**harrychinese 刘忠武**
面朝大海, 春暖花开
我的开源
http://anydbtest.codeplex.com



昵称：harrychinese
园龄：6年4个月
粉丝：55
关注：10
+加关注

< 2017年3月 >

| 日 | 一 | 二 | 三 | 四 | 五 | 六 |
|---|---|---|---|---|---|---|
| 26 | 27 | 28 | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| 19 | 20 | 21 | 22 | 23 | 24 | 25 |
| 26 | 27 | 28 | 29 | 30 | 31 | 1 |
| 2 | 3 | 4 | 5 | 6 | 7 | 8 |

搜索
找找看
谷歌搜索

常用链接
我的随笔
我的评论
我的参与
最新评论
我的标签

最新随笔
1. 利用 yEd 软件做元数据管理
2. jqgrid again
3. Web 前端
4. HTTP Basic Authentication
5. Markdown 写作工具选择
6. jupyter notebook + pyspark 环境搭建
7. 快速搭建大数据分析环境
8. 系统研究Airbnb开源项目airflow
9. pip高级使用技巧以及搭建自己的pypi服务器

# 系统研究Airbnb开源项目airflow

开源项目airflow的一点研究

调研了一些几个调度系统, airflow 更满意一些. 花了些时间写了这个博文, 这应该是国内技术圈中最早系统性研究airflow的文章了. 转载请注明出处 http://www.cnblogs.com/harrychinese/ .

==========
airflow概况
==========
文档:
http://airflow.readthedocs.org/en/latest/
几个调度系统的比较, 可参考:
http://www.rigongyizu.com/about-workflow-schedule-system/

Principles:
动态性: Airflow pipeline是以python code形式做配置, 灵活性没得说了.
扩展性: 可以自定义operator(运算子), 有几个executor(执行器)可供选择.
优雅:pipeline的定义很简单明了, 基于jinja模板引擎很容易做到脚本命令参数化.
扩展性:模块化的结构, 加上使用了message queue来编排多个worker(启用CeleryExcutor), airflow可以做到无限扩展.

竞品:
airflow并不是data streaming方案, 所以不是Spark Streaming/Storm的竞品. 和airflow类似的有: Apache Oozie, Linkedin Azkaban.

比较优势:
linkedin的Azkaban很不错, UI尤其很赞, 使用java properties文件维护上下游关系, 任务资源文件需要打包成zip, 部署不是很方便.
Apache Oozie, 使用XML配置, Oozie任务的资源文件都必须存放在HDFS上. 配置不方便同时也只能用于Hadoop.
Spotify的 Luigi, UI 太烂.
airflow 总体都不错, 有实用的UI, 丰富的cli工具, Task上下游使用python编码, 能保证灵活性和适应性.

==========
概念:
==========
不用多说概念自然非常重要, 这是理解airflow的基础.

---
Operators:
---
基本可以理解为一个抽象化的task, Operator加上必要的运行时上下文就是一个task. 有三类Operator:
1. Sensor(传感监控器), 监控一个事件的发生.
2. Trigger(或者叫做Remote Excution), 执行某个远端动作, (我在代码中没有找到这个类别)
3. Data transfer(数据转换器), 完成数据转换

---
Hooks:
---
Hook是airflow与外部平台/数据库交互的方式, 一个Hook类就像是一个JDBC driver一样. airflow已经实现了jdbc/ftp/http/webhdfs很多hook. 要访问RDBMS数据库 有两类Hook可供选择, 基于原生

10. [转]Informatica vs SSIS

我的标签

Python(57)

工具(33)

数据仓库(21)

AnyDbTest(10)

编程随想(7)

大数据(7)

单元测试(7)

linux(6)

Oracle(6)

web(6)

更多

随笔档案

2017年2月 (1)

2017年1月 (1)

2016年10月 (1)

2016年9月 (2)

2016年3月 (1)

2016年2月 (2)

2016年1月 (3)

2015年11月 (1)

2015年10月 (3)

2015年9月 (1)

2015年8月 (3)

2015年7月 (1)

2015年6月 (2)

2015年3月 (1)

2015年2月 (1)

2015年1月 (1)

2014年9月 (1)

2014年8月 (1)

Python DBAPI的Hook和基于JDBC的Hook, 以Oracle为例,

OracleHook, 是通过cx_Oracle 访问Oracle数据, 即原生Python binding, 有些原生的Hook支持Bulk load.

JdbcHook, 是通过jaydebeapi+Oracle JDBC访问Oracle数据

Tasks: task代表DAG中的一个节点, 它其实是一个BaseOperator子类.

Task instances, 即task的运行态实例, 它包含了task的status(成功/失败/重试中/已启动)

Job: Airflow中Job很少提及, 但在数据库中有个job表, 需要说明的是Job和task并不是一回事, Job可以简单理解为Airflow的批次, 更准确的说法是同一批被调用task或dag的统一代号. 有三类Job, 分别SchedulerJob/BackfillJob/LocalTaskJob, 对于SchedulerJob和BackfillJob, job指的是指定dag这次被调用的运行时代号, LocalTaskJob是指定task的运行时代号.

---

Connections:

---

我们的Task需要通过Hook访问其他资源, Hook仅仅是一种访问方式, 就像是JDBC driver一样, 要连接DB, 我们还需要DB的IP/Port/User/Pwd等信息. 这些信息不太适合hard code在每个task中, 可以把它们定义成Connection, airflow将这些connection信息存放在后台的connection表中. 我们可以在WebUI的Admin->Connections管理这些连接.

---

Variables:

---

Variable 没有task_id/dag_id属性, 往往用来定义一些系统级的常量或变量, 我们可以在WebUI或代码中新建/更新/删除Variable. 也可以在WebUI上维护变量.

Variable 的另一个重要的用途是, 我们为Prod/Dev环境做不同的设置, 详见后面的开发小节.

---

XComs:

---

XCom和Variable类似, 用于Task之间共享一些信息. XCom 包含task_id/dag_id属性, 适合于Task之间传递数据, XCom使用方法比Variables复杂些. 比如有一个dag, 两个task组成(T1->T2), 可以在T1中使用xcom_push()来推送一个kv, 在T2中使用xcom_pull()来获取这个kv.

---

Trigger Rules:

---

可以为dag中的每个task都指定它的触发条件, 这里的触发条件有两个维度, 以T1&T2->T3 这样的dag为例:

一个维度是: 要根据dag上次运行T3的状态确定本次T3是否被调用, 由DAG的default_args.depends_on_past参数控制, 为True时, 只有上次T3运行成功, 这次T3才会被触发

另一个维度是: 要根据前置T1和T2的状态确定本次T3是否被调用, 由T3.trigger_rule参数控制, 有下面6种情形, 缺省是all_success.

all_success: (default) all parents have succeeded

all_failed: all parents are in a failed or upstream_failed state

all_done: all parents are done with their execution

one_failed: fires as soon as at least one parent has failed, it does not wait for all parents to be done

one_success: fires as soon as at least one parent succeeds, it does not wait for all parents to be done

dummy: dependencies are just for show, trigger at will

---

分支的支持:

---

airflow有两个基于PythonOperator的Operator来支持dag分支功能.

ShortCircuitOperator, 用来实现流程的判断. Task需要基于ShortCircuitOperator, 如果本Task返回为False的话, 其下游Task将被skip; 如果为True的话, 其下游Task将会被正常执行. 尤其适合用在其下游都是单线节点的场景.

BranchPythonOperator, 用来实现Case分支. Task需要基于BranchPythonOperator, airflow会根据本task的返回值(返回值是某个下游task的id),来确定哪个下游Task将被执行, 其他下游Task将被

2014年7月 (1)
2014年6月 (1)
2014年5月 (3)
2014年4月 (2)
2014年3月 (3)
2014年2月 (4)
2014年1月 (2)
2013年11月 (3)
2013年10月 (3)
2013年7月 (5)
2013年6月 (3)
2013年3月 (1)
2012年12月 (2)
2012年11月 (2)
2012年10月 (5)
2012年9月 (1)
2012年8月 (2)
2012年4月 (4)
2012年2月 (2)
2012年1月 (5)
2011年12月 (1)
2011年11月 (6)
2011年10月 (4)
2011年9月 (2)
2011年8月 (17)
2011年7月 (2)
2011年6月 (4)
2011年5月 (5)
2011年4月 (2)
2011年3月 (8)
2011年2月 (3)
2011年1月 (8)

skip.

---
airflow系统表:
---
connection 表:
我们的Task往往需要通过jdbc/ftp/http/webhdfs方式访问其他资源, 一般地访问资源时候都需要一些签证, airflow允许我们将这些connection以及鉴证存放在connection表中. 可以现在WebUI的Admin->Connections管理这些连接, 在代码中使用这些连接.
需要说明的是, connection表有2个id栏位, 一个是id, 一个是conn_id, id栏位是该表的PK, conn_id栏位是connection的名义id, 也就是说我们可以定义多个同名的conn_id, 当使用使用时airflow将会从同名的conn_id的列表中随机选一个, 有点基本的load balance的意思.
user 表 :
包含user的username和email信息. 我们可以在WebUI的Admin->Users管理.
variable 表 :
包含定义variable
xcom 表:
包含xcom的数据
dag 表:
包含dag的定义信息, dag_id是PK(字符型)
dag_run 表:
包含dag的运行历史记录, 该表也有两个id栏位, 一个是id, 一个是run_id, id栏位是该表的PK, run_id栏位是这次运行的一个名字(字符型), 同一个dag, 它的run_id 不能重复.
物理PK: 即id栏位
逻辑PK: dag_id + execution_date 组合
execution_date 栏位, 表示触发dag的准确时间
注意: 没有 task 表:
airflow的task定义在python源码中, 不在DB中存放注册信息.
task_instance 表:
物理PK: 该表没有物理PK
逻辑PK: dag_id + task_id + execution_date 组合.
execution_date 栏位, 表示触发dag的准确时间,是datetime类型字段
start_date/end_date 栏位,表示执行task的起始/终止时间, 也是datetime类型字段
job 表:
包含job(这里可以理解为批次)的运行状态信息

==========
操作
==========
---
安装和配置
---
1. 操作系统:
Airflow不能在Windows上部署, 原因是使用了 gunicorn 作为其web server(目前gunicorn还不支持Windows), 另外我也在代码中看到一些hard code了一些bash命令.
2. shell 解释器:
操作系统应该安装bash shell, 运行airflow服务的账号, 最好默认使用bash shell.
3. Python版本:
目前Airflow只是实验性地支持Python3, 推荐使用Python2.7
4. Backend 数据库:
SqlAlchemy 支持的数据库都可以作为Backend, 甚至Sqlite(非常适合做Demo或临时体验一下), 官方推荐采用 MySQL 或 Postgres. 我试了Oracle, 但最终还是以失败告终. MySQL 应该使用 mysqlclient 包, 我简单试了mysql-connector-python 有报错.
5. Executor的选择:
有三个 Executor 可供选择, 分别是: SequentialExecutor 和 LocalExecutor 和 CeleryExecutor, SequentialExecutor仅仅适合做Demo(搭配Sqlite backend), LocalExecutor 和 CeleryExecutor 都可用于生产环境, CeleryExecutor 将使用 Celery 作为Task执行的引擎, 扩展性很好, 当然配置也更复杂, 需要先setup Celery的backend(包括RabbitMQ, Redis)等. 其实真正要求扩展性的场景并不多, 所以LocalExecutor 是一个很不错的选择了.

---
初始化配置:
---

2010年12月 (5)

2010年11月 (1)

personal

Google buzz for mobile

Mobile IM(ebuddy.com)

open source AnyDbTest on codeplex

Web IM(gtalk, skype)

博客同步

我的技术书签

最新评论

1. Re:系统研究Airbnb开源项目airflow

您好，当使用CeleryExecutor时，任务都处于Running状态，而使用LocalExecutor则一切运行正常，推测是Celery检测不到task运行结束，一般是什么原因，谢谢？这是一部分日.....

--ct5869

2. Re:系统研究Airbnb开源项目airflow

给力！

--爱吃猫的鱼

3. Re:系统研究Airbnb开源项目airflow

非常不错

--Gzhifei

4. Re:SqlAlchemy个人学习笔记完整汇总

太感谢了，这篇文章的引用非常好

--林羽飞扬

5. Re:edwin报警和监控平台开源了(python源码)

你好，有没有 类图或者设计图之类的东西，结构图。这样看起来更快

--豆花花水

1. 配置OS环境变量 AIRFLOW_HOME, AIRFLOW_HOME缺省为 ~/airflow
2. 运行下面命令初始化一个Sqlite backend DB, 并生成airflow.cfg文件
   your_python ${AIRFLOW_HOME}\bin\airflow initdb
3. 如果需要修改backend DB类型, 修改$AIRFLOW_HOME/airflow.cfg文件 sql_alchemy_conn后, 然后重新运行 airflow initdb .
   官方推荐使用MySQL/PostgreSQL做DB Server.
   MySQL 应该使用 mysqlclient 驱动, 我试验了mysql-connector-python 驱动, 结果airflow 网页端报错
   我试着用Oracle 做DB, 解决了很多问题, 但终究还是不能完全运行起来.
4. 修改$AIRFLOW_HOME/airflow.cfg文件
   重新设置Backend/Executor, 以及webserver端口, 设置dags_folder目录和base_log_folder目录. 有下面3个参数用于控制Task的并发度,
   parallelism, 一个Executor同时运行task实例的个数
   dag_concurrency, 一个dag中某个task同时运行的实例个数
   max_active_runs_per_dag: 一个dag同时启动的实例个数

---
了解几种作业运行模式
---
test 作业运行模式:
 该task是在本地运行, 不会发送到远端celery worker, 也不检查依赖状态, 也不将结果记录到airflow DB中, log也仅仅会在屏幕输出, 不记录到log文件.
 使用场景: 多用于测试单个作业的code的逻辑. 可以通过test 命令进入test 模式.

mark_success 作业运行模式:
 仅仅将作业在DB中Mark为success, 但并不真正执行作业
 使用场景: 多用于测试整个dag流程控制, 或者为某个task在DB中补一些状态. 可以在backfill命令和run命令中启用.

dry_run 作业运行模式:
 airflow不检查作业的上下游依赖, 也不会将运行结果记录不到airflow DB中. 具体作业的运行内容分情况:
 如果你的Operator没有重载 dry_run()方法的话, 运行作业也仅打印一点作业执行log
 如果重载BaseOperator的dry_run()方法的话, 运行作业即是执行你的dry_run()
 使用场景: 个人觉得 dry_run 模式意义并不大, 可以在backfill命令和 test 命令中启用

---
命令行工具
---
airflow 包安装后, 会往我们your_python\bin目录复制一个名为 airflow 的文件, 可以直接运行.

下面是该命令行工具支持的命令
1. 初始化airflow meta db
airflow initdb [-h]

2. 升级airflow meta db
airflow upgradedb [-h]

3. 开启web server
airflow webserver  --debug=False
开启airflow webserver, 但不进入flask的debug模式

4. 显示task清单
airflow list_tasks --tree=True -sd=/home/docs/airflow/dags
以Tree形式, 显示/home/docs/airflow/dags下的task 清单

5. 检查Task状态
airflow task_state  -sd=/home/docs/airflow/dags dag_id task_id execution_date
这里的 execution_date 是触发dag的准确时间, 是DB的datetime类型, 而不是Date类型

6. 开启一个dag调度器
airflow scheduler [-d DAG_ID] -sd=/home/docs/airflow/dags  [-n NUM_RUNS]
启动dag调度器, 注意启动调度器, 并不意味着dag会被马上触发, dag触发需要符合它自己的schedule

规则.
参数NUM_RUNS, 如果指定的话, dag将在运行NUM_RUNS次后退出. 没有指定时, scheduler将一直运行.
参数DAG_ID可以设定, 也可以缺省, 含义分别是:
如果设定了DAG_ID, 则为该DAG_ID专门启动一个scheduler;
如果缺省DAG_ID, airflow会为每个dag(subdag除外)都启动一个scheduler.

7. 立即触发一个dag, 可以为dag指定一个run id, 即dag的运行实例id.
airflow trigger_dag [-h] [-r RUN_ID] dag_id
立即触发运行一个dag, 如果该dag的scheduler没有运行的话, 将在scheduler启动后立即执行dag

8. 批量回溯触发一个dag
airflow backfill [-s START_DATE] [-e END_DATE] [-sd SUBDIR] --mark_success=False --dry_run=False dag_id
有时候我们需要**立即**批量补跑一批dag, 比如为demo准备点执行历史, 比如补跑错过的运行机会.
DB中dag execute_date记录不是当下时间, 而是按照 START_DATE 和 scheduler_interval 推算出的时间.
如果缺省了END_DATE参数, END_DATE等同于START_DATE.

9. 手工调用一个Task
airflow run [-sd SUBDIR] [-s TASK_START_DATE] --mark_success=False dag_id task_id execution_date
该命令参数很多, 如果仅仅是测试运行, 建议使用test命令代替.

10. 测试一个Task
airflow test -sd=/home/docs/airflow/dags --dry_run=False dag_id task_id execution_date
airflow test -sd=/home/docs/airflow/dags --dry_run=False dag_id task_id 2015-12-31
以 test 或 dry_run 模式 运行作业.

11. 清空dag下的Task运行实例
airflow clear [-s START_DATE] [-e END_DATE] [-sd SUBDIR] dag_id

12. 显示airflow的版本号
airflow version

==========
Airflow 开发
==========
---
dag脚本开发
---
dag脚本可参考example_dags目录中的sample, 然后将脚本存放到airflow.cfg指定的dags_folder下.
airflow 已经包含实现很多常用的 operator, 包括 BashOperator/EmailOperator/JdbcOperator/PythonOperator/ShortCircuitOperator/BranchPythonOperator/TriggerDagRunOperator等, 基本上够用了, 如果要实现自己的Operator, 继承BaseOperator, 一般只需要实现execute()方法即可. execute()约定没有返回值, 如果execute()中抛出了异常, airflow则会认为该task失败.
pre_execute()/post_execute()用处不大, 不用特别关注, 另外post_execute()是在on_failure_callback/on_success_callback回调函数之前执行的, 所以, 也不适合回写作业状态.

作业流程串接的几个小贴士:
使用 DummyOperator 来汇聚分支
使用 ShortCircuitOperator/BranchPythonOperator 做分支
使用 SubDagOperator 嵌入一个子dag
使用 TriggerDagRunOperator 直接trigger 另一个dag
T_B.set_upstream(T_A), T_A->T_B, 通过task对象设置它的上游
T_1.set_downstream(T_2), T_1->T_2 , 通过task对象设置它的下游
airflow.utils.chain(T_1, T_2, T_3), 通过task对象设置依赖关系, 这个方法就能一次设置长的执行流程, T_1->T_2->T_3
dag.set_dependency('T_1_id', 'T_2_id'), 通过id设置依赖关系

---
扩展airflow界面
---
扩展airflow，比如WebUI上增加一个菜单项，可以按照plugin形式实现.

---
自己的表如何关联airflow的表
---
很多时候airflow DB的各个表不够用，我们需要增加自己的表. 比如增加一个my_batch_instance表，一个my_task_instance表，my_batch_instance需要关联airflow dag_run表，my_task_instance需要关联airflow task_instance表.
my_batch_instance表中，增加airflow_dag_id和airflow_execute_date，来对应airflow dag_run表的逻辑PK；my_task_instance表增加airflow_dag_id和airflow_task_id和airflow_execute_date，对应airflow task_instance表的逻辑PK.

接下来的问题是，如何在task的python代码中，获取这些逻辑PK值？其实也很简答，我们的task都继承于BaseOperation类，BaseOperation.execute(self, context)方法，有一个context参数，它包含很丰富的信息，有:
dag定义对象，dag.dag_id 即是 dag_id 值
task定义对象,task.task_id 即是 task_id 值
execution_date相关的几个属性(包括datetime类型的execution_date，字符类型的ds，更短字符型的ds_nodash)
context是一个dict,完整的内容是

```
{
    'dag': task.dag,
    'ds': ds,
    'yesterday_ds': yesterday_ds,
    'tomorrow_ds': tomorrow_ds,
    'END_DATE': ds,
    'ds_nodash': ds_nodash,
    'end_date': ds,
    'dag_run': dag_run,
    'run_id': run_id,
    'execution_date': task_instance.execution_date,
    'latest_date': ds,
    'macros': macros,
    'params': params,
    'tables': tables,
    'task': task,
    'task_instance': task_instance,
    'ti': task_instance,
    'task_instance_key_str': task_instance_key_str,
    'conf': configuration,
}
```

execution_date相关的几个属性具体取值是:

```
ds=execution_date.isoformat()[:10]
ds_nodash = ds.replace('-', '')
ti_key_str_fmt = "{task.dag_id}__{task.task_id}__{ds_nodash}"
task_instance_key_str = ti_key_str_fmt.format(task,ds_nodash)
```

task_instance_key_str 值可以看做是Task instance表的单一的逻辑PK，很可惜的是Task instance没有这个字段.

---
如何及时拿到airflow task的状态
---
举例说明，比如我的task是执行一个bash shell，为了能将task的信息及时更新到自己的表中，需要基于BashOperator的实现一个子类MyBashOperator，在execute(context)方法中，将running状态记录到自己的表中.

另外，在创建MyBashOperator的实例时候，为on_failure_callback和on_success_callback参数设置两个回调函数，我们在回调函数中，将success或failed状态记录到自己的表中.
on_failure_callback/on_success_callback回调函数签名同execute()，都有一个context参数.

---
为生产环境和测试环境提供不同的设置
---
系统级的设置, 见airflow.cfg文档

DAG级别的设置, 我们可为Prod/Dev环境准备不同的default_args,
```
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2015, 6, 1),
    'email': ['airflow@airflow.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
    # 'queue': 'bash_queue',
    # 'pool': 'backfill',
    # 'priority_weight': 10,
    # 'end_date': datetime(2016, 1, 1),
}

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

通过Variable, 加载不同环境的配置. 详细思路如下:
比如我们有一个My_Cfg参数, 在Prod和Dev取值有可能不同.
首先设置一个 Environment_Flag variable, 其取值是Prod或Dev.
然后, 定义为My_Cfg参数设定两个变量, My_Cfg_For_Prod 和 My_Cfg_For_Dev, 并赋值, 分别对应Prod/Dev环境下My_Cfg的取值.
在代码中, 我们就可以通过Environment_Flag的取值, 就知道是该访问 My_Cfg_For_Prod 变量还是My_Cfg_For_Dev 变量, 进而得到My_Cfg的取值.

---
Regular的External trigger触发dag的推荐用法
---
外部触发需要trigger_dag命令行, 命令行最好要加上run_id参数;
同时DAG的schedule_interval参数最好设置成None, 表明这个DAG始终是由外部触发

---
测试运行步骤:
---
1. 先测试pytnon代码正确性
   python ~/airflow/dags/tutorial.py
2. 通过命令行验证DAG/task设置
   # print the list of active DAGs
   airflow list_dags

   # prints the list of tasks the "tutorial" dag_id
   airflow list_tasks tutorial

   # prints the hierarchy of tasks in the tutorial DAG
   airflow list_tasks tutorial --tree
3. 通过test命令行试跑一下, 测试一下code逻辑
   airflow test tutorial my_task_id 2015-06-01
4. 通过 backfill --mark_success=True
   airflow backfill tutorial -s 2015-06-01 -e 2015-06-07

---
我的开发
---
1. 贡献一个ssh hook.
2. 实现几个命令行, airflow_smic 命令行, 实现 terminate, pause, continue, failover 操作.
   failover 操作, 其实就是skip已经完成的作业, 重新跑running的作业.

3. 提供admin界面, 管理依赖关系, 并提供可视化预览
   通过 airflow list_tasks --tree=True, 实现可视化

标签: 工具, ETL



harrychinese
关注 - 10
粉丝 - 55
+加关注

5 0

« 上一篇：pip高级使用技巧以及搭建自己的pypi服务器
» 下一篇：快速搭建大数据分析环境

posted @ 2016-01-05 12:58 harrychinese 阅读(3437) 评论(3) 编辑 收藏

**评论列表**

#1楼 2016-04-13 15:17 Gzhifei

非常不错

支持(0) 反对(0)

#2楼 2016-05-19 20:20 爱吃猫的鱼

给力！

支持(0) 反对(0)

#3楼 2016-05-27 22:59 ct5869

您好，当使用CeleryExecutor时，任务都处于Running状态，而使用LocalExecutor则一切运行正常，推测是Celery检测不到task运行结束，一般是什么原因，谢谢？

这是一部分日志，请查看

[2016-05-27 22:15:08,490] {jobs.py:479} INFO - Checking dependencies on 48 tasks instances, minus 0 skippable ones
[2016-05-27 22:15:08,494] {base_executor.py:34} INFO - Adding to queue: airflow run ct3 print_date 2016-05-27T09:05:00 --local -sd DAGS_FOLDER/ct3.py

[2016-05-27 22:15:08,901] {jobs.py:643} INFO - Done queuing tasks, calling the executor's heartbeat
[2016-05-27 22:15:08,901] {jobs.py:646} INFO - Loop took: 1.675835 seconds
[2016-05-27 22:15:08,903] {models.py:234} INFO - Finding 'running' jobs without a recent heartbeat
[2016-05-27 22:15:08,904] {models.py:240} INFO - Failing jobs without heartbeat after 2016-05-27 22:12:53.904377
[2016-05-27 22:15:08,911] {celery_executor.py:62} INFO - [celery] queuing ('example_bash_operator', 'runme_1', datetime.datetime(2016, 5, 26, 10, 0)) through celery, queue=default
[2016-05-27 22:15:09,078] {celery_executor.py:62} INFO - [celery] queuing ('example_bash_ope

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

**注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。**

**最新IT新闻**:
· 2017 Web开发者学习路线图
· 58同城回应简历数据泄漏：已开展追查并加固信息安全
· 构建下一个计算平台！CB Insights挑选了110家VR/AR初创公司
· 你还在付押金？这份报告说信用时代已经到来
· 亚马逊在芝加哥的书店开了，这些有趣的地方值得一看
» 更多新闻...



**最新知识库文章**:
· 程序员学习能力提升三要素
· 为什么我要写自己的框架？
· 垃圾回收原来是这么回事
· 「代码家」的学习过程和学习经验分享
· 写给未来的程序媛
» 更多知识库文章...

**历史上的今天:**
2011-01-05 MyBatis.Net 资料整理