



Отдельные компоненты

| | |
|-----------------|---|
| ☰ Tags | |
| 📎 Files & media |  |

▼ DAG

 **DAG (Directed acyclic graph)** — это ориентированный ациклический граф. В Apache Airflow сложный конвейер преобразования данных и их последовательного запуска мы можем представить в виде графа. **Основная задача DAG** — организовать выполнение набора операторов.



Обрати внимание!

Чтобы создать DAG, необходимо три основных параметра: `id`, `start_date`, `schedule`. Они **обязательны**.

```
with DAG(
    dag_id="sample_dag",
    start_date=datetime(year=2024, month=1, day=1, hour=9,
    schedule_interval="@daily",
) as dag:
    ...
```

DAG мы описываем на языке **Python** в отдельном файле (и потом помещаем в папку). Каждый файл DAG обычно описывает набор задач для данного графа и **зависимости между задачами**, которые затем анализируются Airflow для определения структуры графа. Помимо этого, эти файлы обычно содержат некоторые дополнительные метаданные о графе, сообщаемые Airflow, как и когда он должен выполняться, и так далее.

▼ Пример кода Dag'ах

Импорты:

```
import json
import pathlib
import airflow
import requests
import requests.exceptions as requests_exceptions
from airflow import DAG
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator
```

Создаем и инициализируем DAG

```
dag = DAG( //dag - имя Dag'a
    dag_id="download_rocket_launches",
    start_date=airflow.utils.dates.days_ago(14), //когда его за
    schedule_interval=None, //интервал запуска
)
```

Пример добавления BashOperator'a

```
download_launches = BashOperator(
    task_id="download_launches",
    bash_command="curl -o /tmp/launches.json -L
'https://11.thespacedevs.com/2.0.0/launch/upcoming'",
    dag=dag, //не забыть указать название дага
)
```

Как показываем очередность в даге:

```
download_launches >> get_pictures >> notify
```

▼ Порядок выполнения задач

В Apache Airflow для настройки зависимостей между задачами в DAG можно использовать несколько методов, чтобы задать условия, последовательное и параллельное выполнение.

1. Последовательное выполнение задач

Чтобы настроить последовательное выполнение, можно использовать операторы `>>` и `<<` для указания порядка выполнения задач:

```
task1 >> task2 >> task3
```

В этом примере **task1** будет выполняться первой, затем **task2**, и в конце — **task3**. Airflow выполнит их строго по очереди, только после завершения предыдущей задачи.

Альтернативно, можно использовать методы `set_upstream` и `set_downstream` для указания зависимостей:

```
task2.set_upstream(task1) # task1 -> task2
task3.set_upstream(task2) # task2 -> task3
```

2. Параллельное выполнение задач

Для параллельного выполнения можно указать несколько задач на одном уровне без зависимостей между ними:

```
task1 >> [task2, task3]
```

Здесь **task2** и **task3** будут выполняться параллельно, как только завершится **task1**. Эти задачи начнутся одновременно и не зависят друг от друга. Это также можно записать так:

```
task2.set_upstream(task1)
task3.set_upstream(task1)
```

3. Условное выполнение задач

Для выполнения задач на основе условий можно использовать `BranchPythonOperator`, который позволяет выбрать, какую задачу запускать в зависимости от логики Python-функции. Пример:

```
from airflow.operators.python import BranchPythonOperator

def choose_branch():
    # Логика выбора задачи для выполнения
```

```

    if some_condition:
        return "task2" # Выполнить task2
    else:
        return "task3" # Выполнить task3

branch_task = BranchPythonOperator(
    task_id="branch_task",
    python_callable=choose_branch,
    dag=dag,
)

branch_task >> [task2, task3]

```

В этом примере `BranchPythonOperator` определяет, будет ли выполнена **task2** или **task3**, в зависимости от условий внутри функции `choose_branch`.

Комбинирование методов

Можно также комбинировать параллельное и последовательное выполнение для построения сложных графов задач:

```

start_task >> [task1, task2] # task1 и task2 выполняются п
араллельно
task1 >> task3 >> end_task # task3 выполняется после task1
task2 >> end_task # end_task выполняется после task2 и tas
k3

```

Таким образом, Apache Airflow предоставляет гибкие инструменты для настройки любых типов зависимостей между задачами в DAG.

▼ Запуск по расписанию

▼ Формат Cron



Формат cron — это широко используемый способ задания расписаний для автоматизированных задач. В Apache Airflow (и многих других системах) он используется для задания временных интервалов выполнения DAG'ов. Формат cron состоит из пяти или шести полей, которые задают конкретное время и даты для запуска задачи.

Каждое поле в cron-формате задаёт определённую часть времени: минуты, часы, дни и т.д. Пример формата cron:

```
* * * * * (или 5 полей)
<Минута> <Час> <День(Число)> <Месяц> <День недели>
```

Поля cron-формата

Значения в cron-формате задаются в таком порядке:

1. **Минуты** (от 0 до 59)
2. **Часы** (от 0 до 23)
3. **Дни месяца** (от 1 до 31)
4. **Месяцы** (от 1 до 12 или сокращения Jan, Feb, ..., Dec)
5. **Дни недели** (от 0 до 7, где 0 и 7 оба обозначают воскресенье, или сокращения Sun, Mon, ..., Sat)

Пример расписания

Каждое поле может принимать различные значения, задавая точное время выполнения. Вот примеры различных расписаний в cron-формате:

- `0 0 * * *` : каждый день в полночь.
- `30 6 * * *` : каждый день в 6:30 утра.
- `15 * * * *` : каждые 15 минут.
- `0 9 * * 1-5` : каждый будний день (с понедельника по пятницу) в 9:00 утра.
- `0 0 1 * *` : первый день каждого месяца в полночь.
- `0 */4 * * *` : каждые 4 часа.

- `0 12 15 5 *` : ежегодно 15 мая в 12:00.

Специальные символы и выражения

Cron-формат позволяет использовать несколько символов для более гибкого задания времени.

1. **Звездочка (*)** — любое значение.

- Например, `* * * * *` — запуск каждую минуту.

2. **Запятая (,)** — перечисление значений.

- Например, `0 9,17 * * *` — запуск каждый день в 9:00 и 17:00.

3. **Дефис (-)** — диапазон значений.

- Например, `0 9-17 * * *` — запуск каждый час с 9 до 17 часов.

4. **Слэш (/) для интервала** — кратность.

- Например, `/15 * * * * *` — запуск каждые 15 минут.

Специальные форматы

Некоторые системы, включая Airflow, поддерживают особые форматы cron, такие как:

- `@hourly` — раз в час.
- `@daily` — раз в день.
- `@weekly` — раз в неделю.
- `@monthly` — раз в месяц.
- `@yearly` или `@annually` — раз в год.

Примеры для Airflow

При создании DAG в Airflow, cron-формат часто используется в параметре

`schedule_interval`. Пример:

```
from airflow import DAG
from datetime import datetime

dag = DAG(
```

```

    "my_dag",
    schedule_interval="0 0 * * *", # запуск каждый день
    в полночь
    start_date=datetime(2023, 1, 1),
)

```

Cron-формат позволяет точно настроить расписание запуска DAG, что полезно для автоматизации регулярных задач.

▼ `schedule_interval`

Airflow запускает задачи в конце интервала. Если разработка ОАГ ведется 1 января 2019 года в 13:00, с `start_date` – 01-01-2019 и интервалом `@daily`, то это означает, что **сначала он запускается в полночь**. Поначалу ничего не произойдет, если вы запустите ОАГ 1 января в 13:00 до полуночи.

- Основной параметр для задания расписания DAG.
- Может принимать значения в формате cron, специальных пресетов или объекта `timedelta` для настройки интервала между запусками.
 - `"@daily"` — запуск каждый день в полночь.
 - `"@hourly"` — запуск каждый час.
 - `"@weekly"` — запуск раз в неделю в полночь по понедельникам.
 - `"@monthly"` — запуск раз в месяц в полночь первого числа.
 - `"@once"` — DAG запускается один раз при первом включении.
 - `timedelta(hours=6)` — запуск каждые 6 часов, `dt.timedelta(days=3)` — запуск каждые три дня.
 - `"0 9 * * *"` — запуск каждый день в 9:00 утра (формат cron).
 - `0 * * * *` = ежечасно (запуск по часам);
 - `0 0 * * *` = ежедневно (запуск в полночь);
 - `0 0 * * 0` = еженедельно (запуск в полночь в воскресенье).
 - `0 0 1 * *` = полночь первого числа каждого месяца;
 - `45 23 * * SAT` = 23:45 каждую субботу.



```
dag = DAG(  
    dag_id="download_rocket_launches",  
    start_date=airflow.utils.dates.days_ago(14),  
    schedule_interval="@daily",  
)
```

▼ start_date

- Устанавливает дату и время, с которых начинается расписание DAG.
- Airflow будет начинать выполнение DAG с этой даты, учитывая заданный `schedule_interval`.
- Важно: `start_date` не должна быть установлена на текущий момент или будущее время, так как Airflow начнёт выполнение DAG только после наступления `start_date`.

▼ `end_date`

- Опциональный параметр, который задаёт конечную дату для запуска DAG.
- После достижения `end_date` DAG перестаёт выполняться по расписанию.

▼ `catchup`

- Булевый параметр (`True` или `False`), указывающий, должен ли DAG "догонять" пропущенные запуски.
- По умолчанию `catchup=True`, то есть, если DAG не выполнялся за прошлые даты, Airflow создаст задачи для всех пропущенных запусков с момента `start_date`.
- Полезен для отключения догоняющего поведения (например, для ежедневных DAG с длинной историей), чтобы запускаться только для текущей даты.

▼ `timezone`

- Устанавливает часовой пояс для запуска DAG.
- По умолчанию используется часовой пояс UTC, но можно указать другие зоны, например, `"Europe/Moscow"`.
- Пример: `timezone="Europe/Moscow"`, чтобы DAG запускался по московскому времени.

Пример настройки DAG с расписанием:

```
from datetime import datetime, timedelta
from airflow import DAG

with DAG(
    dag_id="example_dag",
    start_date=datetime(2023, 1, 1),
```

```
end_date=datetime(2023, 12, 31),
schedule_interval="@daily",
catchup=False,
timezone="Europe/Moscow"
) as dag:
    ..
```

▼ Создание динамического DAG



Динамический DAG — это DAG, который создается на основе входных данных или параметров, что позволяет Airflow генерировать задачи и зависимости в DAG динамически. Такой подход полезен, когда количество задач, их параметры или зависимости неизвестны заранее и могут изменяться в зависимости от условий.

Пример создания динамического DAG с циклом

Допустим, вам нужно создать DAG для обработки данных из нескольких файлов. Количество файлов может меняться, и вы хотите создать задачу для каждого файла динамически:

```
python
Копировать код
from airflow import DAG
from airflow.operators.dummy import DummyOperator
from datetime import datetime

file_list = ["file1.csv", "file2.csv", "file3.csv"] # Список
ок файлов может изменяться

dag = DAG("dynamic_dag_example", start_date=datetime(2023,
1, 1))

start = DummyOperator(task_id="start", dag=dag)
end = DummyOperator(task_id="end", dag=dag)

# Динамически создаем задачи для каждого файла
```

```

for file_name in file_list:
    process_file = DummyOperator(
        task_id=f"process_{file_name}",
        dag=dag,
    )
    start >> process_file >> end

```

Здесь:

- В начале и в конце DAG есть задачи `start` и `end`.
- Для каждого файла в `file_list` создается задача `process_file`, причем имя задачи задается динамически (например, `process_file1.csv`, `process_file2.csv` и т.д.).

Применение динамических DAG

Динамические DAG'и полезны, когда:

- Задачи создаются на основе внешних данных (например, файлов в каталоге или записей в базе данных).
- Требуется гибкость для добавления или удаления задач без изменения основного кода.
- DAG содержит множество однотипных задач, которые можно сгенерировать автоматически для уменьшения кода и ошибок.

▼ Task/Operator/Sensor



Task — это конкретный экземпляр оператора в DAG. Это единичное выполнение операции, созданное на основе оператора и привязанное к расписанию и зависимостям в рамках DAG. Главное отличие от оператора то, что задача — это оболочка, менеджер вокруг оператора, которая шаблон реализует.

Пример: Если вы создаёте Task `task1` с использованием `PythonOperator`, это значит, что `task1` будет выполнять определённую Python-функцию в соответствии с заданным расписанием и зависимостями.

В Apache Airflow есть три основных типа Task:

1. **Операторы (Operators):** заранее определенные задачи, которые можно быстро объединять, чтобы построить большинство частей вашего DAG. Они охватывают различные действия, например, выполнение SQL-запроса, запуск скрипта Python, выполнение команд Bash и т.д.
2. **Сенсоры (Sensors):** это специальный подтип операторов, предназначенный для ожидания определенных внешних событий. Например, сенсоры могут ждать появления файла в заданной папке, завершения процесса или поступления данных.
3. **Задачи с декоратором TaskFlow (@task):** это кастомизированные Python-функции, которые можно обернуть в Task с помощью декоратора `@task` в рамках TaskFlow API. Это упрощает написание пользовательского кода и его интеграцию в DAG, делая каждый шаг задачи отдельной функцией Python.

▼ Operator

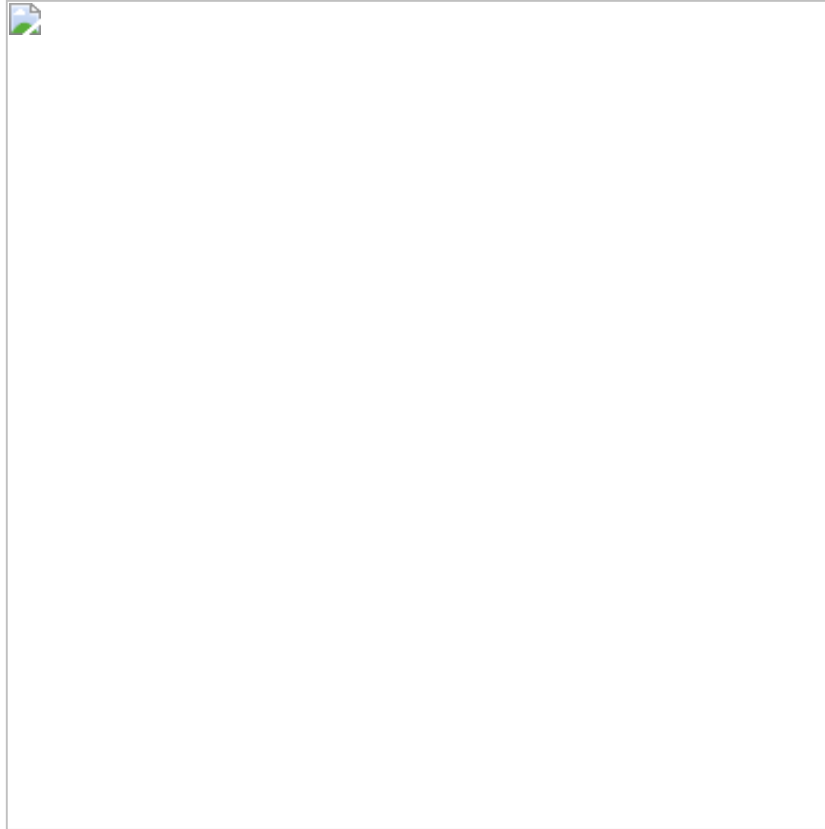


Operator — это шаблон или класс, который описывает тип действия, которое нужно выполнить. Он определяет, какой тип операции будет выполняться, но ещё не запускает саму задачу.


Типы операторов (основные):

- **PythonOperator:** выполняет Python-функцию.
- **BashOperator:** выполняет команду в Bash.
- **SQL оператор:** для выполнения SQL-запросов (например, PostgresOperator, MySqlOperator).
- **SensorOperator:** ожидает выполнения события, например, появления файла в S3.
- **BranchPythonOperator:** используется для реализации условной логики в DAG, когда выбор выполнения зависит от результатов предыдущей задачи.
- **SparkSubmitOperator** используется для запуска Spark приложений в Airflow. Этот оператор позволяет отправлять Spark-задания, указав параметры, такие как путь к Spark-приложению, конфигурации Spark, аргументы и т. д. Конфигурация:
 - `application` : путь к приложению (например, JAR или Python-файлу).
 - `conf` , `executor_memory` , `driver_memory` , `num_executors` : параметры для настройки Spark.

Примеры использования: Операторы — это "шаблоны" действий, такие как "выполнить SQL-запрос" или "выполнить Python-скрипт".



▼ Sensor

 **Sensor** — это специальный оператор, который "ожидает" наступления определённого события или выполнения условия, прежде чем продолжить выполнение DAG. Sensor полезен для проверки, готово ли условие для выполнения следующей задачи, например, для ожидания появления файла или завершения внешнего процесса.

Как работает Sensor

- Sensor переходит в состояние ожидания, проверяя определенное условие.
- Как только условие выполняется, Sensor завершает свою работу, и DAG переходит к следующей задаче.

- Если условие не выполнено, Sensor продолжает ожидание, периодически проверяя его выполнение (этот процесс называется **polling**).

Основные параметры Sensor

1. **poke_interval**: интервал (в секундах) между проверками условия.
2. **timeout**: максимальное время (в секундах), в течение которого Sensor будет пытаться проверить условие. По истечении времени Sensor завершится с ошибкой.
3. **mode**:
 - **"poke"** (режим по умолчанию): Sensor периодически проверяет условие через указанный **poke_interval**.
 - **"reschedule"**: Sensor освобождает worker и ждёт, пока снова наступит время для проверки условия. Это более ресурсосберегающий режим.

Примеры стандартных сенсоров в Airflow

Airflow предоставляет несколько предустановленных сенсоров, среди которых:

- **FileSensor**: ожидает появления файла в указанной директории.
- **ExternalTaskSensor**: ожидает завершения задачи или всего DAG в другом DAG'e.
- **HttpSensor**: ожидает ответа от HTTP-запроса, полезен для проверки доступности API.
- **S3KeySensor**: ожидает появления определенного объекта в бакете Amazon S3.
- **HdfsSensor**: ожидает появления файла или папки в HDFS.

Когда использовать Sensors

Sensors полезны в ситуациях, когда:

- Нужно дождаться, пока файл или данные появятся в системе.
- Нужно дождаться выполнения внешнего процесса (например, завершения другого DAG'a).
- Необходимо синхронизировать задачи, ожидая внешние события, такие как HTTP-ответы или события в хранилищах данных.

Проблемы и рекомендации

- **Режим `reschedule`** рекомендуется для уменьшения нагрузки на ресурсы, так как он освобождает worker, пока Sensor не готов выполнить следующую проверку.
- **Проблема длительного ожидания:** если sensor ожидает слишком долго, это может заблокировать worker, особенно в режиме `poke`.

Пример использования Sensor

Пример настройки `FileSensor`, который ожидает появления файла `/path/to/file.txt`:

```
from airflow import DAG
from airflow.sensors.filesystem import FileSensor
from datetime import datetime

dag = DAG("example_file_sensor", start_date=datetime(2023,
1, 1), schedule_interval="@daily")

wait_for_file = FileSensor(
    task_id="wait_for_file",
    filepath="/path/to/file.txt",
    poke_interval=30, # проверка каждые 30 секунд
    timeout=600,      # таймаут ожидания 10 минут
    mode="poke",      # режим ожидания poke
    dag=dag
)
```

▼ Приоритет

В Apache Airflow для управления приоритетом задач используется параметр `priority_weight`, который определяет порядок выполнения задач в DAG. Задачи с более высоким значением `priority_weight` будут выполняться раньше задач с более низким значением, если ресурсы позволяют.

Как работает `priority_weight`

- **`priority_weight`** задается для каждой задачи в DAG. Он принимает целое число, которое указывает относительный приоритет задачи по сравнению с другими задачами.
- Airflow использует `priority_weight` вместе с настройками параллелизма и `concurrency` для определения очереди задач.

- Более высокое значение `priority_weight` указывает на более высокий приоритет задачи. Если несколько задач одновременно готовы к выполнению, то Airflow выберет задачу с более высоким значением `priority_weight`.

Пример использования `priority_weight`

Пример DAG с тремя задачами, где каждой задаче присваивается свой `priority_weight`:

```
from airflow import DAG
from airflow.operators.dummy import DummyOperator
from datetime import datetime

dag = DAG("example_priority_dag", start_date=datetime(2023,
1, 1))

task1 = DummyOperator(
    task_id="task_low_priority",
    priority_weight=1, # Низкий приоритет
    dag=dag
)

task2 = DummyOperator(
    task_id="task_medium_priority",
    priority_weight=5, # Средний приоритет
    dag=dag
)

task3 = DummyOperator(
    task_id="task_high_priority",
    priority_weight=10, # Высокий приоритет
    dag=dag
)

task1 >> task2 >> task3
```

В этом примере **task3** имеет наибольший приоритет (10), **task2** — средний (5), и **task1** — низкий (1). Если все задачи будут готовы к выполнению одновременно, Airflow сначала выберет **task3**, затем **task2**, а потом **task1**.

▼ Механизм Retry

Механизм повторных попыток (Retry) позволяет Airflow автоматически пытаться выполнить задачу заново, если она завершилась ошибкой. Это полезно для задач, которые могут временно выходить из строя из-за внешних факторов, таких как временные сбои соединения, проблемы с API или временная нехватка ресурсов.

Параметры Retry

Каждая задача в Airflow может иметь два важных параметра для настройки механизма повторных попыток:

- `retries`: определяет количество повторных попыток выполнения задачи при её неудачном завершении.
- `retry_delay`: задает интервал времени между попытками выполнения задачи. Это может быть `timedelta` или выражение типа `"10m"` для 10 минут.

Пример использования Retry

Рассмотрим задачу, которая выполняет подключение к API. В случае временной ошибки она будет пытаться снова подключиться через указанный промежуток времени:

```
python
Копировать код
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime, timedelta

def connect_to_api():
    # Логика подключения к API
    # (код здесь может выбросить ошибку, если API временно
    # недоступен)
    pass

dag = DAG("retry_example_dag", start_date=datetime(2023, 1,
1))

api_task = PythonOperator(
    task_id="connect_to_api",
```

```
python_callable=connect_to_api,  
retries=3, # Повторить попытку 3 раза при ошибке  
retry_delay=timedelta(minutes=5), # Задержка между по-  
пытками 5 минут  
dag=dag  
)
```

В этом примере:

- Если задача `api_task` завершится ошибкой, Airflow подождет 5 минут (заданный `retry_delay`) и снова попытается её выполнить.
- Если три попытки (заданные `retries`) завершатся неудачно, задача перейдёт в статус «failed» и больше не будет выполняться в рамках текущего запуска DAG.

▼ Executor



Executor в Apache Airflow — это компонент, отвечающий за управление выполнением задач (tasks) в рамках DAG'ов. Executor распределяет задачи между воркерами (workers), координирует их выполнение и контролирует параллелизм. Он определяет, где и как будут выполняться задачи DAG, и является важным элементом архитектуры Airflow.

Расположение

Executor в Apache Airflow зависит от **выбранного типа и архитектуры развертывания**

SequentialExecutor

- **Описание:** Выполняет задачи последовательно, одна за другой.
- **Расположение:** работает на той же машине, где установлен Airflow.
- **Преимущества:** Легковесный и простой, так как работает без параллелизма.
- **Когда использовать:** Подходит для тестирования или разработки, особенно если требуется запустить Airflow на одной машине с минимальными ресурсами и без сложной установки.
- **Недостаток:** Не поддерживает параллельное выполнение задач, что делает его непрактичным для production.

LocalExecutor

- **Описание:** Выполняет задачи параллельно на одной машине, используя потоки или процессы.
- **Расположение:** работают на той же машине, где установлен Airflow.
- **Преимущества:** Поддерживает параллелизм, более производителен, чем SequentialExecutor, так как использует ресурсы одного узла для одновременного выполнения задач.
- **Когда использовать:** Хороший выбор для небольших и средних проектов, где задачи могут быть выполнены на одной машине без распределённой архитектуры. Подходит для разработки и небольших production-систем.
- **Недостаток:** Ограничен мощностью одной машины, поэтому не подходит для высоконагруженных систем.

CeleryExecutor

- **Описание:** Работает с распределённой архитектурой, используя Celery для управления задачами, которые обрабатываются на нескольких воркерах.
- **Расположение:** CeleryExecutor требует нескольких компонентов, которые могут быть расположены на разных машинах.
 - **Scheduler и Web-сервер:** часто находятся на одной машине (или нескольких, если настроена отказоустойчивость).
 - **Брокер сообщений** (например, Redis или RabbitMQ): может быть на отдельной машине.
 - **Workers:** могут работать на нескольких машинах, что позволяет распределять задачи.
- **Преимущества:** Масштабируем и может обрабатывать множество задач параллельно, распределяя их по нескольким серверам. Использует брокер сообщений (например, Redis или RabbitMQ) для координации между воркерами.
- **Когда использовать:** Подходит для production-систем с высокой нагрузкой, требующих параллельной и распределённой обработки задач. Особенно эффективен в средах, где есть несколько серверов или виртуальных машин для воркеров.
- **Недостаток:** Требуется настройки и управления брокером сообщений и воркерами, что усложняет установку и обслуживание.

KubernetesExecutor

- **Описание:** Запускает каждую задачу в отдельном поде Kubernetes, что позволяет динамически масштабировать задачи и ресурсы.
- **Расположение:** работает в Kubernetes-кластере, где каждый task запускается в отдельном поде.
- **Преимущества:** Высокая гибкость и масштабируемость, так как Kubernetes автоматически выделяет ресурсы на основе требований задачи. Поддерживает изоляцию задач и позволяет точно настроить ресурсы (например, память, CPU) для каждой задачи.
- **Когда использовать:** Рекомендуется для облачных решений и гибридных архитектур, а также для production-систем с высокой нагрузкой, где требуется гибкость и точный контроль над ресурсами.
- **Недостаток:** Сложность в установке и настройке, требует использования Kubernetes-кластера.

Сравнение CeleryExecutor и LocalExecutor

- **Архитектура:**
 - **LocalExecutor:** Выполняет задачи локально на одной машине, используя потоки или процессы.
 - **CeleryExecutor:** Распределённая архитектура с поддержкой нескольких воркеров, которые могут выполняться на разных серверах.
- **Параллелизм и масштабируемость:**
 - **LocalExecutor:** Параллелизм ограничен ресурсами одного узла (CPU и память).
 - **CeleryExecutor:** Масштабируем за счёт распределённых воркеров, можно добавить больше машин для увеличения параллельности.
- **Использование ресурсов:**
 - **LocalExecutor:** Легче установить и поддерживать, но требует мощного узла для обработки больших объёмов данных.
 - **CeleryExecutor:** Требуется настройки брокера сообщений (например, Redis, RabbitMQ) и может потребовать больше администрирования.

Когда использовать каждый из экзекьюторов:

- **SequentialExecutor**: Для разработки и тестирования без параллелизма.
- **LocalExecutor**: Для небольших или средних проектов, когда всё выполняется на одной машине и требуется ограниченный параллелизм.
- **CeleryExecutor**: Для высоконагруженных production-систем, требующих распределённого выполнения задач на нескольких воркерах.
- **KubernetesExecutor**: Для облачных сред, где необходимы гибкость, высокая масштабируемость и изоляция задач. Подходит для проектов с высокими требованиями к производительности и для сред с динамически меняющимися нагрузками.

▼ Variables



Variables в Apache Airflow — это механизм хранения и управления значениями, которые могут быть использованы в DAG и задачах для настройки или передачи информации. Переменные полезны для хранения значений, которые часто меняются, но должны оставаться доступными в разных задачах или DAG'ах, таких как пути к файлам, параметры подключения, секретные ключи и другие параметры конфигурации.

Глобальная доступность: Переменные создаются на уровне всей системы Airflow и доступны для всех DAG'ов и задач, что позволяет переиспользовать значения в нескольких рабочих процессах.

Хранение и управление через UI, CLI и API: Переменные можно управлять через **веб-интерфейс**, **CLI** и **REST API**:

- **Веб-интерфейс:** Раздел **Admin > Variables**, где можно создать, изменить или удалить переменные.
- **CLI:** Например, команда `airflow variables set key value` для создания переменной.
- **API:** Через вызов API для автоматизированного управления переменными.

Использование в DAG и задачах: Переменные можно использовать прямо в коде DAG и задач с помощью `Variable.get("key")`.

```
from airflow.models import Variable
```

```
# Получение значения переменной
my_value = Variable.get("my_variable_key")
```

Шифрование конфиденциальных данных. Переменные, содержащие конфиденциальные данные (например, пароли или API-ключи), могут быть зашифрованы с помощью ключей шифрования в Airflow. Это обеспечивает дополнительную безопасность для хранения чувствительной информации.

Переменные по умолчанию: При вызове `Variable.get()` можно задать значение по умолчанию, которое будет возвращено, если переменная не найдена.

```
my_value = Variable.get("non_existing_key", default_var="default_value")
```

Поддержка JSON Переменные могут хранить значения в формате JSON, что удобно для хранения сложных структур данных, таких как списки или словари.

```
my_json_data = Variable.get("my_json_key", deserialize_json=True)
```

Примеры использования Variables в Airflow

- **Хранение путей к файлам и параметров конфигурации:** Например, переменные можно использовать для хранения путей к данным, чтобы менять их по мере необходимости, не изменяя код DAG'a.
- **Хранение ключей и токенов:** Переменные удобны для хранения API-ключей и токенов, необходимых для подключения к внешним системам.
- **Параметры для динамических DAG'ов:** Переменные могут быть полезны для параметризации DAG'ов. Например, можно использовать переменную для хранения значения даты или диапазона, которые будут использоваться для фильтрации данных при запуске DAG'a.

▼ XCom



XCom (Cross-communication) в Apache Airflow — это механизм для обмена данными между задачами внутри одного DAG. XCom позволяет передавать небольшие объёмы данных из одной задачи в другую, что полезно для динамического использования результатов, полученных на предыдущих этапах. *Можно им пользоваться, но лимит до 2Гб. Желательно сгружать данные в БД и не передавать их по XCOM*

XCom лучше не использовать для передачи больших объёмов данных, так как они хранятся в базе метаданных Airflow и могут перегружать её.

Основные особенности XCom

1. Передача данных между задачами:

- XCom позволяет одной задаче записывать данные, которые другая задача может затем извлечь и использовать.
- Например, задача А может получить данные из API и передать их задаче В для дальнейшей обработки.

2. Механизм «ключ-значение»:

- XCom использует пары ключ-значение для хранения данных. Когда задача записывает данные в XCom, она может указать ключ, чтобы другая задача могла найти и извлечь это значение.

3. Поддержка произвольных данных:

- XCom поддерживает передачу различных типов данных, включая строки, числа, списки и даже более сложные объекты Python. Однако рекомендуется использовать XCom только для передачи небольших объёмов данных, поскольку все данные хранятся в базе данных метаданных Airflow, и большие объёмы могут перегружать её.

4. Методы работы с XCom:

- **push**: Метод, который передаёт данные в XCom. Обычно вызывается в `return` функции оператора, если передача значений задана в `PythonOperator`.
- **pull**: Метод, который извлекает данные из XCom. Вызывается в другой задаче, которая получает результат от предыдущей.

Пример использования XCom для передачи данных

Допустим, у нас есть DAG, где одна задача извлекает данные из API, а другая задача должна использовать их для обработки.

```
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime

def fetch_data(**kwargs):
    data = {"key": "value"}
    # Сохраняем данные в XCom
    kwargs['ti'].xcom_push(key='api_data', value=data)

def process_data(**kwargs):
    # Извлекаем данные из XCom
    data = kwargs['ti'].xcom_pull(key='api_data')
    print(f"Processing data: {data}")

with DAG(dag_id="xcom_example_dag", start_date=datetime(20
23, 1, 1), schedule_interval="@daily") as dag:
    fetch_task = PythonOperator(
        task_id="fetch_data",
        python_callable=fetch_data,
        provide_context=True
    )

    process_task = PythonOperator(
        task_id="process_data",
        python_callable=process_data,
        provide_context=True
    )

    fetch_task >> process_task
```




Обрати внимание!

Некоторые используют XCOM для написания логики ветвления в зависимости от того, что было в другом даге

```
from airflow.operators.python import BranchPythonOperator

def choose_branch(ti):
    data = ti.xcom_pull(task_ids="generate_data_task", key="data_key")
    return "proceed_task" if data == 42 else "skip_task"

branch_task = BranchPythonOperator(
    task_id="branch_task",
    python_callable=choose_branch,
    dag=dag
)

proceed_task = DummyOperator(task_id="proceed_task", dag=dag)
skip_task = DummyOperator(task_id="skip_task", dag=dag)

generate_data_task >> branch_task >> [proceed_task, skip_task]
```

Здесь

branch_task выбирает, какую задачу выполнить дальше, исходя из данных, переданных через XCom. Если данные соответствуют ожиданию, выполняется `proceed_task`; если нет — `skip_task`.

▼ XCom vs. Variable: Сравнение

| Критерий | XCom | Variable |
|----------|------|----------|
|----------|------|----------|

| | | |
|--------------------------|--|--|
| Основная цель | Передача данных между задачами | Хранение глобальных конфигураций и данных |
| Срок хранения | Только в рамках выполнения DAG | Постоянное (сохраняется между DAG-запусками) |
| Область видимости | Только внутри одного DAG | Глобально для всех DAG |
| Способ работы | <code>push</code> и <code>pull</code> между задачами | Доступ через <code>Variable.get()</code> и <code>Variable.set()</code> |
| Размер данных | Небольшие данные | Может использоваться для любых данных (с осторожностью для больших) |
| Хранение данных | В базе данных Airflow | В базе данных Airflow |
| Типы данных | Поддерживает разные типы данных | Поддерживает строки и JSON |

Когда использовать XCom вместо Variable:

- Используйте **XCom**, если данные нужны только на этапе выполнения DAG и не будут использоваться в других DAG или при следующих запусках.
- Используйте **Variable**, если вам нужно хранить настройки, параметры, данные между запусками DAG или делиться ими между различными DAG.

Таким образом, XCom отлично подходит для временного обмена данными между задачами внутри одного DAG, в то время как Variables лучше использовать для хранения постоянных параметров и конфигураций, доступных глобально.



Обрати внимание!

Данные в XCom не удаляются автоматически, поэтому они будут накапливаться в базе данных метаданных. Это требует регулярной очистки, чтобы не перегружать базу данных и избежать увеличения объема хранимых данных. **Вместо передачи данных через XCom, используйте базу данных для хранения данных и передавайте ссылки на них через XCom.**

```
def save_data_to_db(**kwargs):
    data = {"key": "value"}
    # Подключение и запись в базу данных
    conn = sqlite3.connect('/path/to/my_database.db')
    cursor = conn.cursor()
    cursor.execute("INSERT INTO my_table (data) VALUES (?)")
    conn.commit()
    conn.close()
    # Передача ссылки на данные через XCom
    kwargs['ti'].xcom_push(key='db_reference', value='/path/to/my_database.db')
```

Отдельные темы

▼ Ограничение количества параллельных задач

Чтобы ограничить количество задач, которые могут выполняться параллельно в Airflow, можно настроить следующие параметры:

1. Параметр `max_active_tasks` на уровне DAG

Вы можете задать `max_active_tasks` в DAG, чтобы ограничить количество одновременно выполняющихся задач внутри конкретного DAG.

```
from airflow import DAG
from datetime import datetime

dag = DAG(
```

```

    "my_dag",
    default_args=default_args,
    max_active_tasks=5, # Ограничение на 5 параллельных зада
ч
    schedule_interval="0 12 * * *",
    start_date=datetime(2023, 1, 1),
)

```

В этом примере не более 5 задач будут выполняться параллельно в пределах одного DAG, даже если DAG содержит больше задач.

2. Параметр `max_active_runs` на уровне DAG

Если DAG настроен для выполнения по расписанию, можно также ограничить количество одновременно активных экземпляров (запусков) DAG, используя `max_active_runs`.

```

dag = DAG(
    "my_dag",
    default_args=default_args,
    max_active_runs=1, # Только один активный запуск DAG одн
овременно
    schedule_interval="0 12 * * *",
    start_date=datetime(2023, 1, 1),
)

```

Этот параметр гарантирует, что одновременно будет запущен только один экземпляр DAG, что косвенно ограничивает количество параллельных задач.

3. Параметр `parallelism` на уровне конфигурации Airflow

В глобальной конфигурации `airflow.cfg` можно установить значение `parallelism`, которое задает максимальное количество параллельных задач для всех DAG на уровне всего Airflow-инстанса.

```

# В airflow.cfg
parallelism = 32

```

Здесь `parallelism = 32` ограничит общее число одновременно выполняющихся задач до 32.

4. Параметр `concurrency` в Executor

Некоторые executors, такие как CeleryExecutor, могут также поддерживать параметр `concurrency` для задания лимита числа задач, которые могут одновременно выполняться worker'ами. Это глобальный параметр, который применяется ко всем DAG'ам и ограничивает параллельное выполнение задач на уровне executors.

▼ SLA

SLA (Service Level Agreement) — это соглашение о сроках выполнения, которое определяет максимальное время, за которое задача или весь DAG должны быть завершены. Если задача превышает указанный SLA, Airflow помечает её как "просроченную" (**missed SLA**) и может отправить уведомление (например, по электронной почте), информируя о нарушении SLA.

Настройка SLA

В Airflow SLA задается на уровне конкретной задачи с помощью параметра `sla` в виде значения `timedelta`, которое определяет предельное время для выполнения задачи.

Например, если задача должна завершиться в течение 30 минут, можно задать

```
sla=timedelta(minutes=30) .
```

Пример использования SLA в DAG:

```
python
Копировать код
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime, timedelta

dag = DAG("sla_example_dag", start_date=datetime(2023, 1, 1),
schedule_interval="@daily")

def my_task_function():
    # Код задачи
    pass

task_with_sla = PythonOperator(
    task_id="task_with_sla",
    python_callable=my_task_function,
```

```
sla=timedelta(minutes=30), # Задача должна завершиться з
а 30 минут
dag=dag
)
```

Как работает SLA

- **Контроль времени:** Airflow отслеживает время выполнения задачи. Если задача превышает указанное значение `sla`, Airflow зафиксировывает нарушение SLA.
- **Отправка уведомлений:** При нарушении SLA, Airflow может отправить уведомление по электронной почте или записать событие в журнал.
- **Журнал нарушений SLA:** Все задачи с нарушением SLA фиксируются в журнале, и их можно увидеть в интерфейсе Airflow, что помогает отслеживать задачи, регулярно превышающие время выполнения.

Примечание

Для корректной работы уведомлений по SLA необходимо настроить параметры для отправки электронной почты в `airflow.cfg`.

▼ Мониторинг и оповещения в Airflow

Airflow поддерживает несколько видов оповещений и мониторинга, которые помогают оперативно отслеживать состояние выполнения задач и DAG'ов. Оповещения могут отправляться по электронной почте или через другие системы мониторинга.

Настройка оповещений в Airflow

1. Оповещения по электронной почте

- Airflow позволяет отправлять уведомления о завершении задачи или DAG при успехе, ошибке или нарушении SLA. Для этого нужно настроить параметры электронной почты в конфигурационном файле `airflow.cfg`, такие как `smtp_host`, `smtp_port`, `smtp_user`, `smtp_password` и `smtp_mail_from`.

Пример задания параметров для оповещений по ошибке и повторной попытке задачи:

```
python
Копировать код
task = PythonOperator(
```

```

        task_id="example_task",
        python_callable=my_task_function,
        retries=3,
        email_on_failure=True,      # Отправка письма при ошибке
        email_on_retry=True,       # Отправка письма при повтор
ной попытке
        email=["your_email@example.com"], # Адреса для оповещ
ений
        dag=dag
    )

```

Здесь:

- **email_on_failure** — отправляет уведомление при неудаче задачи.
- **email_on_retry** — отправляет уведомление при повторной попытке выполнения задачи.
- **email** — адрес(а), на которые будут отправлены уведомления.

2. Использование внешних инструментов мониторинга

- **Prometheus и Grafana:** Для более детального мониторинга можно интегрировать Airflow с Prometheus и Grafana. Prometheus может собирать метрики Airflow (например, статус задач, время выполнения и загрузку системы), а Grafana — визуализировать эти метрики для удобного мониторинга.
- **Elasticsearch и Kibana:** Airflow также можно интегрировать с Elasticsearch и Kibana для хранения и анализа логов. В таком случае все журналы выполнения задач можно просматривать и фильтровать через интерфейс Kibana.

Пример интеграции с Prometheus

Для интеграции с Prometheus и сбора метрик требуется установить плагин

`prometheus_exporter` в Airflow. Этот плагин предоставляет метрики для Prometheus, который затем может быть настроен для их периодического сбора.

```
bash
```

Копировать код

```
pip install apache-airflow-exporter
```

После установки в интерфейсе Prometheus можно настроить дашборды в Grafana для визуализации метрик, таких как:

- Количество успешных, завершившихся с ошибкой и отложенных задач.
- Время выполнения DAG'ов.
- Количество задач в очереди и их статус выполнения.

▼ Что такое Backfill и Catchup, и в чём между ними разница?

- **Backfill** — это выполнение DAG задним числом за указанный диапазон дат, что позволяет выполнить DAG за пропущенные периоды.
- **Catchup** — параметр в настройках DAG, который при значении `True` автоматически запускает DAG для всех пропущенных периодов. Если `Catchup=False`, Airflow будет выполнять DAG только для текущей даты без выполнения за пропущенные интервалы.

▼ Чем отличается ETL от ELT, и как Airflow поддерживает оба подхода?

ETL — это процесс, при котором данные **Extract** (извлекаются) из источника, **Transform** (преобразуются) и затем **Load** (загружаются) в целевую систему. В **ELT** данные сначала **Extract** (извлекаются), затем **Load** (загружаются в хранилище данных), и уже там выполняются **Transform** (преобразования). Airflow поддерживает оба подхода за счёт гибкой структуры DAG и операторов, позволяя настраивать этапы в любом порядке.