from airflow.providers.apache.hive.operators.hive import HiveOperator

default_args = {
    'owner':'wangwu',
    'start_date':datetime(2021, 9, 23),
    'retries': 1,  # 失败重试次数
    'retry_delay': timedelta(minutes=5) # 失败重试间隔
}

dag = DAG(
    dag_id = 'execute_hive_sql',
    default_args=default_args,
    schedule_interval=timedelta(minutes=1)
)

first=HiveOperator(
    task_id='person_info',
    hive_cli_conn_id="node1-hive-metastore",
    hql='select id,name,age from person_info',
    dag = dag
)

second=HiveOperator(
    task_id='score_info',
```

```
    hive_cli_conn_id="node1-hive-metastore",
    hql='select id,name,score from score_info',
    dag=dag
)

third=HiveOperator(
    task_id='join_info',
    hive_cli_conn_id="node1-hive-metastore",
    hql='select a.id,a.name,a.age,b.score from person_info a join score_info b on a.id = b.id',
    dag=dag
)

first >> second >>third
```

4） 调度 python 配置脚本

将以上配置好的 python 文件上传至 node4 节点$AIRFLOW_HOME/dags 下，重启 Airflow websever 与 scheduler，登录 webui，开启调度：

调度结果如下：

```
*** Reading local file: /root/airflow/logs/execute_hive_sql/person_info/2021-09-22T16:04:00+00:00/1.log
[2021-09-25 15:41:56,964] {taskinstance.py:903} INFO - Dependencies all met for <TaskInstance: execute_hive_sql.person_info 2021-09-22T16:04:00+00:00 [queued]>
[2021-09-25 15:41:56,977] {taskinstance.py:903} INFO - Dependencies all met for <TaskInstance: execute_hive_sql.person_info 2021-09-22T16:04:00+00:00 [queued]>
[2021-09-25 15:41:56,977] {taskinstance.py:1094} INFO -
--------------------------------------------------------------------------------
[2021-09-25 15:41:56,977] {taskinstance.py:1095} INFO - Starting attempt 1 of 2
[2021-09-25 15:41:56,977] {taskinstance.py:1096} INFO -
--------------------------------------------------------------------------------
[2021-09-25 15:41:56,986] {taskinstance.py:1114} INFO - Executing <Task(HiveOperator): person_info> on 2021-09-22T16:04:00+00:00
[2021-09-25 15:41:56,990] {standard_task_runner.py:52} INFO - Started process 33898 to run task
[2021-09-25 15:41:56,998] {standard_task_runner.py:76} INFO - Running: ['airflow', 'tasks', 'run', 'execute_hive_sql', 'person_info', '2021-09-22T16:04:00+00:00', '--job-id', '530', '--pool', 'default_pool', '--raw', '--subdir', 'DA
[2021-09-25 15:41:56,998] {standard_task_runner.py:77} INFO - Job 530: Subtask person_info
[2021-09-25 15:41:57,008] {logging_mixin.py:109} INFO - Running <TaskInstance: execute_hive_sql.person_info 2021-09-22T16:04:00+00:00 [running]> on host node4
[2021-09-25 15:41:57,108] {taskinstance.py:1253} INFO - Exporting the following env vars:
AIRFLOW_CTX_DAG_OWNER=wangwu
AIRFLOW_CTX_DAG_ID=execute_hive_sql
AIRFLOW_CTX_TASK_ID=person_info
AIRFLOW_CTX_EXECUTION_DATE=2021-09-22T16:04:00+00:00
AIRFLOW_CTX_DAG_RUN_ID=scheduled__2021-09-22T16:04:00+00:00
[2021-09-25 15:41:57,108] {hive.py:137} INFO - Executing: select id,name,age from person_info
[2021-09-25 15:41:57,117] {base.py:79} INFO - Using connection to: id: node1-hive-metastore. Host: 192.168.179.4, Port: 9083, Schema: default, Login: , Password: None, extra: {}
[2021-09-25 15:41:57,117] {hive.py:155} INFO - Passing HiveConf: {'airflow.ctx.dag_owner': 'wangwu', 'airflow.ctx.dag_id': 'execute_hive_sql', 'airflow.ctx.task_id': 'person_info', 'airflow.ctx.execution_date': '2021-09-22T16:04:00+
[2021-09-25 15:41:57,118] {hive.py:247} INFO - hive -hiveconf airflow.ctx.dag_id=execute_hive_sql -hiveconf airflow.ctx.task_id=person_info -hiveconf airflow.ctx.execution_date=2021-09-22T16:04:00+00:00 -hiveconf airflow.ctx.dag_run
[2021-09-25 15:42:05,682] {hive.py:259} INFO - 21/09/25 15:42:05 WARN conf.HiveConf: HiveConf of name hive.metastore.local does not exist
[2021-09-25 15:42:06,017] {hive.py:259} INFO -
[2021-09-25 15:42:06,017] {hive.py:259} INFO - Logging initialized using configuration in jar:file:/software/hive-1.2.1/lib/hive-common-1.2.1.jar!/hive-log4j.properties
[2021-09-25 15:42:09,658] {hive.py:259} INFO - OK
[2021-09-25 15:42:09,670] {hive.py:259} INFO - Time taken: 1.414 seconds
[2021-09-25 15:42:11,228] {hive.py:259} INFO - OK
[2021-09-25 15:42:11,572] {hive.py:259} INFO - 1        zs      18
[2021-09-25 15:42:11,572] {hive.py:259} INFO - 2        ls      19
[2021-09-25 15:42:11,572] {hive.py:259} INFO - 3        ww      20
[2021-09-25 15:42:11,597] {hive.py:259} INFO - Time taken: 1.562 seconds, Fetched: 3 row(s)
[2021-09-25 15:42:12,152] {taskinstance.py:1218} INFO - Marking task as SUCCESS. dag_id=execute_hive_sql, task_id=person_info, execution_date=20210922T160400, start_date=20210925T074156, end_date=20210925T074212
[2021-09-25 15:42:12,198] {local_task_job.py:151} INFO - Task exited with return code 0
[2021-09-25 15:42:12,228] {local_task_job.py:261} INFO - 1 downstream tasks scheduled from follow-on schedule check
```

```
*** Reading local file: /root/airflow/logs/execute_hive_sql/score_info/2021-09-22T16:04:00+00:00/1.log
[2021-09-25 15:42:46,719] {taskinstance.py:903} INFO - Dependencies all met for <TaskInstance: execute_hive_sql.score_info 2021-09-22T16:04:00+00:00 [queued]>
[2021-09-25 15:42:46,732] {taskinstance.py:903} INFO - Dependencies all met for <TaskInstance: execute_hive_sql.score_info 2021-09-22T16:04:00+00:00 [queued]>
[2021-09-25 15:42:46,732] {taskinstance.py:1094} INFO -
--------------------------------------------------------------------------------
[2021-09-25 15:42:46,732] {taskinstance.py:1095} INFO - Starting attempt 1 of 2
[2021-09-25 15:42:46,732] {taskinstance.py:1096} INFO -
--------------------------------------------------------------------------------
[2021-09-25 15:42:46,744] {taskinstance.py:1114} INFO - Executing <Task(HiveOperator): score_info> on 2021-09-22T16:04:00+00:00
[2021-09-25 15:42:46,747] {standard_task_runner.py:52} INFO - Started process 34481 to run task
[2021-09-25 15:42:46,752] {standard_task_runner.py:76} INFO - Running: ['airflow', 'tasks', 'run', 'execute_hive_sql', 'score_info', '2021-09-22T16:04:00+00:00', '--job-id', '533', '--pool', 'default_pool', '--ra
[2021-09-25 15:42:46,754] {standard_task_runner.py:77} INFO - Job 533: Subtask score_info
[2021-09-25 15:42:46,808] {logging_mixin.py:109} INFO - Running <TaskInstance: execute_hive_sql.score_info 2021-09-22T16:04:00+00:00 [running]> on host node4
[2021-09-25 15:42:46,863] {taskinstance.py:1253} INFO - Exporting the following env vars:
AIRFLOW_CTX_DAG_OWNER=wangwu
AIRFLOW_CTX_DAG_ID=execute_hive_sql
AIRFLOW_CTX_TASK_ID=score_info
AIRFLOW_CTX_EXECUTION_DATE=2021-09-22T16:04:00+00:00
AIRFLOW_CTX_DAG_RUN_ID=scheduled__2021-09-22T16:04:00+00:00
[2021-09-25 15:42:46,864] {hive.py:137} INFO - Executing: select id,name,score from score_info
[2021-09-25 15:42:46,874] {base.py:79} INFO - Using connection to: id: node1-hive-metastore. Host: 192.168.179.4, Port: 9083, Schema: default, Login: , Password: None, extra: {}
[2021-09-25 15:42:46,874] {hive.py:155} INFO - Passing HiveConf: {'airflow.ctx.dag_owner': 'wangwu', 'airflow.ctx.dag_id': 'execute_hive_sql', 'airflow.ctx.task_id': 'score_info', 'airflow.ctx.execution_date': '2
[2021-09-25 15:42:46,875] {hive.py:247} INFO - hive -hiveconf airflow.ctx.dag_id=execute_hive_sql -hiveconf airflow.ctx.task_id=score_info -hiveconf airflow.ctx.execution_date=2021-09-22T16:04:00+00:00 -hiveconf a
[2021-09-25 15:42:50,749] {hive.py:259} INFO - 21/09/25 15:42:50 WARN conf.HiveConf: HiveConf of name hive.metastore.local does not exist
[2021-09-25 15:42:50,986] {hive.py:259} INFO -
[2021-09-25 15:42:50,986] {hive.py:259} INFO - Logging initialized using configuration in jar:file:/software/hive-1.2.1/lib/hive-common-1.2.1.jar!/hive-log4j.properties
[2021-09-25 15:42:53,713] {hive.py:259} INFO - OK
[2021-09-25 15:42:53,713] {hive.py:259} INFO - Time taken: 1.026 seconds
[2021-09-25 15:42:54,675] {hive.py:259} INFO - OK
[2021-09-25 15:42:54,832] {hive.py:259} INFO - 1        zs      100
[2021-09-25 15:42:54,832] {hive.py:259} INFO - 2        ls      200
[2021-09-25 15:42:54,832] {hive.py:259} INFO - 3        ww      300
[2021-09-25 15:42:54,854] {hive.py:259} INFO - Time taken: 0.962 seconds, Fetched: 3 row(s)
[2021-09-25 15:42:55,374] {taskinstance.py:1218} INFO - Marking task as SUCCESS. dag_id=execute_hive_sql, task_id=score_info, execution_date=20210922T160400, start_date=20210925T074246, end_date=20210925T074255
[2021-09-25 15:42:55,419] {local_task_job.py:151} INFO - Task exited with return code 0
[2021-09-25 15:42:55,482] {local_task_job.py:261} INFO - 0 downstream tasks scheduled from follow-on schedule check
```

```
[2021-09-25 15:55:42,847] {hive.py:259} INFO - Starting Job = job_1632504247458_0004, Tracking URL = http://node1:8088/proxy/application_1632504247458_0004/
[2021-09-25 15:55:42,847] {hive.py:259} INFO - Kill Command = /software/hadoop-2.9.2/bin/hadoop job  -kill job_1632504247458_0004
[2021-09-25 15:55:56,230] {hive.py:259} INFO - Hadoop job information for Stage-3: number of mappers: 1; number of reducers: 0
[2021-09-25 15:55:56,285] {hive.py:259} INFO - 2021-09-25 15:55:56,279 Stage-3 map = 0%,  reduce = 0%
[2021-09-25 15:56:11,113] {hive.py:259} INFO - 2021-09-25 15:56:11,112 Stage-3 map = 100%,  reduce = 0%, Cumulative CPU 3.09 sec
[2021-09-25 15:56:12,159] {hive.py:259} INFO - MapReduce Total cumulative CPU time: 3 seconds 90 msec
[2021-09-25 15:56:12,198] {hive.py:259} INFO - Ended Job = job_1632504247458_0004
[2021-09-25 15:56:12,253] {hive.py:259} INFO - MapReduce Jobs Launched:
[2021-09-25 15:56:12,254] {hive.py:259} INFO - Stage-Stage-3: Map: 1   Cumulative CPU: 3.09 sec   HDFS Read: 6191 HDFS Write: 36 SUCCESS
[2021-09-25 15:56:12,254] {hive.py:259} INFO - Total MapReduce CPU Time Spent: 3 seconds 90 msec
[2021-09-25 15:56:12,254] {hive.py:259} INFO - OK
[2021-09-25 15:56:12,321] {hive.py:259} INFO - 1        zs      18      100
[2021-09-25 15:56:12,321] {hive.py:259} INFO - 2        ls      19      200
[2021-09-25 15:56:12,321] {hive.py:259} INFO - 3        ww      20      300
[2021-09-25 15:56:12,344] {hive.py:259} INFO - Time taken: 43.2 seconds, Fetched: 3 row(s)
[2021-09-25 15:56:12,756] {taskinstance.py:1218} INFO - Marking task as SUCCESS. dag_id=execute_hive_sql, task_id=join_info, execution_date=20210922T160200, start_date=20210925T075520, end_date=2
[2021-09-25 15:56:12,838] {local_task_job.py:151} INFO - Task exited with return code 0
[2021-09-25 15:56:12,868] {local_task_job.py:261} INFO - 0 downstream tasks scheduled from follow-on schedule check
```

1.6.4 PythonOperator

PythonOperator 可以调用 Python 函数，由于 Python 基本可以调用任何类型的任务，如果实在找不到合适的 Operator，将任务转为 Python 函数，使用 PythonOperator 即可。

关于 PythonOperator 常用参数如下，更多参数可以查看官网：http://airflow.apache.org/docs/apache-airflow/stable/_api/airflow/operators/python/index.html#module-airflow.operators.python

python_callable(python_callable)：调用的 python 函数

op_kwargs(dict)：调用 python 函数对应的 **args 参数，dict 格式，使用参照案例。

op_args(list)：调用 python 函数对应的 *args 参数，多个封装到一个 tuple 中，list 格式，使用参照案例。

- **PythonOperator 调度案例**

```
import random

from datetime import datetime, timedelta
```

```
from airflow import DAG

from airflow.operators.python import PythonOperator


# python 中 *  关键字参数允许你传入 0 个或任意个参数，这些可变参数在函数调用时自动组装为一个
tuple。
# python 中 ** 关键字参数允许你传入 0 个或任意个含参数名的参数,这些关键字参数在函数内部自动组
装为一个 dict。
def print_hello1(*a,**b):
    print(a)
    print(b)
    print("hello airflow1")
    # 返回的值只会打印到日志中
    return{"sss1":"xxx1"}


def print_hello2(random_base):
    print(random_base)
    print("hello airflow2")
    # 返回的值只会打印到日志中
    return{"sss2":"xxx2"}


default_args = {
    'owner':'maliu',
    'start_date':datetime(2021, 10, 1),
    'retries': 1,  # 失败重试次数
    'retry_delay': timedelta(minutes=5) # 失败重试间隔
}


dag = DAG(
```

```
    dag_id = 'execute_pythoncode',
    default_args=default_args,
    schedule_interval=timedelta(minutes=1)
)

first=PythonOperator(
    task_id='first',
    #填写  print__hello1 方法时，不要加上“()”
    python_callable=print__hello1,
    # op_args 对应 print_hello1 方法中的 a 参数
    op_args=[1,2,3,"hello","world"],
    # op_kwargs 对应 print__hello1 方法中的 b 参数
    op_kwargs={"id":"1","name":"zs","age":18},
    dag = dag
)

second=PythonOperator(
    task_id='second',
    #填写  print__hello2 方法时，不要加上“()”
    python_callable=print__hello2,
    # random_base 参数对应 print_hello2 方法中参数“random_base”
    op_kwargs={"random_base":random.randint(0,9)},
    dag=dag
)

first >> second
```

1.7 Airflow 分布式集群搭建及使用

1.7.1 Airflow 分布式集群搭建原因

在稳定性要求较高的场景中，例如：金融交易系统，airflow 一般采用集群、高可用方式搭建部署，airflow 对应的进程分布在多个节点上运行，形成 Airflow 集群、高可用部署，架构图如下：

以上集群、高可用方式搭建 Airflow 好处如下：

- 如果一个 worker 节点崩溃挂掉，集群仍然可以正常利用其他 worker 节点来调度执行任务。
- 当工作流中有内存密集型任务，任务最好分布在多态机器上执行以得到更好效果，airflow 分布式集群满足这点。

1.7.2 Airflow 分布式集群其他扩展

1.7.2.1 扩展 Worker 节点

我们可以通过向集群中添加更多的 worker 节点来水平扩展集群，并使这些新节点使用同一个元数据库，从而分布式处理任务。由于 Worker 不需要再任何进程注册即可执行任务，因此 worker 节点可以在不停机，不重启服务下的情况进行扩展。

我们也可以通过增加单个 worker 节点的进程数来垂直扩展集群，可以通过修改 airflow 配置文件 AIRFLOW_HOME/airflow.cfg 中 celeryd_concurrency 的值来

实现，例如：celeryd_concurrency=30，我们可以根据集群上运行任务性质、CPU 的内核数量等增加 Worker 单节点并发数量来满足实际需求。

扩展 worker 节点后的架构如下：

1.7.2.2 扩展 Master 节点

我们还可以向集群中添加更多的主节点，以扩展主节点上运行的服务。我们可以扩展 webserver，防止太多的 HTTP 请求出现在一台机器上防止 webserver 挂掉，需要注意，Master 节点包含 Scheduler 与 webServer，在一个 Airflow 集群中我们只能一次运行一个 Scheduler 进程，如果有多个 Scheduler 运行，那么可能出现同一个任务被执行多次，导致任务流重复执行。

Master 扩展参照后续 Airflow 分布式集群搭建，扩展 Master 后的架构如下：

1.7.2.3 Scheduler HA

扩展 Master 后的 Airflow 集群中只能运行一个 Scheduler，那么运行的 Scheudler 进程挂掉，任务同样不能正常调度运行，这种情况我们可以在两台机器上部署 scheduler，只运行一台机器上的 Scheduler 进程，一旦运行 Schduler 进程的机器出现故障，立刻启动另一台机器上的 Scheduler 即可，这种就是 Schduler HA，我们可以借助第三方组件 airflow-scheduler-failover-controller 实现 Scheduler 的高可用。

详细操作参照后续 Airflow 分布式集群搭建，加入 Scheduler HA 的架构如下：

1.7.3 Airflow 分布式集群搭建及测试

1.7.3.1 节点规划

节点 IP	节点名称	节点角色	运行服务
192.168.179.4	node1	Master1	webserver，scheduler
192.168.179.5	node2	Master2	websever，scheduler
192.168.179.6	node3	Worker1	worker
192.168.179.7	node4	Worker2	worker

1.7.3.2 airflow 集群搭建步骤

1） 在所有节点安装 python3.7

参照单节点安装 Airflow 中安装 anconda 及 python3.7。

2） 在所有节点上安装 airflow

- **每台节点安装 airflow 需要的系统依赖**

```
yum -y install mysql-devel gcc gcc-devel python-devel gcc-c++ cyrus-sasl
cyrus-sasl-devel cyrus-sasl-lib
```

- **每台节点配置 airflow 环境变量**

```
vim /etc/profile
export AIRFLOW_HOME=/root/airflow

#使配置的环境变量生效
source /etc/profile
```

- **每台节点切换 airflow 环境，安装 airflow，指定版本为 2.1.3**

```
(python37) conda activate python37
(python37) pip install apache-airflow==2.1.3 -i
https://pypi.tuna.tsinghua.edu.cn/simple
```

默认 Airflow 安装在 $ANCONDA_HOME/envs/python37/lib/python3.7/site-packages/airflow 目录下。配置了 AIRFLOW_HOME ,Airflow 安装后文件存储目录在 AIRFLOW_HOME 目录下。可以每台节点查看安装 Airflow 版本信息：

```
(python37) airflow version
2.1.3
```

- **在 Mysql 中创建对应的库并设置参数**

aiflow 使用的 Metadata database 我们这里使用 mysql,在 node2 节点的 mysql 中创建 airflow 使用的库及表信息。

```
CREATE DATABASE airflow CHARACTER SET utf8;
create user 'airflow'@'%' identified by '123456';
grant all privileges on airflow.* to 'airflow'@'%';
flush privileges;
```

在 mysql 安装节点 node2 上修改"/etc/my.cnf"，在[mysqld]下添加如下内容：

```
[mysqld]
explicit_defaults_for_timestamp=1
```

以上修改完成“my.cnf”值后，重启 Mysql 即可，重启之后，可以查询对应的参数是否生效：

```
#重启 mysql
[root@node2 ~]# service mysqld restart

#重新登录 mysql 查询
mysql> show variables like 'explicit_defaults_for_timestamp';
```

```
mysql> show variables like 'explicit_defaults_for_timestamp';
+-----------------------------------+-------+
| Variable_name                     | Value |
+-----------------------------------+-------+
| explicit_defaults_for_timestamp | ON    |
+-----------------------------------+-------+
1 row in set (0.02 sec)
```

- **每台节点配置 Airflow airflow.cfg 文件**

修改 AIRFLOW_HOME/airflow.cfg 文件，确保所有机器使用同一份配置文件，在 node1 节点上配置 airflow.cfg，配置如下：

```
[core]
dags_folder = /root/airflow/dags


#修改时区
default_timezone = Asia/Shanghai


#配置 Executor 类型，集群建议配置 CeleryExecutor

executor = CeleryExecutor


# 配置数据库

sql_alchemy_conn=mysql+mysqldb://airflow:123456@node2:3306/airflow?use_unico
de=true&charset=utf8


[webserver]
#设置时区
default_ui_timezone = Asia/Shanghai
```

```
[celery]

#配置 Celery broker 使用的消息队列

broker_url = redis://node4:6379/0

#配置 Celery broker 任务完成后状态更新使用库

result_backend = db+mysql://root:123456@node2:3306/airflow
```

将 node1 节点配置好的 airflow.cfg 发送到 node2、node3、node4 节点上：

```
(python37) [root@node1 airflow]# scp ./airflow.cfg  node2:`pwd`

(python37) [root@node1 airflow]# scp ./airflow.cfg  node3:`pwd`

(python37) [root@node1 airflow]# scp ./airflow.cfg  node4:`pwd`
```

1.7.3.3 初始化 Airflow

1. 每台节点安装需要的 python 依赖包

初始化 Airflow 数据库时需要使用到连接 mysql 的包，执行如下命令来安装 mysql 对应的 python 包。

```
(python37) # pip install mysqlclient -i https://pypi.tuna.tsinghua.edu.cn/simple
```

2. 在 node1 上初始化 Airflow 数据库

```
(python37) [root@node1 airflow]# airflow db init
```

初始化之后在 MySQL airflow 库下会生成对应的表。

1.7.3.4 创建管理员用户信息

在 node1 节点上执行如下命令，创建操作 Airflow 的用户信息：

```
airflow users create \
    --username airflow \
    --firstname airflow \
    --lastname airflow \
    --role Admin \
    --email xx@qq.com
```

执行完成之后，设置密码为“123456”并确认，完成 Airflow 管理员信息创建。

1.7.3.5 配置 Scheduler HA

1） 下载 failover 组件

登录 https://github.com/teamclairvoyant/airflow-scheduler-failover-controller 下载 airflow-scheduler-failover-controller 第三方组件，将下载好的 zip 包上传到 node1 “/software”目录下。

在 node1 节点安装 unzip，并解压 failover 组件：

```
(python37) [root@node1 software]# yum -y install unzip
(python37) [root@node1 software]# unzip ./airflow-scheduler-failover-controller-master.zip
```

2） 使用 pip 进行安装 failover 需要的依赖包

需要在 node1 节点上安装 failover 需要的依赖包。

```
(python37) [root@node1 software]# cd /software/airflow-scheduler-failover-controller-master
(python37) [root@node1 airflow-scheduler-failover-controller-master]# pip install -e .
```

3） node1 节点初始化 failover

```
(python37) [root@node1 ~]# scheduler_failover_controller init
Adding Scheduler Failover configs to Airflow config file...
Finished adding Scheduler Failover configs to Airflow config file.
Finished Initializing Configurations to allow Scheduler Failover Controller to run. Please update the airflow.cfg with your desired configurations.
```

注意：初始化 airflow 时，会向 airflow.cfg 配置中追加配置，因此需要先安装 airflow 并初始化。

4） 修改 airflow.cfg

首先修改 node1 节点的 AIRFLOW_HOME/airflow.cfg

```
[scheduler_failover]
# 配置 airflow Master 节点，这里配置为 node1,node2，两节点需要免密
scheduler_nodes_in_cluster = node1,node2

#在 1088 行，特别注意，需要去掉一个分号，不然后期自动重启 Scheduler 不能正常启动
airflow_scheduler_start_command = export AIRFLOW_HOME=/root/airflow;nohup airflow scheduler >> ~/airflow/logs/scheduler.logs &
```

配置完成后，可以通过以下命令进行验证 Airflow Master 节点：

```
(python37) [root@node1 airflow]# scheduler_failover_controller test_connection
Testing Connection for host 'node1'
(True, ['Connection Succeeded', ''])
Testing Connection for host 'node2'
(True, ['Connection Succeeded\n'])
```

将 node1 节点配置好的 airflow.cfg 同步发送到 node2、node3、node4 节点上：

```
(python37) [root@node1 ~]# cd /root/airflow/
(python37) [root@node1 airflow]# scp airflow.cfg node2:`pwd`
(python37) [root@node1 airflow]# scp airflow.cfg node3:`pwd`
(python37) [root@node1 airflow]# scp airflow.cfg node4:`pwd`
```

1.7.3.6 启动 Airflow 集群

1） 在所有节点安装启动 Airflow 依赖的 python 包

```
(python37) [root@node1 airflow]# pip install celery==4.4.7 flower==0.9.7 redis==3.5.3
(python37) [root@node2 airflow]# pip install celery==4.4.7 flower==0.9.7 redis==3.5.3
(python37) [root@node3 airflow]# pip install celery==4.4.7 flower==0.9.7 redis==3.5.3
(python37) [root@node4 airflow]# pip install celery==4.4.7 flower==0.9.7 redis==3.5.3
```

2） 在 Master1 节点(node1)启动相应进程

```
#默认后台启动可以使用-D ，这里使用-D 有时不能正常启动 Airflow 对应进程
airflow webserver
airflow scheduler
```

3） 在 Master2 节点(node2)启动相应进程

```
airflow webserver
```

4） 在 Worker1(node3)、Worker2(node4)节点启动 Worker

在 node3、node4 节点启动 Worker：

```
(python37) [root@node3 ~]# airflow celery worker
(python37) [root@node4 ~]# airflow celery worker
```

5） 在 node1 启动 Scheduler HA

```
(python37) [root@node1 airflow]# nohup scheduler_failover_controller start >
/root/airflow/logs/scheduler_failover/scheduler_failover_run.log &
```

至此，Airflow 高可用集群搭建完成。

1.7.3.7 访问 Airflow 集群 WebUI

浏览器输入 node1:8080，查看 Airflow WebUI:

1.7.3.8 测试 Airflow HA

1） 准备 shell 脚本

在 **Airflow 集群所有节点**{AIRFLOW_HOME}目录下创建 dags 目录，准备如下两个

shell 脚本，将以下两个脚本放在$AIRFLOW_HOME/dags 目录下，BashOperator 默认执行脚本时，默认从/tmp/airflow**临时目录查找对应脚本，由于临时目录名称不定，这里建议执行脚本时，在“bash_command”中写上绝对路径。如果要写相对路径，可以将脚本放在/tmp 目录下，在“bash_command”中执行命令写上“sh ../xxx.sh”也可以。

first_shell.sh

```
#!/bin/bash
dt=$1
echo "==== execute first shell ====="
echo "---- first : time is ${dt}"
```

second_shell.sh

```
#!/bin/bash
dt=$1
echo "==== execute second shell ====="
echo "---- second : time is ${dt}"
```

2） 编写 airflow python 配置

```
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.bash import BashOperator

default_args = {
    'owner':'zhangsan',
    'start_date':datetime(2021, 9, 23),
    'retries': 1,  # 失败重试次数
    'retry_delay': timedelta(minutes=5) # 失败重试间隔
}

dag = DAG(
    dag_id = 'execute_shell_sh',
    default_args=default_args,
    schedule_interval=timedelta(minutes=1)
)

first=BashOperator(
    task_id='first',
    #脚本路径建议写绝对路径
    bash_command='sh
/root/airflow/dags/first_shell.sh %s'%datetime.now().strftime("%Y-%m-%d"),
```

```
    dag = dag
)

second=BashOperator(
    task_id='second',
    #脚本路径建议写绝对路径
    bash_command='sh
/root/airflow/dags/second_shell.sh %s'%datetime.now().strftime("%Y-%m-%d"),
    dag=dag
)

first >> second
```

将以上内容写入 execute_shell.py 文件，上传到**所有 Airflow 节点** {AIRFLOW_HOME}/dags 目录下。

3） 重启 Airflow，进入 Airflow WebUI 查看对应的调度

重启Airflow之前首先在node1节点关闭webserver ,Scheduler进程 ,在node2节点关闭 webserver ，Scheduler 进程，在 node3，node4 节点上关闭 worker 进程。

如果各个进程是后台启动，查看后台进程方式：

```
(python37) [root@node1 dags]# ps aux |grep webserver

(python37) [root@node1 dags]# ps aux |grep scheduler

(python37) [root@node2 dags]# ps aux |grep webserver

(python37) [root@node2 dags]# ps aux |grep scheduler

(python37) [root@node3 ~]# ps aux|grep "celery worker"

(python37) [root@node4 ~]# ps aux|grep "celery worker"

找到对应的启动命令对应的进程号，进行 kill。
```

重启后进入 Airflow WebUI 查看任务：

点击“success”任务后，可以看到脚本执行成功日志：

```
AIRFLOW_CTX_EXECUTION_DATE=2021-09-22T17:59:00+00:00
AIRFLOW_CTX_DAG_RUN_ID=scheduled__2021-09-22T17:59:00+00:00
[2021-10-08 23:38:54,290] {subprocess.py:52} INFO - Tmp dir root location:
 /tmp
[2021-10-08 23:38:54,291] {subprocess.py:63} INFO - Running command: ['bash', '-c', 'sh /root/airflow/dags/first_shell.sh 2021-10-08']
[2021-10-08 23:38:54,306] {subprocess.py:74} INFO - Output:
[2021-10-08 23:38:54,311] {subprocess.py:78} INFO - ==== execute first shell ====
[2021-10-08 23:38:54,312] {subprocess.py:78} INFO - ---- first : time is 2021-10-08
[2021-10-08 23:38:54,313] {subprocess.py:82} INFO - Command exited with return code 0
[2021-10-08 23:38:54,419] {taskinstance.py:1218} INFO - Marking task as SUCCESS. dag_id=execute_shell_sh, task_id=first, execution_date=2021
[2021-10-08 23:38:54,499] {local_task_job.py:151} INFO - Task exited with return code 0
[2021-10-08 23:38:54,544] {local_task_job.py:261} INFO - 0 downstream tasks scheduled from follow-on schedule check
```

```
AIRFLOW_CTX_TASK_ID=second
AIRFLOW_CTX_EXECUTION_DATE=2021-09-22T17:59:00+00:00
AIRFLOW_CTX_DAG_RUN_ID=scheduled__2021-09-22T17:59:00+00:00
[2021-10-08 23:38:55,507] {subprocess.py:52} INFO - Tmp dir root location:
 /tmp
[2021-10-08 23:38:55,508] {subprocess.py:63} INFO - Running command: ['bash', '-c', 'sh /root/airflow/dags/second_shell.sh 2021-10-08']
[2021-10-08 23:38:55,523] {subprocess.py:74} INFO - Output:
[2021-10-08 23:38:55,529] {subprocess.py:78} INFO - ==== execute second shell ====
[2021-10-08 23:38:55,529] {subprocess.py:78} INFO - ---- second : time is 2021-10-08
[2021-10-08 23:38:55,530] {subprocess.py:82} INFO - Command exited with return code 0
[2021-10-08 23:38:55,704] {taskinstance.py:1218} INFO - Marking task as SUCCESS. dag_id=execute_shell_sh, task_id=second, execution_date=20
[2021-10-08 23:38:55,809] {local_task_job.py:151} INFO - Task exited with return code 0
```

4） 测试 Airflow HA

当我们把 node1 节点的 websever 关闭后，可以直接通过 node2 节点访问 airflow

webui：

在 node1 节点上，查找“scheduler”进程并 kill，测试 scheduler HA 是否生效：

```
(python37) [root@node1 ~]# ps aux|grep scheduler
root      23744  0.9  3.3 326940 63028 pts/2    S    00:08   0:02 airflow scheduler
-- DagFileProcessorManager
#kill 掉 scheduler 进程
(python37) [root@node1 ~]# kill -9 23744
```

#访问 webserver webui

#在 node1 节点查看 scheduler_failover_controller 进程日志中有启动 schudler 动作，注意：

这里是先从 node1 启动，启动不起来再从其他 Master 节点启动 Schduler。

```
[2021-10-09 00:19:14,616] {failover_controller.py:157} INFO - Starting to Check if Scheduler on host 'node1' is running...
[2021-10-09 00:19:14,640] {failover_controller.py:202} INFO - Finished Checking if Scheduler on host 'node1' is running. is running: False
[2021-10-09 00:19:14,641] {failover_controller.py:95} WARNING - Scheduler is not running on Active Scheduler Node 'node1'
[2021-10-09 00:19:14,641] {failover_controller.py:145} INFO - Starting Scheduler on host 'node1'...
nohup: redirecting stderr to stdout
[2021-10-09 00:19:14,653] {failover_controller.py:148} INFO - Finished starting Scheduler on host 'node1'
[2021-10-09 00:19:14,653] {failover_controller.py:97} INFO - Pausing for 35 seconds to allow the Scheduler to Start
```