Apache Airflow и конвейеры обработки данных

Бас Харенслак
Джулиан де Руйтер

MANNING

ЛАНЬ

ДМК ПРЕСС издательство

Бас Харенслак
Джулиан де Руйтер

Apache Airflow и конвейеры обработки данных

Data Pipelines with Apache Airflow

BAS HARENSLAK
and JULIAN DE RUITER

MANNING
Shelter Island

Apache Airflow и конвейеры обработки данных

БАС ХАРЕНСЛАК
ДЖУЛИАН ДЕ РУЙТЕР

Москва, 2022

УДК 004.4
ББК 32.372
Х20

Харенслак Б., де Руйтер Дж.
Х20 Apache Airflow и конвейеры обработки данных / пер. с англ. Д. А. Беликова. – М.: ДМК Пресс, 2021. – 502 с.: ил.

ISBN 978-5-97060-970-5

Конвейеры обработки данных управляют потоком данных с момента их первоначального сбора до консолидации, очистки, анализа, визуализации и многого другого. Эта книга научит вас создавать и сопровождать эффективные конвейеры обработки данных с использованием платформы Apache Airflow.

Те, кто мало знаком с Airflow, получат базовое представление о принципах работы этой платформы в I части книги. Далее обсуждаются такие темы, как создание собственных компонентов, тестирование, передовые практики и развертывание, – эти главы можно читать в произвольном порядке в зависимости от конкретных потребностей читателя.

Издание предназначено для специалистов по DevOps, обработке и хранению данных, машинному обучению, а также системных администраторов с навыками программирования на Python.

УДК 004.4
ББК 32.372

ISBN 978-1-6172-9690-1 (англ.)
ISBN 978-5-97060-970-5 (рус.)

Оглавление

Содержание

Предисловие

Нам обоим посчастливилось работать инженерами по обработке данных в интересные и трудные времена.

Хорошо это или плохо, но многие компании и организации осознают, что данные играют ключевую роль в управлении их операциями и их улучшении. Последние разработки в области машинного обучения и искусственного интеллекта открыли множество новых возможностей для извлечения выгоды.

Однако внедрение процессов, ориентированных на данные, часто бывает затруднительным, поскольку обычно требуется координировать работу в различных гетерогенных системах и связывать все воедино аккуратно и своевременно для последующего анализа или развертывания продукта.

В 2014 году инженеры Airbnb осознали проблемы управления сложными рабочими процессами данных внутри компании. Чтобы разрешить их, они приступили к разработке Airflow: решения с открытым исходным кодом, позволяющего писать и планировать рабочие процессы, а также отслеживать их выполнение с помощью встроенного веб-интерфейса.

Успех этого проекта быстро привел к тому, что его приняли в рамках Apache Software Foundation, сначала в качестве проекта инкубатора в 2016 году, а затем в качестве проекта верхнего уровня в 2019 году. В результате многие крупные компании теперь используют Airflow для управления многочисленными критически важными процессами обработки данных.

Работая консультантами в GoDataDriven, мы помогли клиентам внедрить Airflow в качестве ключевого компонента в проектах, связанных с созданием озер и платформ данных, моделей машинного обучения и т. д. При этом мы поняли, что передача этих решений может быть непростой задачей, поскольку такие сложные инструменты, как Airflow, трудно освоить в одночасье. По этой причине мы также разработали программу по обучению Airflow в GoData-Driven, часто

организовывали семинары и участвовали в них, чтобы поделиться своими знаниями, мнениями и даже пакетами с открытым исходным кодом. В совокупности эти усилия помогли нам изучить тонкости работы с Airflow, которые не всегда было легко понять, используя доступную нам документацию.

В этой книге мы хотим предоставить исчерпывающее введение в Airflow, которое охватывает все, от построения простых рабочих процессов до разработки собственных компонентов и проектирования / управления развертываниями Airflow. Мы намерены дополнить блоги и другую онлайн-документацию, объединив несколько тем в одном месте в кратком и понятном формате. Тем самым мы надеемся придать вам стимул в работе с Airflow, опираясь на опыт, который мы накопили, и преодолевая трудности, с которыми мы столкнулись за последние годы.

Благодарности

Выход этой книги был бы невозможен без поддержки множества замечательных людей. Коллеги из GoDataDriven и наши друзья поддерживали нас и предоставили ценные предложения и важные наблюдения. Кроме того, читатели, члены программы Manning Early Access Program (MEAP), оставили полезные комментарии на онлайн-форуме.

Рецензенты, участвовавшие в процессе подготовки издания, также предоставили полезные отзывы: Эл Кринкер, Клиффорд Тербер, Дэниел Ламблин, Дэвид Криф, Эрик Платон, Фелипе Ортега, Джейсон Рендель, Джереми Чен, Джири Пик, Джонатан Вуд, Картик Сирасанагандла, Кент Р. Спиллнер, Лин Чен, Филип Бест, Филип Паттерсон, Рамбабу Поза, Ричард Мейнсен, Роберт Г. Гимбель, Роман Павлов, Сальваторе Кампанья, Себастьян Пальма Мардонес, Торстен Вебер, Урсин Стаусс и Влад Навицкий.

Мы выражаем особую благодарность Брайану Сойеру из издательства Manning, редактору отдела закупок, который помог нам сформировать первоначальную заявку на издание этой книги и поверил, что мы сможем довести дело до конца; Трише Лувар, нашему редактору-консультанту по аудитории, которая очень терпеливо отвечала на все наши вопросы, давала важные отзывы по каждой из наших черновиков глав и была для нас важным проводником на протяжении всего пути; а также остальным сотрудникам: редактору проекта Дейрдре Хиам; редактору Мишель Митчелл; корректору Кери Хейлз и техническому корректору Элу Кринкеру.

Бас Харенслак

Я хотел бы поблагодарить своих друзей и семью за их терпение и поддержку в течение этого приключения, длившегося полтора года, которое превратилось из второстепенного проекта в бесчисленное количество дней, ночей и выходных. Стефани, спасибо за то, что все время терпела меня, пока я работал за компьютером. Мириам, Герд и Лотте,

спасибо за то, что терпели меня и верили в меня, пока я писал эту книгу. Я также хотел бы поблагодарить команду GoDataDriven за их поддержку и стремление всегда учиться и совершенствоваться. Пять лет назад я и представить себе не мог, когда начал работать, что стану автором книги.

Джулиан де Руйтер

Прежде всего я хотел бы поблагодарить свою жену Анн Полин и сына Декстера за их бесконечное терпение в течение многих часов, которые я потратил на то, чтобы «еще немного поработать» над книгой. Эта книга была бы невозможна без их непоколебимой поддержки. Также хотел бы поблагодарить нашу семью и друзей за их поддержку и доверие. Наконец, хочу сказать спасибо нашим коллегам из GoDataDriven за их советы и поддержку, от которых я также многому научился за последние годы.

О книге

Эта книга была написана, чтобы помочь вам реализовать рабочие процессы (или конвейеры), ориентированные на обработку данных с помощью Airflow. Она начинается с концепций и механики, участвующих в программном построении рабочих процессов для Apache Airflow с использованием языка программирования Python. Затем мы переходим к более подробным темам, таким как расширение Airflow путем создания собственных компонентов и всестороннего тестирования рабочих процессов. Заключительная часть книги посвящена проектированию и управлению развертывания Airflow, затрагивая такие темы, как безопасность и проектирование архитектур для облачных платформ.

Кому адресована эта книга

Эта книга написана для специалистов и инженеров по обработке данных, которые хотят разрабатывать базовые рабочие процессы в Airflow, а также для инженеров, интересующихся более сложными темами, такими как создание собственных компонентов для Airflow или управление развертываниями Airflow. Поскольку рабочие процессы и компоненты Airflow построены на языке Python, мы ожидаем, что у читателей имеется промежуточный опыт программирования на Python (т. е. они хорошо умеют создавать функции и классы Python, имеют представление о таких понятиях, как * args и ** kwargs и т. д.). Также будет полезен опыт работы с Docker, поскольку большинство примеров кода запускаются с использованием Docker (хотя при желании их можно запускать локально).

Структура книги

Книга состоит из четырех разделов, которые охватывают 18 глав.

Часть I посвящена основам Airflow. В ней объясняется, что такое Airflow, и изложены его основные концепции:

- в главе 1 обсуждается концепция рабочих процессов / конвейеров обработки данных и их создание с помощью Apache Airflow. В ней также рассказывается о преимуществах и недостатках Airflow по сравнению с другими решениями, в том числе приводятся ситуации, при которых вы, возможно, не захотите использовать Apache Airflow;
- глава 2 посвящена базовой структуре конвейеров в Apache Airflow (также известных как ОАГ), с объяснением различных задействованных компонентов и их совместимости;
- в главе 3 показано, как использовать Airflow для планирования работы конвейеров через повторяющиеся промежутки времени, чтобы можно было (например) создавать конвейеры, которые постепенно загружают новые данные с течением времени. В этой главе также рассматриваются тонкости механизма планирования Airflow, который часто является источником путаницы;
- в главе 4 показано, как использовать механизмы создания шаблонов в Airflow для динамического включения переменных в определения конвейера. Это позволяет ссылаться на такие вещи, как даты выполнения расписания в конвейерах;
- в главе 5 демонстрируются различные подходы к определению отношений между задачами в конвейерах, что позволяет создавать более сложные структуры конвейеров с ветками, условными задачами и общими переменными.

В части II подробно рассматривается использование более сложных тем, включая взаимодействие с внешними системами, создание собственных компонентов и разработку тестов для ваших конвейеров:

- в главе 6 показано, как запускать рабочие процессы другими способами, не связанными с фиксированным расписанием, такими как загрузка файлов или вызов по протоколу HTTP;
- в главе 7 демонстрируются рабочие процессы с использованием операторов, которые управляют различными задачами вне Airflow, что позволяет создавать поток событий через системы, которые не связаны между собой;
- в главе 8 объясняется, как создавать собственные компоненты для Airflow, которые позволяют повторно использовать функциональные возможности в разных конвейерах или интегрироваться с системами, не поддерживающими встроенные функции Airflow;
- в главе 9 обсуждаются различные варианты тестирования рабочих процессов Airflow, затрагиваются свойства операторов и способы их применения во время тестирования;

- в главе 10 показано, как использовать рабочие процессы на базе контейнеров для выполнения задач конвейера в Docker или Kubernetes, а также обсуждаются преимущества и недостатки этих подходов.

Часть III посвящена применению Airflow на практике и затрагивает такие темы, как передовые методы, запуск и обеспечение безопасности Airflow и последний демонстрационный пример:

- в главе 11 рассказывается о передовых методах построения конвейеров, которые помогут вам разрабатывать и внедрять эффективные и удобные в сопровождении решения;
- в главе 12 подробно описано несколько тем, которые необходимо учитывать при запуске Airflow в промышленном окружении, например архитектуры для горизонтального масштабирования, мониторинга, журналирования и оповещений;
- в главе 13 обсуждается, как обезопасить установку Airflow, чтобы избежать нежелательного доступа и минимизировать воздействие в случае взлома;
- в главе 14 показан пример проекта Airflow, в котором мы периодически обрабатываем поездки с использованием сервисов Yellow Cab и Citi Bikes по Нью-Йорку, чтобы определить самый быстрый способ передвижения между районами.

В части IV исследуется, как запускать Airflow в облачных платформах. Она включает такие темы, как проектирование развертываний Airflow для облачных платформ и использование встроенных операторов для взаимодействия с облачными сервисами:

- в главе 15 дается общее введение, в котором рассказывается, какие компоненты Airflow участвуют в (облачных) развертываниях, представлена идея, лежащая в основе облачных компонентов, встроенных в Airflow, и рассматриваются варианты самостоятельной реализации развертывания в облаке в сравнении с использованием управляемого решения;
- глава 16 посвящена облачной платформе Amazon AWS. Здесь рассказывается о решениях для развертывания Airflow на AWS и демонстрируется, как применять определенные компоненты для использования сервисов AWS;
- в главе 17 разрабатываются развертывания и демонстрируются облачные компоненты для платформы Microsoft Azure;
- в главе 18 рассматриваются развертывания и облачные компоненты для платформы Google GCP.

Тем, кто плохо знаком с Airflow, следует прочитать главы 1 и 2, чтобы получить внятное представление о том, что такое Airflow, и его возможностях. В главах 3–5 представлена важная информация об основных функциях Airflow. В остальной части книги обсуждаются такие темы, как создание собственных компонентов, тестирование, передовые практики и развертывание, и ее можно читать в произвольном порядке в зависимости от конкретных потребностей читателя.

О коде

Весь исходный код в листингах или тексте набран моноширинным шрифтом, чтобы отделить его от обычного текста. Иногда также используется **жирный шрифт**, чтобы выделить код, который изменился по сравнению с предыдущими шагами в главе, например когда к существующей строке кода добавляется новая функция.

Во многих случаях оригинальный исходный код был переформатирован; мы добавили разрывы строк и переработали отступы, чтобы уместить их по ширине книжных страниц. В редких случаях, когда этого оказалось недостаточно, в листинги были добавлены символы продолжения строки (➥). Кроме того, комментарии в исходном коде часто удаляются из листингов, когда описание кода приводится в тексте. Некоторые листинги сопровождают аннотации, выделяющие важные понятия.

Ссылки на элементы в коде, сценарии или определенные классы, переменные и значения Airflow часто выделяются *курсивом*, чтобы их было легче отличить от окружающего текста.

Исходный код всех примеров и инструкции по их запуску с помощью Docker и Docker Compose доступны в нашем репозитории GitHub (https://github.com/BasPH/data-pipelines-with-apache-airflow), а также на сайте издательства «ДМК Пресс» www.dmkpress.com на странице с описанием соответствующей книги.

ПРИМЕЧАНИЕ В приложении A представлены более подробные инструкции по запуску примеров кода.

Все примеры кода были протестированы с использованием Airflow 2.0. Большинство примеров также следует запускать в более старых версиях Airflow (1.10) с небольшими изменениями. Там, где это возможно, мы включили встроенные указатели, как это сделать. Чтобы вам было легче учитывать различия в путях импорта между Airflow 2.0 и 1.10, в приложении B представлен обзор измененных путей импорта.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning Publications очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторах

БАС ХАРЕНСЛАК – инженер по обработке данных в GoDataDriven, компании, занимающейся разработкой решений на основе данных, расположенной в Амстердаме, Нидерланды. Имея опыт разработки программного обеспечения и информатики, он любит работать над программным обеспечением и данными, как если бы они были сложными головоломками. Он предпочитает работать над ПО с открытым исходным кодом, участвует в проекте Apache Airflow и является одним из организаторов семинара Amsterdam Airflow.

ДЖУЛИАН ДЕ РУИТЕР – инженер машинного обучения, имеет опыт работы в области компьютерных наук и наук о жизни, а также имеет докторскую степень в области вычислительной биологии рака. Будучи опытным разработчиком программного обеспечения, он любит соединять миры науки о данных и инженерии, используя облачное программное обеспечение и программное обеспечение с открытым исходным кодом для разработки решений машинного обучения, готовых к промышленной эксплуатации. В свободное время он с удовольствием разрабатывает собственные пакеты Python, участвует в проектах с открытым исходным кодом и возится с электроникой.

Об иллюстрации на обложке

Рисунок на обложке называется «Femme de l'Isle de Siphanto», или *Женщина с острова Сифанто*. Иллюстрация взята из коллекции костюмов разных стран Жака Грассе де Сен-Совера (1757–1810), «Costumes de Différents Pays», изданной во Франции в 1797 году.

Каждая иллюстрация нарисована и раскрашена вручную. Богатое разнообразие коллекции Грассе де Сен-Совера наглядно напоминает нам о том, насколько обособленными в культурном отношении были города и регионы мира всего 200 лет назад. Изолированные друг от друга люди говорили на разных диалектах и языках. На улице или в деревне было легко определить, где они живут и чем занимаются или каково их положение в обществе, по их одежде.

С тех пор наша манера одеваться изменилась, а разнообразие регионов, столь богатых в то время, исчезло. Сейчас трудно отличить жителей разных континентов, не говоря уже о разных городах, регионах или странах. Возможно, если смотреть на это оптимистично, мы обменяли культурное и визуальное разнообразие на более разнообразную частную жизнь или более разнообразную интеллектуальную и техническую жизнь.

Мы в Manning высоко ценим изобретательность, инициативу и, конечно, радость от компьютерного бизнеса с книжными обложками, основанными на разнообразии жизни в разных регионах два века назад, которое оживает благодаря рисункам Грассе де Сен-Совера.

Часть I

Приступаем к работе

Эта часть подготовит почву на пути к созданию конвейеров для всех видов процессов обработки данных с использованием Apache Airflow. Первые две главы нацелены на то, чтобы предоставить вам обзор того, что такое Airflow и что он умеет.

В главе 1 мы рассмотрим концепции конвейеров обработки данных и обозначим роль, которую выполняет Apache Airflow, помогая реализовать их. Чтобы оправдать ожидания, мы также сравним Airflow с другими технологиями и обсудим, когда он может или не может быть подходящим решением для вашего конкретного случая использования. Далее, в главе 2, вы узнаете, как реализовать свой первый конвейер в Airflow. После его создания мы также рассмотрим, как запустить конвейер и отслеживать его выполнение с помощью веб-интерфейса Airflow.

В главах 3–5 более подробно рассматриваются ключевые концепции Airflow, чтобы вы могли получить внятное представление об основных функциях Airflow.

Глава 3 посвящена семантике планирования, которое позволяет настроить Airflow для запуска конвейеров через равные промежутки времени. Это дает возможность (например) писать конвейеры, которые эффективно загружают и обрабатывают данные ежедневно, еженедельно или ежемесячно. Далее, в главе 4, мы обсудим механизмы шаблонизации в Airflow, которые позволяют динамически ссылаться на такие переменные, как даты выполнения, в ваших конвейерах. Наконец, в главе 5 мы рассмотрим различные подходы к определению

зависимостей задач в конвейерах, которые позволяют определять сложные иерархии задач, включая условные задачи, ветви и т. д.

Если вы новичок в Airflow, то рекомендуем убедиться, что вы понимаете основные концепции, описанные в главах 3–5, поскольку они являются ключом к его эффективному использованию.

Семантика планирования Airflow (описанная в главе 3) может сбивать с толку новичков, поскольку при первом знакомстве эта тема может показаться не совсем логичной.

После завершения части I вы должны быть достаточно подготовлены для написания собственных базовых конвейеров в Apache Airflow и быть готовыми погрузиться в более сложные темы, описанные в частях II–IV.

1 Знакомство с Apache Airflow

Эта глава:

- демонстрирует, как представить конвейеры обработки данных в рабочих процессах в виде графов задач;
- рассказывает, какое место занимает Airflow в экосистеме инструментов управления рабочими процессами;
- поможет определить, подходит ли вам Airflow.

Люди и компании все чаще ориентируются на данные и разрабатывают конвейеры обработки данных как часть своей повседневной деятельности. Объемы данных, задействованных в этих бизнес-процессах, за последние годы значительно увеличились – с мегабайтов в день до гигабайтов в минуту. Хотя обработка такого потока данных может показаться серьезной проблемой, с этими растущими объемами можно справиться с помощью соответствующих инструментов.

В этой книге основное внимание уделяется Apache Airflow, фреймворку для построения конвейеров обработки данных. Ключевая особенность Airflow заключается в том, что он позволяет легко создавать конвейеры обработки данных, запускаемых по расписанию, с использованием гибкой платформы Python, а также предоставляет множество строительных блоков, которые позволяют объединить огромное количество различных технологий, встречающихся в современных технологических ландшафтах.

Airflow очень напоминает паука, сидящего у себя в паутине: он находится в центре ваших процессов обработки данных и координирует работу, происходящую в различных (распределенных) системах. Таким образом, Airflow сам по себе не является инструментом обработки данных. Он управляет различными компонентами, которые отвечают за обработку ваших данных в конвейерах.

В этой главе мы сначала дадим краткое введение в конвейеры обработки данных в Apache Airflow. После этого обсудим ряд соображений, которые следует учитывать при оценке того, подходит ли вам Airflow, и опишем первые шаги по работе с ним.

1.1 Знакомство с конвейерами обработки данных

Обычно конвейеры обработки данных состоят из нескольких задач или действий, которые необходимо выполнить для достижения желаемого результата. Например, предположим, что нам нужно создать небольшое приложение для отображения прогноза погоды на неделю вперед (рис. 1.1). Чтобы реализовать такое приложение, отображающее прогноз в реальном времени, необходимо выполнить следующие шаги:

1. Получить данные прогноза погоды из API погоды.
2. Отфильтровать и преобразовать полученные данные (например, перевести температуру из шкалы Фаренгейта в шкалу Цельсия или наоборот), чтобы данные соответствовали нашей цели.
3. Передать преобразованные данные в приложение.

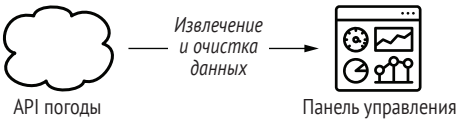


Рис. 1.1 Приложение для отображения прогноза погоды, извлекающее данные из внешнего API и отображающее их в своем интерфейсе

Как видите, этот относительно простой конвейер уже состоит из трех разных задач, каждая из которых выполняет свою часть работы. Более того, эти задачи необходимо выполнять в определенном порядке, поскольку (например) нет смысла пытаться преобразовать данные перед их получением. Точно так же нельзя отправлять новые данные в приложение, пока они не претерпят необходимые преобразования. Таким образом, мы должны убедиться, что этот неявный порядок задач также применяется при запуске данного процесса обработки данных.

1.1.1 Конвейеры обработки данных как графы

Один из способов сделать зависимости между задачами более явными – нарисовать конвейер обработки данных в виде графа. В этом представлении задачам соответствуют узлы графа, а зависимостям между задачами – направленные ребра между узлами задач. Направление ребра указывает направление зависимости, то есть ребро, направленное от задачи A к задаче B, указывает, что задача A должна быть завершена до того, как может начаться задача B. Такие графы обычно называются *ориентированными,* или *направленными*, потому что ребра имеют направление.

Применительно к нашему приложению отображения погоды граф обеспечивает довольно точное представление всего конвейера (рис. 1.2). Просто бегло взглянув на граф, можно увидеть, что наш конвейер состоит из трех разных задач. Помимо этого, направление ребер четко указывает порядок, в котором должны выполняться задачи, достаточно просто проследовать за стрелками.

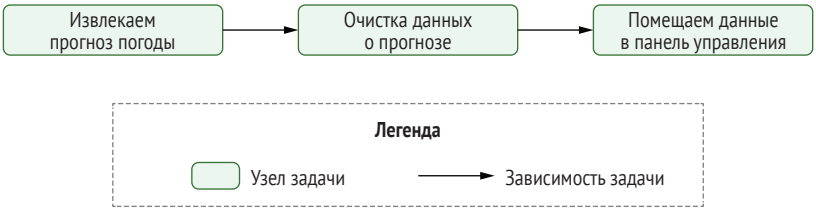


Рис. 1.2 Графовое представление конвейера обработки данных в приложении отображения погоды. Узлы обозначают задачи, а направленные ребра – зависимости между задачами (ребро, направленное от задачи A к задаче B, указывает, что задача A должна быть выполнена раньше задачи B)

Данный тип графа обычно называется *ориентированным ациклическим графом* (ОАГ), поскольку он содержит *ориентированные* ребра и у него нет никаких петель или циклов (*ациклический*). Такое ациклическое свойство чрезвычайно важно, так как оно предотвращает возникновение циклических зависимостей (рис. 1.3) между задачами (где задача A зависит от задачи B и наоборот). Эти циклические зависимости становятся проблемой при попытке выполнить граф, поскольку мы сталкиваемся с ситуацией, когда задача 2 может выполняться только после завершения задачи 3, а задача 3 может выполняться только после завершения задачи 2. Такая логическая непоследовательность приводит к тупиковой ситуации, при которой ни задача 2, ни задача 3 не могут быть запущены, что мешает выполнить граф.

Обратите внимание, что это представление отличается от представлений циклических графов, которые могут содержать циклы для иллюстрации итеративных частей алгоритмов (например), как это

обычно бывает во многих приложениях для машинного обучения. Однако ацикличность ОАГ используется Airflow (и многими другими инструментами управления рабочими процессами) для эффективного разрешения и выполнения этих графов задач.

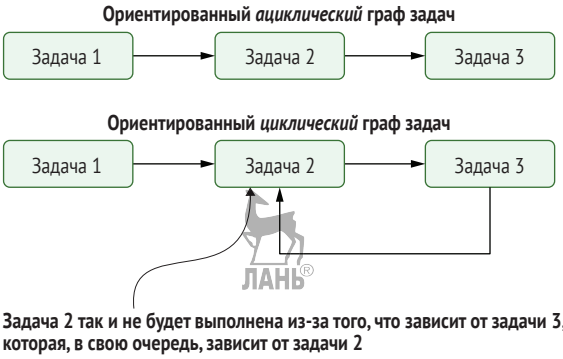


Рис. 1.3 Циклы на графах препятствуют выполнению задачи из-за круговой зависимости. В ациклических графах (вверху) есть четкий путь для выполнения трех разных задач. Однако в циклических графах (внизу) четкого пути выполнения уже нет из-за взаимозависимости между задачами 2 и 3

1.1.2 Выполнение графа конвейера

Прекрасное свойство этого представления состоит в том, что он предоставляет относительно простой алгоритм, который можно использовать для запуска конвейера. Концептуально этот алгоритм состоит из следующих шагов:

1. Для каждой открытой (= незавершенной) задачи в графе выполните следующие действия:
   - для каждого ребра, указывающего *на* задачу, проверьте, завершена ли «вышестоящая» задача на другом конце ребра;
   - если все вышестоящие задачи были выполнены, добавьте текущую задачу в очередь для выполнения.
2. Выполните задачи в очереди, помечая их как выполненные, как только они сделают свою работу.
3. Вернитесь к шагу 1 и повторите действия, пока все задачи в графе не будут выполнены.

Чтобы увидеть, как это работает, проследим за выполнением конвейера в нашем приложении (рис. 1.4). В первой итерации нашего алгоритма мы видим, что задачи *очистки* и *отправки* данных зависят от вышестоящих задач, которые еще не завершены. Таким образом, *зависимости* этих задач не удовлетворены, и их нельзя добавить в очередь выполнения. Однако у задачи *извлечения* данных нет входящих ребер, а это означает, что у нее нет неудовлетворенных вышестоящих зависимостей и, следовательно, ее можно добавить в очередь выполнения.

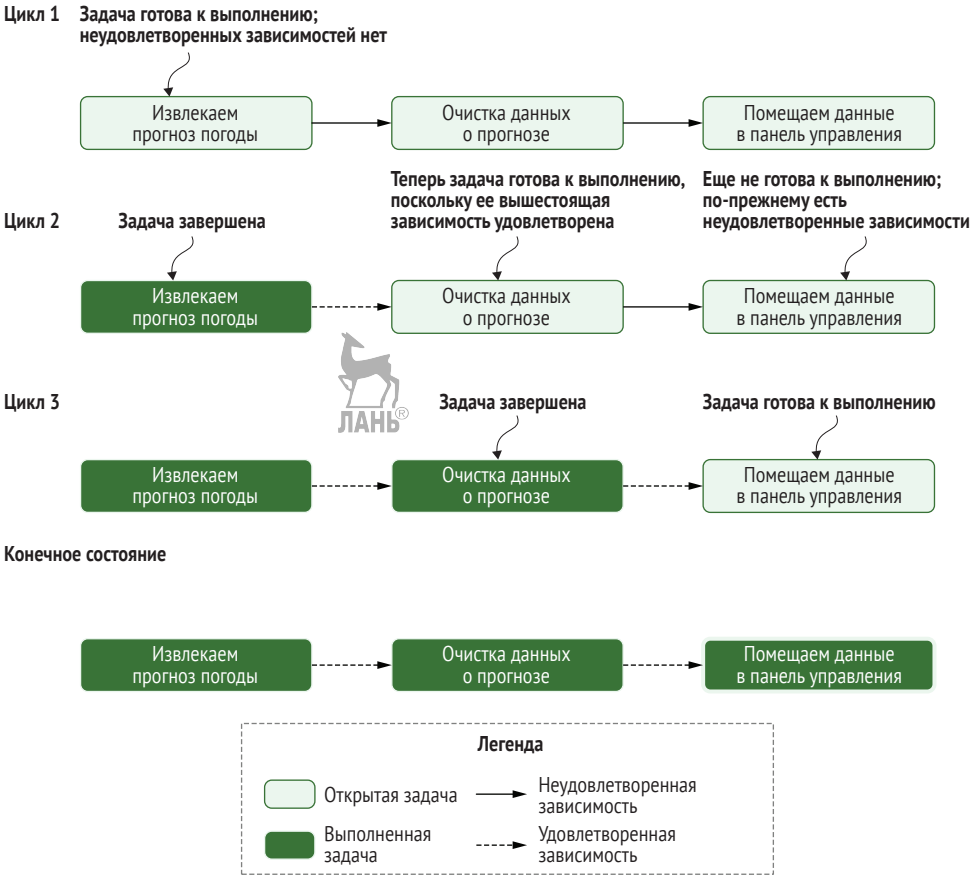


Рис. 1.4 Использование структуры ОАГ для выполнения задач в конвейере обработки данных в правильном порядке: здесь показано, как меняется состояние каждой задачи в каждой итерации алгоритма, что приводит к завершению выполнения конвейера (конечное состояние)

После завершения задачи *извлечения* данных можно перейти ко второй итерации и исследовать зависимости задач *очистки* и *отправки* данных. Теперь мы видим, что задачу *очистки можно* выполнить, поскольку ее вышестоящая зависимость удовлетворена (задача *извлечения* данных выполнена). Таким образом, ее можно добавить в очередь выполнения. Задачу *отправки* данных нельзя добавить в очередь, поскольку она зависит от задачи *очистки*, которую мы еще не запускали.

В третьей итерации, после завершения задачи *очистки*, задача *отправки* данных наконец готова к выполнению, поскольку ее вышестоящая зависимость от задачи *очистки* теперь удовлетворена. В результате мы можем добавить задачу в очередь на выполнение. После того как задача *отправки* данных завершится, у нас не останется задач для выполнения, и выполнение всего конвейера прекратится.

1.1.3 Графы конвейеров и последовательные сценарии

Хотя графовое представление конвейера обеспечивает интуитивно понятный обзор задач в конвейере и их зависимостей, вы можете задаться вопросом, почему бы просто не использовать простой сценарий для выполнения этой линейной цепочки из трех шагов. Чтобы проиллюстрировать преимущества подхода на базе графа, перейдем к более крупному примеру. Здесь к нам обратился владелец компании по производству зонтов, которого вдохновило наше приложение для отображения погоды и теперь он хотел бы попробовать использовать машинное обучение для повышения эффективности работы своей компании. Владельцу компании хотелось бы, чтобы мы реализовали конвейер обработки данных, который создает модель машинного обучения, увязывающую продажи зонтов с погодными условиями. Затем эту модель можно использовать для прогнозирования спроса на зонты в ближайшие недели в зависимости от прогноза погоды на это время (рис. 1.5).

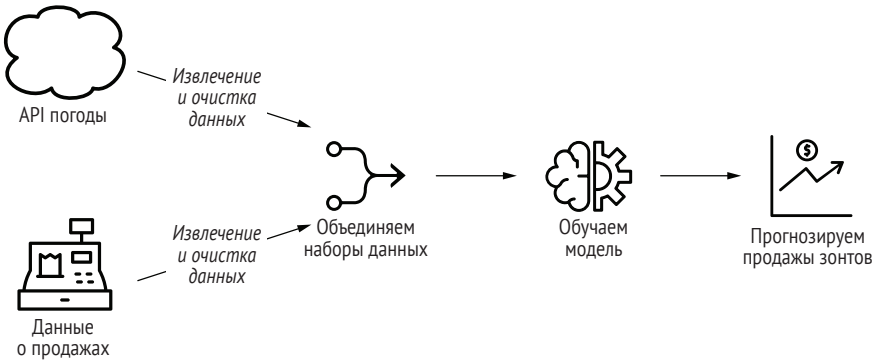


Рис. 1.5 Приложение оценки спроса на зонты, в котором предыдущие данные о погоде и продажах используются для обучения модели, прогнозирующей объем продаж в зависимости от прогноза погоды

Чтобы создать конвейер для обучения такой модели, нужно реализовать следующие шаги.

1. Подготовьте данные о продажах, выполнив такие действия:
   - получите данные о продажах из исходной системы;
   - выполните очистку/преобразование данных о продажах в соответствии с требованиями.
2. Подготовьте данные о погоде, выполнив следующие действия:
   - получите данные прогноза погоды из API;
   - выполните очистку/преобразование данных о погоде в соответствии с требованиями.
3. Объедините наборы данных о продажах и погоде, чтобы создать объединенный набор данных, который можно использовать в качестве входных данных для создания модели машинного обучения.

4 Обучите модель, используя объединенный набор данных.
5 Разверните модель, чтобы ее можно было использовать в бизнесе.

Данный конвейер может быть представлен с использованием того же графового представления, которое мы использовали раньше, путем изображения задач в виде узлов и зависимостей данных между задачами в виде ребер.

Одно из важных отличий от нашего предыдущего примера состоит в том, что первые этапы этого конвейера (получение и очистка данных о погоде и продажах) фактически независимы друг от друга, поскольку включают в себя два отдельных набора данных. Это ясно иллюстрируется двумя отдельными ветвями в графовом представлении конвейера (рис. 1.6), которые могут выполняться параллельно, если мы применяем наш алгоритм выполнения графа, лучше используя доступные ресурсы и потенциально уменьшая время работы конвейера по сравнению с последовательным выполнением задач.

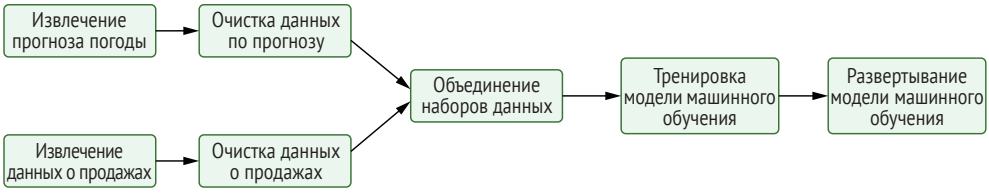


Рис. 1.6 Отсутствие зависимости между задачами, касающимися продаж и погоды в графовом представлении конвейера обработки данных для модели прогноза погоды, для оценки, каким будет спрос на зонты. Два набора задач извлечения и очистки данных независимы, поскольку включают в себя два разных набора данных (наборы данных о погоде и продажах). На эту независимость указывает отсутствие границ между двумя наборами задач

Еще одно полезное свойство графового представления состоит в том, что оно четко разделяет конвейеры на небольшие инкрементные задачи, вместо того чтобы иметь один монолитный сценарий или процесс, который выполняет всю работу. Хотя наличие одного сценария может поначалу казаться не такой уж большой проблемой, это может привести к неэффективности, когда задачи в конвейере будут давать сбой, поскольку нам пришлось бы перезапускать весь сценарий. Напротив, в графовом представлении нужно перезапустить только все неудачные задачи (и все нижестоящие зависимости).

1.1.4 Запуск конвейера с помощью диспетчеров рабочих процессов

Конечно, задача построения графов зависимых задач вряд ли является новой проблемой. За прошедшие годы было разработано множество решений для т. н. «управления рабочими процессами», чтобы

справиться с ней. Обычно эти решения позволяют определять и выполнять графы задач как рабочие процессы или конвейеры.

В табл. 1.1 перечислены некоторые известные диспетчеры рабочих процессов, о которых вы, возможно, слышали.

Таблица 1.1 Обзор известных диспетчеров рабочих процессов и их основных характеристик

Название	Создан[a]	Способ определения рабочих процессов	Написан на	Планирование	Бэкфиллинг (обратное заполнение)	Пользовательский интерфейс[b]	Платформа для установки	Горизонтальная масштабируемость
Airflow	Airbnb	Python	Python	Есть	Есть	Есть	Любая	Есть
Argo	Applatix	YAML	Go	Третья сторона[c]		Есть	Kubernetes	Есть
Azkaban	LinkedIn	YAML	Java	Есть	Нет	Есть	Любая	
Conductor	Netflix	JSON	Java	Нет		Есть	Любая	Есть
Luigi	Spotify	Python	Python	Нет	Есть	Есть	Любая	Есть
Make		Собственный предметно-ориентированный язык	C	Нет	Нет	Нет	Любая	Нет
Metaflow	Netflix	Python	Python	Нет		Нет	Любая	Есть
Nifi	NSA	Пользовательский интерфейс	Java	Есть	Нет	Есть	Любая	Есть
Oozie		XML	Java	Есть	Есть	Есть	Hadoop	Есть

a Некоторые инструменты изначально были созданы (бывшими) сотрудниками компании; однако все они имеют открытый исходный код и не представлены одной отдельной компанией.

b Качество и возможности пользовательских интерфейсов сильно различаются.

c https://github.com/bitphy/argo-cron.

Хотя у каждого из этих диспетчеров имеются свои сильные и слабые стороны, все они предоставляют схожие базовые функции, позволяющие определять и запускать конвейеры, содержащие несколько задач с зависимостями.

Одно из ключевых различий между этими инструментами заключается в том, как они определяют свои рабочие процессы. Например, такие инструменты, как Oozie, используют статические (XML) файлы для их определения, что обеспечивает понятный рабочий процесс, но ограниченную гибкость. Другие решения, такие как Luigi и Airflow, позволяют определять рабочие процессы как код, что дает бóльшую гибкость, но может представлять сложность для чтения и тестирования (в зависимости от навыков программирования у человека, реализующего процесс).

Другие ключевые различия заключаются в объеме функций, предоставляемых диспетчером. Например, такие инструменты, как Make и Luigi, не предоставляют встроенной поддержки для планирования рабочих процессов, а это означает, что вам понадобится дополнительный инструмент, например Cron, если вы хотите запускать рабочий процесс с регулярным расписанием. Другие инструменты могут предоставлять дополнительные функции, такие как планирование, мониторинг, удобные веб-интерфейсы и т. д., встроенные в платформу. Это означает, что вам не нужно самостоятельно объединять несколько инструментов, чтобы получить эти функции.

В целом выбор правильного решения для управления рабочим процессом, отвечающего вашим потребностям, потребует тщательного рассмотрения ключевых особенностей различных решений и того, насколько они соответствуют вашим требованиям. В следующем разделе мы подробнее рассмотрим Airflow, о котором и идет речь в этой книге, и изучим несколько ключевых функций, которые делают его особенно подходящим для обработки процессов или конвейеров, ориентированных на данные.

1.2 Представляем Airflow

В этой книге речь идет о Airflow, решении с открытым исходным кодом для разработки и мониторинга рабочих процессов. В данном разделе мы рассмотрим, в общих чертах, что делает Airflow, после чего перейдем к более подробному изучению того, подходит ли он вам.

1.2.1 Определение конвейеров в коде (Python) гибким образом

Подобно другим диспетчерам рабочих процессов, Airflow позволяет определять конвейеры или рабочие процессы как ОАГ задач. Эти графы очень похожи на примеры, обозначенные в предыдущем разделе, где задачи определены как узлы в графе, а зависимости – как направленные ребра между задачами.

В Airflow ОАГ определяются с помощью кода на языке Python в файлах, которые по сути являются сценариями Python, описывающими структуру соответствующего ОАГ. Таким образом, каждый файл ОАГ обычно описывает набор задач для данного графа и зависимости между задачами, которые затем анализируются Airflow для определения структуры графа (рис. 1.7). Помимо этого, эти файлы обычно содержат некоторые дополнительные метаданные о графе, сообщающие Airflow, как и когда он должен выполняться, и так далее. Подробнее об этом мы поговорим в следующем разделе.

Одно из преимуществ определения ОАГ Airflow в коде Python состоит в том, что этот программный подход обеспечивает большую гибкость при создании графа. Например, как будет показано позже,

вы можете использовать код Python для динамической генерации дополнительных задач в зависимости от определенных условий или даже для генерации целых графов на основе внешних метаданных либо файлов конфигурации. Такая гибкость дает возможность настраивать способ построения конвейеров, позволяя настроить Airflow в соответствии с вашими потребностями при создании конвейеров произвольной сложности.

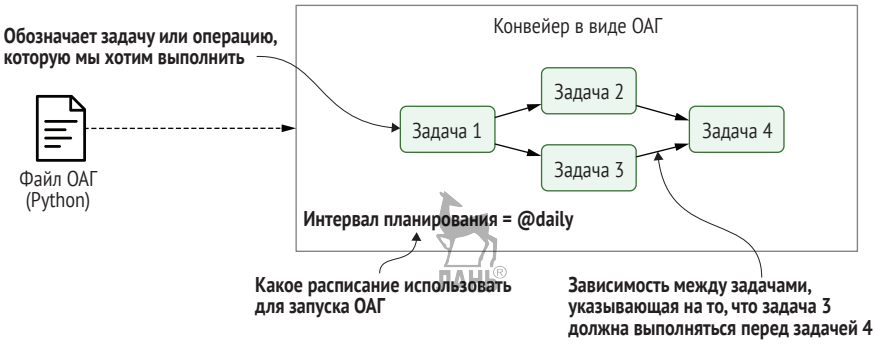


Рис. 1.7 Конвейеры Airflow определены в виде ОАГ с использованием кода Python в файлах ОАГ. Каждый такой файл обычно определяет один граф, который описывает различные задачи и их зависимости. Помимо этого, ОАГ также определяет интервал, который решает, когда Airflow выполняет граф

Помимо этой гибкости, еще одно преимущество того факта, что Airflow написан на языке Python, состоит в том, что задачи могут выполнять любую операцию, которую можно реализовать на Python. Со временем это привело к разработке множества расширений Airflow, позволяющих выполнять задачи в широком спектре систем, включая внешние базы данных, технологии больших данных и различные облачные сервисы, давая возможность создавать сложные конвейеры обработки данных, объединяющие процессы обработки данных в различных системах.

1.2.2 Планирование и выполнение конвейеров

После того как вы определили структуру вашего конвейера (конвейеров) в виде ОАГ, Airflow позволяет вам определить параметр `schedule_interval` для каждого графа, который точно решает, когда ваш конвейер будет запущен Airflow. Таким образом, вы можете дать указание Airflow выполнять ваш граф каждый час, ежедневно, каждую неделю и т. д. Или даже использовать более сложные интервалы, основанные на выражениях, подобных Cron.

Чтобы увидеть, как Airflow выполняет графы, кратко рассмотрим весь процесс, связанный с разработкой и запуском ОАГ. На высоком уровне Airflow состоит из трех основных компонентов (рис. 1.8):

- *планировщик Airflow* – анализирует ОАГ, проверяет параметр `schedule_interval` и (если все в порядке) начинает планировать задачи ОАГ для выполнения, передавая их воркерам Airflow;
- *воркеры*[1] (*workers*) *Airflow* – выбирают задачи, которые запланированы для выполнения, и выполняют их. Таким образом, они несут ответственность за фактическое «выполнение работы»;
- *веб-сервер Airflow* – визуализирует ОАГ, анализируемые планировщиком, и предоставляет пользователям основной интерфейс для отслеживания выполнения графов и их результатов.

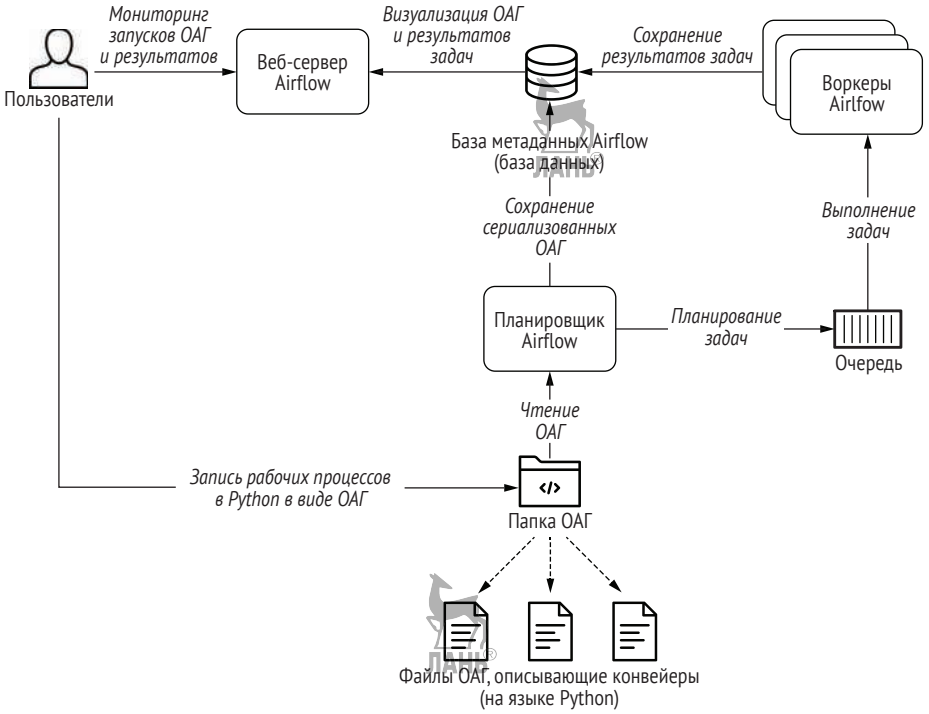


Рис. 1.8 Обзор основных компонентов Airflow (веб-сервер Airflow, планировщик и воркеры)

Сердцем Airflow, вероятно, является планировщик, поскольку именно здесь происходит бóльшая часть магии, определяющей, когда и как будут выполняться ваши конвейеры. На высоком уровне планировщик выполняет следующие шаги (рис. 1.9):

1 После того как пользователи написали свои рабочие процессы в виде ОАГ, файлы, содержащие эти графы, считываются плани-

[1] В англоязычных источниках также встречается термин worker process (рабочий процесс), который, по сути, означает то же самое. Чтобы избежать путаницы со словом workflow (рабочий процесс), в тексте книги используется слово «воркер». – *Прим. перев.*

ровщиком для извлечения соответствующих задач, зависимостей и интервалов каждого ОАГ.

2. После этого для каждого графа планировщик проверяет, все ли в порядке с интервалом с момента последнего чтения. Если да, то задачи в графе планируются к выполнению.
3. Для каждой задачи, запускаемой по расписанию, планировщик затем проверяет, были ли выполнены зависимости (= вышестоящие задачи) задачи. Если да, то задача добавляется в очередь выполнения.
4. Планировщик ждет несколько секунд, прежде чем начать новый цикл, перескакивая обратно к шагу 1.

Рис. 1.9 Схематический обзор процесса, участвующего в разработке и выполнении конвейеров в виде ОАГ с использованием Airflow

Проницательный читатель, возможно, заметит, что фактически шаги, выполняемые планировщиком, очень похожи на алгоритм, приведенный в разделе 1.1. Это не случайно, поскольку Airflow, по сути, выполняет те же шаги, добавляя дополнительную логику для обработки своей логики планирования.

После того как задачи поставлены в очередь на выполнение, с ними уже работает пул воркеров Airflow, которые выполняют задачи параллельно и отслеживают их результаты. Эти результаты передаются в базу метаданных Airflow, чтобы пользователи могли отслеживать

ход выполнения задач и просматривать журналы с помощью веб-интерфейса Airflow (интерфейс, предоставляемый веб-сервером Airflow).

1.2.3 Мониторинг и обработка сбоев

Помимо планирования и выполнения ОАГ, Airflow также предоставляет обширный веб-интерфейс, который можно использовать для просмотра графов и мониторинга результатов их выполнения. После входа (рис. 1.10) на главной странице появляется обширный обзор различных ОАГ со сводными обзорами их последних результатов (рис. 1.11).

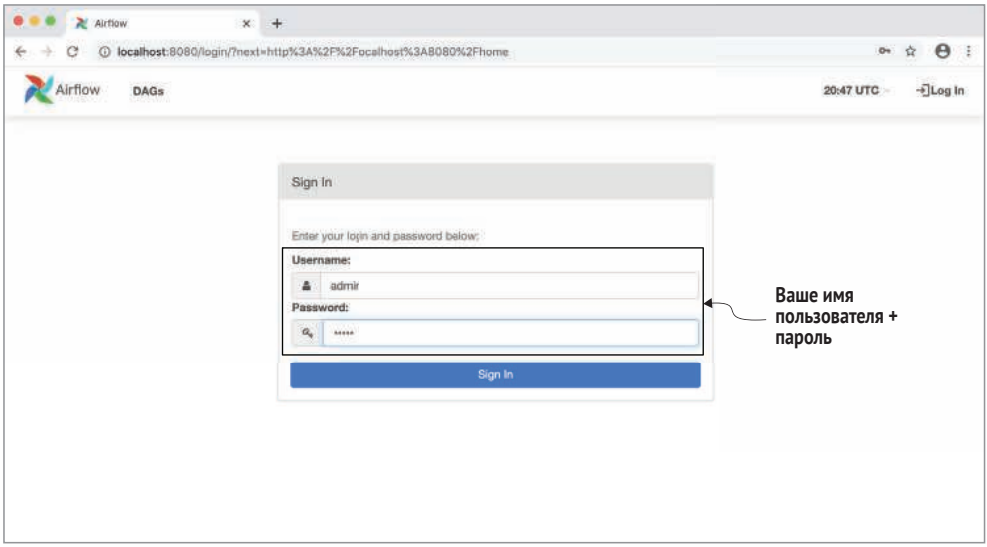


Рис. 1.10 Страница входа в веб-интерфейс Airflow. В примерах кода к этой книге пользователю по умолчанию «admin» предоставляется пароль «admin»

Например, графовое представление отдельного ОАГ дает четкий обзор задач и зависимостей графа (рис 1.12), аналогично схематическим обзорам, которые вы видели в этой главе. Это представление особенно полезно для просмотра структуры графа (обеспечивая подробное понимание зависимостей между задачами), а также для просмотра результатов отдельных запусков ОАГ.

Помимо этого представления, Airflow также предоставляет подробное древовидное представление, в котором показаны все текущие и предыдущие запуски соответствующего ОАГ (рис. 1.13). Это, пожалуй, самое мощное представление, которое дает веб-интерфейс, поскольку здесь приводится беглый обзор того, как работал ОАГ, и оно позволяет покопаться в задачах, завершившихся сбоем, чтобы увидеть, что пошло не так.

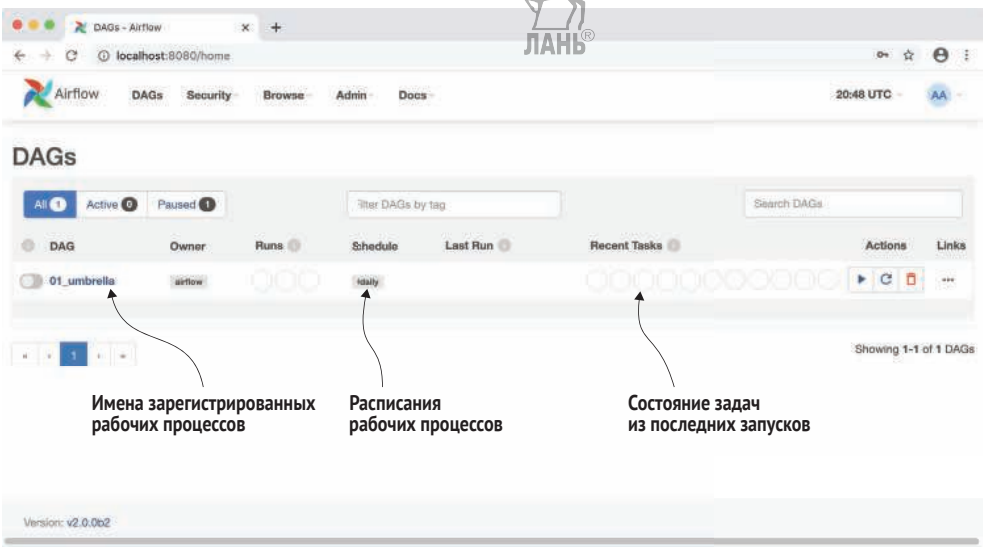


Рис. 1.11 Главная страница веб-интерфейса Airflow с обзором доступных ОАГ и их последние результаты

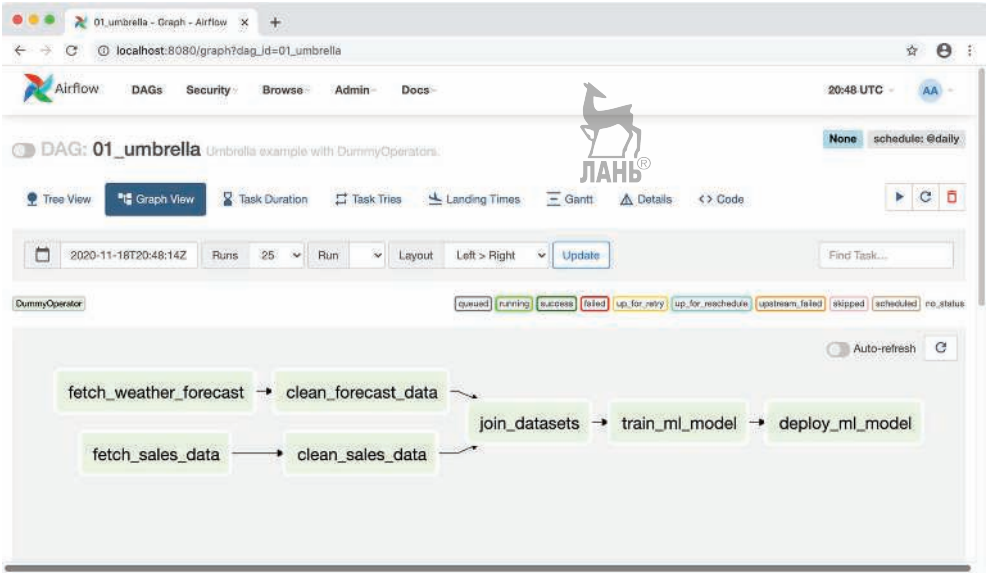


Рис. 1.12 Графовое представление в веб-интерфейсе Airflow, показывающее обзор задач в отдельном ОАГ и зависимости между этими задачами

По умолчанию Airflow может обрабатывать сбои в задачах, повторяя их несколько раз (иногда со временем ожидания между ними), что может помочь задачам восстановиться после периодических сбоев. Если это не помогает, Airflow запишет задачу как неудачную, при желании уведомив вас о сбое, если это предусмотрено настрой-

ками. Отладка сбоев задач довольно проста, поскольку представление в виде дерева позволяет увидеть, какие задачи не удалось выполнить, и изучить их журналы. Это же представление также позволяет очищать результаты отдельных задач для их повторного запуска (наряду со всеми задачами, которые зависят от этой задачи), что дает возможность с легкостью повторно запускать все задачи после внесения изменений в их код.

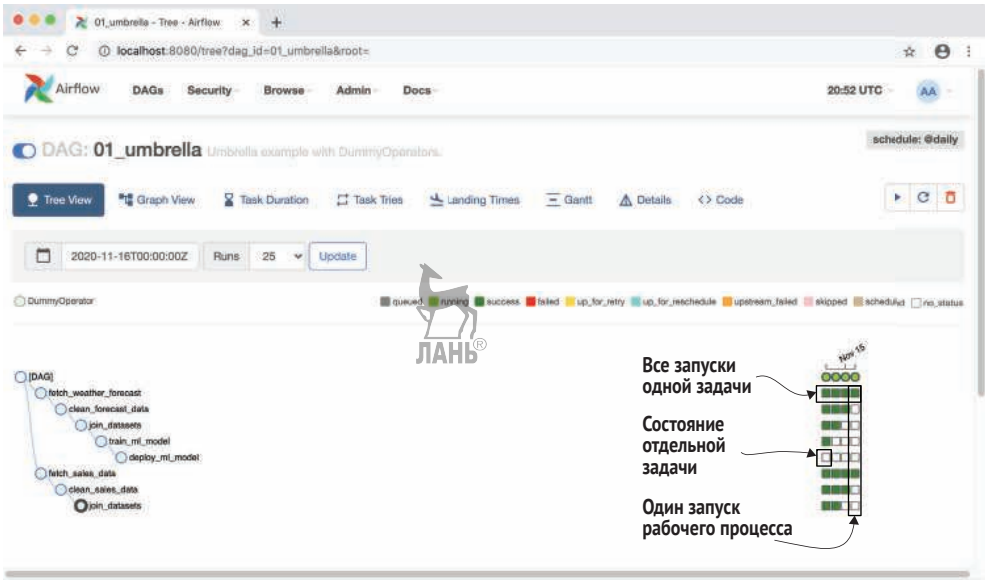


Рис. 1.13 Древовидное представление Airflow, показывающее результаты нескольких запусков ОАГ модели продаж зонтов (самые последние + предыдущие запуски). Столбцы показывают статус одного выполнения ОАГ, а строки показывают статус всех выполнений одной задачи. Цвета (которые вы видите в версии для электронной книги) обозначают результат выполнения соответствующей задачи. Пользователи также могут щелкнуть по «квадратикам», чтобы получить более подробную информацию о данном экземпляре задачи или сбросить состояние задачи, дабы при необходимости ее можно было повторно запустить с помощью Airflow

1.2.4 Инкрементальная загрузка и обратное заполнение

Одно из мощных функций семантики планирования Airflow состоит в том, что вышеуказанные интервалы не только запускают ОАГ в определенные моменты времени (аналогично, например, Cron), но также предоставляют подробную информацию о них и (ожидаемых) следующих интервалах. По сути, это позволяет разделить время на дискретные интервалы (например, каждый день, неделю и т. д.) и запускать ОАГ с учетом каждого из этих интервалов[1].

[1] Если сейчас это звучит для вас немного абстрактно, не волнуйтесь, мы подробно расскажем об этих концепциях позже.

Такое свойство интервалов Airflow неоценимо для реализации эффективных конвейеров обработки данных, поскольку позволяет создавать дополнительные конвейеры. В этих инкрементных конвейерах каждый запуск ОАГ обрабатывает только данные для соответствующего интервала времени (*дельта* данных), вместо того чтобы каждый раз повторно обрабатывать весь набор данных. Это может обеспечить значительную экономию времени и средств, особенно в случае с большими наборами данных, за счет предотвращения дорогостоящего пересчета существующих результатов.

Эти интервалы становятся еще более мощными в сочетании с концепцией обратного заполнения, позволяющей выполнять новый ОАГ для интервалов, которые имели место в прошлом. Эта функция позволяет легко создавать новые наборы архивных данных, просто запуская ОАГ с учетом этих интервалов. Более того, очистив результаты прошлых запусков, вы также можете использовать эту функцию Airflow, чтобы повторно запускать любые архивные задачи, если вы вносите изменения в код задачи, что при необходимости позволяет повторно обрабатывать весь набор данных.

1.3 Когда использовать Airflow

Мы надеемся, что после этого краткого введения в Airflow вы с энтузиазмом познакомились с Airflow и узнали больше о его ключевых функциях. Однако, прежде чем двигаться дальше, для начала рассмотрим причины, по которым вы, возможно, захотите работать с Airflow (а также ряд причин, по которым вы, вероятно, не захотите этого делать), чтобы убедиться, что Airflow – и в самом деле самый подходящий для вас вариант.

1.3.1 Причины выбрать Airflow

В предыдущих разделах мы уже описали несколько ключевых функций, которые делают Airflow идеальным вариантом для реализации конвейеров пакетной обработки данных. Они включают в себя:

- возможность реализовывать конвейеры с использованием кода на языке Python позволяет создавать сколь угодно сложные конвейеры, используя все, что только можно придумать в Python;
- язык Python, на котором написан Airflow, позволяет легко расширять и добавлять интеграции со многими различными системами. Сообщество Airflow уже разработало богатую коллекцию расширений, которые дают возможность Airflow интегрироваться в множество различных типов баз данных, облачных сервисов и т. д.;
- обширная семантика планирования позволяет запускать конвейеры через равные промежутки времени и создавать эффектив-

ные конвейеры, использующие инкрементную обработку, чтобы избежать дорогостоящего пересчета существующих результатов;
- такие функции, как обратное заполнение, дают возможность с легкостью (повторно) обрабатывать архивные данные, позволяя повторно вычислять любые производные наборы данных после внесения изменений в код;
- многофункциональный веб-интерфейс Airflow обеспечивает удобный просмотр результатов работы конвейера и отладки любых сбоев, которые могут произойти.

Дополнительное преимущество Airflow состоит в том, что это фреймворк с открытым исходным кодом. Это гарантирует, что вы можете использовать Airflow для своей работы без какой-либо привязки к поставщику. У некоторых компаний также есть управляемые решения (если вам нужна техническая поддержка), что дает больше гибкости относительно того, как вы запускаете и управляете своей установкой Airflow.

1.3.2 Причины не выбирать Airflow

Хотя у Airflow имеется множество мощных функций, в определенных случаях это, возможно, не то, что вам нужно. Вот некоторые примеры, когда Airflow – не самый подходящий вариант:

- обработка потоковых конвейеров, поскольку Airflow в первую очередь предназначен для выполнения повторяющихся или задач по пакетной обработке данных, а не потоковых рабочих нагрузок;
- реализация высокодинамичных конвейеров, в которых задачи добавляются или удаляются между каждым запуском конвейера. Хотя Airflow может реализовать такое динамическое поведение, веб-интерфейс будет показывать только те задачи, которые все еще определены в самой последней версии ОАГ. Таким образом, Airflow отдает предпочтение конвейерам, структура которых не меняется каждый раз при запуске;
- команды с небольшим опытом программирования (Python) или вообще не имеющие его, поскольку реализация ОАГ в Python может быть сложной задачей для тех, у кого малый опыт работы с Python. В таких командах использование диспетчера рабочих процессов с графическим интерфейсом (например, Azure Data Factory) или определение статического рабочего процесса, возможно, имеет больше смысла;
- точно так же код Python в ОАГ может быстро стать сложным в более крупных примерах. Таким образом, внедрение и поддержка ОАГ в Airflow требуют должной строгости, чтобы поддерживать возможность сопровождения в долгосрочной перспективе.

Кроме того, Airflow – это в первую очередь платформа для управления рабочими процессами и конвейерами, и (в настоящее время)

она не включает в себя более обширные функции, такие как линия данных, управление версиями данных и т. д. Если вам потребуются эти функции, то вам, вероятно, придется рассмотреть возможность объединения Airflow с другими специализированными инструментами, которые предоставляют эти функции.

1.4 Остальная часть книги

К настоящему времени вы должны (мы надеемся) иметь четкое представление о том, что такое Airflow и как его функции могут помочь вам реализовать и запускать конвейеры обработки данных. В оставшейся части этой книги мы начнем с представления основных компонентов Airflow, с которыми вам необходимо ознакомиться, чтобы приступить к созданию собственных конвейеров. Эти первые несколько глав должны иметь большое применение и апеллируют к широкой аудитории. Здесь мы ожидаем, что у вас уже имеется опыт программирования на языке Python (около года). Это означает, что вы должны быть знакомы с такими основными понятиями, как форматирование строк, списковые включения, параметры args и kwargs и т. д. Вы также должны быть знакомы с основами командной строки Linux и иметь хотя бы небольшой опыт использования баз данных (включая SQL) и различных форматов данных.

После этого введения мы углубимся в более сложные функции Airflow, такие как создание динамических ОАГ, реализация собственных операторов, выполнение контейнерных задач и т. д. Эти главы потребуют более глубокого понимания задействованных технологий, включая написание собственных классов Python, основных концепций Docker, форматов файлов и разделения данных. Мы ожидаем, что вторая часть будет особенно интересна data-инженерам.

Наконец, несколько глав в конце книги посвящены темам, связанным с развертыванием Airflow, включая шаблоны развертывания, мониторинг, безопасность и облачные архитектуры. Мы ожидаем, что эти главы будут интересны тем, кто занимается реализацией и управлением развертываний Airflow, например системным администраторам и инженерам DevOps.

Резюме

- Конвейеры обработки данных могут быть представлены в виде ОАГ, которые четко определяют задачи и их зависимости. Эти графы можно выполнять, используя преимущества параллелизма, присущего структуре зависимостей.
- Несмотря на то что на протяжении многих лет для выполнения графов задач было разработано множество диспетчеров рабочих

процессов, Airflow имеет несколько ключевых функций, которые делают его уникальным для реализации эффективных конвейеров пакетной обработки данных.

- Airflow состоит из трех основных компонентов: веб-сервера, планировщика и воркеров, которые работают сообща для планирования задач из конвейеров обработки данных и помогают отслеживать их результаты.

2 Анатомия ОАГ

Эта глава рассказывает:

- о запуске Airflow на собственном компьютере;
- о написании и запуске первого рабочего процесса;
- о первом представлении в интерфейсе Airflow;
- об обработке неудачных задач в Airflow.

В предыдущей главе мы узнали, почему непросто работать с данными и множеством инструментов в ландшафте данных. В этой главе мы приступим к работе с Airflow и рассмотрим пример рабочего процесса, где используются базовые строительные блоки, которые можно встретить во многих рабочих процессах.

При запуске с Airflow полезно иметь некоторый опыт работы с Python, поскольку рабочие процессы определены в коде Python. Пробел в изучении основ Airflow не так уж велик. Как правило, начать работать с базовой структурой рабочего процесса Airflow легко. Рассмотрим пример с фанатом ракет, чтобы увидеть, как Airflow может ему помочь.

2.1 Сбор данных из множества источников

Ракеты – одно из чудес инженерной мысли человечества, и каждый запуск ракеты привлекает внимание во всем мире. В этой главе мы

расскажем о жизни большого поклонника ракет по имени Джон, который следит за каждым запуском ракеты. Новости о запусках содержатся в новостных источниках, которые Джон отслеживает, и, в идеале, ему хотелось бы, чтобы все эти новости были собраны в одном месте. Недавно Джон занялся программированием и хотел бы иметь некоторый автоматизированный способ сбора информации обо всех запусках ракет и в конечном итоге своего рода личное представление о последних новостях, касающихся этих запусков. Для начала Джон решил собрать изображения ракет.

2.1.1 Изучение данных

Для работы с данными мы используем Launch Library 2 (https://thespacedevs.com/llapi), онлайн-хранилище данных о предыдущих и будущих запусках ракет из различных источников. Это бесплатный и открытый API для всех на планете (с учетом ограничений скорости).

В настоящее время Джон интересуется только предстоящими запусками ракет. К счастью, Launch Library предоставляет именно те данные, которые он ищет (https://ll.thespacedevs.com/2.0.0/launch/upcoming/). Здесь есть данные о предстоящих запусках ракет наряду с URL-адресами, где можно найти изображения соответствующих ракет. Вот фрагмент данных, который возвращает этот URL-адрес.

Листинг 2.1 Пример curl-запроса к Launch Library API и ответа

Проверяем ответ в виде URL-адреса с помощью curl из командной строки

```
$ curl -L "https://ll.thespacedevs.com/2.0.0/launch/upcoming"
{
  ...
  "results": [
    {
      "id": "528b72ff-e47e-46a3-b7ad-23b2ffcec2f2",
      "url": "https://.../528b72ff-e47e-46a3-b7ad-23b2ffcec2f2/",
      "launch_library_id": 2103,
      "name": "Falcon 9 Block 5 | NROL-108",
      "net": "2020-12-19T14:00:00Z",
      "window_end": "2020-12-19T17:00:00Z",
      "window_start": "2020-12-19T14:00:00Z",
      "image": "https://spacelaunchnow-prod-
      east.nyc3.digitaloceanspaces.com/media/launch_images/falcon2520925_image
      _20201217060406.jpeg",
      "infographic": ".../falcon2520925_infographic_20201217162942.png",
      ...
    },
    {
      "id": "57c418cc-97ae-4d8e-b806-bb0e0345217f",
      "url": "https://.../57c418cc-97ae-4d8e-b806-bb0e0345217f/",
      "launch_library_id": null,
```

Ответ представляет собой документ в формате JSON, что видно по его структуре

Квадратные скобки обозначают список

Все значения в фигурных скобках относятся к одному запуску ракеты

Здесь мы видим такую информацию, как идентификатор ракеты и окно начала и окончания запуска ракеты

URL-адрес изображения запускаемой ракеты

```
      "name": "Long March 8 | XJY-7 & others",
      "net": "2020-12-22T04:29:00Z",
      "window_end": "2020-12-22T05:03:00Z",
      "window_start": "2020-12-22T04:29:00Z",
      "image": "https://.../long2520march_image_20201216110501.jpeg",
      "infographic": null,
      ...
    },
   ...
 ]
}
```

Как видите, данные представлены в формате JSON и предоставляют информацию о запуске ракеты, а для каждого запуска есть информация о конкретной ракете, такая как идентификатор, имя и URL-адрес изображения. Это именно то, что нужно Джону. Сначала он рисует план, показанный на рис. 2.1, для сбора изображений предстоящих запусков ракет (чтобы использовать изображения из этого каталога в качестве заставки).

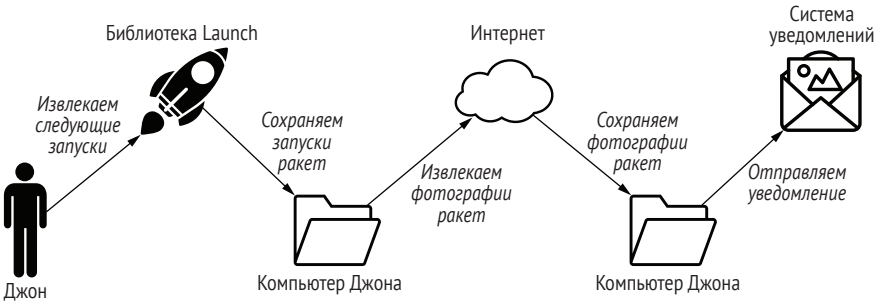


Рис. 2.1 Ментальная модель Джона загрузки изображений ракет

Основываясь на примере из рис. 2.1, мы видим, что в конце дня цель Джона – получить каталог, заполненный изображениями ракет, такими как Ariane 5 ECA на рис. 2.2.

2.2 Пишем наш первый ОАГ

Пример с Джоном прекрасно обрисован, поэтому давайте посмотрим, как запрограммировать его план. Здесь всего лишь несколько шагов, и теоретически можно было бы решить эту задачу с помощью одной строки кода. Тогда зачем нам для этого такая система, как Airflow?

Преимущество Airflow состоит в том, что мы можем разделить большую работу, состоящую из одного или нескольких шагов, на отдельные «задачи», вместе образующие ОАГ. Несколько задач могут выполняться параллельно, и задачи могут использовать разные технологии. Например, сначала можно было бы запустить сценарий

Bash, а затем сценарий на языке Python. Мы разбили ментальную модель Джона его рабочего процесса на три логические задачи в Airflow, как показано на рис. 2.3.

Рис. 2.2 Пример изображения ракеты Ariane 5 ECA

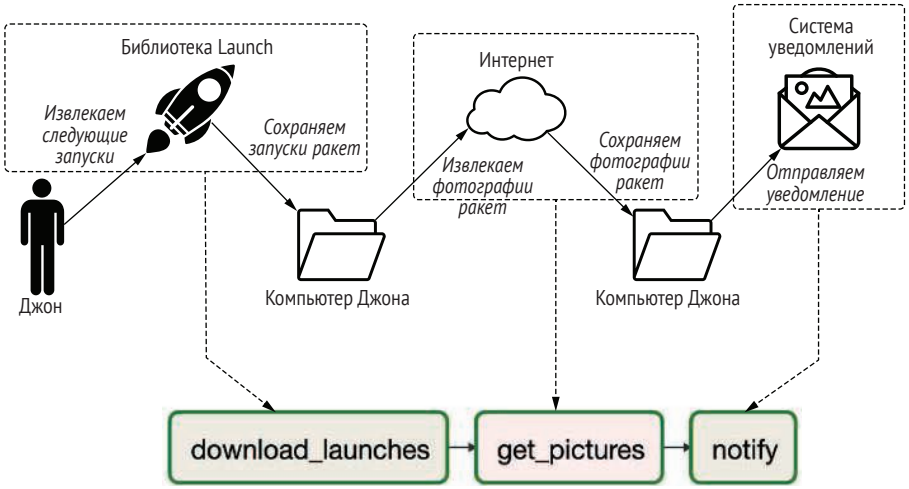


Рис. 2.3 Ментальная модель Джона, сопоставленная с задачами в Airflow

Вы спросите, зачем нужны эти три задачи? Почему бы не скачать запуски и соответствующие изображения в одной задаче? Или не разделить их на пять задач?

В плане Джона у нас есть пять стрелок. Все эти вопросы можно задать при разработке рабочего процесса, но на самом деле здесь нет правильного или неправильного ответа. Однако есть несколько моментов, которые следует принять во внимание, и на протяжении всей книги мы будем прорабатывать многие из этих примеров, чтобы понять, что правильно, а что нет. Код этого рабочего процесса выглядит следующим образом:

Листинг 2.2 ОАГ для скачивания и обработки данных о запуске ракеты

```
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

dag = DAG(
    dag_id="download_rocket_launches",
    start_date=airflow.utils.dates.days_ago(14),
    schedule_interval=None,
)

download_launches = BashOperator(
    task_id="download_launches",
    bash_command="curl -o /tmp/launches.json -L
      'https://ll.thespacedevs.com/2.0.0/launch/upcoming'",
    dag=dag,
)

def _get_pictures():
    # Убеждаемся, что каталог существует
    pathlib.Path("/tmp/images").mkdir(parents=True, exist_ok=True)

    # Скачиваем все изображения в launches.json
    with open("/tmp/launches.json") as f:
        launches = json.load(f)
        image_urls = [launch["image"] for launch in launches["results"]]
        for image_url in image_urls:
            try:
                response = requests.get(image_url)
                image_filename = image_url.split("/")[-1]
                target_file = f"/tmp/images/{image_filename}"
```

Создаем экземпляр объекта ОАГ; это отправная точка любого рабочего процесса

Имя ОАГ

Дата, когда ОАГ должен впервые быть запущен

Интервал, с которым должен запускаться ОАГ

Применяем Bash, чтобы загрузить ответ в виде URL-адреса с помощью curl

Имя задачи

Функция Python проанализирует ответ и загрузит все изображения ракет

```
                with open(target_file, "wb") as f:
                    f.write(response.content)
                print(f"Downloaded {image_url} to {target_file}")
            except requests_exceptions.MissingSchema:
                print(f"{image_url} appears to be an invalid URL.")
            except requests_exceptions.ConnectionError:
                print(f"Could not connect to {image_url}.")
get_pictures = PythonOperator(
    task_id="get_pictures",
    python_callable=_get_pictures,
    dag=dag,
)

notify = BashOperator(
    task_id="notify",
    bash_command='echo "There are now $(ls /tmp/images/ | wc -l) images."',
    dag=dag,
)

download_launches >> get_pictures >> notify
```

Вызываем функцию Python в ОАГ с помощью PythonOperator

Задаем порядок выполнения задач

Разберем этот рабочий процесс. ОАГ – это отправная точка любого рабочего процесса. Все задачи в рамках рабочего процесса ссылаются на этот объект ОАГ, чтобы Airflow знал, какие задачи какому ОАГ принадлежат.

Листинг 2.3 Создание экземпляра объекта ОАГ

```
dag = DAG(
    dag_id="download_rocket_launches",
    start_date=airflow.utils.dates.days_ago(14),
    schedule_interval=None,
)
```

Класс DAG принимает два обязательных аргумента

Имя ОАГ, отображаемое в пользовательском интерфейсе Airflow

Дата и время, когда рабочий процесс должен быть запущен в первый раз

Обратите внимание, что `dag` (нижний регистр) – это имя, присвоенное экземпляру класса `DAG` (верхний регистр). Имя экземпляра может быть любым; можно назвать его `rocket_dag` или как-то еще. Мы будем ссылаться на переменную (`dag`) во всех операторах, которая сообщает Airflow, к какому ОАГ принадлежит оператор.

Также обратите внимание, что мы задали для `schedule_interval` значение `None`. Это означает, что ОАГ не будет запускаться автоматически. На данный момент его можно запустить вручную из пользовательского интерфейса Airflow. Мы перейдем к планированию в разделе 2.4.

Затем сценарий рабочего процесса Airflow состоит из одного или нескольких операторов, которые выполняют фактическую работу. В листинге 2.4 мы применяем `BashOperator` для запуска команды Bash.

Листинг 2.4 Создание экземпляра BashOperator для запуска команды Bash

```
download_launches = BashOperator(
    task_id="download_launches",
    bash_command="curl -o /tmp/launches.json 'https://
      ll.thespacedevs.com/2.0.0/launch/upcoming'",
    dag=dag,
)
```

Имя задачи

Команда Bash для выполнения

Ссылка на переменную DAG

Каждый оператор выполняет одну единицу работы, а несколько операторов вместе образуют рабочий процесс или ОАГ в Airflow. Операторы работают независимо друг от друга, хотя вы можете определить порядок выполнения, который мы называем *зависимостями* в Airflow. В конце концов, рабочий процесс Джона был бы бесполезен, если бы сначала вы попытались скачать изображения, когда у вас нет данных об их местонахождении. Чтобы задачи выполнялись в правильном порядке, можно установить зависимости между задачами.

Листинг 2.5 Определение порядка выполнения задачи

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

Стрелки задают порядок выполнения задач

В Airflow можно использовать *бинарный оператор сдвига вправо* (например, *«rshift»* [>>]) для определения зависимостей между задачами. Это гарантирует, что задача `get_pictures` запустится только после успешного завершения `download _launches`, а задача `notify` запустится только после успешного завершения `get_pictures`.

ПРИМЕЧАНИЕ В Python для сдвига битов используется оператор rshift (>>), который является обычной операцией, например в криптографических библиотеках. В Airflow нет примера сдвига битов, и оператор rshift был переопределен, чтобы обеспечить читабельный способ определения зависимостей между задачами.

2.2.1 Задачи и операторы

Вы можете спросить: в чем разница между задачами и операторами? В конце концов, они оба выполняют фрагмент кода. В Airflow *операторы* имеют единственную ответственность: они существуют для выполнения одной-единственной единицы работы. Некоторые операторы выполняют универсальные вещи, например `BashOperator` (используется для запуска сценария Bash) или `PythonOperator` (используется для запуска функции Python); у других есть более конкретные

варианты использования, такие как `EmailOperator` (используется для отправки электронной почты) или `SimpleHTTPOperator` (используется для вызова конечной точки HTTP).

Роль ОАГ состоит в том, чтобы организовать выполнение набора операторов. Сюда входит запуск и остановка операторов, запуск последовательных задач после выполнения оператора, обеспечение зависимостей между операторами и т. д.

В данном контексте и в документации Airflow мы видим, что термины *оператор* и *задача* используются как взаимозаменяемые. С точки зрения пользователя, они обозначают одно и то же и часто заменяют друг друга во время обсуждений. Операторы обеспечивают реализацию одной единицы работы. У Airflow есть класс `BaseOperator` и множество подклассов, наследуемых от него, такие как `PythonOperator`, `EmailOperator` и `OracleOperator`. В любом случае они выполняют одну единицу работы.

Однако есть разница. Задачи в Airflow управляют выполнением оператора; их можно рассматривать как небольшую оболочку или менеджер вокруг оператора, который обеспечивает его правильное выполнение. Пользователь может сосредоточиться на выполняемой работе с помощью операторов, а Airflow обеспечивает правильное выполнение с помощью задач (рис. 2.4).

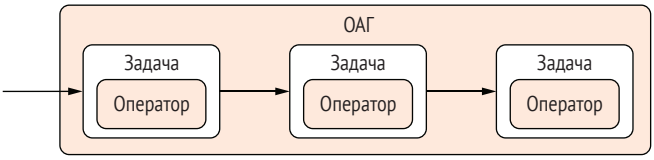


Рис. 2.4 ОАГ и операторы используются пользователями Airflow. Задачи – это внутренние компоненты для управления состоянием оператора и отображения изменений состояния (например, запущено/завершено) для пользователя

2.2.2 Запуск произвольного кода на Python

Извлечение данных по последующим запускам ракет – это одна команда curl в Bash, которая легко выполняется с помощью оператора `BashOperator`. Однако для анализа результата в формате JSON, выбора из него URL-адресов изображений и скачивания соответствующих изображений требуется немного больше усилий. Хотя все это по-прежнему возможно в одной строке кода Bash, часто проще и удобнее читать такой код, если он написан с помощью нескольких строк Python или любого другого языка на ваш выбор. Поскольку код Airflow написан на языке Python, удобно хранить и рабочий процесс, и логику выполнения в одном сценарии. Чтобы скачать изображения ракет, мы реализовали следующий листинг.

Листинг 2.6 Запуск функции Python с помощью PythonOperator

```
def _get_pictures():
    # Убеждаемся, что каталог существует
    pathlib.Path("/tmp/images").mkdir(parents=True, exist_ok=True)

    # Скачиваем все изображения в launches.json
    with open("/tmp/launches.json") as f:
        launches = json.load(f)
        image_urls = [launch["image"] for launch in launches["results"]]
        for image_url in image_urls:
            try:
                response = requests.get(image_url)
                image_filename = image_url.split("/")[-1]
                target_file = f"/tmp/images/{image_filename}"
                with open(target_file, "wb") as f:
                    f.write(response.content)
                print(f"Downloaded {image_url} to {target_file}")
            except requests_exceptions.MissingSchema:
                print(f"{image_url} appears to be an invalid URL.")
            except requests_exceptions.ConnectionError:
                print(f"Could not connect to {image_url}.")

get_pictures = PythonOperator(
    task_id="get_pictures",
    python_callable=_get_pictures,
    dag=dag,
)
```

Функция Python, которую нужно вызвать

Создаем каталог изображений, если его не существует

Открываем результат из предыдущей задачи

Скачиваем каждое изображение

Сохраняем каждое изображение

Вывод в stdout; будет зафиксировано в журналах Airflow

Создаем PythonOperator для вызова функции Python

Указываем на функцию Python, которую нужно выполнить

`PythonOperator` в Airflow отвечает за запуск кода Python. Как и ранее использовавшийся `BashOperator`, этот и все другие операторы требуют `task_id`. На `task_id` ссылаются при запуске задачи, и он отображается в пользовательском интерфейсе. Использование `PythonOperator` всегда состоит из двух частей:

1. мы определяем сам оператор (`get_pictures`);
2. аргумент `python_callable` указывает на вызываемый объект, обычно функцию (`_get_pictures`).

При запуске оператора вызывается функция Python, которая будет выполнять функцию. Давайте разберем ее. Базовое использование `PythonOperator` всегда выглядит так, как показано на рис. 2.5.

Хотя это и не обязательно, для удобства мы оставляем `task_id` в качестве имени переменной `get_pictures`.

Листинг 2.7 Убеждаемся, что выходной каталог существует, и создаем его, если его нет

```
# Убеждаемся, что каталог существует
pathlib.Path("/tmp/images").mkdir(parents=True, exist_ok=True)
```

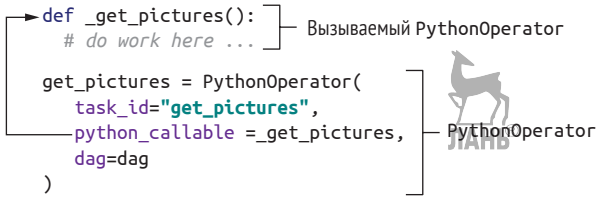


Рис. 2.5 Аргумент `python_callable` в `PythonOperator` указывает на функцию, которую нужно выполнить

Первый шаг – убедиться, что каталог, в котором будут храниться изображения, существует, как показано в листинге 2.7. Затем мы открываем результат, скачанный из Launch Library API, и извлекаем URL-адреса изображений для каждого запуска.

Листинг 2.8 Извлекаем URL-адреса изображений для каждого запуска ракеты

Открываем файл запусков ракет в формате JSON

Читаем как словарь, чтобы можно было смешивать данные

```
with open("/tmp/launches.json") as f:
    launches = json.load(f)
    image_urls = [launch["image"] for launch in launches["results"]]
```

Для каждого запуска извлекаем элемент "image"

Каждый URL-адрес изображения вызывается для скачивания изображения и сохранения его в каталоге /tmp/images.

Листинг 2.9 Скачиваем все изображения из URL-адресов извлеченных изображений

Проходим по всем URL-адресам изображений

```
for image_url in image_urls:
    try:
        response = requests.get(image_url)
        image_filename = image_url.split("/")[-1]
        target_file = f"/tmp/images/{image_filename}"
        with open(target_file, "wb") as f:
            f.write(response.content)
        print(f"Downloaded {image_url} to {target_file}")
    except requests_exceptions.MissingSchema:
        print(f"{image_url} appears to be an invalid URL.")
    except requests_exceptions.ConnectionError:
        print(f"Could not connect to {image_url}.")
```

Получаем изображение

Получаем только имя файла, выбрав все после последнего. Например, https://host/RocketImages/Electron.jpg_1440.jpg → Electron.jpg_1440.jpg

Создаем путь к целевому файлу

Открываем дескриптор целевого файла

Записываем изображение в путь к файлу

Выводим результат

Перехватываем и обрабатываем потенциальные ошибки

2.3 Запуск ОАГ в Airflow

Теперь, когда у нас есть ОАГ, давайте выполним его и рассмотрим его в пользовательском интерфейсе Airflow. В минимальном варианте Airflow состоит из трех основных компонентов: планировщика, веб-сервера и базы данных. Чтобы запустить Airflow, можно либо установить Airflow в окружении Python, либо запустить контейнер Docker.

2.3.1 Запуск Airflow в окружении Python

Чтобы установить и запустить Airflow как пакет Python из PyPi, необходимо выполнить несколько шагов:

```
pip install apache-airflow
```

Убедитесь, что вы устанавливаете `apache-airflow`, а не просто `airflow`. После присоединения к Apache Foundation в 2016 году репозиторий PyPi `airflow` был переименован в `apache-airflow`. Поскольку многие по-прежнему устанавливали `airflow`, вместо того чтобы удалить старый репозиторий, его оставили в качестве пустышки, чтобы выдавать всем сообщение, указывающее на правильный репозиторий.

Некоторые операционные системы поставляются с установкой Python. Если просто выполнить команду `pip install apache-airflow` , то вы установите Airflow в этом «системном» окружении. При работе над проектами Python желательно, чтобы каждый проект находился в своем окружении Python, дабы создать воспроизводимый набор пакетов Python и избежать конфликтов зависимостей. Такие окружения создаются с помощью следующих инструментов:

- pyenv: https://github.com/pyenv/pyenv;
- Conda: https://docs.conda.io/en/latest/;
- virtualenv: https://virtualenv.pypa.io/en/latest/.

После установки Airflow запустите его, инициализировав базу метаданных (где хранится состояние Airflow), создав пользователя, скопировав наш ОАГ в каталог ОАГ и запустив планировщик и веб-сервер:

```
1 airflow db init;
2 airflow users create --username admin --password admin --first-
  name Anonymous --lastname Admin --role Admin --email admin@ex-
  ample.org;
3 cp download_rocket_launches.py ~/airflow/dags/;
4 airflow webserver;
5 airflow scheduler.
```

Обратите внимание, что планировщик и веб-сервер являются непрерывными процессами, которые держат ваш терминал открытым, поэтому они запускаются в фоновом режиме с помощью `airflow webserver` и/или открывают второе окно терминала, чтобы запустить пла-

нировщик и веб-сервер по отдельности. После настройки перейдите по адресу http://localhost: 8080 и выполните вход с именем пользователя «admin» и паролем «admin».

2.3.2 Запуск Airflow в контейнерах Docker

Контейнеры Docker также пользуются популярностью, чтобы создавать изолированные окружения для запуска воспроизводимого набора пакетов Python и предотвращения конфликтов зависимостей. Однако эти контейнеры создают изолированное окружение на уровне операционной системы, тогда как в случае с окружением Python речь идет об изоляции лишь на уровне среды выполнения. В результате вы можете создавать контейнеры Docker, которые содержат не только набор пакетов Python, но и другие зависимости, такие как драйверы базы данных или компилятор GCC. В этой книге мы продемонстрируем работу Airflow в контейнерах Docker, используя для этого несколько примеров.

Для запуска контейнеров Docker на вашем компьютере должен быть установлен Docker Engine. Затем можно запустить Airflow в Docker с помощью следующей команды.

Листинг 2.10 Запуск Airflow в Docker

```
docker run \
-ti \
-p 8080:8080 \
-v ➥ /path/to/dag/download_rocket_launches.py:/opt/airflow/dags/
    download_rocket_launches.py \
--entrypoint=/bin/bash \
--name airflow \
apache/airflow:2.0.0-python3.8 \
-c '( \
airflow db init && \
➥ airflow users create --username admin --password admin --firstname
    Anonymous --lastname Admin --role Admin --email admin@example.org \
); \
airflow webserver & \
airflow scheduler \
'
```

Открываем доступ на хост-порте 8080

Монтируем файл OAГ в контейнере

Образ Airflow Docker

Инициализируем базу метаданных в контейнере

Создаем пользователя

Запускаем веб-сервер

Запускаем планировщик

ПРИМЕЧАНИЕ Если вы знакомы с Docker, то, вероятно, возразите, что нежелательно запускать несколько процессов в одном контейнере Docker, как показано в листинге 2.10. Это отдельная команда, предназначенная для демонстрационных целей, позволяющая быстро приступить к работе. В настройках, предназначенных для промышленного окружения, нужно запускать веб-сервер, планировщик и базу метаданных Airflow в отдельных контейнерах. Об этом подробно написано в главе 10.

После этого будет скачан и запущен образ Airflow Docker `apache/airflow`. После запуска вы можете перейти по адресу http://localhost:8080 и выполнить вход с именем пользователя «admin» и паролем «admin».

2.3.3 Изучаем пользовательский интерфейс Airflow

Первое, что вы увидите в Airflow, – это экран входа, показанный на рис. 2.6.

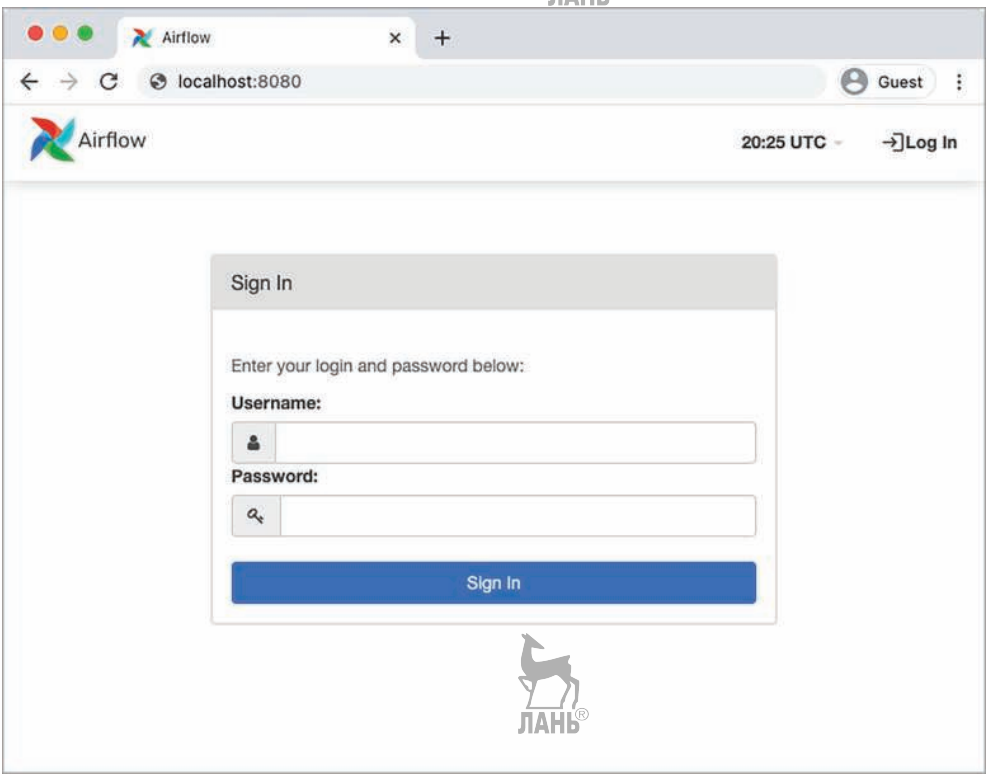


Рис. 2.6 Экран входа в Airflow

Выполнив вход, вы можете просмотреть ОАГ, download_rocket_launches, как показано на рис. 2.7.

Это первое знакомство с Airflow. В настоящее время единственный ОАГ – это download_rocket_launches, который доступен в Airflow в каталоге ОАГ. На главном экране много информации, но вначале посмотрим на download_rocket _launches. Щелкните по имени ОАГ, чтобы открыть его и просмотреть графовое представление (рис. 2.8).

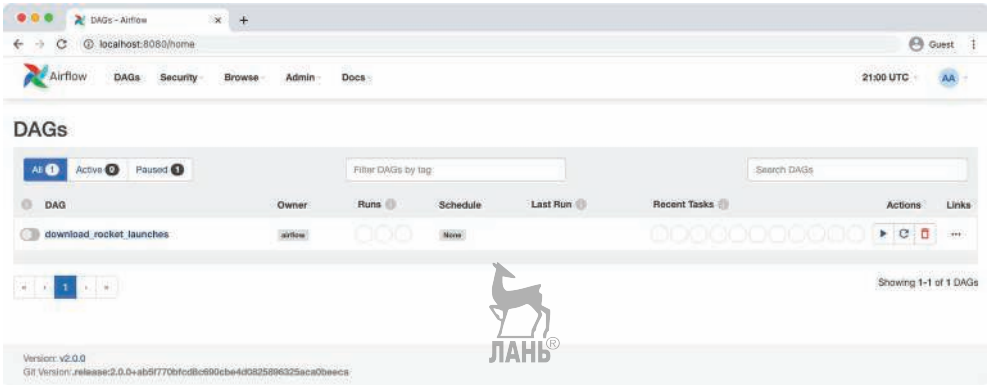


Рис. 2.7 Главный экран Airflow

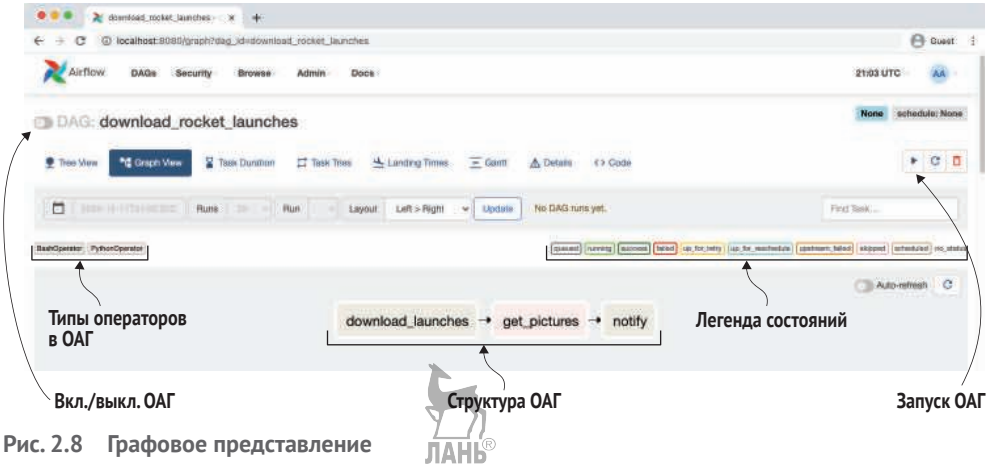


Рис. 2.8 Графовое представление

Данное представление показывает нам структуру сценария ОАГ, предоставленного Airflow. После размещения в каталоге ОАГ Airflow прочитает сценарий и извлечет фрагменты, которые все вместе образуют ОАГ, чтобы их можно было визуализировать в пользовательском интерфейсе. Здесь мы видим структуру ОАГ, а также то, в каком порядке связаны все задачи в ОАГ и как они будут запускаться. Это одно из представлений, которое вы, вероятно, будете чаще всего использовать при разработке рабочих процессов.

В легенде состояния отображаются все цвета, которые можно увидеть во время запуска, поэтому посмотрим, что происходит, и запустим ОАГ. Во-первых, чтобы выполнить запуск, ОАГ должен быть «включен»; для этого переключите кнопку рядом с его именем. Затем нажмите кнопку **Play** (Воспроизвести), чтобы запустить его.

После этого он запустится, и вы увидите текущее состояние рабочего процесса, обозначенное цветами (рис. 2.9). Поскольку мы задаем зависимости между нашими задачами, задачи, идущие друг за другом, начинают выполняться только после завершения предыдущих. Проверим результат задачи *уведомления*. В реальном примере вы, вероятно, захотите отправить электронное письмо или, например, уведомление Slack, чтобы сообщить о новых изображениях. Для простоты сейчас выводится количество скачанных изображений. Проверим журналы.

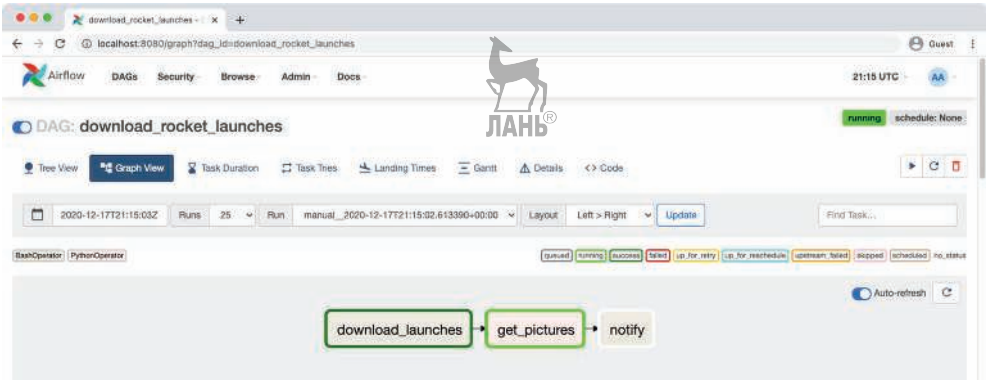


Рис. 2.9 Графовое представление запущенного ОАГ

Все журналы задач собраны в Airflow, поэтому можно искать в пользовательском интерфейсе выходные данные или возможные проблемы в случае сбоя. Щелкните по завершенной задаче уведомления, и вы увидите всплывающее окно с несколькими вариантами, как показано на рис. 2.10.

Щелкните по кнопке **Log** (Журнал) вверху в центре, чтобы просмотреть журналы, как показано на рис. 2.11. По умолчанию журналы содержат довольно много информации, но в них отображается количество скачанных изображений. Наконец, можно открыть каталог /tmp/images и просмотреть их. При запуске в Docker этот каталог существует только внутри контейнера Docker, а не в вашей хост-системе. Поэтому сначала нужно попасть в контейнер Docker:

```
docker exec -it airflow /bin/bash
```

После этого вы получаете терминал Bash в контейнере и можете просматривать изображения в каталоге /tmp/images (рис. 2.12).

Рис. 2.10 Параметры всплывающего окна задачи

```
*** Reading local file: /opt/airflow/logs/download_rocket_launches/notify/2020-12-17T21:15:02.613390+00:00/1.log
[2020-12-17 21:15:30,917] {taskinstance.py:826} INFO - Dependencies all met for <TaskInstance: download_rocket_launches.notify 2020-12-17T2
[2020-12-17 21:15:30,923] {taskinstance.py:826} INFO - Dependencies all met for <TaskInstance: download_rocket_launches.notify 2020-12-17T2
[2020-12-17 21:15:30,923] {taskinstance.py:1017} INFO -
--------------------------------------------------------------------------------
[2020-12-17 21:15:30,923] {taskinstance.py:1018} INFO - Starting attempt 1 of 1
[2020-12-17 21:15:30,923] {taskinstance.py:1019} INFO -
--------------------------------------------------------------------------------
[2020-12-17 21:15:30,931] {taskinstance.py:1038} INFO - Executing <Task(BashOperator): notify> on 2020-12-17T21:15:02.613390+00:00
[2020-12-17 21:15:30,933] {standard_task_runner.py:51} INFO - Started process 1483 to run task
[2020-12-17 21:15:30,937] {standard_task_runner.py:75} INFO - Running: ['airflow', 'tasks', 'run', 'download_rocket_launches', 'notify', '2
[2020-12-17 21:15:30,938] {standard_task_runner.py:76} INFO - Job 6: Subtask notify
[2020-12-17 21:15:30,969] {logging_mixin.py:103} INFO - Running <TaskInstance: download_rocket_launches.notify 2020-12-17T21:15:02.613390+0
[2020-12-17 21:15:30,993] {taskinstance.py:1230} INFO - Exporting the following env vars:
AIRFLOW_CTX_DAG_OWNER=airflow
AIRFLOW_CTX_DAG_ID=download_rocket_launches
AIRFLOW_CTX_TASK_ID=notify
AIRFLOW_CTX_EXECUTION_DATE=2020-12-17T21:15:02.613390+00:00
AIRFLOW_CTX_DAG_RUN_ID=manual__2020-12-17T21:15:02.613390+00:00
[2020-12-17 21:15:30,994] {bash.py:135} INFO - Tmp dir root location:
 /tmp
[2020-12-17 21:15:30,994] {bash.py:158} INFO - Running command: echo "There are now $(ls /tmp/images/ | wc -l) images."
[2020-12-17 21:15:31,002] {bash.py:169} INFO - Output:
[2020-12-17 21:15:31,006] {bash.py:173} INFO - There are now 2 images.
[2020-12-17 21:15:31,006] {bash.py:177} INFO - Command exited with return code 0
[2020-12-17 21:15:31,021] {taskinstance.py:1135} INFO - Marking task as SUCCESS. dag_id=download_rocket_launches, task_id=notify, execution
[2020-12-17 21:15:31,037] {taskinstance.py:1195} INFO - 0 downstream tasks scheduled from follow-on schedule check
[2020-12-17 21:15:31,070] {local_task_job.py:118} INFO - Task exited with return code 0
```

Рис. 2.11 Просмотр журналов

Рис. 2.12 Полученные изображения ракет

2.4 Запуск через равные промежутки времени

Поклонник ракет Джон теперь счастлив, что у него есть рабочий процесс в Airflow, который он может запускать время от времени, чтобы собирать последние изображения ракет. Он может увидеть состояние своего рабочего процесса в пользовательском интерфейсе Airflow, что уже является улучшением по сравнению со сценарием в командной строке, который он запускал раньше. Но ему по-прежнему нужно периодически запускать свой рабочий процесс вручную, а ведь его можно автоматизировать. В конце концов, никому не нравится выполнять повторяющиеся задачи, которые компьютеры умеют делать сами.

В Airflow можно запланировать запуск ОАГ через определенные промежутки времени, например один раз в час, день или месяц. Это можно контролировать, если задать аргумент `schedule_interval`.

Листинг 2.11 Запуск ОАГ раз в день

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

Псевдоним Airflow для 0 0 * * * (т. е. полночь)

Если задать для аргумента `schedule_interval` значение `@daily`, Airflow будет запускать этот рабочий процесс один раз в день, чтобы Джону не приходилось запускать его вручную. Такое поведение лучше всего видно в древовидном представлении, как показано на рис. 2.13.

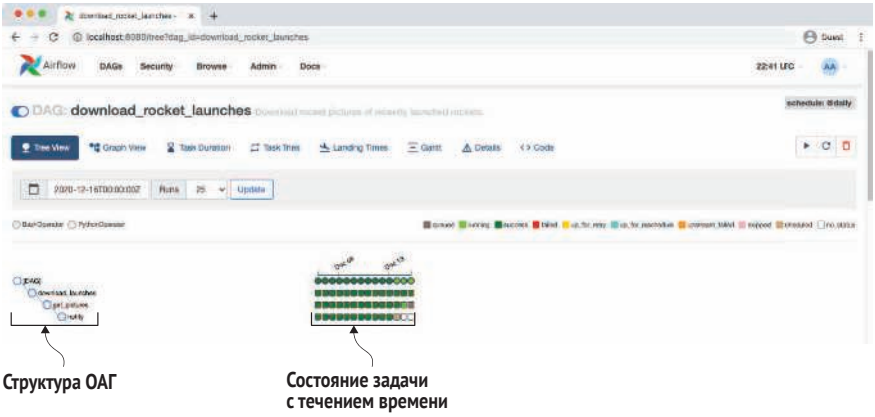


Рис. 2.13 Древовидное представление Airflow

Древовидное представление похоже на графовое, но отображает структуру графа в динамике. Обзор состояния всех запусков одного рабочего процесса можно увидеть на рис. 2.14.

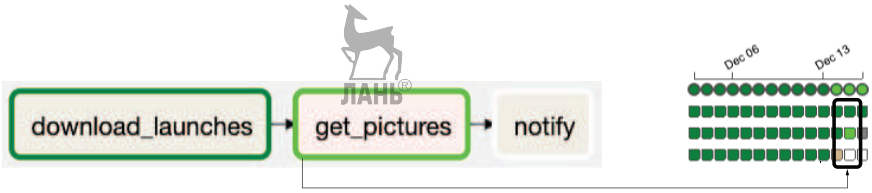


Рис. 2.14 Связь между графовым и древовидным представлениями

Структура ОАГ отображается в соответствии с макетом «строки и столбцы», в частности состояние всех запусков конкретного ОАГ, где каждый столбец представляет собой один запуск в определенный момент времени.

Когда мы задали для `schedule_interval` значение `@daily`, Airflow знал, что должен запускать этот ОАГ один раз в день. Учитывая `start_date`, предоставленный ОАГ 14 дней назад, это означает, что время с момента 14 дней назад до настоящего момента можно разделить на 14 равных интервалов одного дня. Поскольку и дата начала, и дата окончания этих 14 интервалов находятся в прошлом, они начнут выполняться, как только мы предоставим программе Airflow аргумент `schedule_interval`. Семантика этого аргумента и различные способы его настройки более подробно описаны в главе 3.

2.5 Обработка неудачных задач

До сих пор мы видели в пользовательском интерфейсе Airflow только зеленый цвет. Но что будет, если что-то не получится? Задачи нередко терпят неудачу, и причин тому множество (например, внешняя служба не работает, проблемы с сетевым подключением или неисправный диск).

Скажем, например, в какой-то момент у вас случился сбой в сети при получении изображений ракет Джона. Как следствие задача Airflow дает сбой, и мы видим неудачную задачу в пользовательском интерфейсе Airflow. На рис. 2.15 показано, как это выглядело бы.

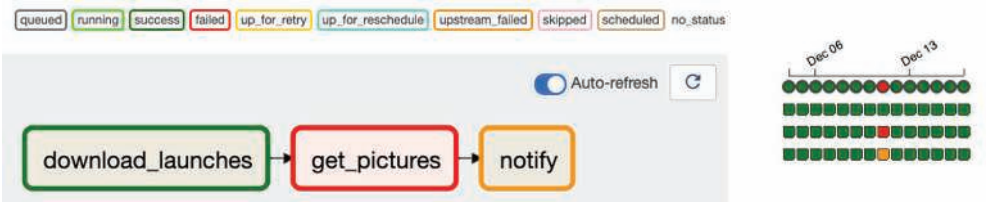


Рис. 2.15 Сбой отображается в графовом и древовидном представлениях

Конкретная неудавшаяся задача будет отображаться красным цветом как в графовом, так и в древовидном представлениях как результат невозможности получить изображения из интернета, что, следовательно, вызовет ошибку. Последующая задача `notify` не будет выполняться вообще, потому что она зависит от успешного состояния задачи `get_pictures`. Такие экземпляры задач отображаются оранжевым цветом. По умолчанию все предыдущие задачи должны выполняться успешно, и никакие последующие задачи в случае неудачи выполняться не будут.

Выясним, в чем проблема, еще раз проверив журналы. Откройте журналы задачи `get_ pictures` (рис. 2.16).

В трассировке стека мы обнаруживаем потенциальную причину проблемы:

```
urllib3.exceptions.NewConnectionError: <urllib3.connection.HTTPSConnection
    object at 0x7f37963ce3a0>: Failed to establish a new connection: [Errno
    -2] Name or service not known
```

Это указывает на то, что urllib3 (т. е. HTTP-клиент для Python) пытается установить соединение, но не может этого сделать. Возможно, дело в правиле брандмауэра, блокирующем соединение, или отсутствии подключения к интернету. Предполагая, что мы устранили проблему (например, подключили интернет-кабель), перезапустим задачу.

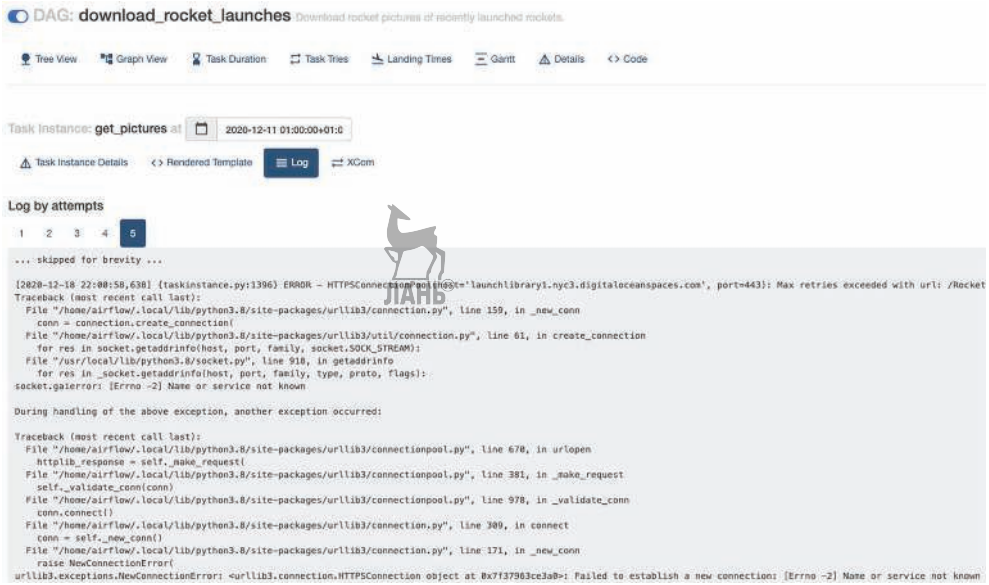


Рис. 2.16 Трассировка стека неудавшейся задачи get_pictures

ПРИМЕЧАНИЕ Нет необходимости перезапускать весь рабочий процесс. Приятная особенность Airflow состоит в том, что вы можете выполнить перезапуск с момента сбоя и далее, без необходимости перезапуска каких бы то ни было ранее выполненных задач.

Щелкните по невыполненной задаче, а затем нажмите кнопку **Clear** (Очистить) во всплывающем окне (рис. 2.17). Вы увидите задачи, которые хотите очистить, то есть вы сбросите состояние этих задач, а Airflow перезапустит их, как показано на рис. 2.18.

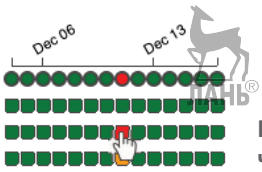


Рис. 2.17 Щелкните по невыполненной задаче, чтобы просмотреть параметры для ее очистки

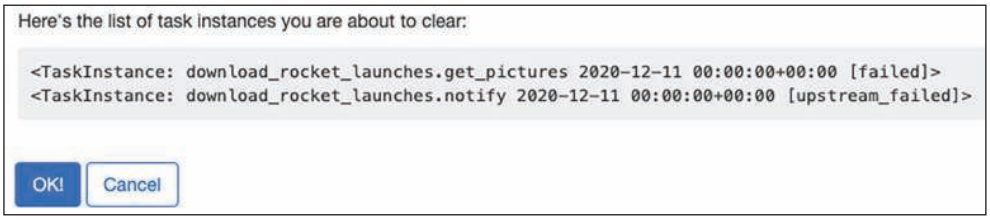


Рис. 2.18 Очистка состояния задачи get_pictures и последующих задач

Щелкните по кнопке **OK!** – и неудавшаяся задача и ее последующие задачи будут очищены, как показано на рис. 2.19.

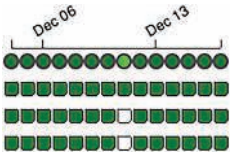


Рис. 2.19 Очищенные задачи, отображаемые в графовом представлении

Предполагая, что проблемы с подключением решены, задачи теперь будут выполняться успешно, и все древовидное представление станет зеленым (рис. 2.20).

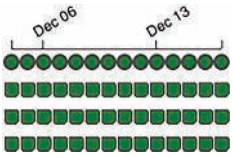


Рис. 2.20 Успешно выполненные задачи после очистки неудавшихся задач

В любом программном обеспечении есть много причин для сбоя. В рабочих процессах Airflow сбой иногда допускается, иногда – нет, а иногда только при определенных условиях. Критерии устранения сбоев можно настроить на любом уровне рабочего процесса. Более подробно они рассматриваются в главе 4.

После очистки невыполненных задач Airflow автоматически перезапустит их. Если все пойдет хорошо, Джон загрузит изображения ракет, полученные в результате неудачного выполнения задач. Обратите внимание, что вызываемый URL-адрес в задаче `download_launches` просто запрашивает следующие запуски ракеты – это означает, что он вернет следующие запуски ракеты во время вызова API. Включение контекста времени выполнения, в котором ОАГ был запущен в код, рассматривается в главе 4.

Резюме

- Рабочие процессы в Airflow представлены в виде ОАГ.
- Операторы представляют собой одну единицу работы.
- Airflow содержит набор операторов как для универсальной, так и для конкретной работы.
- Пользовательский интерфейс Airflow предлагает графовое представление для просмотра структуры ОАГ и древовидное представление для просмотра выполнения ОАГ с течением времени.
- Неудачные задачи можно перезапустить в любом месте ОАГ.

3 Планирование в Airflow

Эта глава рассказывает о:

- запуске ОАГ через равные промежутки времени;
- создании динамических ОАГ для постепенной обработки данных;
- загрузке и повторной обработке наборов архивных данных с помощью обратного заполнения;
- применении лучших практик для надежных задач.

В предыдущей главе мы изучили пользовательский интерфейс Airflow и показали, как определить базовый ОАГ Airflow и запускать его каждый день, определив аргумент `schedule_inteval`. В этой главе мы углубимся в концепцию планирования в Airflow и изучим, как она позволяет обрабатывать данные постепенно и через равные промежутки времени. Вначале мы представим небольшой пример, посвященный анализу пользовательских событий на нашем сайте, и узнаем, как создать ОАГ для регулярного анализа этих событий. Далее мы рассмотрим способы сделать этот процесс более эффективным, применив поэтапный подход к анализу данных и пониманию того, как это связано с концепцией дат выполнения в Airflow. Наконец, мы покажем, как работать с наборами архивных данных, используя обратное заполнение, и обсудим некоторые важные свойства правильных задач Airflow.

3.1 Пример: обработка пользовательских событий

Чтобы понять, как работает планирование в Airflow, сначала рассмотрим небольшой пример. Представьте, что у нас есть служба, которая отслеживает поведение пользователей на нашем сайте и позволяет нам анализировать, к каким страницам обращались пользователи (идентифицированные по IP-адресу). В целях маркетинга мы бы хотели знать, каково число страниц, к которым обращаются пользователи, и сколько времени они тратят во время каждого посещения. Чтобы получить представление о том, как это поведение меняется с течением времени, нам нужно рассчитывать эту статистику ежедневно, поскольку это позволяет сравнивать изменения в разные дни и более длительные периоды времени.

Из практических соображений внешняя служба отслеживания не хранит данные более 30 дней, поэтому нам нужно хранить и аккумулировать эти данные самостоятельно, так как нам нужно хранить историю в течение более длительных периодов времени. Обычно, поскольку необработанные данные могут быть довольно большими, имеет смысл хранить их в облачной службе хранения, такой как Amazon S3 или Google Cloud Storage, ибо они сочетают в себе высокую надежность и относительно низкую стоимость. Однако чтобы было проще, мы не будем заниматься этим и будем хранить наши данные локально.

Чтобы смоделировать этот пример, мы создали простой (локальный) API, который позволяет нам получать пользовательские события. Например, мы можем получить полный список доступных событий последних 30 дней с помощью следующего вызова API:

```
curl -o /tmp/events.json http://localhost:5000/events
```

Этот вызов возвращает список пользовательских событий (в кодировке JSON), которые мы можем проанализировать, чтобы вычислить статистику по пользователям.

Используя этот API, можно разбить наш рабочий процесс на две отдельные задачи: одна для получения пользовательских событий, а другая для расчета статистики. Сами данные можно скачать, используя `BashOperator`, как было показано в предыдущей главе. Для расчета статистики можно использовать `PythonOperator`, который позволяет нам загружать данные в Pandas DataFrame и рассчитывать количество событий, используя *группировку* и *агрегацию*. Все вместе это дает нам ОАГ, показанный в листинге 3.1.

Листинг 3.1 Начальная (без планирования) версия ОАГ события (dags/01_unscheduled.py)

```
import datetime as dt
from pathlib import Path
```

```
import pandas as pd
from airflow import DAG
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator

dag = DAG(
   dag_id="01_unscheduled",
   start_date=dt.datetime(2019, 1, 1),
   schedule_interval=None,
)
fetch_events = BashOperator(
   task_id="fetch_events",
   bash_command=(
      "mkdir -p /data && "
      "curl -o /data/events.json "
      "https://localhost:5000/events"
   ),
   dag=dag,
)
```

Определяем дату запуска OAГ

Указываем, что это версия OAГ без планирования

Извлекаем и сохраняем события из API

```
def _calculate_stats(input_path, output_path):
   """Рассчитывает статистику по событиям"""

   events = pd.read_json(input_path)
   stats = events.groupby(["date", "user"]).size().reset_index()
   Path(output_path).parent.mkdir(exist_ok=True)
   stats.to_csv(output_path, index=False)


calculate_stats = PythonOperator(
   task_id="calculate_stats",
   python_callable=_calculate_stats,
   op_kwargs={
       "input_path": "/data/events.json",
       "output_path": "/data/stats.csv",
   },
   dag=dag,
)
```

Загружаем события и рассчитываем необходимую статистику

Убеждаемся, что выходной каталог существует, и пишем результаты в файл с расширением CSV

```
fetch_events >> calculate_stats
```

Задаем порядок выполнения

Теперь у нас есть базовый OAГ, но нам все еще нужно убедиться, что Airflow регулярно запускает его. Давайте перейдем к планированию, чтобы у нас были ежедневные обновления!

3.2 Запуск через равные промежутки времени

Как было показано в главе 2, OAГ Airflow можно запускать через равные промежутки времени, задав для него запланированный интервал с помощью аргумента `schedule_interval` при инициализации OAГ. По

умолчанию значение этого аргумента равно None. Это означает, что ОАГ не будет запланирован и будет запускаться только при запуске вручную из пользовательского интерфейса или API.

3.2.1 Определение интервалов

В нашем примере мы хотели бы вычислять статистику ежедневно, поэтому имеет смысл запланировать запуск нашего ОАГ один раз в день. Поскольку это распространенный пример, Airflow предоставляет удобный макрос `@daily` для определения интервала, при котором наш ОАГ будет запускаться каждый день в полночь.

Листинг 3.2 Определение ежедневного интервала (dags/02_daily_schedule.py)

```
dag = DAG(
    dag_id="02_daily_schedule",
    schedule_interval="@daily",
    start_date=dt.datetime(2019, 1, 1),
...
)
```

Планируем запуск ОАГ каждый день в полночь

Дата и время начала планирования запусков ОАГ

Airflow также нужно знать, когда мы хотим начать выполнение ОАГ, указав дату запуска. Исходя из этой даты, Airflow запланирует первое выполнение нашего ОАГ, чтобы запустить его в первом интервале *после* даты начала (начало + интервал). Последующие запуски будут выполняться с заданными интервалами после первого интервала.

> **ПРИМЕЧАНИЕ** Обратите внимание на то, что 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.

Например, предположим, что мы определяем наш ОАГ с датой начала первого января, как было показано ранее в листинге 3.2. В сочетании с ежедневным интервалом это приведет к тому, что Airflow будет запускать наш ОАГ в полночь каждого дня после первого января (рис. 3.1). Обратите внимание, что первое выполнение происходит второго января (первый интервал после даты начала), а не первого. Мы рассмотрим аргументацию данного поведения далее в этой главе.

Без даты окончания Airflow (в принципе) будет выполнять наш ОАГ по этому ежедневному расписанию до скончания времен. Однако если мы уже знаем, что наш проект имеет фиксированную продолжительность, то можно сообщить Airflow, чтобы он прекратил запуск ОАГ после определенной даты, используя параметр `end_date`.

Рис. 3.1 Интервалы для ОАГ, запускаемого по расписанию ежедневно, с заданной датой запуска (2019-01-01). Стрелки указывают момент времени, в который выполняется ОАГ. Если дата окончания не указана, ОАГ будет выполняться каждый день до тех пор, пока не будет отключен

Листинг 3.3 Определение даты окончания ОАГ (dags/03_with_end_date.py)

```
dag = DAG(
    dag_id="03_with_end_date",
    schedule_interval="@daily",
    start_date=dt.datetime(year=2019, month=1, day=1),
    end_date=dt.datetime(year=2019, month=1, day=5),
)
```

В результате получится полный набор интервалов, показанный на рис. 3.2.

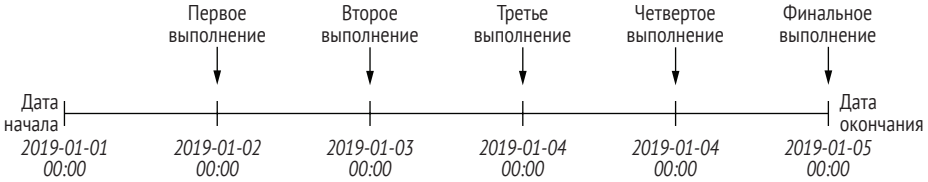


Рис. 3.2 Интервалы для ОАГ, запускаемого по расписанию ежедневно, с указанными датами начала (2019-01-01) и окончания (2019-01-05). Это не дает ОАГ продолжить свое выполнение после этой даты

3.2.2 Интервалы на основе Cron

До сих пор все наши примеры демонстрировали, что ОАГ запускаются с дневными интервалами. Но что, если нам нужно выполнять задания с почасовыми или еженедельными интервалами? А как насчет более сложных интервалов, когда нам нужно запускать ОАГ в 23:45 каждую субботу?

Для поддержки более сложных вариантов Airflow позволяет определять интервалы, используя тот же синтаксис, что и у cron, планировщика заданий на основе времени, используемого Unix-подобными компьютерными операционными системами, такими как macOS и Linux. Этот синтаксис состоит из пяти компонентов и определяется следующим образом:

```
# ┌──────── минута (0-59)
# │ ┌────── час (0-23)
# │ │ ┌──── день месяца (1-31)
# │ │ │ ┌── месяц (1-12)
# │ │ │ │ ┌ день недели (0-6) (с воскресенья по субботу;
# │ │ │ │ │ в некоторых системах 7 - это также воскресенье)
# * * * * *
```

В этом определении задание cron выполняется, когда поля для уточнения времени и даты соответствуют текущему системному времени и дате. Символы звездочки (*) можно использовать вместо чисел для определения неограниченных полей. Это означает, что нас не волнует значение этого поля.

Хотя такое представление на базе cron может показаться немного запутанным, оно предоставляет значительную гибкость для определения временных интервалов. Например, мы можем определить часовые, дневные и недельные интервалы, используя следующие выражения cron:

- 0 * * * * = ежечасно (запуск по часам);
- 0 0 * * * = ежедневно (запуск в полночь);
- 0 0 * * 0 = еженедельно (запуск в полночь в воскресенье).

Помимо этого, также можно определять более сложные выражения, например:

- 0 0 1 * * = полночь первого числа каждого месяца;
- 45 23 * * SAT = 23:45 каждую субботу.

Кроме того, выражения cron позволяют определять коллекции значений с помощью запятой (,) для определения списка значений или тире (-) для определения диапазона значений. Используя этот синтаксис, мы можем создавать выражения, позволяющие запускать задания для нескольких рабочих дней или наборов часов в течение дня:

- 0 0 * * MON, WED, FRI = запускать каждый понедельник, среду, пятницу в полночь;
- 0 0 * * MON-FRI = запускать каждый будний день в полночь;
- 0 0,12 * * * = запускать каждый день в 00:00 и 12:00.

Airflow также обеспечивает поддержку макросов, которые представляют собой сокращенный вариант часто используемых интервалов. Мы уже видели один из этих макросов (`@daily`) для определения дневных интервалов. Обзор других макросов, поддерживаемых Airflow, приведен в табл. 3.1.

Хотя выражения Cron чрезвычайно эффективны, работать с ними может быть непросто. Таким образом, возможно, будет полезно протестировать свое выражение, прежде чем опробовать его в Airflow. К счастью, в интернете доступно множество инструментов[1], которые

[1] https://crontab.guru переводит выражения cron на понятный человеку язык.

могут помочь вам определить, проверить или объяснить выражения Cron простым человеческим языком. Также не помешает задокументировать рассуждения о сложных выражениях cron в своем коде. Это может помочь другим (включая и вас в будущем!) понять данное выражение при повторном обращении к коду.

Таблица 3.1. Макросы Airlfow для часто используемых интервалов планирования

Макрос	Значение
`@once`	Один и только один раз
`@hourly`	Запуск один раз в час в начале часа
`@daily`	Запуск один раз в день в полночь
`@weekly`	Запуск один раз в неделю в полночь в воскресенье утром
`@monthly`	Запуск один раз в месяц в полночь первого числа месяца
`@yearly`	Запуск один раз в год в полночь 1 января

3.2.3 Частотные интервалы

Важное ограничение выражений cron состоит в том, что они не могут представлять определенные расписания на основе частот. Например, как бы вы определили выражение cron, которое запускает ОАГ каждые три дня? Оказывается, что можно было бы написать выражение, которое запускается каждый первый, четвертый, седьмой и так далее день месяца, но такой подход столкнется с проблемами в конце месяца, поскольку ОАГ будет последовательно запускаться и 31-го, и 1-го числа следующего месяца, нарушая желаемый график.

Данное ограничение проистекает из природы выражений cron, поскольку они определяют шаблон, который постоянно сопоставляется с текущим временем, чтобы определить, следует ли выполнять задание. Это дает преимущество, при котором выражения не имеют состояния, а это означает, что вам не нужно помнить, когда было выполнено предыдущее задание для вычисления последующего интервала. Однако, как видите, при этом страдают выразительные возможности.

Что, если мы действительно хотим запускать наш ОАГ раз в три дня? Для поддержки такого типа расписания на базе частот Airflow также позволяет определять интервалы в виде относительного временного интервала. Чтобы использовать такое расписание, можно передать экземпляр `timedelta` (из модуля datetime в стандартной библиотеке) в качестве интервала.

Листинг 3.4 Определение интервала на базе частоты (dags/04_time_delta.py)

```
dag = DAG(
    dag_id="04_time_delta",
    schedule_interval=dt.timedelta(days=3),
```

timedelta дает возможность использовать расписания на базе частоты

```
    start_date=dt.datetime(year=2019, month=1, day=1),
    end_date=dt.datetime(year=2019, month=1, day=5),
)
```

Это приведет к тому, что наш ОАГ будет запускаться каждые три дня после даты начала (4, 7, 10 и т. д. января 2019 г.). Конечно, вы также можете использовать данный подход для запуска ОАГ каждые 10 минут (используя `timedelta(minutes=10)`) или каждые два часа (`timedelta(hours=2)`).

3.3 Инкрементная обработка данных

Хотя теперь у нас есть ОАГ, запускающийся с ежедневным интервалом (при условии что мы придерживаемся расписания, заданного макросом `@daily`), мы пока не достигли своей цели. Например, наш ОАГ скачивает и вычисляет статистику для всего каталога пользовательских событий ежедневно, что вряд ли эффективно. Более того, этот процесс скачивает события только за последние 30 дней, а это означает, что у нас нет истории для более ранних дат.

3.3.1 Инкрементное извлечение событий

Одним из способов решения этих проблем является изменение ОАГ для инкрементной загрузки данных, когда мы загружаем только события соответствующего дня в каждом интервале расписания и рассчитываем статистику лишь для новых событий (рис. 3.3).

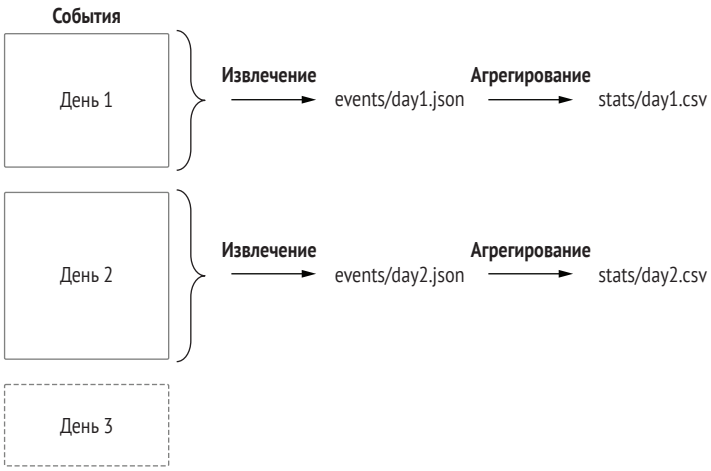


Рис. 3.3 Инкрементное извлечение и обработка данных

Такой подход намного более эффективен, нежели извлечение и обработка всего набора данных, поскольку он значительно сокращает

объем данных, которые необходимо обрабатывать в каждом интервале. Кроме того, поскольку теперь мы храним наши данные в отдельных файлах по дням, у нас также есть возможность приступить к созданию истории файлов, намного превышая 30-дневный лимит нашего API.

Чтобы внедрить инкрементную обработку в наш рабочий процесс, нужно изменить ОАГ, чтобы скачивать данные за определенный день. К счастью, мы можем настроить вызов API для извлечения событий на текущую дату, включив параметры начальной и конечной дат:

```
curl -O http://localhost:5000/events?start_date=2019-01-01&end_date=2019-01-02
```

Эти параметры указывают временной диапазон, для которого мы хотели бы получать события. Обратите внимание, что в этом примере `start_date` является инклюзивным параметром, а `end_date` – эксклюзивным. Это означает, что, по сути, мы получаем события, которые происходят в период между 01.01.2019 00:00:00 и 2019-01-01 23:59:59.

Мы можем реализовать это инкрементное получение данных в ОАГ, изменив bash-команду, чтобы включить в нее две даты.

Листинг 3.5 Получение событий для определенного временного интервала (dags/05_query_with_dates.py)

```
fetch_events = BashOperator(
    task_id="fetch_events",
    bash_command=(
        "mkdir -p /data && "
        "curl -o /data/events.json "
        "http://localhost:5000/events?"
        "start_date=2019-01-01&"
        "end_date=2019-01-02"
    ),
    dag=dag,
)
```

Однако для получения данных на любую дату, отличную от 01.01.2019, нужно изменить команду, чтобы использовать даты начала и окончания, которые отражают день, для которого выполняется ОАГ. К счастью, Airflow предоставляет для этого несколько дополнительных параметров, которые мы рассмотрим в следующем разделе.

3.3.2 Динамическая привязка ко времени с использованием дат выполнения

Для многих рабочих процессов, включающих временные процессы, важно знать, в течение какого временного интервала выполняется данная задача. По этой причине Airflow предоставляет задачам дополнительные параметры, которые можно использовать, чтобы опре-

делить, в каком интервале выполняется задача (более подробно об этих параметрах мы поговорим в следующей главе).

Самый важный из этих параметров – `execution_date`, который обозначает дату и время, в течение которых выполняется ОАГ. Вопреки тому, что предполагает имя параметра, `execution_date` – это не дата, а временная метка, отражающая время начала интервала, для которого выполняется ОАГ. Время окончания интервала указывается другим параметром, `next_execution_date`. Вместе эти даты определяют всю продолжительность интервала задачи (рис. 3.4).

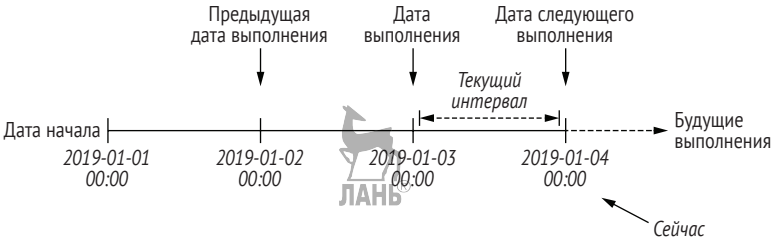


Рис. 3.4 Даты выполнения в Airflow

Airflow также предоставляет параметр `previous_execution_date`, который описывает начало предыдущего интервала. Хотя здесь мы не будем использовать этот параметр, он может быть полезен для выполнения анализа, который сравнивает данные текущего временного интервала с результатами предыдущего интервала.

В Airflow эти даты выполнения можно использовать, ссылаясь на них в операторах. Например, в `BashOperator` можно использовать функцию создания шаблонов, чтобы динамически включать даты выполнения в Bash-команду. Подробнее о создании шаблонов рассказывается в главе 4.

Листинг 3.6 Использование создания шаблонов для указания дат (dags/06_templated_query.py)

```
fetch_events = BashOperator(
    task_id="fetch_events",
    bash_command=(
          "mkdir -p /data && "
          "curl -o /data/events.json "
          "http://localhost:5000/events?"
          "start_date={{execution_date.strftime('%Y-%m-%d')}}"
          "&end_date={{next_execution_date.strftime('%Y-%m-%d')}}"
    ),
    dag=dag,
)
```

Отформатированная дата выполнения, вставленная с использованием шаблонизатора Jinja

next_execution_date содержит дату выполнения следующего интервала

В этом примере синтаксис `{{variable_name}}` представляет собой пример использования (https://jinja.palletsprojects.com/en/3.0.x/) синтаксиса шаблонов на базе Jinja для ссылки на один из специфических

параметров Airflow. Здесь мы используем этот синтаксис, чтобы ссылаться на даты выполнения и форматировать их в ожидаемом строковом формате с помощью метода datetime strftime (так как оба варианта выполнения даты являются объектами datetime).

Поскольку параметры `execute_date` часто используются таким образом, чтобы ссылаться на даты как на форматированные строки, Airflow также предоставляет несколько сокращенных параметров для распространенных форматов дат. Например, параметры `ds` и `ds_nodash` представляют собой разные обозначения `execution_date` в формате ГГГГ-ММ-ДД и ГГГГММДД соответственно. Аналогично `next_ds`, `next_ds_nodash`, `prev_ds` и `prev_ds_nodash` предоставляют сокращенные обозначения для следующей и предыдущей дат выполнения соответственно.

Используя эти сокращения, мы также можем написать нашу команду инкрементального извлечения данных следующим образом.

Листинг 3.7 Использование сокращенного шаблона (dags/07_templated_query_ds.py)

```
fetch_events = BashOperator(
   task_id="fetch_events",
   bash_command=(
     "mkdir -p /data && "
     "curl -o /data/events.json "
     "http://localhost:5000/events?"
     "start_date={{ds}}&"
     "end_date={{next_ds}}"
   ),
   dag=dag,
)
```

ds предоставляет дату выполнения в формате ГГГГММ-ДД

next_ds предоставляет то же самое для next_ execution_date

Эту более короткую версию легче читать. Однако для более сложных форматов даты (или времени и даты) вам, вероятно, все равно придется использовать более гибкий метод strftime.

3.3.3 Разделение данных

Хотя наша новая задача `fetch_events` теперь извлекает события постепенно для каждого нового интервала расписания, проницательный читатель, возможно, заметил, что каждая новая задача просто перезаписывает результат предыдущего дня, а это означает, что фактически мы не создаем никакой истории.

Один из способов решить эту проблему – просто добавить новые события в файл events.json, что позволит нам создать свою историю в одном файле JSON. Однако у такого подхода есть недостаток: он требует, чтобы все нижестоящие задания по обработке загружали весь набор данных, даже если нас интересует только вычисление статистики за данный день. Кроме того, такой файл становится единой точкой отказа, из-за чего мы рискуем потерять весь наш набор данных, если этот файл будет потерян или поврежден.

Есть альтернативный подход – разделить набор данных на ежедневные пакеты, записав вывод задачи в файл с именем соответствующей даты выполнения.

Листинг 3.8 Запись данных о событиях в отдельные файлы по дате (dags/08_templated_path.py)

```
fetch_events = BashOperator(
    task_id="fetch_events",
    bash_command=(
        "mkdir -p /data/events && "
        "curl -o /data/events/{{ds}}.json "
        "http://localhost:5000/events?"
        "start_date={{ds}}&"
        "end_date={{next_ds}}",
    dag=dag,
)
```

Ответ, записанный в шаблонное имя файла

Это приведет к тому, что все данные, скачиваемые с датой выполнения 01.01.2019, будут записаны в файл /data/events/2019-01-01.json.

Такая практика разделения набора данных на более мелкие и управляемые части является распространенной стратегией в системах хранения и обработки данных и обычно называется *секционированием*, когда более мелкие фрагменты данных устанавливают *секции*. Преимущество разделения нашего набора данных по дате выполнения становится очевидным, если мы рассмотрим вторую задачу в нашем ОАГ (`calculate_stats`), в которой мы рассчитываем статистику для пользовательских событий за каждый день. В нашей предыдущей реализации мы загружали весь набор данных и вычисляли статистику для всей истории событий каждый день.

Листинг 3.9 Предыдущая реализация для статистики событий (dags/01_scheduled.py)

```
def _calculate_stats(input_path, output_path):
    """Рассчитываем статистику событий."""
    Path(output_path).parent.mkdir(exist_ok=True)
    events = pd.read_json(input_path)
    stats = events.groupby(["date", "user"]).size().reset_index()
    stats.to_csv(output_path, index=False)
calculate_stats = PythonOperator(
    task_id="calculate_stats",
    python_callable=_calculate_stats,
    op_kwargs={
        "input_path": "/data/events.json",
        "output_path": "/data/stats.csv",
    },
    dag=dag,
)
```

Однако, используя наш секционированный набор данных, мы можем более эффективно рассчитать эту статистику для каждой отдельной секции, изменив пути ввода и вывода этой задачи, чтобы указать на секционированные данные событий и секционированный выходной файл.

Листинг 3.10 Вычисление статистики за каждый интервал выполнения (dags/08_templated_path.py)

```
def _calculate_stats(**context):
    """Расчет статистики."""
    input_path = context["templates_dict"]["input_path"]
    output_path = context["templates_dict"]["output_path"]

    Path(output_path).parent.mkdir(exist_ok=True)

    events = pd.read_json(input_path)
    stats = events.groupby(["date", "user"]).size().reset_index()
    stats.to_csv(output_path, index=False)


calculate_stats = PythonOperator(
    task_id="calculate_stats",
    python_callable=_calculate_stats,
    templates_dict={
        "input_path": "/data/events/{{ds}}.json",
        "output_path": "/data/stats/{{ds}}.csv",
    },
    dag=dag,
)
```

Получаем все контекстные переменные в этом словаре

Получаем шаблонные значения из объекта templates_dict

Передаем значения, которые мы хотим использовать в качестве шаблона

Хотя эти изменения могут показаться несколько сложными, они в основном связаны с шаблонным кодом, обеспечивающим шаблонизацию наших входных и выходных путей. Чтобы реализовать это в PythonOperator, нужно передать все аргументы, которые должны быть шаблонизированы с помощью параметра оператора `templates_dict`. Затем мы можем получить шаблонные значения внутри нашей функции из контекстного объекта, который Airflow[1] передает нашей функции `_calculate_stats`.

Если все это для вас слишком быстро, не волнуйтесь; мы подробнее рассмотрим контекст задачи в следующей главе. Здесь важно понять, что эти изменения позволяют нам вычислять статистику постепенно, обрабатывая только небольшие подмножества данных каждый день.

[1] Для Airflow версии 1.10.x необходимо передать дополнительный аргумент `provide_context=True` в `PythonOperator`; в противном случае функция `_calculate_stats` не получит контекстные значения.

3.4 Даты выполнения

Поскольку даты выполнения являются очень важной частью Airflow, уделим им минуту, чтобы убедиться, что мы полностью понимаем, как они определяются.

3.4.1 Выполнение работы с фиксированными интервалами

Как мы уже видели, мы можем контролировать, когда Airflow запускает ОАГ, с помощью трех параметров: даты начала, интервала и (необязательно) даты окончания. Чтобы фактически начать планирование ОАГ, Airflow использует эти три параметра, дабы разделить время на ряд интервалов, начиная с заданной даты начала и, необязательно, заканчивая датой окончания (рис. 3.5).

Рис. 3.5 Время, представленное в виде интервалов планирования Airflow. Предполагает дневной интервал с датой начала 01.01.2019

В этом представлении ОАГ выполняется для заданного интервала, как только тайм-слот этого интервала будет пройден. Например, первый интервал на рис. 3.5 будет выполнен как можно скорее после 2019-01-01 23:59:59, потому что к этому моменту последняя временная точка в интервале пройдет. Точно так же ОАГ будет выполняться для второго интервала вскоре после 2019-01-02 23:59:59 и т. д., пока мы не достигнем необязательной даты окончания.

Преимущество использования подхода на основе интервалов состоит в том, что он идеально подходит для выполнения инкрементной обработки данных, которая была показана в предыдущих разделах, поскольку мы точно знаем, в течение какого интервала времени выполняется задача, – начало и конец соответствующего интервала. Это резко контрастирует, например, с такой системой планирования, как cron, где нам известно только текущее время, в течение которого выполняется наша задача. Это означает, что, например, в cron мы должны либо вычислить, либо угадать, где остановилось предыдущее выполнение, предполагая, что задача выполняется за предыдущий день (рис. 3.6).

Понимание того, что обработка времени в Airflow строится вокруг интервалов, также помогает понять, как определяются даты выполнения в Airflow. Например, предположим, что у нас есть ОАГ, который следует ежедневному интервалу, а затем учитывает соответствующий интервал, в котором должны обрабатываться данные за 03.01.2019.

В Airflow этот интервал будет запускаться вскоре после 2019-01-04 00:00:00, потому что в данный момент мы знаем, что больше не будем получать новые данные за 2019-01-03. Если вернуться к объяснению использования дат выполнения в наших задачах из предыдущего раздела, то как вы думаете, каким будет значение `execution_date` для этого интервала?

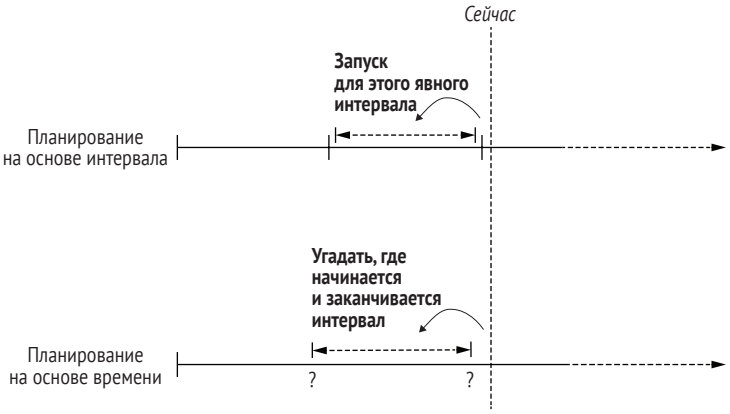


Рис. 3.6 Инкрементальная обработка в окнах планирования на базе интервалов (например, Airflow) в сравнении с окнами, полученными из таких систем, как, например, cron. При инкрементальной обработке (данных) время обычно делится на дискретные временные интервалы, которые обрабатываются сразу после прохождения соответствующего интервала. Подходы к планированию на основе интервалов (такие как Airflow) явно планируют выполнение задач для каждого интервала, предоставляя при этом точную информацию для каждой задачи, касающуюся начала и конца интервала. Напротив, при подходах к планированию как у cron задачи выполняются только в заданное время, оставляя на усмотрение самой задачи определение того, в течение какого инкрементного интервала выполняется задача

Многие ожидают, что дата выполнения этого запуска ОАГ будет 2019-01-04, поскольку это момент, когда ОАГ фактически запускается. Однако если посмотреть на значение переменной `execute_date` при выполнении наших задач, то можно увидеть дату выполнения 2019-01-03. Это связано с тем, что Airflow определяет дату выполнения ОАГ как начало соответствующего интервала. Концептуально это имеет смысл, если считать, что дата выполнения отмечает интервал, а не момент фактического выполнения ОАГ. К сожалению, именование может немного сбивать с толку.

Когда даты выполнения Airflow определены как начало соответствующих интервалов, их можно использовать для определения начала и конца определенного интервала (рис. 3.7). Например, при выполнении задачи начало и конец соответствующего интервала определяются параметрами `execution_date` (начало интервала) и `next_execution_date` (начало следующего интервала). Точно так же

предыдущий интервал может быть получен с помощью параметров `previous_execution_date` и `execution_date`.

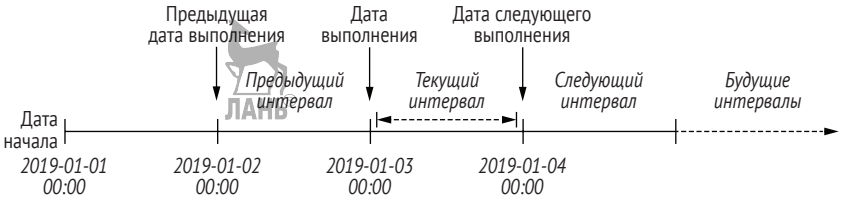


Рис. 3.7 Даты выполнения в контексте интервалов. В Airflow дата выполнения ОАГ определяется как время начала соответствующего интервала, а не время, когда выполняется ОАГ (обычно это конец интервала). Таким образом, значение `execution_date` указывает на начало текущего интервала, в то время как параметры `previous_execution_date` и `next_execution_date` указывают на начало предыдущего и следующего интервалов соответственно. Текущий интервал можно получить из сочетания `execution_date` и `next_execution_date`, что означает начало следующего интервала и, следовательно, конец текущего

Однако следует иметь в виду одну вещь при использовании параметров `previous_execution_date` и `next_execution_date` в своих задачах: они определяются только для запуска ОАГ после интервала. Таким образом, значения этих параметров будут не определены для всех запусков, которые запускаются вручную с помощью пользовательского интерфейса Airflow или интерфейса командной строки, поскольку Airflow не может предоставить информацию о следующих или предыдущих интервалах, если вы не соблюдаете интервал.

3.5 Использование обратного заполнения

Поскольку Airflow позволяет определять интервалы с произвольной даты начала, мы также можем определить интервалы от даты начала в прошлом. Можно использовать это свойство для выполнения запусков ОАГ для загрузки или анализа наборов архивных данных – это процесс, который обычно называют *бэкфиллингом*, или обратным заполнением.

3.5.1 Назад в прошлое

По умолчанию Airflow будет планировать и запускать все прошлые интервалы, которые не были выполнены. Таким образом, указание даты начала в прошлом и активация соответствующего ОАГ даст интервалы, которые прошли до выполнения текущего времени. Такое поведение контролируется параметром ОАГ `catchup`, и его можно отключить, задав для него значение false.

Листинг 3.11 Отключение параметра catchup во избежание выполнения архивных запусков (dags/09_no_catchup.py)

```
dag = DAG(
    dag_id="09_no_catchup",
    schedule_interval="@daily",
    start_date=dt.datetime(year=2019, month=1, day=1),
    end_date=dt.datetime(year=2019, month=1, day=5),
    catchup=False,
)
```

При такой настройке ОАГ будет запускаться только с учетом последнего интервала, вместо того чтобы выполнять все открытые интервалы в прошлом (рис. 3.8). Значением `catchup` по умолчанию можно управлять из конфигурационного файла Airflow, задав значение для параметра конфигурации `catchup_ by_default`.

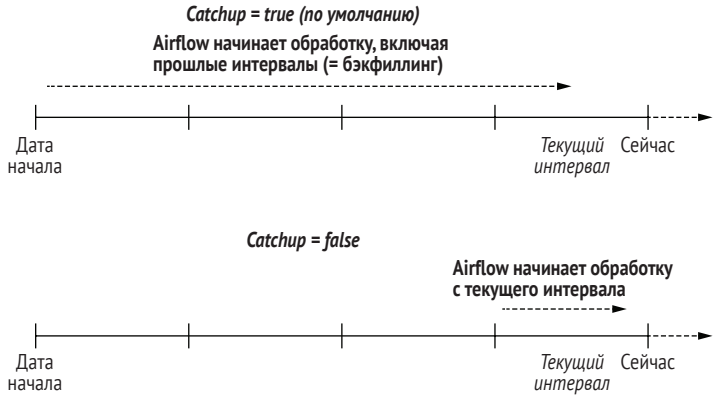


Рис. 3.8 Обратное заполнение в Airflow. По умолчанию Airflow будет запускать задачи для всех прошедших интервалов до текущего времени. Такое поведение можно отключить, задав для параметра `catchup` значение false, и в этом случае Airflow начнет выполнять задачи только с текущего интервала

Хотя обратное заполнение – это мощная концепция, она ограничена доступностью данных в исходных системах. Например, в нашем случае мы можем загрузить прошедшие события из нашего API, указав дату начала до 30 дней в прошлом. Однако поскольку API предоставляет только до 30 дней в прошлом, обратное заполнение нельзя использовать для загрузки данных за более ранние дни.

Обратное заполнение также можно применять для повторной обработки данных, после того как мы внесли изменения в наш код. Например, предположим, что мы вносим изменения в функцию `calc_statistics`, чтобы добавить новую статистику. Используя обратное заполнение, можно очистить прошлые запуски нашей задачи `calc_statistics`, чтобы повторно проанализировать архивные данные с помощью нового кода. Обратите внимание, что в этом случае мы не

ограничены 30-дневным лимитом источника данных, поскольку мы уже загрузили эти более ранние секции данных в рамках прошедших запусков.

3.6 Лучшие практики для проектирования задач

Хотя Airflow и выполняет бóльшую часть тяжелой работы, когда дело доходит до обратного заполнения и повторного выполнения задач, для получения надлежащих результатов необходимо убедиться, что наши задачи соответствуют определенным ключевым свойствам. В этом разделе мы рассмотрим два наиболее важных свойства правильных задач Airflow: атомарность и идемпотентность.

3.6.1 Атомарность

Термин *атомарность* часто используется в системах баз данных, где атомарная транзакция считается неделимой и несводимой серией операций с базой данных: либо происходит все, либо не происходит ничего. Так же и в Airflow: задачи должны быть определены таким образом, чтобы они были успешными и давали надлежащий результат, либо терпели неудачу, не влияя на состояние системы (рис. 3.9).

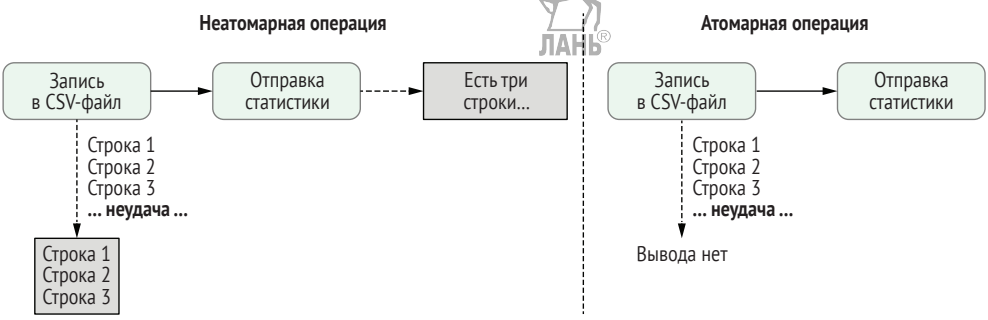


Рис. 3.9 Атомарность гарантирует, что завершится либо все, либо ничего. Вся работа выполняется целиком, и, как следствие, в дальнейшем можно избежать получения неверных результатов

В качестве примера мы рассмотрим простое расширение нашего ОАГ пользовательских событий, в котором мы бы хотели добавить функции, позволяющие отправлять электронные письма 10 нашим лучшим пользователям в конце каждого запуска. Один из простых способов сделать это – расширить нашу предыдущую функцию дополнительным вызовом функции, которая отправляет электронное письмо, содержащее статистику.

Листинг 3.12 Два задания в одной задаче для нарушения атомарности (dags/10_non_atomic_send.py)

```
def _calculate_stats(**context):
    """Вычисляет статистику событий."""
    input_path = context["templates_dict"]["input_path"]
    output_path = context["templates_dict"]["output_path"]

    events = pd.read_json(input_path)
    stats = events.groupby(["date", "user"]).size().reset_index()
    stats.to_csv(output_path, index=False)

    email_stats(stats, email="user@example.com")
```

В ходе отправки электронного письма после записи в файл CSV создаются две единицы работы в одной функции, что нарушает атомарность задачи

К сожалению, недостаток такого подхода состоит в том, что задача больше не является атомарной. Понимаете, почему? Если нет, то подумайте, что произойдет, если наша функция `_send_stats` завершится неудачно (что обязательно произойдет, если наш почтовый сервер ненадежен). В этом случае мы уже запишем статистику в выходной файл в `output_path`, создавая впечатление, что задача выполнена успешно, даже если она закончилась неудачей.

Чтобы реализовать эту функциональность атомарно, можно было бы просто выделить функцию отправки электронной почты в отдельную задачу.

Листинг 3.13 Разделение на несколько задач для улучшения атомарности (dags/ 11_atomic_send.py)

```
def _send_stats(email, **context):
    stats = pd.read_csv(context["templates_dict"]["stats_path"])
    email_stats(stats, email=email)

send_stats = PythonOperator(
    task_id="send_stats",
    python_callable=_send_stats,
    op_kwargs={"email": "user@example.com"},
    templates_dict={"stats_path": "/data/stats/{{ds}}.csv"},
    dag=dag,
)

calculate_stats >> send_stats
```

Выделяем оператор email_stats в отдельную задачу с целью атомарности

Таким образом, неудачная попытка отправить электронное письмо больше не влияет на результат задачи `calculate_stats`, а только приводит к ошибке `send_stats`, что делает обе задачи атомарными.

Исходя из этого примера, можно подумать, что разделения всех операций на отдельные задачи достаточно, чтобы сделать все задачи атомарными. Однако это не всегда так. Чтобы понять, почему, поду-

майте, требовал ли наш API событий, дабы мы выполнили вход перед их запросом. Обычно это требует дополнительного вызова API для получения маркера аутентификации, после чего мы можем начать получать события.

Следуя нашим предыдущим рассуждениям о том, что одна операция = одна задача, нам пришлось бы разделить эти операции на две отдельные задачи. Однако это вызвало бы сильную зависимость между ними, поскольку вторая задача (получение событий) завершится неудачно без запуска первой задачи незадолго до этого. Такая сильная зависимость между ними означает, что, вероятно, лучше сохранить обе операции в рамках одной задачи, позволяя задаче сформировать единую согласованную единицу работы.

Большинство операторов Airflow уже спроектированы как атомарные, поэтому многие операторы включают опции для выполнения тесно связанных операций, таких как внутренняя аутентификация. Однако более гибкие операторы, такие как операторы Python и Bash, могут потребовать от вас тщательно продумать свои операции, чтобы ваши задачи оставались атомарными.

3.6.2 Идемпотентность

Еще одно важное свойство, которое следует учитывать при написании задач Airflow, – это идемпотентность. Задачи называются идемпотентными, если вызов одной и той же задачи несколько раз с одними и теми же входными данными не имеет дополнительного эффекта. Это означает, что повторный запуск задачи без изменения входных данных не должен изменять общий результат.

Например, рассмотрим нашу последнюю реализацию задачи `fetch_events`, которая извлекает результаты за один день и записывает их в секционированный набор данных.

Листинг 3.14 Существующая реализация для извлечения событий (dags/08_templated_paths.py)

```
fetch_events = BashOperator(
    task_id="fetch_events",
    bash_command=(
        "mkdir -p /data/events && "
        "curl -o /data/events/{{ds}}.json "
        "http://localhost:5000/events?"
        "start_date={{ds}}&"
        "end_date={{next_ds}}"
    ),
    dag=dag,
)
```

Разбиение на разделы путем установки шаблонного имени файла

Повторный запуск этой задачи для заданной даты приведет к тому, что задача получит тот же набор событий, что и при предыдущем выполнении (при условии что дата находится в пределах 30-дневного

окна) и перезаписи существующего файла в формате JSON в папке /data/events с тем же результатом. Таким образом, данная реализация задачи `fetch_events` явно идемпотентна.

Чтобы показать пример неидемпотентной задачи, рассмотрим возможность использования файла JSON (/data/events.json), просто добавляя события в этот файл. В этом случае повторный запуск задачи приведет к тому, что события просто будут добавлены к существующему набору данных, дублируя события дня (рис. 3.10). Таким образом, эта реализация не является идемпотентной, поскольку дополнительное выполнение задачи меняет общий результат.

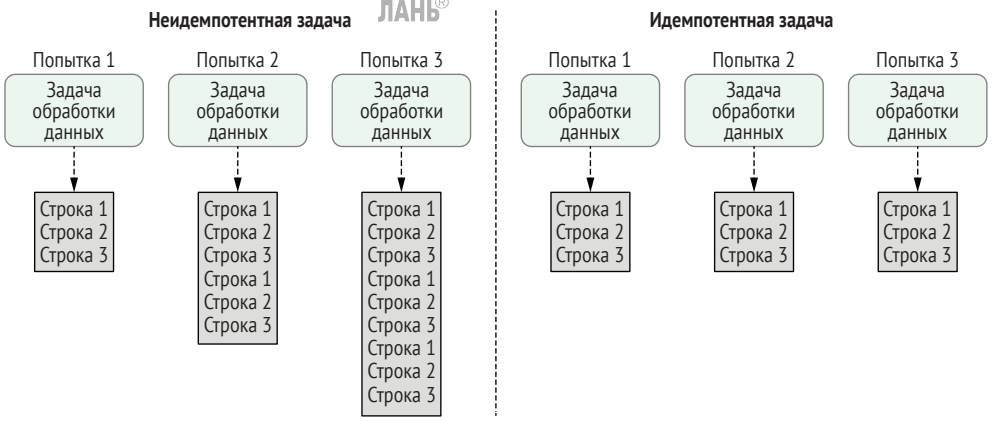


Рис. 3.10 Идемпотентная задача дает один и тот же результат, независимо от того, сколько раз вы ее запускаете. Идемпотентность обеспечивает последовательность и способность справляться со сбоями

В целом задачи, которые записывают данные, можно сделать идемпотентными, проверив существующие результаты или убедившись, что предыдущие результаты перезаписаны задачей. В наборах данных с временным разделением это относительно просто, поскольку мы можем лишь перезаписать соответствующую секцию. Точно так же в случае с системами баз данных можно использовать операции upsert для вставки данных, что позволяет перезаписывать существующие строки, которые были записаны при выполнении предыдущих задач.

Однако в более универсальных приложениях следует тщательно учитывать все побочные эффекты своих задач и убедиться, что они выполняются идемпотентным образом.

Резюме

- ОАГ могут запускаться через равные промежутки времени, задав интервал.

- Запуск задач начинается в конце интервала.
- Интервал расписания можно сконфигурировать с помощью выражений cron и timedelta.
- Данные можно обрабатывать инкреметно, динамически задавая переменные с помощью шаблонизации.
- Дата выполнения относится к дате начала интервала, а не к фактическому времени выполнения.
- ОАГ можно запускать в прошлом с помощью обратного заполнения.
- Идемпотентность обеспечивает возможность повторного выполнения задач с одинаковыми результатами.

4 Создание шаблонов задач с использованием контекста Airflow

Эта глава рассказывает:

- об отображении переменных во время выполнения с использованием шаблонов;
- о создании шаблонов переменных с помощью `PythonOperator` и сравнении с другими операторами;
- об отображении шаблонных переменных с целью отладки;
- о выполнении операций с внешними системами.

В предыдущих главах мы коснулись того, как ОАГ и операторы работают вместе и как спланировать рабочий процесс в Airflow. В этой главе мы подробно рассмотрим, что представляют собой операторы, как они функционируют, а также когда и как они выполняются. Мы продемонстрируем, как использовать операторы для обмена данными с удаленными системами с помощью хуки, что позволяет выполнять такие задачи, как загрузка данных в базу, запуск команды в удаленном окружении и выполнение рабочих нагрузок вне Airflow.

4.1 Проверка данных для обработки с помощью Airflow

В этой главе мы разработаем несколько компонентов операторов с помощью (фиктивного) инструмента прогнозирования фондового рынка, применяющего анализ настроений, который мы назовем StockSense. «Википедия» – один из крупнейших общедоступных информационных ресурсов в интернете. Помимо страниц «Википедии», в открытом доступе существуют и другие вещи, например количество просмотров страниц. Для этого примера мы применим аксиому, согласно которой увеличение количества просмотров страниц компании свидетельствует о положительном настроении, и акции компании, скорее всего, вырастут. С другой стороны, уменьшение количества просмотров говорит нам о падении интереса, и цена акций, вероятно, снизится.

4.1.1 Определение способа загрузки инкрементальных данных

Фонд «Викимедиа» (организация, стоящая за «Википедией») предоставляет все данные о просмотрах страниц с 2015 года в машиночитаемом формате[1]. Данные можно загрузить в формате gzip. Они собраны по часам для каждой страницы. Каждый ежечасный дамп составляет примерно 50 МБ в сжатых текстовых файлах, а в разархивированном виде его размер составляет от 200 до 250 МБ.

При работе с данными это важные детали. Любые данные, и маленькие, и большие, могут быть сложными, и перед созданием конвейера важно иметь технический план подхода. Решение всегда зависит от того, что вы или другие пользователи хотите делать с данными, поэтому спросите себя и других: «Хотим ли мы снова обрабатывать данные в будущем?», «Как мне получить данные (например, частоту, размер, формат, тип источника)?» и «Что мы будем создавать на основе этих данных?». Получив ответы на эти вопросы, можно обратиться к техническим деталям.

Скачаем один из дампов и проверим данные вручную. Чтобы разработать конвейер обработки данных, мы должны понимать, как загружать их поэтапно и как работать с данными (рис. 4.1).

Мы видим, что URL-адреса следуют фиксированному шаблону, который можно использовать при загрузке данных в пакетном режиме (кратко упоминается в главе 3). В качестве мысленного эксперимента и для проверки данных посмотрим, какие коды доменов наиболее часто используются для даты 7 июля, 10:00–11:00 (рис. 4.2).

---

[1] https://dumps.wikimedia.org/other/pageviews. Структура и технические детали данных о просмотрах страниц «Википедии» можно найти здесь: https://meta.wikimedia.org/wiki/Research:Page_view и https://wikitech.wikimedia.org/wiki/Analytics/Data_Lake/Traffic/Pageviews.

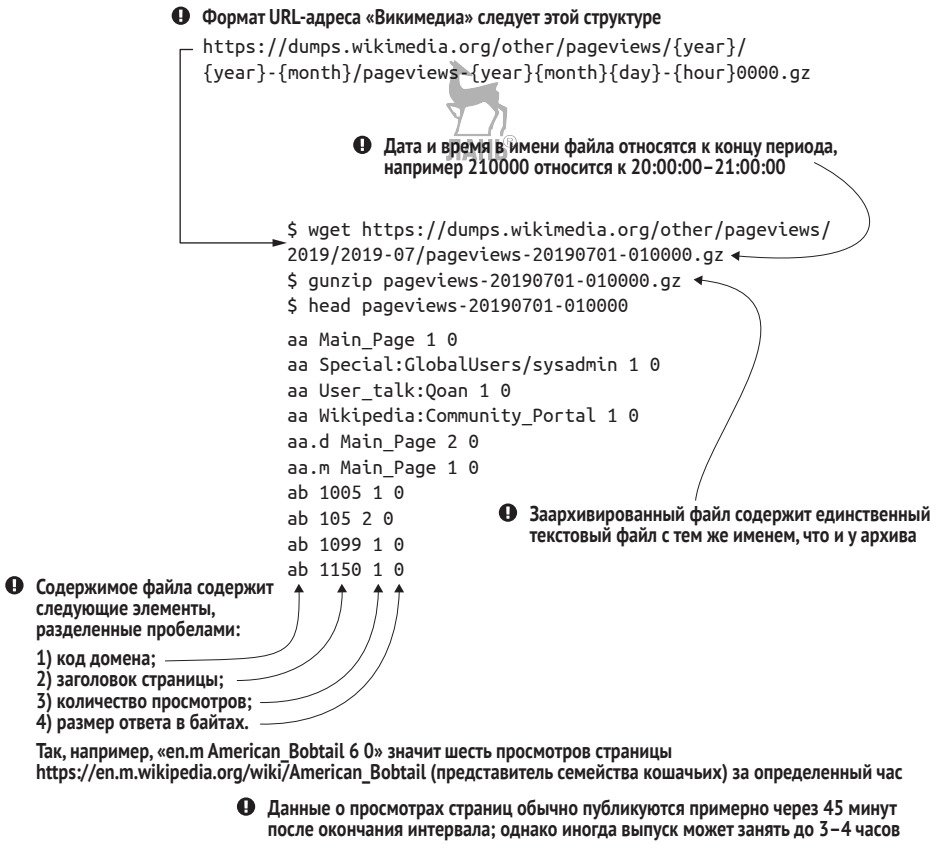


Рис. 4.1 Скачивание и проверка данных о просмотрах страниц «Викимедиа»

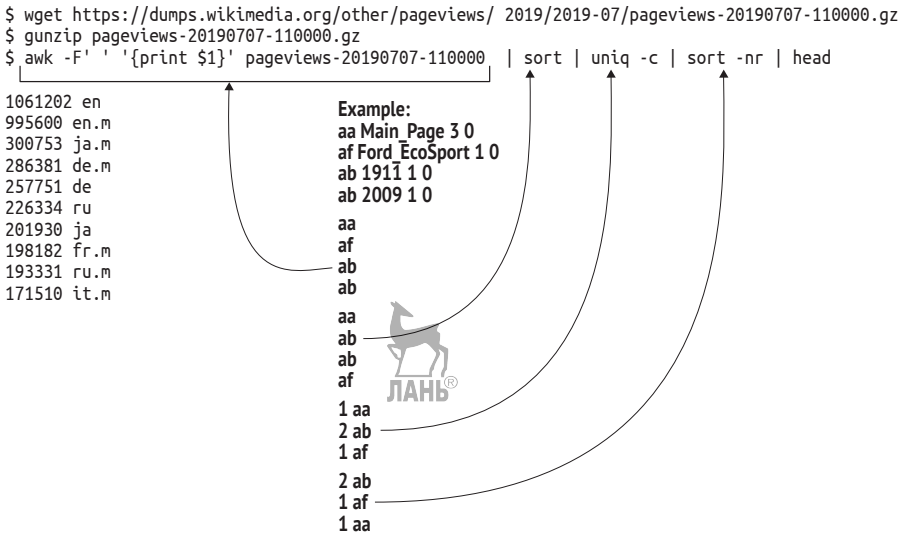


Рис. 4.2 Первый простой анализ данных о просмотрах страниц «Викимедиа»

Если посмотреть на результаты, находящиеся в топе, 1061202 en и 995600 en.m, то мы увидим, что наиболее просматриваемыми доменами в период с 10:00 до 11:00 7 июля являются «en» и «en.m» (мобильная версия .en), что имеет смысл, учитывая, что английский язык – наиболее часто используемый язык в мире. Кроме того, результаты возвращаются в том виде, в каком мы ожидаем их увидеть, что подтверждает отсутствие неожиданных символов или несовпадения столбцов. Это означает, что нам не нужно выполнять дополнительную обработку для очистки данных. Часто очистка и преобразование данных в согласованное состояние – весомая часть работы.

4.2 Контекст задачи и шаблонизатор Jinja

Теперь соберем все это вместе и создадим первую версию ОАГ, учитывая количество просмотров страниц в «Википедии». Начнем с простого: скачивания, распаковки и чтения данных. Мы выбрали пять компаний (Amazon, Apple, Facebook, Google и Microsoft) для первоначального отслеживания и проверки своей гипотезы (рис. 4.3).

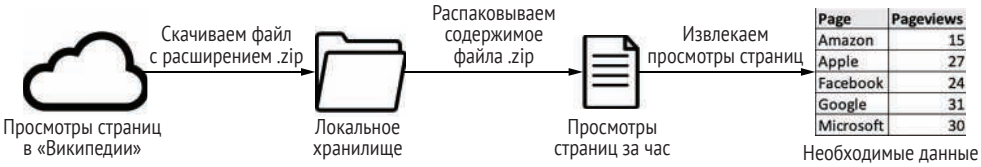


Рис. 4.3 Первая версия рабочего процесса StockSense

Первый шаг – скачать файл с расширением .zip для каждого интервала. URL-адрес состоит из различных компонентов даты и времени:

```
https://dumps.wikimedia.org/other/pageviews/
{year}/{year}-{month}/pageviews-{year}{month}{day}-{hour}0000.gz
```

Для каждого интервала нам нужно будет вставить дату и время в URL-адрес. В главе 3 мы кратко коснулись планирования и того, как использовать дату выполнения в коде для выполнения одного определенного интервала. Давайте подробнее рассмотрим, как это работает. Есть много способов загрузить данные о просмотрах страниц; однако мы сосредоточимся на операторах `BashOperator` и `PythonOperator`. Метод вставки переменных во время выполнения в эти операторы можно распространить на все другие типы операторов.

4.2.1 Создание шаблонов аргументов оператора

Для начала скачаем просмотры страниц «Википедии» с помощью оператора `BashOperator`, принимающего аргумент `bash_command`, которому мы предоставляем команду Bash для выполнения – в начале

и в конце каждого компонента URL-адреса, куда нам нужно вставить переменную, стоят двойные фигурные скобки.

Листинг 4.1 Загрузка просмотров страниц «Википедии» с помощью BashOperator

```
import airflow.utils.dates
from airflow import DAG
from airflow.operators.bash import BashOperator

dag = DAG(
  dag_id="chapter4_stocksense_bashoperator",
  start_date=airflow.utils.dates.days_ago(3),
  schedule_interval="@hourly",
)

get_data = BashOperator(
  task_id="get_data",
  bash_command=(
    "curl -o /tmp/wikipageviews.gz "
    "https://dumps.wikimedia.org/other/pageviews/"
    "{{ execution_date.year }}/"
    "{{ execution_date.year }}-"
    "{{ '{:02}'.format(execution_date.month) }}/"
    "pageviews-{{ execution_date.year }}"
    "{{ '{:02}'.format(execution_date.month) }}"
    "{{ '{:02}'.format(execution_date.day) }}-"
    "{{ '{:02}'.format(execution_date.hour) }}0000.gz"
 ),
 dag=dag,
)
```

Двойные фигурные скобки обозначают переменную, вставленную во время выполнения

Может быть указана любая переменная или выражение Python

Как было кратко сказано в главе 3, `execution_date` – это одна из переменных, которые «волшебным образом» доступны во время выполнения задачи. Двойные фигурные скобки – это синтаксис Jinja. Jinja – это шаблонизатор, который заменяет переменные и/или выражения в шаблонной строке во время выполнения. Создание шаблонов используется, когда вам как программисту неизвестна ценность чего-либо на момент написания, но известна ценность во время выполнения. Например: у вас есть форма, в которую вы можете ввести свое имя, и код выводит введенное имя (рис. 4.4).

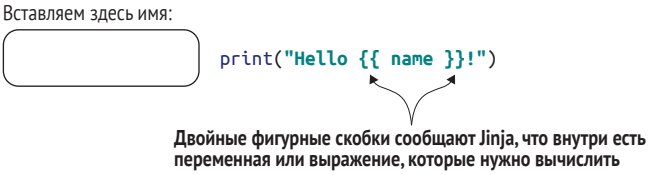


Рис. 4.4 Не все переменные известны заранее при написании кода, например при использовании таких интерактивных элементов, как формы

Значение `name` неизвестно при программировании, потому что пользователь введет свое имя в форму во время выполнения. Что мы действительно знаем, так это то, что вводимое значение присваивается переменной `name`, после чего мы можем предоставить шаблонную строку «`Hello {{name}}!`», чтобы отобразить и вставить значения `name` во время выполнения.

В Airflow есть ряд переменных, доступных во время выполнения из контекста задачи. Одна из таких переменных – `execution_date`. Airflow использует библиотеку Pendulum (https://pendulum.eustace.io) для работы с модулем datetime, а `execute_date` – это объект datetime. Это прямая замена встроенному модулю datetime в Python, поэтому все методы, которые могут быть применены к Python, также можно применить и к Pendulum. Равно как можно использовать `datetime.now().year`, вы можете получить тот же результат с `pendulum.now().year`.

Листинг 4.2 В Pendulum мы наблюдаем аналогичное поведение, что и в Python

```
>>> from datetime import datetime
>>> import pendulum
>>> datetime.now().year
2020
>>> pendulum.now().year
2020
```

URL-адрес просмотров страниц «Википедии» требует использования нулей при заполнении месяцев, дней и часов (например, «07» – это седьмой час). Поэтому внутри строки шаблонизатора Jinja мы применяем форматирование строк:

```
{{ '{:02}'.format(execution_date.hour) }}
```

Какие аргументы являются шаблонными?

Важно знать, что не все аргументы оператора могут быть шаблонами! Каждый оператор может вести список разрешенных атрибутов, которые можно превратить в шаблоны. По умолчанию это не так, поэтому строка `{{name}}` будет интерпретирована буквально как `{{name}}` и не будет шаблонизирована Jinja, если только она не включена в список атрибутов, которые можно шаблонизировать. Этот список задается атрибутом `template_fields` для каждого оператора. Атрибуты можно увидеть в документации (https://airflow.apache.org/docs/); перейдите к выбранному оператору и просмотрите элемент `template_fields`.

Обратите внимание, что элементы в `template_fields` – это имена атрибутов класса. Обычно имена аргументов, предоставленные `__init__`, совпадают с именами атрибутов класса, поэтому все, что перечислено в `template_fields`, – это один к одному аргументы `__init__`. Однако технически возможно, что это не так, и следует задокументировать, какому атрибуту класса соответствует аргумент.

4.2.2 Что доступно для создания шаблонов?

Теперь, когда мы понимаем, какие аргументы оператора можно шаблонизировать, какие есть в нашем распоряжении переменные для этого? Мы видели, что ранее в ряде примеров использовалась переменная `execution_date`, но доступны и другие. С помощью `PythonOperator` можно вывести полный контекст задачи и изучить его.

Листинг 4.3 Вывод контекста задачи

```
import airflow.utils.dates
from airflow import DAG
from airflow.operators.python import PythonOperator

dag = DAG(
   dag_id="chapter4_print_context",
   start_date=airflow.utils.dates.days_ago(3),
   schedule_interval="@daily",
)


def _print_context(**kwargs):
   print(kwargs)


print_context = PythonOperator(
   task_id="print_context",
   python_callable=_print_context,
   dag=dag,
)
```

При выполнении этой задачи выводится словарь всех доступных переменных в контексте задачи.

Листинг 4.4 Все переменные контекста для заданной даты выполнения

```
{
   'dag': <DAG: print_context>,
   'ds': '2019-07-04',
   'next_ds': '2019-07-04',
   'next_ds_nodash': '20190704',
   'prev_ds': '2019-07-03',
   'prev_ds_nodash': '20190703',
   ...
}
```

Все переменные записываются в ** kwargs и передаются функции `print()`. Все эти переменные доступны в среде выполнения. В табл. 4.1 содержится описание всех доступных переменных контекста задачи.

Таблица 4.1 Все переменные контекста задачи

Ключ	Описание	Пример
`conf`	Предоставляет доступ к конфигурации Airflow	Объект `airflow.configuration.AirflowConfigParser`
`dag`	Текущий объект ОАГ	Объект ОАГ
`dag_run`	Текущий объект `DagRun`	Объект `DagRun`
`ds`	`execution_date` в формате %Г-%м-%д	“2019-01-01”
`ds_nodash`	`execution_date` в формате %Г%м%д	“20190101”
`execution_date`	Дата начала и время интервала задачи	Объект `pendulum.datetime.DateTime`
`inlets`	Сокращение для `task.inlets`, функции для отслеживания источников входных данных для происхождения данных	[]
`macros`	Модуль `airflow.macros`	Модуль `macros`
`next_ds`	`execution_date` следующего интервала (= конец текущего интервала) в формате %Г-% м-%д	“2019-01-02”
`next_ds_nodash`	`execution_date` следующего интервала (= конец текущего интервала) в формате %Г%м%д	“20190102”
`next_execution_date`	Дата начала и время следующего интервала задачи (= конец текущего интервала)	Объект `pendulum.datetime.DateTime`
`outlets`	Сокращение для `task.outlets`, функции для отслеживания источников выходных данных для происхождения данных	[]
`params`	Пользовательские переменные для контекста задачи	{}
`prev_ds`	`execution_date` предыдущего интервала в формате %Г-%м-%д	“2018-12-31”
`prev_ds_nodash`	`execution_date` предыдущего интервала в формате %Г%м%д	“20181231”
`prev_execution_date`	Дата начала и время предыдущего интервала задачи	Объект `pendulum.datetime.DateTime`
`prev_execution_date_success`	Дата начала и время последнего успешно завершенного запуска одной той же задачи (только в прошлом)	Объект `pendulum.datetime.DateTime`
`prev_start_date_success`	Дата и время начала последнего успешного запуска одной и той же задачи (только в прошлом)	Объект `pendulum.datetime.DateTime`
`run_id`	`run_id` объекта `DagRun` (обычно ключ состоит из префикса + datetime)	“manual__2019-01-01T00:00:00+00:00”
`task`	Текущий оператор	Объект `PythonOperator`
`task_instance`	Текущий объект `TaskInstance`	Объект `TaskInstance`
`task_instance_key_str`	Уникальный идентификатор текущего объекта `TaskInstance` (`{dag_id}__{task_id}__{ds_nodash}`)	“dag_id__task_id__20190101”
`templates_dict`	Пользовательские переменные для контекста задачи	{}
`test_mode`	Работает ли Airflow в тестовом режиме (свойство конфигурации)	False
`ti`	Текущий объект `TaskInstance`, то же, что и `task_instance`	Объект `TaskInstance`

Таблица 4.1 (*окончание*)

Ключ	Описание	Пример
`tomorrow_ds`	ds плюс один день	“2019-01-02”
`tomorrow_ds_nodash`	ds_nodash плюс один день	“20190102”
`ts`	execution_date, отформатированная в соответствии с форматом ISO8601	“2019-01-01T00:00:00+00:00”
`ts_nodash`	execution_date в формате %Г%м%дВ%Ч%М%С	“20190101T000000”
`ts_nodash_with_tz`	ts_nodash с информацией о часовом поясе	“20190101T000000+0000
`var`	Объекты-помощники для работы с переменными Airflow	{}
`yesterday_ds`	ds минус один день	“2018-12-31”
`yesterday_ds_nodash`	ds_nodash минус один день	“20181231”

Выведено с использованием PythonOperator, запущенного вручную в ОАГ с датой выполнения 2019-01-01T00: 00: 00, интервал с @daily.

4.2.3 Создание шаблона для PythonOperator

`PythonOperator` является исключением из того, что было показано в разделе 4.2.1. С помощью `BashOperator` (и всех других операторов в Airflow) вы предоставляете строку аргументу `bash_command` (или любому другому аргументу в других операторах), который автоматически шаблонизируется во время выполнения. `PythonOperator` является исключением из этого стандарта, потому что он не принимает аргументы, которые можно шаблонизировать, используя контекст среды выполнения. Вместо этого он принимает аргумент `python_callable`, в котором можно применить данный контекст.

Проверим код для скачивания просмотров страниц «Википедии», как показано в листинге 4.1, с помощью `BashOperator`, но теперь уже реализованного с использованием `PythonOperator`. Функционально это приводит к тому же поведению.

Листинг 4.5 Загрузка просмотров страниц «Википедии» с помощью PythonOperator

```
from urllib import request

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

dag = DAG(
    dag_id="stocksense",
    start_date=airflow.utils.dates.days_ago(1),
    schedule_interval="@hourly",
)
```

```
def _get_data(execution_date):
    year, month, day, hour, *_ = execution_date.timetuple()
    url = (
        "https://dumps.wikimedia.org/other/pageviews/"
        f"{year}/{year}-{month:0>2}/"
        f"pageviews-{year}{month:0>2}{day:0>2}-{hour:0>2}0000.gz"
    )
    output_path = "/tmp/wikipageviews.gz"
    request.urlretrieve(url, output_path)


get_data = PythonOperator(
    task_id="get_data",
    python_callable=_get_data,
    dag=dag,
)
```

PythonOperator принимает функцию Python, тогда как BashOperator принимает команду Bash в качестве строки для выполнения

Функции – это объекты первого класса в Python, и мы предоставляем *вызываемый объект*[1] (функция – это вызываемый объект) аргументу `python_callable` оператора `PythonOperator`. При выполнении `PythonOperator` выполняет предоставленный вызываемый объект, которым может быть любая функция. Поскольку это функция, а не строка, как у всех других операторов, код внутри функции нельзя шаблонизировать автоматически. Вместо этого в данной функции можно указать и использовать переменные контекста задачи, как показано на рис. 4.5.

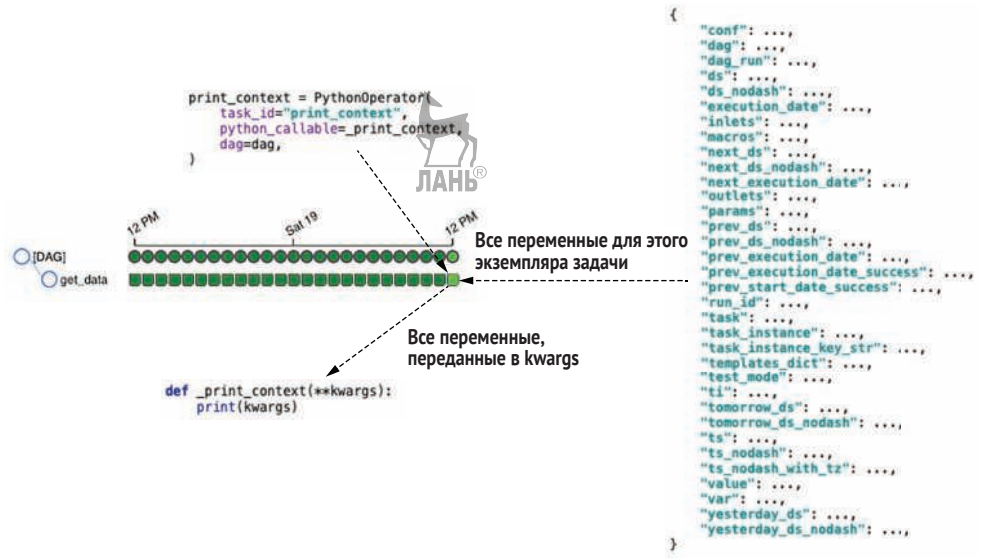


Рис. 4.5 Предоставление контекста задачи с помощью PythonOperator

[1] В Python любой объект, реализующий `__call__()`, считается вызываемым.

provide_context в Airflow 1 и PythonOperator в Airflow 2

В Airflow 1 переменные контекста задачи должны быть указаны явно путем задания в `PythonOperator` значения `True` аргументу `provide_context`: `provide_context=True`. Так вы передаете все (!) переменные контекста задачи вызываемому объекту:

```
PythonOperator(
   task_id="pass_context",
   python_callable=_pass_context,
   provide_context=True,
   dag=dag,
)
```

В Airflow 2 `PythonOperator` определяет, какие контекстные переменные должны быть переданы вызываемому объекту, выводя их из имен вызываемых аргументов. Поэтому больше не нужно задавать значение `True`:

```
PythonOperator(
   task_id="pass_context",
   python_callable=_pass_context,
   dag=dag,
)
```

Для обеспечения обратной совместимости аргумент `provide_context` по-прежнему поддерживается в Airflow 2; однако при работе в Airflow 2 его можно безопасно удалить.

Python позволяет собирать ключевые аргументы в функции. Здесь есть различные варианты, в основном если вам неизвестны заранее предоставленные ключевые аргументы и чтобы избежать необходимости явно выписывать все ожидаемые имена ключевых аргументов.

Листинг 4.6 Ключевые аргументы, хранящиеся в kwargs

```
def _print_context(**kwargs):
   print(kwargs)
```

При сборе ключевых аргументов можно использовать два символа **. По соглашению собирающий аргумент называется kwargs

Чтобы сообщить себе в будущем и другим специалистам, которые будут читать код, о своих намерениях собирать переменные контекста задачи в ключевых аргументах, рекомендуется присвоить этому аргументу соответствующее имя (например, «context»).

Листинг 4.7 Переименовываем kwargs в context, чтобы выразить намерение сохранить контекст задачи

```
def _print_context(**context):
   print(context)


print_context = PythonOperator(
```

Переименовывая этот аргумент в context, мы указываем на то, что ожидаем контекст задачи Airflow

```
    task_id="print_context",
    python_callable=_print_context,
    dag=dag,
)
```

Контекстная переменная – это словарь всех переменных контекста, что позволяет нам задавать задаче различное поведение для интервала, в котором она выполняется, например для вывода даты и времени начала и окончания текущего интервала:

Листинг 4.8 Вывод даты начала и окончания интервала

```
def _print_context(**context):
    start = context["execution_date"]
    end = context["next_execution_date"]
    print(f"Start: {start}, end: {end}")


print_context = PythonOperator(
    task_id="print_context", python_callable=_print_context, dag=dag
)

# Выводим, например:
# Start: 2019-07-13T14:00:00+00:00, end: 2019-07-13T15:00:00+00:00
```

Извлекаем execution_date из контекста

Теперь, когда мы рассмотрели несколько основных примеров, разберем оператор `PythonOperator`, скачивающий ежечасные просмотры страниц «Википедии», как показано в листинге 4.5 (рис. 4.6).

```
def _get_data(**context):
    year, month, day, hour, *_ = context["execution_date"].timetuple()
    url = (
        "https://dumps.wikimedia.org/other/pageviews/"
        f"{year}/{year}-{month:0>2}/pageviews-{year}{month:0>2}{day:0>2}-{hour:0>2}0000.gz"
    )
    output_path = "/tmp/wikipageviews.gz"
    request.urlretrieve(url, output_path)
```

Переменные контекста задачи

Извлекаем компоненты datetime из execution_date

Форматируем URL-адрес с компонентами datetime

Получаем данные

Рис. 4.6 `PythonOperator` принимает функцию вместо строковых аргументов, и, следовательно, в данном случае нельзя использовать шаблонизатор Jinja. В этой вызываемой функции мы извлекаем компоненты datetime из `execution_date` для динамического создания URL-адреса

Функция `_get_data`, вызываемая `PythonOperator`, принимает один аргумент: `**context`. Как было показано ранее, мы можем принять все ключевые аргументы в одном аргументе: `**kwargs` (двойная звездочка указывает на все ключевые аргументы, а kwargs – это фактическое имя переменной). Чтобы указать на то, что мы ожидаем переменные

контекста задачи, можно было бы переименовать его в `**context`. Хотя в Python есть еще один способ принимать ключевые аргументы.

Листинг 4.9 Явное ожидание переменной execute_date

```
def _get_data(execution_date, **context):
    year, month, day, hour, *_ = execution_date.timetuple()
    # ...
```

Так мы сообщаем Python, что ожидаем получить аргумент execute_date. Он не будет собран в аргумент context

Вот что происходит под капотом: функция `_get_data` вызывается со всеми контекстными переменными в качестве ключевых аргументов:

Листинг 4.10 Все переменные контекста передаются в виде ключевых аргументов

```
_get_data(conf=..., dag=..., dag_run=..., execution_date=..., ...)
```

Затем Python проверит, ожидается ли какой-либо из указанных аргументов в сигнатуре функции (рис. 4.7).

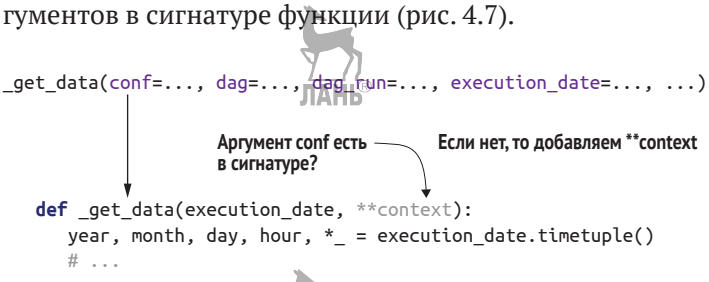


Рис. 4.7 Python определяет, передается ли заданный ключевой аргумент одному конкретному аргументу функции или аргументу **, если подходящее имя не найдено

Проверяется первый аргумент `conf`; и в сигнатуре функции `_get_data` он не обнаружен (ожидаются аргументы). Таким образом, он добавляется в `**context`. То же самое проделывается для `dag` и `dag_run`, поскольку оба аргумента не входят в ожидаемые аргументы функции. Далее это переменная `execution_date`, которую мы ожидаем получить, и, таким образом, значение передается аргументу `execution_date` в функции `_get_data()` (рис. 4.8).

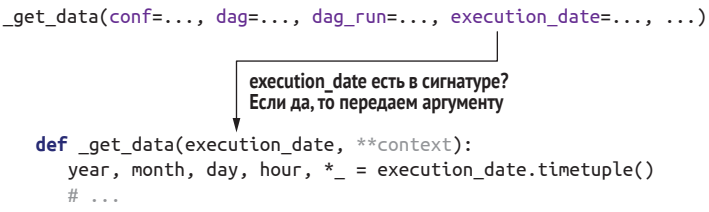


Рис. 4.8 Функция `_get_data` ожидает аргумент `execution_date`. Значение по умолчанию не задано, поэтому случится сбой, если оно не будет предоставлено

Конечный результат этого примера – ключевое слово `execution_date` передается аргументу `execution_date`, а все другие переменные передаются `**context`, поскольку они явно не ожидаются в сигнатуре функции (рис. 4.9).

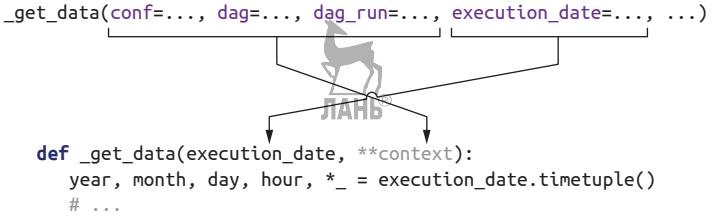


Рис. 4.9 Любой именованный аргумент может быть передан функции `_get_data()`. `execution_date` нужно предоставить явно, потому что он указан как аргумент, все остальные аргументы собирает `**context`

Теперь мы можем напрямую использовать переменную `execution_date`, вместо того чтобы извлекать ее из `**context` с помощью `context["execution_date"]`. Кроме того, ваш код будет более понятным, а такие инструменты, как статические анализаторы и механизм, позволяющий указывать на ожидаемый тип значения (type hinting), выиграют от явного определения аргументов.

4.2.4 Предоставление переменных PythonOperator

Теперь, когда мы увидели, как контекст задачи работает в операторах и как Python работает с ключевыми аргументами, представьте, что нам нужно загрузить данные из нескольких источников данных. Функцию `_get_data()` можно продублировать и немного изменить, чтобы поддерживать второй источник данных. Однако `PythonOperator` также поддерживает предоставление дополнительных аргументов вызываемой функции. Например, предположим, что мы начинаем с конфигурирования `output_path`, чтобы, в зависимости от задачи, можно было настроить `output_path`, вместо того чтобы дублировать всю функцию, просто чтобы изменить выходной путь (рис. 4.10).

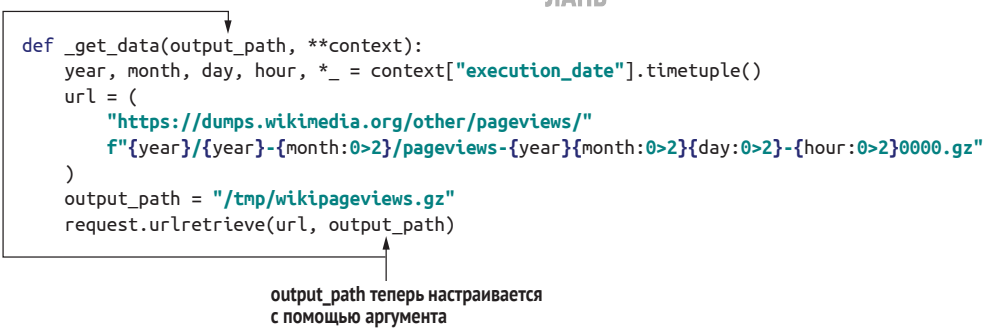


Рис. 4.10 Теперь `output_path` можно сконфигурировать с помощью аргумента

Значение `output_path` можно указать двумя способами. Первый – через аргумент: `op_args`.

Листинг 4.11 Предоставляем определяемые пользователем переменные вызываемому объекту `PythonOperator`

```
get_data = PythonOperator(
    task_id="get_data",
    python_callable=_get_data,
    op_args=["/tmp/wikipageviews.gz"],
    dag=dag,
)
```

Предоставляем дополнительные переменные вызываемому объекту с помощью op_args

При выполнении оператора каждое значение в списке, предоставленное аргументу `op_args`, передается в вызываемую функцию (т. е. здесь тот же эффект, что и при прямом вызове функции: `_get_data("/tmp/wikipageviews.gz")`).

Поскольку `output_path` на рис. 4.10 – это первый аргумент функции `_get_data`, при запуске его значение будет выглядеть так: /tmp/wikipageviews.gz (мы называем их *неключевые аргументы*). Второй подход – использовать аргумент `op_kwargs`, показанный в листинге 4.12.

Листинг 4.12 Предоставляем определяемый пользователем kwargs вызываемому объекту

```
get_data = PythonOperator(
    task_id="get_data",
    python_callable=_get_data,
    op_kwargs={"output_path": "/tmp/wikipageviews.gz"},
    dag=dag,
)
```

Словарь, предоставленный op_kwargs, будет передан в качестве ключевых аргументов вызываемому объекту

Подобно `op_args`, все значения в `op_kwargs` передаются вызываемой функции, но на этот раз как ключевые аргументы. Эквивалентный вызов `_get_data` будет выглядеть так:

```
_get_data(output_path="/tmp/wikipageviews.gz")
```

Обратите внимание, что эти значения могут содержать строки, а следовательно, их можно шаблонизировать. Это означает, что мы могли бы избежать извлечения компонентов datetime внутри самой вызываемой функции и вместо этого передать шаблонные строки вызываемой функции.

Листинг 4.13 Предоставление шаблонных строк в качестве ввода для вызываемой функции

```
def _get_data(year, month, day, hour, output_path, **_):
    url = (
        "https://dumps.wikimedia.org/other/pageviews/"
        f"{year}/{year}-{month:0>2}/"
```

```
        f"pageviews-{year}{month:0>2}{day:0>2}-{hour:0>2}0000.gz"
    )
    request.urlretrieve(url, output_path)


get_data = PythonOperator(
    task_id="get_data",
    python_callable=_get_data,
    op_kwargs={
        "year": "{{ execution_date.year }}",
        "month": "{{ execution_date.month }}",
        "day": "{{ execution_date.day }}",
        "hour": "{{ execution_date.hour }}",
        "output_path": "/tmp/wikipageviews.gz",
    },
    dag=dag,
)
```

Определяемые пользователем ключевые аргументы шаблонизируются перед передачей вызываемому объекту

4.2.5 Изучение шаблонных аргументов

Полезный инструмент для отладки проблем с шаблонными аргументами – пользовательский интерфейс Airflow. Вы можете проверить значения шаблонных аргументов после запуска задачи, выбрав ее на диаграмме или в дереве и щелкнув по кнопке **Rendered Template** (Визуализированный шаблон) (рис. 4.11).

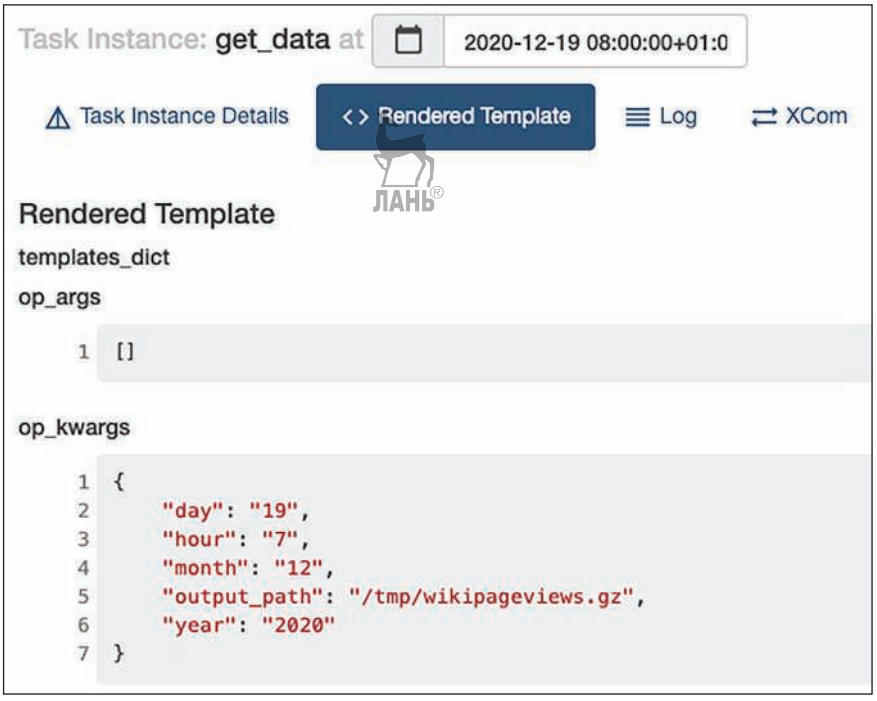


Рис. 4.11 Проверка отображаемых шаблонных значений после запуска задачи

Это представление отображает все атрибуты данного оператора, которые можно визуализировать, и их значения. Оно отображается для каждого экземпляра задачи. Следовательно, задача должна быть запланирована Airflow, прежде чем можно будет проверить отображаемые атрибуты для данного экземпляра задачи (то есть нужно дождаться, пока Airflow спланирует следующую задачу, например). Во время разработки это может оказаться непрактичным. Интерфейс командной строки Airflow позволяет отображать шаблонные значения для любого заданного datetime.

Листинг 4.14 Отображение шаблонных значений для любой заданной даты выполнения

```
# airflow tasks render stocksense get_data 2019-07-19T00:00:00
# ----------------------------------------------------------
# property: templates_dict
# ----------------------------------------------------------
None
# ----------------------------------------------------------
# property: op_args
# ----------------------------------------------------------
[]
# ----------------------------------------------------------
# property: op_kwargs
# ----------------------------------------------------------
{'year': '2019', 'month': '7', 'day': '19', 'hour': '0', 'output_path':
     '/tmp/wikipageviews.gz'}
```

Интерфейс командной строки предоставляет нам ту же информацию, которая показана в пользовательском интерфейсе Airflow, без необходимости запускать задачу, что упрощает проверку результата. Команда для визуализации шаблонов с помощью интерфейса командной строки выглядит так:

```
airflow tasks render [dag id] [task id] [desired execution date]
```

Вы можете ввести любое значение даты и времени, а интерфейс командной строки Airflow отобразит все шаблонные атрибуты, как если бы задача выполнялась в нужное время и дату. При использовании интерфейса командной строки в базе метаданных ничего не регистрируется. Таким образом, это более легковесный и гибкий вариант.

4.3 Подключение других систем

Теперь, когда мы разобрались, как работает шаблонизация, продолжим наш пример, обработав ежечасные просмотры страниц «Википедии». Следующие два оператора будут извлекать архив и обрабатывать извлеченный файл, просматривая его и выбирая количество

просмотров страниц для заданных имен страниц. Результат затем будет выведен в журналах.

Листинг 4.15 Чтение просмотров страниц для заданных имен страниц

```
extract_gz = BashOperator(
    task_id="extract_gz",
    bash_command="gunzip --force /tmp/wikipageviews.gz",
    dag=dag,
)


def _fetch_pageviews(pagenames):
    result = dict.fromkeys(pagenames, 0)
    with open(f"/tmp/wikipageviews", "r") as f:
        for line in f:
        domain_code, page_title, view_counts, _ = line.split(" ")
        if domain_code == "en" and page_title in pagenames:
            result[page_title] = view_counts
    print(result)
    # Выводит, например, "{'Facebook': '778', 'Apple': '20', 'Google': '451',
     'Amazon': '9', 'Microsoft': '119'}"

fetch_pageviews = PythonOperator(
    task_id="fetch_pageviews",
    python_callable=_fetch_pageviews,
    op_kwargs={
        "pagenames": {
            "Google",
            "Amazon",
            "Apple",
            "Microsoft",
            "Facebook",
         }
    },
    dag=dag,
)
```

Открываем файл, написанный в предыдущем задании

Извлекаем элементы на строке

Фильтруем только домен "en"

Проверяем, находится ли page_ title в pagenames

Будет выведено, например, следующее: `{'Apple': '31', 'Microsoft': '87', 'Amazon': '7', 'Facebook': '228', 'Google': '275'}`. В качестве первого улучшения мы бы хотели записать эти цифры в собственную базу данных, что позволит нам обращаться к ней с помощью SQL-запросов и задавать такие вопросы, как «Каково среднее количество просмотров страницы в час на странице Google в “Википедии”»?» (рис. 4.12).

У нас есть база данных Postgres для хранения почасовых просмотров страниц. Таблица для хранения данных состоит из трех столбцов, как показано в листинге 4.16.

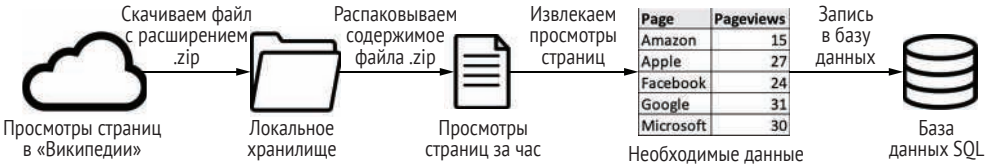


Рис. 4.12 Концептуальное представление рабочего процесса. После извлечения просмотров страниц запишите количество просмотров в базу данных SQL

Листинг 4.16 Оператор CREATE TABLE для хранения вывода

```
CREATE TABLE pageview_counts (
   pagename VARCHAR(50) NOT NULL,
   pageviewcount INT NOT NULL,
   datetime TIMESTAMP NOT NULL
);
```

Столбцы `pagename` и `pageviewcount` содержат название страницы «Википедии» и количество просмотров этой страницы за определенный час соответственно. Столбец `datetime` будет содержать дату и время для подсчета, что соответствует `execution_date` для интервала в Airflow. Пример запроса с использованием операции `INSERT` будет выглядеть следующим образом.

Листинг 4.17 Операция INSERT, сохраняющая вывод в таблице pageview_counts

```
INSERT INTO pageview_counts VALUES ('Google', 333, '2019-07-17T00:00:00');
```

На данный момент этот код выводит найденное количество просмотров страницы, и теперь нам нужно соединить точки, записав эти результаты в таблицу Postgres. В настоящее время `PythonOperator` выводит результаты, но не записывает их в базу данных, поэтому нам понадобится вторая задача для записи результатов. В Airflow есть два способа передачи данных между задачами:

- использовать базу метаданных Airflow для записи и чтения результатов между задачами. Она называется XCom и рассматривается в главе 5;
- записывать результаты в постоянное место и из него (например, на диск или в базу данных) между задачами.

Задачи Airflow выполняются независимо друг от друга, возможно, на разных физических машинах в зависимости от настройки, и поэтому не могут совместно использовать объекты в памяти. Следовательно, данные между задачами должны храниться в другом месте, где они находятся после завершения задачи и могут быть прочитаны другой задачей.

Airflow предоставляет один механизм «из коробки» под названием XCom, позволяющий сохранять и позже читать любой объект, кото-

рый можно сериализовать с помощью модуля pickle, в базу метаданных Airflow. Pickle – это протокол сериализации Python, а сериализация означает преобразование объекта в памяти в формат, который можно сохранить на диске для повторного чтения позже, возможно, другим процессом. По умолчанию все объекты, построенные на основе базовых типов Python (например, string, int, dict, list), могут быть сериализованы.

Примерами объектов, которые нельзя сериализовать, являются подключения к базам данных и обработчики файлов. Использование XComs для хранения сериализуемых объектов подходит только для небольших объектов. Поскольку база метаданных Airflow (обычно это база данных MySQL или Postgres) имеет конечный размер, а сериализуемые с помощью модуля pickle объекты хранятся в больших двоичных объектах в базе метаданных, обычно рекомендуется применять XComs только для передачи небольших фрагментов данных, например нескольких строк (таких как список имен).

Альтернативой для передачи данных между задачами является хранение данных за пределами Airflow. Количество способов хранения данных безгранично, но обычно для этого создается файл на диске. В нашем примере мы извлекли несколько строк и целых чисел, которые сами по себе не занимают много места. Помня о том, что может быть добавлено больше страниц и, таким образом, размер данных в будущем может вырасти, мы будем думать наперед и сохранять результаты на диске, вместо того чтобы использовать XCom'ы.

Чтобы решить, как хранить промежуточные данные, мы должны знать, где и как данные будут использоваться снова. Поскольку целевая база данных – это Postgres, мы будем использовать `PostgresOperator` для вставки данных. Во-первых, нужно установить дополнительный пакет, чтобы импортировать класс `PostgresOperator` в свой проект:

```
pip install apache-airflow-providers-postgres
```

Пакеты провайдеров Airflow 2

Начиная с Airflow 2 большинство операторов устанавливаются с помощью отдельных пакетов pip. Это позволяет избежать установки зависимостей, которые вы, вероятно, не будете использовать, сохраняя небольшой размер основного пакета Airflow. Все дополнительные пакеты pip называются

`apache-airflow-providers-*`

В Airflow осталось лишь несколько основных операторов, таких как `BashOperator` и `PythonOperator`. Обратитесь к документации Airflow, чтобы найти пакет apache-airflow-providers для своих нужд.

`PostgresOperator` выполнит любой запрос, который вы ему предоставите. Поскольку этот оператор не поддерживает вставки из дан-

ных в формате CSV, для начала мы напишем SQL-запросы в качестве промежуточных данных.

Листинг 4.18 Написание инструкций INSERT для передачи в PostgresOperator

```
def _fetch_pageviews(pagenames, execution_date, **_):
    result = dict.fromkeys(pagenames, 0)                    # Инициализируем результат для всех просмотров страниц, используя 0
    with open("/tmp/wikipageviews", "r") as f:
        for line in f:
            domain_code, page_title, view_counts, _ = line.split(" ")
            if domain_code == "en" and page_title in pagenames:
                result[page_title] = view_counts            # Счетчик просмотров страниц

    with open("/tmp/postgres_query.sql", "w") as f:
        for pagename, pageviewcount in result.items():      # Для каждого результата пишем SQL-запрос
            f.write(
              "INSERT INTO pageview_counts VALUES ("
              f"'{pagename}', {pageviewcount}, '{execution_date}'"
              ");\n"
            )


fetch_pageviews = PythonOperator(
    task_id="fetch_pageviews",
    python_callable=_fetch_pageviews,
    op_kwargs={"pagenames": {"Google", "Amazon", "Apple", "Microsoft",
      "Facebook"}},
    dag=dag,
)
```

При выполнении этой задачи будет создан файл (/tmp/postgres_query.sql) для заданного интервала, содержащий все SQL-запросы, которые будут выполняться PostgresOperator. *См.* следующий пример.

Листинг 4.19 Несколько запросов с INSERT для оператора PostgresOperator

```
INSERT INTO pageview_counts VALUES ('Facebook', 275, '2019-07-18T02:00:00+00:00');
INSERT INTO pageview_counts VALUES ('Apple', 35, '2019-07-18T02:00:00+00:00');
INSERT INTO pageview_counts VALUES ('Microsoft', 136, '2019-07-18T02:00:00+00:00');
INSERT INTO pageview_counts VALUES ('Amazon', 17, '2019-07-18T02:00:00+00:00');
INSERT INTO pageview_counts VALUES ('Google', 399, '2019-07-18T02:00:00+00:00');
```

Теперь, когда мы сгенерировали запросы, пришло время соединить последний фрагмент головоломки.

Листинг 4.20 Вызов PostgresOperator

```
from airflow.providers.postgres.operators.postgres import PostgresOperator

dag = DAG(..., template_searchpath="/tmp")      # Путь для поиска sql-файла
```

```
write_to_postgres = PostgresOperator(
   task_id="write_to_postgres",
   postgres_conn_id="my_postgres",
   sql="postgres_query.sql",
   dag=dag,
)
```

Идентификатор учетных данных, используемых для подключения

SQL-запрос или путь к файлу, содержащему SQL-запросы

Соответствующее графовое представление будет выглядеть, как показано на рис. 4.13.

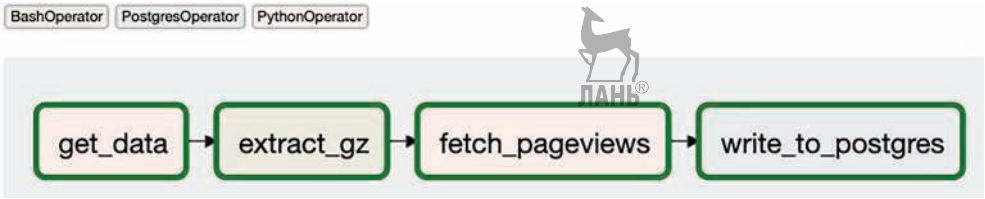


Рис. 4.13 ОАГ, извлекающий ежечасные просмотры страниц из «Википедии» и записывающий результаты в Postgres

`PostgresOperator` требует заполнить только два аргумента для выполнения запроса к базе данных Postgres. Сложные операции, такие как установка соединения с базой данных и его закрытие после завершения, выполняются под капотом. Аргумент `postgres_conn_i` указывает на идентификатор, содержащий учетные данные для базы данных Postgres. Airflow может управлять такими учетными данными (хранящимися в зашифрованном виде в базе метаданных), а операторы при необходимости могут извлечь их. Не вдаваясь в подробности, можно добавить идентификатор подключения `my_postgres` в Airflow с помощью интерфейса командной строки.

Листинг 4.21 Сохранение учетных данных в Airflow с помощью интерфейса командной строки

```
airflow connections add \
--conn-type postgres \
--conn-host localhost \
--conn-login postgres \
--conn-password mysecretpassword \
my_postgres
```

Идентификатор подключения

После этого соединение отображается в пользовательском интерфейсе (его также можно создать оттуда). Перейдите в **Admin > Connections** (Администратор> Подключения), чтобы просмотреть все подключения, хранящиеся в Airflow (рис. 4.14).

После завершения ряда запусков ОАГ база данных Postgres будет содержать следующие данные:

```
"Amazon",12,"2019-07-17 00:00:00"
"Amazon",11,"2019-07-17 01:00:00"
```

```
"Amazon",19,"2019-07-17 02:00:00"
"Amazon",13,"2019-07-17 03:00:00"
"Amazon",12,"2019-07-17 04:00:00"
"Amazon",12,"2019-07-17 05:00:00"
"Amazon",11,"2019-07-17 06:00:00"
"Amazon",14,"2019-07-17 07:00:00"
"Amazon",15,"2019-07-17 08:00:00"
"Amazon",17,"2019-07-17 09:00:00"
```

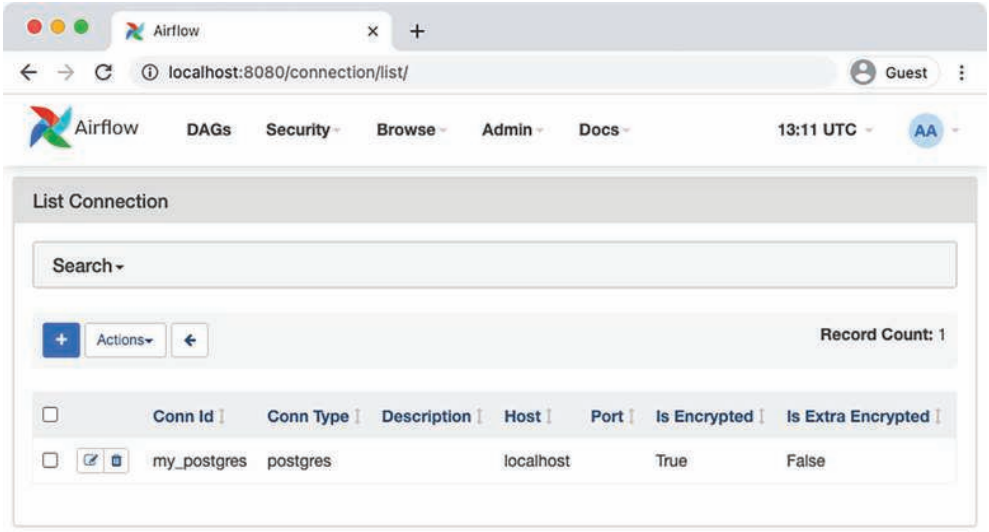


Рис. 4.14 Соединение, указанное в пользовательском интерфейсе Airflow

На этом этапе нужно отметить несколько моментов. У OAG есть дополнительный аргумент: `template_searchpath`. Помимо строки `INSERT INTO ...`, содержимое файлов также можно шаблонизировать. Каждый оператор может читать и создавать шаблоны файлов с определенными расширениями, предоставляя оператору путь к файлу. В случае с `PostgresOperator` аргумент SQL можно шаблонизировать, и, таким образом, также может быть предоставлен путь к файлу, содержащему SQL-запрос. Будет прочитан любой путь к файлу, заканчивающийся расширением .sql, шаблоны в файле будут визуализированы, а `PostgresOperator` выполнит запросы из файла. Обратитесь к документации операторов и проверьте поле `template_ext`, содержащее расширения файлов, которые могут быть созданы оператором.

ПРИМЕЧАНИЕ Jinja требует, чтобы вы указали путь для поиска файлов, которые можно шаблонизировать. По умолчанию ищется только путь к файлу OAG, но, поскольку мы сохранили его в /tmp, Jinja его не найдет. Чтобы добавить пути для поиска, задайте в OAG аргумент `template_searchpath`, и Jinja обойдет путь по умолчанию плюс дополнительные предоставленные пути для поиска.

Postgres – это внешняя система, а Airflow поддерживает подключение к широкому спектру внешних систем с помощью множества операторов в своей экосистеме. Это не лишено смысла: подключение к внешней системе часто требует установки определенных зависимостей, которые позволяют подключаться к внешней системе и обмениваться с ней данными. То же самое относится и к Postgres; нужно установить пакет `apache-airflow-providerpostgres`, чтобы установить дополнительные зависимости Postgres в Airflow. Множество зависимостей – одна из характеристик любой системы оркестровки; чтобы обмениваться данными со множеством внешних систем, установка большого числа зависимостей неизбежна.

После выполнения `PostgresOperator` происходит ряд вещей (рис. 4.15). `PostgresOperator` создает т. н. *хук* для обмена данными с Postgres. Хук занимается созданием подключения, отправкой запросов в Postgres и последующим закрытием подключения. В данной ситуации оператор просто передает запрос от пользователя точке подключения.

ПРИМЕЧАНИЕ Оператор определяет, что нужно сделать; хук определяет, как это сделать.

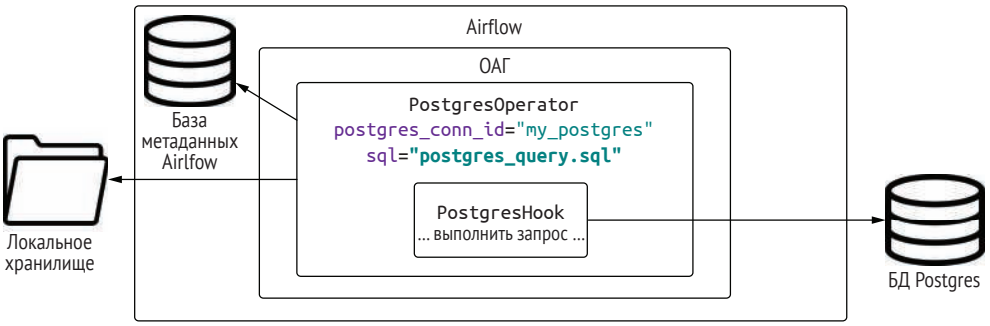


Рис. 4.15 Запуск сценария SQL для базы данных Postgres включает несколько компонентов. Укажите правильные настройки для `PostgresOperator`, а `PostgresHook` выполнит всю работу под капотом

Создавая подобные конвейеры, вы будете иметь дело только с операторами и не будете иметь никакого представления о хуках, потому что они используются внутри операторов.

После нескольких запусков ОАГ база данных Postgres будет содержать несколько записей извлеченных из просмотров страниц «Википедии». Теперь Airflow раз в час автоматически скачивает новый набор данных почасовых просмотров страниц, распаковывает его, извлекает желаемое количество и записывает их в базу данных Postgres. И мы можем задавать такие вопросы, как «В какое время каждая из страниц наиболее популярна?».

Листинг 4.22 SQL-запрос, спрашивающий, в какое время каждая из страниц пользуется наибольшей популярностью

```
SELECT x.pagename, x.hr AS "hour", x.average AS "average pageviews"
FROM (
 SELECT
   pagename,
   date_part('hour', datetime) AS hr,
   AVG(pageviewcount) AS average,
   ROW_NUMBER() OVER (PARTITION BY pagename ORDER BY AVG(pageviewcount) DESC)
 FROM pageview_counts
 GROUP BY pagename, hr
) AS x
WHERE row_number=1;
```

Этот листинг показывает, что наиболее популярное время для просмотра данных страниц – с 16:00 до 21:00, как показано в табл. 4.2.

Таблица 4.2 Результаты запроса, показывающие, какое время является наиболее популярным для каждой страницы

Название страницы	Время	Среднее количество просмотров
Amazon	18	20
Apple	16	66
Facebook	16	500
Google	20	761
Microsoft	21	181

С помощью этого запроса мы завершили предусмотренный рабочий процесс для «Википедии», который выполняет полный цикл загрузки данных о ежечасных просмотрах страниц, обработки данных и записи результатов в базу данных Postgres для будущего анализа. Airflow отвечает за оркестровку правильного времени и порядка запуска задач. С помощью контекста среды выполнения задачи и создания шаблонов код выполняется в течение заданного интервала, используя значения datetime, которые идут с этим интервалом. Если все настроено правильно, рабочий процесс может работать до бесконечности.

Резюме

- Некоторые аргументы операторов можно шаблонизировать.
- Создание шаблонов происходит во время выполнения.
- Шаблонизация `PythonOperator` отличается от других операторов; переменные передаются указанному вызываемому объекту.
- Результат шаблонных аргументов можно проверить с помощью команды `airflow tasks render`.
- Операторы могут обмениваться данными с другими системами с помощью хуков.
- Операторы описывают, *что* нужно делать; хуки определяют, *как* это сделать.

5 Определение зависимостей между задачами

Эта глава:

- показывает, как определять зависимости задач в DAG;
- объясняет, как реализовать соединения с помощью правил триггеров;
- демонстрирует, как ставить задачи в зависимость от определенных условий;
- дает общее представление о том, как правила триггеров влияют на выполнение ваших задач;
- демонстрирует, как использовать XCom'ы для использования состояния в задачах;
- показывает, как Taskflow API в Airflow 2 может помочь упростить DAG, которые интенсивно используют задачи Python.

В предыдущих главах мы видели, как создать базовый DAG и определить простые зависимости между задачами. В этой главе мы подробнее рассмотрим, как именно определяются зависимости задач в Airflow и как использовать эти возможности для реализации более сложных шаблонов, включая условные задачи, ветви и объединения. Ближе к концу главы мы также подробно рассмотрим XCom'ы и обсудим достоинства и недостатки использования данного подхода. Мы покажем, как новый Taskflow API в Airflow 2 может помочь упростить DAG, которые интенсивно используют задачи Python и XCom'ы.

5.1 Базовые зависимости

Прежде чем переходить к более сложным шаблонам зависимостей задач, таким как ветвление и условные задачи, для начала рассмотрим шаблоны зависимостей задач, с которыми мы столкнулись в предыдущих главах. Сюда входят линейные цепочки задач (задачи, которые выполняются одна за другой) и паттерны «один-ко-многим» и «многие-к-одному» (включающие одну задачу, связанную с несколькими нижестоящими задачами, или наоборот). Чтобы убедиться, что мы все находимся на одной волне, мы кратко рассмотрим значение этих паттернов в последующих разделах.

5.1.1 Линейные зависимости

Пока что мы в основном рассматривали примеры OAG, состоящих из одной линейной цепочки задач. Например, OAG из главы 2 (рис. 5.1) состоял из цепочки из трех задач: одна для скачивания метаданных запуска, вторая для скачивания изображений и третья, чтобы уведомить нас о завершении процесса.

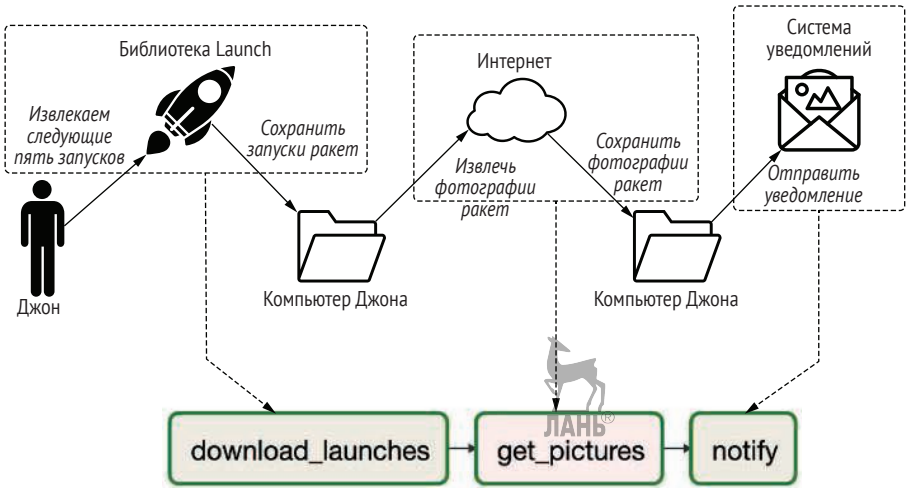


Рис. 5.1 OAG из главы 2 (первоначально показанный на рис. 2.3) состоит из трех задач: скачивание метаданных, извлечение изображений и отправка уведомления

Листинг 5.1 Задачи в OAG из главы 2 (chapter02/dags/isting_2_10.py)

```
download_launches = BashOperator(...)
get_pictures = PythonOperator(...)
notify = BashOperator(...)
```

В этом OAG каждая задача должна быть завершена перед тем, как перейти к следующей, потому что результат предыдущей задачи требуется в качестве ввода для следующей. Как уже было показано,

Airflow позволяет указывать этот тип связи между двумя задачами, создавая зависимость между ними с использованием оператора битового сдвига.

Листинг 5.2 Добавление зависимостей между задачами (chapter02/dags /isting_2_10.py)

```
download_launches >> get_pictures
get_pictures >> notify

download_launches >> get_pictures >> notify
```

Поочередно устанавливаем зависимости задач...

... или устанавливаем несколько зависимостей за раз

Зависимости задач фактически сообщают Airflow, что он может начать выполнение заданной задачи только после того, как ее вышестоящие зависимости будут успешно выполнены. В данном примере это означает, что `get_pictures` может начать выполнение только после успешного выполнения `download_launches`. Точно так же задача `notify` может начаться только после того, как задача `get_pictures` будет завершена без ошибок.

Одно из преимуществ явного указания зависимостей состоит в том, что это четко определяет (неявный) порядок в наших задачах. Это позволяет Airflow планировать задачи только тогда, когда выполняются их зависимости, что более надежно, чем (например) планировать отдельные задачи одну за другой, используя Cron и надеясь, что предыдущие задачи будут выполнены к моменту запуска второй задачи (рис. 5.2). Более того, все ошибки будут распространяться на нижестоящие задачи, и их выполнение, по сути, будет отложено. Это означает, что в случае сбоя в задаче `download_launches` Airflow не будет пытаться выполнить задачу `get_pictures` на этот день, пока не будет решена проблема с `download_launches`.

5.1.2 Зависимости «один-ко-многим» и «многие-к-одному»

Помимо линейных цепочек задач, зависимости задач Airflow могут использоваться для создания более сложных структур зависимостей между задачами. Вернемся к нашему примеру из главы 1, где мы хотели обучить модель для прогнозирования спроса на зонты в ближайшие недели на основе прогноза погоды.

Как вы, возможно, помните, основной целью нашего ОАГ было ежедневное получение данных о погоде и продажах из двух разных источников и объединение данных в набор данных для обучения модели. Таким образом, ОАГ (рис. 5.2) начинается с двух наборов задач для извлечения и очистки входных данных, один для данных о погоде (`fetch_weather` и `clean_weather`) и второй для данных о продажах (`fetch_sales` и `clean_sales`). За этими задачами следует задача (`join_datasets`), которая берет полученные очищенные данные о продажах и погоде и объединяет их в комбинированный набор данных

для обучения модели. Наконец, этот набор данных используется для обучения модели (`train_model`), после чего финальная задача выполняет развертывание модели (`deploy_model`).

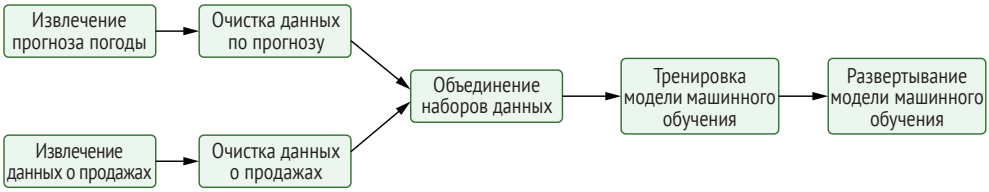


Рис. 5.2 Обзор ОАГ из примера в главе 1

Если рассматривать этот ОАГ с точки зрения зависимостей, существует линейная зависимость между задачами `fetch_weather` и `clean_weather`, поскольку нам нужно извлечь данные из удаленного источника данных, прежде чем мы сможем выполнить какую-либо очистку данных. Однако, поскольку извлечение и очистка данных о погоде не зависят от данных о продажах, между задачами, касающимися погоды и продаж, нет взаимозависимости. Это означает, что мы можем определить зависимости для задач `fetch` и `clean` следующим образом.

Листинг 5.3 Добавление линейных зависимостей, которые выполняются параллельно (dags / 01_start.py)

```
fetch_weather >> clean_weather
fetch_sales >> clean_sales
```

Перед двумя задачами `fetch` также можно было бы добавить фиктивную задачу `start`, обозначающую начало нашего ОАГ. В данном случае эта задача не является строго необходимой, однако она помогает проиллюстрировать неявную зависимость «один-ко-многим», имеющую место в начале ОАГ, когда при его запуске выполняются задачи `fetch_weather` и `fetch_sales`. Такую зависимость (связывание одной задачи с несколькими нижестоящими задачами) можно определить следующим образом:

Листинг 5.4 Добавление зависимости «один-ко-многим» (dags/01_start.py)

```
from airflow.operators.dummy import DummyOperator

start = DummyOperator(task_id="start")   # Создаем фиктивную задачу start
start >> [fetch_weather, fetch_sales]    # Создаем зависимость «один-ко-многим»
```

В отличие от параллелизма задач `fetch` и `clean`, создание комбинированного набора данных требует ввода данных от веток погоды и продаж. Таким образом, задача `join_ datasets` зависит от задач `clean_weather` и `clean_sales` и может запускаться только после успеш-

ного завершения обеих вышестоящих задач. Такой тип структуры, в которой одна задача зависит от нескольких вышестоящих задач, часто называют структурой «многие-к-одному», поскольку она состоит из нескольких вышестоящих задач, которые веером «сходятся» в одной нижестоящей задаче. В Airflow такие зависимости можно определить следующим образом:

Листинг 5.5 Добавление зависимости «многие-к-одному» (dags/ 01_start.py)

```
[clean_weather, clean_sales] >> join_datasets
```

После этого оставшаяся часть ОАГ представляет собой линейную цепочку задач по обучению и развертыванию модели.

Листинг 5.6 Добавление оставшихся зависимостей (dags/01_start.py)

```
join_datasets >> train_model >> deploy_model
```

В совокупности это должно дать нечто похожее на ОАГ, изображенный на рис. 5.3.

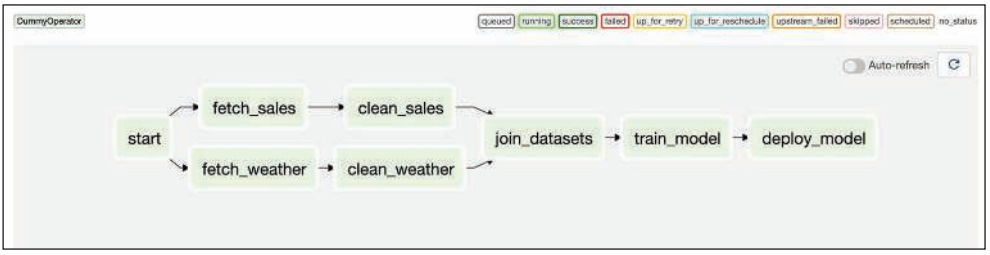


Рис. 5.3 ОАГ, отображаемый в графовом представлении Airflow. Он выполняет несколько задач, в том числе извлечение и очистку данных о продажах, объединение их в набор данных и использование набора данных для обучения модели. Обратите внимание, что обработка данных о продажах и погоде происходит в отдельных ветвях ОАГ, поскольку эти задачи не зависят друг от друга напрямую

Как вы думаете, что произойдет, если мы сейчас начнем выполнение этого ОАГ? Какие задачи начнут выполняться в первую очередь? Как вы думаете, какие задачи (не) будут запускаться параллельно?

Как и следовало ожидать, если мы запустим ОАГ, то Airflow начнет с запуска задачи `start` (рис. 5.4). После ее завершения он инициирует задачи `fetch_sales` и `fetch_weather`, которые будут выполняться параллельно (при условии что Airflow настроен на наличие нескольких воркеров). Завершение любой из задач `fetch` приведет к запуску соответствующих задач очистки (`clean_sales` или `clean_weather`). Только после того, как обе задачи по очистке будут выполнены, Airflow сможет приступить к выполнению задачи `join_datasets`. Наконец, остальная часть ОАГ будет выполняться линейно, при этом зада-

ча train _model будет запущена, как только будет завершена задача join_datasets, а deploy_model – после завершения задачи train_model.

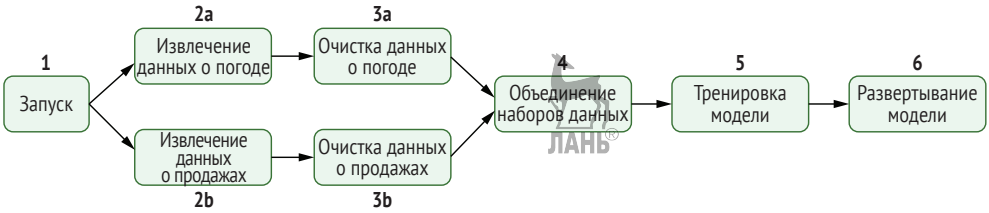


Рис. 5.4 Порядок выполнения задач в ОАГ с номерами, указывающими порядок выполнения задач. Airflow запускается с выполнения задачи *start*, после чего он может запускать задачи fetch_sales и fetch_weather и задачи по очистке параллельно (на что указывает суффикс a/b). Обратите внимание, это означает, что пути weather и sales работают независимо, а значит, 3b может, например, начать выполнение до 2a. После выполнения обеих задач clean остальная часть ОАГ линейно переходит к выполнению задач join, train и deployment

5.2 Ветвление

Представьте, что вы только что закончили вводить данные о продажах в свой ОАГ, когда приходит ваш коллега с новостями. Судя по всему, руководство решило, что будет переходить на ERP-системы, а это значит, что данные о продажах будут поступать из другого источника (и, конечно, в другом формате) через одну-две недели. Очевидно, что такое изменение не должно привести к прерыванию обучения нашей модели. Более того, они бы хотели, чтобы мы поддерживали совместимость нашего потока как со старой, так и с новой системами, чтобы мы могли продолжать использовать прошлые данные о продажах в будущем анализе. Как бы вы подошли к решению этой проблемы?

5.2.1 Ветвление внутри задач

В качестве одного из подходов можно переписать задачи приема данных о продажах, чтобы проверить текущую дату выполнения и использовать ее для выбора между двумя отдельными путями кода для приема и обработки данных о продажах. Например, задачу clean_sales можно было бы переписать примерно так:

Листинг 5.7 Ветвление в задаче clean_sales (dags/02_branch_task.py)

```
def _clean_sales(**context):
    if context["execution_date"] < ERP_CHANGE_DATE:
        _clean_sales_old(**context)
    else
        _clean_sales_new(**context)

...
```

```
clean_sales_data = PythonOperator(
    task_id="clean_sales",
    python_callable=_clean_sales,
)
```

В данном примере `_clean_sales_old` — это функция, выполняющая очистку для старого формата продаж, а `_clean_sales_new` делает то же самое для нового формата. Пока результат является совместимым (с точки зрения столбцов, типов данных и т. д.), остальная часть ОАГ может оставаться неизменной, и ей не нужно беспокоиться о различиях между двумя ERP-системами.

Точно так же мы могли бы сделать наш начальный этап приема данных совместимым с обеими ERP-системами, добавив пути кода из обеих систем.

Листинг 5.8 Ветвление в задаче `fetch_sales` (dags/02_branch_task.py)

```
def _fetch_sales(**context):
    if context["execution_date"] < ERP_CHANGE_DATE:
        _fetch_sales_old(**context)
    else:
        _fetch_sales_new(**context)
    ...
```

В совокупности эти изменения позволят ОАГ относительно прозрачно обрабатывать данные из обеих систем, поскольку наши первоначальные задачи по извлечению и очистке данных гарантируют, что данные о продажах будут поступать в том же (обработанном) формате независимо от соответствующего источника данных.

Преимущество такого подхода состоит в том, что он позволяет добавить некоторую гибкость в наши ОАГ без необходимости изменять их структуру. Однако такой подход работает только в тех случаях, когда ветви в коде состоят из похожих задач. Здесь, например, в коде есть две ветви, каждая из которых выполняет операции по извлечению и очистке с минимальными различиями. Но что, если для загрузки данных из нового источника данных требуется совсем другая цепочка задач (рис. 5.5)? В таком случае, возможно, лучше будет разделить прием данных на два отдельных набора задач.

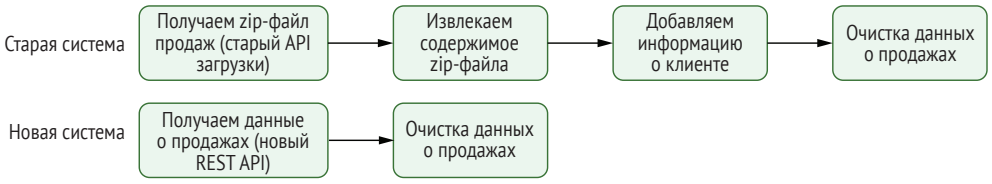


Рис. 5.5 Возможный пример различных наборов задач между двумя ERP-системами. Если между разными случаями много общего, возможно, вам удастся обойтись одним набором задач и внутренним ветвлением. Однако если между двумя потоками есть много различий (например, как показано здесь), вероятно, лучше выбрать другой подход

Еще один недостаток этого подхода заключается в том, что трудно увидеть, какая ветка кода используется Airflow во время определенного запуска ОАГ. Например, глядя на рис. 5.6, можете ли вы определить, какая ERP-система использовалась для этого конкретного запуска ОАГ? На этот, казалось бы, простой вопрос довольно сложно ответить, используя только данное представление, поскольку фактическое ветвление скрыто в задачах. Один из способов решить эту проблему – включить в задачи более эффективное журналирование, но, как мы увидим, есть и другие способы сделать ветвление более явным в самом ОАГ.

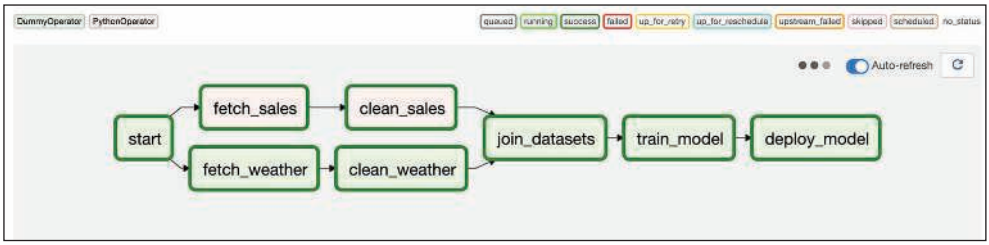


Рис. 5.6 Пример запуска ОАГ, который разветвляется между двумя ERP-системами в рамках задач `fetch_sales` и `clean_sales`. Поскольку это ветвление происходит в этих двух задачах, невозможно увидеть, какая ERP-система использовалась в данном запуске. Это означает, что нам нужно будет проверить свой код (или, возможно, журналы), чтобы определить это

Наконец, можно реализовать эту гибкость в своих задачах, только вернувшись к универсальным операторам Airflow, таким как `PythonOperator`. Это не дает нам использовать функции, предоставляемые более специализированными операторами, которые позволяют выполнять более сложную работу, прилагая минимальные усилия для написания кода. Например, если бы одним из наших источников данных оказалась база данных SQL, это сэкономило бы нам много работы, если бы мы могли просто использовать `MysqlOperator` для выполнения SQL-запроса, поскольку это позволяет нам делегировать фактическое выполнение запроса (наряду с аутентификацией и т. д.) предоставленному оператору.

К счастью, проверка условий в задачах – не единственный способ выполнить ветвление в Airflow. В следующем разделе мы покажем, как вплетать ветви в структуру ОАГ, что обеспечивает бóльшую гибкость, по сравнению с подходом на основе задач.

5.2.2 Ветвление внутри ОАГ

Еще один способ поддержки двух разных ERP-систем в одном ОАГ – разработать два разных набора задач (по одному на каждую систему) и предоставить ОАГ выбирать, из какой ERP-системы выполнять задачи по извлечению данных: из старой или из новой (рис. 5.7).

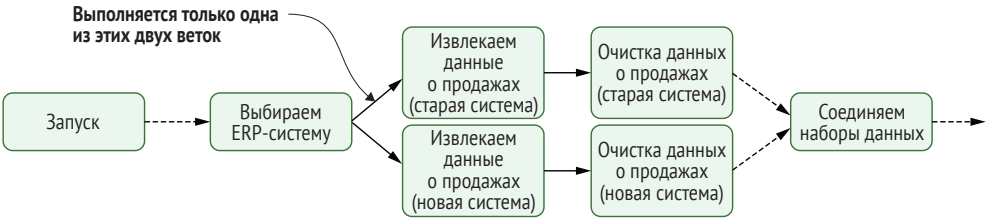


Рис. 5.7 Поддержка двух ERP-систем с использованием ветвления внутри ОАГ, реализуя разные наборы задач для обеих систем. Airflow может выбирать между этими двумя ветвями, используя определенную задачу ветвления (здесь это «Выбрать ERP-систему»), которая сообщает Airflow, какой набор нижестоящих задач выполнить

Создать два набора задач относительно просто: можно просто создать задачи для каждой ERP-системы по отдельности, используя нужные операторы, и связать соответствующие задачи.

Листинг 5.9 Добавление дополнительных задач по извлечению и очистке (dags/03_branch_dag.py)

```
fetch_sales_old = PythonOperator(...)
clean_sales_old = PythonOperator(...)

fetch_sales_new = PythonOperator(...)
clean_sales_new = PythonOperator(...)

fetch_sales_old >> clean_sales_old
fetch_sales_new >> clean_sales_new
```

Теперь нам нужно подключить эти задачи к остальной части ОАГ и убедиться, что Airflow знает, что и когда нужно выполнить.

К счастью, Airflow предоставляет встроенную поддержку для выбора между наборами нижестоящих задач с помощью оператора `BranchPythonOperator`. Этот оператор (как следует из названия) похож на `PythonOperator` в том смысле, что принимает вызываемый объект Python в качестве одного из своих основных аргументов.

Листинг 5.10 Ветвление с использованием BranchPythonOperator (dags/03_branch_dag.py)

```
def _pick_erp_system(**context):
    ...

pick_erp_system = BranchPythonOperator(
    task_id="pick_erp_system",
    python_callable=_pick_erp_system,
)
```

Однако, в отличие от `PythonOperator`, вызываемые объекты, передаваемые в `BranchPythonOperator`, должны возвращать идентификатор нижестоящей задачи в качестве результата их вычисления. Воз-

вращаемый идентификатор определяет, какая из нижестоящих задач будет выполнена после завершения задачи ветвления. Обратите внимание, что вы также можете вернуть список идентификаторов задач, и тогда Airflow выполнит все указанные задачи.

В данном случае мы можем реализовать наш выбор между двумя ERP-системами, используя вызываемый объект для возврата соответствующего `task_id` в зависимости от даты выполнения ОАГ.

Листинг 5.11 Добавление функции условия ветвления (dags/03_branch_dag.py)

```
def _pick_erp_system(**context):
    if context["execution_date"] < ERP_SWITCH_DATE:
        return "fetch_sales_old"
    else:
        return "fetch_sales_new"

pick_erp_system = BranchPythonOperator(
    task_id="pick_erp_system",
    python_callable=_pick_erp_system,
)

pick_erp_system >> [fetch_sales_old, fetch_sales_new]
```

Таким образом, Airflow выполнит наш набор задач «старой» ERP-системы для дат выполнения, наступающих до даты перехода, при выполнении новых задач после этой даты. Теперь все, что нужно сделать, – это связать эти задачи с остальной частью ОАГ.

Чтобы связать задачу ветвления с ОАГ, можно добавить зависимость между предыдущей задачей `start` и задачей `pick_erp_system`.

Листинг 5.12 Подключение ветви к задаче `start` (dags/03_branch_dag.py)

```
start_task >> pick_erp_system
```

Можно ожидать, что соединить две задачи `clean` так же просто, как добавить зависимость между задачами `clean` и задачей `join_datasets` (аналогично предыдущей ситуации, когда задача `clean_sales` была подключена к задаче `join_datasets`).

Листинг 5.13 Подключение ветви к задаче join_datasets (dags/03_branch_dag.py)

```
[clean_sales_old, clean_sales_new] >> join_datasets
```

Однако если вы это сделаете, то запуск ОАГ приведет к тому, что Airflow пропустит задачу `join_datasets` и все нижестоящие задачи. (Можете попробовать, если хотите.)

Причина этого состоит в том, что по умолчанию Airflow требует, чтобы все задачи, стоящие перед данной задачей, успешно заверши-

лись, прежде чем сама задача может быть выполнена. Подключив обе задачи `clean` к задаче `join_datasets`, мы создали ситуацию, когда этого не может произойти, поскольку выполняется только одна из задач `clean`. В результате задача `join_datasets` так и не сможет быть выполнена, и Airflow пропустит ее (рис. 5.8).

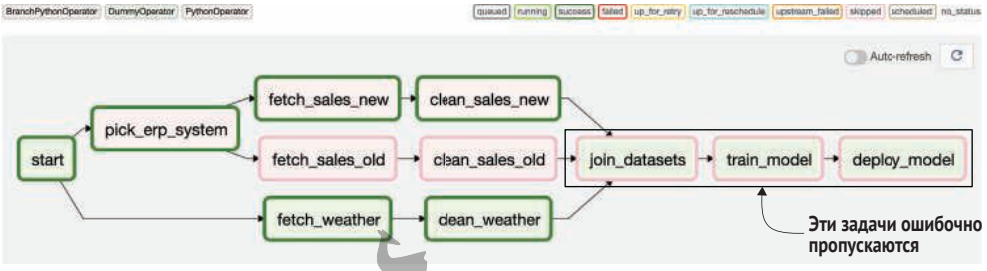


Рис. 5.8 Объединение ветвления с неправильными правилами триггеров приведет к пропуску нижестоящих задач. В этом примере задача `fetch_sales_new` пропускается из-за ветви sales. Это приводит к тому, что все задачи после задачи `fetch_sales_new` так же пропускаются, а это явно не то, что нам нужно

Такое поведение, определяющее, когда выполняются задачи, контролируется так называемыми *правилами триггеров* в Airflow. Правила триггеров можно определить для отдельных задач с помощью аргумента `trigger_rule`, который можно передать любому оператору. По умолчанию для правил триггера установлено значение `all_success`. Это означает, что все родители соответствующей задачи должны завершиться успешно до того, как задачу можно будет запустить. Такого никогда не происходит при использовании `BranchPythonOperator`, поскольку он пропускает все задачи, не выбранные веткой. Это объясняет, почему задача `join_datasets` и все ее нижестоящие задачи также были пропущены Airflow.

Чтобы исправить ситуацию, можно изменить правило триггеров `join_datasets`, чтобы оно сработало, если одна из вышестоящих задач пропущена. Один из способов добиться этого – изменить правило на `none_failed`. Это указывает на то, что задача должна выполняться, как только все ее родительские задачи будут выполнены и ни одна из них не завершилась сбоем.

Листинг 5.14 Исправляем правило триггеров для задачи join_datasets (dags/03_branch_dag.py)

```
join_datasets = PythonOperator(
    ...,
    trigger_rule="none_failed",
)
```

Таким образом, задача `join_datasets` начнет выполняться, как только все ее родительские задачи будут выполнены без каких-либо сбоев, что позволит продолжить ее выполнение (рис. 5.9).

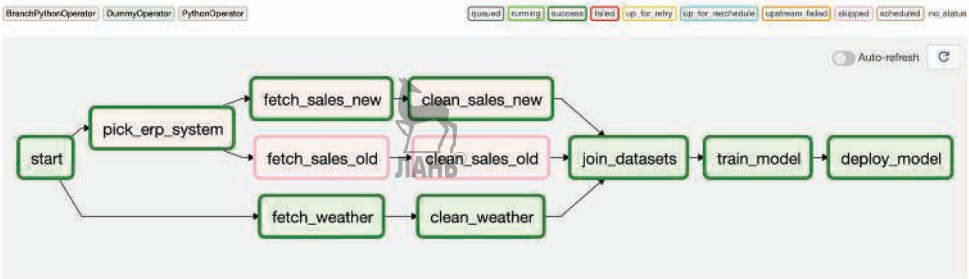


Рис. 5.9 Ветвление в ОАГ с использованием правила триггеров `none_failed` для задачи `join_datasets`, что позволяет ей (и ее нижестоящим зависимостям) по-прежнему выполняться

Один из недостатков такого подхода состоит в том, что теперь у нас есть три ребра, входящих в задачу `join_datasets`. На самом деле это не отражает характер нашего потока, в котором мы, по сути, хотим получить данные о продажах и погоде (вначале выбирая между двумя ERP-системами), а затем загрузить эти два источника данных в `join_datasets`. По этой причине многие предпочитают сделать условие ветвления более явным, добавляя фиктивную задачу, которая объединяет разные ветви, прежде чем продолжить работу с ОАГ (рис. 5.10).

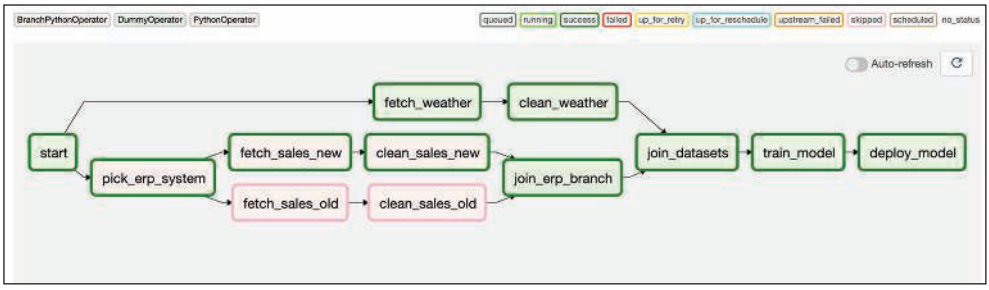


Рис. 5.10 Чтобы сделать структуру ветвления более понятной, можно добавить дополнительную задачу соединения после ветви, которая связывает разные ветви перед продолжением работы с остальной частью ОАГ. У этой задачи имеется дополнительное преимущество, заключающееся в том, что вам не нужно изменять какие-либо правила триггеров для других задач в ОАГ, поскольку вы можете задать необходимое правило для задачи соединения. (Обратите внимание: это означает, что вам больше не нужно задавать правило триггеров для задачи `join_datasets`)

Чтобы добавить такую фиктивную задачу в ОАГ, можно использовать встроенный оператор `DummyOperator`, предоставляемый Airflow.

Листинг 5.15 Добавление фиктивной задачи соединения для ясности (dags/04_branch_dag_join.py)

```
from airflow.operators.dummy import DummyOperator

join_branch = DummyOperator(
```

```
    task_id="join_erp_branch",
    trigger_rule="none_failed"
)

[clean_sales_old, clean_sales_new] >> join_branch
join_branch >> join_datasets
```

Это изменение также означает, что нам больше не нужно изменять правило триггеров для задачи `join_datasets`, что делает нашу ветку более автономной по сравнению с исходной.

5.3 Условные задачи

Airflow также предоставляет другие механизмы для пропуска определенных задач в ОАГ в зависимости от определенных условий. Это позволяет запускать определенные задачи только при наличии определенных наборов данных или лишь в том случае, если ваш ОАГ выполняется для самой последней даты выполнения.

Например, в ОАГ, показанном на рис. 5.3, у нас есть задача, которая развертывает каждую модель, которую мы обучаем. Однако подумайте, что произойдет, если коллега внесет изменения в код очистки и захочет использовать обратное заполнение, чтобы применить эти изменения ко всему набору данных. Это приведет к развертыванию большого числа старых экземпляров нашей модели, которые нас определенно не интересуют.

5.3.1 Условия в задачах

Можно избежать данной проблемы, изменив ОАГ для развертывания модели только для самого последнего запуска, поскольку это гарантирует, что мы развернем лишь одну версию нашей модели: ту, которая обучена на самом последнем наборе данных. Один из способов сделать это – реализовать развертывание с помощью оператора `PythonOperator` и явно проверить дату выполнения ОАГ в функции развертывания.

Листинг 5.16 Реализация условия в задаче (dags/05_condition_task.py)

```
def _deploy(**context):
    if context["execution_date"] == ...:
        deploy_model()

deploy = PythonOperator(
    task_id="deploy_model",
    python_callable=_deploy,
)
```

Хотя такая реализация должна иметь ожидаемый эффект, у нее есть те же недостатки, что и у соответствующей реализации ветвления:

она смешивает логику развертывания с условием. Мы больше не можем использовать какие-либо другие встроенные операторы, кроме `PythonOperator`, и отслеживание результатов задачи в пользовательском интерфейсе Airflow становится менее явным (рис. 5.11).

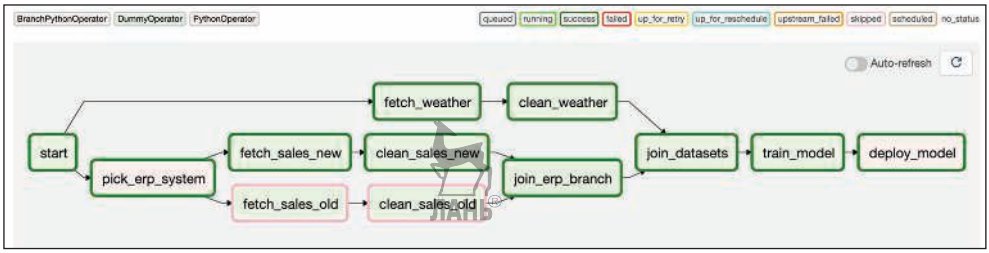


Рис. 5.11 Пример запуска ОАГ с условием внутри задачи `deploy_model`. Это гарантирует, что развертывание выполняется только для последнего запуска. Поскольку условие проверяется внутри задачи `deploy_model`, исходя из этого представления, нельзя определить, действительно ли модель была развернута

5.3.2 Делаем задачи условными

Еще один способ реализовать условное развертывание – сделать саму задачу развертывания условной. Это означает, что она выполняется только на основе заранее определенного условия (в этом случае является ли запуск ОАГ самым последним). В Airflow можно сделать задачи условными, добавив задачу в ОАГ, который проверяет указанное условие и обеспечивает пропуск всех нижестоящих задач в случае невыполнения условия.

Кроме того, можно сделать развертывание условным, добавив задачу, которая проверяет, является ли текущее выполнение самым последним выполнением ОАГ, и добавив задачу развертывания после этой задачи.

Листинг 5.17 Встраивание условия в ОАГ (dags/06_condition_dag.py)

```
def _latest_only(**context):
    ...

latest_only = PythonOperator(
    task_id="latest_only",
    python_callable=_latest_only,
    dag=dag,
)

latest_only >> deploy_model
```

Теперь это означает, что наш ОАГ должен выглядеть примерно так, как показано на рис. 5.12, с задачей `train_model`, подключенной к новой задаче, и задачей `deploy_model` после этой новой задачи.

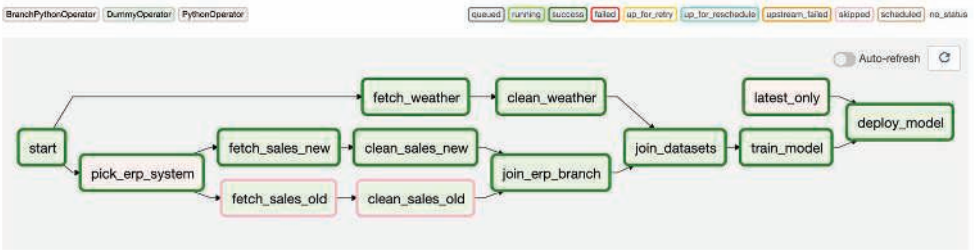


Рис. 5.12 Альтернативная реализация ОАГ с условным развертыванием, где условие включается в качестве задачи в ОАГ, что делает условие более явным, чем в предыдущей реализации

Затем нам нужно заполнить функцию `_latest_only`, чтобы убедиться, что нижестоящие задачи пропускаются, если `execution_date` не принадлежит самому последнему запуску. Для этого нужно проверить дату выполнения и, при необходимости, вызвать исключение `AirflowSkipException` из нашей функции. Это способ Airflow, позволяющий указать, что условие и все его нижестоящие задачи должны быть пропущены, тем самым пропуская развертывание.

Таким образом, мы получаем следующую реализацию условия:

Листинг 5.18 Реализация условия _latest_only (dags/06_condition_dag.py)

```
from airflow.exceptions import AirflowSkipException

def _latest_only(**context):
    left_window = context["dag"].following_schedule(context["execution_date"])
    right_window = context["dag"].following_schedule(left_window)

    now = pendulum.utcnow()
    if not left_window < now <= right_window:
        raise AirflowSkipException("Not the most recent run!")
```

Находим границы для нашего окна выполнения

Проверяем, находится ли наше текущее время в рамках окна

Можно проверить, получим ли мы то, чего ожидаем, выполнив наши ОАГ для нескольких дат. Мы должны увидеть нечто похожее на то, что изображено рис. 5.13. Здесь показано, что наша задача развертывания была пропущена во всех запусках ОАГ, кроме последнего.

Как это работает? По сути, происходит следующее: когда наша задача условия (`latest _only`) возбуждает исключение `AirflowSkipException`, задача завершается, и ей назначается состояние пропуска. Затем Airflow проверяет правила триггеров всех нижестоящих задач, чтобы определить, должны ли они запускаться. В этом случае у нас есть только одна нижестоящая задача (задача развертывания), в которой используется правило триггеров по умолчанию, `all_success`, указывающее на то, что задача должна выполняться только в том случае, если все ее вышестоящие задачи были выполнены успешно. В данном случае это неверно, поскольку родительская задача (задача условия)

имеет пропущенное состояние, она не является успешной, и поэтому развертывание пропускается.

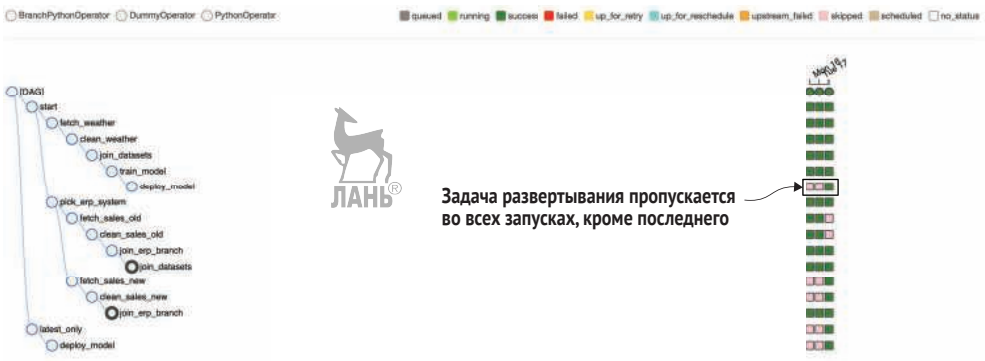


Рис. 5.13 Результат условия `latest_only` для трех запусков ОАГ. Это древовидное представление показывает, что задача развертывания была запущена только для самого последнего окна выполнения, поскольку задача развертывания была пропущена при предыдущих выполнениях. Видно, что наше условие и в самом деле работает, как и ожидалось

И наоборот, если задача условия не возбуждает исключение `AirflowSkipException`, то она успешно завершается и получает статус «успешно». Таким образом, запускается задача развертывания, поскольку все ее родительские задачи завершились успешно, и мы получаем наше развертывание.

5.3.3 Использование встроенных операторов

Поскольку обычно выполняются только задачи для последнего запуска ОАГ, Airflow также предоставляет встроенный класс `LatestOnlyOperator`. По сути, этот оператор выполняет ту же работу, что и наша пользовательская реализация на основе `PythonOperator`. Используя `LatestOnlyOperator`, мы также можем реализовать условное развертывание таким образом, что избавляет нас от написания собственной сложной логики.

Листинг 5.19 Использование встроенного оператора LatestOnlyOperator (dags/07_condition_dag_op.py)

```
from airflow.operators.latest_only import LatestOnlyOperator

latest_only = LatestOnlyOperator(
    task_id="latest_only",
    dag=dag,
)

train_model >> latest_only >> deploy_model
```

Конечно, для более сложных случаев маршрут на основе `PythonOperator` предоставляет больше гибкости для реализации собственных условий.

5.4 Подробнее о правилах триггеров

В предыдущих разделах мы видели, как Airflow позволяет нам создавать ОАГ с динамическим поведением, что дает возможность писать код ветвей или условные операторы непосредственно в ОАГ. Бóльшая часть этого поведения регулируется так называемыми правилами триггеров Airflow, которые точно определяют, когда задача выполняется. В предыдущих разделах мы относительно быстро пробежались по правилам триггеров, поэтому теперь мы более подробно рассмотрим, что они собой представляют и что можно с ними делать.

Чтобы разбираться в правилах триггеров, сначала нужно изучить, как Airflow выполняет задачи в рамках запуска ОАГ. По сути, когда Airflow выполняет ОАГ, он постоянно проверяет каждую из ваших задач, чтобы узнать, можно ли ее выполнить. Как только задача будет признана готовой к выполнению, она выбирается планировщиком, после чего планируется ее выполнение. В результате задача выполняется, как только Airflow получает доступный слот выполнения.

Итак, как же Airflow определяет, когда задачу можно выполнить? Вот тут вступают в действие правила триггера.

5.4.1 Что такое правило триггеров?

Правила триггеров – это, по сути, условия, которые Airflow применяет к задачам, чтобы определить, готовы ли они к выполнению, ориентируясь на их зависимости (= предшествующие задачи в ОАГ). Правило триггеров по умолчанию – это `all_success`, которое гласит, что все зависимости задачи должны быть успешно завершены, прежде чем саму задачу можно будет выполнить.

Чтобы понять, что это означает, вернемся к нашей первоначальной реализации ОАГ (рис. 5.4), в которой не используется никаких правил, кроме правила `all_success` по умолчанию. Если бы мы начали выполнение этого ОАГ, Airflow начал бы перебирать его задачи, чтобы определить, какие из них можно выполнить (т. е. у каких задач нет зависимостей, которые не были завершены успешно).

В данном случае этому условию удовлетворяет только задача `start`, у которой нет никаких зависимостей. Таким образом, Airflow приступает к выполнению ОАГ, сначала выполняя задачу `start` (рис. 5.14a). После ее успешного завершения задачи `fetch_weather` и `fetch_sales` уже готовы к выполнению, поскольку их единственная зависимость теперь удовлетворяет правилу триггеров (рис. 5.14b). Следуя этому шаблону выполнения, Airflow может продолжить выполнение оставшихся задач в ОАГ, пока тот не будет выполнен целиком.

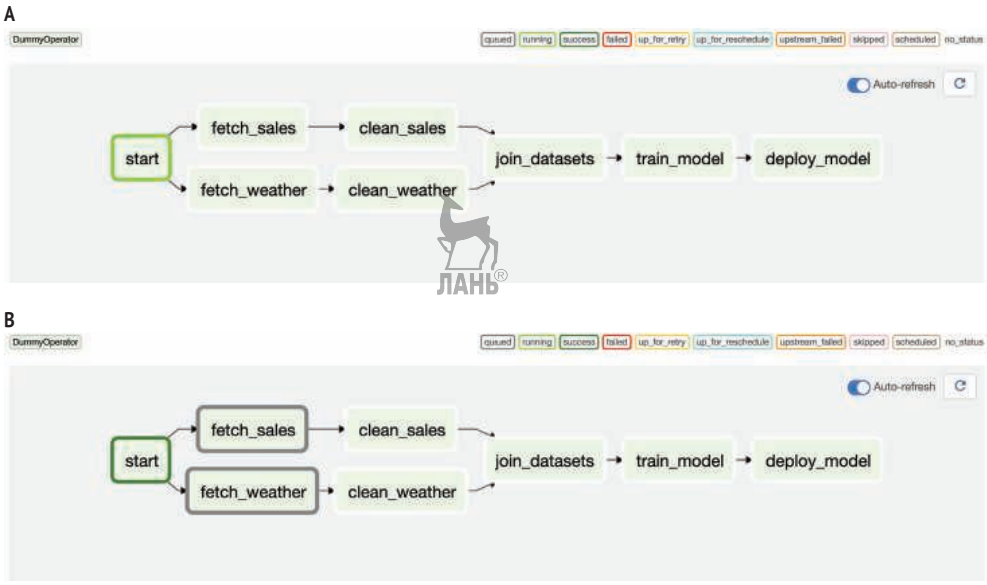


Рис. 5.14 Отслеживание выполнения базового ОАГ (рис. 5.4) с использованием правила триггеров по умолчанию, `all_success`. (A) Airflow изначально запускает выполнение ОАГ, выполняя единственную задачу, у которой нет предыдущих задач, которые не были выполнены успешно: задача `start`. (B) После ее успешного выполнения другие задачи уже готовы к выполнению, и Airflow переходит к ним

5.4.2 Эффект неудач

Конечно, это только наброски для удачной ситуации, в которой все наши задачи выполняются успешно. Что, например, произойдет, если одна из наших задач обнаружит ошибку во время выполнения?

Это легко проверить, смоделировав сбой в одной из задач. Например, моделируя неудавшуюся задачу `fetch_sales`, мы видим, что Airflow записывает сбой, назначая `fetch_sales` состояние сбоя, а не состояние успеха (рис. 5.15). Это означает, что нижестоящую задачу `clean_sales` уже нельзя выполнить, поскольку для ее успешного выполнения необходима задача `fetch_sales`. В результате задаче `clean_sales` присваивается состояние `upstream_failed`. Это указывает на то, что она не может продолжаться в результате сбоя.

Такой тип поведения, при котором результат вышестоящих задач также влияет на нижестоящие задачи, часто называется *распространением*, поскольку в этом случае сбой по восходящей *распространяется* на нижестоящие задачи. Эффекты пропущенных задач могут распространяться и по нисходящей, в результате чего все задачи после пропущенной задачи тоже пропускаются.

Подобное распространение является прямым результатом определения правила триггеров `all_success`, которое требует, чтобы все его зависимости были успешно выполнены. Таким образом, если он об-

наруживает пропуск или сбой в зависимости, у него нет другого выбора, кроме как также дать сбой, таким образом распространяя пропуск или неудачное завершение.

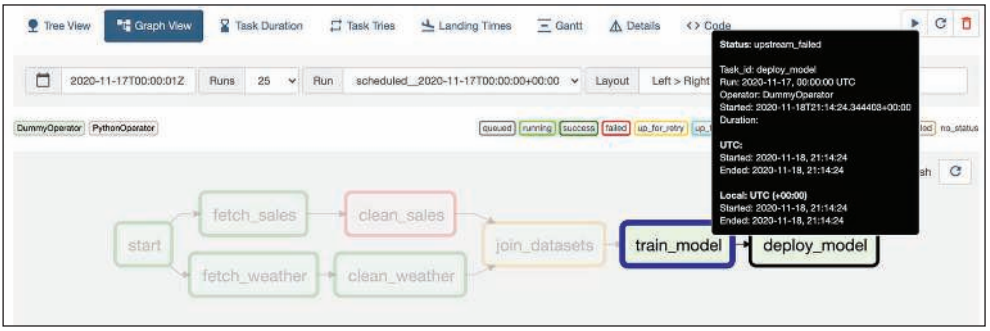


Рис. 5.15 Сбой останавливает выполнение нижестоящих задач с помощью правила триггеров по умолчанию, `all_success`, которое требует, чтобы все вышестоящие задачи были выполнены успешно. Обратите внимание, что Airflow продолжает выполнять задачи, которые не зависят от невыполненной задачи (`fetch_weather` и `clean_weather`)

5.4.3 Другие правила

Airflow поддерживает и другие правила триггеров, которые допускают различные типы поведения при ответе на успешные, неудачные или пропущенные задачи.

Например, вернемся к нашей схеме ветвления между двумя ERP-системами из раздела 5.2. В этом случае нам пришлось настроить правило триггеров для задачи, соединяющей ветки (задачи `join_datasets` или `join_erp_branch`), чтобы избежать пропуска нижестоящих задач, потому что с правилом триггеров по умолчанию пропуски, идущие из ветки, будут распространяться по нисходящей, в результате чего все задачи после ветки также будут пропущены. Правило `none_failed`, напротив, проверяет, все ли вышестоящие задачи были выполнены без сбоев. Это означает, что оно допускает как успешные, так и пропущенные задачи, по-прежнему ожидая завершения всех вышестоящих задач, перед тем как продолжить, что делает это правило подходящим для объединения двух ветвей. Обратите внимание, что с точки зрения распространения это означает, что правило не распространяет пропуски. Однако оно по-прежнему распространяет сбои, а это означает, что любые сбои в задачах `fetch/clean` по-прежнему будут останавливать выполнение нижестоящих задач.

Для обработки иных ситуаций могут использоваться другие правила. Например, правило `all_done` можно использовать для определения задач, которые выполняются, как только их зависимости завершают выполнение, независимо от результатов. Например, его можно использовать для выполнения кода очистки (например, выключения компьютера или очистки ресурсов), который вы хотели бы запустить

независимо от того, что произойдет. Еще одна категория правил включает в себя активные правила, такие как `one_failed` или `one_success`, которые не ждут завершения всех вышестоящих задач перед запуском, а требуют только одной вышестоящей задачи для выполнения своего условия. Таким образом, эти правила можно использовать, чтобы подать сигнал о раннем сбое задач или для ответа, как только одна задача из группы задач будет успешно завершена.

Хотя мы не будем здесь подробно рассматривать эти правила, мы надеемся, что это даст вам представление о роли правил триггеров в Airflow и о том, как их можно использовать для введения более сложного поведения в OAГ. Полный обзор правил триггеров и некоторых возможных вариантов их использования приводится в табл. 5.1.

Таблица 5.1 Обзор правил триггеров, поддерживаемых Airflow

Правило триггера	Поведение	Пример использования
`all_success`	Срабатывает, когда все родительские задачи были завершены успешно	Правило триггера по умолчанию для обычного рабочего процесса
`all_failed`	Срабатывает при сбое всех родительских задач (или дали сбой в результате сбоя в родительских задачах)	Запуск обработки ошибок в ситуациях, когда вы ожидали, по крайней мере, одного успешного завершения среди группы задач
`all_done`	Срабатывает, когда все родительские задачи завершили свое выполнение, независимо от их конечного состояния	Выполнение кода очистки, который вы хотите выполнить, когда все задачи будут завершены (например, выключение машины или остановка кластера)
`one_failed`	Срабатывает, как только, по крайней мере, одна родительская задача потерпела неудачу; не дожидается завершения выполнения других родительских задач	Быстрый запуск кода обработки ошибок, например уведомления или откаты
`one_success`	Срабатывает, как только одна из родительских задач завершается успешно; не дожидается завершения выполнения других родительских задач	Быстрый запуск нижестоящих вычислений или уведомлений, как только становится доступен один результат
`none_failed`	Срабатывает, если ни одна из родительских задач не была неудачной, а либо завершилась успешно, либо была пропущена	Объединение условных ветвей в OA, как показано в разделе 5.2
`none_skipped`	Срабатывает, если ни одна из родительских задач не была пропущена, а завершилась успешно или неудачно	Запуск задачи, если все вышестоящие задачи были выполнены, игнорируя их результат (результаты)
`dummy`	Срабатывает независимо от состояния любых вышестоящих задач	Тестирование

5.5 Обмен данными между задачами

Airflow также позволяет обмениваться небольшими фрагментами данных между задачами с помощью механизма XCom[1]. Идея XCom

[1] XCom – это сокращение от слова *cross-communication*.

заключается в том, что они, по сути, позволяют обмениваться сообщениями между задачами, обеспечивая некоторый уровень общего состояния.

5.5.1 Обмен данными с помощью XCom

Чтобы увидеть, как это работает, вернемся к нашему примеру с зонтами (рис. 5.3). Представьте, что при обучении нашей модели (в задаче `train_model`) модель регистрируется в реестре моделей с использованием случайно сгенерированного идентификатора. Чтобы развернуть обученную модель, нам нужно каким-то образом передать этот идентификатор задаче `deploy_model`, чтобы она знала, какую версию модели следует развернуть.

Один из способов решить эту проблему – использовать XCom, чтобы сообщить идентификатор модели задачам `train_model` и `deploy_model`. В этом случае задача `train_model` отвечает за передачу значения XCom, которое, по сути, публикует его и делает доступным для других задач. Мы можем публиковать значения XCom явно в своей задаче с помощью метода `xcom_push`, который доступен в экземпляре задачи в контексте Airflow.

Листинг 5.20 Явная передача значений Xcom с помощью метода xcom_push (dags/09_xcoms.py)

```
def _train_model(**context):
    model_id = str(uuid.uuid4())
    context["task_instance"].xcom_push(key="model_id", value=model_id)
train_model = PythonOperator(
    task_id="train_model",
    python_callable=_train_model,
)
```

Этот вызов `xcom_push` по сути сообщает Airflow о необходимости регистрации значения `model_id` в качестве значения XCom для соответствующей задачи (`train_model`) и соответствующего ОАГ и даты выполнения. После запуска этой задачи вы можете просмотреть эти опубликованные значения в веб-интерфейсе в разделе **Admin > XComs** (Администрирование > XCom) (рис. 5.16), где показан обзор всех опубликованных значений XCom.

Вы можете получить значение XCom в других задачах, используя метод `xcom_pull`, который является обратной версией метода `xcom_push`.

Листинг 5.21 Получение значений XCom с помощью метода xcom_pull (dags/09_xcoms.py)

```
def _deploy_model(**context):
    model_id = context["task_instance"].xcom_pull(
        task_ids="train_model", key="model_id"
    )
```

```
    print(f"Deploying model {model_id}")

deploy_model = PythonOperator(
    task_id="deploy_model",
    python_callable=_deploy_model,
)
```

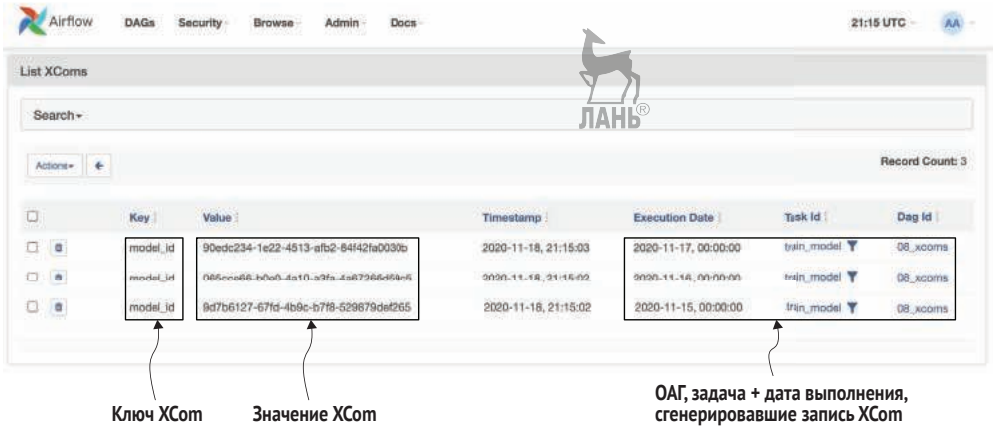


Рис. 5.16 Обзор зарегистрированных значений XCom (в разделе Admin > XComs в веб-интерфейсе)

Этот код сообщает Airflow, что нужно извлечь значение XCom с ключом `model_id` из задачи `train_model`, соответствующий `model_id`, который мы ранее поместили в задачу `train_model`. Обратите внимание, что метод `xcom_pull` также позволяет определять `dag_id` и дату выполнения при извлечении значений XCom. По умолчанию для этих параметров задан текущий ОАГ и дата выполнения, чтобы `xcom_pull` извлекал только значения, опубликованные текущим запуском ОАГ[1].

Можно проверить, как это работает, запустив ОАГ, который должен дать нам что-то наподобие следующего результата для задачи `deploy_model`:

```
[2020-07-29 20:23:03,581] {python.py:105} INFO - Exporting the following env
➥ vars:
AIRFLOW_CTX_DAG_ID=chapter5_09_xcoms
AIRFLOW_CTX_TASK_ID=deploy_model
AIRFLOW_CTX_EXECUTION_DATE=2020-07-28T00:00:00+00:00
AIRFLOW_CTX_DAG_RUN_ID=scheduled__2020-07-28T00:00:00+00:00
[2020-07-29 20:23:03,584] {logging_mixin.py:95} INFO - Deploying model
➥ f323fa68-8b47-4e21-a687-7a3d9b6e105c
[2020-07-29 20:23:03,584] {python.py:114} INFO - Done.
➥ Returned value was: None
```

[1] Вы можете указать иные значения для получения значений из других ОАГ или иные даты выполнения, но мы настоятельно рекомендуем не делать этого, если у вас нет на это очень веских причин.

Также можно ссылаться на переменные XCom в шаблонах.

Листинг 5.22 Использование значений XCom в шаблонах (dags/10_xcoms_template.py)

```
def _deploy_model(templates_dict, **context):
    model_id = templates_dict["model_id"]
    print(f"Deploying model {model_id}")

deploy_model = PythonOperator(
    task_id="deploy_model",
    python_callable=_deploy_model,
    templates_dict={
        "model_id": "{{task_instance.xcom_pull(
         ➥ task_ids='train_model', key='model_id')}}"
    },
)
```

Наконец, некоторые операторы также поддерживают автоматическую передачу значений XCom. Например, у оператора `BashOperator` есть параметр `xcom_push`, который, если для него задано значение true, сообщает оператору поместить последнюю строку, записанную в `stdout` командой bash в качестве значения XCom. Точно так же оператор `PythonOperator` опубликует любое значение, возвращаемое из вызываемого объекта Python как значение XCom. Это означает, что наш пример также можно записать следующим образом:

Листинг 5.23 Использование return (dags/11_xcoms_return.py)

```
def _train_model(**context):
    model_id = str(uuid.uuid4())
    return model_id
```

Мы регистрируем XCom с ключом по умолчанию `return_value`, что можно увидеть, заглянув в раздел администратора (рис. 5.17).

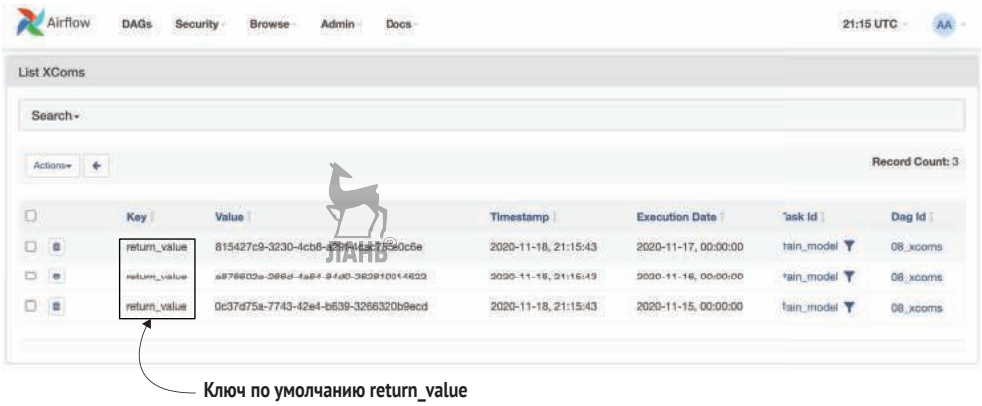


Рис. 5.17 Неявные XCom'ы из оператора `PythonOperator` регистрируются с ключом `return_value`

5.5.2 Когда (не) стоит использовать XCom

Хотя данный механизм может показаться довольно полезным для разделения состояния между задачами, его использование также имеет некоторые недостатки. Например, один из них заключается в том, что он добавляет скрытую зависимость между задачами, поскольку задача по извлечению (pulling task) неявно зависит от задачи, размещающей требуемое значение. В отличие от явных зависимостей задач, эту зависимость не видно в ОАГ, и она не будет учитываться при планировании задач. Таким образом, вы несете ответственность за то, чтобы задачи с зависимостями XCom выполнялись в правильном порядке; Airflow не сделает этого за вас. Такие скрытые зависимости становятся еще более сложными при обмене значениями XCom между разными ОАГ или датами выполнения, что также не является практикой, которой мы бы рекомендовали следовать.

Механизм XCom еще может быть своего рода антипаттерном, когда он нарушает атомарность оператора. Например, мы видели, как некоторые использовали оператор для получения API-токена в одной задаче, а затем передавали его в следующую задачу с помощью XCom. В этом случае недостаток состоял в том, что срок действия токена истекал через пару часов, а это означало, что любой повторный запуск второй задачи окончится неудачей. Возможно, лучше было бы объединить извлечение токена во второй задаче, поскольку таким образом и обновление токена, и выполнение связанной работы происходит за один раз (то есть задача остается атомарной).

Наконец, техническое ограничение XCom состоит в том, что любое значение, хранимое XCom, должно поддерживать сериализацию. Это означает, что некоторые типы Python, такие как лямбда-выражения или многие классы, связанные с многопроцессорностью, обычно нельзя сохранить в XCom (хотя вам, вероятно, все равно не стоит этого делать). Кроме того, размер значения XCom может быть ограничен бэкендом, используемым для их хранения. По умолчанию они хранятся в базе метаданных Airflow и имеют следующие ограничения по размеру:

- *SQLite* – хранятся как тип BLOB, ограничение 2 ГБ;
- *PostgreSQL* – хранятся как тип BYTEA, ограничение 1 ГБ;
- *MySQL* – хранятся как тип BLOB, ограничение 64 КБ.

При этом XCom может быть мощным инструментом при правильном использовании. Просто убедитесь, что вы тщательно продумали его использование и четко задокументировали зависимости, которые он вводит между задачами, чтобы избежать сюрпризов в будущем.

5.5.3 Использование настраиваемых XCom-бэкендов

Ограничение при использовании базы метаданных Airflow для хранения значений XCom заключается в том, что в целом оно плохо масштабируется, когда речь идет о больших объемах данных. Это озна-

чает, что обычно нужно использовать XCom для хранения отдельных значений или небольших результатов, но не для больших наборов данных.

Чтобы обеспечить бóльшую гибкость, в Airflow 2 появилась возможность указать настраиваемый XCom-бэкенд для развертывания в Airflow. По сути, эта опция позволяет определить собственный класс, который Airflow будет использовать для хранения и получения XCom'ов. Единственное требование – этот класс наследует от базового класса `BaseXCom` и реализует два статических метода для сериализации и десериализации значений соответственно.

Листинг 5.24 Каркас для настраиваемого XCom-бэкенда (lib/custom_xcom_backend.py)

```
from typing import Any
from airflow.models.xcom import BaseXCom

class CustomXComBackend(BaseXCom):

    @staticmethod
    def serialize_value(value: Any):
        ...

    @staticmethod
    def deserialize_value(result) -> Any:
        ...
```

В этом классе метод сериализации вызывается всякий раз, когда значение XCom помещается внутри оператора, тогда как метод десериализации вызывается, когда значения XCom извлекаются из бэкенда. После того как у вас появился желаемый бэкенд-класс, вы можете настроить Airflow, чтобы использовать этот класс с параметром `xcom_backend` в конфигурации Airflow.

Настраиваемые XCom-бэкенды значительно расширяют возможности для хранения значений XCom. Например, если вы хотите хранить более крупные значения в относительно дешевом и масштабируемом облачном хранилище, то можете реализовать настраиваемый бэкенд для облачных сервисов, таких как хранилище Azure Blob, Amazon S3 или Google GCS. По мере развития Airflow 2 мы ожидаем, что бэкенды для распространенных сервисов станут более общедоступными, то есть вам не нужно будет создавать для них собственные бэкенды.

5.6 Связывание задач Python с помощью Taskflow API

Хотя XCom можно использовать для обмена данными между задачами Python, API может быть громоздким в использовании, особенно если вы объединяете в цепочку большое количество задач. Чтобы ре-

шить эту проблему, Airflow 2 добавил новый API на основе декоратора для определения задач Python и их зависимостей – *Taskflow API*. Хотя и не без недостатков, Taskflow API может значительно упростить ваш код, если вы в основном используете оператор `PythonOperator` и передаете данные между ними в виде XCom'ов.

5.6.1 Упрощение задач Python с помощью Taskflow API

Чтобы увидеть, как выглядит Taskflow API, вернемся к нашим задачам по тренировке и развертыванию модели машинного обучения. В нашей предыдущей реализации эти задачи и их зависимости были определены следующим образом.

Листинг 5.25 Определение задач обучения и развертывания с использованием обычного API (dags/09_xcoms.py)

Определение функций обучения и развертывания

```
def _train_model(**context):
    model_id = str(uuid.uuid4())
    context["task_instance"].xcom_push(key="model_id", value=model_id)

def _deploy_model(**context):
    model_id = context["task_instance"].xcom_pull(
        task_ids="train_model", key="model_id"
    )
    print(f"Deploying model {model_id}")

with DAG(...) as dag:
    ...

    train_model = PythonOperator(
        task_id="train_model",
        python_callable=_train_model,
    )

    deploy_model = PythonOperator(
        task_id="deploy_model",
        python_callable=_deploy_model,
    )

    ...
    join_datasets >> train_model >> deploy_model
```

Отправка идентификатора модели с помощью механизма XCom

Создание задач обучения и развертывания с помощью оператора PythonOperator

Установка зависимостей между задачами

Недостаток этого подхода состоит в том, что сначала он требует от нас определения функции (например, `_train_model` и `_deploy_model`), которые затем нужно обернуть в `PythonOperator` для создания задачи Airflow. Более того, чтобы передавать идентификатор модели между двумя задачами, нужно явно использовать методы `xcom_push` и `xcom_pull` в функциях для отправки или получения значения идентификатора модели. Определение такой зависимости данных обременительно и может дать сбой, если мы изменим ключ передаваемого значения, на который есть ссылки в двух разных местах.

Taskflow API призван упростить определение этого типа (на основе `PythonOperator`) задачи, облегчив преобразование функций Python в задачи и сделав обмен переменными через механизм XCom между этими задачами более явным в определении ОАГ. Чтобы увидеть, как это работает, начнем с преобразования данных функций для использования этого альтернативного API.

Вначале мы можем изменить определение задачи `train_model` на относительно простую функцию Python, декорированную новым декоратором `@task`, добавленным Taskflow API.

Листинг 5.26 Определение задачи обучения с помощью Taskflow API (dags/12_taskflow.py)

```
...
from airflow.decorators import task
...

with DAG(...) as dag:
    ...
    @task
    def train_model():
        model_id = str(uuid.uuid4())
        return model_id
```

По сути, этот код дает Airflow указание обернуть функцию `train_model`, чтобы мы могли использовать ее для определения задач Python с помощью Taskflow API. Обратите внимание, что мы больше не отправляем явно идентификатор модели в виде XCom, а просто возвращаем его из функции, чтобы Taskflow API позаботился о передаче его следующей задаче.

Точно так же можно определить задачу `deploy_model`:

Листинг 5.27 Определение задачи развертывания с помощью Taskflow API (dags/12_taskflow.py)

```
@task
def deploy_model(model_id):
    print(f"Deploying model {model_id}")
```

Здесь идентификатор модели также больше не извлекается с помощью метода `xcom_pull`, а просто передается функции Python в качестве аргумента. Теперь осталось только соединить две задачи, что можно сделать, используя синтаксис, подозрительно похожий на обычный код Python.

Листинг 5.28 Определение зависимостей между задачами Taskflow (dags/12_taskflow.py)

```
model_id = train_model()
deploy_model(model_id)
```

Этот код должен привести к созданию ОАГ с двумя задачами (`train_model` и `deploy_model`) и зависимостью между двумя задачами (рис. 5.18).

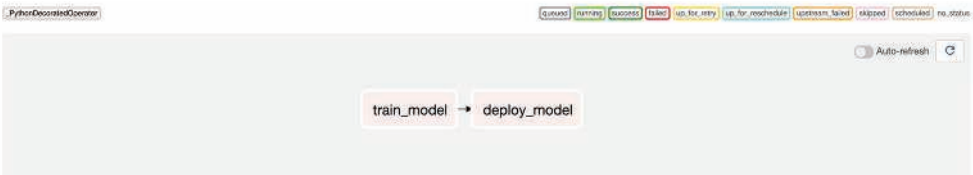


Рис. 5.18 Подмножество предыдущего ОАГ, содержащее задачи `train` и `deploy`, в которых задачи и их зависимости определены с помощью Taskflow API

Сравнивая новый код с предыдущей реализацией, видно, что подход на основе Taskflow API дает аналогичные результаты, при этом код проще читать и он больше похож на обычный код Python. Но как это работает?

По сути, когда мы вызываем декорированную функцию `train_model`, она создает новый экземпляр оператора для задачи `train_model` (показанный как `_PythonDecoratedOperator` на рис. 5.18). По оператору `return` из функции `train_model` Airflow распознает, что мы возвращаем значение, которое автоматически будет зарегистрировано в качестве XCom, возвращаемого из задачи. Для задачи `deploy_model` мы также вызываем декорированную функцию для создания экземпляра оператора, но теперь мы еще передаем вывод `model_id` из задачи `train_model`. При этом мы фактически сообщаем Airflow, что вывод `model_id` должен быть передан в качестве аргумента декорированной функции `deploy_model`. Таким образом, Airflow поймет, что между двумя задачами существует зависимость, и позаботится о передаче значений XCom между двумя задачами.

5.6.2 Когда (не) стоит использовать Taskflow API

Taskflow API обеспечивает простой подход к определению задач Python и их зависимостей с использованием синтаксиса, который ближе к использованию обычных функций Python, чем к более объектно-ориентированному API операторов. Это позволяет API значительно упростить ОАГ, которые интенсивно используют операторы Python, и передавать данные между результирующими задачами с помощью XCom'ов. API также устраняет одно из наших предыдущих критических замечаний по поводу использования механизма XCom, обеспечивая явную передачу значений между задачами, вместо того чтобы скрывать зависимости между задачами в рамках соответствующих функций.

Однако один из недостатков Taskflow API состоит в том, что его использование в настоящее время ограничено задачами Python, ко-

торые в противном случае были бы реализованы с помощью `PythonOperator`. Таким образом, задачи с участием любых других операторов Airflow потребуют использования обычного API для определения задач и их зависимостей. Хотя это не мешает смешивать и сопоставлять два этих стиля, результирующий код может вас запутать, если вы не будете осторожны. Например, при объединении новых задач по обучению и развертыванию обратно в исходный ОАГ (рис. 5.19) нужно определить зависимость между задачей `join_datasets` и ссылкой на `model_id`, что интуитивно не слишком понятно.

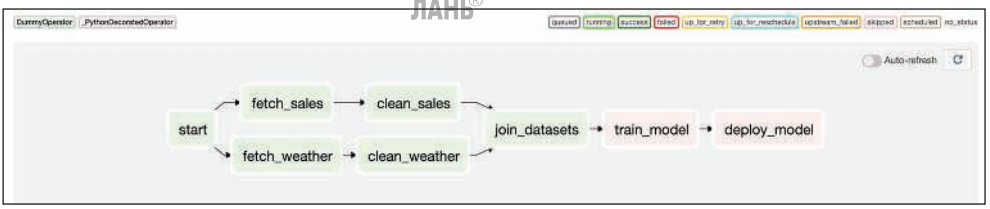


Рис. 5.19 Объединение задач обучения и развертывания в стиле Taskflow обратно в исходный ОАГ, который также содержит другие (не основанные на `PythonOperator`) операторы

Листинг 5.29 Объединение других операторов с Taskflow API (dags/13_taskflow_full.py)

```
with DAG(...) as dag:
    start = DummyOperator(task_id="start")
    ...
    [clean_sales, clean_weather] >> join_datasets

    @task
    def train_model():
        model_id = str(uuid.uuid4())
        return model_id

    @task
    def deploy_model(model_id: str):
        print(f"Deploying model {model_id}")

    model_id = train_model()
    deploy_model(model_id)

    join_datasets >> model_id
```

Определение задач и зависимостей с помощью обычного API

Использование Taskflow API для задач и зависимостей Python

Смешивание двух стилей

Все данные, передаваемые между задачами в стиле Taskflow, будут храниться в виде XCom'ов. Это означает, что все переданные значения подлежат техническим ограничениям XCom (т. е. должны быть сериализуемыми). Более того, размер наборов данных, передаваемых между задачами, может быть ограничен XCom-бэкендом, который используется при развертывании в Airflow, как обсуждалось в предыдущем разделе.

Резюме

- Зависимости базовых задач Airflow можно использовать для определения линейных зависимостей задач и структур «один-ко-многим» и «многие-к-одному» в ОАГ.
- Используя оператор `BranchPythonOperator`, можно встраивать ветви в ОАГ, что дает вам возможность выбирать несколько путей выполнения в зависимости от определенных условий.
- Используя условные задачи, можно выполнять задачи в зависимости от конкретных условий.
- Явное кодирование ветвей или условий в структуре ОАГ обеспечивает существенные преимущества с точки зрения интерпретируемости того, как выполнялся ОАГ.
- Запуск задач Airflow контролируется правилами триггеров, которые регулируют поведение и которые можно настроить таким образом, чтобы задачи могли реагировать на различные ситуации.
- Задачи могут обмениваться данными между собой с помощью механизма XCom.
- Taskflow API может помочь упростить ОАГ, интенсивно использующие задачи Python.

Часть II

За пределами основ

Теперь, когда вы познакомились с основами Airflow и можете создавать собственные конвейеры обработки данных, вы готовы к тому, чтобы перейти к изучению более продвинутых методов, позволяющих создавать более сложные кейсы с участием внешних систем, собственных компонентов и т. д.

В главе 6 мы рассмотрим, как запускать конвейеры способами, не связанными с фиксированным расписанием. Это позволяет запускать конвейеры в ответ на определенные события, такие как поступление новых файлов или вызов HTTP-службы.

В главе 7 будет показано, как использовать встроенные функции Airflow для запуска задач во внешних системах. Это чрезвычайно мощная функция Airflow, позволяющая создавать конвейеры, координирующие потоки данных во многих системах, таких как базы данных, вычислительные фреймворки, допустим Apache Spark, и системы хранения.

Далее в главе 8 будет показано, как создавать собственные компоненты для Airflow, позволяющие выполнять задачи в системах, не поддерживаемых встроенными функциями Airflow. Эту функциональность также можно использовать для создания компонентов, которые можно с легкостью повторно использовать в своих конвейерах для поддержки распространенных рабочих процессов.

Чтобы повысить надежность конвейеров, в главе 9 подробно рассматриваются различные стратегии, которые можно использовать для тестирования конвейеров обработки данных и собственных ком-

понентов. Эта тема часто поднимается на семинарах, посвященных Airflow, поэтому мы потратим некоторое время на ее изучение.

Наконец, в главе 10 рассматривается использование подходов на базе контейнеров для реализации задач в конвейерах. Мы покажем вам, как запускать задачи, используя Docker и Kubernetes, и обсудим преимущества и недостатки применения контейнеров. После завершения части II вы должны быть на пути к тому, чтобы стать продвинутым пользователем Airflow, имея возможность писать сложные (и тестируемые) конвейеры, которые необязательно включают собственные компоненты и/или контейнеры. Однако в зависимости от ваших интересов вы можете выбрать конкретные главы, на которых нужно сосредоточиться, поскольку не все из них могут быть актуальными для вашего варианта.

6 Запуск рабочих процессов

Эта глава рассказывает:

- об ожидании выполнения определенных условий с использованием сенсоров;
- о задании зависимостей между задачами в разных ОАГ;
- о выполнении рабочих процессов через интерфейс командной строки и REST API.

В главе 3 мы изучили, как спланировать рабочие процессы в Airflow на основе временного интервала. Временные интервалы можно указать в виде удобных строк (например, «`@daily`"), объектов timedelta (например, `timedelta(days=3)`) или строк cron (например, `"30 14 * * *"`). Все это обозначения, указывающие на то, что рабочий процесс запускается в определенное время или через определенный интервал. Airflow вычислит, когда в следующий раз должен запускаться рабочий процесс с учетом интервала, и запустит первую (первые) задачу (задачи) в рабочем процессе в следующую дату и время.

В этой главе мы рассмотрим другие способы запуска рабочих процессов. Это часто требуется после определенного действия, в отличие от временных интервалов, когда рабочие процессы запускаются в заранее определенное время. Действия, связанные с запуском, часто являются результатом внешних событий; представьте себе файл, который загружается на общий диск, разработчика, который помещает

свой код в репозиторий или раздел в таблице Hive. Что угодно из вышеперечисленного может стать причиной, чтобы приступить к запуску рабочего процесса.

6.1 Опрос условий с использованием сенсоров

Один из распространенных вариантов запуска рабочего процесса – это поступление новых данных; представьте, что третья сторона осуществляет доставку ежедневного дампа данных в общей системе хранения данных между собой и вашей компанией. Предположим, что мы разрабатываем популярное мобильное приложение для использования купонов и контактируем со всеми брендами супермаркетов, чтобы предоставлять ежедневный экспорт их рекламных акций, которые будут отображаться в нашем приложении. В настоящее время рекламные акции в основном проводятся вручную: в большинстве супермаркетов работают аналитики по ценообразованию, которые принимают во внимание многие факторы и проводят точные рекламные акции. Некоторые акции хорошо продуманы на недели вперед, а некоторые представляют собой спонтанные однодневные флеш-распродажи. Аналитики внимательно изучают конкурентов, и иногда акции проводятся поздно вечером. Следовательно, данные о ежедневных рекламных акциях часто поступают в случайные периоды времени. Мы видели, как данные поступают в общее хранилище с 16:00 до 2:00 следующего дня, хотя ежедневные данные могут быть доставлены в любое время суток.

Давайте разработаем исходную логику для такого рабочего процесса (рис. 6.1).

Рис. 6.1 Исходная логика обработки данных об акциях супермаркета

В этом рабочем процессе мы копируем данные, предоставленные супермаркетами (1–4), в собственное хранилище необработанных данных, из которого мы всегда можем воспроизвести результаты. Затем задачи `process_supermarket_{1,2,3,4}` преобразуют и сохра-

нят все необработанные данные в базе данных результатов, который приложение может прочитать. И наконец, задача `create_metrics` вычисляет и объединяет ряд показателей, которые дают представление о рекламных акциях для дальнейшего анализа.

Поскольку данные из супермаркетов поступают в разное время, хронология этого рабочего процесса может выглядеть так, как показано на рис. 6.2.

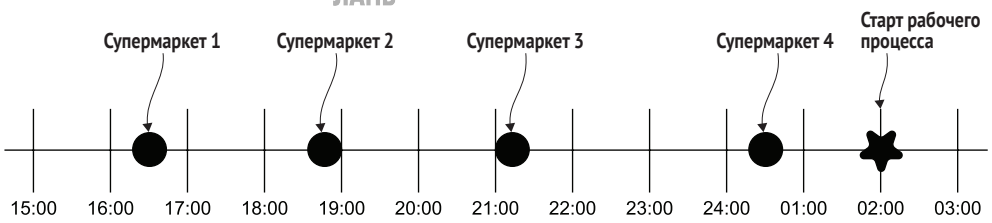


Рис. 6.2 Хронология обработки данных рекламной акции супермаркета

Здесь мы видим время доставки данных по супермаркетам и время запуска нашего рабочего процесса. Поскольку мы видели, что супермаркеты доставляют данные только в 2:00, безопаснее было бы начинать рабочий процесс в 2:00, чтобы убедиться, что все супермаркеты доставили свои данные. Однако это приводит к большому времени ожидания. Супермаркет 1 доставил свои данные в 16:30, в то время как рабочий процесс начинает обработку в 2:00, но ничего не делает 9 с половиной часов (рис. 6.3).

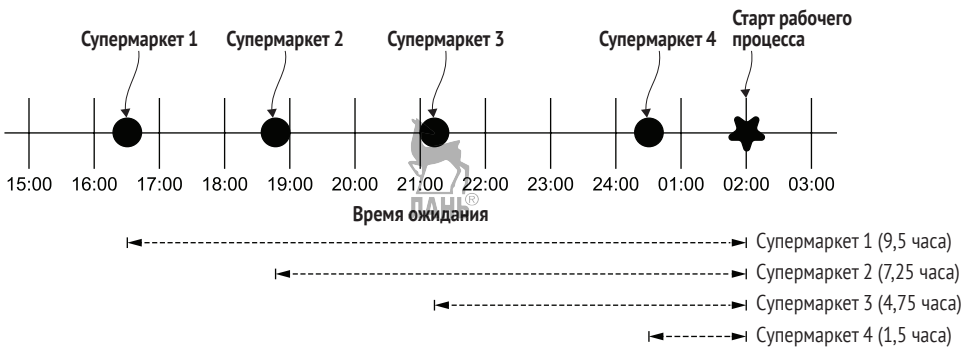


Рис. 6.3 Хронология рабочего процесса рекламной акции супермаркета с указанием времени ожидания

Один из способов решить эту проблему в Airflow – использовать *сенсоры*, представляющие собой особый тип (подкласс) операторов. Сенсоры непрерывно опрашивают определенные условия, чтобы определить их истинность, и если условие истинно, то все успешно. В противном случае сенсор будет ждать и повторять попытку до тех пор, пока условие не будет истинно, или время ожидания истечет.

Листинг 6.1 FileSensor ожидает существования пути к файлу

```
from airflow.sensors.filesystem import FileSensor

wait_for_supermarket_1 = FileSensor(
   task_id="wait_for_supermarket_1",
   filepath="/data/supermarket1/data.csv",
)
```

Здесь `FileSensor` выполняет проверку на предмет наличия файла /data/supermarket1/data.csv и возвращает `true`, если файл существует. В противном случае возвращается `false`, и сенсор будет ждать в течение заданного периода (по умолчанию 60 секунд) и повторит попытку. У операторов (сенсоры – это тоже операторы) и ОАГ есть настраиваемые тайм-ауты, и сенсор продолжит проверку условия, пока не истечет время ожидания. Вывод сенсоров можно посмотреть в журналах задач:

```
{file_sensor.py:60} INFO - Poking for file /data/supermarket1/data.csv
{file_sensor.py:60} INFO - Poking for file /data/supermarket1/data.csv
{file_sensor.py:60} INFO - Poking for file /data/supermarket1/data.csv
{file_sensor.py:60} INFO - Poking for file /data/supermarket1/data.csv
{file_sensor.py:60} INFO - Poking for file /data/supermarket1/data.csv
```

Здесь мы видим, что примерно раз в минуту[1] сенсор осуществляет *покинг* на предмет наличия определенного файла. Покинг – так в Airflow называется запуск сенсора и проверка условия.

При включении сенсоров в этот рабочий процесс следует внести одно изменение. Теперь, когда мы знаем, что не будем ждать до 2:00 и полагать, что все данные доступны, а вместо этого начнем делать все непрерывно, проверяя, доступны ли данные, время запуска ОАГ должно быть помещено в начало границ поступления данных (рис. 6.4).

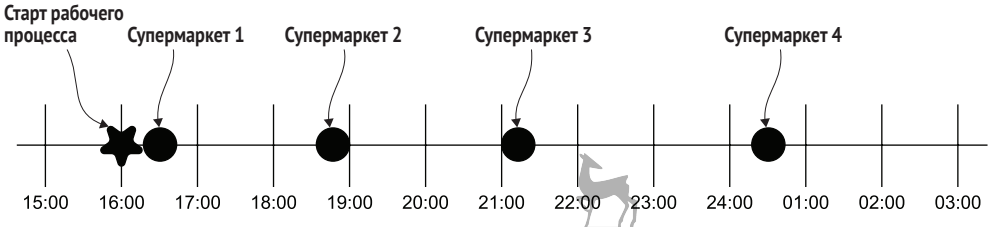


Рис. 6.4 Хронология рекламных акций супермаркетов с сенсорами

У соответствующего ОАГ будет задача (`FileSensor`), добавленная в начало обработки данных каждого супермаркета. Это будет выглядеть, как показано на рис. 6.5.

[1] Настраивается аргументом `poke_interval`.

Рис. 6.5 ОАГ рекламной акции супермаркета с сенсорами

На рис. 6.5 сенсоры были добавлены в начало ОАГ, а `schedule_interval` ОАГ был установлен на запуск *до* ожидаемой доставки данных. Таким образом, сенсоры в начале ОАГ будут непрерывно выполнять опрос на предмет наличия данных и переходить к следующей задаче после выполнения условия (т. е. как только данные будут доступны по заданному пути).

Здесь мы видим, что супермаркет 1 уже доставил данные. Таким образом, состояние соответствующего сенсора установлено как успешное, и обработка нижестоящих задач продолжается. В результате данные были обработаны сразу после доставки, без излишнего ожидания окончания предполагаемого времени доставки.

6.1.1 Опрос пользовательских условий

Некоторые наборы данных имеют большой размер и состоят из нескольких файлов (например, data-01.csv, data-02.csv, data-03.csv и т. д.). `FileSensor` поддерживает подстановочные знаки для соответствия, например, data-*.csv, что соответствует любому файлу, совпадающему с шаблоном. Так, если, допустим, первый файл data-01.csv был доставлен, а другие все еще загружаются в общее хранилище супермаркетом, `FileSensor` вернет `true`, и рабочий процесс продолжит выполнение задачи `copy_to_raw`, что нежелательно.

Поэтому мы договорились с супермаркетами записывать файл с именем _SUCCESS в качестве последней части загрузки, чтобы указать, что был загружен полный набор ежедневных данных. Группа, работающая с данными, решила, что им нужно проверить наличие одного или нескольких файлов с именем data-*.csv и одного файла с именем _SUCCESS. Под капотом `FileSensor` использует шаблоны поиска (https://ru.wikipedia.org/wiki/Шаблон_поиска) для сопоставления шаблонов с именами файлов или каталогов. Используя шаблоны поиска (они похожи на регулярные выражения, но более ограничены по функциональности) можно было бы сопоставить несколько шаблонов со сложным шаблоном, однако есть более читабельный подход – реализовать две проверки с помощью `PythonSensor`.

PythonSensor похож на PythonOperator в том смысле, что вы предоставляете вызываемый объект Python (функция, метод и т. д.), который нужно выполнить. Однако вызываемый объект PythonSensor ограничен возвратом логического значения: true, чтобы указать, что условие выполнено успешно, и false, чтобы указать, что это не так. Посмотрим, как вызываемый объект PythonSensor выполняет проверку этих двух условий.

Листинг 6.2 Реализация собственного условия с помощью PythonSensor

```
from pathlib import Path

from airflow.sensors.python import PythonSensor

def _wait_for_supermarket(supermarket_id):
    supermarket_path = Path("/data/" + supermarket_id)
    data_files = supermarket_path.glob("data-*.csv")
    success_file = supermarket_path / "_SUCCESS"
    return data_files and success_file.exists()


wait_for_supermarket_1 = PythonSensor(
    task_id="wait_for_supermarket_1",
    ython_callable=_wait_for_supermarket,
    op_kwargs={"supermarket_id": "supermarket1"},
    dag=dag,
)
```

Инициализируем объект Path

Собираем файлы data-*.csv

Собираем файл _SUCCESS

Используем return, если данные и файлы SUCCESS существуют

Вызываемый объект, предоставленный PythonSensor, выполняется и, как и ожидается, возвращает логическое значение true или false. Вызываемый объект, показанный в листинге 6.2, теперь проверяет два условия, а именно существуют ли данные и файл success. За исключением использования другого цвета, задачи PythonSensor отображаются в пользовательском интерфейсе так же (рис. 6.6).

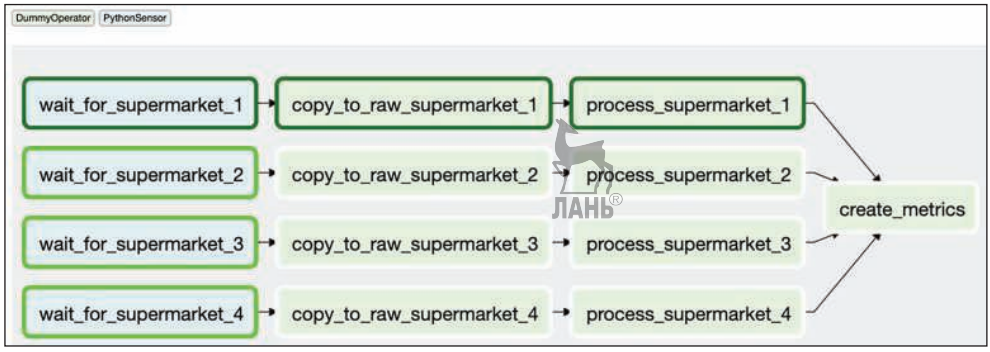


Рис. 6.6 ОАГ рекламных акций супермаркета с использованием PythonSensor для пользовательских условий

6.1.2 Использование сенсоров в случае сбоя

Теперь, когда мы убедились, что сенсоры работают успешно, что произойдет, если однажды супермаркет не предоставит свои данные? По умолчанию сенсоры будут давать сбой, как и операторы (рис. 6.7).

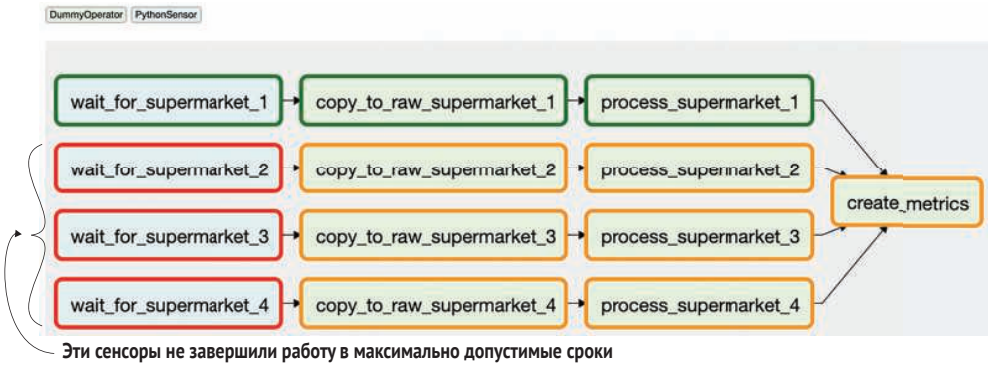


Рис. 6.7 Сенсоры, превышающие максимальный таймфрейм, дадут сбой

Сенсоры принимают аргумент `timeout`, который содержит максимальное количество секунд, в течение которого сенсор может работать. Если в начале очередного покинга количество этих секунд превысит число, заданное для `timeout`, то это приведет к сбою сенсора:

```
INFO - Poking callable: <function wait_for_supermarket at 0x7fb2aa1937a0>
INFO - Poking callable: <function wait_for_supermarket at 0x7fb2aa1937a0>
ERROR - Snap. Time is OUT.
Traceback (most recent call last):
  ➥ File "/usr/local/lib/python3.7/site-
     packages/airflow/models/taskinstance.py", line 926, in _run_raw_task
    result = task_copy.execute(context=context)
  ➥ File "/usr/local/lib/python3.7/site-
     packages/airflow/sensors/base_sensor_operator.py", line 116, in execute
    raise AirflowSensorTimeout('Snap. Time is OUT.')
airflow.exceptions.AirflowSensorTimeout: Snap. Time is OUT.
INFO - Marking task as FAILED.
```

По умолчанию тайм-аут сенсора установлен на семь дней. Если для `schedule_interval` задано значение раз в день, то это приведет к нежелательному эффекту снежного кома, с которым на удивление легко столкнуться при большом количестве ОАГ! ОАГ запускается раз в день, а супермаркеты 2, 3 и 4 дадут сбой через семь дней, как показано на рис. 6.7. Однако новые запуски ОАГ добавляются каждый день, и на эти дни запускаются сенсоры, в результате чего появляется все больше и больше задач. Вот в чем загвоздка: количество задач (на разных уровнях), которые Airflow может обработать и будет запускать, ограничено.

Важно понимать, что в Airflow есть ограничения на максимальное количество выполняемых задач на разных уровнях; количество задач на каждый ОАГ, количество задач на глобальном уровне Airflow, количество запусков ОАГ на каждый ОАГ и т. д. На рис. 6.8 видно 16 запущенных задач (все они являются сенсорами). У класса dag есть аргумент `concurrency`, определяющий, сколько параллельных задач разрешено в рамках этого ОАГ.

Листинг 6.3 Задаем максимальное количество параллельных задач

```
dag = DAG(
   dag_id="couponing_app",
   start_date=datetime(2019, 1, 1),
   schedule_interval="0 0 * * *",
   concurrency=50,
)
```

Этот код позволяет параллельно запускать 50 задач

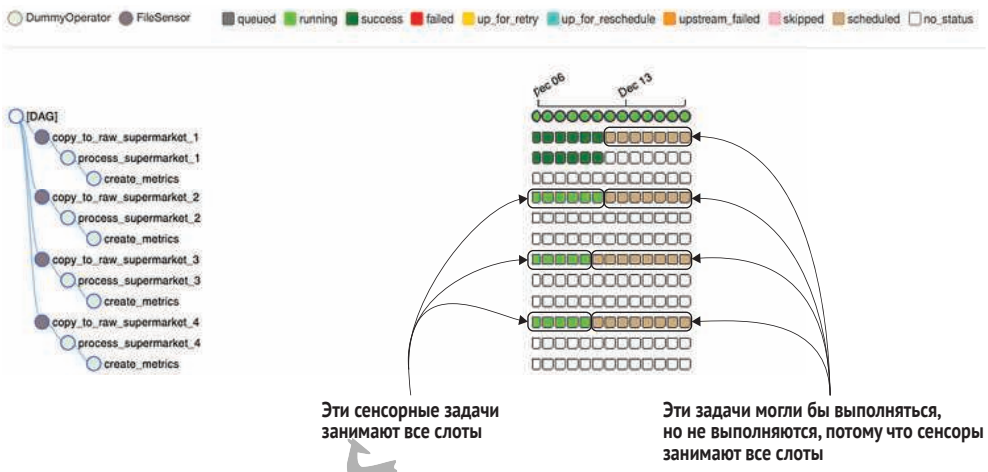


Рис. 6.8 Тупик сенсора: все выполняемые задачи – это сенсоры, ожидающие выполнения условия, чего никогда не произойдет, и, таким образом, они занимают все слоты

На рис. 6.8 мы запустили ОАГ со всеми значениями по умолчанию, а это 16 параллельных задач на каждый ОАГ. Произошел эффект снежного кома:

- день 1: супермаркет 1 – все успешно; супермаркеты 2, 3 и 4 – опрос, 3 задания заняты;
- день 2: супермаркет 1 – все успешно; супермаркеты 2, 3 и 4 – опрос, 6 заданий заняты;
- день 3: супермаркет 1 – все успешно; супермаркеты 2, 3 и 4 – опрос, 9 заданий заняты;
- день 4: супермаркет 1 – все успешно; супермаркеты 2, 3 и 4 – опрос, 12 заданий заняты;
- день 5: супермаркет 1 – все успешно; супермаркеты 2, 3 и 4 – опрос, 15 заданий заняты;

- день 6: супермаркет 1 – все успешно; супермаркеты 2, 3 и 4 – опрос, 16 заданий заняты; две новые задачи нельзя запустить, и любая другая задача, пытающаяся запуститься, блокируется.

Такое поведение часто называют *тупиком сенсора*. В этом примере достигается максимальное количество запущенных задач в ОАГ, и, таким образом, влияние ограничивается этим ОАГ, в то время как другие ОАГ все еще могут работать. Однако как только будет достигнут глобальный предел максимального количества задач, вся ваша система встанет, что явно нежелательно. Решить эту проблему можно разными способами.

Класс сенсора принимает аргумент `mode`, для которого можно задать значение `poke` или `reschedule` (начиная с Airflow версии 1.10.2). По умолчанию задано значение `poke`, что приводит к блокировке. Это означает, что задача сенсора занимает слот, пока выполняется. Время от времени она выполняет покинг, осуществляя проверку условия, а затем ничего не делает, но по-прежнему занимает слот. Режим сенсора `reschedule` освобождает слот после заверения покинга, поэтому слот занят, *только* пока выполняется работа (рис. 6.9).

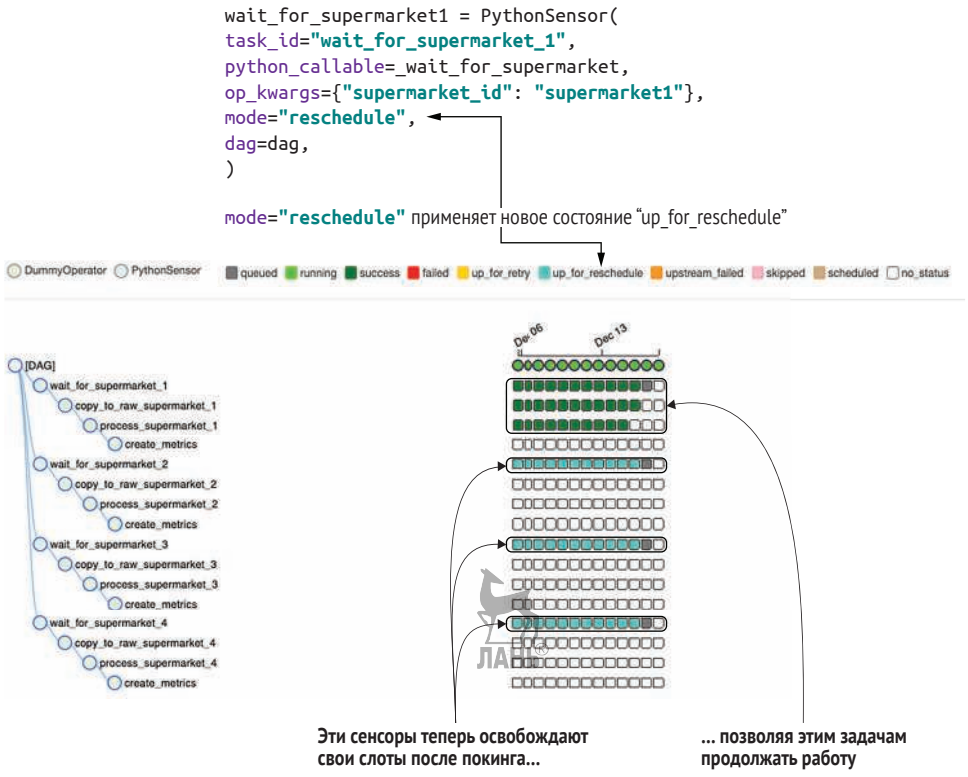


Рис. 6.9 Сенсоры с `mode="reschedule"` освобождают свой слот после покинга, позволяя запускать другие задачи

Количество одновременных задач также можно контролировать с помощью нескольких параметров конфигурации на глобальном уровне Airflow, которые рассматриваются в разделе 12.6. В следующем разделе мы посмотрим, как разделить один ОАГ на несколько ОАГ меньшего размера, которые запускают друг друга, чтобы разделить ответственности.

6.2 Запуск других ОАГ

В какой-то момент к нашему сервису по выдаче купонов добавляется больше супермаркетов. Все больше и больше людей хотели бы получать информацию об акциях супермаркетов, и этап `create_metrics` в конце выполняется только один раз в день после того, как данные всех супермаркетов были доставлены и обработаны. В текущей настройке все зависит от успешного состояния задач `process_supermarket_{1,2,3,4}` (рис. 6.10).

Рис. 6.10 Разная логика выполнения между задачами для конкретного супермаркета, и задача `create_metrics` указывает на потенциальное разделение на отдельные ОАГ

Мы получили вопрос от группы аналитиков относительно того, можно ли сделать показатели доступными сразу после обработки, вместо того чтобы ждать, пока другие супермаркеты доставят свои данные и запустят их по конвейеру. Здесь у нас есть несколько вариантов (в зависимости от выполняемой логики). Мы могли бы установить задачу `create_metrics` как нижестоящую после каждой задачи `process_supermarket_*` (рис. 6.11).

Предположим, что задача `create_metrics` превратилась в несколько задач, что сделало структуру ОАГ более сложной и привело к увеличению количества повторяющихся задач (рис. 6.12).

Один из способов обойти повторяющиеся задачи с (почти) равной функциональностью – разделить ОАГ на несколько более мелких ОАГ, каждый из которых берет на себя часть общего рабочего процесса. Одно из преимуществ такого способа заключается в том, что вы мо-

жете вызывать ОАГ 2 несколько раз из ОАГ 1 вместо одного ОАГ, содержащего несколько (дублированных) задач из ОАГ 2. Возможно ли это или желательно, зависит от многих факторов, таких как сложность рабочего процесса. Если, например, вы хотите иметь возможность создавать метрики, не дожидаясь завершения рабочего процесса в соответствии с его расписанием, а вместо этого запускать его вручную, когда захотите, то имеет смысл разделить его на два отдельных ОАГ.

Рис. 6.11 Репликация задач, чтобы не ждать завершения всех процессов

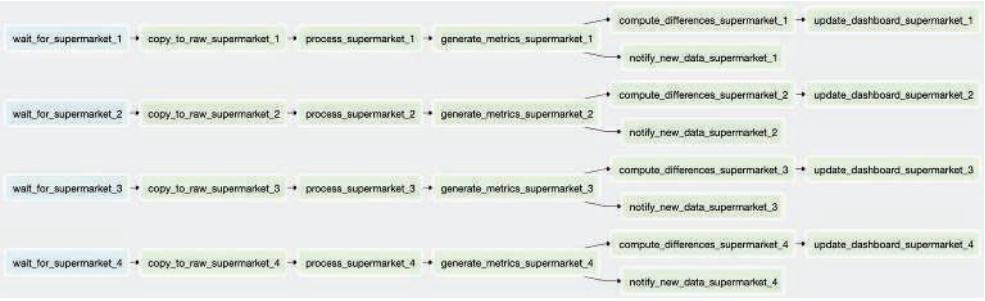


Рис. 6.12 Дополнительная логика снова указывает на возможное разделение на отдельные ОАГ

Данный сценарий можно реализовать с помощью оператора `TriggerDagRunOperator`. Он позволяет запускать другие ОАГ, которые можно применять для разделения частей рабочего процесса.

Листинг 6.4 Запуск других ОАГ с помощью TriggerDagRunOperator

```
import airflow.utils.dates
from airflow import DAG
from airflow.operators.dummy import DummyOperator
from airflow.operators.trigger_dagrun import TriggerDagRunOperator

dag1 = DAG(
   dag_id="ingest_supermarket_data",
   start_date=airflow.utils.dates.days_ago(3),
   schedule_interval="0 16 * * *",
)
```

```
for supermarket_id in range(1, 5):
    # ...
    trigger_create_metrics_dag = TriggerDagRunOperator (
        task_id=f"trigger_create_metrics_dag_supermarket_{supermarket_id}",
        trigger_dag_id="create_metrics",
        dag=dag1,
    )

dag2 = DAG(
    dag_id="create_metrics",
    start_date=airflow.utils.dates.days_ago(3),
    schedule_interval=None,
)
# ...
```

dag_id должны совпадать

schedule_interval присваивается значение None

Строка, предоставленная аргументу `trigger_dag_id` оператора `TriggerDagRunOperator`, должна соответствовать `dag_id` ОАГ, который должен быть запущен. В конечном результате теперь у нас есть два ОАГ: один для приема данных из супермаркетов, второй для вычисления показателей по данным (рис. 6.13).

Рис. 6.13 ОАГ поделены на два, причем ОАГ 1 запускает ОАГ 2 с помощью `TriggerDagRunOperator`. Логика в ОАГ 2 теперь определяется только один раз, что упрощает ситуацию, показанную на рис. 6.12

Визуально в пользовательском интерфейсе Airflow почти нет разницы между ОАГ, запускаемым по расписанию, ОАГ, запускаемым вручную, или ОАГ, запускаемым автоматически. Две маленькие детали в древовидном представлении сообщают вам, был ли ОАГ запущен по расписанию. Во-первых, запланированные запуски ОАГ и их экземпляры задач обведены черной рамкой (рис. 6.14).

Во-вторых, каждый запуск ОАГ содержит поле `run_id`. Значение `run_id` начинается с одного из следующих префиксов:

- `scheduled__`, чтобы указать, что запуск ОАГ начался по расписанию;
- `backfill__`, чтобы указать, что ОАГ запускается с использованием обратного заполнения;
- `manual__`, чтобы указать, что ОАГ запускается вручную (например, при нажатии кнопки **Trigger Dag** или инициируется оператором `TriggerDagRunOperator`).

При наведении курсора на кружок запуска ОАГ отображается всплывающая подсказка, показывающая значение `run_id`, сообщая нам, как был запущен ОАГ (рис. 6.15).

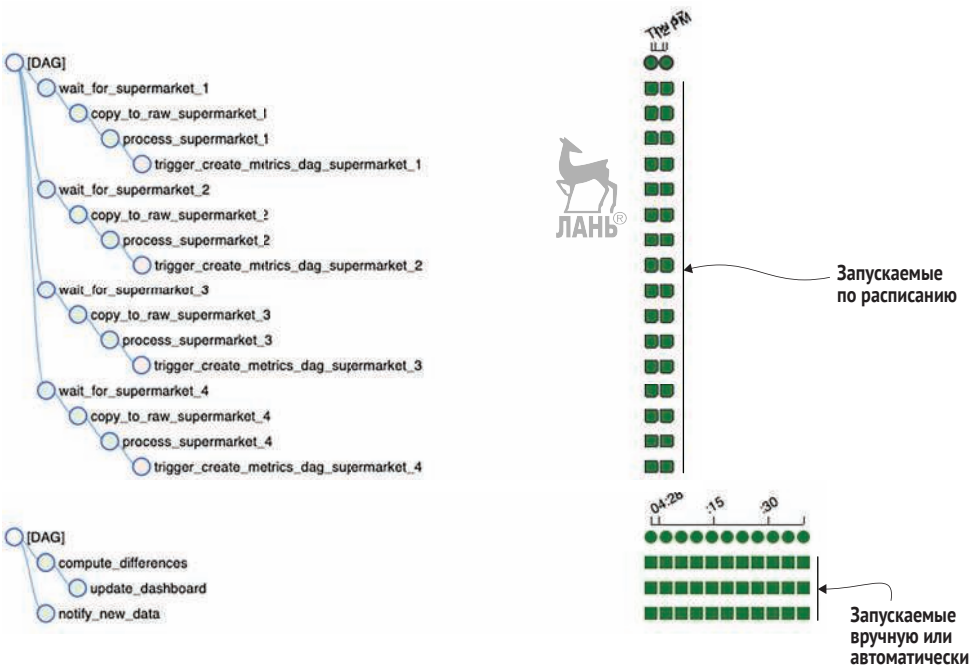


Рис. 6.14 Черная рамка означает запуск по расписанию; отсутствие рамки указывает на запуск вручную или автоматически

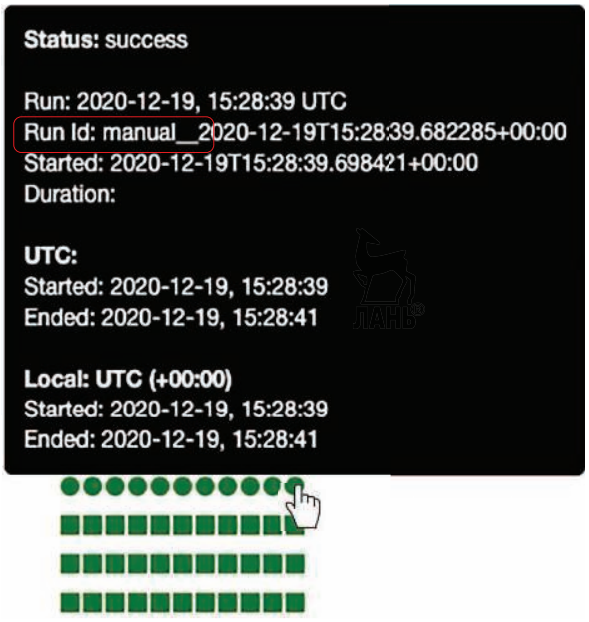


Рис. 6.15 `run_id` сообщает нам источник запуска DAG

6.2.1 Обратное заполнение с помощью оператора TriggerDagRunOperator

Что делать, если вы изменили логику в задачах `process_*` и захотели перезапустить ОАГ оттуда? В одном ОАГ можно очистить состояние `process_*` и соответствующих нижестоящих задач. Однако при очистке задач удаляются *только* задачи в пределах одного и того же ОАГ. Задачи, идущие после `TriggerDagRunOperator` в другом ОАГ, не очищаются, поэтому следует учитывать такое поведение.

Очистка задач в ОАГ, включая `TriggerDagRunOperator`, вызовет новый запуск ОАГ вместо очистки соответствующих ранее инициированных запусков (рис. 6.16).

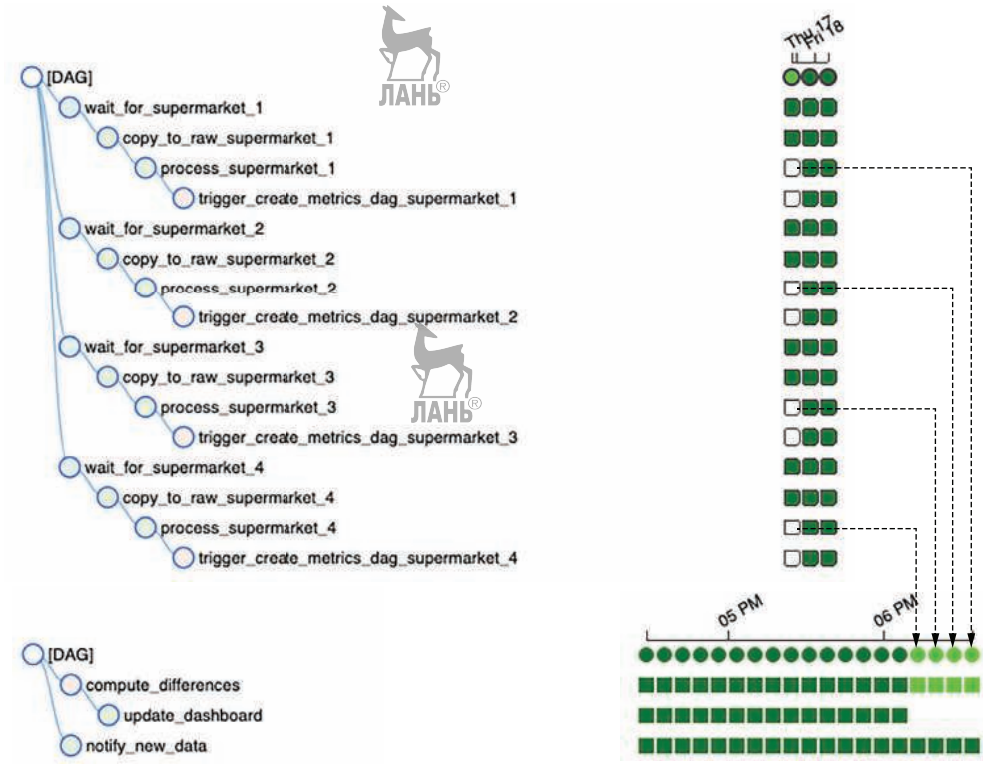


Рис. 6.16 Очистка `TriggerDagRunOperators` не очищает задачи в инициированном ОАГ; вместо этого создаются новые запуски ОАГ

6.2.2 Опрос состояния других ОАГ

Пример на рис. 6.13 работает до тех пор, пока нет зависимости от инициируемых ОАГ обратно к инициирующему ОАГ. Другими словами, первый ОАГ может инициировать нижестоящий ОАГ всякий раз, без необходимости проверять какие-либо условия.

Если ОАГ становятся очень сложными, чтобы внести ясность, первый ОАГ можно разделить на несколько ОАГ, а соответствующую задачу `TriggerDagRunOperator` можно выполнить для каждого соответствующего ОАГ, как показано на рис. 6.17 посередине. Кроме того, один ОАГ, инициирующий несколько исходящих ОАГ, является возможным сценарием при использовании `TriggerDagRunOperator`, как показано на рис. 6.17 справа.

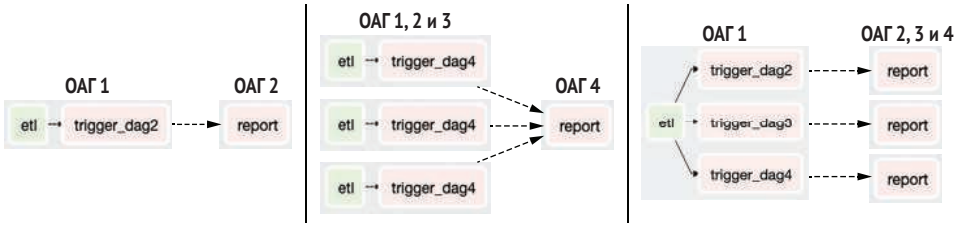


Рис. 6.17 Различные зависимости между ОАГ возможны с помощью `TriggerDagRunOperator`

Но что, если несколько инициирующих ОАГ должны завершиться, прежде чем другой ОАГ сможет начать работу? Например, что, если ОАГ 1, 2 и 3 извлекают, преобразуют и загружают набор данных, а вы хотите запускать ОАГ 4 *только* после завершения всех трех ОАГ, например чтобы вычислить набор агрегированных метрик? Airflow управляет зависимостями между задачами в рамках одного ОАГ; однако он *не* предоставляет механизма для зависимостей между ОАГ (рис. 6.18)[1].

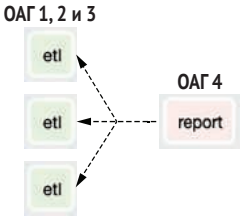


Рис. 6.18 Иллюстрация зависимости между ОАГ, которую нельзя решить с помощью `TriggerDagRunOperator`

В этой ситуации можно бы применить `ExternalTaskSensor`, представляющий собой сенсор, осуществляющий покинг состояния задач в других ОАГ, как показано на рис. 6.19. Таким образом, задачи `wait_for_etl_dag{1,2,3}` действуют как прокси, чтобы гарантировать завершенное состояние всех трех ОАГ перед окончательным выполнением задачи report.

[1] Этот плагин Airflow визуализирует зависимости между ОАГ, сканируя все ваши ОАГ на предмет использования `TriggerDagRunOperator` и `ExternalTaskSensor`: https://github.com/ms32035/airflow-dag-dependencies.

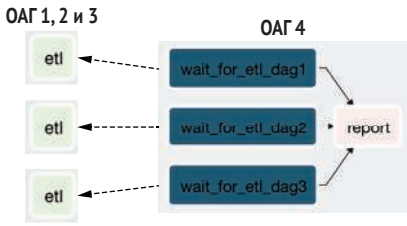


Рис. 6.19 Вместо того чтобы размещать выполнение с помощью `TriggerDagRunOperator`, в некоторых ситуациях, таких как обеспечение завершенного состояния для ОАГ 1, 2 и 3, нужно извлечь выполнение к ОАГ 4 с помощью `ExternalTaskSensor`

Принцип работы `ExternalTaskSensor` показан на рис. 6.20.

```
import airflow.utils.dates
from airflow import DAG
from airflow.operators.dummy import DummyOperator
from airflow.sensors.external_task import ExternalTaskSensor

dag1 = DAG(dag_id="ingest_supermarket_data", schedule_interval="0 16 * * *", ...)
dag2 = DAG(schedule_interval="0 16 * * *", ...)

DummyOperator(task_id="copy_to_raw", dag=dag1) >> DummyOperator(task_id="process_supermarket", dag=dag1)

wait = ExternalTaskSensor(
    task_id="wait_for_process_supermarket",
    external_dag_id="ingest_supermarket_data",
    external_task_id="process_supermarket",
    dag=dag2,
)
report = DummyOperator(task_id="report", dag=dag2)
wait >> report
```

Рис. 6.20 Пример использования `ExternalTaskSensor`

Поскольку от ОАГ 1 до ОАГ 2 событий нет, ОАГ 2 опрашивает состояние задачи в ОАГ 1, но здесь есть ряд недостатков. В мире Airflow одни ОАГ не имеют понятия о других ОАГ. Хотя технически возможно выполнить запрос к основной базе метаданных (что и делает `ExternalTaskSensor`) или считывать сценарии ОАГ с диска и делать выводы о деталях выполнения других рабочих процессов, в Airflow они никак не связаны. В случае использования `ExternalTaskSensor` требуется совпадение между ОАГ. По умолчанию `ExternalTaskSensor` просто проверяет наличие успешного состояния задачи с точно такой же датой выполнения. Таким образом, если `ExternalTaskSensor` запускается с датой выполнения 2019-10-12T18: 00: 00, он будет выполнять запрос к базе метаданных Airflow для данной задачи, также с датой выполнения 2019-10-12T18: 00: 00. Теперь предположим, что у обоих ОАГ разный интервал; тогда ни о каком совпадении не может быть и речи, и, следовательно, `ExternalTaskSensor` так и не найдет соответствующую задачу (рис. 6.21)!

```
from airflow  import DAG
from airflow.operators.dummy  import DummyOperator
from airflow.sensors.external_task  import ExternalTaskSensor
dag1 = DAG(dag_id="dag1", schedule_interval="0 16 * * *")
dag2 = DAG(dag_id="dag2", schedule_interval="0 20 * * *")
DummyOperator(task_id="etl", dag=dag1)
ExternalTaskSensor(task_id="wait_for_etl", external_dag_id="dag1", external_task_id="etl", dag=dag2)
```

schedule_interval не совпадают, поэтому ExternalTaskSensor не найдет соответствующую задачу

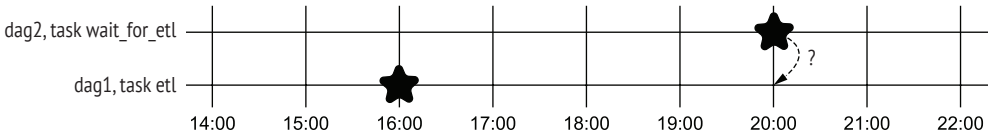


Рис. 6.21 `ExternalTaskSensor` проверяет завершение задачи в другом ОАГ, следуя собственному `schedule_interval`, который так и не будет найден, если интервалы не совпадут

В случае если интервалы не совпадают, можно прибегнуть к смещению, по которому `ExternalTaskSensor` должен искать задачу в другом ОАГ. Это смещение контролируется аргументом `execution_delta` в `ExternalTaskSensor`. Он ожидает объект `timedelta`, и важно знать, что он работает вразрез с вашими ожиданиями. `timedelta` вычитается из `execution_date`, а это означает, что положительная `timedelta` фактически оглядывается назад в прошлое (рис. 6.22).

```
from airflow  import DAG
from airflow.operators.dummy  import DummyOperator
from airflow.sensors.external_task  import ExternalTaskSensor
dag1 = DAG(dag_id="dag1", schedule_interval="0 16 * * *")
dag2 = DAG(dag_id="dag2", schedule_interval="0 20 * * *")
DummyOperator(task_id="etl", dag=dag1)
ExternalTaskSensor(
    task_id="wait_for_etl",
    external_dag_id="dag1",
    external_task_id="etl",
    execution_delta=datetime.timedelta(hours=4),
    dag=dag2,
)
```

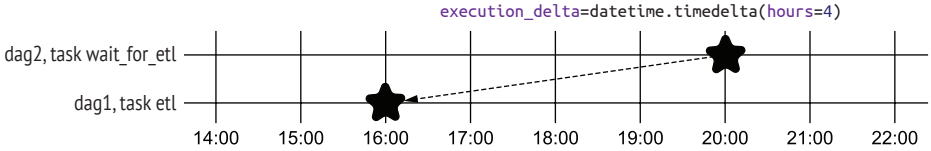


Рис. 6.22 Для `ExternalTaskSensor` можно использовать смещение с помощью аргумента `execution_delta`, чтобы обеспечить соответствие с интервалами других ОАГ

Обратите внимание, что проверка задачи с помощью `ExternalTaskSensor`, где у другого ОАГ иной период интервала, например ОАГ 1 запускается раз в день, а ОАГ 2 запускается каждые пять часов, усложняет задание подходящего значения для `execution_delta`. Для этого варианта можно предоставить функцию, возвращающую список объектов timedelta через аргумент `execution_date_fn`. Подробности *см.* в документации Airflow.

6.3 Запуск рабочих процессов с помощью REST API и интерфейса командной строки

Помимо запуска ОАГ из других ОАГ, их также можно запускать через REST API и интерфейс командной строки. Это может быть полезно, если вы хотите запускать рабочие процессы за пределами Airflow (например, как часть конвейера непрерывной интеграции и доставки). Или данные, поступающие в случайное время в бакет AWS S3, можно обрабатывать, задав лямбда-функцию для вызова REST API, запуская ОАГ, вместо того чтобы постоянно запускать опрос с сенсорами.

Используя интерфейс командной строки Airflow, можно запустить ОАГ следующим образом.

Листинг 6.5 Запуск ОАГ с помощью интерфейса командной строки Airflow

```
airflow dags trigger dag1

➥ [2019-10-06 14:48:54,297] {cli.py:238} INFO - Created <DagRun dag1 @ 2019-
    10-06 14:48:54+00:00: manual__2019-10-06T14:48:54+00:00, externally
    triggered: True>
```

Этот код запускает `dag1` с датой выполнения, установленной на текущую дату и время. Идентификатор запуска имеет префикс «`manual__`», указывая на то, что он был запущен вручную или за пределами Airflow. Интерфейс командной строки принимает дополнительную конфигурацию для запущенного ОАГ.

Листинг 6.6 Запуск ОАГ с дополнительной конфигурацией

```
airflow dags trigger -c '{"supermarket_id": 1}' dag1
airflow dags trigger --conf '{"supermarket_id": 1}' dag1
```

Данная часть конфигурации теперь доступна во всех задачах инициированного ОАГ, запущенного через переменные контекста задачи.

Листинг 6.7 Применение конфигурации

```
import airflow.utils.dates
from airflow import DAG
from airflow.operators.python import PythonOperator

dag = DAG(
    dag_id="print_dag_run_conf",
    start_date=airflow.utils.dates.days_ago(3),
    schedule_interval=None,
)
```

```
def print_conf(**context):
    print(context["dag_run"].conf)

process = PythonOperator(
    task_id="process",
    python_callable=print_conf,
    dag=dag,
)
```

Конфигурация, предоставленная при запуске ОАГ, доступна в контексте задачи

Эти задачи выводят конфигурацию, предоставленную для запуска ОАГ, которую можно применять как переменную во всей задаче:

```
{cli.py:516} INFO - Running <TaskInstance: print_dag_run_conf.process 2019-
    10-15T20:01:57+00:00 [running]> on host ebd4ad13bf98
{logging_mixin.py:95} INFO - {'supermarket': 1}
{python_operator.py:114} INFO - Done. Returned value was: None
{logging_mixin.py:95} INFO - [2019-10-15 20:03:09,149]
    {local_task_job.py:105} INFO - Task exited with return code 0
```

В результате, если у вас есть ОАГ, в котором вы запускаете копии задач просто для поддержки различных переменных, с конфигурацией запуска ОАГ этот процесс становится намного короче, поскольку это позволяет вам вставлять переменные в конвейер (рис. 6.23). Однако обратите внимание, что у ОАГ из листинга 6.8 нет интервала (т. е. он запускается только при инициализации). Если логика в вашем ОАГ основана на конфигурации запуска ОАГ, то запуск по расписанию будет невозможен, поскольку он не предоставляет никакой конфигурации запуска ОАГ.

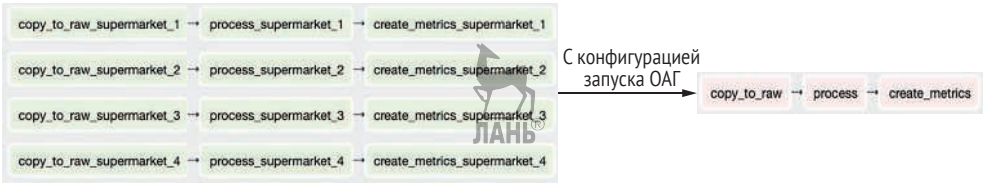


Рис. 6.23 Упрощаем ОАГ за счет предоставления полезной нагрузки во время выполнения

Для обеспечения аналогичного результата можно использовать REST API (например, если у вас нет доступа к интерфейсу командной строки, но к вашему экземпляру Airflow можно подключиться по протоколу HTTP).

Листинг 6.8 Запуск ОАГ с помощью REST API

```
# URL-адрес - это /api/v1

curl \
-u admin:admin \
-X POST \
"http://localhost:8080/api/v1/dags/print_dag_run_conf/dagRuns" \
```

Отправка имени пользователя / пароля в виде простого текста нежелательна; обратитесь к документации по аутентификации с Airflow API, чтобы узнать о других методах аутентификации

```
-H "Content-Type: application/json" \
-d '{"conf": {}}'
```

Конечной точке требуется часть данных, даже если не задана дополнительная конфигурация

```
{
  "conf": {},
  "dag_id": "print_dag_run_conf",
  "dag_run_id": "manual__2020-12-19T18:31:39.097040+00:00",
  "end_date": null,
  "execution_date": "2020-12-19T18:31:39.097040+00:00",
  "external_trigger": true,
  "start_date": "2020-12-19T18:31:39.102408+00:00",
  "state": "running"
}
```

```
curl \
-u admin:admin \
-X POST \
"http://localhost:8080/api/v1/dags/print_dag_run_conf/dagRuns" \
-H "Content-Type: application/json" \
-d '{"conf": {"supermarket": 1}}'

{
  "conf": {
    "supermarket": 1
  },
  "dag_id": "listing_6_08",
  "dag_run_id": "manual__2020-12-19T18:33:58.142200+00:00",
  "end_date": null,
  "execution_date": "2020-12-19T18:33:58.142200+00:00",
  "external_trigger": true,
  "start_date": "2020-12-19T18:33:58.153941+00:00",
  "state": "running"
}
```

Это может быть удобно при запуске ОАГ за пределами Airflow, например из системы непрерывной интеграции и доставки.

Резюме

- Сенсоры – это особый тип операторов, которые непрерывно выполняют опрос, чтобы проверить, является ли заданное условие истинным.
- Airflow предоставляет набор сенсоров для различных систем / сценариев использования; пользовательское условие также можно выполнить с помощью `PythonSensor`.
- `TriggerDagRunOperator` может запускать ОАГ из другого ОАГ, тогда как `ExternalTaskSensor` может опрашивать состояние в другом ОАГ.
- Запуск ОАГ за пределами Airflow возможен с помощью REST API и/или интерфейса командной строки.

7 Обмен данными с внешними системами

Эта глава рассказывает о:

- работе с операторами Airflow, выполняющими действия в системах за пределами Airflow;
- применении операторов для конкретных внешних систем;
- реализации операторов в Airflow, выполняющих операции от A до B;
- тестировании задач при подключении к внешним системам.

Во всех предыдущих главах мы рассматривали различные аспекты написания кода Airflow, в основном продемонстрированные на примерах с использованием универсальных операторов, таких как `BashOperator` и `PythonOperator`. Хотя эти операторы могут запускать произвольный код и, таким образом, выполнять любую рабочую нагрузку, в Airflow есть и другие операторы для более конкретных случаев использования, например для выполнения запроса к базе данных Postgres. У этих операторов есть один и только один конкретный вариант использования, допустим выполнение запроса. В результате их легко применять, просто предоставив запрос оператору, а оператор выполняет внутреннюю обработку логики запросов. В случае с `PythonOperator` вам придется писать такую логику запросов самостоятельно.

Для протокола: под фразой *внешняя система* мы подразумеваем любую технологию, кроме Airflow и машины, на которой работает Air-

flow. Это может быть, например, Microsoft Azure Blob Storage, кластер Apache Spark или хранилище данных Google BigQuery.

Чтобы увидеть, когда и как использовать такие операторы, в этой главе мы займемся разработкой двух ОАГ, которые подключаются к внешним системам и перемещают и преобразуют данные, используя эти системы. Мы рассмотрим различные варианты, которые есть (и которых нет)[1] у Airflow, для работы с этим вариантом и внешними системами.

В разделе 7.1 мы разработаем модель машинного обучения для AWS, работая с бакетами AWS S3 и AWS SageMaker, решением для разработки и развертывания моделей машинного обучения. Далее, в разделе 7.2, мы продемонстрируем, как перемещать данные между различными системами, используя базу данных Postgres, содержащую информацию об аренде жилья в Амстердаме с помощью сервиса Airbnb. Данные поступают с сайта Inside Airbnb (http://insideairbnb.com), которым управляет Airbnb, содержащего записи о списках, обзорах и многом другом. Раз в день мы будем скачивать последние данные из базы данных Postgres в наш бакет AWS S3. После этого мы запустим задание Pandas в контейнере Docker, чтобы определить колебания цен, а результат будет сохранен в S3.

7.1 Подключение к облачным сервисам

В настоящее время большая часть программного обеспечения работает на облачных сервисах. Такими сервисами обычно можно управлять через API – интерфейсом для подключения и отправки запросов облачному провайдеру. API обычно поставляется с клиентом в виде пакета Python, например клиент AWS называется boto3 (https://github.com/boto/boto3), клиент GCP – Cloud SDK (https://cloud.google.com/sdk), а клиент Azure – Azure SDK для Python (https://docs.microsoft.com/ru-ru/azure/developer/python/). Такие клиенты предоставляют удобные функции, где вы вводите необходимые данные для запроса, а клиенты занимаются техническими вопросами обработки запроса и ответа.

В контексте Airflow для программиста интерфейс – это оператор. Операторы – это удобные классы, которым вы можете предоставить необходимые детали, чтобы выполнить запрос к облачному сервису, а техническая реализация выполняется внутри оператора. Эти операторы используют Cloud SDK для отправки запросов и обеспечивают небольшой слой вокруг Cloud SDK, предоставляющий программисту определенные функции (рис. 7.1).

[1] Операторы всегда находятся в стадии разработки. Эта глава была написана в 2020 году; обратите внимание, что на момент ее прочтения могут появиться новые операторы, подходящие для вашего варианта использования, которые не были описаны здесь.

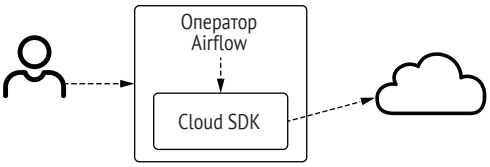


Рис. 7.1 Оператор Airflow преобразует заданные аргументы в операции в Cloud SDK

7.1.1 Установка дополнительных зависимостей

Пакет Python `apache-airflow` включает в себя несколько основных операторов, но у него нет компонентов для подключения к облаку. Для работы с облачными сервисами можно установить пакеты провайдеров из табл. 7.1.

Таблица 7.1 Дополнительные пакеты для установки дополнительных компонентов облачных провайдеров

Облачный сервис	Команда установки
AWS	pip install `apache-airflow-providers-amazon`
GCP	pip install `apache-airflow-providers-google`
Azure	pip install `apache-airflow-providers-microsoft-azure`

Это касается не только поставщиков облачных услуг, но и других внешних сервисов. Например, чтобы установить операторы и соответствующие зависимости, необходимые для запуска `PostgresOperator`, установите пакет `apache-airflow-provider-postgres`. Полный список всех доступных дополнительных пакетов *см.* в документации Airflow (https://airflow.apache.org/docs/).

Посмотрим на оператора, выполняющего действие в AWS. Взять, например, оператор `S3CopyObjectOperator`. Он копирует объект из одного бакета в другой и принимает несколько аргументов (пропустим нерелевантные аргументы для этого примера).

Листинг 7.1 S3CopyObjectOperator требует, чтобы вы указали только необходимые данные

```
➥ from airflow.providers.amazon.aws.operators.s3_copy_object import
    S3CopyObjectOperator

S3CopyObjectOperator(
    task_id="...",
    source_bucket_name="databucket",
    source_bucket_key="/data/{{ ds }}.json",
    dest_bucket_name="backupbucket",
    dest_bucket_key="/data/{{ ds }}-backup.json",
)
```

Бакет, из которого нужно копировать

Имя объекта для копирования

Бакет, в который нужно копировать

Имя целевого объекта

Этот оператор превращает копирование объекта в другое место в простое упражнение, представляющее собой заполнение пробелов, без необходимости углубляться в детали клиента AWS boto3[1].

7.1.2 Разработка модели машинного обучения

Рассмотрим более сложный пример и поработаем с операторами AWS, разработав конвейер обработки данных, создавая классификатор рукописных чисел. Для обучения модели будет использоваться база данных MNIST (http://yann.lecun.com/exdb/mnist/), содержащая примерно 70 000 изображений рукописного написания цифр от 0 до 9 (рис. 7.2).

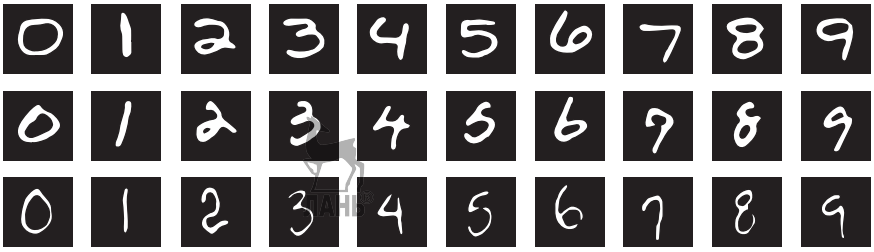


Рис. 7.2 Пример рукописного написания цифр из базы данных MNIST

После обучения модели мы должны иметь возможность отправить ей новое, ранее неизвестное рукописное число, и модель должна классифицировать его (рис. 7.3).

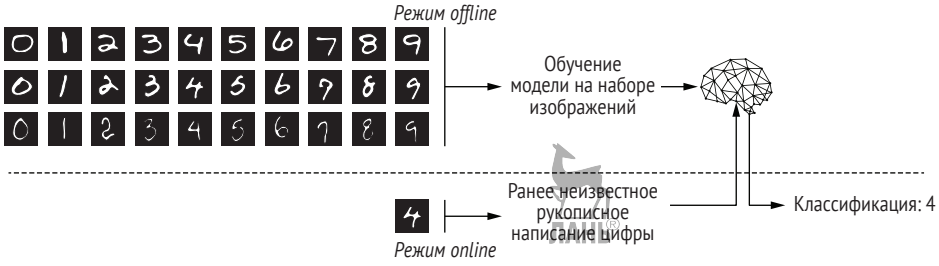


Рис. 7.3 Примерный план того, как модель обучается на одном этапе и классифицирует ранее неизвестные образцы на другом

Модель состоит из двух частей: автономной и онлайн-части. Автономная часть принимает большой набор рукописных цифр и учит модель классифицировать их. Результат (набор параметров модели) сохраняется. Этот процесс можно выполнять периодически при сборе новых данных. Онлайн-часть отвечает за загрузку модели и класси-

[1] Если просмотреть реализацию оператора, то можно увидеть, что внутри он вызывает метод `copy_object()`.

фикацию ранее неизвестных цифр. Это должно выполняться мгновенно, поскольку пользователи ожидают прямой обратной связи.

Рабочие процессы Airflow обычно отвечают за автономную часть модели. Обучение модели включает в себя загрузку данных, их предварительную обработку в формате, подходящем для модели, и обучение модели, которое может стать сложным. Кроме того, периодическое переобучение модели хорошо согласуется с парадигмой пакетной обработки Airflow. Онлайн-часть обычно представляет собой API, например REST API или HTML-страницу с вызовами REST API под капотом. Такой API обычно развертывается только один раз или как часть конвейера непрерывной интеграции/доставки. Для повторного еженедельного развертывания API нет варианта использования, и поэтому обычно это не является частью рабочего процесса Airflow.

Для обучения классификатора мы разработаем конвейер Airflow. В конвейере будет использоваться AWS SageMaker, сервис AWS, облегчающий разработку и развертывание моделей машинного обучения. В конвейере мы сначала копируем образцы данных из общедоступного места в собственный бакет S3. Далее мы преобразуем данные в формат, который можно использовать для модели, обучим модель с помощью AWS SageMaker и, наконец, развернем ее, чтобы классифицировать конкретную рукописную цифру. Конвейер будет выглядеть так, как показано на рис. 7.4.

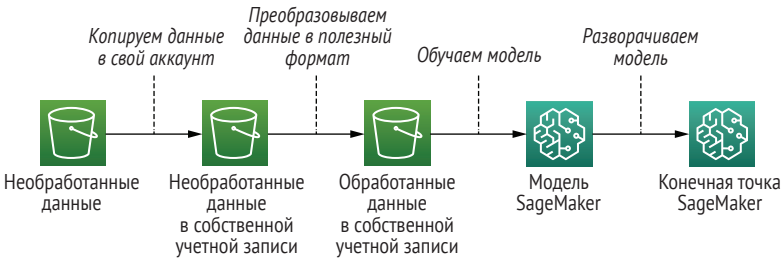


Рис. 7.4 Логические этапы для создания классификатора рукописных цифр

Этот конвейер можно запустить только один раз, и модель SageMaker можно развернуть только один раз. Сильной стороной Airflow является возможность запланировать такой конвейер и при необходимости повторно запустить (частичные) конвейеры в случае появления новых данных или изменений в модели. Если необработанные данные постоянно обновляются, конвейер Airflow будет периодически повторно загружать необработанные данные и повторно развертывать модель, обученную на новых данных. Кроме того, специалист по обработке и анализу данных может настроить модель по своему вкусу, а конвейер Airflow может автоматически повторно развернуть модель без необходимости запускать что-либо вручную.

Airflow обслуживает несколько операторов на различных сервисах платформы AWS. Хотя их список никогда не бывает полным, поскольк-

ку сервисы постоянно добавляются, изменяются или удаляются, большинство сервисов AWS поддерживаются оператором Airflow. Операторы AWS предоставляются пакетом `apache-airflow-provider-amazon`.

Посмотрим на конвейер (рис. 7.5).

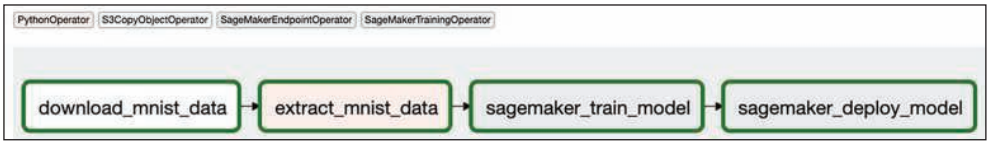


Рис. 7.5 Логические этапы, реализованные в ОАГ

Несмотря на то что задач всего четыре, в AWS SageMaker многое нужно настраивать, поэтому код у ОАГ длинный. Но не беспокойтесь; мы разберем его позже.

Листинг 7.2 ОАГ для обучения и развертывания классификатора рукописных цифр

```
import gzip
import io
import pickle

import airflow.utils.dates
from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow.providers.amazon.aws.hooks.s3 import S3Hook
➥ from airflow.providers.amazon.aws.operators.s3_copy_object import
    S3CopyObjectOperator
➥ from airflow.providers.amazon.aws.operators.sagemaker_endpoint import
    SageMakerEndpointOperator
➥ from airflow.providers.amazon.aws.operators.sagemaker_training import
    SageMakerTrainingOperator
from sagemaker.amazon.common import write_numpy_to_dense_tensor

dag = DAG(
   dag_id="chapter7_aws_handwritten_digits_classifier",
   schedule_interval=None,
   start_date=airflow.utils.dates.days_ago(3),
)

download_mnist_data = S3CopyObjectOperator(
   task_id="download_mnist_data",
   source_bucket_name="sagemaker-sample-data-eu-west-1",
   source_bucket_key="algorithms/kmeans/mnist/mnist.pkl.gz",
   dest_bucket_name="[your-bucket]",
   dest_bucket_key="mnist.pkl.gz",
  dag=dag,
)


def _extract_mnist_data():
```

S3CopyObjectOperator копирует объекты между двумя местоположениями S3

Иногда желаемая функциональность не поддерживается ни одним из операторов, и нужно реализовать логику самостоятельно

```
s3hook = S3Hook()

# Скачиваем набор данных S3 в память
mnist_buffer = io.BytesIO()
mnist_obj = s3hook.get_key(
    bucket_name="[your-bucket]",
    key="mnist.pkl.gz",
)
mnist_obj.download_fileobj(mnist_buffer)

# Распаковываем gzip-файл, извлекаем набор данных, конвертируем и загружаем
# обратно в S3
mnist_buffer.seek(0)
with gzip.GzipFile(fileobj=mnist_buffer, mode="rb") as f:
    train_set, _, _ = pickle.loads(f.read(), encoding="latin1")
    output_buffer = io.BytesIO()
    write_numpy_to_dense_tensor(
        file=output_buffer,
        array=train_set[0],
        labels=train_set[1],
    )
        output_buffer.seek(0)
        s3hook.load_file_obj(
            output_buffer,
            key="mnist_data",
            bucket_name="[your-bucket]",
            replace=True,
        )

extract_mnist_data = PythonOperator(
    task_id="extract_mnist_data",
    python_callable=_extract_mnist_data,
    dag=dag,
)

sagemaker_train_model = SageMakerTrainingOperator(
    task_id="sagemaker_train_model",
    config={
        ➥ "TrainingJobName": "mnistclassifier-{{ execution_date
    .strftime('%Y-%m-%d-%H-%M-%S') }}",
        "AlgorithmSpecification": {
            ➥ "TrainingImage": "438346466558.dkr.ecr.eu-west-
      1.amazonaws.com/kmeans:1",
            "TrainingInputMode": "File",
        },
        "HyperParameters": {"k": "10", "feature_dim": "784"},
        "InputDataConfig": [
            {
                "ChannelName": "train",
                "DataSource": {
                    "S3DataSource": {
                        "S3DataType": "S3Prefix",
                        "S3Uri": "s3://[your-bucket]/mnist_data",
```

Можно использовать S3Hook для операций в S3

Скачиваем объект S3

Загружаем извлеченные данные обратно в S3

Иногда желаемая функциональность не поддерживается ни одним из операторов, и вам придется реализовать ее самостоятельно и вызвать с помощью PythonOperator

SageMakerTrainingOperator создает обучающее задание SageMaker

Конфигурация представляет собой JSON, содержащий конфигурацию обучающего задания

```
                        "S3DataDistributionType": "FullyReplicated",
                    }
                },
            }
        ],
        ➥ "OutputDataConfig": {"S3OutputPath": "s3://[your-bucket]/
      mnistclassifier-output"},
        "ResourceConfig": {
            "InstanceType": "ml.c4.xlarge",
            "InstanceCount": 1,
            "VolumeSizeInGB": 10,
        },
        ➥ "RoleArn": "arn:aws:iam::297623009465:role/service-role/
       AmazonSageMaker-ExecutionRole-20180905T153196",
         "StoppingCondition": {"MaxRuntimeInSeconds": 24 * 60 * 60},
    },
    wait_for_completion=True,
    print_log=True,
    check_interval=10,
    dag=dag,
)
sagemaker_deploy_model = SageMakerEndpointOperator(
    task_id="sagemaker_deploy_model",
    wait_for_completion=True,
    config={
        "Model": {
            ➥ "ModelName": "mnistclassifier-{{ execution_date.strftime('%Y-
    %m-%d-%H-%M-%S') }}",
            "PrimaryContainer": {
                ➥ "Image": "438346466558.dkr.ecr.eu-west-1.amazonaws.com/
    kmeans:1",
                "ModelDataUrl": (
                    "s3://[your-bucket]/mnistclassifier-output/"
                    ➥ "mnistclassifier-{{ execution_date.strftime('%Y-%m-%d-
    %H-%M-%S') }}/"
                    "output/model.tar.gz"
                ), # Это свяжет модель и задачу по обучению
            },
            ➥ "ExecutionRoleArn": "arn:aws:iam::297623009465:role/service-
    role/AmazonSageMaker-ExecutionRole-20180905T153196",
        },
        "EndpointConfig": {
            ➥ "EndpointConfigName": "mnistclassifier-{{
    execution_date.strftime('%Y-%m-%d-%H-%M-%S') }}",
            "ProductionVariants": [
                {
                    "InitialInstanceCount": 1,
                    "InstanceType": "ml.t2.medium",
                    "ModelName": "mnistclassifier",
                    "VariantName": "AllTraffic",
                }
            ],
```

Оператор ожидает завершения обучения и выводит журналы CloudWatch во время обучения

SageMakerEndpointOperator развертывает обученную модель, что делает ее доступной за конечной точкой HTTP

```
        },
        "Endpoint": {
            ➥ "EndpointConfigName": "mnistclassifier-{{
    execution_date.strftime('%Y-%m-%d-%H-%M-%S') }}",
            "EndpointName": "mnistclassifier",
        },
},
dag=dag,
)

➥ download_mnist_data >> extract_mnist_data >> sagemaker_train_model >>
    sagemaker_deploy_model
```

С внешними сервисами сложность часто заключается не в Airflow, а в обеспечении правильной интеграции различных компонентов в конвейер. Есть довольно много настроек, связанных с SageMaker, поэтому разберем задачи фрагмент за фрагментом.

Листинг 7.3 Копирование данных между двумя бакетами S3

```
download_mnist_data = S3CopyObjectOperator(
    task_id="download_mnist_data",
    source_bucket_name="sagemaker-sample-data-eu-west-1",
      source_bucket_key="algorithms/kmeans/mnist/mnist.pkl.gz",
      dest_bucket_name="[your-bucket]",
      dest_bucket_key="mnist.pkl.gz",
      dag=dag,
)
```

После инициализации ОАГ первая задача копирует набор данных MNIST из общедоступного бакета в наш. Мы храним его там для дальнейшей обработки. `S3CopyObjectOperator` запрашивает имя бакета и объекта в источнике и месте назначения и копирует выбранный объект. Итак, как же убедиться во время разработки, что все правильно, без предварительного написания кода всего конвейера и скрещивания пальцев, чтобы увидеть, работает ли это в промышленном окружении?

7.1.3 Локальная разработка с использованием внешних систем

В частности, в случае с AWS, если у вас есть доступ к облачным ресурсам из машины, используемой для разработки, с ключом доступа, вы можете запускать задачи Airflow локально. С помощью команды интерфейса командной строки `airflow tasks test` можно запустить отдельную задачу для заданной даты выполнения. Поскольку задача `download_mnist_data` не использует дату выполнения, не имеет значения, какое значение мы предоставляем. Однако предположим, что для `dest_bucket_key` было задано значение `mnist-{{ds}}.pkl.gz`; тогда нужно будет подумать о том, с какой датой выполнения мы будем

проводить тестирование. Из командной строки выполните действия, указанные в следующем листинге.

Листинг 7.4 Настройка для локального тестирования операторов AWS

```
# Add secrets in ~/.aws/credentials:
  # [myaws]
  # aws_access_key_id=AKIAEXAMPLE123456789
  # aws_secret_access_key=supersecretaccesskeydonotshare!123456789

export AWS_PROFILE=myaws
export AWS_DEFAULT_REGION=eu-west-1
export AIRFLOW_HOME=[your project dir]
airflow db init                                    ← Инициализируем локальную базу метаданных Airflow
airflow tasks test chapter7_aws_handwritten_digits_classifier
     download_mnist_data 2020-01-01                ← Запускаем одну задачу
```

С помощью данного кода мы запускаем задачу `download_mnist_data` и отображаем журналы.

Листинг 7.5 Ручная проверка задачи с помощью команды `airflow tasks test`

```
➥ $ airflow tasks test chapter7_aws_handwritten_digits_classifier
     download_mnist_data 2019-01-01

INFO - Using executor SequentialExecutor
INFO - Filling up the DagBag from .../dags
➥ INFO - Dependencies all met for <TaskInstance:
     chapter7_aws_handwritten_digits_classifier.download_mnist_data 2019-01-
     01T00:00:00+00:00 [None]>
--------------------------------------------------------------------------------
INFO - Starting attempt 1 of 1
--------------------------------------------------------------------------------
➥ INFO - Executing <Task(PythonOperator): download_mnist_data> on 2019-01-
     01T00:00:00+00:00
INFO - Found credentials in shared credentials file: ~/.aws/credentials
INFO - Done. Returned value was: None
➥ INFO - Marking task as SUCCESS.dag_id=chapter7_aws_handwritten_digits
     _classifier, task_id=download_mnist_data, execution_date=20190101T000000,
     start_date=20200208T110436, end_date=20200208T110541
```

После этого мы видим, что данные были скопированы в наш бакет (рис. 7.6).

Что сейчас произошло? Мы сконфигурировали учетные данные AWS, что дает нам возможность получить доступ к облачным ресурсам с локального компьютера. Хотя это характерно для AWS, аналогичные методы аутентификации применимы к GCP и Azure. Клиент AWS boto3, используемый в операторах Airflow, будет искать учетные данные в разных местах на машине, где запущена задача. В листинге 7.4 мы задаем переменную окружения `AWS_PROFILE`, которую клиент boto3 использует для аутентификации. После этого задаем еще одну

переменную окружения: `AIRFLOW_HOME`. Это место, где Airflow будет хранить журналы и т. п. Внутри этого каталога Airflow будет искать каталог `/dags`. Если это произойдет в другом месте, то можно воспользоваться еще одной переменной окружения: `AIRFLOW __CORE__DAGS_FOLDER`.

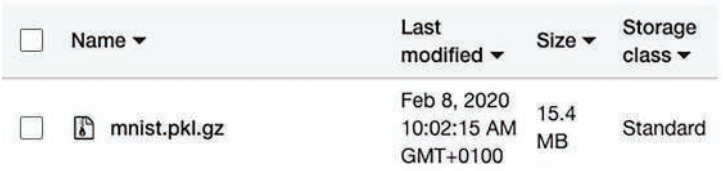


Рис. 7.6 После локального запуска задачи с помощью команды `airflow tasks test` данные копируются в наш бакет AWS S3

Далее мы выполняем команду `airflow db init`. Перед тем как сделать это, убедитесь, что вы не использовали переменную `AIRFLOW__CORE__SQL_ALCHEMY_CONN` (унифицированный идентификатор ресурса базы данных, где хранятся все состояния) или задали для нее значение URI базы данных специально с целью тестирования. Без настройки `AIRFLOW __CORE__SQL_ALCHEMY_CONN` команда `airflow db init` инициализирует локальную базу данных SQLite (один файл, конфигурация не требуется) внутри переменной окружения `AIRFLOW_HOME`[1]. Команда `airflow tasks test` существует для запуска и проверки одной задачи и не записывает состояние в базу данных; однако для хранения журналов требуется база данных, поэтому мы должны инициализировать ее с помощью команды `airflow db init`.

После всего этого можно запустить задачу из командной строки с помощью команды `airflow tasks test chapter7_aws_handwritten_digits_classifier extract_mnist_data 2020-01-01`. После того как мы скопировали файл в наш собственный бакет S3, нужно преобразовать его в формат, который ожидает модель SageMaker KMeans, а именно формат RecordIO[2].

Листинг 7.6 Преобразование данных MNIST в формат RecordIO для модели SageMaker KMeans

```
import gzip
import io
import pickle
```

[1] База данных будет создана в файле airflow.db в каталоге, заданном переменной `AIRFLOW_HOME`. Его можно открыть и проверить, например с помощью DBeaver.

[2] Документацию по MIME-типу `application/x-recordio-protobuf` можно найти на странице https://docs.aws.amazon.com/de_de/sagemaker/latest/dg/cdf-inference.html.

```
from airflow.operators.python import PythonOperator
from airflow.providers.amazon.aws.hooks.s3 import S3Hook
from sagemaker.amazon.common import write_numpy_to_dense_tensor

def _extract_mnist_data():
    s3hook = S3Hook()

    # Скачиваем набор данных S3 в память
    mnist_buffer = io.BytesIO()
    mnist_obj = s3hook.get_key(
        bucket_name="your-bucket",
        key="mnist.pkl.gz",
    )
    mnist_obj.download_fileobj(mnist_buffer)

    # Распаковываем файл gzip, извлекаем набор данных, выполняем преобразование
    # и загружаем обратно в S3
    mnist_buffer.seek(0)
    with gzip.GzipFile(fileobj=mnist_buffer, mode="rb") as f:
        train_set, _, _ = pickle.loads(f.read(), encoding="latin1")
        output_buffer = io.BytesIO()
        write_numpy_to_dense_tensor(
            file=output_buffer,
            array=train_set[0],
            labels=train_set[1],
        )
        output_buffer.seek(0)
        s3hook.load_file_obj(
            output_buffer,
            key="mnist_data",
            bucket_name="your-bucket",
            replace=True,
        )


extract_mnist_data = PythonOperator(
    task_id="extract_mnist_data",
    python_callable=_extract_mnist_data,
    dag=dag,
)
```

Инициализируем S3Hook для обмена данными с S3

Скачиваем данные в двоичный поток в памяти

Выполняем разархивирование и десериализацию

Преобразовываем массив Numpy в записи RecordIO

Загружаем результат в S3

Сам по себе Airflow – это универсальный фреймворк оркестровки с управляемым набором функций, которые необходимо изучать. Однако работа с данными часто требует времени и опыта, чтобы быть в курсе всех технологий и знать, какие точки и каким образом соединять. Процесс разработки в Airflow никогда не осуществляется сам по себе; вы часто подключаетесь к другим системам и читаете документацию по этой конкретной системе. Хотя Airflow инициирует выполнение такой задачи, сложность разработки конвейера обработки данных часто лежит за пределами Airflow и связана с системой, с которой вы обмениваетесь данными. Хотя эта книга посвящена исключительно Airflow, в связи с особенностями работы с другими инструментами

обработки данных мы попытаемся продемонстрировать на этих примерах, что значит разработка конвейера обработки данных.

Для этой задачи в Airflow нет функций для скачивания данных и их извлечения, преобразования и выгрузки результата в S3. Следовательно, нужно реализовать собственную функцию. Она будет скачивать данные в двоичный поток в памяти (io.BytesIO), чтобы данные никогда не сохранялись в файле в файловой системе и чтобы после выполнения задачи файлов не оставалось. Размер базы MNIST невелик (15 МБ), поэтому она будет работать на любой машине. Однако нужно подумать о реализации; в случае с большими данными, возможно, разумнее будет хранить данные на дисках и обрабатывать их частями.

Аналогично эту задачу можно запустить или протестировать локально:

```
airflow tasks test chapter7_aws_handwritten_digits_classifier extract_mnist_data
2020-01-01
```

После завершения данные будут видны в S3 (рис. 7.7).

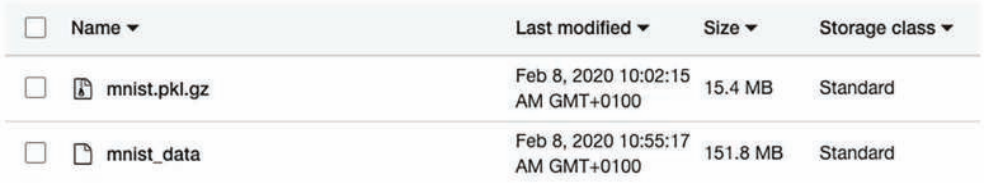


Рис. 7.7 Заархивированные и сериализованные данные были прочитаны и преобразованы в пригодный для использования формат

Следующие две задачи обучают и развертывают модель SageMaker. Операторы SageMaker принимают аргумент `config`, что влечет за собой конфигурацию, предназначенную конкретно для SageMaker, обсуждение которой выходит за рамки этой книги. Сосредоточимся на других аргументах.

Листинг 7.7 Обучение модели SageMaker

```
sagemaker_train_model = SageMakerTrainingOperator(
   task_id="sagemaker_train_model",
   config={
      ➥ "TrainingJobName": "mnistclassifier-{{ execution_date.strftime('%Y-
     %m-%d-%H-%M-%S') }}",
          ...
      },
      wait_for_completion=True,
      print_log=True,
      check_interval=10,
      dag=dag,
   )
```

Многие детали в аргументе `config` относятся к SageMaker, и о них можно узнать, прочитав документацию по SageMaker. Однако можно сделать два урока, применимых к работе с любой внешней системой.

Во-первых, AWS ограничивает параметр `TrainingJobName`, чтобы сделать его уникальным в рамках учетной записи AWS и региона. Дважды запустив этого оператора с одним и тем же именем `TrainingJobName`, вы получите ошибку. Скажем, мы предоставили `TrainingJobName` фиксированное значение `mnistclassifier`; если запустить его во второй раз, то это приведет к сбою:

```
botocore.errorfactory.ResourceInUse: An error occurred (ResourceInUse) when
calling the CreateTrainingJob operation: Training job names must be unique
within an AWS account and region, and a training job with this name already
exists (arn:aws:sagemaker:eu-west-1:[account]:training-job/mnistclassifier)
```

Аргумент `config` можно шаблонизировать, следовательно, если вы планируете периодически переобучать свою модель, то должны предоставить ей уникальное имя `TrainingJobName`, что можно сделать, создав шаблон с `execution_date`. Таким образом мы гарантируем идемпотентность нашей задачи, и существующие обучающие задания не будут приводить к конфликту имен.

Во-вторых, обратите внимание на аргументы `wait_for_completion` и `check_interval`. Если для аргумента `wait_for_completion` задано значение `false`, команда сработает по принципу *fire and forget* (выстрелил и забыл) (так и работает клиент boto3): AWS приступит к обучению, но мы так и не узнаем, прошло ли оно успешно. Поэтому все операторы SageMaker ждут (по умолчанию `wait_for_completion=True`) завершения данной задачи. Внутри операторы осуществляют опрос каждые X секунд, проверяя, выполняется ли задание. Это гарантирует, что наши задачи Airflow будут завершены только после окончания (рис. 7.8). Если у вас есть нижестоящие задачи и вы хотите обеспечить правильное поведение и порядок конвейера, нужно дождаться завершения.

Рис. 7.8 Операторы SageMaker срабатывают успешно только после успешного завершения задания в AWS

После завершения всего конвейера мы успешно развернули модель SageMaker и конечную точку, чтобы предоставить доступ (рис. 7.9).

Рис. 7.9 В меню модели SageMaker видно, что модель развернута, а конечная точка находится в рабочем состоянии

Однако в AWS конечная точка SageMaker не доступна внешнему миру. Она доступна через API-интерфейсы AWS, но, например, не через доступную везде конечную точку HTTP.

Конечно, чтобы завершить конвейер обработки данных, нам нужен хороший интерфейс или API для ввода рукописных цифр и получения результата. В AWS, чтобы сделать его доступным через интернет, можно было бы воспользоваться AWS Lambda (https://aws.amazon.com/ru/lambda/), чтобы запустить конечную точку SageMaker и API Gateway (https://aws.amazon.com/ru/api-gateway/) для создания конечной точки, перенаправляя запросы в Lambda[1], так почему бы не интегрировать их в наш конвейер (рис. 7.10).

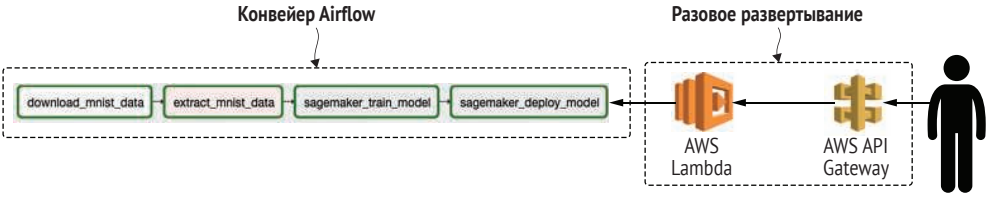


Рис. 7.10 Классификатор рукописных цифр состоит из большего числа компонентов, нежели простой конвейер Airflow

Причина отказа от развертывания инфраструктуры заключается в том, что Lambda и API Gateway будут развертываться разово, а не периодически. Они работают на онлайн-этапе модели, и поэтому их лучше развертывать как часть конвейера непрерывной интеграции и доставки. Для полноты картины API можно реализовать с помощью Chalice.

Листинг 7.8 Пример API, с которым могут взаимодействовать пользователи, с использованием AWS Chalice

```
import json
from io import BytesIO
import boto3

import numpy as np
from PIL import Image
from chalice import Chalice, Response
from sagemaker.amazon.common import numpy_to_record_serializer

app = Chalice(app_name="number-classifier")
@app.route("/", methods=["POST"], content_types=["image/jpeg"])
def predict():
    ""
    Предоставляем этой конечной точке изображение в формате jpeg.
```

[1] Chalice (https://github.com/aws/chalice) – это фреймворк на языке Python, аналогичный Flask, который используется для разработки API и автоматического создания базового шлюза API и лямбда-ресурсов в AWS.

```
    Изображение должно быть по размеру равно изображениям, используемым
    для обучения(28×28).
    """
    img = Image.open(BytesIO(app.current_request.raw_body)).convert("L")
    img_arr = np.array(img, dtype=np.float32)
    runtime = boto3.Session().client(
        service_name="sagemaker-runtime",
        region_name="eu-west-1",
    )
    response = runtime.invoke_endpoint(
        EndpointName="mnistclassifier",
        ContentType="application/x-recordio-protobuf",
        Body=numpy_to_record_serializer()(img_arr.flatten()),
    )
    result = json.loads(response["Body"].read().decode("utf-8"))
    return Response(
        result,
        status_code=200,
        headers={"Content-Type": "application/json"},
)
```

Преобразуем исходное изображение в массив numpy в оттенках серого

Вызываем конечную точку SageMaker, которую развернул OAG

Ответ SageMaker возвращается в виде байтов

API содержит одну конечную точку, которая принимает изображение в формате JPEG.

Листинг 7.9 Классификация изображения рукописной цифры путем его отправки в API

```
curl --request POST \
  --url http://localhost:8000/ \
  --header 'content-type: image/jpeg' \
  --data-binary @'/path/to/image.jpeg'
```

При правильном обучении результат выглядит, как показано на рис. 7.11.

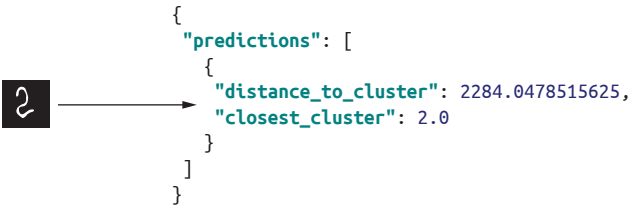


Рис. 7.11 Пример ввода и вывода API. В результате у вас может получиться прекрасный пользовательский интерфейс для загрузки изображений и отображения прогнозируемого числа

API преобразует данное изображение в формат RecordIO, который использовался для обучения модели SageMaker. Затем объект RecordIO пересылается в конечную точку SageMaker, развернутую конвейером Airflow, и, наконец, возвращает прогнозируемое число.

7.2 Перенос данных из одной системы в другую

Классический вариант использования Airflow – это периодическое задание ETL, при котором данные скачиваются ежедневно и преобразуются в другом месте. Такое задание часто используется в аналитических целях, когда данные экспортируются из рабочей базы данных и хранятся в другом месте для последующей обработки. Рабочая база данных чаще всего (в зависимости от модели данных) не способна возвращать архивные данные (например, состояние базы данных, каким оно было месяц назад). Поэтому нередко создается периодический экспорт, который сохраняется для последующей обработки. Дампы архивных данных быстро повысят ваши требования к хранилищу и потребуют распределенной обработки для обработки всех данных. В этом разделе вы узнаете, как оркестрировать такую задачу с помощью Airflow.

Мы разработали репозиторий в GitHub с примерами кода для этой книги. Он содержит файл Docker Compose для развертывания и запуска следующего варианта использования, в котором мы извлекаем данные листинги Airbnb и обрабатываем их в контейнере Docker с помощью библиотеки Pandas. При обработке данных большого масштаба контейнер Docker можно заменить заданием Spark, которое распределяет работу по нескольким машинам. Файл Docker Compose содержит:

- один контейнер Postgres со списками по Амстердаму от Airbnb;
- один контейнер, совместимый с API AWS S3. Поскольку у нас нет «AWS S3-в-Docker», мы создали контейнер MinIO (хранилище объектов, совместимое с API AWS S3) для чтения и записи данных;
- один контейнер Airflow.

Визуально поток будет выглядеть, как показано на рис. 7.12.

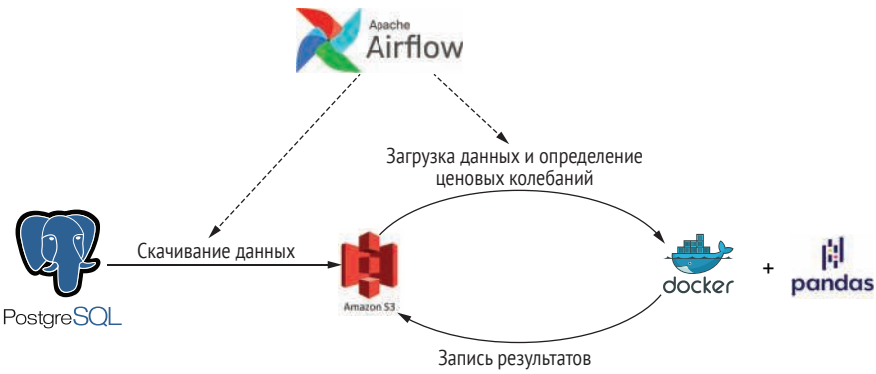


Рис. 7.12 Airflow управляет заданиями, перемещая данные из одной системы в другую

Airflow ведет себя как «паук в паутине», запускает задания и управляет ими, обеспечивая их успешное завершение в правильном порядке, в противном случае произойдет сбой конвейера.

Контейнер Postgres – это специально созданный образ Postgres, содержащий базу данных, заполненную данными из Inside Airbnb, которые доступны в Docker Hub как airflowbook / insideairbnb. В базе данных содержится одна таблица с названием «listings», в которой содержатся записи о местах в Амстердаме, указанных на сайте Airbnb в период с апреля 2015 года по декабрь 2019 года (рис. 7.13).

Рис. 7.13 Пример структуры таблицы базы данных Inside Airbnb

Для начала выполним запрос к базе данных и экспортируем данные в S3. Оттуда мы будем читать и обрабатывать данные с помощью Pandas.

Передача данных из одной системы в другую, возможно, с промежуточным преобразованием – распространенная задача в Airflow. Запрос к базе данных MySQL и сохранение результата в Google Cloud Storage, копирование данных с SFTP-сервера в озеро данных в AWS S3 или вызов HTTP REST API и сохранение вывода имеют одну общую особенность, а именно: они имеют дело с двумя системами, одна из которых используется для ввода, а другая – для вывода.

В экосистеме Airflow это привело к появлению множества операторов типа «A-to-B». Для этих примеров у нас есть `MySql-ToGoogle-CloudStorageOperator`, `SFTPToS3Operator` и `SimpleHttpOperator`. Хотя существует в экосистеме Airflow множество примеров операторов, оператора `Postgres-query-to-AWS-S3` (на момент написания книги) не существует. Так что же делать?

7.2.1 Реализация оператора PostgresToS3Operator

Во-первых, мы могли бы обратить внимание на то, как работают другие похожие операторы, и разработать собственный оператор `PostgresToS3Operator`. Рассмотрим оператор, тесно связанный с нашим примером, `MongoToS3Operator` из `airflow.providers.amazon.aws.transfers.mongo_to_s3` (после установки `apache-airflow-provider-amazon`). Этот оператор выполняет запрос к базе данных MongoDB и сохраняет результат в бакете AWS S3. Изучим его и выясним, как заменить MongoDB на Postgres. Метод `execute()` реализован следующим образом (часть кода «запутана»).

Листинг 7.10 Реализация MongoToS3Operator

```
def execute(self, context):
    s3_conn = S3Hook(self.s3_conn_id)

    results = MongoHook(self.mongo_conn_id).find(
        mongo_collection=self.mongo_collection,
        query=self.mongo_query,
        mongo_db=self.mongo_db
    )
    docs_str = self._stringify(self.transform(results))

    # Загрузка в S3
    s3_conn.load_string(
        string_data=docs_str,
        key=self.s3_key,
        bucket_name=self.s3_bucket,
        replace=self.replace
    )
```

Создается экземпляр S3Hook

Создается экземпляр MongoHook, который используется для запроса данных

Результаты трансформируются

Вызывается метод load_string() для записи преобразованных результатов

Важно отметить, что этот оператор не использует файловые системы на машине Airflow, а сохраняет все результаты в памяти. Процесс в основном выглядит так:

```
MongoDB → Airflow in operator memory → AWS S3.
```

Поскольку этот оператор сохраняет промежуточные результаты в памяти, подумайте о последствиях для памяти при выполнении очень больших запросов, потому что очень большой результат потенциально может истощить доступную память на машине Airflow. А пока не будем забывать о реализации `MongoToS3Operator` и рассмотрим еще один оператор, `S3ToSFTPOperator`.

Листинг 7.11 Реализация S3ToSFTPOperator

```
def execute(self, context):
  ssh_hook = SSHHook(ssh_conn_id=self.sftp_conn_id)
  s3_hook = S3Hook(self.s3_conn_id)
```

```
s3_client = s3_hook.get_conn()
sftp_client = ssh_hook.get_conn().open_sftp()

with NamedTemporaryFile("w") as f:
    s3_client.download_file(self.s3_bucket, self.s3_key, f.name)
    sftp_client.put(f.name, self.sftp_path)
```

NamedTemporaryFile используется для временного скачивания файла, который удаляется после выхода из контекста

Этот оператор, опять же, создает два хука: SSHHook (SFTP – это FTP через SSH) и S3Hook. Однако в этом операторе промежуточный результат записывается в NamedTemporaryFile, временное место в локальной файловой системе экземпляра Airflow. В этой ситуации мы не сохраняем в памяти весь результат, но должны убедиться, что на диске достаточно места.

У обоих операторов есть два общих хука: один для обмена данными с системой A и второй для системы B. Однако способ извлечения и передачи данных из системы A в систему B отличается и зависит от того, кто реализует конкретный оператор. В конкретном случае с Postgres курсоры базы данных могут выполнять итерацию для извлечения и загрузки фрагментов результатов. Однако такие подробности реализации выходят за рамки этой книги. Будем проще и предположим, что промежуточный результат соответствует границам ресурсов экземпляра Airflow.

Очень минимальная реализация оператора PostgresToS3Operator может выглядеть следующим образом.

Листинг 7.12 Пример реализации PostgresToS3Operator

```
def execute(self, context):
  postgres_hook = PostgresHook(postgres_conn_id=self._postgres_conn_id)
  s3_hook = S3Hook(aws_conn_id=self._s3_conn_id)

  results = postgres_hook.get_records(self._query)
  s3_hook.load_string(
     string_data=str(results),
     bucket_name=self._s3_bucket,
     key=self._s3_key,
  )
```

Извлекаем записи из базы данных PostgreSQL

Загружаем записи в объект S3

Изучим этот код. Инициализация обоих хуков проста; мы инициализируем их, предоставляя имя идентификатора подключения, указанного пользователем. Хотя использовать аргументы ключевого слова необязательно, вы, возможно, заметили, что S3Hook принимает аргумент aws_conn_id (а не s3_conn_id, как вы могли ожидать). Во время разработки такого оператора и использования подобных хуков иногда неизбежно приходится погружаться в исходный код или внимательно читать документацию, чтобы просмотреть все доступные аргументы и понять, как осуществляется распространение на классы. В случае с S3Hook он является подклассом AwsHook и наследует ряд методов и атрибутов, например aws_conn_id.

`PostgresHook` также является подклассом, а именно класса `DbApiHook`. При этом он наследует несколько методов, таких как `get_records()`, который выполняет заданный запрос и возвращает результаты. Тип возврата – это последовательность последовательностей (точнее, список кортежей[1]). Затем мы преобразовываем результаты в строку и вызываем метод `load_string()`, который записывает закодированные данные в заданный бакет или ключ в AWS S3. Вы, возможно, думаете, что это не очень практично, и вы правы. Хотя это минимальный поток для выполнения запроса в Postgres и записи результата в AWS S3, список кортежей преобразован в строку, который ни один из фреймворков обработки данных не может интерпретировать как обычный формат файла, такой как CSV или JSON (рис. 7.14).

123 id	ABC name
2.818	Quiet Garden View Room & Super Fast WiFi
20.168	Studio with private bathroom in the centre 1
25.428	Lovely apt in City Centre (w.lift) near Jordaan

Рис. 7.14 Экспорт данных из базы данных Postgres в кортежи, которые были преобразованы в строку

Сложная часть разработки конвейеров обработки данных часто заключается не в оркестровке заданий с Airflow, а в гарантии того, что все части и детали различных заданий правильно настроены и сочетаются друг с другом, как кусочки головоломки. Итак, запишем результаты в формат CSV; это позволит фреймворкам обработки данных, таким как Apache Pandas и Spark, с легкостью интерпретировать выходные данные.

Для загрузки данных в S3 `S3Hook` предоставляет разные удобные методы. В случае с файловыми объектами[2] можно применить метод `load_file_obj()`.

Листинг 7.13 Преобразование в памяти результатов запроса к Postgres в формат CSV и загрузка в S3

```
def execute(self, context):
    postgres_hook = PostgresHook(postgres_conn_id=self._postgres_conn_id)
    s3_hook = S3Hook(aws_conn_id=self._s3_conn_id)

    results = postgres_hook.get_records(self.query)

    data_buffer = io.StringIO()
```

Для удобства мы сначала создаем строковый буфер, похожий на файл в памяти, в который можно записывать строки. После записи мы преобразуем его в двоичный формат

[1] Как указано в спецификации PEP 249.

[2] Объекты в памяти с методами для операций с файлами для чтения или записи.

```
    csv_writer = csv.writer(data_buffer, lineterminator=os.linesep)
    csv_writer.writerows(results)
    data_buffer_binary = io.BytesIO(data_buffer.getvalue().encode())
    s3_hook.load_file_obj(
        file_obj=data_buffer_binary,
        bucket_name=self._s3_bucket,
        key=self._s3_key,
        replace=True,
    )
```

Требуется файловый объект в двоичном режиме

Обеспечиваем идемпотентность, заменив файлы, если они уже существуют

Буферы находятся в памяти, что может быть удобно, поскольку после обработки в файловой системе не остается файлов. Однако мы должны понимать, что вывод запроса к Postgres должен умещаться в памяти. Ключ к идемпотентности – задать для `replace` значение `True`. Так вы гарантируете перезапись существующих файлов. Мы можем перезапустить конвейер, например после изменения кода, и тогда конвейер завершится ошибкой, если для `replace` задано значение `True` из-за существующего файла.

С помощью этих дополнительных строк мы теперь можем хранить файлы с расширением CSV в S3. Посмотрим, как это выглядит на практике.

Листинг 7.14 Запуск PostgresToS3Operator

```
download_from_postgres = PostgresToS3Operator(
    task_id="download_from_postgres",
    postgres_conn_id="inside_airbnb",
    query="SELECT * FROM listings WHERE download_date={{ ds }}",
    s3_conn_id="s3",
    s3_bucket="inside_airbnb",
    s3_key="listing-{{ ds }}.csv",
    dag=dag,
)
```

Теперь у нас есть удобный оператор, превращающий запросы к Postgres и запись результата в CSV-файл в упражнение по заполнению пропусков.

7.2.2 Привлекаем дополнительные ресурсы для тяжелой работы

В сообществе Airflow часто обсуждают, следует ли рассматривать Airflow не только как систему оркестровки задач, но и как систему их выполнения, поскольку многие ОАГ написаны с помощью `BashOperator` и `PythonOperator`, которые выполняют работу в рамках одной и той же среды выполнения Python как Airflow. Противники этой точки зрения утверждают, что рассматривают Airflow только как систему для запуска задач, и полагают, что внутри самого Airflow не следует выполнять

никаких фактических работ. Вместо этого всю работу нужно передать системе, предназначенной для работы с данными, такой как Apache Spark.

Представим, что у нас есть очень большое задание, на которое потребуются все ресурсы компьютера, на котором работает Airflow. В этом случае лучше выполнить его в другом месте; Airflow запустит задание и дождется его завершения. Идея состоит в том, что должно быть четкое разделение между оркестровкой и выполнением, чего можно достичь с помощью Airflow, запускающего задание и ожидающего его завершения, и фреймворка обработки данных, такого как Spark, выполняющего фактическую работу.

В Spark есть несколько способов для запуска задания:

- *использовать* `SparkSubmitOperator` – для этого требуется утилита spark-subnit и режим yarn-client на компьютере, где работает Airflow, чтобы найти экземпляр Spark;
- *применить* `SSHOperator` – для этого требуется доступ к экземпляру Spark по протоколу SSH, но не нужен режим spark-client в экземпляре Airflow;
- *использовать* `SimpleHTTPOperator` – для этого надо запустить Livy, REST API для Apach Spark, чтобы получить доступ к Spark.

Ключ к работе с любым оператором в Airflow – это чтение документации, чтобы выяснить, какие аргументы предоставить. Посмотрим на оператор `DockerOperator`, который запускает контейнер Docker для обработки данных Inside Airbnb с помощью Pandas.

Листинг 7.15 Запуск контейнера Docker с помощью DockerOperator

```
crunch_numbers = DockerOperator(
    task_id="crunch_numbers",
    image="airflowbook/numbercruncher",
    api_version="auto",
    auto_remove=True,
    docker_url="unix://var/run/docker.sock",
    network_mode="host",
    environment={
        "S3_ENDPOINT": "localhost:9000",
        "S3_ACCESS_KEY": "[insert access key]",
        "S3_SECRET_KEY": "[insert secret key]",
    },
    dag=dag,
)
```

Удаляем контейнер после завершения

Чтобы подключиться к другим службам на хост-машине через http://localhost, нужно совместно использовать пространство имен хост-сети, задав для network_mode значение host

`DockerOperator` оборачивается вокруг клиента Python Docker и, учитывая список аргументов, позволяет запускать контейнеры Docker. В листинге 7.15 для `docker_url` задано значение в виде сокета Unix, а для этого требуется, чтобы Docker работал на локальном компьютере. Мы запускаем образ Docker `airflowbook/numbercruncher`, включающий в себя сценарий Pandas, который загружает внутренние данные Airbnb из S3, обрабатывает их и записывает результаты в S3.

Листинг 7.16 Пример результатов из сценария numbercruncher

```
[
  {
    "id": 5530273,
    "download_date_min": 1428192000000,
    "download_date_max": 1441238400000,
    "oldest_price": 48,
    "latest_price": 350,
    "price_diff_per_day": 2
  },
  {
    "id": 5411434,
    "download_date_min": 1428192000000,
    "download_date_max": 1441238400000,
    "oldest_price": 48,
    "latest_price": 250,
    "price_diff_per_day": 1.3377483444
  },
  ...
]
```

Airflow управляет запуском контейнера, извлечением журналов и в конечном итоге удалением контейнера, если потребуется. Ключ в том, чтобы гарантировать, что ни одно состояние не осталось позади, чтобы ваши задачи могли выполняться идемпотентно и не было ничего лишнего.

Резюме

- Операторы внешних систем предоставляют функции, вызывая клиента для данной системы.
- Иногда эти операторы просто передают аргументы клиенту Python.
- В других случаях они предоставляют дополнительные возможности, такие как оператор `SageMakerTrainingOperator`, который непрерывно опрашивает AWS и выполняет блокировку до завершения.
- Если можно получить доступ к внешним сервисам с локального компьютера, то можно тестировать задачи с помощью команды `airflow tasks test`.

8 Создание пользовательских компонентов

Эта глава рассказывает о:

- создании более модульных и лаконичных ОАГ с помощью пользовательских компонентов;
- проектировании и реализации собственного хука;
- проектировании и реализации собственного оператора;
- проектировании и реализации собственного сенсора;
- распространении пользовательских компонентов в качестве базовой библиотеки Python.

Одна из сильных сторон Airflow состоит в том, что его можно легко расширить для координации заданий в различных типах систем. Мы уже видели некоторые из этих функций в предыдущих главах, где нам удалось выполнить задание по обучению модели в Amazon SageMaker с помощью оператора `S3CopyObjectOperator`, но можно также использовать Airflow (например) для выполнения заданий в кластере ECS (Elastic Container Service) в AWS с помощью `ECSOperator` для выполнения запросов к базе данных Postgres с `PostgresOperator` и делать многое другое.

Однако в какой-то момент вам может понадобиться выполнить задачу в системе, которая не поддерживается Airflow, или у вас может быть задача, которую можно реализовать с помощью `PythonOperator`, но для этого требуется много шаблонного кода, что не позволяет

другим с легкостью повторно использовать ваш код в разных ОАГ. Что же делать?

К счастью, Airflow дает возможность создавать новые операторы для реализации пользовательских операций. Он позволяет запускать задания в системах, которые не поддерживаются иным образом, или просто упростить выполнение распространенных операций в ОАГ. Фактически именно так и было реализовано множество операторов в Airflow: кому-то нужно было запустить задание в определенной системе и создать для него оператор.

В этой главе мы покажем вам, как создавать собственные операторы и использовать их в своих ОАГ. Мы также рассмотрим, как упаковать эти компоненты в пакет Python, чтобы упростить их установку и повторное использование в разных окружениях.

8.1 Начнем с PythonOperator

Прежде чем создавать какие-либо пользовательские компоненты, попробуем решить нашу проблему с помощью (теперь уже знакомого) оператора `PythonOperator`. В данном случае нас интересует создание рекомендательной системы, которая будет предлагать новые фильмы в зависимости от нашей истории просмотров. Однако в качестве первоначального пилотного проекта мы решили сосредоточиться на простом получении данных, которые касаются прошлых оценок пользователей по данному набору фильмов и рекомендации фильмов, которые кажутся наиболее популярными в целом, на основе их рейтинга.

Данные по рейтингу фильмов будут предоставляться через API, который можно использовать для получения оценок пользователей за определенный период времени. Это позволяет нам, например, ежедневно получать новые оценки и использовать их для обучения нашей рекомендательной системы. Для нашего пилотного проекта нам нужно настроить этот ежедневный процесс импорта и создать рейтинг самых популярных фильмов. Он будет использоваться в дальнейшем для рекомендации популярных фильмов (рис. 8.1).

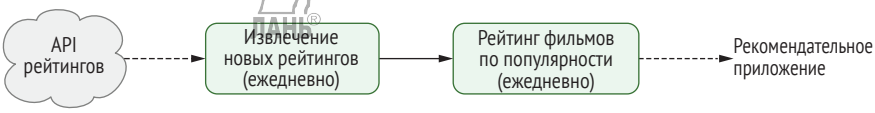


Рис. 8.1 Создание простого пилотного проекта рекомендательной системы

8.1.1 Имитация API для рейтинга фильмов

Чтобы смоделировать данные для этого примера, мы используем данные из базы MovieLens 25M (https://grouplens.org/datasets/movielens/), которая находится в свободном доступе и содержит 25 млн оценок

62 000 фильмов от 162 000 пользователей. Поскольку сама база предоставляется в виде неструктурированного файла, мы создали небольшой REST API с использованием Flask[1], который обслуживает части базы на разных конечных точках.

Чтобы начать обслуживание API, мы предоставили небольшой файл Docker Compose, который создает несколько контейнеров: один для нашего REST API и еще пару для запуска самого Airflow. Оба контейнера можно запустить с помощью следующих команд:

```
$ cd chapter08
$ docker-compose up
```

После того как оба контейнера запустятся, вы сможете получить доступ к API на порту 5000 на локальном хосте (http://localhost:5000). Перейдя по этому адресу, вы должны увидеть на экране приветствие: *Hello from the Movie Rating API!* (рис. 8.2).

Рис. 8.2 Приветствие, которое вы увидите на экране

Нас в основном интересует получение рейтингов фильмов, которые предоставляются конечной точкой API /rating. Чтобы получить доступ к ней, перейдите на страницу http://localhost:5000/ratings. Вы должны увидеть на экране поля для ввода логина и пароля (рис. 8.3), поскольку эта часть API возвращает данные, которые могут содержать (потенциально) конфиденциальную информацию о пользователе. По умолчанию мы используем комбинацию airflow/airflow в качестве имени пользователя и пароля.

После ввода учетных данных вы должны получить начальный список рейтингов (рис. 8.4). Как видите, рейтинги возвращаются в формате JSON. В этом файле фактические рейтинги содержатся в ключе `result`, а два дополнительных поля, `limit` и `offset`, указывают на то, что мы просматриваем только одну страницу результатов (первые 100 оценок) и что потенциально доступно больше рейтингов (обозначается полем total, где описано общее количество записей, доступных для запроса).

[1] Код API доступен в репозитории, прилагаемом к этой книге.

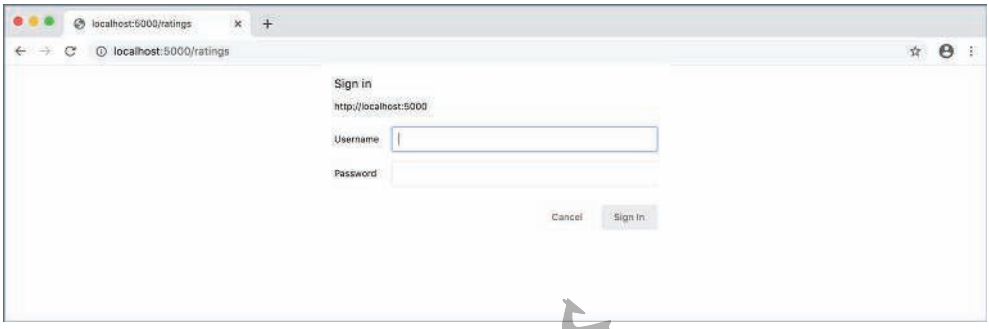


Рис. 8.3 Аутентификация

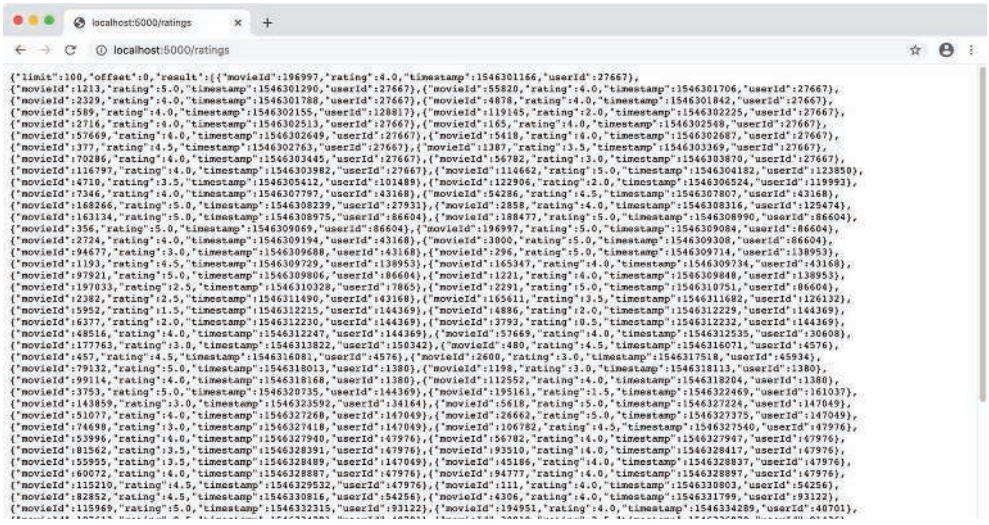


Рис. 8.4 Рейтинги, возвращаемые конечной точкой

Чтобы просмотреть результат запроса с разбивкой на страницы, можно использовать параметр `offset`. Например, чтобы получить следующий набор из 100 записей, можно добавить этот параметр со значением 100:

http://localhost:5000/ratings?offset=100

Также можно увеличить количество записей, извлекаемых в одном запросе, используя параметр `limit`:

http://localhost:5000/ratings?limit=1000

По умолчанию конечная точка возвращает все рейтинги, доступные в API. Чтобы получить рейтинги для определенного периода времени,

можно выбрать рейтинги между заданной датой начала и окончания, используя параметры[1] `start_date` и `end_date`:

http://localhost:5000/ratings?start_date=2019-01-01&end_date=2019-01-02

Такая возможность фильтрации позволит нам загружать данные из API по возрастающей (ежедневно), без необходимости загружать весь набор данных.

8.1.2 Получение оценок из API

Теперь, когда мы познакомились с API MovieLens, нам нужно приступить к извлечению рейтингов программными средствами, чтобы (позже) можно было автоматизировать этот процесс с помощью Airflow.

Для доступа к нашему API из Python можно использовать `requests` (https://docs.python-requests.org/en/master/), популярную и простую в применении библиотеку для выполнения HTTP-запросов. Чтобы начать отправку запросов с помощью нашего API, для начала нужно создать сеанс `requests` с использованием класса `Session`:

```
import requests
session = requests.Session()
```

Этот сеанс позволит нам получать рейтинги из API с помощью метода `get`, который выполняет HTTP-запрос:

```
response = session.get("http://localhost:5000/ratings")
```

Метод `get` также позволяет передавать дополнительные аргументы, такие как параметры (например, дата начала/окончания), чтобы включить их в запрос:

```
response = session.get(
    "http://localhost:5000/ratings",
    params={
        "start_date": "2019-01-01",
        "end_date": "2019-01-02",
    },
)
```

Вызов `get` вернет *объект ответа*, представляющий собой результат запроса. Этот объект можно использовать для проверки успешности запроса с помощью метода `raise_for_status`, который возбуждает исключение, если запрос вернул неожиданный код состояния. Мы мо-

[1] API предоставляет данные только 30-дневной давности, поэтому обязательно обновите параметры `start_date` и `end_date` на более свежие даты, чтобы получить результаты.

жем получить доступ к результату запроса с помощью атрибута content или, в этом случае, с помощью метода json (поскольку мы знаем, что наш API возвращает данные в формате JSON):

```
response.raise_for_status()
response.json()
```

Если выполнить этот запрос, то можно увидеть, что наши запросы не работают, так как мы забыли включить сюда аутентификацию. Поскольку наш API использует базовую HTTP-аутентификацию, мы можем настроить наш сеанс, чтобы включить сюда данные аутентификации:

```
movielens_user = "airflow"
movielens_password = "airflow"

session.auth = (movielens_user, movielens_password)
```

Это гарантирует, что сеанс запросов включает сюда аутентификацию по имени пользователя и паролю с запросами.

Инкапсулируем эту функциональность в функцию _get_session, которая будет обрабатывать настройку сеанса с аутентификацией, чтобы нам не пришлось беспокоиться об этом в других частях кода, и позволим этой функции возвращать базовый URL-адрес API, чтобы он также был определен в одном месте.

Листинг 8.1 Функция, создающая HTTP-сеанс для API

```
def _get_session():
    """ Создает сеанс requests для API Movielens."""
        session = requests.Session()
        session.auth = ("airflow", "airflow")

    base_url = "http://localhost:5000"

    return session, base_url
```

Создаем сеанс requests

Конфигурируем сеанс для базовой HTTP-аутентификации с этим именем пользователя и паролем

Возвращаем сеанс вместе с базовым URL-адресом API, чтобы мы также знали, где получить доступ к API

Чтобы это выглядело более приемлемо с точки зрения конфигурации, мы также можем указать имя пользователя и пароль и различные части нашего URL-адреса с использованием переменных окружения:

Листинг 8.2 Создание настраиваемой функции _get_session (dags/01_python.py)

Получаем сведения о конфигурации API из необязательных переменных окружения

```
MOVIELENS_HOST = os.environ.get("MOVIELENS_HOST", "movielens")
MOVIELENS_SCHEMA = os.environ.get("MOVIELENS_SCHEMA", "http")
MOVIELENS_PORT = os.environ.get("MOVIELENS_PORT", "5000")
```

```
MOVIELENS_USER = os.environ["MOVIELENS_USER"]
MOVIELENS_PASSWORD = os.environ["MOVIELENS_PASSWORD"]

def _get_session():
    """ Создает сеанс requests для API Movielens."""

    session = requests.Session()
    session.auth = (MOVIELENS_USER, MOVIELENS_PASSWORD)

    base_url = f"{MOVIELENS_SCHEMA}://{MOVIELENS_HOST}:{MOVIELENS_PORT}"

    return session, base_url

session, base_url = _get_session()
```

Извлекаем имя пользователя и пароль из двух обязательных переменных окружения

Используем полученную конфигурацию для создания сеанса и базового URL-адреса

Позже это позволит нам легко изменять данные параметры при запуске нашего сценария, определяя значения переменных окружения.

Теперь, когда у нас есть элементарная настройка для сеанса requests, нам нужно реализовать функции, которые будут прозрачно обрабатывать пагинацию API.

Один из способов сделать это – обернуть наш вызов `session.get` в код, который проверяет ответ API и продолжает запрашивать новые страницы, пока мы не достигнем общего количества записей.

Листинг 8.3 Вспомогательная функция для пагинации (dags/01_python.py)

```
def _get_with_pagination(session, url, params, batch_size=100):
    """
    Извлекает записи с помощью запроса GET с заданными URL и параметрами с учетом
    разбивки на страницы
    """

    offset = 0
    total = None
    while total is None or offset < total:
        response = session.get(
            url,
            params={
                **params,
                **{"offset": offset, "limit": batch_size}
            }
        )
        response.raise_for_status()
        response_json = response.json()

        yield from response_json["result"]

        offset += batch_size
        total = response_json["total"]
```

Отслеживаем, сколько записей мы получили и сколько записей следует ожидать

Продолжаем обход, пока не извлечем все записи. Обратите внимание, что проверка None предназначена для первого цикла, так как общее количество записей неизвестно до окончания первого цикла

Извлекаем новую страницу, начиная с заданного параметра offset

Проверяем состояние результата и выполняем анализ полученных данных в формате JSON

Передаем вызывающему объекту все полученные записи

Обновляем текущий offset и общее количество записей

Когда мы используем `yield from` для возврата результатов, эта функция, по сути, возвращает генератор отдельных записей, что означает, что нам больше не нужно беспокоиться о страницах результатов[1].

Единственное, чего не хватает, – это функции, которая связывает все это воедино и позволяет нам выполнять запросы к конечной точке, указывая даты начала и окончания для желаемого диапазона дат.

Листинг 8.4 Связываем все воедино в _get_ratings (dags/01_python.py)

```
def _get_ratings(start_date, end_date, batch_size=100):
    session, base_url = _get_session()

    yield from _get_with_pagination(
        session=session,
        url=base_url + "/ratings",
        params="start_date": start_date, "end_date": end_date},
        batch_size=batch_size,
    )

ratings = _get_ratings(session, base_url + "/ratings")
next(ratings)
list(ratings)
```

Получаем сеанс requests (с аутентификацией) плюс базовый URL-адрес для API

Используем функцию разбивки на страницы, чтобы прозрачно получить коллекцию записей

Убеждаемся, что мы используем конечную точку ratings

Ограничиваем страницы до определенного размера пакета

Извлекаем записи между заданными датами начала и окончания

Извлекаем одну запись ... или весь пакет

Пример использования функции _get_ratings

Теперь у нас есть красивая и лаконичная функция для получения рейтингов, которую мы можем использовать в нашем OAГ.

8.1.3 Создание фактического OAГ

Теперь, когда у нас есть функция `_get_ratings`, можно вызвать ее с помощью `PythonOperator`, чтобы получить рейтинги для каждого интервала. Когда у нас будут рейтинги, мы сможем выгрузить результаты в выходной файл в формате JSON с разбивкой по датам, чтобы при необходимости можно было с легкостью повторно запустить выборку.

Такую функциональность можно реализовать, написав небольшую функцию-обертку, которая позаботится о предоставлении дат начала и окончания и записи рейтингов в функцию вывода.

Листинг 8.5 Использование функции _get_ratings (dags/01_python.py)

```
def _fetch_ratings(templates_dict, batch_size=1000, **_):
    logger = logging.getLogger(__name__)
```

Используем журналирование, чтобы предоставить полезную обратную связь о том, что делает функция

[1] Дополнительное преимущество такой реализации состоит в том, что она ленивая: она будет извлекать новую страницу только тогда, когда записи из текущей страницы будут исчерпаны.

```
    start_date = templates_dict["start_date"]
    end_date = templates_dict["end_date"]
    output_path = templates_dict["output_path"]

    logger.info(f"Fetching ratings for {start_date} to {end_date}")
    ratings = list(
        _get_ratings(
            start_date=start_date,
            end_date=end_date,
            batch_size=batch_size,
        )
    )
    logger.info(f"Fetched {len(ratings)} ratings")

    logger.info(f"Writing ratings to {output_path}")

    output_dir = os.path.dirname(output_path)
    os.makedirs(output_dir, exist_ok=True)

    with open(output_path, "w") as file_:
        json.dump(ratings, fp=file_)

fetch_ratings = PythonOperator(
    task_id="fetch_ratings",
    python_callable=_fetch_ratings,
    templates_dict={
        "start_date": "{{ds}}",
          "end_date": "{{next_ds}}",
          "output_path": "/data/python/ratings/{{ds}}.json",
        },
    )
```

Извлекаем шаблонные даты начала и окончания и выходной путь

С помощью функции _get_ratings извлекаем записи о рейтингах

Создаем выходной каталог, если его не существует

Записываем выходные данные в формате JSON

Создаем задачу с помощью PythonOperator

Обратите внимание, что параметры `start_date/end_date/output_path` передаются с использованием `templates_dict`, что позволяет нам ссылаться на переменные контекста, такие как `execution_date`, в их значениях.

После извлечения рейтингов мы включаем сюда еще один шаг, `rank_movies`, для ранжирования фильмов. Здесь используется `PythonOperator`, чтобы применить функцию `rank_movies_by_rating`, которая ранжирует фильмы по среднему рейтингу с возможностью фильтрации по минимальному количеству оценок.

Листинг 8.6 Вспомогательная функция для ранжирования фильмов (dags/custom / rank.py)

```
import pandas as pd

def rank_movies_by_rating(ratings, min_ratings=2):
    ranking = (
        ratings.groupby("movieId")
        .agg(
            avg_rating=pd.NamedAgg(column="rating", aggfunc="mean"),
```

Вычисляем средний рейтинг и общее количество рейтингов

```
    Фильтрация по минимальному количеству требуемых рейтингов
                    num_ratings=pd.NamedAgg(column="userId", aggfunc="nunique"),
                )
                .loc[lambda df: df["num_ratings"] > min_ratings]
                .sort_values(["avg_rating", "num_ratings"], ascending=False)
            )                                                  Сортируем по средней оценке
            return ranking
```

Листинг 8.7 Добавление задачи rank_movies (dags / 01_python.py)

```
def _rank_movies(templates_dict, min_ratings=2, **_):
    input_path = templates_dict["input_path"]
           output_path = templates_dict["output_path"]

    ratings = pd.read_json(input_path)
    ranking = rank_movies_by_rating(ratings, min_ratings=min_ratings)

    output_dir = os.path.dirname(output_path)
    os.makedirs(output_dir, exist_ok=True)

    ranking.to_csv(output_path, index=True)


rank_movies = PythonOperator(
    task_id="rank_movies",
    python_callable=_rank_movies,
    templates_dict={
        "input_path": "/data/python/ratings/{{ds}}.json",
        "output_path": "/data/python/rankings/{{ds}}.csv",
    },
)

        fetch_ratings >> rank_movies
```

Читаем рейтинги из заданного (шаблонного) пути ввода

Создаем выходной каталог, если его не существует

Используем вспомогательную функцию для ранжирования фильмов

Используем функцию _rank_movies в PythonOperator

Записываем ранжированные фильмы в файл CSV

Подключаем задачи по извлечению и ранжированию

В результате мы получаем ОАГ, состоящий из двух этапов: один для извлечения рейтингов и второй для ранжирования фильмов. Если запланировать его таким образом, чтобы он запускался ежедневно, то это позволит составить рейтинг самых популярных фильмов на этот день (конечно, более умный алгоритм может учитывать историю, но с чего-то нужно начинать, верно?).

8.2 Создание собственного хука

Как видите, требуются некоторые усилия (и код), чтобы приступить к извлечению рейтингов из нашего API и использовать их для ранжирования. Интересно, что бóльшая часть нашего кода касается взаимодействия с API, в котором мы должны получить API-адрес и данные аутентификации, настроить сеанс для взаимодействия с API и включить дополнительные функции, например для пагинации.

Один из способов справиться со сложностью взаимодействия с API – инкапсулировать весь этот код в хук, который можно будет использовать повторно. Таким образом, мы можем хранить весь предназначенный для API код в одном месте и просто использовать этот хук в разных местах наших ОАГ, что позволяет сократить усилия по извлечению рейтингов.

Листинг 8.8 Использование MovielensHook для извлечения рейтингов

```
hook = MovielensHook(conn_id="movielens")
ratings = hook.get_ratings(start_date, end_date)
hook.close()
```

Создаем хук

Закрываем хук, освободив все использованные ресурсы

Используем хук для выполнения работы

Хуки также позволяют нам использовать функциональность Airflow для управления учетными данными для подключения через базу данных и пользовательский интерфейс, а это значит, что нам не нужно вручную указывать учетные данные API для своего ОАГ. В следующих разделах мы узнаем, как написать собственный хук, и приступим к созданию хука для API фильмов.

8.2.1 Создание собственного хука

В Airflow все хуки создаются в виде подклассов абстрактного класса `BaseHook`.

Листинг 8.9 Каркас пользовательского хука

```
from airflow.hooks.base_hook import BaseHook

class MovielensHook(BaseHook):
    …
```

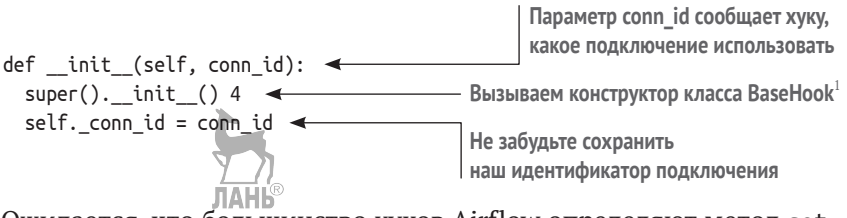
Чтобы приступить к созданию хука, нужно определить метод `__init__`, который указывает, какое подключение использует хук (если это применимо), и любые другие дополнительные аргументы, которые ему могут понадобиться. В данном случае мы хотим, чтобы наш хук получал нужные сведения из определенного подключения, но никаких дополнительных аргументов не требуется.

Листинг 8.10 Начальная часть класса MovielensHook (dags/custom/hooks.py)

```
from airflow.hooks.base_hook import BaseHook

class MovielensHook(BaseHook):
```

[1] В Airflow 1 конструктор класса BaseHook требует передачи аргумента `source`. Обычно можно просто передать `source = None`, так как вы не будете нигде его использовать.



Ожидается, что большинство хуков Airflow определяют метод `get_conn`, который отвечает за установку соединения с внешней системой. В нашем случае это означает, что мы можем повторно использовать бóльшую часть ранее определенной функции `_get_session`, которая уже предоставляет нам предварительно сконфигурированный сеанс для API. Это означает, что простейшая реализация `get_conn` может выглядеть примерно так:

Листинг 8.11 Начальная реализация метода get_conn

```
class MovielensHook(BaseHook):

    ...

    def get_conn(self):
        session = requests.Session()
        session.auth = (MOVIELENS_USER, MOVIELENS_PASSWORD)

        schema = MOVIELENS_SCHEMA
        host = MOVIELENS_HOST
        port = MOVIELENS_PORT

        base_url = f"{schema}://{host}:{port}"

        return session, base_url
```

Однако, вместо того чтобы вшивать в код наши учетные данные, мы предпочитаем получать их из хранилища учетных данных Airflow, что безопаснее и проще в управлении. Для этого нам сначала нужно добавить наше соединение в базу метаданных Airflow. Это можно сделать, открыв раздел **Admin > Connections** (Администратор > Подключения) с помощью веб-интерфейса Airflow и щелкнув **Create** (Создать), чтобы добавить новое подключение.

На экране создания подключения (рис. 8.5) необходимо указать сведения о подключении нашего API. В этом случае мы назовем его «movielens». Мы будем использовать этот идентификатор позже в коде для обозначения подключения. В типе подключения мы выбираем HTTP. В разделе **Host** нужно указать имя хоста API в настройке Docker Compose, которая называется «movielens». Далее можно (по желанию) указать схему, которую мы будем использовать для подключения (HTTP), и добавить необходимые учетные данные для входа (имя пользователя: airflow, пароль: airflow).

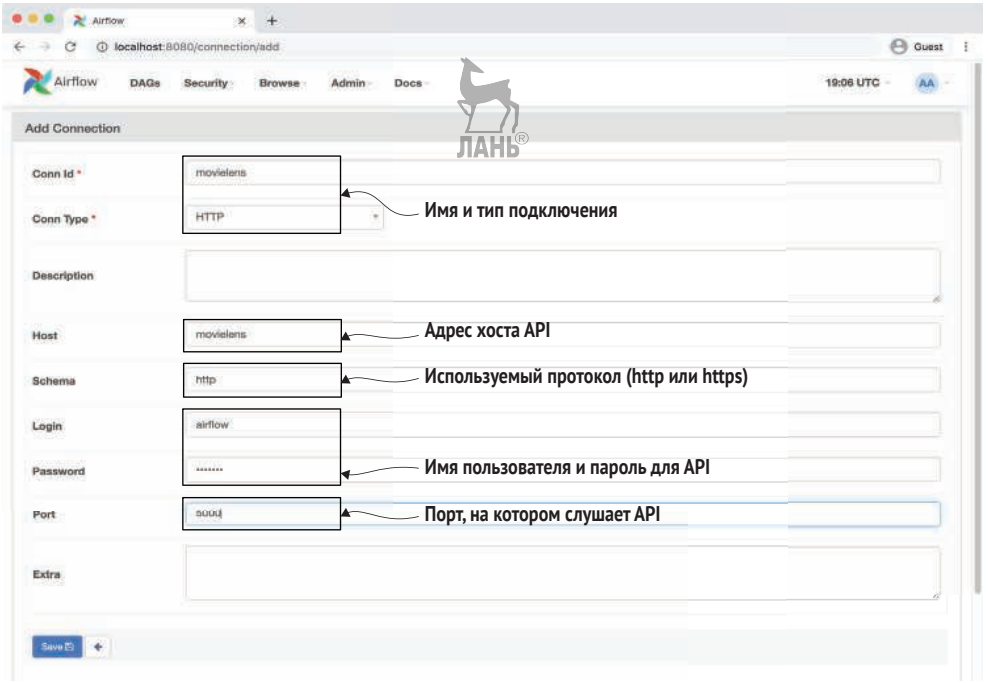


Рис. 8.5 Добавляем наше подключение в веб-интерфейсе Airflow

Наконец, нужно указать, на каком порту будет доступен наш API. В нашей настройке Docker Compose это порт 5000 (как мы уже видели ранее при доступе к API вручную).

Теперь, когда у нас есть подключение, нам нужно изменить метод `get_conn`, чтобы получить детали подключения из базы метаданных. Для этого класс `BaseHook` предоставляет удобный метод `get_connection`, который может получать нужные сведения для заданного идентификатора подключения из базы метаданных:

```
config = self.get_connection(self._conn_id)
```

У этого объекта есть поля, соответствующие различным деталям, которые мы только что заполнили при создании подключения. Таким образом, мы можем использовать объект `config`, чтобы приступить к определению хоста и порта и имени пользователя и пароля для нашего API. Сначала мы используем поля `schema`, `host` и `port`, чтобы определить URL-адрес нашего API, как и раньше:

```
schema = config.schema or self.DEFAULT_SCHEMA
host = config.host or self.DEFAULT_HOST
port = config.port or self.DEFAULT_PORT

base_url = f"{schema}://{host}:{port}/"
```

Обратите внимание, что мы определяем значения по умолчанию в нашем классе (аналогично константам, которые мы определили до этого) в случае, если эти поля не указаны в подключении. Если мы хотим, чтобы они были указаны в самом подключении, то можем выдать ошибку, вместо того чтобы указывать значения по умолчанию.

Теперь, когда мы получили наш базовый URL-адрес из базы метаданных, нужно только сконфигурировать детали аутентификации в сеансе:

```
if config.login:
   session.auth = (config.login, config.password)
```

что дает нам следующую новую реализацию метода `get_conn`:

Листинг 8.12 Создание конфигурируемого метода get_conn (dags/custom/hooks.py)

```
class MovielensHook(BaseHook):
   DEFAULT_HOST = "movielens"
   DEFAULT_SCHEMA = "http"
   DEFAULT_PORT = 5000

   def __init__(self, conn_id):
       super().__init__()
       self._conn_id = conn_id

   def get_conn(self):
       config = self.get_connection(self._conn_id)

       schema = config.schema or self.DEFAULT_SCHEMA
       host = config.host or self.DEFAULT_HOST
       port = config.port or self.DEFAULT_PORT

       base_url = f"{schema}://{host}:{port}"

       session = requests.Session()

       if config.login:
           session.auth = (config.login, config.password)

       return session, base_url
```

Значения подключения по умолчанию как сохраненные переменные класса для удобства

Получение конфигурации подключения с использованием заданного идентификатора

Создание базового URL-адреса с использованием конфигурации подключения и значений по умолчанию

Создание сессии requests с использованием логина и пароля из конфигурации подключения

Возврат сеанса requests и базового URL-адреса

Один из недостатков такой реализации состоит в том, что каждый вызов функции `get_conn` будет приводить к вызову к базе метаданных Airflow, поскольку этой функции необходимо получать учетные данные из базы данных. Можно избежать этого ограничения, также кешируя `session` и `base_url` в нашем экземпляре как защищенные переменные.

Листинг 8.13 Добавление кеширования для сеанса API (dags/custom/hooks.py)

```
class MovielensHook(BaseHook):

    def __init__(self, conn_id, retry=3):
        ...
        self._session = None
        self._base_url = None
    def get_conn(self):
        ""
        Возвращает соединение, используемое хуком для запроса данных.
        В принципе, не следует использовать это напрямую.
        """

        if self._session is None:
            config = self.get_connection(self._conn_id)
              ...
            self._base_url = f"{schema}://{config.host}:{port}"
          self._session = requests.Session()
           ...

        return self._session, self._base_url
```

Две дополнительные переменные экземпляра, используемые для кеширования сеанса и базового URL-адреса

Проверяем, есть ли у нас активный сеанс, прежде чем создавать его

Таким образом, при первом вызове метода `get_conn` `self._session` имеет значение `None`, поэтому в конечном итоге мы получаем данные о нашем подключении из базы метаданных и настраиваем базовый URL-адрес и сеанс. Сохраняя эти объекты в переменных экземпляра `_session` и `_base_url`, мы гарантируем, что они кешируются для последующих вызовов. Таким образом, во время второго вызова `get_conn` увидит, что у `self._session` больше нет значения `None`, и вернет кешированный сеанс и базовый URL-адрес.

ПРИМЕЧАНИЕ Лично мы не являемся поклонниками использования метода `get_conn` непосредственно за пределами хука, даже если доступ к нему открыт, поскольку этот метод раскрывает внутренние детали того, как ваш хук обращается к внешней системе, нарушая инкапсуляцию. Это доставит вам много головной боли, если вы когда-нибудь захотите изменить эту внутреннюю деталь, поскольку ваш код будет сильно привязан к типу внутреннего соединения. Это также было проблемой в кодовой базе Airflow, например в случае с `HdfsHook`, где реализация хука была тесно связана с библиотекой только для Python 2.7 (snakebite).

Теперь, когда мы завершили реализацию метода `get_conn`, можно создать аутентифицированное соединение с API. Это означает, что мы, наконец, можем приступить к встраиванию полезных методов в хук, которые мы можем затем использовать, чтобы сделать что-нибудь полезное с нашим API.

Для получения рейтингов можно повторно использовать код из предыдущей реализации, которая извлекала рейтинги из конечной точки /rating и использовала функцию `get_with_ pagination` для разбивки на страницы. Основное отличие от предыдущей версии заключается в том, что теперь мы используем `get_conn` в рамках функции разбивки на страницы для получения сеанса API.

Листинг 8.14 Добавление метода get_ratings (dags/custom/hooks.py)

```
class MovielensHook(BaseHook):
    ...

    def get_ratings(self, start_date=None, end_date=None, batch_size=100):
        """
        Извлекает рейтинги между заданной датой начала и окончания.
        Параметры
        -----
        start_date : str
            Дата начала, с которой начинается получение рейтингов (включительно).
            Ожидаемый формат - ГГГГ-ММ-ДД (соответствует форматам ds в Airflow).
        end_date : str
            Дата окончания для получения оценок до (исключая). Ожидаемый
            формат - ГГГГ-ММ-ДД (соответствует форматам ds в Airflow).
        batch_size : int
            Размер пакетов (страниц), которые нужно извлечь из API. Большие
            значения означают меньше запросов, но больше данных передается
            на каждый запрос
        """

        yield from self._get_with_pagination(
            endpoint="/ratings",
            params={"start_date": start_date, "end_date": end_date},
            batch_size=batch_size,
        )

    def _get_with_pagination(self, endpoint, params, batch_size=100):
        """
        Извлекает записи, используя запрос get с заданными url и params,
        с учетом разбивки на страницы.
        """

        session, base_url = self.get_conn()

        offset = 0
        total = None
        while total is None or offset < total:
            response = session.get(
                url, params={
                    **params,
                    **{"offset": offset, "limit": batch_size}
                }
            )
```

Открытый метод, который будет вызываться пользователями хука

Наш внутренний вспомогательный метод, занимающийся пагинацией (та же реализация, что и раньше)

```
        response.raise_for_status()
        response_json = response.json()

        yield from response_json["result"]

        offset += batch_size
        total = response_json["total"]
```

В целом это дает нам базовый хук Airflow, который обрабатывает подключения к API MovieLens. Добавление дополнительных функций (кроме извлечения рейтингов) можно легко выполнить, добавив в хук дополнительные методы.

Хотя создание хука может показаться трудоемким делом, бóльшая часть работы была перенесена на функции, которые мы написали ранее, в единый консолидированный класс. Преимущество нашего нового хука состоит в том, что он обеспечивает красивую инкапсуляцию логики API MovieLens в одном классе, который легко использовать в разных OAГ.

8.2.2 Создание OAГ с помощью MovielensHook

Теперь, когда у нас есть хук, можно приступить к его использованию для получения рейтингов в нашем OAГ. Однако сначала нужно где-то сохранить наш класс, чтобы мы могли импортировать его в OAГ. Один из способов – создать пакет в том же каталоге, что и папка DAG[1], и сохранить хук в модуле `hooks.py` внутри этого пакета.

Листинг 8.15 Структура каталога OAГ с пакетом custom

```
chapter08
├── dags
│   ├── custom          <-- Пример пакета с именем «custom»
│   │   ├── __init__.py
│   │   └── hooks.py    <-- Модуль, содержащий код хука
│   ├── 01_python.py
│   └── 02_hook.py
├── docker-compose.yml
└── ...
```

Когда у нас появится этот пакет, мы сможем импортировать хук из нового пакета `custom`, который содержит код нашего хука:

```
from custom.hooks import MovielensHook
```

После импорта хука извлечь рейтинг становится довольно просто. Нам нужно только создать экземпляр хука с правильным идентификатором соединения, а затем вызвать его метод `get_ratings` с желаемыми датами начала и окончания.

[1] Позже в этой главе мы покажем другой подход на базе пакетов.

Листинг 8.16 Использование MovielensHook для получения рейтингов

```
hook = MovielensHook(conn_id=conn_id)
ratings = hook.get_ratings(
   start_date=start_date,
   end_date=end_date,
   batch_size=batch_size
)
```

Этот код возвращает генератор рейтингов, которые мы затем записываем в выходной (JSON) файл.

Чтобы использовать хук в нашем ОАГ, нам все еще нужно обернуть этот код в `PythonOperator`, который позаботится о предоставлении правильных дат начала и окончания запуска ОАГ, а также, собственно, о записи рейтингов в нужный выходной файл. Для этого мы, по сути, можем использовать ту же функцию `_fetch_ratings`, которую мы определили для нашего исходного ОАГ, заменив вызов `_get_ratings` на вызов нового хука.

Листинг 8.17 Использование MovielensHook в DAG (dags/02_hook.py)

```
def _fetch_ratings(conn_id, templates_dict, batch_size=1000, **_):
   logger = logging.getLogger(__name__)

   start_date = templates_dict["start_date"]
   end_date = templates_dict["end_date"]
   output_path = templates_dict["output_path"]

   logger.info(f"Fetching ratings for {start_date} to {end_date}")
   hook = MovielensHook(conn_id=conn_id)
   ratings = list(
       hook.get_ratings(
           start_date=start_date, end_date=end_date, batch_size=batch_size
       )
   )
   logger.info(f"Fetched {len(ratings)} ratings")

   logger.info(f"Writing ratings to {output_path}")

   output_dir = os.path.dirname(output_path)
   os.makedirs(output_dir, exist_ok=True)

   with open(output_path, "w") as file_:
       json.dump(ratings, fp=file_)

PythonOperator(
   task_id="fetch_ratings",
   python_callable=_fetch_ratings,
   op_kwargs={"conn_id": "movielens"},
   templates_dict={
       "start_date": "{{ds}}",
       "end_date": "{{next_ds}}",
```

Используем хук для извлечения рейтингов из API

Создаем экземпляр MovielensHook с соответствующим идентификатором подключения

Записываем полученные рейтинги, как и раньше

Указываем, какое подключение использовать

```
        "output_path": "/data/custom_hook/{{ds}}.json",
    },
)
```

Обратите внимание, что мы добавили в `fetch_ratings` параметр `conn_id`, чтобы указать подключение, используемое для хука. Таким образом, нам также необходимо включить этот параметр при вызове функции `_fetch_ratings` из оператора `PythonOperator`.

Это дает нам то же поведение, что и раньше, но теперь у нас гораздо более простой и не такой большой файл ОАГ, так как бóльшая часть сложностей, связанных с API MovieLens, теперь отдана на откуп `MovielensHook`.

8.3 Создание собственного оператора

Хотя создание `MovielensHook` позволило нам перенести значительную часть сложного кода из нашего ОАГ в хук, нам все равно приходится писать значительное количество шаблонного кода для определения дат начала и окончания и записи рейтингов в выходной файл. Это означает, что если мы захотим повторно использовать такую функциональность в нескольких ОАГ, у нас по-прежнему будет много дублированного кода и придется потратить дополнительные усилия.

К счастью, Airflow позволяет создавать собственные операторы, которые можно использовать для выполнения повторяющихся задач с минимальным количеством шаблонного кода. В этом случае можно было бы, например, использовать эту возможность для создания `MovielensFetchRatingsOperator`, что позволит нам получать рейтинги фильмов с помощью специального класса.

8.3.1 Определение собственного оператора

В Airflow все операторы созданы в виде подклассов класса `BaseOperator`.

Листинг 8.18 Каркас пользовательского оператора

```
from airflow.models import BaseOperator
from airflow.utils.decorators import apply_defaults

class MyCustomOperator(BaseOperator):
    @apply_defaults
    def __init__(self, conn_id, **kwargs):
        super().__init__(self, **kwargs)
        self._conn_id = conn_id
        ...
```

Наследуем от класса BaseOperator

Декоратор, который гарантирует, что аргументы ОАГ по умолчанию передаются нашему оператору

Передаем все дополнительные ключевые аргументы конструктору BaseOperator

Можно явно указать любые аргументы, относящиеся к вашему оператору (например, `conn_id` в этом примере) в методе конструктора `__init__`. Как вы будете использовать эти аргументы, конечно, зависит от вас. Аргументы конкретного оператора различаются в зависимости от оператора, но обычно включают в себя идентификаторы подключения (для операторов, использующих удаленные системы) и любые сведения, необходимые для операции (например, даты начала и окончания, запросы и т. д.).

Класс `BaseOperator` также принимает большое количество (в основном необязательных) универсальных аргументов, которые определяют базовое поведение оператора. Примеры универсальных аргументов включают в себя `task_id`, созданный оператором для задачи, а также множество аргументов, таких как `retries` и `retry_delay`, которые влияют на планирование итоговой задачи. Чтобы избежать явного перечисления всех этих задач, мы используем синтаксис Python `**kwargs` для перенаправления этих универсальных аргументов в метод конструктора `__init__` класса `BaseOperator`.

Вспоминая предыдущие ОАГ из этой книги, вы, наверное, помните, что Airflow также предоставляет возможность определения конкретных аргументов в качестве аргументов по умолчанию для всего ОАГ. Это делается с помощью параметра `default_args` самого объекта ОАГ.

Листинг 8.19 Применение аргументов по умолчанию к операторам

```
default_args = {
   "retries": 1,
   "retry_delay": timedelta(minutes=5),
}

with DAG(
   ...
   default_args=default_args
) as dag:
   MyCustomOperator(
       ...
   )
```

Чтобы гарантировать, что эти аргументы по умолчанию применяются к вашему пользовательскому оператору, Airflow предоставляет декоратор `apply_defaults`, который применяется к методу `__init__` оператора (как показано в начальном примере). На практике это означает, что вы всегда должны включать этот декоратор при определении собственных операторов; в противном случае вы непроизвольно нарушите поведение Airflow.

Теперь, когда у нас есть базовый класс пользовательских операторов, нам еще нужно определить, что на самом деле делает оператор, реализуя метод `execute`, основной метод, который Airflow вызывает, когда оператор фактически выполняется как часть запуска ОАГ.

Листинг 8.20 Метод оператора execute

```
class MyCustomOperator(BaseOperator):
    ...

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

Основной метод, вызываемый при выполнении нашего оператора

Как видите, метод `execute` принимает единственный параметр `context`, представляющий собой словарь, содержащий все контекстные переменные Airflow. Затем метод может продолжить выполнение той функции, для которой был разработан оператор, с учетом переменных из контекста Airflow (например, даты выполнения и т. д.).

8.3.2 Создание оператора для извлечения рейтингов

Теперь, когда мы знаем основы создания оператора, посмотрим, как создать собственный оператор для извлечения рейтингов. Идея состоит в том, что этот оператор извлекает рейтинги из API MovieLens между заданной датой начала и датой окончания и записывает их в файл JSON, аналогично тому, что наша функция `_fetch_ratings` делала в предыдущем ОАГ.

Мы можем начать с заполнения необходимых параметров оператора в его методе `__init__`, которые включают в себя даты начала и окончания, какое подключение использовать и выходной путь, куда нужно вести запись.

Листинг 8.21 Начало пользовательского оператора (dags/custom/operator.py)

```
class MovielensFetchRatingsOperator(BaseOperator):
    """
    Оператор, извлекающий рейтинги из API Movielens.

    Параметры
    -----
    conn_id : str
        ID подключения, который будет использоваться для подключения к API
        Movielens.
        Ожидается, что подключение будет включать в себя данные аутентификации
        (логин/пароль) и хост, обслуживающий API.
    output_path : str
        Путь для записи полученных рейтингов.
    start_date : str
        (Шаблонная) дата начала, с которой начинается извлечение рейтингов
        (включительно).
        Ожидаемый формат - ГГГГ-ММ-ДД (соответствует форматам ds в Airflow).
    end_date : str
        (Шаблонная) дата окончания для извлечения рейтингов до (исключая).
        Ожидаемый формат - ГГГГ-ММ-ДД (соответствует форматам ds в Airflow).
    """
```

```
@apply_defaults
def __init__(
   self, conn_id, output_path, start_date, end_date, **kwargs,
):
   super(MovielensFetchRatingsOperator, self).__init__(**kwargs)

   self._conn_id = conn_id
   self._output_path = output_path
   self._start_date = start_date
   self._end_date = end_date
```

Затем нужно реализовать тело оператора, который фактически извлекает рейтинги и записывает их в выходной файл. Для этого можно заполнить метод оператора `execute` модифицированной версией нашей реализации функции `_fetch_ratings`.

Листинг 8.22 Добавление метода execute (dags/custom/operator.py)

```
class MovielensFetchRatingsOperator(BaseOperator):
   ...

   def execute(self, context):
       hook = MovielensHook(self._conn_id)

       try:
           self.log.info(
               f"Fetching ratings for {self._start_date} to {self._end_date}"
           )
           ratings = list(
               hook.get_ratings(
                   start_date=self._start_date,
                   end_date=self._end_date,
               )
           )
           self.log.info(f"Fetched {len(ratings)} ratings")
       finally:
           hook.close()

       self.log.info(f"Writing ratings to {self._output_path}")

       output_dir = os.path.dirname(self._output_path)
       os.makedirs(output_dir, exist_ok=True)
       with open(self._output_path, "w") as file_:
           json.dump(ratings, fp=file_)
```

Создаем экземпляр MovielensHook

Используем хук для извлечения рейтингов

Закрываем хук, чтобы высвободить ресурсы

Создаем выходной каталог, если его не существует

Выписываем результаты

Как видите, перенос кода в пользовательский оператор потребовал относительно небольшого количества изменений в коде. Подобно функции `_fetch_ratings`, метод `execute` начинается с создания экземпляра `MovielensHook` и использования этого хука для извлечения рейтингов между заданными датами начала и окончания. Есть одно отличие, которое состоит в том, что теперь код принимает параметры от `self`, не забывая использовать значения, переданные при создании экземпляра оператора. Кроме того, теперь мы используем регист-

ратор, предоставленный классом `BaseOperator`, который доступен в свойстве `self.log`. Наконец, мы добавили обработку исключений, чтобы наш хук всегда закрывался должным образом, даже если вызов метода `get_ratings` не сработает. Таким образом, мы не тратим ресурсы впустую, забывая закрыть сеансы API, а это хорошая практика при реализации кода, применяющего хуки.

Использовать этот оператор относительно просто, поскольку мы можем просто создать экземпляр оператора и включить его в свой ОАГ.

Листинг 8.23 Использование MovielensFetchRatingsOperator

```
fetch_ratings = MovielensFetchRatingsOperator(
    task_id="fetch_ratings",
    conn_id="movielens",
    start_date="2020-01-01",
    end_date="2020-01-02",
    output_path="/data/2020-01-01.json"
)
```

Недостаток этой реализации состоит в том, что для нее требуются заранее определенные даты, для которых оператор будет извлекать рейтинги. Таким образом, оператор будет извлекать их только за один период времени, вшитый в код, не принимая во внимание дату выполнения.

К счастью, Airflow также позволяет делать определенные переменные оператора шаблонизируемыми. Это означает, что они могут ссылаться на переменные контекста, такие как `execution_date`. Чтобы разрешить шаблонизацию конкретных переменных экземпляра, нужно указать Airflow, чтобы он шаблонизировал их с помощью переменной класса `templates_field`.

Листинг 8.24 Добавляем `templates_field` (dags/custom/operator.py)

```
class MovielensFetchRatingsOperator(BaseOperator):
    ...
    template_fields = ("_start_date", "_end_date", "_output_path")
    ...

    @apply_defaults
    def __init__(
        self,
        conn_id,
        output_path,
        start_date="{{ds}}",
        end_date="{{next_ds}}",
        **kwargs,
    ):
        super(MovielensFetchRatingsOperator, self).__init__(**kwargs)

        self._conn_id = conn_id
        self._output_path = output_path
```

Даем указание Airflow шаблонизировать эти переменные экземпляра в нашем операторе

```
        self._start_date = start_date
        self._end_date = end_date
```

По сути, этот код сообщает Airflow, что переменные `_start_date`, `_end_date` и `_output_path` (которые создаются в методе `__init__`) доступны для создания шаблонов. Это означает, что если мы используем шаблон Jinja в этих строковых параметрах, то Airflow позаботится о том, чтобы эти значения были шаблонизированы до того, как будет вызван метод `execute`. В результате мы теперь можем использовать наш оператор с шаблонными аргументами следующим образом:

Листинг 8.25 Использование шаблонизации в операторе (dags/03_operator.py)

```
from custom.operators import MovielensFetchRatingsOperator

fetch_ratings = MovielensFetchRatingsOperator(
   task_id="fetch_ratings",
   conn_id="movielens",
   start_date="{{ds}}",
   end_date="{{next_ds}}",
   output_path="/data/custom_operator/{{ds}}.json"
)
```

Таким образом, Airflow заполнит значения начала окна выполнения (`ds`) для даты начала и конца окна выполнения (`next_ds`) для даты окончания. Он также обеспечит запись вывода в файл с тегом начала окна выполнения (`ds`).

8.4 Создание нестандартных сенсоров

За всеми этими разговорами об операторах вы можете задаться вопросом, сколько усилий нужно, чтобы создать собственный сенсор. Если вы пропустили эту тему в предыдущих главах, напомним, что сенсор – это особый тип оператора, который можно использовать для ожидания выполнения определенного условия, прежде чем выполнить какие-либо последующие задачи в ОАГ. Например, возможно, вы захотите использовать сенсор для проверки наличия определенных файлов или данных в исходной системе, прежде чем попытаться использовать данные для последующего анализа.

По своей реализации сенсоры очень похожи на операторов, за исключением того, что они наследуют от класса `BaseSensorOperator`, а не от `BaseOperator`.

Листинг 8.26 Каркас пользовательского сенсора

```
from airflow.sensors.base import BaseSensorOperator

class MyCustomSensor(BaseSensorOperator):
    ...
```

Как следует из названия, это показывает, что сенсоры на самом деле являются особым типом оператора. Класс `BaseSensorOperator` предоставляет базовые функции сенсора и требует, чтобы сенсоры реализовали специальный метод `poke` вместо метода `execute`.

Листинг 8.27 Метод сенсора poke

```
class MyCustomSensor(BaseSensorOperator):

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

Сигнатура метода `poke` похожа на метод `execute` в том смысле, что он принимает единственный аргумент, содержащий контекст Airflow. Однако, в отличие от метода `execute`, ожидается, что `poke` вернет логическое значение, которое указывает, истинно ли условие сенсора. Если это так, то сенсор завершит свое выполнение, позволяя запускать нижестоящие задачи. В противном случае сенсор бездействует несколько секунд перед повторной проверкой условия. Этот процесс повторяется до тех пор, пока условие не станет истинным или сенсор не достигнет тайм-аута.

Хотя у Airflow имеется множество встроенных сенсоров, вы можете создать собственный сенсор для проверки любого типа состояния. Например, в нашем случае использования нам может понадобиться реализовать сенсор, который сначала проверяет, доступны ли данные рейтинга на заданную дату, прежде чем продолжить выполнение ОАГ.

Чтобы приступить к созданию `MovielensRatingsSensor`, сначала нужно определить метод класса пользовательского сенсора, `__init__`, который должен принимать идентификатор подключения (он определяет, какие сведения о подключении использовать для API), и диапазон дат начала и окончания, который определяет, для какого диапазона дат мы хотим посмотреть рейтинги. Выглядит это примерно так:

Листинг 8.28 Начало класса сенсора (dags/custom/sensor.py)

```
from airflow.sensors.base import BaseSensorOperator
from airflow.utils.decorators import apply_defaults

class MovielensRatingsSensor(BaseSensorOperator):
    """
    Сенсор, ожидающий, пока API Movielens получит рейтинги за определенный период
    времени.

    start_date : str
        (Шаблонная) дата начала, с которой начинается извлечение рейтингов
        (включительно).
        Ожидаемый формат - ГГГГ-ММ-ДД (соответствует форматам ds в Airflow).
    end_date : str
        (Шаблонная) дата окончания для извлечения рейтингов до (исключая).
        Ожидаемый формат - ГГГГ-ММ-ДД (соответствует форматам ds в Airflow).
    """
```

```
template_fields = ("_start_date", "_end_date")

@apply_defaults
def __init__(self, conn_id, start_date="{{ds}}",
             end_date="{{next_ds}}", **kwargs):
    super().__init__(**kwargs)
    self._conn_id = conn_id
    self._start_date = start_date
    self._end_date = end_date
```

Поскольку сенсоры – это особый тип оператора, можно использовать ту же базовую настройку, что и для реализации оператора

После указания конструктора единственное, что нам нужно реализовать, – это метод poke. В нем мы можем проверить, есть ли рейтинги для определенного диапазона дат, просто запросив рейтинги между заданными датами начала и окончания. При наличии каких-либо записей возвращается логическое значение true. Обратите внимание, что для этого не требуется извлекать все записи; нам нужно только продемонстрировать, что в диапазоне есть хотя бы одна запись.

Используя MovielensHook, реализовать этот алгоритм довольно просто. Сначала мы создаем экземпляр хука, а затем вызываем метод get_ratings, чтобы приступить к извлечению записей. Поскольку нас интересует только наличие хотя бы одной записи, можно попробовать вызвать метод генератора, next, возвращаемого get_ratings, что приведет к вызову StopIteration, если генератор пуст. Таким образом, можно проверить исключение, используя конструкцию try/except, и вернуть True, если исключение не возбуждается, или в противном случае вернуть False (это указывает на то, что записей не было).

Листинг 8.29 Реализация метода poke (dags/custom/sensor.py)

```
class MovielensRatingsSensor(BaseSensorOperator):
    def poke(self, context):
        hook = MovielensHook(self._conn_id)

        try:
            next(
                hook.get_ratings(
                    start_date=self._start_date,
                    end_date=self._end_date,
                    batch_size=1
                )
            )
            self.log.info(
                f"Found ratings for {self._start_date} to {self._end_date}"
            )
            return True
        except StopIteration:
            self.log.info(
                f"Didn't find any ratings for {self._start_date} "
                f"to {self._end_date}, waiting..."
            )
            return False
        finally:
            hook.close()
```

Пытаемся извлечь одну запись из хука (используя метод next, чтобы извлечь первую запись)

Если все прошло успешно, у нас будет хотя бы одна запись, поэтому возвращаем true

Если с StopIteration ничего не получилось, значит, набор записей пуст, поэтому возвращаем false

Закрываем хук для высвобождения ресурсов

Обратите внимание, что повторное использование `MovielensHook` делает этот код относительно лаконичным, демонстрируя мощь содержания деталей взаимодействия с API MovieLens внутри хука.

Данный сенсор теперь можно использовать для проверки ОАГ и ожидания новых рейтингов, прежде чем продолжить выполнение остальной части ОАГ.

Листинг 8.30 Использование сенсора для ожидания рейтингов (dags/04_sensor.py)

```
...

from custom.operators import MovielensFetchRatingsOperator
from custom.sensors import MovielensRatingsSensor

with DAG(
   dag_id="04_sensor",
   description="Fetches ratings with a custom sensor.",
   start_date=airflow_utils.dates.days_ago(7),
   schedule_interval="@daily",
) as dag:
   wait_for_ratings = MovielensRatingsSensor(
      task_id="wait_for_ratings",
      conn_id="movielens",
      start_date="{{ds}}",
      end_date="{{next_ds}}",
   )

   fetch_ratings = MovielensFetchRatingsOperator(
       task_id="fetch_ratings",
       conn_id="movielens",
       start_date="{{ds}}",
       end_date="{{next_ds}}",
       output_path="/data/custom_sensor/{{ds}}.json"
   )

   ...

   wait_for_ratings >> fetch_ratings >> rank_movies
```

Сенсор, ожидающий, когда записи будут доступны

Оператор, извлекающий записи после завершения работы сенсора

8.5 Упаковка компонентов

До сих пор мы использовали включение своих пользовательских компонентов в подпакет в каталоге ОАГ, чтобы эти ОАГ могли их импортировать. Однако такой подход не всегда идеален, если вы хотите иметь возможность использовать эти компоненты в других проектах, делиться ими с другими или провести более тщательное тестирование.

Более подходящий подход к распространению компонентов – поместить свой код в пакет Python. Хотя это и требует дополнительных

затрат на настройку, это дает вам возможность устанавливать свои компоненты в окружении Airflow, как и в случае с любым другим пакетом. Более того, сохраняя код отдельно от ОАГ, вы можете настроить надлежащий процесс непрерывной интеграции и доставки для своего пользовательского кода, и вам будет проще делиться им или совместно работать над ним с другими программистами.

8.5.1 Создание пакета Python

К сожалению, упаковка – сложная тема. В данном случае мы сосредоточимся на самом простом примере, который включает в себя использование `setuptools` для создания простого пакета Python[1]. Используя данный подход, нам нужно создать небольшой пакет, `airflow_movielens`, который будет содержать хук, оператор и сенсор, написанные в предыдущих разделах.

Чтобы приступить к сборке нашего пакета, для начала создадим для него каталог:

```
$ mkdir -p airflow-movielens
$ cd airflow-movielens
```

Затем начнем с включения кода, создав основу для пакета. С этой целью мы создадим подкаталог `src` в нашем каталоге `airflow-movielens` и каталог `airflow_movielens` (имя нашего пакета) внутри каталога `src`.

Чтобы превратить `airflow_movielens` в пакет, мы также создадим файл `__init__.py` внутри каталога[2]:

```
$ mkdir -p src/airflow_movielens
$ touch src/airflow_movielens/__init__.py
```

Затем мы можем создать файлы `hooks.py`, `sensor.py` и `operator.py` в каталоге `airflow_movielens` и копировать реализации наших пользовательских хуков, сенсоров и операторов в соответствующие файлы. После этого вы должны получить примерно такой результат:

```
$ tree airflow-movielens/
airflow-movielens/
└── src
    └── airflow_movielens
        ├── __init__.py
        ├── hooks.py
        ├── operators.py
        └── sensors.py
```


[1] Более подробное обсуждение упаковки и различных подходов к ней выходит за рамки этого издания и объясняется во многих книгах по Python и/или онлайн-статьях.

[2] Технически с использованием стандарта PEP420 файл `__init__.py` уже не нужен, но мы хотим, чтобы все было явным.

Теперь, когда у нас есть базовая структура, все, что нам нужно сделать, чтобы превратить это в пакет, – включить сюда файл `setup.py`, сообщающий `setuptools`, как его установить. Базовый файл `setup.py` обычно выглядит примерно так:

Листинг 8.31 Пример файла setup.py (package/airflow-movielens/setup.py)

```
#!/usr/bin/env python
import setuptools
requirements = ["apache-airflow", "requests"]

setuptools.setup(
    name="airflow_movielens",
    version="0.1.0",
    description="Hooks, sensors and operators for the Movielens API.",
    author="Anonymous",
    author_email="anonymous@example.com",
    install_requires=requirements,
    packages=setuptools.find_packages("src"),
    package_dir={"": "src"},
    url="https://github.com/example-repo/airflow_movielens",
    license="MIT license",
)
```

Список пакетов Python, от которых зависит наш пакет

Название, версия и описание нашего пакета

Сведения об авторе (метаданные)

Информирует setuptools о наших зависимостях

Домашняя страница пакета

Лицензия MIT

Сообщает setuptools, где искать файлы Python нашего пакета

Самая важная часть этого файла – это вызов `setuptools.setup`, который предоставляет `setuptools` подробные метаданные о нашем пакете. Наиболее важные поля в этом вызове:

- *name* – определяет имя пакета (как он будет называться при установке);
- *version* – номер версии пакета;
- *install_requires* – список зависимостей, необходимых для пакета;
- *packages/package_dir* – сообщает setuptools, какие пакеты следует включить при установке и где их искать. Здесь мы используем макет каталога src для нашего пакета Python[1].

Кроме того, `setuptools` позволяет включать множество дополнительных полей[2] для описания пакета, среди которых:

- *author* – имя автора пакета (вы);
- *author_email* – контактные данные автора;
- *description* – краткое, читабельное описание пакета (обычно одна строка). Более подробное описание можно дать с помощью аргумента `long_description`;

[1] *См.* этот блог для получения более подробной информации о структурах на базе src и других вариантах: https://blog.ionelmc.ro/2014/05/25/python-packaging/#the-structure.

[2] Чтобы получить полный справочник параметров, которые можно передать в `setuptools.setup`, обратитесь к документации по `setuptools`.

- *url* – где искать ваш пакет в интернете;
- *license* – лицензия, под которой выпущен код вашего пакета (если таковая имеется).

Если посмотреть на реализацию `setup.py`, то это означает, что мы сообщаем `setuptools`, что наши зависимости включают `apache-airflow` и `requests`, что наш пакет должен называться `airflow_movielens` и иметь версию 0.1 и что он должен содержать файлы из пакета `airflow_movielens`, расположенного в каталоге `src`, включая дополнительные сведения о нас и лицензию.

Когда мы закончим писать файл `setup.py`, пакет должен выглядеть так:

```
$ tree airflow-movielens
airflow-movielens
├── setup.py
└── src
    └── airflow_movielens
        ├── __init__.py
        ├── hooks.py
        ├── operators.py
        └── sensors.py
```

Это означает, что теперь у нас есть базовый пакет Python `airflow_movielens`, который можно попробовать установить в следующем разделе.

Конечно, более сложные пакеты обычно включают в себя тесты, документацию и т. д. Все это мы здесь не описываем. Если вы хотите увидеть обширную настройку упаковки в Python, то рекомендуем обратиться к большому количеству шаблонов, доступных в интернете (например, https://github.com/audreyfeldroy/cookiecutter-pypackage), которые служат отличной отправной точкой для разработки пакетов Python.

8.5.2 Установка пакета

Теперь, когда у нас есть базовый пакет, мы можем установить `airflow_movielens` в окружении Python. Можно попробовать сделать это, выполнив команду pip, чтобы установить пакет в активном окружении:

```
$ python -m pip install ./airflow-movielens
Looking in indexes: https://pypi.org/simple
Processing ./airflow-movielens
Collecting apache-airflow
...
Successfully installed ... airflow-movielens-0.1.0 ...
```

После того как установка пакета и зависимостей будет завершена, можно проверить, был ли он установлен, путем запуска Python и попытки импортировать один из классов из пакета:

```
$ python
Python 3.7.3 | packaged by conda-forge | (default, Jul 1 2019, 14:38:56)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from airflow_movielens.hooks import MovielensHook
>>> MovielensHook
<class 'airflow_movielens.hooks.MovielensHook'>
```

Развертывание пакета в окружении Airflow не потребует гораздо больших усилий, чем его установка в окружении Python в Airflow. Однако в зависимости от настройки нужно убедиться, что ваш пакет и все его зависимости установлены во всех окружениях, используемых Airflow (имеются в виду планировщик, веб-сервер и воркеры).

Распространение пакета может осуществляться либо путем установки непосредственно из репозитория GitHub:

```
$ python -m pip install git+https://github.com/...
```

либо можно использовать PyPI (или частный фид):

```
$ python -m pip install airflow_movielens
```

либо путем установки из файлового расположения (как мы изначально сделали здесь). В последнем случае нужно убедиться, что окружение Airflow может получить доступ к каталогу, из которого вы хотите установить пакет.

Резюме

- Можно расширить встроенную функциональность Airflow, создавая собственные компоненты, которые подходят для ваших конкретных случаев использования. По нашему опыту, два варианта использования, в которых пользовательские операторы являются особенно мощными, – это:
  - запуск задач в системах, которые изначально не поддерживаются Airflow (например, новые облачные сервисы, базы данных и т. д.);
  - предоставление операторов, сенсоров или хуков для часто выполняемых операций, чтобы члены вашей команды могли с легкостью реализовать их в OAГ.
- Конечно, это далеко не полный список, и может возникнуть много других ситуаций, в которых вам понадобится создать собственные компоненты.
- Пользовательские хуки позволяют взаимодействовать с системами, для которых нет встроенной поддержки в Airflow.
- Пользовательские операторы можно создавать для выполнения задач, конкретно для ваших рабочих процессов, для которых недостаточно встроенных операторов.

- Пользовательские сенсоры позволяют создавать компоненты для ожидания (внешних) событий.
- Код, содержащий пользовательские операторы, хуки, сенсоры и т. д., можно структурировать, если реализовать их в (распространяемой) библиотеке Python.
- Пользовательские хуки, операторы и сенсоры требуют, чтобы их устанавливали с зависимостями на кластере Airflow, прежде чем их можно будет использовать. Это может быть непросто, если у вас нет полномочий на установку программного обеспечения в кластере или есть программное обеспечение с конфликтующими зависимостями.
- Некоторые предпочитают использовать универсальные операторы, такие как встроенный `DockerOperator` и `KubernetesPodOperator` для выполнения своих задач. Преимущество такого подхода заключается в том, что вы можете сохранить свою установку Airflow компактной, поскольку Airflow занимается только координацией контейнерных заданий; вы можете хранить все зависимости конкретных задач, используя контейнер. Мы остановимся на этом подходе в следующих главах.

9 Тестирование

Эта глава рассказывает:

- о тестировании задач Airflow в конвейере непрерывной интеграции и доставки;
- о структурировании проекта для тестирования с pytest;
- об имитации запуска ОАГ для тестирования задач, применяющих создание шаблонов;
- об имитации событий внешней системы;
- о тестировании поведения во внешних системах с помощью контейнеров.

Во всех предыдущих главах мы подробно рассматривали различные фрагменты разработки Airflow. Как же убедиться, что написанный вами код является валидным, прежде чем использовать его при развертывании в рабочей системе? Тестирование является неотъемлемой частью разработки программного обеспечения, и никому не хочется писать код, проводить его через процесс развертывания и скрещивать пальцы, чтобы все было в порядке. Такой вариант разработки явно неэффективен и не дает никаких гарантий правильного функционирования программного обеспечения, будь то допустимая ситуация или нет.

В этой главе мы обратимся к тестированию методом серого ящика, которое часто считается сложной задачей. Это связано с тем, что Airflow взаимодействует со множеством внешних систем, а также потому, что это система оркестровки, которая запускает и останавливает задачи, выполняющие логику, в то время как сам Airflow (часто) никакой логики не выполняет.

9.1 Приступаем к тестированию

Тесты могут применяться на разных уровнях. Небольшие отдельные единицы работы (например, отдельные функции) можно протестировать с помощью модульных тестов. Хотя такие тесты могут подтверждать правильное поведение, они не подтверждают поведение системы, состоящей из нескольких таких единиц в целом. Для этого пишутся интеграционные тесты, которые проверяют, как ведут себя несколько компонентов вместе. В книгах по тестированию следующим используемым уровнем тестирования является приемочное тестирование (оценка соответствия требованиям спецификации или контракта), что не относится к данной главе. Здесь мы будем рассматривать модульное и интеграционное тестирования.

В этой главе мы демонстрируем различные фрагменты кода, написанные с помощью pytest (https://docs.pytest.org/en/6.2.x/). В то время как в Python имеется встроенный фреймворк для тестирования под названием unittest, pytest представляет собой один из самых популярных сторонних фреймворков для тестирования различных функций, таких как фикстуры, которыми мы воспользуемся в этой главе. Никаких предварительных знаний pytest не требуется.

Поскольку код из этой книги есть на GitHub, мы продемонстрируем конвейер непрерывной интеграции и доставки, запуская тесты с помощью GitHub Actions (https://github.com/features/actions), системы непрерывной интеграции и доставки, которая интегрируется с GitHub. Используя идеи и код из примеров GitHub Actions, вы сможете запускать этот конвейер в любой системе. Все популярные системы непрерывной интеграции и доставки, такие как GitLab, Bitbucket, CircleCI, Travis CI и т. д., работают, определяя конвейер в формате YAML в корне каталога проекта. Мы будем делать так же.

9.1.1 Тест на благонадежность ОАГ

В контексте Airflow первым шагом в тестировании обычно является *тест на благонадежность ОАГ*, термин, который стал известен из поста в блоге под названием *Data's Inferno: 7 Circles of Data Testing Hell with Airflow* (https://medium.com/wbaa/datas-inferno-7-circles-of-data-testing-hell-with-airflow-cef4adff58d8). Такой тест проверяет *благонадежность* всех ваших ОАГ (т. е. их правильность, например не содер-

жат ли они циклов; уникальны ли идентификаторы задач и т. д.). Тест на благонадежность ОАГ часто фильтрует простые ошибки. Например, нередко совершается ошибка при генерации задач в цикле for с фиксированным идентификатором задачи вместо динамически устанавливаемого идентификатора, в результате чего каждая сгенерированная задача имеет один и тот же идентификатор. При загрузке ОАГ Airflow также выполняет такие проверки самостоятельно и при обнаружении отображает ошибку (рис. 9.1). Чтобы избежать прохождения цикла развертывания, обнаружив в конечном итоге, что ваш ОАГ содержит простую ошибку, разумно выполнять такие тесты в своем наборе тестов.

Рис. 9.1 Ошибка цикла ОАГ, отображаемая Airflow

Следующий ОАГ из листинга 9.1 будет отображать ошибку в пользовательском интерфейсе, потому что здесь есть цикл: `t1 >> t2 >> t3 >> t1`. Это нарушает свойство, согласно которому ОАГ должен иметь определенные начальные и конечные узлы.

Листинг 9.1 Пример цикла в ОАГ, приводящий к ошибке

```
t1 = DummyOperator(task_id="t1", dag=dag)
t2 = DummyOperator(task_id="t2", dag=dag)
t3 = DummyOperator(task_id="t3", dag=dag)

t1 >> t2 >> t3 >> t1
```

Теперь перехватим эту ошибку, используя тест на благонадежность. Для начала установим pytest:

Листинг 9.2 Установка pytest

```
pip install pytest

Collecting pytest
.................
Installing collected packages: pytest
Successfully installed pytest-5.2.2
```

Так мы получаем CLI-утилиту pytest. Чтобы увидеть все доступные параметры, выполните `pytest --help`. Пока не нужно знать все параметры; достаточно знать, что вы можете запускать тесты с помощью `pytest [file/directory]` (где каталог содержит тестовые файлы). Давайте создадим такой файл. По соглашению в корне проекта создается папка tests/, которая содержит все тесты и зеркально отображает

ту же структуру каталогов, что и в остальной части проекта[1]. Таким образом, если структура вашего проекта подобна той, что показана на рис. 9.2:

```
.
├── dags
│   ├── dag1.py
│   ├── dag2.py
│   └── dag3.py
└── mypackage
    ├── airflow
    │   ├── hooks
    │   │   ├── __init__.py
    │   │   └── movielens_hook.py
    │   ├── operators
    │   │   ├── __init__.py
    │   │   └── movielens_operator.py
    │   └── sensors
    │       ├── __init__.py
    │       └── movielens_sensors.py
    └── movielens
        ├── __init__.py
        └── utils.py
```

Рис. 9.2 Пример структуры пакета Python

тогда структура каталога tests/ будет выглядеть так, как показано на рис. 9.3:

```
.
├── dags
├── mypackage
└── tests
    ├── dags
    │   └── test_dag_integrity.py
    └── mypackage
        ├── airflow
        │   ├── hooks
        │   │   └── test_movielens_hook.py
        │   ├── operators
        │   │   └── test_movielens_operator.py
        │   └── sensors
        │       └── test_movielens_sensor.py
        └── movielens
            └── test_utils.py
```

Рис. 9.3 Структура каталога tests/ следует структуре, показанной на рис. 9.2

[1] Pytest называет такую структуру «Тесты вне кода приложения». Другая поддерживаемая pytest структура – хранение тестовых файлов непосредственно рядом с кодом вашего приложения, который он называет «тесты как часть кода вашего приложения».

Обратите внимание, что все файлы зеркально отражают имена файлов, которые (предположительно) тестируются, с префиксом `test_`. Опять же, хотя зеркальное отображение имени файла для тестирования не требуется, это очевидное соглашение, чтобы сообщить о содержимом файла. Тесты, которые перекрывают несколько файлов или предоставляют другие виды тестов (например, тест на благонадежность ОАГ), обычно помещаются в файлы, названные в соответствии с тем, что они тестируют. Однако здесь требуется префикс `test_`; pytest просматривает заданные каталоги и ищет файлы с префиксом `test_` или суффиксом `_test`[1]. Также обратите внимание, что в каталоге tests/ нет файлов `__init__.py`; каталоги не являются модулями, и тесты должны иметь возможность работать независимо друг от друга, не импортируя друг друга. Pytest сканирует каталоги и файлы и автоматически находит тесты; не нужно создавать модули с файлами `__init__.py`.

Создадим файл с именем tests/dags/test_dag_integrity.py.

Листинг 9.3 Тест на благонадежность ОАГ

```
import glob
import importlib.util
import os

import pytest
from airflow.models import DAG

DAG_PATH = os.path.join(
   os.path.dirname(__file__), "..", "..", "dags/**/*.py"
)
DAG_FILES = glob.glob(DAG_PATH, recursive=True)
@pytest.mark.parametrize("dag_file", DAG_FILES)
def test_dag_integrity(dag_file):
   module_name, _ = os.path.splitext(dag_file)
   module_path = os.path.join(DAG_PATH, dag_file)
   ➥ mod_spec = importlib.util.spec_from_file_location(module_name,
     module_path)
   module = importlib.util.module_from_spec(mod_spec)
   mod_spec.loader.exec_module(module)

   ➥ dag_objects = [var for var in vars(module).values() if isinstance(var,
     DAG)]

   assert dag_objects

   for dag in dag_objects:
       dag.test_cycle()
```

Здесь мы видим одну функцию с именем `test_dag_integrity`, которая выполняет тест. Код на первый взгляд может показаться немного

[1] В pytest параметры обнаружения тестов можно настроить, например, для поддержки тестовых файлов с именем `check_ *`.

неясным, поэтому давайте разберем его. Помните структуру папок, описанную ранее? Здесь есть папка dags/, в которой хранятся все файлы ОАГ, и файл test_dag_integrity.py, который находится в каталоге tests/dags/test_dag_integrity.py. На рис. 9.4 показан каталог, содержащий все файлы ОАГ.

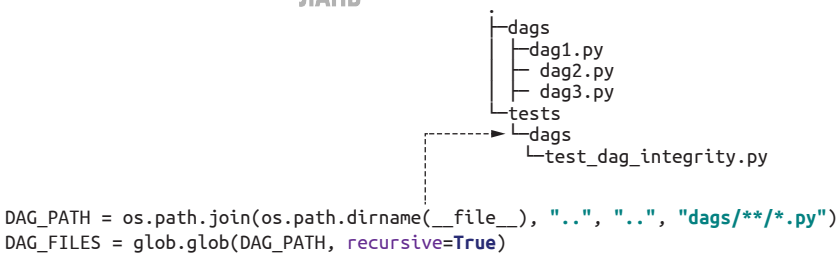


Рис. 9.4 Каталог, содержащий все файлы ОАГ

Метод `dirname()` возвращает каталог `test_dag_integrity.py`, а затем мы просматриваем два каталога вверх, сначала до tests/, потом до корня, а оттуда ищем все, что соответствует шаблону `dags/**/*.py`. Синтаксис `"**"` указывает на то, что поиск будет вестись рекурсивно, поэтому файлы ОАГ, например, в dags/dir1/dir2/dir3/mydag.py также будут найдены. Наконец, переменная `DAG_FILES` содержит список файлов, находящихся в папке dags/ с расширением `.py`. Далее декоратор `@pytest.mark.parametrize` запускает тест для каждого найденного файла Python (рис. 9.5).

Рис. 9.5 Декоратор запускает тест для каждого файла `dag_file`

Первая часть теста немного непонятна. Мы не будем вдаваться в подробности, но все сводится к загрузке и выполнению файла, как это сделал бы сам Python, и извлечению из него объектов ОАГ.

Листинг 9.4 Попытка создать экземпляр каждого найденного объекта ОАГ

Загружаем файл

```
modul_name, _ = os.path.splitext(dag_file)
module_path = os.path.join(DAG_PATH, dag_file)
mod_spec = importlib.util.spec_from_file_location(module_name, module_path)
module = importlib.util.module_from_spec(mod_spec)
mod_spec.loader.exec_module(module)

dag_objects = [var for var in vars(module).values() if isinstance(var, DAG)]
```

Все объекты класса DAG, находящиеся в файле

Теперь, когда объекты ОАГ извлечены из файла, можно применить определенные проверки. В коде мы применили две проверки. Сначала было утверждение: `assert dag_objects`. Так мы проверили, находятся ли объекты ОАГ в файле (все прошло успешно). Добавляя это утверждение, мы проверяем все файлы Python, найденные в папке /dags, если они содержат хотя бы один объект ОАГ. Например, функции Scripts, хранящиеся в этой папке, в которых не создаются экземпляры объектов ОАГ, не сработают. Желательно это или нет, решать вам, но наличие одного каталога, который не содержит ничего, кроме файлов ОАГ, не обеспечивает четкого разделения обязанностей.

Вторая проверка (`for dag in dag_objects: dag.test_cycle()`) выясняет, нет ли циклов в объектах ОАГ. Она вызывается явно, и на то есть причина. До Airflow версии 1.10.0 ОАГ проверялись на предмет наличия циклов при каждом изменении их структуры. Эта проверка становится сложнее с точки зрения вычислений по мере добавления все большего количества задач. Для ОАГ с большим количеством задач это стало обузой, потому что для каждой новой задачи выполнялась проверка цикла, что приводило к долгому чтению. Поэтому проверка цикла перенесена в точку, где Airflow осуществляет анализ и кеширование ОАГ (в структуру под названием DagBag), и проверка цикла выполняется только один раз после анализа всего ОАГ, что сокращает время чтения. В результате можно вполне объявить `t1 >> t2 >> t1` и выполнить вычисление. Только когда работающий в реальном времени экземпляр Airflow прочитает ваш сценарий, он будет жаловаться на цикл. Таким образом, чтобы избежать цикла развертывания, мы явно вызываем метод `test_cycle()` для каждого ОАГ, найденного в тесте.

Это два примера проверки, но вы, конечно же, можете добавить свою. Если, скажем, вы хотите, чтобы каждое имя ОАГ начиналось со слов «import» или «export», то можно использовать следующий код для проверки идентификаторов `dag_id`:

```
assert dag.dag_id.startswith(("import", "export"))
```

Теперь проведем тест на благонадежность. В командной строке запустите pytest (можно намекнуть pytest, где искать с помощью `pytest tests/`, чтобы избежать сканирования других каталогов).

Листинг 9.5 Вывод запущенного pytest

```
$ pytest tests/
======================== test session starts ========================
....
collected 1 item

tests/dags/test_dag_integrity.py F
[100%]

============================ FAILURES ============================
```

```
_____________ test_dag_integrity[..../dag_cycle.py] _____________

dag_file = '..../dag_cycle.py'

    @pytest.mark.parametrize("dag_file", DAG_FILES)
    def test_dag_integrity(dag_file):
        "" Импорт файлов ОАГ и проверка на наличие ОАГ.""
        module_name, _ = os.path.splitext(dag_file)
        module_path = os.path.join(DAG_PATH, dag_file)
        ➥ mod_spec = importlib.util.spec_from_file_location(module_name,
     module_path)
        module = importlib.util.module_from_spec(mod_spec)
        mod_spec.loader.exec_module(module)

        ➥ dag_objects = [
            var for var in vars(module).values() if isinstance(var, DAG)
        ]

        assert dag_objects

        for dag in dag_objects:
            # Test cycles
>           dag.test_cycle()

tests/dags/test_dag_integrity.py:29:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
.../site-packages/airflow/models/dag.py:1427: in test_cycle
    self._test_cycle_helper(visit_map, task_id)
.../site-packages/airflow/models/dag.py:1449: in _test_cycle_helper
    self._test_cycle_helper(visit_map, descendant_id)
.../site-packages/airflow/models/dag.py:1449: in _test_cycle_helper
    self._test_cycle_helper(visit_map, descendant_id)
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
➥ self = <DAG: chapter8_dag_cycle>, visit_map = defaultdict(<class 'int'>,
     {'t1': 1, 't2': 1, 't3': 1}), task_id = 't3'

    def _test_cycle_helper(self, visit_map, task_id):
        ""
        Проверяет, существует ли цикл из входной задачи, используя поиск
        в глубину
        ...

        task = self.task_dict[task_id]
        for descendant_id in task.get_direct_relative_
              if visit_map[descendant_id] == DagBag.CYCLE_IN_PROGRESS:
                  ➥ msg = "Cycle detected in DAG. Faulty task: {0} to
        {1}".format(
                      task_id, descendant_id)
>                 raise AirflowDagCycleException(msg)
➥ E                   airflow.exceptions.AirflowDagCycleException: Cycle
     detected in DAG. Faulty task: t3 to t1

..../airflow/models/dag.py:1447: AirflowDagCycleException
========================== 1 failed in 0.21s ==========================
```

Результат теста довольно длинный, но обычно ответы можно найти в верхней части и внизу. Вверху вы найдете, какой тест не прошел, а внизу ответы на вопрос, почему это произошло.

Листинг 9.6 Причина исключения, найденная в листинге 9.5

```
airflow.exceptions.AirflowDagCycleException: Cycle detected in DAG.
Faulty task: t3 to t1
```

Этот пример показывает нам (как и ожидалось), что от t3 до t1 был обнаружен цикл. После создания экземпляров ОАГ и операторов сразу же выполняется несколько других проверок из коробки. Допустим, вы используете `BashOperator`, но забыли добавить (обязательный) аргумент `bash_command`. Тест на благонадежность проверит все операторы в сценарии и завершится ошибкой при вычислении `BashOperator`.

Листинг 9.7 Неправильное создание экземпляра BashOperator

```
BashOperator(task_id="this_should_fail", dag=dag)
```

Тест на благонадежность обнаружит исключение и завершится ошибкой.

Листинг 9.8 Исключение, возбуждаемое при неправильном создании экземпляра из листинга 9.7

```
airflow.exceptions.AirflowException: Argument ['bash_command'] is required
```

Теперь, когда у нас есть тест на благонадежность, давайте запустим его автоматически в конвейере непрерывной интеграции и доставки.

9.1.2 Настройка конвейера непрерывной интеграции и доставки

Если выразить все одной строкой, то конвейер непрерывной интеграции и доставки – это система, которая запускает предопределенные сценарии, когда вы вносите изменения в репозиторий кода. *Непрерывное интегрирование* означает проверку и валидацию измененного кода, чтобы убедиться, что он соответствует стандартам кодирования и набору тестов. Например, при отправке изменений в репозиторий вы можете проверить наличие Flake8 (https://flake8.pycqa.org/en/latest/), Pylint (https://www.pylint.org) и Black (https://github.com/psf/black) и запустить серию тестов. *Непрерывное развертывание* указывает на автоматическое развертывание кода в рабочих системах без вмешательства человека. Цель состоит в том, чтобы максимизировать продуктивность написания кода, не имея дело с ручной проверкой и развертыванием.

Существует широкий спектр систем непрерывной интеграции и доставки. В этой главе мы рассмотрим GitHub Actions (https://github.com/features/actions); в целом он должен быть применим к любой системе непрерывной интеграции и развертывания. Большинство таких систем начинаются с файла конфигурации YAML, в котором определяется конвейер: последовательность шагов, выполняемых при изменении кода. Каждый шаг должен завершиться успешно, чтобы конвейер был успешным. В репозитории Git можно затем применить такие правила, как «слияние ветки в master только при наличии успешного завершения конвейера».

Определения конвейера обычно находятся в корне проекта; GitHub Actions требуются файлы YAML, хранящиеся в каталоге: .github/workflows. Имя файла YAML не имеет значения, поэтому можно создать файл с именем airflow-tests.yaml следующего содержания:

Листинг 9.9 Пример конвейера GitHub Actions для проекта Airflow

```
name: Python static checks and tests

on: [push]

jobs:
 testing:
   runs-on: ubuntu-18.04
   steps:
     - uses: actions/checkout@v1
     - name: Setup Python
       uses: actions/setup-python@v1
       with:
         python-version: 3.6.9
         architecture: x64

     - name: Install Flake8
       run: pip install flake8
     - name: Run Flake8
       run: flake8

     - name: Install Pylint
       run: pip install pylint
     - name: Run Pylint
       run: find . -name "*.py" | xargs pylint --output-format=colorized

     - name: Install Black
       run: pip install black
     - name: Run Black
       run: find . -name "*.py" | xargs black -check

     - name: Install dependencies
       run: pip install apache-airflow pytest

   - name: Test DAG integrity
     run: pytest tests/
```

Ключевые слова, показанные в этом YAML-файле, уникальны для GitHub Actions, хотя общие идеи применимы и к другим системам непрерывной интеграции и доставки. Важно отметить задачи в GitHub Actions, определенные в разделе «steps». На каждом этапе выполняется фрагмент кода. Например, Flake8 выполняет статический анализ кода и даст сбой, если возникнут какие-либо проблемы, например неиспользованный импорт. В строке 3 мы указываем `on: [push]`, сообщая GitHub запускать полный конвейер каждый раз при отправке изменений в репозиторий. В полностью автоматизированной системе непрерывной доставки будут содержаться фильтры для шагов в определенных ветвях, таких как master, чтобы выполнять шаги и развертывать код только в том случае, если конвейер успешно работает в этой ветви.

9.1.3 Пишем модульные тесты

Теперь, когда у нас есть запущенный и работающий конвейер, который изначально проверяет валидность всех ОАГ в проекте, пора немного глубже погрузиться в код Airflow и приступить к модульному тестированию отдельных фрагментов.

Посмотрим на пользовательские компоненты, продемонстрированные в главе 8; есть несколько вещей, которые можно протестировать, чтобы проверить правильность поведения. Поговорка гласит: «Никогда не доверяйте пользовательскому вводу», поэтому мы хотим быть уверены, что наш код работает правильно, будь то допустимая ситуация или нет. Возьмем, к примеру, `MovielensHook` из главы 8, который содержит метод `get_ratings()`. Этот метод принимает несколько аргументов; один из них – `batch_size`, который контролирует размер пакетов, запрашиваемых из API. Можно представить, что допустимым вводом будет любое положительное число (возможно, с некоторым верхним пределом). Но что, если пользователь вводит отрицательное число (например, –3)? Возможно, API правильно обработает недопустимый размер пакета и вернет код состояния HTTP, например 400 или 422, но как на это отреагирует `MovielensHook`? Возможно, разумнее было бы обработать входное значение еще до отправки запроса или правильно обработать ошибки, если API возвращает ошибку. Это поведение мы и хотим проверить.

Продолжим то, что мы делали в главе 8, и реализуем `MovielensPopularityOperator` – оператор, возвращающий *N*-е количество самых популярных фильмов между двумя заданными датами.

Листинг 9.10 Пример оператора MovielensPopularityOperator

```
class MovielensPopularityOperator(BaseOperator):
    def __init__(
        self,
        conn_id,
```

```
        start_date,
        end_date,
        min_ratings=4,
        top_n=5,
        **kwargs,
    ):
        super().__init__(**kwargs)
        self._conn_id = conn_id
        self._start_date = start_date
        self._end_date = end_date
        self._min_ratings = min_ratings
        self._top_n = top_n
    def execute(self, context):
        with MovielensHook(self._conn_id) as hook:
            ratings = hook.get_ratings(
                start_date=self._start_date,
                end_date=self._end_date,
            )

            rating_sums = defaultdict(Counter)
            for rating in ratings:
                ➥ rating_sums[rating["movieId"]].update(count=1,
        rating=rating["rating"])

            averages = {
                ➥ movie_id: (rating_counter["rating"] /
        rating_counter["count"], rating_counter["count"])
                for movie_id, rating_counter in rating_sums.items()
                if rating_counter["count"] >= self._min_ratings
            }
            ➥ return sorted(averages.items(), key=lambda x: x[1],
        reverse=True)[: self._top_n]
```

Получаем необработанные рейтинги

Суммируем рейтинги по идентификатору фильма

Фильтруем min_ratings и рассчитываем средний рейтинг на каждый идентификатор фильма

Возвращаем результат, отсортированный по среднему рейтингу и количеству рейтингов

Как проверить правильность этого оператора? Для начала можно было бы протестировать его целиком, просто запустив оператор с заданными значениями и проверив результат, как и ожидалось. Нам потребуется пара компонентов pytest, чтобы запустить оператор отдельно, за пределами действующей системы Airflow и внутри модульного теста. Это позволяет нам запускать его в разных условиях и проверять, правильно ли он ведет себя.

9.1.4 Структура проекта Pytest

При использовании pytest перед тестовым сценарием должен стоять префикс `test_`. Так же, как и в структуре каталогов, мы имитируем имена файлов, поэтому тест кода из файла `movielens_operator.py` будет храниться в файле `test_movielens_operator.py`. Внутри этого файла мы создадим функцию, которая будет вызываться в качестве теста.

Листинг 9.11 Пример функции, тестирующей BashOperator

```
def test_example():
    task = BashOperator(
        task_id="test",
        bash_command="echo 'hello!'",
        xcom_push=True,
    )
    result = task.execute(context={})
    assert result == "hello!"
```

В этом примере мы создаем экземпляр `BashOperator` и вызываем функцию `execute()`, учитывая пустой контекст (пустой словарь). Когда Airflow запускает оператор в реальной ситуации, до и после этого происходит ряд вещей, например визуализация шаблонных переменных, настройка контекста экземпляра задачи и предоставление его оператору. В этом тесте мы не работаем в реальной ситуации, а вызываем метод `execute()` напрямую. Это функция самого низкого уровня, которую можно вызвать для запуска оператора, и это метод, который каждый оператор реализует для выполнения своих функций. Нам не нужен контекст экземпляра задачи для запуска `BashOperator`; поэтому мы предоставляем ему пустой контекст. Если тест зависит от обработки чего-либо из контекста экземпляра задачи, то можно заполнить его необходимыми ключами и значениями[1].

Выполним этот тест.

Листинг 9.12 Результат выполнения теста из листинга 9.11

```
$ pytest tests/dags/chapter9/custom/test_operators.py::test_example
========================= test session starts ==========================
platform darwin -- Python 3.6.7, pytest-5.2.2, py-1.8.0, pluggy-0.13.0
rootdir: .../data-pipelines-with-apache-airflow
collected 1 item

tests/dags/chapter9/custom/test_operators.py .
```

Теперь применим это к `MovielensPopularityOperator`.

Листинг 9.13 Пример функции, тестирующей MovielensPopularityOperator

```
def test_movielenspopularityoperator():
    task = MovielensPopularityOperator(
        task_id="test_id",
        start_date="2015-01-01",
        end_date="2015-01-03",
```

[1] Аргумент `xcom_push=True` возвращает стандартный вывод в `Bash_command` в виде строки, которую мы используем в этом тесте для выборки и проверки `Bash_command`. В реальной ситуации Airflow любой объект, возвращаемый оператором, автоматически отправляется в XCom.

```
        top_n=5,
    )
    result = task.execute(context={})
    assert len(result) == 5
```

Первое, что мы видим, – это текст, выделенный красным, который сообщает нам, что у оператора отсутствует обязательный аргумент.

Листинг 9.14 Результат выполнения теста из листинга 9.13

```
➥ $ pytest tests/dags/chapter9/custom/test_operators.py::test
    _movielenspopularityoperator
========================== test session starts ==========================
platform darwin -- Python 3.6.7, pytest-5.2.2, py-1.8.0, pluggy-0.13.0
rootdir: /.../data-pipelines-with-apache-airflow
collected 1 item

tests/dags/chapter9/custom/test_operators.py F
[100%]
============================== FAILURES ==============================
____________________ test_movielenspopularityoperator ____________________

mocker = <pytest_mock.plugin.MockFixture object at 0x10fb2ea90>

    def test_movielenspopularityoperator(mocker: MockFixture):
        task = MovielensPopularityOperator(
➥ >           task_id="test_id", start_date="2015-01-01", end_date="2015-01-
    03", top_n=5
        )
➥ E       TypeError: __init__() missing 1 required positional argument:
    'conn_id'

tests/dags/chapter9/custom/test_operators.py:30: TypeError
========================== 1 failed in 0.10s ==========================
```

Сейчас мы видим, что тест провалился из-за отсутствия обязательного аргумента `conn_id`, который указывает на идентификатор подключения в базе метаданных. Но как указать его в тесте? Тесты должны быть изолированы друг от друга; у них не должно быть возможности влиять на результаты других тестов, поэтому база данных, совместно используемая в тестах, не является идеальной ситуацией. В этом случае на помощь приходит *мокирование*.

Мокирование – это «имитация» определенных операций или объектов. Например, вызов базы данных, которая, как ожидается, будет существовать в промышленном окружении, но не во время тестирования, можно «*сымитировать*», сообщая Python, чтобы он возвращал определенное значение, вместо того чтобы делать фактический вызов базы данных (отсутствующей во время тестирования). Это позволяет разрабатывать и запускать тесты, не требуя подключения к внешним системам, но требует понимания внутреннего устройства того, что вы тестируете, и поэтому иногда для этого нужно погрузиться в сторонний код.

В Pytest имеется набор вспомогательных плагинов (официально они не от pytest), которые упрощают использование таких понятий, как имитация. Для этого можно установить пакет `pytest-mock`:

```
pip install pytest-mock
```

`pytest-mock` – это пакет Python, предоставляющий небольшую удобную обертку для встроенного пакета имитаций. Чтобы использовать его, передайте вашей функции аргумент «mocker»[1], который является точкой входа для использования чего-либо в пакете `pytest-mock`.

Листинг 9.15 Имитация объекта в тесте

```
def test_movielenspopularityoperator(mocker):
    mocker.patch.object(
        MovielensHook,
        "get_connection",
        return_value=Connection(
            conn_id="test",
            login="airflow",
                password="airflow",
            ),
        )
        task = MovielensPopularityOperator(
            task_id="test_id",
            conn_id="test",
            start_date="2015-01-01",
            end_date="2015-01-03",
            top_n=5,
        )
        result = task.execute(context=None)
        assert len(result) == 5
```

С помощью этого кода к вызову метода `get_connection()` в `MovielensHook` применяется «обезьяний патч» (заменяя его функциональность во время выполнения, чтобы вернуть данный объект вместо запроса в базу метаданных Airflow), а выполнение `MovielensHook.get_connection()` не завершится ошибкой при запуске теста, поскольку во время тестирования не производится никаких вызовов несуществующей базы данных. Вместо этого возвращается предопределенный ожидаемый объект `Connection`.

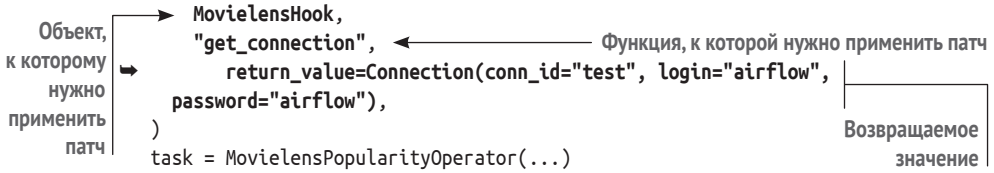
Листинг 9.16 Замена вызова внешней системы в тесте

```
def test_movielenspopularityoperator(mocker):
    mock_get = mocker.patch.object(
```

Объект mocker волшебным образом существует во время выполнения; импорт не требуется

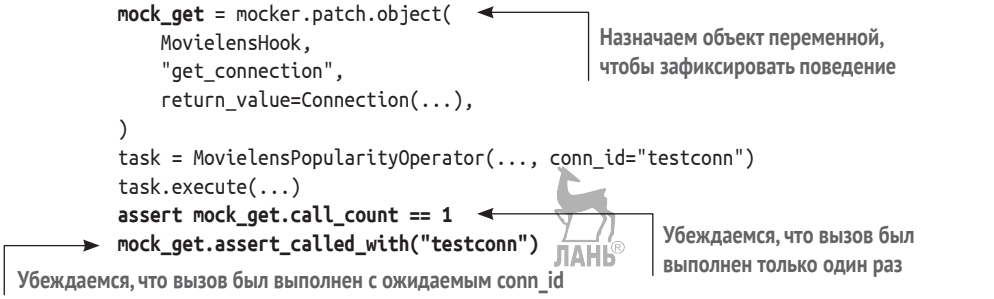
Применяем патч, используя имитационный объект

[1] Если вы хотите типизировать свои аргументы, то `mocker` имеет тип `pytest_mock.MockFixture`.



В этом примере показано, как заменить вызов внешней системы (база метаданных Airflow) во время тестирования, возвращая предопределенный объект `Connection`. Что, если вам нужно подтвердить, что вызов действительно сделан в вашем тесте? Можно назначить объект с патчем переменной, которая содержит ряд свойств, собранных при вызове объекта. Например, нам нужно убедиться, что метод `get_connection()` вызывается только один раз, а аргумент `conn_id`, предоставленный `get_connection()`, имеет то же значение, что и в случае с `MovielensPopularityOperator`.

Листинг 9.17 Проверка поведения имитационной функции



Назначая возвращаемое значение `mocker.patch.object` переменной `mock_get`, мы перехватываем все вызовы, осуществляемые к имитационному объекту, что дает нам возможность проверить данный ввод, количество вызовов и многое другое. В этом примере мы утверждаем, может ли `call_count` проверить, что `MovielensPopularityOperator` случайно не выполняет несколько вызовов к базе метаданных Airflow в режиме реального времени. Кроме того, поскольку мы предоставляем `MovielensPopularityOperator` идентификатор «testconn», мы ожидаем, что этот идентификатор будет запрошен из базы метаданных Airflow, а для валидации используем метод `assert_called_with()`[1]. Объект `mock_get` содержит больше свойств для проверки (например, вызываемое свойство, чтобы просто подтвердить, был ли объект вызван [любое количество раз]) (рис. 9.6).

Одна из самых больших ошибок при использовании мокирования в Python – это имитирование неправильного объекта. В этом примере мы имитируем метод `get_connection()`. Он вызывается в классе

[1] Для двух этих утверждений существует удобный метод `assert_called_once_with()`.

MovielensHook, который наследует от класса BaseHook (пакет airflow.hooks.base). Метод get_connection() определяется в классе BaseHook. Поэтому интуитивно вы, вероятно, «сымитировали» бы BaseHook.get_connection(), а это неправильно.

Правильный способ – это имитировать место вызова, а не место определения[1]. Проиллюстрируем это с помощью кода.

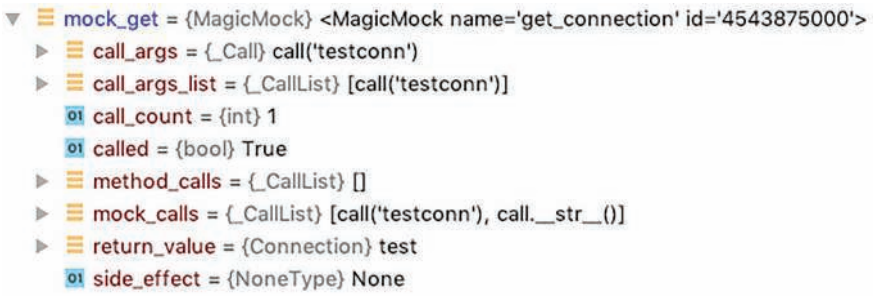


Рис. 9.6 Объект mock_get содержит несколько свойств, которые можно использовать для проверки поведения (этот скриншот был сделан с помощью отладчика Python в PyCharm)

Листинг 9.18 Обращаем внимание на правильное место импорта при имитировании

```
from airflowbook.operators.movielens_operator import (
    MovielensPopularityOperator,
    MovielensHook,
)


def test_movielenspopularityoperator(mocker):
    mock_get = mocker.patch.object(
        MovielensHook,
        "get_connection",
        return_value=Connection(...),
    )
    task = MovielensPopularityOperator(...)
```

Мы должны импортировать метод, который нужно сымитировать, из того места, откуда он был вызван

Внутри кода MovielensPopularityOperator вызывается MovielensHook.get_connection()

9.1.5 Тестирование с файлами на диске

Рассмотрим оператор, который читает один файл, содержащий список в формате JSON, и записывает все это в формат CSV (рис. 9.7).

[1] Это объяснено в документации Python: https://docs.python.org/3/library/unittest.mock.html#whereto-patch, а также продемонстрировано на странице https://alexmarandon.com/articles/python_mock_gotchas/.

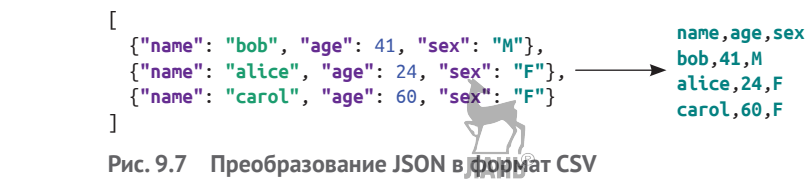


Рис. 9.7 Преобразование JSON в формат CSV

Оператор для данного действия может выглядеть следующим образом:

Листинг 9.19 Пример оператора с использованием локального диска

```
class JsonToCsvOperator(BaseOperator):
    def __init__(self, input_path, output_path, **kwargs):
        super().__init__(**kwargs)
        self._input_path = input_path
        self._output_path = output_path

    def execute(self, context):
        with open(self._input_path, "r") as json_file:
            data = json.load(json_file)

        columns = {key for row in data for key in row.keys()}

        with open(self._output_path, mode="w") as csv_file:
            writer = csv.DictWriter(csv_file, fieldnames=columns)
            writer.writeheader()
            writer.writerows(data)
```

Оператор `JsonToCsvOperator` принимает два входных аргумента: входной путь (JSON) и выходной путь (CSV). Чтобы протестировать этот оператор, мы могли бы сохранить статический файл в нашем тестовом каталоге, дабы использовать его в качестве входных данных для теста, но где хранить выходной файл?

В Python есть модуль tempfile для задач, связанных с временным хранилищем. Он не оставляет остатков в файловой системе, поскольку каталог и его содержимое стираются после использования. pytest предоставляет удобную точку доступа к этому модулю, `tmp_dir` (дает объект `os.path`) и `tmp_path` (дает объект `pathlib`). Рассмотрим пример с применением `tmp_path`.

Листинг 9.20 Тестирование с использованием временных путей

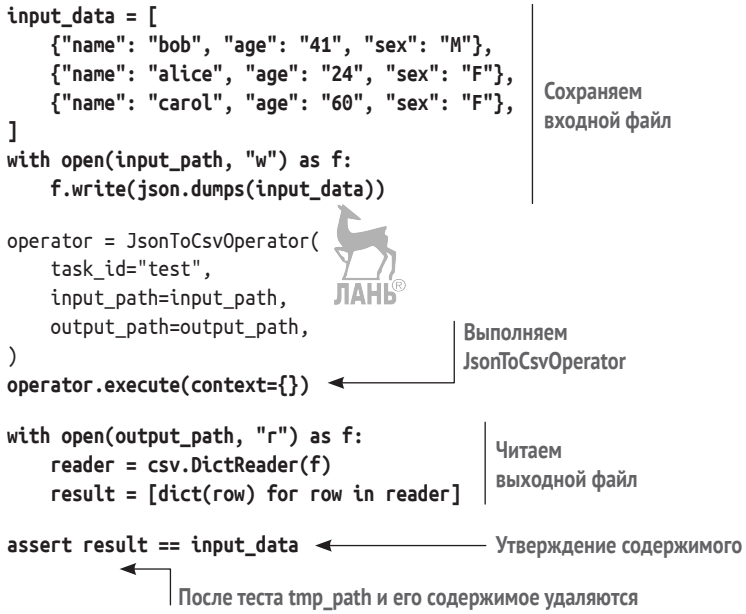
```
import csv
import json
from pathlib import Path

from airflowbook.operators.json_to_csv_operator import JsonToCsvOperator


def test_json_to_csv_operator(tmp_path: Path):
    input_path = tmp_path / "input.json"
    output_path = tmp_path / "output.csv"
```

Используем фикстуру tmp_path

Определяем пути



При запуске теста создается временный каталог. Фактически аргумент `tmp_path` относится к функции, которая выполняется для каждого теста, в котором она вызывается. В pytest они называются *фикстурами* (https://docs.pytest.org/en/stable/fixture.html). У фикстур есть некоторое сходство с методами `setUp()` и `tearDown()` фреймворка unittest, и они обеспечивают больше гибкости, потому что их можно смешивать и сопоставлять (например, одна фикстура может инициализировать временный каталог для всех тестов в классе, в то время как другая инициализируется только для одного теста)[1]. По умолчанию область действия фикстур – это каждая тестовая функция. Это можно увидеть, если вывести путь и запустить разные тесты или даже один и тот же тест дважды:

```
print(tmp_path.as_posix())
```

Соответственно, будет выведено следующее:

- /private/var/folders/n3/g5l6d1j10gxfsdkphhgkgn4w0000gn/T/pytest-of-basharenslak/pytest-**19**/test_json_to_csv_operator0;
- /private/var/folders/n3/g5l6d1j10gxfsdkphhgkgn4w0000gn/T/pytest-of-basharenslak/pytest-**20**/test_json_to_csv_operator0.

Существуют и другие фикстуры, которые можно использовать, а у фикстур pytest есть много функций, которые не демонстрируются в этой книге. Если вы серьезно интересуетесь функциями pytest, обратитесь к документации.

[1] Поищите информацию о «pytest scope», если хотите узнать, как использовать фикстуры в тестах.

9.2 Работа с OAG и контекстом задачи в тестах

Некоторым операторам требуется больше контекста (например, шаблонизация переменных) или использование контекста экземпляра задачи для выполнения. Нельзя просто запустить `operator.execute(context={})`, как мы это делали в предыдущих примерах, потому что мы не предоставляем контекст задачи оператору, который ему нужен для выполнения кода.

В этих случаях нам хотелось бы запускать оператор в более реалистичном сценарии, как если бы Airflow запускал задачу в действующей системе и, таким образом, создавал контекст экземпляра задачи, шаблоны для всех переменных и т. д. На рис. 9.8 показаны шаги, которые выполняются при выполнении задачи в Airflow[1].

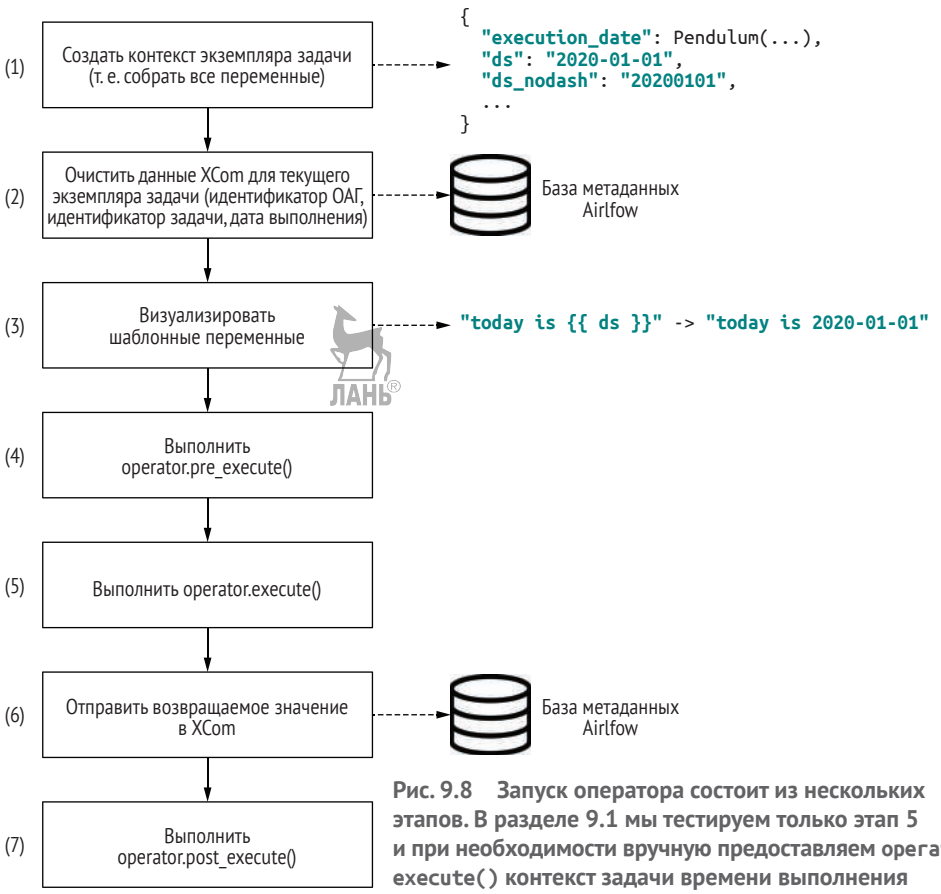


Рис. 9.8 Запуск оператора состоит из нескольких этапов. В разделе 9.1 мы тестируем только этап 5 и при необходимости вручную предоставляем `operator.execute()` контекст задачи времени выполнения

[1] В TaskInstance это `_run_raw_task()`.

Как видите, шаг 5 – единственный, который мы выполняли в примерах до сих пор (листинги 9.15, 9.17 и 9.20). Если вы используете реальную систему Airflow, при выполнении оператора совершается гораздо больше шагов, некоторые из которых нужно реализовать, чтобы протестировать, например, правильно ли осуществляется создание шаблонов.

Допустим, мы реализовали оператор, извлекающий рейтинги фильмов между двумя заданными датами, которые пользователь может предоставить с помощью шаблонных переменных.

Листинг 9.21 Пример оператора с использованием шаблонных переменных

```
class MovielensDownloadOperator(BaseOperator):
    template_fields = ("_start_date", "_end_date", "_output_path")

    def __init__(
        self,
        conn_id,
        start_date,
        end_date,
        output_path,
        **kwargs,
    ):
        super().__init__(**kwargs)
        self._conn_id = conn_id
        self._start_date = start_date
        self._end_date = end_date
        self._output_path = output_path

    def execute(self, context):
        with MovielensHook(self._conn_id) as hook:
            ratings = hook.get_ratings(
                start_date=self._start_date,
                end_date=self._end_date,
            )

        with open(self._output_path, "w") as f:
            f.write(json.dumps(ratings))
```

Этот оператор нельзя протестировать, как в предыдущих примерах, поскольку он (потенциально) требует контекста экземпляра задачи для выполнения. Например, аргумент `output_path` можно предоставить в виде `/output/{{ds}}.json`, а переменная `ds` недоступна при тестировании с `operator.execute(context={})`.

Поэтому для этого мы вызовем фактический метод, который сам Airflow также использует для запуска задачи, а именно `operator.run()` (метод класса `BaseOperator`). Чтобы использовать его, ОАГ должен быть назначен оператор. Хотя предыдущий пример можно было запустить как есть, не прибегая к созданию ОАГ для тестирования, чтобы использовать метод `run()`, нужно предоставить оператору ОАГ, потому

что когда Airflow запускает задачу, он несколько раз ссылается на объект ОАГ (например, при построении контекста экземпляра задачи).

Мы могли бы определить ОАГ в наших тестах следующим образом:

Листинг 9.22 ОАГ с аргументами по умолчанию для тестирования

```
dag = DAG(
    "test_dag",
    default_args={
        "owner": "airflow",
        "start_date": datetime.datetime(2019, 1, 1),
    },
    schedule_interval="@daily",
)
```

Значения, которые мы предоставляем ОАГ, не важны, но мы будем ссылаться на них при утверждении результатов оператора. Затем можно определить нашу задачу и запустить ее.

Листинг 9.23 Тестирование с ОАГ для визуализации шаблонных переменных

```
def test_movielens_operator(tmp_path, mocker):
    mocker.patch.object(
        MovielensHook,
        "get_connection",
        return_value=Connection(
            conn_id="test", login="airflow", password="airflow"
        ),
    )

    dag = DAG(
        "test_dag",
        default_args={
            "owner": "airflow",
            "start_date": datetime.datetime(2019, 1, 1),
        },
        schedule_interval="@daily",
    )

    task = MovielensDownloadOperator(
        task_id="test",
        conn_id="testconn",
        start_date="{{ prev_ds }}",
        end_date="{{ ds }}",
        output_path=str(tmp_path / "{{ ds }}.json"),
        dag=dag,
    )

    task.run(
        start_date=dag.default_args["start_date"],
        end_date=dag.default_args["start_date"],
    )
```

Если вы запустите тест в том виде, в каком мы его сейчас определили, то, вероятно, столкнетесь с ошибкой, похожей на ту, что содержится в следующем листинге:

Листинг 9.24 Первый запуск теста, включая ОАГ

```
.../site-packages/sqlalchemy/engine/default.py:580: OperationalError

The above exception was the direct cause of the following exception:

➥ > task.run(start_date=dag.default_args["start_date"],
end_date=dag.default_args["start_date"])

...
cursor = <sqlite3.Cursor object at 0x1110fae30>
➥ statement = 'SELECT task_instance.try_number AS task_instance_try_number,
     task_instance.task_id AS task_instance_task_id, task_ins...\nWHERE
     task_instance.dag_id = ? AND task_instance.task_id = ? AND
     task_instance.execution_date = ?\n LIMIT ? OFFSET ?'
parameters = ('test_dag', 'test', '2015-01-01 00:00:00.000000', 1, 0)
...

    def do_execute(self, cursor, statement, parameters, context=None):
>       cursor.execute(statement, parameters)
➥ E       sqlalchemy.exc.OperationalError: (sqlite3.OperationalError) no such
     column: task_instance.max_tries
➥ E       [SQL: SELECT task_instance.try_number AS task_instance_try_number,
     task_instance.task_id AS task_instance_task_id, task_instance.dag_id AS
     task_instance_dag_id, task_instance.execution_date AS
     task_instance_execution_date, task_instance.start_date AS
     task_instance_start_date, task_instance.end_date AS task_instance_end_date,
     task_instance.duration AS task_instance_duration, task_instance.state AS
     task_instance_state, task_instance.max_tries AS task_instance_max_tries,
     task_instance.hostname AS task_instance_hostname, task_instance.unixname AS
     task_instance_unixname, task_instance.job_id AS task_instance_job_id,
     task_instance.pool AS task_instance_pool, task_instance.queue AS
     task_instance_queue, task_instance.priority_weight AS
     task_instance_priority_weight, task_instance.operator AS
     task_instance_operator, task_instance.queued_dttm AS
     task_instance_queued_dttm, task_instance.pid AS task_instance_pid,
     task_instance.executor_config AS task_instance_executor_config
E      FROM task_instance
➥ E      WHERE task_instance.dag_id = ? AND task_instance.task_id = ? AND
     task_instance.execution_date = ?
E       LIMIT ? OFFSET ?]
➥ E       [parameters: ('test_dag', 'test', '2015-01-01 00:00:00.000000', 1, 0)]
E       (Background on this error at: http://sqlalche.me/e/e3q8)
```

Как видно из сообщения об ошибке, в базе метаданных Airflow что-то не так. Чтобы запустить задачу, Airflow запрашивает в базе данных некую информацию, например предыдущие экземпляры задачи с той же датой выполнения. Но если вы не инициализировали базу

данных Airflow (`airflow db init`) в пути, заданном для `AIRFLOW_HOME` (`~/airflow`, если значение не задано), или не сконфигурировали Airflow под работающую базу данных, тогда базы данных для чтения или записи не будет. Также при тестировании нам понадобится база метаданных. Существует несколько подходов к работе с базой метаданных во время тестирования.

Во-первых, гипотетически можно было бы имитировать каждый вызов базы данных, как показано ранее, при запросе учетных данных для подключения. Хотя такое возможно, выглядеть это будет очень громоздко. Более практичный подход – запустить реальную базу метаданных, к которой Airflow может отправлять запросы при запуске тестов.

Для этого нужно выполнить команду `airflow db init`, которая инициализирует базу данных. Это будет база данных SQLite, которая хранится в `~/airflow/airflow.db`. Если задать значение для переменной окружения `AIRFLOW_HOME`, Airflow сохранит базу данных в этом каталоге. Убедитесь, что при выполнении тестов вы предоставляете то же значение `AIRFLOW_HOME`, чтобы Airflow мог найти вашу базу метаданных[1].

Теперь, когда вы настроили базу метаданных Airflow, к которой можно выполнить запрос, можно запустить тест и посмотреть, будет ли он успешным. Кроме того, теперь мы видим, что в базу метаданных Airflow была записана строка во время теста (рис. 9.9).

	task_id	dag_id	execution_date	start_date	end_date	duration	state	try_number
1	test	test_dag	2019-01-01 00:00:00.000000	2019-12-22 21:52:13.111447	2019-12-22 21:52:13.283970	0,172523	success	1

Рис. 9.9 Вызов метода `task.run()` приводит к сохранению сведений о запуске задачи в базе данных

Здесь следует отметить два момента. Если у вас несколько тестов с использованием ОАГ, то есть удобный способ, позволяющий повторно использовать его с помощью pytest. Мы уже рассказывали о фикстурах pytest, и их можно повторно использовать для нескольких файлов в (под)каталогах с файлом conftest.py. В этом файле может содержаться фикстура для создания экземпляра ОАГ.

Листинг 9.25 Пример фикстуры pytest для повторного использования ОАГ во время тестов

```
import datetime

import pytest
from airflow.models import DAG
@pytest.fixture
def test_dag():
    return DAG(
        "test_dag",
```

[1] Чтобы ваши тесты выполнялись изолированно, можно использовать контейнер Docker с пустой инициализированной базой данных Airflow.

```
        default_args={
            "owner": "airflow",
            "start_date": datetime.datetime(2019, 1, 1),
        },
        schedule_interval="@daily",
    )
```

Теперь каждый тест, требующий объекта OAG, может просто создать его экземпляр, добавив `test_dag` в качестве аргумента теста, который выполняет функцию `test_dag()` в начале теста.

Листинг 9.26 Создание требуемых объектов путем включения фикстур с тестом

```
def test_movielens_operator(tmp_path, mocker, test_dag):
    mocker.patch.object(
        MovielensHook,
        "get_connection",
        return_value=Connection(
            conn_id="test",
            login="airflow",
            password="airflow",
        ),
    )

    task = MovielensDownloadOperator(
        task_id="test",
        conn_id="testconn",
        start_date="{{ prev_ds }}",
        end_date="{{ ds }}",
        output_path=str(tmp_path / "{{ ds }}.json"),
        dag=test_dag,
    )

    task.run(
        start_date=dag.default_args["start_date"],
        end_date=dag.default_args["start_date"],
    )
```

`task.run()` – это метод класса `BaseOperator`. `run()` принимает две даты и, учитывая `schedule_interval` OAG, вычисляет экземпляры задачи для запуска между двумя заданными датами. Поскольку мы указываем те же две даты (дату начала OAG), здесь будет только один экземпляр задачи, который нужно выполнить.

9.2.1 Работа с внешними системами

Предположим, что мы работаем с оператором, который подключается к базе данных, скажем `MovielensToPostgresOperator`, который считывает рейтинги MovieLens и записывает результаты в базу данных Postgres. Это часто встречающийся пример, когда источник пре-

доставляет данные только в том виде, в котором они есть во время запроса, но не может предоставить прошлые данные, и некоторым хотелось бы создать историю источника. Например, если сегодня вы выполнили запрос к API MovieLens, где вчера Джон поставил фильму «*Мстители*» четыре звезды, а сегодня изменил рейтинг на пять звезд, API вернет только пятизвездочный рейтинг. Задание Airflow может один раз в день получать все данные и сохранять ежедневный экспорт наряду со временем записи.

Оператор для этого действия может выглядеть так.

Листинг 9.27 Пример оператора, подключающегося к базе данных PostgreSQL

```
from airflow.hooks.postgres_hook import PostgresHook
from airflow.models import BaseOperator

from airflowbook.hooks.movielens_hook import MovielensHook

class MovielensToPostgresOperator(BaseOperator):
   template_fields = ("_start_date", "_end_date", "_insert_query")

   def __init__(
       self,
       movielens_conn_id,
       start_date,
       end_date,
       postgres_conn_id,
       insert_query,
       **kwargs,
   ):
       super().__init__(**kwargs)
       self._movielens_conn_id = movielens_conn_id
       self._start_date = start_date
       self._end_date = end_date
       self._postgres_conn_id = postgres_conn_id
       self._insert_query = insert_query

def execute(self, context):
   with MovielensHook(self._movielens_conn_id) as movielens_hook:
       ratings = list(movielens_hook.get_ratings(
           start_date=self._start_date,
           end_date=self._end_date),
       )

   postgres_hook = PostgresHook(
       postgres_conn_id=self._postgres_conn_id
   )
   insert_queries = [
                    ➥ self._insert_query.format(",".join([str(_[1]) for _ in
 sorted(rating.items())]))
        for rating in ratings
   ]
   postgres_hook.run(insert_queries)
```

Разберемся с методом `execute()`. Он соединяет API MovieLens и базу данных Postgres, извлекая данные и преобразовывая результаты в запросы для Postgres (рис. 9.10).

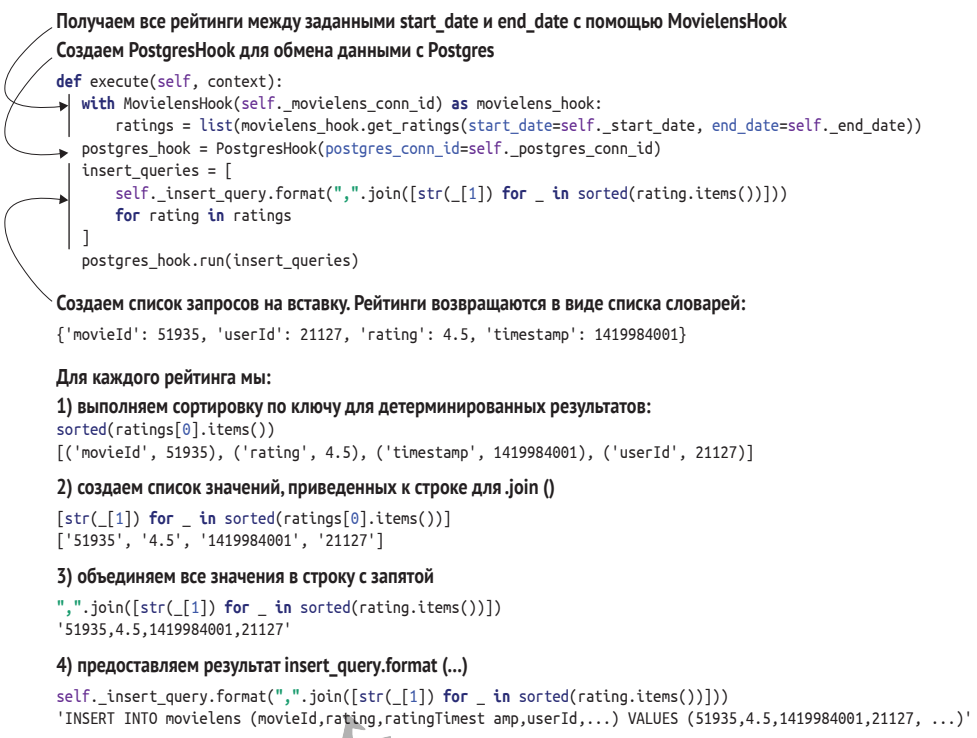


Рис. 9.10 Разбор преобразования данных JSON в запросы Postgres

Как это протестировать, если мы не можем получить доступ к нашей рабочей базе данных Postgres с ноутбука? К счастью, создать локальную базу данных Postgres для тестирования с помощью Docker несложно. В Python есть несколько пакетов, которые предоставляют удобные функции для управления контейнерами Docker в рамках тестов pytest. В следующем примере мы будем использовать pytest-docker-tools (https://github.com/Jc2k/pytest-docker-tools). Этот пакет предоставляет набор удобных вспомогательных функций, с помощью которых можно создать контейнер Docker для тестирования.

Мы не будем вдаваться в детали, но продемонстрируем, как создать образец контейнера Postgres для записи результатов MovieLens. Если оператор работает правильно, то у нас должны быть результаты, записанные в базу данных Postgres в контейнере в конце теста. Тестирование с использованием контейнеров Docker позволяет использовать реальные методы хуков, без необходимости имитировать вызовы, чтобы тестирование выглядело как можно реалистичнее.

Сначала установите pytest-docker-tools в своем окружении с помощью команды `pip install pytest_ docker_tools`. Так мы получаем

ряд вспомогательных функций, таких как `fetch` и `containter`. Сначала мы извлечем контейнер.

Листинг 9.28 Извлекаем образ Docker для тестирования с помощью pytest_docker_tools

```
from pytest_docker_tools import fetch

postgres_image = fetch(repository="postgres:11.1-alpine")
```

Функция `fetch` запускает команду `docker pull` на компьютере, на котором ведется работа (и, следовательно, требуется, чтобы вы установили на нем Docker), и возвращает скачанный образ. Обратите внимание, что сама эта функция представляет собой фикстуру pytest, а это значит, что ее нельзя вызвать напрямую. Мы должны предоставить ее в качестве параметра для теста.

Листинг 9.29 Использование образа Docker в тесте с фикстурами pytest_docker_tools

```
from pytest_docker_tools import fetch

postgres_image = fetch(repository="postgres:11.1-alpine")

def test_call_fixture(postgres_image):
    print(postgres_image.id)
```

При запуске этого теста будет выведено следующее:

```
Fetching postgres:11.1-alpine
PASSED                      [100%]
sha256:b43856647ab572f271decd1f8de88b590e157bfd816599362fe162e8f37fb1ec
```

Теперь мы можем использовать этот идентификатор образа для настройки и запуска контейнера Postgres.

Листинг 9.30 Запуск контейнера Docker для теста с фикстурами pytest_docker_tools

```
from pytest_docker_tools import container

postgres_container = container(
    image="{postgres_image.id}",
    ports={"5432/tcp": None},
)

def test_call_fixture(postgres_container):
    print(
        f"Running Postgres container named {postgres_container.name} "
        f"on port {postgres_container.ports['5432/tcp'][0]}."
    )
```

Функция `containter` из `pytest_docker_tools` также представляет собой фикстуру, поэтому ее тоже можно вызвать, только предоставив ее в качестве аргумента для теста. Требуется несколько аргументов, которые настраивают контейнер для запуска, в данном случае это идентификатор образа, который был возвращен из фикстуры `fetch()`, и порты, которые нужно открыть. Так же, как мы запускаем контейнеры Docker из командной строки, мы можем сконфигурировать переменные окружения, тома и многое другое.

Конфигурация портов требует пояснения. Обычно порт контейнера отображается в тот же порт в хост-системе (например, `docker run -p 5432:5432 postgres`). Контейнер для тестов не должен работать до бесконечности, и, кроме того, мы не хотим конфликтовать с другими портами, используемыми в хост-системе.

Предоставляя словарь ключевому аргументу `ports`, где ключи – это порты контейнера, а значения отображаются в хост-систему, и оставляя значения `None`, мы отображаем порт хоста в случайный открытый порт на хосте (как при запуске `docker run -P`). Если предоставить тесту фикстуру, то это приведет к ее выполнению (т. е. запуску контейнера), а pytest-dockertools затем внутренне отобразит назначенные порты в хост-системе в атрибут `ports` в самой фикстуре. `postgres_container.ports['5432/tcp'][0]` дает нам назначенный номер порта на хосте, который мы затем можем использовать в тесте для подключения.

Чтобы максимально сымитировать реальную базу данных, нужно задать имя пользователя и пароль и инициализировать ее, используя схему и данные для запроса. И то, и другое можно указать в фикстуре `containter`.

Листинг 9.31 Инициализация контейнера Postgres для тестирования с реальной базой данных

```
postgres_image = fetch(repository="postgres:11.1-alpine")
postgres = container(
   image="{postgres_image.id}",
   environment={
       "POSTGRES_USER": "testuser",
       "POSTGRES_PASSWORD": "testpass",
   },
   ports={"5432/tcp": None},
   volumes={
       os.path.join(os.path.dirname(__file__), "postgres-init.sql"): {
           "bind": "/docker-entrypoint-initdb.d/postgres-init.sql"
       }
   },
)
```

Структуру базы данных и данные можно инициализировать в `postgres-init.sql`.

Листинг 9.32 Инициализация схемы для тестовой базы данных

```
SET SCHEMA 'public';
CREATE TABLE movielens (
   movieId integer,
   rating float,
   ratingTimestamp integer,
   userId integer,
   scrapeTime timestamp
);
```

В фикстуре `containter` мы предоставляем имя пользователя и пароль Postgres через переменные окружения. Это особенность образа Postgres Docker, позволяющая сконфигурировать несколько настроек через переменные окружения. Обратитесь к документации по образу Postgres Docker, чтобы узнать обо всех переменных окружения. Еще одна функция образа Docker – возможность инициализировать контейнер сценарием запуска, поместив файл с расширением *.sql, *.sql.gz или *.sh в каталог `/docker-entrypoint-initdb.d`. Они выполняются при загрузке контейнера, перед запуском фактического сервиса Postgres, и их можно использовать для инициализации нашего тестового контейнера с таблицей для запроса.

В листинге 9.31 мы монтируем файл `postgres-init.sql` в контейнер, используя ключевое слово `volume`:

```
volumes={
       os.path.join(os.path.dirname(__file__), "postgres-init.sql"): {
           "bind": "/docker-entrypoint-initdb.d/postgres-init.sql"
       }
    }
```

Мы предоставляем ему словарь, где ключи показывают (абсолютное) местоположение в хост-системе. В данном случае мы сохранили файл `postgres-init.sql` в том же каталоге, что и наш тестовый сценарий, поэтому `os.path.join(os.path.dirname(__file__), "postgres-init.sql")` даст нам абсолютный путь к нему. Значения также являются словарем, где ключ указывает тип монтирования (bind) и значение местоположения внутри контейнера, `/docker-entrypoint-initdb.d` для запуска сценария с расширением *.sql во время загрузки контейнера.

Соберите все это в сценарии, и мы наконец сможем протестировать его на реальной базе данных Postgres.

Листинг 9.33 Завершаем тест с использованием контейнера Docker для тестирования внешних систем

```
import os

import pytest
from airflow.models import Connection
```

```
from pytest_docker_tools import fetch, container

➥ from airflowbook.operators.movielens_operator import MovielensHook,
    MovielensToPostgresOperator, PostgresHook

postgres_image = fetch(repository="postgres:11.1-alpine")
postgres = container(
   image="{postgres_image.id}",
   environment={
        "POSTGRES_USER": "testuser",
        "POSTGRES_PASSWORD": "testpass",
   },
   ports={"5432/tcp": None},
   volumes={
       os.path.join(os.path.dirname(__file__), "postgres-init.sql"): {
           "bind": "/docker-entrypoint-initdb.d/postgres-init.sql"
       }
   },
)

def test_movielens_to_postgres_operator(mocker, test_dag, postgres):
   mocker.patch.object(
       MovielensHook,
       "get_connection",
       return_value=Connection(
          conn_id="test",
          login="airflow",
          password="airflow",
                                  ),
      )
  mocker.patch.object(
      PostgresHook,
      "get_connection",
      return_value=Connection(
         conn_id="postgres",
         conn_type="postgres",
         host="localhost",
         login="testuser",
         password="testpass",
         port=postgres.ports["5432/tcp"][0],
      ),
)

task = MovielensToPostgresOperator(
    task_id="test",
    movielens_conn_id="movielens_id",
    start_date="{{ prev_ds }}",
    end_date="{{ ds }}",
    postgres_conn_id="postgres_id",
    insert_query=(
        "INSERT INTO movielens
  (movieId,rating,ratingTimestamp,userId,scrapeTime) "
        "VALUES ({0}, '{{ macros.datetime.now() }}')"
```

```
    ),
    dag=test_dag,
)

pg_hook = PostgresHook()

row_count = pg_hook.get_first("SELECT COUNT(*) FROM movielens")[0]
assert row_count == 0

task.run(
    start_date=test_dag.default_args["start_date"],
    end_date=test_dag.default_args["start_date"],
)

row_count = pg_hook.get_first("SELECT COUNT(*) FROM movielens")[0]
assert row_count > 0
```

Полный тест получился немного длинным из-за инициализации контейнера и имитации подключения, которое мы должны выполнить. После этого мы создаем экземпляр `PostgresHook` (который использует тот же имитированный метод `get_connection()`, что и в `MovielensToPostgresOperator`, и таким образом подключается к контейнеру Docker Postgres). Сначала мы используем утверждение, что количество строк равно нулю, запускаем оператор и, наконец, тестируем, если были вставлены какие-либо данные.

Что происходит за пределами самой логики тестирования? Во время запуска теста pytest выясняет, какие тесты используют фикстуру, и выполнение будет осуществляться только при использовании данной фикстуры (рис. 9.11).

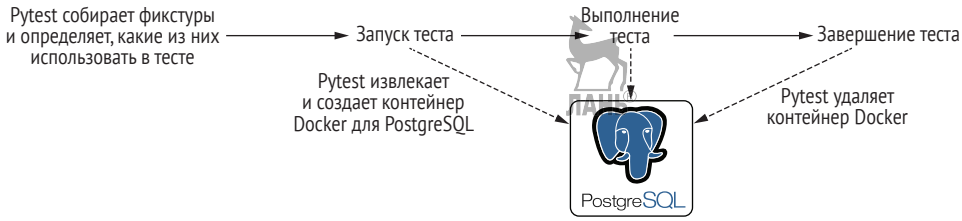


Рис. 9.11 Процесс запуска теста с помощью pytest-docker-tools. Запуск контейнеров Docker во время тестов позволяет проводить тестирование на реальных системах. Жизненный цикл контейнера Docker управляется pytest-docker-tools, а пользователь должен реализовать тест

Когда pytest решает запустить фикстуру `containter`, он будет извлекать, запускать и инициализировать контейнер. На это уходит пара секунд, поэтому в наборе тестов будет небольшая задержка в несколько секунд. По окончании теста фикстуры завершат выполнение. Pytest-docker-tools помещает небольшую функцию-обертку вокруг клиента Python Docker, предоставляя пару удобных конструкций и фикстур для использования в тестах.

9.3 Использование тестов для разработки

Тесты не только помогают проверить правильность кода. Они также полезны во время разработки, потому что позволяют запускать небольшой фрагмент кода без необходимости использовать реальную систему. Посмотрим, как они могут помочь нам при разработке рабочих процессов. Мы продемонстрируем пару скриншотов из PyCharm, но любая современная интегрированная среда разработки позволяет установить точки останова и выполнить отладку.

Вернемся к `MovielensPopularityOperator` из раздела 9.1.3. В методе `execute()` он запускает серию операторов, и мы хотели бы знать состояние на полпути. С помощью PyCharm это можно сделать, установив точку останова и запустив тест, который доходит до строки кода, для которой установлена эта точка (рис. 9.12).

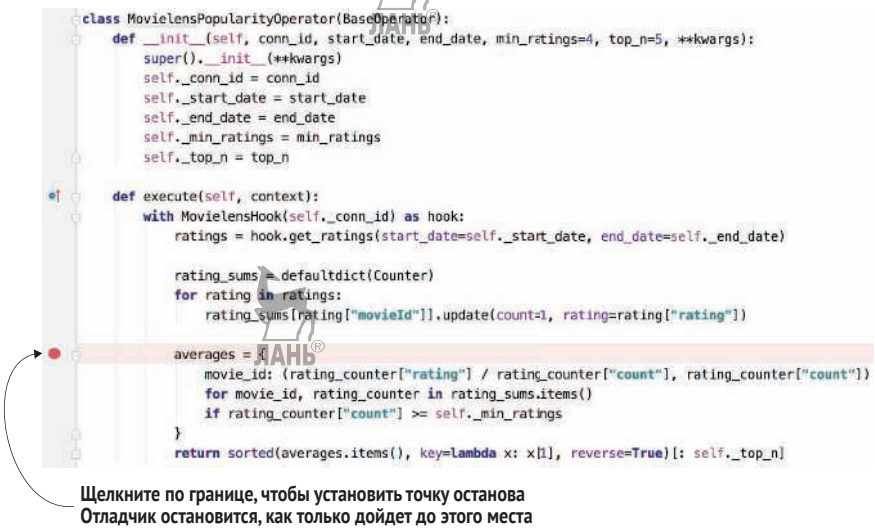


Рис. 9.12 Установка точки останова в интегрированной среде разработки. Этот скриншот был сделан в PyCharm, но любая среда разработки позволяет устанавливать точки останова и выполнять отладку

Теперь запустите тест `test_movielenspopularityoperator` в режиме отладки (рис. 9.13).

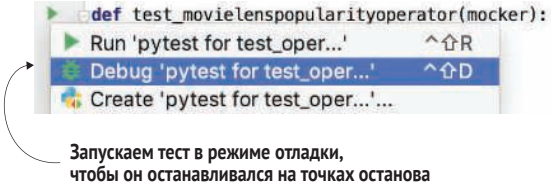


Рис. 9.13 Запуск теста в режиме отладки, чтобы он остановился на точке останова

Как только тест дойдет до строки кода, для которой вы установили точку останова, можно проверить текущее состояние переменных, а также выполнить код в этот момент. Здесь, например, можно проверить контекст экземпляра задачи на полпути через метод `execute()` (рис. 9.14).

Рис. 9.14 Отладка позволяет нам проверить состояние программы в установленной точке останова. Здесь мы проверяем значения контекста

Иногда ваш код работает локально, но на рабочей машине возвращает ошибку. Как выполнить отладку в такой ситуации? Существует способ сделать это удаленно, но данная тема выходит за рамки этой книги. При удаленной отладке вы можете подключить локальный отладчик PyCharm (или другой интегрированной среды разработки) к удаленному работающему процессу Python. (Для получения дополнительной информации поищите в сети «удаленная отладка с PyCharm».)

Есть и другая альтернатива, на которую стоит обратить внимание, если по какой-либо причине вы не можете использовать реальный отладчик: воспользуйтесь отладчиком командной строки (для этого нужен доступ к командной строке на удаленной машине). В Python есть встроенный отладчик pdb (Python Debugger). Он добавляет эту строку кода в то место, которое вы хотите отладить[1].

[1] В Python 3.7 и PEP553 представлен новый способ установки точек останова: просто вызовите метод `breakpoint()`.

Листинг 9.34 Установка точки останова в коде

```
import pdb; pdb.set_trace()
```

Теперь вы можете запустить свой код из командной строки, выполнив тест с pytest или запустив задачу Airflow в ОАГ с помощью интерфейса командной строки:

```
airflow tasks test [dagid] [taskid] [execution date]
```

Вот пример:

```
airflow tasks test movielens_download fetch_data 2019-01-01T12:00:00
```

Команда `airflow tasks test` запускает задачу без регистрации каких-либо записей в базе метаданных. Она полезна для запуска и тестирования отдельных задач в промышленном окружении. Как только точка останова pdb будет достигнута, вы можете выполнить код и управлять отладчиком с помощью определенных ключей, таких как `n` для выполнения оператора и перехода к следующей строке и `l` для отображения окружающих строк (рис. 9.15). (Просмотрите полный список команд, выполнив поиск по запросу «pdb cheat sheet» в интернете.)

Оператор, в котором pdb делает паузу

на «1» для проверки окружающих строк,

показывает следующую строку для выполнения

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> PDB set_trace >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> >>>>>
> /src/airflowbook/operators/movielens_operator.py( 70)execute()
-> postgres_hook = PostgresHook(postgres_conn_id=se lf._postgres_conn_id)
(Pdb) l
65 with MovielensHook(self._movielens_conn_id) as m ovielens_hook:
66 ratings = list(movielens_hook.get_ratings(start_ date=self._start_date, end_date=self._end_date))
67
68 import pdb; pdb.set_trace()
69
70 -> postgres_hook = PostgresHook(postgres_conn_id =self._postgres_conn_id)
71 insert_queries = [
72 self._insert_query.format(",".join([str(_[1]) fo r _ in sorted(rating.items())]))
73 for rating in ratings
74 ]
75 postgres_hook.run(insert_queries)
(Pdb) len(ratings)
3103
(Pdb) n
> /src/airflowbook/operators/movielens_operator.py( 72)execute()
-> self._insert_query.format(",".join([str(_[1]) fo r _ in sorted(rating.items())]))
```

Проверяем, содержат ли рейтинги переменных какие-либо значения, выводя длину

Вычисляем строку и переходим к следующей

Рис. 9.15 Отладка из командной строки с помощью pdb

9.3.1 Тестирование полных ОАГ

До сих пор мы рассматривали различные аспекты тестирования отдельных операторов: тестирование с контекстом экземпляра задачи и без него, операторы, использующие локальную файловую систему, и операторы, использующие внешние системы с помощью Docker. Но все это было сосредоточено на тестировании одного оператора. Большой и важный аспект разработки рабочих процессов – убедиться, что все строительные блоки хорошо сочетаются друг с другом. Хотя один оператор может работать правильно с логической точки зрения, он может, например, преобразовать данные неожиданным способом, что приведет к сбою последующего оператора. Как же убедиться, что все операторы в ОАГ работают, как и ожидалось?

К сожалению, ответить на этот вопрос непросто. Имитация реального окружения не всегда возможна по разным причинам. Например, в случае с разделенной системой DTAP (аббревиатура, образованная из начальных букв слов **D**evelopment, **T**esting, **A**cceptance и **P**roduction, поэтапный подход, применяемый к тестированию и развертыванию программного обеспечения) часто нельзя создать идеальную копию рабочего варианта в окружении разработки из-за правил конфиденциальности или размера данных. Допустим, в промышленном окружении содержится петабайт данных; в этом случае было бы непрактично (мягко говоря) поддерживать синхронизацию данных во всех четырех окружениях. Поэтому создается максимально реалистичное промышленное окружение, которое можно использовать для разработки и проверки программного обеспечения. В случае с Airflow это не исключение, и мы уже видели несколько подходов к решению данной проблемы. Мы кратко опишем два подхода в разделах 9.4 и 9.5.

9.4 Эмулируйте промышленное окружение с помощью Whirl

Один из подходов к воссозданию промышленного окружения – проект Whirl (https://github.com/godatadriven/whirl). Его суть состоит в том, чтобы моделировать все компоненты промышленного окружения в контейнерах Docker и управлять всем этим с помощью инструмента Docker Compose. Whirl поставляется с утилитой командной строки, чтобы можно было с легкостью управлять этими окружениями. Хотя Docker – отличный инструмент для разработки, один из его недостатков состоит в том, что не все может быть доступно в качестве образа Docker, например Google Cloud Storage.

9.5 Создание окружений

Локальная эмуляция промышленного окружения с помощью Docker или работа с таким инструментом, как Whirl, не всегда возможна. Одна из причин – безопасность (например, иногда невозможно подключить локальную установку Docker к FTP-серверу, используемому в рабочих ОАГ, потому что FTP-сервер включен в список разрешенных IP-адресов).

Один из подходов, который часто обсуждается с сотрудником службы безопасности, – это настройка изолированных окружений. Иногда бывает неудобно настраивать и сопровождать четыре полноценных окружения, поэтому в небольших проектах с небольшим количеством людей иногда используются всего два (окружение разработки и промышленное окружение). У каждого окружения могут быть определенные требования, такие как фиктивные данные в окружениях разработки и тестирования. Реализация такого DTAP-конвейера часто касается конкретного проекта и инфраструктуры и выходит за рамки этой книги.

В контексте проекта Airflow целесообразно создать одну выделенную ветку в репозитории GitHub для каждого окружения: окружение разработки > ветка development, промышленное окружение > production / main и т. д. Таким образом вы можете вести разработку локально, в ветках. Сначала выполните слияние (ветка development) и запустите ОАГ в окружении разработки. Как только вы будете удовлетворены результатами, внесите свои изменения в следующую ветку, скажем main, и запустите рабочие процессы в соответствующем окружении.

Резюме

- Тест на благонадежность фильтрует основные ошибки в ваших ОАГ.
- Модульное тестирование проверяет правильность отдельных операторов.
- pytest и плагины предоставляют ряд полезных конструкций для тестирования, таких как временные каталоги и плагины для управления контейнерами Docker во время тестов.
- Операторы, которые не используют контекст экземпляра задачи, можно просто запускать с помощью метода `execute()`.
- Операторы, использующие контекст экземпляра задачи, должны запускаться вместе с ОАГ.
- Для интеграционного тестирования нужно максимально точно смоделировать свое промышленное окружение.

10 Запуск задач в контейнерах

Эта глава рассказывает:

- о том, как выявить проблемы, связанные с управлением развертыванием Airflow;
- об упрощении развертывания Airflow;
- о запуске контейнерных задач в Airflow с использованием Docker;
- о высокоуровневом обзоре рабочих процессов при разработке контейнерных ОАГ.

В предыдущих главах мы реализовали несколько ОАГ с использованием различных операторов Airflow, каждый из которых специализировался на выполнении определенного типа задач. В этой главе мы поговорим о недостатках использования множества разных операторов, особенно обращая внимание на создание ОАГ, которые легко создавать, развертывать и сопровождать. В свете этих проблем мы рассмотрим, как использовать Airflow для запуска задач в контейнерах, применение Docker и Kubernetes и преимущества данного подхода.

10.1 Проблемы, вызываемые множеством разных операторов

Операторы, возможно, являются одной из сильных сторон Airflow, поскольку обеспечивают отличную гибкость для координации заданий в различных типах систем. Однако создание и управление ОАГ со множеством разных операторов могут оказаться довольно сложной задачей.

Чтобы понять, почему, рассмотрим ОАГ на базе нашей рекомендательной системы из главы 8. Он показан на рис. 10.1. Этот ОАГ состоит из трех разных задач: извлечение рекомендаций по фильмам из нашего API, ранжирование фильмов на основе полученных рекомендаций и отправка этих фильмов в базу данных MySQL для дальнейшего использования. Обратите внимание, что этот относительно простой ОАГ уже использует три разных оператора: `HttpOperator` (или какой-либо другой оператор API) для доступа к API, `PythonOperator` для выполнения функции рекомендательной системы и `MySQLOperator` для сохранения результатов.

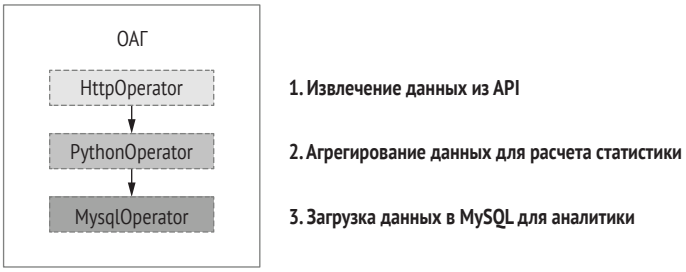


Рис. 10.1 Иллюстрация ОАГ рекомендательной системы. Он извлекает рекомендации по фильмам, использует их для ранжирования и сохраняет результат в базе данных. Каждый из этих этапов включает в себя разного оператора, что усложняет разработку и сопровождение ОАГ

10.1.1 Интерфейсы и реализации операторов

Недостаток использования разных операторов для каждой из этих задач состоит в том, что нам нужно ознакомиться с интерфейсами и внутренним устройством каждого оператора, чтобы эффективно использовать их. Кроме того, если мы столкнемся с ошибками в каком-либо из операторов[1], нам потребуется потратить драгоценное время и ресурсы на отслеживание основных проблем и их устранение. Хотя эти усилия могут показаться приемлемыми для такого небольшого примера, представьте себе сопровождение развертывания Air-

[1] К сожалению, это не редкость, особенно в случае с малораспространенными и не особо часто используемыми операторами Airflow.

flow с большим количеством разных ОАГ, где используется множество различных операторов. При таком сценарии работать со всеми этими операторами будет сложнее.

10.1.2 Сложные и конфликтующие зависимости

Еще одна проблема, возникающая при использовании множества разных операторов, заключается в том, что каждому из них обычно требуется свой набор зависимостей (Python или что-то иное). Например, `HttpOperator` зависит от библиотеки requests для выполнения HTTP-запросов, тогда как `MySQLOperator` нужны зависимости на уровне Python и/или на уровне системы для взаимодействия с MySQL. Точно так же код рекомендательной системы, вызываемый `PythonOperator`, скорее всего, будет иметь собственный набор зависимостей (таких как pandas, scikit-learn и т. д., если используется машинное обучение).

Из-за способа настройки Airflow все эти зависимости должны быть установлены в окружении, в котором работает планировщик Airflow, а также сами воркеры. При использовании множества разных операторов для этого требуется установка большого количества зависимостей[1], что приводит к потенциальным конфликтам (рис. 10.2) и большой сложности в настройке и сопровождении этих окружений (не говоря уже о потенциальных рисках безопасности при установке такого количества программных пакетов). Конфликты представляют собой особую проблему в окружениях Python, поскольку Python не дает никакого механизма для установки нескольких версий одного и того же пакета в одном окружении.

10.1.3 Переход к универсальному оператору

Из-за этих проблем с использованием и сопровождением множества различных операторов и их зависимостей некоторые утверждают, что лучше было бы сосредоточиться на использовании одного универсального оператора для выполнения задач. Положительный момент такого подхода состоит в том, что нам нужно знать только один вид операторов, а это означает, что наши ОАГ внезапно становятся намного проще для понимания, поскольку состоят только из одного типа задач. Более того, если все будут использовать один и тот же оператор для запуска своих задач, то мы с меньшей вероятностью столкнемся с ошибками в таком операторе. Наконец, наличие только одного оператора означает, что нам нужно беспокоиться лишь об одном наборе зависимостей Airflow – тех, что необходимы для этого оператора.

[1] Просто посмотрите на файл setup.py, чтобы получить представление об огромном количестве зависимостей, участвующих в поддержке всех операторов Airflow.

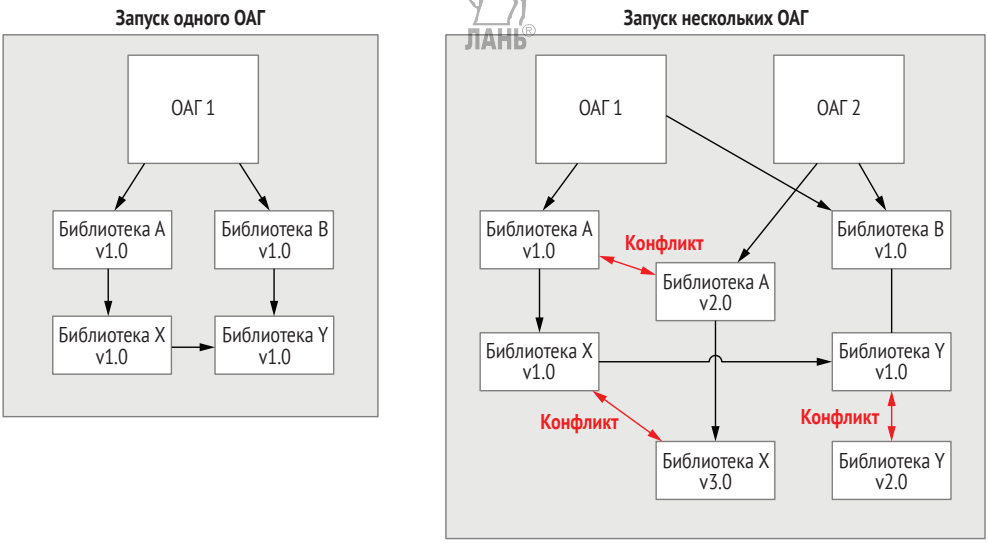


Рис. 10.2 Запуск большого количества ОАГ в одном окружении может привести к конфликтам, когда ОАГ зависят от разных версий одних и тех же (или связанных) пакетов. В частности, Python не поддерживает установку разных версий одного и того же пакета в одном окружении. Это означает, что любые конфликты в пакетах (справа) необходимо будет решить, заново написав ОАГ (или их зависимости), чтобы использовать те же версии пакета

Но где же найти такой универсальный оператор, способный выполнять множество различных задач, который в то же время не требует от нас установки и управления зависимостями для каждой из них? Вот тут-то и пригодятся контейнеры.

10.2 Представляем контейнеры

Контейнеры рекламируются как одна из основных последних разработок, позволяющих с легкостью упаковывать приложения с необходимыми зависимостями и легко развертывать их в различных окружениях единообразно. Прежде чем перейти к тому, как использовать контейнеры в Airflow, для начала дадим краткое введение[1], чтобы убедиться, что все мы находимся на одной волне. Если вы уже знакомы с Docker и концепциями контейнеров, переходите к разделу 10.3.

[1] Для полного введения мы с радостью дадим вам ссылки на множество книг о виртуализации на основе контейнеров и связанных с ними технологиях, таких как Docker/Kubernetes.

10.2.1 Что такое контейнеры?

Исторически сложилось так, что одной из самых больших проблем при разработке программных приложений было их развертывание (т. е. обеспечение правильной и стабильной работы приложений на целевой машине (машинах)). Обычно это связано с жонглированием и учетом множества различных факторов, в том числе различий между операционными системами, вариаций установленных зависимостей и библиотеки, различного оборудования и т. д.

Один из подходов к управлению этой сложностью – использовать виртуализацию, при которой приложения устанавливаются на виртуальную машину, работающую поверх клиентской операционной системы хоста (рис. 10.3). При использовании такого подхода приложения видят только операционную систему (ОС) виртуальной машины, а это означает, что нам нужно лишь убедиться, что виртуальная ОС соответствует требованиям нашего приложения, вместо того чтобы изменять ОС хоста. Таким образом, чтобы развернуть приложение, можно просто установить его со всеми необходимыми зависимостями в виртуальную ОС, которую мы затем можем отправить нашим клиентам.

Недостаток виртуальных машин состоит в том, что они довольно тяжелые, поскольку требуют запуска всей операционной системы (виртуальной или гостевой ОС) поверх операционной системы хоста. Более того, каждая новая виртуальная машина будет работать под управлением собственной гостевой операционной системы, то есть для запуска нескольких приложений в виртуальных машинах на одном компьютере потребуются значительные ресурсы.

Это ограничение привело к развитию виртуализации на базе контейнеров, которая представляет собой гораздо более легкий подход, нежели виртуальные машины (рис. 10.3). В отличие от виртуальных машин, подходы к виртуализации на основе контейнеров используют функциональные возможности на уровне ядра в ОС хоста для виртуализации приложений. Это означает, что контейнеры могут разделять приложения и их зависимости так же, как и виртуальные машины, не требуя при этом, чтобы каждое приложение запускало собственную операционную систему; они могут просто использовать эту функциональность из ОС хоста.

Взаимодействием между контейнерами и операционной системой хоста часто управляет служба, которую называют менеджер контейнеров. Она предоставляет API для управления и запуска различных контейнеров приложений и их образов, а также часто предоставляет инструменты командной строки, которые помогают пользователям создавать свои контейнеры и взаимодействовать с ними. Самым известным таким решением является Docker, который за долгие годы приобрел большую популярность благодаря своей относительной простоте в использовании и большому сообществу.

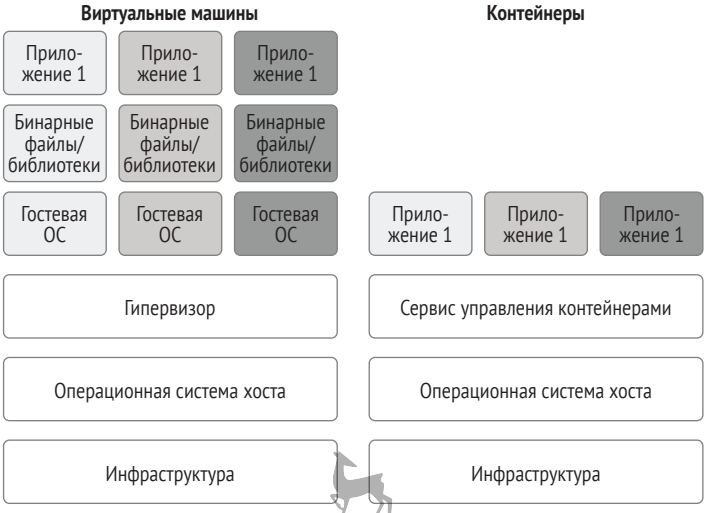


Рис. 10.3 Сравнение виртуальных машин (ВМ) и контейнеров. Обратите внимание, что контейнеры намного легче, поскольку им не требуется запускать всю гостевую ОС для каждого приложения

10.2.2 Запуск нашего первого контейнера Docker

Чтобы изучить жизненный цикл создания и запуска контейнера, попробуем создать небольшой контейнер с помощью Docker. Надеемся, это даст вам прочувствовать работу с контейнерами и задействованный здесь рабочий процесс разработки. Перед тем как начать, убедитесь, что у вас установлен Docker. Инструкции по установке Docker Desktop можно найти на странице https://www.docker.com/get-started. После того как вы установите и запустите Docker, можно запустить наш первый контейнер, используя следующую команду в вашем терминале:

Листинг 10.1 Запуск контейнера Docker

```
$ docker run debian:buster-slim echo Hello, world!
```

После выполнения этой команды вы должны получить примерно такой результат:

```
Unable to find image 'debian:buster-slim' locally
latest: Pulling from library/debian
...
Digest: sha256:76c15066d7db315b42dc247b6d439779d2c6466f
➥ 7dc2a47c2728220e288fc680
Status: Downloaded newer image for debian:buster-slim
Hello, world!
```

Что произошло, когда мы выполнили эту команду? Если говорить коротко, Docker сделал для нас следующее:

1. Клиент docker связался с демоном Docker (контейнерная служба, работающая на нашем локальном компьютере).
2. Демон извлек образ Debian Docker, содержащий базовые двоичные файлы и библиотеки Debian из реестра Docker Hub (онлайн-сервис для хранения образов Docker).
3. Используя этот образ, демон создал новый контейнер.
4. Контейнер выполнил нашу команду `echo Hello, world` внутри контейнера.
5. Демон передал вывод команды клиенту Docker, показав его на нашем терминале.

Это означает, что мы смогли выполнить команду `echo Hello, world` внутри контейнера Ubuntu на своей локальной машине, независимо от операционной системы хоста. Довольно круто!

Точно так же можно выполнять команды в Python, используя следующую команду:

Листинг 10.2 Выполнение команды внутри контейнера Python

```
$ docker run python:3.8 python -c 'import sys; print(sys.version)'
```

По сути, мы выполняем команду Python внутри контейнера Python. Обратите внимание, что здесь мы указываем тег для образа (3.8), который в этом случае гарантирует, что мы используем версию образа Python, содержащего Python 3.8.

10.2.3 Создание образа Docker

Хотя запустить существующий образ довольно просто, что, если нам нужно включить собственное приложение в образ, чтобы мы могли запускать его с помощью Docker? Проиллюстрируем этот процесс небольшим примером.

В этом примере у нас есть небольшой скрипт (fetch_weather.py), который извлекает прогнозы погоды из API wttr.in (http://wttr.in) и записывает вывод этого API в выходной файл. У этого сценария есть пара зависимостей, и мы хотим упаковать все это в виде образа Docker, чтобы конечным пользователям было легче его запускать.

Можно начать создание образа Docker с создания файла Dockerfile, который, по сути, представляет собой текстовый файл, описывающий Docker, как создать образ. Базовая структура этого файла выглядит примерно так:

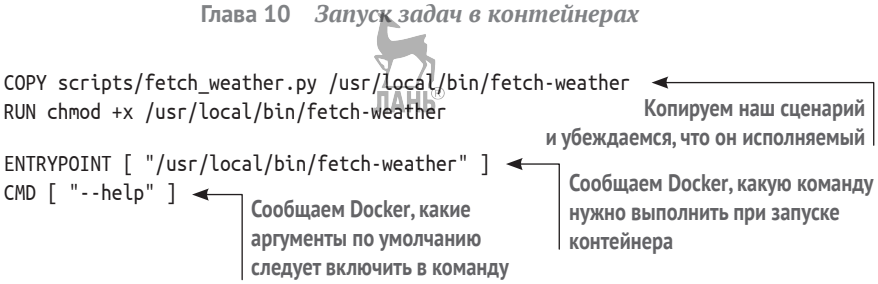
Листинг 10.3 Файл Dockerfile для извлечения прогнозов погоды из API wttr

```
FROM python:3.8-slim

COPY requirements.txt /tmp/requirements.txt
RUN pip install -r /tmp/requirements.txt
```

Сообщаем Docker, какой образ использовать в качестве основы для создания нашего образа

Копируем файл requirements и запускаем pip, чтобы установить требования



Каждая строка файла Dockerfile – это, по сути, инструкция, сообщающая Docker о необходимости выполнения конкретной задачи при построении образа. Большинство файлов Dockerfile начинаются с инструкции `FROM`, которая сообщает Docker, какой базовый образ нужно использовать в качестве отправной точки. Остальные инструкции (`COPY`, `ADD`, `ENV` и т. д.) затем сообщают Docker, как добавить дополнительные слои к базовому образу, содержащему ваше приложение и его зависимости.

Чтобы создать образ с помощью этого файла Dockerfile, можно использовать следующую команду:

Листинг 10.4 Создание образа Docker с использованием файла Dockerfile

```
$ docker build --tag manning-airflow/wttr-example .
```

Здесь мы даем Docker указание создать образ Docker, используя текущий каталог (.) в качестве контекста сборки. Затем Docker будет искать в этом каталоге файл Dockerfile, а также любые файлы, включенные в операторы `ADD`/`COPY` (например, наш сценарий и требования к файлу). Аргумент `--tag` сообщает Docker, какое имя присвоить созданному образу (в данном случае это `manning-airflow/wttr-example`).

Выполнение команды `build` даст примерно следующий вывод:

```
Sending build context to Docker daemon 5.12kB
Step 1/7 : FROM python:3.8-slim
 ---> 9935a3c58eae
Step 2/7 : COPY requirements.txt /tmp/requirements.txt
 ---> 598f16e2f9f6
Step 3/7 : RUN pip install -r /tmp/requirements.txt
 ---> Running in c86b8e396c98
Collecting click
...
Removing intermediate container c86b8e396c98
 ---> 102aae5e3412
Step 4/7 : COPY scripts/fetch_weather.py /usr/local/bin/fetch-weather
 ---> 7380766da370
Step 5/7 : RUN chmod +x /usr/local/bin/fetch-weather
 ---> Running in 7d5bf4d184b5
Removing intermediate container 7d5bf4d184b5
 ---> cae6f678e8f8
```

```
Step 6/7 : ENTRYPOINT [ "/usr/local/bin/fetch-weather" ]
 ---> Running in 785fe602e3fa
Removing intermediate container 785fe602e3fa
 ---> 3a0b247507af
Step 7/7 : CMD [ "--help" ]
 ---> Running in bad0ef960f30
Removing intermediate container bad0ef960f30
 ---> ffabdb642077
Successfully built ffabdb642077
Successfully tagged wttr-example:latest
```

По сути, здесь показан весь процесс сборки, связанный с созданием нашего образа, начиная с базового образа Python (этап 1) и заканчивая последней инструкцией `CMD` (этап 7). В конце Docker заявляет, что пометил созданный образ указанным именем.

Чтобы выполнить тестовый запуск созданного образа, можно использовать следующую команду.

Листинг 10.5 Запуск контейнера Docker с использованием образа wttr

```
$ docker run manning-airflow/wttr-example:latest
```

После этого должно быть выведено такое сообщение из нашего сценария внутри контейнера:

```
Usage: fetch-weather [OPTIONS] CITY

  CLI application for fetching weather forecasts from wttr.in.

Options:
  --output_path FILE  Optional file to write output to.
  --help              Show this message and exit.
```

Теперь, когда у нас есть образ контейнера, можно начать использовать его для извлечения прогнозов погоды из API wttr в следующем разделе.

10.2.4 Сохранение данных с использованием томов

Можно запустить образ `wttr-example`, который мы создали в предыдущем разделе, чтобы извлечь прогнозы погоды для такого города, как Амстердам, используя следующую команду Docker.

Листинг 10.6 Запуск контейнера wttr для определенного города

```
$ docker run wttr-example:latest Amsterdam
```

Предположим, что все идет правильно, и в результате мы должны получить прогнозы погоды в Амстердаме, которые будут выведены в терминале наряду с причудливыми графиками (рис. 10.4).

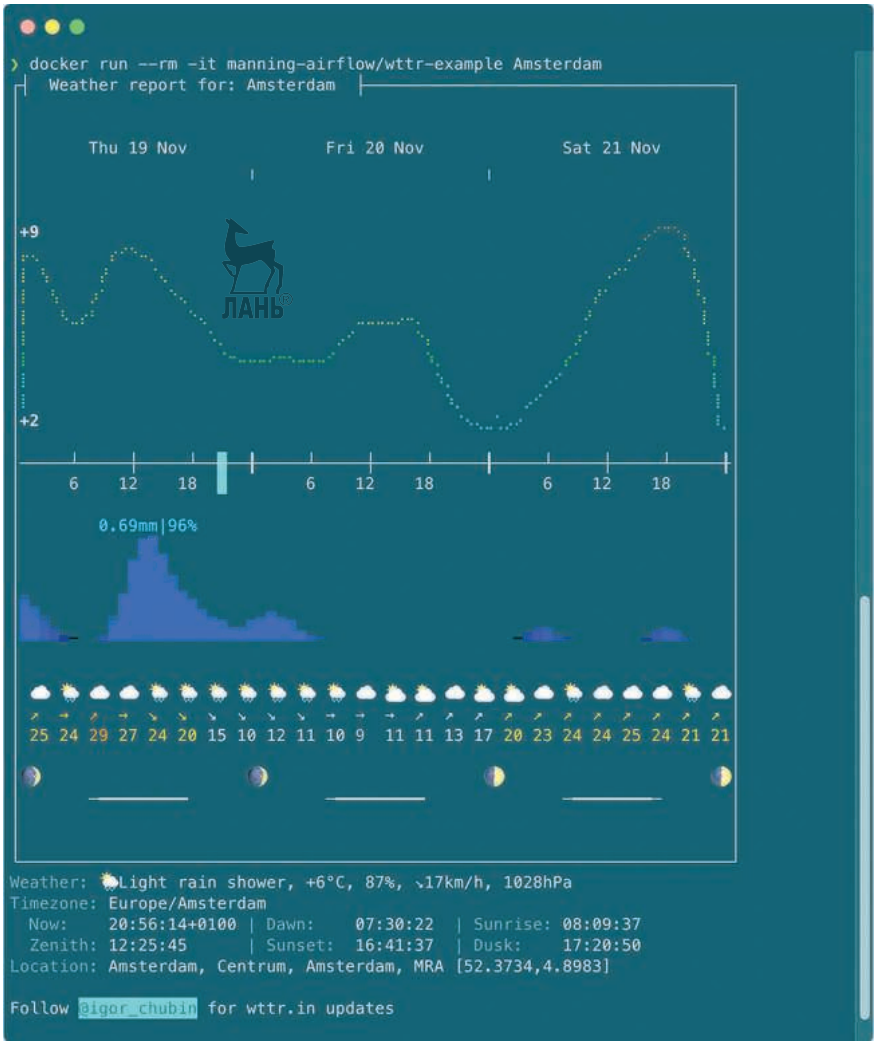


Рис. 10.4 Пример вывода из контейнера `wttr-example` для Амстердама

Чтобы создать историю прогнозов погоды, также можно записать прогнозы в какой-нибудь выходной файл (файлы), который можно использовать для дальнейшего применения или анализа. К счастью, наш сценарий включает дополнительный аргумент `--output_path`, который позволяет нам указать путь к выходному файлу для записи прогнозов, вместо того чтобы писать их в консоль.

Однако если вы попытаетесь выполнить эту команду с путем к локальному файлу, то увидите, что фактически никаких выходных данных в формате JSON в вашей локальной файловой системе не создается:

```
$ docker run wttr-example:latest Amsterdam --output_path amsterdam.json
$ ls amsterdam.json
ls: amsterdam.json: No such file or directory
```

Это связано с тем, что окружение контейнера изолировано от операционной системы хоста, а это означает, что он (среди прочего) имеет изолированную файловую систему, отделенную от файловой системы хоста.

Чтобы поделиться файлами с контейнером, необходимо убедиться, что файлы доступны в файловой системе, к которой имеет доступ контейнер. Один из часто используемых вариантов – чтение и запись файлов с применением хранилища, к которому можно получить доступ через интернет (например, хранилище Amazon S3) или локальную сеть. Кроме того, можно монтировать файлы или папки из вашей хост-системы в контейнер, дабы они были доступны изнутри контейнера.

Чтобы смонтировать файл или папку в свой контейнер, необходимо указать для команды `docker run` аргумент `-volume`, который определяет, какой файл или папку нужно монтировать и желаемый путь внутри контейнера.

Листинг 10.7 Монтирование тома при запуске контейнера

```
$ docker run --volume `pwd`/data:/data wttr-example ...
```

Монтируем данные локального каталога (слева) в контейнере в /data

Здесь мы сообщаем Docker о необходимости монтировать данные локальной папки по пути `/data` внутри контейнера. Это означает, что теперь мы можем записать выходные данные о погоде в смонтированный том данных с помощью следующей команды:

Листинг 10.8 Сохранение вывода из контейнера wttr

```
$ docker run --rm --volume `pwd`/data:/data \
➥ wttr-example Amsterdam --output_path /data/amsterdam.json
```

Передаем в контейнер дополнительные аргументы из Amsterdam и --output_path

Можно удостовериться, что все работает, проверив, действительно ли текстовый файл существует, после того как наш контейнер закончил работу:

```
$ ls data/amsterdam.json
data/amsterdam.json
```

Когда вы закончите работу с контейнерами, то можете использовать следующую команду, чтобы проверить, не работают ли еще какие-либо контейнеры:

```
$ docker ps
```

Остановить все запущенные контейнеры можно с помощью команды `docker stop`, используя идентификаторы, полученные из предыдущей команды, для ссылки на запущенные контейнеры:

```
$ docker stop <container_id>
```

Остановленные контейнеры все еще находятся в приостановленном состоянии в фоновом режиме на случай, если позже вы захотите снова запустить их. Если вам больше не нужен контейнер, можно полностью удалить его с помощью команды `docker rm`:

```
$ docker rm <container_id>
```

Обратите внимание, что остановленные контейнеры не отображаются по умолчанию при использовании команды `docker ps` для поиска работающих контейнеров. Остановленные контейнеры можно просмотреть, используя параметр `-a`, выполнив команду `docker ps`:

```
$ docker ps -a
```

10.3 Контейнеры и Airflow

Теперь, когда у нас есть базовое представление о том, что такое контейнеры Docker и как их использовать, вернемся к Airflow. В этом разделе мы рассмотрим, как применять контейнеры в Airflow и каковы их потенциальные преимущества.

10.3.1 Задачи в контейнерах

Airflow позволяет запускать задачи в качестве контейнеров. На практике это означает, что вы можете использовать контейнерные операторы (такие как `DockerOperator` и `KubernetesPodOperators`) для определения задач. Эти операторы при выполнении начнут запускать контейнер и будут ждать, пока он завершит выполнение того, что предполагалось (аналогично команде `docker run`).

Результат каждой задачи зависит от выполненной команды и программного обеспечения внутри образа контейнера. В качестве примера рассмотрим ОАГ рекомендательной системы (рис. 10.1). В исходном примере используются три оператора для выполнения трех разных задач, а именно: для извлечения рейтингов (с использованием `HttpOperator`), ранжирования фильмов (с помощью `PythonOperator`) и публикации результатов (с использованием оператора на базе MySQL). Используя подход на базе Docker (рис. 10.5), можно было бы заменить эти задачи с помощью `DockerOperator` и использовать его для выполнения команд в трех разных контейнерах Docker с соответствующими зависимостями.

10.3.2 Зачем использовать контейнеры?

Конечно, такой подход на основе контейнеров требует создания образов для каждой из задач (хотя иногда можно использовать образы в связанных или похожих задачах). Таким образом, вы можете спросить себя,

зачем создавать и сопровождать эти образы Docker, когда можно реализовать все с помощью нескольких сценариев или функций Python?

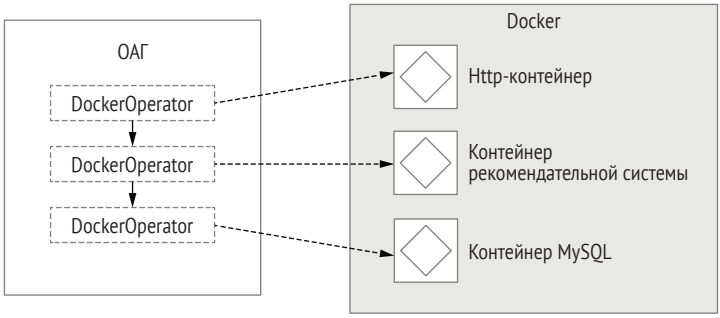


Рис. 10.5 Docker-версия ОАГ рекомендательной системы из рис. 10.1

Легкое управление зависимостями

Одно из самых больших преимуществ использования контейнеров (Docker) состоит в том, что они обеспечивают более простой подход к управлению зависимостями. Создавая разные образы для разных задач, вы можете устанавливать точные зависимости, необходимые для каждой из них, в соответствующий образ. Поскольку затем задачи выполняются изолированно в этих образах, вам больше не нужно иметь дело с конфликтами в зависимостях между задачами (рис. 10.6). В качестве дополнительного преимущества можно упомянуть тот факт, что вам не нужно устанавливать какие-либо зависимости задач в окружении воркеров Airflow (только в Docker), поскольку задачи больше не запускаются в них.

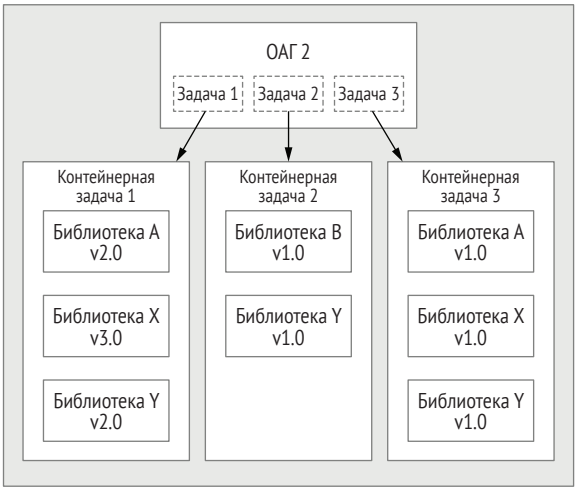


Рис. 10.6 Управление зависимостями в разных задачах с использованием контейнеров

Единый подход к запуску различных задач

Еще одно преимущество использования контейнеров для задач состоит в том, что каждая контейнерная задача имеет один и тот же интерфейс, поскольку все они фактически представляют собой одну и ту же операцию (запуск контейнера), выполняемую одним и тем же оператором (например, `DockerOperator`). Единственные отличия – задействованные образы с небольшими вариациями в конфигурации и выполняемая команда. Такое единообразие упрощает разработку ОАГ, поскольку нужно выучить только одного оператора. И если возникают какие-либо проблемы, связанные с оператором, нужно лишь отладить и исправить проблемы в одном этом операторе, а не разбираться во множестве разных операторов.

Улучшенная тестируемость

Наконец, еще одно преимущество использования образов контейнеров состоит в том, что их можно разрабатывать и сопровождать отдельно от ОАГ Airflow, в котором они работают. Это означает, что у каждого образа может быть собственный жизненный цикл разработки и его можно подвергнуть специальному тесту (например, запуск с имитированными данными), чтобы проверить, делает ли программное обеспечение в образе то, что мы ожидаем. Разделение на контейнеры делает это тестирование проще, чем, например, при использовании `PythonOperator`, который часто включает в себя задачи, тесно связанные с самим ОАГ, что затрудняет тестирование функций отдельно от слоя оркестровки Airflow.

10.4 Запуск задач в Docker

После этого введения пришло время приступить к реализации нашего ОАГ рекомендательной системы в контейнерах. В этом разделе мы расскажем, как запустить существующий ОАГ в контейнерах с помощью Docker.

10.4.1 Знакомство с DockerOperator

Самый простой способ запустить задачу в контейнере с Airflow – использовать `DockerOperator`, доступный из пакета провайдера `apache-airflow-provider-docker`[1]. Из названия оператора видно, что он позволяет запускать задачи в контейнерах с использованием Docker. Базовый API оператора выглядит так:

[1] Для Airflow версии 1.10.x можно установить `DockerOperator`, используя пакет `apache-airflow-backport-sizes-docker-backport`.

Листинг 10.9 Пример использования DockerOperator

```
rank_movies = DockerOperator(
        task_id="rank_movies",
        image="manning-airflow/movielens-ranking",
        command=[
            "rank_movies.py",
            "--input_path",
            "/data/ratings/{{ds}}.json",
            "--output_path",
            "/data/rankings/{{ds}}.csv",
        ],
        volumes=["/tmp/airflow/data:/data"],
    )
```

Сообщаем DockerOperator, какой образ использовать

Указываем, какую команду запускать в контейнере

Определяем, какие тома монтировать внутри контейнера (формат: host_path: container_path)

Идея `DockerOperator` заключается в том, что он является эквивалентом команды `docker run` (как показано в предыдущем разделе) для запуска конкретного образа с конкретными аргументами и ждет, пока контейнер завершит свою работу. В данном случае мы даем Airflow указание запустить сценарий `rank_movies.py` внутри образа `manningairflow/movielens-ranking`, с дополнительными аргументами, указывающими на то, где сценарий должен читать или записывать свои данные. Обратите внимание, что мы также предоставляем дополнительный аргумент `volumes`, который монтирует каталог данных в контейнер, чтобы мы могли предоставить контейнеру входные данные, а также сохранить результаты после завершения задачи или работы контейнера.

Что происходит, когда выполняется этот оператор? По сути, происходящее показано на рис. 10.7. Сначала Airflow дает воркеру указание выполнить задачу, запланировав ее (1). Затем `DockerOperator` выполняет команду `docker run` на машине, где функционирует воркер, с соответствующими аргументами (2). Потом, при необходимо-

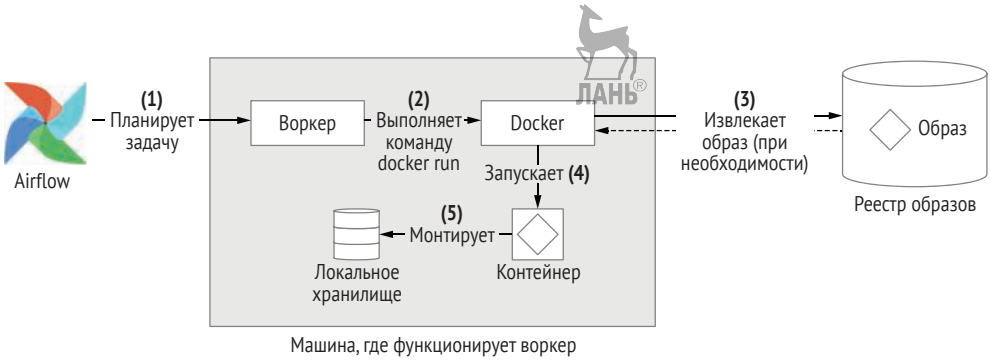


Рис. 10.7 Иллюстрация того, что происходит, когда задача выполняется с помощью `DockerOperator`. В реестре образов хранится коллекция образов Docker. Это может быть закрытый реестр (содержащий наши собственные образы) или общедоступный реестр, такой как DockerHub (который используется по умолчанию при извлечении образов). При извлечении образы кешируются локально, поэтому вам нужно делать это только один раз (за исключением обновлений образа)

сти, демон Docker извлекает требуемый образ Docker из реестра (3). Наконец, Docker создает контейнер, в котором запускается образ (4), и монтирует локальный том в контейнер (5). По окончании команды работа контейнера завершается, и `DockerOperator` получает результаты в воркере.

10.4.2 Создание образов для задач

Прежде чем мы сможем запускать задачи с помощью `DockerOperator`, нам нужно создать все необходимые образы Docker для различных задач. Чтобы создать образ для задачи, необходимо точно определить, какое программное обеспечение (и соответствующие зависимости) требуется для выполнения задачи. Как только это станет ясно, мы можем приступить к созданию файла Dockerfile (наряду со вспомогательными файлами) и использовать команду `docker build` для создания необходимого образа.

В качестве примера рассмотрим первую задачу в OAГ рекомендательной системы: задачу извлечения рейтингов (рис. 10.1). Эта задача должна связаться с внешним API для извлечения рейтингов фильмов от пользователей на заданный диапазон дат, чтобы мы могли использовать их в качестве входных данных для модели рекомендательной системы в следующей задаче.

Чтобы иметь возможность запускать этот процесс в контейнере, для начала нужно преобразовать код, который мы написали для извлечения рейтингов в главе 8, в сценарий, который можно легко запустить внутри контейнера. Первым шагом к его созданию является использование небольшого каркаса для создания CLI-сценария на Python, который затем можно заполнить нужными функциями. С помощью популярной библиотеки Python, click[1], такой каркас может выглядеть примерно так, как показано в листинге 10.10.

Листинг 10.10 Каркас для CLI-сценария на базе библиотеки click

```
#!/usr/bin/env python

import logging
import click

logging.basicConfig(level=logging.INFO)

@click.command()
@click.option(
    "--start_date",
    type=click.DateTime(formats=["%Y-%m-%d"]),
    required=True,
    help="Start date for ratings.",
```

Строка, сообщающая Linux, что нужно выполнить сценарий с использованием Python

Настройка журналирования для обратной связи с пользователем

Преобразовывает функцию main в команду CLI

Добавляет параметр в команду CLI с соответствующими типами и аннотациями

[1] Вы, конечно же, также можете использовать встроенную библиотеку `argparse`, но лично нам очень нравится краткость API библиотеки click для создания приложения командной строки.

```
)
@click.option(
    ...
)
@click.option(
    ...
)
...
def main(start_date, ...):
    """CLI script for fetching ratings from the movielens API."""
    ...

if __name__ == "__main__":
    main()
```

Добавляет дополнительные параметры, необходимые для команды

Параметры передаются в качестве ключевых аргументов в функцию main и могут использоваться с этого момента

Способ, которым Python гарантирует, что функция или команда main вызываются при выполнении этого сценария

В этом каркасе мы определяем одну функцию, `main`, которая выполняется при запуске нашего сценария и, следовательно, должна реализовывать нашу функцию извлечения рейтингов. Мы также используем декоратор `click.command` для преобразования функции `main` в команду CLI, которая позаботится об анализе аргументов из командной строки и представлении полезного отзыва пользователю. Декораторы `click.option` используются для того, чтобы сообщить библиотеке click, какие аргументы должен принимать наш интерфейс командной строки и какие типы значений следует ожидать. Приятный момент здесь состоит в том, что click также выполняет за нас анализ и валидацию аргументов, поэтому нам не нужно самостоятельно обрабатывать данный тип логики.

Используя этот каркас, можно приступить к заполнению функции `main` той же логикой, с которой мы начали в главе 8[1].

Листинг 10.11 Сценарий с рейтингами (docker/images/movielens-fetch/scripts/fetch_ratings.py)

```
...
from pathlib import Path

@click.command()
@click.option(...)
...
def main(start_date, end_date, output_path,
         host, user, password, batch_size):
    """CLI-скрипт для извлечения рейтингов из API Movielens."""

    session = requests.Session()
    session.auth = (user, password)

    logging.info("Fetching ratings from %s (user: %s)", host, user)
    ratings = list(
```

Определяем различные аргументы интерфейса командной строки для библиотеки click. Опущено здесь для краткости; полная реализация доступна в примерах кода

Настраивает сеанс requests для выполнения HTTP-запросов с правильными деталями аутентификации

Журналирование используется для обратной связи с пользователем

[1] Это адаптированный код из примера на базе `PythonOperator`, с которого мы начали в главе 8.

```
    _get_ratings(
        session=session,
        host=host,
        start_date=start_date,
        end_date=end_date,
        batch_size=batch_size,
    )
)
logging.info("Retrieved %d ratings!", len(ratings))

output_path = Path(output_path)

output_dir = output_path.parent
output_dir.mkdir(parents=True, exist_ok=True)

logging.info("Writing to %s", output_path)
with output_path.open("w") as file_:
    json.dump(ratings, file_)
```

Использует нашу функцию _get_ratings (опущена для краткости) для извлечения рейтингов с помощью предоставленного сеанса

Проверяет, что выходной каталог существует

Записывает вывод в формате JSON в выходной каталог

Говоря кратко, этот код начинается с настройки сеанса requests для выполнения HTTP-запросов, после чего используется функция `_get_ratings`[1] для извлечения из API рейтингов за определенный период времени. Результатом вызова этой функции является список записей (в виде словарей), который затем записывается в выходной путь в формате JSON. В промежутке мы также используем операторы журналирования, чтобы предоставить пользователю обратную связь.

Теперь, когда у нас есть сценарий, можно приступить к созданию образа Docker. Для этого нам нужно создать файл Dockerfile, который устанавливает зависимости для нашего сценария (click и requests), копирует его в образ и проверяет, находится ли он в PATH[2]. В результате у нас должен получиться примерно следующий файл Dockerfile:

Листинг 10.12 Встраивание сценария с оценками (docker/images/movielens-fetch/Dockerfile)

```
FROM python:3.8-slim
RUN pip install click==7.1.1 requests==2.23.0
COPY scripts/fetch_ratings.py /usr/bin/local/fetch-ratings
RUN chmod +x /usr/bin/local/fetch-ratings
ENV PATH="/usr/local/bin:${PATH}"
```

Устанавливаем необходимые зависимости

Копируем сценарий fetch_ratings и делаем его исполняемым

Убеждаемся, что сценарий находится в PATH (чтобы его можно было запустить без указания полного пути)

Обратите внимание: код предполагает, что мы поместили сценарий `fetch_ratings.py` в каталог scripts рядом с файлом Dockerfile. Наши зависимости устанавливаются путем указания их непосредственно

[1] Функция `_get_ratings` здесь опущена для краткости, но доступна в исходном коде, прилагаемом к книге.

[2] Это сделано для того, чтобы мы могли запустить сценарий с помощью команды fetch-ratings, вместо того чтобы указывать полный путь к нему.

в файле Dockerfile, хотя вместо него также можно использовать файл `requirements.txt`, который копируется в образ перед запуском pip.

Листинг 10.13 Использование файла requirements.txt (docker/images/movielens-fetch-reqs/Dockerfile)

```
COPY requirements.txt /tmp/requirements.txt
RUN pip install -r /tmp/requirements.txt
```

С помощью этого файла Dockerfile мы, наконец, можем создать образ для извлечения рейтингов:

```
$ docker build -t manning-airflow/movielens-fetch .
```

Чтобы протестировать созданный образ, можно попробовать запустить его с помощью команды `docker run`:

```
$ docker run --rm manning-airflow/movielens-fetch fetch-ratings --help
```

Эта команда должна вывести сообщение-справку из нашего сценария, которое выглядит примерно так:


```
Usage: fetch-ratings [OPTIONS]

  CLI script for fetching movie ratings from the movielens API.

Options:
  --start_date [%Y-%m-%d]  Start date for ratings. [required]
  --end_date [%Y-%m-%d]    End date for ratings. [required]
  --output_path FILE       Output file path. [required]
  --host TEXT              Movielens API URL.
  --user TEXT              Movielens API user. [required]
  --password TEXT          Movielens API password. [required]
  --batch_size INTEGER     Batch size for retrieving records.
  --help                   Show this message and exit.
```


Это означает, что теперь у нас есть образ контейнера для нашей первой задачи. Можно использовать аналогичный подход к созданию разных образов для других задач. В зависимости от количества общего кода вы также можете создавать образы, которые используются в задачах, но могут запускаться с разными аргументами или даже с разными сценариями. Как вы все это организуете, зависит от вас.

10.4.3 Создание ОАГ с задачами Docker

Теперь, когда мы знаем, как создавать образы Docker для каждой из наших задач, можно приступить к созданию ОАГ для запуска задач. Процесс создания такого ОАГ на основе Docker относительно прост: нужно только заменить существующие задачи на операторы `DockerOperator` и убедиться, что каждый `DockerOperator` запускает свою задачу с правильными аргументами. Также нужно подумать о том, как

обмениваться данными между задачами, так как файловые системы контейнеров Docker перестанут существовать по истечении срока выполнения задачи.

Начнем с извлечения рейтингов. Первая часть нашего ОАГ – это просто `DockerOperator`, вызывающий сценарий `fetch-ratings` внутри контейнера `manning-airflow/movielensfetch`, который мы создали в предыдущем разделе.

Листинг 10.14 Запуск контейнера fetch (docker/dags/01_docker.py)

```
import datetime as dt


from airflow import DAG
from airflow.providers.docker.operators.docker import DockerOperator


with DAG(
   dag_id="01_docker",
   description="Fetches ratings from the Movielens API using Docker.",
   start_date=dt.datetime(2019, 1, 1),
   end_date=dt.datetime(2019, 1, 3),
   schedule_interval="@daily",
) as dag:
   Fetch
ratings = DockerOperator(
       task_id="fetch_ratings",
       image="manning-airflow/movielens-fetch",
       command=[
           "fetch-ratings",
           "--start_date",
           "{{ds}}",
           "--end_date",
           "{{next_ds}}",
           "--output_path",
           "/data/ratings/{{ds}}.json",
           "--user",
           os.environ["MOVIELENS_USER"],
           "--password",
           os.environ["MOVIELENS_PASSWORD"],
           "--host",
           os.environ["MOVIELENS_HOST"],
       ],
       volumes=["/tmp/airflow/data:/data"],
       network_mode="airflow",
)
```

Сообщаем DockerOperator, чтобы он использовал образ movielens-fetch

Запускаем сценарий fetch-rating в контейнере с необходимыми аргументами

Предоставляем информацию о хосте и аутентификации для нашего API

Монтируем том для хранения данных. Обратите внимание, что этот путь к хосту находится на хосте Docker, а не в контейнере Airflow

Убеждаемся, что контейнер подключен к сети Docker, чтобы он мог получить доступ к API (который работает в той же сети)

При запуске контейнера из оператора убедитесь, что вы включили аргументы, сообщающие оператору, как подключиться к API MovieLens (`host`, `user`, `password`), для какого диапазона дат нужно извлечь рейтинги (`start_date`/`end_date`) и куда их писать (`output_path`).

Мы также сообщаем Docker монтировать путь файловой системы хоста к контейнеру в `/data`, чтобы можно было сохранить полученные рейтинги вне контейнера. Кроме того, мы даем Docker указание запускать контейнер в определенной сети (Docker) под названием Airflow. Это то место, где работает наш контейнер MovieLens API, если вы используете шаблоны `docker-compose` для запуска Airflow[1].

Для нашей второй задачи, ранжирования фильмов, можно использовать аналогичный подход, чтобы создать контейнер Docker для задачи, который затем можно запустить с помощью `DockerOperator`.

Листинг 10.15 Добавляем задачу ранжирования в ОАГ (docker/dags/01_docker.py)

```
rank_movies = DockerOperator(
    task_id="rank_movies",
    image="manning-airflow/movielens-ranking",
    command=[
        "rank-movies",
        "--input_path",
        "/data/ratings/{{ds}}.json",
        "--output_path",
        "/data/rankings/{{ds}}.csv",
    ],
    volumes=["/tmp/airflow/data:/data"],
)
fetch_ratings >> rank_movies
```

Используем образ movielens-ranking

Вызываем сценарий rank-movies с необходимым входным или выходным путем

Здесь вы также можете увидеть одно из существенных преимуществ использования `DockerOperator`: хотя эти задачи выполняют разные функции, интерфейс для запуска задач тот же (за исключением аргументов `command`, которые передаются в контейнер). Таким образом, теперь эта задача выполняет команду `rank-movies` внутри образа `manning-airflow/movielens-rank`, следя за тем, чтобы данные читались и записывались на тот же том hostPath, как и в случае с предыдущей задачей. Это позволяет задаче ранжирования считывать вывод задачи `fetch_ratings` и сохранять ранжированные фильмы в той же структуре каталогов.

Теперь, когда у нас есть первые две задачи[2] в ОАГ, можно попробовать запустить ее из Airflow. Для этого откройте веб-интерфейс Airflow и активируйте ОАГ. Дождавшись завершения работы, вы должны увидеть пару успешных запусков за последние несколько дней (рис. 10.8).

---

[1] Мы не будем здесь подробно рассматривать работу в сети, говоря о Docker, поскольку это небольшая деталь реализации; вам не нужно настраивать сеть, если вы обращаетесь к API в интернете. Если вам интересно, то для знакомства с этой темой воспользуйтесь хорошей книгой по Docker или онлайн-документацией.

[2] Мы оставим третью задачу по загрузке рекомендаций в базу данных в качестве упражнения.

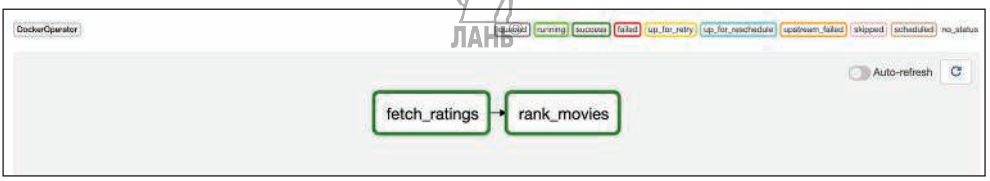


Рис. 10.8 ОАГ на базе Docker в пользовательском интерфейсе Airflow

Результат запуска можно проверить, щелкнув по задаче и открыв журналы, нажав на **View logs** (Просмотр журналов). В случае с задачей `fetch_ratings` вы должны увидеть что-то наподобие вывода, показанного в следующем листинге. Здесь видно, что `DockerOperator` запустил наш образ и зарегистрировал выходные журналы из контейнера.

Листинг 10.16 Вывод журнала из задачи fetch_ratings

```
[2020-04-13 11:32:56,780] {docker.py:194} INFO -
➥ Starting docker container from image manning-airflow/movielens-fetch
[2020-04-13 11:32:58,214] {docker.py:244} INFO -
➥ INFO:root:Fetching ratings from http://movielens:5000 (user: airflow)
[2020-04-13 11:33:01,977] {docker.py:244} INFO -
➥ INFO:root:Retrieved 3299 ratings!
[2020-04-13 11:33:01,979] {docker.py:244} INFO -
➥ INFO:root:Writing to /data/ratings/2020-04-12.json
```

Вы также можете проверить выходные файлы из ОАГ, просмотрев выходные файлы, которые (в нашем примере) были записаны в каталог `/tmp/airflow/data` на хосте Docker.

Листинг 10.17 Вывод из ОАГ

```
$ head /tmp/airflow/data/rankings/*.csv | head
==> /tmp/airflow/data/rankings/2020-04-10.csv <==
movieId,avg_rating,num_ratings
912,4.833333333333333,6
38159,4.833333333333333,3
48516,4.833333333333333,3
4979,4.75,4
7153,4.75,4
```

10.4.4 Рабочий процесс на базе Docker

Как мы уже видели, рабочий процесс для создания ОАГ с использованием контейнеров Docker немного отличается от того подхода, который мы применяли для других ОАГ. Самое большое отличие заключается в том, что сначала вам нужно создать контейнеры Docker для разных заданий. Таким образом, весь рабочий процесс обычно состоит из нескольких этапов (показано на рис. 10.9):

1. Разработчик создает файл (файлы) Dockerfile для необходимого образа (образов), который устанавливает требуемое программ-

ное обеспечение и зависимости. Разработчик (или процесс непрерывной интеграции и доставки) затем сообщает Docker, что нужно создать образ (образы), используя файл (файлы) Dockerfile.

2. Демон Docker создает соответствующий образ (образы) на компьютере, используемом для разработки (или машине в окружении непрерывной интеграции и доставки).
3. Демон Docker помещает созданный образ (образы) в реестр контейнеров, чтобы предоставить к нему доступ для дальнейшего использования.
4. Разработчик создает ОАГ, используя операторы `DockerOperator`, которые ссылаются на созданный образ (образы).
5. После активации ОАГ Airflow приступает к запуску ОАГ и планированию задач `DockerOperator` для соответствующих запусков.
6. Воркеры Airflow берут задачи `DockerOperator` и извлекают необходимый образ (образы) из реестра контейнеров.
7. Для каждой задачи воркер запускает контейнер с соответствующим образом и аргументами с помощью демона Docker.

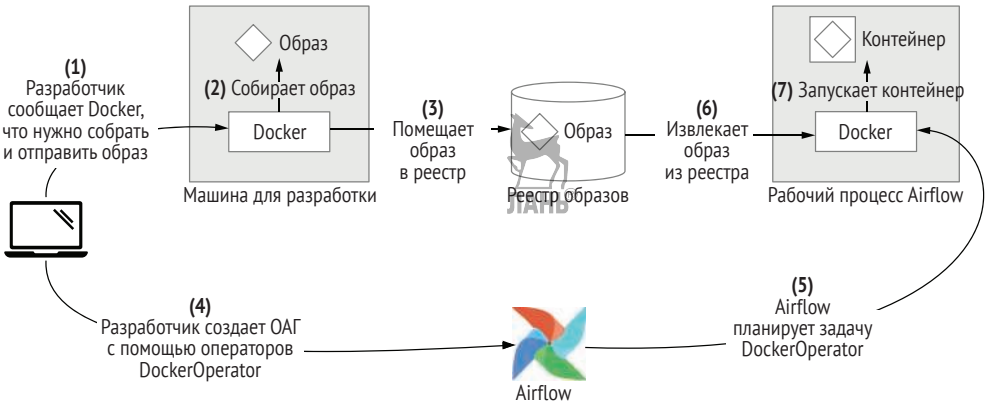


Рис. 10.9 Обычный рабочий процесс для работы с образами Docker в Airflow

Одно из преимуществ данного подхода состоит в том, что, по сути, он разделяет разработку программного обеспечения для запуска задачи, которая теперь хранится внутри образа Docker, от разработки всего ОАГ. Это позволяет разрабатывать образы в рамках их собственного жизненного цикла и тестировать их отдельно от самого ОАГ.

10.5 Запуск задач в Kubernetes

Хотя Docker предоставляет удобный подход для запуска контейнерных задач на одной машине, он не поможет вам с оркестровкой и распределением работы на нескольких машинах, что ограничивает

масштабируемость подхода. Данное ограничение в Docker привело к разработке систем оркестровки контейнеров, таких как Kubernetes, которые помогают масштабировать контейнеризированные приложения в компьютерных кластерах. В этом разделе мы покажем, как запускать контейнеризированные задачи в Kubernetes вместо Docker, и проиллюстрируем преимущества и недостатки использования Kubernetes поверх Docker.

10.5.1 Представляем Kubernetes

Поскольку Kubernetes – сама по себе отдельная тема, мы не будем подробно рассказывать о том, что это такое. Наша цель – дать вам общее представление о том, что он может для вас сделать[1]. Kubernetes – это платформа оркестровки контейнеров с открытым исходным кодом, ориентированная на развертывание, масштабирование и управление контейнеризированными приложениями. По сравнению с Docker, Kubernetes помогает масштабировать контейнеры, управляя их развертыванием на нескольких рабочих узлах, также учитывая такие вещи, как требуемые ресурсы (ЦП и/или память), хранилище и специальные требования к аппаратному обеспечению (например, доступ к графическому процессору) при планировании размещения контейнеров на узлах.

Kubernetes состоит из двух основных компонентов: мастера Kubernetes (или плоскости управления) и узлов (рис. 10.10). Мастер Kubernetes отвечает за запуск множества различных компонентов, включая API-сервер, планировщик и другие службы, отвечающие за управление развертыванием, хранением и т. д. API-сервер Kubernetes используется такими клиентами, как `kubectl` (инструмент командной строки для управления кластерами Kubernetes) или Kubernetes Python SDK для выполнения запросов к Kubernetes и команд для запуска развертываний. Это делает мастер Kubernetes основной контактной точкой для управления контейнеризированными приложениями в кластере Kubernetes.

Рабочие узлы Kubernetes отвечают за запуск контейнерных приложений, назначенных им планировщиком. В Kubernetes эти приложения называются *поды*. Они могут содержать один или несколько контейнеров, которые необходимо запускать вместе на одной машине.

На данный момент все, что вам нужно знать, – это то, что под – это наименьшая рабочая единица внутри Kubernetes. В контексте Airflow каждая задача будет запускаться как контейнер внутри одного пода.

Kubernetes также предоставляет встроенные функции для управления секретами и хранилищем. По сути, это означает, что мы можем,

[1] Чтобы получить полный обзор Kubernetes, рекомендуем вам прочитать исчерпывающую книгу по данной теме, например «Kubernetes в действии» (М.: ДМК Пресс, 2019).

например, запросить объем хранилища у мастера Kubernetes и смонтировать его в качестве постоянного хранилища внутри контейнера. Как таковые эти тома функционируют аналогично монтируемым томам Docker, которые мы видели в предыдущем разделе, но управляются Kubernetes. Это означает, что нам не нужно беспокоиться о том, откуда берется хранилище (если только, конечно, вы не несете ответственности за работу кластера), а можно просто запросить и использовать предоставленный том.

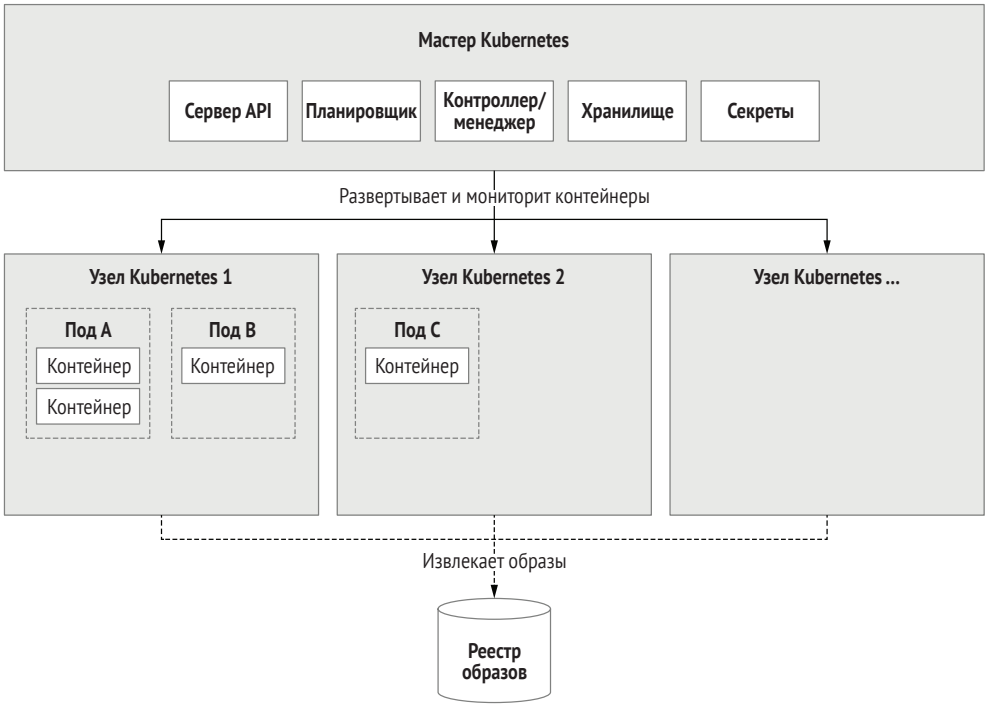


Рис. 10.10 Высокоуровневый обзор Kubernetes

10.5.2 Настройка Kubernetes

Прежде чем подробно заняться настройкой нашего ОАГ для запуска его в Kubernetes, начнем с настройки необходимых нам ресурсов. Во-первых, убедитесь, что у вас есть доступ к кластеру Kubernetes и локально установлен клиент `kubectl`. Самый простой способ получить доступ – установить его локально (используя, например, Docker для Mac/Windows или Minikube) или настроить его в одном из облачных провайдеров.

После того как вы правильно настроили Kubernetes, можно проверить, работает ли он, выполнив команду

```
$ kubectl cluster-info.
```

При использовании Docker в Mac вы должны увидеть что-то наподобие следующего вывода:

```
Kubernetes master is running at https://kubernetes.docker.internal:6443
KubeDNS is running at https://kubernetes.docker.internal:6443/api/v1/
    namespaces/kube-system/services/kube-dns:dns/proxy
```

Если ваш кластер Kubernetes запущен и работает, можно продолжить создание ресурсов. Во-первых, нам нужно создать пространство имен Kubernetes, которое будет содержать все наши ресурсы и поды задач, связанные с Airflow.

Листинг 10.18 Создание пространства имен Kubernetes

```
$ kubectl create namespace airflow
namespace/airflow created
```

Затем мы создадим ресурсы хранилища для нашего OAГ, что позволит нам сохранять результаты наших задач. Эти ресурсы определяются с использованием синтаксиса YAML для Kubernetes.

Листинг 10.19 Спецификация YAML для хранилища (kubernetes/resources/data-volume.yml)

```
apiVersion: v1
kind: PersistentVolume
metadata:
 name: data-volume
 Labels:
   type: local
Spec:
 storageClassName: manual
 capacity:
   storage: 1Gi
 accessModes:
   - ReadWriteOnce
 hostPath:
   path: "/tmp/data"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: data-volume
spec:
 storageClassName: manual
 accessModes:
   - ReadWriteOnce
 resources:
   requests:
     storage: 1Gi
```

Спецификация Kubernetes для определения постоянного тома, виртуального диска, предоставляющего пространство для подов для хранения данных

Имя, которое нужно присвоить тому

Размер тома

Разрешаем доступ для чтения и записи по одному контейнеру за раз

Указываем путь к файлу на хосте, где будет находиться это хранилище

Спецификация Kubernetes для заявки на постоянный том, которая представляет собой резервирование части хранилища в указанном томе

Имя тома, на котором будет запрошено место для хранилища

Разрешенные режимы доступа для заявки на хранилище

Объем хранилища, на который будет подаваться заявка

По сути, эта спецификация определяет два ресурса, используемых для хранилища. Первый — это том Kubernetes, а второй – заявка на хранилище, которая, по сути, сообщает Kubernetes, что нам нужно хранилище, которое будет использоваться для наших контейнеров. Эта заявка может использоваться любыми подами Kubernetes, запускаемыми Airflow, для хранения данных (как будет показано в следующем разделе).

Используя этот YAML, можно создать необходимые ресурсы хранилища.

Листинг 10.20 Развертывание ресурсов храненилища с помощью kubectl

```
$ kubectl --namespace airflow apply -f resources/data-volume.yml
persistentvolumeclaim/data-volume created
persistentvolume/data-volume created
```

Нам также необходимо создать развертывание API MovieLens, к которому мы будем делать запросы, используя наш OAГ. Следующий код позволяет создавать развертывание и служебные ресурсы для API MovieLens, сообщающие Kubernetes, как запустить наш API-сервис.

Листинг 10.21 Спецификация YAML для API (kubernetes/resources/api.yml)

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: movielens-deployment
 labels:
   app: movielens
spec:
 replicas: 1
selector:
  matchLabels:
    app: movielens
template:
  metadata:
    Labels:
      app: movielens
  spec:
    containers:
    - name: movielens
      image: manning-airflow/movielens-api
      ports:
      - containerPort: 5000
      env:
      - name: API_USER
        value: airflow
      - name: API_PASSWORD
```

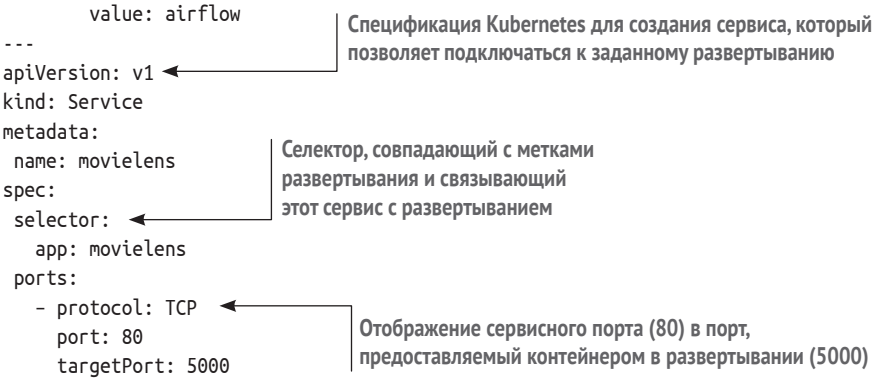
Спецификация Kubernetes для создания развертывания контейнера

Имя развертывания

Ярлыки для развертывания (которые совпадают в сервисе)

Указываем, какие контейнеры следует включить в развертывание, наряду с соответствующими портами, переменными окружения и т. д.

Сообщаем Kubernetes использовать последнюю версию образа movielens-api (последний – это тег образа по умолчанию, используемый Docker/Kubernetes, если не указан конкретный тег версии)



Сервис можно создать таким же образом, как и для ресурсов хранилища.

Листинг 10.22 Развертывание API MovieLens

```
$ kubectl --namespace airflow apply -f resources/api.yml
deployment.apps/movielens-deployment created
service/movielens created
```

Подождав пару секунд, вы должны увидеть поды для API, которые будут подключены к сети:

```
$ kubectl --namespace airflow get pods
NAME                          READY    STATUS    RESTARTS   AGE
movielens-deployment-...      1/1      Running   0          11s
```

Вы можете проверить, работает ли API-сервис:

```
$ kubectl --namespace airflow port-forward svc/movielens 8000:80
```

а затем перейдя по адресу http://localhost: 8000 в браузере. Если все работает правильно, то теперь вы должны увидеть фразу «hello, world» из API в своем браузере.

10.5.3 Использование KubernetesPodOperator

После создания необходимых ресурсов Kubernetes можно приступить к настройке ОАГ рекомендательной системы на базе Docker для использования кластера Kubernetes вместо Docker.

Чтобы запустить задачи в Kubernetes, нужно заменить операторы `DockerOperator` экземплярами `KubernetesPodOperator`, которые доступны из пакета поставщиков `apacheairflow-provider-cncf-kubernetes`[1]. Как следует из названия, `KubernetesPodOperator` запускает

[1] Для Airflow версии 1.10.x можно установить `KubernetesPodOperator`, используя пакет `apache-airflow-backportproviders-cncf-kubernetes backport`.

задачи внутри подов в кластере Kubernetes. Базовый API оператора выглядит следующим образом.

Листинг 10.23 Использование KubernetesPodOperator (kubernetes/dags/02_kubernetes.py)

```
fetch_ratings = KubernetesPodOperator(
        task_id="fetch_ratings",
        image="manning-airflow/movielens-fetch",
        cmds=["fetch-ratings"],
        arguments=[
            "--start_date",
            "{{ds}}",
            "--end_date",
            "{{next_ds}}",
            "--output_path",
            "/data/ratings/{{ds}}.json",
            "--user",
            os.environ["MOVIELENS_USER"],
            "--password",
            os.environ["MOVIELENS_PASSWORD"],
            "--host",
            os.environ["MOVIELENS_HOST"],
        ],
        namespace="airflow",
        name="fetch-ratings",
        cluster_context="docker-desktop",
        in_cluster=False,
        volumes=[volume],
        volume_mounts=[volume_mount],
        image_pull_policy="Never",
        is_delete_operator_pod=True,
    )
```

Какой образ использовать

Исполняемый файл для запуска внутри контейнера

Аргументы для передачи исполняемому файлу (здесь указаны отдельно, в отличие от DockerOperator)

Название, которое будет использоваться для пода

Пространство имен Kubernetes, которое будет использоваться для запуска пода

Имя используемого кластера (если у вас зарегистрировано несколько кластеров Kubernetes)

Указывает на то, что мы не запускаем Airflow внутри Kubernetes

Аргументы volume и volume_mounts для использования в поде

Указываем политику извлечения образов, при которой Airflow должен использовать наши локально созданные образы, а не пытаться извлекать их из Docker Hub

Автоматически удаляет поды, когда они заканчивают работу

Как и в случае с `DockerOperator`, первые несколько аргументов сообщают `KubernetesPodOperator`, как запустить задачу в качестве контейнера: аргумент `image` говорит Kubernetes, какой образ Docker использовать, а параметры `cmds` и `arguments` определяют, какой исполняемый файл запускать (`fetch-rating`) и какие аргументы передать исполняемому файлу. Остальные аргументы сообщают Kubernetes, какой кластер использовать (`cluster_context`), в каком пространстве имен запускать под (`namespace`) и какое имя применять для контейнера (`name`).

Мы также предоставляем два дополнительных аргумента: `volume` и `volume_mounts`, которые определяют, как тома, которые мы создали в предыдущем разделе, должны монтироваться в задачи в поде Kubernetes. Эти конфигурационные значения создаются с использованием двух классов из Kubernetes Python SDK: `V1Volume` и `V1VolumeMount`.

Листинг 10.24 Аргументы volume и volume_mounts (kubernetes/dags/02_kubernetes.py)

```
from kubernetes.client import models as k8s

...

volume_claim = k8s.V1PersistentVolumeClaimVolumeSource(
   claim_name="data-volume"
)
volume = k8s.V1Volume(
   name="data-volume",
   persistent_volume_claim=volume_claim
)
volume_mount = k8s.V1VolumeMount(
   name="data-volume",
   mount_path="/data",
   sub_path=None,
   read_only=False,
)
```

Ссылки на ранее созданный том хранилища и заявку

Куда монтировать том

Монтируем том как доступный для записи

Здесь мы сначала создаем конфигурационный объект `V1Volume`, ссылающийся на заявку `data-volume`, которую мы создали в качестве ресурса Kubernetes в предыдущем разделе. Затем мы создаем конфигурационный объект `V1VolumeMount`, который относится к только что созданной конфигурации тома (`data-volume`), и указываем, где нужно монтировать этот том в контейнере пода. Эти объекты потом можно передать операторам `KubernetesPodOperator`, используя аргументы `volumes` и `volume_mounts`.

Теперь осталось только создать вторую задачу для ранжирования фильмов.

Листинг 10.25 Добавляем задачу ранжирования фильмов (kubernetes/dags/02_kubernetes.py)

```
rank_movies = KubernetesPodOperator(
    task_id="rank_movies",
    image="manning-airflow/movielens-rank",
    cmds=["rank-movies"],
    arguments=[
        "--input_path",
        "/data/ratings/{{ds}}.json",
        "--output_path",
        "/data/rankings/{{ds}}.csv",
    ],
    namespace="airflow",
    name="fetch-ratings",
    cluster_context="docker-desktop",
    in_cluster=False,
    volumes=[volume],
       volume_mounts=[volume_mount],
```

```
        image_pull_policy="Never",
        is_delete_operator_pod=True,
    )
```

Затем связываем все это вместе в окончательный ОАГ.

Листинг 10.26 Реализация ОАГ (kubernetes/dags/02_kubernetes.py)

```
import datetime as dt
import os

from kubernetes.client import models as k8s

from airflow import DAG
from airflow.providers.cncf.kubernetes.operators.kubernetes_pod import (
    KubernetesPodOperator,
)

with DAG(
    dag_id="02_kubernetes",
    description="Fetches ratings from the Movielens API using kubernetes.",
    start_date=dt.datetime(2019, 1, 1),
    end_date=dt.datetime(2019, 1, 3),
    schedule_interval="@daily",
) as dag:
    volume_claim = k8s.V1PersistentVolumeClaimVolumeSource(...)
    volume = k8s.V1Volume(...)
    volume_mount = k8s.V1VolumeMount(...)

    fetch_ratings = KubernetesPodOperator(...)
    rank_movies = KubernetesPodOperator(...)

    fetch_ratings >> rank_movies
```

После окончания можно перейти к запуску ОАГ, активировав его в веб-интерфейсе Airflow. Подождав несколько секунд, мы должны увидеть, как Airflow начинает планировать и запускать задачи (рис. 10.11). Для получения более подробной информации можно открыть журнал экземпляра отдельной задачи, щелкнув по задаче, а затем щелкнув **View Logs** (Просмотр журналов). Вы увидите вывод задачи, который должен быть примерно таким.

Рис. 10.11 Несколько успешных запусков ОАГ рекомендательной системы на базе `KubernetesPodOperator`

Листинг 10.27 Журналы задачи fetch_ratings на базе Kubernetes

```
...
[2020-04-13 20:28:45,067] {logging_mixin.py:95} INFO -
➥ [[34m2020-04-13 20:28:45,067[0m] {[34mpod_launcher.py:[0m122}
➥ INFO[0m - Event: [1mfetch-ratings-0a31c089[0m had an event
➥ of type [1mPending[0m[0m
[2020-04-13 20:28:46,072] {logging_mixin.py:95} INFO -
➥ [[34m2020-04-13 20:28:46,072[0m] {[34mpod_launcher.py:[0m122}
➥ INFO[0m - Event: [1mfetch-ratings-0a31c089[0m had an event
➥ of type [1mRunning[0m[0m
[2020-04-13 20:28:48,926] {logging_mixin.py:95} INFO -
➥ [[34m2020-04-13 20:28:48,926[0m] {[34mpod_launcher.py:[0m105}
➥ INFO[0m - b'Fetching ratings from
➥ http://movielens.airflow.svc.cluster.local:80 (user: airflow)\n'[0m
[2020-04-13 20:28:48,926] {logging_mixin.py:95} INFO -
➥ [[34m2020-04-13 20:28:48,926[0m] {[34mpod_launcher.py:[0m105}
➥ INFO[0m - b'Retrieved 3372 ratings!\n'[0m
[2020-04-13 20:28:48,927] {logging_mixin.py:95} INFO -
➥ [[34m2020-04-13 20:28:48,927[0m] {[34mpod_launcher.py:[0m105}
➥ INFO[0m - b'Writing to /data/ratings/2020-04-10.json\n'[0m
[2020-04-13 20:28:49,958] {logging_mixin.py:95} INFO -
➥ [[34m2020-04-13 20:28:49,958[0m] {[34mpod_launcher.py:[0m122}
➥ INFO[0m - Event: [1mfetch-ratings-0a31c089[0m had an event
➥ of type [1mSucceeded[0m[0m
...
```

10.5.4 Диагностика проблем, связанных с Kubernetes

Если вам не повезло, то может случиться так, что ваши задачи зависнут в состоянии выполнения, вместо того чтобы завершиться правильно. Обычно это происходит из-за того, что Kubernetes не может запланировать под задач. Это означает, что под зависнет в состоянии ожидания, вместо того чтобы запускаться в кластере. Чтобы проверить, так ли это на самом деле, можно посмотреть журналы соответствующей задачи (задач), которые могут предоставить вам дополнительную информацию о состоянии подов в кластере.

Листинг 10.28 Вывод журнала, показывающий, что задача зависла в состоянии ожидания

```
[2020-04-13 20:27:01,301] {logging_mixin.py:95} INFO -
➥ [[34m2020-04-13 20:27:01,301[0m] {[34mpod_launcher.py:[0m122}
➥ INFO[0m - Event: [1mfetch-ratings-0a31c089[0m had an event of type
➥ [1mPending[0m[0m
[2020-04-13 20:27:02,308] {logging_mixin.py:95} INFO -
➥ [[34m2020-04-13 20:27:02,308[0m] {[34mpod_launcher.py:[0m122}
➥ INFO[0m - Event: [1mfetch-ratings-0a31c089[0m had an event
➥ of type [1mPending[0m[0m
[2020-04-13 20:27:03,315] {logging_mixin.py:95} INFO -
➥ [[34m2020-04-13 20:27:03,315[0m] {[34mpod_launcher.py:[0m122}
```

```
➥ INFO[0m - Event: [1mfetch-ratings-0a31c089[0m had an event
➥ of type [1mPending[0m[0m
...
```

Здесь видно, что поды и в самом деле зависли.

Чтобы диагностировать проблему, можно найти поды задач:

```
$ kubectl --namespace airflow get pods
```

После того как вы определили имя соответствующего пода, можно попросить Kubernetes предоставить дополнительные сведения о состоянии пода, используя подкоманду `describe` из `kubectl`.

Листинг 10.29 Описание конкретного пода для выявления проблем

```
$ kubectl --namespace describe pod [NAME-OF-POD]
...
Events:
  Type Reason                Age   From                Message
  ---- ------                ----  ----                -------
Warning FailedScheduling     82s   default-scheduler   persistentvolumeclaim
➥ "data-volume" not found
```

Эта команда выдает большое количество деталей о соответствующем поде, включая недавние события (в разделе `Events`). Здесь видно, что наш под не был запланирован, потому что заявка на постоянный том не была создана должным образом.

Чтобы исправить это, можно попробовать исправить ресурсы, должным образом применив нашу спецификацию ресурсов (что мы, вероятно, забыли сделать), а затем выполнить проверку на предмет наличия новых событий.

Листинг 10.30 Устранение проблемы путем создания недостающих ресурсов

```
$ kubectl --namespace airflow apply -f resources/data-volume.yml
persistentvolumeclaim/data-volume created
persistentvolume/data-volume created

$ kubectl --namespace describe pod [NAME-OF-POD]
...
Events:
  Type     Reason           Age    From               Message
  ----     ------           ----   ----               ------
  Warning FailedScheduling  33s    default-scheduler  persistentvolumeclaim
➥ "data-volume" not found
Warning FailedScheduling     6s     default-scheduler  pod has unbound
➥ immediate PersistentVolumeClaims
Normal Scheduled             3s     default-scheduler  Successfully assigned
➥ airflow/fetch-ratings-0a31c089 to docker-desktop
Normal Pulled                2s      kubelet, ...       Container image
```

```
➥ "manning-airflow/movielens-fetch" already present on machine
Normal Created              2s     kubelet, ...       Created container base
Normal Started              2s     kubelet, ...       Started container base
```

Данный код показывает, что Kubernetes действительно смог запланировать наш под после создания заявки на том, тем самым исправив предыдущую проблему.

ПРИМЕЧАНИЕ В целом мы рекомендуем начинать диагностику любых проблем, сначала проверив журналы Airflow на предмет наличия полезных отзывов. Если вы видите что-либо, напоминающее проблемы с планированием, kubectl – лучшее, на что вы можете надеяться при выявлении проблем с вашим кластером или конфигурацией Kubernetes.

Хотя этот пример далеко не исчерпывающий, надеемся, он даст вам некоторое представление о подходах, которые вы можете использовать для отладки проблем, связанных с Kubernetes, при использовании `KubernetesPodOperator`.

10.5.5 Отличия от рабочих процессов на базе Docker

Рабочий процесс на базе Kubernetes (рис. 10.12) относительно похож на рабочий процесс на базе Docker (рис. 10.9). Однако помимо необходимости заниматься настройкой и сопровождением кластера Kubernetes (что не обязательно может быть тривиально) стоит иметь в виду и другие отличия.

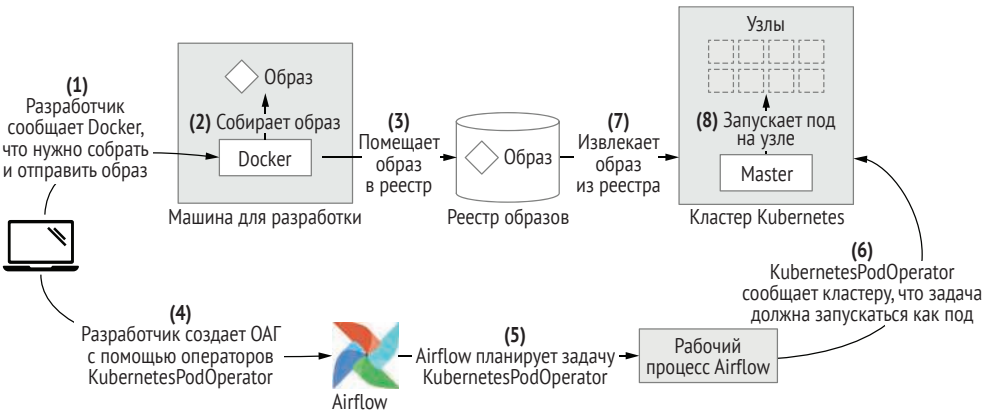


Рис. 10.12 Рабочий процесс для создания ОАГ с помощью KubernetesPodOperator

Во-первых, контейнеры задач уже выполняются не на рабочем узле Airflow, а на отдельном узле (Kubernetes) в кластере Kubernetes. Это означает, что любые ресурсы, используемые на рабочем узле, довольно минимальны, и вы можете использовать функции Kubernetes,

чтобы убедиться, что ваша задача развернута на узле с правильными ресурсами (например, ЦП, память, графический процессор).

Во-вторых, ни одно хранилище больше не будет доступно из воркера Airflow, но оно должно быть доступно для пода Kubernetes. Обычно это означает использование предоставленного хранилища через Kubernetes (как было показано с томами Kubernetes и заявками на хранилище); однако вы также можете применять различные типы сетевого или облачного хранилища, если у пода есть соответствующий доступ к хранилищу.

В целом Kubernetes предоставляет значительные преимущества по сравнению с Docker, особенно в отношении того, что касается масштабируемости, гибкости (например, предоставления разных ресурсов или узлов для разных рабочих нагрузок) и управления другими ресурсами, такими как хранилище, секреты и т. д. Кроме того, сам Airflow можно запускать поверх Kubernetes, а это значит, что вы можете использовать всю настройку Airflow в единой масштабируемой инфраструктуре на основе контейнеров.

Резюме

- Развертываниями Airflow может быть сложно управлять, если они связаны со множеством различных операторов, так как это требует знания различных API и усложняет отладку и управление зависимостями.
- Один из способов решения этой проблемы – использовать систему управления контейнерами, такую как Docker, чтобы инкапсулировать задачи в образы контейнеров и запускать эти образы изнутри Airflow.
- У такого подхода есть ряд преимуществ, в том числе более простое управление зависимостями, более унифицированный интерфейс для запуска задач и улучшенная тестируемость задач.
- Применяя `DockerOperator`, можно запускать задачи в образах контейнеров напрямую, используя Docker, что аналогично команде `docker run`.
- Вы можете воспользоваться `KubernetesPodOperator` для запуска контейнеризированных задач в подах на кластере Kubernetes.
- Kubernetes позволяет масштабировать контейнеризированные задачи в вычислительном кластере, что обеспечивает (помимо прочего) бóльшую масштабируемость и гибкость с точки зрения вычислительных ресурсов.

Часть III

Airflow на практике

Теперь, когда вы узнали, как создавать сложные конвейеры, используем их в промышленном окружении! Чтобы помочь вам приступить к работе, в этой части обсуждается несколько тем, касающихся использования Airflow в промышленном окружении. Сначала, в главе 11, дается обзор методов, которые мы уже встречали, для реализации конвейеров, и особое внимание уделяется передовым практикам, которые должны помочь вам в создании эффективных конвейеров, удобных в сопровождении. В главах 12 и 13 рассматриваются детали, которые следует учитывать при запуске Airflow в промышленном окружении. В главе 12 описано, как развернуть Airflow. Здесь мы коснемся таких тем, как архитектуры масштабирования Airflow, мониторинг, журналирование и оповещение. В главе 13 особое внимание уделяется обеспечению безопасности Airflow, чтобы избежать нежелательного доступа и минимизировать последствия брешей в системе безопасности. В главе 14 все предыдущие главы объединяются в общий пример. После завершения этой части вы сможете писать эффективные и удобные в сопровождении конвейеры. У вас также должно быть хорошее представление о том, как развертывать Airflow и какие детали реализации следует учитывать для надежного и безопасного развертывания.

11 Лучшие практики

Эта глава рассказывает:

- о том, как писать чистые и понятные ОАГ, используя соглашения о стилях;
- об использовании последовательных подходов к управлению учетными данными и параметрами конфигурации;
- о создании повторяющихся ОАГ и задач с помощью фабричных функций;
- о проектировании воспроизводимых задач путем наложения ограничений идемпотентности и детерминизма;
- об эффективной обработке данных путем ограничения количества данных, обрабатываемых в ОАГ;
- об использовании эффективных подходов к обработке и хранению наборов (промежуточных) данных;
- об управлении параллелизмом с использованием пулов ресурсов.

В предыдущих главах мы описали большинство основных элементов, которые входят в построение и проектирование процессов обработки данных с использованием ОАГ. В этой главе мы подробнее рассмотрим передовые практики, которые могут помочь вам в написании хорошо спроектированных ОАГ, которые просты для понимания и эффективны с точки зрения того, как они обрабатывают ваши данные и ресурсы.

11.1 Написание чистых ОАГ

Написание ОАГ легко может превратиться в путаницу. Например, код ОАГ может быстро стать чрезмерно сложным или трудным для чтения – особенно если его пишут члены команды с очень разными стилями программирования. В этом разделе мы коснемся советов, которые помогут вам структурировать и стилизовать код ОАГ и, мы надеемся, внесут (часто необходимую) ясность в сложные процессы обработки данных.

11.1.1 Используйте соглашения о стилях

Как и во всех упражнениях по программированию, один из первых шагов к написанию чистых и последовательных ОАГ – принятие общего и чистого стиля программирования и его последовательное применение. Хотя тщательное изучение практик по написанию чистого кода выходит за рамки этой книги, мы можем дать несколько советов в качестве отправных точек.

Следуем руководству по стилям

Самый простой способ сделать свой код чище и понятнее – использовать общепринятый стиль при написании кода. В сообществе доступно несколько руководств по стилю, в том числе широко известное руководство PEP8 (https://www.python.org/dev/peps/pep-0008/) и руководства от таких компаний, как Google (https://google.github.io/styleguide/pyguide.html). Как правило, они включают рекомендации по отступам, максимальной длине строк, стилям именования для переменных, классов, функций и т. д.

Листинг 11.1 Примеры кода, несовместимого с PEP8

```
spam( ham[ 1 ], { eggs: 2 } )

i=i+1
submitted +=1

my_list = [
    1, 2, 3,
    4, 5, 6,
    ]
```

Листинг 11.2 Приведение примеров из листинга 11.1 в соответствии с PEP-8

```
spam(ham[1], {eggs: 2})        <-- Меньше ненужных пробелов

i = i + 1                      | Согласованные пробелы
submitted += 1                 | вокруг операторов
```

```
my_list = [    <-- Более читабельные отступы вокруг скобок списка
    1, 2, 3,
    4, 5, 6,
]
```

Использование статических анализаторов для проверки качества кода

Сообщество Python также создало множество программных инструментов, которые можно использовать для проверки соответствия вашего кода соглашениям о кодировании и/или стилях.

Два популярных инструмента – это Pylint и Flake8. Они используются в качестве статических анализаторов кода, а это означает, что вы можете запускать их для проверки своего кода, чтобы получить отчет о том, соответствует ли ваш код их предусмотренным стандартам.

Например, чтобы запустить Flake8, можно установить его с помощью pip, а затем указать соответствующий путь.

Листинг 11.3 Установка и запуск Flake8

```
python -m pip install flake8
python -m flake8 dags/*.py
```

Эта команда запускает Flake8 для всех файлов Python в папке ОАГ, предоставляя отчет о воспринимаемом качестве кода этих файлов. Обычно отчет выглядит примерно так.

Листинг 11.4 Пример вывода Flake8

```
$ python -m flake8 chapter08/dags/
chapter08/dags/04_sensor.py:2:1: F401
➥ 'airflow.operators.python.PythonOperator' imported but unused
chapter08/dags/03_operator.py:2:1: F401
➥ 'airflow.operators.python.PythonOperator' imported but unused
```

И Flake8, и Pylint широко используются в сообществе, хотя обычно считается, что в конфигурации Pylint по умолчанию имеется более обширный набор проверок[1]. Конечно, оба инструмента можно настроить для включения или отключения определенных проверок, в зависимости от предпочтений, и их можно сочетать для предоставления исчерпывающей обратной связи. Чтобы получить более подробную информацию, посетите соответствующие сайты обоих инструментов.

Использование инструментов для форматирования кода с целью обеспечения общепринятого форматирования

Хотя статические анализаторы кода предоставляют обратную связь о качестве вашего кода, такие инструменты, как Pylint или Flake8, не

[1] Можно считать это сильной или слабой стороной Pylint, в зависимости от предпочтений, поскольку некоторые находят его слишком педантичным.

предъявляют слишком строгих требований к его форматированию (например, когда начинать новую строку, как делать отступы в заголовках функций и т. д.). Таким образом, код Python, написанный разными людьми, может иметь самые разные стили форматирования в зависимости от предпочтений автора.

Один из подходов к уменьшению неоднородности форматирования кода внутри команд – использовать инструмент для форматирования кода, чтобы передать ему управление (и все заботы), дабы обеспечить повторное форматирование кода в соответствии с его рекомендациями. Таким образом, применяя его на протяжении всего проекта, вы гарантируете, что весь код будет следовать единому согласованному стилю форматирования: стилю, реализуемому данным инструментом.

Два часто используемых инструмента для форматирования кода Python – это YAPF (https://github.com/google/yapf) и Black (https://github.com/psf/black). Оба они используют похожий стиль: берут ваш код, написанный на Python, и повторно форматируют его в соответствии со своими стилями, с небольшими различиями. Таким образом, выбор между Black и YAPF может зависеть от личных предпочтений, хотя Black приобрел большую популярность в сообществе Python за последние несколько лет.

Чтобы показать небольшой пример, рассмотрим следующий (надуманный) пример некрасивой функции.

Листинг 11.5 Пример кода до форматирования с использованием Black

```
def my_function(
    arg1, arg2,
    arg3):
    """Function to demonstrate black."""
    str_a = 'abc'
    str_b = "def"
    return str_a + \
        str_b
```

После применения Black вы получите следующий (более чистый) результат.

Листинг 11.6 Тот же код после форматирования с использованием Black

```
def my_function(arg1, arg2, arg3):  <-- Более последовательный отступ для аргументов
    """Function to demonstrate black."""
    str_a = "abc"                   <-- Последовательное использование
    str_b = "def"                       двойных кавычек
    return str_a + str_b            <-- Удален ненужный разрыв строки
```

Чтобы запустить Black самостоятельно, установите его с помощью pip и примените к своим файлам Python:

Листинг 11.7 Установка и запуск Black

```
python -m pip install black
python -m black dags/
```

Вы должны получить что-то наподобие следующего вывода, показывающего, переформатировал ли Black какие-либо файлы Python.

Листинг 11.8 Пример вывода

```
reformatted dags/example_dag.py
All done! ✨ 🍰 ✨
1 file reformatted.
```

Обратите внимание, что вы также можете выполнить пробный запуск Black, используя параметр `--check`, который заставляет Black указывать только то, будет ли он переформатировать какие-либо файлы, вместо того чтобы выполнять фактическое переформатирование.

Многие редакторы (например, Visual Studio Code, Pycharm) поддерживают интеграцию с этими инструментами, позволяя переформатировать код из редактора. Подробнее о том, как настроить такой тип интеграции, *см.* в документации соответствующего редактора.

Соглашения о стилях, относящихся к Airflow

Также неплохо договориться о соглашениях по стилям для кода Airflow, особенно в случаях, когда он предоставляет несколько способов достижения одинаковых результатов. Например, у него есть два разных стиля для определения ОАГ.

Листинг 11.9 Два стиля для определения DAG

```
with DAG(...) as dag:                    <-- Использование диспетчера контекста
    task1 = PythonOperator(...)
    task2 = PythonOperator(...)

dag = DAG(...)                           <-- Без диспетчера контекста
task1 = PythonOperator(..., dag=dag)
task2 = PythonOperator(..., dag=dag)
```

В принципе, оба этих определения ОАГ делают одно и то же, а это означает, что нет реальной причины выбирать что-то одно, помимо стилевых предпочтений. Однако если говорить о членах вашей команды, то может быть полезно выбрать один из двух стилей и следовать им в кодовой базе, чтобы все было более последовательным и понятным.

Такая согласованность еще более важна при определении зависимостей между задачами, поскольку Airflow предоставляет несколько различных способов определения одной и той же зависимости задачи.

Листинг 11.10 Различные стили для определения зависимостей задач

```
task1 >> task2
task1 << task2
[task1] >> task2
task1.set_downstream(task2)
task2.set_upstream(task1)
```

Хотя у этих разных определений есть свои достоинства, сочетание разных стилей определений зависимостей в одном ОАГ может сбивать с толку.

Листинг 11.11 Смешивание различных нотаций зависимостей задач

```
task1 >> task2
task2 << task3
task5.set_upstream(task3)
task3.set_downstream(task4)
```

Таким образом, ваш код, как правило, будет более читабельным, если вы будете придерживаться единого стиля для определения зависимостей между задачами.

Листинг 11.12 Использование единого стиля для определения зависимостей задач

```
task1 >> task2 >> task3 >> [task4, task5]
```

Как и прежде, не обязательно отдавать четкое предпочтение какому-либо конкретному стилю; просто убедитесь, что вы выбрали тот стиль, который нравится вам (и вашей команде), и последовательно применяйте его.

11.1.2 Централизованное управление учетными данными

В ОАГ, которые взаимодействуют со множеством различных систем, вы можете столкнуться с манипуляциями с различными типами учетных данных – базами данных, вычислительными кластерами, облачным хранилищем и т. д. Как было показано в предыдущих главах, Airflow позволяет сохранять эти учетные данные в своем хранилище подключений, что обеспечивает безопасное[1] хранение ваших учетных данных в центральной базе данных.

Хотя хранилище подключений – это самое простое место для хранения учетных данных для встроенных операторов, у вас может возникнуть соблазн хранить секреты ваших пользовательских функций `PythonOperator` (и других функций) в менее безопасных местах для

[1] Предполагая, что Airflow сконфигурирован с учетом стандартов безопасности. *См.* главы 12 и 13 для получения дополнительной информации о настройке развертываний и безопасности в Airflow.

облегчения доступа. Например, мы видели довольно много реализаций ОАГ с ключами безопасности, вшитыми в код самого ОАГ или во внешних файлах конфигурации.

К счастью, относительно легко использовать хранилище подключений Airflow для сохранения учетных данных и для вашего пользовательского кода, извлекая сведения о подключении из хранилища в вашем коде и используя полученные учетные данные для выполнения своей работы.

Листинг 11.13 Извлечение учетных данных из базы метаданных Airflow

```
from airflow.hooks.base_hook import BaseHook

def _fetch_data(conn_id, **context)
    credentials = BaseHook.get_connection(conn_id)
    ...

fetch_data = PythonOperator(
    task_id="fetch_data",
    op_kwargs={"conn_id": "my_conn_id"},
    dag=dag
)
```

Извлечение учетных данных с использованием данного идентификатора

Преимущество такого подхода заключается в том, что он использует тот же метод хранения учетных данных, что и все другие операторы Airflow, а это означает, что управление учетными данными осуществляется в одном месте. Как следствие вам нужно волноваться только о защите и сохранении учетных данных в этой центральной базе данных.

Конечно, в зависимости от развертывания вам может понадобиться хранить свои секреты в других внешних системах (например, секреты Kubernetes, облачные хранилища секретов), перед тем как передать их Airflow. В этом случае рекомендуется убедиться, что эти учетные данные передаются в Airflow (например, с использованием переменных окружения), и вы обращаетесь к ним с помощью хранилища подключений Airflow.

11.1.3 Единообразно указывайте детали конфигурации

У вас могут быть другие параметры, которые нужно передать в качестве конфигурации своему ОАГ, такие как пути к файлам, имена таблиц и т. д. Поскольку они написаны на языке Python, ОАГ предоставляют множество различных вариантов конфигурации, в том числе глобальные переменные (в рамках ОАГ), конфигурационные файлы (например, YAML, INI, JSON), переменные окружения, модули конфигурации на базе Python и т. д. Airflow также позволяет хранить конфигурации в базе метаданных, используя Airflow Variables (https://airflow.apache.org/docs/apache-airflow/stable/concepts/index.html).

Например, чтобы загрузить параметры конфигурации из файла YAML[1], можно использовать что-то вроде этого:

Листинг 11.14 Загрузка параметров конфигурации из файла YAML

```
import yaml

with open("config.yaml") as config_file:
    config = yaml.load(config_file)
...
fetch_data = PythonOperator(
    task_id="fetch_data",
    op_kwargs={
        "input_path": config["input_path"],
        "output_path": config["output_path"],
    },
    ...
)
```

Чтение файла конфигурации с помощью PyYAML

Листинг 11.15 Пример файла конфигурации YAML

```
input_path: /data
output_path: /output
```

Точно так же вы можете загрузить конфигурацию с помощью Airflow Variables. По сути, это функция Airflow для хранения (глобальных) переменных в базе метаданных[2].

Листинг 11.16 Сохранение параметров конфигурации в Airflow Variables

```
from airflow.models import Variable

input_path = Variable.get("dag1_input_path")
output_path = Variable.get("dag1_output_path")

fetch_data = PythonOperator(
    task_id="fetch_data",
    op_kwargs={
        "input_path": input_path,
        "output_path": output_path,
    },
    ...
)
```

Извлечение глобальных переменных с помощью механизма Airflow Variables

[1] Обратите внимание: вы должны быть осторожны и не хранить конфиденциальные данные в таких файлах конфигурации, поскольку, как правило, они хранятся в виде обычного текста. Если вы храните их в конфигурационных файлах, убедитесь, что только нужные люди имеют полномочия для доступа к файлу. В противном случае рассмотрите возможность хранения этих данных в более безопасных местах, таких как база метаданных Airflow.

[2] Обратите внимание, что извлечение таких переменных в глобальной области видимости вашего OAG обычно плохо сказывается на его производительности. Прочитайте следующий подраздел, чтобы узнать, почему.

Обратите внимание, что такое извлечение переменных в глобальной области видимости может быть плохой идеей, поскольку это означает, что Airflow будет извлекать их из базы данных каждый раз, когда планировщик читает определение ОАГ.

В целом у нас нет никаких реальных предпочтений в отношении того, как хранить свою конфигурацию, пока вы последовательны в этом. Например, если вы храните свою конфигурацию для одного ОАГ в виде файла YAML, имеет смысл следовать тому же соглашению и в отношении других ОАГ.

Что касается конфигурации, которая используется в нескольких ОАГ, настоятельно рекомендуется указывать значения конфигурации в одном месте (например, в общем файле YAML), следуя принципу DRY (Don't repeat yourself – Не повторяйся). Таким образом, вы с меньшей вероятностью столкнетесь с проблемами, если измените параметр конфигурации в одном месте, но при этом забудете сделать это в другом месте.

Наконец, полезно понимать, что параметры конфигурации можно загружать в разные контексты в зависимости от того, где на них ссылаются в рамках ОАГ. Например, если вы загружаете файл конфигурации в основную часть ОАГ следующим образом:

Листинг 11.17 Загрузка параметров конфигурации в определение ОАГ (неэффективно)

```
import yaml

with open("config.yaml") as config_file:
    config = yaml.load(config_file)

fetch_data = PythonOperator(...)
```

В глобальной области видимости эта конфигурация будет загружена в планировщике

Файл `config.yaml` загружается из локальной файловой системы компьютеров, на которых запущены веб-сервер и/или планировщик Airflow. Это означает, что оба этих компьютера должны иметь доступ к пути к файлу конфигурации. Напротив, вы также можете загрузить файл конфигурации как часть задачи (Python).

Листинг 11.18 Загрузка параметров конфигурации внутри задачи (более эффективно)

```
import yaml

def _fetch_data(config_path, **context):
    with open(config_path) as config_file:
        config = yaml.load(config_file)
    ...

fetch_data = PythonOperator(
    op_kwargs={"config_path": "config.yaml"},
    ...
)
```

В области видимости задачи эта конфигурация будет загружена в воркер

В данном случае файл конфигурации не будет загружен, пока ваша функция не будет выполнена воркером Airflow. Это означает, что конфигурация загружается в контексте воркера. В зависимости от того, как вы настроили развертывание, это может быть совсем другое окружение (с доступом к другим файловым системам и т. д.), что приводит к ошибочным результатам или сбоям. Подобные ситуации могут возникнуть и при других подходах к настройке.

Их можно избежать, выбрав один подход к конфигурации, который хорошо работает, и придерживаясь его в ОАГ. Кроме того, при загрузке параметров конфигурации помните о том, где выполняются различные части вашего ОАГ, и желательно используйте подходы, доступные для всех компонентов Airflow (например, нелокальные файловые системы и т. д.).

11.1.4 Избегайте вычислений в определении ОАГ

ОАГ написаны на Python, что обеспечивает значительную гибкость. Однако недостаток такого подхода состоит в том, что Airflow необходимо выполнить файл Python, чтобы получить соответствующий ОАГ. Более того, чтобы улавливать любые изменения, которые вы могли внести в ОАГ, Airflow должен перечитывать файл через регулярные промежутки времени и синхронизировать все изменения его внутреннего состояния.

Как вы понимаете, повторный анализ файлов ОАГ может привести к проблемам, если какой-либо из них будет долго загружаться. Это может произойти, например, если вы выполняете длительные или тяжелые вычисления при определении ОАГ.

Листинг 11.19 Выполнение вычислений в определении ОАГ (неэффективно)

```
...
task1 = PythonOperator(...)
my_value = do_some_long_computation()
task2 = PythonOperator(op_kwargs={"my_value": my_value})
...
```

Такое долгое вычисление будет происходить каждый раз при анализе ОАГ

Такая реализация заставит Airflow выполнять `do_some_long_computation` каждый раз при загрузке файла ОАГ, блокируя весь процесс анализа до тех пор, пока не будет завершено вычисление.

Один из способов избежать этой проблемы – отложить вычисление до выполнения задачи, для которой требуется вычисляемое значение.

Листинг 11.20 Выполнение вычислений внутри задач (более эффективно)

```
def _my_not_so_efficient_task(value, ...):
    ...

PythonOperator(
```

```
    task_id="my_not_so_efficient_task",
    ...
    op_kwargs={
        "value": calc_expensive_value()
    }
)


def _my_more_efficient_task(...):
    value = calc_expensive_value()
    ...

PythonOperator(
    task_id="my_more_efficient_task",
    python_callable=_my_more_efficient_task,
    ...
)
```

Здесь значение будет вычисляться каждый раз при анализе ОАГ

Перенося вычисление в задачу, значение будет вычисляться только при выполнении задачи

Еще один подход – написать собственный хук или оператор, который извлекает учетные данные только тогда, когда это необходимо для выполнения, но для этого может потребоваться немного больше работы.

Нечто подобное может произойти в более специфичных случаях, когда конфигурация загружается из внешнего источника данных или файловой системы в ваш основной файл ОАГ. Например, нам может понадобиться загрузить учетные данные из базы метаданных Airflow и использовать их в нескольких задачах, сделав что-то вроде этого:

Листинг 11.21 Извлечение учетных данных из базы метаданных в определении ОАГ (неэффективно)

```
from airflow.hooks.base_hook import BaseHook

api_config = BaseHook.get_connection("my_api_conn")
api_key = api_config.login
api_secret = api_config.password

task1 = PythonOperator(
    op_kwargs={"api_key": api_key, "api_secret": api_secret},
    ...
)
...
```

Этот вызов будет обращаться к базе данных каждый раз при анализе ОАГ

Однако недостаток такого подхода состоит в том, что он извлекает учетные данные из базы данных каждый раз при анализе ОАГ, вместо того чтобы делать это только при его выполнении. Таким образом, мы будем видеть повторяющиеся запросы каждые 30 секунд или около того (в зависимости от конфигурации Airflow) к нашей базе данных просто для того, чтобы получить эти учетные данные.

Подобных проблем с производительностью обычно можно избежать, отложив извлечение учетных данных до выполнения функции задачи.

Листинг 11.22 Извлечение учетных данных в задаче (более эффективно)

```
from airflow.hooks.base_hook import BaseHook

def _task1(conn_id, **context):
    api_config = BaseHook.get_connection(conn_id)
    api_key = api_config.login
    api_secret = api_config.password
    ...

task1 = PythonOperator(op_kwargs={"conn_id": "my_api_conn"})
```

Этот вызов будет обращаться к базе данных только при выполнении задачи

Таким образом, учетные данные извлекаются только тогда, когда задача фактически выполняется, что делает наши ОАГ намного эффективнее. Такой тип вычислений, при котором вы случайно включаете вычисления в определения ОАГ, может быть незаметным и требует бдительности, чтобы его избежать. Кроме того, иногда могут встречаться ситуации и похуже: возможно, вы не возражаете против неоднократной загрузки файла конфигурации из локальной файловой системы, но повторная загрузка из облачного хранилища или базы данных может быть менее предпочтительной.

11.1.5 Используйте фабричные функции для генерации распространенных шаблонов

В некоторых случаях вы можете оказаться в ситуации, когда вам снова и снова приходится писать варианты одного и того же ОАГ. Такое часто происходит, когда вы получаете данные из связанных источников данных, с небольшими вариациями в исходных путях и преобразованиях, применяемых к данным. Или у вас могут быть общие процессы обработки данных в компании, которые требуют шагов или преобразований, многие из которых одинаковые, в результате чего они повторяются во множестве разных ОАГ.

Один из эффективных способов ускорить процесс создания таких распространенных структур ОАГ – написать *фабричную функцию*. Идея подобной функции заключается в том, что она принимает любую необходимую конфигурацию для соответствующих шагов и генерирует соответствующий ОАГ или набор задач (таким образом производя его, как фабрика). Например, если у нас есть общий процесс, включающий в себя извлечение данных из внешнего API и их предварительную обработку, используя заданный сценарий, можно было бы написать фабричную функцию, которая выглядит примерно так:

Листинг 11.23 Генерация наборов задач с помощью фабричной функции (dags / 01_task_factory.py)

```
def generate_tasks(dataset_name, raw_dir, processed_dir,
                   preprocess_script, output_dir, dag):
    raw_path = os.path.join(raw_dir, dataset_name, "{ds_nodash}.json")
    processed_path = os.path.join(
        processed_dir, dataset_name, "{ds_nodash}.json"
    )
    output_path = os.path.join(output_dir, dataset_name, "{ds_nodash}.json")
    fetch_task = BashOperator(
        task_id=f"fetch_{dataset_name}",
        bash_command=f"echo 'curl http://example.com/{dataset_name}.json
          ➥ > {raw_path}.json'",
        dag=dag,
    )

    preprocess_task = BashOperator(
        task_id=f"preprocess_{dataset_name}",
        bash_command=f"echo '{preprocess_script} {raw_path}
          ➥ {processed_path}'",
        dag=dag,
    )

    export_task = BashOperator(
        task_id=f"export_{dataset_name}",
        bash_command=f"echo 'cp {processed_path} {output_path}'",
        dag=dag,
    )

fetch_task >> preprocess_task >> export_task

return fetch_task, export_task
```

Параметры, конфигурирующие задачи, которые будут созданы фабричной функцией

Пути к файлам, используемые разными задачами

Создание отдельных задач

Определение зависимостей задач

Возвращаем первую и последнюю задачи в цепочке, чтобы мы могли связать их с другими задачами в более крупном графе (при необходимости)

Затем мы могли бы использовать эту фабричную функцию для приема нескольких таких наборов данных.

Листинг 11.24 Применение функции task_factory (dags/01_task_factory.py)

```
import airflow.utils.dates
from airflow import DAG

with DAG(
    dag_id="01_task_factory",
    start_date=airflow.utils.dates.days_ago(5),
    schedule_interval="@daily",
```

```
) as dag:
    for dataset in ["sales", "customers"]:
        generate_tasks(
            dataset_name=dataset,
            raw_dir="/data/raw",
            processed_dir="/data/processed",
            output_dir="/data/output",
            preprocess_script=f"preprocess_{dataset}.py",
            dag=dag,
        )
```

Создание наборов задач с разными значениями конфигурации

Передача экземпляра ОАГ для подключения задач к ОАГ

У нас должен получиться ОАГ, похожий на тот, что показан на рис. 11.1. Конечно, для независимых наборов данных, вероятно, не имеет смысла принимать два набора данных в одном ОАГ. Однако можно легко разделить задачи между несколькими ОАГ, вызвав фабричный метод `generate_tasks` из разных файлов ОАГ.

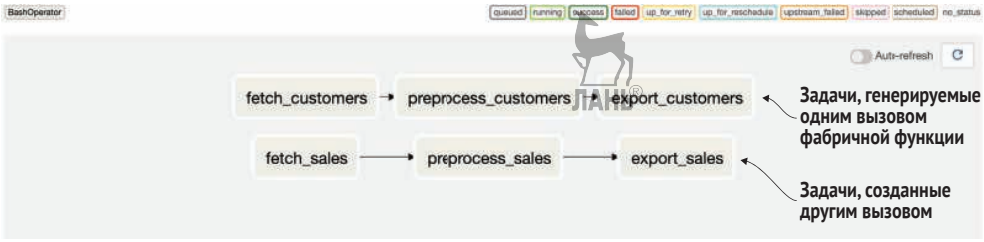


Рис. 11.1 Генерирование повторяющихся шаблонов задач с использованием фабричных методов. Этот пример содержит несколько наборов практически идентичных задач, которые были сгенерированы из объекта конфигурации с помощью метода `task_factory`

Также можно писать фабричные методы для генерации целых ОАГ, как показано в листинге 11.25.

Листинг 11.25 Генерация ОАГ с помощью фабричной функции (dags/02_dag_factory.py)

```
def generate_dag(dataset_name, raw_dir, processed_dir, preprocess_script):
    with DAG(
        dag_id=f"02_dag_factory_{dataset_name}",
        start_date=airflow.utils.dates.days_ago(5),
        schedule_interval="@daily",
    ) as dag:
        raw_file_path = ...
        processed_file_path = ...

        fetch_task = BashOperator(...)
        preprocess_task = BashOperator(...)

        fetch_task >> preprocess_task

    return dag
```

Генерация экземпляра ОАГ в рамках фабричной функции

Это позволит вам сгенерировать ОАГ, используя следующий скромный файл.

Листинг 11.26 Применение фабричной функции

```
...

dag = generate_dag(
    dataset_name="sales",
        raw_dir="/data/raw",
        processed_dir="/data/processed",
        preprocess_script="preprocess_sales.py",
)
```

Создание ОАГ с использованием фабричной функции

Данный подход также можно использовать для создания нескольких ОАГ с помощью файла ОАГ.

Листинг 11.27 Создание нескольких ОАГ с помощью фабричной функции (dags/02_dag_factory.py)

```
...

for dataset in ["sales", "customers"]:
    globals()[f"02_dag_factory_{dataset}"] = generate_dag(
        dataset_name=dataset,
        raw_dir="/data/raw",
        processed_dir="/data/processed",
        preprocess_script=f"preprocess_{dataset}.py",
    )
```

Создание нескольких ОАГ с разными конфигурациями. Обратите внимание, что мы должны присвоить каждому ОАГ уникальное имя в глобальном пространстве имен (используя трюк с глобальными переменными), чтобы убедиться, что они не перезаписывают друг друга

Этот цикл, по сути, генерирует несколько объектов ОАГ в глобальной области видимости файла ОАГ, который Airflow использует как отдельные ОАГ (рис. 11.2). Обратите внимание, что объектам нужны разные имена переменных, чтобы они не перезаписывали друг друга; иначе Airflow увидит только один экземпляр ОАГ (последний экземпляр, сгенерированный циклом).

Мы рекомендуем соблюдать осторожность при генерации нескольких ОАГ из одного файла, поскольку это может сбивать с толку, если вы этого не ожидаете. (Более общий паттерн – наличие одного файла для каждого ОАГ.) Таким образом, этот шаблон лучше всего использовать экономно, когда он обеспечивает значительные преимущества.

Фабричные методы могут быть особенно эффективны в сочетании с файлами конфигурации или другими формами внешней конфигурации. Это позволяет, например, создать фабричную функцию, которая принимает файл YAML в качестве входных данных и генерирует ОАГ на основе конфигурации, определенной в этом файле. Таким образом, можно сконфигурировать повторяющиеся ETL-процессы, используя набор относительно простых файлов конфигурации, которые

также могут редактироваться пользователями, которые мало знакомы с Airflow.

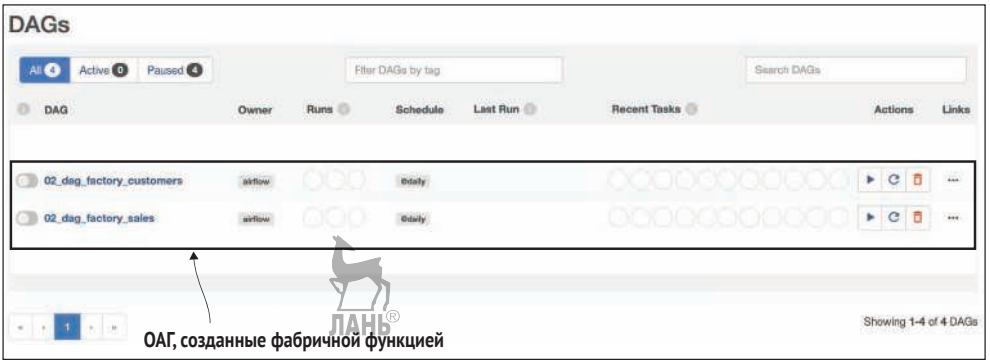


Рис. 11.2 Несколько ОАГ, сгенерированных из одного файла с помощью фабричной функции (это скриншот из пользовательского интерфейса Airflow, показывающий несколько ОАГ, которые были созданы из одного файла с использованием фабричной функции)

11.1.6 Группируйте связанные задачи с помощью групп задач

Сложные ОАГ, особенно те, что созданы с использованием фабричных методов, часто бывает трудно понять из-за сложной структуры или огромного количества задействованных задач. Чтобы помочь привести в порядок эти сложные структуры, у Airflow 2 есть новая функция – *группы задач*. Группы задач позволяют (визуально) группировать наборы задач на более мелкие группы, что упрощает контроль и понимание структуры ОАГ.

Группы задач можно создавать с помощью диспетчера контекста TaskGroup. Например, если взять предыдущий пример фабрики задач, то можно сгруппировать задачи, созданные для каждого набора данных:

Листинг 11.28 Использование TaskGroups для визуальной группировки задач (dags/03_task_groups.py)

```
...
for dataset in ["sales", "customers"]:
    with TaskGroup(dataset, tooltip=f"Tasks for processing {dataset}"):
        generate_tasks(
            dataset_name=dataset,
            raw_dir="/data/raw",
            processed_dir="/data/processed",
            output_dir="/data/output",
            preprocess_script=f"preprocess_{dataset}.py",
            dag=dag,
    )
```

Этот код группирует набор задач, созданных для наборов данных `sales` и `customers`, в две группы, по одной для каждого набора. В ре-

зультате сгруппированные задачи отображаются в виде единой группы в веб-интерфейсе, которую можно расширить, щелкнув по соответствующей группе (рис. 11.3).

Рис. 11.3 Группы задач могут помочь организовать ОАГ путем группировки связанных задач. Изначально группы задач изображены в виде отдельных узлов в ОАГ, как показано для группы задач `customers` на этом рисунке. Нажав на группу задач, вы можете развернуть ее и просмотреть задачи в группе, как показано здесь для группы задач `sales`. Обратите внимание, что группы задач могут быть вложенными. Это означает, что у вас могут быть группы задач внутри других групп

Хотя это относительно простой пример, группы задач могут быть довольно эффективны для уменьшения количества визуального шума в более сложных случаях. Например, в нашем ОАГ для тренировки моделей машинного обучения из главы 5 мы создали значительное количество задач для извлечения и очистки данных о погоде и продажах из разных систем. Группы задач позволяют снизить очевидную сложность этого ОАГ за счет группировки задач, связанных с продажами и погодой, в соответствующие группы. Это позволяет нам скрыть сложность задач выборки набора данных по умолчанию, но по-прежнему увеличивать масштаб отдельных задач, когда это необходимо (рис. 11.4).

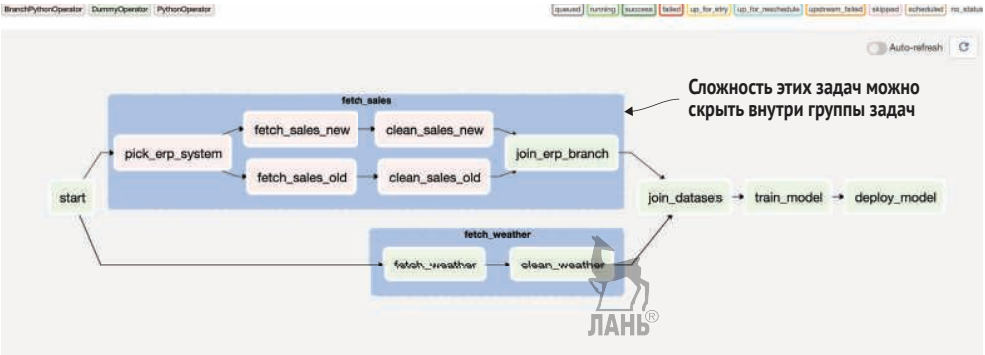


Рис. 11.4 Использование групп задач для организации ОАГ из главы 5. Здесь группировка задачи для наборов по извлечению и очистке данных о погоде и продажах помогает значительно упростить сложные структуры задач, участвующих в этих процессах (пример кода приведен в dags/04_task_groups_umbrella.py)

11.1.7 Создавайте новые DAG для больших изменений

После запуска DAG база данных планировщика содержит экземпляры запусков этого DAG. Значительные изменения в нем, например в дате начала и/или интервале планирования, могут сбить планировщика с толку, поскольку изменения больше не соответствуют предыдущим запускам DAG. Точно так же удаление или переименование задач не позволит вам получить доступ к истории этих задач из пользовательского интерфейса, поскольку они больше не будут соответствовать текущему состоянию DAG и, следовательно, не будут отображаться.

Лучший способ избежать этих проблем – создавать новую версию DAG всякий раз, когда вы решите внести большие изменения в существующие DAG, поскольку в настоящее время Airflow не поддерживает версионированные DAG. Это можно сделать, создав новую версионированную копию DAG (т. е. dag_v1, dag_v2) перед внесением желаемых изменений. Таким образом, вы можете избежать путаницы в планировщике, сохранив при этом предыдущую информацию о доступной старой версии. Поддержка версионированных DAG, возможно, будет добавлена в будущем, поскольку в сообществе есть сильное желание сделать это.

11.2 Проектирование воспроизводимых задач

Помимо кода, одна из самых больших проблем при написании хорошего DAG – это спроектировать свои задачи таким образом, чтобы их можно было воспроизводить. Это означает, что вы можете с легкостью перезапустить задачу и ожидать того же результата – даже если задача выполняется в разные моменты времени. В этом разделе мы пересмотрим некоторые ключевые идеи и дадим несколько советов по обеспечению соответствия ваших задач этой парадигме.

11.2.1 Всегда требуйте, чтобы задачи были идемпотентными

Как кратко обсуждалось в главе 3, одно из ключевых требований хорошей задачи заключается в том, что задача должна быть идемпотентной, то есть повторное выполнение одной и той же задачи несколько раз дает тот же общий результат (при условии, что сама задача не изменилась).

Идемпотентность – важная особенность, потому что существует множество ситуаций, когда вы или Airflow можете повторно запустить задачу. Например, вам может понадобиться выполнить повторный запуск некоторых DAG после изменения кода, что приведет к повторному выполнению данной задачи. В других случаях Airflow сам может повторно запустить невыполненную задачу, используя механизм повтора, хотя данная задача все же успела записать некоторые результаты до того, как дала сбой. В обоих случаях нужно избежать созда-

ния нескольких копий одних и тех же данных в своем окружении или столкновения с другими нежелательными побочными эффектами.

Идемпотентность обычно можно обеспечить, потребовав перезаписи всех выходных данных при повторном запуске задачи, поскольку это гарантирует, что любые данные, записанные в предыдущем запуске, будут перезаписаны новым результатом. Точно так же вы должны внимательно рассмотреть другие побочные эффекты задачи (например, отправку уведомлений и т. д.) и определить, нарушают ли они ее идемпотентность каким-либо пагубным образом.

11.2.2 Результаты задачи должны быть детерминированными

Задачи могут быть воспроизводимы только в том случае, если они детерминированы. Это означает, что задача всегда должна возвращать один и тот же вывод для данного ввода. Напротив, недетерминированные задачи не позволяют создавать воспроизводимые ОАГ, поскольку каждый запуск задачи может дать другой результат, даже с теми же входными данными.

Недетерминированное поведение можно ввести разными способами:

- полагаясь на неявный порядок данных или структур данных внутри функции (например, неявное упорядочивание словаря Python или порядок строк, в котором набор данных возвращается из базы данных, без какого-либо определенного упорядочивания);
- использованием внешнего состояния в функции, включая случайные значения, глобальные переменные, внешние данные, хранящиеся на диске (не переданные в качестве входных данных в функцию) и т. д.;
- параллельной обработкой данных (в нескольких процессах или потоках) без явного упорядочивания результата;
- состоянием гонки в многопоточном коде;
- неправильной обработкой исключений.

В целом проблем с недетерминированными функциями можно избежать, тщательно подумав об источниках недетерминированности, которые могут возникнуть в вашей функции. Например, можно избежать недетерминированности в упорядочивании набора данных, применив явную сортировку. Точно так же можно избежать проблем с алгоритмами, которые включают случайность, задав случайное начальное число перед выполнением соответствующей операции.

11.2.3 Проектируйте задачи с использованием парадигмы функционального программирования

Один из подходов, который может помочь в создании задач, – проектировать их в соответствии с парадигмой функционального программирования. Функциональное программирование – это подход

к созданию компьютерных программ, которые, по сути, рассматривают вычисления как применение математических функций, избегая изменения состояния и изменяемых данных. Кроме того, функции в языках функционального программирования обычно должны быть чистыми. Это означает, что они могут вернуть результат, но в остальном не имеют побочных эффектов.

Одно из преимуществ такого подхода состоит в том, что результат чистой функции в языке функционального программирования всегда должен быть одинаковым для заданного ввода. Такие чистые функции обычно являются идемпотентными и детерминированными – а это именно то, чего мы и пытаемся добиться для наших задач в функциях Airflow. Поэтому сторонники парадигмы функционального программирования утверждают, что аналогичные подходы можно применять к приложениям обработки данных, вводя парадигму функционального дата-инжиниринга.

Подходы к функциональному дата-инжинирингу, по сути, направлены на применение тех же концепций языков функционального программирования к задачам дата-инжиниринга. Сюда входит требование, согласно которому задачи не должны иметь побочных эффектов и у них всегда должен быть одинаковый результат, если применять их к одному и тому же набору входных данных. Главное преимущество соблюдения этих ограничений заключается в том, что они имеют большое значение для достижения идеалов идемпотентных и детерминированных задач, что делает наши ОАГ и задачи воспроизводимыми.

Для получения дополнительной информации обратитесь к этому посту в блоге Максима Бошмана (одной из ключевых фигур, стоящих за Airflow), который представляет собой отличное введение в концепцию функционального дата-инжиниринга для конвейеров обработки данных в Airflow: http://mng.bz/2eqm.

11.3 Эффективная обработка данных

ОАГ, предназначенные для обработки больших объемов данных, должны быть тщательно спроектированы, чтобы иметь возможность делать это наиболее эффективным способом. В этом разделе мы обсудим пару советов относительно того, как эффективно обрабатывать большие объемы данных.

11.3.1 Ограничьте объем обрабатываемых данных

Хотя это и может показаться несколько банальным, лучший способ эффективно обрабатывать данные – ограничить обработку до минимального количества данных, необходимых для получения желаемого результата. Ведь обработка данных, которые в любом случае будут отброшены, – пустая трата времени и ресурсов.

На практике это означает, что нужно тщательно продумать источники данных и определить, все ли они необходимы. Если говорить

о необходимых наборах данных, посмотрите, можно ли уменьшить их размер, отбросив неиспользуемые строки или столбцы. Агрегирование на ранней стадии также может существенно повысить производительность, поскольку правильное агрегирование может значительно уменьшить размер промежуточного набора данных, тем самым сократив объем работ, которые необходимо выполнить.

В качестве примера представьте себе процесс обработки данных, в котором нам нужно рассчитать ежемесячные объемы продаж нашей продукции для определенной клиентской базы (рис. 11.5).

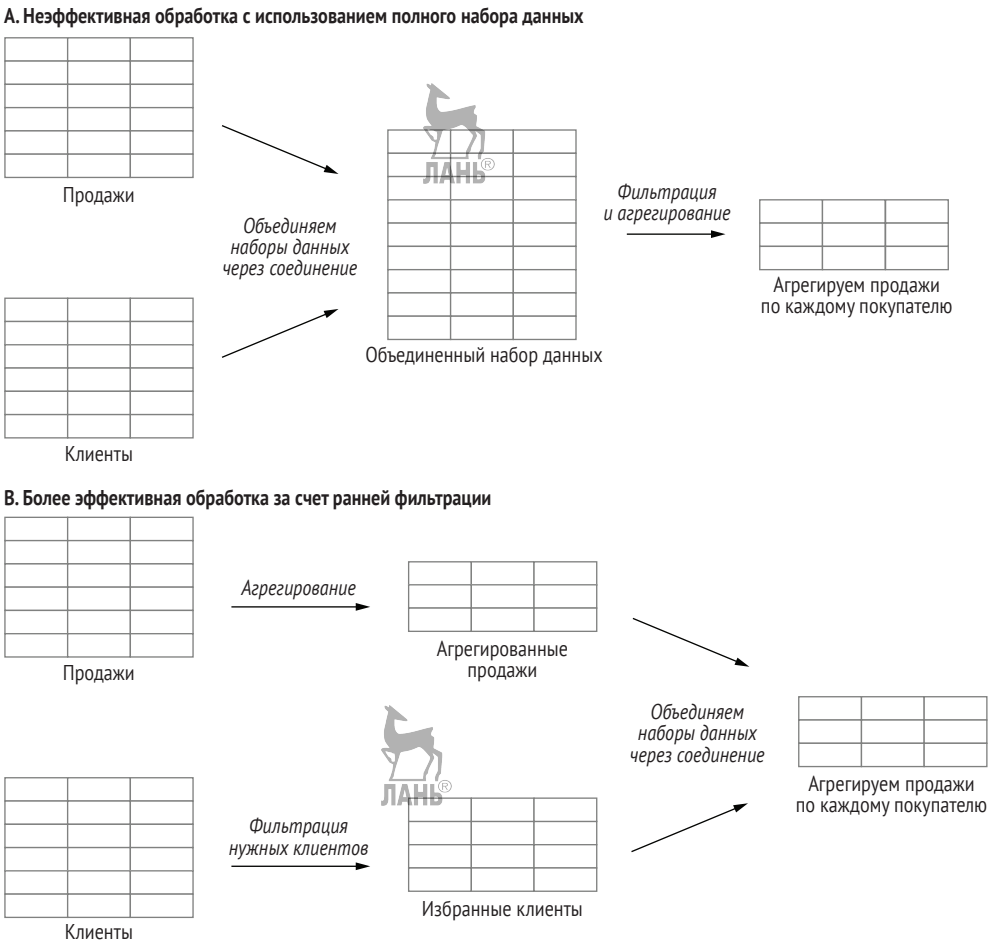


Рис. 11.5 Пример неэффективного процесса обработки данных по сравнению с более эффективным: (A) один из способов рассчитать совокупные продажи на каждого покупателя – сначала полностью объединить оба набора данных, а затем агрегировать продажи до необходимой степени детализации и выполнить фильтрацию по интересующим нас клиентам. Хотя это и может дать желаемый результат, это не очень эффективно из-за потенциально большого размера объединенной таблицы; (B) более эффективный подход состоит в том, чтобы сначала отфильтровать или агрегировать таблицы продаж и клиентов до минимально необходимой детализации, что позволяет объединить два небольших набора данных

В этом примере мы можем рассчитать совокупные продажи, сначала объединив два набора данных, за которыми следует этап агрегации и фильтрации, на котором мы агрегируем продажи до требуемой степени детализации, а затем фильтруем их по нужным нам клиентам. Недостаток такого подхода заключается в том, что мы соединяем два потенциально больших набора данных, чтобы получить результат, а это может потребовать значительных затрат времени и ресурсов.

Более эффективный подход – продвинуть этапы фильтрации и агрегирования вперед, что позволит уменьшить размер наборов данных о клиентах и продажах перед их объединением. Потенциально это позволяет значительно уменьшить размер объединенного набора данных, делая вычисления намного эффективнее.

Хотя, возможно, это немного абстрактный пример, мы сталкивались с большим количеством аналогичных случаев, когда интеллектуальная агрегация или фильтрация наборов данных (как в отношении строк, так и столбцов) значительно повысила производительность задействованных процессов обработки данных. Таким образом, может быть полезно внимательно посмотреть на свои ОАГ и определить, обрабатывают ли они больше данных, чем это необходимо.

11.3.2 Инкрементальная загрузка и обработка

Во многих случаях нельзя уменьшить размер набора данных с помощью умной агрегации или фильтрации. Однако, особенно если речь идет о наборах данных временных рядов, вы также часто можете ограничить объем обработки, который вам нужно выполнять каждый раз, используя инкрементальную обработку данных.

Основная идея инкрементальной обработки (которую мы затронули в главе 3) состоит в том, чтобы разделить данные на (временные) разделы и обрабатывать их индивидуально в каждом из запусков ОАГ. Таким образом вы ограничиваете объем данных, обрабатываемых при каждом запуске размером соответствующего раздела, который обычно намного меньше размера всего набора данных. Однако, добавляя результаты каждого запуска в качестве приращений к набору выходных данных, вы все равно будете наращивать весь набор с течением времени (рис. 11.6).

Преимущество проектирования инкрементального процесса состоит в том, что ошибка в одном из запусков не потребует от вас повторного выполнения анализа для всего набора данных; если запуск оказался неудачным, можно просто выполнить его снова. Конечно, в некоторых случаях вам все же может потребоваться провести анализ всего набора данных. Тем не менее вы все равно можете получить выгоду от инкрементальной обработки, выполнив этапы фильтрации и агрегирования в инкрементальной части процесса и проведя крупномасштабный анализ уменьшенного результата.

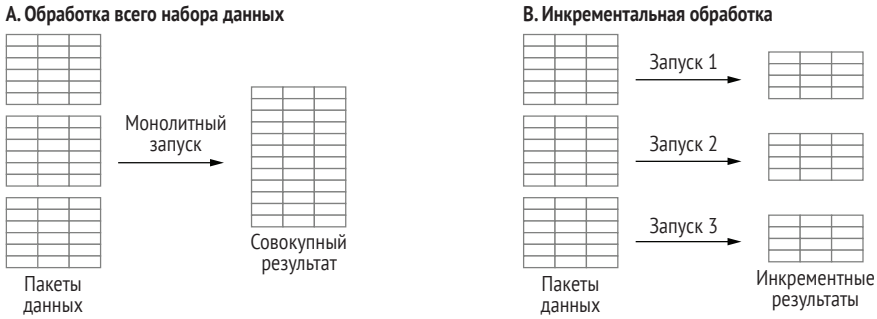


Рис. 11.6 Иллюстрация монолитной обработки (A), при которой весь набор данных обрабатывается при каждом запуске, по сравнению с инкрементальной обработкой (B), при которой набор данных анализируется инкрементными партиями по мере поступления данных

11.3.3 Кешируйте промежуточные данные

В большинстве рабочих процессов обработки данных ОАГ состоят из нескольких этапов, каждый из которых выполняет дополнительные операции с данными, полученными из предыдущих этапов. Преимущество такого подхода (как было описано ранее в этой главе) состоит в том, что он разбивает ОАГ на четкие, атомарные этапы, которые легко повторить, если мы обнаружим ошибки.

Однако, чтобы иметь возможность повторно выполнить какие-либо шаги в таком ОАГ, нужно убедиться, что данные, необходимые для этих шагов, легко доступны (рис. 11.7). В противном случае мы не сможем повторно запустить какой-либо отдельный этап без повторного запуска всех его зависимостей, что частично сводит на нет цель разделения нашего рабочего процесса на задачи.

Рис. 11.7 Сохранение промежуточных данных из задач гарантирует, что каждую задачу можно легко запустить повторно независимо от других задач. В этом примере облачное хранилище (обозначенное бакетом) используется для хранения промежуточных результатов задач *fetch/preprocess*

Недостаток кеширования промежуточных данных состоит в том, что на это может потребоваться чрезмерный объем памяти, если у вас несколько промежуточных версий больших наборов данных. В этом случае можно подумать о том, чтобы пойти на компромисс: вы будете хранить промежуточные наборы данных только в течение ограниченного периода времени, что даст вам время для повторного выполнения отдельных задач, если вы столкнетесь с проблемами в недавних запусках.

Тем не менее мы рекомендуем, чтобы у вас под рукой всегда была самая «сырая» версия ваших данных (например, данные, которые вы только что получили из внешнего API). Это гарантирует, что у вас всегда будет копия данных в том виде, в котором они были на тот момент. Данный тип снимка, говоря об управлении версиями данных, часто недоступен в исходных системах, таких как базы данных (при условии что снимки не были сделаны) или API. Хранение такой необработанной копии данных гарантирует, что вы всегда сможете обработать их по мере необходимости, например всякий раз, когда вы вносите изменения в свой код или если возникают проблемы во время начальной обработки.

11.3.4 Не храните данные в локальных файловых системах

При обработке данных в рамках задания Airflow у вас может возникнуть соблазн записать промежуточные данные в локальную файловую систему. Это особенно актуально при использовании операторов, которые запускаются локально, таких как операторы Bash и Python, поскольку локальная файловая система легко доступна изнутри них.

Однако недостаток записи файлов в локальные системы состоит в том, что нижестоящие задачи могут не иметь к ним доступа, потому что Airflow запускает свои задачи в нескольких воркерах. Это позволяет ему запускать несколько задач параллельно. В зависимости от развертывания это может означать, что две зависимые задачи (т. е. одна задача ожидает данные от другой) могут запускаться на двух разных воркерах, у которых нет доступа к файловым системам друг друга, поэтому они не могут получить доступ к файлам.

Самый простой способ избежать подобной проблемы – использовать общее хранилище, к которому можно одинаково просто получить доступ из любого воркера Airflow. Например, обычно используется паттерн для записи промежуточных файлов в общий бакет облачного хранилища, доступ к которому можно получить из каждого воркера, используя одни и те же URL-адреса файлов и учетные данные. Точно так же общие базы данных или другие системы хранения могут использоваться для хранения данных, в зависимости от типа данных.

11.3.5 Переложите работу на внешние/исходные системы

В целом Airflow по-настоящему эффективен, когда применяется в качестве инструмента оркестровки, а не использует воркеры для выполнения фактической обработки данных. Например, при небольших наборах данных обычно можно обойтись загрузкой данных непосредственно в воркеры с помощью оператора `PythonOperator`. Однако для больших наборов данных это может стать проблемой, поскольку они будут требовать от вас запуска воркеров на все более крупных машинах.

В таких случаях можно получить гораздо большую производительность от небольшого кластера Airflow, переложив вычисления или за-

просы на внешние системы, которые лучше всего подходят для данного типа работы. Например, при запросе данных из базы данных можно сделать свою работу более эффективной, направляя всю необходимую фильтрацию или агрегацию в саму систему базы данных, вместо того чтобы получать данные локально и выполнять вычисления в Python в воркере. Точно так же в случае приложений с большими данными обычно можно повысить производительность, используя Airflow для выполнения вычислений во внешнем кластере Spark.

Ключевой момент здесь состоит в том, что изначально Airflow разрабатывался как инструмент оркестровки, поэтому вы получите лучшие результаты, если будете использовать его таким образом. Другие инструменты, как правило, лучше подходят для выполнения фактической обработки данных, поэтому обязательно используйте их для этого, давая им возможность продемонстрировать свои сильные стороны.

11.4 Управление ресурсами

При работе с большими объемами данных можно запросто перегрузить свой кластер Airflow или другие системы, используемые для обработки данных. В этом разделе мы рассмотрим несколько советов по эффективному управлению ресурсами и надеемся, что вы получите ряд идей, которые помогут вам справиться с подобными проблемами.

11.4.1 Управление параллелизмом с помощью пулов

При параллельном выполнении множества задач можно столкнуться с ситуациями, когда нескольким задачам требуется доступ к одному и тому же ресурсу. Это может привести к быстрой перегрузке указанного ресурса, если он не предназначен для такого рода параллелизма. Среди примеров можно упомянуть общие ресурсы, такие как база данных или система графического процессора, но это также могут быть кластеры Spark, если, например, вы хотите ограничить количество заданий, выполняемых в данном кластере.

Airflow позволяет контролировать количество задач, имеющих доступ к данному ресурсу, с помощью *пулов ресурсов*, где каждый пул содержит фиксированное количество *слотов*, которые предоставляют доступ к соответствующему ресурсу. Отдельные задачи, которым необходим доступ к ресурсу, можно назначить пулу ресурсов, по сути сообщая планировщику Airflow, что ему необходимо получить слот из пула, прежде чем он сможет запланировать соответствующую задачу.

Пул ресурсов можно создать, перейдя в раздел **Admin > Pools** (Администратор > Пулы) в пользовательском интерфейсе Airflow. Это представление покажет вам обзор пулов, которые были определены внутри Airflow (рис. 11.8). Чтобы создать новый пул ресурсов, нажмите **Create** (Создать). В новом экране (рис. 11.9) вы можете ввести на-

звание и описание нового пула ресурсов наряду с количеством слотов, которые хотите ему назначить. Количество слотов определяет степень параллелизма для пула ресурсов. Это означает, что пул с 10 слотами позволит 10 задачам одновременно обращаться к соответствующему ресурсу.

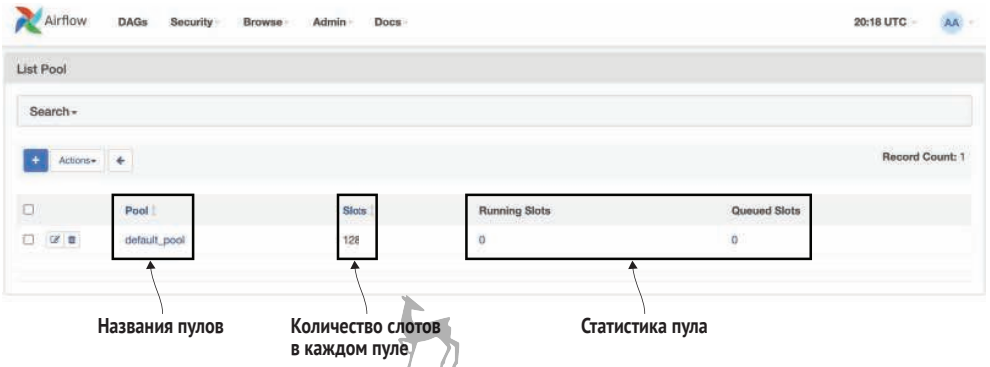


Рис. 11.8 Обзор пулов ресурсов Airflow в веб-интерфейсе

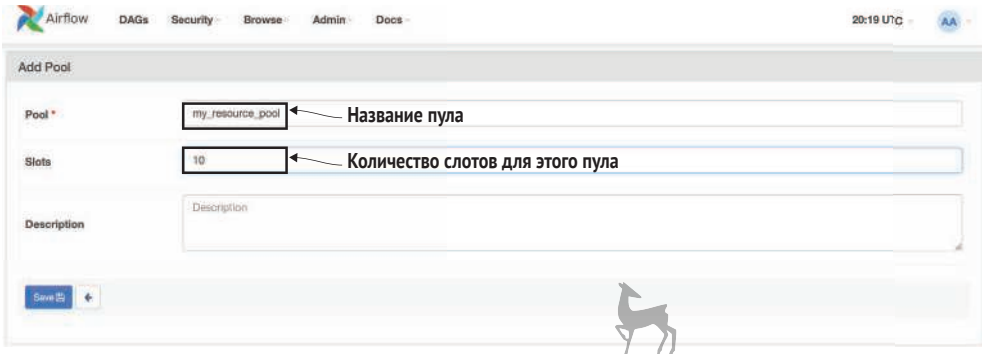


Рис. 11.9 Создание нового пула ресурсов в веб-интерфейсе Airflow

Чтобы ваши задачи использовали новый пул ресурсов, необходимо назначить его при создании задачи.

Листинг 11.29 Назначение определенного пула ресурсов задаче

```
PythonOperator(
    task_id="my_task",
    ...
    pool="my_resource_pool"
)
```

Таким образом, Airflow проверит, доступны ли еще какие-либо слоты в `my_resource_pool` перед планированием задачи в данном запуске. Если пул все еще содержит свободные слоты, планировщик потребует

пустой слот (уменьшив количество доступных слотов на один) и запланирует выполнение задачи. Если свободных слотов в пуле нет, планировщик отложит планирование задачи до тех пор, пока не появится доступный слот.

11.4.2 Обнаружение задач с длительным временем выполнения с помощью соглашений об уровне предоставления услуг и оповещений

В некоторых случаях выполнение задач или ОАГ может занять больше времени, чем обычно, из-за непредвиденных проблем с данными, ограниченных ресурсов и т. д. Airflow позволяет отслеживать поведение задач с помощью механизма *SLA* (service-level agreement – соглашение об уровне предоставления услуг). Эта функция дает возможность назначать тайм-ауты ОАГ или задачам, и в этом случае Airflow предупредит вас, если какая-либо из ваших задач или ОАГ не соответствует соглашению (т. е. занимает больше времени, чем указано в соглашении).

На уровне ОАГ можно назначить тайм-аут, передав `default_args` аргумент `sla`.

Листинг 11.30 Назначаем тайм-аут всем задачам в ОАГ (dags/05_sla_misses.py)

```
from datetime import timedelta

default_args = {
    "sla": timedelta(hours=2),
    ...
}

with DAG(
    dag_id="...",
    ...
    default_args=default_args,
) as dag:
    ...
```

Применяя SLA на уровне ОАГ, Airflow будет проверять результат каждой задачи после ее выполнения, чтобы определить, превышено ли время начала или окончания задачи (по сравнению со временем запуска ОАГ). В случае превышения Airflow сгенерирует оповещение, уведомляющее пользователей о том, что произошло. После создания оповещения Airflow продолжит выполнение остальной части ОАГ, генерируя аналогичные оповещения для других задач, у которых отмечено превышение.

По умолчанию несоблюдения соглашения записываются в базу метаданных Airflow. Их можно просмотреть с помощью веб-интерфейса

в разделе **Browse > SLA misses** (Просмотр > Несоблюдения соглашения об уровне предоставления услуг). Письма с оповещениями также отправляются на адрес электронной почты, определенный в ОАГ (с использованием аргумента `email`), предупреждая пользователей, что для соответствующей задачи SLA было превышено.

Вы также можете определить собственные обработчики, передав ОАГ функцию обработчика с помощью параметра `sla_miss_callback`.

Листинг 11.31 Определяем пользовательский обработчик (dags/05_sla_misses.py)

```
def sla_miss_callback(context):
    send_slack_message("Missed SLA!")

...

with DAG(
     ...
     sla_miss_callback=sla_miss_callback
) as dag:
     ...
```

Также можно указать SLA на уровне задачи, передав оператору задачи аргумент `sla`.

Листинг 11.32 Назначение SLA определенным задачам

```
PythonOperator(
    ...
    sla=timedelta(hours=2)
)
```

Этот код обеспечивает выполнение SLA только для соответствующих задач. Однако важно отметить, что Airflow по-прежнему будет сравнивать время окончания задачи со временем запуска ОАГ при соблюдении SLA, а не время начала задачи. Это связано с тем, что в Airflow эти соглашения всегда определяются относительно времени запуска ОАГ, а не отдельных задач.

Резюме

- Принятие общих соглашений о стилях наряду с вспомогательными инструментами проверки соблюдения стандарта оформления кода и форматирования может значительно повысить читабельность кода вашего ОАГ.
- Фабричные методы позволяют эффективно создавать повторяющиеся ОАГ или структуры задач, фиксируя различия между экземплярами в небольших объектах или файлах конфигурации.

- Идемпотентные и детерминированные задачи являются ключом к созданию воспроизводимых задач и ОАГ, которые легко запускать повторно и применять к ним обратное заполнение. Концепции функционального программирования могут помочь вам разработать задачи с этими характеристиками.
- Процессы обработки данных можно реализовать эффективно, если тщательно продумать, как обрабатываются данные (т. е. обработка в соответствующих системах, ограничение количества загружаемых данных и использование инкрементальной загрузки) и путем кеширования наборов промежуточных данных в доступных файловых системах, которые доступны в воркерах.
- Вы можете управлять доступом к своим ресурсам в Airflow или ограничивать его с помощью пулов ресурсов.
- Задачи или ОАГ с длительным временем выполнения можно обнаружить и пометить с помощью соглашений об уровне предоставления услуг.

12 Эксплуатация Airflow в промышленном окружении

Эта глава рассказывает:

- о планировщике Airflow;
- о настройке Airflow для горизонтального масштабирования с помощью разных исполнителей;
- о визуальном мониторинге статуса и производительности Airflow;
- об отправке оповещений при сбое задачи.

В большинстве предыдущих глав мы рассматривали различные части Airflow с точки зрения программиста. Цель данной главы – изучить Airflow с точки зрения эксплуатации. Предполагается общее понимание таких концепций, как архитектура (распределенного) программного обеспечения, журналирование, мониторинг и оповещение. Однако знаний никаких специальных технологий не требуется.

Конфигурация Airflow

В этой главе мы часто упоминаем конфигурацию Airflow. Она интерпретируется в следующем порядке предпочтения:

1. переменная окружения (AIRFLOW__[SECTION]__[KEY]);
2. переменная окружения команды (AIRFLOW__[SECTION]__[KEY]_CMD);
3. в airflow.cfg;
4. команда в airflow.cfg;
5. значение по умолчанию.

Всякий раз, когда речь идет о параметрах конфигурации, мы будем демонстрировать вариант 1. Например, возьмем элемент конфигурации `web_server_port` из раздела `webserver`. Он будет продемонстрирован как `AIRFLOW__WEBSERVER__WEB_SERVER_PORT`.

Чтобы найти текущее значение любого элемента конфигурации, можно прокрутить вниз страницу **Configurations** (Конфигурации) в пользовательском интерфейсе Airflow до таблицы **Running Configuration** (Текущая конфигурация), где показаны все параметры конфигурации, их текущее значение и какой из пяти вариантов был использован для настройки конфигурации.

12.1 Архитектура Airflow

В минимальном варианте Airflow состоит из трех компонентов (рис. 12.1):

- веб-сервер;
- планировщик;
- база данных.

Рис. 12.1 Самая простая архитектура Airflow

Веб-сервер и планировщик – это процессы Airflow. База данных – отдельная служба, которую вы должны предоставить Airflow для хранения метаданных с веб-сервера и планировщика. Папка с определениями ОАГ должна быть доступна планировщику.

Веб-сервер и развертывание ОАГ в Airflow 1

В Airflow 1 файлы ОАГ должны быть доступны как для веб-сервера, так и для планировщика. Это усложняет развертывание, поскольку использование файлом несколькими машинами или процессами – нетривиальная задача.

В Airflow 2 ОАГ записываются в базу данных в сериализованном формате. Веб-сервер читает этот формат из базы данных, и ему не требуется доступ к файлам ОАГ.

Сериализация ОАГ стала возможной начиная с Airflow версии 1.10.10, хотя это необязательно. Чтобы активировать сериализацию ОАГ в Airflow 1 (только для версии 1.10.10 или выше), необходимо задать значение `True` для следующих переменных:

- AIRFLOW__CORE__STORE_DAG_CODE=True;
- AIRFLOW__CORE__STORE_SERIALIZED_DAGS=True.

Веб-сервер отвечает за визуальное отображение информации о текущем состоянии конвейеров и позволяет пользователю выполнять определенные действия, такие как запуск ОАГ.

Планировщик несет ответственность за:

1. анализ файлов ОАГ (т. е. читает их, извлекает фрагменты и сохраняет их в базе метаданных);
2. определение задач, которые нужно выполнить, и размещение этих задач в очередь.

Мы более подробно рассмотрим обязанности планировщика в разделе 12.1.3. Airflow можно установить разными способами: с одной машины (что требует минимальных усилий для настройки, но это не масштабируемый вариант), на несколько машин (что требует дополнительной подготовки, но этот вариант обладает горизонтальной масштабируемостью). В Airflow различные режимы выполнения настраиваются по типу *исполнителя*. На момент написания этой книги существовало четыре типа исполнителей:

- `SequentialExecutor` (по умолчанию);
- `LocalExecutor`;
- `CeleryExecutor`;
- `KubernetesExecutor`.

Тип исполнителя можно настроить, задав для переменной окружения `AIRFLOW _CORE_ EXECUTOR` один из типов исполнителей из списка (табл. 12.1). Посмотрим, как функционируют эти четыре исполнителя.

Таблица 12.1 Обзор режимов исполнителей Airflow

Исполнитель	Является ли распределенным	Насколько легко его установить	Хорошо подходит для
`SequentialExecutor`	Нет	Очень просто	Демонстрации и тестирования
`LocalExecutor`	Нет	Просто	Достаточно неплох при работе на одной отдельной машине
`CeleryExecutor`	Да	Умеренная сложность	Если нужно масштабировать, используя несколько машин
`KubernetesExecutor`	Да	Сложно	Если вы знакомы с Kubernetes и предпочитаете вариант настройки с использованием контейнеров

12.1.1 Какой исполнитель мне подходит?

`SequentialExecutor` – самый простой исполнитель, который вы получаете автоматически с Airflow. Как следует из названия, он выполняет задачи последовательно, в основном используется для тестирования и демонстрационных целей и будет запускать задачи довольно медленно. Он будет работать только на одной машине.

Следующий исполнитель – это `LocalExecutor`, который не ограничивается одной задачей за раз, а может запускать несколько задач параллельно. Он регистрирует задачи для запуска в очереди вида *FIFO* (англ. first in, first out – «первым пришёл – первым ушёл»), которые рабочие процессы читают и выполняют. По умолчанию `LocalExecutor` может запускать до 32 параллельных процессов (это число можно настроить).

Если вы хотите распределить свои рабочие нагрузки на несколько машин, у вас есть два варианта: `CeleryExecutor` и `KubernetesExecutor`. Такое распределение может выполняться по разным причинам: вы исчерпали лимит ресурсов одной машины, вам нужна избыточность, запуская задания на нескольких машинах, или вы просто хотите ускорить выполнение рабочих нагрузок за счет распределения работы по нескольким машинам.

`CeleryExecutor` применяет Celery (https://docs.celeryproject.org/en/stable/) в качестве механизма постановки задач в очередь для выполнения, а воркеры читают и обрабатывают задачи из очереди. С точки зрения пользователя он работает так же, как и `LocalExecutor`, отправляя задачи в очередь, а воркеры читают задачи, которые нужно обработать, из очереди. Однако главное отличие состоит в том, что все компоненты могут работать на разных машинах, распространяя рабочую нагрузку. В настоящее время Celery поддерживает RabbitMQ, Redis и AWS SQS для механизма постановки в очередь (в Celery это называется *брокер*). Celery также поставляется с инструментом мониторинга под названием Flower для проверки состояния системы Celery. Celery – это библиотека Python, поэтому она прекрасно интегрируется с Airflow. Например, команда `airflow celery worker` фактически запускает воркер Celery. Единственная реальная внешняя зависимость для этой настройки – это механизм постановки в очередь.

Наконец, `KubernetesExecutor`, как следует из названия, запускает рабочие нагрузки, используя Kubernetes (https://kubernetes.io). Для этого требуется установка и настройка кластера Kubernetes, на котором будет запускаться Airflow, а исполнитель интегрируется с API Kubernetes для распределения задач. Kubernetes – это решение де-факто для запуска контейнеризированных рабочих нагрузок. Это подразумевает, что каждая задача в ОАГ Airflow выполняется в модуле Kubernetes. Kubernetes легко настраивается и масштабируется и часто уже используется в компаниях; поэтому многие с удовольствием применяют Kubernetes в сочетании с Airflow.

12.1.2 Настройка базы метаданных для Airflow

Все, что происходит в Airflow, регистрируется в базе данных, которую мы также называем *базой метаданных* в Airflow. Сценарий рабочего процесса состоит из нескольких компонентов, которые планировщик интерпретирует и сохраняет в базе метаданных. Airflow выполняет все операции с базой данных с помощью SQLAlchemy, ORM-фреймворка на языке Python, для удобной записи объектов Python непосредственно в базу данных без необходимости вручную записывать SQL-запросы. В результате внутреннего использования SQLAlchemy только поддерживаемые им базы данных также поддерживаются Airflow. Из всех поддерживаемых баз данных Airflow рекомендует использовать PostgreSQL или MySQL. Также можно применять SQLite, но только в сочетании с `SequentialExecutor`, поскольку она не поддерживает параллельную запись и поэтому не подходит для рабочей системы. Однако она очень удобна для тестирования и разработки благодаря простой настройке.

Без какой-либо конфигурации выполнив команду `airflow db init`, вы создаете базу данных SQLite в `$AIRFLOW_HOME/airflow.db`. Если вы хотите настроить рабочую систему и работать с MySQL или Postgres, то сначала должны отдельно создать базу данных. Далее нужно предоставить Airflow информацию о базе данных, задав значение для переменной окружения `AIRFLOW__CORE__SQL_ALCHEMY_CONN`.

Значение этого элемента конфигурации должно быть указано в формате URI (`protocol://[username:password@]host[:port]/path`). *См.* следующие примеры:

- MySQL: mysql://username:password@localhost:3306/airflow;
- PostgreSQL: postgres://username:password@localhost:5432/airflow.

Интерфейс командной строки Airflow предоставляет три команды для настройки базы данных:

- `airflow db init`: создает схему Airflow в пустой базе данных;
- `airflow db reset`: очищает существующую базу данных и создает новую. Это деструктивная операция!
- `airflow db upgrade`: применяет недостающие обновления схемы базы данных (если вы обновили версию Airflow) к базе данных. Выполнение этой команды на уже обновленной схеме базы данных не приведет ни к какому действию, и, следовательно, ее можно безопасно выполнять несколько раз. В случае если ни одна база данных не была инициализирована, эффект будет таким же, как и при использовании команды `airflow db init`. Обратите внимание, однако, что в отличие от `airflow db init` она не создает подключения по умолчанию.

Выполнение любой из этих команд даст примерно следующий вывод:

Листинг 12.1 Инициализация базы метаданных Airflow

```
$ airflow db init
DB: sqlite:////home/airflow/airflow.db
```

```
[2020-03-20 08:39:17,456] {db.py:368} INFO - Creating tables
INFO [alembic.runtime.migration] Context impl SQLiteImpl.
INFO [alembic.runtime.migration] Will assume non-transactional DDL.
... Running upgrade -> e3a246e0dc1, current schema
... Running upgrade e3a246e0dc1 -> 1507a7289a2f, create is_encrypted
...
```

То, что вы видите, – это вывод Alembic, еще одного фреймворка, используемого для миграции баз данных, написанного на Python. Каждая строка в листинге 12.1 – это вывод миграции одной-единственной базы данных. При обновлении до более новой версии Airflow, содержащей миграции базы данных (в примечаниях к выпуску указано, содержит ли новая версия обновления базы данных), необходимо также обновить соответствующую базу данных. Выполняя команду `airflow db upgrade`, вы проверяете, на каком этапе миграции находится ваша текущая база данных, и применяете этапы миграции, которые были добавлены в новом выпуске.

На данном этапе у вас есть полнофункциональная база данных Airflow, и вы можете выполнить команды `airflow webserver` и `airflow scheduler`. При открытии веб-сервера по адресу http://localhost:8080 вы увидите много примеров ОАГ и подключений (рис. 12.2).

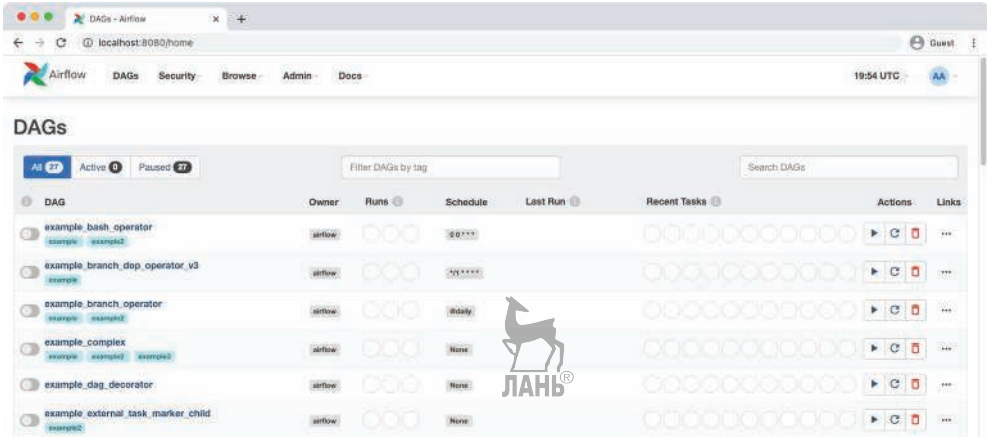


Рис. 12.2 По умолчанию Airflow загружает примеры ОАГ (и подключений, которые здесь не отображаются)

Эти примеры могут пригодиться во время разработки, но, вероятно, нежелательны для рабочей системы. Их можно исключить, задав для переменной окружения `AIRFLOW__CORE__LOAD_EXAMPLES` значение `False`.

Однако после перезапуска планировщика и веб-сервера вы, вероятно, будете удивлены, по-прежнему наблюдая ОАГ и подключения. Причина состоит в том, что когда вы задаете для вышеуказанной переменной значение `False`, то тем самым даете Airflow указание не загружать примеры ОАГ (не относится к подключениям!), и Airflow их не перезагружает. Однако уже загруженные ОАГ остаются в базе данных и не удаляются. Такое же поведение применяется к подключени-

ям по умолчанию, которые можно исключить, задав для переменной окружения `AIRFLOW__CORE__LOAD_DEFAULT_CONNECTIONS` значение `False`.

Учитывая это, «чистую» (т. е. без примеров) базу данных можно получить, выполнив следующие шаги:

1. установите Airflow;
2. задайте для переменной окружения `AIRFLOW__CORE__LOAD_EXAMPLES` значение `False`;
3. задайте для переменной окружения `AIRFLOW__CORE__LOAD_DEFAULT_CONNECTIONS` значение `False`;
4. выполните команду `airflow db init`.

12.1.3 Присмотримся к планировщику

Чтобы понять, как и когда выполняются задачи, подробнее рассмотрим планировщик. Планировщик выполняет несколько задач:

- анализ файлов ОАГ и сохранение извлеченной информации в базе данных;
- определение того, какие задачи готовы к выполнению, и их размещение в очереди;
- извлечение и выполнение задач в очереди.

Airflow выполняет все задачи в ОАГ в рамках *задания*. Хотя классы заданий являются внутренними по отношению к Airflow, запущенные задания можно просмотреть в его пользовательском интерфейсе. Планировщик также запускается в задании, хотя это и особое задание, а именно SchedulerJob. Все задания можно просмотреть в пользовательском интерфейсе Airflow в разделе **Browse > Jobs** (Просмотр > Задания) (рис. 12.3).

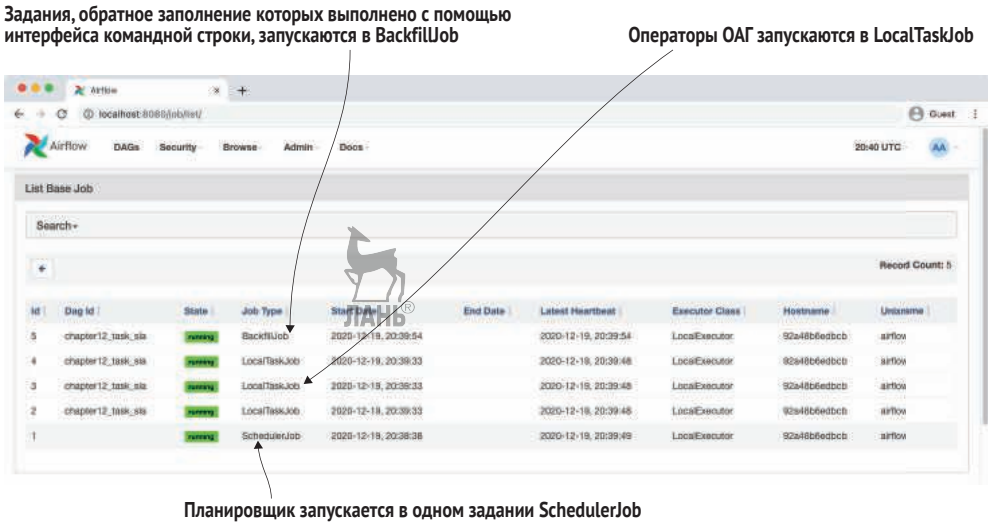


Рис. 12.3 Планировщик, обычные задачи и задачи обратного заполнения запускаются в рамках задания в Airflow. Все вакансии можно просмотреть в пользовательском интерфейсе Airflow

SchedulerJob делает три вещи. Во-первых, он отвечает за анализ файлов ОАГ и сохранение извлеченной информации в базе данных. Рассмотрим этот момент подробнее.

Процессор ОАГ

Планировщик Airflow периодически обрабатывает файлы Python в каталоге ОАГ (каталог, заданный `AIRFLOW__CORE__DAGS_FOLDER`). Это означает, что даже если в файл ОАГ не было внесено никаких изменений[1], он периодически проверяет каждый файл и сохраняет найденные ОАГ в базе метаданных Airflow, потому что вы можете создавать динамические ОАГ (эта структура изменений на основе внешнего источника в Airflow), в то время как код остается прежним. В качестве примера можно привести ОАГ, в котором считывается файл YAML и на основе его содержимого создаются задачи. Чтобы работать с изменениями в динамических ОАГ, планировщик периодически обрабатывает файлы.

Обработка ОАГ требует вычислительных мощностей. Чем больше вы повторно обрабатываете файлы, тем быстрее будет идти работа с изменениями, но за счет увеличения мощности процессора. Если вы знаете, что ваши ОАГ не изменяются динамически, то можно увеличить интервалы по умолчанию, чтобы уменьшить нагрузку на процессор, ничего не опасаясь. Интервал обработки ОАГ связан с четырьмя конфигурациями (*см.* табл. 12.2).

Таблица 12.2 Параметры конфигурации Airflow, связанные с обработкой ОАГ

Элемент конфигурации	Описание
`AIRFLOW__SCHEDULER__PROCESSOR_POLL_INTERVAL`	Время ожидания после завершения цикла планировщика. Внутри цикла (помимо других операций) осуществляет анализ ОАГ, поэтому чем меньше это число, тем быстрее будет выполняться анализ
`AIRFLOW__SCHEDULER__MIN_FILE_PROCESS_INTERVAL`	Минимальный интервал обработки файлов (по умолчанию 0). Обратите внимание: нет гарантии, что файлы будут обрабатываться с этим интервалом; это всего лишь нижняя граница, а не фактический интервал
`AIRFLOW__SCHEDULER__DAG_DIR_LIST_INTERVAL`	Минимальное время обновления списка файлов в папке ОАГ (по умолчанию 300). Уже перечисленные файлы хранятся в памяти и обрабатываются с другим интервалом. Обратите внимание, что данный параметр – это нижняя граница, а не фактический интервал
`AIRFLOW__SCHEDULER__PARSING_PROCESSES`	Максимальное количество процессов (не потоков), которое используется для анализа всех файлов ОАГ. Обратите внимание, что данный параметр – это нижняя граница, а не фактическое количество процессов

Оптимальная конфигурация для вашей системы зависит от количества ОАГ, их размера (то есть сколько времени требуется процессо-

[1] В то время как в сообществе Airflow продолжаются дискуссии по поводу того, чтобы сделать анализ ОАГ событийно-ориентированным, путем прослушивания изменений в файлах и явной настройки ОАГ для повторной обработки, если это необходимо, что могло бы облегчить использование ЦП планировщиком, на момент написания этой книги такой возможности пока не существует.

ру для их вычисления) и доступных ресурсов на машине, на которой запущен планировщик. Все интервалы определяют границу того, как часто следует выполнять процесс; время от времени значение интервала сравнивается, но возможно, например, что DAG_DIR_LIST_INTERVAL проверяется через 305 секунд, в то время как значение составляет 300 секунд.

AIRFLOW__SCHEDULER__DAG_DIR_LIST_INTERVAL особенно полезно для уменьшения значения. Если вы часто добавляете новые ОАГ и ждете их появления, проблему можно решить, уменьшив это значение.

Вся обработка ОАГ происходит в рамках цикла while True, в котором Airflow повторяет серию шагов для обработки файлов снова и снова. В файлах журнала вы увидите вывод обработки: /logs/dag_processor_manager/dag_processor_manager.log.

Листинг 12.2 Пример вывода диспетчера процессора ОАГ

```
================================================================
DAG File Processing Stats

File Path   PID Runtime # DAGs # Errors Last Runtime Last Run
----------- --- ------- ------ -------- ------------ ------------
.../dag1.py                 1        0  0.09s       ... 18:55:15
.../dag2.py                 1        0  0.09s       ... 18:55:15
.../dag3.py                 1        0  0.10s       ... 18:55:15
.../dag4.py 358 0.00s       1        0  0.08s       ... 18:55:15
.../dag5.py 359 0.07s       1        0  0.08s       ... 18:55:15
================================================================
... - Finding 'running' jobs without a recent heartbeat
... - Failing jobs without heartbeat after 2020-12-20 18:50:22.255611
... - Finding 'running' jobs without a recent heartbeat
... - Failing jobs without heartbeat after 2020-12-20 18:50:32.267603
... - Finding 'running' jobs without a recent heartbeat
... - Failing jobs without heartbeat after 2020-12-20 18:50:42.320578
```

Обратите внимание, что эта статистика обработки файлов выводится не с каждой итерацией, а каждые X секунд, которые заданы в качестве значения для AIRFLOW__SCHEDULER__PRINT_STATS_INTERVAL (по умолчанию 30 секунд). Также обратите внимание, что отображаемая статистика представляет информацию из последнего запуска, а не результаты последнего количества секунд PRINT_STATS_INTERVAL.

Планировщик заданий

Планировщик отвечает за определение того, какие экземпляры задачи могут быть выполнены. Цикл while True периодически проверяет для каждого экземпляра задачи, выполняется ли набор условий, например (среди прочего) удовлетворены ли все вышестоящие зависимости, достигнут ли конец интервала, успешно ли запущен экземпляр задачи в предыдущем ОАГ, если для depends_on_past задано значение True, и так далее. Всякий раз, когда экземпляр задачи соответствует

всем условиям, он устанавливается в запланированное состояние, а это означает, что планировщик решил, что он удовлетворяет всем условиям и готов к выполнению.

Еще один цикл в планировщике определяет другой набор условий, при которых задачи переходят из запланированного в состояние очереди. Здесь условия включают в себя (среди прочего) проверку на предмет того, достаточно ли открытых слотов и имеют ли одни задачи приоритет над другими (с учетом аргумента `priority_weight`). После выполнения всех этих условий планировщик помещает команду в очередь для запуска задачи и задает для состояния экземпляра задачи значение queued. Это означает, что после того, как экземпляр задачи был помещен в очередь, планировщик больше не несет за него ответственности. На данном этапе ответственность за задачи теперь лежит на исполнителе, который будет читать экземпляр задачи из очереди и запускать задачу в воркере.

ПРИМЕЧАНИЕ Планировщик задач отвечает за задачи вплоть до помещения их в очередь. После задача становится обязанностью исполнителя, который будет запускать ее.

Тип очереди и способ обработки экземпляра задачи, после того как она была помещена в очередь, содержится в процессе, который называют *исполнитель*. Эту часть планировщика можно настроить различными способами, начиная от одного процесса на одной машине и заканчивая несколькими процессами, распределенными по нескольким машинам, как описано в разделе 12.1.1.

Исполнитель задач

Как правило, исполнитель задачи будет ждать, пока планировщик разместит экземпляры задач для выполнения в очереди. После помещения в очередь исполнитель извлекает экземпляр задачи из очереди и выполняет его. Airflow регистрирует каждое изменение состояния в базе метаданных. Сообщение, помещенное в очередь, содержит несколько деталей экземпляра задачи. В исполнителе *выполнение* задач означает создание нового процесса для задачи, которую нужно запустить, чтобы Airflow не отключился, если что-то пойдет не так. В новом процессе он выполняет команду `airflow tasks` для запуска одного экземпляра задачи, как показано в следующем примере (используется LocalExecutor).

Листинг 12.3 Команда, выполняемая для любой заданной задачи

```
➥ airflow tasks run [dag_id] [task_id] [execution date] -local -pool [pool
    id] -sd [dag file path]

For example:
➥ airflow tasks run chapter12_task_sla sleeptask 2020-04-04T00:00:00+00:00
    --local --pool default_pool -sd /..../dags/chapter12/task_sla.py
```

Прямо перед выполнением команды Airflow регистрирует состояние экземпляра задачи как запущенное в базе метаданных. После этого он выполняет задачу и периодически проверяет, отправляя контрольный сигнал в эту базу. Контрольный сигнал – это еще один цикл `while True`, в котором Airflow выполняет следующие действия:

- проверяет, завершена ли задача;
- если она завершена и код завершения равен нулю, то задача выполнена успешно;
- если она завершена и код завершения не равен нулю, то задача не выполнена;
- если она не завершена,
  - регистрируем контрольный сигнал и ждем *X* секунд. Это значение задается с помощью переменной окружения `AIRFLOW__SCHEDULER__JOB_HEARTBEAT_SEC` (по умолчанию 5);
  - повторяем.

Для успешной задачи данный процесс повторяется определенное количество раз, пока задача не будет завершена. Если ошибок не произошло, состояние задачи меняется на успешное. Идеальная последовательность выполнения задачи изображена на рис. 12.4.

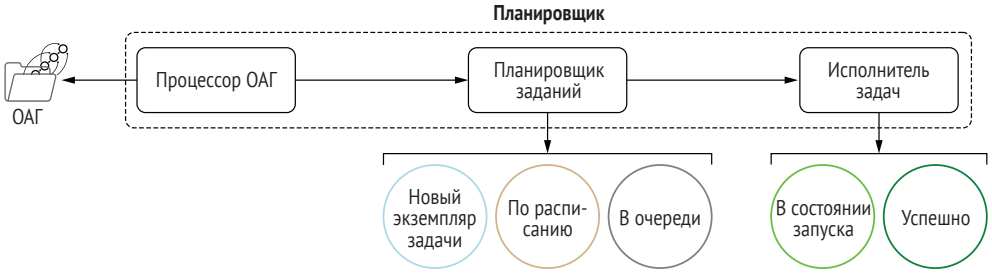


Рис. 12.4 Идеальный поток и состояние задачи, за которую отвечают компоненты планировщика. Пунктирная линия обозначает полную ответственность планировщика. При запуске режима `SequentialExecutor`/`LocalExecutor` это единый процесс. `CeleryExecutor` и `KubernetesExecutor` запускают исполнитель задач в отдельных процессах, предназначенных для масштабирования на нескольких машинах

12.2 Установка исполнителей

Есть много способов установить и настроить Airflow; следовательно, нецелесообразно разговаривать обо всех в этой книге. Однако мы продемонстрируем основные элементы, необходимые для подготовки каждого исполнителя к работе.

Как уже объяснялось в разделе 12.1, исполнитель является частью планировщика Airflow. Процессор ОАГ и планировщик задач можно запустить только одним способом, выполнив команду `airflow scheduler`. Однако исполнителя задач можно установить разными спосо-

бами, начиная от одного процесса на одной машине и заканчивая несколькими процессами на нескольких машинах для повышения производительности и/или избыточности.

Тип исполнителя задается с помощью переменной окружения `AIRFLOW__CORE__EXECUTOR`, с использованием одного из следующих значений:

- `SequentialExecutor` (по умолчанию);
- `LocalExecutor`;
- `CeleryExecutor`;
- `KubernetesExecutor`.

Правильность установки любого исполнителя можно проверить, запустив ОАГ. Если какая-либо задача переходит в состояние выполнения, это означает, что она прошла цикл планирования, постановки в очередь и выполнения и ей занимается исполнитель.

12.2.1 Настройка SequentialExecutor

Исполнителем по умолчанию в Airflow является `SequentialExecutor` (рис. 12.5). Исполнитель задач запускается в одном подпроцессе, в рамках которого выполняются задачи одна за другой, поэтому это самый медленный метод выполнения задач. Однако он удобен для тестирования, потому что не требует настройки.

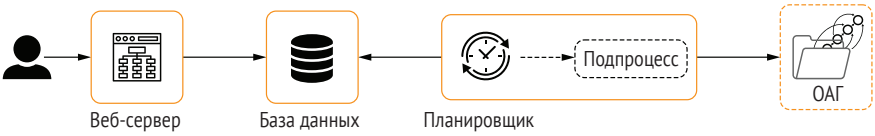


Рис. 12.5 При использовании `SequentialExecutor` все компоненты должны работать на одной машине

`SequentialExecutor` работает с базой данных SQLite. При выполнении команды `airflow db init` без какой-либо конфигурации вы инициализируете базу данных SQLite в каталоге `$AIRFLOW_HOME`, который представляет собой единственный файл airflow.db. После этого запускаем два процесса:

- airflow scheduler;
- airflow webserver.

12.2.2 Настройка LocalExecutor

Настройка Airflow с помощью `LocalExecutor` не сильно отличается от настройки `SequentialExecutor` (рис. 12.6). Его архитектура похожа на `SequentialExecutor`, только здесь имеется несколько подпроцессов, поэтому задачи могут выполняться параллельно, и, таким образом, он выполняется быстрее. Каждый подпроцесс выполняет одну задачу, и подпроцессы можно запускать параллельно.

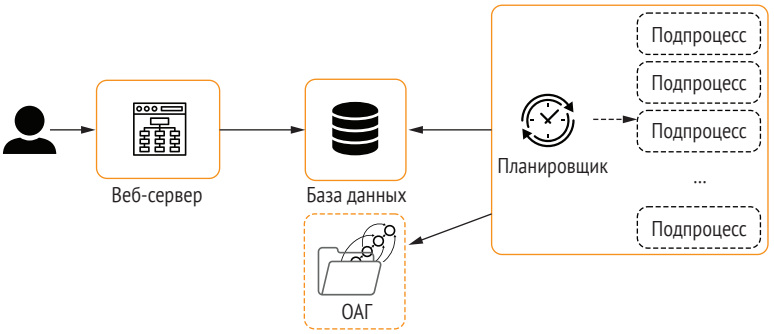


Рис. 12.6 При использовании `LocalExecutor` все компоненты могут работать на отдельной машине. Однако все подпроцессы, созданные планировщиком, выполняются на одной машине

Кроме того, `SequentialExecutor` привязан к базе данных SQLite, в то время как все остальные исполнители могут работать с более сложными базами данных, такими как MySQL и PostgreSQL, что обеспечивает лучшую производительность.

Чтобы настроить `LocalExecutor`, задайте для переменной окружения `AIRFLOW__CORE__EXECUTOR` значение `LocalExecutor`. Планировщик может порождать максимальное количество подпроцессов, настроенных `AIRFLOW__CORE__PARALLELISM` (по умолчанию их 32). С технической точки зрения это не новые процессы, а скорее процессы, отходящие от родительского процесса (планировщика).

Есть и другие способы ограничить количество параллельных задач (например, уменьшив размер пула по умолчанию, `AIRFLOW__CORE__DAG_CONCURRENCY` или `AIRFLOW__CORE__MAX_ACTIVE_RUNS_PER_DAG`).

Что касается базы данных, установите Airflow с дополнительными зависимостями для соответствующей системы базы данных:

- MySQL: `pip install apache-airflow[mysql]` ;
- PostgreSQL: `pip install apache-airflow[postgres]`.

`LocalExecutor` прост в настройке и может обеспечить достойную производительность. Система ограничена ресурсами машины планировщика. Если `LocalExecutor` больше вам не подходит (например, с точки зрения производительности или избыточности), то следующий логический этап – это `CeleryExecutor` и `KubernetesExecutor`, которые мы рассмотрим в разделах 12.2.3 и 12.2.4.

12.2.3 Настройка CeleryExecutor

`CeleryExecutor` создан на базе проекта Celery. Celery предоставляет фреймворк для рассылки сообщений воркерам через систему очередей (рис. 12.7).

Как видно на рис. 12.7, и планировщику, и воркерам Celery требуется доступ к ОАГ и базе данных. Что касается базы данных, то это не проблема, поскольку к ней можно подключиться с помощью клиента.

А вот что касается папки OAG, здесь могут возникнуть сложности с настройкой. Вы делаете OAG доступными для всех машин либо через общую файловую систему, либо используя контейнеризацию, когда OAG встраиваются в образ с помощью Airflow. При использовании контейнеризации любое изменение в коде OAG приведет к повторному развертыванию программного обеспечения.

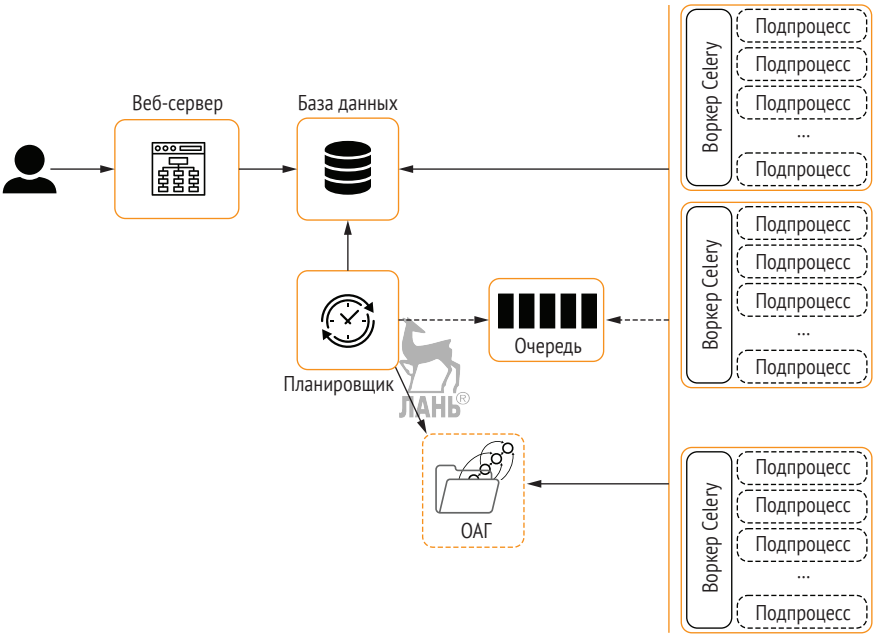


Рис. 12.7 В `CeleryExecutor` задачи распределяются между несколькими машинами, на которых работают воркеры Celery. Воркеры ждут поступления задач в очередь

Чтобы начать работу с Celery, сначала установите Airflow с дополнительными зависимостями Celery и выполните настройку исполнителя:

- `pip install apache-airflow[celery]`;
- `AIRFLOW__CORE__EXECUTOR=CeleryExecutor`.

На момент написания книги Celery использует Redis, RabbitMQ и AWS SQS для работы с очередями сообщений. В Celery очередь называется *брокером*. Установка брокера выходит за рамки этой книги, но после инсталляции вы должны задать его, используя для этого переменную окружения `AIRFLOW__CELERY__BROKER_URL`:

- Redis: `AIRFLOW__CELERY__BROKER_URL=redis://localhost:6379/0`;
- RabbitMQ: `AIRFLOW__CELERY__BROKER_URL=amqp://user:pass@localhost:5672//`.

Соответствующий формат URI *см.* в документации к программному обеспечению, которое вы используете. `BROKER_URL` позволяет

планировщику отправлять сообщения в очередь. Чтобы воркеры Celery могли обмениваться данными с базой метаданных Airflow, также нужно настроить переменную `AIRFLOW__CELERY__RESULT_BACKEND`. В Celery префикс `db+` используется для обозначения подключения к базе данных:

- MySQL: `AIRFLOW__CELERY__RESULT_BACKEND=db+mysql://user:pass@localhost/airflow`;
- PostgreSQL: `AIRFLOW__CELERY__RESULT_BACKEND=db+postgresql://user:pass@localhost/airflow`.

Убедитесь, что папка ОАГ также доступна на машинах, где функционируют воркеры, по тому же пути, как настроено в `AIRFLOW__CORE__DAGS_FOLDER`. После этого вы должны быть готовы:

1. запустить веб-сервер Airflow;
2. запустить планировщик;
3. запустить воркер Celery.

`airflow celery worker` – это небольшая команда-оболочка, запускающая воркер Celery. Теперь все должно быть готово и работать.

ПРИМЕЧАНИЕ Чтобы проверить установку, можно запустить ОАГ вручную. Если какая-либо задача завершится успешно, она пройдет через все компоненты установки `CeleryExecutor`, а это означает, что все работает, как и было задумано.

Чтобы отслеживать состояние системы, можно настроить Flower, веб-инструмент мониторинга, в котором можно проверять (среди прочего) воркеры, задачи и состояние всей системы Celery. Интерфейс командной строки Airflow также предоставляет удобную команду для запуска Flower: `airflow celery flower`. По умолчанию Flower работает на порту 5555. После запуска перейдите по адресу http://localhost: 5555 (рис. 12.8).

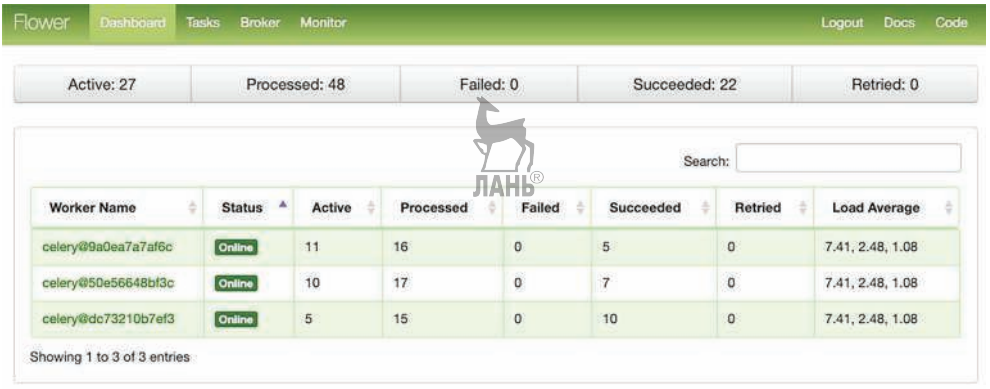


Рис. 12.8 Панель управления Flower показывает состояние всех воркеров Celery

В первом представлении Flower мы видим количество зарегистрированных воркеров Celery, их статус и информацию высокого уровня о количестве задач, обработанных каждым воркером. Как узнать, хорошо ли работает система? Самая полезная страница в интерфейсе Flower – это страница мониторинга на рис. 12.9, где показано состояние системы на нескольких графиках.

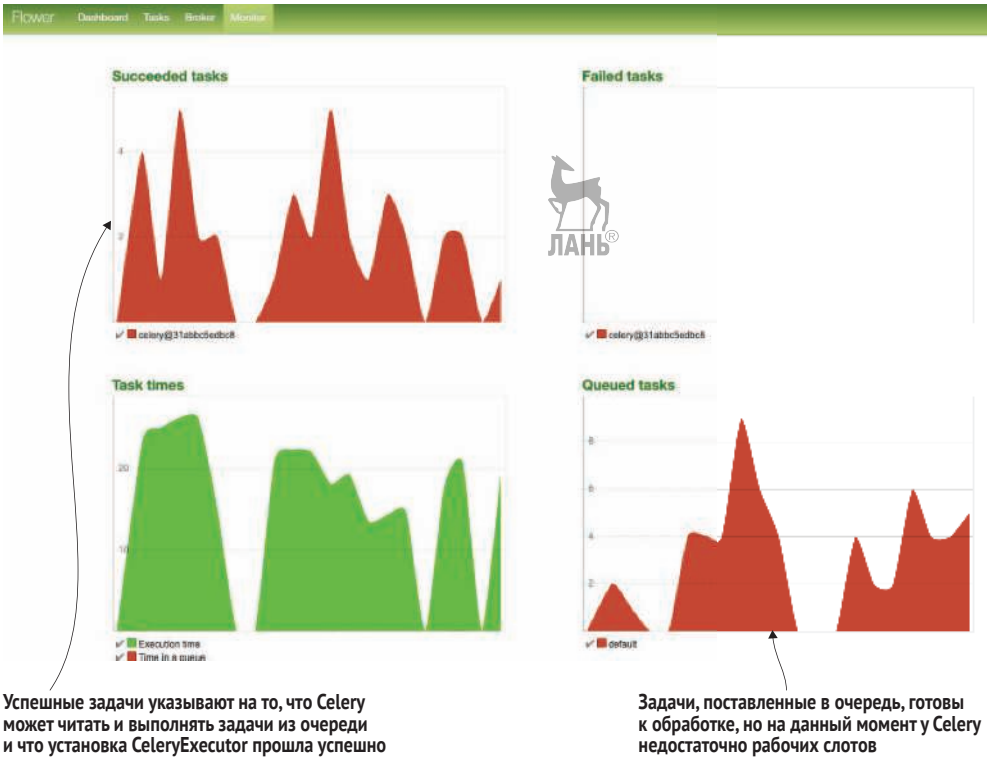


Рис. 12.9 Вкладка мониторинга Flower показывает графики, что помогает получить представление о производительности системы Celery

Из двух распределенных режимов исполнителя, предлагаемых Airflow (Celery и Kubernetes), `CeleryExecutor` легче настроить с нуля, потому что вам нужно настроить только еще один дополнительный компонент: очередь. Воркеры Celery и панель управления Flower интегрированы в Airflow, что упрощает настройку и масштабирование выполнения задач на нескольких машинах.

12.2.4 Настройка KubernetesExecutor

И последний, но не менее важный исполнитель – это `KubernetesExecutor`. Чтобы его использовать, задайте для переменной окружения `AIRFLOW__CORE__EXECUTOR` значение `KubernetesExecutor`. Как следует из

названия, данный тип исполнителя привязан к Kubernetes – наиболее часто используемой системе для запуска и управления программным обеспечением в контейнерах. Многие компании запускают свое программное обеспечение на Kubernetes, поскольку контейнеры предоставляют изолированное окружение, которое гарантирует, что то, что вы разрабатываете у себя на компьютере, также будет работать и в рабочей системе. Поэтому сообщество Airflow заявило о своем твердом намерении использовать Kubernetes для запуска Airflow. С точки зрения архитектуры `KubernetesExecutor` выглядит так, как показано на рис. 12.10.

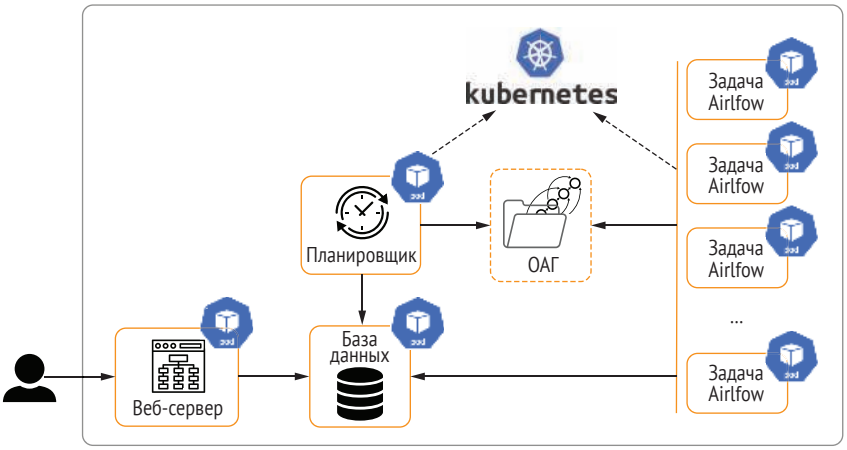


Рис. 12.10 В случае с `KubernetesExecutor` все задачи выполняются в поде Kubernetes. Хотя в Kubernetes нет необходимости запускать веб-сервер, планировщик и базу данных, при использовании KubernetesExecutor будет разумно сделать это

При работе с `KubernetesExecutor` полезно иметь какие-либо предварительные знания Kubernetes. Kubernetes может быть большим и сложным. Однако `KubernetesExecutor` использует лишь небольшую часть всех доступных компонентов на платформе Kubernetes.

На данный момент полезно знать, что *под* – это самая маленькая рабочая единица в Kubernetes, которая может запускать один или несколько контейнеров. В контексте Airflow одна задача будет выполняться в одном поде.

Под создается каждый раз при выполнении задачи. Когда планировщик решает запустить задачу, он отправляет запрос на создание пода в API Kubernetes, который затем создает под, на котором работает контейнер Airflow, используя команду `airflow tasks run ...`, как показано в листинге 12.3 (без учета ряда деталей). Kubernetes сам отслеживает состояние пода.

С настройками других исполнителей между физическими машинами было четкое разделение. В Kubernetes все процессы выполняются

в подах, где их можно распределить на нескольких машинах, хотя они также могут работать и на одной машине. С точки зрения пользователя, процессы выполняются в подах, и пользователь не знает о базовых машинах.

Самый распространенный способ развертывания программного обеспечения в Kubernetes – использовать Helm, менеджер пакетов для Kubernetes. Различные сторонние диаграммы Helm для Airflow доступны в Helm Hub, хранилище диаграмм Helm. На момент написания книги официальная диаграмма для Airflow доступна в главной ветви проекта Airflow, однако ее еще нет в открытых репозиториях Helm. Поэтому минимальные инструкции по установке (при условии что у вас есть функционирующий кластер Kubernetes и Helm версии 3 или выше) выглядят следующим образом:

Листинг 12.4 Установка Airflow в Kubernetes с помощью диаграммы Helm

Создаем пространство имен Airflow в Kubernetes

Скачиваем исходный код Airflow, содержащий диаграмму Helm

Скачиваем указанные версии зависимых диаграмм Helm

Устанавливаем диаграмму

```
$ curl -OL https://github.com/apache/airflow/archive/master.zip
$ unzip master.zip
$ kubectl create namespace airflow
$ helm dep update ./airflow-master/chart
$ helm install airflow ./airflow-master/chart --namespace airflow

NAME: airflow
LAST DEPLOYED: Wed Jul 22 20:40:44 2020
NAMESPACE: airflow
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
Thank you for installing Airflow!

Your release is named airflow.

➥ You can now access your dashboard(s) by executing the following
command(s) and visiting the corresponding port at localhost in your browser:

Airflow dashboard:
➥ kubectl port-forward svc/airflow-webserver 8080:8080 --namespace airflow
```

Одна из самых сложных частей настройки `KubernetesExecutor` – определить, как распространять файлы ОАГ между процессами Airflow. Для этого есть три метода:

1. использовать ОАГ в подах с помощью PersistentVolume;
2. извлечь последний код ОАГ из репозитория;
3. встроить ОАГ в образ Docker.

Сперва выясним, как развернуть код ОАГ без использования контейнеров. Все процессы Airflow должны иметь доступ к каталогу,

содержащему файлы ОАГ. На одной машине это не так уж сложно: запустите все процессы Airflow и пропишите путь к каталогу, содержащему код ОАГ.

Однако при запуске процессов Airflow на разных машинах все уже не так просто. В этом случае вам нужен способ сделать код ОАГ доступным для обеих машин, например используя общую файловую систему (рис. 12.11).

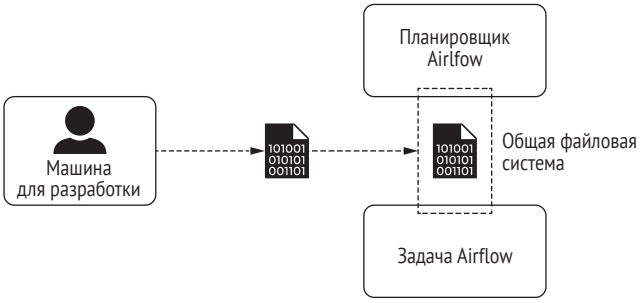


Рис. 12.11 Без контейнеров разработчик помещает код в репозиторий, после чего нужно сделать так, чтобы код был доступен для обоих процессов Airflow

Однако получить код в общей файловой системе – задача непростая. Файловая система предназначена для хранения и извлечения файлов на носителе, а не для предоставления интерфейса в сети для простого обмена файлами. Обмен файлами через интернет будет обрабатываться приложением, работающим на том же компьютере, где смонтирована файловая система.

Выражаясь более практично, скажем, у вас есть общая файловая система, такая как *NFS* (сетевая файловая система) для обмена файлами между планировщиком Airflow и машинами, где функционируют воркеры. Вы пишете код у себя на компьютере, но не можете копировать файлы напрямую в NFS, потому что у нее нет интерфейса для доступа в сеть. Чтобы скопировать файлы в NFS, нужно смонтировать ее на машине, и файлы должны записываться туда через приложение, работающее на том же компьютере, например FTP (рис. 12.12).

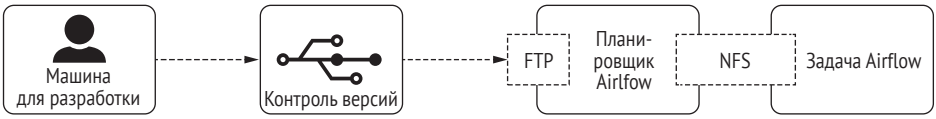


Рис. 12.12 Файлы нельзя записывать напрямую в NFS, потому что у нее нет интернет-интерфейса. Для отправки и получения файлов по сети можно было бы использовать FTP, чтобы хранить файлы на том же компьютере, куда монтируется NFS

На рис. 12.12 разработчик или система непрерывной интеграции и доставки могут отправить код Airflow в систему Airflow через FTP-

сервер, который работает на одной из машин. Через FTP-сервер NFS-том должен быть доступен для системы непрерывной интеграции и доставки, чтобы отправлять файлы ОАГ и делать их доступными для всех машин Airflow.

Что делать, если такой механизм не подходит? Это распространенная проблема по разным причинам, например из-за ограничений безопасности или сети. В этом случае часто используется следующее решение – скачать код из машины Airflow, используя «DAG puller».

Листинг 12.5 Скачивание последней версии кода с помощью DAG puller

```
import datetime

from airflow.models import DAG
from airflow.operators.bash import BashOperator

dag = DAG(
    dag_id="dag_puller",
    default_args={"depends_on_past": False},
    start_date=datetime.datetime(2020, 1, 1),
    schedule_interval=datetime.timedelta(minutes=5),
    catchup=False,
)

fetch_code = BashOperator(
    task_id="fetch_code",
    bash_command=(
        "cd /airflow/dags && "
        "git reset --hard origin/master"
    ),
    dag=dag,
)
```

Игнорируем все зависимости; всегда запускаем задачи

Скачиваем последний код каждые пять минут

Требует установки и настройки Git

С помощью DAG puller последний код из главной ветви загружается в машину Airflow каждые пять минут (рис. 12.13). Это, очевидно, приводит к задержке между кодом в главной ветке и развертыванием кода в Airflow, но иногда это наиболее практичное решение.

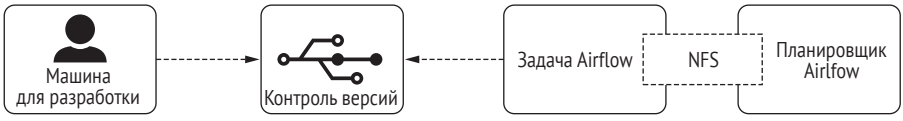


Рис. 12.13 Используя DAG puller, вы извлекаете код из машины Airflow

Теперь, когда мы знаем проблемы и потенциальные решения для развертывания ОАГ с Airflow в распределенной настройке, посмотрим, как использовать ОАГ с подами в Kubernetes.

Использование ОАГ с подами с помощью PersistentVolume

PersistentVolume – это абстракция Kubernetes поверх хранилища, позволяющая монтировать общие тома в контейнеры без необходимости знать базовую технологию, такую как NFS, Azure File Storage или AWS EBS. Один из самых сложных моментов – настроить конвейер непрерывной интеграции и доставки, где код ОАГ передается в общий том, который обычно не предоставляет готовых функций для прямой отправки на общий том. Чтобы разрешить совместное использование ОАГ с PersistentVolume, задайте для элемента конфигурации `AIRFLOW__KUBERNETES__DAGS_VOLUME_CLAIM` значение в виде имени тома (запрос на выделение тома в Kubernetes) в поде Airflow. Код ОАГ необходимо скопировать на том либо с помощью метода, показанного на рис. 12.12, либо используя способ, показанный в листинге 12.5. Решение может зависеть от выбранного вами типа тома, поэтому обратитесь к документации по Kubernetes для получения дополнительной информации о томах.

Извлекаем последний код ОАГ из репозитория с помощью init-контейнера синхронизации с Git

Конфигурация Airflow содержит список настроек для извлечения репозитория Git посредством паттерна Sidecar (Прицеп) перед запуском задачи Airflow (это не полный список):

- `AIRFLOW__KUBERNETES__GIT_REPO = https://mycompany.com/repository/airflow`;
- `AIRFLOW__KUBERNETES__GIT_BRANCH = master`;
- `AIRFLOW__KUBERNETES__GIT_SUBPATH = dags`;
- `AIRFLOW__KUBERNETES__GIT_USER = username`;
- `AIRFLOW__KUBERNETES__GIT_PASSWORD = password`;
- `AIRFLOW__KUBERNETES__GIT_SSH_KEY_SECRET_NAME = airflow-secrets`;
- `AIRFLOW__KUBERNETES__GIT_DAGS_FOLDER_MOUNT_POINT = /opt/airflow/dags`;
- `AIRFLOW__KUBERNETES__GIT_SYNC_CONTAINER_REPOSITORY = k8s.gcr.io/git-sync`;
- `AIRFLOW__KUBERNETES__GIT_SYNC_CONTAINER_TAG = v3.1.2`;
- `AIRFLOW__KUBERNETES__GIT_SYNC_INIT_CONTAINER_NAME = git-sync-clone`.

Хотя не нужно заполнять все данные, если задать значение для `GIT_REPO` и учетные данные (`USER` + `PASSWORD` или `GIT_SSH_KEY_SECRET_NAME`), то вы активируете синхронизацию с Git. Airflow создаст контейнер синхронизации, который извлекает код из сконфигурированного репозитория перед запуском задачи.

Встраивайте ОАГ в образ Docker

Наконец, встраивание файлов ОАГ в образ Airflow также является популярным вариантом для его неизменности; любое изменение фай-

лов ОАГ приводит к созданию и развертыванию нового образа Docker, чтобы вы всегда были уверены в том, с какой версией кода вы работаете. Для сообщения `KubernetesExecutor`, что вы встроили файлы ОАГ в образ, задайте для переменной `AIRFLOW__KUBERNETES__DAGS_IN_IMAGE` значение `True`.

Процесс сборки и развертывания теперь выглядит немного иначе (рис. 12.14).

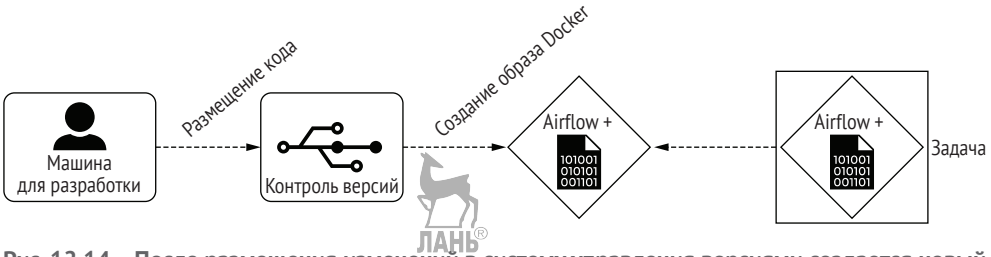


Рис. 12.14 После размещения изменений в систему управления версиями создается новый образ Docker

Создание образа Airflow вместе с кодом ОАГ дает несколько преимуществ:

- мы точно знаем, какая версия кода сейчас развернута;
- мы можем запускать Airflow локально и в промышленном окружении;
- конфликты между новыми зависимостями обнаруживаются во время сборки, а не во время выполнения.

Если увеличить масштаб сборки, то для повышения производительности предпочтительнее установить Airflow и добавить код ОАГ в два отдельных этапа:

1. зависимости установки;
2. добавьте только код ОАГ.

Причина этого разделения состоит в том, что Airflow содержит множество зависимостей, создание которых может занять несколько минут. Вероятно, вы не будете часто менять зависимости во время разработки, а в основном будете изменять код ОАГ. Чтобы избежать повторной установки зависимостей при каждом небольшом изменении, скопируйте код своего ОАГ в образ на отдельном этапе. Если ваша система непрерывной интеграции и доставки кеширует слои Docker, это можно сделать в отдельном операторе Docker, потому что вы можете быстро получить базовые слои. Если ваша система не кеширует слои Docker, то разумно будет создать один базовый образ для Airflow и зависимостей и второй образ только для добавления туда кода ОАГ. Проиллюстрируем, как последний вариант работает с двумя файлами Dockerfile[1]. Первый файл – это базовый файл Dockerfile.

[1] Оба файла Dockerfile предназначены для демонстрационных целей.

Листинг 12.6 Пример базового файла Dockerfile

```
FROM apache/airflow:2.0.0-python3.8

USER root

RUN apt-get update && \
    apt-get install -y gcc && \
    apt-get autoremove -y && \
    apt-get clean -y && \
    rm -rf /var/lib/apt/lists/*

USER airflow
COPY requirements.txt /opt/airflow/requirements.txt
RUN pip install --user -r /opt/airflow/requirements.txt && \
    rm /opt/airflow/requirements.txt
```

На основе официального образа Airflow

Пользователь по умолчанию – это непривилегированный пользователь Airflow, поэтому для установки переключитесь на root

Переключитесь обратно на Airflow после установки

Этот базовый файл Dockerfile начинается с официального образа Docker для Airflow версии 2.0.0 и устанавливает дополнительные зависимости, перечисленные в файле requirements.txt. Наличие отдельного файла дополнительных зависимостей упрощает конвейер непрерывной интеграции и доставки, поскольку любое изменение в этом файле всегда должно запускать повторное создание базового образа. Команда `docker build -f Dockerfile.base -t myrepo/airflow-base.` создаст базовый образ.

Листинг 12.7 Финальный пример файла Dockerfile

```
FROM myrepo/airflow-base:1.2.3

COPY dags/ /opt/airflow/dags/
```

Наличие предварительно созданного базового образа со всеми зависимостями превращает создание окончательного образа в очень быстрый процесс, поскольку единственный необходимый шаг – это копирование файлов ОАГ. Чтобы создать его, используйте команду `docker build -t myrepo/airflow.` Однако этот образ будет создаваться с каждым изменением. В зависимости от устанавливаемых зависимостей разница во времени между сборкой базового и окончательного образов может быть очень большой.

Листинг 12.8 Пример файла requirements.txt

```
python-dotenv~=0.10.3
```

Разделив процесс сборки образа Docker на отдельные операторы или отдельные образы, можно значительно ускорить время сборки, поскольку в образ Docker копируются только наиболее часто изменяемые файлы (скрипты ОАГ). Более трудоемкое и полное повторное создание образа Docker будет выполняться только при необходимости.

Со стороны Kubernetes убедитесь, что тег образа Airflow определен в YAML с помощью AIRFLOW__KUBERNETES__POD_TEMPLATE_FILE, или убедитесь, что для переменной AIRFLOW__KUBERNETES__WORKER_CONTAINER_TAG в качестве значения задан тег, который должен быть развернут подами. Если вы используете диаграмму Helm, то можете обновить развернутую версию с помощью интерфейса командной строки Helm, задав тег только что созданного образа.

Листинг 12.9 Обновление развернутого образа Airflow с помощью Helm

```
helm upgrade airflow ./airflow-master/chart \
  --set images.airflow.repository=yourcompany/airflow \
  --set images.airflow.tag=1234abc
```

12.3 Работа с журналами всех процессов Airflow

А как насчет журналирования? Все системы производят какой-то вывод, и иногда нам нужно знать, что происходит. В Airflow есть три типа журналов:

- *журналы веб-сервера* – содержат информацию об активности в сети, т. е. какие запросы отправляются на веб-сервер;
- *журналы планировщика* – содержат информацию обо всех действиях планировщика, включая анализ ОАГ, планирование задач и многое другое;
- *журналы задач* – содержат журналы одного экземпляра задачи в каждом файле журнала.

По умолчанию журналы записываются в каталог $AIRFLOW_HOME/logs в локальной файловой системе. Журналирование настраивается различными способами. В этом разделе мы продемонстрируем вариант по умолчанию, а также покажем, как писать журналы в удаленную систему хранения в разделе 12.3.4.

12.3.1 Вывод веб-сервера

Веб-сервер обслуживает статические файлы, и каждый запрос к файлу отображается в выводе веб-сервера. *См.* следующий пример:

- ➥ 127.0.0.1 - - [24/Mar/2020:16:50:45 +0100] "GET / HTTP/1.1" 302 221 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.149 Safari/537.36";
- ➥127.0.0.1--[24/Mar/2020:16:50:46+0100]"GET/admin/HTTP/1.1" 200 44414 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_5) AppleWeb-Kit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.149 Safari/537.36";

- ➥ 127.0.0.1 - - [24/Mar/2020:16:50:46 +0100] "GET /static/bootstrap-theme.css HTTP/1.1" 200 0 "http://localhost:8080/admin/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.149 Safari/537.36".

При запуске веб-сервера из командной строки вы увидите, что этот вывод будет выведен в stdout или stderr. Что делать, если вы хотите сохранить журналы после отключения веб-сервера? На веб-сервере существует два типа журналов: журналы доступа, как показано, и журналы ошибок, которые содержат не только ошибки, но и системную информацию, например:

- [2020-04-13 12:22:51 +0200] [90649] [INFO] Listening at: http://0.0.0.0:8080 (90649);
- [2020-04-13 12:22:51 +0200] [90649] [INFO] Using worker: sync;
- [2020-04-13 12:22:51 +0200] [90652] [INFO] Booting worker with pid: 90652.

Оба типа журналов можно записать в файл, указав параметр при запуске веб-сервера Airflow:

- airflow webserver --access_logfile [filename];
- airflow webserver --error_logfile [filename].

Имя файла будет относиться к AIRFLOW_HOME, поэтому, если, например, задать «accesslogs.log» в качестве имени файла, то вы создадите файл: /path/to/airflow/home/accesslogs.log.

12.3.2 Вывод планировщика

По умолчанию планировщик записывает журналы в файлы, в отличие от веб-сервера. Если снова посмотреть на каталог $AIRFLOW_HOME/logs, то можно увидеть различные файлы, связанные с журналами планировщика.

Листинг 12.10 Файлы журнала, сгенерированные планировщиком

```
.
├── dag_processor_manager
│   └── dag_processor_manager.log
└── scheduler
    └── 2020-04-14
        ├── hello_world.py.log
        └── second_dag.py.log
```

Это дерево каталогов – результат обработки двух ОАГ: hello_world и second_dag. Каждый раз, когда планировщик обрабатывает файл ОАГ, несколько строк записываются в соответствующий файл. Эти строки – ключ к пониманию того, как работает планировщик. Посмотрим на файл hello_world.py.log.

Листинг 12.11 Планировщик, читающий файлы ОАГ и создающий соответствующие ОАГ или задачи

```
... Started process (PID=46) to work on /opt/airflow/dags/hello_world.py
... Processing file /opt/airflow/dags/hello_world.py for tasks to queue
... Filling up the DagBag from /opt/airflow/dags/hello_world.py
➥ ... DAG(s) dict_keys(['hello_world']) retrieved from
    /opt/airflow/dags/hello_world.py
... Processing hello_world
... Created <DagRun hello_world @ 2020-04-11 00:00:00 ...>
... Examining DAG run <DagRun hello_world @ 2020-04-11 00:00:00 ...>
➥ ... Skipping SLA check for <DAG: hello_world> because no tasks in DAG have
    SLAs
➥ ... Creating / updating <TaskInstance: hello_world.hello 2020-04-11 ...>
    in ORM
➥ ... Creating / updating <TaskInstance: hello_world.world 2020-04-11 ...>
    in ORM
... Processing /opt/airflow/dags/hello_world.py took 0.327 seconds
```

ОАГ hello_world был получен из файла

Начинаем обработку этого файла

Создан DagRun, потому что достигнут конец интервала

Проверяем, нужно ли отправлять пропущенные SLA-уведомления

Проверяем, запущен ли ОАГ, можно ли создать соответствующие экземпляры задач с учетом их расписания и не были ли пропущены какие-либо SLA

Обработка этого файла завершена

Проверяем, есть ли задачи, которые нужно создать и установить их в запланированное состояние

Проверяем, нужно ли запустить какие-либо существующие экземпляры задач

Эти этапы обработки файла ОАГ, загрузки объекта ОАГ из файла и проверки выполнения множества условий, таких как расписания, выполняются многократно и являются частью основных функций планировщика. На основании этих журналов можно определить, работает ли планировщик должным образом.

Также существует файл dag_processor_manager.log (ротация журналов выполняется при достижении показателя в 100 МБ), в котором отображается агрегированное представление (по умолчанию последние 30 секунд) того, какие файлы обработал планировщик.

12.3.3 Журналы задач

Наконец, у нас есть журналы задач, где каждый файл представляет собой одну попытку одной задачи.

Листинг 12.12 Файлы журнала, созданные при выполнении задачи

```
.
├── hello_world  <-- Имя ОАГ
│   ├── hello  <-- Название задачи
│   │   └── 2020-04-14T16:00:00+00:00  <-- Дата выполнения
│   │       ├── 1.log  <-- Номер попытки
│   │       └── 2.log
```

```
│ ├── world
│ │ └──   2020-04-14T16:00:00+00:00
│ │       ├──  1.loge
│ │       └──  2.log
└── second_dag
    └── print_context
        ├── 2020-04-11T00:00:00+00:00
│       └── 1.log
└──   2020-04-12T00:00:00+00:00
      └── 1.log
```

Содержимое этих файлов отражает то, что мы видим при открытии задачи в пользовательском интерфейсе веб-сервера.

12.3.4 Отправка журналов в удаленное хранилище

В зависимости от настроек Airflow вам может понадобиться отправить журналы в другое место, например при запуске Airflow во временных контейнерах, в которых журналы пропадают, когда контейнер останавливается или с целью архивации. В Airflow есть функция под названием «удаленное журналирование», которая позволяет отправлять журналы в удаленную систему. На момент написания книги поддерживаются следующие удаленные системы:

- AWS S3 (требуется команда `pip install apache-airflow [amazon]`);
- Azure Blob Storage (требуется команда `pip install apache-airflow [microsoft.azure]`);
- Elasticsearch (требуется команда `pip install apache-airflow [elasticsearch]`);
- Google Cloud Storage (требуется команда `pip install apache-airflow [google]`).

Чтобы настроить Airflow для удаленного журналирования, задайте следующие конфигурации:

- `AIRFLOW__CORE__REMOTE_LOGGING=True`;
- `AIRFLOW__CORE__REMOTE_LOG_CONN_ID=`.

`REMOTE_LOG_CONN_ID` указывает на идентификатор подключения, содержащего учетные данные для вашей удаленной системы. После этого каждая система удаленного журналирования может читать конфигурацию для этой конкретной системы. Например, путь, куда нужно писать журналы, в Google Cloud Storage можно настроить как `AIRFLOW__CORE__REMOTE_BASE_LOG_FOLDER=gs://my-bucket/path/to/logs`. Подробную информацию по каждой системе *см.* в документации Airflow.

12.4 Визуализация и мониторинг метрик Airflow

В какой-то момент вам, возможно, понадобится узнать больше о производительности настройки Airflow. В этом разделе мы сосредото-

чимся на числовых данных о состоянии системы, которые называют *метриками*, например количество секунд задержки между постановкой задачи в очередь и фактическим выполнением задачи. В литературе по мониторингу наблюдаемость и полное понимание системы достигаются за счет сочетания трех элементов: журналов, метрик и трассировки. Журналы (текстовые данные) рассматривались в разделе 12.3, в этом разделе мы рассмотрим метрики, а тема трассировки выходит за рамки данной книги.

У каждой настройки Airflow есть свои особенности. Иногда установка бывает большой, иногда – нет. У некоторых мало ОАГ и много задач; у некоторых – много ОАГ и лишь несколько заданий. Попытку охватить в книге все возможные ситуации вряд ли можно считать практичной, поэтому мы продемонстрируем основные идеи для мониторинга Airflow, которые должны применяться к любой установке. Конечная цель – начать собирать метрики и активно использовать их в своих интересах, например как показано на рис. 12.15.

Рис. 12.15 Пример визуализации количества запущенных задач. Здесь у параллелизма было значение по умолчанию 32, до которого иногда виден резкий скачок количества задач

12.4.1 Сбор метрик из Airflow

Airflow оснащен демоном StatsD (https://github.com/statsd/statsd). Что значит оснащен? «Оснащен» в контексте StatsD и Airflow означает, что определенные события в Airflow приводят к отправке информации о событии, поэтому ее можно собирать, агрегировать, визуализировать или сообщать. Например, всякий раз, когда задача не выполняется, событие с именем «ti_failures» отправляется со значением 1. Это означает, что произошел сбой одной из задач.

Отправка и извлечение

При сравнении систем метрик часто ведется дискуссия по поводу модели *push vs pull*. В случае с *push*-моделью метрики отправляются (push) в систему сбора метрик. В *pull*-модели доступ к метрикам предоставляется системой для мониторинга определенной конечной точки,

а система сбора метрик должна извлекать (pull) метрики из системы из данной конечной точки. Отправка может привести к переполнению системы сбора метрик, когда много систем начинают отправлять большое количество метрик одновременно с системой сбора.

StatsD работает с push-моделью. Поэтому, начиная мониторинг в Airflow, нужно настроить систему сбора, в которую StatsD сможет отправлять свои метрики до того, как мы сможем их просмотреть.

Какую систему сбора метрик выбрать?

statsD – одна из множества доступных систем сбора метрик. Среди прочих можно упомянуть Prometheus и Graphite. Клиент StatsD устанавливается вместе с Airflow. Однако сервер, который будет собирать метрики, нужно настроить самостоятельно. Клиент StatsD передает метрики на сервер в определенном формате, и многие системы сбора метрик могут обмениваться компонентами, считывая форматы друг друга.

Например, сервер Prometheus можно использовать для хранения метрик из Airflow. Однако метрики отправляются в формате StatsD, поэтому Prometheus нужен «перевод», чтобы понять их. Кроме того, Prometheus применяет pull-модель, тогда как StatsD применяет push-модель, поэтому нужно установить некоего посредника, которому StatsD сможет отправлять метрики, а Prometheus будет извлекать их, потому что Airflow не предоставляет формат метрик Prometheus, и, таким образом, Prometheus не может извлекать метрики напрямую из Airflow.

Зачем все смешивать и сочетать? В основном потому, что Prometheus – инструмент, который выбирают многие разработчики и системные администраторы для сбора метрик. Он используется во многих компаниях и превосходит StatsD по многим параметрам, таким как гибкая модель данных, простота эксплуатации и интеграция практически с любой системой. Поэтому мы также предпочитаем Prometheus при работе с метриками, и продемонстрируем, как преобразовать метрики StatsD в метрики Prometheus, после чего можно будет визуализировать собранные метрики с помощью Grafana. Grafana – это удобный инструмент для визуализации метрик мониторинга.

Все шаги, начиная от Airflow и заканчивая Grafana, будут выглядеть так, как показано на рис. 12.16.

Рис. 12.16 Программное обеспечение и шаги, необходимые для сбора и визуализации метрик из Airflow. Prometheus собирает метрики, а Grafana визуализирует их в дашбордах. Экспортер переводит метрики StatsD в формат метрик Prometheus и предоставляет их Prometheus

Давайте настроим эту систему слева (Airflow) направо (Grafana), чтобы создать дашборд для визуализации метрик из Airflow.

12.4.2 Настройка Airflow для отправки метрик

Чтобы Airflow отправлял метрики StatsD, нужно установить Airflow, используя дополнительную зависимость `statsd`:

```
pip install apache-airflow[statsd]
```

Затем настроим место, в которое Airflow должен отправлять метрики. В настоящее время у нас нет системы для сбора метрик, но мы настроим ее далее в разделе 12.4.3.

- `AIRFLOW__METRICS__STATSD_ON=True`;
- `AIRFLOW__METRICS__STATSD_HOST=localhost` (значение по умолчанию);
- `AIRFLOW__METRICS__STATSD_PORT=9125`;
- `AIRFLOW__METRICS__STATSD_PREFIX=airflow` (значение по умолчанию).

Что касается Airlfow, то здесь все готово. В этой конфигурации Airflow будет отправлять события на порт 9125 (по протоколу UDP).

12.4.3 Настройка Prometheus для сбора метрик

Prometheus – это программа для мониторинга систем. Она обладает широким набором функций, но, по сути, это база данных временных рядов, к которой можно выполнять запросы с помощью языка PromQL. Здесь нельзя вставлять данные вручную, например используя запрос `INSERT INTO ...`, как в случае с реляционной базой данных, но можно извлекать метрики в базу данных. Каждые *X* секунд она извлекает самые последние метрики из сконфигурированных вами целей. При чрезмерной загрузке Prometheus автоматически замедляется при мониторинге целей. Однако для этого требуется обработать большое количество метрик, поэтому на данный момент этот вариант не применим.

Для начала нужно установить экспортер StatsD, который переводит метрики StatsD в метрики Prometheus. Проще всего это сделать с помощью Docker.

Листинг 12.13 Запуск экспортера StatsD с Docker

```
docker run -d -p 9102:9102 -p 9125:9125/udp prom/statsd-exporter
```

Метрики Prometheu будут отображаться по адресу http://localhost: 9102

Убедитесь, что этот номер порта совпадает с портом, заданным AIRFLOW__SCHEDULER__STATSD_PORT

Если не использовать Docker, то экспортер StatsD можно скачать на странице https://github.com/prometheus/statsd_exporter/releases.

Для начала можно запустить экспортер без конфигурации. Перейдите по адресу http://localhost:9102/metrics, где вы должны увидеть первые метрики Airflow.

Листинг 12.14 Примеры метрик Prometheus, представленных с помощью экспортера StatsD

У каждой метрики есть такой тип, как датчик

К каждой метрике по умолчанию сообщение HELP

```
# HELP airflow_collect_dags Metric autogenerated by statsd_exporter.
# TYPE airflow_collect_dags gauge
airflow_collect_dags 1.019871
# HELP airflow_dag_processing_processes Metric autogenerated by statsd_exporter.
# TYPE airflow_dag_processing_processes counter
airflow_dag_processing_processes 35001
# HELP airflow_dag_processing_total_parse_time Metric autogenerated by
    statsd_exporter.
# TYPE airflow_dag_processing_total_parse_time gauge
airflow_dag_processing_total_parse_time 1.019871
➥ # HELP airflow_dagbag_import_errors Metric autogenerated by statsd_exporter.
# TYPE airflow_dagbag_import_errors gauge
airflow_dagbag_import_errors 0
# HELP airflow_dagbag_size Metric autogenerated by statsd_exporter.
# TYPE airflow_dagbag_size gauge
airflow_dagbag_size 4
```

Метрика airflow_collect_dags в настоящее время имеет значение 1.019871. Prometheus регистрирует временную метку мониторинга вместе с этим значением

Теперь, когда мы сделали метрики доступными по адресу http://localhost: 9102, можно установить и настроить Prometheus для мониторинга этой конечной точки. Самый простой способ сделать это – еще раз использовать Docker для запуска контейнера Prometheus. Сначала нужно настроить экспортер StatsD в качестве цели в Prometheus, чтобы он знал, откуда брать метрики.

Листинг 12.15 Минимальная конфигурация Prometheus

```
scrape_configs:
  - job_name: 'airflow'
    static_configs:
    - targets: ['localhost:9102']
```

Определяет задание по мониторингу метрик Prometheus

Целевой URL-адрес задания мониторинга

Сохраните содержимое листинга 12.15 в файл, например, /tmp/prometheus.yml. Потом запустите Prometheus и смонтируйте файл.

Листинг 12.16 Запуск Prometheus с Docker для сбора метрик

```
➥ docker run -d -p 9090:9090 -v /tmp/prometheus.yml:/etc/prometheus/
    prometheus.yml prom/prometheus
```

Prometheus теперь запущен и работает по адресу http://localhost:9090. Чтобы это проверить, перейдите по адресу http://localhost:9090/target и убедитесь, что цель Airflow доступна (рис. 12.17).

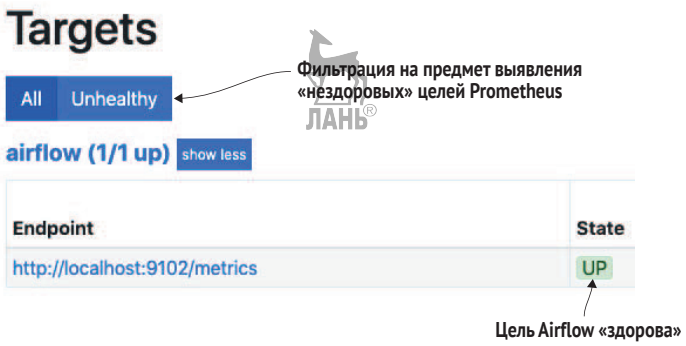


Рис. 12.17 Если все настроено правильно, на странице целей в Prometheus должно быть отображено состояние цели Airlfow – UP. В противном случае можно считать, что с целью что-то не так

Активно работающая цель означает, что Prometheus мониторит метрики, и мы можем приступить к визуализации показателей в Grafana.

Модели данных метрик

Модель данных Prometheus идентифицирует уникальные метрики по имени (например, `task_duration`) и набор меток «ключ-значение» (например, `dag_id=mydag` и `task_id=first_task`). Это обеспечивает значительную гибкость, поскольку вы можете выбирать метрики с любой желаемой комбинацией меток, например `task_duration{task_id="first_task"}`, чтобы выбрать только `task_duration` задач с именем «first_task». Альтернативная модель данных, встречающаяся во многих других системах метрик, таких как StatsD, основана на иерархии, где метки определены в именах метрик, разделенных точками:

- task_duration.my_dag.first_task -> 123;
- task_duration.my_other_dag.first_task -> 4.

Это может стать проблемой, если у вас появится желание выбрать метрику `task_duration` из всех задач с именем `first_task`. Это одна из причин, по которой Prometheus приобрела популярность.

Экспортер StatsD применяет универсальные правила к предоставленным метрикам, чтобы преобразовать их из иерархической модели, используемой StatsD, в модель меток, используемую Prometheus. Иногда правила преобразования по умолчанию работают прекрасно, а иногда – нет, и метрика StatsD дает уникальное имя метрики в Prometheus. Например, в метрике `dag.<dag_id>.<task_id>.duration`, `dag_id` и `task_id` не преобразуются автоматически в метки в Prometheus.

Хотя технически такой вариант подходит для Prometheus, он не оптимален. Таким образом, экспортер StatsD можно настроить для преобразования определенных метрик, разделенных точками, в метрики Prometheus. *См.* приложение C, где содержится такой конфигурационный файл. Для получения дополнительной информации обратитесь к документации по экспортеру StatsD.

12.4.4 Создание дашбордов с Grafana

После сбора метрик с помощью Prometheus последняя часть головоломки – визуализировать эти показатели на дашборде. Это должно дать нам быстрое понимание того, как функционирует система. Grafana – основной инструмент для визуализации метрик. Самый простой способ запустить Grafana – снова использовать Docker.

Листинг 12.17 Запуск Grafana с Docker для визуализации метрик

```
docker run -d -p 3000:3000 grafana/grafana
```

Перейдя по адресу http://localhost: 3000, вы увидите первое представление Grafana (рис. 12.18).

Рис. 12.18 Экран приветствия Grafana

Щелкните **Add your first data source** (Добавить свой первый источник данных), чтобы добавить Prometheus в качестве источника данных. Вы увидите список доступных источников данных. Щелкните **Prometheus**, чтобы настроить его (рис. 12.19).

Рис. 12.19 На странице «Добавить источник данных» выберите Prometheus, чтобы настроить его в качестве источника для чтения метрик

На следующем экране укажите URL-адрес Prometheus: http://localhost:9090 (рис. 12.20).

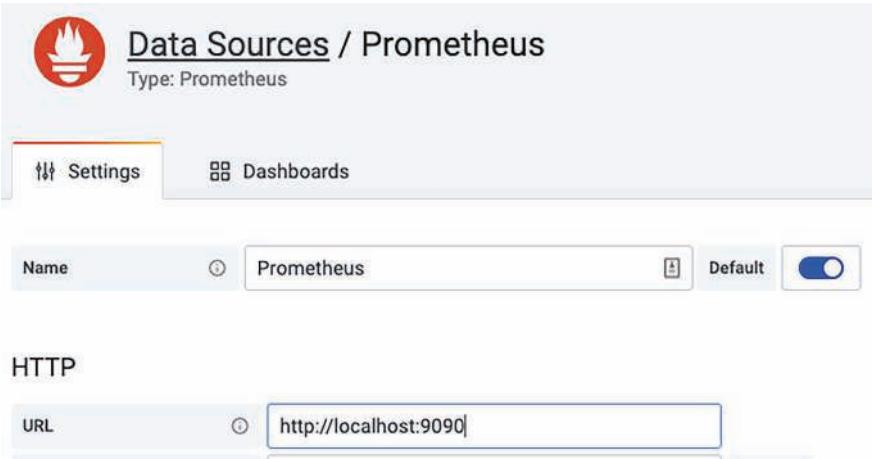


Рис. 12.20 Укажите в Grafana URL-адрес Prometheus

Когда Prometheus будет настроен в качестве источника данных в Grafana, наступает время визуализировать первую метрику. Создайте новый дашборд и панель на этом дашборде. Вставьте следующую метрику в поле запроса: `airflow_dag_processing_total_parse_time` (количество секунд, затраченных на обработку всех ОАГ). Теперь появится визуализация для этой метрики (рис. 12.21).

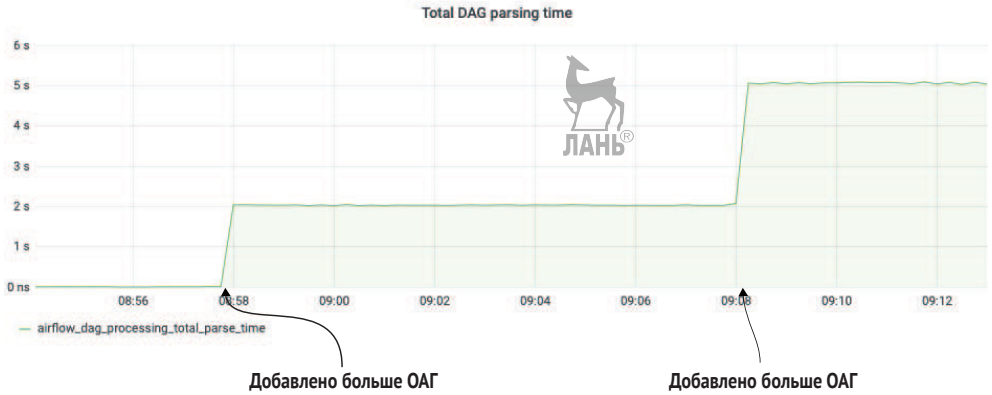


Рис. 12.21 График количества секунд для обработки всех файлов ОАГ. Мы видим две точки изменения, в которых были добавлены дополнительные файлы ОАГ. Резкий всплеск на этом графике может указывать на проблему с планировщиком Airflow или файлом ОАГ

Теперь, когда у нас есть Prometheus и Grafana, Airflow передает показатели экспортеру StatsD, которые в конечном итоге отображаются в Grafana. В этой настройке следует отметить два момента. Во-первых, метрики в Grafana близки к реальному времени, но здесь не идет речь о миллисекундах. Prometheus мониторит метрики с учетом интервалов (по умолчанию это одна минута, и это значение можно уменьшить), что в худшем случае вызовет одноминутную задержку. Кроме того, Grafana периодически выполняет запрос к Prometheus (обновление запроса по умолчанию отключено), поэтому в Grafana у нас тоже есть небольшая задержка. В целом задержка между событием в Airflow и графиком в Grafana составляет максимум минут, чего обычно более чем достаточно.

Во-вторых, эта установка использует Prometheus, которая отлично подходит для мониторинга метрик и оповещения. Однако это не система отчетов, и она не хранит отдельные события. Если вы планируете сообщать об отдельных событиях в Airflow, то можете рассмотреть InfluxDB как базу данных временных рядов, поскольку она больше ориентирована на журналирование событий.

12.4.5 Что следует мониторить?

Теперь, когда у нас есть настройка мониторинга, что нужно мониторить, чтобы понять, как функционирует Airflow? Начнем с того, что, когда вы что-то мониторите, необходимо отслеживать четыре основных сигнала.

Задержка

Сколько времени уходит на обработку служебных запросов? Подумайте, сколько времени требуется веб-серверу, чтобы ответить, или сколько времени нужно планировщику, чтобы переместить задачу из состояния очереди в состояние запуска. Эти показатели выражаются

в виде продолжительности (например, «среднее количество миллисекунд для возврата запроса веб-сервера» или «среднее время в секундах для перевода задач из состояния очереди в состояние запуска»).

Трафик

Насколько востребована система? Подумайте, сколько задач ваша система Airflow должна обработать, или сколько открытых слотов пула доступно Airflow. Эти показатели обычно выражаются как среднее значение за продолжительность (например, «количество выполняемых задач в минуту» или «открытых слотов пула в секунду»).

Ошибки

Какие ошибки возникли? В контексте Airflow это может быть «число задач-зомби» (задач, в которых основной процесс исчез), «количество ответов на веб-сервере, чей код состояния не относится к 200» или «количество истекших задач».

Насыщенность

Какая часть мощности вашей системы используется? Измерение метрик машины, на которой работает Airflow, может быть хорошим индикатором, например «текущая загрузка процессора» или «количество выполняемых в данный момент задач». Чтобы определить, насколько заполнена система, вы должны знать ее верхний предел, который иногда бывает непросто определить.

Prometheus предлагает широкий спектр экспортеров, предоставляющих всевозможные метрики системы. Начните с установки нескольких экспортеров Prometheus, чтобы узнать больше обо всех задействованных системах:

- *экспортер узлов* – для мониторинга компьютеров, на которых работает Airflow (ЦП, память, дисковый ввод-вывод, сетевой трафик);
- *экспортер сервера PostgreSQL/MySQL* – для мониторинга базы данных;
- *один из нескольких (неофициальных) экспортеров Celery* – для мониторинга Celery при использовании `CeleryExecutor`;
- *экспортер Blackbox* – для опроса данной конечной точки, чтобы проверить, возвращается ли предопределенный код HTTP;
- *при использовании Kubernetes, одного из многих экспортеров Kubernetes* – для мониторинга ресурсов Kubernetes. *См.* документацию по мониторингу Kubernetes.

Обзор всех доступных метрик приведен в документации Airflow; обратитесь к ней для своей версии Airflow. Вот несколько хороших метрик для определения статуса Airflow:

- чтобы узнать о правильном функционировании ваших ОАГ:
  - `dag_processing.import_errors` – показывает количество ошибок, обнаруженных при обработке ОАГ. Все, что больше нуля, не годится;

  - `dag_processing.total_parse_time` – внезапное сильное увеличение после добавления или изменения ОАГ – это нехорошо;
  - `ti_failures` – количество неудачных экземпляров задачи;
- чтобы разобраться с производительностью Airflow:
  - `dag_processing.last_duration.[filename]` – время, затраченное на обработку файла ОАГ. Высокие значения указывают на то, что что-то не так;
  - `dag_processing.last_run.seconds_ago.[filename]` – количество секунд с момента последней проверки планировщиком файла, содержащего ОАГ. Чем выше это значение, тем хуже; это означает, что планировщик слишком занят. Значения должны быть порядка нескольких секунд максимум;
  - `dagrun.[dag_id].first_task_scheduling_delay` – задержка между запланированной и фактической датами выполнения запуска ОАГ;
  - `execor.open_slots` – количество свободных слотов исполнителя;
  - `executeor.queued_tasks` – количество задач в очереди;
  - `executeor.running_tasks` – количество задач с состоянием запуска.

12.5 Как получить уведомление о невыполненной задаче

При запуске любых критически важных конвейеров нам нужно получать уведомления об инциденте в тот момент, когда что-то пойдет не так. Представьте себе невыполненную задачу или задачу, которая не завершилась в ожидаемые сроки и задерживает другие процессы. Рассмотрим различные варианты, которые Airflow предоставляет как для определения условий, требующих оповещений, так и для отправки фактических оповещений.

12.5.1 Оповещения в ОАГ и операторах

В Airflow есть несколько уровней для настройки оповещений. Во-первых, в рамках определения ОАГ и операторов можно настроить т. н. *функции обратного вызова* (т. е. функции для вызова определенных событий).

Листинг 12.18 Определение функции обратного вызова для выполнения при сбое ОАГ

```
def send_error():
    print("ERROR!")

dag = DAG(
    dag_id="chapter12",
    on_failure_callback=send_error,
    ...
)
```

send_error выполняется при сбое запуска ОАГ

`on_failure_callback` – это аргумент, который выполняется всякий раз при сбое запуска ОАГ. Представьте себе отправку сообщения Slack в канал ошибок, уведомления в систему сообщений об инцидентах, такую как PagerDuty, или в обычную старую электронную почту. Однако выполняемую функцию вам нужно будет реализовать самостоятельно.

На уровне задачи есть дополнительные параметры для настройки. У вас вряд ли есть желание настраивать каждую задачу отдельно, чтобы можно было распространять конфигурацию с помощью параметра `default_args` на все задачи.

Листинг 12.19 Определение функции обратного вызова для выполнения при сбое задачи

```
def send_error():
    print("ERROR!")


dag = DAG(
    dag_id="chapter12_task_failure_callback",
    default_args={"on_failure_callback": send_error},
    on_failure_callback=send_error,
    ...
)

failing_task = BashOperator(
    task_id="failing_task",
    bash_command="exit 1",
    dag=dag,
)
```

default_args распространяет аргументы на задачи

Обратите внимание, что здесь будут отправлены два уведомления: одно для сбоя задачи и одно для сбоя ОАГ

Эта задача не вернет код выхода 0 и, следовательно, завершится ошибкой

Родительский класс всех операторов (`BaseOperator`) содержит аргумент `on_failure_callback`; следовательно, его содержат все операторы. При настройке этого аргумента в `default_args` вы задаете сконфигурированные аргументы для всех задач в ОАГ, поэтому все задачи будут вызывать `send_error` всякий раз, когда возникает ошибка в листинге 12.19. Также можно настроить аргумент `on_success_callback` (в случае успеха) и аргумент `on_retry_callback` (в случае повторного выполнения задачи).

Хотя можно самостоятельно отправить электронное письмо внутри функции, вызываемой `on_failure_callback`, Airflow предоставляет удобный аргумент `email_on_failure`, который отправляет электронное письмо без необходимости настраивать сообщение. Однако нужно настроить протокол SMTP в конфигурации Airflow; в противном случае электронные письма нельзя будет отправить. Такая конфигурация предназначена для Gmail.

Листинг 12.20 Пример конфигурации SMTP для отправки автоматических электронных писем

```
AIRFLOW__SMTP__SMTP_HOST=smtp.gmail.com
AIRFLOW__SMTP__SMTP_MAIL_FROM=myname@gmail.com
```

```
AIRFLOW__SMTP__SMTP_PASSWORD=abcdefghijklmnop
AIRFLOW__SMTP__SMTP_PORT=587
AIRFLOW__SMTP__SMTP_SSL=False
AIRFLOW__SMTP__SMTP_STARTTLS=True
AIRFLOW__SMTP__SMTP_USER=myname@gmail.com
```

Фактически Airflow настроен на отправку писем по умолчанию, то есть в `BaseOperator` есть аргумент `email_on_failure`, который по умолчанию имеет значение `True`. Тем не менее без правильной конфигурации SMTP он не будет отправлять электронную почту. Кроме того, в аргументе оператора `email` также нужно задать адрес электронной почты получателя.

Листинг 12.21 Настройка адреса электронной почты для отправки оповещений

```
dag = DAG(
   dag_id="task_failure_email",
   default_args={"email": "bob@work.com"},
   ...
)
```

При правильной конфигурации SMTP и настроенном адресе электронной почты получателя Airflow теперь отправит вам электронное письмо с уведомлением о невыполненной задаче (рис. 12.22).

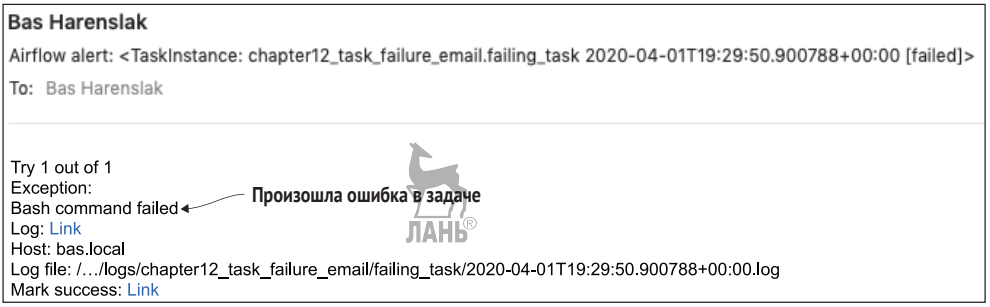


Рис. 12.22 Пример оповещения по электронной почте

Журналы задач также сообщают нам, что письмо было отправлено:

```
INFO - Sent an alert email to ['bob@work.com']
```

12.5.2 Определение соглашений об уровне предоставления услуги

В Airflow также существует концепция SLA (соглашение об уровне предоставления услуги). Это определенный стандарт, которому необходимо соответствовать в отношении услуги или продукта. Например, ваш поставщик телевизионных услуг может гарантировать, что

99,999 % времени телевизор будет работать безотказно. Это означает, что допустимый простой в год составляет 5,26 минуты. В Airflow можно настроить соглашение об уровне предоставления услуги на уровне задачи для настройки последней приемлемой даты и времени завершения задачи. Если это соглашение не соблюдается, отправляется электронное письмо или вызывается самоопределяемая функция обратного вызова. Чтобы настроить дату и время крайнего срока для выполнения задачи с помощью SLA, *см.* следующий код.

Листинг 12.22 Настройка SLA

```
dag = DAG(
    dag_id="chapter12_task_sla",
    default_args={"email": "bob@work.com"},
    schedule_interval=datetime.timedelta(minutes=30),
    start_date=datetime.datetime(2020, 1, 1, 12),
       end_date=datetime.datetime(2020, 1, 1, 15),
)

sleeptask = BashOperator(
    task_id="sleeptask",
    bash_command="sleep 60",
    sla=datetime.timedelta(minutes=2),
    dag=dag,
)
```

ОАГ срабатывает каждые 30 минут, например в 12:30

Эта задача бездействует 60 секунд

SLA определяет максимальную разницу между запланированным запуском ОАГ и завершением задачи (например, 12:32)

Эти соглашения работают несколько нелогично. Хотя вы можете ожидать, что они будут функционировать как максимальное время выполнения для заданных задач, они функционируют как максимальная разница во времени между запланированным началом запуска ОАГ и завершением задачи.

Итак, если ваш ОАГ начинается в 12:30 и вы хотите, чтобы ваша задача завершилась не позднее 14:30, нужно задать для timedelta значение, равное двум часам, даже если вы ожидаете, что задача будет выполняться всего за пять минут. Примером аргумента в пользу такого, на первый взгляд, непонятного поведения может быть ситуация, когда вы хотите, чтобы отчет был отправлен не позднее определенного времени, скажем 14:30. Если обработка данных для отчета занимает больше времени, чем ожидалось, задача *отправить электронное письмо с отчетом* будет завершена после крайнего срока, 14:30, и будет инициировано соглашение об уровне предоставления услуги. Само условие SLA срабатывает примерно во время крайнего срока вместо ожидания завершения задачи. Если задача не будет завершена до установленного срока, будет отправлено электронное письмо.

Листинг 12.23 Пример сообщения электронной почты о пропущенном SLA

```
Here's a list of tasks that missed their SLAs:
sleeptask on 2020-01-01T12:30:00+00:00
```

```
Blocking tasks:


     =,             .=
    =.|    ,---.    |.=
    =.| "-(:::::)-" |.=
     \\__/`-.|.-'\__//
      `-| .::| .::|-'          Pillendreher
        _|`-._|_.-'|_          (Scarabaeus sacer)
     /.-| |    .::|-.\
    // ,| .::|::::|. \\
   || //\::::|::' /\\ ||
   /'\|| `.__|__.' ||/'\
  ^    \\         //   ^
       /'\       /'\
  ^                 ^
```

Да, в письме был этот жук в кодировке ASCII! Хотя задача из листинга 12.22 служит примером, желательно настроить SLA для обнаружения отклонений в вашем задании. Например, если входные данные вашего задания внезапно увеличиваются в пять раз, а это приводит к тому, что задание занимает значительно больше времени, можно рассмотреть возможность повторного вычисления определенных параметров задания. Дрейф в размере данных и результирующую продолжительность задания можно обнаружить с помощью SLA.

Сообщение по электронной почте уведомляет вас только о нарушении SLA, поэтому можно рассмотреть что-то другое, кроме электронной почты или вашего собственного формата. Этого можно добиться с помощью аргумента `sla_miss_callback`. Как ни странно, это аргумент класса DAG, а не класса `BaseOperator`.

Если вы ищете максимальное время выполнения задачи, настройте аргумент `execution_timeout` для своего оператора. Если продолжительность задачи превышает настроенное значение `execution_timeout`, она завершается ошибкой.

12.6 Масштабируемость и производительность

В разделах 12.1 и 12.2 мы рассмотрели типы исполнителей, которые предлагает Airflow:

- SequentialExecutor (по умолчанию);
- LocalExecutor;
- CeleryExecutor;
- KubernetesExecutor.

Давайте подробнее рассмотрим, как настроить Airflow и эти типы исполнителей для обеспечения адекватной масштабируемости и производительности. Под производительностью мы понимаем способность быстро реагировать на события без задержек и с минимальным

ожиданием. Под масштабируемостью понимается способность обрабатывать большую (увеличивающуюся) нагрузку без воздействия на службу.

Мы бы хотели подчеркнуть важность мониторинга, как описано в разделе 12.4. Без измерения и знания состояния вашей системы оптимизация остается лишь догадкой. Измеряя то, что вы делаете, вы знаете, положительно ли влияет изменение на вашу систему.

12.6.1 Контроль максимального количества запущенных задач

В табл. 12.3 перечислены конфигурации Airflow, с помощью которых можно контролировать количество задач, которые вы можете запускать параллельно. Обратите внимание, что элементы конфигурации имеют несколько странное название, поэтому внимательно прочтите их описание.

Таблица 12.3 Обзор конфигураций Airflow, связанных с количеством выполняемых задач

Элемент конфигурации	Значение по умолчанию	Описание
`AIRFLOW__CORE__DAG_CONCURRENCY`	16	Максимальное количество задач в очереди или в состоянии запуска на каждый ОАГ
`AIRFLOW__CORE__MAX_ACTIVE_RUNS_PER_DAG`	16	Максимальное количество параллельных запусков ОАГ на каждый ОАГ
`AIRFLOW__CORE__PARALLELISM`	32	Максимальное количество экземпляров задачи для параллельного запуска, глобально
`AIRFLOW__CELERY__WORKER_CONCURRENCY`	16	Максимальное количество задач на один воркер Celery (только для Celery)

Если вы запускаете ОАГ с большим количеством задач, то значения по умолчанию ограничивают это число до 16 параллельных задач из-за того, что для параметра `dag_concurrency` задано значение 16, даже если для параметра `parallelism` установлено значение 32. Второй ОАГ с большим количеством задач также будет ограничен 16 параллельными задачами, но вместе они достигнут глобального предела 32, заданного параметром `parallelism`.

Есть еще один фактор, ограничивающий глобальное количество параллельных задач: по умолчанию все задачи выполняются в пуле «default_pool» со 128 слотами по умолчанию. Однако перед достижением предела этого пула необходимо увеличить значения `dag_concurrency` и `parallelism`.

В частности, для `CeleryExecutor` параметр `AIRFLOW__CELERY__WORKER_ CONCURRENCY` контролирует количество процессов на одного воркера, которые будет обрабатывать Celery. По нашему опыту, Airflow может потребовать значительных ресурсов; поэтому учитывайте как минимум 200 МБ ОЗУ на процесс в качестве базового показателя,

для того чтобы просто иметь воркер с настроенным числом `concurrency` в рабочем состоянии. Кроме того, нужно рассмотреть худший сценарий, при котором наиболее ресурсоемкие задачи выполняются параллельно, чтобы оценить, сколько параллельных задач может обработать воркер Celery. Для определенных ОАГ значение по умолчанию `max_active_runs_per_dag` можно переопределить аргументом `concurrency` в классе DAG.

На уровне отдельной задачи можно задать аргумент `pool` для запуска конкретной задачи в пуле, который ограничивает количество задач, которые он может выполнить. Пулы можно применять для определенных групп задач. Например, хотя для вашей системы может быть нормально запускать 20 задач, выполняющих запросы к базе данных и ожидающих возврата результата, при запуске 5 задач, интенсивно использующих ЦП, могут возникнуть проблемы. Чтобы ограничить такие ресурсоемкие задачи, можно назначить им выделенный пул `high_resource` с низким максимальным количеством задач.

Кроме того, на уровне задачи можно задать аргумент `task_concurrency`, который обеспечивает дополнительное ограничение конкретной задачи на несколько запусков. Это может быть полезно в случае ресурсоемкой задачи, которая может потребовать все ресурсы машины при параллельном запуске со множеством экземпляров (рис. 12.23).

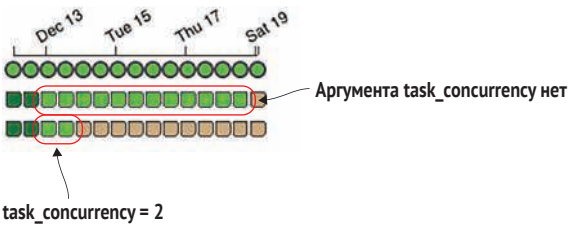


Рис. 12.23 `task_concurrency` может ограничить количество параллельных выполнений задачи

12.6.2 Конфигурации производительности системы

При выполнении значительного количества задач вы можете заметить, что нагрузка на базу метаданных возрастает. Airflow в значительной степени полагается на базу данных для хранения всего состояния. Каждая новая версия Airflow обычно включает в себя несколько улучшений, связанных с производительностью, поэтому это помогает регулярно выполнять обновления. Мы также можем настроить количество выполняемых запросов к базе данных.

Повышение значения `AIRFLOW__SCHEDULER__SCHEDULER_HEARTBEAT_SEC` (по умолчанию 5) может снизить количество проверок, которые Airflow выполняет в задании планировщика, что приводит к меньшему числу запросов к базе данных. 60 секунд – разумное значение. Пользовательский интерфейс Airflow отобразит предупреждение

через 30 секунд после получения последнего контрольного сигнала планировщика, но это число можно настроить с помощью `AIRFLOW__SCHEDULER__SCHEDULER_HEALTH_CHECK_THRESHOLD`.

Значение `AIRFLOW__SCHEDULER__PARSING_PROCESSES` (по умолчанию 2; изменяется на 1, если используется SQLite) контролирует, сколько процессов выполняет планировщик задач одновременно для обработки состояния ОАГ; каждый процесс заботится о проверке необходимости создания новых запусков ОАГ, планирования или постановки новых экземпляров задач в очередь и т. д. Чем выше это число, тем больше ОАГ будет проверяться одновременно и меньше задержка между задачами. Повышение данного значения происходит за счет увеличения использования ЦП, поэтому увеличивайте и измеряйте изменения осторожно.

Наконец, с точки зрения пользователя, может быть интересно настроить `AIRFLOW_SCHEDULER__DAG_DIR_LIST_INTERVAL` (по умолчанию 300 секунд). Этот параметр определяет, как часто планировщик проверяет каталог ОАГ на наличие новых, ранее неизвестных файлов. Если вы часто добавляете новые файлы ОАГ, то вам придется ждать, пока они появятся в пользовательском интерфейсе Airflow. Уменьшение этого значения заставит Airflow сканировать каталог ОАГ на предмет наличия новых файлов чаще, но за счет бóльшей загрузки ЦП, поэтому также осторожно уменьшайте это значение.

12.6.3 Запуск нескольких планировщиков

Долгожданная функция Airflow 2 – возможность горизонтального масштабирования планировщика (в Airflow 1 этой функции нет). Поскольку планировщик – это сердце и мозги Airflow, способность запускать несколько экземпляров планировщика, для масштабируемости и избыточности, были давней мечтой сообщества Airflow.

Распределенные системы сложны, и большинство систем требуют добавления алгоритма консенсуса, чтобы определить, какой процесс является лидером. В Airflow цель заключалась в том, чтобы максимально упростить системные операции, а лидерство было реализовано за счет блокировки на уровне строк (`SELECT ... FOR UPDATE`) на уровне базы данных. В результате несколько планировщиков могут работать независимо друг от друга, не требуя дополнительных инструментов для достижения консенсуса. Единственный момент – база данных должна поддерживать определенные концепции блокировки. На момент написания книги тестируются и поддерживаются следующие базы и версии:

- PostgreSQL версии 9.6 и выше;
- MySQL версии 8 и выше.

Для масштабирования планировщика просто запустите еще один процесс планировщика:

```
airflow scheduler
```

Каждый экземпляр планировщика определит, какие задачи (представленные строками в базе данных) доступны для обработки по принципу «первым пришел – первым обслужен», и никакой дополнительной настройки не требуется. После запуска нескольких экземпляров, если одна из машин, на которой работает один из планировщиков, выйдет из строя, он больше не будет отключать Airflow, поскольку другие экземпляры планировщика будут продолжать работать.

Резюме

- `SequentialExecutor` и `LocalExecutor` ограничены одной машиной, но их легко настроить.
- Для настройки `CeleryExecutor` и `KubernetesExecutor` требуется больше работы, но они позволяют масштабировать задачи на нескольких машинах.
- Prometheus и Grafana можно использовать для хранения и визуализации метрик из Airflow.
- Функции обратного вызова при сбоях и соглашения об уровне предоставления услуги могут отправлять электронные письма или настраиваемые уведомления в случае определенных событий.
- Развернуть Airflow на нескольких машинах не так просто, поскольку задачи Airflow и планировщик (планировщики) требуют доступа к каталогу ОАГ.

13 Безопасность в Airflow

Эта глава рассказывает:

- о настройке интерфейса RBAC для контроля доступа;
- о предоставлении доступа центральному набору пользователей с помощью подключения к службе LDAP;
- о настройке ключа Fernet для шифрования секретов в базе данных;
- об обеспечении безопасности трафика между вашим браузером и веб-сервером;
- об извлечении секретов из центральной системы управления секретами.

Учитывая природу Airflow, который напоминает паука в паутине, управляющего серией задач, он должен подключаться ко множеству систем и, следовательно, является желанной целью, к которой некоторые хотят получить доступ. Чтобы избежать такой нежелательной ситуации, в этой главе мы обсудим безопасность Airflow. Мы расскажем о различных вариантах использования, связанных с вопросами безопасности, и подробно рассмотрим их на практических примерах. Безопасность часто рассматривается как нечто, связанное с черной магией, где необходимо понимание множества технологий, сокращений и сложных деталей. Хотя это и не так, мы написали данную главу для читателей, мало знающих о безопасности, и выделили различные ключевые моменты, чтобы вы могли избежать нежелательных действий при установке Airflow, которые должны послужить отправной точкой.

Интерфейсы Airflow

В Airflow 1.x есть два интерфейса:

- оригинальный интерфейс, разработанный поверх Flask-Admin;
- интерфейс RBAC, разработанный поверх Flask-AppBuilder (FAB).

Первоначально Airflow поставлялся с исходным интерфейсом и впервые представил интерфейс управления доступом на основе ролей (RBAC) в Airflow версии 1.10.0. Интерфейс RBAC предоставляет механизм, который ограничивает доступ, определяя роли с соответствующими полномочиями и назначая эти роли пользователям. Исходный интерфейс по умолчанию открыт для всех. Интерфейс RBAC имеет больше функций безопасности.

Во время написания этой книги исходный интерфейс устарел и был удален в Airflow 2.0. RBAC теперь является единственным интерфейсом, поэтому в этой главе мы рассматриваем только его. Чтобы активировать его интерфейс RBAC с Airflow 1.x, задайте для `AIRFLOW__WEBSERVER__RBAC` значение `True`.

13.1 Обеспечение безопасности веб-интерфейса Airflow

Запустите веб-сервер Airflow, выполнив команду `airflow webserver`, и перейдите по адресу http://localhost:8080, где вы увидите экран входа (рис. 13.1).

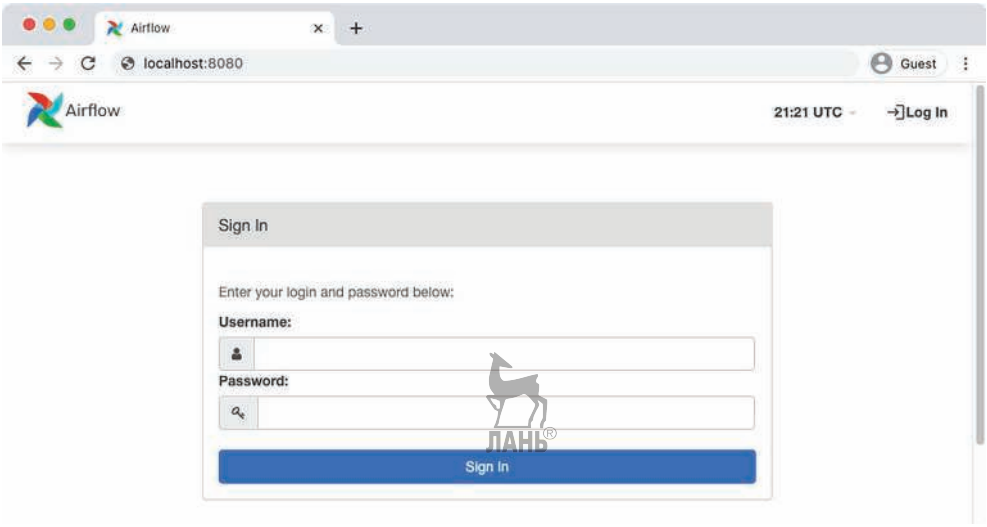


Рис. 13.1 Главный экран интерфейса RBAC. Аутентификация по паролю активирована по умолчанию. Пользователя по умолчанию не существует

Это первое представление интерфейса RBAC. На этом этапе веб-сервер запрашивает имя пользователя и пароль, но пользователей пока нет.

13.1.1 Добавление пользователей в интерфейс RBAC

Мы создадим учетную запись для пользователя по имени Боб Смит.

Листинг 13.1 Регистрация пользователя для интерфейса RBAC

```
airflow users create \
--role Admin \
--username bobsmith \
--password topsecret \
--email bobsmith@company.com \
--firstname Bob \
--lastname Smith
```

Роль Admin предоставляет все полномочия этому пользователю

Опускаем параметр –password для запроса пароля

С помощью этого кода мы создаем пользователя с ролью «Администратор». Модель RBAC состоит из пользователей, которым назначена (единственная) роль с полномочиями (определенными операциями), назначенными этим ролям, которые применяются к определенным компонентам интерфейса веб-сервера (рис. 13.2).

Рис. 13.2 Модель полномочий RBAC

В листинге 13.1 пользователю «bobsmith» была назначена роль «Адмиистратор». Определенным операциям (например, `edit`) в определенных компонентах (таких как меню и конкретные страницы, например «Подключения») затем можно назначить роли. Например, наличие полномочий типа «можно редактировать в ConnectionModelView» позволяет редактировать подключения.

По умолчанию есть пять ролей. Роль администратора предоставляет все полномочия, в том числе доступ к представлению безопасности. Однако нужно не торопясь подумать, какую роль вы предоставите пользователю в рабочей системе.

На данном этапе можно выполнить вход с именем пользователя «bobsmith» и паролем «topsecret». Главный экран будет выглядеть так же, как и исходный интерфейс, но на верхней панели есть несколько новых элементов, показанных на рис. 13.3.

Представление безопасности – самая интересная особенность интерфейса RBAC. При открытии меню будет отображено несколько параметров (рис. 13.4).

Щелкните **List Roles** (Список ролей), чтобы изучить все роли по умолчанию (рис. 13.5).

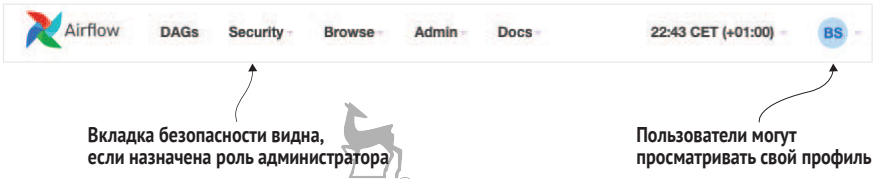


Рис. 13.3 Верхняя панель, отображающая пункты меню в зависимости от роли и соответствующих полномочий, которые были предоставлены вашему пользователю

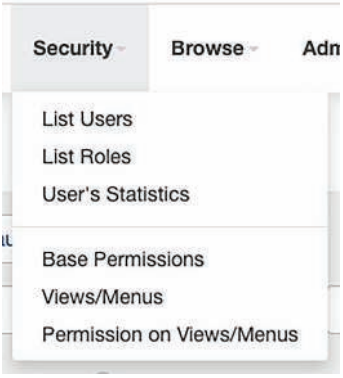


Рис. 13.4 Параметры на вкладке Security (Безопасность)

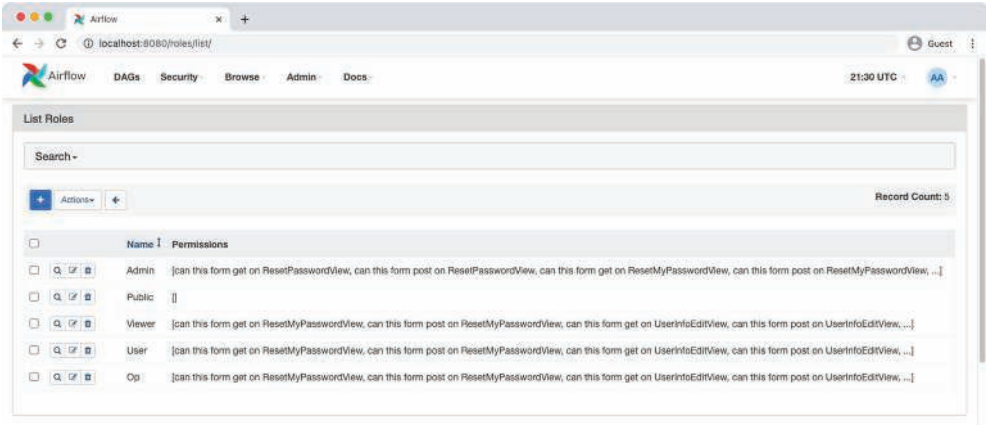


Рис. 13.5 Роли по умолчанию и соответствующие полномочия в Airflow. Некоторые полномочия опущены, чтобы было удобнее читать

В этом представлении мы видим пять ролей, доступных для использования по умолчанию. Полномочия по умолчанию для этих ролей приведены в табл. 13.1.

Только что созданному пользователю «bobsmith» была назначена роль администратора, что предоставляет ему все полномочия (неко-

торые из них не показаны на рис. 13.5, чтобы было удобнее читать). Возможно, вы заметили, что у роли Public нет полномочий. Как следует из названия роли, все полномочия, прикрепленные к ней, являются общедоступными (т. е. вам не нужно выполнять вход). Допустим, вы хотите разрешить лицам без учетной записи Airflow просматривать меню **Docs** (Документы) (рис. 13.6).

Таблица 13.1 Полномочия ролей по умолчанию для интерфейса RBAC

Роль	Предполагаемые пользователи / использование	Полномочия по умолчанию
Admin	Требуется только при управлении полномочиями безопасности	Все полномочия
Public	Неаутентифицированные пользователи	Отсутствие полномочий
Viewer	Представление Airflow только для чтения	Доступ к ОАГ для чтения
User	Полезна, если вам необходимо строгое разделение в своей команде между разработчиками, которые могут или не могут редактировать секреты (подключения, переменные и т. д.). Эта роль предоставляет полномочия только на создание ОАГ, а не секретов	То же, что и Viewer, но с полномочиями на редактирование (очистка, запуск, пауза и т. д.) в ОАГ
Op	Все полномочия, необходимые для разработки ОАГ	То же, что и User, но с дополнительными полномочиями на просмотр и редактирование подключений, пулов, переменных, XCom'ов и конфигурации

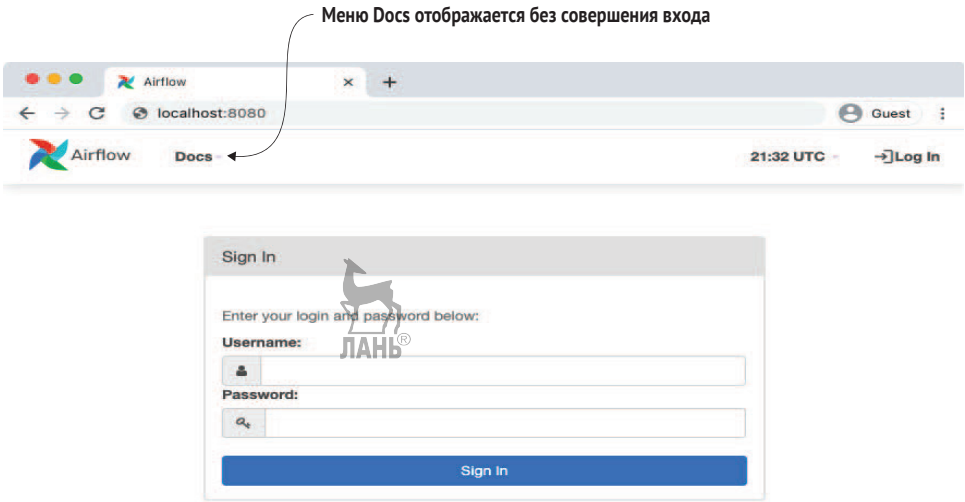


Рис. 13.6 Предоставление полномочий роли Public делает компоненты пользовательского интерфейса доступными для всех

Чтобы разрешить доступ к этим компонентам, нужно отредактировать роль Public и добавить ей правильные полномочия (рис. 13.7).

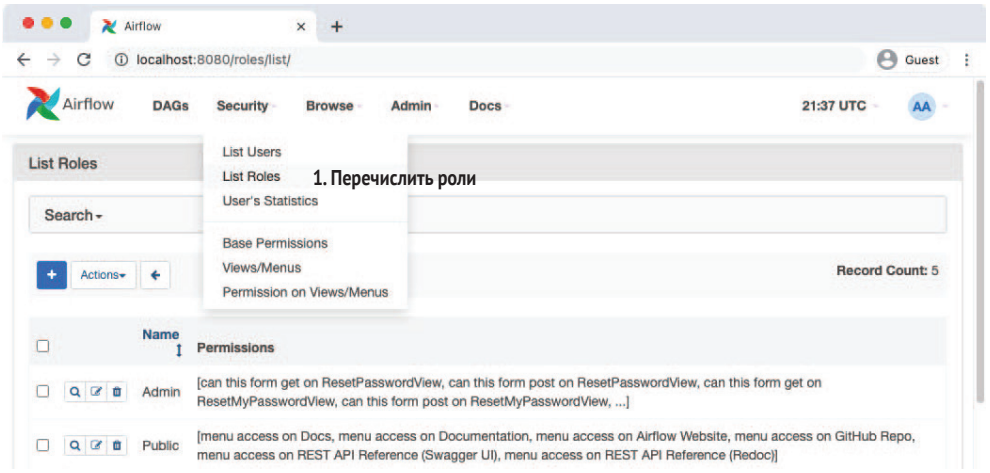


Рис. 13.7 Добавление полномочия для роли Public

Полномочия достаточно детализированы; доступ к каждому меню и пункту меню контролируется полномочиями. Например, чтобы сделать меню «Документы» видимым, нужно добавить полномочие «доступ в меню “Документы”». А чтобы сделать элемент меню «Документация» видимым в меню «Документы», нужно добавить полномочие «доступ в меню “Документация”». Иногда поиск правильных полномочий может быть обременительным.

Проще всего проверить другие роли, чтобы узнать, какие полномочия доступны. Полномочия отображаются в виде строки, которая в большинстве случаев не требует пояснений относительно предоставляемого доступа.

13.1.2 Настройка интерфейса RBAC

Как уже отмечалось, интерфейс RBAC разработан на основе фреймворка Flask-AppBuilder (FAB). При первом запуске веб-сервера RBAC вы найдете файл webserver_config.py в `$AIRFLOW_HOME`. FAB можно настроить с помощью файла config.py, но для ясности этот же файл в Airflow был назван webserver_config.py. Итак, этот файл содержит конфигурацию для FAB, фреймворка, лежащего в основе интерфейса RBAC.

Вы можете предоставить собственную конфигурацию интерфейсу RBAC, поместив файл webserver_ config.py в `$AIRFLOW_HOME`. Если Airflow не может найти файл, он сгенерирует для вас файл по умолчанию. Для получения всех подробностей и доступных опций в этом файле обратитесь к документации FAB. В ней содержатся все конфигурации для интерфейса RBAC (а не только те, которые связаны с безопасностью). Например, чтобы настроить тему для интерфейса RBAC, задайте для `APP_THEME` значение `"sandstone.css"` в файле webserver_config.py. Просмотрите документацию по FAB, чтобы увидеть все доступные темы (рис. 13.8).

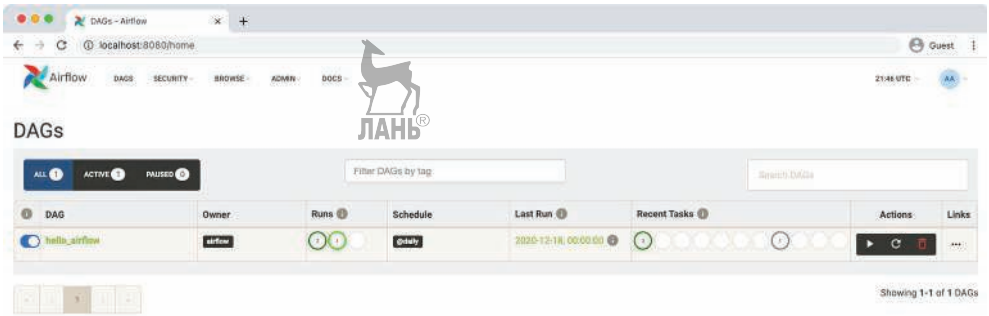


Рис. 13.8 Настройка интерфейса RBAC с использованием темы Sandstone

13.2 Шифрование хранимых данных

Интерфейс RBAC требует, чтобы пользователи находились в базе данных с именем пользователя и паролем. Это предотвращает доступ к Airflow со стороны случайных незнакомцев, которые «просто осматриваются», но это далеко не идеальный вариант. Прежде чем погрузиться в шифрование, вернемся к базовой архитектуре Airflow из рис. 12.1.

Airflow состоит из нескольких компонентов. Каждый из них представляет собой потенциальную угрозу, поскольку служит путем, через который незваные гости могут получить доступ к вашим системам (рис. 13.9). Уменьшение количества открытых точек входа (т. е. сужение *поверхности атаки*) всегда является хорошей идеей. Если вы должны открыть доступ к службе по практическим причинам, например к веб-серверу Airflow, нужно непременно убедиться, что он не является общедоступным[1].

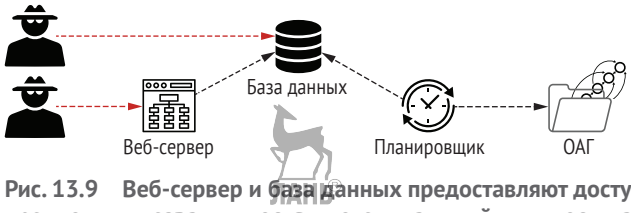


Рис. 13.9 Веб-сервер и база данных предоставляют доступ к службе и могут предложить незваным гостям потенциальный путь доступа к Airflow. Защищая их, вы сокращаете поверхность атаки

13.2.1 Создание ключа Fernet

Вы также хотите, чтобы ваши данные были в безопасности *после того*, как злоумышленник сумел получить к ним доступ. До того, как соз-

[1] В любом облаке легко можно предоставить доступ к службе через интернет. Простые меры, которые можно предпринять, чтобы избежать этого, включают в себя отказ от использования внешнего IP-адреса и/или блокировку всего трафика, допуская только свой диапазон IP-адресов.

давать пользователей и пароли, убедитесь, что в Airflow включено шифрование. Без шифрования пароли (и другие секреты, такие как подключения) хранятся в базе данных в незашифрованном виде. Любой, у кого есть доступ к базе данных, также может прочитать пароли. В зашифрованном виде они хранятся в виде последовательности кажущихся случайными символов, что, по сути, бесполезно. Airflow может шифровать и расшифровывать секреты, используя т. н. ключ Fernet (рис. 13.10).

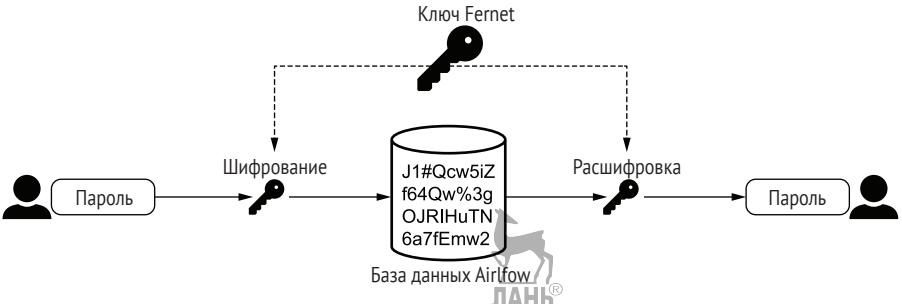


Рис. 13.10 Ключ Fernet шифрует данные, перед тем как сохранить их в базе данных, и расшифровывает данные перед их чтением из базы данных. Без доступа к ключу Fernet пароли бесполезны для злоумышленника. Когда один и тот же ключ используется для шифрования и дешифровки, такой способ называется *симметричным шифрованием*

Ключ Fernet – это строка, используемая для шифрования и дешифрования. Если этот ключ каким-то образом будет утерян, то зашифрованные сообщения больше нельзя будет расшифровать. Чтобы предоставить Airflow ключ Fernet, его можно сгенерировать.

Листинг 13.2 Создание ключа Fernet

```
from cryptography.fernet import Fernet

fernet_key = Fernet.generate_key()
print(fernet_key.decode())
# YlCImzjge_TeZc7jPJ7Jz2pgOtb4yTssA1pVyqIADWg=
```

Затем мы можем предоставить его Airflow, задав его в качестве значения элементу конфигурации `AIRFLOW__CORE__FERNET_KEY`:

```
AIRFLOW__CORE__FERNET_KEY=YlCImzjge_TeZc7jPJ7Jz2pgOtb4yTssA1pVyqIADWg=
```

Теперь Airflow будет использовать данный ключ для шифрования и дешифрования подключений, переменных и паролей пользователей. И мы можем создать своего первого пользователя и безопасно сохранить его пароль. Храните этот ключ в безопасности, поскольку любой, у кого есть доступ к нему, сможет расшифровать секреты; кроме того, вы не сможете расшифровать их, если вдруг потеряете его!

Чтобы не хранить ключ Fernet в виде обычного текста в переменной окружения, можно настроить Airflow для чтения значения из команды Bash (например, `cat/path/to/secret`). Саму команду можно задать в переменной окружения: `AIRFLOW__CORE __FERNET_KEY_CMD=cat/path/to/secret`. После этого файл, содержащий секретное значение, может быть доступен только для чтения и только пользователю Airflow.

13.3 Подключение к службе LDAP

Как было показано в разделе 13.1, мы можем создавать и хранить пользователей в самом Airflow. Однако в большинстве компаний обычно существуют системы управления пользователями. Разве не было бы удобнее подключить Airflow к такой системе, вместо того чтобы управлять собственным набором пользователей с еще одним паролем?

Популярный метод управления пользователями осуществляется через сервис, поддерживающий *протокол LDAP* (англ. Lightweight Directory Access Protocol – «легковесный протокол доступа к каталогам), например Azure AD или OpenLDAP, которые называются *службы каталогов*.

ПРИМЕЧАНИЕ В этом разделе мы будем использовать термин «служба LDAP» для обозначения службы каталогов, поддерживающей запросы по протоколу LDAP. Служба каталогов – это система хранения, обычно используемая для хранения информации о ресурсах, например пользователях и службах. LDAP – это протокол, по которому можно получить доступ к большинству этих служб каталогов.

Когда Airflow подключен к службе LDAP, информация о пользователе извлекается из службы в фоновом режиме после входа в систему (рис. 13.11).

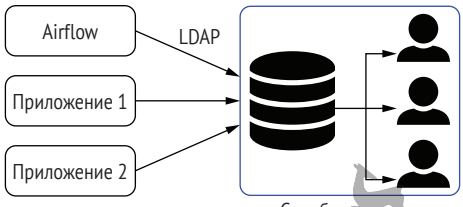


Рис. 13.11 Пользователи хранятся в службе каталогов, такой как Azure AD или OpenLDAP, доступ к которой можно получить с помощью протокола LDAP. Таким образом, пользователь создается только один раз и подключается ко всем приложениям

Сначала мы дадим небольшое введение в LDAP и соответствующие технологии (раздел 13.3.1), а затем продемонстрируем, как подключить Airflow к службе LDAP (раздел 13.3.2).

13.3.1 Разбираемся с LDAP

Связь между SQL и реляционной базой данных (например, PostgreSQL или MySQL) аналогична связи между LDAP и службой каталогов (например, Azure AD или OpenLDAP). Точно тоже, как в реляционной базе данных хранятся данные, а SQL используется для запроса данных, служба каталогов также хранит данные (хотя и в другой структуре), а LDAP используется для выполнения запросов к службе каталогов.

Однако реляционные базы данных и службы каталогов созданы для разных целей: реляционные базы данных предназначены для транзакционного использования любых данных, которые вы хотите сохранить, в то время как службы каталогов предназначены для больших объемов операций чтения, где данные соответствуют структуре, подобной телефонной книге (допустим, сотрудники компании или устройства в здании). Например, реляционная база данных больше подходит для поддержки платежной системы, поскольку платежи производятся часто и анализ платежей включает в себя разные типы агрегирования. С другой стороны, служба каталогов больше подходит для хранения учетных записей пользователей, поскольку они часто запрашиваются, но обычно не меняются.

В службе каталогов сущности (например, пользователи, принтеры или общие сетевые ресурсы) хранятся в иерархической структуре – *информационном дереве каталога*. Каждая сущность называется *записью*, где информация хранится в виде пар «ключ-значение» – *атрибутов* и *значений*. Кроме того, каждая запись обладает *уникальным именем*. Визуально данные в службе каталогов представлены на рис. 13.12.

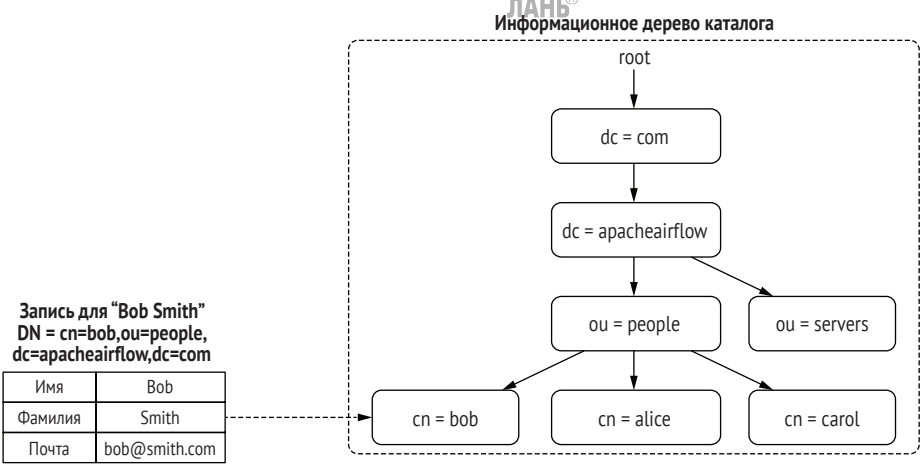


Рис. 13.12 Информация в службе каталогов хранится в виде иерархической структуры – информационном дереве каталога. Записи представляют собой сущность, например человек, и содержат атрибуты «ключ-значение»

Вы можете спросить, зачем мы демонстрируем эту иерархию и что означают сокращения *dc*, *ou* и *cn*. Хотя служба каталогов – это база данных, в которой теоретически можно хранить любые данные, существуют установленные требования LDAP в отношении того, как хранить и структурировать данные[1]. Согласно одному из соглашений дерево должно начинаться с т. н. *компонента домена* (dc), который мы видим на рис. 13.12, представленного как `dc=com` и `dc=apacheairflow`. Как следует из названия, это *компоненты* домена, поэтому домен вашей компании разделен точками, например `apacheairflow` и `com`.

Далее идут `ou=people` и `cn=bob`. ou – это сокращение от *organizational unit* (организационная единица), а cn – это *common name* (обычное имя). Хотя это ничего не говорит вам о том, как структурировать дерево, это часто используемые компоненты.

Стандарт LDAP определяет различные *объектные классы*, которые определяют сущность наряду с определенными ключами. Например, объектный класс *person* определяет человека с такими ключами, как *sn* (фамилия, обязательно) и *initials* (необязательно). Поскольку в стандарте определены такие объектные классы, приложения, читающие службу LDAP, обязательно всегда найдут фамилию человека в поле sn, и, таким образом, любое приложение, которое может выполнить запрос к службе LDAP, будет знать, где найти нужную информацию.

Теперь, когда мы знаем основные компоненты службы каталогов и то, как хранится информация, что же такое LDAP и как он соединяется со службой каталогов? Подобно тому, как SQL предоставляет определенные инструкции, такие как SELECT, INSERT, UPDATE и DELETE, LDAP предоставляет набор операций для службы каталогов (табл. 13.2).

Таблица 13.2 Обзор операций LDAP

Операция LDAP	Описание
`Abandon`	Отмена ранее запрошенной операции
`Add`	Создание новой записи
`Bind`	Авторизация в качестве заданного пользователя. Технически первое подключение к службе каталогов – анонимное. Затем операция bind изменяет личность данного пользователя, что позволяет выполнять определенные операции со службой каталогов
`Compare`	Проверяет, содержит ли данная запись заданное значение атрибута
`Delete`	Удаление записи
`Extended`	Запрос операции, не определенной стандартом LDAP, но доступной в службе каталогов (зависит от типа службы каталогов, к которой вы подключаетесь)
`Modify DN`	Изменяет уникальное имя записи
`Modify`	Редактирует атрибуты записи
`Search`	Поиск и возврат записей, соответствующих заданным критериям
`Unbind`	Закрывает соединение со службой каталогов

[1] Эти стандарты определены в RFC 4510-4519.

Только для извлечения информации о пользователе нам потребуются операции `bind` (чтобы аутентифицироваться как пользователь с полномочиями на чтение пользователей в службе каталогов), `search` (для поиска заданного уникального имени) и `unbind`, чтобы закрыть соединение.

Поисковый запрос содержит набор фильтров, обычно уникальное имя, выбирающее часть информационного дерева, а также несколько условий, которым должны соответствовать записи, например `uid=bsmith`. Это то, что делает любое приложение, запрашивающее службу LDAP, под капотом[1].

Листинг 13.3 Пример поиска в LDAP

```
ldapsearch -b "dc=apacheairflow,dc=com"
ldapsearch -b "dc=apacheairflow,dc=com" "(uid=bsmith)"
```

Перечисляет все записи в dc = apacheairflow, dc = com

Перечисляет все записи в dc = apacheairflow, dc = com, где uid = bsmith

Приложения, взаимодействующие со службой LDAP, будут выполнять такой поиск, чтобы извлечь и проверить информацию о пользователе для аутентификации в приложении.

13.3.2 Извлечение пользователей из службы LDAP

Аутентификация с протоколом LDAP поддерживается через FAB; поэтому нужно настроить его в файле webserver_config.py (в `$AIRFLOW_HOME`). При правильной настройке и после выполнения входа FAB будет искать в службе LDAP указанное имя пользователя и пароль.

Листинг 13.4 Настройка синхронизации LDAP в webserver_config.py

```
from flask_appbuilder.security.manager import AUTH_LDAP

AUTH_TYPE = AUTH_LDAP
AUTH_USER_REGISTRATION = True
AUTH_USER_REGISTRATION_ROLE = "User"

AUTH_LDAP_SERVER = "ldap://openldap:389"
AUTH_LDAP_USE_TLS = False
AUTH_LDAP_SEARCH = "dc=apacheairflow,dc=com"
AUTH_LDAP_BIND_USER = "cn=admin,dc=apacheairflow,dc=com"
AUTH_LDAP_BIND_PASSWORD = "admin"
AUTH_LDAP_UID_FIELD = "uid"
```

Роль по умолчанию, назначаемая любому пользователю, выполнившему вход

Раздел информационного дерева для поиска пользователей

Пользователь службы LDAP для подключения (привязки) и поиска

Имя поля в службе LDAP для поиска имени пользователя

Если они будут найдены, FAB разрешит найденному пользователю доступ к роли, настроенной с использованием `AUTH_USER_REGISTRA-`

[1] ldapsearch требует установки пакета ldap-utils.

TION_ROLE. На момент написания книги не существовало функции для сопоставления групп LDAP с ролями RBAC[1].

После настройки LDAP вам больше не нужно вручную создавать и сопровождать пользователей в Airflow. Все пользователи хранятся в службе LDAP, единственной системе, в которой будет храниться информация о пользователе, и все приложения (включая Airflow) смогут проверять учетные данные пользователя в службе LDAP без необходимости сопровождения собственных пользователей.

13.4 Шифрование трафика на веб-сервер

Злоумышленник может получить данные в различных местах вашей системы. Одно из таких мест – момент передачи данных между двумя системами, также известный как *передаваемые данные*. *Человек посередине* – это атака, при которой две системы или люди обмениваются данными друг с другом, в то время как третье лицо перехватывает трафик, читая сообщение (потенциально содержащее пароли и т. д.), и перенаправляет его так, чтобы никто не заметил перехват (рис. 13.13).

Перехват конфиденциальных данных неизвестным лицом – нежелательная ситуация, так как же защитить Airflow, чтобы передаваемые данные были в безопасности?

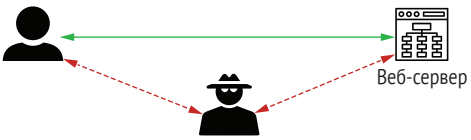


Рис. 13.13 При осуществлении атаки «человек посередине» злоумышленник перехватывает трафик между пользователем и веб-сервером Airflow. Трафик считывается и перенаправляется, при этом пользователь не замечает перехват, а злоумышленник получает отправленные данные

Подробнее описание того, как осуществляется данная атака, выходит за рамки этой книги, но мы обсудим, как смягчить ее последствия.

13.4.1 Разбираемся с протоколом HTTP

Мы можем работать с веб-сервером Airflow через браузер, который обменивается данными с Airflow по протоколу HTTP (рис. 13.14). Чтобы делать это безопасно, нужно использовать протокол HTTPS (HTTP Secure). Прежде чем защищать трафик, идущий на веб-сервер, разберемся в различиях между HTTP и HTTPS. Если вы уже знаете о них, можете перейти к разделу 13.4.2.

---

[1] Можно вручную отредактировать таблицу ab_user_role в базе метаданных, чтобы назначить другую роль (после первого входа).

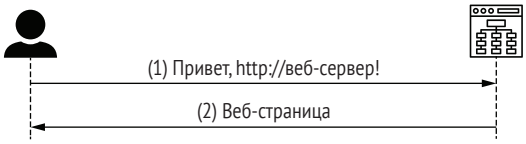


Рис. 13.14 При использовании протокола HTTP валидность вызывающего объекта не проверяется, а данные передаются в виде обычного текста

В чем их отличие? Чтобы понять, как работает HTTPS и для чего нужны закрытый ключ и сертификат, для начала выясним, как работает HTTP.

При переходе на сайт, где используется протокол HTTP, проверка подлинности запроса не выполняется ни с одной стороны (браузер или веб-сервер пользователя). Все современные браузеры отображают предупреждение о небезопасном соединении (рис. 13.15).

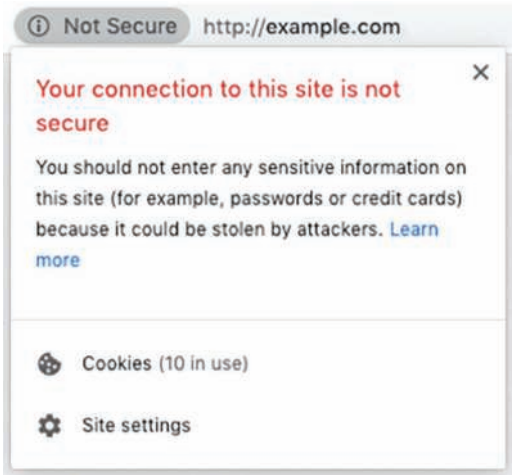


Рис. 13.15 При переходе на страницу http://example.com в Google Chrome будет отображаться сообщение «Небезопасно», поскольку трафик не защищен

Теперь, когда мы знаем, что HTTP-трафик небезопасен, чем нам может помочь HTTPS? Во-первых, с точки зрения пользователя, в современных браузерах отображается значок в виде замка или нечто зеленого цвета, указывая на действительный сертификат (рис. 13.16).

Когда ваш браузер и веб-сервер обмениваются данными по протоколу HTTPS, первоначальное квитирование включает в себя больше шагов для проверки действительности удаленной стороны (рис. 13.17).

Данные в протоколе HTTPS передаются поверх криптографических протоколов TLS (безопасность транспортного уровня), в которых используется *асимметричное* и *симметричное шифрование*. В то время как в симметричном шифровании для шифрования и дешифрования применяется один ключ, в асимметричном шифровании использует-

ся два ключа: открытый и закрытый. Магия асимметричного шифрования заключается в том, что данные, зашифрованные открытым ключом, можно расшифровать только с помощью закрытого ключа (который знает только веб-сервер), а данные, зашифрованные с помощью закрытого ключа, можно расшифровать только с помощью открытого ключа (рис. 13.18).

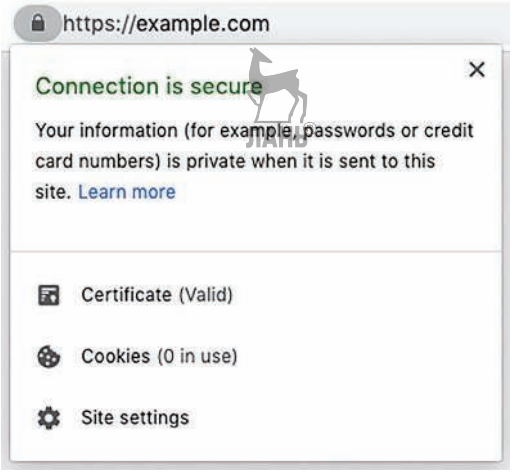


Рис. 13.16 При переходе на сайт, где используется протокол HTTPS в Google Chrome, отображается значок в виде замка (если сертификат действителен) для обозначения безопасного соединения

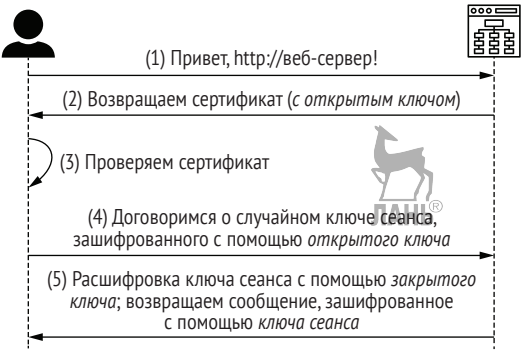


Рис. 13.17 В начале сеанса HTTPS браузер и веб-сервер согласовывают общий ключ сеанса для шифрования и дешифрования трафика

В начале HTTPS-сеанса веб-сервер сначала возвращает сертификат, представляющий собой файл с открытым ключом. Браузер возвращает случайно сгенерированный ключ сеанса на веб-сервер, зашифрованный открытым ключом. Только закрытый ключ может расшифровать это сообщение, доступ к которому должен иметь лишь веб-сервер. По этой причине важно никогда и никому не передавать закрытый ключ; любой, у кого он есть, может расшифровать трафик.

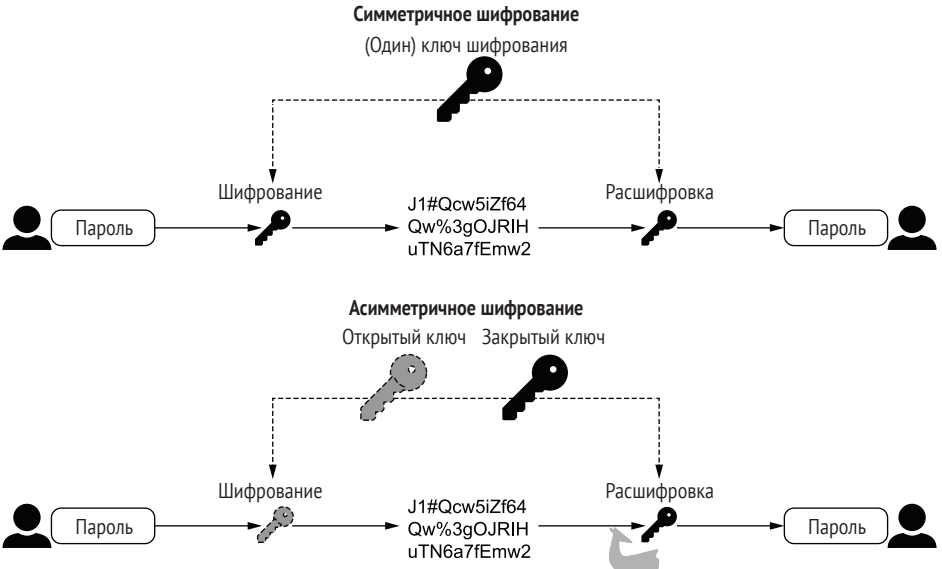


Рис. 13.18 Используя симметричное шифрование, потеря ключа шифрования позволяет другим шифровать и расшифровывать сообщения. При асимметричном шифровании открытый ключ используется совместно с другими, но его потеря не ставит под угрозу безопасность данных

13.4.2 Настройка сертификата для HTTPS

Airflow состоит из различных компонентов, и нужно избегать атак на них и между ними, независимо от того, используются ли они извне (например, когда речь идет о веб-сервере) или внутренне (если говорить о трафике между планировщиком и базой данных). Обнаружить и избежать атаки «злоумышленник посередине» может быть сложно. Однако сделать данные бесполезными для злоумышленника несложно, зашифровав трафик.

По умолчанию мы обмениваемся данными с Airflow по протоколу HTTP. Можно определить, зашифрован ли трафик, если посмотреть на URL-адрес, например http(s)://localhost: 8080. Весь HTTP-трафик передается в виде обычного текста; во время атаки «человек посередине» злоумышленник, читающий трафик, может перехватывать и читать пароли по мере их передачи. Когда вы используете протокол HTTPS, это означает, что данные шифруются с одной стороны и расшифровываются с другой. Во время вышеупомянутой атаки злоумышленник не сможет интерпретировать данные, поскольку они зашифрованы.

Посмотрим, как защитить одну общедоступную конечную точку в Airflow: веб-сервер. Нам понадобятся две вещи:

- открытый ключ (держите его в тайне);
- сертификат (им можно безопасно делиться с другими).

О том, что делать с ними дальше, мы поговорим позже. На данный момент важно знать, что закрытый ключ и сертификат – это файлы, предоставляемые *центром сертификации* или *самозаверенным сертификатом* (сертификатом, который вы создаете сами и который не подписан официальным центром сертификации).

Листинг 13.5 Создание самозаверенного сертификата

```
openssl req \
-x509 \
-newkey rsa:4096 \
-sha256 \
-nodes \
-days 365 \
-keyout privatekey.pem \
-out certificate.pem \
-extensions san \
-config \
 <(echo "[req]";
   echo distinguished_name=req;
   echo "[san]";
   echo subjectAltName=DNS:localhost,IP:127.0.0.1
   ) \
-subj "/CN=localhost"
```

Генерируем ключ сроком на один год

Имя файла закрытого ключа

Имя файла сертификата

Большинству браузеров требуется расширение SAN по соображениям безопасности

И закрытый ключ, и сертификат должны храниться в пути, доступном для Airflow, и Airflow следует запускать, используя:

- AIRFLOW__WEBSERVER__WEB_SERVER_SSL_CERT=/путь/к/certificate.pem;
- AIRFLOW__WEBSERVER__WEB_SERVER_SSL_KEY=/путь/к/privatekey.pem.

Запустите веб-сервер, и вы увидите, что он больше не обслуживается по адресу http://localhost: 8080. Теперь адрес выглядит так: https://localhost:8080 (рис. 13.19).

На данном этапе трафик между вашим браузером и веб-сервером Airflow зашифрован. Хотя злоумышленник может перехватить трафик, для него он будет бесполезен, поскольку он зашифрован, а следовательно, его нельзя прочитать. Расшифровать данные можно только с помощью закрытого ключа; вот почему так важно никогда и никому не давать закрытый ключ и хранить его в надежном месте.

При использовании самозаверенного сертификата, созданного в листинге 13.5, вы сначала получите предупреждение (на рисунке показана страница из Chrome 13.20).

На вашем компьютере есть список доверенных сертификатов и их расположение в зависимости от операционной системы. В большинстве систем Linux доверенные сертификаты хранятся в каталоге /etc/ssl/certs. Эти сертификаты предоставляются с вашей операционной системой и согласованы с различными органами. Они

позволяют перейти на страницу https://www.google.com, получить сертификат Google и проверить его в списке сертификатов, потому что сертификат Google поставляется вместе с вашей операционной системой[1].

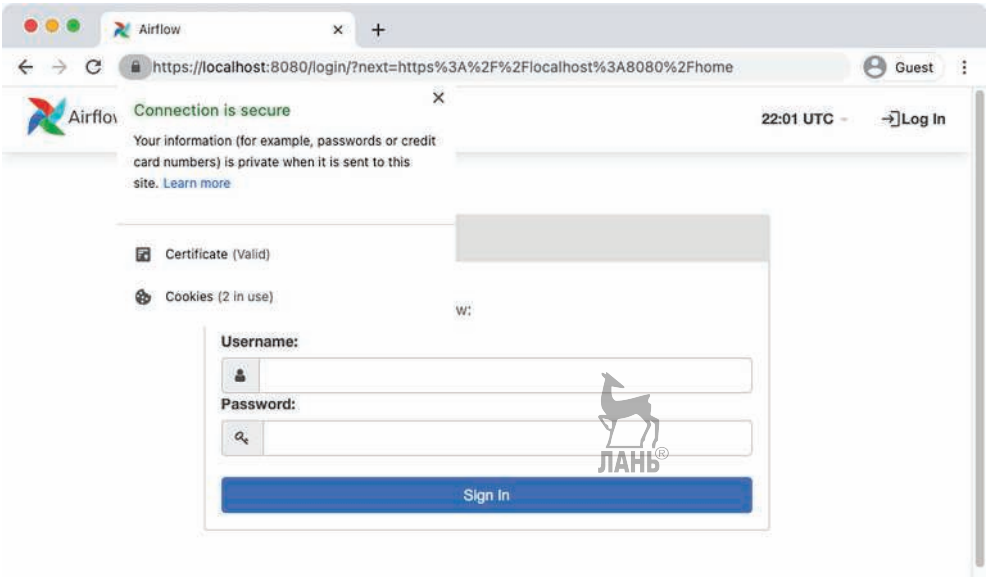


Рис. 13.19 После предоставления сертификата и закрытого ключа веб-сервер обслуживается по адресу https://localhost:8080. Обратите внимание, что для localhost нельзя выдать официальный сертификат; следовательно, он должен быть самозаверенным. Самозаверенные сертификаты по умолчанию не являются доверенными, поэтому вы должны добавить его к своим доверенным сертификатам

Если вы переходите на сайт, сообщающий о том, что сертификата нет в списке, браузер отобразит предупреждение, как в случае использования нашего самозаверенного сертификата. Следовательно, нужно дать компьютеру указание доверять нашему сгенерированному сертификату. Мы знаем, что сами сгенерировали его, поэтому ему можно доверять.

Как велеть компьютеру доверять сертификату? Все зависит от используемой операционной системы. В случае с macOS это включает в себя открытие приложения Keychain Access и импорт вашего сертификата в системную связку ключей (рис. 13.21).

[1] Для ясности различные технические детали опущены. Хранить миллиарды доверенных сертификатов для всех сайтов непрактично. Вместо этого на вашем компьютере хранится несколько сертификатов наверху цепочки. Сертификаты выдаются определенными доверенными центрами.

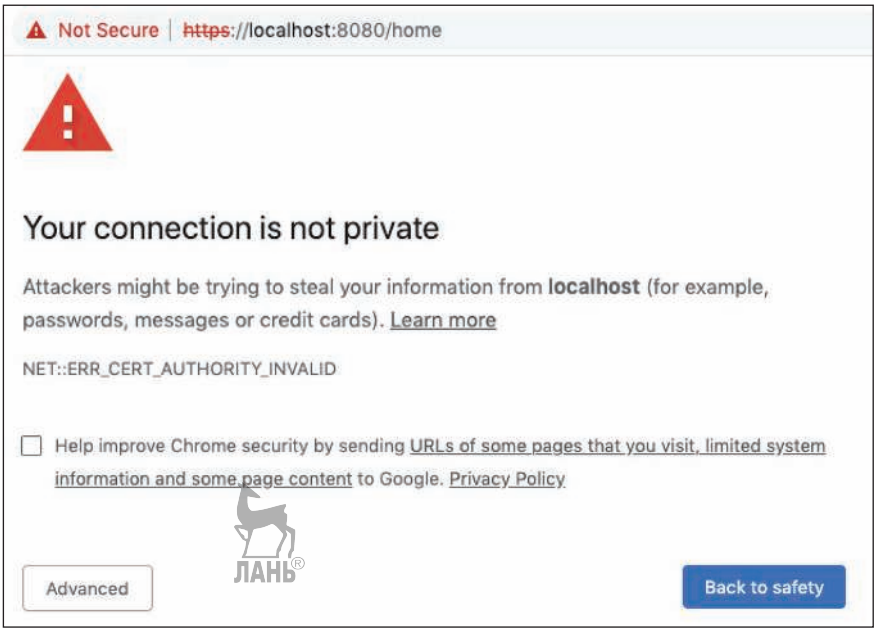


Рис. 13.20 Большинство браузеров отображают предупреждения при использовании самозаверенных сертификатов, потому что их действительно нельзя проверить

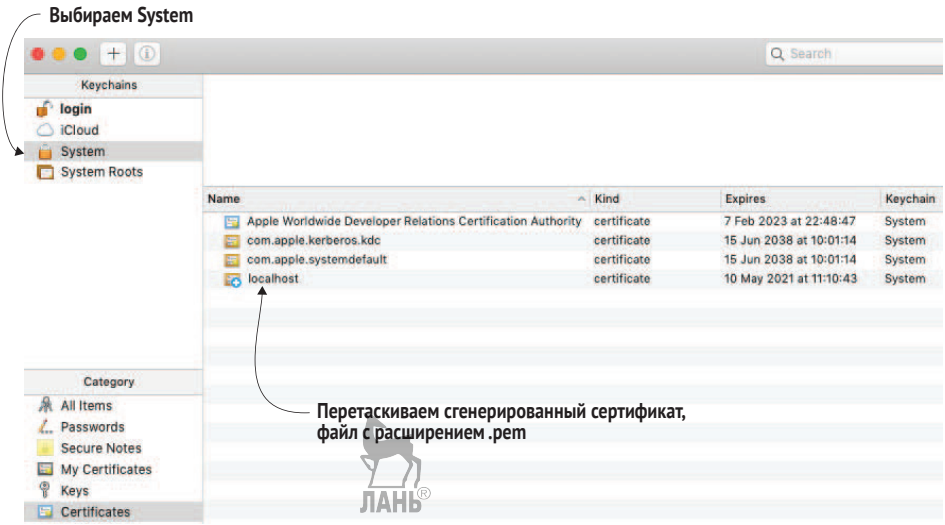


Рис. 13.21 Добавление самозаверенного сертификата к системным сертификатам в macOS

После этого сертификат станет известен системе, но все еще не будет доверенным. Чтобы доверять ему, нужно явно доверять SSL при обнаружении самозаверенного сертификата (рис. 13.22).

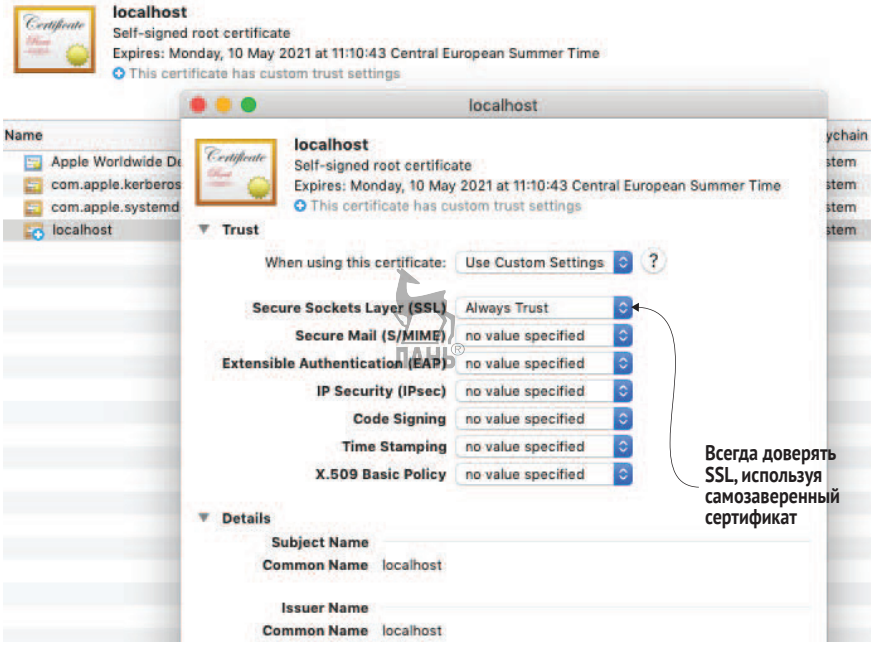


Рис. 13.22 Доверие к SSL с использованием самозаверенного сертификата обеспечивает доверие между компьютером и веб-сервером Airflow

Если вы размещаете Airflow по адресу, доступному для других лиц (т. е. не на локальном хосте), всем придется пройти через трудности, связанные с доверием самозаверенному сертификату. Это явно нежелательно; поэтому сертификаты выпускаются через доверенный центр, и их можно проверить. Для получения дополнительной информации поищите в интернете «Сертификат TLS» (для приобретения сертификата) или «Let's Encrypt» (для получения бесплатных криптографических сертификатов для шифрования передаваемых через интернет данных).

13.5 Извлечение учетных данных из систем управления секретами

Многие компании применяют центральную систему хранилища секретов, позволяющую сохранять конфиденциальные данные (пароли, сертификаты, ключи и т. д.) только один раз в одной системе. Приложения имеют возможность запрашивать эти данные, когда им это нужно, без необходимости хранить собственные. В качестве примера можно привести HashiCorp Vault, Azure Key Vault, AWS SSM и GCP Secrets Manager. Это позволяет избежать разброса конфиденциальных данных по разным системам и вместо этого хранить все в одной си-

стеме, разработанной специально для хранения и управления секретами. Кроме того, эти системы предоставляют такие функции, как ротация секретов и управление версиями, чего нет в Airflow.

Секретные значения в Airflow могут храниться в переменных и соединениях. Разве не было бы удобнее и безопаснее подключиться к одной из этих систем хранения, вместо того чтобы копировать и вставлять конфиденциальные данные в Airflow? В Airflow версии 1.10.10 была представлена новая функция под названием *Secrets Backend*, которая предоставляет механизм извлечения конфиденциальных данных из внешних систем хранения, при этом используя существующие классы переменных и соединений.

На момент написания данной главы поддерживаются такие системы, как AWS SSM, GCP Secret Manager и HashiCorp Vault. Вам предоставляется универсальный класс, который можно разделить на подклассы для реализации вашей собственной системы хранения секретов и подключения к ней. Рассмотрим пример с использованием HashiCorp Vault.

Листинг 13.6 Извлечение сведений о подключении

```
import airflow.utils.dates
from airflow.models import DAG
from airflow.providers.http.operators.http import SimpleHttpOperator
dag = DAG(
   dag_id="secretsbackend_with_vault",
   start_date=airflow.utils.dates.days_ago(1),
   schedule_interval=None,
)

call_api = SimpleHttpOperator(
   task_id="call_api",
   http_conn_id="secure_api",
   method="GET",
   endpoint="",
   log_response=True,
   dag=dag,
)
```

Обозначает идентификатор секрета в Vault

Как видно из листинга 13.5, в коде вашего OAГ нет явной ссылки на HashiCorp Vault. `SimpleHttpOperator` выполняет HTTP-запрос, в данном случае к URL-адресу, заданному в подключении. Раньше нужно было сохранять URL-адреса в подключении. Теперь мы можем сохранить его (среди прочего) в HashiCorp Vault. При этом следует отметить несколько моментов:

- бэкенды секретов должны быть настроены с помощью `AIRFLOW__SECRETS__BACKEND` и `AIRFLOW__SECRETS__BACKEND_KWARGS`;
- все секреты должны иметь общий префикс;
- все соединения должны храниться в ключе «conn_uri»;
- все переменные должны храниться в ключе «value».

Имя секрета хранится в виде пути (это относится ко всем менеджерам секретов), например `secret/connections/secure_api`, где `secret` и `connections` можно рассматривать как папки, используемые для организации, а `secure_api` – это имя, идентифицирующее фактический секрет.

ПРИМЕЧАНИЕ Префикс «secret» используется конкретно для бэкенда Vault. Обратитесь к документации по Airflow, чтобы получить всю информацию о выбранном вами бэкенде.

Иерархическая организация секретов во всех системах управления секретами позволяет Airflow предоставить универсальный бэкенд секретов для взаимодействия с такими системами. В разделе «Secrets Engines» в HashiCorp Vault секрет будет храниться, как показано на рис. 13.23.

Рис. 13.23 Секреты в Vault хранятся в разделе «Secrets Engines», который может хранить секреты в различных системах. По умолчанию вы получаете движок с именем «секрет» для хранения секретов типа «ключ-значение»

В движке Vault мы создаем секрет connections/secure_api. Хотя префикс «connections/» не является обязательным, бэкэнд секретов Airflow принимает префикс, под которым он может искать секреты, что удобно для поиска только в одной части иерархии секретов в Vault.

Сохранение соединения Airflow в любом бэкэнде требует установки ключа с именем conn_uri, который будет запрашивать Airflow (рис. 13.24). Соединение должно быть указано в формате URI. Эти данные будут переданы в класс соединения Airflow, где из URI извлекаются необходимые детали.

Скажем, у нас есть API, работающий на имени хоста secure_api, порт 5000, и для аутентификации ему требуется заголовок с именем «token» и значением «supersecret». Для анализа в соединение Airflow сведения об API должны быть сохранены в формате URI, как показано на рис. 13.24: `http://secure_api:5000?token=supersecret`.

В Airflow мы должны задать два параметра конфигурации для извлечения учетных данных. Во-первых, для `AIRFLOW__SECRETS__BACKEND` нужно задать класс, читающий секреты:

- HashiCorp Vault: airflow.providers.hashicorp.secrets.vault.VaultBackend;
- AWS SSM: airflow.providers.amazon.aws.secrets.systems_manager.Systems- ManagerParameterStoreBackend;

- GCP Secret Manager: airflow.providers.google.cloud.secrets.secrets_manager.CloudSecretsManagerBackend.

Рис. 13.24 Для сохранения сведений о подключении Airflow в Vault необходимо задать ключ: conn_uri

Затем в `AIRFLOW__SECRETS__BACKEND_KWARGS` необходимо настроить различные детали, относящиеся к выбранному бэкенду секретов. Для получения всех подробностей о бэкендах обратитесь к документации по Airflow. Возьмем, например, `BACKEND_KWARGS` для Vault:

```
{"url":"http://vault:8200","token":"airflow","connections_path":"connections"}.
```

Здесь `"url"` указывает на URL-адрес Vault, `"token"` означает токен аутентификации в Vault, а `"connections_path»` относится к префиксу для запроса всех подключений. В бэкенде Vault префикс по умолчанию для всех секретов (как подключений, так и переменных) задан как `secret`. В результате полный поисковый запрос с conn_id, «secure_api» принимает следующий вид: `secret/connections/secure_api`.

Бэкенд не заменяет секреты, хранящиеся в переменных окружения или базе метаданных Airflow. Это альтернативное место для хранения секретов. Порядок их извлечения выглядит так:

1. бэкенд секретов;
2. переменные окружения (`AIRFLOW_CONN_*` и `AIRFLOW_VAR_*`);
3. база метаданных Airflow.

Создав бэкенд, мы передали хранение конфиденциальной информации и управление ею системе, разработанной специально для этой цели. Другие системы также могут подключаться к системе управления секретами, чтобы вы сохраняли секретное значение только один раз, вместо того чтобы распределять его среди множества систем,

каждую из которых можно взломать. В результате поверхность атаки становится меньше.

Технически возможности взлома систем безграничны. Тем не менее мы продемонстрировали различные способы защиты данных как внутри, так и за пределами Airflow – и все это с целью ограничения количества возможных вариантов, которыми может воспользоваться злоумышленник, и защиты от некоторых наиболее распространенных способов получения нежелательного доступа. В заключение убедитесь, что вы всегда следите за новыми релизами Airflow, поскольку иногда они содержат исправления безопасности, закрывающие ошибки в старых версиях.

Резюме

- В целом безопасность не сосредоточивается на одном элементе, а включает в себя обеспечение различных уровней вашего приложения, чтобы ограничить потенциальную поверхность атаки.
- В интерфейсе RBAC есть механизм безопасности на основе ролей, позволяющий выполнять определенные действия группам, в которые организованы пользователи.
- Перехват трафика между клиентом и веб-сервером Airflow можно сделать бесполезным, применив шифрование по протоколу TLS.
- Учетные данные в базе данных Airflow можно сделать нечитаемыми для злоумышленника с помощью ключа Fernet.
- Систему управления секретами, такую как HashiCorp Vault, можно использовать для хранения и управления секретами, чтобы все конфиденциальные данные находились в одном месте и использовались только при необходимости такими приложениями, как Airflow.

14 Проект: поиск самого быстрого способа передвижения по Нью-Йорку

Эта глава рассказывает:

- как настроить конвейер Airflow с нуля;
- о структурировании промежуточных выходных данных;
- о разработке идемпотентных задач;
- о реализации одного оператора для обработки нескольких похожих преобразований.

Транспорт в Нью-Йорке может быть перегружен. Всегда час пик, но, к счастью, альтернативных способов передвижения сейчас больше, чем когда-либо. В мае 2013 года в Нью-Йорке начала свою деятельность система проката велосипедов Citi Bike с 6000 велосипедов. С годами Citi Bike выросла и расширилась, и велосипед стал популярным средством передвижения по городу.

Еще один знаковый способ передвижения – такси Yellow Cab. Такси появились в Нью-Йорке в конце 1890-х годов и всегда пользовались популярностью. Однако в последние годы количество водителей такси резко упало, и многие водители стали работать с такими сервисами, как Uber и Lyft.

Независимо от того, какой вид транспорта вы выберете для передвижения по Нью-Йорку, обычно ваша цель состоит в том, чтобы как можно быстрее добраться из пункта А в пункт Б. К счастью, Нью-Йорк

активно публикует данные, включая информацию о поездках Citi Bikes и Yellow Cab.

В этой главе мы пытаемся ответить на вопрос: «Если бы я поехал из пункта А в пункт Б в Нью-Йорке прямо сейчас, какой способ передвижения самый быстрый?» Мы создали мини-проект Airflow, чтобы извлекать и загружать данные, преобразовывать их в пригодный для использования формат и запрашивать, какой способ передвижения быстрее, в зависимости от районов, между которыми вы путешествуете, и времени суток[1].

Чтобы сделать этот мини-проект воспроизводимым, был создан файл Docker Compose, запускающий несколько служб в контейнерах Docker. Он включает в себя:

- один REST API, обслуживающий данные по Citi Bike;
- один общий файловый ресурс, обслуживающий данные по Yellow Cab;
- MinIO, хранилище объектов, поддерживающее протокол S3;
- базу данных PostgreSQL для запросов и хранения данных;
- приложение Flask, отображающее результаты.

Таким образом, мы получаем строительные блоки, показанные на рис. 14.1.

Рис. 14.1 Файл Docker Compose создает несколько сервисов. Наша задача – загрузить данные из REST API, поделиться ими и преобразовать их, чтобы в конечном итоге просмотреть самый быстрый способ передвижения на получившейся веб-странице

Наша цель в этой главе – использование этих строительных блоков для извлечения данных из REST API, а также совместное применение и разработка конвейера обработки данных, соединяющего эти точки. Мы выбрали MinIO, поскольку AWS S3 часто используется для хранения данных, а MinIO поддерживает протокол S3. Результаты анализа будут записаны в базу данных PostgreSQL, а итоги будут отображены на веб-странице. Для начала убедитесь, что в вашем текущем каталоге находится файл docker-compose.yml, и создайте все контейнеры.

Листинг 14.1 Запуск стандартных блоков для нашего примера в контейнерах Docker

```
$ docker-compose up -d
Creating network "airflow-use-case_default" with the default driver
```

[1] Некоторые идеи в этой главе основаны на посте из блога Тодда Шнайдера (https://toddwschneider.com/posts/taxi-vs-citi-bike-nyc/), где он анализирует самый быстрый способ передвижения, применяя симуляцию Монте-Карло.

```
Creating volume "airflow-use-case_logs" with default driver
Creating volume "airflow-use-case_s3" with default driver
Creating airflow-use-case_result_db_1              ... done
Creating airflow-use-case_citibike_db_1            ... done
Creating airflow-use-case_minio_1                  ... done
Creating airflow-use-case_postgres_1
Creating airflow-use-case_nyc_transportation_api_1 ... done
Creating airflow-use-case_taxi_db_1                ... done
Creating airflow-use-case_webserver_1              ... done
Creating airflow-use-case_initdb_adduser_1         ... done
Creating airflow-use-case_scheduler_1              ... done
Creating airflow-use-case_minio_init_1             ... done
Creating airflow-use-case_citibike_api_1           ... done
Creating airflow-use-case_taxi_fileserver_1        ... done
```

Это предоставляет нам доступ к следующим службам на локальном хосте: [порт], с [именем пользователя] / [паролем], указанными в скобках:

- 5432: хранилище метаданных PostgreSQL Airflow (`airflow/airflow`);
- 5433: база данных Postgres для такси Нью-Йорка (`taxi/ridetlc`);
- 5434: база данных Postgres для Citi Bike (`citi/cycling`);
- 5435: база данных Postgres с результатами передвижения по Нью-Йорку (`nyc/tr4N5p0RT4TI0N`);
- 8080: веб-сервер Airflow (`airflow/airflow`);
- 8081: статический файловый сервер такси Нью-Йорка;
- 8082: API Citi Bike (`citibike/cycling`);
- 8083: веб-страница транспорта Нью-Йорка;
- 9000: MinIO (`AKIAIOSFODNN7EXAMPLE/wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY`).

Данные о поездках как для Yellow Cab, так и для Citi Bikes предоставляются ежемесячными партиями:

- желтое такси Нью-Йорка: https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page;
- Citi Bike: https://www.citibikenyc.com/system-data.

Цель данного проекта – продемонстрировать реальное окружение с реальными проблемами, с которыми вы можете столкнуться, и как с ними бороться в Airflow. Наборы данных выходят раз в месяц. Интервалы длительностью в один месяц довольно велики, поэтому мы создали два API в настройке Docker Compose, которые предоставляют одни и те же данные, но с настраиваемыми интервалами до одной минуты. Кроме того, API имитируют ряд характеристик рабочих систем, таких как аутентификация.

Посмотрим на карту Нью-Йорка, чтобы понять, как определить самый быстрый способ передвижения (рис. 14.2).

Рис. 14.2 Зоны желтого такси Нью-Йорка с указанием местоположения станций Citi Bike

Ясно видно, что станции Citi Bike расположены только в центре Нью-Йорка. Поэтому, чтобы дать какой-либо значимый совет о наиболее быстром способе передвижения, мы ограничены теми зонами, где присутствуют как Citi Bikes, так и Yellow Cab. В разделе 14.1 мы проверим данные и разработаем план подхода.

14.1 Разбираемся с данными

Файл Docker Compose предоставляет две конечные точки с данными по Yellow Cab и Citi Bike:

- данные по Yellow Cab на http://localhost:8081;
- данные по Citi Bike на http://localhost:8082.

Посмотрим, как делать запрос к этим конечным точкам и какие данные они возвращают.

14.1.1 Файловый ресурс Yellow Cab

Данные Yellow Cab доступны по адресу http://localhost:8081. Они представлены в виде статических CSV-файлов, где каждый файл содержит поездки на такси, завершенные за последние 15 минут. Он будет хранить только один полный час данных; данные старше одного часа автоматически удаляются. Никакой аутентификации не требуется.

Листинг 14.2 Пример запроса к общему файловому ресурсу Yellow Cab

```
$ curl http://localhost:8081
[
  ➥ { "name":"06-27-2020-16-15-00.csv", "type":"file", "mtime":"Sat, 27 Jun
     2020 16:15:02 GMT", "size":16193 },
  ➥ { "name":"06-27-2020-16-30-00.csv", "type":"file", "mtime":"Sat, 27 Jun
     2020 16:30:01 GMT", "size":16580 },
  ➥ { "name":"06-27-2020-16-45-00.csv", "type":"file", "mtime":"Sat, 27 Jun
     2020 16:45:01 GMT", "size":13728 },
  ➥ { "name":"06-27-2020-17-00-00.csv", "type":"file", "mtime":"Sat, 27 Jun
     2020 17:00:01 GMT", "size":15919 }
]
```

Индекс возвращает список доступных файлов. Каждый из них представляет собой CSV-файл, содержащий информацию о поездках в Yellow Cab, завершенных за последние 15 минут, в то время, которое указано в имени файла.

Листинг 14.3 Пример фрагмента файла Yellow Cab

```
$ curl http://localhost:8081/06-27-2020-17-00-00.csv
➥ pickup_datetime,dropoff_datetime,pickup_locationid,dropoff_locationid,
     trip_distance
2020-06-27 14:57:32,2020-06-27 16:58:41,87,138,11.24
2020-06-27 14:47:40,2020-06-27 16:46:24,186,35,11.36
2020-06-27 14:47:01,2020-06-27 16:54:39,231,138,14.10
2020-06-27 15:39:34,2020-06-27 16:46:08,28,234,12.00
2020-06-27 15:26:09,2020-06-27 16:55:22,186,1,20.89
...
```

Мы видим, что каждая строка представляет собой одну поездку на такси с указанием времени начала и окончания, а также идентификаторов начальной и конечной зон.

14.1.2 REST API Citi Bike

Данные по Citi Bike доступны по адресу http://localhost: 8082, который предоставляет данные через REST API. Этот API обеспечивает базовую аутентификацию, то есть мы должны указать имя пользователя и пароль. API возвращает поездки Citi Bike, завершенные в течение настраиваемого периода времени.

Листинг 14.4 Пример запроса к REST API Citi Bike

```
$ date
Sat 27 Jun 2020 18:41:07 CEST

$ curl --user citibike:cycling http://localhost:8082/recent/hour/1
[
 {
  "end_station_id": 3724,
  "end_station_latitude": 40.7667405590595,
  "end_station_longitude": -73.9790689945221,
  "end_station_name": "7 Ave & Central Park South",
  "start_station_id": 3159,
  "start_station_latitude": 40.77492513,
  "start_station_longitude": -73.98266566,
  "start_station_name": "W 67 St & Broadway",
  "starttime": "Sat, 27 Jun 2020 14:18:15 GMT",
  "stoptime": "Sat, 27 Jun 2020 15:32:59 GMT",
  "tripduration": 4483
 },
 {
  "end_station_id": 319,
  "end_station_latitude": 40.711066,
  "end_station_longitude": -74.009447,
  "end_station_name": "Fulton St & Broadway",
  "start_station_id": 3440,
  "start_station_latitude": 40.692418292578466,
  "start_station_longitude": -73.98949474096298,
  "start_station_name": "Fulton St & Adams St",
  "starttime": "Sat, 27 Jun 2020 10:47:18 GMT",
  "stoptime": "Sat, 27 Jun 2020 16:27:21 GMT",
  "tripduration": 20403
 },
 ...
]
```

Запрос данных за последний час

Каждый объект JSON представляет собой одну поездку на Citi Bike

Здесь мы запрашиваем количество поездок Citi Bike, завершенных за последний час. Каждая запись в ответе обозначает одну поездку и предоставляет координаты широты и долготы начального и конечного местоположений, а также время начала и окончания поездки. Конечную точку можно настроить на возврат поездок с меньшими или большими интервалами:

```
http://localhost:8082/recent/<period>/<amount>
```

где `<period>` может быть минута, час или день. `<amount>` – это целое число, обозначающее количество заданных периодов. Например, запрос `http://localhost:8082/latest/day/3` вернет все поездки Citi Bike, завершенные за последние три дня.

API не знает ограничений по размеру запроса. Теоретически мы могли бы запрашивать данные за бесконечное количество дней. На

практике API-интерфейсы часто ограничивают вычислительную мощность и размер передаваемых данных. Например, API может ограничить количество результатов до 1000. При таком ограничении вам нужно будет знать, сколько поездок на велосипеде (приблизительно) выполняется в течение определенного времени, и достаточно часто отправлять запросы, чтобы получить все данные, не превышая максимальное число: 1000 результатов.

14.1.3 Выбор плана подхода

Теперь, когда мы ознакомились с примерами данных из листингов 14.3 и 14.4, изложим факты и решим, что делать дальше. Чтобы сравнить одно с другим, нужно сопоставить местоположения в обоих наборах данных с чем-то общим. Данные о поездках в Yellow Cab содержат идентификаторы зон такси, а данные Citi Bike – координаты широты и долготы велосипедных станций. Давайте сделаем все проще, задействовав наш пример, но немного пожертвовав точностью, и сопоставим широту и долготу станций Citi Bike с зонами такси (рис. 14.3).

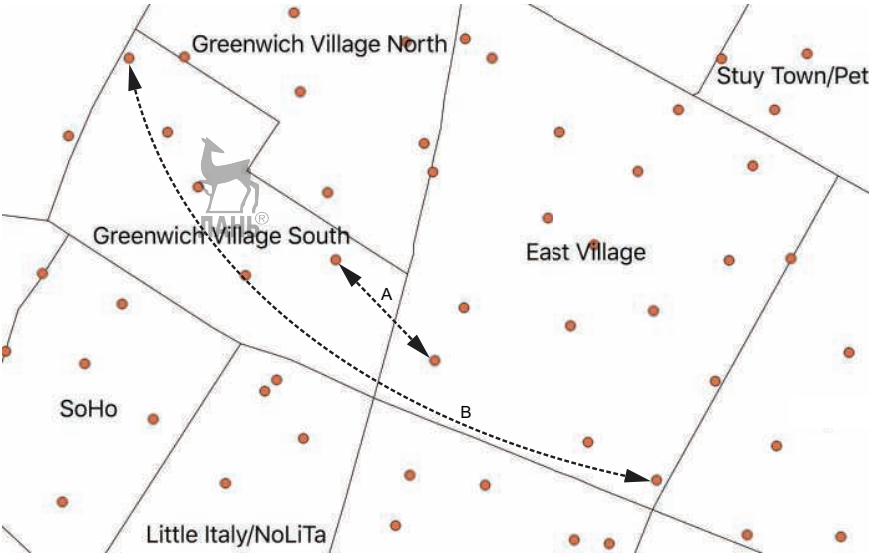


Рис. 14.3 Сопоставление станций Citi Bike (точки) с зонами Yellow Cab обеспечивает точное сравнение, но не учитывает тот факт, что поездки в пределах одной зоны могут отличаться по расстоянию. Поездка A, очевидно, короче, чем поездка B. Усредняя время поездки из юга Гринвич-Виллидж до Ист-Виллидж, вы теряете эту информацию

Поскольку данные по Yellow Cab предоставляются в общей папке только за один час, мы должны скачать и сохранить их в собственных системах. Таким образом, мы создаем коллекцию архивных данных о поездках на такси с течением времени и всегда можем вернуться

к скачанным данным, если изменим обработку. Как уже упоминалось, файл Docker Compose создает службу MinIO, которая представляет собой службу хранилища объектов, поэтому мы будем использовать ее для хранения извлеченных данных.

14.2 Извлечение данных

При извлечении нескольких источников данных важно учитывать временные интервалы данных. Данные по Yellow Cab доступны с 15-минутными интервалами, а интервал данных по Citi Bike можно настроить. Чтобы упростить задачу, давайте также будем запрашивать данные по Citi Bike с 15-минутными интервалами. Это позволяет делать два запроса с одним и тем же интервалом в одном ОАГ и обрабатывать все данные параллельно. Если мы выберем другой интервал, то должны будем по-разному согласовать обработку обоих наборов данных.

Листинг 14.5 ОАГ, запускаемый каждые 15 минут

```
import airflow.utils.dates
from airflow.models import DAG

dag = DAG(
    dag_id="nyc_dag",
    schedule_interval="*/15 * * * *",
    start_date=airflow.utils.dates.days_ago(1),
    catchup=False,
)
```

Запуск каждые 15 минут

14.2.1 Скачиваем данные по Citi Bike

В Airflow у нас есть оператор `SimpleHttpOperator` для выполнения вызовов по протоколу HTTP. Однако быстро оказалось, что он не подходит для нашего случая использования: этот оператор просто выполняет HTTP-запрос, но не предоставляет никаких функций для хранения ответа[1]. В такой ситуации мы вынуждены реализовать собственные функции и использовать `PythonOperator`.

Посмотрим, как выполнить запрос к API Citi Bike и сохранить результат в хранилище объектов MinIO.

Листинг 14.6 Скачиваем данные из REST API Citi Bike в MinIO

```
import json

import requests
from airflow.hooks.base import BaseHook
```

[1] Задав для `xcom_push` значение `True`, можно сохранить вывод в XCom.

```
from airflow.models import DAG
from airflow.operators.python import PythonOperator
from airflow.providers.amazon.aws.hooks.s3 import S3Hook
from requests.auth import HTTPBasicAuth

def _download_citi_bike_data(ts_nodash, **_):
    citibike_conn = BaseHook.get_connection(conn_id="citibike")

    ➥ url = f"http://{citibike_conn.host}:{citibike_conn.port}/recent/minute/15"
    ➥ response = requests.get(url, auth=HTTPBasicAuth(citibike_conn.login,
      citibike_conn.password))
    data = response.json()

    s3_hook = S3Hook(aws_conn_id="s3")
    s3_hook.load_string(
        string_data=json.dumps(data),
        key=f"raw/citibike/{ts_nodash}.json",
        bucket_name="datalake"
    )


download_citi_bike_data = PythonOperator(
    task_id="download_citi_bike_data",
    python_callable=_download_citi_bike_data,
    dag=dag,
)
```

Загружаем учетные данные Citi Bike из соединения Airflow

Используем временную метку задачи Airflow в итоговом имени файла

Используем S3Hook для обмена данными с MinIO

У нас нет оператора Airflow, который можно было бы использовать для этой конкретной операции «из HTTP-в-S3», но мы можем применить хуки и подключения. Сначала нужно подключиться к API Citi Bike (используя библиотеку запросов Python) и хранилищу MinIO (используя S3Hook). Поскольку оба они требуют учетных данных для аутентификации, мы сохраним их в Airflow для загрузки во время выполнения.

Листинг 14.7 Задаем детали подключения через переменные окружения

```
➥ export AIRFLOW_CONN_CITIBIKE=http://citibike:cycling@citibike_api:5000
➥ export AIRFLOW_CONN_S3="s3://@?host=http://minio:9000&aws_access_key_id
  =AKIAIOSFODNN7EXAMPLE&aws_secret_access_key=wJalrXUtnFEMI/K7MDENG/bPxRfi
  CYEXAMPLEKEY"
```

Пользовательский хост S3 должен быть предоставлен таким образом

По умолчанию хук S3 обменивается данными с AWS S3 на сайте https://aws.amazon.com/ru/s3/. Поскольку мы запускаем MinIO по другому адресу, то должны указать его в деталях подключения. К сожалению, это непростая задача, и иногда такие странности приводят к необходимости читать реализацию хука, чтобы понять, как он устроен. В случае с S3 имя хоста можно предоставить через ключевой хост (рис.14.4).

```
AIRFLOW_CONN_S3=s3://@?host=http://minio:9000&aws_access_key_id=...&aws_secret_access_key=...
```

Где вы ожидаете задать имя хоста

Имя хоста

Рис. 14.4 Пользовательское имя хоста S3 можно задать, но не там, где вы этого ожидали

Теперь, когда у нас настроены подключения, перенесем данные.

Листинг 14.8 Загрузка части данных в MinIO с помощью S3Hook

```
s3_hook = S3Hook(aws_conn_id="s3")
s3_hook.load_string(
    string_data=json.dumps(data),
    key=f"raw/citibike/{ts_nodash}.json",
    bucket_name="datalake"
)
```

Запись в объект с временной меткой задачи, шаблонированной в имени ключа

Если все прошло успешно, мы можем войти в интерфейс MinIO по адресу http://localhost: 9000 и просмотреть первый скачанный файл (рис. 14.5).

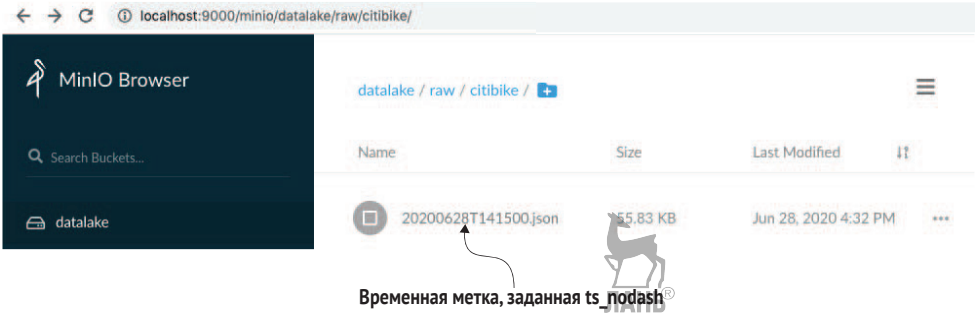


Рис. 14.5 Скриншот интерфейса MinIO, показывающий файл, записанный в каталог /data lake/raw/citibike, и имя файла, шаблонированное с использованием ds_nodash

Если бы вам приходилось чаще выполнять операцию «из HTTP-в-S3» с другими параметрами, то вы, вероятно, захотели бы написать оператор для этой задачи, чтобы избежать дублирования кода.

14.2.2 Загрузка данных по Yellow Cab

Нам также нужно скачать данные о такси в хранилище объектов MinIO. Это тоже операция вида «из HTTP-в-S3», но у нее есть ряд отличительных характеристик:

- файловый ресурс обслуживает файлы, тогда как для данных Citi Bike нам пришлось создавать новые файлы для MinIO;
- это файлы с расширением .cvs, а API Citi Bike возвращает данные в формате JSON;
- мы не знаем заранее имена файлов; чтобы получить список файлов, нужно поработать с индексом.

Когда вы сталкиваетесь с такими специфическими функциями, то обычно это приводит к необходимости реализовать собственное поведение, вместо того чтобы применить встроенный оператор Airflow. Некоторые операторы легко настроить, а некоторые – нет, но при таких ситуациях в основном приходится прибегать к реализации собственных функций. С учетом всего сказанного посмотрим на возможную реализацию.

Листинг 14.9 Скачивание данных из файлового ресурса Yellow Cab в хранилище MinIO

```
def _download_taxi_data():
    taxi_conn = BaseHook.get_connection(conn_id="taxi")
    s3_hook = S3Hook(aws_conn_id="s3")

    url = f"http://{taxi_conn.host}"
    response = requests.get(url)
    files = response.json()
    for filename in [f["name"] for f in files]:
        response = requests.get(f"{url}/{filename}")
        s3_key = f"raw/taxi/{filename}"
        ➥ s3_hook.load_string(string_data=response.text, key=s3_key,
      bucket_name="datalake")

download_taxi_data = PythonOperator(
    task_id="download_taxi_data",
    python_callable=_download_taxi_data,
    dag=dag,
)
```

Получаем список файлов

Получаем один-единственный файл

Загружаем файл в MinIO

Этот код скачает данные с файлового сервера и загрузит их в MinIO, но возникает проблема. Заметили ее?

`s3_hook.load_string()` не идемпотентная операция. Она не переопределяет файлы и загружает только один файл (или в данном случае строку), если его еще не существует. Если файл с таким именем уже есть, произойдет сбой:

```
➥ [2020-06-28 15:24:03,053] {taskinstance.py:1145} ERROR - The key
    raw/taxi/06-28-2020-14-30-00.csv already exists.
...
    raise ValueError("The key {key} already exists.".format(key=key))
ValueError: The key raw/taxi/06-28-2020-14-30-00.csv already exists.
```

Чтобы избежать сбоев в работе с существующими объектами, мы могли бы применить принцип EAFP (сначала перехватывайте исключения, вместо того чтобы проверять все возможные условия) и перехватывать исключения ValueError, возникающие, если файл уже существует.

Листинг 14.10 Скачиваем данные из файлового ресурса Yellow Cab в хранилище MinIO

```
def _download_taxi_data():
    taxi_conn = BaseHook.get_connection(conn_id="taxi")
    s3_hook = S3Hook(aws_conn_id="s3")

    url = f"http://{taxi_conn.host}"
    response = requests.get(url)
    files = response.json()

    for filename in [f["name"] for f in files]:
        response = requests.get(f"{url}/{filename}")
        s3_key = f"raw/taxi/{filename}"
        try:
            s3_hook.load_string(
                string_data=response.text,
                key=s3_key,
                bucket_name="datalake",
            )
            print(f"Uploaded {s3_key} to MinIO.")
        except ValueError:
            print(f"File {s3_key} already exists.")
```

Перехватываем исключения ValueError, возникающие, если файл уже существует

Добавление этой проверки для существующих файлов больше не приведет к сбою нашего конвейера! Теперь у нас есть две задачи, которые загружают данные в хранилище MinIO (рис. 14.6).

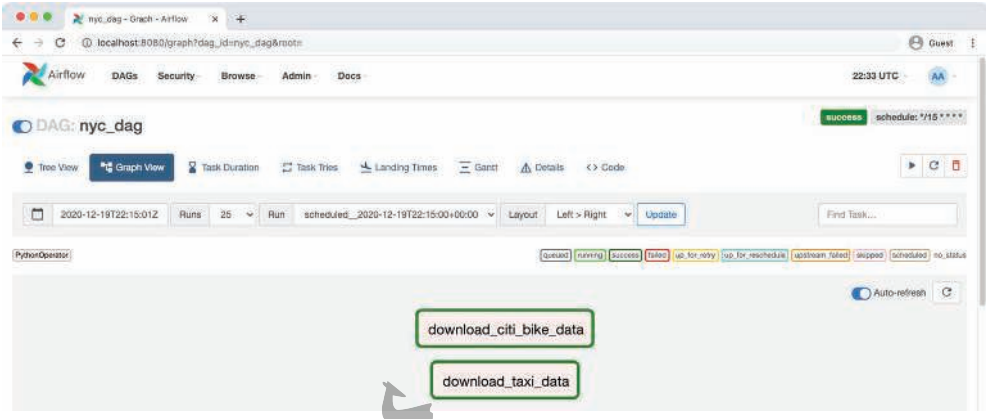


Рис. 14.6 Первые две задачи download_citi_bike_data и download_taxi_data

Данные для API Citi Bike и файлового ресурса Yellow Cab загружаются в хранилище MinIO (рис. 14.7).

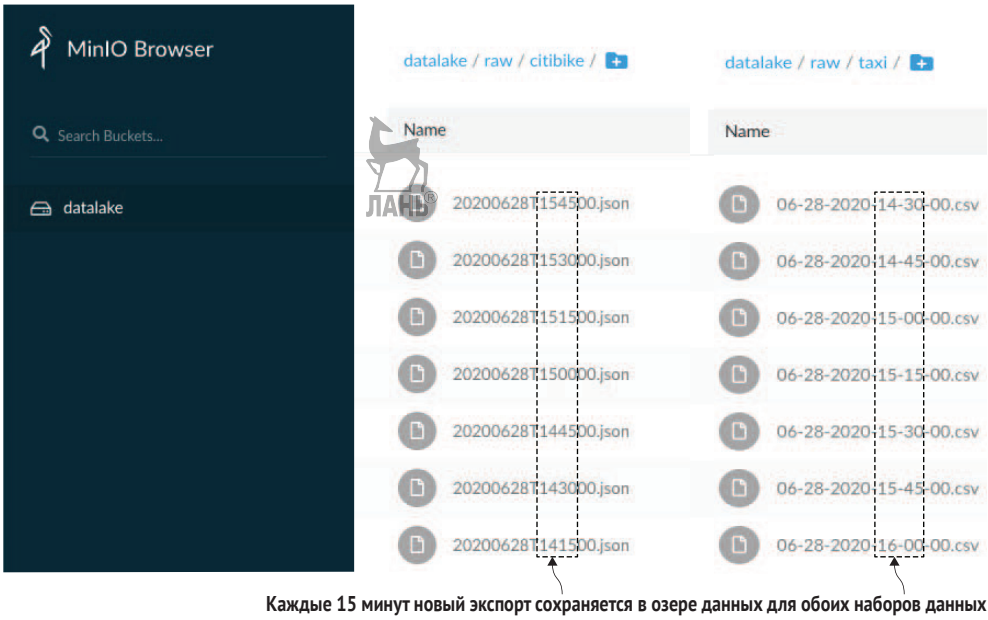


Рис. 14.7 Данные, экспортированные в хранилище MinIO. Мы установили собственный контроль над MinIO и всегда можем обратиться к этим файлам позже

14.3 Применение аналогичных преобразований к данным

После того как мы скачали данные по Citi Bike и Yellow Cab, мы применяем ряд преобразований, чтобы сопоставить координаты прокатных станций Citi Bike с зонами Yellow Cab и начать их точное сравнение. Сделать это можно разными способами, в зависимости от размера данных.

В сценарии с большими данными нужно применить Apache Spark для обработки данных с помощью кластера машин. Задание Spark можно запустить с помощью `SparkSubmitOperator` или другого оператора, который может инициировать задание Spark, например `SSHOperator`. Затем задание Spark будет читать данные из S3, применять к ним преобразования и записывать их обратно в S3.

Если речь идет о меньшем масштабе (т. е. о данных, обрабатываемых на одной машине), то можно применить для этой задачи Pandas, но на момент написания данной книги оператора `PandasOperator` не существует, поэтому код Pandas обычно выполняется с использованием `PythonOperator`. Обратите внимание, что код Python запускается на том же компьютере, где установлен Airflow, тогда как задание Spark обычно выполняется на других машинах, выделенных для этой задачи, что не влияет на ресурсы компьютера, где установлен Airflow.

В последнем случае Airflow отвечает только за запуск и мониторинг задания Spark. Если задание Pandas выходит за пределы ресурсов машины, теоретически это может вывести ее из строя, а вместе с ней и Airflow.

Еще один способ избежать проблем с ресурсами – передать задание Kubernetes с помощью `KubernetesPodOperator` или аналогичной платформе для оркестровки контейнеризированных приложений, такой как AWS ECS, используя оператор `ECSOperator`.

Предположим, что мы применяем Pandas для обработки небольших данных. Вместо демонстрации того, как использовать еще один `PythonOperator`, посмотрим, как обобщить некоторые компоненты для повторного использования и дедупликации кода. У нас есть два набора данных, которые хранятся в /raw:

- /raw/citibike/{ts_nodash}.json;
- /raw/taxi/*.csv.

Оба набора данных будут прочитаны с помощью Pandas, и будет применено несколько преобразований, в конечном итоге результат будет записан в:

- /processed/citibike/{ts_nodash}.parquet;
- /processed/taxi/{ts_nodash}.parquet.

Хотя входные форматы различаются, тип объекта, в который они загружены, и выходные форматы – нет. Абстракция, к которой применяются операции в Pandas, – это Pandas DataFrame (похожий на Spark DataFrame). Есть несколько небольших отличий между нашими преобразованиями, наборами входных данных и местоположениями выходных файлов, но основная абстракция та же: Pandas DataFrame. Следовательно, мы могли бы реализовать один оператор для работы с обоими преобразованиями.

Листинг 14.11 Единый оператор для всех операций Pandas DataFrame

```
import logging

from airflow.models import BaseOperator
from airflow.utils.decorators import apply_defaults
   class PandasOperator (BaseOperator):
      template_fields = (
          "_input_callable_kwargs",
          "_transform_callable_kwargs",
          "_output_callable_kwargs",
      )

      @apply_defaults
      def __init__(
         self,
         input_callable,
         output_callable,
```

Все аргументы kwargs могут содержать шаблонные значения

```
        transform_callable=None,
        input_callable_kwargs=None,
        transform_callable_kwargs=None,
        output_callable_kwargs=None,
        **kwargs,
    ):
        super().__init__(**kwargs)

        # Атрибуты для чтения данных
        self._input_callable = input_callable
        self._input_callable_kwargs = input_callable_kwargs or {}

        # Атрибуты для преобразований
        self._transform_callable = transform_callable
        self._transform_callable_kwargs = transform_callable_kwargs or {}

        # Атрибуты для записи данных
        self._output_callable = output_callable
        self._output_callable_kwargs = output_callable_kwargs or {}

    def execute(self, context):
        df = self._input_callable(**self._input_callable_kwargs)
        logging.info("Read DataFrame with shape: %s.", df.shape)

        if self._transform_callable:
            df = self._transform_callable(
                df,
                **self._transform_callable_kwargs,
            )
            logging.info("DataFrame shape after transform: %s.", df.shape)

        self._output_callable(df, **self._output_callable_kwargs)
```

Вызываем входной вызываемый объект для возврата Panda DataFrame

Применяем преобразования к DataFrame

Записываем DataFrame

Разберемся, как использовать оператор `PandasOperator`. Как уже упоминалось, между различными преобразованиями существует некий общий элемент – это Pandas DataFrame. Мы используем его для составления операций над DataFrame с учетом трех функций:

- `input_callable`;
- `transform_callable` (не является обязательной);
- `output_callable`.

Функция `input_callable` считывает данные в Pandas DataFrame, `transform_callable` применяет к нему преобразования, а `output_callable` записывает DataFrame. Пока ввод и вывод всех трех функций представляют собой Pandas DataFrame, мы можем смешивать и сопоставлять вызываемые объекты для обработки данных с помощью `PandasOperator`. Рассмотрим один пример.

Листинг 14.12 Применение оператора PandasOperator из листинга 14.11

```
process_taxi_data = PandasOperator (
    task_id="process_taxi_data",
```

```
    input_callable=get_minio_object,
    input_callable_kwargs={
        "pandas_read_callable": pd.read_csv,
        "bucket": "datalake",
        "paths": "{{ ti.xcom_pull(task_ids='download_taxi_data') }}",
    },
    transform_callable=transform_taxi_data,
    output_callable=write_minio_object,
    output_callable_kwargs={
        "bucket": "datalake",
        "path": "processed/taxi/{{ ts_nodash }}.parquet",
        "pandas_write_callable": pd.DataFrame.to_parquet,
        "pandas_write_callable_kwargs": {"engine": "auto"},
    },
    dag=dag,
)
```

Читаем CSV-файл из хранилища MinIO

Применяем преобразования к DataFrame

Пишем файл с расширением .parquet в хранилище MinIO

Цель `PandasOperator` – предоставить единый оператор, который позволяет смешивать и сопоставлять различные функции ввода, преобразования и вывода. В результате, определяя задачу Airflow, вы склеиваете эти функции, указывая на них и предоставляя их аргументы. Начнем с функции ввода, которая возвращает Pandas DataFrame, как показано ниже.

Листинг 14.13 Пример функции, считывающей объекты MinIO и возвращающей Pandas DataFrame

```
def get_minio_object(
    pandas_read_callable,
    bucket,
    paths,
    pandas_read_callable_kwargs=None,
):
    s3_conn = BaseHook.get_connection(conn_id="s3")
    minio_client = Minio(
        s3_conn.extra_dejson["host"].split("://")[1],
        access_key=s3_conn.extra_dejson["aws_access_key_id"],
        secret_key=s3_conn.extra_dejson["aws_secret_access_key"],
        secure=False,
    )

    if isinstance(paths, str):
        paths = [paths]
    if pandas_read_callable_kwargs is None:
        pandas_read_callable_kwargs = {}

    dfs = []
    for path in paths:
        minio_object = minio_client.get_object(
            bucket_name=bucket,
            object_name=path,
        )
```

Инициализируем клиента MinIO

```
        df = pandas_read_callable(                    ← Читаем файл из MinIO
            minio_object,
            **pandas_read_callable_kwargs,
        )
        dfs.append(df)
    return pd.concat(dfs)                             ← Возвращаем Pandas DataFrame
```

Функция преобразования, которая следует принципу «DataFrame вошел, DataFrame вышел», выглядит следующим образом:

Листинг 14.14 Пример функции преобразования данных по такси

```
def transform_taxi_data(df):                          ← DataFrame вошел
    ➥ df[["pickup_datetime", "dropoff_datetime"]] = df[["pickup_datetime",
      "dropoff_datetime"]].apply(
        pd.to_datetime
    )
    ➥ df["tripduration"] = (df["dropoff_datetime"] - df["pickup_datetime"])
      .dt.total_seconds().astype(int)
    df = df.rename(
        columns={
            "pickup_datetime": "starttime",
            "pickup_locationid": "start_location_id",
            "dropoff_datetime": "stoptime",
            "dropoff_locationid": "end_location_id",
        }
    ).drop(columns=["trip_distance"])
    return df                                         ← DataFrame вышел
```

И наконец, функция вывода, принимающая Pandas DataFrame, выглядит следующим образом:

Листинг 14.15 Пример функции записи преобразованного DataFrame обратно в хранилище MinIO

```
def write_minio_object(
    df,
    pandas_write_callable,
    bucket,
    path,
    pandas_write_callable_kwargs=None
):
    s3_conn = BaseHook.get_connection(conn_id="s3")
    minio_client = Minio(
        s3_conn.extra_dejson["host"].split("://")[1],
        access_key=s3_conn.extra_dejson["aws_access_key_id"],
        secret_key=s3_conn.extra_dejson["aws_secret_access_key"],
        secure=False,
    )
    bytes_buffer = io.BytesIO()
    pandas_write_method = getattr(df, pandas_write_callable.__name__)
```

Извлекаем ссылку на метод записи DataFrame (например, pd.DataFrame.to_parquet)

```
pandas_write_method(bytes_buffer, **pandas_write_callable_kwargs)
nbytes = bytes_buffer.tell()
bytes_buffer.seek(0)
minio_client.put_object(
    bucket_name=bucket,
    object_name=path,
    length=nbytes,
    data=bytes_buffer,
)
```

Вызываем метод записи DataFrame для записи DataFrame в байтовый буфер, который можно хранить в MinIO

Сохраняем байтовый буфер в MinIO

Передача Pandas DataFrame между функциями ввода, преобразования и вывода теперь предоставляет возможность изменить входной формат набора данных, просто изменив аргумент `"pandas_read_callable": pd.read_csv`, например, на `"pandas_read_callable ": pd.read_parquet`. В результате нам не нужно повторно реализовывать логику с каждым изменением или каждым новым набором данных, что исключает дублирование кода и повышает гибкость.

ПРИМЕЧАНИЕ Всякий раз, когда вы обнаруживаете, что повторяете логику и хотите разработать единый фрагмент логики, охватывающий несколько случаев, подумайте о чем-то общем, что есть у ваших операций, например о Pandas DataFrame или файловом объекте Python.

14.4 Структурирование конвейера обработки данных

Как было упомянуто в предыдущем разделе, мы создали папки «Raw» и «Processed» в бакете «datalake». Откуда они к нам попали и почему? С точки зрения эффективности, мы могли бы, в принципе, написать одну-единственную функцию Python, которая извлекает данные, преобразовывает их и записывает результаты в базу данных, сохраняя при этом данные в памяти и не касаясь файловой системы. Так было бы намного быстрее. Почему же мы этого не сделали?

Во-первых, данные часто используются несколькими лицами или конвейерами обработки данных. Для распространения и повторного использования они хранятся в месте, где другие люди и процессы могут читать данные.

Но что еще более важно, нам нужно сделать наш конвейер воспроизводимым. Что означает воспроизводимость с точки зрения конвейера обработки данных? Данные никогда не бывают идеальными, а программное обеспечение всегда находится в стадии разработки; это означает, что нам нужна возможность вернуться к предыдущим запускам ОАГ и повторно запустить конвейер с данными, которые были обработаны. Если мы извлекаем данные из веб-сервиса, например REST API, который возвращает результат только для состояния

в данный момент времени, мы не можем вернуться к API и запросить тот же результат, что и два месяца назад. В такой ситуации лучше сохранить неотредактированную копию результата. По соображениям конфиденциальности некоторые части данных иногда редактируются, что неизбежно, но отправной точкой воспроизводимого конвейера обработки данных должно быть сохранение копии входных данных (которые редактируются как можно реже). Эти данные обычно хранятся в папке raw (рис. 14.8).

Рис. 14.8 Мы не можем контролировать структуру данных во внешних системах. В собственных системах логично хранить данные в соответствии с их жизненным циклом. Например, неотредактированные данные хранятся в папке Raw, производные и преобразованные данные – в папке Processed, а наборы данных, готовые к передаче, – в папке Export

Используя данные из папки raw, вы (и другие пользователи) можете изменять, обогащать, уточнять, преобразовывать и смешивать их столько, сколько захотите, после чего они записываются обратно в папку processed.

Преобразования часто требуют больших затрат времени и вычислений, поэтому мы стараемся избегать повторного запуска задачи и сохранять результаты, чтобы обработанные результаты можно было легко прочитать снова.

На практике многие организации применяют более детальное разделение между этапами данных, например Raw > Preprocessed > Enriched > Processed > Export. Но не бывает так, чтобы одна структура подходила всем; ваш проект и его требования определят, как лучше всего структурировать передвижение данных.

14.5 Разработка идемпотентных конвейеров обработки данных

Теперь, когда у нас есть данные в папке raw, мы обработаем их и вставим результаты в базу данных Postgres. Поскольку эта глава не посвящена лучшему способу обработки данных с помощью Pandas или Spark, мы не будем обсуждать подробности этого преобразования. Вместо этого повторим важный аспект конвейеров обработки данных в целом, а именно обеспечение возможности выполнения конвейера обработки данных несколько раз без необходимости вручную сбрасывать состояние или вносить изменения в результаты (идемпотентность).

В этом конвейере обработки данных есть две точки, куда можно было бы ввести идемпотентность. Первый этап прост: при преобразовании необработанных данных в обработанное состояние и сохранении их в папке /processed нужно задать параметр для перезаписи файлов назначения. Это гарантирует, что повторный запуск задачи не завершится ошибкой из-за уже существующего пути вывода.

Второй этап, когда мы записываем результаты в базу данных, менее очевиден. Повторный запуск задачи записи результатов в базу данных может и не завершиться ошибкой, но может привести к появлению повторяющихся строк, что может испортить результаты. Как гарантировать, что результаты записываются в базу данных идемпотентным образом, чтобы мы могли повторно запускать конвейеры без дублирования результатов?

Один из способов – добавить в таблицу столбец, который может идентифицировать нечто уникальное касательно задания, например дату выполнения задания Airflow. Допустим, мы используем Pandas для записи DataFrame в базу данных, как показано в следующем листинге.

Листинг 14.16 Запись Pandas DataFrame в базу данных SQL

```
--CREATE TABLE citi_bike_rides(
--    tripduration INTEGER,
--    starttime TIMESTAMP,
--    start_location_id INTEGER,
--    stoptime TIMESTAMP,
--    end_location_id INTEGER
--);

df = pd.read_csv(... citi bike data ...)
engine = sqlalchemy.create_engine(
    BaseHook.get_connection(self._postgres_conn_id).get_uri()
)
df.to_sql("citi_bike_rides", con=engine, index=False, if_exists="append")
```

Pandas DataFrame и структура таблицы должны совпадать

Невозможно определить при выполнении `df.to_sql()`, собираемся ли мы уже вставить существующие строки в таблицу. В этой ситуации можно было бы изменить таблицу базы данных, чтобы добавить столбец для даты выполнения Airflow.

Листинг 14.17 Запись Pandas DataFrame в базу данных SQL за одну операцию

```
--CREATE TABLE citi_bike_rides(
--    tripduration INTEGER,
--    starttime TIMESTAMP,
--    start_location_id INTEGER,
--    stoptime TIMESTAMP,
--    end_location_id INTEGER,
```

```
--    airflow_execution_date TIMESTAMP
--);

df = pd.read_csv(... citi bike data ...)
df["airflow_execution_date"] = pd.Timestamp(
   context["execution_date"].timestamp(),
   unit='s',
)
engine = sqlalchemy.create_engine(
   BaseHook.get_connection(self._postgres_conn_id).get_uri()
)
with engine.begin() as conn:
   conn.execute(
       "DELETE FROM citi_bike_rides"
       f"WHERE airflow_execution_date='{context['execution_date']}';"
   )
   df.to_sql("citi_bike_rides", con=conn, index=False, if_exists="append")
```

Добавляем execute_date в качестве столбца в Pandas Dataframe

Начинаем транзакцию

Сначала удаляем все существующие записи с текущей датой выполнения

В этом примере мы запускаем транзакцию базы данных, потому что взаимодействие с базой данных состоит из двух частей: сначала мы удаляем все существующие строки с заданной датой выполнения, а затем вставляем новые строки. Если существующих строк с заданной датой выполнения нет, то ничего не удаляется. Два оператора SQL (`df.to_sql()` выполняет SQL под капотом) заключены в транзакцию, которая является атомарной операцией, то есть либо оба запроса завершаются успешно, либо ни один из них не выполняется. Это гарантирует отсутствие остатков в случае сбоя. После того как данные будут обработаны и успешно сохранены в базе данных, мы можем запустить веб-приложение по адресу http://localhost: 8083, которое запрашивает результаты в базе данных (рис. 14.9).

Start location	End location	Weekday	Time group	Avg time Citi Bike	Avg time Taxi
Alphabet City	East Village	Sunday	8 AM - 11 AM	1057.2	330.0
Alphabet City	Penn Station/Madison Sq West	Sunday	8 AM - 11 AM	1023.0	1318.0
Astoria	Long Island City/Hunters Point	Sunday	8 AM - 11 AM	700.0	358.0
Astoria	Old Astoria	Sunday	10 PM - 8 AM	206.0	1757.0
Astoria	Steinway	Sunday	8 AM - 11 AM	725.0	705.0
Battery Park City	Clinton East	Sunday	8 AM - 11 AM	1551.0	1788.0
Battery Park City	East Chelsea	Saturday	4 PM - 7 PM	715.0	913.0
Battery Park City	Financial District North	Sunday	8 AM - 11 AM	388.5	415.75

Рис. 14.9 Веб-приложение, отображающее результаты, хранящиеся в базе данных PostgreSQL, постоянно обновляемые ОАГ

Результаты показывают, какой способ передвижения между двумя районами в данный момент является более быстрым. Например (строка 1), в воскресенье с 8:00 до 11:00 ехать из Алфабет-Сити в Ист-Виллидж (в среднем) быстрее на такси: 330 секунд (5,5 минуты) на такси по сравнению с 1057,2 секунды (17,62 минуты) на велосипеде Citi Bike.

Теперь Airflow приступает к скачиванию, преобразованию и сохранению данных в базе данных Postgres с 15-минутными интервалами. В случае с реальным приложением, ориентированным на пользователя, вам, вероятно, понадобится более привлекательный интерфейс с большей доступностью для поиска, но с точки зрения серверной части у нас теперь есть автоматизированный конвейер обработки данных, который автоматически запускается с 15-минутными интервалами и показывает, что быстрее: такси или велосипед при передвижении из одного района в другой в заданное время, что показано в таблице на рис. 14.9.

Резюме

- Разработка идемпотентных задач может быть разной в зависимости от случая.
- Сохранение промежуточных данных гарантирует, что мы можем возобновить (частичные) конвейеры.
- Когда функциональные возможности оператора не выполняются, вы должны постоянно вызывать функцию с помощью `PythonOperator` или реализовать собственный оператор.

Часть IV

Облако

На этом этапе вы должны быть на правильном пути к освоению Airflow – вы можете писать сложные конвейеры и знаете, как развернуть Airflow в рабочих условиях.

До сих пор мы рассматривали запуск Airflow в локальной системе либо изначально, либо с использованием контейнерных технологий, таких как Docker. Распространенный вопрос – как запустить и использовать Airflow в облаке, поскольку многие современные технологические ландшафты включают в себя облачные платформы. Эта часть полностью посвящена запуску Airflow в облаке, включая такие темы, как проектирование архитектур для развертываний Airflow и использование встроенных функций Airflow для вызова различных облачных сервисов.

Вначале в главе 15 мы даем краткое введение в различные компоненты, участвующие в проектировании развертывания Airflow на базе облака. Мы также кратко обсудим встроенные функции Airflow для взаимодействия с различными облачными сервисами и коснемся развертываний Airflow, управляемых поставщиками, которые могут избавить вас от реализации собственного развертывания в облаке.

После этого введения мы рассмотрим конкретные реализации Airflow для нескольких облачных платформ: Amazon AWS (глава 16), Microsoft Azure (глава 17) и Google Cloud Platform (глава 18). В каждой из этих глав мы спроектируем архитектуры для развертывания Airflow с использованием сервисов соответствующей платформы и обсудим встроенные функции Airflow для взаимодействия с сервисами конкретной платформы. Каждая глава завершается примером.

После завершения части IV вы должны иметь четкое представление о том, как спроектировать развертывание Airflow для интересующей вас облачной платформы. Вы также должны уметь создавать конвейеры, которые плавно интегрируются с облачными сервисами, чтобы использовать масштаб облака в ваших рабочих процессах.

15 Airflow и облако

Эта глава рассказывает:

- о компонентах, необходимых для создания развертываний Airflow в облаке;
- об облачных хуки и операторах, использующихся для интеграции с облачными сервисами;
- о сервисах, управляемых поставщиком, как альтернативе реализации собственного развертывания.

В этой главе мы изучим, как развернуть и интегрировать Airflow в облачном окружении. Вначале мы еще раз рассмотрим различные компоненты Airflow и то, как они сочетаются друг с другом при развертывании в облаке. Мы будем использовать этот обзор, чтобы сопоставить каждый из компонентов с их облачными аналогами в Amazon AWS (глава 16), Microsoft Azure (глава 17) и Google Cloud Platform (глава 18). Затем кратко расскажем об облачных хуках и операторах, которые можно использовать для интеграции с конкретными облачными сервисами, а также предоставим несколько управляемых альтернатив для развертывания Airflow и обсудим ряд критериев, которые следует учитывать при сравнении реализации собственного развертывания с использованием решения, управляемого поставщиком.

15.1 Проектирование стратегий (облачного) развертывания

Прежде чем приступить к проектированию стратегий развертывания Airflow в разных облачных сервисах (AWS, Azure и GCP), начнем с обзора компонентов Airflow (например, веб-сервера, планировщика, воркеров) и того, к каким (общим) ресурсам этим компонентам потребуется доступ (например, ОАГ, хранилище журналов и т. д.). Это поможет нам позже при сопоставлении данных компонентов с соответствующими облачными сервисами.

Чтобы было проще, мы начнем с развертывания Airflow на основе `LocalExecutor`. В этом типе настройки воркеры Airflow работают на том же компьютере, что и планировщик. Это означает, что нам нужно настроить только два вычислительных ресурса для Airflow: один для веб-сервера и один для планировщика (рис. 15.1).

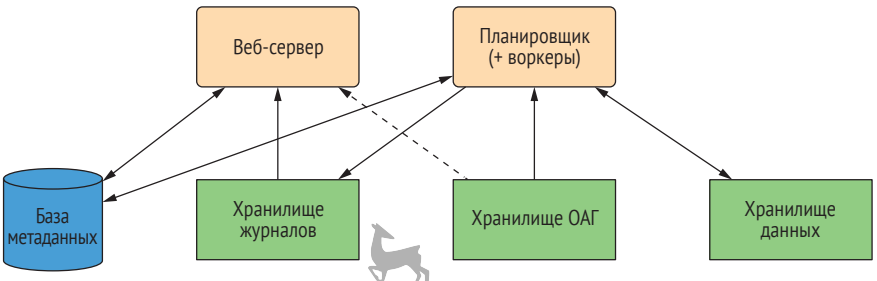


Рис. 15.1 Обзор различных вычислительных компонентов и компонентов хранения, задействованных в развертывании Airflow на основе `LocalExecutor`

Веб-серверу и планировщику потребуется доступ к общей базе данных (базе метаданных Airflow) и (в зависимости от версии и конфигурации Airflow[1]) общее хранилище для ОАГ и журналов. В зависимости от того, как вы управляете своими данными, вам также может потребоваться настройка внешнего хранилища для хранения наборов входных и выходных данных.

Помимо этих вычислительных ресурсов и ресурсов хранения, также необходимо подумать о работе в сети. Здесь у нас есть две основные проблемы: как мы будем соединять различные сервисы и как

[1] В Airflow 1 и веб-серверу Airflow, и планировщику по умолчанию требуется доступ к хранилищу ОАГ. В Airflow версии 1.10.10 была добавлена опция, чтобы веб-сервер хранил ОАГ в базе метаданных, поэтому ему больше не требуется доступ к хранилищу ОАГ, если эта опция активирована. В Airflow 2 эта опция всегда активирована, поэтому веб-серверу никогда не требуется доступ к хранилищу ОАГ.

организуем настройку сети для защиты внутренних сервисов. Как мы увидим далее, обычно это включает в себя настройку разных сегментов сети (общедоступных и закрытых подсетей) и подключение различных сервисов к соответствующим подсетям (рис. 15.2). Кроме того, полная установка должна также включать в себя защиту всех общедоступных сервисов от несанкционированного доступа.

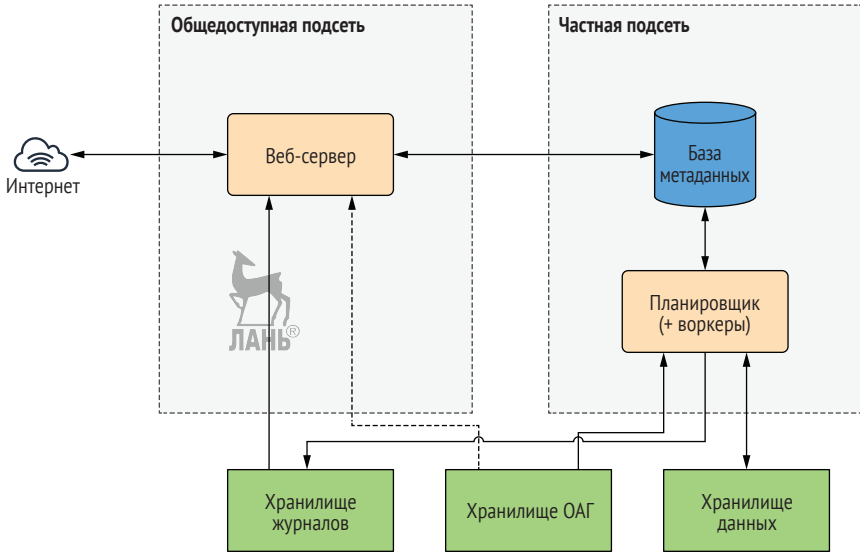


Рис. 15.2 Обзор работы в сети при развертывании на основе LocalExecutor. Компоненты разделены на две общедоступные и закрытые подсети. В общедоступную подсеть следует размещать только общедоступные сервисы. Обратите внимание, что сервисы хранения создаются за пределами обеих подсетей, поскольку многие облачные сервисы хранения (например, AWS S3) не обязательно привязаны к данной подсети. Тем не менее эти учетные записи должны быть защищены от открытого доступа

Это дает нам довольно полный обзор необходимых компонентов для развертывания на основе `LocalExecutor`.

Переход к `CeleryExecutor` (который обеспечивает лучшее масштабирование за счет запуска воркеров на отдельных машинах) требует немного больше усилий, поскольку для развертывания на базе Celery требуются два дополнительных ресурса: пул дополнительных вычислительных ресурсов для воркеров и брокер сообщений, который передает им сообщения (рис. 15.3).

Надеемся, что эти наброски дадут вам представление о ресурсах, необходимых для реализации развертываний Airflow в облачной среде. В следующих главах мы рассмотрим реализацию этих архитектур в различных облачных сервисах.

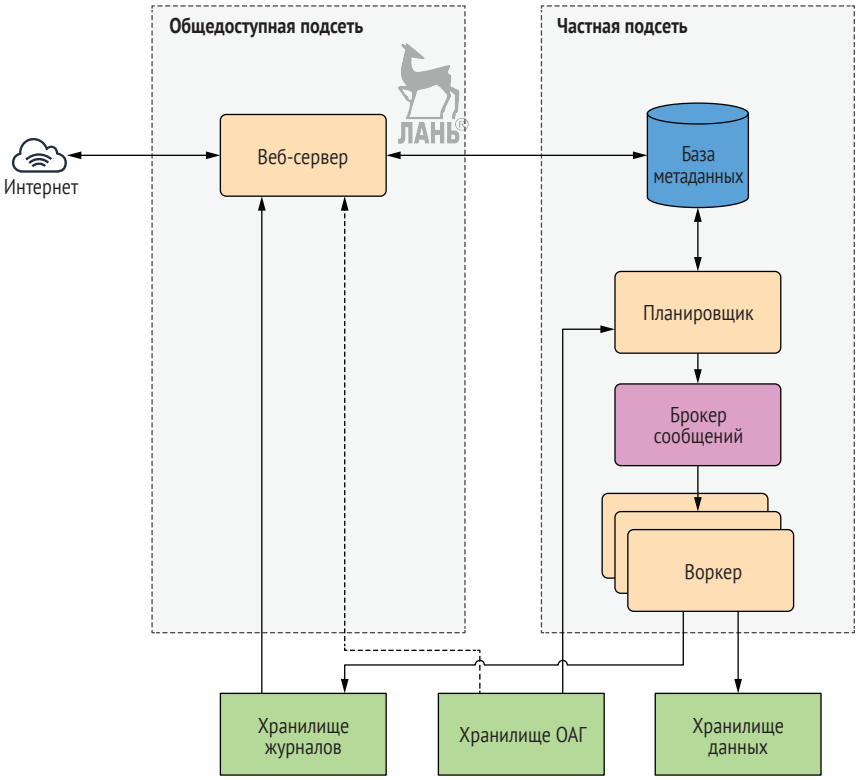


Рис. 15.3 Обзор архитектуры для развертывания Airflow на базе `CeleryExecutor`. Основные дополнения включают в себя дополнительный пул вычислительных компонентов для воркеров Airflow и брокер сообщений для ретрансляции задач. Обратите внимание, что для настройки на основе Celery больше не требуется, чтобы планировщик имел доступ к хранилищам данных и журналов, поскольку вычислительные ресурсы воркера будут отвечать за фактическое выполнение работы (и, следовательно, фактически будут читать и записывать данные и генерировать сообщения журналов)

15.2 Операторы и хуки, предназначенные для облака

За прошедшие годы программисты, участвующие в разработке Airflow, создали большое количество операторов и хуков, которые позволяют выполнять задачи с использованием различных облачных сервисов. Например, `S3Hook` дает возможность взаимодействовать с сервисом хранения AWS S3 (например, для загрузки и скачивания файлов), а `BigQueryExecuteQueryOperator` позволяет выполнять запросы в сервисе Google, BigQuery.

В Airflow 2 эти хуки и операторы можно использовать, установив соответствующие пакеты поставщика. В более ранних версиях Airflow

можно использовать те же функции, установив эквивалентные бэкпорты из PyPI.

15.3 Управляемые сервисы

Хотя реализация собственного развертывания Airflow может дать вам максимальную гибкость относительно того, как вы его используете, его настройка и сопровождение могут потребовать много работы. Один из способов избежать этого бремени – использовать сервис, управляемый поставщиком, где вы можете переложить бóльшую часть работы на внешнего поставщика. Этот поставщик обычно предоставляет инструменты для простого создания и управления новыми развертываниями Airflow без лишних хлопот, связанных с собственными развертываниями. Обычно поставщик также обещает предоставить сопровождение базовой инфраструктуры, чтобы вам не пришлось беспокоиться о том, чтобы ваша операционная система и/или установка Airflow обновлялись с помощью последних исправлений безопасности, о мониторинге системы и т. д.

Есть три известных управляемых сервиса для Airflow – это Astronomer.io, Google Cloud Composer и Amazon MWAA. В следующих разделах мы вкратце рассмотрим их и опишем их ключевые функции.

15.3.1 Astronomer.io

Astronomer.io – это решение на базе Kubernetes для Airflow, которое можно использовать как *SaaS*-решение (англ. software as a service – программное обеспечение как услуга) (облако Astronomer) или развернуть в собственном кластере Kubernetes (Astronomer Enterprise). По сравнению с Airflow, Astronomer также предоставляет дополнительные инструменты, которые помогут с легкостью развернуть экземпляры Airflow из пользовательского интерфейса или из их настраиваемого интерфейса командной строки. Интерфейс командной строки также позволяет запускать локальные экземпляры Airflow для разработки, что может упростить разработку ОАГ (при условии что Kubernetes доступен на вашем компьютере, используемом для разработки).

Будучи созданным на базе Kubernetes, Astronomer.io должен хорошо интегрироваться с любыми рабочими процессами на основе Kubernetes и Docker, к которым вы, возможно, привыкли. Это упрощает (например) выполнение задач в контейнерах с помощью `KubernetesExecutor` и `KubernetesPodOperator`. Также поддерживаются другие режимы развертывания с использованием `LocalExecutor` или `CeleryExecutor`, что обеспечивает значительную гибкость при выполнении заданий. Astronomer также позволяет настроить развертывание Airflow, указав дополнительную ОС или зависимости Python, которые

необходимо установить в кластер. В качестве альтернативы вы можете создать собственный базовый образ Airflow, если вам потребуется дополнительная гибкость.

Стоимость решения SaaS рассчитывается с использованием *единиц Astronomer*, при этом разные конфигурации требуют разного количества единиц. Обзор этих затрат *см.* на сайте Astronomer (https://www.astronomer.io).

Также стоит упомянуть, что Astronomer.io работает с некоторыми ключевыми участниками проекта Airflow. Они вносят большой вклад в проект Airflow и регулярно стимулируют разработку важных улучшений в версии Airflow с открытым исходным кодом, чтобы каждый мог воспользоваться этими новыми функциями. Их диаграммы Helm для развертывания Airflow в Kubernetes также находятся в свободном доступе в интернете, если вы захотите попробовать их за пределами платформы Astronomer.

15.3.2 Google Cloud Composer

Google Cloud Composer – это управляемая версия Airflow, работающая поверх набора облачных сервисов Google Cloud Platform (GCP). Таким образом, Cloud Composer предоставляет простое решение для развертывания Airflow в GCP, которое хорошо интегрируется с его сервисами. GCP также позаботится об управлении базовыми ресурсами, но вы платите только за те ресурсы, которые они используют. Вы можете взаимодействовать с Cloud Composer с помощью интерфейса командной строки GCP и/или отслеживать состояние своих кластеров из веб-интерфейса GCP.

Подобно Astronomer.io, Cloud Composer также основан на Kubernetes и работает на Google Kubernetes Engine (GKE). Приятной особенностью Cloud Composer является то, что он хорошо интегрируется с различными сервисами в GCP (такими как Google Cloud Storage, BigQuery и т. д.), что упрощает доступ к ним из ваших ОАГ. Cloud Composer также обеспечивает большую гибкость относительно того, как вы настраиваете кластер Kubernetes с точки зрения ресурсов и т. д., чтобы вы могли настроить развертывание в соответствии с вашими конкретными потребностями. Подобно Astronomer.io, вы можете установить зависимости Python в свои кластеры Airflow с помощью веб-интерфейса или интерфейса командной строки GCP.

Google Cloud Composer включает плату за само окружение (количество узлов, хранение в базе данных, трафик, исходящий из сети, и т. д.), помимо затрат на базовые сервисы (GKE, Google Cloud Storage[1]). Актуальный обзор этих затрат можно найти на сайте GCP (https://cloud.google.com).

Будучи ярым сторонником программного обеспечения с открытым исходным кодом, Google также регулярно вносит свой вклад в проект

[1] Используется Cloud Composer для хранения ОАГ и журналов и т. д.

с открытым исходным кодом Airflow и помог разработать обширный набор операторов для различных сервисов, чтобы обеспечить их использование в рамках Airflow[1].

15.3.3 Amazon Managed Workflows for Apache Airflow

Amazon Managed Workflows for Apache Airflow (MWAA) – это сервис AWS, позволяющий с легкостью создавать управляемые развертывания Airflow в облаке AWS, аналогично Cloud Composer от Google. При использовании MWAA для запуска Airflow сервис будет управлять базовой инфраструктурой и масштабировать развертывание в соответствии с требованиями ваших рабочих процессов. Кроме того, предполагается, что развертывание Airflow в MWAA будет хорошо интегрироваться с такими сервисами AWS, как S3, RedShift, Sagemaker, а также с AWS CloudWatch для журналирования и оповещений и AWS IAM для безопасного управления доступом к сервисам и ресурсам AWS.

Подобно другим управляемым решениям, MWAA использует `CeleryExecutor` для масштабирования воркеров на базе текущей рабочей нагрузки с базовой инфраструктурой, управляемой за вас. Вы можете добавлять или редактировать ОАГ, загружая их в заранее определенный бакет S3, где они будут развернуты в вашем окружении Airflow. Аналогичные подходы на основе S3 можно использовать для установки дополнительных плагинов Airflow или требований Python в кластер по мере необходимости.

Цена включает базовую плату за само окружение Airflow и дополнительную плату за каждый экземпляр воркера Airflow. В обоих случаях у вас есть возможность выбирать небольшие, средние или большие машины, чтобы адаптировать развертывание к вашему конкретному варианту использования. Динамическое масштабирование воркеров означает, что их использование должно быть относительно рентабельным. Существует также дополнительная (ежемесячная) стоимость хранения для базы метаданных Airflow, а также любого хранилища, необходимого для ваших ОАГ или данных. Посетите сайт AWS, чтобы получить последний обзор и дополнительные сведения (https://aws.amazon.com/ru/).

15.4 Выбор стратегии развертывания

При выборе платформы для запуска рабочих нагрузок Airflow мы рекомендуем подробно изучить особенности различных предложений (и их цены), чтобы определить, какой сервис лучше всего подходит

[1] Обратите внимание, что не обязательно применять Google Composer для использования этих операторов, поскольку они отлично работают и в Airflow (при правильной настройке полномочий).

для вашей ситуации. В целом реализация собственного развертывания в одном из облачных сервисов дает вам максимальную гибкость в выборе компонентов для запуска Airflow и способов их интеграции в любое существующее облачное или локальное решение, которое у вас уже есть. С другой стороны, реализация собственного облачного развертывания требует значительных усилий и опыта, особенно если вы хотите внимательно следить за такими важными факторами, как безопасность и управление затратами.

Использование управляемого решения позволяет возложить многие из этих обязанностей на поставщика, что позволяет вам сосредоточиться на фактическом создании ОАГ, а не на создании и обслуживании необходимой инфраструктуры. Однако управляемые решения не всегда могут быть достаточно гибкими для ваших нужд, если у вас сложные требования. Например, вот на что нужно обратить внимание:

- вы хотите использовать рабочий процесс на базе Kubernetes? Если да, то Astronomer.io или Google Cloud Platform обеспечивают простой подход. В качестве альтернативы вы можете развернуть собственный кластер Kubernetes, например используя диаграмму Helm из Astronomer.io;
- к каким сервисам вы хотите подключаться из ОАГ? Если вы много инвестируете в технологии GCP, использование Google Cloud Composer может оказаться бесполезным из-за простой интеграции между Composer и другими сервисами GCP. Однако если вы хотите подключаться к локальным сервисам или сервисам в других облаках, запуск Airflow в GCP, возможно, и не имеет особого смысла;
- как вы хотите развертывать свои ОАГ? И Astronomer.io, и Google Cloud Composer предоставляют простой способ развертывания ОАГ с помощью интерфейса командной строки (Astronomer.io) или бакета (Cloud Composer). Однако можно подумать над тем, как связать эти функции с конвейерами непрерывной интеграции и доставки для автоматического развертывания новых версий ОАГ и т. д.;
- сколько вы хотите потратить на развертывание Airflow? Развертывание на основе Kubernetes может быть дорогостоящим делом из-за стоимости базового кластера. Другие стратегии развертывания (с использованием других вычислительных решений в облаке) или решения SaaS (например, Astronomer.io) могут быть более дешевыми вариантами. Если у вас уже есть кластер Kubernetes, вы также можете рассмотреть возможность запуска Airflow в собственной инфраструктуре Kubernetes;
- вам нужен более точный контроль или гибкость, по сравнению с теми, что обеспечиваются управляемыми сервисами? В этом случае вы, возможно, захотите применить собственную стратегию развертывания (конечно же, за счет дополнительных усилий по настройке и сопровождению развертывания).

Как уже видно из этого краткого списка, при выборе решения для развертывания кластера Airflow необходимо учитывать множество факторов. Хотя мы не можем принять это решение за вас, мы надеемся, что вы учтете данные рекомендации.

Резюме

- Airflow состоит из нескольких компонентов (веб-сервер, планировщик, база метаданных, хранилище), которые необходимо реализовать с использованием облачных сервисов при развертывании в облаке.
- При развертывании Airflow с разными исполнителями (например, `LocalExecutor` или `CeleryExecutor`) требуются разные компоненты, которые необходимо учитывать при стратегии развертывания.
- Для интеграции с облачными сервисами Airflow предоставляет облачные хуки и операторов, которые позволяют вам взаимодействовать с соответствующим сервисом.
- Сервисы, управляемые поставщиком (например, Astronomer.io, Google Cloud Composer, Amazon MWAA), предоставляют простую альтернативу реализации собственного развертывания, управляя множеством деталей за вас.
- Что выбрать: сервис, управляемый поставщиком, или создать собственное облачное развертывание? Это будет зависеть от многих факторов, поскольку управляемые решения обеспечивают бóльшую простоту развертывания и управления за счет меньшей гибкости и (возможно) более высоких эксплуатационных расходов.

16 Airflow и AWS

Эта глава рассказывает о:

- проектировании стратегии развертывания AWS с использованием ECS, S3, EFS и RDS;
- хуках и операторах, предназначенных для AWS, и о том, как их использовать.

После краткого введения в предыдущей главе в этой главе мы подробнее рассмотрим, как развернуть и интегрировать Airflow с облачными сервисами в Amazon AWS. Мы начнем с проектирования развертывания Airflow, сопоставив различные компоненты Airflow с сервисами AWS. Затем мы рассмотрим хуки и операторы, которые предоставляет Airflow для интеграции с ключевыми сервисами AWS. Наконец, мы покажем, как использовать эти операторы и хуки для создания рекомендаций по фильмам.

16.1 Развертывание Airflow в AWS

В предыдущей главе мы описали различные компоненты, из которых состоит развертывание Airflow. В этом разделе мы спроектируем

несколько шаблонов развертывания для AWS, сопоставив их с конкретными облачными сервисами AWS. Это должно дать вам внятное представление о процессе проектирования развертывания Airflow для AWS и послужить хорошей отправной точкой для его реализации.

16.1.1 Выбор облачных сервисов

Если начать с веб-сервера и планировщика Airflow, то одним из самых простых подходов к запуску этих компонентов, вероятно, является Fargate, ядро для бессерверных вычислений на базе контейнеров. Одно из главных преимуществ Fargate (по сравнению с другими сервисами AWS, такими как ECS[1] или EKS[2]), заключается в том, что они позволяют с легкостью запускать контейнеры в AWS, не беспокоясь о выделении базовых вычислительных ресурсов и управлении ими. Это означает, что мы можем просто предоставить Fargate определение контейнерных задач веб-сервера и планировщика, а Fargate позаботится о развертывании, запуске и мониторинге задач за нас.

Если речь идет о базе метаданных Airflow, то рекомендуем обратить внимание на RDS-решения (например, Amazon RDS[3]), которые помогают настраивать реляционные базы данных в облаке, беря на себя такие трудоемкие административные задачи, как выделение оборудования, настройка базы данных, установка исправлений и резервные копии. Amazon RDS работает с такими базами данных, как MySQL, Postgres и Amazon Aurora (совместимая с MySQL и PostgreSQL реляционная база данных, созданная для облака). Как правило, Airflow поддерживает использование всех их для своей базы метаданных, поэтому ваш выбор может зависеть от других требований, таких как стоимость, или таких функций, как высокая доступность.

AWS предоставляет несколько вариантов общего хранилища. Самый известный – S3, масштабируемая система хранения объектов. S3 обычно отлично подходит для хранения больших объемов данных с высокой долговечностью и доступностью при относительно невысокой стоимости. Таким образом, это идеальный вариант для хранения больших наборов данных (которые мы можем обрабатывать в наших OAГ) или временных файлов, например журналов воркеров Airflow (которые Airflow может изначально записывать в S3). Недостаток S3 заключается в том, что его нельзя смонтировать как локальную файловую систему на машинах с веб-сервером или планировщиком, что делает его менее идеальным выбором для хранения таких файлов, как OAГ, к которым Airflow требуется локальный доступ.

---

[1] Elastic Compute Service похож на Fargate, но требует, чтобы вы самостоятельно управляли базовыми машинами.

[2] Elastic Kubernetes Service, управляемое решение AWS для развертывания и запуска Kubernetes.

[3] Amazon RDS предлагает на выбор различные базы данных, такие как PostgreSQL, MySQL и Aurora.

Напротив, облачное хранилище AWS EFS совместимо с протоколом NFS, и поэтому его можно установить непосредственно в контейнеры Airflow, что делает его пригодным для хранения ОАГ. Однако EFS немного дороже S3, что делает его менее идеальным вариантом для хранения данных или файлов журналов. Еще один недостаток EFS состоит в том, что в EFS файлы загружать сложнее, чем в S3, поскольку AWS не предоставляет простой веб-интерфейс или интерфейс командной строки для копирования файлов в EFS. По этим причинам все же, возможно, имеет смысл поискать другие варианты, такие как S3 (или Git) для хранения ОАГ, а затем использовать автоматизированный процесс для синхронизации ОАГ с EFS (как будет показано позже в этой главе).

Все это дает нам следующую картину (рис. 16.1):

- Fargate для вычислительных компонентов (веб-сервер и планировщик Airflow);
- Amazon RDS (например, Aurora) для базы метаданных Airflow;
- S3 для хранения журналов (а также, необязательно, данных);
- EFS для хранения ОАГ.

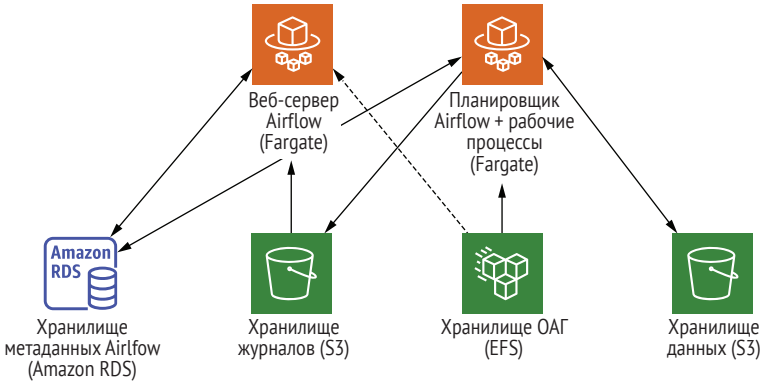


Рис. 16.1 Сопоставление компонентов Airflow из рис. 15.1 с сервисами AWS. Fargate используется для вычислительных компонентов (веб-сервер, планировщик и воркеры), поскольку предоставляет простой и гибкий сервис вычислений на базе контейнеров. Amazon RDS применяется в качестве управляемого сервиса реляционных баз данных для базы метаданных, а EFS и S3 – для хранения. Стрелки указывают на зависимости между сервисами

16.1.2 Проектирование сети

Нам также необходимо подумать о том, как эти сервисы будут связаны и как управлять доступом к Airflow через интернет. Типичный вариант – создать VPC (виртуальное частное облако), содержащее общедоступные и частные подсети. В данном типе настройки частные подсети внутри VPC могут использоваться для сервисов, которые не должны быть доступны напрямую через интернет, в то время как общедоступ-

ные подсети могут использоваться для предоставления внешнего доступа к сервисам и исходящего подключения к интернету.

У нас есть несколько сервисов, которые необходимо подключить к сети для развертывания Airflow. Например, контейнеры веб-сервера и планировщика должны иметь доступ к базе метаданных Airflow RDS и EFS для получения ОАГ. Мы можем организовать этот доступ, подключив оба контейнера, RDS и наш экземпляр EFS, к нашей частной подсети. Это также гарантирует, что эти сервисы не будут доступны напрямую из интернета (рис. 16.2). Чтобы обеспечить контейнерам доступ к S3, мы также можем разместить частную конечную точку S3 в частной подсети, которая гарантирует, что любой связанный с S3 трафик не покинет VPC.

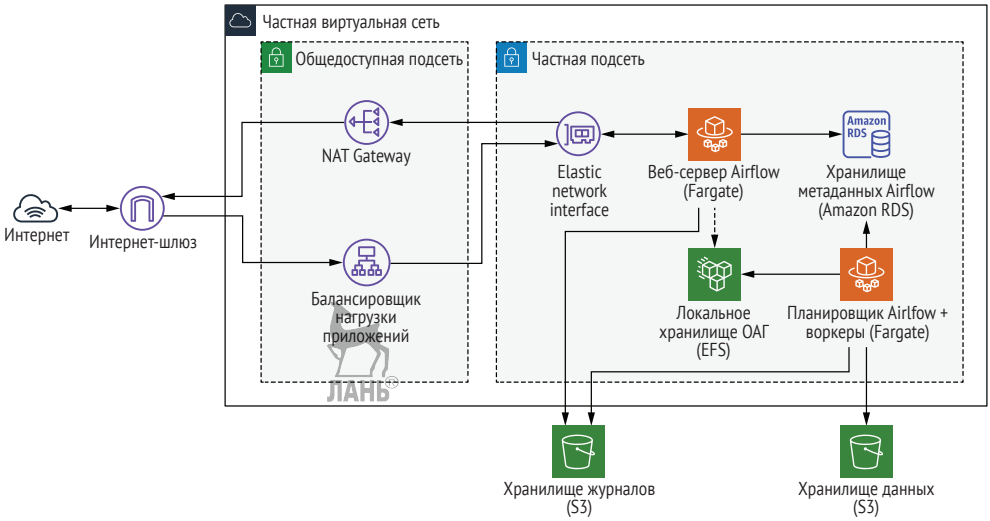


Рис. 16.2 Проецирование компонентов на схему сети с общедоступными и частными подсетями. Общедоступная подсеть обеспечивает доступ к веб-серверу через интернет с помощью балансировщика нагрузки приложений в сочетании с интернет- и NAT-шлюзами для маршрутизации трафика в интернет и из него. Частная подсеть гарантирует, что наши вычислительные компоненты и компоненты хранилища могут связываться друг с другом без непреднамеренного к ним доступа по сети. Стрелки указывают направление потока информации между сервисами

Также нужно предоставить доступ к нашему веб-серверу Airflow через интернет (конечно же, с надлежащим контролем доступа), чтобы мы могли получить доступ к нему из нашего рабочего пространства. Типичный подход – разместить его за балансировщиком нагрузки приложений, который доступен в открытой подсети через интернет-шлюз. Этот балансировщик будет обрабатывать все входящие соединения и при необходимости перенаправлять их в контейнер нашего веб-сервера, если это необходимо. Чтобы убедиться, что наш веб-сервер также может отправлять ответы на наши запросы, нам необходимо разместить NAT Gateway в общедоступной подсети.

16.1.3 Добавление синхронизации ОАГ

Как упоминалось ранее, недостаток использования EFS для хранения ОАГ состоит в том, что к EFS не очень легко получить доступ с помощью веб-интерфейсов или инструментов командной строки. Таким образом, вам, возможно, понадобится настроить процесс автоматической синхронизации ОАГ из другого хранилища, такого как S3 или репозиторий Git.

Одно из возможных решений – создать функцию Lambda, которая позаботится о синхронизации ОАГ из git или S3 с EFS (рис. 16.3). Эту функцию можно активировать (либо с помощью событий S3, либо конвейером сборки в случае Git) для синхронизации любых измененных ОАГ с EFS, делая изменения доступными для Airflow.

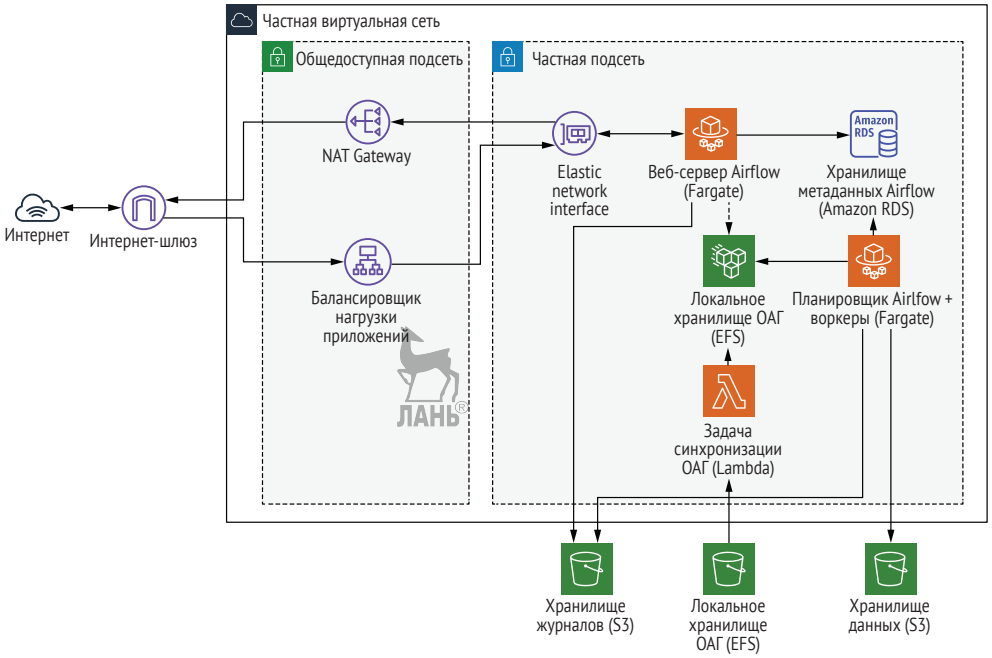


Рис. 16.3 Добавляем автоматическую синхронизацию ОАГ в нашу архитектуру. Это позволяет нам сохранять и редактировать ОАГ в S3, к которому, как правило, легче получить доступ и с которым проще взаимодействовать, чем с EFS. Lambda автоматически позаботится о синхронизации новых ОАГ из S3 с EFS

16.1.4 Масштабирование с помощью CeleryExecutor

Хотя такая установка должна быть достаточно надежной, чтобы справиться со множеством рабочих нагрузок, можно улучшить масштабируемость развертывания Airflow, переключившись на `CeleryExecutor`. Основное преимущество этого перехода заключается в том, что `CeleryExecutor` позволяет запускать каждый воркер Airflow в собствен-

ном экземпляре контейнера, тем самым существенно увеличивая ресурсы, доступные каждому воркеру.

Чтобы использовать `CeleryExecutor`, нужно внести ряд изменений в наш дизайн (рис. 16.4). Во-первых, необходимо настроить отдельный пул задач Fargate для воркеров Airflow, которые выполняются в отдельных процессах в настройке на базе Celery. Обратите внимание, что эти задачи также должны иметь доступ к базе метаданных Airflow и бакету журналов, чтобы иметь возможность сохранять свои журналы и результаты. Во-вторых, нужно добавить брокера сообщений, который передает задания от планировщика воркерам. Хотя можно было бы разместить собственного брокера сообщений (например, RabbitMQ или, возможно, Redis) в Fargate, возможно, проще использовать сервис AWS, SQS, который предоставляет простой бессерверный брокер сообщений, требующий минимальных усилий в сопровождении.

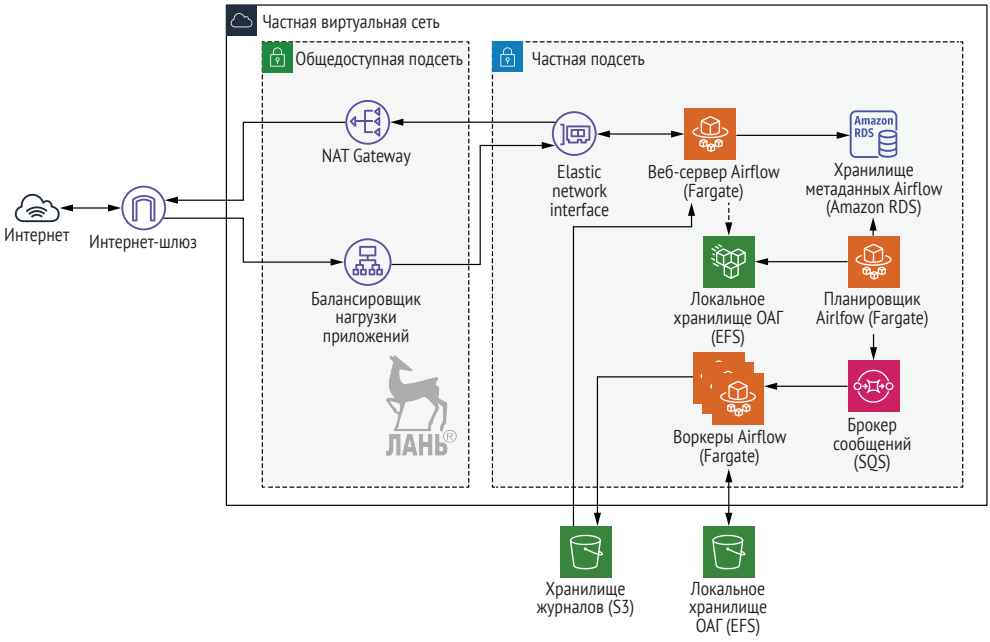


Рис. 16.4 Альтернативное развертывание на базе `CeleryExecutor`. `CeleryExecutor` запускает воркеры в отдельных вычислительных процессах, которые выполняются как отдельные экземпляры контейнера в Fargate. Сервис SQS используется в качестве брокера сообщений для передачи задач воркерам, после того как они были запланированы

Конечно, недостаток использования `CeleryExecutor` состоит в том, что настройка здесь немного сложнее, чем у `LocalExecutor`, и поэтому требует больше усилий. Добавленные компоненты (в первую очередь дополнительные задачи воркеров) могут также привести к значительным затратам на дополнительные вычислительные ресурсы, необходимые для каждого воркера.

16.1.5 Дальнейшие шаги

Хотя мы набросали базовые стратегии развертывания Airflow в AWS, нужно отметить, что эти настройки не следует рассматривать как готовые к промышленной эксплуатации, поскольку еще необходимо учесть ряд факторов.

Прежде всего важным фактором при развертывании в промышленном окружении является безопасность. Несмотря на то что мы приложили определенные усилия для защиты различных компонентов от открытого доступа через интернет, все же необходимо рассмотреть возможность дальнейшего ограничения доступа к компонентам с помощью групп безопасности и сетевых списков управления доступом, ограничивая доступ к ресурсам AWS с использованием соответствующих ролей и политик IAM (Identity and Access Management). На уровне Airflow вы также должны подумать, как хотите обезопасить Airflow (использовать механизм RBAC и т. д.).

Кроме того, мы ожидаем, что развертывание в промышленном окружении будет иметь надежный подход к журналированию, аудиту и отслеживанию метрик, а также к созданию оповещений в случае возникновения проблем с каким-либо из развернутых сервисов. Для этого мы рекомендуем приглядеться к соответствующим сервисам, предоставляемым AWS, включая CloudTrail и CloudWatch.

16.2 Хуки и операторы, предназначенные для AWS

Airflow предоставляет значительное количество встроенных хуков и операторов, которые позволяют взаимодействовать с большим количеством сервисов AWS. Они дают возможность (например) координировать процессы, включающие перемещение и преобразование данных между различными сервисами, а также развертывание всех необходимых ресурсов. Обзор всех доступных хуков и операторов *см.* в пакете поставщика Amazon/AWS[1].

Ввиду их большого количества мы не будем вдаваться в подробности, а предлагаем обратиться к соответствующей документации. Однако в табл. 16.1 и 16.2 представлен краткий обзор хуков и операторов с сервисами AWS, с которыми они связаны, и их соответствующими приложениями. В следующем разделе представлена демонстрация некоторых из этих хуков и операторов.

[1] Их можно установить в Airflow 2 с использованием пакета `apache-airflow-provider-amazon` или в Airflow 1.10 с помощью пакета `backport apache-airflow-backport-sizes-amazon`.

Таблица 16.1 Хуки, предназначенные для AWS

Сервис	Описание	Хук	Приложения
Athena	Бессерверные запросы больших данных	`AWSAthenaHook`	Выполнение запросов, опрос статуса запроса, получение результатов
CloudFormation	Управление ресурсами (стеками) инфраструктуры	`AWSCloudFormationHook`	Создание и удаление стеков CloudFormation
EC2	Виртуальные машины	`EC2Hook`	Получение сведений о виртуальных машинах; ожидание изменения состояния
Glue	Управляемый ETL-сервис	`AwsGlueJobHook`	Создание заданий Glue и проверка их статуса
Lambda	Бессерверные функции	`AwsLambdaHook`	Вызов функций Lambda
S3	Простой сервис хранения	`S3Hook`	Перечисление и загрузка/скачивание файлов
SageMaker	Управляемый сервис машинного обучения	`SageMakerHook`	Создание заданий машинного обучения, конечных точек и т. д. и управление ими

Таблица 16.2 Операторы, предназначенные для AWS

Оператор	Сервис	Описание
`AWSAthenaOperator`	Athena	Выполнение запроса к Athena
`CloudFormationCreateStackOperator`	CloudFormation	Создание стека CloudFormation
`CloudFormationDeleteStackOperator`	CloudFormation	Удаление стека CloudFormation
`S3CopyObjectOperator`	S3	Копирование объектов в S3
`SageMakerTrainingOperator`	SageMaker	Создание учебного задания SageMaker

Особого упоминания заслуживает `AwsBaseHook`, который предоставляет универсальный интерфейс для сервисов AWS с помощью библиотеки boto3. Чтобы использовать его, создайте его экземпляр со ссылкой на соединение Airflow, которое содержит соответствующие учетные данные AWS:

```
from airflow.providers.amazon.aws.hooks.base_aws import AwsBaseHook

hook = AwsBaseHook("my_aws_conn")
```

Требуемое соединение можно создать в Airflow с помощью веб-интерфейса (рис. 16.5) или других подходов к настройке (например, переменные окружения). По сути, требуется две вещи: ключ доступа и секрет, указывающие на пользователя IAM в AWS[1].

После того как мы создали экземпляр хука, мы можем использовать его для создания клиентов boto3 для разных сервисов с помощью метода `get_client_type`. Например, можно создать клиента для сервиса AWS Glue:

```
glue_client = hook.get_client_type("glue")
```

[1] В следующем разделе мы приведем пример того, как получить эти сведения.

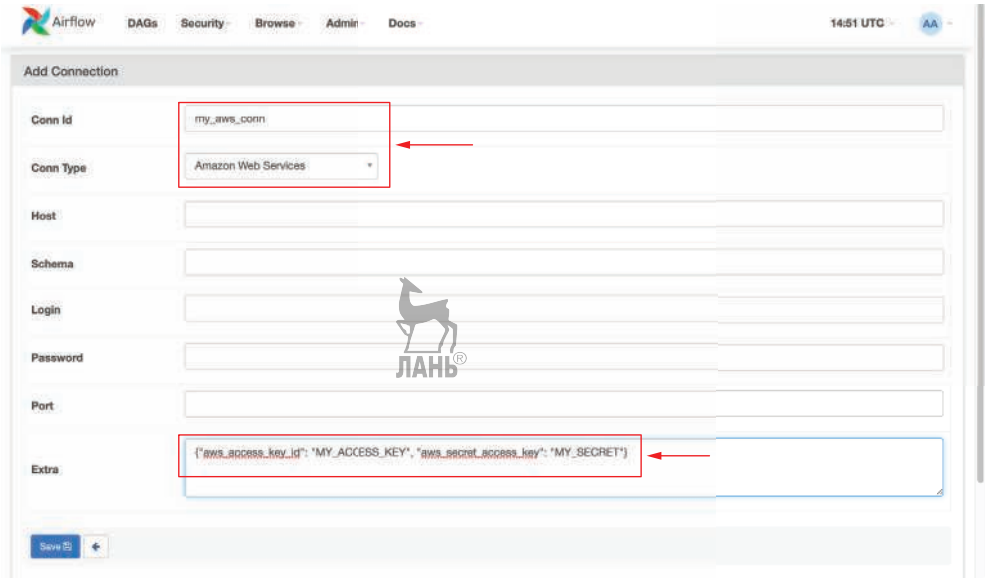


Рис. 16.5 Создание подключения для хука AWS в Airflow. Обратите внимание, что ключ доступа и секрет следует вводить в формате JSON в поле Extra, а не в поля Login и Password (вопреки тому, что вы могли ожидать)

С помощью этого клиента мы можем выполнять все виды операций с сервисом Glue в AWS. Для получения дополнительных сведений о различных типах клиентов и поддерживаемых операциях вы можете обратиться к документации boto3 (https://boto3.amazonaws.com/v1/documentation/api/latest/index.html). Чтобы иметь возможность выполнять любую из этих операций, у IAM-пользователя должны быть соответствующие полномочия в AWS. Таким образом, убедитесь, что вы назначили их соответствующему пользователю с помощью политик IAM.

В следующем разделе мы покажем пример создания собственного оператора на основе `AwsBaseHook`, который демонстрирует, как все это взаимосвязано.

16.3 Пример использования: бессерверное ранжирование фильмов с AWS Athena

Чтобы изучить некоторые из этих функций, рассмотрим небольшой пример.

16.3.1 Обзор

В этом примере нас интересует использование бессерверных сервисов в AWS (S3, Glue, Athena) для анализа данных фильмов, с которыми мы

столкнулись в предыдущих главах. Наша цель – найти самые популярные фильмы, ранжируя их по средней оценке (используя все рейтинги до этого момента). Одно из преимуществ использования бессерверных сервисов для данной задачи заключается в том, что нам не нужно беспокоиться о запуске и сопровождении каких-либо серверов, что делает общую настройку относительно дешевой (мы платим, только пока все работает) и требует относительно небольшого сопровождения.

Чтобы создать бессерверный процесс ранжирования фильмов, нужно выполнить ряд шагов:

- сначала мы извлекаем рейтинги фильмов из нашего API и загружаем их в S3, чтобы они были доступны в AWS. Мы планируем загружать данные ежемесячно, дабы можно было рассчитывать рейтинги за каждый месяц по мере поступления новых данных;
- потом мы используем AWS Glue (бессерверный сервис ETL) для сканирования данных рейтингов в S3. Таким образом, Glue создает табличное представление данных, хранящихся в S3, которое мы можем впоследствии запросить для расчета наших рейтингов;
- наконец, мы используем AWS Athena (бессерверный сервис SQL-запросов) для выполнения SQL-запроса к таблице рейтингов для расчета рейтингов фильмов. Вывод этого запроса записан в S3, поэтому мы можем использовать рейтинги в любых приложениях.

Это дает нам относительно простой подход (рис. 16.6) к ранжированию фильмов, которые должны легко масштабироваться до больших наборов данных (поскольку S3 и Glue/Athena – это высокомасштабируемые технологии). Более того, понятие *бессерверный* означает, что нам не нужно платить ни за какие серверы, чтобы выполнить этот процесс, а это снижает затраты. Здорово, правда?

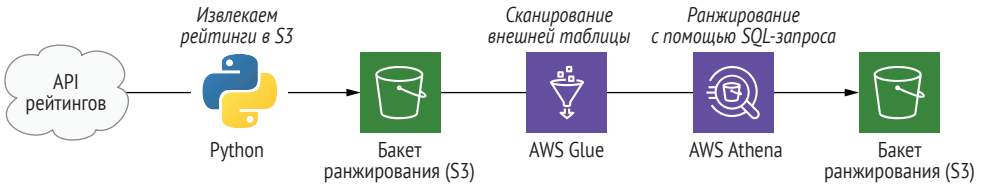


Рис. 16.6 Обзор процесса обработки данных в случае использования бессерверного ранжирования фильмов. Стрелки указывают на преобразования данных, выполняемые в Airflow, отмеченные соответствующим сервисом AWS, используемым для выполнения преобразования данных (если это применимо)

16.3.2 Настройка ресурсов

Перед реализацией ОАГ начнем с создания необходимых ресурсов в AWS. Нашему ОАГ потребуются следующие облачные ресурсы:

- бакет S3, который будет содержать наши данные по рейтингам;
- второй бакет S3, в котором будут храниться результаты ранжирования;

- поисковый робот Glue, который создаст таблицу на основе наших данных о рейтингах;
- пользователь IAM, который позволит нам получить доступ к бакетам S3 и вызывать такие сервисы, как Glue и Athena.

Один из способов настройки этих ресурсов – открыть AWS Console (http://console.aws.amazon.com) и создать необходимые ресурсы вручную в соответствующих разделах консоли. Однако из соображений воспроизводимости мы рекомендуем определять ресурсы и управлять ими с помощью решения Iac (Infrastructure-as-code – инфраструктура как код), такого как CloudFormation (решение для создания шаблонов для определения облачных ресурсов в коде). Для этого примера мы предоставили шаблон CloudFormation, который создает все необходимые ресурсы в вашей учетной записи. Для краткости мы не будем здесь углубляться в детали шаблона, но с радостью порекомендуем вам ознакомиться с ним в интернете.

Чтобы создать необходимые ресурсы с помощью нашего шаблона, откройте консоль AWS, перейдите в раздел **CloudFormation** и щелкните **Create Stack** (Создать стек) (рис. 16.7). На следующей странице загрузите предоставленный шаблон и нажмите **Next** (Далее). На странице сведений о стеке введите имя своего стека (= этот набор ресурсов) и уникальный префикс для имен бакета S3 (это необходимо для того, чтобы сделать их уникальными глобально). Теперь щелкните «Далее» еще несколько раз (не забудьте выбрать **I ac-**

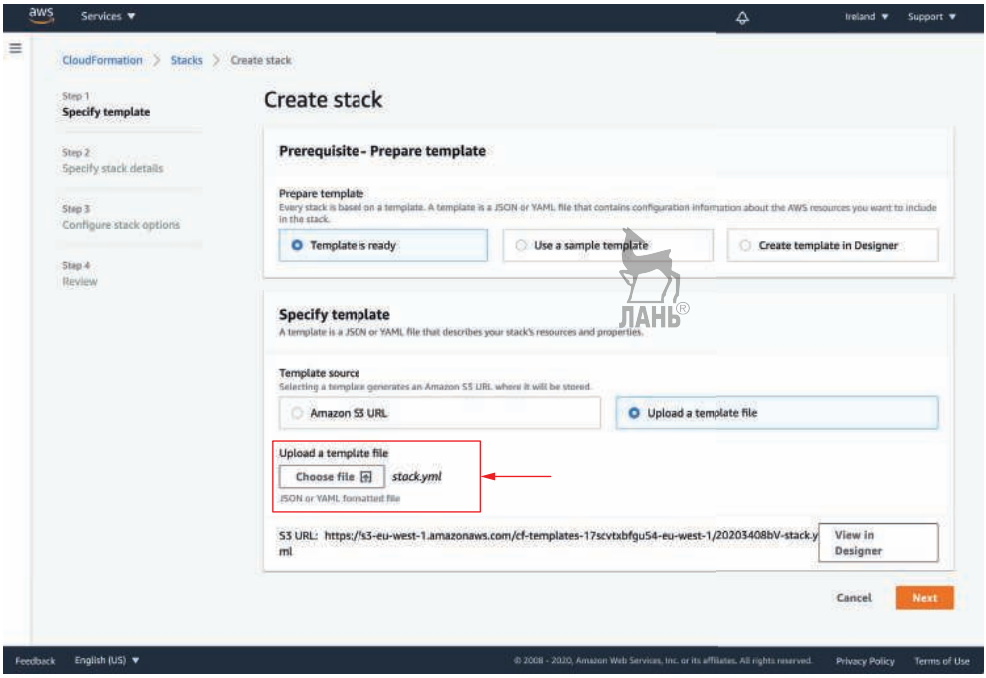


Рис. 16.7 Создание стека CloudFormation в консоли AWS

knowledge that AWS CloudFormation might create IAM resources with custom names (Я подтверждаю, что AWS CloudFormation может создавать ресурсы IAM с настраиваемыми именами) на странице обзора), после чего CloudFormation должен приступить к созданию ваших ресурсов.

После завершения вы сможете увидеть статус созданного стека на странице обзора стека CloudFormation (рис. 16.8). Также на вкладке **Resources** (Ресурсы) можно увидеть, какие ресурсы CloudFormation создал для вас (рис. 16.9). Здесь должны быть IAM-пользователь и набор политик доступа, два бакета S3 и наш поисковый робот Glue. Обратите внимание, что вы можете перейти к другим ресурсам, щелкнув по ссылке физического идентификатора каждого ресурса. Так вы сможете перейти к соответствующему ресурсу в соответствующем разделе консоли AWS.

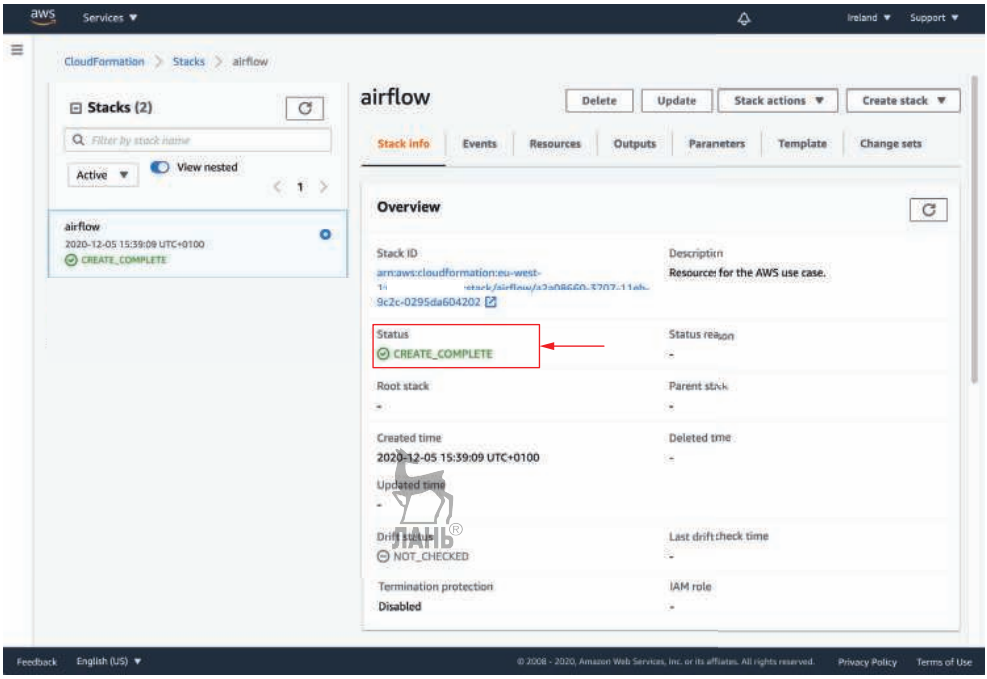


Рис. 16.8 Обзор созданного стека CloudFormation в консоли AWS. На этой странице показано общее состояние стека. При необходимости она также предоставляет вам элементы управления для обновления или удаления стека

Если во время создания стека что-то пошло не так, то можно попробовать определить проблему, используя события на вкладке **Events** (События). Такое может произойти, если, например, имена ваших сегментов конфликтуют с уже существующими сегментами другого пользователя (поскольку они должны быть уникальными глобально).

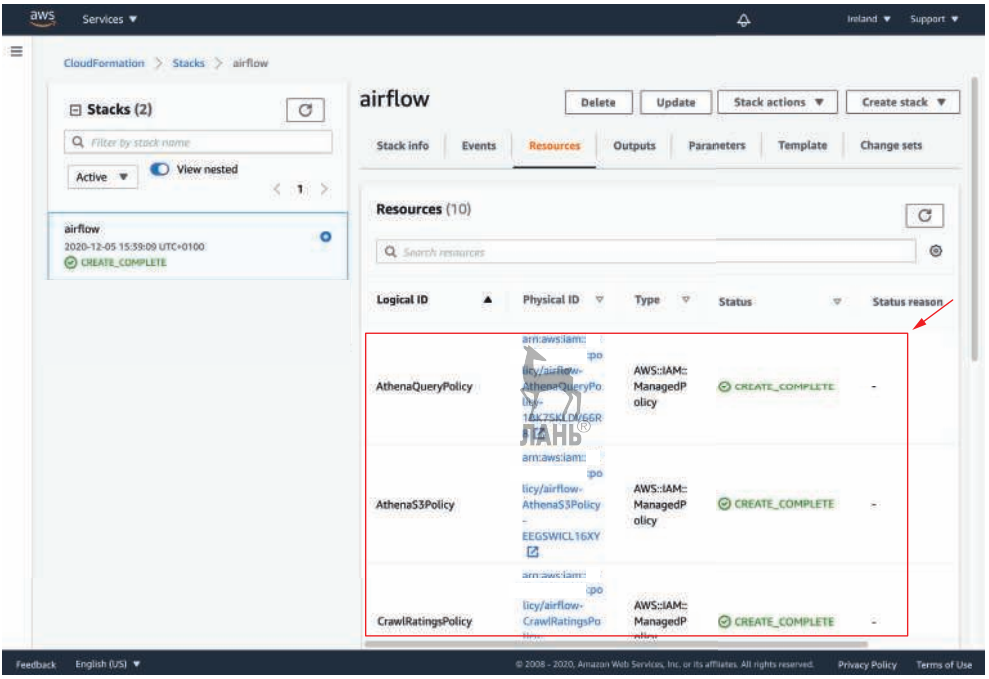


Рис. 16.9 Обзор созданных стеком CloudFormation ресурсов. Данный вывод можно использовать для перехода к различным ресурсам, созданным стеком

Когда у нас есть необходимый набор ресурсов, нам остается сделать одно. Чтобы иметь возможность использовать IAM-пользователя, созданного стеком в нашем ОАГ, необходимо создать ключ доступа и секрет, которые можно использовать совместно с Airflow. Чтобы создать их, прокрутите страницу вниз, пока не найдете ресурс `AWS: IAM: USER`, созданный стеком, и щелкните по ссылке его физического идентификатора. Перед вами должен открыться обзор пользователей в консоли IAM. Затем перейдите на вкладку **Security credentials** (Учетные данные безопасности) и нажмите **Create access key** (Создать ключ доступа) (рис. 16.10). Запишите сгенерированные ключ доступа и секрет и храните их в безопасном месте, так как они нам понадобятся позже.

16.3.3 Создание ОАГ

Теперь, когда у нас есть все необходимые ресурсы, начнем реализацию нашего ОАГ с поиска подходящих хуков и операторов. На первом этапе нам понадобится оператор, который извлекает данные из нашего API рейтингов фильмов и загружает их в S3. Хотя Airflow предоставляет ряд встроенных операторов S3, ни один из них не позволяет получать рейтинги из наших API и загружать их напрямую в S3. К счастью, мы можем реализовать этот шаг, объединив `Python-`

Operator и S3Hook. Вместе эти классы позволяют получать рейтинги с помощью наших собственных функций Python, а затем загружать результаты в S3.

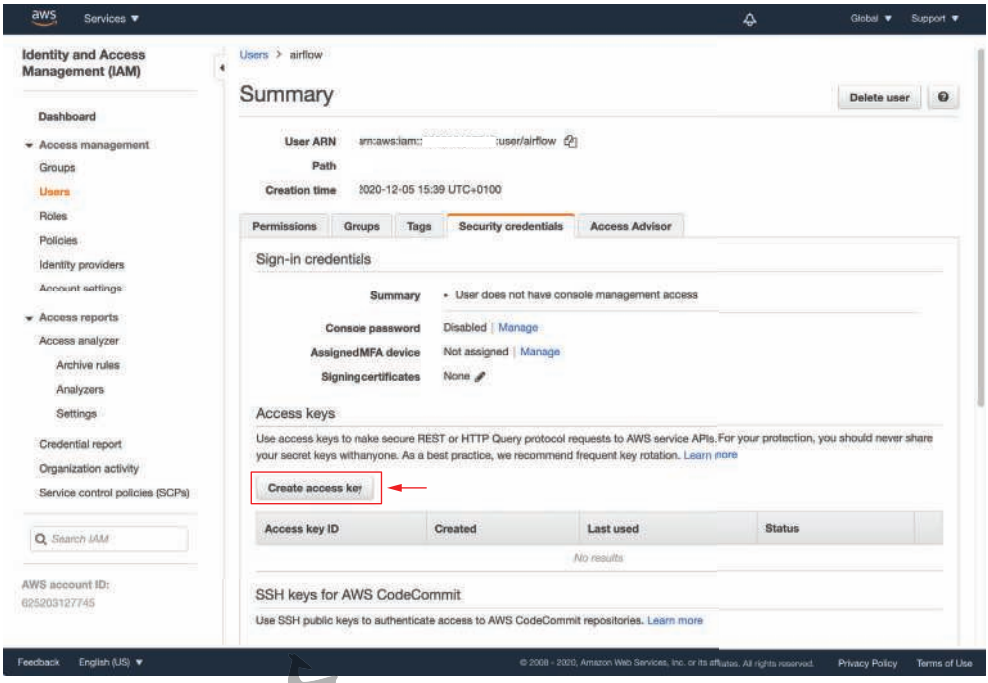


Рис. 16.10 Создание ключа доступа и секрета для сгенерированного пользователя

Листинг 16.1 Загрузка рейтингов с помощью S3Hook (dags/01_aws_usecase.py)

```
from airflow.operators.python import PythonOperator
from airflow.providers.amazon.aws.hooks.s3 import S3Hook

from custom.hooks import MovielensHook

def _fetch_ratings(api_conn_id, s3_conn_id, s3_bucket, **context):
    year = context["execution_date"].year
    month = context["execution_date"].month

    logging.info(f"Fetching ratings for {year}/{month:02d}")
    api_hook = MovielensHook(conn_id=api_conn_id)
    ratings = pd.DataFrame.from_records(
        api_hook.get_ratings_for_month(year=year, month=month),
        columns=["userId", "movieId", "rating", "timestamp"],
    )

    logging.info(f"Fetched {ratings.shape[0]} rows")
```

Извлекаем рейтинги из API с помощью MovielensHook из главы 8 (код хука доступен в dags/custom/hooks.py)

Запись рейтингов во временный каталог

```
with tempfile.TemporaryDirectory() as tmp_dir:
    tmp_path = path.join(tmp_dir, "ratings.csv")
    ratings.to_csv(tmp_path, index=False)

    logging.info(f"Writing results to ratings/{year}/{month:02d}.csv")
    s3_hook = S3Hook(s3_conn_id)
    s3_hook.load_file(
        tmp_path,
        key=f"ratings/{year}/{month:02d}.csv",
        bucket_name=s3_bucket,
        replace=True,
    )
```

Загружаем записанные рейтинги в S3 с помощью S3Hook

```
fetch_ratings = PythonOperator(
    task_id="fetch_ratings",
    python_callable=_fetch_ratings,
    op_kwargs={
        "api_conn_id": "movielens",
        "s3_conn_id": "my_aws_conn",
        "s3_bucket": "my_ratings_bucket",
      },
)
```

Обратите внимание, что для `S3Hook` требуется идентификатор подключения, который указывает, какое подключение (т. е. какие учетные данные) использовать для присоединения к S3. Таким образом, нам необходимо убедиться, что Airflow настроен с подключением, у которого имеется ключ доступа и секрет для пользователя с достаточными полномочиями. К счастью, мы уже создали такого пользователя в предыдущем разделе (используя стек CloudFormation) и теперь можем использовать учетные данные для создания подключения Airflow (рис. 16.5). После создания подключения не забудьте подставить его имя и имя своего бакета S3 (в аргументе `op_kwargs` в `PythonOperator`).

Для второго шага нам понадобится оператор, который может подключиться к AWS для запуска поискового робота Glue (который также был создан стеком CloudFormation). К сожалению, Airflow не предоставляет оператора для этого действия, а это значит, что нужно создать собственного. Однако мы можем использовать встроенный `AwsBaseHook` в качестве основы для нашего оператора, который обеспечивает легкий доступ к различным сервисам AWS с помощью boto3.

Используя `AwsBaseHook`, мы можем создать собственного оператора[1] (`GlueTriggerCrawlerOperator`), который, по сути, получает клиента Glue с помощью `AwsBaseHook` и использует его для запуска нашего поискового робота с помощью метода этого клиента, `start_crawler`. После проверки успешности запуска поискового робота мы можем проверить его статус с помощью метода клиента `get_crawler`, который (помимо прочего) возвращает статус робота. Как только робот

[1] *См.* главу 8 для получения дополнительной информации о создании собственных операторов.

достигнет состояния готовности, мы можем быть уверены[1], что он завершил свою работу, а это означает, что мы можем продолжить выполнение всех нижестоящих задач. В целом реализация этого оператора могла бы выглядеть примерно так:

Листинг 16.2 Оператор для запуска поисковых роботов Glue (dags/custom/operator.py)

```
import time
from airflow.models import BaseOperator
from airflow.providers.amazon.aws.hooks.base_aws import AwsBaseHook
from airflow.utils.decorators import apply_defaults

class GlueTriggerCrawlerOperator(BaseOperator):
    """
    Оператор, инициирующий запуск робота в AWS Glue.

    Параметры
    ----------
    aws_conn_id
        Соединие для подключения к AWS. Для этого необходимы соответствующие
        полномочия (Glue:StartCrawler and Glue:GetCrawler) в AWS.
    crawler_name
        Имя робота.
    region_name
        Имя региона AWS, в котором находится робот.
    kwargs
        Любые аргументы kwargs, передаваемые оператору BaseOperator.
    """
    @apply_defaults
    def __init__(
        self,
        aws_conn_id: str,
        crawler_name: str,
        region_name: str = None,
        **kwargs
    ):
        super().__init__(**kwargs)
        self._aws_conn_id = aws_conn_id
        self._crawler_name = crawler_name
        self._region_name = region_name

    def execute(self, context):
        hook = AwsBaseHook(
            self._aws_conn_id, client_type="glue",
            region_name=self._region_name
        )
        glue_client = hook.get_conn()
```

Создаем экземпляр AwsBaseHook и получаем клиента для AWS Glue

[1] Этот пример, возможно, можно было бы сделать понадежнее, добавив больше проверок неожиданных ответов, статусов и т. д.

```
    self.log.info("Triggering crawler")
    response = glue_client.start_crawler(Name=self._crawler_name)
    if response["ResponseMetadata"]["HTTPStatusCode"] != 200:
        raise RuntimeError(
            "An error occurred while triggering the crawler: %r" % response
        )

    self.log.info("Waiting for crawler to finish")
    while True:
        time.sleep(1)
        crawler = glue_client.get_crawler(Name=self._crawler_name)
        crawler_state = crawler["Crawler"]["State"]

        if crawler_state == "READY":
            self.log.info("Crawler finished running")
            break
```

Используем клиента Glue, чтобы запустить поискового робота

Убеждаемся, что запуск робота прошел успешно

Выполняем цикл, чтобы проверить состояние робота

Останавливаемся, как только робот завершил работу (отображено состояние READY)

Мы можем использовать `GlueTriggerCrawlerOperator` следующим образом.

Листинг 16.3 Использование GlueTriggerCrawlerOperator (dags/01_aws_usecase.py)

```
from custom.operators import GlueTriggerCrawlerOperator

trigger_crawler = GlueTriggerCrawlerOperator(
    aws_conn_id="my_aws_conn",
    task_id="trigger_crawler",
    crawler_name="ratings-crawler",
)
```

Наконец, для третьего этапа нам понадобится оператор, который позволяет выполнять запрос в Athena. На этот раз нам повезло, поскольку Airflow предоставляет такого оператора – это `AwsAthenaOperator`. Ему требуется ряд аргументов: соединение с Athena, база данных (которая должна была быть создана поисковым роботом Glue), запрос на выполнение и место вывода в S3 для записи результатов запроса. В целом использование оператора будет выглядеть примерно так.

Листинг 16.4 Ранжирование фильмов с помощью AWSAthenaOperator (dags/01_aws_usecase.py)

```
from airflow.providers.amazon.aws.operators.athena import AWSAthenaOperator

rank_movies = AWSAthenaOperator(
    task_id="rank_movies",
    aws_conn_id="my_aws_conn",
    database="airflow",
    query="""
```

```
        SELECT movieid, AVG(rating) as avg_rating, COUNT(*) as num_ratings
        FROM (
            SELECT movieid, rating,
                CAST(from_unixtime(timestamp) AS DATE) AS date
            FROM ratings
        )
        WHERE date <= DATE('{{ ds }}')
        GROUP BY movieid
        ORDER BY avg_rating DESC
        """,
        output_location=f"s3://my_rankings_bucket/{{ds}}",
    )
```

Получаем идентификатор фильма, значение рейтинга и дату каждого рейтинга

Выбираем все рейтинги до даты выполнения

Группируем по идентификатору фильма, чтобы рассчитать средний рейтинг для каждого фильма

Теперь, когда мы создали все необходимые задачи, можно начать связывать все воедино в общем ОАГ.

Листинг 16.5 Создание общего ОАГ рекомендательной системы (dags/01_aws_usecase.py)

```
import datetime as dt
import logging
import os
from os import path
import tempfile

import pandas as pd

from airflow import DAG
from airflow.providers.amazon.aws.hooks.s3 import S3Hook
from airflow.providers.amazon.aws.operators.athena import AWSAthenaOperator
from airflow.operators.dummy import DummyOperator
from airflow.operators.python import PythonOperator

from custom.operators import GlueTriggerCrawlerOperator
from custom.ratings import fetch_ratings

with DAG(
    dag_id="01_aws_usecase",
    description="DAG demonstrating some AWS-specific hooks and operators.",
    start_date=dt.datetime(year=2019, month=1, day=1),
    end_date=dt.datetime(year=2019, month=3, day=1),
    schedule_interval="@monthly",
    default_args={
        "depends_on_past": True
    }
) as dag:
    fetch_ratings = PythonOperator(...)
    trigger_crawler = GlueTriggerCrawlerOperator(...)
    rank_movies = AWSAthenaOperator(...)
    fetch_ratings >> trigger_crawler >> rank_movies
```

Задаем даты начала и окончания в соответствии с набором данных по рейтингам

Используем depends_on_past, чтобы избежать выполнения запросов до того, как будут загружены архивные данные (что может дать неполные результаты)

Теперь, когда все готово, мы можем запустить наш ОАГ в Airflow (рис. 16.11). Предполагая, что все настроено правильно, запуск дол-

жен быть успешным, и вы должны увидеть, как выходные данные в формате CSV из Athena появятся в вашем бакете (рис. 16.12). Если у вас возникнут проблемы, убедитесь, что ресурсы AWS были настроены правильно, а ваш ключ доступа и секрет были сконфигурированы должным образом.

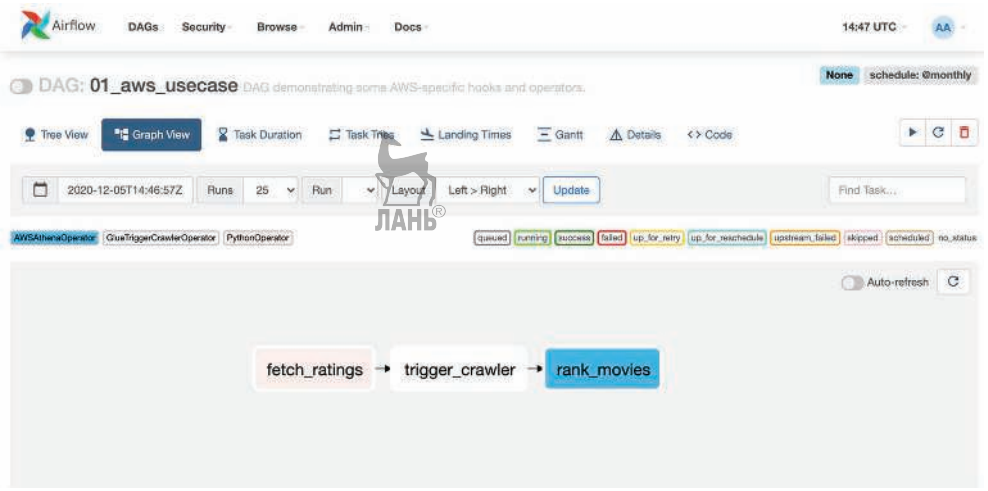


Рис. 16.11 Полученный в результате ОАГ, иллюстрирующий три разные задачи и соответствующих операторов, задействованных в каждой задаче

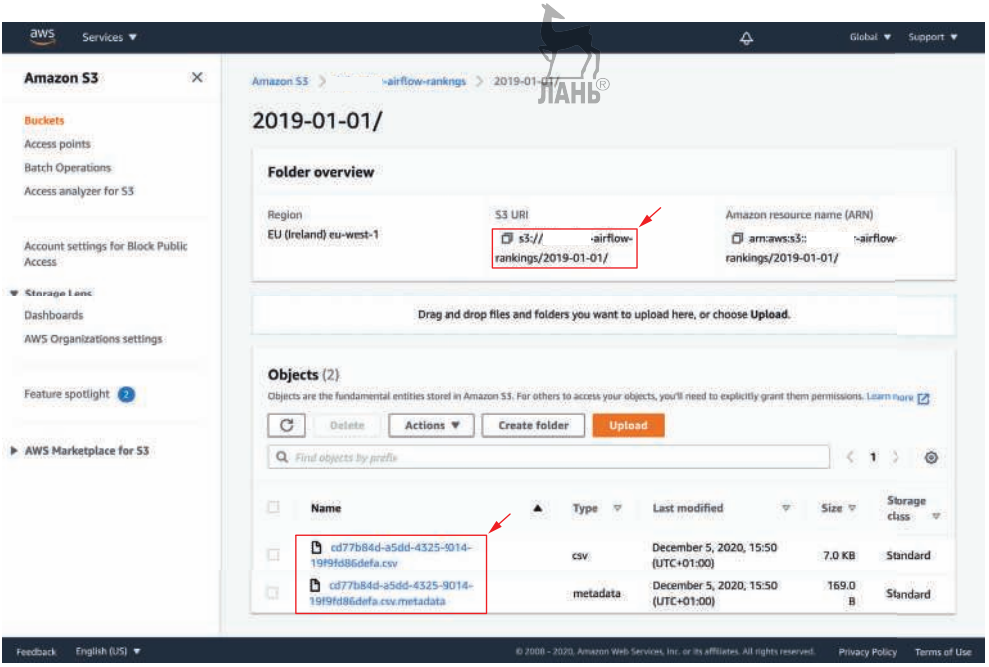


Рис. 16.12 Результаты запроса Athena в бакете ранжирования

16.3.4 Очистка

После завершения этого примера не забудьте очистить все ресурсы, которые вы создали в AWS, чтобы избежать ненужных затрат. Если вы использовали шаблон CloudFormation для создания ресурсов, то можете убрать большую часть, удалив стек. Обратите внимание, что некоторые ресурсы, такие как бакеты S3, придется удалить вручную, даже если вы используете шаблон, поскольку CloudFormation не позволит автоматически удалить заполненные бакеты. Обязательно проверьте, все ли созданные ресурсы были успешно удалены, уделяя особое внимание проверке всех ресурсов, которые вы могли создать вручную.

Резюме

- Airflow можно развернуть в AWS с помощью таких сервисов, как ECS/Fargate для запуска процессов планировщика и веб-сервера, EFS/S3 для хранения и Amazon RDS для базы метаданных Airflow.
- Airflow предоставляет множество хуков и операторов, предназначенных для AWS, которые позволяют интегрировать различные сервисы с облачной платформой AWS.
- Класс `AwsBaseHook` обеспечивает низкоуровневый доступ ко всем сервисам в AWS с помощью библиотеки boto3, позволяя вам реализовать собственные высокоуровневые хуки и операторы, если они еще не существуют.
- Использование хуков и операторов, предназначенных для AWS, обычно требует настройки необходимых ресурсов и полномочий для доступа к AWS и Airflow, чтобы Airflow мог выполнять требуемые операции.

17 Airflow и Azure

Эта глава рассказывает о:

- проектировании стратегии развертывания для Azure;
- хуках и операторах, предназначенных для Azure, и как их использовать.

В данной главе мы подробнее рассмотрим, как развернуть и интегрировать Airflow с облачными сервисами в облаке Microsoft Azure. Мы начнем проектировать развертывание Airflow с сопоставления различных компонентов Airflow со службами Azure. Затем мы рассмотрим хуки и операторы, которые Airflow предоставляет для интеграции с ключевыми сервисами Azure. Наконец, покажем, как использовать эти операторы и хуки для создания рекомендаций по фильмам.

17.1 Развертывание Airflow в Azure

В главе 15 мы описали различные компоненты, из которых состоит развертывание Airflow. В этом разделе мы спроектируем несколько шаблонов развертывания для Azure, сопоставив их с конкретными

облачными сервисами Azure. Это должно дать вам внятное представление о процессе проектирования развертывания Airflow для Azure и послужить хорошей отправной точкой для его реализации.

17.1.1 Выбор сервисов

Начнем с веб-сервера и планировщика Airflow. Один из самых простых подходов к их запуску – использовать управляемые контейнерные сервисы Azure, такие как Azure Container Instances (ACI) или Azure Kubernetes Service (AKS). Однако для веб-сервера у нас также есть другой вариант: Azure App Service.

Azure App Service – это, по утверждению Microsoft, «полностью управляемая платформа для создания, развертывания и масштабирования ваших веб-приложений». На практике она обеспечивает удобный подход для развертывания веб-сервисов на управляемой платформе, который включает в себя такие функции, как аутентификация и мониторинг. Важно отметить, что Azure App Service поддерживает развертывание приложений в контейнерах, а это означает, что ее можно использовать для развертывания веб-сервера Airflow и переложить на нее все обязанности, касающиеся аутентификации. Конечно, планировщику не нужны какие-либо связанные с сетью функции, предоставляемые App Service. Таким образом, по-прежнему имеет смысл развернуть планировщик в ACI, который предоставляет более простую среду выполнения контейнеров.

Что касается базы метаданных Airflow, имеет смысл обратить внимание на управляемые службы баз данных Azure, такие как Azure SQL Database. Этот сервис предоставляет удобное решение для хранения и обработки реляционных данных, при этом нам не нужно беспокоиться об обслуживании базовой системы.

Azure предоставляет ряд различных решений для хранения, включая Azure File Storage, Azure Blob Storage и Azure Data Lake Storage. Azure File Storage – наиболее удобное решение для размещения ОАГ, поскольку его тома можно подключать непосредственно к контейнерам, работающим в App Service и ACI. Более того, к Azure File Storage легко получить доступ с помощью вспомогательных пользовательских приложений, таких как Azure Storage Explorer, что делает относительно простым добавление или обновление любых ОАГ. Что касается хранилища данных, имеет смысл обратить внимание на Azure Blob или Data Lake Storage, поскольку они лучше подходят для рабочих нагрузок данных, чем File Storage.

Таким образом, мы имеем следующую картину (рис. 17.1):

- App Service для веб-сервера Airflow;
- ACI для планировщика Airflow;
- Azure SQL Database для базы метаданных Airflow;
- Azure File Storage для хранения ОАГ;
- Azure Blob Storage для данных и журналов.

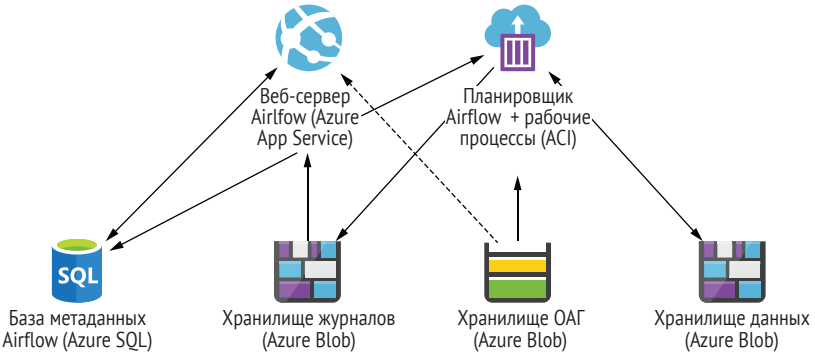


Рис. 17.1 Сопоставление компонентов Airflow, изображенных на рис. 15.1, с сервисами Azure. App Service и ACI используются для вычислительных компонентов (веб-сервер, планировщик и воркер соответственно), поскольку предоставляют удобные вычислительные сервисы на базе контейнеров. App Service применяется для веб-сервера вместо ACI, поскольку она предоставляет дополнительные функции для аутентификации доступа к веб-серверу и т. д. Azure SQL Database используется в качестве управляемой службы базы данных для базы метаданных, в то время как Azure File Storage и Azure Blob Storage используются для хранения ОАГ, журналов и данных

17.1.2 Проектирование сети

Теперь, когда мы выбрали сервисы для каждого компонента, можно приступить к проектированию сетевого взаимодействия между ними. В данном случае нам нужно предоставить доступ к веб-серверу Airflow через интернет, чтобы можно было обращаться к нему удаленно. Однако мы хотим оставить другие компоненты, такие как база метаданных Airflow и планировщик, в частной сети, чтобы закрыть к ним доступ со стороны посторонних лиц.

К счастью, Azure App Service упрощает предоставление доступа к веб-серверу в качестве веб-приложения; это именно то, для чего она и предназначена. Таким образом, мы можем дать ей возможность взять на себя заботу о предоставлении доступа к веб-серверу и подключении его к интернету, а также можем использовать ее встроенные функции, чтобы добавить брандмауэр или уровень аутентификации (который может быть интегрирован с Azure AD и т. д.) перед веб-службой, предотвращая доступ пользователей, не прошедших авторизацию, к веб-серверу.

Для планировщика и базы метаданных можно создать виртуальную сеть с частной подсетью и разместить эти компоненты внутри частной сети (рис. 17.2). Это обеспечит связь между базой метаданных и планировщиком. Чтобы разрешить веб-серверу доступ к базе метаданных, необходимо активировать интеграцию виртуальной сети для App Service.

Azure File Storage и Azure Blob Storage можно интегрировать со службой приложений и ACI. По умолчанию оба этих сервиса доступ-

ны через интернет, а это означает, что их не нужно интегрировать в нашу виртуальную сеть. Однако мы также рекомендуем рассмотреть возможность использования частных конечных точек для подключения учетных записей хранения к частным ресурсам. Это обеспечивает бóльшую безопасность, гарантируя, что трафик данных не будет проходить через общедоступную сеть.

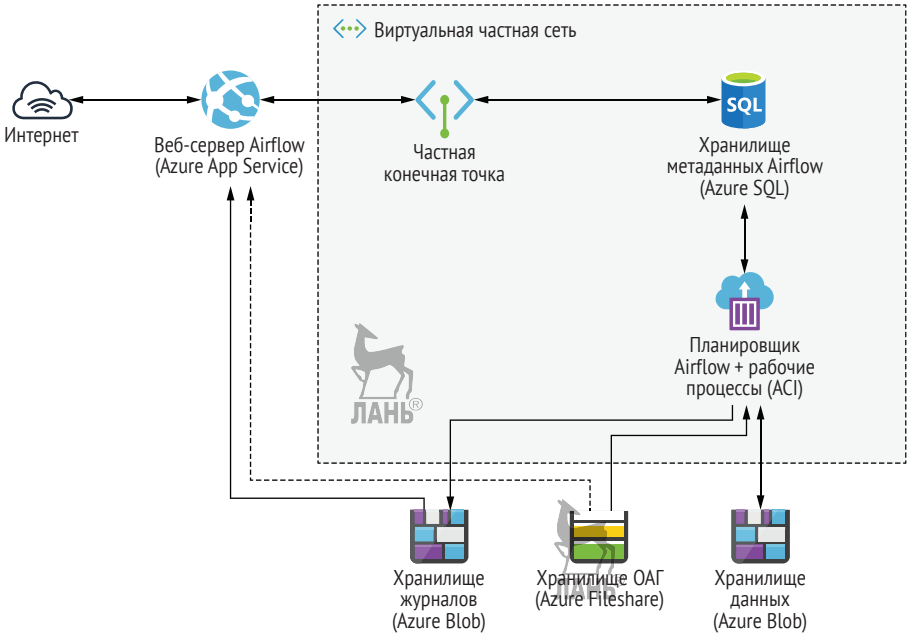


Рис. 17.2 Проецируем наши компоненты на схему частной виртуальной сети. Частная виртуальная сеть изолирует наши внутренние ресурсы (например, базу метаданных и планировщик) от открытого доступа через интернет. Веб-сервер доступен в интернете через Azure App Service, поэтому к нему можно получить удаленный доступ. Интеграция с виртуальной сетью осуществляется с использованием частной конечной точки, чтобы веб-сервер мог подключиться к базе метаданных. Стрелки указывают направление потока информации между службами. Здесь сервисы хранения не ограничены виртуальной сетью, но при желании такое возможно[1]

17.1.3 Масштабирование с помощью CeleryExecutor

Подобно AWS, мы можем улучшить масштабируемость развертывания Azure, переключившись с `LocalExecutor` на `CeleryExecutor`. В Azure переход с одного исполнителя на другой также требует создания пула воркеров Airflow, которые может использовать `CeleryExecutor`. Поскольку мы уже запускаем наш планировщик в ACI, имеет

[1] Доступность сервисов хранения может быть ограничена виртуальной сетью с помощью комбинации частных конечных точек и правил брандмауэра для обеспечения дополнительного уровня безопасности.

смысл создать дополнительные воркеры в качестве дополнительных контейнерных служб, работающих в ACI (рис. 17.3).

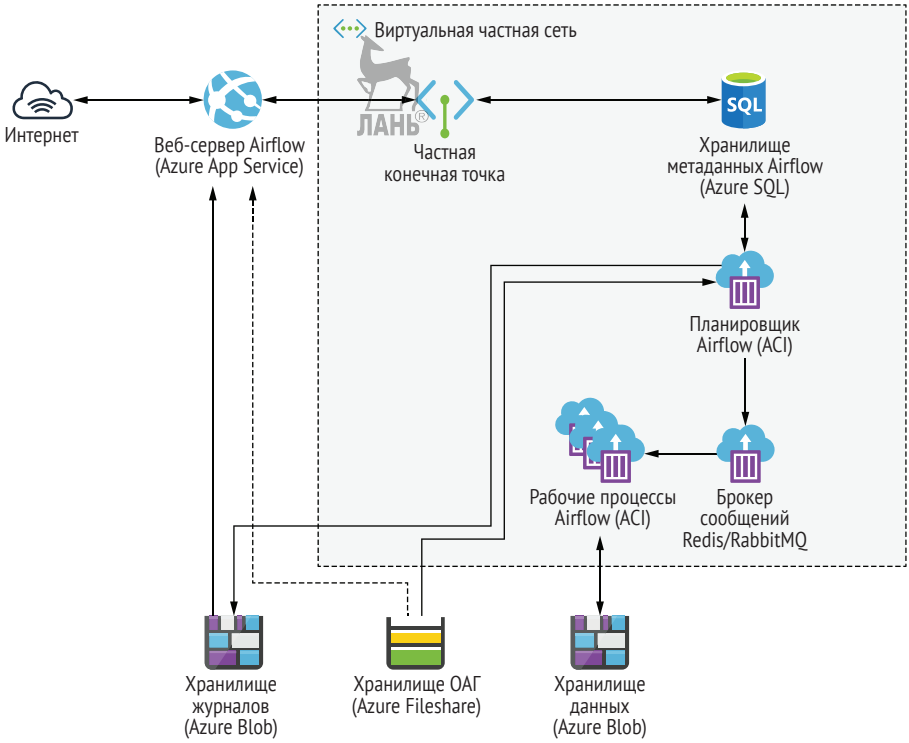


Рис. 17.3 Альтернативное развертывание на базе `CeleryExecutor`. `CeleryExecutor` запускает воркеры в отдельных вычислительных процессах, которые запускаются как отдельные экземпляры контейнера в ACI. Кроме того, экземпляр Redis или RabbitMQ запускается в ACI, чтобы функционировать в качестве брокера сообщений для передачи задач воркерам, после того как они были запланированы

Затем нам также необходимо реализовать брокер сообщений для передачи заданий между планировщиком и воркерами. К сожалению, (насколько нам известно) в Azure нет управляемых решений, которые хорошо интегрируются с Airflow для этой цели. Таким образом, самый простой подход – запустить службу с открытым исходным кодом в ACI, которая может функционировать как брокер сообщений. Для этого можно использовать инструменты с открытым исходным кодом, такие как RabbitMQ и Redis.

17.1.4 Дальнейшие шаги

Хотя это иллюстрирует базовые стратегии развертывания Airflow в Azure, нужно отметить, что они не готовы к промышленной эксплуатации. Как и в случае с AWS, для любой готовой к промышленной эксплуатации настройки все равно потребуется предпринять до-

полнительные шаги, такие как настройка надлежащих межсетевых экранов и средств контроля доступа. На уровне Airflow также следует учитывать, как вы хотите обезопасить Airflow (например, используя механизм RBAC и т. д.).

Мы также ожидаем, что развертывание в промышленном окружении будет иметь надежный подход к журналированию, аудиту, отслеживанию метрик и созданию оповещений в случае возникновения проблем с каким-либо из развернутых сервисов. Для этого мы рекомендуем обратить внимание на соответствующие сервисы, предоставляемые Azure, включая Azure Log Analytics, App Insights и т. д.

17.2 Хуки и операторы, предназначенные для Azure

На момент написания этой книги в Airflow было относительно мало встроенных обработчиков и операторов для облачных сервисов Azure. Вероятно, это отражает предвзятость сообщества Airflow; однако реализовать (и предоставить) собственные хуки и операторы с помощью Azure Python SDK должно быть довольно просто. Кроме того, к некоторым сервисам можно получить доступ с помощью более универсальных интерфейсов (например, ODBC, как будет показано в примере), а это означает, что Airflow по-прежнему может хорошо взаимодействовать с облачными сервисами Azure.

Хуки и операторы Airflow, предназначенные для Azure (табл.17.1 и 17.2), предоставляются пакетом поставщика Microsoft/Azure[1]. Некоторые из этих хуков и операторов можно использовать для взаимо-

Таблица 17.1 Некоторые хуки для Azure

Сервис	Описание	Хук	Приложения
Azure Blob Storage	Хранилище BLOB-объектов	`WasbHook`[a]	Загрузка и скачивание файлов
Azure Container Instances	Управляемый сервис для запуска контейнеров	`AzureContainer-InstanceHook`	Запуск и мониторинг контейнеризированных заданий
Azure Cosmos DB	Мультимодальная база данных	`AzureCosmosDBHook`	Вставка и получение документов
Azure Data Lake Storage	Облачная платформа для аналитики больших данных	`AzureDataLakeHook`	Загрузка и скачивание файлов в/из Azure Data Lake Storage
Azure File Storage	Сервис хранения файлов, совместимый с NFS	`AzureFileShareHook`	Загрузка и скачивание файлов

[a] Windows Azure Storage Blob.

[1] Можно установить в Airflow 2 с помощью пакета поставщиков `apache-airflow-providers-microsoft-azure` или в Airflow 1.10 с использованием пакета `apache-airflow-backport-sizes-microsoft-azure`.

действия с различными сервисами хранилищ Azure (например, Blob, File Share и Data Lake Storage). Дополнительные хуки обеспечивают доступ к специализированным базам данных (например, CosmosDB) и среде запуска контейнеров (например, Azure Container Service).

Таблица 17.2 Операторы для Azure

Оператор	Сервис	Описание
`AzureDataLakeStorageList-Operator`	Azure Data Lake Storage	Перечисляет файлы в определенном пути к файлу
`AzureContainerInstances-Operator`	Azure Container Instances	Запускает контейнеризированное задание
`AzureCosmosInsertDocument-Operator`	Azure Cosmos DB	Вставляет документ в экземпляр базы данных
`WasbDeleteBlobOperator`	Azure Blob Storage	Удаляет конкретный BLOB-объект

17.3 Пример: бессерверное ранжирование фильмов с Azure Synapse

Чтобы познакомиться с использованием сервисов Azure в Airflow, мы реализуем небольшую рекомендательную систему для фильмов с использованием ряда бессерверных сервисов (аналогично варианту использования AWS, но теперь применительно к Azure). В данном случае мы заинтересованы в том, чтобы определять популярные фильмы, ранжируя их на основе среднего пользовательского рейтинга. Используя бессерверные технологии для этой задачи, мы надеемся сохранить относительно простую и экономичную настройку, не беспокоясь о запуске и сопровождении серверов, отдавая все это на откуп Azure.

17.3.1 Обзор

Хотя, вероятно, в Azure существует много разных способов выполнения такого рода анализа, мы сосредоточимся на использовании службы аналитики Azure Synapse, поскольку она позволяет выполнять бессерверные SQL-запросы с использованием возможности «SQL по запросу». Это означает, что нам нужно платить только за объем данных, которые мы обрабатываем в Azure Synapse, и нам не нужно беспокоиться о расходах и сопровождении используемых вычислительных ресурсов.

Чтобы реализовать наш пример с помощью Synapse, необходимо выполнить следующие шаги.

1. Извлечь рейтинги за определенный месяц из API рейтингов и загрузить их в Azure Blob Storage для дальнейшего анализа.
2. Использовать Azure Synapse, чтобы выполнить SQL-запрос, который ранжирует наши фильмы. Полученный список ранжированных фильмов будет записан обратно в Azure Blob Storage для дальнейшего потребления.

Это дает нам процесс обработки данных, показанный на рис. 17.4. Проницательный читатель заметит, что здесь у нас на один шаг меньше, чем в примере с AWS, где мы использовали Glue и Athena. Это связано с тем, что наш пример с Azure будет напрямую ссылаться на файлы в Azure Blob Storage при выполнении запроса (как будет показано далее), вместо того чтобы сначала индексировать их в каталог (ценой необходимости вручную указывать схему в запросе).

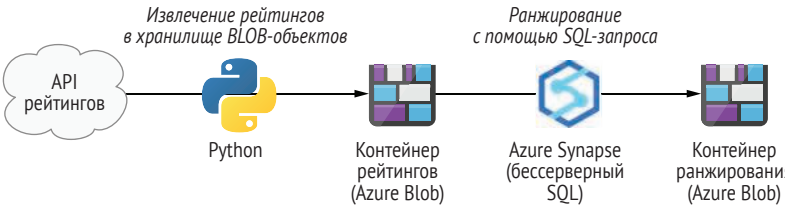


Рис. 17.4 Обзор процесса обработки данных в случае использования бессерверного ранжирования фильмов. Стрелки указывают на преобразования данных, выполняемые в Airflow, отмеченные соответствующим сервисом Azure, который используется для выполнения преобразования данных (где это применимо)

17.3.2 Настройка ресурсов

Перед созданием ОАГ для начала нужно создать необходимые ресурсы. Мы сделаем это на портале Azure (https://portal.azure.com), к которому вы можете получить доступ при наличии соответствующей подписки.

Мы начнем с создания группы ресурсов (рис. 17.5), которая представляет виртуальный контейнер наших ресурсов для данного примера. Здесь мы назвали группу ресурсов «airflow-azure», но, в принципе, имя может быть любое.

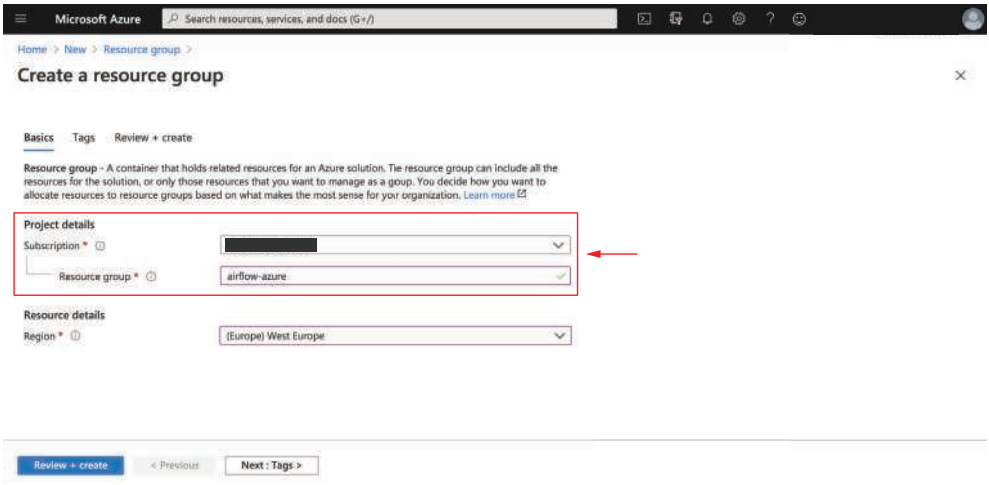


Рис. 17.5 Создание группы ресурсов Azure для хранения ресурсов

После настройки группы ресурсов можно приступить к созданию рабочей области Azure Synapse, которая в настоящее время называется «Azure Synapse Analytics», на портале Azure. Чтобы создать рабочее пространство Synapse, откройте страницу сервиса на портале и щелкните **Create Synapse workspace** (Создать рабочее пространство Synapse). На первой странице мастера создания (рис. 17.6) выберите ранее созданную группу ресурсов и введите имя своего рабочего пространства Synapse. В разделе параметров хранения обязательно создайте новую учетную запись хранилища и файловую систему (выберите любые имена, которые вам нравятся).

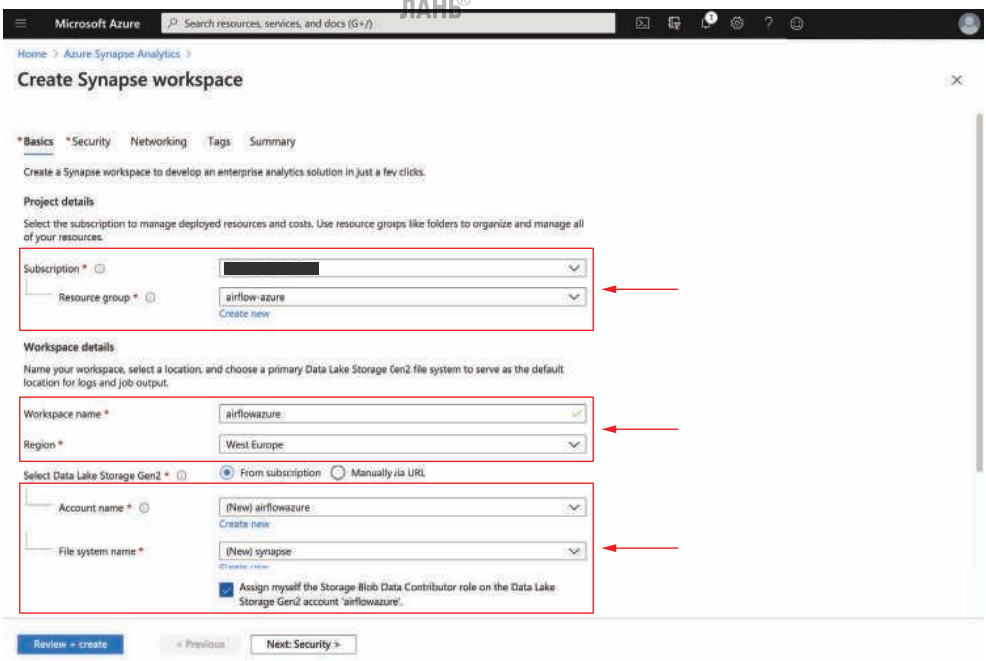


Рис. 17.6 Первая страница мастера создания рабочего пространства Synapse. Обязательно укажите правильную группу ресурсов и имя вашей рабочей области. Чтобы настроить хранилище, нажмите Create new (Создать новое) для учетной записи и файловой системы и введите их имена

На следующей странице мастера (рис. 17.7) у нас есть возможность указать имя пользователя и пароль для учетной записи администратора SQL. Введите, что вам нравится, но запомните данные, которые вы указали (они понадобятся нам при создании ОАГ).

На третьей странице (рис. 17.8) у вас также есть возможность ограничить доступ к сети, сняв флажок рядом с надписью **Allow connections from all IP addresses** (Разрешить соединения со всех IP-адресов), но не забудьте добавить свой личный IP-адрес в исключения брандмауэра, если вы снимете этот флажок. Нажмите **Review+create** (Обзор+создать), чтобы приступить к созданию рабочего пространства.

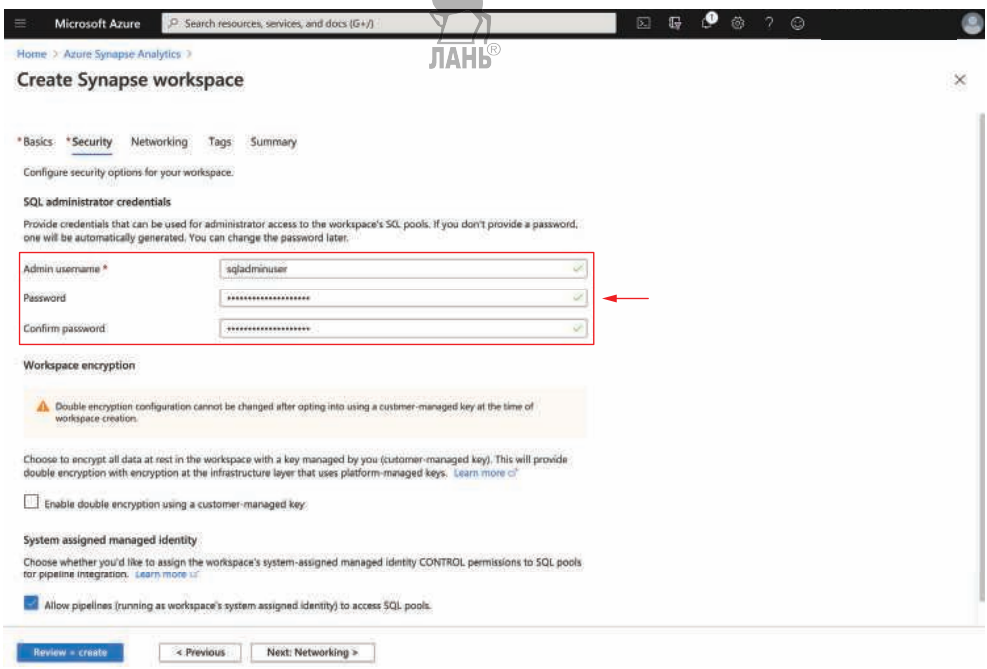


Рис. 17.7 Настройка параметров безопасности для рабочего пространства Synapse

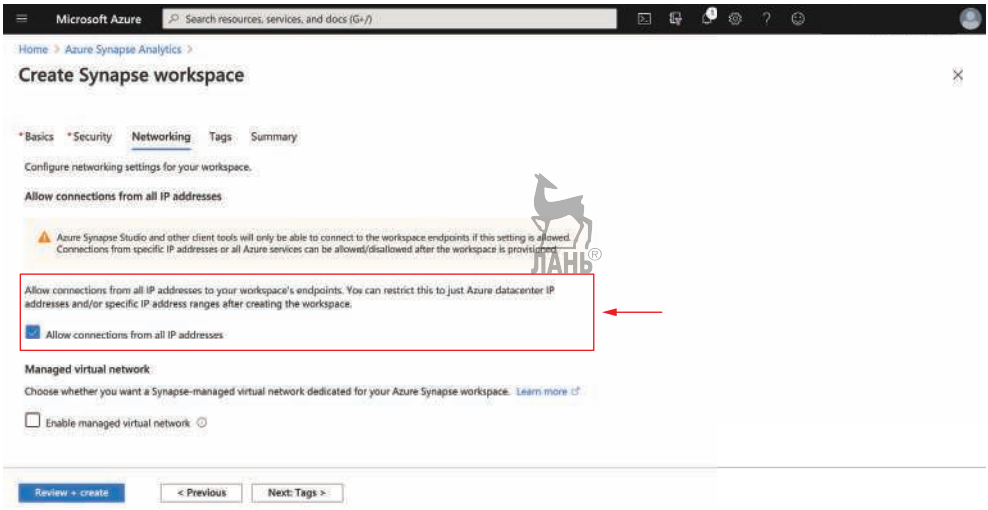


Рис. 17.8 Настройка сетевых параметров для рабочего пространства Synapse

Теперь, когда у нас есть рабочее пространство Synapse и соответствующая учетная запись хранилища, можно приступать к созданию контейнеров (своего рода подпапки), в которых будут храниться наши рейтинги и данные ранжирования в хранилище BLOB-объектов.

Для этого откройте учетную запись хранилища (если вы ее потеряли, то сможете найти ее в своей группе ресурсов), перейдите на страницу **Overview** (Обзор) и нажмите **Containers** (Контейнеры). На странице «Контейнеры» (рис. 17.9) создайте два новых контейнера, ratings и rankings, нажав **+ Container** и введя соответствующее имя контейнера.

Наконец, чтобы убедиться, что мы можем получить доступ к нашей учетной записи из Airflow, нужно получить ключ доступа и секрет. Чтобы получить эти учетные данные, щелкните **Access keys** (Ключи доступа) на панели слева (рис. 17.10).

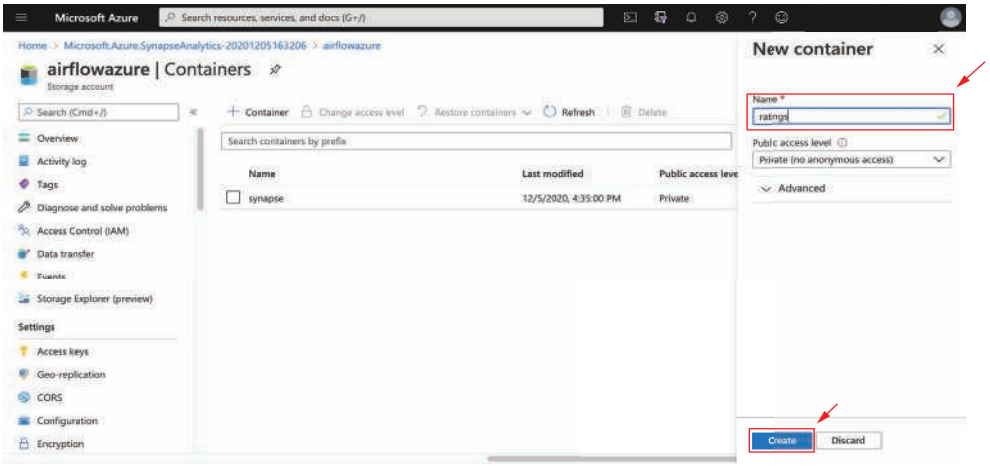


Рис. 17.9 Создание контейнеров BLOB-объектов для хранения рейтингов и данных ранжирования в учетной записи

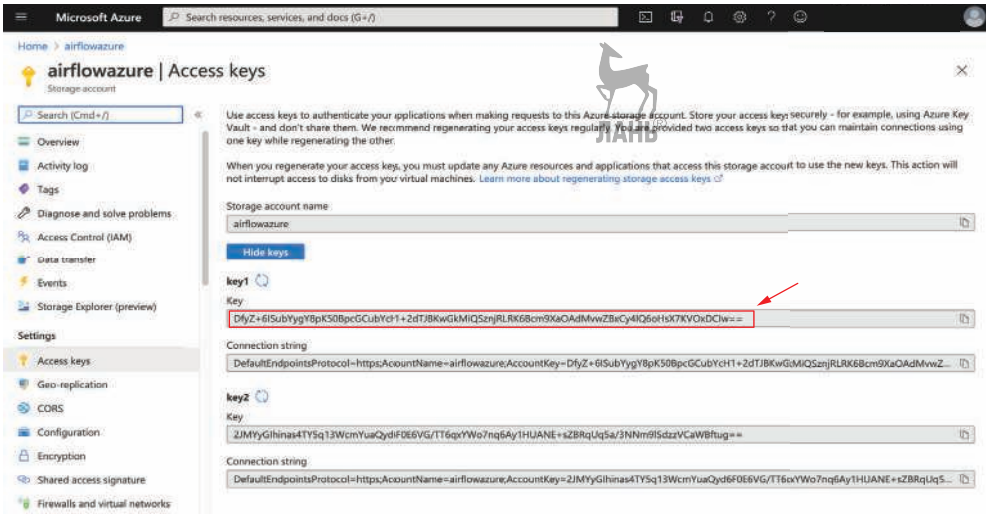


Рис. 17.10 Получение имени учетной записи и ключа для доступа к учетной записи хранилища BLOB-объектов из Airflow

Запишите имя учетной записи хранилища и один из двух ключей, которые мы передадим в качестве деталей подключения в Airflow при реализации ОАГ.

17.3.3 Создание ОАГ

Теперь, когда у нас есть все необходимые ресурсы, можно приступить к созданию ОАГ. Для первого из двух шагов нам нужно реализовать операцию, которая извлекает данные из нашего API рейтингов и отправляет их в Azure Blob Storage. Самый простой способ сделать это – объединить `PythonOperator` с `WasbHook` из пакета поставщика Microsoft/Azure. Такая комбинация позволяет нам извлекать рейтинги с помощью собственных функций, а затем выгружать результаты в Azure Blob Storage с помощью хука.

Листинг 17.1 Загрузка рейтингов с помощью WasbHook (dags/01_azure_usecase.py)

```
import logging
from os import path
import tempfile

from airflow.operators.python import PythonOperator
from airflow.providers.microsoft.azure.hooks.wasb import WasbHook

from custom.hooks import MovielensHook

def _fetch_ratings(api_conn_id, wasb_conn_id, container, **context):
    year = context["execution_date"].year
    month = context["execution_date"].month
    logging.info(f"Fetching ratings for {year}/{month:02d}")
    api_hook = MovielensHook(conn_id=api_conn_id)
    ratings = pd.DataFrame.from_records(
        api_hook.get_ratings_for_month(year=year, month=month),
        columns=["userId", "movieId", "rating", "timestamp"],
    )
    logging.info(f"Fetched {ratings.shape[0]} rows")

    with tempfile.TemporaryDirectory() as tmp_dir:
        tmp_path = path.join(tmp_dir, "ratings.csv")
        ratings.to_csv(tmp_path, index=False)

        logging.info(f"Writing results to
        ➥ {container}/{year}/{month:02d}.csv")
        hook = WasbHook (wasb_conn_id)
        hook.load_file(
            tmp_path,
            container_name=container,
            blob_name=f"{year}/{month:02d}.csv",
        )

fetch_ratings = PythonOperator(
```

Извлекаем рейтинги из API с помощью MovielensHook из главы 8 (код хука доступен в dags/custom/hooks.py)

Записываем рейтинги во временный каталог

Загружаем записанные рейтинги в Azure Blob с помощью WasbHook

```
    task_id="upload_ratings",
    python_callable=_upload_ratings,
    op_kwargs={
        "wasb_conn_id": "my_wasb_conn",
        "container": "ratings"
    },
)
```

`WasbHook` требуется идентификатор подключения, который указывает, какой из них следует использовать для подключения к учетной записи хранилища. Это подключение можно создать в Airflow с использованием учетных данных, полученных в предыдущем разделе, используя имя учетной записи в качестве логина и ключ учетной записи в качестве пароля (рис. 17.11). Код довольно простой: мы извлекаем рейтинги, записываем их во временный файл и загружаем его в контейнер рейтингов с помощью `WasbHook`.

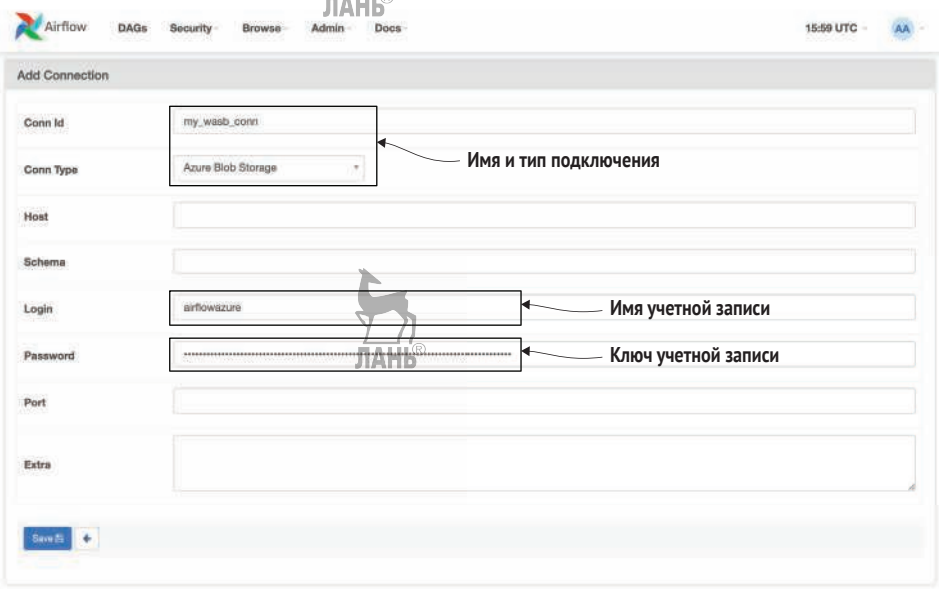


Рис. 17.11 Создание подключения Airflow для учетной записи Azure Blob Storage с помощью имени учетной записи и ключа, полученных на портале Azure

Для второго шага нам понадобится оператор, который может подключиться к Azure Synapse, выполнить запрос, генерирующий наши рейтинги, и записать результаты в контейнер рейтингов в нашей учетной записи хранилища. Хотя ни один обработчик или оператор Airflow не предоставляет такую функциональность, мы можем использовать `OdbcHook` (из пакета поставщика ODBC[1]) для подключения

[1] Можно установить в Airflow 2 с помощью пакета `apache-airflow-provider-odbc` или в Airflow 1.10 с использованием пакета `apache-airflow-backport-sizes-odbc`.

к Synapse через подключение по ODBC. Затем этот хук позволяет нам выполнить запрос и получить результаты, которые потом можно записать в Azure Blob Storage с помощью `WasbHook`.

Фактическое ранжирование будет выполнено SQL-запросом Synapse из листинга 17.2.

Листинг 17.2 SQL-запрос Synapse для ранжирования фильмов (dags/01_azure_usecase.py)

```
RANK_QUERY = """
    SELECT movieId, AVG(rating) as avg_rating, COUNT(*) as num_ratings
    FROM OPENROWSET(
        BULK
            'https://{blob_account_name}.blob.core.windows.net/
            ➥ {blob_container}/*/*.csv',
        FORMAT = 'CSV',
        PARSER_VERSION = '2.0',
        HEADER_ROW = TRUE,
        FIELDTERMINATOR =',',
        ROWTERMINATOR = '\n',
    )
    WITH (
        [userId] bigint,
        [movieId] bigint,
        [rating] float,
        [timestamp] bigint
    ) AS [r]
    WHERE (
        (r.filepath(1) < '{year}') OR
        (r.filepath(1) = '{year}' AND r.filepath(2) <= '{month:02d}')
    )
    GROUP BY movieId
    ORDER BY avg_rating DESC
"""
```

Сообщаем Synapse, чтобы он нашел наш набор данных в формате CSV в нашей учетной записи

Получаем идентификатор фильма, значение рейтинга и дату каждого рейтинга

Определяем схему, которая будет использоваться при чтении данных в формате CSV

Выбираем все рейтинги до даты выполнения на основе имен файлов разделов

Выполняем группировку по идентификатору фильма, чтобы вычислить средний рейтинг

В этом SQL-запросе оператор `OPENROWSET` сообщает Synapse о необходимости загрузить требуемый набор данных из нашей учетной записи хранилища (на которую указывает URL-адрес) и что файлы данных находятся в формате CSV. Далее оператор `WITH` сообщает Synapse, какую схему использовать для данных, считываемых из внешнего набора данных, чтобы мы могли гарантировать, что столбцы данных имеют правильные типы. Наконец, оператор `WHERE` использует разные части путей к файлам, чтобы гарантировать, что мы читаем данные только до текущего месяца, в то время как остальная часть оператора выполняет фактическое ранжирование (используя операторы `SELECT AVG`, `GROUP BY` и `ORDER BY`).

ПРИМЕЧАНИЕ В этом случае у Synapse есть доступ к учетной записи хранилища, потому что мы поместили наши файлы в учетную запись, связанную с рабочим пространством Synapse.

Если вы поместили файлы в другую учетную запись (не связанную напрямую с рабочим пространством), то вам необходимо либо предоставить идентификационный доступ вашего рабочего пространства Synapse к соответствующей учетной записи, либо зарегистрировать ее с соответствующими учетными данными в качестве внешнего хранилища данных в рабочем пространстве.

Этот запрос можно выполнить, используя следующую функцию, которая выполняет запрос с помощью `OdbcHook`[1], преобразует строки в датафрейм Pandas, а затем загружает его содержимое в хранилище BLOB-объектов с помощью `WasbHook`.

Листинг 17.3 Выполнение запроса Synapse с использованием ODBC (dags/01_azure_usecase.py)

```
def _rank_movies(
    odbc_conn_id, wasb_conn_id, ratings_container, rankings_container,
    **context
):
    year = context["execution_date"].year
    month = context["execution_date"].month

    blob_account_name = WasbHook.get_connection(wasb_conn_id).login

    query = RANK_QUERY.format(
        year=year,
        month=month,
        blob_account_name=blob_account_name,
        blob_container=ratings_container,
    )
    logging.info(f"Executing query: {query}")

    odbc_hook = OdbcHook(
        odbc_conn_id,
        driver="ODBC Driver 17 for SQL Server",
    )
    with odbc_hook.get_conn() as conn:
        with conn.cursor() as cursor:
            cursor.execute(query)

            rows = cursor.fetchall()
            colnames = [field[0] for field in cursor.description]
```

Получаем имя нашей учетной записи хранилища BLOB-объектов (такое же, как и у логина учетной записи)

Добавляем параметры запуска в SQL-запрос

Подключаемся к Synapse с помощью хука ODBC

Выполняем запрос и получаем результирующие строки

[1] Обратите внимание, что для этого необходимо установить соответствующие драйверы ODBC. Этот драйвер уже должен быть установлен в нашем образе Docker. Если вы не используете его, то можете найти дополнительные сведения о том, как установить драйверы самостоятельно, на сайте Microsoft. Убедитесь, что вы используете правильную версию для своей операционной системы.

```
    ranking = pd.DataFrame.from_records(rows, columns=colnames)
    logging.info(f"Retrieved {ranking.shape[0]} rows")

    logging.info(f"Writing results to
         ➥ {rankings_container}/{year}/{month:02d}.csv")
    with tempfile.TemporaryDirectory() as tmp_dir:
        tmp_path = path.join(tmp_dir, "ranking.csv")
        ranking.to_csv(tmp_path, index=False)

        wasb_hook = WasbHook (wasb_conn_id)
        wasb_hook.load_file(
            tmp_path,
            container_name=rankings_container,
            blob_name=f"{year}/{month:02d}.csv",
        )
```

Преобразовываем полученные строки в датафрейм Pandas

Записываем результат во временный файл в формате CSV

Загружаем CSV-файл с данными о ранжировании в хранилище BLOB-объектов

Подобно предыдущему шагу, мы можем выполнить эту функцию с помощью `PythonOperator`, передав оператору необходимые ссылки на подключение и пути к контейнерам в качестве аргументов.

Листинг 17.4 Вызов функции ранжирования фильмов (dags/01_azure_usecase.py)

```
rank_movies = PythonOperator(
    task_id="rank_movies",
    python_callable=_rank_movies,
    op_kwargs={
        "odbc_conn_id": "my_odbc_conn",
        "wasb_conn_id": "my_wasb_conn",
        "ratings_container": "ratings",
        "rankings_container": "rankings",
    },
)
```

Конечно, нам по-прежнему нужно предоставить подробную информацию об ODBC-подключении к Airflow (рис. 17.12). URL-адрес хоста для своего экземпляра Synapse можно найти на странице обзора рабочей области Synapse на портале Azure в разделе «Конечная точка SQL по запросу» (по запросу = бессерверный SQL). Для схемы базы данных мы просто будем использовать базу данных по умолчанию (master). Наконец, что касается логина и пароля, мы можем использовать имя пользователя и пароль, которые мы предоставили нашему пользователю с правами администратора при создании рабочего пространства. Конечно, здесь мы используем учетную запись администратора только в демонстрационных целях. При более реалистичных настройках мы рекомендуем создать отдельного пользователя SQL с необходимыми полномочиями и использовать его для подключения к Synapse.

Остается только объединить два этих оператора в OAГ, который мы будем запускать с ежемесячным интервалом для создания ежемесячного рейтинга фильмов.

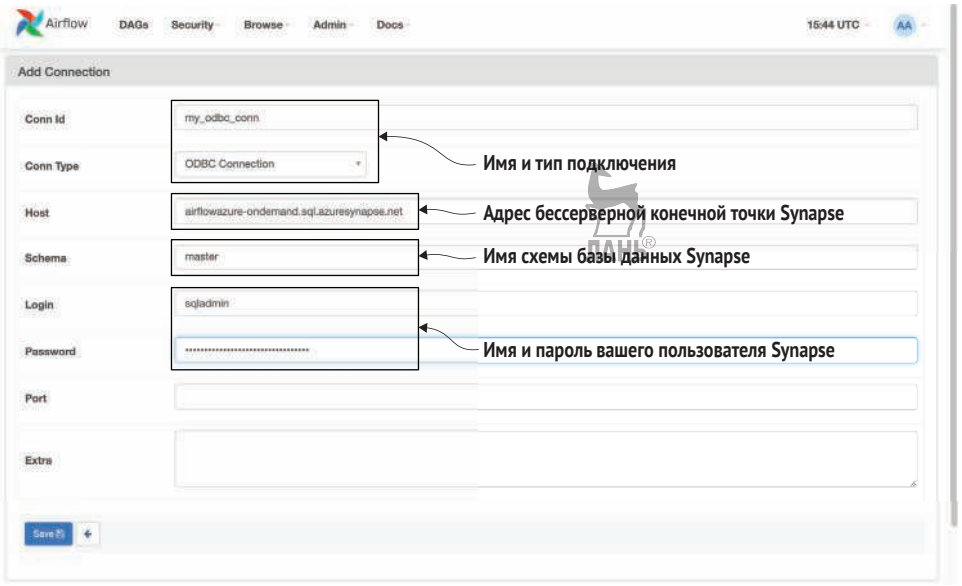


Рис. 17.12 Создание подключения Airflow для ODBC-подключения с Synapse. Соответствующие пользовательские данные должны быть указаны при создании рабочего пространства Synapse

Листинг 17.5 Создание OAГ рекомендательной системы целиком (dags/01_azure_usecase.py)

```
import datetime as dt
import logging
from os import path
import tempfile

import pandas as pd

from airflow import DAG

from airflow.providers.microsoft.azure.hooks.wasb import WasbHook
from airflow.providers.odbc.hooks.odbc import OdbcHook
from airflow.operators.python import PythonOperator

from custom.hooks import MovielensHook

RANK_QUERY = ...

def _fetch_ratings(api_conn_id, wasb_conn_id, container, **context):
    ...

def _rank_movies(odbc_conn_id, wasb_conn_id, ratings_container,
        rankings_container, **context):
    ...
```

```
with DAG(
    dag_id="01_azure_usecase",
    description="DAG demonstrating some Azure hooks and operators.",
    start_date=dt.datetime(year=2019, month=1, day=1),
    end_date=dt.datetime(year=2019, month=3, day=1),
    schedule_interval="@monthly",
    default_args={"depends_on_past": True},
) as dag:
    fetch_ratings = PythonOperator(...)
    rank_movies = PythonOperator(...)
    upload_ratings >> rank_movies
```

Задаем даты начала и окончания в соответствии с набором рейтинговых данных

Используем depends_on_past, чтобы избежать выполнения запросов до того, как будут загружены архивные данные (что может привести к неполным результатам)

Когда все будет готово, мы наконец сможем запустить наш OAГ в Airflow. Если все пойдет хорошо, мы должны увидеть, как наши задачи загружают данные из API рейтингов и обрабатывают их в Synapse (рис. 17.13). Если вы столкнетесь с какими-либо проблемами, убедитесь, что пути к данным и учетные данные для доступа к Azure Blob Storage и Synapse верны.

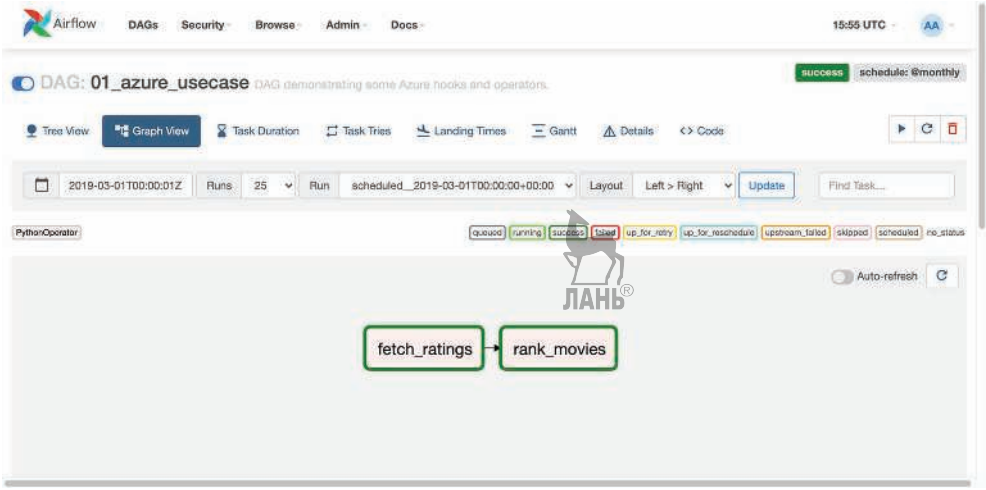


Рис. 17.13 Успешное создание рейтингов фильмов с помощью Azure Synapse в OAГ

17.3.4 Очистка

После того как вы закончите работать с этим примером в Azure Synapse, можете избавиться ото всех созданных ресурсов, удалив группу ресурсов, которую мы создали вначале. Для этого откройте страницу группы ресурсов **Overview** (Обзор) на портале Azure и щелкните **Delete resource group** (Удалить группу ресурсов) (рис. 17.14). Подтвердите удаление.

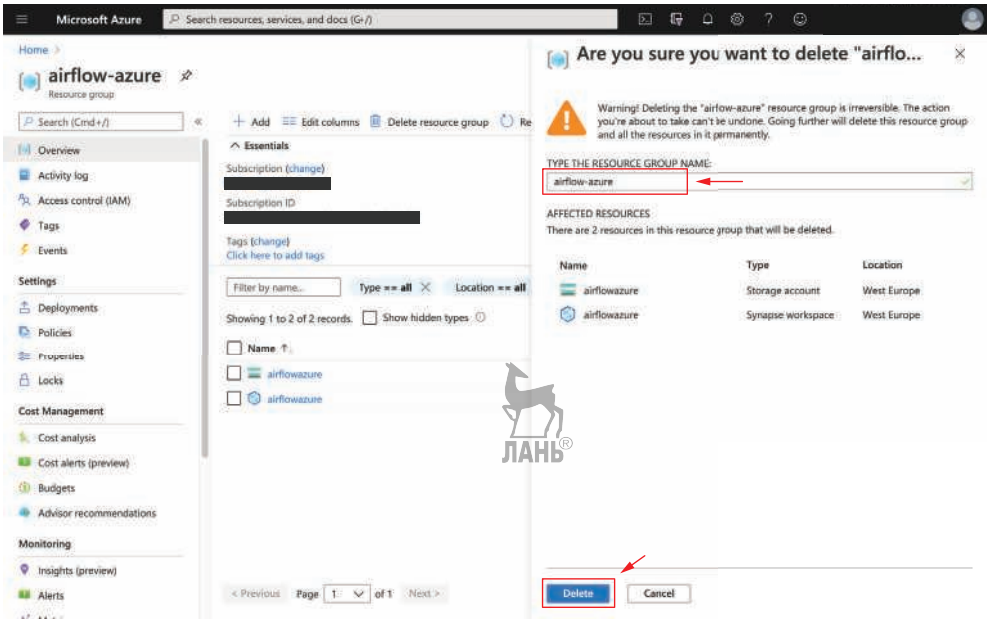


Рис. 17.14 Очистка созданных ресурсов путем удаления соответствующей группы ресурсов

Резюме

- Airflow можно развернуть в Azure с помощью таких сервисов, как ACI и App Service для запуска процессов планировщика и веб-сервера, Azure File и Blob Storages для хранения файлов и Azure SQL Database для базы метаданных Airflow.
- Airflow предоставляет ряд хуков и операторов специально для Azure, которые позволяют интегрировать различные сервисы с облачной платформой Azure.
- К некоторым сервисам Azure можно получить доступ с помощью универсальных хуков, таких как `ODBChook`, если они соответствуют этим стандартизованным протоколам.
- Использование хуков и операторов для Azure обычно также требует настройки необходимых ресурсов и полномочий на доступ в Azure и Airflow, чтобы Airflow мог выполнять необходимые операции.

18 Airflow в GCP

Эта глава рассказывает о:

- проектировании стратегии развертывания GCP;
- хуках и операторах, предназначенных для GCP, как их использовать.

Последний крупный поставщик облачных услуг, Google Cloud Platform (GCP), фактически является лучшей поддерживаемой облачной платформой с точки зрения количества хуков и операторов. Практически всеми сервисами Google можно управлять с помощью Airflow. В этой главе мы подробно разберем настройку Airflow в GCP (раздел 18.1), операторы и хуки для сервисов GCP (раздел 18.2), а также вариант использования, который мы продемонстрировали для AWS и Azure, но применительно к GCP (раздел 18.3).

Также нужно отметить, что в GCP есть управляемый сервис Airflow «Cloud Composer», о котором более подробно говорится в разделе 15.3.2. В этой главе рассматривается, как самостоятельно настроить Airflow для GCP, а не для Cloud Composer.

18.1 Развертывание Airflow в GCP

GCP предоставляет различные сервисы для запуска программного обеспечения. Здесь не существует универсального подхода, вот поче-

му Google (и все другие облачные провайдеры) предоставляют разные сервисы для запуска программного обеспечения.

18.1.1 Выбор сервисов

Эти сервисы можно отобразить в виде шкалы от полностью самоуправляемых с максимальной гибкостью до полностью управляемых GCP без необходимости сопровождения (рис. 18.1).

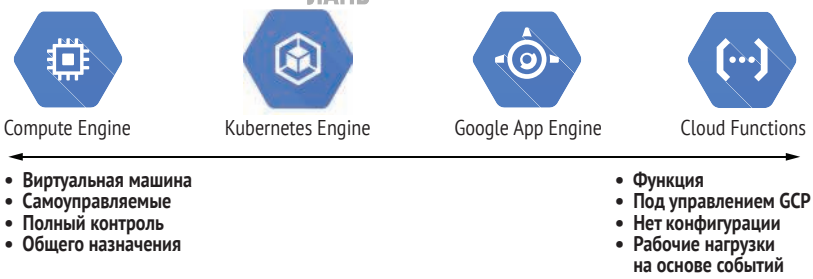


Рис. 18.1 Обзор сервисов, доступных в Google Cloud Platform

С левой стороны у нас идет Compute Engine, который предоставляет вам виртуальную машину для запуска любого программного обеспечения, которое вы пожелаете. Compute Engine дает вам полную свободу и контроль, что может быть положительным моментом, но также требует, чтобы вы самостоятельно управляли виртуальной машиной и настраивали ее. Например, если трафик к сервису, который вы используете на Compute Engine, растет, то можно воспользоваться вертикальным масштабированием, создав новую виртуальную машину с крупным типом экземпляра, или масштабировать по горизонтали, настраивая политику автомасштабирования для создания большего количества одинаковых экземпляров.

С правой стороны у нас идет Cloud Functions, которому вы можете предоставить функцию на одном из поддерживаемых языков (на момент написания книги это Node.js, Python, Go и Java), например функцию Python, которая возвращает текущее время в заданном часовом поясе. Итак, если вы вызываете функцию с аргументом, CEST, функция вернет время для часового пояса CEST. Функции обрабатывают небольшие рабочие нагрузки и работают на основе событий. Google управляет вашей функцией (то есть базовой инфраструктурой) и автоматически масштабирует количество развернутых функций. Если у вашей функции запрашивается высокая нагрузка, масштабирование происходит автоматически. Google занимается журналированием, мониторингом и тому подобным; нужно только предоставить функцию. Если ваш вариант использования соответствует характеристикам функции, это может значительно повысить вашу производительность.

Настроить Airflow непросто из-за общего хранилища, которое требуется для хранения и обмена файлами DAG (в основном применяется

при запуске `CeleryExecutor` или `KubernetesExecutor`). Вот что ограничивает наши возможности в GCP:

- Cloud Functions обслуживает событийно-ориентированные функции без сохранения состояния. Airflow таковым не является, и поэтому его нельзя развернуть в Cloud Functions;
- запуск Airflow в App Engine может быть технически возможен, но с некоторыми оговорками: App Engine ожидает наличия одного контейнера Docker, в то время как минимальная установка Airflow уже поделена между веб-сервером и процессом планировщика. Это создает проблему: обычно приложения, которые предоставляют доступ к чему-либо (например, интерфейс или REST API), запускаются в App Engine, который автоматически масштабируется в зависимости от нагрузки. Airflow не подходит для этой модели, поскольку по умолчанию это распределенное приложение. Веб-сервер может быть хорошим кандидатом для работы в GAE. Планировщик Airflow не соответствует модели App Engine, поэтому остается два варианта: GCE и GKE. Kubernetes уже подробно обсуждался в главе 10;
- Kubernetes Engine хорошо подходит для Airflow. Доступны диаграммы Helm для развертывания Airflow в Kubernetes, а также предоставляются абстракции для монтирования файловых систем, совместно используемых несколькими подами;
- Compute Engine дает вам полную свободу запуска и настройки вашего экземпляра. Можно выделить две разновидности Compute Engine: виртуальную машину на базе Linux и виртуальную машину с оптимизированной для контейнеров ОС (COS). Система COS идеально подходит для запуска контейнеров Docker и поэтому кажется привлекательной с точки зрения развертывания, но, к сожалению, создает проблему в сочетании с Airflow. Airflow требуется файловая система для хранения ОАГ (потенциально совместно используемая несколькими машинами), для которой хранилище, доступное через NFS, является распространенным решением. Однако COS не поставляется с библиотеками NFS. Хотя технически их можно установить, это непростая задача, поэтому проще переключиться на виртуальную машину на базе Linux, что дает полный контроль.

Для общей файловой системы есть два (из множества) варианта для GCP:

- Google Cloud Filestore (сервис NAS, управляемый GCP);
- GCS, монтируемое с FUSE.

Общие файловые системы долгое время были проблемой, и у каждой из них есть свои плюсы и минусы. По возможности, мы предпочитаем избегать файловых систем FUSE, поскольку они применяют интерфейс, подобный файловой системе, поверх того, что никогда не планировалось как файловая система (например, GCS – это хранилище объектов). Это приводит к проблемам, связанным с низкой про-

изводительностью и согласованностью, особенно при использовании несколькими клиентами.

Для других компонентов Airflow количество вариантов меньше и, следовательно, проще. Для базы метаданных GCP предоставляет сервис Cloud SQL, который подходит как для MySQL, так и для PostgreSQL. Для хранения журналов мы будем применять Google Cloud Storage (GCS), сервис хранения объектов GCP.

При запуске в GCP развертывание в Google Kubernetes Engine (GKE), вероятно, является самым простым подходом (рис. 18.2). GKE – это управляемый Google сервис Kubernetes, который обеспечивает простой способ развертывания и управления контейнеризированным программным обеспечением. Другой очевидный вариант – запускать все на виртуальных машинах Compute Engine на базе Linux: для установки и запуска требуется больше работы и времени, так как вам придется настраивать все самостоятельно. Google уже предоставляет управляемый сервис Airflow под названием Composer, но мы продемонстрируем, как развернуть Airflow на GKE и как интегрировать его с другими сервисами GCP.

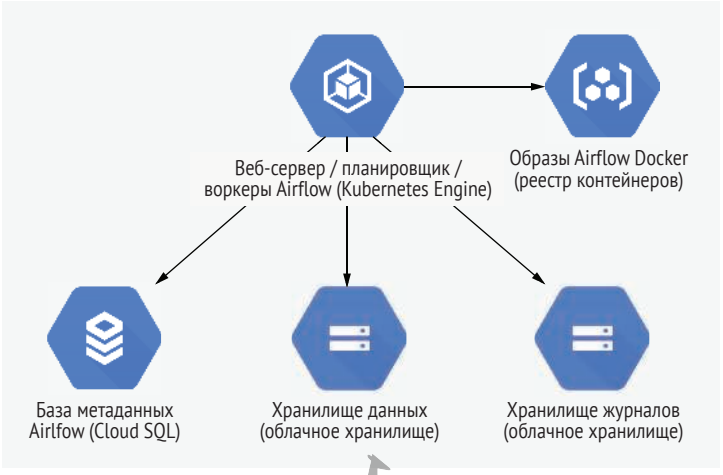


Рис. 18.2 Сопоставление компонентов Airflow с GCP в развертывании Airflow на базе Kubernetes

18.1.2 Развертывание в GKE с помощью Helm

Начнем с запуска GKE. В этом разделе мы хотим предоставить основные команды для запуска Airflow, поэтому пропустим разные детали, которые часто требуются при рабочей настройке, такие как запрет на предоставление доступа к сервисам на общедоступных IP-адресах. Команда из листинга 18.1 создает кластер GKE с общедоступной конечной точкой.

Работа с интерфейсом командной строки gcloud

Чтобы указать Google использовать конкретный проект, можно настроить значение по умолчанию:

```
gcloud config set project [my-project-id]
```

или добавить параметр к каждой команде, например:

```
gcloud compute instances list --project [my-project-id]
```

Для показанных здесь команд gcloud параметр `--project` не отображается. Мы предполагаем, что вы устанавливаете значение по умолчанию или добавляете к команде этот параметр.

Листинг 18.1 Команда gcloud для создания кластера GKE

```
gcloud container clusters create my-airflow-cluster \
--machine-type n1-standard-4 \
--num-nodes 1 \
--region "europe-west4"
```

Затем используйте команду из следующего листинга, чтобы подключить клиента kubectl к кластеру.

Листинг 18.2 Команда gcloud для настройки конфигурационной записи kubectl

```
gcloud container clusters get-credentials my-airflow-cluster \
--region europe-west4
```

В этом кластере мы развернем полностью работоспособную установку Airflow с помощью Helm, менеджера пакетов для Kubernetes. На момент написания книги диаграмма Helm была включена в репозиторий Airflow на GitHub, но не выпущена через официальный канал. Поэтому нужно скачать ее, чтобы установить. Обратитесь к документации по Airflow для получения самой последней информации.

Листинг 18.3 Скачивание и установка диаграммы Helm

```
$ curl -OL https://github.com/apache/airflow/archive/master.zip
$ unzip master.zip
$ kubectl create namespace airflow
$ helm dep update ./airflow-master/chart
$ helm install airflow ./airflow-master/chart -namespace airflow

NAME: airflow
LAST DEPLOYED: Wed Jul 22 20:40:44 2020
NAMESPACE: airflow
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

Скачиваем исходный код Airflow

Устанавливаем диаграмму, это займет некоторое время

Скачиваем указанные версии зависимых диаграмм Helm

Создаем пространство имен Kubernetes для Airflow

```
NOTES:
Thank you for installing Airflow!

Your release is named airflow.

➥ You can now access your dashboard(s) by executing the following command(s)
    and visiting the corresponding port at localhost in your browser:

➥ Airflow dashboard:        kubectl port-forward svc/airflow-webserver
    8080:8080 --namespace airflow
```

Диаграмма Helm из листинга 18.3 обеспечивает полную установку Airflow, работающую в Kubernetes. Это означает, что все работает внутри Kubernetes. Многие части можно настроить, но по умолчанию запускается `KubernetesExecutor` с базой метаданных Postgres, OAГ встроены в образы Docker, а имя пользователя / пароль веб-сервера – «admin»/«admin» (который вы, вероятно, захотите изменить). Веб-сервер работает в качестве сервиса Kubernetes, ClusterIP, предоставляющего вам сервис внутри вашего кластера, к которому другие приложения могут получить доступ, но он не доступен извне. Чтобы получить к нему доступ, мы можем перенаправить его на под.

Листинг 18.4 Перенаправление порта на веб-сервер Airflow

```
kubectl port-forward svc/airflow-webserver 8080:8080 --namespace airflow
```

Так мы делаем веб-сервер доступным по адресу http://localhost:8080.

OAГ можно добавить двумя способами.

1. Метод развертывания по умолчанию с помощью диаграммы Helm заключается в создании OAГ вместе с образом Airflow Docker. Чтобы создать новый образ и обновить образ Docker, используйте следующий код:

   Листинг 18.5 Обновление развернутого образа Airflow с помощью Helm

   ```
   helm upgrade airflow ./airflow-master/chart \
     --set images.airflow.repository=yourcompany/airflow \
     --set images.airflow.tag=1234abc
   ```

2. Или можно указать путь к репозиторию Git и настроить sidecar-контейнер, используя команду git-sync (https://github.com/kubernetes/git-sync), чтобы извлекать код из репозитория Git каждые X секунд (по умолчанию 60).

   Листинг 18.6 Настройка sidecar-контейнера с диаграммой Helm

   ```
   helm upgrade airflow ./airflow-master/chart \
     --set dags.persistence.enabled=false \
     --set dags.gitSync.enabled=true
   ```

Все подробности и параметры конфигурации *см.* в документации Airflow.

18.1.3 Интеграция с сервисами Google

После запуска Airflow на GKE можно увидеть, как более эффективно использовать управляемые сервисы Google, чтобы вам не приходилось самостоятельно управлять приложениями в Kubernetes. Мы продемонстрируем, как создать балансировщик нагрузки GCP для внешнего доступа к веб-серверу. Для этого нужно изменить тип сервиса веб-сервера. По умолчанию это ClusterIP.

ClusterIP может направлять запросы к правильному поду, но она не предоставляет внешней конечной точки для подключения, требуя, чтобы пользователь настроил прокси-сервер для подключения к сервису (рис. 18.3, слева). Это неудобно для пользователя, поэтому нам нужен другой механизм, к которому пользователь может подключаться напрямую, без какой-либо настройки. Для этого есть несколько вариантов, и один из них – создать сервис Kubernetes, LoadBalancer (рис. 18.3, справа). Данный тип сервиса применяется в файле chart/values.yaml в разделе «веб-сервер». Измените тип сервиса с ClusterIP на LoadBalancer и примените измененную диаграмму Helm.

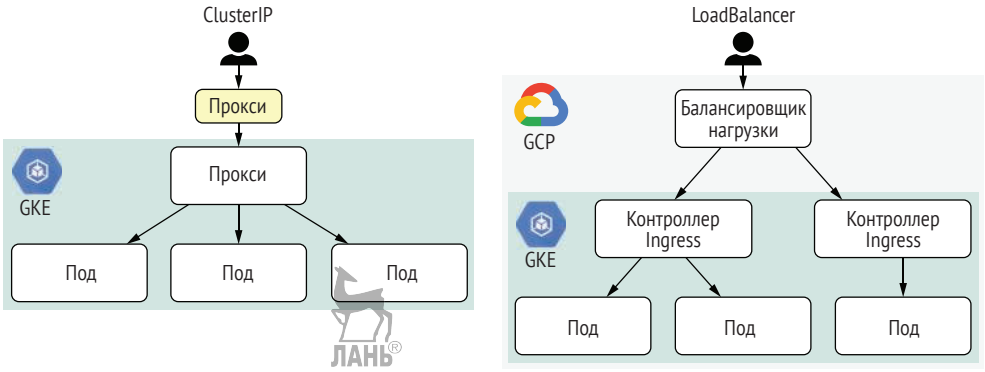


Рис. 18.3 Различные шаблоны доступа для сервисов, работающих в Kubernetes

Листинг 18.7 Установка новой версии диаграммы Helm

```
helm upgrade --install airflow ./airflow-master/chart --namespace airflow
```

GKE получает запрос на применение изменений в кластере GKE и замечает изменение с ClusterIP на LoadBalancer. GKE интегрируется с различными сервисами GCP, и один из них – балансировщик нагрузки. При создании Kubernetes LoadBalancer в GKE GCP создаст балансировщик нагрузки в меню сетевых служб, обслуживая трафик в вашем кластере GKE (рис. 18.4).

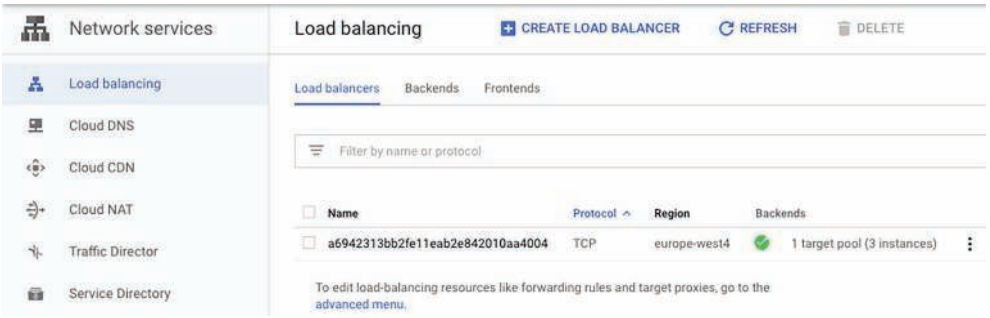


Рис. 18.4 Создание балансировщика нагрузки в консоли GCP

Выбор только что созданного балансировщика нагрузки покажет адрес, который теперь доступен извне (рис. 18.5). На этом скриншоте веб-сервер Airflow доступен по адресу http://34.90.59.14:8080.

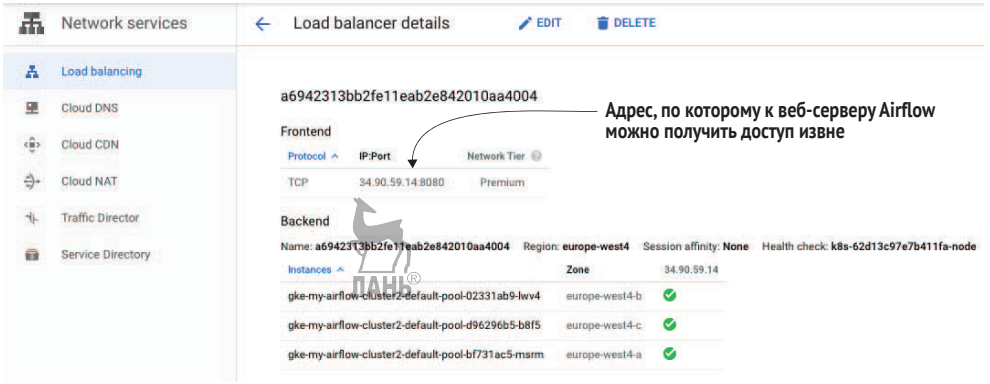


Рис. 18.5 Определение внешнего адреса балансировщика нагрузки в консоли GCP

Другие компоненты установки Airlfow Helm также могут быть переданы на откуп сервисам GCP; однако необходимая работа более сложна:

- база данных Postgres может работать в Cloud SQL;
- мы можем запускать собственные образы из Google Cloud Repository (GCR);
- мы можем настроить удаленное журналирование в GCS (описано в разделе 12.3.4).

18.1.4 Проектирование сети

Структура сети – это персональный выбор, и количество вариантов здесь безгранично. Например, нормально ли пропускать трафик через интернет и использовать внешние IP-адреса, или параметры безопасности требуют, чтобы мы маршрутизировали весь трафик внутри GCP и использовали только внутренние IP-адреса? Мы хотим предоставить макет сети, который поможет вам начать работу. Он

подходит не всем (да это и невозможно), но может служить отправной точкой. Использование упомянутых компонентов дает результат, показанный на рис. 18.6.

Как уже упоминалось, Airflow установлен на GKE. Веб-сервер может быть открыт для внешнего мира через балансировщик нагрузки. Cloud Storage – это глобально доступный сервис, который не ограничивается виртуальным частным облаком. Однако GCP предоставляет сервис под названием VPC Service Controls (VPC SC), чтобы ограничить обмен данными с выбранными сервисами (включая Cloud Storage), к которым можно получить доступ только из вашего VPC. База данных Cloud SQL, обслуживающая базу метаданных Airflow, не может работать в той же подсети, что и ваши службы. Google создает для вас полностью управляемую базу данных по своему периметру. Таким образом, подключение к базе данных должно быть создано либо через общий доступ в интернет, либо путем пиринга собственного виртуального частного облака с виртуальным облаком Google.

18.1.5 Масштабирование с помощью CeleryExecutor

Celery использует брокера сообщений для распределения задач между воркерами. GCP предлагает сервис обмена сообщениями Pub/Sub; однако он не поддерживается Celery. Таким образом, вы ограничены использованием инструментов с открытым исходным кодом, которые поддерживает Celery: RabbitMQ или Redis. С архитектурной точки зрения это не изменит то, что изображено на рис. 18.6, поскольку эти сервисы могут работать вместе с контейнерами Airflow в GKE.

По умолчанию Airflow Helm запускается с `KubernetesExecutor`. К счастью, настроить `CeleryExecutor` очень просто. Необходимые компоненты (например, Redis) автоматически устанавливаются одной командой.

Листинг 18.8 Настройка CeleryExecutor

```
$ helm upgrade airflow ./airflow-master/chart --set executor=CeleryExecutor

Release "airflow" has been upgraded. Happy Helming!
...

➥ You can now access your dashboard(s) by executing the following command(s)
    and visiting the corresponding port at localhost in your browser:

➥ Airflow dashboard:        kubectl port-forward svc/airflow-webserver
    8080:8080 --namespace airflow
➥ Flower dashboard:         kubectl port-forward svc/airflow-flower
   5555:5555 --namespace airflow
```

Дашборд Flower установлен для мониторинга

Количество воркеров Celery можно контролировать вручную с помощью свойства Helm, `worker.replicas`, для которого по умолчанию задано значение 1. Оно не масштабируется автоматически. Однако

для этого есть решение, а именно Kubernetes Event-Driven Autoscaling, более известное как KEDA. На основе определенного условия KEDA автоматически вертикально масштабирует количество контейнеров (в Kubernetes это известно как HPA, или горизонтальное автомасштабирование пода), например рабочую нагрузку для вашей настройки Airflow. Диаграмма Helm предоставляет настройки для активации автомасштабирования KEDA и определяет нагрузку для Airflow и соответствующих воркеров в виде следующего запроса к базе метаданных Airflow:

```
CEIL((RUNNING + QUEUED tasks) / 16)
```

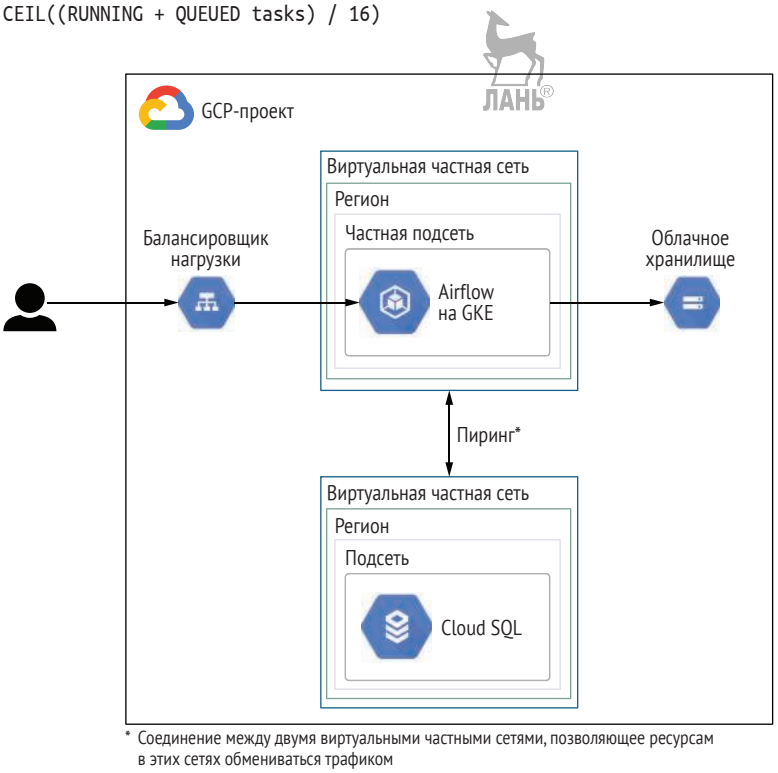


Рис. 18.6 Пример схемы сети GCP с Airflow, работающим в GKE, Cloud SQL для базы метаданных и веб-сервер Airflow, доступный через балансировщик нагрузки

Например, предположим, что у нас есть 26 запущенных задач и 11 задач в очереди: CEIL ((26 + 11) / 16) = 3 воркера. По умолчанию KEDA выполняет запрос к базе данных каждые 30 секунд и изменяет количество воркеров, если оно отличается от текущего числа, активируя автомасштабирование воркеров Celery, как показано на рис. 18.7.

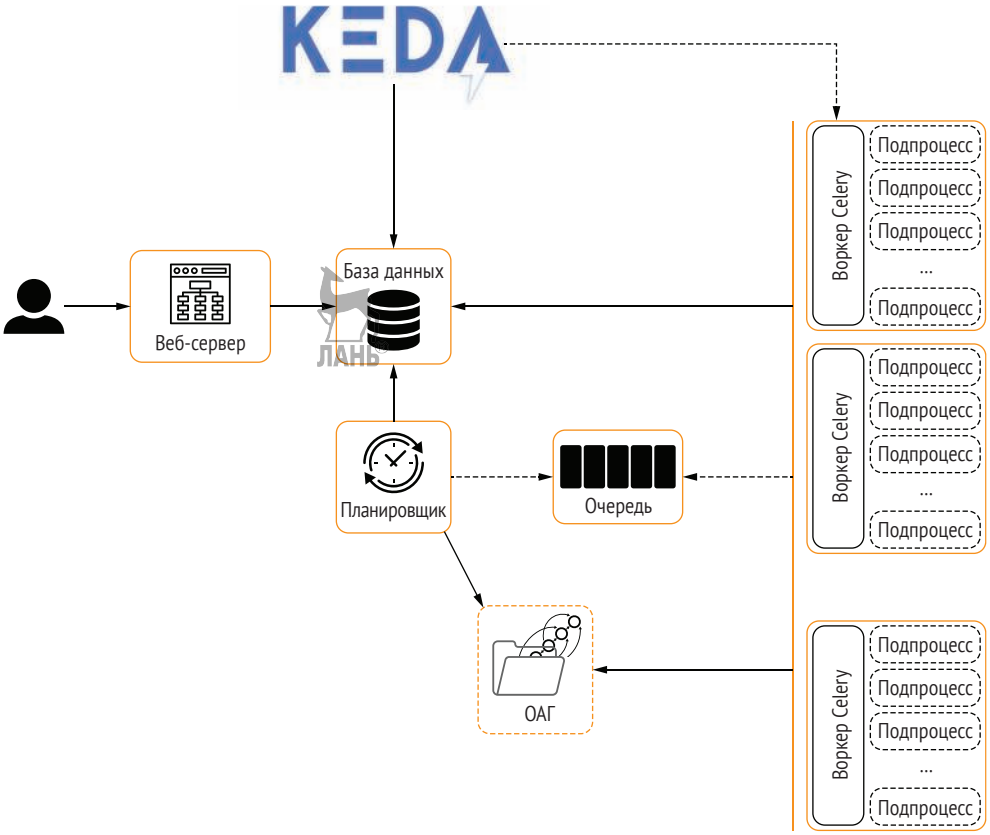


Рис. 18.7 Airlfow, запускающий `CeleryExecutor` с KEDA, автоматически масштабирует количество воркеров Celery в зависимости от рабочей нагрузки. Такая настройка работает только при установке на Kubernetes

Активируйте автомасштабирование с помощью диаграммы Helm.

Листинг 18.9 Настройка CeleryExecutor и автомасштабирования

```
helm repo add kedacore https://kedacore.github.io/charts

helm repo update

kubectl create namespace keda

helm install \
  --set image.keda=docker.io/kedacore/keda:1.2.0 \
  ➥ --set image.metricsAdapter=docker.io/kedacore/keda-metrics-adapter:1.2.0 \
  --namespace keda \
  keda kedacore/keda
```

```
helm upgrade airflow ./airflow-master/chart \
  --set executor=CeleryExecutor \
  --set workers.keda.enabled=true \
  --set workers.persistence.enabled=false
```

KEDA не поддерживает Kubernetes StatefulSets, поэтому его необходимо отключить

Так почему бы вы отдали предпочтение настройке Celery и KEDA, а не `KubernetesExecutor`? Хотя оба они могут масштабироваться по горизонтали, установка Celery и KEDA более желательна с точки зрения производительности, поскольку она поддерживает работу определенного числа воркеров Celery, которые незамедлительно обрабатывают новые задачи, поступающие в очередь. Однако `KubernetesExecutor` должен создать новый под Airflow для выполнения данной задачи, что приведет к расходам при запуске для каждой задачи.

Все упомянутые настройки можно изменить; обратитесь к документации, чтобы получить всю подробную информацию. На момент написания книги настройка с KEDA считалась экспериментальной; *см.* последнюю информацию в документации по Airflow.

18.2 Хуки и операторы, предназначенные для GCP

Многие сервисы GCP используют операторы Airflow, хуки, сенсоры и т. д., что обеспечивает гораздо больший охват, чем в случае с AWS и Azure. Из-за их огромного количества мы рекомендуем вам обратиться к пакету поставщиков Google/Cloud `apache-airflow-providers-google` для получения полного обзора доступных хуков и операторов.

Связанные с Google хуки наследуют не от `airflow.hooks.BaseHook`, а от класса `airflow.providers.google.common.hooks.base_google.GoogleBaseHook`. Этот базовый класс предоставляет тот же механизм аутентификации для Google REST API, поэтому всем производным хукам и операторам, использующим его, не требуется реализовывать аутентификацию.

Поддерживаются три метода аутентификации:

1. путем задания для переменной окружения `GOOGLE_APPLICATION_CREDENTIALS` (за пределами Airflow) значения в виде пути к файлу ключей в формате JSON;
2. путем задания значений для полей «Идентификатор проекта» и «Путь к ключевому файлу» в подключении Airflow типа Google Cloud Platform;
3. предоставляя содержимое файла ключей в формате JSON для подключения Airflow типа «Google Cloud Platform» в поле «Keyfile JSON».

После выполнения любого оператора, относящегося к GCP, в GCP будет отправлен запрос, требующий аутентификации. Эта аутенти-

фикация может быть представлена в GCP сервисным аккаунтом, который может использоваться приложением (например, Airflow), а не человеком. Airflow требует одного из трех вариантов аутентификации GCP с помощью данного аккаунта. Например, предположим, что мы хотим разрешить Airflow запускать задания BigQuery. Создадим сервисный аккаунт, предоставляющий такие полномочия.

Сначала в консоли GCP перейдите в раздел **Service Accounts** (рис. 18.8).

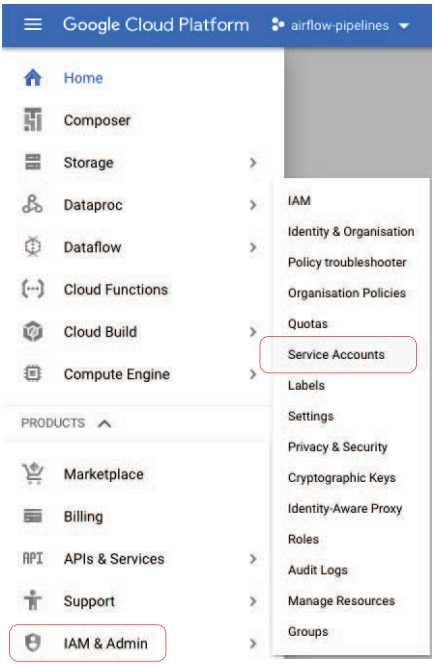


Рис. 18.8 Создание сервисной учетной записи в консоли GCP

Щелкните **Create Service Account** (Создать учетную запись сервиса) и введите имя, например «run-bigquery-jobs». Затем укажите роль, BigQuery Job User, которая имеет полномочия на запуск заданий BigQuery (рис. 18.9).

После добавления роли нажмите **Continue** (Продолжить), чтобы перейти к следующему экрану, где можно создать ключ. Нажмите **Create Key** (Создать ключ), после чего вам будет предложено два варианта загрузки файла ключа.

JSON – рекомендуемый метод, поэтому выберите его и нажмите **Create** (Создать), чтобы скачать файл в формате JSON, содержащий ключ (рис. 18.10).

Только что загруженный файл в формате JSON содержит несколько значений, которые можно использовать для аутентификации с помощью GCP.

Рис. 18.9 Добавление соответствующих полномочий BigQuery в вашу учетную запись

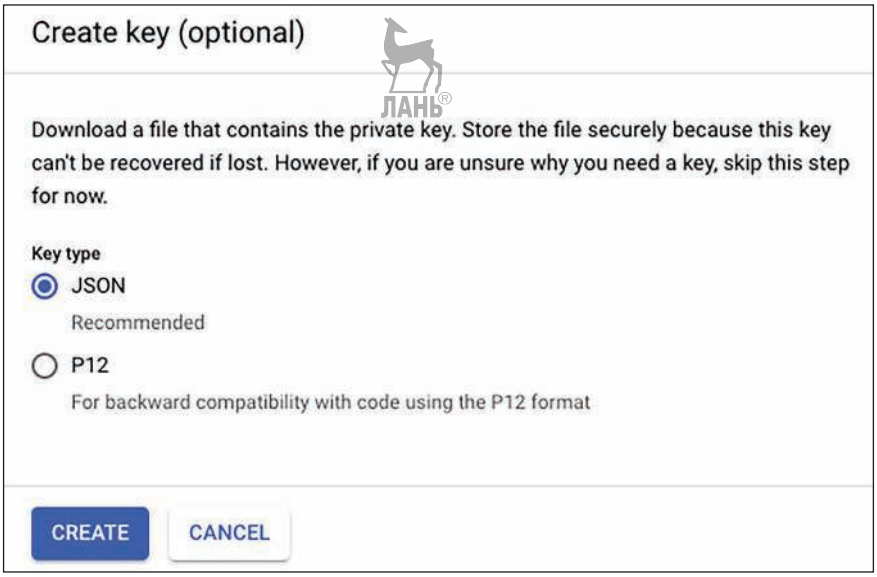


Рис. 18.10 Создание и загрузка ключа доступа

Листинг 18.10 Содержимое JSON-ключа учетной записи сервиса

```
$ cat airflow-pipelines-4aa1b2353bca.json
{
  "type": "service_account",
  "project_id": "airflow-pipelines",
  "private_key_id": "4aa1b2353bca412363bfa85f95de6ad488e6f4c7",
  ➥ "private_key": "-----BEGIN PRIVATE KEY-----\nMIIz...LaY=\n-----END
     PRIVATE KEY-----\n",
  "client_email": "run-bigquery-jobs@airflow-pipelines.iam...com",
  "client_id": "936502912366591303469",
  "auth_uri": "https://accounts.google.com/o/oauth2/auth",
  "token_uri": "https://oauth2.googleapis.com/token",
```

```
  "auth_provider_x509_cert_url": "https://www.googleapis.com/oauth2/...",
  "client_x509_cert_url": "https://...iam.gserviceaccount.com"
}
```

Храните этот файл в надежном и безопасном месте. Любой, у кого есть доступ к нему, может пройти аутентификацию в GCP и использовать предоставленные полномочия. Предоставим его Airflow, чтобы можно было запустить задание BigQuery. Имея три варианта, можно предоставить ключ тремя способами.

1. Задав значение для переменной окружения, `GOOGLE_APPLICATION_CREDENTIALS`.

Листинг 18.11 Задаем учетные данные Google с использованием переменной окружения

```
export GOOGLE_APPLICATION_CREDENTIALS=/path/to/key.json
```

Обратите внимание, что так мы задаем учетные данные глобально, и все приложения, проходящие аутентификацию с Google, прочитают этот ключ в формате JSON.

2. Настроив подключение Airflow (рис. 18.11):

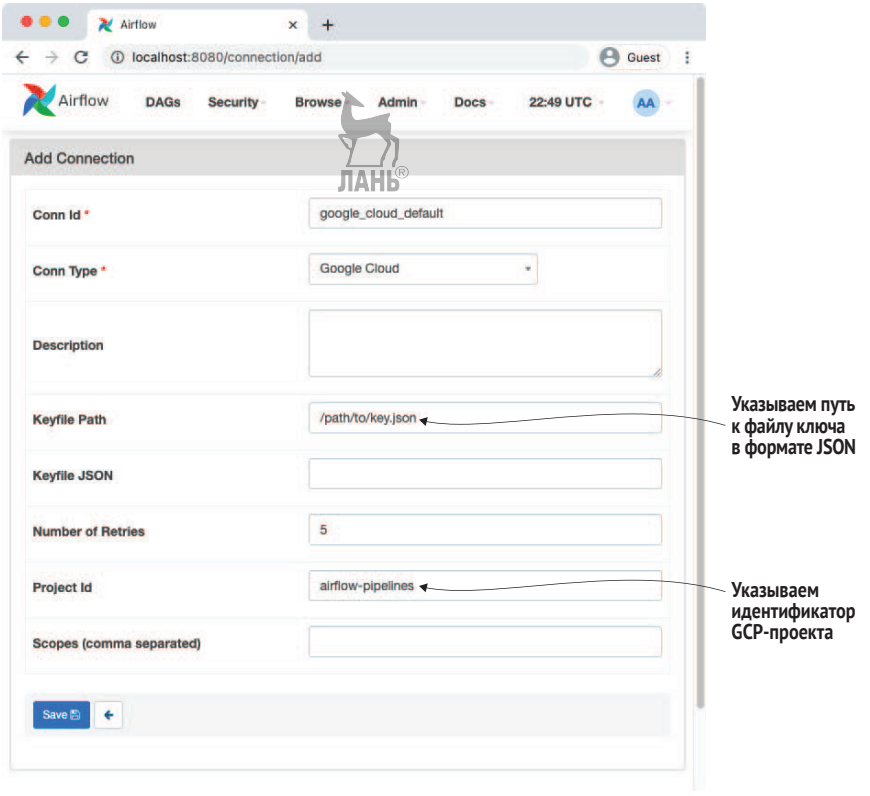


Рис. 18.11 Создание подключения Airflow с использованием файла ключа доступа

3 Предоставив содержимое файла в формате JSON для подключения Airflow (рис. 18.12):

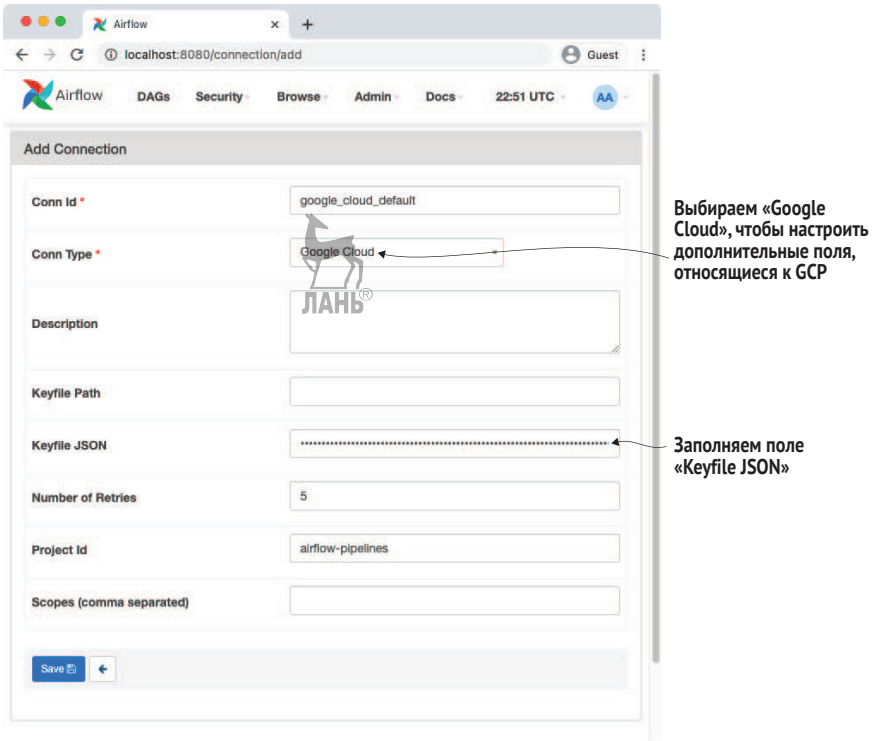


Рис. 18.12 Создание подключения Airflow с использованием ключа доступа в формате JSON

Все три варианта пройдут аутентификацию. Обратите внимание, что JSON-ключ относится к одному проекту. При использовании первого варианта вы установите ключ в вашей системе глобально: все приложения, подключенные к Google, будут проходить аутентификацию с помощью этого ключа и использовать одни и те же полномочия. Второй вариант также указывает на расположение файла ключа в формате JSON, но из подключения Airflow. Таким образом, вы можете предоставить разные идентификаторы подключения для разных задач, используя разные наборы полномочий между задачами, а также, возможно, подключение к разным проектам GCP. Разница между вторым и третьим вариантами заключается в том, что в третьем варианте ваш ключ хранится *только* в Airflow, а не в виде файла в вашей файловой системе; это может быть желательным вариантом, но если в вашей системе есть и другие приложения, использующие тот же ключ, выберите второй вариант.

18.3 Пример использования: бессерверный рейтинг фильмов в GCP

Вернемся к примеру, который ранее был применен к AWS и Azure. Как бы это работало в GCP? Многие облачные сервисы можно сопоставить друг с другом (табл. 18.1).

Таблица 18.1 Сравнение похожих сервисов в AWS, Azure и GCP

AWS	Azure	GCP
S3	Blob Storage	GCS
Glue	Synapse	Dataflow
Athena	Synapse	BigQuery

Упомянутые здесь сервисы предоставляют сопоставимые функции, но они не идентичны. Они могут использоваться для аналогичных целей, но у них разные функции и особенности. Например, AWS Glue – это управляемый сервис Apache Spark с базой метаданных. GCP Dataflow – это управляемый сервис Apache Beam. И Spark, и Beam нацелены на обработку больших данных, но делают это по-разному. В нашем случае они оба сделают свою работу.

18.3.1 Загрузка в GCS

Как и в главах 16 и 17, первая часть рабочего процесса извлекает оценки из нашего API рейтингов и загружает их в GCS. Хотя большинством сервисов GCP может управлять оператор Airflow, очевидно, что для обмена данными с нашим пользовательским API оператора нет. Хотя технически мы могли бы разделить эту работу, сначала извлекая данные рейтингов, записывая их в локальный файл, а затем загружая файл в GCS на втором этапе с помощью `LocalFilesystemToGCSOperator`, для краткости мы выполним это действие в одной задаче. Единственный компонент от Airflow, который можно применить здесь, – это `GCSHook` для выполнения действий с GCS.

Листинг 18.12 DAG извлекает рейтинги и загружает их в GCS

```
import datetime
import logging
import os
import tempfile
from os import path

import pandas as pd
from airflow.models import DAG
from airflow.operators.python import PythonOperator
from airflow.providers.google.cloud.hooks.gcs import GCSHook
```

```
from custom.hooks import MovielensHook

dag = DAG(
    "gcp_movie_ranking",
    start_date=datetime.datetime(year=2019, month=1, day=1),
    end_date=datetime.datetime(year=2019, month=3, day=1),
    schedule_interval="@monthly",
    default_args={"depends_on_past": True},
)

def _fetch_ratings(api_conn_id, gcp_conn_id, gcs_bucket, **context):
    year = context["execution_date"].year
    month = context["execution_date"].month

    logging.info(f"Fetching ratings for {year}/{month:02d}")

    api_hook = MovielensHook(conn_id=api_conn_id)
    ratings = pd.DataFrame.from_records(
        api_hook.get_ratings_for_month(year=year, month=month),
        columns=["userId", "movieId", "rating", "timestamp"],
    )
    logging.info(f"Fetched {ratings.shape[0]} rows")

    with tempfile.TemporaryDirectory() as tmp_dir:
        tmp_path = path.join(tmp_dir, "ratings.csv")
        ratings.to_csv(tmp_path, index=False)

        # Загружаем файл в GCS.
        logging.info(f"Writing results to ratings/{year}/{month:02d}.csv")
        gcs_hook = GCSHook (gcp_conn_id)
        gcs_hook.upload(
            bucket_name=gcs_bucket,
            object_name=f"ratings/{year}/{month:02d}.csv",
            filename=tmp_path,
        )

fetch_ratings = PythonOperator(
    task_id="fetch_ratings",
    python_callable=_fetch_ratings,
    op_kwargs={
        "api_conn_id": "movielens",
        "gcp_conn_id": "gcp",
        "gcs_bucket": os.environ["RATINGS_BUCKET"],
    },
    dag=dag,
)
```

Сначала извлекаем и записываем результаты в локальный файл

Инициализируем подключение к GCS

Загружаем локальный файл в GCS

Бакет GCS, в который будет загружен файл

Ключ GCS, в который будут записываться данные

Если все прошло успешно, то теперь у нас есть данные в бакете GCS, показанной на рис. 18.13.

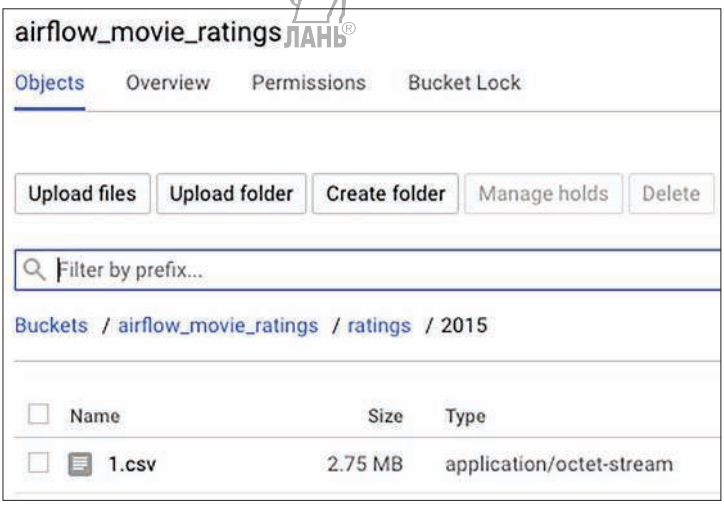


Рис. 18.13 Результаты успешного запуска начального ОАГ с рейтингами, загруженными в бакет в Google Cloud Storage

18.3.2 Загрузка данных в BigQuery

После загрузки данных в GCS мы загрузим данные в BigQuery, чтобы можно было запросить их. Хотя BigQuery может работать с внешними данными, его параметры при разделении данных несколько ограничены, особенно при создании внешних таблиц. Лучше выполнить внутреннюю загрузку данных в BigQuery. Есть несколько операторов Airflow, связанных с операциями для BigQuery; `GCSToBigQueryOperator` предназначен специально для загрузки данных, хранящихся в GCS, в BigQuery.

Листинг 18.13 Импорт секционированных данных из GCS в BigQuery

```
➥ from airflow.providers.google.cloud.transfers.gcs_to_bigquery import
    GCSToBigQueryOperator


import_in_bigquery = GCSToBigQueryOperator (
   task_id="import_in_bigquery",
   bucket="airflow_movie_ratings",
   source_objects=[
       "ratings/{{ execution_date.year }}/{{ execution_date.month }}.csv"
   ],
   source_format="CSV",
   create_disposition="CREATE_IF_NEEDED",
   write_disposition="WRITE_TRUNCATE",
   bigquery_conn_id="gcp",
   autodetect=True,
```

Создаем таблицу, если ее не существует

Перезаписываем данные секции, если они уже существуют

```
        destination_project_dataset_table=(
            "airflow-pipelines:",
            "airflow.ratings${{ ds_nodash }}",
        ),
        dag=dag,
    )

fetch_ratings >> import_in_bigquery
```

Пытаемся автоматически определить схему

Значение после символа $ определяет секцию для записи под названием «декоратор секции»

Так мы создаем вторую часть этого ОАГ (рис. 18.14).

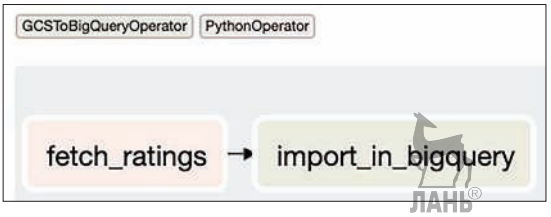


Рис. 18.14 Загрузка и импорт данных в GCP BigQuery

Как видите, мы определяем источник (файл в бакете GCS) и цель (раздел таблицы BigQuery), но есть и другие конфигурации. Например, аргументы `create_disposition` и `write_disposition` определяют поведение в случае, если таблица не существует или раздел уже существует. Их значения (`CREATE_IF_NEEDED` и `WRITE_TRUNCATE`) могут показаться неожиданными. Операторы Airflow, связанные с GCP, предоставляют удобные оболочки для базового запроса к Google. Они предоставляют вам как разработчику интерфейс для вызова базовой системы при использовании функций Airflow, например переменных, которые можно шаблонировать. Но такие аргументы, как `create_disposition`, предназначены конкретно для GCP и передаются непосредственно в запрос. Таким образом, единственный способ узнать их ожидаемые значения – это внимательно прочитать документацию по Airflow, документацию GCP или в крайнем случае проверить исходный код.

После запуска этого рабочего процесса мы можем проверить данные в BigQuery (рис. 18.15).

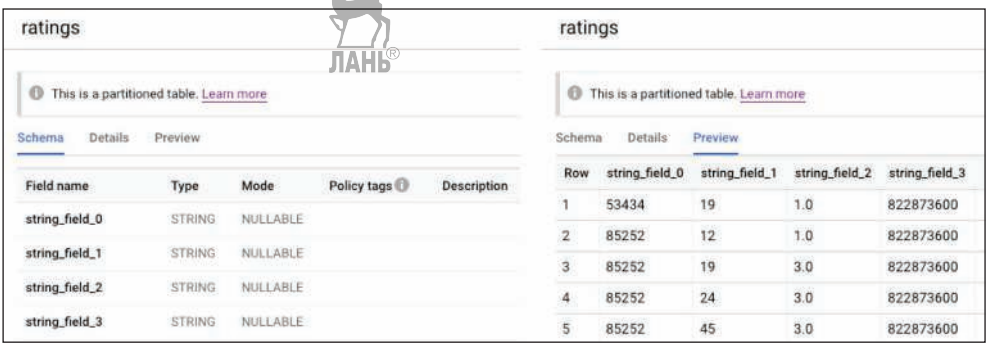


Рис. 18.15 Проверка импортированных данных в BigQuery

Однако, как видно слева, автоопределение схемы (для которого мы задали значение `True`) не смогло автоматически определить схему, что видно из имен столбцов «string_field_0», «string_field_1» и т. д. Хотя автоопределение и делает свою работу, нет никаких гарантий, что в итоге все сработает правильно. В этой ситуации мы знаем, что структура данных не изменится. Таким образом, можно безопасно предоставить схему вместе с запросом.

Листинг 18.14 Импорт данных из GCS в BigQuery со схемой

```
➥ from airflow.providers.google.cloud.transfers.gcs_to_bigquery import
    GCSToBigQueryOperator

import_in_bigquery = GCSToBigQueryOperator (
   task_id="import_in_bigquery",
   bucket="airflow_movie_ratings",
   source_objects=[
       "ratings/{{ execution_date.year }}/{{ execution_date.month }}.csv"
   ],
   source_format="CSV",
   create_disposition="CREATE_IF_NEEDED",
   write_disposition="WRITE_TRUNCATE",
   bigquery_conn_id="gcp",
   skip_leading_rows=1,        <-- Пропускаем строку заголовка
   schema_fields=[             <-- Определяем схему вручную
       {"name": "userId", "type": "INTEGER"},
       {"name": "movieId", "type": "INTEGER"},
       {"name": "rating", "type": "FLOAT"},
       {"name": "timestamp", "type": "TIMESTAMP"},
   ],
   destination_project_dataset_table=(
       "airflow-pipelines:",
       "airflow.ratings${{ ds_nodash }}",
   ),
   dag=dag,
)
```

Теперь проверка не только показывает нам правильную схему, но и отображает изящно отформатированную временную метку (рис. 18.16).

18.3.3 Извлечение рейтингов, находящихся в топе

Наконец, нам нужно вычислить рейтинги, находящиеся в топе, в BigQuery и сохранить результаты. Ни BigQuery, ни Airflow не предоставляют для этого готового решения. Хотя мы можем выполнять запросы и экспортировать полные таблицы, нельзя напрямую экспортировать результат запроса. Обходной путь – сначала сохранить результат запроса в новой таблице, экспортировать таблицу, а затем удалить промежуточную таблицу для очистки.

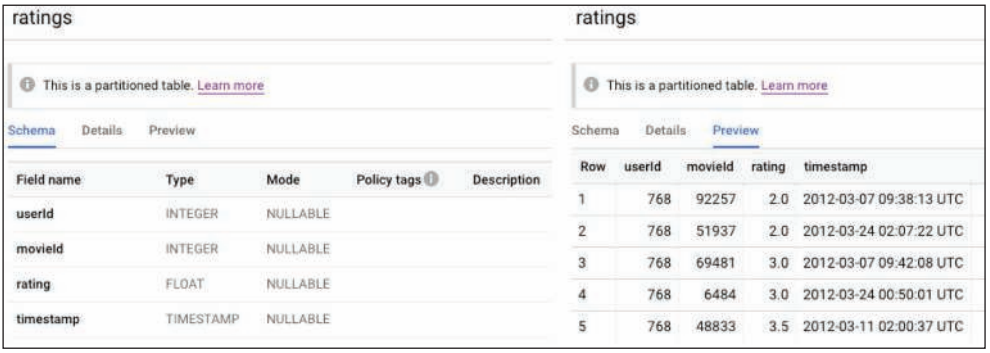


Рис. 18.16 Проверка импортированных данных в BigQuery с помощью предопределенной схемы

Листинг 18.15 Экспорт результатов запроса BigQuery через промежуточную таблицу

```
from airflow.providers.google.cloud.operators.bigquery import
BigQueryExecuteQueryOperator, BigQueryDeleteTableOperator
➥ from airflow.providers.google.cloud.transfers.bigquery_to_gcs import
    BigQueryToGCSOperator

query_top_ratings = BigQueryExecuteQueryOperator (
   task_id="query_top_ratings",
   destination_dataset_table=(
       "airflow-pipelines:",
       "airflow.ratings_{{ ds_nodash }}",
   ),
   sql="""SELECT
movieid,
AVG(rating) as avg_rating,
COUNT(*) as num_ratings
FROM airflow.ratings
WHERE DATE(timestamp) <= DATE("{{ ds }}")
GROUP BY movieid
ORDER BY avg_rating DESC
""",
   write_disposition="WRITE_TRUNCATE",
   create_disposition="CREATE_IF_NEEDED",
   bigquery_conn_id="gcp",
   dag=dag,
)

extract_top_ratings = BigQueryToGCSOperator(
   task_id="extract_top_ratings",
   source_project_dataset_table=(
       "airflow-pipelines:",
       "airflow.ratings_{{ ds_nodash }}",
   ),
   destination_cloud_storage_uris=(
       "gs://airflow_movie_results/{{ ds_nodash }}.csv"
```

Таблица BigQuery

SQL-запрос для выполнения

Таблица BigQuery для извлечения

Извлекаем целевой путь

```
    ),
    export_format="CSV",
    bigquery_conn_id="gcp",
    dag=dag,
)
delete_result_table = BigQueryTableDeleteOperator(
    task_id="delete_result_table",
    deletion_dataset_table=(
        "airflow-pipelines:",
        "airflow.ratings_{{ ds_nodash }}",
    ),
    bigquery_conn_id="gcp",
    dag=dag,
)

➥ fetch_ratings >> import_in_bigquery >> query_top_ratings >>
    extract_top_ratings >> delete_result_table
```

Таблица BigQuery, которую нужно удалить

На веб-сервере Airflow результат выглядит, как показано на рис. 18.17.

Рис. 18.17 Полный ОАГ для скачивания рейтингов, загрузки и обработки с использованием BigQuery

Используя контекстную переменную `ds_nodash`, нам удалось объединить серию задач, выполняющих различные действия в BigQuery. В каждом запуске ОАГ ее значение остается неизменным и, таким образом, может использоваться для соединения результатов задачи, избегая при этом их переопределения одной и той же задачей через разные промежутки времени. Результат – бакет, заполненный файлами с расширением CSV (рис. 18.18).

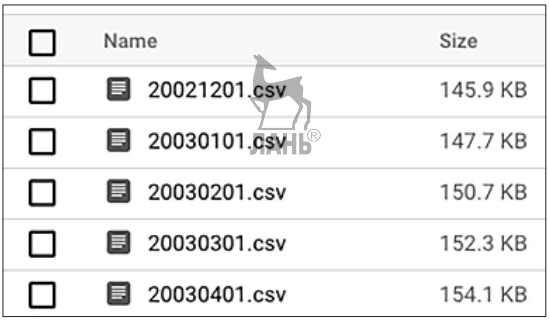


Рис. 18.18 Результаты экспортируются и сохраняются в виде CSV-файлов с соответствующими датами и временем

На стороне BigQuery, если мы выполняем несколько запусков ОАГ одновременно, будет создано несколько промежуточных таблиц. Они удобно сгруппированы в BigQuery (рис. 18.19).

Рис. 18.19 BigQuery группирует таблицы с одинаковыми суффиксами. При одновременном запуске нескольких ОАГ это может привести к созданию нескольких промежуточных таблиц

Последняя задача в этом ОАГ очищает промежуточную таблицу результатов. Обратите внимание, что операция запроса к BigQuery, извлечения результатов и удаления промежуточной таблицы теперь разделена на три задачи. Нет такой операции, чтобы выполнить все это в одной задаче, ни в BigQuery, ни в Airflow. Теперь предположим, что `extract_top_ratings` по какой-то причине не работает, тогда у нас останется таблица BigQuery. Цены на BigQuery состоят из нескольких элементов, включая хранилище данных, поэтому будьте осторожны, оставляя остатки, поскольку это может повлечь за собой расходы (как в любом облаке). Когда вы все закончите, не забудьте удалить все ресурсы. В Google Cloud это просто делается путем удаления соответствующего проекта (при условии что все ресурсы находятся в одном проекте). В меню **IAM & Admin > Manage Resources** выберите проект и нажмите **Delete** (Удалить).

После нажатия кнопки **Shut Down** (Завершить работу) ваш проект будет удален. Примерно через 30 дней Google удаляет все ресурсы, хотя никаких гарантий не дается, и некоторые ресурсы могут быть удалены (намного) раньше, чем другие.

Резюме

- Самый простой способ установить и запустить Airflow в GCP – это GKE, используя диаграмму Helm в качестве отправной точки.
- Airflow предоставляет множество привязок и операторов для GCP, которые позволяют интегрироваться с различными сервисами в облачной платформе Google, установленной вместе с пакетом `apache-airflow-provider-google`.

- Класс `GoogleBaseHook` обеспечивает аутентификацию для GCP, позволяя вам сосредоточиться на деталях сервиса при реализации собственных хуков и операторов GCP.
- Использование хуков и операторов GCP обычно требует, чтобы вы настроили необходимые ресурсы и права доступа в GCP и Airflow, дабы Airflow мог выполнять требуемые операции.

Приложение A
Запуск примеров кода

К этой книге прилагается репозиторий кода на сайте GitHub (https://github.com/BasPH/data-pipelines-with-apache-airflow). В нем содержится тот же код, что и в данной книге, а также легко исполняемые окружения Docker, так что вы можете запускать все примеры самостоятельно. В этом приложении объясняется, как организован код и как запускать примеры.

A.1 Структура кода

Код разбит по главам, и у каждой главы одинаковая структура. Верхний уровень репозитория состоит из нескольких каталогов глав (пронумерованных 01–18), которые содержат автономные примеры кода для соответствующих глав. Каждый каталог содержит как минимум следующие файлы и каталоги:

- dags – каталог, содержащий файлы ОАГ, показанные в главе;
- docker-compose.yml – файл, описывающий настройку Airflow, необходимую для запуска ОАГ;
- README.md – файл, в котором представлены примеры глав и поясняются детали, относящиеся к каждой главе, о том, как запускать примеры.

Там, где это возможно, листинги кода из книги ссылаются на соответствующий файл в каталоге главы. Для некоторых глав списки кодов, показанные в главах, будут соответствовать отдельным ОАГ. В других случаях (особенно для более сложных примеров) несколько листингов будут объединены в один ОАГ, в результате чего будет получен один файл ОАГ.

Помимо файлов OAГ и кода на Python, некоторые примеры из книги (особенно в главах 16, 17 и 18) требуют дополнительных вспомогательных ресурсов или конфигурации для запуска примеров. Дополнительные шаги, необходимые для запуска этих примеров, будут описаны в соответствующей главе и в файле README для данной главы.

A.2 Запуск примеров

Каждая глава поставляется с окружением Docker, которое можно использовать для запуска соответствующих примеров кода.

A.2.1 Запуск окружения Docker

Чтобы начать работу с примерами глав, выполните команду

```
$ docker-compose up -build
```

Эта команда запускает окружение Docker, содержащее ряд контейнеров, необходимых для работы Airflow, включая следующие контейнеры:

- веб-сервер Airflow;
- планировщик Airflow;
- база данных Postgres для базы метаданных Airflow.

Чтобы не видеть вывод всех трех контейнеров в своем терминале, также можно запустить окружение Docker в фоновом режиме:

```
$ docker-compose up --build -d
```

В некоторых главах создаются дополнительные контейнеры, которые предоставляют другие сервисы или API, необходимые для примеров. Например, в главе 12 демонстрируются следующие сервисы мониторинга, которые также созданы в Docker, чтобы примеры были по возможности максимально реалистичными:

- Grafana;
- Prometheus;
- Flower;
- Redis.

К счастью, о запуске всех этих сервисов позаботится файл docker-compose. Не стесняйтесь подробнее изучить его, если вам интересно.

A.2.2 Проверка запущенных сервисов

После запуска примера вы можете проверить, какие контейнеры работают, используя команду `docker ps`:

```
$ docker ps
CONTAINER ID   IMAGE                                ... NAMES
d7c68a1b9937   apache/airflow:2.0.0-python3.8       ... chapter02_scheduler_1
557e97741309   apache/airflow:2.0.0-python3.8       ... chapter02_webserver_1
742194dd2ef5   postgres:12-alpine                   ... chapter02_postgres_1
```

По умолчанию префиксы docker-compose, запускающие контейнеры с именем содержащей их папки, означают, что контейнеры, относящиеся к каждой главе, должны распознаваться по именам контейнеров.

Вы также можете проверить журналы отдельных контейнеров, используя команду `docker logs`:

```
$ docker logs -f chapter02_scheduler_1
➥ [2020-11-30 20:17:36,532] {scheduler_job.py:1249} INFO - Starting the
    scheduler
➥ [2020-11-30 20:17:36,533] {scheduler_job.py:1254} INFO - Processing each
    file at most -1 times
➥ [2020-11-30 20:17:36,984] {dag_processing.py:250} INFO - Launched
    DagFileProcessorManager with pid: 131
```

Надеемся, что эти журналы смогут предоставить вам ценные отзывы, если что-то пойдет не так.

A.2.3 Завершение работы с окружением

Когда вы закончите запускать пример, то можете выйти из docker-compose с помощью комбинации клавиш **CTRL**+**C**. (Обратите внимание, что в этом нет необходимости, если вы запускаете docker-compose в фоновом режиме.) Чтобы полностью выйти из окружения Docker, можно выполнить следующую команду из каталога главы:

```
$ docker-compose down -v
```

Помимо остановки различных контейнеров, она также должна позаботиться об удалении всех сетей и томов Docker, используемых в примере.

Для проверки, действительно ли все контейнеры были полностью удалены, можно использовать следующую команду, чтобы увидеть все контейнеры, которые были остановлены, но еще не удалены:

```
$ docker ps -a
```

Может получиться так, что вы по-прежнему будете видеть список контейнеров, которые хотите удалить. Их можно удалять один за другим, используя следующую команду:

```
$ docker rm <container_id>
```

где мы получаем `container_id` из списка контейнеров, показанного командой `docker ps`. В качестве альтернативы можно использовать следующий сокращенный вариант для удаления всех контейнеров:

```
$ docker rm $(docker ps -aq)
```

Наконец, вы также можете удалить все неиспользуемые тома, которые ранее использовались этими контейнерами:

```
$ docker volume prune
```

Однако мы настоятельно рекомендуем вам соблюдать осторожность при использовании данной команды, так как это может привести к непреднамеренной потере данных, если вы избавитесь не от тех томов Docker.

Приложение B
Структуры пакетов Airflow 1 и 2

Большая часть этой книги основана на Airflow 1. Незадолго до ее выпуска вышел Airflow 2, и мы решили обновить весь код для Airflow 2.

Одно из наиболее значительных изменений – это новые пакеты поставщиков. Многие модули были удалены из основного Airflow и теперь устанавливаются через отдельный пакет «провайдеров», чтобы уменьшить основной пакет Airflow. В этом приложении мы перечислим весь импорт Airflow, использованный в книге, и его пути в Airflow 1 и Airflow 2.

B.1 Структура пакета Airflow 1

В Airflow 1 разделение было выполнено на «основные» (core) компоненты (операторы/хуки/сенсоры и т. д.) и contrib-компоненты, например `airflow.operators.python_operator.PythonOperator` и `airflow.contrib.sensors.python_sensor.PythonSensor`.

Это был исторический артефакт со времени разработки Airflow в Airbnb, где деление компонентов на «core» и «contrib» имело смысл внутри Airbnb. Когда проект Airflow приобрел популярность в качестве проекта с открытым исходным кодом, разделение на core и contrib стало серой зоной и частой темой обсуждения в сообществе. На протяжении всей разработки Airflow 1 модули из пакета contrib сохранялись в contrib, чтобы избежать критических изменений.

B.2 Структура пакета Airflow 2

С появлением Airflow 2 сообщество наконец достигло момента, когда оно могло допустить критические изменения, и, таким образом, было принято решение реструктурировать пакет Airflow, чтобы создать структуру, которая соответствует тому глобальному масштабу проекта, в котором он сейчас работает. Еще один распространенный источник раздражения – большое количество зависимостей, установки которых требует Airflow.

Поэтому сообщество решило разделить проект Airflow на отдельные проекты:

- «базовый» проект, содержащий всего несколько универсальных операторов, хуков и т. д.;
- другие компоненты, которые можно установить через отдельные пакеты, что позволяет разработчикам выбирать, какие компоненты установить, сохраняя управляемый набор зависимостей. Эти дополнительные пакеты называются «поставщиками». Каждый такой пакет называется `apache-airflow-providers-[имя]`, например `apache-airflow-providers-postgres`.

Все компоненты, которые теперь содержатся в поставщиках, удалены из ядра Airflow. Например, класса `airflow.hooks.postgres_hook.PostgresHook` больше нет в Airflow 2. Чтобы добавить его, установите пакет

```
pip install apache-airflow-providers-postgres
```

и импортируйте `airflow.providers.postgres.operators.postgres.PostgresOperator`.

> **ПРИМЕЧАНИЕ** Если вы хотите подготовить свои файлы ОАГ к плавному переходу с Airflow 1 на Airflow 2, каждый пакет поставщиков также существует в виде «бэкпортов». Эти пакеты содержат структуру Airflow 2, но все компоненты совместимы с Airflow 1. Например, чтобы использовать новую структуру поставщиков postgres в Airflow 1, применяйте
>
> ```
> pip install apache-airflow-backport-providers-postgres
> ```

В табл. B.1 перечислены все импорты Airflow, выполненные в примерах кода в этой книге, показаны пути как в Airflow 1, так и в Airflow 2 и, если применимо, дополнительный пакет поставщиков для установки в Airflow 2.

Таблица B.1 Импорты Airflow

Путь импорта Airflow 2	Дополнительный пакет Airflow 2	Путь импорта Airflow 1
`airflow.providers.amazon.aws.hooks.base_aws.AwsBaseHook`	`apache-airflow-providers-amazon`	`airflow.contrib.hooks.aws_hook.AwsHook`

Таблица B.1 (*продолжение*)

Путь импорта Airflow 2	Дополнительный пакет Airflow 2	Путь импорта Airflow 1
airflow.providers.microsoft.azure.hooks.wasb.WasbHook	apache-airflow-providers-microsoft-azure	airflow.contrib.hooks.wasb_hook.WasbHook
kubernetes.client.models.V1Volume	kubernetes	airflow.contrib.kubernetes.volume.Volume
kubernetes.client.models.V1VolumeMount	Kubernetes	airflow.contrib.kubernetes.volume_mount.VolumeMount
airflow.providers.amazon.aws.operators.athena.AWSAthenaOperator	apache-airflow-providers-amazon	airflow.contrib.operators.aws_athena_operator.AWSAthena-Operator
airflow.providers.google.cloud.operators.bigquery.BigQueryExecuteQueryOperator	apache-airflow-providers-google	airflow.contrib.operators.bigquery_operator.BigQuery-Operator
airflow.providers.google.cloud.operators.bigquery.BigQueryDeleteTableOperator	apache-airflow-providers-google	airflow.contrib.operators.bigquery_table_delete_operator.BigQueryTableDeleteOperator
airflow.providers.google.cloud.transfers.bigquery_to_gcs.BigQueryToGCSOperator	apache-airflow-providers-google	airflow.contrib.operators.bigquery_to_gcs.BigQueryTo-CloudStorageOperator
airflow.providers.google.cloud.transfers.local_to_gcs.LocalFilesystemToGCSOperator	apache-airflow-providers-google	airflow.contrib.operators.gcs_to_bq.GoogleCloudStorageToBig-QueryOperator
airflow.providers.google.cloud.transfers.gcs_to_bigquery.GCSToBigQueryOperator	apache-airflow-providers-google	airflow.contrib.operators.gcs_to_bq.GoogleCloudStorageToBig-QueryOperator
airflow.providers.cncf.kubernetes.operators.kubernetes_pod.KubernetesPodOperator	apache-airflow-providers-cncfkubernetes	airflow.contrib.operators.kubernetes_pod_operator.Kubernetes-PodOperator
airflow.providers.amazon.aws.operators.s3_copy_object.S3CopyObjectOperator	apache-airflow-providers-amazon	airflow.contrib.operators.s3_copy_object_operator.S3Copy-ObjectOperator
airflow.providers.amazon.aws.operators.sagemaker_endpoint.SageMakerEndpointOperator	apache-airflow-providers-amazon	airflow.contrib.operators.sagemaker_endpoint_operator.SageMakerEndpointOperator
airflow.providers.amazon.aws.operators.sagemaker_training.SageMakerTrainingOperator	apache-airflow-providers-amazon	airflow.contrib.operators.sagemaker_training_operator.SageMakerTrainingOperator
airflow.sensors.filesystem.FileSensor		airflow.contrib.sensors.file_sensor.FileSensor
airflow.sensors.python.PythonSensor		airflow.contrib.sensors.python_sensor.PythonSensor
airflow.DAG		airflow.DAG
airflow.exceptions.Airflow-SkipException		airflow.exceptions.AirflowSkip-Exception
airflow.hooks.base_hook.BaseHook		airflow.hooks.base_hook.BaseHook
airflow.providers.postgres.hooks.postgres.PostgresHook	apache-airflow-providers-postgres	airflow.hooks.postgres_hook.PostgresHook
airflow.providers.amazon.aws.hooks.s3.S3Hook	apache-airflow-providers-postgres	airflow.hooks.S3_hook.S3Hook
airflow.models.BaseOperator		airflow.models.BaseOperator
airflow.models.Connection		airflow.models.Connection
airflow.models.DAG		airflow.models.DAG

Таблица B.1 (*окончание*)

Путь импорта Airflow 2	Дополнительный пакет Airflow 2	Путь импорта Airflow 1
airflow.models.Variable		airflow.models.Variable
airflow.operators.bash.BashOperator		airflow.operators.bash_operator.BashOperator
airflow.operators.dagrun_operator.TriggerDagRunOperator		airflow.operators.dagrun_operator.TriggerDagRunOperator
airflow.providers.docker.operators.docker.DockerOperator	apache-airflow-providers-docker	airflow.operators.docker_operator.DockerOperator
airflow.operators.dummy_operator.DummyOperator		airflow.operators.dummy_operator.DummyOperator
airflow.providers.http.operators.http.SimpleHttpOperator	apache-airflow-providers-http	airflow.operators.http_operator.SimpleHttpOperator
airflow.operators.latest_only.LatestOnlyOperator		airflow.operators.latest_only_operator.LatestOnlyOperator
airflow.providers.postgres.operators.postgres.PostgresOperator	apache-airflow-providers-postgres	airflow.operators.postgres_operator.PostgresOperator
airflow.operators.latest_only.LatestOnlyOperator		airflow.operators.latest_only_operator.LatestOnlyOperator
airflow.providers.postgres.operators.postgres.PostgresOperator		airflow.operators.postgres_operator.PostgresOperator
airflow.operators.python.PythonOperator		airflow.operators.python_operator.PythonOperator
airflow.utils		airflow.utils
airflow.utils.decorators.apply_defaults		airflow.utils.apply_defaults
airflow.utils.dates		airflow.utils.dates
airflow.utils.decorators.apply_defaults		airflow.utils.decorators.apply_defaults

Приложение C
Сопоставление метрик в Prometheus

Это приложение содержит сопоставлене метрик из формата StatsD в формат Prometheus, как описано в главе 12. Оно также содержится в сопутствующем репозитории GitHub (https://github.com/BasPH/data-pipelines-with-apache-airflow), где демонстрируется с помощью экспортера Prometheus, StatsD. Экспортер StatsD принимает метрики StatsD (предоставленные Airflow) и предоставляет к ним доступ в формате, который может прочитать Prometheus. Однако некоторые преобразования неэффективны или не соответствуют правилам именования Prometheus. Таким образом, здесь метрики StatsD явно сопоставляются с метриками Prometheus. Ввиду того, что Airflow – это проект с открытым исходным кодом, данное сопоставление может быть изменено.

Листинг C.1 Сопоставление метрик с помощью экспортера StatsD

```
mappings:

- match: "airflow.dag_processing.total_parse_time"
 help: Number of seconds taken to process all DAG files
 name: "airflow_dag_processing_time"

- match: "airflow.dag.*.*.duration"
 name: "airflow_task_duration"
 labels:
   dag_id: "$1"
   task_id: "$2"

- match: "airflow.dagbag_size"
 help: Number of DAGs
 name: "airflow_dag_count"
```

```
- match: "airflow.dag_processing.import_errors"
 help: The number of errors encountered when processing DAGs
 name: "airflow_dag_errors"

- match: "airflow.dag.loading-duration.*"
 help: Loading duration of DAGs grouped by file. If multiple DAGs are found
     in one file, DAG ids are concatenated by an underscore in the label.
 name: "airflow_dag_loading_duration"
 labels:
   dag_ids: "$1"

- match: "airflow.dag_processing.last_duration.*"
 name: "airflow_dag_processing_last_duration"
 labels:
   filename: "$1"

- match: "airflow.dag_processing.last_run.seconds_ago.*"
 name: "airflow_dag_processing_last_run_seconds_ago"
 labels:
   filename: "$1"

- match: "airflow.dag_processing.last_runtime.*"
 name: "airflow_dag_processing_last_runtime"
 labels:
   filename: "$1"

- match: "airflow.dagrun.dependency-check.*"
 name: "airflow_dag_processing_last_runtime"
 labels:
   dag_id: "$1"

- match: "airflow.dagrun.duration.success.*"
 name: "airflow_dagrun_success_duration"
 labels:
   dag_id: "$1"

- match: "airflow.dagrun.schedule_delay.*"
 name: "airflow_dagrun_schedule_delay"
 labels:
   dag_id: "$1"

- match: "airflow.executor.open_slots"
 help: The number of open executor slots
 name: "airflow_executor_open_slots"

- match: "airflow.executor.queued_tasks"
 help: The number of queued tasks
 name: "airflow_executor_queued_tasks"

- match: "airflow.executor.running_tasks"
 help: The number of running tasks
 name: "airflow_executor_running_tasks"
```

Предметный указатель

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **http://www.galaktika-dmk.com/**.

Бас Харенслак, Джулиан де Руйтер

Apache Airflow и конвейеры обработки данных

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Беликов Д. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 40,79. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

Apache Airflow и конвейеры обработки данных

Конвейеры обработки данных управляют потоком данных с момента их первоначального сбора до консолидации, очистки, анализа, визуализации и многого другого. Apache Airflow предоставляет единую платформу, которую можно использовать для проектирования, реализации, мониторинга и обслуживания конвейеров. Простота пользовательского интерфейса, возможности plug-and-play и гибкие сценарии Python оптимизируют решение любых задач, касающихся управления данными.

Эта книга научит вас создавать и сопровождать эффективные конвейеры обработки данных. Вы изучите наиболее распространенные модели их использования, особенности ориентированных ациклических графов (ОАГ), которые обеспечивают работу Airflow, а также способы их настройки в соответствии с потребностями вашего конвейера.

«Настольная книга по Airflow. Для пользователей всех уровней, от новичка до эксперта».

Рамбабу Поса, Sai AAshika Consultancy

«Справочное издание, где представлено все, что вам нужно знать о создании, разработке, планировании и мониторинге рабочих процессов с помощью Apache Airflow. Однозначно рекомендую».

Торстен Вебер, bbv Software Services AG

«Безусловно, лучший ресурс по Airflow».

Джонатан Вуд, LexisNexis

Издание предназначено для специалистов по DevOps, обработке и хранению данных, машинному обучению, а также системных администраторов с навыками программирования на Python.

Вы научитесь:

- создавать, тестировать и развертывать конвейеры Airflow как ОАГ;
- автоматизировать перемещение и преобразование данных;
- анализировать наборы архивных данных с помощью обратного заполнения;
- разрабатывать собственные компоненты;
- настраивать Airflow в промышленном окружении.

Бас Харенслак и Джулиан де Руйтер — эксперты, имеющие широкий опыт использования Airflow при разработке конвейеров для крупных компаний. Бас также принимает участие в разработке Airflow.

ISBN 978-5-97060-970-5

Интернет-магазин: www.dmkpress.com

Оптовая продажа: КТК «Галактика»
books@alians-kniga.ru

ДМК ПРЕСС издательство
www.дмк.рф