Data Pipelines with Apache Airflow

Bas P. Harenslak
Julian R. de Ruiter

MEAP

MANNING

MEAP版Manning Early Access计划

数据流水线

使用Apache Airflow

版本1

欢迎

感谢您购买用于Apache Airflow数据管道的MEAP。我们希望这本书在目前的状态下对你来说已经很有价值，并希望利用你的反馈使这本书的最终版本更好！

本书旨在为精通数据的专业人士提供对AirFlow的深入介绍，AirFlow是一种工具/框架，可帮助您开发由许多任务和（可能）跨越不同技术组成的复杂数据管道。当我们开始编写第一个Airflow管道时，看到一个简单的Python脚本将各种任务粘合在一起，并处理依赖关系、重试、日志记录等复杂逻辑，这让我们松了一口气。

Apache Airflow框架拥有许多用于编写、运行和监视管道的可能选项。虽然这为您提供了很大的自由来以您喜欢的任何方式定义管道，但它也不会产生单一的好方法或最好的方法。本书旨在从头到尾提供Airflow框架的指南，以及从我们使用Apache Airflow的经验中学到的最佳实践和经验教训。

由于Airflow本身是用Python编写的，因此假定您有一定的Python编程经验。除此之外，在数据领域拥有一些经验也很有帮助，因为Airflow只是一种用于协调技术的编排工具，它本身并不是一种数据处理工具。因此，您需要对要协调的工具有一定的了解。数据领域是快速和不断变化的-因此，它有助于在该领域的经验，以快速了解新技术，以及如何适应他们在您的数据管道。

本书的第1部分涵盖了每个人都应该知道的气流的基本知识-框架的构建模块。第2部分为更高级的用户提供了深入探讨，包括开发和测试自定义操作符等主题。第3部分将研究如何在生产中运行Airflow-执行CI/CD、横向扩展、安全性等。

我们希望您喜欢这本书，并希望它能在您的实体书架上占据显著位置。我们鼓励您在LiveBook论坛。 任何反馈都非常感谢，并将有助于改进这本书。

— 巴斯·哈伦斯拉克和朱利安·德·鲁伊特

简要内容

1

认识Apache Airflow

本章包括：

- 什么是Apache Airflow
- 气流解决了什么问题
- 气流是适合您的工具吗

人们和公司越来越精通数据，并将开发数据管道作为其日常业务的一部分。多年来，数据量大幅增加，从早期应用程序中的每天仅兆字节增加到今天遇到的每天数Pb的数据集。不过，我们可以管理数据泛滥。

有些管道使用实时数据，有些则使用批量数据。这两种方法都有各自的好处。Apache Airflow是一个用于开发和监控批量数据管道的平台。

Airflow提供了一个框架来集成不同技术的数据管道。Airflow工作流是在Python脚本中定义的，它提供了一组构建块来与各种技术进行通信。

把气流想象成网中的蜘蛛。它以分布式体系结构的形式控制系统。它本身并不是一个数据处理工具，而是协调这样做的过程。这本书孤立和教导的部分过程中蔓延的网络。我们将详细检查气流的所有组件。首先，在我们分解Airflow架构的鸟瞰图并确定此工具是否适合您之前，让我们快速浏览一下工作流管理器，以调整我们的工作思路。

1.1 工作流管理器简介

许多计算过程由多个动作组成，这些动作需要以特定的顺序执行，可能以规则的间隔执行。这些过程可以用图形表示。

它定义了（a）哪些单独的操作组成了流程，以及（B）这些操作需要以什么顺序执行。这些图表通常称为工作流。

计算过程和物理过程通常都可以被认为是一系列任务，当以正确的顺序执行时，这些任务可以实现特定的目标。从这个角度来看，任务是形成整体的一小部分的工作的一小部分，但它们共同组成了产生预期结果所需的步骤。根据这一思想，实现（软件）流程的一种方法是简单地将这些任务编码到一个线性脚本中，该脚本一个接一个地执行任务，如下面的示例清单所示。

然而，如果任务可以并行执行，则这种方法不能很好地利用我们的资源，因为它一次只能执行一个任务。此外，如果任何任务失败，整个脚本也会失败。这意味着，要重新执行失败的任务，我们需要重新启动进程，这涉及到重新执行可能已成功执行的任务。最后，就其本身而言，这种方法不会为我们提供有关单个任务的结果的任何详细反馈，例如这些任务是何时执行的以及完成这些任务需要多长时间。

1.1.1 作为一系列任务的工作流

协调任务的挑战并不是计算领域的新问题。最常见的方法之一是根据工作流来定义流程，工作流表示要运行的任务的集合以及这些任务之间的依赖关系（即，需要首先运行哪些任务）。

在过去的几十年里，已经开发了许多工作流管理系统。大多数这些系统的核心是将任务定义为"工作单元"并表示这些任务之间的依赖关系的概念。这允许工作流管理系统跟踪哪些任务应该在何时运行，同时还负责诸如并行、失败作业的自动重试和报告等功能。

1.1.2 表示任务相关性

为了确定何时运行给定的任务，工作流管理系统需要知道在可以运行该任务之前需要执行哪些任务。任务之间的这种关系通常称为任务依赖性，因为任务的执行依赖于这些较早的任务（其依赖性）。

任务相关性的定义方式因工作流管理工具而异，但定义通常涉及说明哪些任务是给定任务的上游（指向任务相关性）或说明哪些任务是该任务的下游（将该任务标记为其他任务的相关性）。

可视化这些依赖关系的一种常见方法是将工作流描述为

有向图。在该表示中，任务通常被绘制为图的节点，而任务依赖性被描绘为节点之间的有向边。

假设您正在为一家雨伞公司开发一个模型，该公司希望根据历史销售数据和天气预测来预测未来世界各地的雨伞销售情况。这种过程的工作流程可以如下所示：

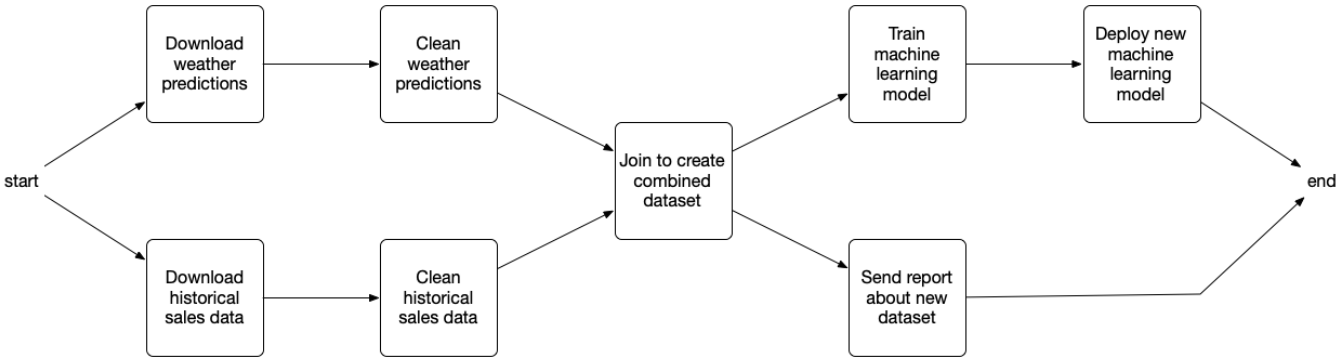


图1.1按一定顺序执行的一系列任务的示例，这些任务共同形成了一个工作流。箭头显示了任务之间的顺序，并告诉我们在连续任务开始之前必须完成哪些任务。

1.1.3 工作流管理系统

所有工作流管理器都有自己独特的方法来定义工作流。市场上并不缺少工作流管理器。在大数据兴起和处理数据工作流的过程中，一些公司/团体遇到了类似的挑战，他们开发了自己的工作流管理系统。虽然不是一个完整的列表，但这里有一些众所周知的选项[1]：

- 制作
- 奥齐
- 阿尔戈
- 库贝弗洛
- 阿兹卡班（源自LinkedIn）
- Luigi（源自Spotify）
- 指挥家（源自Netflix）
- 弹珠台（源自Pinterest）
- Airflow（源自Airbnb）

所有系统都有其优点和缺点。有些是面向特定平台或使用情形的。总的来说，可以指出它们之间的一些重要区别。

在Oozie中，工作流是在静态XML文件中定义的。另一方面，Airflow允许动态和灵活的工作流，因为它们是在Python代码中定义的。此外，Oozie还绑定到了Hadoop平台。它与Hadoop专用工具（如Hive和Spark）紧密集成。还可以运行shell脚本，这允许隐式运行任何类型的任务。

1有些工具最初是由公司的（前）员工创建的，但所有工具都是开源的，不是由一家公司提供的。

另一方面，Airflow本质上是一个Python框架，它允许运行可由Python执行的任何类型的任务，Hadoop-Tasks就是其中之一。Airflow生态系统为多种类型的任务提供了构建模块，例如发送电子邮件、运行Spark作业以及在Postgres数据库上运行SQL查询。

其他竞争对手提供类似的"通用"构建模块，如Luigi，其不同之处在于各种功能。Airflow和Luigi之间的几个关键区别是Airflow Resp中调度程序的可用性/缺乏。Luigi，它允许以一定的时间间隔调度工作流，而使用Luigi，您必须在系统外部调度工作流，例如使用cron。Airflow和Luigi都有一个用户界面，可以显示工作流的状态，但Airflow UI允许更多操作，例如重新运行工作流，并且通常被认为功能更丰富。

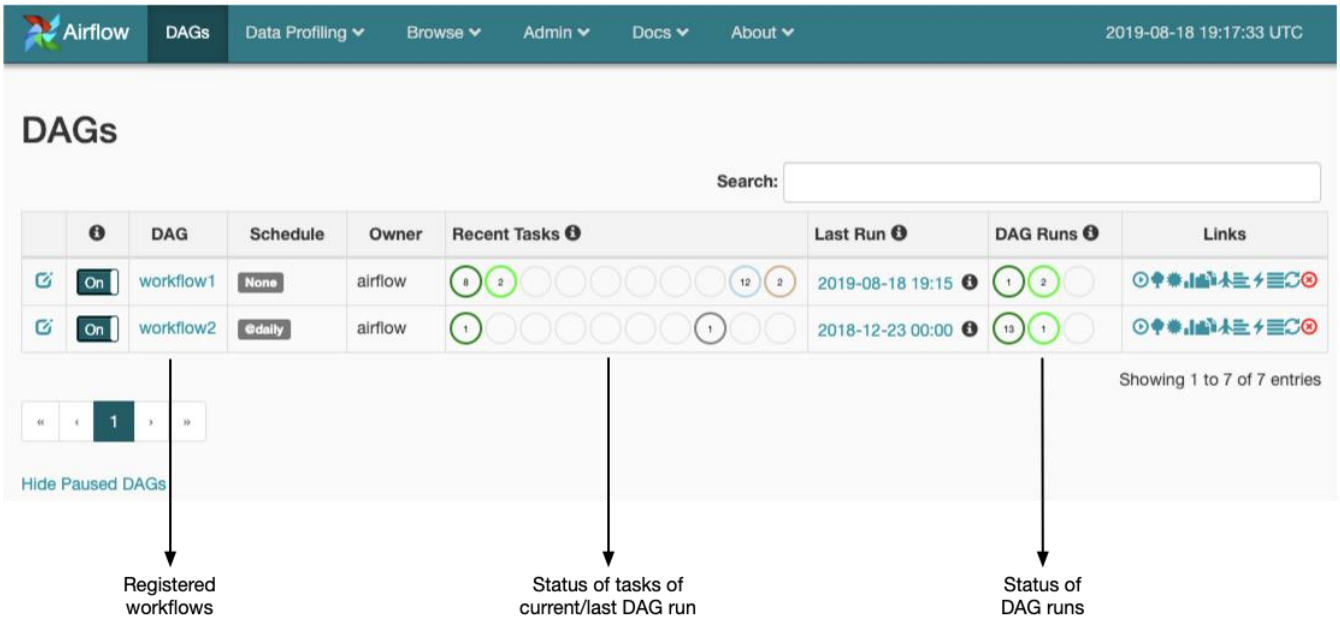


图1.2气流用户界面主页。它显示您的工作流及其当前状态。

最后，Airflow是一个不断增长的组件生态系统，可在任何系统上运行操作。虽然有许多方法可以处理您的数据，但我们作为数据工程师的经验告诉我们，Airflow是一个具有组件生态系统的卓越平台，可在各种系统上运行。

1.2 气流组件概述

气流由各种部件组成。一般体系结构由Web服务器、调度程序和数据库组成。数据库保存了气流系统的所有状态，以便您可以

随时重新启动气流过程，而不会丢失状态。数据库的常见选择是PostgreSQL和MySQL[2].

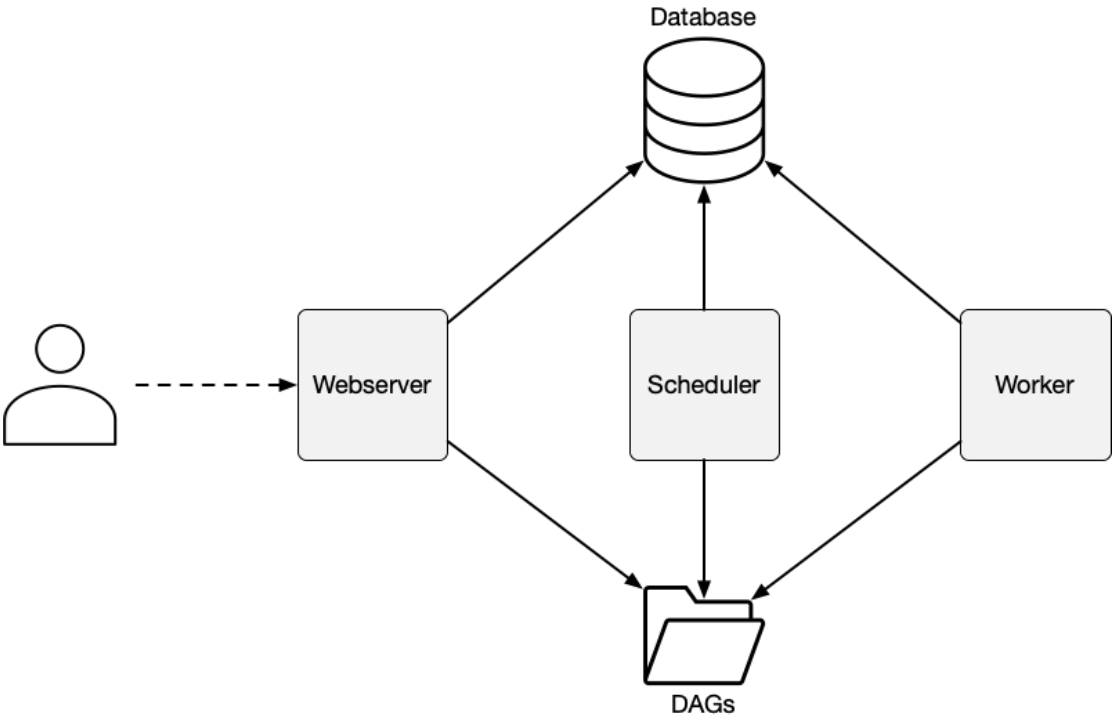


图1.3一般气流架构

Airflow Web服务器在基于Web的用户界面中显示工作流的状态，并支持对工作流进行手动操作，例如运行完整的工作流或重新启动单个失败的任务。它显示工作流程级别和单个任务级别的信息，并且可以显示每个任务运行的日志。

Airflow Scheduler进程负责检查是否满足运行任务的标准；依赖任务是否成功完成，任务是否有资格在给定调度间隔的情况下运行，是否满足执行的其他条件等。一旦满足所有条件，就会将此事实记录在数据库中，并由要执行的工作进程挑选各个任务。根据您选择的设置，此工作进程可以是单个

2在内部，Airflow使用SQLAlchemy与数据库通信。因此，SQLAlchemy支持的任何数据库都将得到Airflow的支持。

单个机器上的进程、单个机器上的多个进程或分布在多个机器上的多个进程。对于测试或第一次查看气流，单一流程设置是最简单和最容易的，但不建议在生产设置中使用，因为它速度慢且不能横向扩展。

所有Airflow进程都需要访问保存工作流的共享位置（称为有向非循环图或"DAG"）。Airflow反复扫描此目录中的新文件和更改，并将工作流的各种属性存储在数据库中。但是，在某些情况下，必须重新读取工作流文件，因此所有Airflow进程都必须可以访问这些文件。

1.2.1 有向无圈图

Airflow中的工作流被建模为DAG：有向无环图。Airflow针对的是批处理数据管道，其中任务的集合和任务之间的依赖关系共同形成了一个具有明确定义的开始和结束的图形。有了图中的循环，我们永远不会知道工作流何时完成；因此，工作流不能包含循环，也不能包含非循环属性。

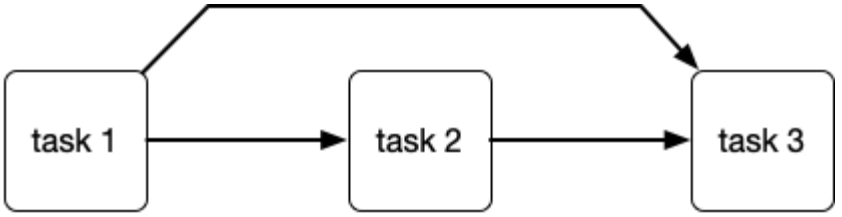


图1.4有效DAG的示例。这种结构只有一种可能的结束状态；任务3完成后，DAG即告完成。

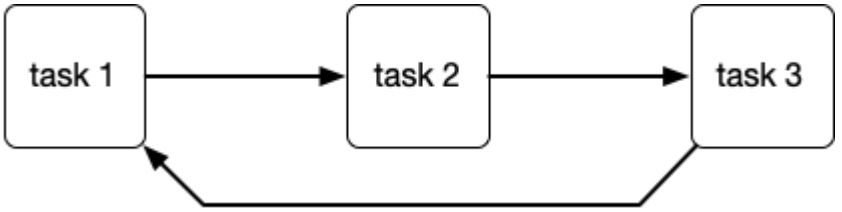


图1.5无效DAG的示例。任务3完成后，任务1将再次运行，我们无法知道工作流何时完成。

气流中的DAG由各种积木块构成。在接下来的章节中，我们将展开并详细介绍DAG组件。DAG构成核心结构，Airflow围绕该结构构建其工作流。用户界面是检查DAG的结构和状态的有用工具。

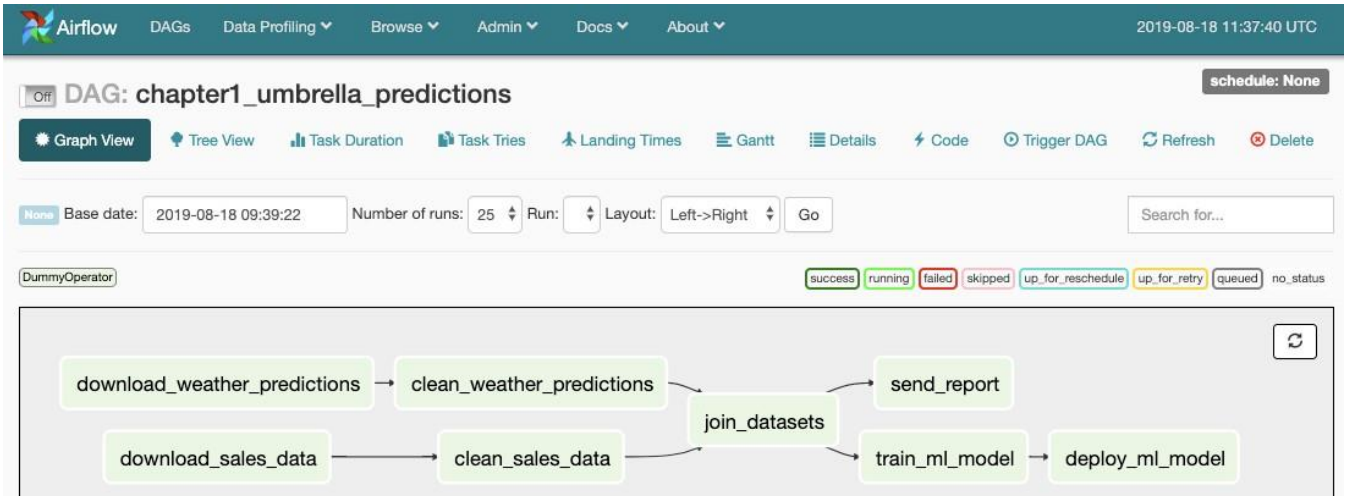


图1.6图1.1中的雨伞销售预测模型定义并可视化为气流中的DAG。

1.2.2 成批处理

气流在间歇过程的空间中运行；一系列具有明确定义的开始和结束任务的有限任务，以一定的时间间隔或触发器运行。虽然工作流的概念也存在于流空间中，但气流在那里不起作用。诸如Apache Spark之类的框架通常用于由Airflow中的任务触发的Airflow作业，以运行给定的Spark作业。当与Airflow结合使用时，这始终是Spark批处理作业，而不是Spark流式作业，因为批处理作业是有限的，而流式作业可以永远运行。

1.2.3 配置为代码

通常，工作流管理系统可以分为两个阵营：以代码（例如Python）配置的系统和以静态文件（例如XML）配置的系统。Python代码允许使用例如FOR循环和其他编程构造生成动态和灵活的工作流。另一方面，静态配置在这个意义上不太灵活，并且通常遵循严格的模式。

作为代码的配置既可以是正的，也可以是负的；代码的动态特性可以为许多任务生成一段简洁的代码。然而，定义代码的方法有很多，因此对工作流的定义也不太严格。

1.2.4 调度与回填

气流工作流程可以通过多种方式启动：手动操作、外部触发或按计划的时间间隔启动。在所有方式中，每次启动工作流时都会运行一系列任务。只要您运行工作流，这可能就可以正常工作，但在某些时候，您可能需要更改逻辑。

对于图1.1中的Umbrella示例，假设您希望每日工作流在"Clean Weather Predictions"任务中以不同的方式清理数据。您可以更改工作流以生成这些新的已清理数据，并从现在开始运行新的工作流。但是，也可以在所有以前完成的工作流和所有历史数据上运行此新逻辑。这在Airflow中是可能的，它有一种称为回填的机制-及时运行工作流程。当然，只有在满足数据可用性等外部依赖性的情况下，这才是可能的。使回填特别有用的是重新运行部分工作流的能力。如果无法及时获取数据，或者您希望避免一个非常漫长的过程，则可以通过回填重新运行部分工作流。

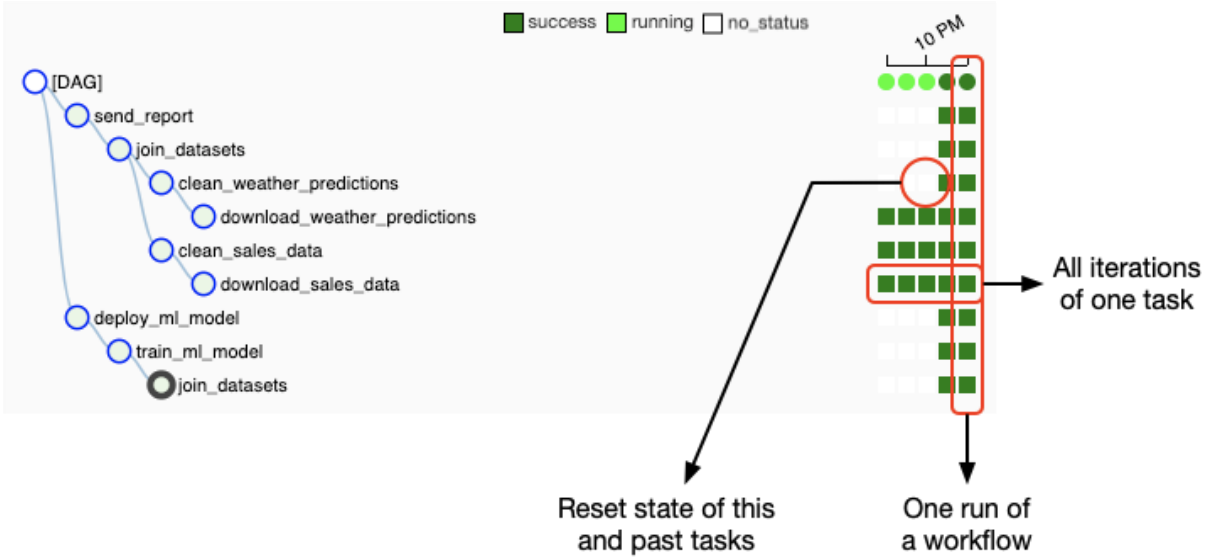


图1.7图1.1中的雨伞销售预测模型的执行随时间的变化。列显示工作流的一个执行的状态，行显示单个任务的所有执行的状态。红圈所覆盖的任务、其在工作流中的任务以及其过去的任务，都用单个操作进行了回填，并且正在进行中，需要再次执行。

回填保留任务的所有约束并运行它们，就像时间倒转到该时间点一样。对于每个任务，Airflow提供运行时上下文，并且当重新运行历史任务时，Airflow提供运行时上下文，就像时间被还原一样。除了回填之外，Airflow还提供了几个构造来管理任务和工作流的生命周期，这些任务和工作流在本书中都有介绍。

1.2.5 处理故障

在一个理想的世界里，所有的任务都能成功完成。然而，软件可能由于几乎任何原因而失败，因此我们需要自动处理这种失败的方法，因为

没人喜欢半夜被吵醒。有时失败是可以接受的，有时则不然。Airflow提供了各种处理故障的方法，例如，从声明"可以-继续运行其他任务"到"重试此失败任务最多5次"，再到"此处失败不可接受-整个管道失败"。如何以及何时处理故障的业务逻辑可能很复杂，而Airflow可以在任务和工作流级别处理故障。

1.3 为您的工作流程选择气流

想象一下，在您的公司中，您的任务是开发用于在多个系统之间处理数据的工作流。这类任务的几个例子可以是：

- 执行从MySQL数据库到分区配置单元表的每日数据转储
- 对存储在FTP服务器上的历史数据重新运行数据管道
- 计算昨天的数据报告，并根据星期几将报告通过电子邮件发送给适当的人员
- 根据每小时的数据转储重新培训和部署数据科学模型
- 在凌晨1点到5点之间等待文件到达，到达后立即处理并将其存储在AWS S3存储桶中

1.3.1 什么时候不使用气流？

我们之前提到过，由于DAG是在Python代码中定义的，而不是使用XML或JSON等更静态的定义，因此Airflow框架如何允许灵活和动态的工作流。两者都有各自的优点，例如，XML和JSON可以定义一个模式，这限制了更好地定义某些内容的可能方法的数量。尽管有约定，但Python代码不受任何限制的约束，实际上有无限多的方法来定义您的工作流。与XML在代码的可读性和简洁性方面的限制相比，这通常被认为是积极的。然而，它可以被视为一个负面因素，因为每个人都倾向于编写略有不同的Python代码。

"动态"这个词我们也提到了好几次，这个问题必须澄清。事实上，Airflow允许在Python代码中定义工作流，这可以产生简洁的工作流定义。例如，如果我们有一个包含5个表名的列表，我们可以创建一个任务来处理for循环中的每个表，只定义一次实际任务并避免代码重复。这就是我们所说的工作流的"动态性"。

在Airflow中，还有"动态DAG"的概念，它指的是导致不同行为的相同脚本，即当基于外部文件的内容生成任务集合时。虽然工作流脚本本身不会更改，但在工作流脚本中读取的外部文件内容可能会更改，从而导致工作流的不同行为。Airflow倾向于支持结构不是动态的工作流，这意味着每次运行时其结构都会发生变化的工作流在Airflow中不能很好地工作。

1.3.2 谁会觉得气流和这本书有用？

总而言之，我们希望读者是一个精通数据的人。数据场是不断增长的，因此永远不可能事先知道一切。但是，我们希望您不要害怕处理数据和编程。必须了解SQL（以及至少一种编程语言）的使用方法。

本书的前几章旨在介绍您作为开发人员必须了解的气流生态系统的基本组件。我们希望有一些Python经验，比如大约一年。这些章节应包括作为气流组件用户所需了解的所有内容。具备这些技能的典型职位是数据分析师和数据科学家。我们希望您具备以下基本知识：

- 数据库（例如，至少有使用关系型数据库（如MySQL/Postgres）的经验）
- 文件系统（例如，从FTP读取/写入文件）
- 数据类型（例如，知道如何与日期时间混合）
- Python（例如，了解参数、kwargs、列表理解和模板化）
- SQL语言

如果你有兴趣扩展自己的气流组件，这本书也会对你感兴趣。我们在这里看到的典型读者是具有更多Python经验的数据工程师。您有使用Python处理数据的经验，了解OOP、数据格式（例如Parquet/Avro）、数据结构（例如分区/分组）的概念，并且有使用Linux文件系统的经验。后续章节将介绍如何构建您自己的Airflow组件。

负责监控数据管道和维护气流安装的DevOps工程师和系统管理员也会发现这本书很有用。实际设置和维护气流的最佳实践将在后面的章节中介绍。我们涵盖了日志记录、监控、安全和以分布式方式设置气流。

1.4 启动并运行Airflow

为了让Airflow启动并运行，您可以在Python环境中安装Airflow或运行Docker容器。Docker Way是一行程序：

```
Docker运行-P 8080：8080 AirFlowBook/AirFlow
```

这需要在你的机器上安装一个Docker引擎。它将下载并运行Airflow Docker容器。运行后，您可以在http：//localhost：8080上查看气流。第二个选项是从PyPI安装Airflow并将其作为Python包运行：

```
PIP安装Apache-Airflow Airflow
Web服务器
```

再次浏览到http：//localhost：8080，您将看到气流的第一眼。此时，只有Web服务器已启动，并提供只读接口。启动调度程序以启动并运行完整的工作气流实例：

气流排程器

此时，您可以运行示例工作流，单击界面并查看正在运行和已完成任务的状态。在下一章中，我们将开始开发我们自己的。

1.5 总结

- 工作流管理系统处理工作流的执行和管理。
- 工作流可以表示为DAG，其中任务由节点描述，任务依赖关系由节点之间的有向边描述。
- DAG具有结束状态，因为它们不能包含循环。
- Airflow中的工作流被建模为DAG。
- Airflow中的任务可以在任何语言、软件或系统中运行。
- Airflow支持通过回填及时运行任务和（部分）工作流。

2

气流DAG的剖析

本章包括

- 在您自己的机器上实施气流设置
- 编写并运行您的第一个工作流
- 检查气流接口处的第一个视图
- 检测气流中的故障

在上一章中，我们了解了为什么在数据环境中使用数据和许多工具不是一件容易的事。在本章中，我们将开始使用Airflow，并查看一个示例工作流，该工作流使用了许多工作流中的基本构造块。在开始使用Airflow时，拥有一些Python经验会有所帮助。由于工作流是在Python代码中定义的，因此学习Airflow基础知识的差距并不大。使用Airflow启动和运行工作流通常并不是一项艰巨的任务；新手需要学习的概念数量很少。更复杂的部分是知道什么时候该做什么时候不该做某些选择。一些通常需要实践经验的东西。

2.1 跟踪火箭发射

火箭是人类的工程奇迹之一，每一次火箭发射都吸引着全世界的目光。在这一章中，我们讲述了一位名叫约翰的火箭爱好者的生活，他跟踪了每一次火箭发射。关于火箭发射的新闻可以在约翰跟踪的许多新闻来源中找到，理想情况下，约翰希望将他所有的火箭新闻聚合在一个位置。约翰最近学会了编程，并希望有某种自动化的方式来收集所有火箭发射的信息，并最终对最新的火箭新闻有某种个人见解。为了从小处着手，约翰决定首先收集火箭的图像。

2.1.1 启动库

对于数据，我们使用Launch Library[3]（https://launchlibrary.net），这是一个从各种来源收集的有关历史和未来火箭发射数据的在线存储库。它是一个免费开放的API，适用于地球上的任何人。约翰目前只对即将到来的火箭发射感兴趣。幸运的是，Launch Library正好提供了他在这个URL上寻找的数据：https://launchlibrary.net/1.4/launch?next=5&mode=verbose. 它提供了关于未来5次即将发射的火箭的数据，以及在哪里可以找到几枚火箭的图像的URL。此URL返回的数据片段：


```
{
   ""启动"": [
       {
           ""ID"": 1343,
           "名称": "Ariane 5 ECA|Eutelsat 7C&AT&T-16", "WindowStart":
           "2019年6月20日21: 43: 00 UTC",
           "窗口结束": "世界协调时2019年6月20日23: 30: 00",
           ...
          ""火箭": {
               ""ID"": 27,
               ""名称": ""阿丽亚娜5 ECA"",
               ...
              ""图像URL"": ""unk 1.JPG""
           },
           ...
       },
       {
           ""ID"": 1112,
           "名称": "Proton-M/Blok DM-03|Spektr-RG", "窗口开始": "世界
           协调时2019年6月21日12: 17: 14",
           "WindowEnd": "世界协调时2019年6月21日12: 17: 14",
           ...
          ""火箭": {
               ""ID"": 62,
               "名称": "质子-M/块DM-03",
               ...
              ""图像URL"": ""隐藏1.PNG""
           },
           ...
       },
        ...
   ],
   "总计": 202,
   "偏移量": 0,
   "计数": 5
}
```


[3] API文档：https://launchlibrary.net/docs/1.4/api.html

如您所见，数据采用JSON格式，提供火箭发射信息，每次发射都有一个字段“Rocket”，其中包含有关特定火箭的信息，如ID、名称和图像URL。这正是约翰所需要的，他最初制定了以下计划来收集图像：

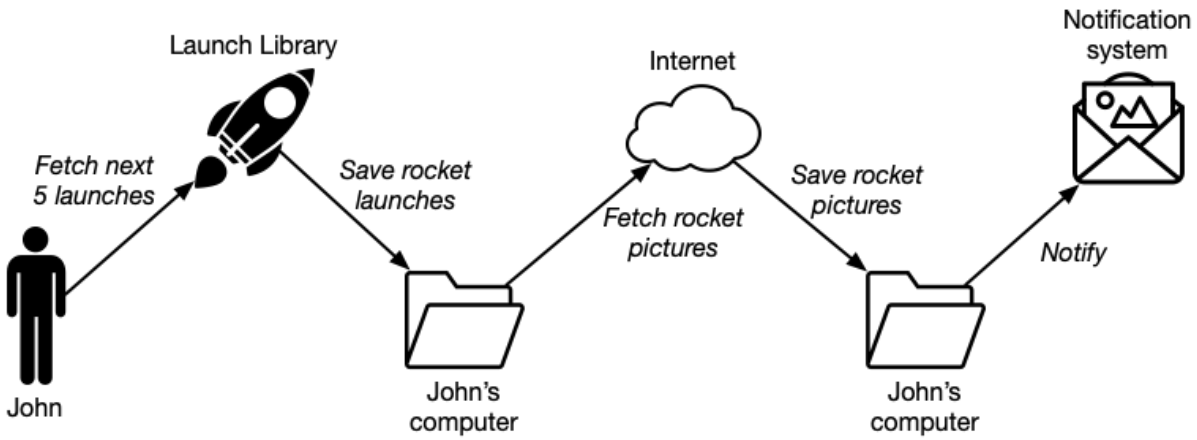


图2.1 John下载火箭图片的心理模型

在一天结束的时候，约翰的目标是有一个充满火箭图像的目录，例如阿丽亚娜5型ECA火箭的图像：

图2.2阿丽亚娜5型ECA火箭的示例图片

2.2 编写第一个Airflow DAG

John的用例很好地限定了范围，所以让我们看看如何对他的计划进行编程。这只是几个步骤，理论上，你可以用一行程序解决这个问题。那么，为什么我们需要一个像气流这样的系统来完成这项工作呢？

Airflow的好处在于，我们可以将由一个或多个步骤组成的大型作业拆分为单独的"任务"，并共同形成一个"DAG"。DAG中的每个任务负责一项工作。可以并行运行多个任务，可以在DAG中的任何时间点重新启动任务，并且任务可以执行不同类型的工作，因此我们可以首先运行bash脚本，然后运行Python脚本。我们制定了John在Airflow DAG中的使用案例：

```
导入JSON导入
PathLib

导入气流导入请求
从气流导入DAG
从airflow.operators.bash_operator导入bashOperator从
airflow.operators.python_operator导入pythonOperator

DAG=DAG（DAG_ID="下载火箭发射"，
```

```
    start_date=airflow.utils.dates.days_ago (14) ,
    schedule_interval=无,
)

download_launches =
    BashOperator( task_id="download_launches",
    bash_command="curl -o /tmp/launches.json
        "1.4/启动"？下一步=5&mode=verbose"", dag=dag,
)


def _get_pictures():
    #确保目录存在
    pathlib.path ("/TMP/images") .mkdir (parent=true, exist_OK=true)

    #下载launches.JSON中的所有图片
    使用open ("/TMP/launches.JSON") 作为f:
        launches=JSON.load (f)
        IMAGE_URLs=[Launch["Rocket"]["ImageUrl"]for Launch in Launches["Launches"]for IMAGE_URLs in
        IMAGEURL:
            response=requests.get (image_URL)
            image_filename=image_URL.split ("/") [-
            1]target_file=f"/TMP/images/{image_filename}"使用open
             (target_file, "WB") 作为f:
                f.write (响应.内容)
            打印 (f"已将{image_URL}下载到{target_file}")


get_pictures=pythonOperator
     (task_ID=""get_pictures"",
    python_callable=_get_pictures,
    dag=DAG,
)

NOTIFY=BASHOPERATOR
     (task_ID=""notify"",
    bash_command='echo""现在有$ (ls/TMP/images/|WC-l) 映像。"", DAG=DAG,
)
```

我们在Airflow中将John工作流程的心理模型分解为三个任务：

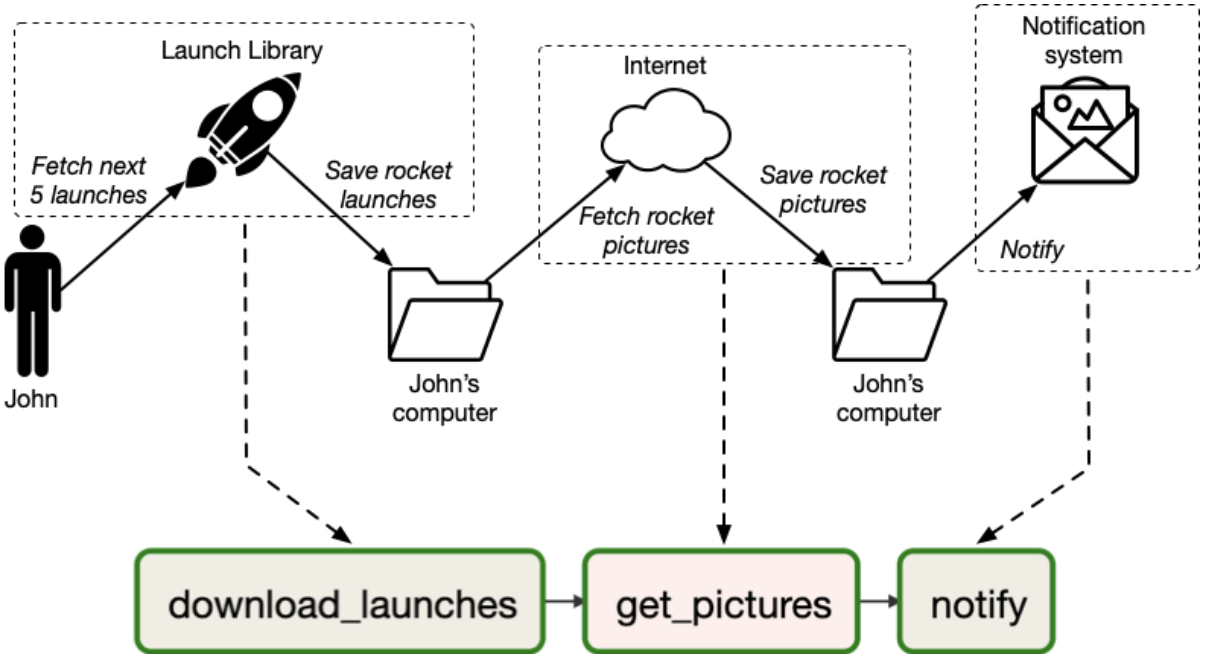


图2.3 John的心理模型映射到Airflow中的任务

你可能会问，为什么要做这三件事？你可能想知道，为什么不在一个单一的任务中下载发射和相应的图片呢？或者为什么不分成五个任务呢？毕竟，在约翰的计划中我们有五支箭。这些都是在开发工作流程时要问自己的有效问题，但事实是没有正确或错误的答案。不过，有几点需要考虑，在这本书中，我们研究了许多这样的用例，以了解什么是对的，什么是错的。让我们首先分解一下工作流程。

每个工作流都从一个DAG对象开始。这是任何工作流的主要起点，没有DAG就没有工作流。

```
DAG=DAG (DAG_ID="下载火箭发射",
    start_date=airflow.utils.dates.days_ago (14) ,
    schedule_interval=无,
)
```

DAG是任何工作流的起点；工作流中的所有任务都引用此DAG对象，以便Airflow知道哪些任务属于哪个DAG。DAG类采用两个必需的参数：

1. DAG_ID：显示在Airflow UI中的DAG的名称
2. START_DATE：工作流首次开始运行的日期时间

接下来，Airflow工作流脚本由一个或多个执行实际工作的操作员组成。在这种情况下，我们可以（例如）使用bashOperator来运行bash命令：

```
download_launches =
   BashOperator( task_id="download_launches",
   bash_command="curl -o /tmp/launches.json
      "l.4/启动"？下一步=5&mode=verbose"", dag=dag,
)
```

每个操作员执行单个工作单元，多个操作员一起在Airflow中形成工作流或DAG。运算符彼此独立运行，但您可以定义依赖关系。毕竟，如果您第一次尝试下载图片时还没有关于图片位置的数据，那么John的工作流程就不会有用。为了确保任务以正确的顺序运行，我们可以按如下方式设置任务之间的依赖关系：

```
下载_启动>>获取_图片>>通知
```

这确保了get_pictures任务仅在download_launches成功完成后运行，而notify任务仅在get_pictures成功完成后运行。

2.2.1 任务与操作员

您可能想知道任务和操作符之间的区别。毕竟，它们都会执行一些代码。在Airflow中，操作员只有一项职责：他们的存在就是为了执行一项工作。一些操作符执行通用工作，如BashOperator（运行bash脚本）和Python操作符（运行Python函数），其他操作符具有更具体的用例，如EmailOperator（发送电子邮件）或HttpOperator（调用HTTP端点）。无论哪种方式，他们都只做一件工作。

DAG的作用是协调操作符集合的执行。这包括操作员的启动和停止、操作员完成后启动连续任务、确保满足操作员之间的依赖关系等。

在本文和整个Airflow文档中，我们看到术语"操作员"和"任务"可互换使用。从用户的角度来看，它们指的是同一件事，并且在讨论中两者经常相互替换。操作符提供了实际的实现，Airflow有一个名为BaseOperator的类，以及许多继承自BaseOperator的子类，如BashOperator和PythonOperator。

不过，这是有区别的。Airflow中的任务管理操作员的执行；它们可以被认为是操作符周围的小型"包装器"或"管理器"，用于确保操作符正确执行。通过使用和创建操作符，用户可以专注于要完成的工作，而Airflow则通过任务确保正确执行工作：

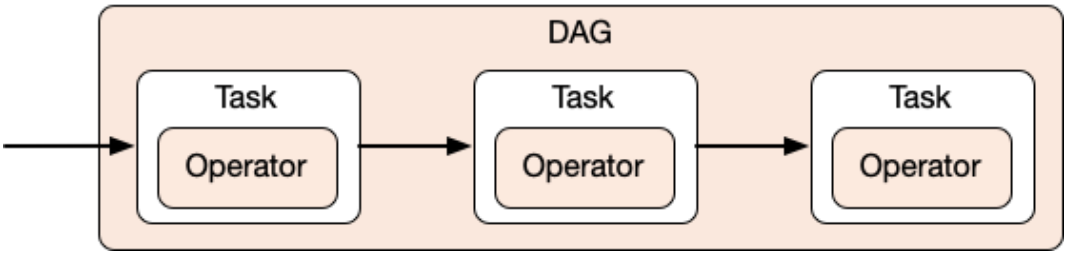


图2.4 DAG和操作员由Airflow用户使用，任务是管理操作员状态的内部元素。

2.2.2 运行任意Python代码

为接下来的5次火箭发射获取数据是Bash中的一个curl命令，它很容易使用BashOperator执行。然而，解析JSON结果、从中选择图像URL并下载相应的图像需要付出更多的努力。尽管所有这些在bash一行程序中仍然是可能的，但使用Python或您选择的任何其他语言的几行代码通常更容易且更具可读性。由于Airflow代码是用Python定义的，因此将工作流和执行逻辑保存在同一个脚本中非常方便。为了下载火箭图片，我们执行了以下操作：

```
def _get_pictures():
    #确保目录存在
    pathlib.path ("/TMP/images") .mkdir (parent=true, exist_OK=true)

    #下载launches.JSON中的所有图片
    使用open ("/TMP/launches.JSON") 作为f:
        launches=JSON.load (f)
        IMAGE_URLs=[Launch["Rocket"]["ImageUrl"]for Launch in Launches["Launches"]for IMAGE_URLs in
        IMAGEURL:
            response=requests.get (image_URL)
            image_filename=image_URL.split ("/") [-
            1]target_file=f"/TMP/images/{image_filename}"使用open
             (target_file, "WB") 作为f:
                f.write (响应.内容)
            打印 (f"已将{image_URL}下载到{target_file}")


get_pictures=pythonOperator
    (task_ID="get_pictures",
    python_callable=_get_pictures,
```

Airflow中的PythonOperator负责运行任何Python代码。就像之前使用的BashOperator一样，这个操作符和所有其他操作符都需要一个task_ID。task_ID在运行任务时引用，并显示在UI中。

PythonOperator的使用始终是双重的：

1. 我们定义操作符本身（get_pictures）和

2. 这个 python_callable参数 点数 向 a 可调用， 典型地 a 函数 (_get_pictures)

当运行操作符时，将调用Python函数并执行该函数。让我们来分析一下。PythonOperator的基本用法通常如下所示：

```
def _get_pictures():
    # do work here...

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

PythonOperator callable

PythonOperator

虽然不是必需的，但为了方便起见，我们将变量名称"get_pictures"与task_ID保持一致。

```
#确保目录存在
pathlib.path ("/TMP/images") .mkdir (parent=true, exist OK=true)
```

Callable中的第一步是确保存储图像的目录存在。接下来，我们打开从Launch Library API下载的结果，并提取每次启动的图像URL：

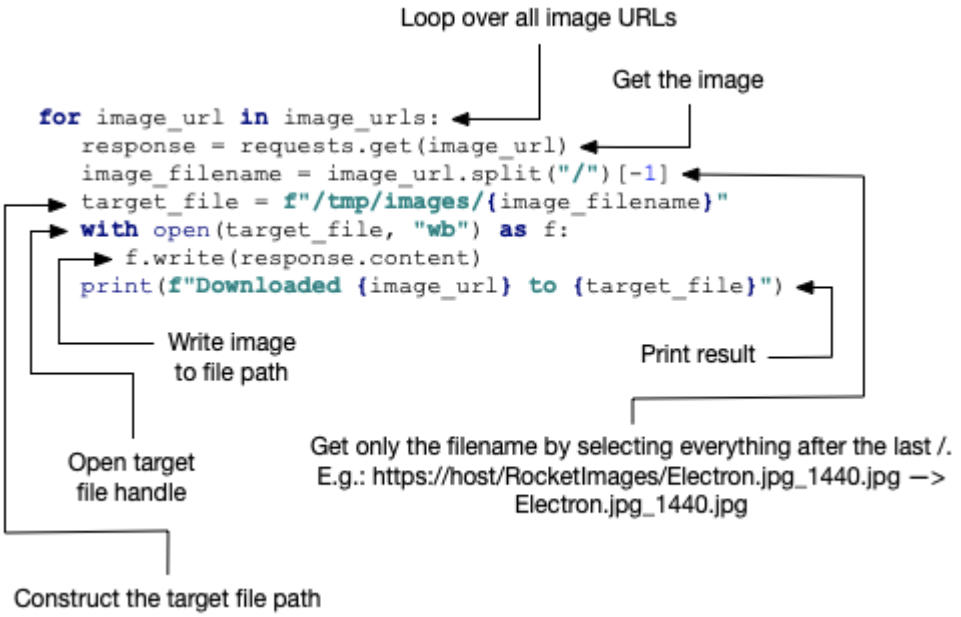
Open the rocket launches JSON

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

Read as a dict so we can mingle the data

For every launch, fetch the field rocket -> imageURL

调用每个图像URL以下载图像并将其保存在/TMP/images中：



2.3 在Airflow中运行DAG

现在我们有了基本的火箭发射DAG，让我们启动并运行它，并在Airflow UI中查看它。Bare Minimum Airflow由两个核心组件组成：（1）调度器和（2）Web服务器。要启动并运行Airflow，您可以在Python环境中安装Airflow或运行Docker容器。Docker Way是一行程序：

```
Docker运行-P 8080：8080 AirFlowBook/AirFlow
```

这需要在你的机器上安装一个Docker引擎。它将下载并运行Airflow Docker容器。运行后，您可以在http：//localhost：8080上查看气流。第二个选项是从PyPI安装Airflow并将其作为Python包运行：

```
PIP安装Apache-Airflow
```

确保您安装的是Apache-Airflow，而不仅仅是Airflow。随着2016年加入Apache基金会，PyPI Airflow存储库更名为Apache-Airflow。由于许多人仍在安装Airflow，而不是删除旧的存储库，它被保留为一个虚拟对象，以便为每个人提供指向正确存储库的消息。

现在您已经安装了Airflow，通过初始化MetaStore（存储所有Airflow状态的数据库），将Rocket Launch DAG复制到DAGS目录，并启动Scheduler和Web服务器来启动它：

```
气流INITDB
cp download_rocket_launches.py ~/airflow/dags/
```

```
Airflow Scheduler
Airflow Web服务器
```

请注意，Scheduler和WebServer都是保持终端打开的连续进程，因此可以在后台使用Airflow Scheduler运行，也可以打开第二个终端窗口分别运行Scheduler和WebServer。设置完成后，浏览到"http：//localhost：8080"以查看气流。

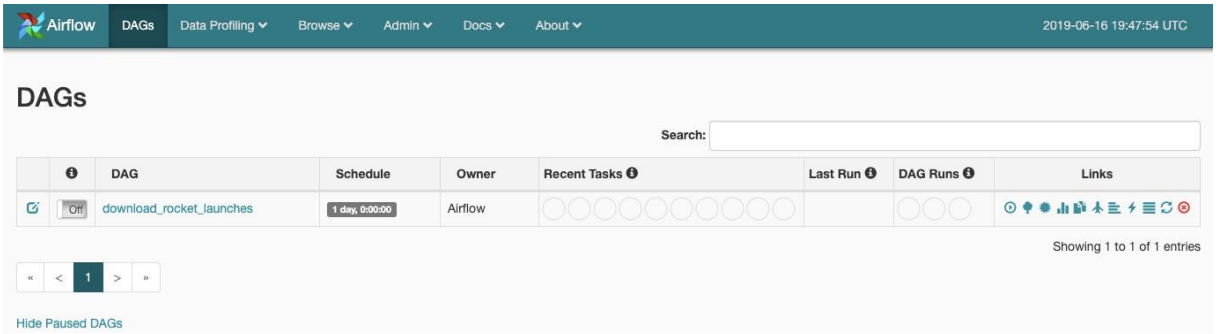


图2.5气流主屏幕

这是你将看到的气流的第一眼。目前，唯一的DAG是DAGS目录中Airflow可用的download_rocket_launches。在主视图上有很多信息，但是让我们首先检查download_rocket_launches DAG。单击DAG名称将其打开并检查所谓的图形视图：

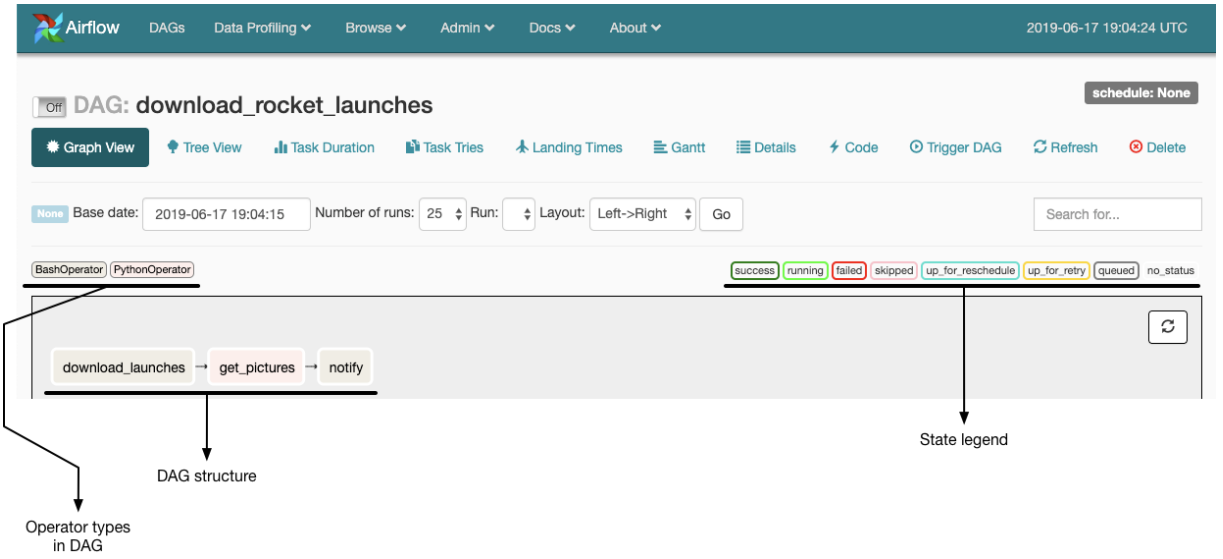


图2.6气流图视图

此视图向我们显示了提供给Airflow的DAG脚本的结构。一旦放置在DAG目录中，Airflow将读取脚本并提取出一起形成DAG的零碎内容，以便可以在UI中可视化。图形视图向我们显示了DAG的结构、DAG中所有任务的连接方式和运行顺序。这可能是您在开发工作流程时使用最多的视图之一。

状态图例显示您在运行时可能看到的所有颜色，因此让我们看看发生了什么并运行DAG。首先，DAG需要"打开"才能运行，为此切换"关闭"按钮。接下来，单击"Trigger DAG"以运行它。

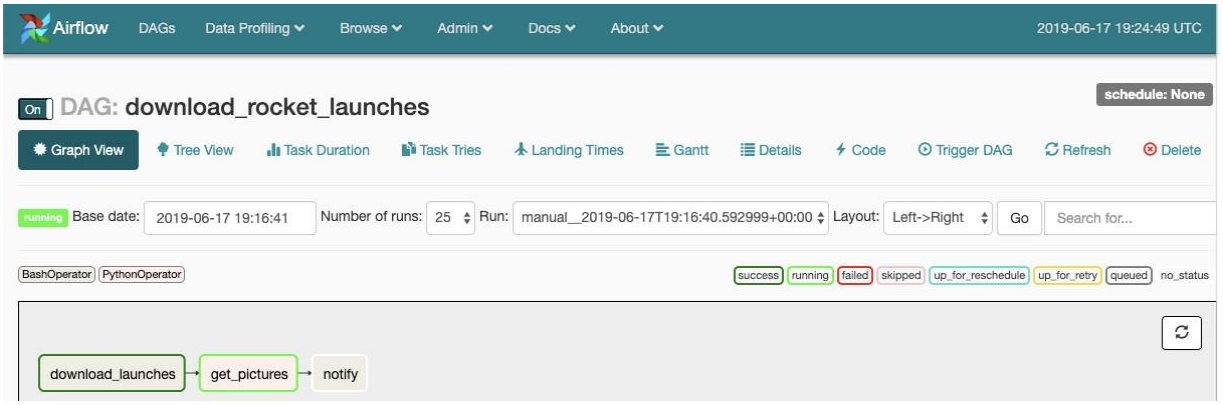


图2.7显示运行DAG的图形视图

触发DAG后，它将开始运行，您将看到由颜色表示的工作流的当前状态。由于我们在任务之间设置了依赖关系，因此只有在前面的任务完成后，连续的任务才会开始运行。让我们检查"通知"任务的结果。在实际使用案例中，您可能需要发送电子邮件或Slack通知来通知有关新映像的信息。为了简单起见，它现在打印已下载图像的数量。让我们检查一下日志。

所有任务日志都收集在Airflow中，因此我们可以在UI中搜索输出或故障情况下的潜在问题。单击已完成的"通知"任务，您将看到一个包含多个选项的弹出窗口：

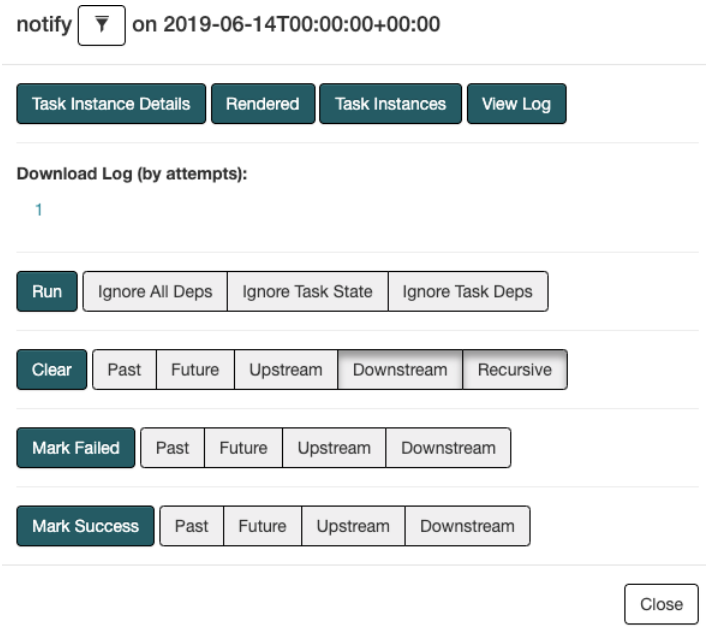


图2. 8任务弹出选项

单击右上角的"查看日志"按钮以检查日志：

```
*** Reading local file: /root/airflow/logs/download_rocket_launches/notify/2019-06-18T19:06:28.102026+00:00/1.log
[2019-06-18 19:06:58,698] {__init__.py:1139} INFO - Dependencies all met for <TaskInstance: download_rocket_launches.notify 2019-06-18T
[2019-06-18 19:06:58,705] {__init__.py:1139} INFO - Dependencies all met for <TaskInstance: download_rocket_launches.notify 2019-06-18T
[2019-06-18 19:06:58,705] {__init__.py:1353} INFO -
--------------------------------------------------------------------------------
[2019-06-18 19:06:58,705] {__init__.py:1354} INFO - Starting attempt 1 of 1
[2019-06-18 19:06:58,705] {__init__.py:1355} INFO -
--------------------------------------------------------------------------------
[2019-06-18 19:06:58,715] {__init__.py:1374} INFO - Executing <Task(BashOperator): notify> on 2019-06-18T19:06:28.102026+00:00
[2019-06-18 19:06:58,716] {base_task_runner.py:119} INFO - Running: ['airflow', 'run', 'download_rocket_launches', 'notify', '2019-06-1
[2019-06-18 19:06:59,871] {base_task_runner.py:101} INFO - Job 85: Subtask notify [2019-06-18 19:06:59,871] {__init__.py:51} INFO - Us
[2019-06-18 19:07:00,126] {base_task_runner.py:101} INFO - Job 85: Subtask notify [2019-06-18 19:07:00,126] {__init__.py:305} INFO - F
[2019-06-18 19:07:00,153] {base_task_runner.py:101} INFO - Job 85: Subtask notify [2019-06-18 19:07:00,152] {cli.py:517} INFO - Running
[2019-06-18 19:07:00,165] {bash_operator.py:81} INFO - Tmp dir root location:
 /tmp
[2019-06-18 19:07:00,165] {bash_operator.py:90} INFO - Exporting the following env vars:
AIRFLOW_CTX_DAG_ID=download_rocket_launches
AIRFLOW_CTX_TASK_ID=notify
AIRFLOW_CTX_EXECUTION_DATE=2019-06-18T19:06:28.102026+00:00
AIRFLOW_CTX_DAG_RUN_ID=manual__2019-06-18T19:06:28.102026+00:00
[2019-06-18 19:07:00,165] {bash_operator.py:104} INFO - Temporary script location: /tmp/airflowtmptdhnydwi/notify3obku1dp
[2019-06-18 19:07:00,165] {bash_operator.py:114} INFO - Running command: echo "There are now $(ls /tmp/images/ | wc -l) images."
[2019-06-18 19:07:00,173] {bash_operator.py:123} INFO - Output:
[2019-06-18 19:07:00,177] {bash_operator.py:127} INFO - There are now 5 images.
[2019-06-18 19:07:00,177] {bash_operator.py:131} INFO - Command exited with return code 0
[2019-06-18 19:07:03,692] {logging_mixin.py:95} INFO - [2019-06-18 19:07:03,692] {jobs.py:2562} INFO - Task exited with return code 0
```

图2.9打印日志中显示的语句

默认情况下，日志非常详细，但会在输出日志中显示下载的映像数量。最后，我们可以打开/TMP/images目录并查看它们：

图2.10生成的火箭图片

2.4 每隔一定时间运行一次

火箭爱好者约翰很高兴，现在他有一个工作流程，并在气流中运行，他可以不时地触发收集最新的火箭图片。他可以在Airflow UI中看到工作流的状态，与他之前在命令行上运行的脚本相比，这已经是一种改进。但是他仍然需要手动触发他的工作流程

有时应该是自动化的。毕竟，没有人喜欢做计算机擅长的重复性工作。

在Airflow中，我们可以安排DAG以特定的时间间隔运行，例如，每小时、每天或每月运行一次。这是通过在DAG上设置schedule_interval参数来控制的：

```
DAG=DAG（DAG_ID="下载火箭发射"，
    start_date=airflow.utils.dates.days_ago（14），
    schedule_interval="@daily"，
）
```

将SCHEDULE_INTERVAL设置为"@DAILY"将告知Airflow每天运行一次此工作流，这样John就不必每天手动触发一次。最好在树视图中查看此行为：

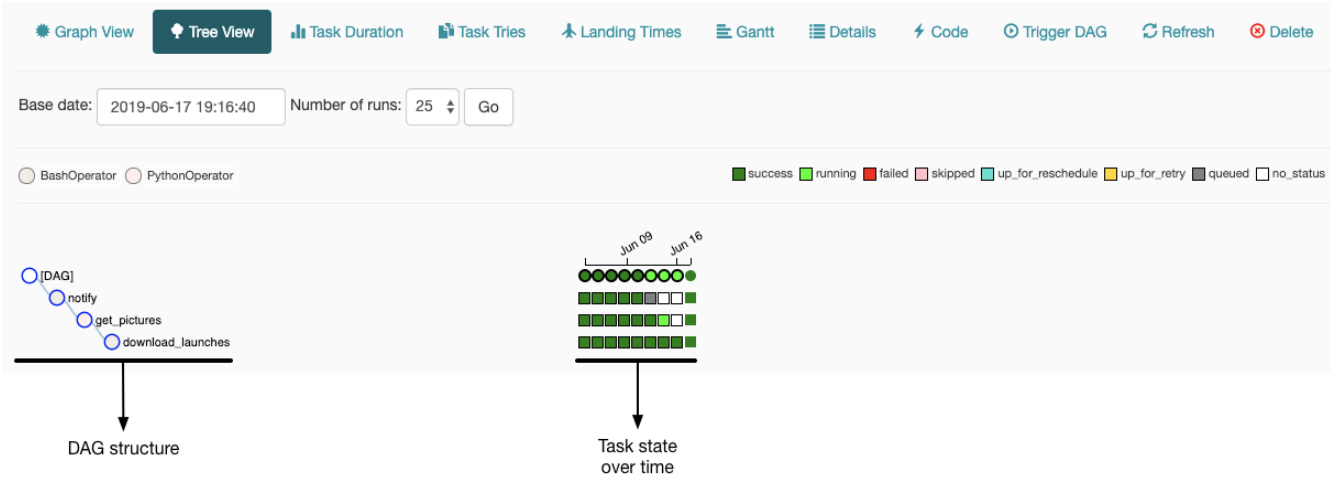


图2.11气流树视图

树视图类似于图形视图，但在超时运行时显示图形结构。可以在此处查看单个工作流的所有运行状态的概述。

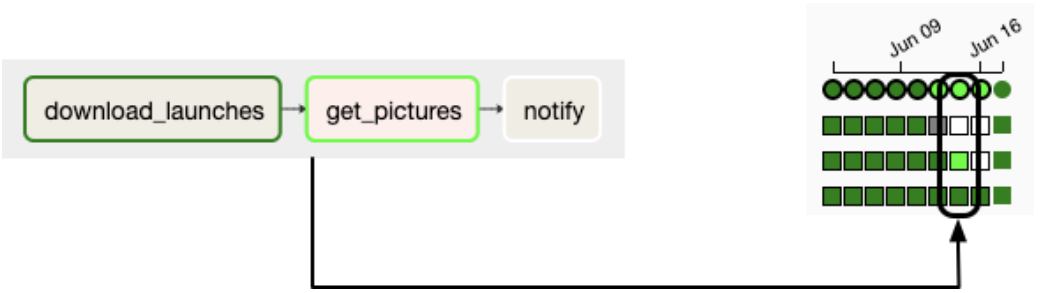


图2.12图视图与树视图的关系

DAG的结构显示为适合"行和列"布局，特别是特定DAG的所有运行的状态，其中每一列表示某个时间点的单个运行。

当我们将schedule_interval设置为"@daily"时，Airflow知道它必须每天运行一次此DAG。假设提供给DAG的start_date为14天前，这意味着从14天前到现在的时间可以划分为14个相等的1天间隔。由于这14个间隔的开始和结束日期时间都在过去，因此一旦我们向airflow提供schedule_interval，它们就会开始运行。第3章将更详细地介绍调度间隔的语义和配置调度间隔的各种方法。

2.5 处理失败的任务

到目前为止，我们在Airflow UI中只看到了绿色。但如果失败了会发生什么呢？任务失败的情况并不少见，这可能是由于多种原因造成的，例如外部服务中断、网络连接问题或磁盘损坏。假设我们在获取图片时遇到了网络问题，我们会在Airflow中看到一个失败的任务，如下所示：

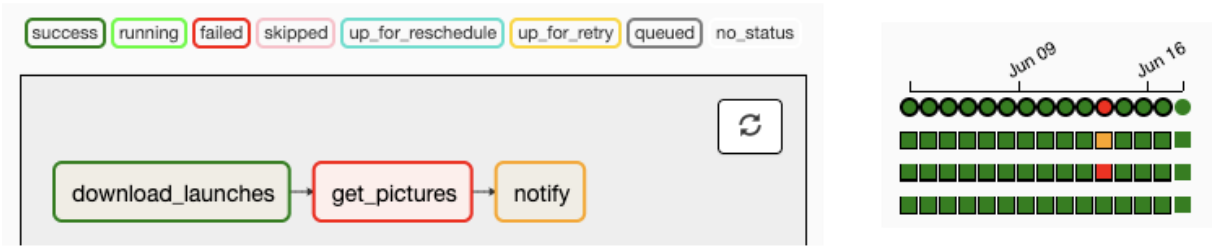


图2.13图形和树视图中显示的故障

由于无法从互联网获取图像并因此引发错误，特定的失败任务将在图形和树视图中显示为红色。这个

连续的"通知"任务根本不会运行，因为它依赖于"get_pictures"任务的成功状态。此类任务实例以橙色显示。默认情况下，所有先前的任务必须成功运行，失败任务的任何后续任务将不会运行。

让我们通过再次检查日志来解决这个问题。打开"get_pictures"任务的日志：

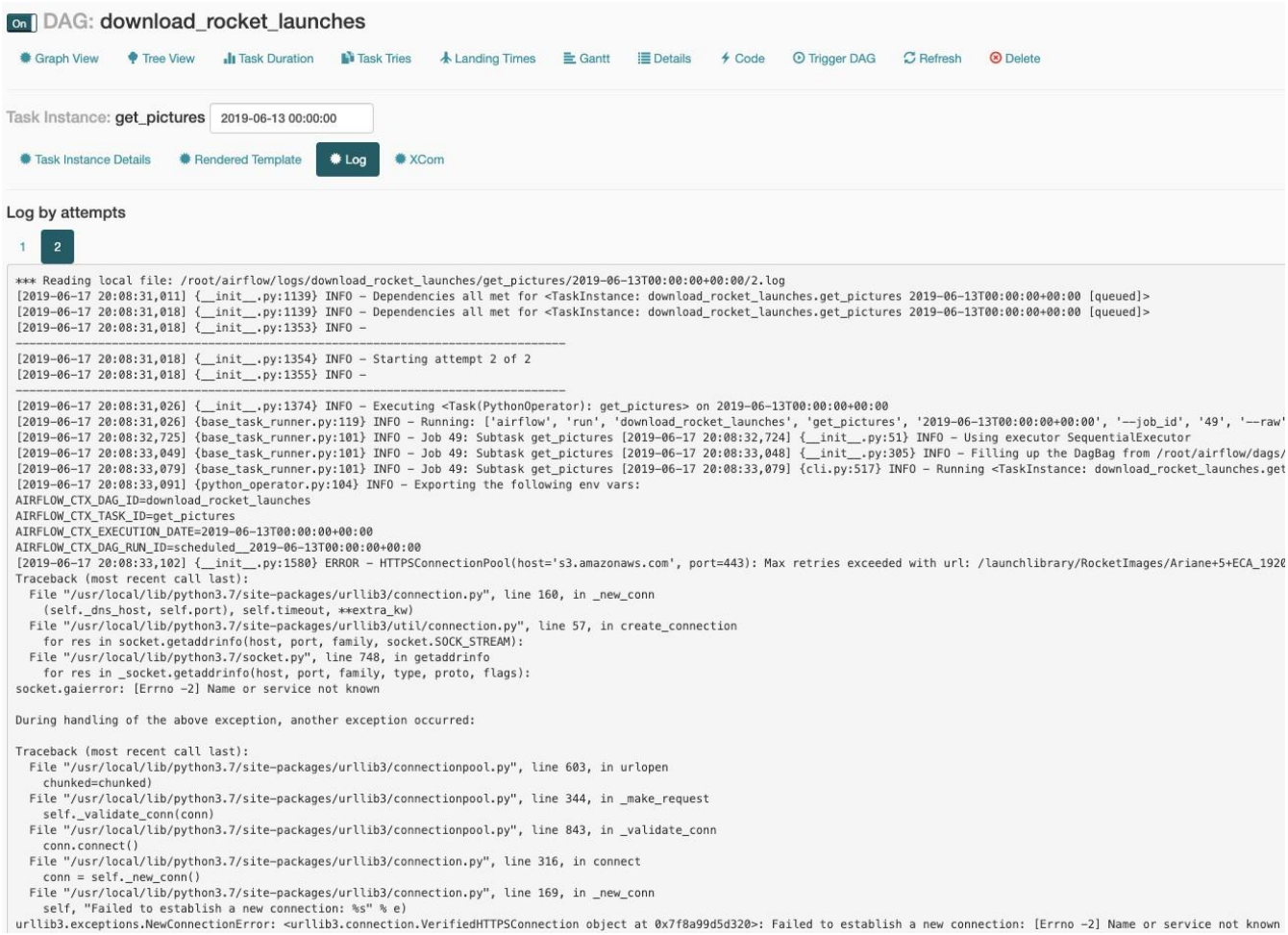


图2.14失败的get_pictures任务的堆栈跟踪

在堆栈跟踪中，我们发现了问题的潜在原因：

```
Urllib3.Exceptions.NewConnectionError：<Urllib3.Connection.VerifiedHttpsConnection对象位于
        0x7f8a99d5d320>：无法建立新连接：[errno-2]名称或服务未知
```

这表示urllib3正在尝试建立连接，但无法建立，这可能暗示防火墙规则阻止连接或没有Internet连接。现在我们发现了问题，并假定已修复了该问题，让我们重新启动该任务。没有必要重新启动整个工作流，Airflow的一个很好的功能是，您可以从故障点重新启动，而不必重新启动任何以前成功的任务。

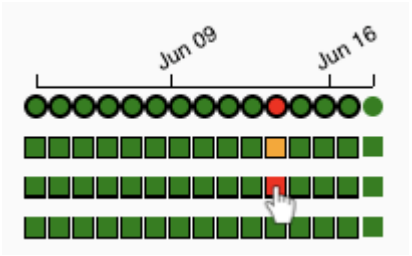


图2.15单击失败的任务可获得清除该任务的选项

单击失败的任务，然后单击弹出窗口中的"清除"按钮。它将显示您将要清除的任务。这意味着您将"重置"这些任务的状态，Airflow将重新运行这些任务：

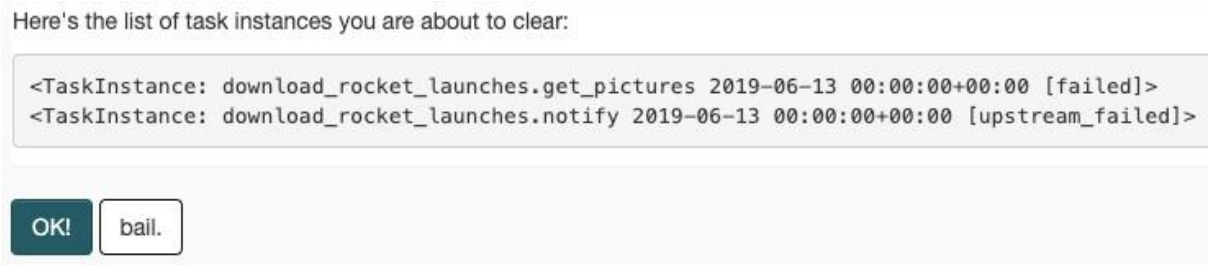


图2.16清除get_pictures和后续任务的状态

单击"确定！"失败的任务及其后续任务将被清除：

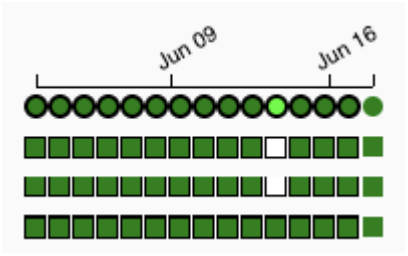


图2.17图形视图中显示的已清除任务

如果互联网连接问题得到解决，任务现在将成功运行，并使整个树视图变为绿色：

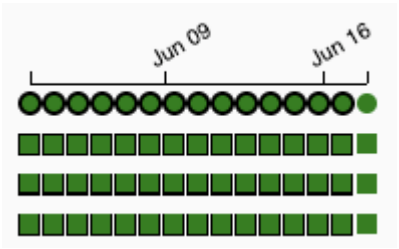


图2.18清除失败任务后成功完成的任务

在任何一款软件中，失败的原因都有很多。在Airflow工作流中，有时接受失败，有时不接受失败，有时仅在特定条件下接受失败。失败时的处理标准可以在工作流中的任何级别上配置，在第4章中有更详细的介绍。

2.6 总结

- Airflow中的工作流以DAG表示。
- 运算符表示单个工作单元。
- Airflow包含一系列用于一般和特定类型工作的操作符。
- Airflow UI提供了用于查看DAG结构的图形视图，以及用于查看DAG超时运行的树视图。
- 失败的任务可以在DAG中的任何位置重新启动。

3

在AirFlow中调度

本章包括

- 使用计划间隔定期运行DAG
- 构建以增量方式加载和处理数据的高效DAG
- 设计DAG以使用回填重新处理过去的数据集

在上一章中，我们研究了Airflow的UI，并向您展示了如何定义基本的Airflow DAG，以及如何通过定义计划间隔每天运行此DAG。在本章中，我们将更深入地探讨Airflow中的调度概念，并探讨这如何使您能够定期增量处理数据。首先，我们将介绍一个小型用例，重点分析我们网站上的用户事件，并探讨如何构建DAG来定期分析这些事件。接下来，我们将探讨如何通过采用增量方法来分析我们的数据，以及如何将其与Airflow的执行日期概念联系起来，从而使这一过程更加高效。最后，我们将展示如何使用回填来填补数据集中过去的空白，并讨论适当的气流任务的一些重要属性。

3.1 定期运行任务

正如我们在第2章中所看到的，通过为DAG定义计划间隔，可以定期运行Airflow DAG。在构建DAG时，可以使用schedule_interval参数定义计划间隔。默认情况下，此参数的值为none，这意味着不会计划DAG，并且仅在从UI或API手动触发时才会运行DAG。在本节中，我们将探讨几种不同类型的计划间隔，并研究它们如何影响DAG的计划。

3.1.1 用例：处理用户事件

为了理解Airflow的调度是如何工作的，我们将首先考虑一个小例子。想象一下，我们有一项服务，可以跟踪用户在我们网站上的行为，并允许我们查看用户（通过IP地址标识）在我们网站上访问的页面。出于营销目的，我们想知道我们的用户访问了多少不同的页面，以及他们在每次访问中花费了多少时间。为了了解这种行为如何随着时间的推移而变化，我们希望每天计算这些统计数据，因为这样可以比较不同日期和更长时间段内的变化。

出于实际原因，外部跟踪服务不会存储超过30天的数据。这意味着我们需要自己存储和积累这些数据，因为我们想要更长时间地保留我们的历史。通常，由于原始数据可能相当大，因此将这些数据存储在云存储服务（如Amazon的S3或Google的云存储服务）中是有意义的，因为这些服务将高持久性与相对较低的成本结合在一起。然而，为了简单起见，我们不会担心这些事情，并将我们的数据保存在本地。

为了模拟这个例子，我们创建了一个简单的（本地）API，它允许我们检索用户事件。例如，我们可以使用以下API调用检索过去30天内可用事件的完整列表：

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

该调用返回一个（JSON编码的）用户事件列表，我们可以对其进行分析以计算用户统计信息。

使用此API，我们可以将工作流分解为两个单独的任务：一个用于获取用户事件，另一个用于计算统计信息。数据本身可以使用BashOperator下载，其方式与我们在上一章中看到的类似。为了计算统计数据，我们可以使用Python操作符，该操作符允许我们将数据加载到Pandas DataFrame中，并使用GroupBy和聚合计算事件的数量：

```
def calculate_stats (input_path, output_path) : “”计算
        事件统计信息。””events=PD.read_JSON (输入路
        径)
        stats = events.groupby(["date", "user"]).size().reset_index()
        stats.to_csv(output_path, index=False)
```

总之，这为我们的工作流提供了以下DAG：

```
从气流导入DAG
从airflow.operators.bash_operator导入bashoperator从
airflow.operators.python_operator导入pythonoperator从datetime导入
datetime, timedelta

将Pandas导入为PD

DAG=DAG (
      DAG_ID=“”用户事件“”,
      start_date=日期时间 (2015, 6, 1) ,
```

```
)

fetch_events=bashoperator
        (task_ID=""fetch_events"",
       bash_command=""curl-o data/events.JSON https: //localhost: 5000/events"", dag=DAG,
)

def_calculate_stats (input_path, output_path) : ""计算
       事件统计信息。""events=PD.read_JSON (输入路径)
       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={
    "输入路径": "数据/事件.JSON", "输出路
    径": "数据/状态.CSV"
},
       dag=dag,
)

获取_事件>>计算_统计
```

从Airflow的角度来看，这应该会产生一个包含两个任务的DAG，如图3.1所示。

图3.1。我们的事件用例的基本DAG。在此用例中，我们首先使用bash命令（fetch_events）从外部API加载事件，然后在Python中的几个统计信息（calculate_stats）中汇总这些事件。原则上，我们的想法是每天运行这个DAG，以随着时间的推移积累每日统计数据的数据集。

现在我们有了基本的DAG，但我们仍然需要确保它通过气流定期运行。让我们把它安排好，这样我们就有了每日更新！

3.1.2 定义计划间隔

在我们接收用户事件的示例中，我们希望每天计算统计数据，这表明安排DAG每天运行一次是有意义的。由于这是一种常见的使用情形，Airflow提供了方便的宏"@Daily"，用于定义每天午夜运行一次DAG的每日计划间隔：

```
dag = DAG(
       DAG_ID="用户事件",
       SCHEDULE_INTERVAL="@每日",
       ...
)
```

然而，我们还没有完全完成。为了让Airflow知道应该从哪一天开始计划DAG运行，我们还需要提供DAG的开始日期。根据此开始日期，Airflow将计划DAG的第一次执行，以在开始日期（开始+间隔）之后的第一个计划间隔运行。后续运行将在此第一个间隔之后的计划间隔继续执行。

例如，假设我们将DAG的开始日期定义为1月1日：

```
将日期时间作为DT导入

dag = DAG(
       DAG_ID="用户事件",
       SCHEDULE_INTERVAL="@每日",
       start_date=DT.dateTime（年=2019，月=1，日=1）
)
```

结合每日调度间隔，这将导致Airflow在1月1日之后的每天午夜运行我们的DAG（图3.2）。请注意，我们的第一次执行发生在1月2日（开始日期之后的第一个间隔），而不是1月1日。我们将在本章后面进一步探讨这种行为背后的原因。

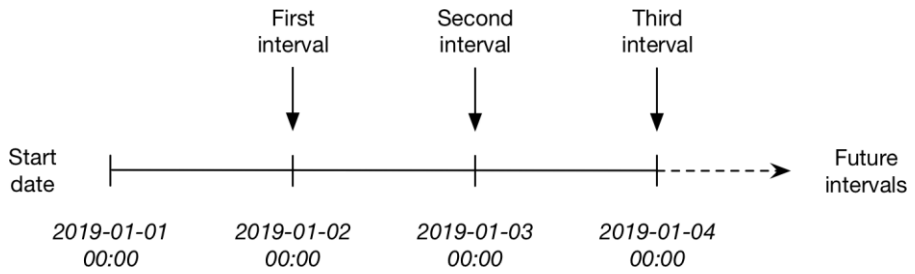


图3.2。具有指定开始日期的每日计划DAG的计划间隔。这将显示开始日期为2019-01-01的DAG的每日时间间隔。箭头指示执行DAG的时间点。如果没有指定的结束日期，DAG将保持每天执行，直到DAG关闭。

在没有结束日期的情况下，Airflow将（原则上）继续按此每日计划执行我们的DAG，直到时间结束。但是，如果我们已经知道项目具有固定的持续时间，则可以使用"end_date"参数告诉Airflow在特定日期后停止运行DAG：

```
dag = DAG(
       DAG_ID="用户事件",
       SCHEDULE_INTERVAL="@每日",
       start_date=DT.dateTime（年=2019，月=1，日=1），
       end_date=DT.dateTime（年=2019，月=1，日=5）
```

这将产生图3.3所示的全套计划间隔。

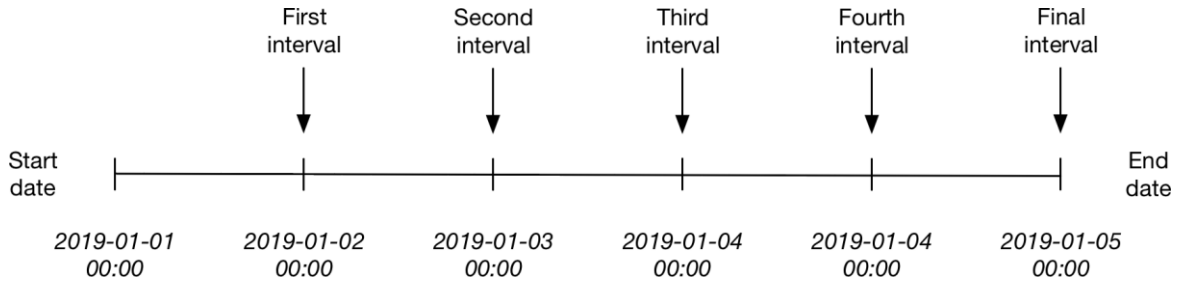


图3.3。具有指定开始和结束日期的每日计划DAG的计划间隔。显示与图3.2相同的DAG的时间间隔，但现在的结束日期为2019-01-05，这将阻止DAG在此日期之后执行。

原则上，确保您的开始/结束日期与您的计划间隔保持一致是一种很好的做法。例如，对于每日计划间隔，使用2019-01 00：00：00（午夜）的开始日期比使用2019-01 05：50：00的开始日期更有意义，因为前者与每日计划间隔（计划在午夜运行）很好地保持一致。但是，如果使用未对齐的开始日期，Airflow将自动对齐开始日期和计划间隔，方法是使用开始日期作为开始查找下一个计划间隔的时刻。在实践中，这意味着两种情况（对齐和非对齐）都将导致气流在2019-01-02 00：00：00执行其第一个间隔。

除静态日期外，还可以在Airflow中使用开始/结束日期的动态值。例如，通过将start_date设置为类似"Airflow.utils.dates.days_ago（14）"的内容，这在Airflow教程中经常用于运行过去X天的DAG。尽管此功能对于测试或示例非常有用，但在实践中，我们建议不要为DAG使用动态开始/结束日期，因为它们可能会造成混乱。此外，某些组合可能会导致您的DAG永远不会被触发，如果它们导致气流永远不会完成计划间隔。例如，将"DateTime.Now（）"的开始日期与每小时调度间隔相结合将导致DAG永远不会被触发，因为DAG永远不会到达开始日期之后的一小时，这与"Now"一起移动。

3.1.3 基于Cron的间隔

到目前为止，我们的所有示例都显示DAG以每日间隔运行。但是，如果我们希望以每小时或每周为间隔运行作业，该怎么办呢？对于更复杂的时间间隔，例如，我们可能希望在每周六23：45运行DAG？

为了支持更复杂的调度间隔，Airflow允许我们使用与Cron相同的语法来定义调度间隔，Cron是类Unix计算机操作系统（如MacOS和Linux）使用的基于时间的作业调度程序。该语法由5个组件组成，定义如下：

```
##————————分钟（0-59）#|————————
小时（0-23）
#||————————月之日（1-31）#|||————————月（1-
12）
#||||————星期几（0-6）（星期日至星期六；# | | | | |    7在某些系统上也
是星期日）
```

在此定义中，当时间/日期规范字段与当前系统时间/日期匹配时，将执行cron作业。可以使用星号（'*'）代替数字来定义不受限制的字段，这意味着我们不关心该字段的值。

尽管这种基于cron的表示法可能看起来有点复杂，但它为我们定义时间间隔提供了相当大的灵活性。例如，我们可以使用以下cron表达式定义每小时、每天和每周的时间间隔：

- 0****=每小时（整点运行）
- 00***=每日（午夜运行）
- 00**0=每周（周日午夜运行）

除此之外，我们还可以定义更复杂的表达式，例如：

- 001**=每月第一天的午夜
- 4523**星期六=每周六23：45

此外，Cron表达式允许您使用逗号（','）定义值列表或使用破折号（'-'）定义值范围来定义值集合。使用此语法，我们可以构建表达式，以支持在多个工作日或一天中的多个小时组运行作业：

- 00**周一、周三、周五=每周一、周三、周五午夜运行
- 00**周一至周五=在每个工作日的午夜运行
- 00，12***=每天上午00：00和下午12：00运行

Airflow还提供了对几个宏的支持，这些宏表示常用调度间隔的简写。我们已经看到了其中一个用于定义每日间隔的宏（@daily）。Airflow支持的其他宏的概述如表3.1所示。

表3.1。经常使用的计划间隔的气流预设。

预调	意义	等价的Cron表达式
无（不是字符串）	不要计划DAG。用于外部触发的DAG。	-
@一次	计划一次且仅一次。	-
@每小时	每小时运行一次，在整点开始的时候。	`0 * * * *`
@每日	每天午夜跑一次。	`0 0 * * *`
@每周	每周一次，在周日凌晨的午夜跑步。	`0 0 * * 0`
@每月	每月运行一次，在每月第一天的午夜运行。	`0 0 1 * *`
@每年	每年1月1日午夜运行一次。	`0 0 1 1 *`

尽管cron表达式非常强大，但它们可能很难使用。因此，在气流中尝试之前，先测试一下你的表情可能是个好主意。幸运的是，有许多工具（例如https://crontab.guru），它可以帮助您用简单的英语定义、验证或解释您的cron表达式。在代码中记录复杂的cron表达式背后的推理也没有坏处。这可能会帮助其他人（包括未来的你！）在重新访问代码时理解表达式。

3.1.4 基于频率的间隔

Cron表达式的一个重要限制是它们无法表示某些基于频率的计划。例如，如何定义每三天运行一次DAG的cron表达式？事实证明，你可以写一个表达式，运行在每1，4，7，等等。但这种方法在月底会遇到问题，因为DAG将在下个月的31日和1日连续运行，违反了所需的计划。

cron的这种局限性源于cron表达式的性质，因为cron表达式定义了一种模式，该模式不断与当前时间进行匹配，以确定是否应该执行作业。这样做的好处是使表达式无状态，这意味着您不必记住上一个作业何时运行来计算下一个时间间隔。然而，正如您所看到的，这是以牺牲一些表现力为代价的。

那么，如果我们真的想按三天一次的计划运行DAG，该怎么办呢？

为了支持这种基于频率的计划，Airflow还允许您根据相对时间间隔来定义计划间隔。要使用这种基于频率的计划，您可以传递"TimeDelta"实例（来自标准库中的DateTime模块）作为计划间隔：

```
从日期时间导入时间增量

dag = DAG(
    DAG_ID="用户事件"，计划间隔=时间增量（天
    数=3）
```

```
    start_date=DT.dateTime（年=2019，月=1，日=1）
）
```

这将导致我们的DAG在开始日期之后每三天运行一次（在4号、7号、10号等）。2019年1月）。当然，您也可以使用此方法每10分钟（使用timedelta（minutes=10））或每2小时（使用timedelta（hours=2））运行一次DAG。

3.1.5 何时使用哪个区间表达式？

鉴于Airflow至少有三种不同的方法来定义调度间隔，您可能想知道何时使用哪种类型的表达式来定义调度间隔（预设、cron或基于频率）。原则上，我们建议使用最简单的表达式来适合您的情况，因为这将产生最具可读性的结果。在实践中，这意味着我们建议在可能的情况下使用Airflow的预设而不是Cron表达式，因为这些预设通常被认为更容易阅读。当然，如果您需要比预置提供的粒度更细的粒度，请毫不犹豫地使用cron表达式（但请考虑在代码中添加简短的解释性注释）。

基于频率的表达式通常用于覆盖cron不支持的间隔。但是，在许多情况下，它们可能与等效的预设或cron表达式一样具有可读性。例如，使用"TimeDelta（Days=1）"的计划间隔并不比使用@Daily预设的可读性差很多。但是，需要注意的是，这两个计划间隔不一定相同，具体取决于使用的开始日期。在非舍入开始日期（如2019-01-01 05：00：00）的情况下，基于TimeDelta的表达式将导致DAG在开始日期之后的整整一天（即2019-01-02 05：00：00）触发。相反，@Daily预设将触发DAG在午夜运行，这意味着DAG将在2019-01-02 00：00：00触发。根据您的意图，最好记住基于频率的计划和cron/预设计划之间的细微差别。

3.2 以增量方式处理数据

尽管我们现在以每天为间隔运行DAG（假设我们坚持使用@Daily计划），但我们还没有完全实现我们的目标。首先，我们的DAG每天都在下载和计算整个用户事件目录的统计数据，这几乎没有效率。此外，此过程仅下载过去30天的事件，这意味着我们不会为过去的日期建立任何历史记录。

解决这些问题的一种方法是将我们的DAG更改为仅获取和处理一天的数据。这样，我们只处理每天收到的新数据，而不会浪费时间重新计算前几天的统计数据，从而节省了宝贵的计算资源。这种类型的方法通常称为增量处理，因为它涉及在构建数据集时为每个计划间隔处理数据的小增量。

在本节中，我们将探讨如何重写事件DAG以增量加载数据。在此过程中，我们将向您展示气流调度间隔如何定义数据增量，以及增量处理对数据存储方式的影响。

3.2.1 以增量方式获取事件

要将我们的事件DAG转换为使用增量方法，我们需要重写我们的DAG以获取事件并计算每天的统计信息，仅针对相应日期的事件。实现这一点的一种方法是重写获取任务，使其仅获取一天的事件，并将其写入包含当天所有事件的输出JSON文件。随后，聚合（Calculate_Statistics）任务可以读入该天的JSON文件（events/day1.JSON），并使用该文件中的数据来计算该特定天的统计信息，还可以将其写入包含该天统计信息的输出文件（stats/day1.CSV，图3.4）。

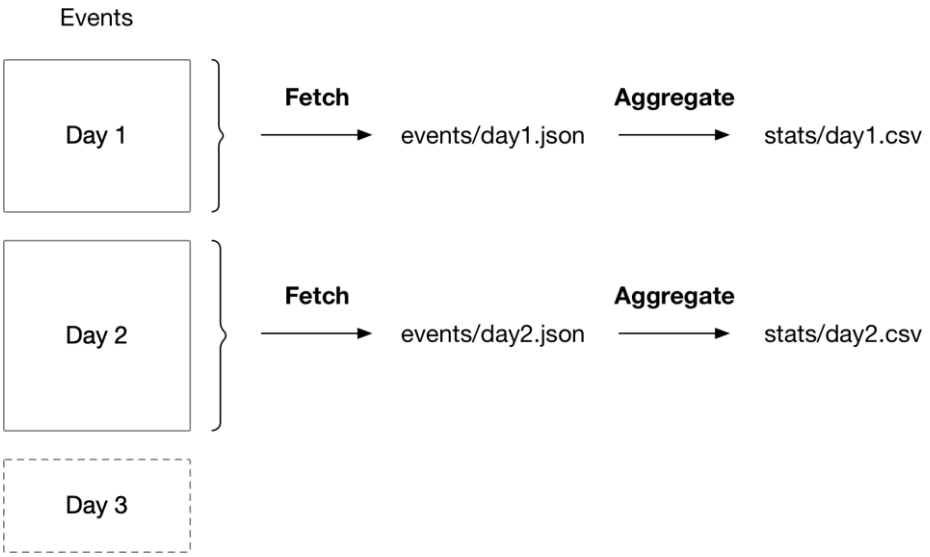


图3.4。以增量方式获取和处理事件。通过将DAG调整为仅获取和聚合一天的事件，可以将事件DAG转换为以增量方式加载和处理事件。通过每天重复这种对日常数据的增量处理，DAG实现了与以前相同的结果，而不必处理整个数据集。请注意，每日任务需要将每天的事件写入单独的文件，以便我们知道在哪里可以找到一天的数据。

如前所述，这种增量方法比获取和处理整个数据集的效率高得多，因为它显著减少了每个计划间隔中必须处理的数据量。此外，由于我们现在每天都将数据存储在单独的文件中，因此我们也有机会随着时间的推移开始构建文件的历史记录，远远超过了API的30天限制。

要在工作流中实现增量处理，我们需要修改DAG以下载特定日期的数据。幸运的是，我们可以通过包含开始和结束日期参数来调整API调用，以获取当前日期的事件：

```
旋度-O http: //localhost: 5000/events？开始日期=2019-01-01&结束日期=2019-01-02
```

这两个日期参数共同指示我们要获取事件的时间范围。请注意，在此示例中，start_date是包含性的，而end_date是排他性的，这意味着我们有效地获取了在2019-01-01 00：00：00和2019-01-01 23：59之间发生的事件。

我们可以在DAG中实现这种增量数据获取，方法是将bash命令更改为包含两个日期：

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

但是，要获取除2019-01-01以外的任何其他日期的数据，我们需要更改命令以使用反映DAG执行日期的开始和结束日期。幸运的是，Airflow为我们提供了几个额外的参数，我们将在下一节中对此进行探讨。

3.2.2 使用执行日期的动态时间引用

对于许多涉及基于时间的流程的工作流，了解给定任务的执行时间间隔非常重要。因此，Airflow为任务提供了额外的参数，这些参数可用于确定执行任务的计划间隔。

这些参数中最重要的是execution_date，它表示执行DAG的日期和时间。与参数名称所暗示的相反，EXECUTION_DATE不是日期，而是时间戳，它反映了DAG正在执行的计划间隔的开始时间。计划间隔的结束时间由另一个名为next_execution_date的参数指示。这两个日期一起定义了任务计划间隔的整个长度（图3.5）。

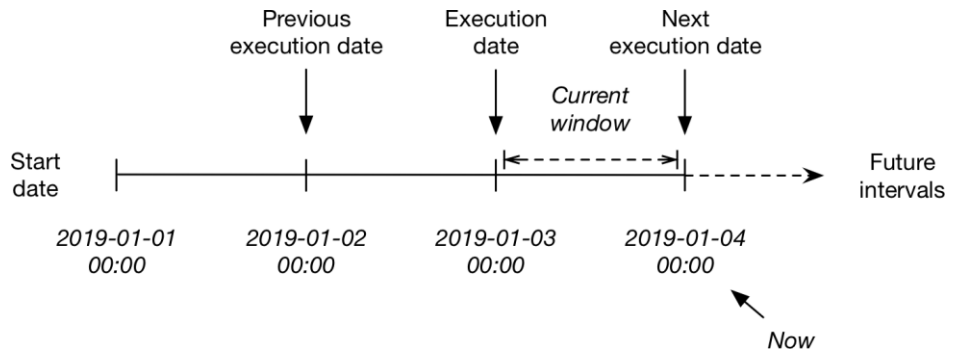


图3.5。Airflow中执行日期的定义。箭头指示在Airflow上下文中定义的执行日期，以及这些日期与当前执行窗口的关系。注意，执行日期是指执行窗口的开始，而下一执行日期是指下一执行日期（=当前执行日期+调度间隔）。因此，当前执行窗口在当前执行日期和下一个执行日期之间运行。

除这两个参数外，Airflow还提供了PREVIOUS_EXECUTION_DATE参数，该参数描述了上一个计划间隔的开始时间。虽然我们不会在这里使用此参数，但它对于执行将当前时间间隔的数据与前一时间间隔的结果进行对比的分析非常有用。

在Airflow中，我们可以通过在运算符中引用这些执行日期来使用它们。例如，在BashOperator中，我们可以使用Airflow的模板化功能将执行日期动态包含在bash命令中：

```
fetch_events =
      BashOperator( task_id="fetch_events",
      bash_command="curl -o data/events.json
      http: //localhost: 5000/events？开始日期={{执行日期.strfTime (%y-%m-
      %d') }&end_date={{next_execution_date.strftime ('%y-%m-%d') }", DAG=DAG,
)
```

在此示例中，语法{{variable_name}}是使用基于Jinja的AirFlows的示例[4] 用于引用Airflow的特定参数之一的模板语法。在这里，我们使用此语法来引用两个执行日期，并使用DateTimes strfTime方法将它们格式化为预期的字符串格式（因为两个执行日期都是DateTime对象）。

由于执行日期参数通常以这种方式用于将日期引用为格式化的字符串，因此Airflow还为常见的日期格式提供了几个简写参数。例如，DS和DS_NODASH参数是

[4] http://jinja.pocoo.org/

执行日期，格式分别为YYYY-MM-DD和YYYYMMDD。类似地，NEXT_DS、NEXT_DS_NODASH、Prev_DS和Prev_DS_NODASH分别提供下一个和上一个执行日期的简写。

使用这些简写，我们还可以按如下方式编写增量获取命令：

```
fetch_events =
       BashOperator( task_id="fetch_events",
       bash_command="curl -o data/events.json
       http: //localhost: 5000/events？开始日期={DS}&结束日期={next_DS}″, DAG=DAG,
)
```

这个较短的版本更容易阅读。但是，对于更复杂的日期（或日期时间）格式，您可能仍然需要使用更灵活的strftime方法。

3.2.3 对数据进行分区

尽管我们的新FETCH_EVENTS任务现在为每个新的计划间隔增量地获取事件，但精明的读者可能已经注意到，每个新任务只是覆盖前一天的结果，这意味着我们实际上没有建立任何历史记录。

解决此问题的一种方法是简单地将新事件附加到events.JSON文件，这将允许我们在单个JSON文件中构建历史记录。然而，这种方法的一个缺点是，它需要任何下游处理作业来加载整个数据集，即使我们只对计算某一天的统计数据感兴趣。此外，它还使此文件成为单点故障，如果此文件丢失或损坏，我们可能会面临丢失整个数据集的风险。

另一种方法是通过将任务的输出写入具有相应执行日期名称的文件，将数据集划分为每日批处理：

```
fetch_events=bashoperator
        (task_ID="″fetch_events"",
       bash_command="″curl-o data/events/{{DS}.JSON http: //localhost: 5000/events？开始日
       期={DS}&结束日期={next_DS}″, DAG=DAG,
)
```

这将导致下载的执行日期为2019-01-01的任何数据被写入文件data/2019-01-01.JSON。

这种将数据集划分为更小、更易于管理的部分的做法是数据存储和处理系统中的一种常见策略。这种做法通常称为分区，数据集的较小部分称为分区。

当我们考虑DAG中的第二个任务（Calculate_Stats）时，按执行日期对数据集进行分区的优势变得显而易见，在该任务中，我们计算每天用户事件的统计信息。在我们以前的实现中，我们每天加载整个数据集并计算整个事件历史的统计信息：

```
def_calculate_stats (input_path, output_path) : "″计算
       事件统计信息。"″
```

```
        events=PD.read_JSON (输入路径)
        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={
    "输入路径": "数据/事件.JSON", "输出路
    径": "数据/状态.CSV"
},
        dag=dag,
```

但是，使用我们的分区数据集，通过将此任务的输入和输出路径更改为指向分区事件数据和分区输出文件，我们可以更高效地为每个单独的分区计算这些统计信息：

```
def_calculate_stats (**context) : ""计算事件统计信
        息。""

        input_path=context["templates_dict"]["input_path"]output_path=c
        ontext["templates_dict"]["' output_path'"']

        events=PD.read_JSON (输入路径)
        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={
    "输入路径": "数据/事件/{DS}.JSON", "输出路径":
    "数据/统计信息/{DS}.CSV"
},
provide_context=True,
```

尽管这些更改可能看起来有些复杂，但它们主要涉及样板代码，以确保我们的输入和输出路径是模板化的。为了在PythonOperator中实现这种模板化，我们需要传递任何应该使用运算符templates_dict参数（lines...）进行模板化的参数。.为了在我们的Python函数中检索这些模板化参数，我们需要从任务上下文中检索它们的值（行..和..）在确保上下文被传递给我们的函数之后（行...）.

如果这一切进行得有点太快，不要担心-我们将在下一章中更详细地深入到任务上下文。这里需要理解的重点是，这些变化允许我们通过每天只处理数据的一小部分来逐步计算统计数据。

3.3 了解Airflow的执行日期

由于执行日期是Airflow的重要组成部分，让我们花点时间来确保我们完全理解这些日期是如何定义的。

正如我们所看到的，我们可以通过三个参数控制Airflow何时运行DAG：开始日期、计划间隔和（可选）结束日期。要实际开始计划我们的DAG，Airflow使用这三个参数将时间划分为一系列计划间隔，从给定的开始日期开始，并可选择在结束日期结束（图3.6）。

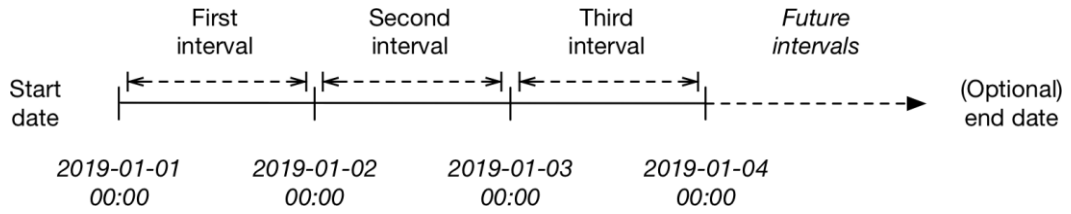


图3.6。以气流的调度间隔表示的时间。假设开始日期为2019-01-01的每日间隔。计划时间间隔从DAG开始日期到可选的结束日期。

在该基于间隔的时间表示中，只要给定间隔的时隙已经过去，就针对该给定间隔执行DAG。例如，图3.6中的第一个间隔将在2019-01-01 23：59：59之后尽快执行，因为此时间隔中的最后一个时间点已经过去。类似地，DAG将在2019-01-02 23：59：59之后不久执行第二个间隔，依此类推，直到到达可选的结束日期。

使用这种基于间隔的方法的一个优点是，它非常适合执行我们在前面几节中看到的增量数据处理类型，因为我们确切地知道任务执行的时间间隔-相应间隔的开始和结束。这与基于时间点的调度系统（如Cron）形成了鲜明的对比，在基于时间点的调度系统中，我们只知道执行任务的当前时间。这意味着，例如在cron中，我们必须计算或"猜测"上一次执行停止的位置，例如假设任务在前一天执行（图3.7）。

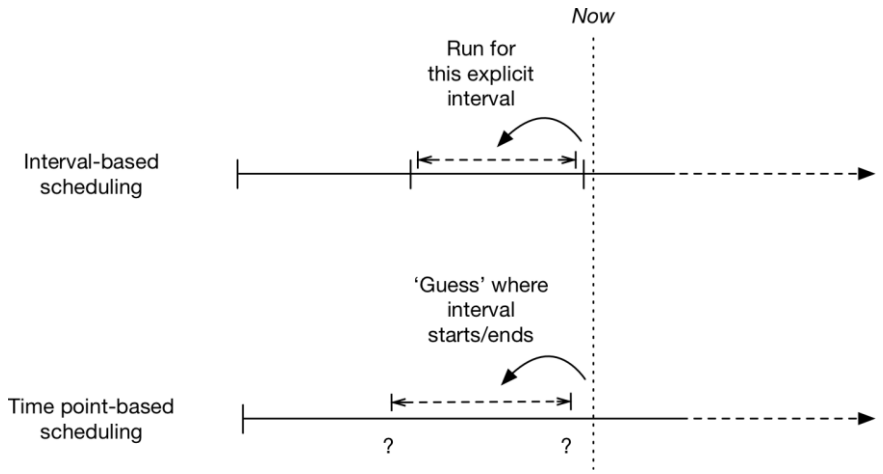


图3.7。基于间隔的调度窗口（如Airflow）中的增量处理与基于时间点的系统（如Cron）中的窗口。对于增量（数据）处理，时间通常被划分为离散的时间间隔，一旦经过相应的时间间隔，就对其进行处理。基于间隔的调度方法（如Airflow）明确地调度任务在每个间隔运行，同时向每个任务提供关于间隔开始和结束的确切信息。相反，基于时间点的调度方法只在给定的时间执行任务，让任务自己来确定执行任务的增量间隔。

了解Airflow对时间的处理是围绕计划间隔构建的，还有助于了解如何在Airflow中定义执行日期。例如，假设我们有一个遵循每日计划间隔的DAG，并考虑应处理2019-01-03这一天的数据的相应间隔。在Airflow中，此间隔将在2019-01-04 00：00：00之后不久运行，因为在该时间点，我们知道我们将不再接收2019-01-03这一天的任何新数据。回想一下我们在上一节中对在任务中使用执行日期的解释，您认为对于此间隔，EXECUTION_DATE的值是多少？

许多人期望的是，此DAG运行的执行日期将是2019-01-04，因为这是DAG实际运行的时刻。但是，如果我们在执行任务时查看EXECUTION_DATE变量的值，我们实际上会看到执行日期为2019-01-03。这是因为Airflow将DAG的执行日期定义为相应间隔的开始。从概念上讲，如果我们考虑到执行日期标记我们的计划间隔，而不是DAG实际执行的时刻，这是有意义的。不幸的是，命名可能有点混乱。

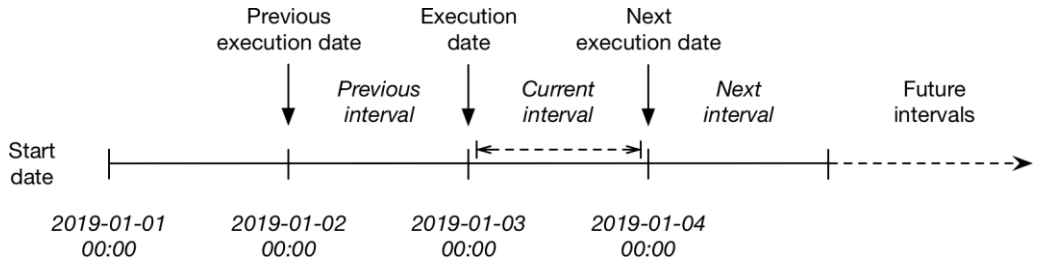


图3.8。计划间隔上下文中的执行日期。在Airflow中，DAG的执行日期定义为相应计划间隔的开始时间，而不是执行DAG的时间（通常是间隔的结束时间）。因此，execution_date的值指向当前间隔的开始，而previous_execution_date和next_execution_date参数分别指向上一个和下一个调度间隔的开始。当前时间间隔可以从EXECUTION_DATE和NEXT_EXECUTION_DATE的组合中得出，这表示下一个时间间隔的开始，因此也表示当前时间间隔的结束。

气流执行日期定义为相应计划间隔的开始日期，可用于推导特定间隔的开始和结束日期（图3.8）。例如，在执行任务时，相应间隔的开始和结束由EXECUTION_DATE（间隔的开始）和NEXT_EXECUTION DATE（下一个间隔的开始）参数定义。类似地，可以使用previous_execution_date和execution_date参数来导出先前的调度间隔。

但是，在任务中使用PREVIOUS_EXECUTION_DATE和NEXT_EXECUTION_DATE参数时需要注意的一点是，这些参数仅为遵循计划间隔的DAG运行定义。因此，对于使用Airflow UI或CLI手动触发的任何运行，这些参数的值将是未定义的。这样做的原因是，如果您不遵循计划间隔，则Airflow无法为您提供有关下一个或上一个计划间隔的信息。

3.4 使用回填来填补过去的空白

由于Airflow允许我们定义从任意开始日期开始的计划间隔，因此我们还可以定义从过去的开始日期开始的过去间隔。我们可以使用此属性来执行DAG的历史运行，以加载或分析过去的数据集-这一过程通常称为回填（图3.9）。

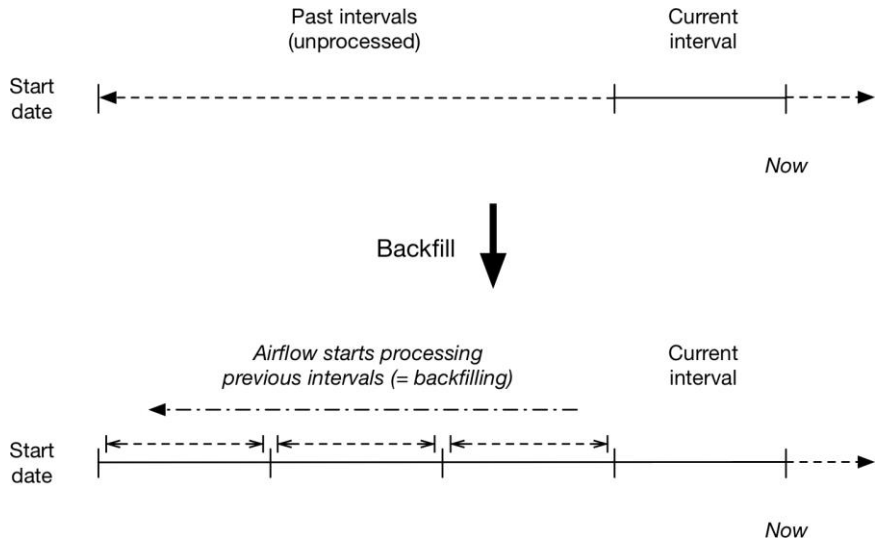


图3.9。使用回填来填充过去的间隙。由于Airflow定义计划间隔的方式，它还能够为过去的计划间隔执行运行。为计划间隔执行运行通常称为回填，因为回填允许您填充数据集中的（过去的）漏洞。

默认情况下，Airflow将计划并运行尚未运行的任何过去的计划间隔。因此，指定过去的开始日期并激活相应的DAG将导致执行在当前时间之前经过的所有间隔。此行为由DAG CatchUp参数控制，可通过将CatchUp设置为false来禁用：

```
dag = DAG(
    DAG_ID="用户事件", SCHEDULE_INTERVAL=时间增量（天数=3），
    START_DATE=DT.DATETIME（年份=2019，月份=1，日期=1），
    CATCHUP=假
)
```

使用此设置，DAG将仅在最近的计划间隔运行，而不是执行所有打开的过去的间隔。通过设置CATCHUP_BY_DEFAULT配置设置的值，可以从Airflow配置文件中控制CATCHUP的默认值。

尽管回填是一个强大的概念，但它受到源系统中数据可用性的限制。例如，在我们的示例用例中，我们可以通过指定过去30天内的开始日期来从API加载过去的事件。但是，由于API最多只能提供30天的历史记录，因此我们无法使用回填来加载较早日期的数据。

在我们对代码进行更改后，回填还可用于重新处理数据。例如，假设我们对_calc_statistics函数进行了更改，以添加新的统计信息。使用回填，我们可以清除过去运行的calc_statistics任务，以重新分析历史数据

使用新代码的数据。请注意，在这种情况下，我们不受数据源的30天限制，因为我们已经在过去的运行中加载了这些较早的数据分区。

3.5 设计任务的最佳实践

尽管Airflow在回填和重新运行任务时承担了大量繁重的工作，但我们需要确保我们的任务满足某些关键属性，以确保获得适当的结果。在本节中，我们将深入研究正常气流任务的两个最重要的属性：原子性和幂等性。

3.5.1 原子性

在数据库系统中经常使用术语原子性，其中原子事务被认为是一系列不可分割且不可简化的数据库操作，使得要么全部发生，要么什么都不发生。类似地，在Airflow中，任务应被定义为它们要么成功并产生一些适当的最终结果，要么以不影响系统状态的方式失败。

例如，考虑对我们的用户事件DAG的简单扩展，我们希望在其中添加一些功能，以便在每次运行结束时发送前10个用户的电子邮件。添加此功能的一种简单方法是扩展我们前面的函数，额外调用某个发送包含统计信息的电子邮件的函数：

```
def_calculate_stats (**context) : "”计算事件统计信
        息。“”

        input_path=context[“templates_dict”][“input_path”]output_path=con
        text[“templates_dict”][“’output_path’”]

        events=PD.read_JSON (输入路径)
        stats = events.groupby(["date", "user"]).size().reset_index()
        stats.to_csv(output_path, index=False)
```

不幸的是，这种方法的缺点是任务不再是原子的。你知道为什么吗？如果不是，考虑一下如果我们的send_stats函数失败会发生什么（如果我们的电子邮件服务器有点不稳定，这肯定会发生）。在这种情况下，我们已经将统计信息写入到输出文件的OUTPUT_PATH中，使其看起来好像我们的任务成功了，尽管它以失败告终（图3.10a）。

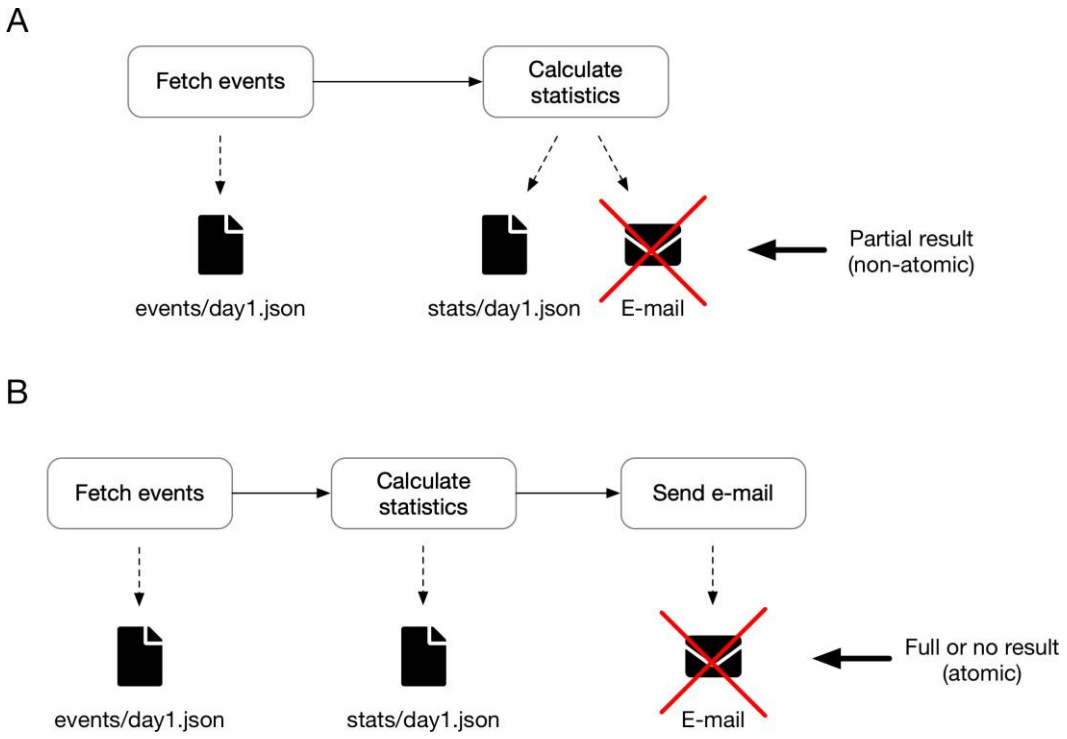


图3.10。我们的事件DAG的原子和非原子实现的说明。（a）将电子邮件通知添加到“计算统计数据”任务可能导致任务成功写入输出文件但无法发送电子邮件的情况，在这种情况下，整个任务失败。但是，由于输出文件在失败之前已经写入，因此我们只能看到任务的部分输出。因此，该任务不是原子任务，因为原子任务应该在没有任何副作用的情况下成功或失败。（B）通过将电子邮件通知添加为单独的任务，我们可以使两个任务都是原子任务。在这种情况下，如果通知任务失败，则整个任务失败，对我们的系统没有任何影响，因此可以认为是原子的。类似地，“计算统计信息”任务也可以成功写入输出文件或失败，因此是原子的。

要以原子方式实现此功能，我们可以简单地将电子邮件功能拆分为单独的任务：

```
def_send_stats（电子邮件，**上下文）：
        stats=PD.read_CSV（context[“templates_dict”][“stats_path”]）email_stats
         （stats，email=电子邮件）


send_stats=pythonOperator
         （task_ID=“”send_stats“”，
        python_callable=_send_stats，
        op_kwargs={
            “电子邮件”：”我们er@example.com“”
```

```
        模板_字典={
            ""stats_path"": ""data/stats/{DS}.CSV""
        },
        提供上下文=TRUE,
        DAG=DAG,
)

计算_统计信息>>发送_统计信息
```

这样，发送电子邮件失败不再影响Calculate_Stats任务的结果，而只是使Send_Stats失败，从而使两个任务都成为原子任务（图3.10B）。

从这个例子中，您可能会认为将所有操作分离为单独的任务足以使我们的所有任务原子化。然而，这不一定是真的。要了解原因，请想一想，如果我们的事件API要求我们在查询事件之前登录，会发生什么。这通常需要一个额外的API调用来获取一些身份验证令牌，之后我们就可以开始检索我们的事件了。

根据我们前面的一个操作=一个任务的推理，我们必须将这些操作拆分为两个单独的任务。但是，这样做会在两个任务之间产生很强的依赖性，因为第二个任务（获取事件）将失败，而不会在不久之前运行第一个任务。两个任务之间的这种强烈依赖性意味着，我们最好将两个操作都放在一个任务中，让任务形成一个连贯的工作单元。

大多数Airflow操作器已经设计为原子级，这就是为什么许多操作器包括用于执行紧密耦合操作（如内部身份验证）的选项。但是，更灵活的操作符（如Python和Bash操作符）可能需要您仔细考虑您的操作，以确保您的任务保持原子性。

3.5.2 幂等性

除了原子性之外，在编写Airflow任务时要考虑的另一个重要属性是幂等性。如果用相同的输入多次调用相同的任务没有额外的效果，则称任务是幂等的。这意味着，例如，在不更改输入的情况下重新运行任务不应更改总体输出。

例如，考虑我们上一次实现的fetch_events任务，该任务获取一天的结果并将其写入我们的分区数据集：

```
fetch_events=bashoperator
        (task_ID=""fetch_events"",
       bash_command=""curl-o data/events/{{DS}.JSON http: //localhost: 5000/events？开始日
       期={DS}&结束日期={next_DS}", DAG=DAG,
)
```

在给定日期重新运行此任务将导致任务获取与其上一次执行相同的事件集（假设日期在我们的30天窗口内），并覆盖Data/Events文件夹中的现有JSON文件，从而产生相同的最终结果。因此，获取事件任务的实现显然是幂等的（图3.11A）。

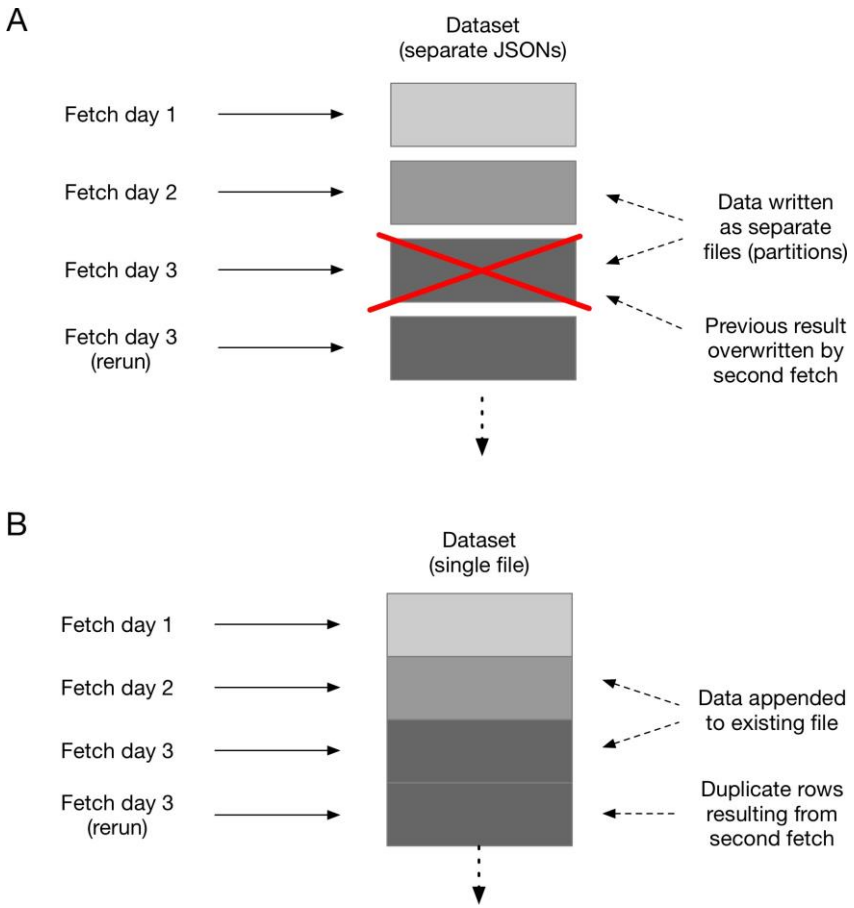


图3.11。我们的事件DAG中的幂等性。（a）将每日结果写入单个文件时，重新运行给定日期的获取任务会导致该日期的JSON文件被覆盖。这种覆盖确保我们得到与运行任务之前相同的结果：三个JSON文件，其中包含第1-3天的数据。（B）相反，如果我们要使用单个数据集来存储我们的结果，则我们的实现可能涉及每当我们获取给定日期的数据时将数据附加到该文件。在这种情况下，如果我们要重新运行其中一个获取任务，则简单的实现将导致重复的行，因为我们最终会将相同的数据附加到文件两次。因此，重新运行任务会影响系统的整体状态，从而使任务不是幂等的。

对于非幂等任务的示例，考虑我们讨论的使用单个JSON文件（data/events.JSON）并简单地将事件附加到该文件的情况。在这种情况下，重新运行任务将导致事件简单地附加到现有数据集，从而在数据集中复制Days事件。因此，该实现不是幂等的，因为任务的额外执行会改变总体结果（图3.11b）。

通常，通过检查现有结果或确保以前的结果被任务覆盖，可以使写入数据的任务成为幂等的。在时间分区数据集中，这是相对简单的，因为我们可以简单地覆盖相应的分区。类似地，对于数据库系统，我们可以使用upsert操作来插入数据，这允许我们覆盖由先前的任务执行所写入的现有行。但是，在更一般的应用程序中，您应该仔细考虑任务的所有副作用，并确保所有这些副作用都以幂等方式执行。

3.6 总结

- 使用DAG的start_date、schedule_interval和end_date属性在Airflow中计划DAG。
- 基于cron或基于频率的表达式，用于以适当的时间间隔计划DAG。
- 使用由Airflow的计划间隔定义的时间窗口来执行数据集的增量处理，这使您可以在数据子集进入时高效地处理它们。
- 使用Airflow提供的不同执行日期参数来引用（分区）数据集的特定子集。
- 执行日期是根据调度间隔定义的，这意味着调度间隔的执行日期是指间隔的开始日期，而不是DAG实际执行的时刻。
- 使用回填来填充数据集中过去的空白。
- 正确设计的气流任务应该是原子和幂等的，以确保在重新运行任务时结果仍然正确。