# 常用命令及操作

## CLI命令

airflow webserver -p 8080 运行webserver ,默认访问端口-p 8080 (可以去掉)

ps -ef|grep -Ei '(airflow-webserver)'| grep master | awk '{print $2}'|xargs -i kill {} 关闭webserver

airflow scheduler 守护进程运行调度器

airflow worker 守护进程运行celery worker

airflow worker 启动celery worker (不能用根用户)：

airflow worker -c 1 守护进程运行celery worker并指定任务并发数为1

airflow pause dag\_id 暂停任务

airflow unpause dag\_id 取消暂停，等同于在管理界面打开off按钮

airflow list\_dags 查看dags列表

airflow list\_tasks dag\_id 查看task列表

airflow clear dag\_id 清空任务实例 -r自动清理掉所有state为running的任务

airflow trigger\_dag dag\_id -r RUN\_ID -e EXEC\_DATE 运行整个dag文件

airflow run dag\_id task\_id execution\_date 运行task

airflow resetdb 重置数据库

airflow initdb 初始化数据库

airflow clear -h 查看参数 其他也可以-h

airflow backfill -s START -e END --mark\_success DAG\_ID 在特定情况下，修改DAG后，为了避免当前日期之前任务的运行，可以使用填补特定时间段的任务。

## 从数据库中完全删除某个DAG的信息

set @dag\_id = 'BAD\_DAG';

delete from airflow.xcom where dag\_id = @dag\_id;

delete from airflow.task\_instance where dag\_id = @dag\_id;

delete from airflow.sla\_miss where dag\_id = @dag\_id;

delete from airflow.log where dag\_id = @dag\_id;

delete from airflow.job where dag\_id = @dag\_id;

delete from airflow.dag\_run where dag\_id = @dag\_id;

delete from airflow.dag where dag\_id = @dag\_id;

比如在数据库进行如下查询

delete from xcom where dag\_id = 'sleep4';

delete from task\_instance where dag\_id = 'sleep4';

delete from sla\_miss where dag\_id = 'sleep4';

delete from log where dag\_id = 'sleep4';

delete from job where dag\_id = 'sleep4';

delete from dag\_run where dag\_id = 'sleep4';

delete from dag where dag\_id = 'sleep4';

# 安装和配置

## 在线安装

pip install apache-airflow

## 离线安装

以 airflow 1.10 版本为例，其他版本的操作也是一致的。

先在联网环境下载安装包，联网的计算机操作系统与 Python 版本最好与生产环境一致，如果不一致，需要为 pip 指定操作系统和 Python 版本。

$ mkdir airflow

$ cd airflow

$ pip download apache-airflow

也可以是apache-airflow[all],这里我们不需要全部，全部可能会有很多依赖

请等待下载完成。

然后将上述文件打包传输至生产环境解压后，进入 airflow目录，执行：

$cd airflow

$ pip install apache-airflow --no-index -f ./

## 配置

airflow version

#首次运行该命令之后才会产生配置文件【第一次运行Airflow时，它将创建一个名为airflow.cfg在你的$AIRFLOW\_HOME文件夹(~/airflow默认情况下）】

### 存储路径

先在/etc/profile中配置airflow路径export AIRFLOW\_HOME=/mnt/airflow

然后刷新source /etc/profile

接着使用airflow version，就会在你设置的AIRFLOW\_HOME的目录创建文件和文件夹

### 配置选项

#### 配置初始化数据库，用来储存元数据等

这里用postgresql做例子

格式：sql\_alchemy\_conn =   
postgresql+psycopg2://usernm:passwd@172.16.100.10:5432/dbname

例：sql\_alchemy\_conn =   
postgresql+psycopg2://postgres:postgres@172.16.100.10:5432/airflow1

数据库需要自己先创建好

#### 设置不使用默认dag例子

load\_examples = False

#### 设置不自动backfill

详细信息见下面4.Tips中的Backfill

catchup\_by\_default = False

#### 设置默认时区

详细信息见下面4.Tips中的TimeZone

# 实例

## 编写

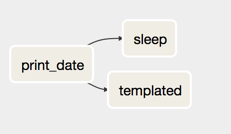
DAG脚本就是一个Python脚本，这个脚本需要生成一个Dag对象和若干个参数化了的Operator对象（即Task）。   
DAG的编写十分简单，主要有四个步骤

1. import DAG类和若干operater类以及必要的Python模块
2. 设定默认参数，创建DAG对象。
3. 提供必要的参数（比如task\_id和dag)，创建Task（即Operator对象）
4. 设定Task的上下游依赖关系

下面结合[官网教程的例子](http://incubator-airflow.readthedocs.io/en/latest/tutorial.html)来说明这四个步骤和一些参数的含义。

特别提醒，对于Python2，脚本中含有中文的话（即使是注释），就必须要加下面的文件头来设定文件编码。

1. *# -\*- coding: UTF-8 -\*-*
2. """
3. Code that goes along with the Airflow tutorial located at:
4. https://github.com/airbnb/airflow/blob/master/airflow/example\_dags/tutorial.py
5. """
6. *# ==================== import DAG和若干模块 ====================*
7. from airflow import DAG
8. from airflow.operators.bash\_operator import BashOperator
9. from datetime import datetime, timedelta
10. *# ==================== 创建一个DAG对象 ====================*
11. *# default\_args设置的是DAG的通用参数，这些通用参数会直接传递给DAG下属的所有Task，这些参数也可以在创建Task时传入，*
12. default\_args = {
13. 'owner': 'airflow', *# 这个DAG的所有者，会在Web UI上显示，主要用于方便管理*
14. 'depends\_on\_past': False, *# 是否依赖于过去。如果为True，那么必须要昨天的DAG执行成功了，今天的DAG才能执行。*
15. 'start\_date': datetime(2015, 6, 1), *# DAG的开始时间，比如这里就是从2015年6月1日开始执行第一个DAG。这个参数会影响到部署上线时回填DAG的数量。一般建议写成上线时间的前一天（因为这里的start\_date指的是execute\_date，而Airflow执行的逻辑是，今天的同一时间执行昨天的任务，比如execute\_date=2018-03-01，每天凌晨3点执行，则会在2018-03-02 03:00:00启动这个DAG。特别地，这个参数必须一个datetime对象，不可以用字符串。*
16. 'email': ['airflow@example.com'],# 出问题时，发送报警Email的地址，可以填多个，用逗号隔开。
17. 'email\_on\_failure': False, *# 任务失败且重试次数用完时是否发送Email，推荐填True。*
18. 'email\_on\_retry': False, *# 任务重试时是否发送Email*
19. 'retries': 1, *# 任务失败后的重试次数*
20. 'retry\_delay': timedelta(minutes=5), *# 重试间隔，必须是timedelta对象。*
21. *# 'queue': 'bash\_queue', 队列，默认是default，决定实际执行任务会发送到哪个worker*
22. *# 'pool': 'backfill', pool是一个分类限制并发量的设计，目前来说可以忽略，默认所有的Task都在一个pool里。*
23. *# 'priority\_weight': 10, 优先级权重，在任务需要排队时而你需要优先执行某些任务时会有用*
24. *# 'end\_date': datetime(2016, 1, 1), # 结束时间，一般线上任务都会一直跑下去，所以没必要设置。*
25. }
26. dag = DAG('tutorial', default\_args=default\_args,schedule\_interval="0 3 \* \* \*") *# 第一个参数固定为dag的名字,schedule\_interval为执行时间间隔，同crontab的语法，在这个例子中表示每天凌晨3点执行*
27. *# ==================== 创建若干Task ====================*
28. *# t1, t2 and t3 are examples of tasks created by instantiating operators*
29. *# 下面三个Task，都是BashOperator对象，这在Airflow乃至所有的调度系统中都是最常用的对象之一，其效果是执行传入的bash 命令。除了例子中的这些命令之外，和Spark结合，可以写spark 命令，和sqoop结合，可以写sqoop 命令，适用面非常广，而且编写简单，直接，只需要三个参数：task\_id-任务id，bash\_command-bash命令，dag-对应的dag对象*
30. t1 = BashOperator(
31. task\_id='print\_date',
32. bash\_command='date',
33. dag=dag)
34. t2 = BashOperator(
35. task\_id='sleep',
36. bash\_command='sleep 5',
37. retries=3,
38. dag=dag)
39. *# 下面的bash命令用到了macros（宏）这个特性，这里用'{{ ds }}'来指代任务的执行日期，比如这个任务的某一天的执行日期是2018-01-01，那么'{{ ds }}'就会被替换成'2018-01-01'，这在做一些和时间密切相关的变量时非常有用。这个特性背后的机制是Jinja2模版的render(渲染)功能，每个Operator都可以指定某些参数用于render，比如BashOperator就特别指定了bash\_command，要有更深入全面的认识，则需要去查看Operator的源码。*
40. templated\_command = """
41. {% for i in range(5) %}
42. echo "{{ ds }}"
43. echo "{{ macros.ds\_add(ds, 7)}}"
44. echo "{{ params.my\_param }}"
45. {% endfor %}
46. """
47. t3 = BashOperator(
48. task\_id='templated',
49. bash\_command=templated\_command,
50. params={'my\_param': 'Parameter I passed in'},
51. dag=dag)
52. *# ==================== 设定Task依赖关系 ====================*
53. *# 设定依赖有两种方式，一种是使用对象的方法，set\_upstream和set\_downstream来设置上下游依赖，另一种是使用运算符，比如下面的 t3 << t1，表示t3是t1的下游对象。推荐使用第二种，更简单直观，而且可以在一行里写出整个依赖链条，比如说t1 下游t2，t2下游t3，就可以写成t1 >> t2 >> t3。*
54. t2.set\_upstream(t1)
55. t3 << t1

上面这个脚本最终会得到下图的任务依赖关系   


## 调试

在编写完成之后就进入调试阶段，这一阶段主要的目的就是DeBug，步骤如下：   
1. **语法检查**：用Python命令执行该脚本，确保没有语法错误。   
2. **上传到服务器**：将脚本上传至服务器，使用airflow list\_dags | grep your\_dag\_name（你的dag的名字，比如上面的例子里，就是tutorial)来检查Airflow是否识别出了你的dag。一般来说，只要第一个Python运行没问题，就能识别出来。   
3. **DAG测试**：在服务器上输入命令 airflow backfill your\_dag\_name -s '2018-03-06' -e '2018-03-07' -l。backfill命令本身是用于回填的，-s表示开始时间，-e表示结束时间(到该时间为止，比如在这条语句中，如果DAG的执行间隔为1天，则只会生成03-06的DAG Run，不会生成03-07的），使用该命令可以回填多个DAG Run，按当前语句设置则只会生成一个，适合用于测试整个DAG的运行状况，最后有个-l参数，表示些任务在本地机器上执行（在部署好的集群上做测试时，这样做可以仅仅更新用于测试服务器上的DAG文件，直到测试完成之后再更新worker服务器上的DAG文件）。   
4. **Task Debug**:如果在3中的测试出了问题，系统会提示是哪个Task出了问题，这时候首先修改代码，然后重复1,2的操作，再执行airflow test your\_dag\_name bug\_task\_id 2018-03-06，单独测试这个Task的执行情况。依此法修复所有有问题的task。然后再执行命令airflow clear your\_dag\_id，清理干净所有的Task，最后重复3的操作，直到能成功运行完所有的task。整个调试阶段就算完成了。

## 部署

当完成了调试之后，就可以把代码部署到线上了。   
Airflow的生产环境一般使用Celery Worker来执行任务，可以搞出分布式的worker，方便在任务量增加之后通过增加机器的方式来保证计算的效率。相应的，这就要求所有机器上的DAG脚本都是一致的。因此在调试完成之后，就需要把所有Worker上的DAG脚本都更新好。最后在Scheduler所在的服务器，执行airflow unpause your\_dag\_id，激活这个DAG。然后在Airflow Web UI上检查这个DAG是否已经在激活列表里，是否已经在回填DAG Run（在激活之后会自动回填，比如设定DAG 的start\_date为2018-03-05，在2018-03-07上线激活的话，会自动回填2018-03-05和2018-03-06两天的DAG Run）

## 总结运维的一般流程

经过一段时间的使用，结合多次救火的经验，我大致总结出了一套简单的运维流程，可以节约维护现有DAG的时间，腾出更多的时间来开发新的DAG。

1. 每天上班第一件事，打开Airflow的Web面板，查看DAG列表中的Recent Tasks，可以了解到最近的DAG Run的Task的执行情况概括，如果全是深绿色(success)，万事OK，又度过了安全的一天，可以开始干点别的了。
2. 如果不幸看到了浅绿色(running)或者红色(failed)，那么就是出问题了。如果是running，说明Task被卡住，一直无法完成。这个处理相对麻烦些，首先，要让DAG停下来，也就是把DAG Run的state设定为failed，让DAG Run停止运行。接着，登陆到Web所在的服务器，执行airflow clear you\_dag -r，会自动清理掉所有state为running的任务，也可以再加上-s execute\_date来限定时间区间，更多细节请使用airflow clear -h命令来学习。总之这一步主要是让DAG停下来。然后下一步再开始排查。使用airflow test dag\_id task\_id execute\_date来测试被卡住的task。如果test过程中报错，就根据log来排查，直到能顺利运行完成。由于Airflow是使用Operator来封装各种操作的，可能实际上有些关键的log信息，可能并没有传递给Airflow的log。如果出现了这样的问题，就需要手动地在服务器上执行相应操作的shell命令来重现问题了。当Task能够顺利运行之后，就可以尝试把DAG Run的state改为running，这样它就会自动启动那些被clear命令删掉的Task Instance，自动跳过已经执行成功的Task Instance，直接从卡住的Task开始执行。   
   在实际使用中，DAG Run failed的情况更常见，Debug方法与running的相似，但因为DAG Run已经结束了运行，所以不再需要在服务器上执行clear命令来停止任务，只需要在测试Task能顺利运行之后，在Browse –> Task Instance下把相应的Task Instance删除掉，然后再把DAG Run设置为running就可以了。
3. 补充说明：为什么在2.中，需要先删除掉有问题的Task Instance再重新启动DAG Run呢？这是因为，一个DAG Run，管理着它相应Task的Instance，在实际执行时，会按照执行图，从所有的无上游结点开始向下遍历，逐个启动Task Instance并等其执行完成。当从失败中恢复时，如果之前的Task Instance已经执行结束（success或者failed），那么DAG Run会直接跳过，success是跳过当前Task Instance，前进到下游Task，而failed则是直接让整个DAG Run failed。所以，为了保证DAG Run能顺利执行完成全部任务，就需要删除掉这些有问题的Task Instance，这样以来，当DAG Run重启后，会自动生成这些Task Instance。这样的机制在一定程度上节约了出错恢复需要的时间，还是挺有用的，代价就是恢复时稍微麻烦了一点，不能简单粗暴地直接重启。

# 分布式使用celery

celery使用rabbitmq作为broker

需先安装celery

pip install apache-airflow[celery]

#在airflow.cfg中配置

executor = CeleryExecutor

#celery4.x的broker需要填pyamqp而不是amqp

broker\_url = pyamqp://myuser:mypassword@centos01:5672/myvhost

# 设定结果存储后端result\_backend，可以与broker\_url相同，不需是pyamqp

result\_backend = amqp://myuser:mypassword@centos01:5672/myvhost

或者

result\_backend = db+postgresql+psycopg2://postgres:postgres@192.168.3.191:5432/airflow

注意账号myuser密码mypassword还有myvhost别写错

注：！！！启动worker时不能用root用户启动，否则会报错，除非进行修改。

但是分布式worker节点必须用普通用户启动，否则节点会不执行操作，即使修改成可以使用root用户也不行

# Tips

Airflow有自己的一套设计逻辑，有一些地方与一般的认知不一样，这里特别说明一下。

* 每个DAG Run都有一个execute\_date，这个日期就是宏变量中的ds，代表执行日期。但这个执行日期不是我们理解的，启动任务的日期，而是启动任务日期的前一天。也就是说，execute\_date=2018-03-06的DAG Run,会在2018-03-07这一天，相同的时间点上生成。这是需要特别注意的。
* 如果ETL任务与执行时间无关，只是要求每隔一段时间执行一次，那么可以加一个前置的Task，将所有无上游的任务的上游都设置为这个Task。这个Task会自动跳过之前若干天的任务，保证只执行最近一天的任务。在激活DAG时，这样做可以大大减少初始化消耗的资源。这个Task需要用LatestOnlyOperator来创建，属于模块airflow.operators.latest\_only\_operator，比如在上面的代码例子中，t1(print\_date)是唯一的无上游任务，我们只要加上以下语句，在激活就可以自动跳过前面若干天的任务，只执行最新任务。代码如下：

1. from airflow.operators.latest\_only\_operator import LatestOnlyOperator
2. lastest = LatestOnlyOperator(task\_id='lastest',dag=dag)
3. lastest >> t1

## DAG Runs和Task Instances

前面所写的都是针对一个DAG的，显然我们实际工作中肯定需要管理多个DAG的，如果我们就是想迅速找出某些有问题的DAG Run或者Task Instance呢？这个时候就需要弄个表格+筛选功能了。在页面的顶部导航栏上，Browse –> Task Instance/DAG Runs，就可以看到所有的Task Instance或DAG Run的情况。可以使用Add Filter添加筛选条件，或者使用search 框来快速搜索，然后勾选想要处理的Task Instance或DAG Run，使用With selected来执行相应的批量操作。

## **TimeZone**

### **在airflow家目录下修改airflow.cfg，设置**

**default\_timezone = Asia/Shanghai**

### **进入airflow包的安装位置,也就是site-packages的位置,以下修改文件均为相对位置**

**# 这我安装airflow包的位置(大家自行参考)**

**cd /opt/dist/anaconda3/lib/python3.6/site-packages**

### **修改airflow/utils/timezone.py**

**3.1 在 utc = pendulum.timezone(‘UTC’) 这行(第27行)代码下添加,**

**from airflow import configuration as conf**

**try:**

**tz = conf.get("core", "default\_timezone")**

**if tz == "system":**

**utc = pendulum.local\_timezone()**

**else:**

**utc = pendulum.timezone(tz)**

**except Exception:**

**pass**

**3.2 修改utcnow()函数 (在第69行)**

**原代码 d = dt.datetime.utcnow()**

**修改为 d = dt.datetime.now()**

### **修改airflow/utils/sqlalchemy.py**

**在utc = pendulum.timezone(‘UTC’) 这行(第37行)代码下添加**

**from airflow import configuration as conf**

**try:**

**tz = conf.get("core", "default\_timezone")**

**if tz == "system":**

**utc = pendulum.local\_timezone()**

**else:**

**utc = pendulum.timezone(tz)**

**except Exception:**

**pass**

### **修改airflow/www/templates/admin/master.html(第31行)**

**把代码 var UTCseconds = (x.getTime() + x.getTimezoneOffset()\*60\*1000);**

**改为 var UTCseconds = x.getTime();**

**把代码 "timeFormat":"H:i:s %UTC%",**

**改为 "timeFormat":"H:i:s",**

**最后重启airflow-webserver即可**

## ****Backfill设置不自动回填运行****

可以把它简单的理解成当你错过了某一次执行时间之后，往回去补充执行的行为。我们可以使用手动方法来执行这个行为

airflow backfill sensors -s 2015-06-01 -e 2015-06-07

他会回补这个时间段开始的 和 -e 后面时间段结束期间所有的任务执行。回补的意思就是把没有执行的操作都执行一遍。

这个特性想法很好，但是自自动触发的时候不注意就会产生非常不可预期的问题。

比如刚才在上面我们谈到的在给 dag 配置的时候指定的 default\_args 上面有一个参数 start\_date。如果我们不给 dag 指定不回补，那么 airflow 会默认回补从系统当前时间到我们指定的 start\_date 期间的任务。如果这个参数设置得不恰当会打来恐怖的回补，所以一般我都会禁用回补。

sensors\_dag = DAG(

'sensors\_dag',

default\_args=default\_args,

schedule\_interval=u'0 0 \* \* \*',

catchup=False)

指定 catchup=False 。让他从最新的任务时间点开始执行。这个在官方文档 Scheduling & Triggers 一章有详细提到。

如果不想回填，可以在dag参数default\_args中设置catchup=False或'catchup':False。或者在airflow.cfg中配置catchup\_by\_default=False

## 不能用root用户启动celery worker

解决方案一：修改airlfow源码，在celery\_executor.py中强制设置C\_FORCE\_ROOT

from celery import Celery, platforms

在app = Celery(…)后新增

platforms.C\_FORCE\_ROOT = True

重启即可

解决方案二：在容器初始化环境变量的时候，设置C\_FORCE\_ROOT参数，以零侵入的方式解决问题

强制celery worker运行采用root模式

vim /etc/profiole

export C\_FORCE\_ROOT=True

# 编写

超时标记为失败，进行重试，只有一个dag会运行，重试10次任务失败

*# coding: utf-8***from** datetime **import** datetime, timedelta  
**import** airflow   
**from** airflow **import** DAG  
**from** airflow.operators.bash\_operator **import** BashOperator  
  
bash1 = **'''  
path='/mnt'  
tee -a $path/aa.txt <<< 'task4 start'  
tee -a $path/aa.txt <<< `date`  
tee -a $path/aa.txt <<< 'start sleep 20'  
sleep 20  
tee -a $path/aa.txt <<< `date`  
'''**  
default\_args = {  
 **'start\_date'**: datetime(2019, 7, 7),  
 **'retries'**: 10,  
 **'retry\_delay'**: timedelta(seconds=5),  
 **'execution\_timeout'**: timedelta(seconds=10),  
 *# 'depends\_on\_past': True,  
 #'wait\_for\_downstream': True 上游seccess完毕下游才会开始*}  
  
dag = DAG(**'sleep4'**, default\_args=default\_args, schedule\_interval=timedelta(seconds=30), max\_active\_runs=1)  
*# max\_active\_runs=1为最多只运行一个dag*  
*# dagrun\_timeout=timedelta(seconds=10))用这个不会重试任务，整个dag会直接失败*t1 = BashOperator(task\_id=**'task4'**, bash\_command=bash1, dag=dag)  
*# , trigger\_rule="all\_done"*

# 项目调度

*# coding: utf-8***from** datetime **import** datetime, timedelta  
**import** airflow   
**from** airflow **import** DAG  
**from** airflow.operators.bash\_operator **import** BashOperator  
  
bash1 = **'''  
path='/mnt/crontab\_test/DataFusion\_V2.0/logs'  
path1='/mnt/crontab\_test/DataFusion\_V2.0/DataAggregate\_v2'  
path2='/mnt/crontab\_test/DataFusion\_V2.0/AutoMapBuilding'  
logname="$(date +"%Y-%m-%d-%H")"  
tee -a $path/$logname <<< `date`  
cd $path1  
python -u horizon.py >> $path/$logname 2>&1  
cd $path2  
python -u AutoMapBuilding.py >> $path/$logname 2>&1  
'''**default\_args = {  
 **'start\_date'**: datetime(2019, 7, 11),  
 **'retries'**: 10,  
 **'retry\_delay'**: timedelta(seconds=10),  
 **'execution\_timeout'**: timedelta(hours=2),  
 *# 'depends\_on\_past': True,* **'wait\_for\_downstream'**: **True**,  
}  
  
dag = DAG(**'main\_process'**, default\_args=default\_args, schedule\_interval=timedelta(hours=1), max\_active\_runs=1)  
*# dagrun\_timeout=timedelta(seconds=30))用这个不会重试任务，整个dag会直接失败*t1 = BashOperator(task\_id=**'task4'**, bash\_command=bash1, dag=dag)  
*# , trigger\_rule="all\_done"*