Airflow原理解析与常见坑答疑 – 基于v1.8.1

杭州优行科技-大数据部

汪涛

目录

CONTENTS

01

基础介绍

基础介绍 – 基础概念

DAG: Directed Acyclic Graph，is a collection of all the tasks you want to run, organized in a way that reflects their relationships and dependencies.

DAG Run: A DAG run is a physical instance of a DAG, containing task instances that run for a specific execution_date

execution_date: The execution_date is the *logical* date and time which the DAG Run, and its task instances, are running for

Operators: While DAGs describe *how* to run a workflow, Operators determine what actually gets done.

Sensor: waits for a certain time, file, database row.

tasks: Once an operator is instantiated, it is referred to as a "task". The instantiation defines specific values when calling the abstract operator, and the parameterized task becomes a node in a DAG

Task Instances: A task instance represents a specific run of a task

Workflows: By combining DAGs and Operators to create TaskInstances, you can build complex workflows

---

Hooks: Hooks are interfaces to external platforms and databases like Hive, S3, MySQL, Postgres, HDFS, and Pig.
Pools: Some systems can get overwhelmed when too many processes hit them at the same time, Airflow pools can be used to **limit the execution parallelism** on arbitrary sets of tasks
Connections: The connection information to external systems .
Queues: When using the CeleryExecutor, the Celery queues that tasks are sent to can be specified

基础介绍 - 基础组件

配置文件：airflow.cfg ($AIRFLOW_HOME目录下)
airflow webserver
airflow scheduler
airflow worker

基础介绍 - 基础组件

```
[admin@test31 ~]$ ps -ef | grep webserver | grep -v grep
admin     9404 19348 10 12:16 ?        00:00:16 [ready] gunicorn: worker [airflow-webserver]
admin    11026 19348 14 12:17 ?        00:00:16 [ready] gunicorn: worker [airflow-webserver]
admin    12867 19348 23 12:18 ?        00:00:16 [ready] gunicorn: worker [airflow-webserver]
admin    14641 19348 83 12:19 ?        00:00:16 [ready] gunicorn: worker [airflow-webserver]
admin    19100  1102  8 09:19 ?        00:14:35 /home/admin/program/airflow_venv/bin/python2 /home/admin/program/airflow_venv/bin/airflow webserver
admin    19348 19100  0 09:19 ?        00:00:06 gunicorn: master [airflow-webserver]
```

```
[admin@test31 ~]$ ps -ef | grep 'airflow sch' | grep -v grep
admin    17055 19073  0 12:20 ?        00:00:00 airflow scheduler /home/admin/program/airflow/dags/python/dag/cc_ads/t_red_bag_car.py
admin    17056 19073  0 12:20 ?        00:00:00 airflow scheduler /home/admin/program/airflow/dags/python/dag/openwork/jobchain/dep_day_86.py
admin    17057 19073  0 12:20 ?        00:00:00 airflow scheduler /home/admin/program/airflow/dags/python/dag/cc_ads/t_show_city_stat_1d_new.py
admin    17058 19073  0 12:20 ?        00:00:00 airflow scheduler /home/admin/program/airflow/dags/python/dag/bi/order/t_hermes_upms_weekly.py
admin    17059 19073  0 12:20 ?        00:00:00 airflow scheduler /home/admin/program/airflow/dags/python/dag/cc_ads/t_caocao_recharge_pay_withdraw_1d.py
admin    17060 19073  0 12:20 ?        00:00:00 airflow scheduler /home/admin/program/airflow/dags/python/dag/cc_ads/t_user_carbon_stat_insert.py
admin    17061 19073  0 12:20 ?        00:00:00 airflow scheduler /home/admin/program/airflow/dags/python/dag/cc_model/demand-fcst/t_real_damend_10_30.py
admin    17062 19073  0 12:20 ?        00:00:00 airflow scheduler /home/admin/program/airflow/dags/python/dag/cc_ods/cclog_recover_partitions.py
admin    19073  1102  6 12:05 ?        00:01:05 /home/admin/program/airflow_venv/bin/python2 /home/admin/program/airflow_venv/bin/airflow scheduler
```

基础介绍 - 基础组件

```
[admin@test31 ~]$ ps -ef | grep celery | grep -v grep
admin     5356  1102  0 02:53 ?        00:00:27 [celeryd: celery@test31:MainProcess] -active- (worker)
admin     5435  5356  0 02:53 ?        00:00:00 [celeryd: celery@test31:ForkPoolWorker-1]
admin     5436  5356  0 02:53 ?        00:00:00 [celeryd: celery@test31:ForkPoolWorker-2]
admin     5437  5356  0 02:53 ?        00:00:00 [celeryd: celery@test31:ForkPoolWorker-3]
admin     5438  5356  0 02:53 ?        00:00:00 [celeryd: celery@test31:ForkPoolWorker-4]
admin     5439  5356  0 02:53 ?        00:00:00 [celeryd: celery@test31:ForkPoolWorker-5]
admin     5440  5356  0 02:53 ?        00:00:00 [celeryd: celery@test31:ForkPoolWorker-6]
admin     5441  5356  0 02:53 ?        00:00:00 [celeryd: celery@test31:ForkPoolWorker-7]
admin     5442  5356  0 02:53 ?        00:00:00 [celeryd: celery@test31:ForkPoolWorker-8]
admin     5443  5356  0 02:53 ?        00:00:00 [celeryd: celery@test31:ForkPoolWorker-9]
admin     5444  5356  0 02:53 ?        00:00:00 [celeryd: celery@test31:ForkPoolWorker-10]
admin     5445  5356  0 02:53 ?        00:00:00 [celeryd: celery@test31:ForkPoolWorker-11]
admin     5446  5356  0 02:53 ?        00:00:00 [celeryd: celery@test31:ForkPoolWorker-12]
admin     5447  5356  0 02:53 ?        00:00:00 [celeryd: celery@test31:ForkPoolWorker-13]
admin     5448  5356  0 02:53 ?        00:00:00 [celeryd: celery@test31:ForkPoolWorker-14]
admin     5449  5356  0 02:53 ?        00:00:00 [celeryd: celery@test31:ForkPoolWorker-15]
admin     5450  5356  0 02:53 ?        00:00:00 [celeryd: celery@test31:ForkPoolWorker-16]
```

02

调度原理

调度原理 - 目录

调度原理 - 常见疑问

1、重新运行一个任务，clear之后还需要点击run吗

2、一个dag中有a、b任务，重跑过去时间的任务时，为什么clear a之后b会运行

3、任务正在运行，点击clear或者mark success之后正在运行的任务会终止吗

4、新上线dag为什么在页面上看不到

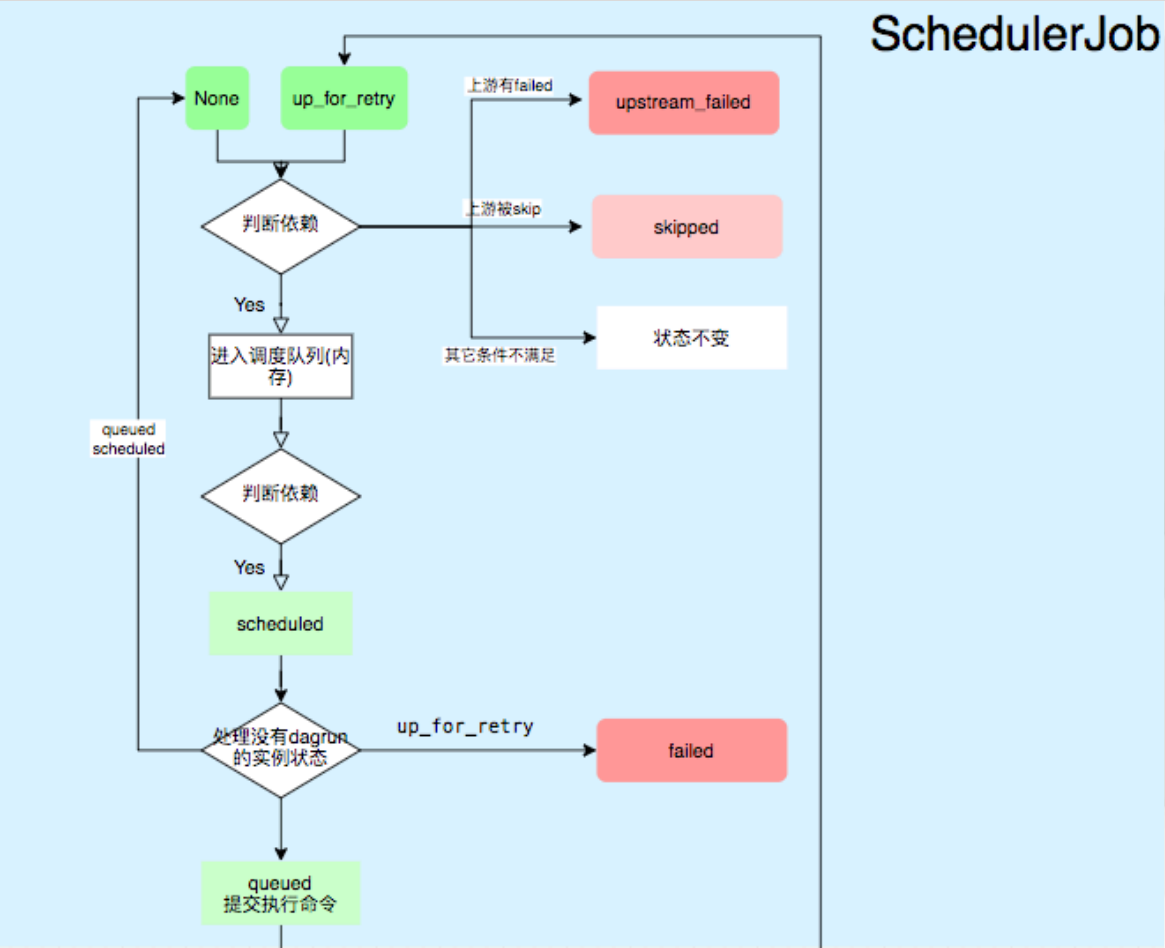
5、新上线一个dag为什么一直跑历史任务

6、为什么有的任务达到超时时间没有停止运行，有时会同时跑两个任务

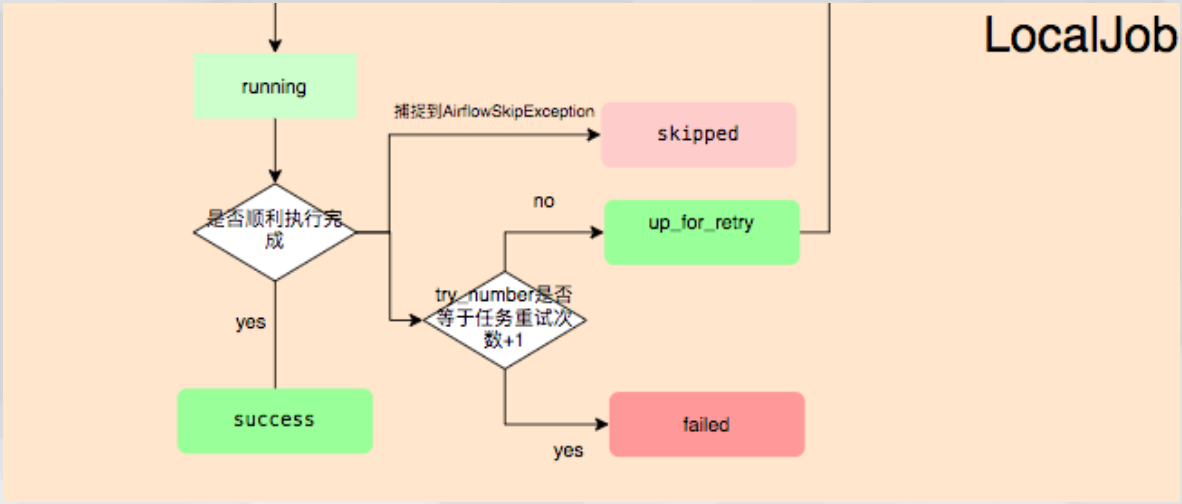
调度原理 - 状态流转


success
running
failed
skipped
rescheduled
retry
queued
no status

调度原理 - 状态流转


SchedulerJob
None
up_for_retry
上游有failed
upstream_failed
判断依赖
上游被skip
skipped
Yes
进入调度队列(内存)
其它条件不满足
状态不变
queued
scheduled
判断依赖
Yes
scheduled
处理没有dagrun的实例状态
up_for_retry
failed
queued
提交执行命令

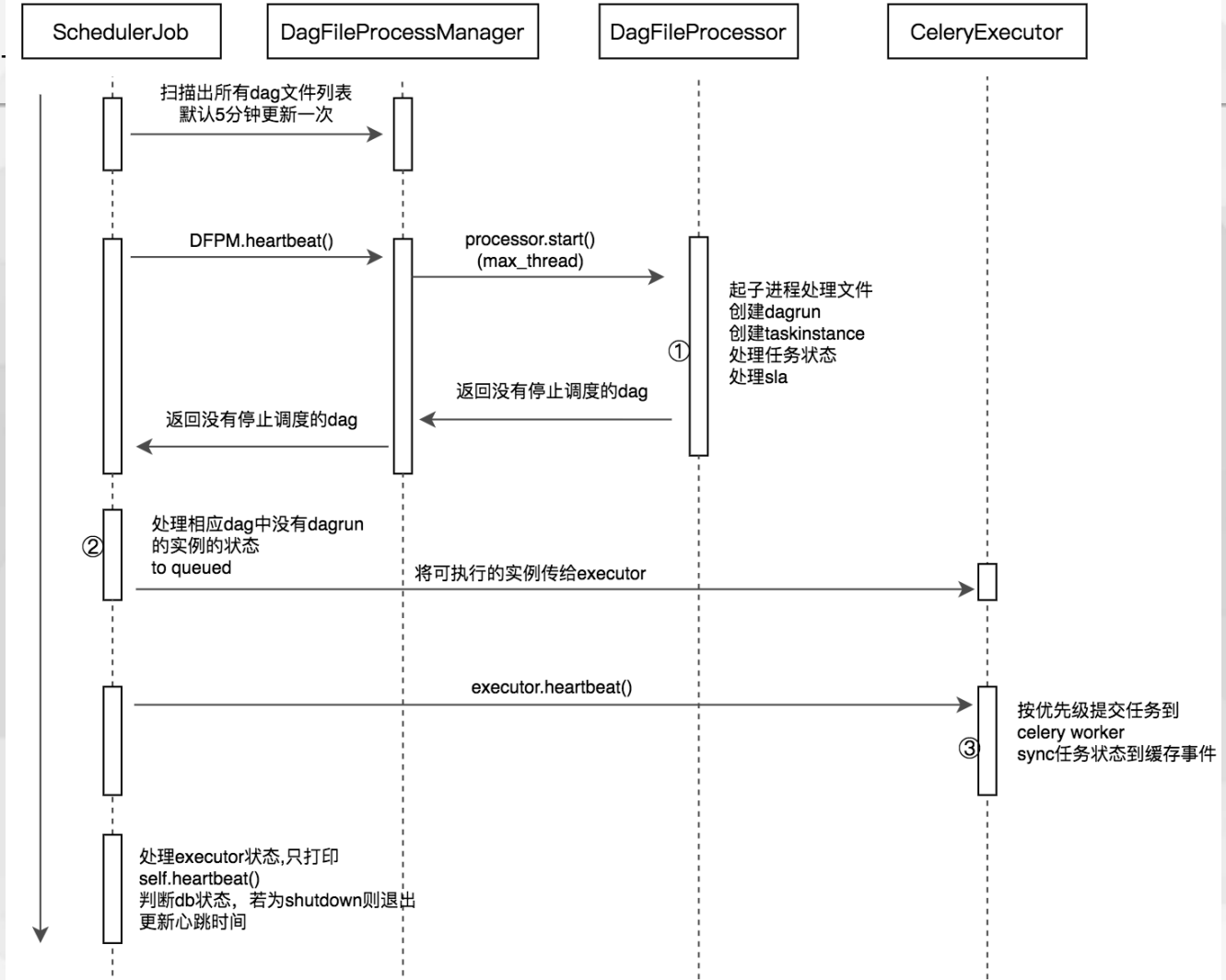
调度原理 - 状态流转


LocalJob
running
捕捉到AirflowSkipException
skipped
是否顺利执行完成
no
up_for_retry
try_number是否等于任务重试次数+1
yes
success
failed
yes

调度原理 - 调度核心类

SchedulerJob(BaseJob)
DagFileProcessManager
DagFileProcessor(AbstractDagFileProcessor)
CeleryExecutor(BaseExecutor)
DagRun
TaskInstance
LocalTaskJob(BaseJob)
BashTaskRunner(BaseTaskRunner)
...Operator(BaseOperator)
.....

调度原理


SchedulerJob
DagFileProcessManager
DagFileProcessor
CeleryExecutor
扫描出所有dag文件列表
默认5分钟更新一次
DFPM.heartbeat()
processor.start()
(max_thread)
起子进程处理文件
创建dagrun
创建taskinstance
处理任务状态
处理sla
①
返回没有停止调度的dag
返回没有停止调度的dag
②
处理相应dag中没有dagrun
的实例的状态
to queued
将可执行的实例传给executor
executor.heartbeat()
③
按优先级提交任务到
celery worker
sync任务状态到缓存事件
处理executor状态,只打印
self.heartbeat()
判断db状态，若为shutdown则退出
更新心跳时间

调度原理 – scheduler过程

```
[operadmin@test31 ~]$ ps -ef | grep 'airflow sch'
admin     3817  1102 34 15:54 ?        00:00:03 /home/admin/program/airflow_venv/bin/python2 /home/admin/program/airflow_venv/bin/airflow scheduler
admin     4002  3817 99 15:54 ?        00:00:01 airflow scheduler /home/admin/program/airflow/dags/python/dag/cc_dw/dwdnew.py
admin     4018  3817  0 15:54 ?        00:00:00 airflow scheduler /home/admin/program/airflow/dags/python/dag/app/tags_finish_moiter.py
admin     4019  3817  0 15:54 ?        00:00:00 airflow scheduler /home/admin/program/airflow/dags/python/dag/cc_ads/bi_new_gift_pack_stat_monthly.py
admin     4020  3817  0 15:54 ?        00:00:00 airflow scheduler /home/admin/program/airflow/dags/python/dag/cc_ads/t_zy_rush_order.py
admin     4021  3817  0 15:54 ?        00:00:00 airflow scheduler /home/admin/program/airflow/dags/python/dag/risk/operation_cclog_start_up.py
admin     4022  3817  0 15:54 ?        00:00:00 airflow scheduler /home/admin/program/airflow/dags/python/dag/cc_ads/t_dws_cnt_interactive_count.py
admin     4023  3817  0 15:54 ?        00:00:00 airflow scheduler /home/admin/program/airflow/dags/python/dag/cc_ads/zhongyue_ads.py
admin     4024  3817  0 15:54 ?        00:00:00 airflow scheduler /home/admin/program/airflow/dags/python/dag/openwork/jobchain/dep_day_66.py
```

①

增加dag run
1、将超时的dag run状态置为failed
2、根据最新的dagrun的execution_date以及是否catchup创建新的dag run
处理task instance
1、根据dag对所有在running中的dagrun增加和删除taskinstance，一次性补齐
2、根据dagrun所有taskinstance的状态更新dagrun的状态
3、将无状态和up_for_retry，判断依赖，如果上游有失败，状态置为upstream_failed,上游有skipped状态置为skipped，依赖完成的任务放入调度队列
处理sla
将调度队列中的taskinstance再判断一次依赖，依赖完成后将taskinstance状态置为scheduled
处理僵尸任务：将job状态为running但ti不为running以及job心跳超时的ti标记为重试状态或失败状态，并调用相应的回调函数
返回脚本包含的没有停止调度的dag对象

调度原理 – scheduler过程

create_dag_run

```
if not dag.catchup:
    ...
    now = datetime.now()
    next_start = dag.following_schedule(now)
    last_start = dag.previous_schedule(now)
    if next_start <= now:
        new_start = last_start
    else:
        new_start = dag.previous_schedule(last_start)

    if dag.start_date:
        if new_start >= dag.start_date:
            dag.start_date = new_start
    else:
        dag.start_date = new_start
```

```
if dag.schedule_interval == '@once':
    period_end = next_run_date
elif next_run_date:
    period_end = dag.following_schedule(next_run_date)
```

```
if next_run_date and period_end and period_end <= datetime.now():
    next_run = dag.create_dagrun(
        run_id='scheduled__' + next_run_date.isoformat(),
        execution_date=next_run_date,
        start_date=datetime.now(),
        state=State.RUNNING,
        external_trigger=False
    )
    return next_run
```

调度原理 – scheduler过程

_process_task_instances

```
active_dag_runs = []
for run in dag_runs:
    self.logger.info("Examining DAG run {}".format(run))
    # don't consider runs that are executed in the future
    if run.execution_date > datetime.now():
        self.logger.error("Execution date is in future: {}"
                          .format(run.execution_date))
        continue

    if len(active_dag_runs) >= dag.max_active_runs:
        self.logger.info("Active dag runs > max_active_run.")
        continue

    # skip backfill dagruns for now as long as they are not really sch
    if run.is_backfill:
        continue

    # todo: run.dag is transient but needs to be set
    run.dag = dag
    # todo: preferably the integrity check happens at dag collection t
    run.verify_integrity(session=session)
    run.update_state(session=session)
    if run.state == State.RUNNING:
        make_transient(run)
        active_dag_runs.append(run)
```

```
for run in active_dag_runs:
    self.logger.debug("Examining active DAG run {}".format(run))
    # this needs a fresh session sometimes tis get detached
    tis = run.get_task_instances(state=(State.NONE,
                                        State.UP_FOR_RETRY))

    ...
    for ti in tis:
        task = dag.get_task(ti.task_id)

        # fixme: ti.task is transient but needs to be set
        ti.task = task

        # future: remove adhoc
        if task.adhoc:
            continue

        if ti.are_dependencies_met(
                dep_context=DepContext(flag_upstream_failed=True),
                session=session):
            self.logger.debug('Queuing task: {}'.format(ti))
            queue.append(ti.key)

session.close()
```

调度原理 – scheduler过程

进入SCHEDULED状态

```
for ti_key in ti_keys_to_schedule:
    dag = dagbag.dags[ti_key[0]]
    task = dag.get_task(ti_key[1])
    ti = models.TaskInstance(task, ti_key[2])

    ti.refresh_from_db(session=session, lock_for_update=True)
    ...
    dep_context = DepContext(deps=QUEUE_DEPS, ignore_task_deps=True)

    ...
    if ti.are_dependencies_met(
            dep_context=dep_context,
            session=session,
            verbose=True):
        ...
        ti.state = State.SCHEDULED

    # Also save this task instance to the DB.
    self.logger.info("Creating / updating {} in ORM".format(ti))
    session.merge(ti)
    session.commit()
```

```
if flag_upstream_failed:
    if tr == TR.ALL_SUCCESS:
        if upstream_failed or failed:
            ti.set_state(State.UPSTREAM_FAILED, session)
        elif skipped:
            ti.set_state(State.SKIPPED, session)
    elif tr == TR.ALL_FAILED:
        if successes or skipped:
            ti.set_state(State.SKIPPED, session)
    elif tr == TR.ONE_SUCCESS:
        if upstream_done and not successes:
            ti.set_state(State.SKIPPED, session)
    elif tr == TR.ONE_FAILED:
        if upstream_done and not (failed or upstream_failed):
            ti.set_state(State.SKIPPED, session)
```

调度原理 – scheduler过程

②

1、将dagrun不是running的taskinstance，retry状态置为failed，queued与scheduled置为None
2、将scheduled状态的taskinstance根据优先级和对应pool的slot数量(没有指定pool的默认128)将实例状态置为queued
3、将任务实例、优先级、队列(worker)以及执行命令发送到executor的queue(scheduler)中

调度原理 – scheduler过程

```
if len(simple_dags) > 0:
    simple_dag_bag = SimpleDagBag(simple_dags)

    ...
    self._change_state_for_tis_without_dagrun(simple_dag_bag,
                                              [State.UP_FOR_RETRY],
                                              State.FAILED)
    ...
    self._change_state_for_tis_without_dagrun(simple_dag_bag,
                                              [State.QUEUED,
                                               State.SCHEDULED],
                                              State.NONE)

    self._execute_task_instances(simple_dag_bag,
                                 (State.SCHEDULED,))
```

```
current_task_concurrency = dag_id_to_possibly_running_task_count[dag_id]
task_concurrency_limit = simple_dag_bag.get_dag(dag_id).concurrency
self.logger.info("DAG {} has {}/{} running and queued tasks"
                 .format(dag_id,
                         current_task_concurrency,
                         task_concurrency_limit))
if current_task_concurrency >= task_concurrency_limit:
    self.logger.info("Not executing {} since the number "
                     "of tasks running or queued from DAG {}"
                     " is >= to the "
                     "DAG's task concurrency limit of {}"
                     .format(task_instance,
                             dag_id,
                             task_concurrency_limit))
    continue
```

调度原理 – scheduler过程

```
command = " ".join(TI.generate_command(
    task_instance.dag_id,
    task_instance.task_id,
    task_instance.execution_date,
    local=True,
    mark_success=False,
    ignore_all_deps=False,
    ignore_depends_on_past=False,
    ignore_task_deps=False,
    ignore_ti_state=False,
    pool=task_instance.pool,
    file_path=simple_dag_bag.get_dag(task_instance.dag_id).full_filepath,
    pickle_id=simple_dag_bag.get_dag(task_instance.dag_id).pickle_id))

priority = task_instance.priority_weight
queue = task_instance.queue
self.executor.queue_command(
    task_instance,
    command,
    priority=priority,
    queue=queue)
```

调度原理 – scheduler过程

③

1、根据parallelism限制按照优先级将状态不为running的实例异步执行(提交任务到celery worker)
2、sync任务状态，将任务状态缓存到执行器的对象中(event_buffer)

调度原理 – scheduler过程

```
def heartbeat(self):

    # Triggering new jobs
    if not self.parallelism:
        open_slots = len(self.queued_tasks)
    else:
        open_slots = self.parallelism - len(self.running)

    self.logger.debug("{} running task instances".format(len(self.running)))
    self.logger.debug("{} in queue".format(len(self.queued_tasks)))
    self.logger.debug("{} open slots".format(open_slots))

    sorted_queue = sorted(
        [(k, v) for k, v in self.queued_tasks.items()],
        key=lambda x: x[1][1],
        reverse=True)
    for j in range(min((open_slots, len(self.queued_tasks)))):
        key, (command, _, queue, ti) = sorted_queue.pop(0)
        ...
        self.queued_tasks.pop(key)
        ti.refresh_from_db()
        if ti.state != State.RUNNING:
            self.running[key] = command
            self.execute_async(key, command=command, queue=queue)
        else:
            self.logger.debug(
                'Task is already running, not sending to '
                'executor: {}'.format(key))

    # Calling child class sync method
    self.logger.debug("Calling the {} sync method".format(self.__class__))
    self.sync()
```

调度原理 – scheduler过程

补充

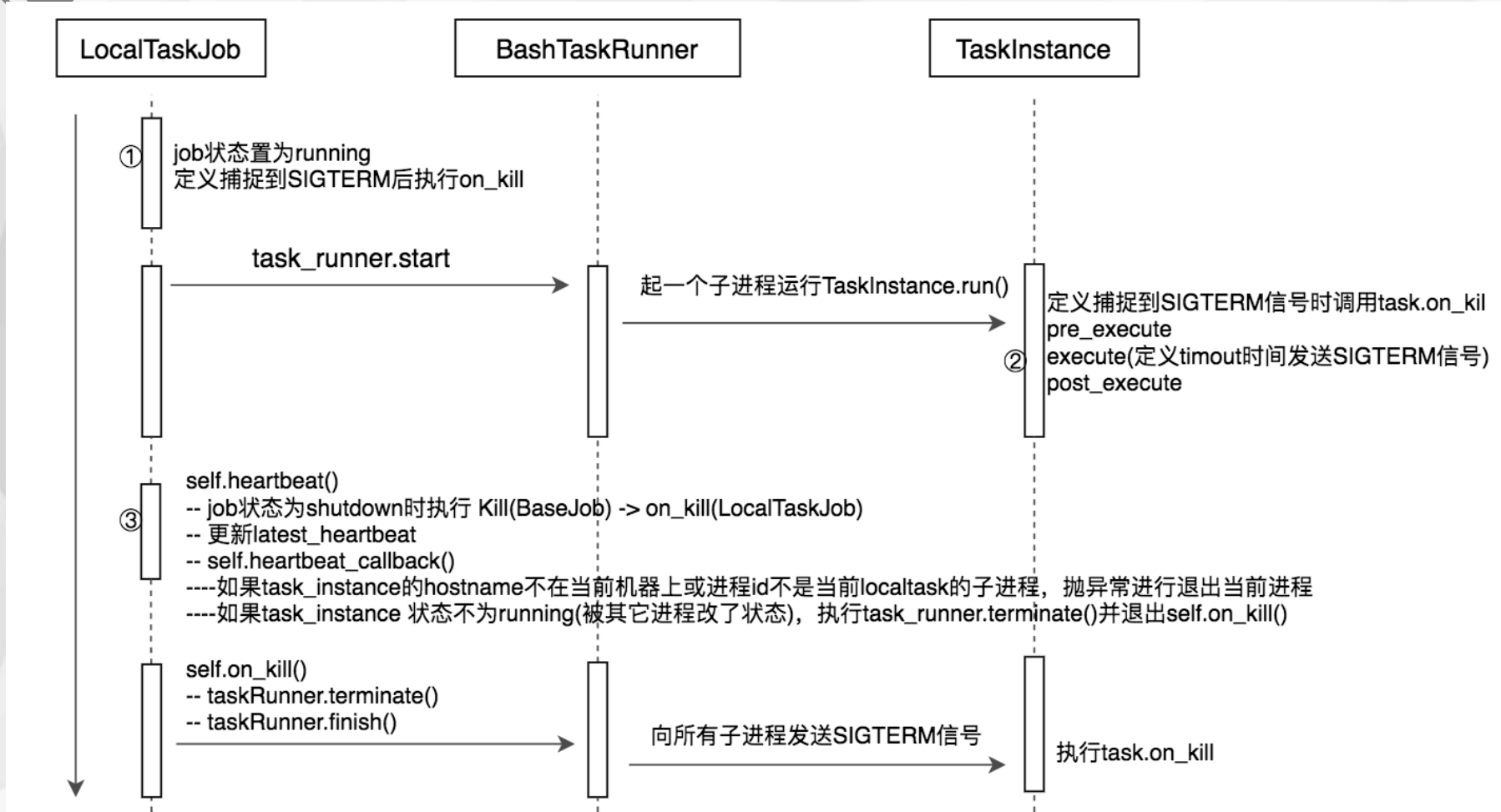
Start

生成SchedulerJob实例，置为running状态
扫描出dag文件列表
生成DagFileProcessorManager实例
初始化Executor(CeleryExecutor)

End

捕捉异常，kill掉所有子进程(先SIGTERM，再SIGKILL)

调度原理 - job执行过程



调度原理-job执行过程

```
[admin@bd-spark03 ~]$ ps -ef | grep -v grep | grep 2913
admin     2913  2708  0 09:11 ?        00:00:02 /home/admin/program/airflow_venv/bin/python2 /home/admin/program/airflow_venv/bin/airflow run openwork.jobchain.dep_day_57 ningbo_yonghu_data 2020-04-20T09:10:00 --job_id 3978212 --raw -sd DAGS_FOLDER/python/dag/openwork/job
chain/dep_day_57.py
admin     3049  2913  0 09:11 ?        00:00:00 sh -c spark-sql --queue default --executor-memory 6G --num-executors 5 --executor-cores 3 --driver-memory 2G --hiveconf hive.cli.print.header=true -f/home/admin/program/airflow/dags/python/dag/openwork/job/sql/ningbo_yonghu_
data.sql --hiveconf current_date=2020-04-21 > /data1/data/bi/order/ningbo_yonghu_data/ningbo_yonghu_data2020-04-20.csv
[admin@bd-spark03 ~]$ ps -ef | grep -v grep | grep 2708
admin     2708  9818  1 09:11 ?        00:00:18 /home/admin/program/airflow_venv/bin/python2 /home/admin/program/airflow_venv/bin/airflow run openwork.jobchain.dep_day_57 ningbo_yonghu_data 2020-04-20T09:10:00 --local -sd /home/admin/program/airflow/dags/python/dag/openwo
rk/jobchain/dep_day_57.py
admin     2913  2708  0 09:11 ?        00:00:02 /home/admin/program/airflow_venv/bin/python2 /home/admin/program/airflow_venv/bin/airflow run openwork.jobchain.dep_day_57 ningbo_yonghu_data 2020-04-20T09:10:00 --job_id 3978212 --raw -sd DAGS_FOLDER/python/dag/openwork/job
chain/dep_day_57.py
[admin@bd-spark03 ~]$ ps -ef | grep -v grep | grep 9818
admin     2708  9818  1 09:11 ?        00:00:18 /home/admin/program/airflow_venv/bin/python2 /home/admin/program/airflow_venv/bin/airflow run openwork.jobchain.dep_day_57 ningbo_yonghu_data 2020-04-20T09:10:00 --local -sd /home/admin/program/airflow/dags/python/dag/openwo
rk/jobchain/dep_day_57.py
admin     9818  9776  0 Mar20 ?        00:00:18 [celeryd: celery@bd-spark03:ForkPoolWorker-14]
root     29818     2  0 09:17 ?        00:00:00 [kworker/6:1]
[admin@bd-spark03 ~]$ ps -ef | grep -v grep | grep 9776
admin     8267  7972  0 05:36 ?        00:00:02 /home/admin/program/airflow_venv/bin/python2 /home/admin/program/airflow_venv/bin/airflow run bi.bi_cc_center_all t_caocao_city_operate_1d_part2 2020-04-20T04:27:00 --job_id 3977622 --raw -sd DAGS_FOLDER/python/dag/bi/bi_cc_
center_all.py
admin     9776  6434  0 Mar20 ?        04:05:24 [celeryd: celery@bd-spark03:MainProcess] -active- (worker)
admin     9791  9776  0 Mar20 ?        00:06:28 /home/admin/program/airflow_venv/bin/python2 /home/admin/program/airflow_venv/bin/airflow serve_logs
admin     9805  9776  0 Mar20 ?        00:00:19 [celeryd: celery@bd-spark03:ForkPoolWorker-1]
admin     9806  9776  0 Mar20 ?        00:00:18 [celeryd: celery@bd-spark03:ForkPoolWorker-2]
admin     9807  9776  0 Mar20 ?        00:00:18 [celeryd: celery@bd-spark03:ForkPoolWorker-3]
admin     9808  9776  0 Mar20 ?        00:00:19 [celeryd: celery@bd-spark03:ForkPoolWorker-4]
admin     9809  9776  0 Mar20 ?        00:00:19 [celeryd: celery@bd-spark03:ForkPoolWorker-5]
admin     9810  9776  0 Mar20 ?        00:00:18 [celeryd: celery@bd-spark03:ForkPoolWorker-6]
admin     9811  9776  0 Mar20 ?        00:00:19 [celeryd: celery@bd-spark03:ForkPoolWorker-7]
admin     9812  9776  0 Mar20 ?        00:00:16 [celeryd: celery@bd-spark03:ForkPoolWorker-8]
admin     9813  9776  0 Mar20 ?        00:00:19 [celeryd: celery@bd-spark03:ForkPoolWorker-9]
admin     9814  9776  0 Mar20 ?        00:00:18 [celeryd: celery@bd-spark03:ForkPoolWorker-10]
admin     9815  9776  0 Mar20 ?        00:00:18 [celeryd: celery@bd-spark03:ForkPoolWorker-11]
admin     9816  9776  0 Mar20 ?        00:00:18 [celeryd: celery@bd-spark03:ForkPoolWorker-12]
admin     9817  9776  0 Mar20 ?        00:00:18 [celeryd: celery@bd-spark03:ForkPoolWorker-13]
admin     9818  9776  0 Mar20 ?        00:00:18 [celeryd: celery@bd-spark03:ForkPoolWorker-14]
admin     9819  9776  0 Mar20 ?        00:00:18 [celeryd: celery@bd-spark03:ForkPoolWorker-15]
admin     9820  9776  0 Mar20 ?        00:00:18 [celeryd: celery@bd-spark03:ForkPoolWorker-16]
admin     9821  9776  0 Mar20 ?        00:00:19 [celeryd: celery@bd-spark03:ForkPoolWorker-17]
admin     9822  9776  0 Mar20 ?        00:00:18 [celeryd: celery@bd-spark03:ForkPoolWorker-18]
```

调度原理 -job执行过程

①

```
def on_kill(self):
    self.task_runner.terminate()
    self.task_runner.on_finish()
```

```
    def terminate(self):
        if self.process and psutil.pid_exists(self.process.pid):
            kill_process_tree(self.logger, self.process.pid)

    def on_finish(self):
        super(BashTaskRunner, self).on_finish()
```

调度原理-job执行过程

②

1、先判断依赖，依赖没通过，退出执行
检查的依赖项有: 实例要在以下状态中：scheduled, queued, failed, up_for_retry, upstream_failed，以及一些可执行条件(执行日期超过当前时间、task超出有效期等)要求
2、taskinstance状态置为running,try_number+1
3、初始化context
4、定义收到SIGTERM信号时调用task.on_kill
5、清理taskinstance的XCom数据
6、渲染模板变量
7、执行task的pre_execute
8、执行execute，若定义了timeout时间则发送一个定时器超时信SIGALRM，超时后执行task.on_kill再抛异常退出当前进程(先定义了收到SIGALRM信号时的抛AirflowTaskTimeout异常，再捕捉AirflowTaskTimeout异常再执行task.on_kill)
9、执行post_execute
10、第3到第9步没有任务异常则将taskinstance状态置为success
11、第3到第9步有异常处理：
捕捉到AirflowSkipException时则将实例状态置为skipped
捕捉到其它异常则先在task_fail中增加一条失败记录，如果实例的try_number次数不等于配置的任务重试次数+1的倍数则将状态置为up_for_retry，否则置为failed(如果配置了发送邮件，则会将异常信息发送至任务指定的email中)，并调用重试或失败回调函数

调度原理 -job执行过程

```
try:
    logging.info(msg.format(self=self))
    if not mark_success:
        context = self.get_template_context()

        task_copy = copy.copy(task)
        self.task = task_copy

        def signal_handler(signum, frame):
            '''Setting kill signal handler'''
            logging.error("Killing subprocess")
            task_copy.on_kill()
            raise AirflowException("Task received SIGTERM signal")
        signal.signal(signal.SIGTERM, signal_handler)

        # Don't clear Xcom until the task is certain to execute
        self.clear_xcom_data()

        self.render_templates()
        task_copy.pre_execute(context=context)

        # If a timeout is specified for the task, make it fail
        # if it goes beyond
        result = None
        if task_copy.execution_timeout:
            try:
                with timeout(int(
                        task_copy.execution_timeout.total_seconds())):
                    result = task_copy.execute(context=context)
            except AirflowTaskTimeout:
                task_copy.on_kill()
                raise
        else:
            result = task_copy.execute(context=context)

        # If the task returns a result, push an XCom containing it
        if result is not None:
            self.xcom_push(key=XCOM_RETURN_KEY, value=result)

        task_copy.post_execute(context=context)
        Stats.incr('operator_successes_{}'.format(
            self.task.__class__.__name__), 1, 1)
    self.state = State.SUCCESS
except AirflowSkipException:
    self.state = State.SKIPPED
except (Exception, KeyboardInterrupt) as e:
    self.handle_failure(e, test_mode, context)
    raise
```

```
try:
    if task.retries and self.try_number % (task.retries + 1) != 0:
        self.state = State.UP_FOR_RETRY
        logging.info('Marking task as UP_FOR_RETRY')
        if task.email_on_retry and task.email:
            self.email_alert(error, is_retry=True)
    else:
        self.state = State.FAILED
        if task.retries:
            logging.info('All retries failed; marking task as FAILED')
        else:
            logging.info('Marking task as FAILED.')
        if task.email_on_failure and task.email:
            self.email_alert(error, is_retry=False)
except Exception as e2:
    logging.error(
        'Failed to send email to: ' + str(task.email))
    logging.exception(e2)

# Handling callbacks pessimistically
try:
    if self.state == State.UP_FOR_RETRY and task.on_retry_callback:
        task.on_retry_callback(context)
    if self.state == State.FAILED and task.on_failure_callback:
        task.on_failure_callback(context)
except Exception as e3:
    logging.error("Failed at executing callback")
    logging.exception(e3)

if not test_mode:
    session.merge(self)
session.commit()
logging.error(str(error))
```

调度原理 – 跨DAG依赖

1、定义CrossDagDepPythonOperator(PythonOperator)

2、在pre_execute实现：
   1) 记录该实例到表task_instance_ext中
   2) 判断依赖的其它dag任务是否成功，没有成功，则抛出AirflowSkipException

3、在post_execute实现: 将task_instance_ext中实例状态置为success

4、起一个服务无限循环监控task_instance_ext表中状态，如果依赖的跨dag任务success，则clear该实例及其下游

调度原理 – 跨DAG依赖

```
is_task_ready = CrossDagDepPythonOperator.are_cross_dag_dep_conditions_met(
    super(CrossDagDepPythonOperator, self).dag_id,
    self.dag_dep_conditions, session, context['execution_date'])
if (not is_task_ready):
    ts_ext.state = STATE_WAITING
    session.merge(ts_ext)
    session.commit()
    raise AirflowSkipException("task dependencies are not ready yet!")
```

```
def post_execute(self, context):
    logging.info("post_execute in")
    session = settings.Session()
    execution_date = context['execution_date'].strftime('%Y-%m-%d %H:%M:%S')
    logging.info("execution_date:{}".format(execution_date))
    ts_ext = TaskInstanceExt(self.task_id, self.dag_id, execution_date)
    ts_ext.refresh_from_db(session)
    ts_ext.state = State.SUCCESS
    ts_ext.wait_time = (datetime.now() - ts_ext.start_time).total_seconds()
    session.merge(ts_ext)
    session.commit()
    session.close()
```

```
qry = session.query(TaskInstanceExt).filter(
    TaskInstanceExt.state == airflow_plugins.cross_dag_dep_python_operatior.STATE_WAITING,
    TaskInstanceExt.operator == CrossDagDepPythonOperator.__name__,
    TaskInstanceExt.execution_date > str_start_day,
    TaskInstanceExt.flag == 0,
    TaskInstanceExt.last_check_time < datetime.now() - timedelta(seconds=10),
)
```

03

常用操作含义

常用操作含义

clear

1、如果不在RUNNING中直接删除
2、在RUNNING中将job置为shutdown
3、将dagrun置为running

```
def clear_task_instances(tis, session, activate_dag_runs=True):
    """
    Clears a set of task instances, but makes sure the running ones
    get killed.
    """
    job_ids = []
    for ti in tis:
        if ti.state == State.RUNNING:
            if ti.job_id:
                ti.state = State.SHUTDOWN
                job_ids.append(ti.job_id)
        # todo: this creates an issue with the webui tests
        # elif ti.state != State.REMOVED:
        #     ti.state = State.NONE
        #     session.merge(ti)
        else:
            session.delete(ti)
    if job_ids:
        from airflow.jobs import BaseJob as BJ
        for job in session.query(BJ).filter(BJ.id.in_(job_ids)).all():
            job.state = State.SHUTDOWN
    if activate_dag_runs:
        execution_dates = {ti.execution_date for ti in tis}
        dag_ids = {ti.dag_id for ti in tis}
        drs = session.query(DagRun).filter(
            DagRun.dag_id.in_(dag_ids),
            DagRun.execution_date.in_(execution_dates),
        ).all()
        for dr in drs:
            dr.state = State.RUNNING
            dr.start_date = datetime.now()
```

常用操作含义

run

依赖通过后传给executor执行

```
# Make sure the task instance can be queued
dep_context = DepContext(
    deps=QUEUE_DEPS,
    ignore_all_deps=ignore_all_deps,
    ignore_task_deps=ignore_task_deps,
    ignore_ti_state=ignore_ti_state)
failed_deps = list(ti.get_failed_dep_statuses(dep_context=dep_context))
if failed_deps:
    failed_deps_str = ", ".join(
        ["{}: {}".format(dep.dep_name, dep.reason) for dep in failed_deps])
    flash("Could not queue task instance for execution, dependencies not met: "
          "{}".format(failed_deps_str),
          "error")
    return redirect(origin)

executor.start()
executor.queue_task_instance(
    ti,
    ignore_all_deps=ignore_all_deps,
    ignore_task_deps=ignore_task_deps,
    ignore_ti_state=ignore_ti_state)
executor.heartbeat()
flash(
    "Sent {} to the message queue, "
    "it should start any moment now.".format(ti))
return redirect(origin)
```

常用操作含义

Mark success

1、task_instance查漏补缺
2、将状态置为success

```
confirmed_dates = []
drs = DagRun.find(dag_id=dag.dag_id, execution_date=dates)
for dr in drs:
    dr.dag = dag
    dr.verify_integrity()
    confirmed_dates.append(dr.execution_date)
```

```
qry_dag = session.query(TI).filter(
    TI.dag_id==dag.dag_id,
    TI.execution_date.in_(confirmed_dates),
    TI.task_id.in_(task_ids)).filter(
    or_(TI.state.is_(None),
        TI.state != state)
)
```

```
if commit:
    tis_altered = qry_dag.with_for_update().all()
    if len(sub_dag_ids) > 0:
        tis_altered += qry_sub_dag.with_for_update().all()
    for ti in tis_altered:
        ti.state = state
    session.commit()
```

常用操作含义- 常见疑问

1、重新运行一个任务，clear之后还需要点击run吗

2、一个dag中有a、b任务，重跑过去时间的任务时，为什么clear a之后b会运行

3、任务正在运行，点击clear或者mark success之后正在运行的任务会终止吗

4、新上线dag为什么在页面上看不到

5、新上线一个dag为什么一直跑历史任务

6、为什么有的任务达到超时时间还在跑

常用操作含义- 常见疑问

```python
class PythonOperator(BaseOperator):
    """Executes a Python callable..."""
    template_fields = ('templates_dict',)
    template_ext = tuple()
    ui_color = '#ffefeb'

    @apply_defaults
    def __init__(
            self,
            python_callable,
            op_args=None,
            op_kwargs=None,
            provide_context=False,
            templates_dict=None,
            templates_exts=None,
            *args, **kwargs):...

    def execute(self, context):...
```

```python
class CCPythonOperator(PythonOperator):

    def on_kill(self):
        import psutil
        p = psutil.Process(os.getpid())
        childern = [x for x in
                    p.children(recursive=True)
                    if x.is_running()]
        for child in childern:
            if child.is_running():
                child.terminate()
        for child in childern:
            if child.is_running():
                child.kill()
```

from airflow.operators.python_operator import PythonOperator
to
from airflow_plugins.python_operator import CCPythonOperator as PythonOperator

04

新增插件介绍

常用操作含义- 新增插件介绍

```
class HiveToMysqlByDataxOperator(BaseOperator):

    def __init__(self):
        pass

    def execute(self, context):
        pass

    def on_kill(self):
        u"""如果会产生子进程则应实现此方法清理子进程"""
        pass
```

常用操作含义- 新增插件介绍

```python
class HiveToMysqlByDataxOperator(BaseOperator):

    ui_color = '#ededed'
    template_fields = (
        'hive_table',
        'partition_path',
        'mysql_table',
        'columns_map',
        'clear_where_condition',
        'write_mode'
    )

    job_text = """..."""


    def __init__(self,
                 hive_table,
                 partition_path,
                 mysql_table,
                 mysql_conn,
                 columns_map,
                 clear_where_condition=None,
                 write_mode='insert',
                 channel=1,
                 datax_conn_id='datax_default',
                 hive_metastore_conn_id='metastore_default',
                 hdfs_file_encoding='utf-8',
                 null_format=None,
                 compress=None,
                 jdbc_url_suffix="characterEncoding=utf-8",
                 datax_jvm_para=None,
                 datax_run_mode=None,
                 datax_loglevel=None,
                 datax_debug=None,
                 error_limit_record=0,
                 error_limit_percentage=0.0,
                 *args,
                 **kwargs):...

    def execute(self, context):...


    def on_kill(self):...
```

05

QA

Thanks!