Оркестрация загрузок (Apache Airflow)

Раздаточный материал к 6-му уроку, посвященному изучению оркестратору загрузок Airflow.

|  |  |
| --- | --- |
| СОДЕРЖАНИЕ | |
| 1 | Оркестраторы загрузок |
| 2 | Обзор Apache Airflow |
| 3 | Компоненты Apache Airflow |
| 4 | Apache Airflow DAG |
| 5 | Источники информации (ссылки) |

Оркестраторы загрузок

Оркестратор загрузок позволяет нам осуществлять мониторинг, управлять потоками данных в режиме реального времени с использованием графического интерфейса. В любом оркестраторе должны быть следующие возможности и особенности:

* автоматический перезапуск упавших задач
* легко читаемые связи внутри потока данных
* возможность мониторинга всех процессов
* удобный интерфейс для пользователя
* высокая доступность и масштабируемость
* возможность агрегировать логи с различных серверов

Выделим задачи, которые может помочь нам решать ELT-инструмент.

Основная функция — это **планирование задач**. Оркестратор позволит нам избавиться от ручного запуска рутинных задач по расчёту витрин, загрузке данных, резервному копированию и так далее.

Мы сможем **управлять зависимостями**. Часто задачу нужно запустить не только в определённый промежуток времени, но и с учётом статуса других задач. Например, расчёт витрины данных нужно запустить только после загрузки сырых данных на кластер.

Выполнение **репроцессинга**. Если известно, что какая-то задача требует перезапуска (например, были загружены неполные данные на предыдущем этапе), то перезапуска требуют и задачи, зависящие от неё. Кроме того, перезапуск может быть необходим за несколько временных периодов. В итоге нужно будет руками запустить несколько десятков задач, да ещё и в правильном порядке. Оркестратор позволяет выполнить эту утомительную работу за пару кликов.

Осуществление **мониторинга**. Есть множество причин, почему задача может не выполниться — опоздала загрузка данных, не хватило ресурсов, сервис, необходимый для расчётов, был временно недоступен. Поэтому одна из задач оркестратора — отображать наличие проблемы и уведомлять через специальные каналы людей, ответственных за поддержку. Большинство оркестраторов также поддерживают SLA-уведомления. Это уведомления, которые генерируются не в случае, когда задача «сломалась», а когда её выполнение задерживается, а значит, задерживается и загрузка критичных для бизнеса данных.

**Управление ресурсами** — это не основная задача оркестратора, но часто в их функционале можно встретить пулы и очереди задач, которые позволяют ограничить одновременное выполнение задач одного типа или использующих один ресурс. Пример: есть несколько задач на импорт из источника данных. Мощность кластера позволяет запустить сразу несколько задач импорта параллельно, но источник может не выдержать резкий рост нагрузки. В этом случае такие задачи объединяются в один пул и ограничивается число одновременных запусков, что позволяет выполнять задачи по очереди и распределить нагрузку на источник во времени.

Внедрение оркестратора позволит повысить качество данных за счёт возможностей быстрого репроцессинга и удобного мониторинга. А главное, он сделает работу дата-инженера приятнее, сняв с него кучу скучных рутинных задач.

Примеры оркестраторов загрузок: Apache Airflow, Apache NiFi, Luigi, Prefect, Dagster.

Обзор Apache Airflow

Рассмотрим подробнее продукт Apache Airflow. Это open-source инструмент с открытым исходным кодом, написанном на Python в 2014 году в компании Airbnb. Выделим основные преимущества и недостатки.

* как было сказано ранее, это **открытый исходный код**. AirFlow активно поддерживается сообществом и имеет хорошо описанную документацию.
* **небольшой, но полноценный инструментарий** создания процессов обработки данных и управления ими, расписание запусков для каждой цепочки задач, логирование сбоев.
* **графический веб-интерфейс** для создания конвейеров данных (data pipeline), который обеспечивает относительно низкий порог входа в технологию, позволяя работать с Airflow не только инженеру данных (Data Engineer), но и аналитику, разработчику, администратору и DevOps-инженеру. Пользователь может наглядно отслеживать жизненный цикл данных в цепочках связанных задач, представленных в виде направленного ациклического графа (Directed Acyclic Graph)
* **расширяемый REST API**, который позволяет относительно легко интегрировать Airflow в существующий ИТ-ландшафт корпоративной инфраструктуры и гибко настраивать конвейеры данных, например, передавать POST-параметры в DAG
* **программный код на Python.** Python считается относительно простым языком для освоения и общепризнанным стандартом для специалистов в области Big Data и Data Science. Когда ETL-процессы определены как код, они становятся более удобными для разработки, тестирования и сопровождения.
* **интеграция со множеством источников данных и сервисов**. AirFlow поддерживает множество баз данных (MySQL, PostgreSQL, DynamoDB, Hive), Big Data хранилищ (HDFS, Amazon S3) и облачных платформ (Google Cloud Platform, Amazon Web Services, Microsoft Azure).
* **наличие собственного репозитория** метаданных на базе библиотеки SqlAlchemy, где хранятся состояния задач, DAG’ов, глобальные переменные и пр.
* **мониторинг и алертинг**. Поддерживается интеграция с Statsd и FluentD — для сбора и отправки метрик и логов. Также доступен Airflow-exporter для интеграции с Prometheus.
* **возможность настройки ролевого доступа**. По умолчанию AirFlow предоставляет 5 ролей с различными уровнями доступа: Admin, Public, Viewer, Op, User. Также допускается создание собственных ролей с доступом к ограниченному числу DAG. Дополнительно возможна интеграция с Active Directory и гибкая настройка доступов с помощью RBAC (Role-Based Access Control).
* **поддержка тестирования**. Можно добавить базовые Unit-тесты, которые будут проверять как пайплайны в целом, так и конкретные задачи в них.
* **масштабируемость** за счет модульной архитектуры и очереди сообщений для неограниченного числа DAG’ов.
* **кастомизация**. Есть возможность настройки собственных операторов.

К недостаткам можно отнести следующее:

* **наличие неявных зависимостей при установке**, например, некоторые дополнительные пакеты могут усложнить быстрое конфигурирование фреймворка;
* некоторые пользователи отмечают незначительные **временные задержки** в запуске задач из-за нюансов работы планировщика, связанных с накладными расходами на постановку задач в очередь и их приоритезацию. Однако в версии Airflow 2 подобные задержки были сведены к минимуму, а также появилась возможность запуска нескольких планировщиков для достижения максимальной производительности;
* **необходимость наличия свободного слота в пуле задач** **и воркера**. Очередь сортируется в зависимости от приоритета задач (приоритетами тоже можно управлять), и если в пуле есть свободный слот, задача может браться в работу;
* **постфактум оповещения о сбоях в конвейере данных**, в частности, в интерфейсе Airflow логи появятся только после того, как задание, к примеру, Spark-job, отработано. Поэтому следить в режиме онлайн, как выполняется data pipeline, приходится из других мест, например веб-интерфейса YARN. Именно такое решение по работе с Apache Spark и Airflow было принято в онлайн-кинотеатре IVI;
* **разделение по операторам** — каждый оператор Airflow исполняется в своем python-интерпретаторе. Файл, который создается для определения DAG – это не просто скрипт, который обрабатывает какие-то данные, а объект. В процессе выполнения задачи DAG не могут пересекаться, так как они выполняются на разных объектах и в разное время. При этом на практике иногда возникает потребности, чтобы несколько операторов Airflow могли выполняться в одном Spark-контексте над общим пространством dataframe’ов. Реализовать это можно с помощью Apache Livy – REST-API сервиса для взаимодействия с кластером Spark.
* **нет возможности спроектировать DAG в графическом виде**, как это, например, доступно в Apache NiFi. Многие видят в этом, напротив, плюс, так как ревью кода проводится легче, чем ревью схем.
* **при проектировании задач важно соблюдать идемпотентность**: задачи должны быть написаны так, чтобы независимо от количества их запусков, для одних и тех же входных параметров возвращался одинаковый результат.
* **особенности в механизмах обработки execution\_date (даты выполнения DAG)**. Важно понимать, что корректировки кода задач будут отражаться на всех их запусках за предыдущее время. Это исключает воспроизводимость результатов, но, с другой стороны, позволяет получить результаты работы новых алгоритмов за прошлые периоды. Другими словами, если правится реализация задачки в DAG’е, то выполнение в предыдущих Execution Date пойдет уже с учетом корректировок. Это хорошо, если нужно пересчитать данные в прошлых периодах новым алгоритмом, но плохо, потому что теряется воспроизводимость результата (конечно, никто не мешает вернуть из Git’а нужную версию исходника и разово посчитать то, что нужно, так, как нужно).

Компоненты Apache Airflow

Рассмотрим архитектуру Airflow и принцип его работы.

Изображение выглядит как диаграмма

Автоматически созданное описание

Основу составляют следующие компоненты:

* **Web Server** — отвечает за пользовательский интерфейс, где предоставляется возможность настраивать DAG и их расписание, отслеживать статус их выполнения и так далее.
* **Scheduler (планировщик)** — планировщик, который запускает по расписанию в определённой последовательности задания внутри пайплайна. Отслеживая все созданные Task и DAG, планировщик инициализирует Task Instance — по мере выполнения необходимых для их запуска условий. По умолчанию раз в минуту планировщик анализирует результаты парсинга DAG и проверяет, нет ли задач, готовых к запуску. Для выполнения активных задач планировщик использует указанный в настройках Executor. Для определенных версий БД (PostgreSQL 9.6+ и MySQL 8+) поддерживается одновременная работа нескольких планировщиков — для максимальной отказоустойчивости.
* **Executor (исполнитель)** — механизм, с помощью которого запускаются экземпляры задач. Работает в связке с планировщиком в рамках одного процесса. Поддерживаемые типы исполнителей приведены ниже.

1. **SequentialExecutor** – последовательно запускает поступающие задачи и на время их выполнения приостанавливает планировщик, в связи с чем рекомендован исключительно для тестирования — для продуктивной среды он не подходит.
2. **LocalExecutor** – на каждую задачу запускает новый дочерний процесс, позволяя обрабатывать несколько задач параллельно. Отлично имитирует продуктивную среду в тестовой системе, но для реального использования не рекомендуется ввиду низкой отказоустойчивости: в случае сбоя на машине, где запущен Executor, задача не может быть передана другим узлам.
3. **CeleryExecutor** – основан на Celery (асинхронной очереди задач), позволяет иметь несколько Worker, работающих на разных машинах. Требует дополнительной настройки брокера сообщений, например, Redis либо RabbitMQ. Обладает высокой масштабируемостью и отказоустойчивостью: при увеличении нагрузки достаточно подключить дополнительный Worker, а в случае падения любого Worker его работа будет передана остальным узлам.
4. **DaskExecutor** – во многом похож на CeleryExecutor, но вместо Celery использует Dask – библиотеку для параллельных вычислений.
5. **KubernetesExecutor** – на каждый экземпляр задачи Task Instance запускает новый Worker на отдельном поде в Kubernetes.
6. **CeleryKubernetesExecutor** – Позволяет одновременно запускать CeleryExecutor и KubernetesExecutor. Конкретный тип будет выбираться в зависимости от состояния очереди задач.
7. **DebugExecutor** – создан для запуска и отладки пайплайнов из IDE.

* **Worker** — отдельный процесс, в котором выполняются задачи. Размещение Worker — локально или на отдельной машине — определяется выбранным типом Executor.
* **Metadata DB (база метаданных)** — собственный репозиторий метаданных на базе библиотеки SqlAlchemy для хранения глобальных переменных, настроек соединений с источниками данных, статусов выполнения Task Instance, DAG Run и так далее. Требует установки совместимой с SqlAlchemy базы данных, например, MySQL или PostgreSQL.
* **DAG Directory** - это место для хранения настроенных пользовательских DAG’ов.

Apache Airflow DAG

**DAG** — это описание пайплайна в Python коде, содержащее определение задач (Task) и их последовательность между собой. Представляет собой смысловое объединение задач. **Task** — это логическое представление одного из действий пайплайна (они же узлы DAG). Физически представлено оператором. Это непосредственно операции, применяемые к данным – ELT процессы. То есть DAG — это группа Задач, которые были скомпонованы для последующего выполнения направленным, ациклическим образом. Планировщик Airflow анализирует DAG, чтобы найти Задачи, готовые к выполнению, на основе их зависимостей. Если Задача готова к выполнению, Планировщик отправляет ее Исполнителю.

Выполнение Задачи в реальном времени называется task instance - **Экземпляром Задачи** (это также принято называть task run - **Выполнением Задачи**). Airflow регистрирует информацию об Экземплярах Задач, включая время их выполнения и статус, в базе метаданных (metadata database).

**Operator** — реализация Task. Если задачи описывают, какие действия выполнять с данными, то операторы — как эти действия выполнять. Операторы являются строительными блоками Airflow. Операторы содержат логику того, как данные обрабатываются в конвейере. Существуют разные Операторы для разных типов работы: некоторые Операторы выполняют общие типы кода, в то время как другие предназначены для выполнения очень специфических типов работ. Для того чтобы Оператор мог совершить работу в контексте DAG, он должен быть создан с помощью Задачи.

Примеры операторов приведены ниже:

* PythonOperator — исполнение Python-кода;
* BranchPythonOperator — запуск задач в зависимости от выполнения заданного условия;
* BashOperator — запуск Bash-скриптов;
* SimpleHttpOperator — отправка HTTP-запросов;
* MySqlOperator — отправка запросов к базе данных MySQL;
* PostgresOperator — отправка запросов к базе данных PostgreSQL;
* S3FileTransformOperator — загрузка данных из S3 во временную директорию в локальной файловой системе, преобразование согласно указанному сценарию и сохранение результатов обработки в S3;
* DockerOperator — запуск Docker-контейнера под выполнение задачи;
* KubernetesPodOperator — создание отдельного Pod под выполнение задачи. Используется совместно с K8s;
* SqlSensor — проверка выполнения SQL-запроса;
* SlackAPIOperator — отправка сообщений в Slack;
* EmailOperator — отправка электронных писем;
* DummyOperator — «пустой» оператор, который можно использовать для группировки задач;

**Sensor’ы** – операторы**,** которые позволяют прописывать реакцию на определенное событие. В качестве триггера может выступать наступление конкретного времени, получение некоторого файла или строки с данными, другого DAG/Task.

Возможные настройки для дага.

* owner – владелец DAGа;
* depends\_on\_past – запуск вне зависимости или в зависимости от статуса предыдущей задачи;
* email – список емейлов для рассылки алертов;
* email\_on\_failure – отправка сообщения на почту при падении задачи;
* email\_on\_retry – отправка сообщения на почту при повторении задачи;
* retries – количество повторений в случае падения;
* retries\_delay – ожидание между повторениями;
* queue – очередь для задач;
* pool – пул, в котором будут запущены задачи;
* priority\_weight – приоритет задачи в очереди задач;
* end\_date – дата окончания работы дага;
* wait\_for\_downstream – означает, что все предыдущие задачи должны или не должны быть выполнены успешно. Улучшает depends\_on\_past и проверяет, что все последующие задачи предыдущего запуска также были успешными.
* dag – ссылка на dag, к которому прикреплена задача
* sla – рассылка уведомления в случае, если даг всё еще не в статусе успешно выполненного по истечении указанного времени
* execution\_timeout – максимально отведённое время для выполнения таска
* on\_failure\_callback – функция, которая вызовется при падении.
* on\_success\_callback – функция, которая вызовется при успешном выполнении.
* on\_retry\_callback – функция, которая вызовется при повторении.
* sla\_miss\_callback – функция, которая вызовется в случае если таск не соответствует указанному sla.
* trigger\_rule– правило, которое будет являться триггером для запуска таска.

Airflow не предоставляет средств для автоматического обновления или версионирования пайплайнов. Каждая компания по-своему решают эту проблему. Чаще всего можно встретить следующие подходы:

1. Ручной. Py-файлы с дагом и сопутствующие зависимости заливаются вручную. Требуется осторожность, чтобы случайно не задеть другие даги. При уходе разработчика может нарушиться релизный цикл его пайплайна;
2. Git. Все даги компании хранятся в одном git репозитории, который обновляется в фоновом режиме.

Airflow каждые 300 секунд (настраивается в airflow.cfg) сканирует директорию dags и подхватывает любые изменения в ней.

В качестве дополнительной теории можно рассмотреть функции-декораторы на Python. Функции – это объекты. Соответственно, их можно возвращать из другой функции или передавать в качестве аргумента. Также функция в Python может быть объявлена внутри другой функции.

Декораторы – это «обёртки», которые дают нам возможность изменить процесс выполнения функции, добавив в неё дополнительную логику функции-декоратора, не изменяя код исходной функции. С помощью декораторов можно значительно упростить написание кода, уменьшить его объём, улучшить читаемость. Одну такую «обёртку» можно использовать сразу для нескольких функций.

Посмотреть примеры можно в презентации к уроку. Декорировать функции можно с помощью использования специального синтаксиса. Перед объявлением можно добавить знак @ и название функции-декоратора.

Изображение выглядит как текст

Автоматически созданное описание

Встроенные декораторы в Apache Airflow:

Изображение выглядит как стол

Автоматически созданное описание

Стоит отметить, что Apache Airflow предназначен для того, чтобы просто вызывать необходимые команды, в нашем случае запускать выполнение функций в Greenplum. Через сам ELT инструмент данные не перекачиваются.

Источники информации (ссылки)

1. [Зачем вам Dagster, если есть AirFlow: сравнение ETL-оркестраторов](https://www.bigdataschool.ru/blog/dagster-vs-airflow-dag-orchestration-in-big-data.html)
2. [Что такое AirFlow и как работает технология](https://mcs.mail.ru/blog/airflow-what-it-is-how-it-works)
3. [What is Airflow? — Airflow Documentation](https://airflow.apache.org/docs/apache-airflow/stable/index.html)
4. [Airflow.operators — Airflow Documentation](https://airflow.apache.org/docs/apache-airflow/1.10.3/_api/airflow/operators/index.html)
5. [4 альтернативы Apache Airflow для оркестровки Big Data Pipeline'ов](https://www.bigdataschool.ru/blog/airflow-alternatives.html#:~:text=%D0%9D%D0%B0%D0%B8%D0%B1%D0%BE%D0%BB%D0%B5%D0%B5%20%D0%B8%D0%B7%D0%B2%D0%B5%D1%81%D1%82%D0%BD%D0%BE%D0%B9%20%D0%B0%D0%BB%D1%8C%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%82%D0%B8%D0%B2%D0%BE%D0%B9%20Apache%20Airflow,%D0%B2%D0%B8%D0%B7%D1%83%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D0%B8%20%D0%B2%20%D0%B2%D0%B8%D0%B4%D0%B5%20DAG-%D0%BA%D0%BE%D0%BD%D0%B2%D0%B5%D0%B9%D0%B5%D1%80%D0%B0)
6. [Зачем дата-инженеру нужен оркестратор?](https://otus.ru/nest/post/722/)
7. [7 Best Airflow Alternatives for 2023](https://hevodata.com/learn/airflow-alternatives/#tools)
8. [4 альтернативы Apache Airflow для оркестровки Big Data Pipeline'ов](https://www.bigdataschool.ru/blog/airflow-alternatives.html#:~:text=%D0%9D%D0%B0%D0%B8%D0%B1%D0%BE%D0%BB%D0%B5%D0%B5%20%D0%B8%D0%B7%D0%B2%D0%B5%D1%81%D1%82%D0%BD%D0%BE%D0%B9%20%D0%B0%D0%BB%D1%8C%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%82%D0%B8%D0%B2%D0%BE%D0%B9%20Apache%20Airflow,%D0%B2%D0%B8%D0%B7%D1%83%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D0%B8%20%D0%B2%20%D0%B2%D0%B8%D0%B4%D0%B5%20DAG-%D0%BA%D0%BE%D0%BD%D0%B2%D0%B5%D0%B9%D0%B5%D1%80%D0%B0)
9. [7 достоинств и 5 недостатков Apache AirFlow](https://medium.com/@bigdataschool/7-%D0%B4%D0%BE%D1%81%D1%82%D0%BE%D0%B8%D0%BD%D1%81%D1%82%D0%B2-%D0%B8-5-%D0%BD%D0%B5%D0%B4%D0%BE%D1%81%D1%82%D0%B0%D1%82%D0%BA%D0%BE%D0%B2-apache-airflow-39fbbc80e702)
10. [Чем хорош Apache Airflow и что с ним не так: обзор плюсов и минусов](https://www.bigdataschool.ru/blog/airflow-features.html)
11. [Знакомство с Apache Airflow: установка и запуск первого DAGа](https://habr.com/ru/companies/alfa/articles/676926/)
12. [Как мы оркестрируем процессы обработки данных с помощью Apache Airflow](https://habr.com/ru/companies/lamoda/articles/518620/)
13. [DAG’и без напрягов: наш опыт использования метаданных при работе с Apache Airflow](https://habr.com/ru/companies/leroy_merlin/articles/564276/#habracut)
14. [Airflow — инструмент, чтобы удобно и быстро разрабатывать и поддерживать batch-процессы обработки данных](https://habr.com/ru/companies/vk/articles/339392/)
15. [Working with TaskFlow — Airflow Documentation](https://airflow.apache.org/docs/apache-airflow/stable/tutorial/taskflow.html)
16. [Декораторы | Python 3 для начинающих и чайников](https://pythonworld.ru/osnovy/dekoratory.html)