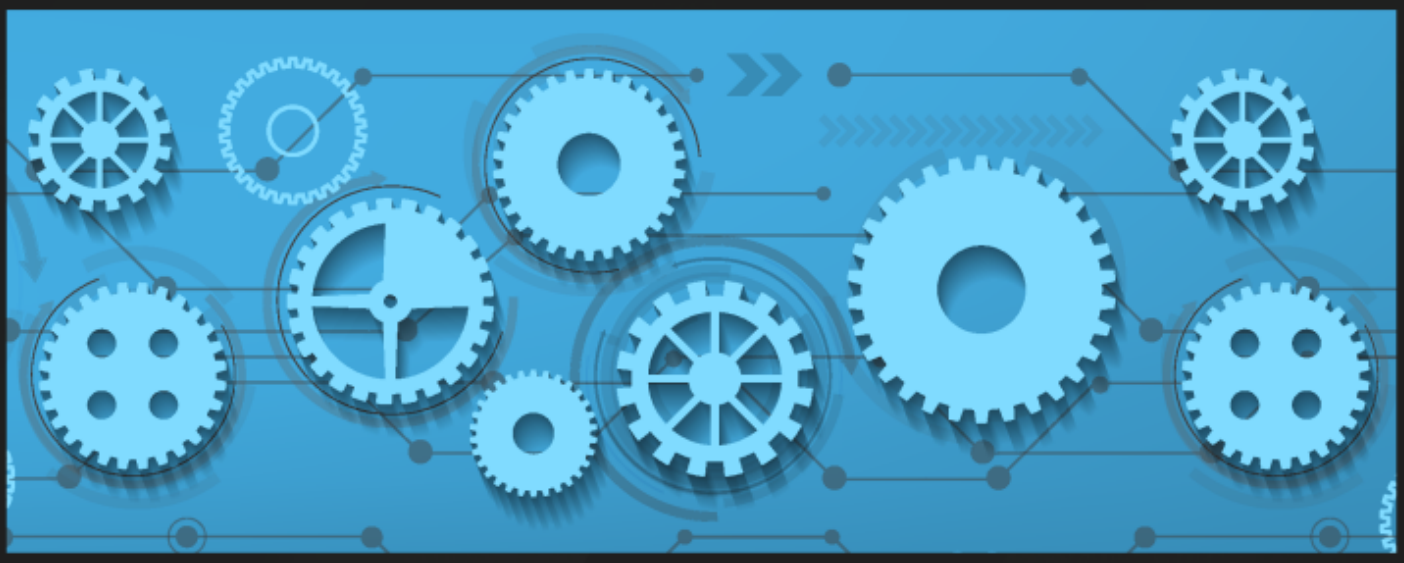
packt



1ST EDITION

GitHub Actions Cookbook

A practical guide to automating repetitive tasks and streamlining your development process

MICHAEL KAUFMANN

GitHub Actions 实用指南

自动化重复任务和优化开发流程

作者： Michael Kaufmann

译者：陈晓伟

目录

关于作者

Michael Kaufmann 相信开发者和工程师能够在工作中感到快乐和高效。他热爱 DevOps、GitHub、Azure 和现代工作方式。Microsoft 授予他 Microsoft 区域总监（RD）和 Microsoft 最有价值专家（MVP）称号 – 后者属于 DevOps 和 GitHub 类别。Michael 同时也是德国 Xebia Microsoft 服务公司的创始人兼总经理，这是一家咨询公司，通过支持客户进行云、DevOps 和数字化转型来帮助他们成为数字领导者。Michael 通过书籍、培训，以及经常在国际会议上进行分享。

前言

随着来自世界各地的数百万开发人员，以各种类型和规模的项目在 Github 平台上进行进行合作，GitHub 不仅仅是托管和共享代码的平台，它已成为开源社区的核心。借助 GitHub 操作，其拥有自己的工作流平台，可以自动化各种重复的工程任务 - 从从持续集成（CI）和持续部署（CD）到 IssueOps、问题的自动分类和 ChatOps。

本书将展示如何在日常生活中充分利用 GitHub Actions。这是一本实用的书 — 会让你动手实践，并在每个示例中结合理论进行讲解。

适读人群

如果正在寻找一种学习 GitHub 动作的实用方法，那么本书将非常适合您，无论您是软件开发人员还是 DevOps 工程师。如果已经独自发挥了动作，但想了解更多；或有其他 CI/CD 工具的经验，例如 Jenkins 或 Azure Pipelines；或者对于相关领域从未涉及 — 没关系，本书会在这方面对您进行帮助。

翻开本书之前，您应该至少了解一种编程或脚本语言、作为版本控制系统的 Git，以及包括 Docker、Linux 和 Windows 文件系统、认证等知识。

本书内容

第 1 章，*GitHub Actions 工作流*，介绍 GitHub Actions 工作流及其功能，将学习 YAML 基础、触发工作流的事件和表达式，以及如何从市场中使用 GitHub Actions 进行任务的自动化。

第 2 章，*编写与调试工作流*，介绍编写工作流的最佳实践：如何使用 Visual Studio Code 和 GitHub Codespaces 及各种插件，高效创建、编辑和运行工作流；使用强大的代码检查工具排查错误，在分支中开发工作流并在本地运行。还将会了解如何调试工作流，并启用高级日志记录。

第 3 章，*构建 GitHub Actions*，介绍不同类型的 GitHub Actions，并了解如何使用输入和输出。将尝试自行编写 Docker 容器动作、TypeScript 动作和复合动作。

第 4 章，*工作流运行时*，介绍了工作流的不同运行时选项。将了解如何使用由 GitHub 托管的运行器，以及如何在 Docker 容器和 Kubernetes 中使用 GitHub Actions Controller（GHAC）进行设置和扩展临时的、自托管的运行器。

第 5 章，*使用 GitHub Actions 自动化任务*，将展示如何使用 Issue-Ops 在 GitHub 内自动化常见任务。将了解如何通过 GitHub Apps 进行身份验证，使用 GITHUB_TOKEN 和工作流权限，使用 GitHub CLI 自动化任务，使用环境进行审批和检查，以及使用可重用的工作流和组合操作，在不同工作流和库之间共享逻辑。

第 6 章，*构建和验证代码*，主要介绍持续集成（CI）。将了解如何使用同一工作流构建和测试不同版本的代码，使用 CodeQL 查找代码中的安全漏洞，将软件物料清单（SBOM）添加到发布中，自动化软件版本控制，并使用缓存加快工作流的速度。

第 7 章，*使用 GitHub Actions 发布软件*，涵盖了持续交付和持续部署（CD）。将了解如何使用 OpenID Connect（OIDC）安全地部署到云端，以及如何将容器部署到 Kubernetes — 无论是 Microsoft Azure Kubernetes 服务（AKS）、Google Kubernetes 引擎（GKE），还是 Elastic Container Services(ECS)。此外，还将了解如何将 Dependabot 与 GitHub Actions 结合使用，以完全自动化更新依赖项。

环境配置

软件要求	操作系统
GitHub	适用于所有操作系统，需要在 https://github.com 上拥有一个账户。
Visual Studio Code	适用于所有操作系统。可以选择使用 GitHub Codespaces 来完成所有示例，无需配置本地环境。如果想要在本地进行实践，则需要安装 Visual Studio Code(下载地址：https://code.visualstudio.com/download）以及其他一些后续的工具。
Git	仅在本地实践时需要。适用于所有操作系统，应该安装最新版本的 Git(至少版本 2.23)。
Node.js	仅在本地实践时需要。需要安装最新版本的 Node.js(本书撰写时使用的是版本 21)。适用于所有操作系统，可以从这里下载最新版本：https://nodejs.org/en/download/current。
Docker	仅在本地实践时需要。可以在此处获取适用于所有操作系统的 Docker：https://docs.docker.com/get-docker/。
Azure 和 Azure CLI	某些章节需要一个 Azure 账户和 Azure CLI，免费试用版即可(https://azure.microsoft.com/en-us/free)。如果想在本地进行实践，还需要安装 Azure CLI。

所有示例都可以使用免费的 GitHub 账户在公共库中完成。你可以使用 GitHub Codespaces 在虚拟环境中完成所有操作。这将消耗你每月 120 小时的免费时长（GitHub Pro 用户为 180 小时），请注意这一点。一旦免费时长用尽，你将需要按分钟支付 Codespaces 的使用费用。

下载源码

可以从 GitHub 下载本书的示例代码文件，网址为 https://github.com/PacktPublishing/github-actions-cookbook。如果代码有更新，将在 GitHub 库中更新。

还有丰富的书籍和视频目录中的其他代码包，可在 https://github.com/PacktPublishing/ 上找到。快来看看吧！

第 1 章 GitHub 动作工作流

自 2008 年创立以来, GitHub 已经成长为拥有超过 2 亿个库和 1 亿用户的平台, 仅去年一年就产生了惊人的 35 亿次贡献提交。通过 GitHub Actions, 工程师和开发者现在可以自动化各种类型的工作流和重复性的工程任务 – 从持续集成 (CI) 和持续部署 (CD) 到 IssueOps、自动问题分类和 ChatOps。GitHub Actions 远不仅是一个 CI/CD 工具 – 它是一个全自动化平台, 可以优化整个开发流程。

本书将展示如何在日常开发中使用 GitHub Actions – 书中有很多需要动手的实例, 并会结合这些实例来讲解相关的理论知识。

本章中, 将了解 GitHub 中工作流的基础知识: 工作流文件、工作流与 YAML 语法、触发工作流的事件、表达式、密钥 (secrets) 与环境 (environments), 并动手完成第一个工作流。

本章主要内容:

- GitHub 的生态系统
- Github 的托管和定价
- GitHub Actions 的定价
- GitHub Marketplace(市场)
- 使用工作流编辑器编写工作流
- 使用密钥 (secrets) 和变量 (variables)
- 创建和使用环境 (environments)

1.1. 环境要求

本章, 需要一个免费的 GitHub 账户和浏览器。如果还没有 GitHub 账户, 只需在https://github.com/signup注册一个。

可以这里找到所有的示例和代码: https://github.com/wulfland/githubactionscookbook

1.2. GitHub 生态系统

GitHub 基于去中心化的 Git 版本控制系统 (VCS) 构建, 改变了软件开发的方式。而 GitHub 并不仅只是托管 Git 库的平台 – 已经发展成为一个全面的 DevOps 平台, 具备以下领域的功能:

- 协作编码
- 计划和跟踪
- 工作流和 CI/CD
- 开发者生产力
- 客户端应用
- 安全性

GitHub 的初心是以开发者为中心，因此该平台高度重视 Webhook 和 API 的使用。开发者可以使用 REST 或 GraphQL API 来与 GitHub 平台进行互动。此外，开发者还可以将 GitHub 用作身份提供者（IdP，identity provider），用于访问其应用。这种方式促进了与其他工具和平台的无缝集成，使 GitHub 成为一个全球开发者共建软件的平台。

要理解 GitHub Actions 的强大功能，必须知道可以在整个生态系统中自动化的各种任务 — 不仅限于代码。这包括：

- 计划与追踪：GitHub 提供了 Issues（问题）、Milestones（里程碑）、GitHub Discussions（讨论区）以及 GitHub Projects（项目管理），用于进行项目规划和进度跟踪，还能与 Jira、Trello 或 Azure Boards 等计划与追踪工具无缝集成。
- 客户端应用：GitHub 提供了可直接在浏览器中访问的代码编辑器 Visual Studio Code（https://github.dev），还有适用于 iOS 和 Android 平台的移动应用，可随时随地参与协作，以及跨平台的桌面应用，并且 CLI（命令行接口）也具有良好的扩展性。GitHub 还集成了所有主流 IDE，如 Visual Studio、Visual Studio Code 和 Eclipse，并支持 Slack 和 Teams 等聊天平台。
- 安全性：GitHub 高级安全（Advanced Security）提供软件供应链安全功能，包括 Dependabot、密钥扫描（Secret Scanning），以及 CodeQL 的代码扫描。还可以与 Snyk、Veracode 或 Checkmarx 等工具集成，并集成到 Microsoft 的 Defender for DevOps 中。
- 开发者生产力：GitHub 能提供了一个虚拟的、基于容器的开发环境 — GitHub Codespaces，以及由 AI 驱动的助手 GitHub Copilot，可以帮助开发者编写和理解代码。GitHub 还提供代码搜索、命令面板等功能，进一步提升开发者的生产力。
- 工作流与 CI/CD：除了 GitHub Actions 外，GitHub 还支持市场上大多数 CI/CD 工具。此外，GitHub 提供了与各大云服务商的安全集成，通过 Open ID Connect（OIDC）实现 CI/CD 工作流的身份验证。GitHub Packages 提供了一个支持多种包格式（包括原生 npm 支持）的包注册表 — 而且所有主流的包注册表也都可以与 GitHub 集成。

GitHub Actions 可用于在 GitHub 生态系统中自动化任务，并构建解决方案（见图 1.1）：

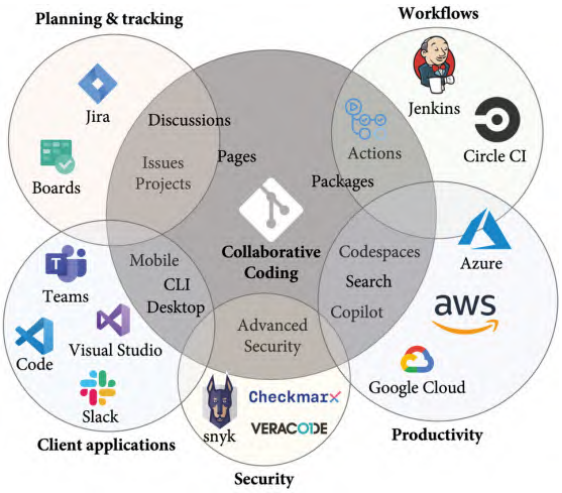


图 1.1 — GitHub 生态系统及其集成

在本书中，我将为提供所有主要领域的实用工作流示例，帮助各位读者自动化各种实际开发任务。

1.3. Github 的托管和定价

本书中的所有示例可在 https://github.com 上完成，这是 GitHub 提供的“软件即服务”（SaaS）版本。注册 GitHub 完全免费，用户可以拥有无限的私有和公共库。

对于开源项目（公共库），GitHub 上几乎所有功能都可以免费使用；但对于私有库，则需要购买付费许可证。公共库中，对于 GitHub Actions 的使用并无时间限制，建议将所有示例操作都在公共库中进行– 否则，可能会迅速耗尽每月仅有 2,000 分钟的免费额度。

GitHub 的定价模型按月进行计费，包含三个层级：免费版（Free）、团队版（Team）和企业版（Enterprise)(见图 1.2):

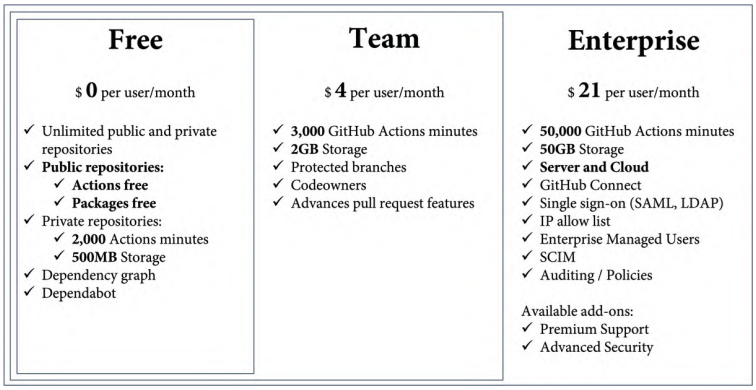


图 1.2 – GitHub 定价层级

如前所述，公共库完全免费 - 包括 GitHub Actions、Packages 以及 Dependabot 和 Secret Scanning 等安全特性。私有库同样免费，但仅限于有限的协作功能，不包括受保护的分支、代码所有者（Codeowners）及一些高级拉取请求（PR）功能。对于私有库，免费版每月有 2,000 分钟的 GitHub Actions 使用额度。

要解锁更多的协作功能，需要获取团队版许可证，每个用户每月费用为 4 美元。团队版方案还包括 3,000 分钟的 GitHub Actions 使用时限。

GitHub Enterprise 方案则提供了所有企业级特性，例如：通过 Security Assertion Markup Language（SAML）和跨域身份管理系统（SCIM）实现的单点登录（SSO)、企业托管用户和 IP 白名单等。还提供了 50,000 分钟的 GitHub Actions 使用时限 – 但每位用户每月需支付 21 美元。

GHES 与 GitHub Actions

如果使用的是 GHES(GitHub Enterprise Server)，则无法使用 GitHub 托管的运行器来运行工作流。需要自行提供工作流所需的运行器，并确保其安全性，以及清理其工作流产生的产物（artifacts)。通常情况下，这可以通过 Kubernetes 配合 Actions Runner Controller(ARC –https://github.com/actions/actions-runner-controller）完成。我们将在第 4 章中了解到更多。

1.4. GitHub Actions 的价格

因为使用自己的计算资源，所以自我托管运行器上运行工作流完全免费。公共库中运行工作流也免费——即使在 GitHub 也是如此。GitHub 托管的运行器在 Linux、Windows 和 macOS 上可用，并且有不同的规格。如果想在私有库中使用这些运行器，将按分钟计费。不同的运行器使用不同的分钟乘数（见表 1.1）。在 Linux 上运行工作流程将每分钟消耗 1 分钟的免费时长——如果超出免费时长，则需支付 $0.008。

Windows 将以两倍的速度消耗免费时长，之后每分钟费用为 $0.08。macOS 将以十倍的速度消耗时长，当达到包含的分钟数限制时，每分钟收费 $0.016

操作系统	分钟倍率	每分钟价格
Linux	1	$0.008
Windows	2	$0.080
macOS	10	$0.016

表 1.1 – GitHub 托管运行器的每分钟定价

这就是为什么本书中的大多数示例中使用 Linux，并且我会鼓励我的客户尽可能在 Linux 上运行更多任务的原因。

如果使用 GHEC 或团队计划，并且需要功能更强大的机器，可以使用更大的 GitHub 托管运行器。它们按分钟计费（见表 1.2），并具有静态 IP 范围等功能：

虚拟 CPU 数量	Linux	Windows	macOS
2	$0.008	$0.016	
3			$0.08
4	$0.016		
8	$0.032	$0.064	
12			$0.32
16	$0.064	$0.128	
32	$0.128	$0.256	
64	$0.256	$0.512	

表 1.2 – 更大运行器的每分钟费率

私有网络

除了静态 IP 范围，还可以使用 Azure 私有网络直接将 GitHub 托管的运行器连接到相关资源。在撰写本文时，此功能仍处于测试阶段，可能会发生变化。有关更多信息，请参阅以下链接：https://docs.github.com/en/enterprise-cloud@latest/admin/configuration/configuringprivate-networking-for-hosted-compute-products/about-networkingfor-hosted-compute-products

GitHub Actions 还会消耗存储空间 — 例如，用于日志、工作流程构件或缓存。如果超出包含的存储空间，将收取每天每 GB $0.008 的费用。

价格可能会变化，并参考 GitHub 文档以获取最新信息（https://docs.github.com/en/billing/managing-billing-forgithub-actions/about-billing-for-github-actions）。

要学习 GitHub Actions 并尝试工作流程 — 只需在公共库中执行所有操作，则无需支付计算或存储费用。

1.5. GitHub 市场

Github 提供了一个社区驱动的市场（https://github.com/marketplace），目前包含超过 20,000 个 GitHub Actions，可以在工作流程中使用这些 Actions 作为构建块（见图 1.3）：

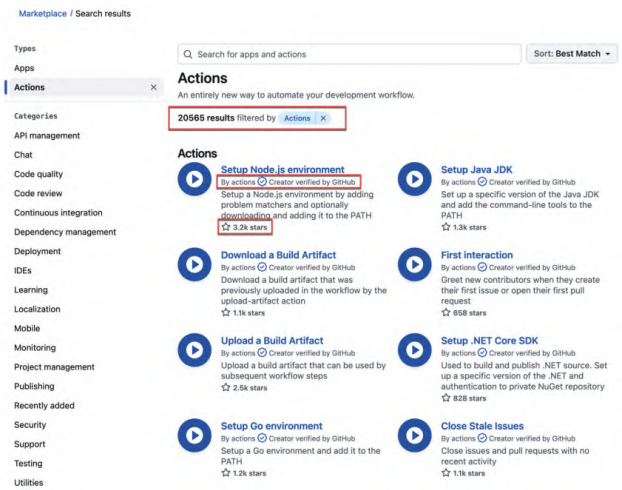


图 1.3 — Github 市场包含超过 20,000 个可重用的 Actions

若 Action 的作者是 actions，则是 GitHub 的原生 Action。并且可以在概况中看到 Action 的点赞数，可以对 Action 的受欢迎程度有一个直观的了解。还会看到一个蓝色徽章，表明该 Action 的作者已经过 GitHub 的验证。

可以通过多个类别对 Actions 进行过滤，也可以通过术语进行搜索，并且可以更改结果的排序方式为 **Most installed/starred**(安装/标星最多)、**Best Match**(最佳匹配) 或 **Recently added**(最近添加)(见图 1.4)：

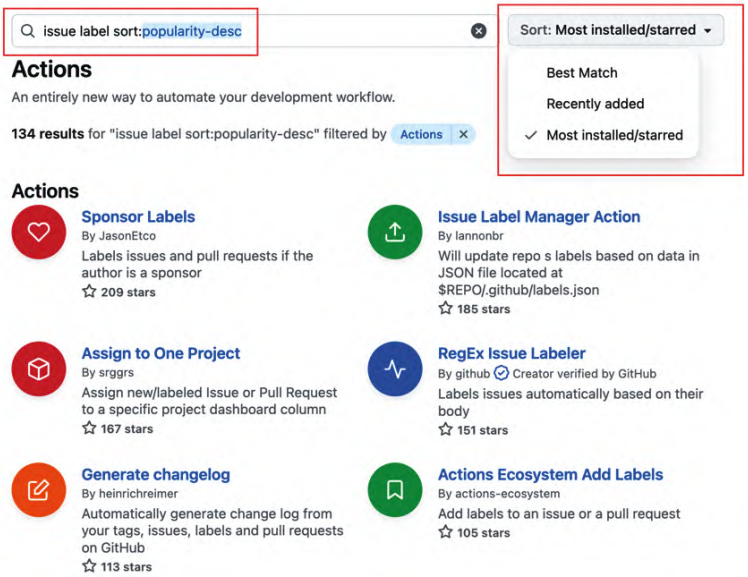


图 1.4 – 中搜索市场并对结果进行排序

通过这种方式，可以轻松地到有相关的 Actions。

点击其中一个结果，将进入详细信息页面（见图 1.5）：

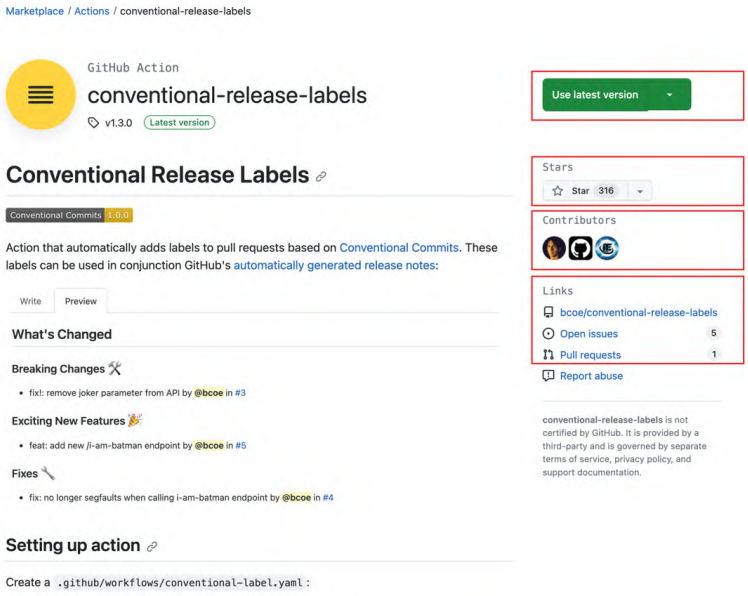


图 1.5 – 详细信息

您可以找到发布的版本、标星数、贡献者，以及指向源码库的链接（包括问题和拉取请求的数量，并且所有 Actions 都开源）。这能对 Action 的使用活跃度有很好的了解——并且还可以深入研究其代码。

市场的结果也会显示在工作流程编辑器中，我们将在相关的示例中使用。

1.6. 使用工作流编辑器编写工作流

GitHub 在工作流程设计器中会指导用户如何编写工作流，可以使用工作流编辑器编写工作流。所以，最好的办法是，编写自己的工作流，并熟悉这个平台。

1.6.1 Getting ready

创建第一个工作流程之前，必须先在 GitHub 上创建一个库，导航到https://github.com/new。如果尚未认证，请进行认证，并填写如图 1.6 所示的数据：

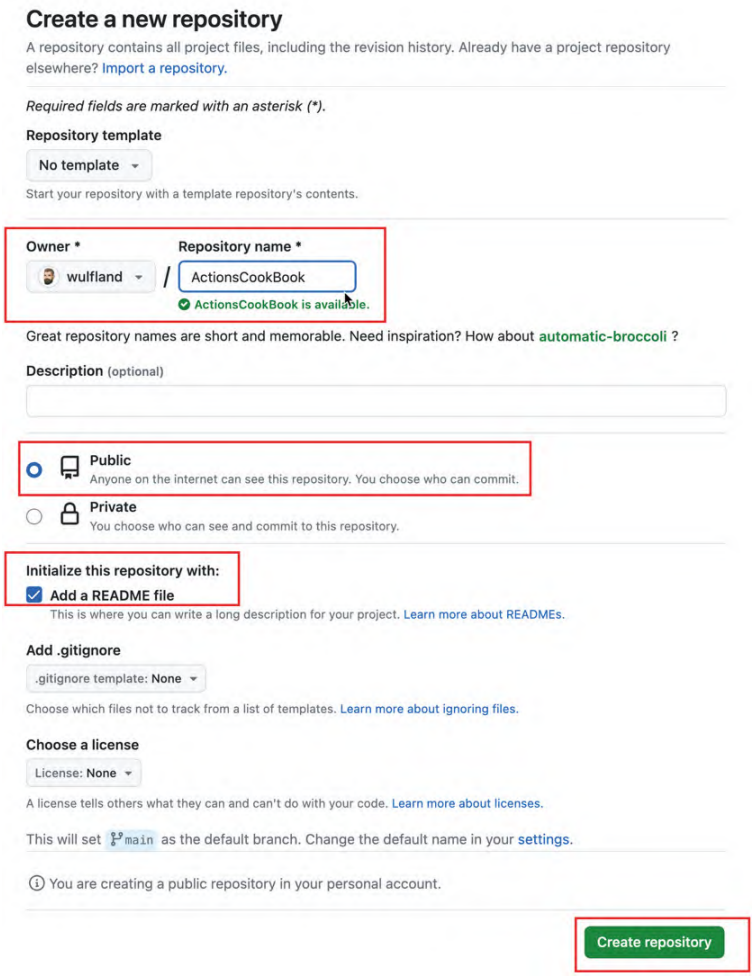


图 1.6 – 创建一个新库

选择 GitHub 用户作为所有者，并设置名称——例如，ActionsCookBook。将其设为公共库，以便所有工作流程和存储都免费。使用 README 文件初始化库——这样，库中就已经有了文件，工作流也可以进行操作了。

1.6.2 How to do it…

GitHub Action 工作流程是位于库的.github/workflows 文件夹中，扩展名为.yml 或.yaml 的 **YAML** 文件。可以手动创建该文件，这样的话工作流编辑器只有在第一次提交后才能工作，我建议从菜单中创建一个新的工作流。

1. 在新库中，点击到 **Actions**。由于库是新的，还没有任何工作流，所以会重定向到 **Create new workflow**(创建新的工作流程）页面（actions/new)。如果库包含工作流，可以在这里看到这些工作流（如图 1.16 所示），需要点击 **New workflow**(新工作流程）按钮才能到达该页面。

在这个页面上，有很多可以作为起点使用的模板工作流。适配大多数云的部署、适配大多数语言的 CI、代码安全扫描、通用自动化，以及将内容部署到 GitHub Pages 的启动工作流。本示例中，我们将专注于熟悉编辑器，并通过点击 **set up a workflow yourself**(自己设置工作流程)(见图 1.7) 从头创建一个工作流：

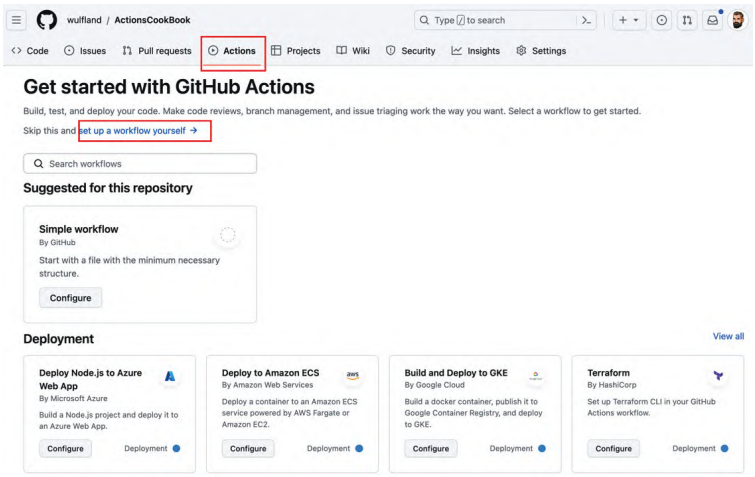


图 1.7 — 在 GitHub 中创建一个新的工作流

2. GitHub 会在默认分支的.github/workflows 中创建一个新的 main.yml 文件，并在 Web 编辑器中显示。编辑器的右侧是文档，可以在 GitHub Marketplace 中搜索 actions。在编辑器中，可以使用 Ctrl + Space(或 Option + Space) 自动完成。编辑器会捕获 Tab 键，并且默认情况下将其用于两个空格的缩进。要使用 Tab 键导航到页面上的其他控件，必须使用 Esc 或 Ctrl + Shift + M 先进行退出。

   将文件名修改为 MyFirstWorkflow.yml(见图 1.8)：

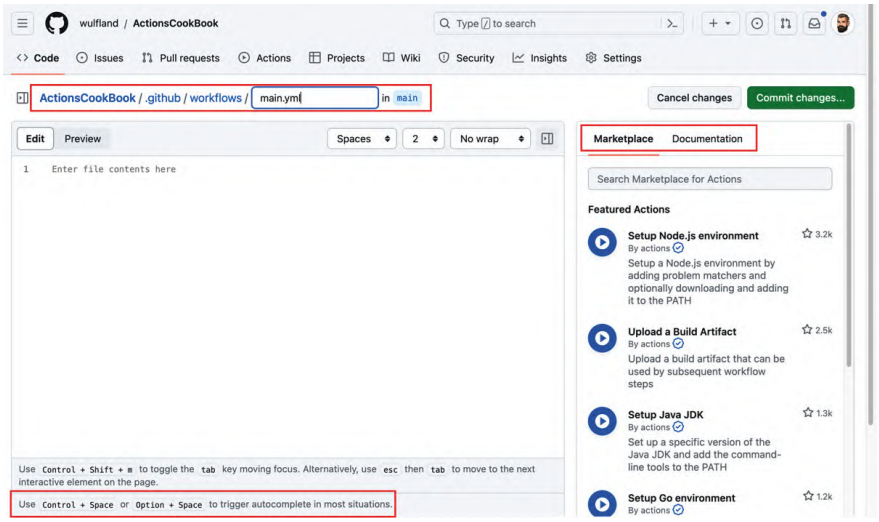


图 1.8 — GitHub Actions 的工作流编辑器

3. 在编辑器中，点击 Ctrl + Space(或 Option + Space) 查看在工作流文件中有效的根元素列表（见图 1.9)：

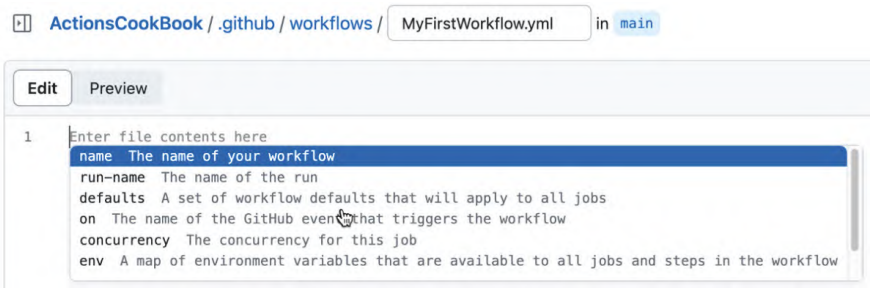


图 1.9 – 编辑器显示工作流文件中某个级别的有效选项

4. 可在文件顶部添加注释，并使用自动完成设置 name 属性。请注意，编辑器具有错误检查功能，并指示当前缺少所需的根键 on(见图 1.10):

图 1.10 – 代码编辑器中的错误检查

通常，工作流以 name 属性开始，该属性在工作流 UI 中设置显示名称。在文件顶部添加注释以总结工作流的意图，是一个好习惯。

5. 接下来，配置应触发工作流的事件。请注意，工作流程可以有多个触发器，可根据在工作流设计器中的位置自动完成，并得到出不同的结果。如果与 on: 在同一行，将得到 JSON 语法的结果（参见 YAML 集合类型部分）；也就是，on: [push]。
如果在第一个元素后面添加一个逗号，然后再次点击 Control + Space，则可以从自动完成中选择其他元素（见图 1.11):

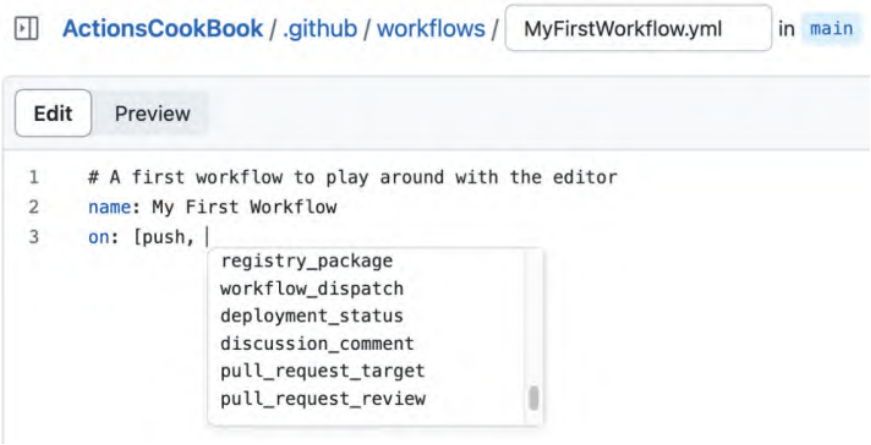


图 1.11 – 自动补全也适用于方括号内

每个触发器都是一个映射，并且可以包含附加参数。如果将光标放在 on: 下面的行上并添加两个空格的缩进，自动完成将使用完整的 YAML 语法给出结果，还会给出可用于配置每个触发器的属性（见图 1.12):

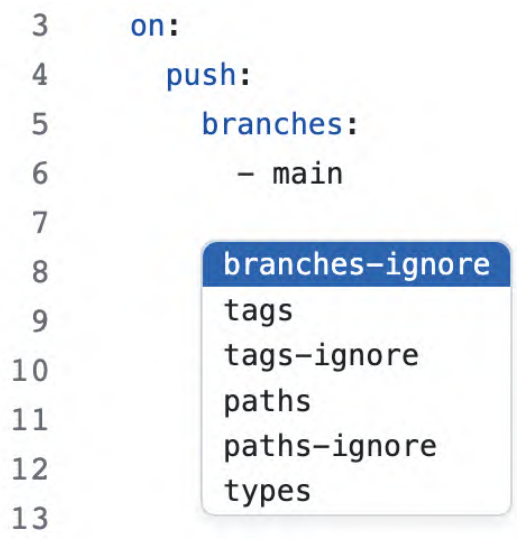


图 1.12 – 自动完成有助于触发器的选项

请注意，大多数参数 - 例如，分支或路径 - 都是序列，如果不使用 JSON 语法，则需要每个条目使用一个短划线。

我们希望测试工作流程在每次推送到 main 分支时运行，还希望能够手动进行触发（请参阅触发工作流的事件部分）。工作流触发器代码好似如下所示：

```
on:
  push:
    branches:
      - main
  workflow_dispatch:
```

通配符

* 字符可以用作路径中的通配符，** 用作递归通配符。* 是 YAML 中的特殊字符，所以需要使用引号：

```
push:
  branches:
    - 'release/**'
  paths:
    - 'doc/**'
```

6. 配置好工作流程的触发器后，下一步是添加另一个根元素：作业（jobs）。作业在 YAML 中是一个映射 - 所以下一行使用两个空格缩进，自动完成将不起作用，编辑器期望设置一个名称。可将作业命名为 first_job，并转到下一行。作业对象的名称只能包含字母数字值、短划线（-）和下划线（_）。如果希望在工作流程中显示其他字符，可以使用 name 属性：

```
jobs:
  first_job:
    name: My first job
```

7. 每个作业都需要一个执行运行器。运行器通过标签标识，将在第 4 章中了解更多关于运行器的信息。我们希望工作流在 GitHub 提供的最新版本的 Ubuntu 运行器上执行，因此使用 ubuntu-latest 标签：

```
runs-on: ubuntu-latest
```

8. 作业由一系列依次执行的步骤组成。最基本的步骤是 run: 命令，将执行一个命令行命令：

```
steps:
  - name: Greet the user
    run: echo "Hello world"
    shell: bash
```

name 是可选的，并设置步骤在日志中的输出。shell 也可选，并在非 Windows 平台上默认为 bash，可再回退到 sh。Windows 上，默认是 PowerShell Core(pwsh)，可再回退到 cmd。可以使用 {0} 占位符，为步骤输入配置 shell(例如，shell: perl {0})。要添加变量输出，可以使用写在 ${{ 和 }} 之间的表达式。在表达式中，可以使用来自上下文对象的值，例如 GitHub 上下文。请注意，自动完成也适用于这些上下文对象（见图 1.13）：

图 1.13 – 自动完成也适用于上下文对象

从值列表中选择 actor：

```
- run: echo "Hello world 👋 from ${{ github.actor }}."
```

可参考表达式（https://docs.github.com/en/actions/learn-github-actions/expressions）和上下文（https://docs.github.com/en/actions/learn-github-actions/contexts）的在线文档。

9. YAML 允许编写多行脚本，而无需与引号和换行符作斗争。只需在 run: 后面添加管道操作符（|），并在下一行使用四个空格缩进编写脚本。YAML 将其视为一个块，直到下一个元素 – 即使其中包含新行和空行：

```
- run: |
    echo "Hello world 👋 from ${{ github.actor }}."
    echo "Current branch is '${{ github.ref }}'."
```

10. GitHub Actions 工作流程不会自动下载库中的代码。如果想对代码库中的文件进行操作，必须先 checkout 内容。这可以使用 GitHub Action 完成 - 一个可重用的流程步骤，可以轻松地与多个流程共享。
在工作流编辑器的右侧是市场，可以直接在那里搜索 Action。搜索检出（checkout）并找到来自 actions 的 Action(这些是 GitHub 的内置 Action)。列表包含一个安装部分，可以将其复制到工作流中以使用该 Action(见图 1.14):

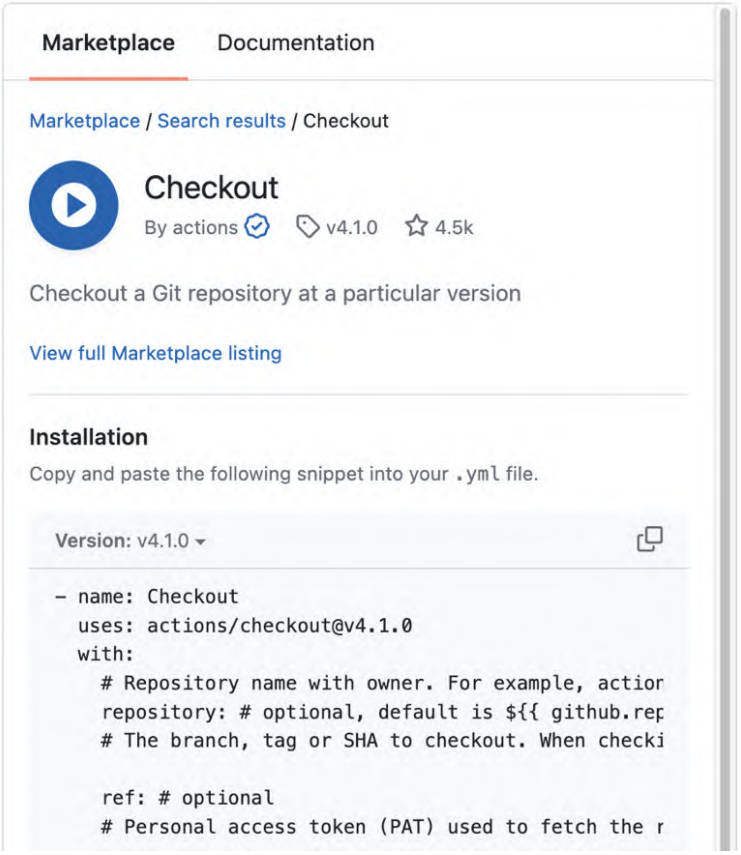


图 1.14 – 工作流程编辑器中的市场列表

请注意，许多参数是可选的。为了 checkout 代码库，只需要以下几行信息即可:

```
- name: Checkout
  uses: actions/checkout@v4.1.0
```

使用 GitHub Actions

指向 GitHub 上的一个位置。语法是 {path}@{ref}。路径指向 GitHub 上的物理位置，如果 Actions 在代码库的根目录中，则为 {owner}/{repo}，如果 Actions 在子文件夹中，则为 {owner}/{repo}/{path}。@{ref} 之后的引用指向提交的 git 引用，可以是一个标签、分支或单个提交 SHA 码。

11. 为了在检出后显示库中的文件，需要添加一个步骤：

```
- run: tree
```

这将以树形结构输出代码库中的文件。

12. 要运行工作流程，只需将工作流程文件提交到 main 分支。点击 **Commit changes…**（提交更改…），保留提交信息和分支，并在对话框中点击 **Commit changes**（提交更改）以完成操作（见图 1.15）：

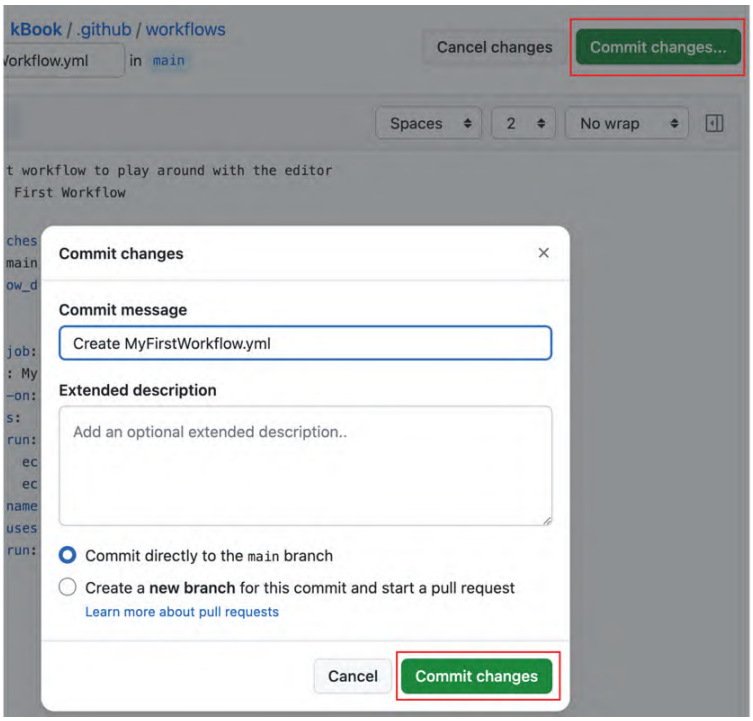


图 1.15 – 提交工作流文件

13. 已经为主分支设置了一个推送触发器，所以提交已经自动触发了工作流程。如果现在导航到库中的操作，将能够看到工作流和最新的工作流运行的情况（见图 1.16）：

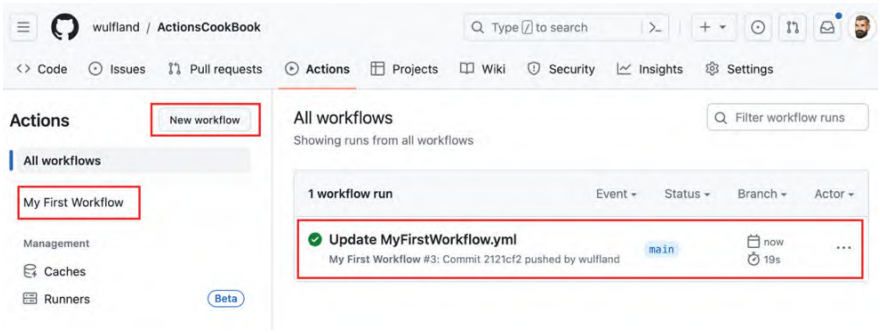


图 1.16 – 操作的默认视图显示所有工作流程的最新运行情况

请注意，工作流程运行的名称是提交信息，还可以看到触发工作流程的提交和推送更改的操作者。

14. 点击工作流程运行以查看更多详细信息，工作流摘要页面包含左侧的作业和一个右侧的可视化表示（见图 1.17），包含触发器、状态和持续时间等元数据：

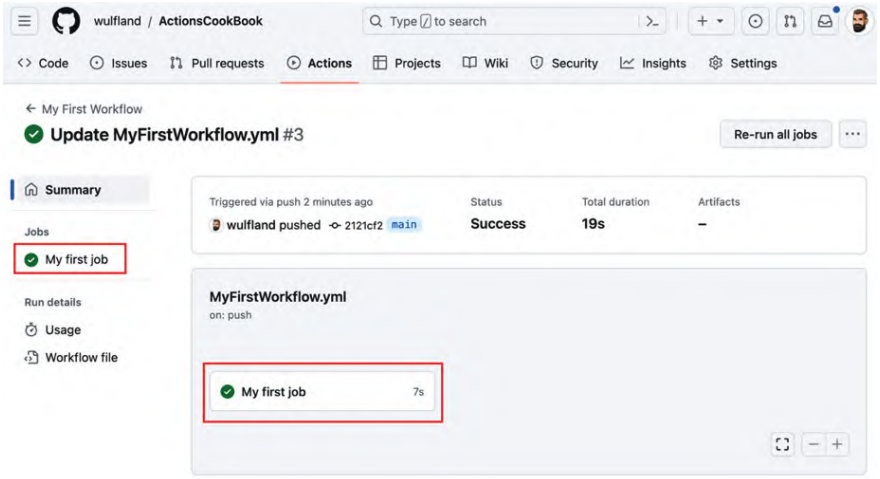


图 1.17 – 工作流摘要页面

15. 点击作业以查看更多详细信息。在工作流程日志中，可以检查各个步骤。请注意，工作流文件的每一行都有一个可点击的数字 - 那是一个 URL，可以用来识别每一行。设置作业步骤是个特殊步骤，提供了有关工作流运行者和工作流程权限的许多背景信息（见图 1.18）。检查工作流程所有步骤的输出：

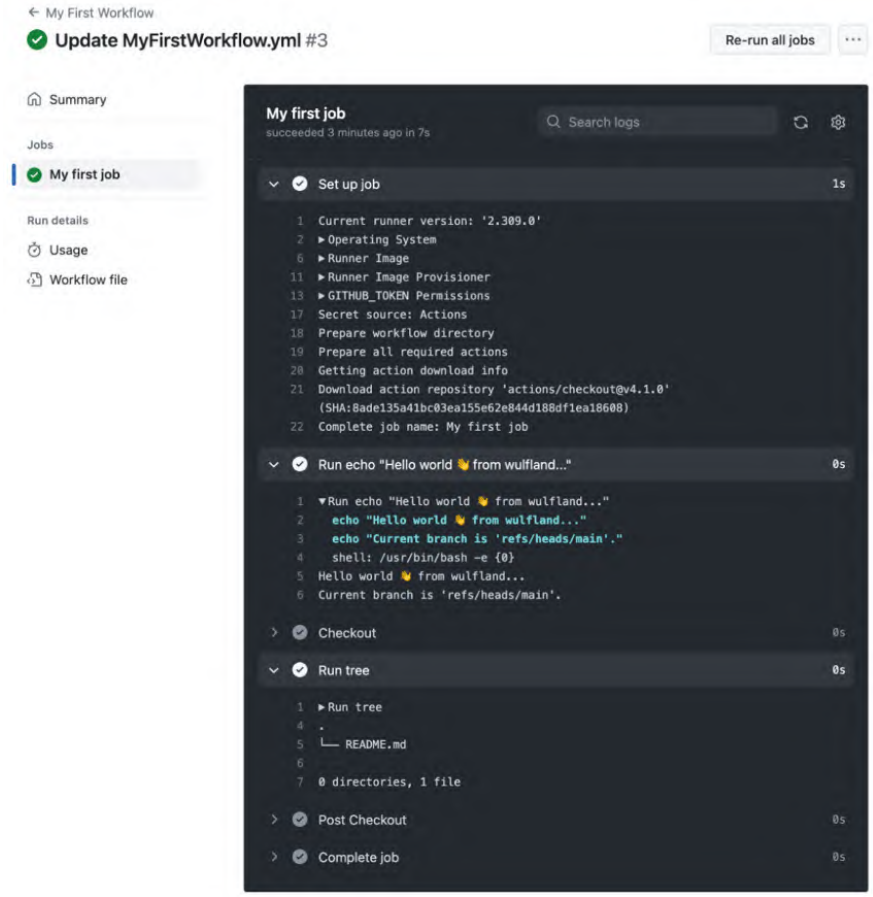


图 1.18 – 单个作业工作流的日志

16. 作为最后一步，可以手动触发工作流程，以便也查看工作流程运行中的差异。返回操作，选择左侧的工作流程（见图 1.19），然后运行工作流程：

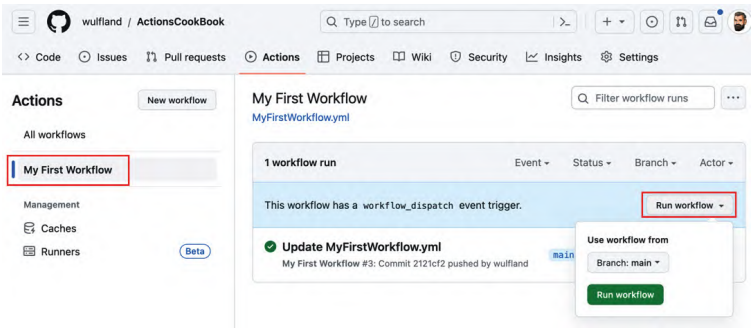


图 1.19 – 通过 UI 手动触发工作流

检查新的工作流及其输出。

1.6.3 How it works…

工作流文件是位于代码库.github/workflows 文件夹中的 YAML 文件。

YAML 基础知识

YAML 是“YAML Ain’t Markup Language”的缩写，是一种数据序列化语言，经过优化，可以直接由人类编写和阅读。它是 JSON 的一个严格超集，但使用语法上重要的换行符和缩进代替了大括号。

可以在文本前加上井号（#）来编写注释。

YAML 中，可以使用以下语法将值分配给变量：key: value。

key 是变量的名称。根据 value 的数据类型，变量的类型将有所不同。请注意，键和值可以包含空格，不需要引号！只有在使用某些特殊字符或想强制某些值为字符串时才添加，可以使用单引号或双引号引用键和值。双引号使用反斜杠作为转义模式("Foo \"bar \" foo ")，而单引号则使用单引号('Foo''bar''foo')。

YAML 集合类型

在 YAML 中，有两种不同的集合类型：嵌套类型称为映射（map）和列表（list)- 也称为序列（sequence)。映射使用两个空格的缩进：

```
parent_type:
  key1: value1
  key2: value2
  nested_type:
    key1: value1
```

一个序列是一个有序的项目列表，每行前都有一个减号：

```
sequence:
  - item1
  - item2
  - item3
```

由于 YAML 是 JSON 的一个超集，也可以使用 JSON 语法在一行中放置集合：

```
key: [item1, item2, item3]
key: {key1: value1, key2: value2}
```

触发工作流的事件

工作流程有三种类型的触发器：webhook 触发器、定时触发器和手动触发器。

Webhook 触发器根据 GitHub 中的事件启动工作流程。有许多可用的 webhook 触发器，例如：可以在 issues 事件、代码库事件或 discussions 事件上运行工作流程。我们示例中的 push 触发器是一个 webhook 触发器。

定时触发器可以在多个预定时间运行工作流程，语法与用于 cron 作业的语法相同：

```
on:
  schedule:
    # Runs at every 15th minute
    - cron: '*/15 * * * *'
    # Runs every hour from 9am to 5pm
    - cron: '0 9-17 * * *'
    # Runs every Friday at midnight
    - cron: '0 2 * * FRI'
```

手动触发器允许手动启动工作流程。workflow_dispatch 触发器将使用 Web UI 或 GitHub CLI 启动工作流程。可以使用 inputs 属性为此触发器定义输入参数，repository_dispatch 触发器可用于使用 API 触发工作流程。此触发器也可以通过某些事件类型进行过滤，并可在工作流中访问的 JSON 的有效信息。

要了解更多关于触发器的信息，请查看文档https://docs.github.com/en/actions/anction/lusther-workflows/events-th-trigger-workflows。

工作

每个作业都需要一个执行运行器，运行器通过标签标识。我们的示例中，使用 ubuntu-latest 标签。这意味着我们的作业将在 GitHub 托管的最新 Ubuntu 镜像上执行。

使用 GitHub Actions

Actions 指向 GitHub 上的一个位置。语法是 {path}@{ref}。路径指向 GitHub 上的物理位置，如果 Actions 在代码库的根目录中，则为 {owner}/{repo}，如果 Actions 在子文件夹中，则为 {owner}/{repo}/{path}。@{ref} 之后的引用指向提交的 git 引用，可以是一个标签、分支或单个提交 SHA 码。

```
# Reference a version using a tag
- uses: actions/checkout@v4.1.0

# Reference the current head of a branch
- uses: actions/checkout@main

# Reference a specific commit
- uses: actions/checkout@8e5e7e5ab8b370d6c329ec480221332ada57f0ab
```

对于同一代码库中的本地 actions，如果仅检出库，则可以省略引用。

如果 action 已定义输入，可以使用 with 属性进行指定：

```
- uses: ActionsInAction/HelloWorld@v1
  with:
    WhoToGreet: Mona
```

输入为可选或必需，还可以使用 env 属性为步骤设置环境变量：

```
- uses: ActionsInAction/HelloWorld@v1
  env:
    GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

1.6.4 There’s more…

这只是一个非常基础的工作流程，使用一个 action 来检出代码并在命令行上运行一些命令。接下来的两个示例中，将展示如何使用密钥、变量和保护环境来实现更复杂的工作流程。

1.7. 使用密钥和变量

可以在库中设置变量和密钥，以便在工作流程中访问。这个示例中，我们将添加这两者并在工作流程中进行访问。

1.7.1 Getting ready

在这个示例中，将使用 Web UI 来设置变量和密钥，也可以使用 GitHub CLI(https://cli.github.com）来完成这项工作（需要安装)。但对于这个示例来说，GitHub CLI 并不是必需的。

1.7.2 How to do it…

1. 导航到 **Settings** | **Secrets and Variables** | **Actions**(设置 | 密钥和变量 | 操作)，可查看库中现有的密钥，并且可以在 **Secrets**(密钥)(settings/secrets/actions) 和 **Variables**(变量)(settings/variables/actions) 选项卡之间切换，参考图 1.20：

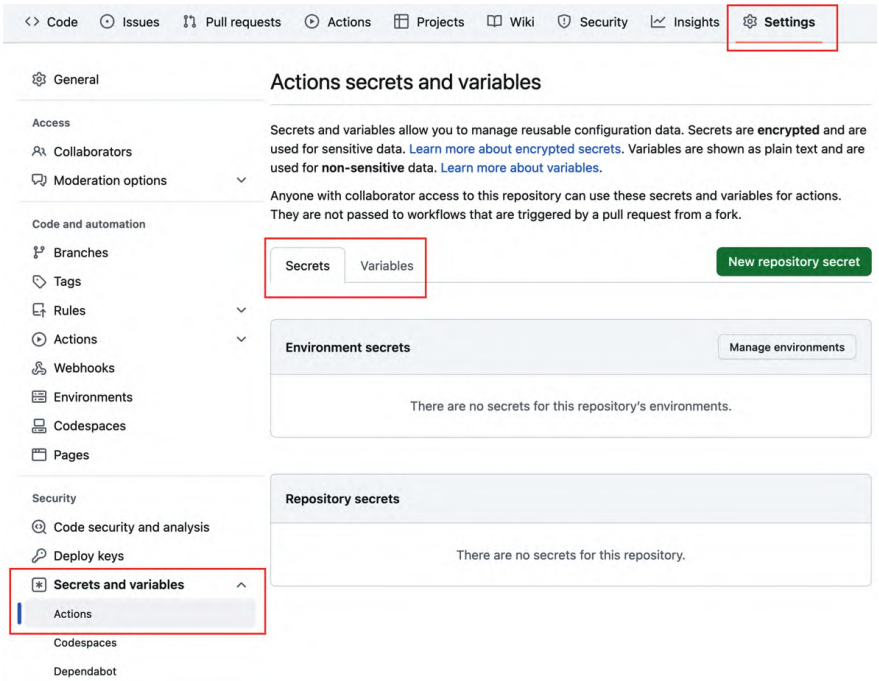


图 1.20 – 为代码库配置密钥和变量

2. 点击 **New repository secret**(新建代码库密钥) 将打开 **New secret**(新建密钥) 对话框 (settings/secrets/actions/new; 请参阅图 1.21):

图 1.21 – 添加新密钥

将 MY_SECRET 作为密钥名称, 将随机单词 (例如 Abracadabra) 作为密钥, 然后点击 **Add secret**(添加密钥), 日志中将屏蔽密钥! 因此, 不要使用可能在其他作业或步骤的输出中出现的常见单词。

密钥和变量的命名约定

密钥名称不区分大小写, 并且只能包含普通字符 ([a-z] 和 [A-Z])、数字 ([0-9]) 和下划线 ( )。它们不能以 GITHUB 或数字开头。
最好是使用大写单词, 并通过下划线字符分隔来命名密钥。

3. 重复该过程以 **New repository variable**(创建新的代码库变量)(settings/variables/actions/new), 并创建一个名为 WHO_TO_GREET 且值为 World 的变量。

4. 打开来自上一个示例的.github/workflows/MyFirstWorkflow.yml 文件，并点击编辑图标（请参阅图 1.22）：

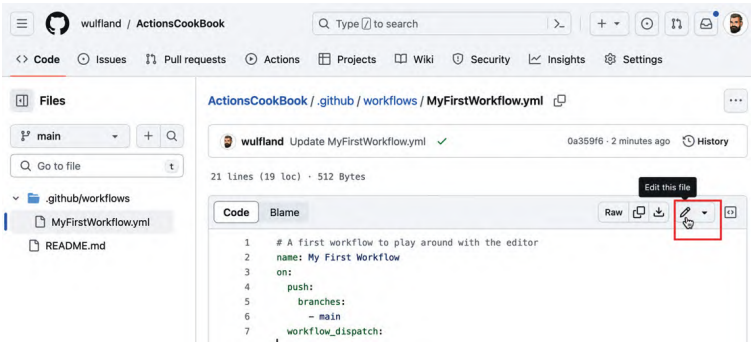


图 1.22 – 编辑 MyFirstWorkflow.yml

将单词 World 更改为${{vars.WHO_TO_GREET}}表达式,并使用${{ secrets.MY_SECRET }}添加秘钥：

```
- run: |
    echo "Hello ${{ vars.WHO_TO_GREET }} 👋 from ${{ github.actor }}."
    echo "My secret is 🤫 '${{ secrets.MY_SECRET }}'."
```

5. 提交更改。工作流将自动运行，检查工作流程日志中的输出，看起来类似图 1.23：

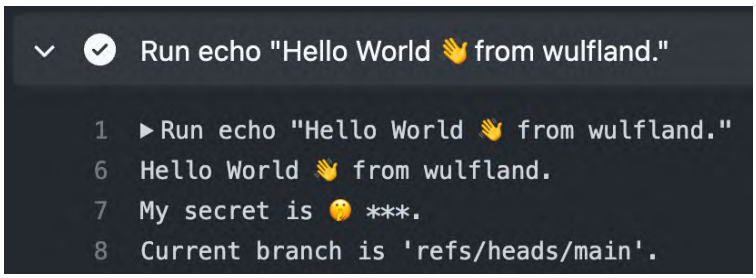


图 1.23 – 日志中密钥和变量的输出

1.7.3 There’s more…

可以通过在以下某个级别定义，来创建用于多个工作流的配置变量：

- 组织级别
- 库级别
- 环境级别

这三个级别像一个层次结构：可以通过为相同的键提供新值，来覆盖较低级别的变量或密钥。图 1.24 展示了这个层次结构：

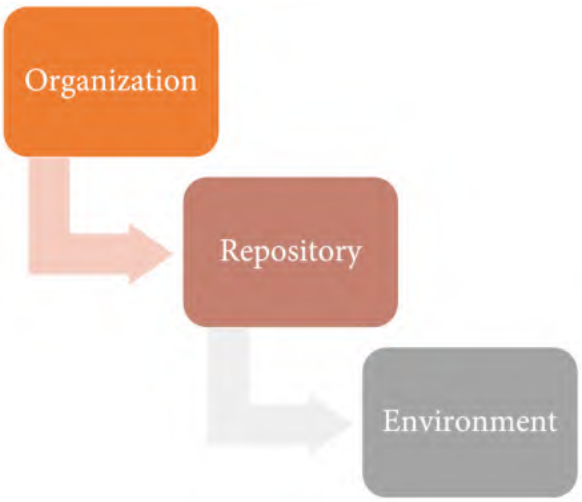


图 1.24 – 配置变量和密钥的层次结构

组织的密钥和变量与代码库的密钥和变量工作方式相同。您可以在 **Settings | Secrets and variables | Actions**（设置 | 密钥和变量 | 操作）下创建一个密钥或变量。新的组织密钥或变量可以为以下内容设置访问策略：

- 所有库（All repositories）
- 私有库（Private repositories）
- 选定库（Selected repositories）

选择“选定库”时，可以授予对单个库的访问。除了通过 UI 设置这些值外，还可以使用 GitHub CLI。您可以使用 gh secret 或 gh variable 来创建新条目：

```
$ gh secret set secret-name
$ gh variable set var-name
```

系统会提示输入密钥或变量的值，或者从文件中读取该值，将其通过管道传递给命令，或使用 -b 或 --body 选项进行指定：

```
$ gh secret set secret-name < secret.txt
$ gh variable set var-name --body config-value
```

1.8. 创建和使用环境

环境用于描述一般的部署目标，例如：开发、测试、暂存或生产。可以使用保护规则来保护环境，并且可以为特定环境提供配置变量和密钥。

1.8.1 Getting ready

我们将首先使用 Web UI 创建一些环境，并添加一些保护规则、密钥和变量。然后，将它们添加到现有的工作流中。

1.8.2 How to do it…

1. 导航到 **Settings | Environments**(设置 | 环境) 并点击 **New environment**(新环境)(见图 1.25):

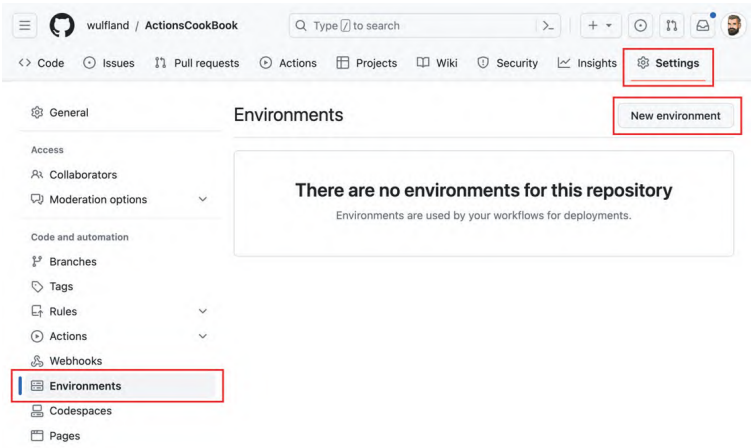


图 1.25 – 库中管理环境

输入名称 Production 并点击 **Configure environment**(配置环境)(见图 1.26):

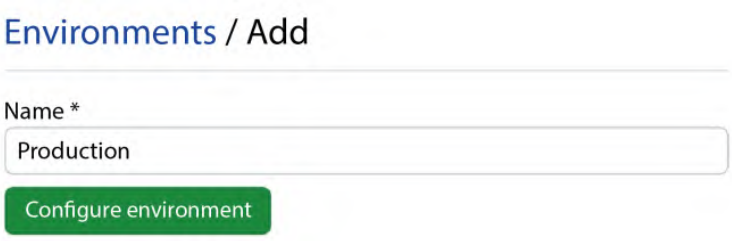


图 1.26 – 创建新环境

2. 将自己添加为必需的审查者, 并点击 **Save protection rule**(保存保护规则)(见图 1.27):

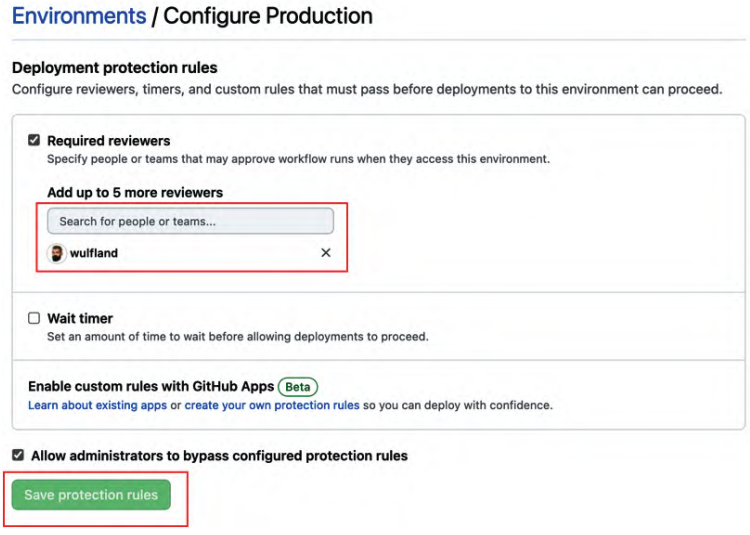


图 1.27 – 配置部署保护规则

3. 在 **Deployment branches and tags**(部署分支和标签) 下,选择 **Selected branches and tags**(选定的分支和标签),点击加号,并为 main 分支添加一个名称模式 (见图 1.28):

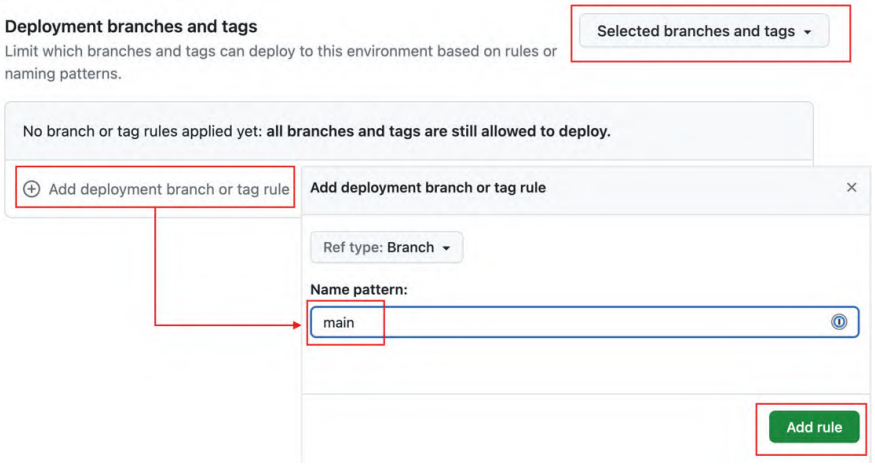


图 1.28 – 配置部署分支和标签

4. 在 **Environment secrets**(环境密钥) 下，点击 **Add secret**(添加密钥) 并添加一个新的 MY_SECRET 密钥，其值为 Open Sesame(见图 1.29)。重复此操作，使用 **Add variable**(添加变量) 添加一个 WHO_TO_GREET 变量，其值为 Production users:

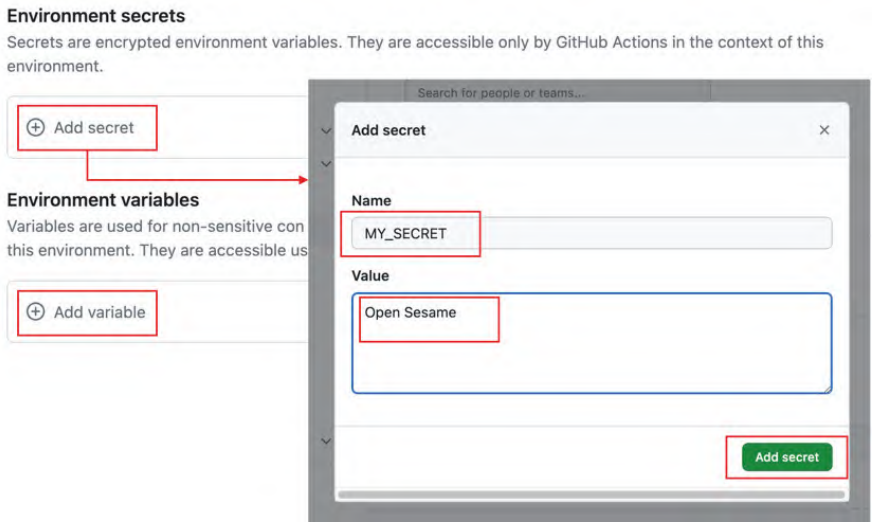


图 1.29 – 向环境添加密钥和变量

5. 重复步骤 1 并创建两个附加环境，Test 和 Load-Test。我们将在接下来的步骤中使用这些环境来展示如何并行执行作业。不必配置部署分支或必需的审查者，只需添加一个 WHO_TO_GREET 变量并设置相应的值即可。结果应该类似于图 1.30:

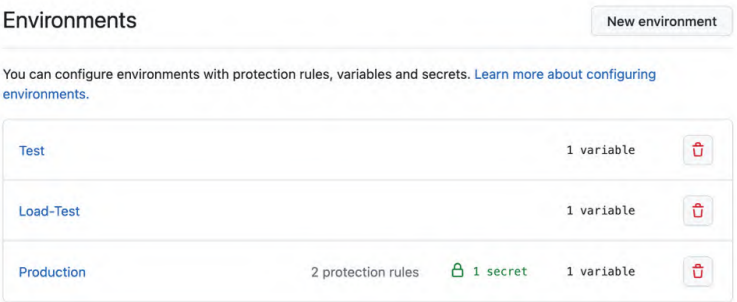


图 1.30 – 设置多个环境

6. 现在，返回工作流程文件并编辑。在 first_job 下面添加一个名为 Test 的新作业，在最新的 Ubuntu 镜像上运行。我们将此作业与 Test 环境关联，为了在 first_job 之后运行此作业，可以使用 needs 属性，并将其设置为所依赖的作业：

```
Test:
  runs-on: ubuntu-latest
  environment: Test
  needs: first_job
```

为了查看密钥如何被环境所覆盖，可以使用一个小技巧。由于 GitHub 会搜索日志输出的密钥值对其进行屏蔽，所以必须修改实际文本。例如，可以使用 sed 's/./& /g'命令来完成此操作。这将把空格添加到密钥的每个字符之间。通过这个小技巧，Test 作业的步骤如下所示：

```
 - run: |
     echo "Hello ${{ vars.WHO_TO_GREET }} 👋 from ${{ github.actor }}."
     sec=$(echo ${{ secrets.MY_SECRET }} | sed 's/./& /g')
     echo "My secret is 👀 '$sec'."
```

7. 接下来，添加一个新的 Load-Test 作业，将其与 Load-Test 环境关联，并在 first_job 之后执行：

```
Load-Test:
  runs-on: ubuntu-latest
  environment: Load-Test
  needs: first_job
```

只需复制 Test 中的步骤即可，无需进行更改。

8. 最后一个作业是 Production 作业。除了名称之外，environment 属性还接受一个 URL，该 URL 将在工作流程设计器中显示。将其设置为相应的 URL。为了展示在工作流程并行执行作业后如何再次合并，我们将在 Test 和 Load-Test 之后运行 Production：

```
Production:
  runs-on: ubuntu-latest
  environment:
    name: Production
    url: https://writeabout.net
  needs: [Test, Load-Test]
```

只需复制前一个作业的步骤即可。

9. 将更改提交到 main 分支，工作流程将自动运行。导航到新的工作流程运行并检查工作流程设计器，其很好地展示了并行执行。工作流程在执行 Production 之前将暂停，并等待批准（请参阅图 1.31）：

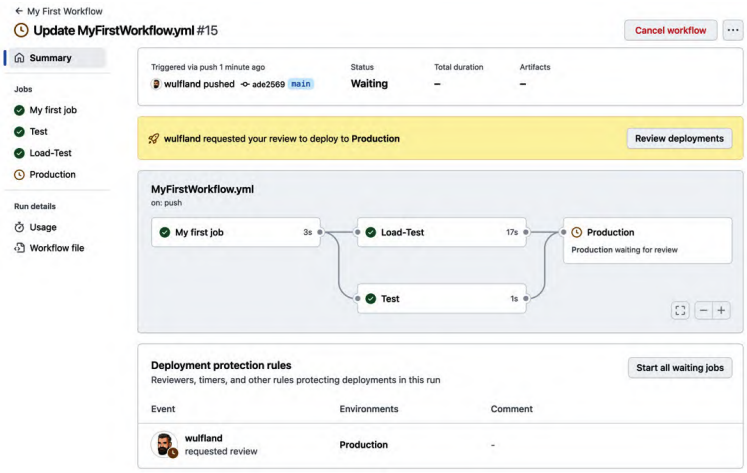


图 1.31 – 工作流程将在具有必需审查者的环境之前停止，并等待批准

10. 点击 **Review deployment**(审查部署)，选中 **Production** 并添加可选评论。点击 **Approve and deploy**(批准并部署) 开始执行 Production 作业（见图 1.32）：

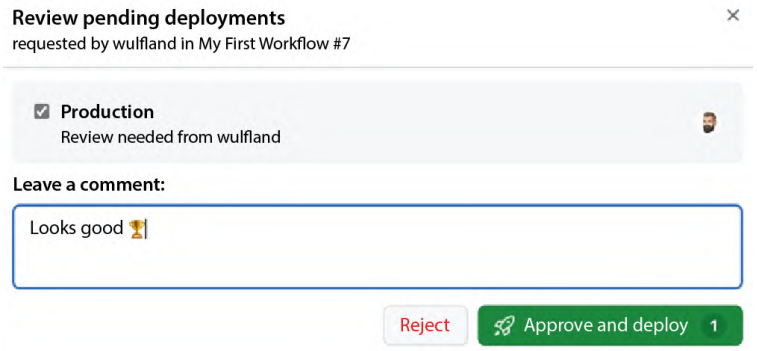


图 1.32 – 批准受保护的环境

工作流将完全执行，结果应该看起来像图 1.33。请注意，URL 显示在 **Production** 环境中。同时，请注意工作流程摘要中的批准历史记录：

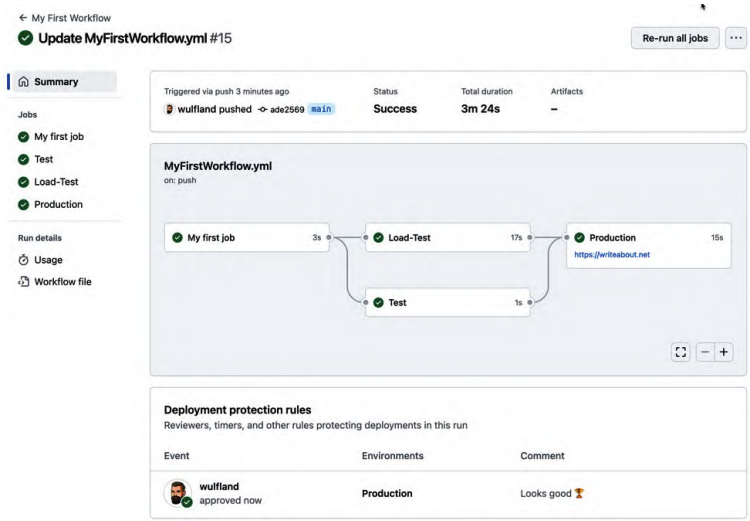


图 1.33 – 工作流的最终摘要

打开各个作业并检查添加的步骤的输出（见图 1.34）。密钥和变量来自代码库，并且只有在设置环境时才会进行覆盖：

图 1.34 – 生产密钥只有在批准后才对生产环境可用

1.8.3 There’s more…

如果正在使用 GitHub CLI 为环境设置密钥或变量，可以使用 --env（-e）参数指定。对于组织密钥，可以将可见性（--visibility 或 -v）设置为 all、private 或 selected。对于 selected，必须使用 --repos（-r）指定一个或多个代码库：

```
$ gh secret set secret-name --env environment-name
$ gh secret set secret-name --org org -v private
$ gh secret set secret-name --org org -v selected -r repo
```

环境比本指南中使用的选项更多。还可以配置一个等待计时器，在工作流执行特定环境的部署作业之前，将工作流暂停 n 分钟（最多 30 天）。

还有一个名为“自定义部署保护规则”的新功能仍处于测试阶段。此功能允许创建 GitHub 应用程序，该应用程序可以暂停部署并等待特定条件。Datadog,Honeycomb,Sentry,New Relic 和 Service Now 已经有应用程序（请参阅https://doc.github.com/en/actions/deployment/deployment/protecting-deployments/configuring-configuring-configuring-custom-custom-deployment-protection-protection-protection-protection-protection-protection-protection-protection-protection-custom-custom-customdeployment-prection-prection-prection-es）。

环境保护规则的力量在于部署分支或标签规则，这可以限制不适用于分支保护规则的代码部署到某些环境。可以包括各种检查 - Codeowners 批准、代码审查者、部署到某些其他环境、SonarQube 质量门[1]和其他许多自动代码检查（请参阅https://docs.github.com/en/repositories/configuring-branches-and-merges-inyour-repository/managing-protected-branches/about-protected-branches）。

[1]质量门：是一种创新的管理策略，它将产品设计流程中的关键节点设定为严格的检查点，就像一道道严密的门坎。

第 2 章 编写和调试工作流

本章更进一步，将了解编写工作流程的最佳实践。这包括使用 Visual Studio Code、本地运行工作流、代码检查、针对分支开发工作流，以及使用高级日志记录和监控。这将是其他章节的基础，并提供了多种编写工作流的选项。

主要内容有：

- 使用 Visual Studio Code 编写工作流
- 分支中开发工作流
- 对工作流程进行代码检查
- 将消息写入日志
- 启用调试记录
- 本地运行工作流

2.1. 环境要求

为了学习本章内容，需要在本地计算机上安装 Visual Studio Code（VS Code），其适用于 Windows（x64、x86 和 Arm64）、Linux（x64、x86 和 Arm64）以及 Mac（Intel 和 Apple silicon）系统。如果还没有安装，可以从这里进行下载安装：https://code.visualstudio.com/download。

此外，请检查本地机器上是否安装了最新版本的 Git。可以从这里获取安装 Git 客户端的说明：https://git-scm.com/downloads。为了在本地运行工作流，还需要安装 Docker。如果使用的是 macOS 系统，请务必按照 Docker 文档中关于如何安装 Docker Desktop for Mac 的步骤进行操作（https://docs.docker.com/docker-for-mac/install）。如果使用的是 Windows 系统，请按照在 Windows 上安装 Docker Desktop 的步骤进行操作（https://docs.docker.com/docker-for-windows/install）。如果你使用的是 Linux 系统，需要安装 Docker Engine(https://docs.docker.com/engine/install)。

2.2. 使用 Visual Studio Code 编写工作流

Visual Studio Code（VS Code）是世界上最受欢迎和广泛使用的代码编辑器之一。由于其灵活性、广泛的扩展生态系统和强大的社区支持，它在开发者社区中获得了显著的人气。

VS Code 与 GitHub 高度集成，提供了诸如 Git 集成、使用 GitHub 账户同步设置、直接访问库，以及使用 GitHub 提供的扩展在编辑器内创建、编辑和管理 GitHub Action 工作流程等功能。这种紧密的集成，简化了工作流程的创建过程，还简化了在 GitHub Action 工作流程上的协作。

这个示例中，我们将安装 GitHub Actions 的 VS Code 扩展，并展示可以用它做什么。

2.2.1 Getting ready

开始之前，请检查电子邮件地址和名称是否正确设置在 git 中：

```
$ git config --global user.email
$ git config --global user.name
```

请记住，我们在公共库中工作。如果想要保持电子邮件地址为隐私，请使用来自 https://github.com/settings/emails 的邮件地址（见图 2.1）：

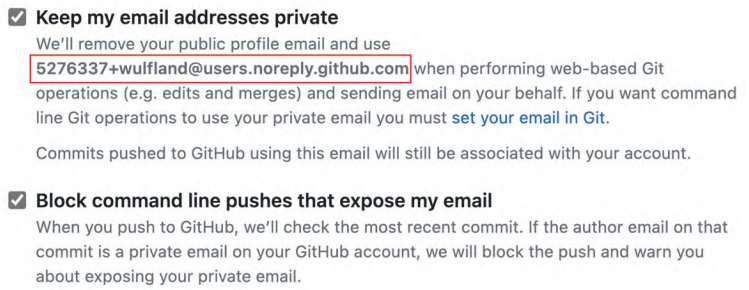


图 2.1 – 公共库中保持电子邮件地址为隐私

这个电子邮件地址由你的 GitHub 用户 ID 和名称组成，位于 users.noreply.github.com 域中：

```
$ git config --global user.email 5276337+wulfland@users.noreply.github.com
```

GitHub 将自动将提交与账户关联，而不会暴露电子邮件地址。

然后，从第 1 章中本地克隆库，以在库的 **Code**（代码）部分的 **Code | Local**（代码 | 本地）下找到相应的 URL（见图 2.2）：

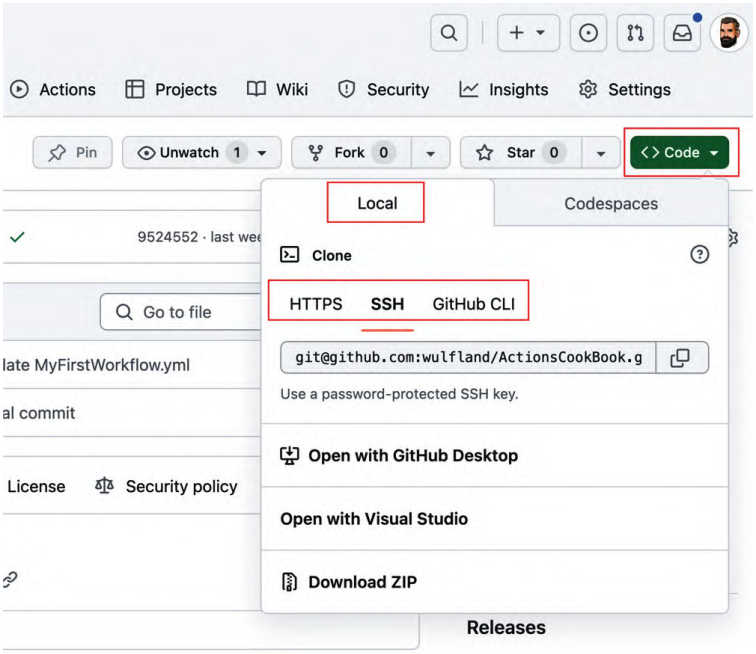


图 2.2 – 本地克隆库

这里使用 SSH 进行身份验证，这样就可以在 1Password 中管理 SSH 密钥——但也可以使用 HTTPS 和一个 PAT 令牌。可以在这里找到关于本地克隆库的更多信息：https://docs.github.com/en/repositories/creating-and-managing-repositories/cloning-a-repository。

2.2.2 How to do it…

1. 打开 VS Code，通过组合键 [Shift]+[Command]+[X] 或 [Ctrl]+[Shift]+[X] 或只需单击左侧栏中的扩展图标来打开扩展窗口（见图 2.3）。搜索 github actions 并安装带有验证徽章的 GitHub 的操作（https://marketplace.visualstudio.com/items?itemName=GitHub.vscodegithub-actions）。如有必要，重启 VS Code，并使用 GitHub 账户登录：

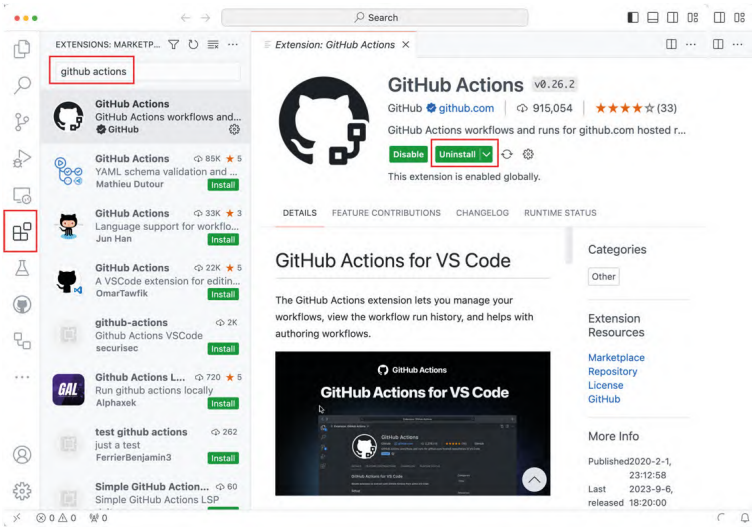


图 2.3 – 为 VS Code 安装 GitHub Actions 扩展

该扩展提供了以下功能：

- 管理工作流和监控工作流运行
- 手动触发工作流
- 工作流和表达式的语法高亮
- 集成文档
- 验证和代码补全
- 智能验证

智能验证尤其有帮助，支持对引用的操作和可重用工作流程进行代码补全，将解析引用操作的参数、输入和输出，并提供验证、代码补全和内联文档功能。

2. 打开本地克隆库。可以在本地克隆的库文件夹的命令行中输入 code .(. 代表当前文件夹)。这将打开一个新的 VS Code 实例并打开当前文件夹。或者在 VS Code 中使用 **File | Open Folder**(文件 | 打开文件夹)，并选择克隆库的所在文件夹。
3. 单击左侧的 GitHub Actions 图标（见图 2.4），并检查 **Current Branch**(当前分支）窗口。可以看到当前分支中的所有工作流运行，每次运行都有由工作流名称和哈希值组

成 ID。还可以展开工作流运行状态，以查看作业和步骤。可以直接在 VS Code 中打开作业日志，或者可以在浏览器中打开步骤日志：

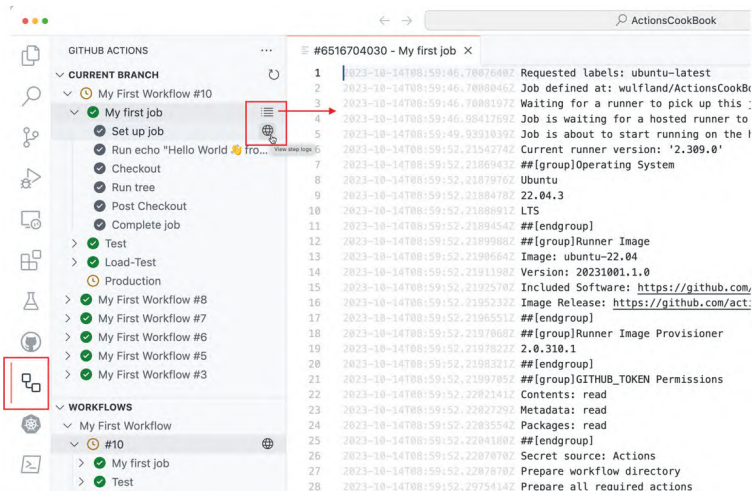


图 2.4 – 检查当前分支中的工作流运行

4. WORKFLOWS 窗口显示了主分支中库的所有工作流程，可以将工作流程固定在列表顶部，打开工作流程文件进行编辑，或者如果工作流具有 workflow_dispatch 触发器，则可以运行工作流的触发器（见图 2.5）：

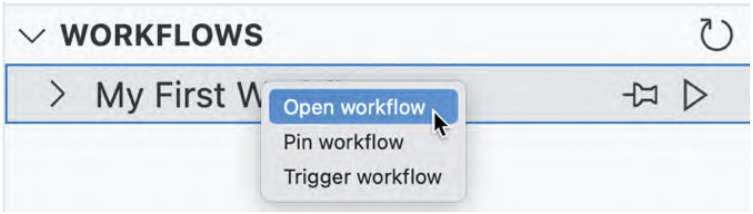


图 2.5 – 工作流窗口允许触发和打开工作流

打开工作流文件。

5. 接下来，在工作流程文件中尝试使用自动补全功能。删除第 15 行中 vars 后面的值。请注意，扩展程序知晓所有的变量和秘钥（见图 2.6）

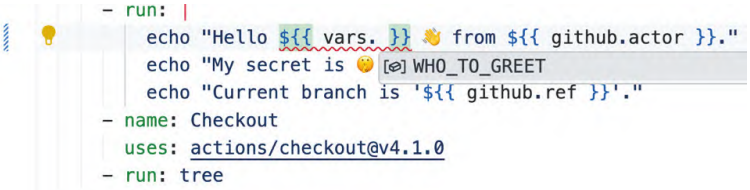


图 2.6 – 自动补全所有已知的秘钥和变量

还具有完整的 workflow 语法，可以在工作流文件中按组合键 [Control]+[Space] 来获取有效元素的列表。

VS Code 会检测到你工作流文件中的问题，并为其提供快速修复（见图 2.7）：

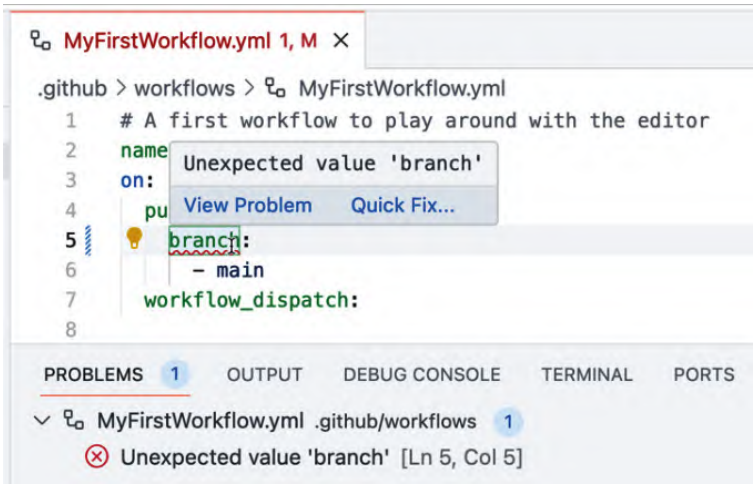


图 2.7 – VS Code 为问题提供快速修复

6. 在 SETTINGS 窗口中，可以找到库中所有的环境、密钥和变量。但不能创建新的环境，可以为环境或库级别添加密钥和变量，以及编辑或删除它们（见图 2.8）：

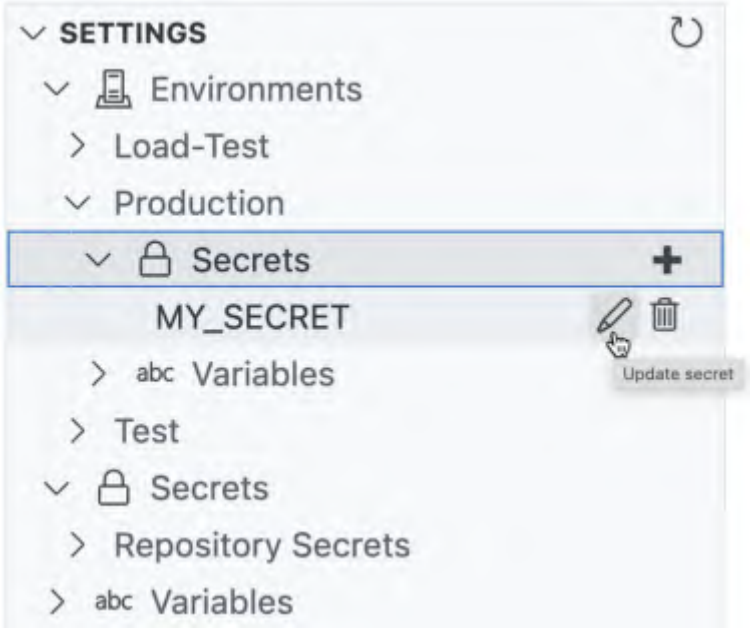


图 2.8 – 在 VS Code 中管理环境、密钥和变量

2.2.3 How it works…

对于 GitHub Actions 扩展，VS Code 是编写和执行工作流的完美编辑器，可以在一个地方完成所有操作，也可以离线工作，并且具有高级语法高亮和针对工作流和表达式的自动补全功能。这就是为什么我们将在本书的其余部分中，使用 VS Code 来编写工作流。

2.2.4 There’s more…

VS Code 不仅仅可以本地安装——也可以直接在浏览器中使用。在 GitHub 库中，只需按下点号 [.] 键，即可直接在浏览器中用 VS Code 打开当前库，或按

下 [Shift]+[>] 在新标签页中打开。要直接在 VS Code 中打开库，也可以导航到 https://github.dev/<owner>/<repository>。

在 GitHub.dev 中，可以像在本地一样处理文件和 Git：首先，提交更改，然后推送到 GitHub。可以安装扩展，并且可以使用 GitHub 账户同步 VS Code 设置。

然而，如果需要终端或安装一些框架，必须在本机进行操作或使用 GitHub Codespaces (https://github.com/features/codespaces)。Codespaces 提供了在 Microsoft Azure 上运行的全功能远程开发环境。每月有 120 小时和 15 GB 的免费存储空间（GitHub Pro 计划则有 180 分钟和 20 GB）；之后，将按分钟和 GB 付费。每 GB 每月成本为 0.07 美元，机器的计算费用每小时在 0.18 美元（双核机器）到 2.88 美元（32 核机器）之间。对于本书来说，我选择了本地版本，以避免用完免费时长的限制，但如果各位读者从未尝试过 Codespaces，我鼓励各位进行尝试，这是为每个项目配备专业开发环境的绝佳方式。

2.3. 分支中开发工作流

全新的库中，最好在主分支上创建工作流。如果必须在开发人员正在工作的活跃库中创建工作流，并且不想影响到其他人的工作，那么可以在一个分支中编写工作流程，并通过拉取请求将其合并回主分支。

然而，某些触发器可能不会按预期工作。如果想要使用 workflow_dispatch 触发器手动运行工作流，第一个操作必须是合并带有触发器的工作流程回主分支，或者使用 API 来触发工作流程。之后，可以在一个分支中编写工作流程，并在通过 UI 触发工作流程时选择该分支。

如果工作流需要 webhook 触发器，如 push、pull_request 或 pull_request_target，根据计划如何使用这些触发器，可能需要在库的分支中创建工作流程。这样，可以在不干扰开发人员工作的前提下测试和调试工作流程，完成时可以将其合并回原始库。

2.3.1 Getting ready

如果尝试前一个示例中的工作流后仍有本地更改，请撤销所有更改，使库保持一个干净的状态。可以通过执行以下命令来实现：

```
$ git reset --hard HEAD
```

也可以在 VS Code 的 Git 窗口中这样做（见图 2.9）：

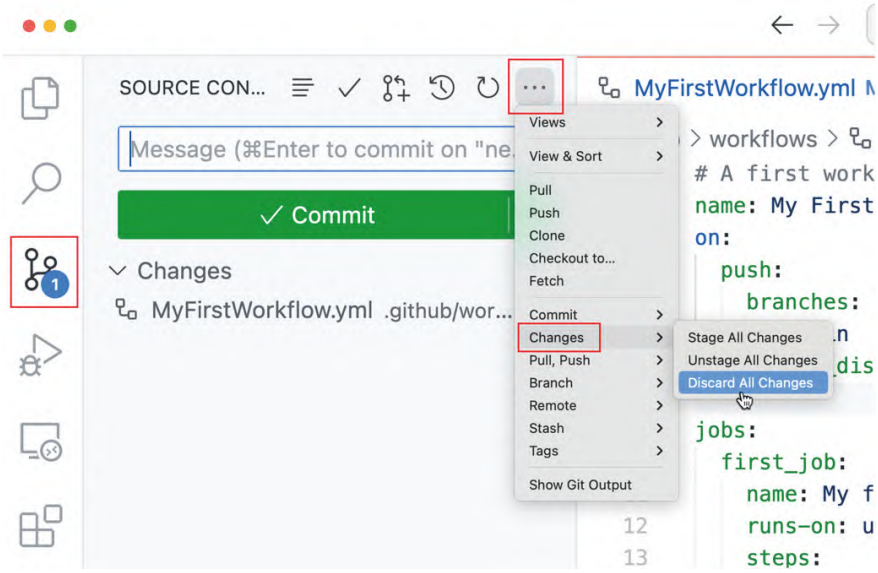


图 2.9 – 丢弃本地更改

2.3.2 How to do it…

1. 在 VS Code 中，点击左下角的 main，选择命令面板中的 **+ Create new branch…**，输入 new-workflow 作为新分支的名称，然后按 [Enter]（见图 2.10）：

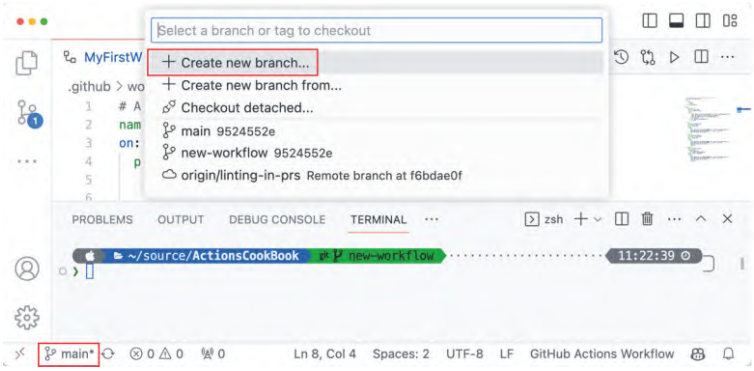


图 2.10 – 在 VS Code 中创建一个新分支

或者，可以使用以下命令：

```
$ git switch -c new-workflow
```

2. 在 VS Code 的资源管理器窗口中创建一个新的工作流文件。找到并标记.github/workflow 文件夹，然后点击 **New file…**（新建文件…）图标。输入 DevelopInBranch.yml 作为文件名，然后点击回车（见图 2.11）：

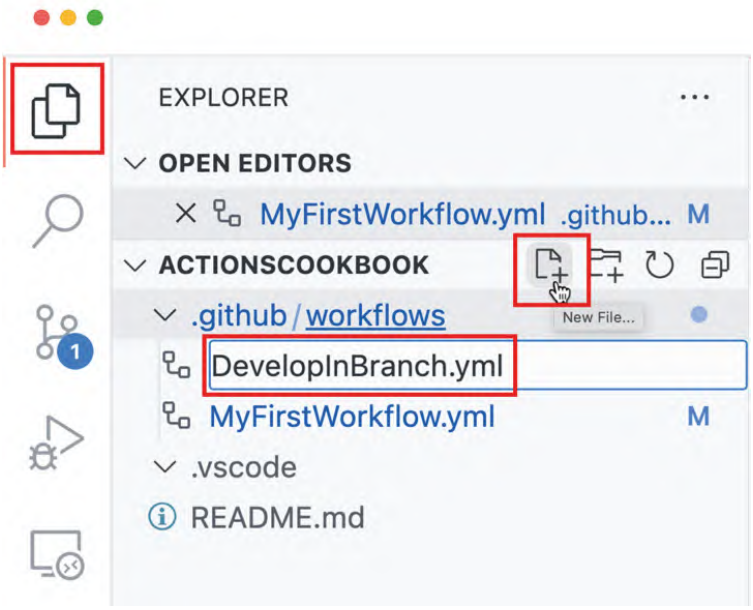


图 2.11 – 在 VS Code 中创建一个新的工作流文件

请注意，VS Code 会自动检测这是一个工作流文件。创建具有 pull_request 和 workflow_dispatch 触发器的工作流，该触发器将一些上下文值输出到控制台，如清单 2.1 所示：

清单 2.1 - 在分支中创建的工作流

```
# workflow to show how to develop workflows in branches
name: Develop in a branch

on: [pull_request, workflow_dispatch]
jobs:
  job1:
    runs-on: ubuntu-latest
    steps:
      - run: |
          echo "Workflow triggered in branch '${{ github.ref }}'."
          Echo "Workflow triggered by event '${{ github.event_name }}'."
          Echo "Workflow triggered by actor '${{ github.actor}}''."
```

3. 添加新文件（暂存更改），输入提交信息，并提交更改（见图 2.12）：

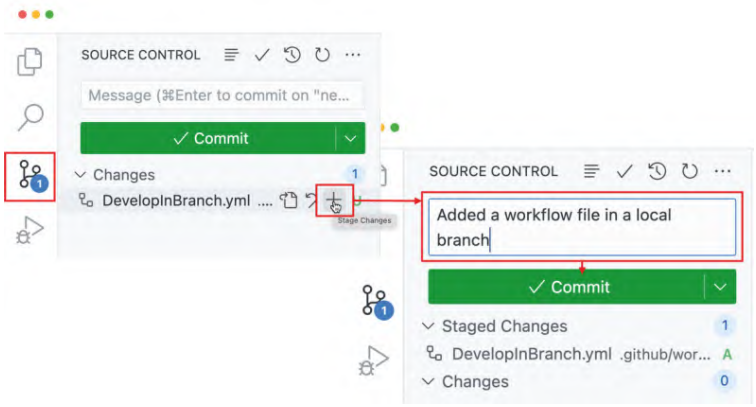


图 2.12 – 在 VS Code 中提交新文件

也可以使用命令行：

```
$ git add .
$ git commit -m "Added a workflow file in local branch"
```

4. 在 VS Code 中，可以通过点击 **Publish Branch**（发布分支）直接推送更改。使用命令行，可以这样：

```
$ git push -u origin new-workflow
```

5. 接下来，我们将为新分支创建一个拉取请求。因为使用了 pull_request 触发器，这将自动运行新工作流。在浏览器中转到库并导航到 **Pull requests**（拉取请求）。Git 检测到已经推送了一个新分支，并会提供给创建拉取请求的选项（比较 & 拉取请求，见图 2.13）：

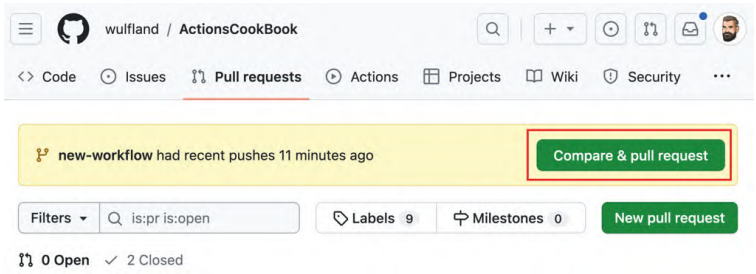


图 2.13 – 浏览器中创建一个新的拉取请求

只需保留默认标题（你之前添加的提交信息）然后点击、**Create pull request**（创建拉取请求）（见图 2.14）：

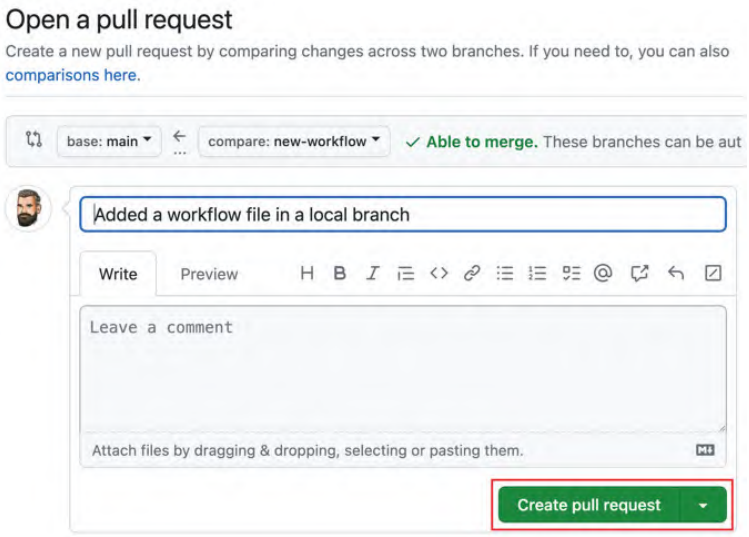


图 2.14 – 创建一个带有标题和描述的拉取请求

也可以使用 GitHub CLI 创建拉取请求：

```
$ gh pr create --fill
```

GitHub CLI

在本书中，将大量使用 GitHub CLI（https://cli.github.com/），其适用于所有平台，可以通过许多包管理器（Homebrew、WinGet、RPM 等）获取。有关更多安装说明，请参阅 https://github.com/cli/cli#installation。安装后，需要使用 gh auth login 进行身份验证（请参阅 https://cli.github.com/manual/gh_auth_login）。

6. 打开你的拉取请求，已经自动执行了工作流程作为检查（见图 2.15）：

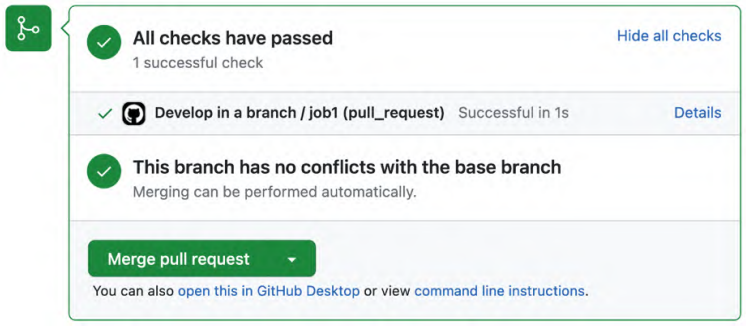


图 2.15 – 由于 pull_request 触发器，工作流将自动运行

你还可以在 **Actions**（操作）选项卡中看到工作流，但不能手动运行工作流。即使有 workflow_dispatch 触发器，只有该触发器的工作流与 main 合并后，运行工作流程的按钮才可用。之后，也将能够在分支上手动运行。

不过，可以在 VS Code 中运行工作流程。打开 **GitHub Actions** 扩展，请使用右上角的刷新图标刷新 **Workflows**（工作流）窗口。现在应该看到新的工作流，并且可以使用箭头按钮手动触发（见图 2.16）：

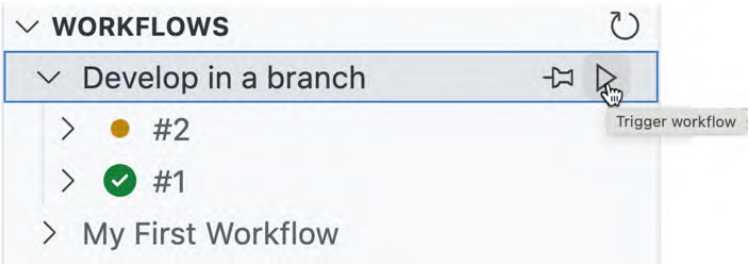


图 2.16 – 在 VS Code 中手动运行工作流

2.3.3 How it works…

当涉及到某些触发器时，存在一些限制。但总的来说，在单独的分支中开发工作流，并使用拉取请求协作更改，这种方法非常有效。

2.3.4 There’s more…

为了进一步发展，我们将向工作流程添加一个代码检查工具，能够在库中的所有工作流程中找出错误、安全问题，以及缺陷。

2.4. 检查工作流

在这个示例中，我们将添加一个代码检查操作，将检查工作流，并直接在拉取请求中给出反馈。

2.4.1 Getting ready

打开开放的拉取请求中拥有的分支上的工作流程文件，先不要合并更改。

2.4.2 How to do it…

1. 转到市场并搜索 `actionlint`，找的操作来自 `devops-actions`。该操作需要访问工作流文件，所以必须使用 `checkout` 操作检出库。在作业的末尾添加以下两个步骤：

```
- uses: actions/checkout@v4.1.0
- uses: devops-actions/actionlint@v0.1.2
```

2. 由于我们希望该操作在拉取请求中注释错误，必须给工作流写入拉取请求的权限，稍后再对此进行解释。现在，只需向作业添加一个 `permissions` ：

```
jobs:
  job1:
    runs-on: ubuntu-latest
    permissions:
```

```
      contents: read
      pull-requests: write
```

3. 将更改提交到 new-workflow 分支并推送到远程。这将触发构建，工作流就应该能够无错误地完成。
4. 为了测试代码检查过程，需要添加一些恶意代码。如果在运行事件中使用用户控制的输入，例如：拉取请求的标题，攻击者可能会通过在标题中注入脚本来利用这一点。如果一个公共库，人们可以从分支创建拉取请求，这会变得相当危险。假设使用 echo 输出拉取请求的标题：

```
  - run: echo "${{ github.event.pull_request.title }}"
```

创建一个标题为 "Hi";ls $GITHUB_WORKSPACE;echo "-" 的拉取请求将执行以下命令：

```
$ echo "Hi";ls $GITHUB_WORKSPACE;echo "-"
```

脚本ls $ GITHUB_WORKSPACE将在无错误的情况下执行，还可以找到其他方法来注入更有害的内容。

在作业的步骤中，添加以下行：

```
steps:
  - run: echo "PR title is '${{ github.event.pull_request.title
}}'.
```

5. 将更改提交到 new-workflow 分支并推送到远程。这次，工作流程应该会失败，拉取请求的检查会显示一个错误，如图 2.17 所示：

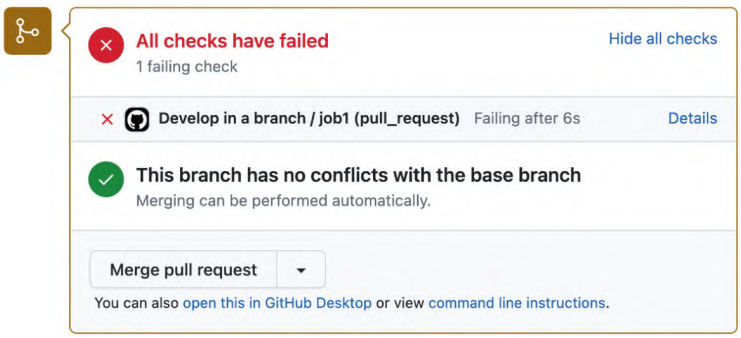


图 2.17 – 如果工作流中存在潜在的脚本攻击，代码检查可能会使拉取请求失败

6. 在拉取请求中，导航到 **Files changes**（文件更改）。请注意，代码检查操作已在正确的行号上对拉取请求进行了注释，该行包含潜在的脚本注入攻击（见图 2.18）：

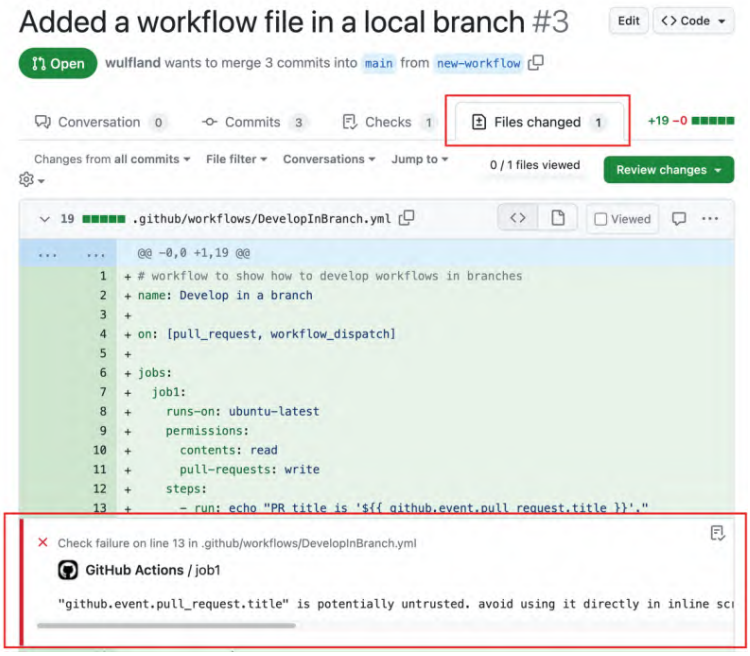


图 2.18 – 工作流文件中的拉取请求注释

2.4.3 How it works…

devops-actions/actionlint（https://github.com/marketplace/actions/rhysd-actionlint）操作是一个封装器，用于 @rhysd（这是 GitHub 用户）的 actionlint 容器（见 https://github.com/rhysd/actionlint），可以在本地或网络上运行 actionlint。其对工作流进行大量的检查（见 https://github.com/rhysd/actionlint/blob/main/docs/checks.md 获取完整列表），该操作是一个封装器，会在库中找到的所有工作流程上运行 actionlint。因此，必须先检出库。然后，该操作使用问题匹配器（见 https://github.com/actions/toolkit/blob/main/docs/problem-matchers.md）来在工作流程文件中注释拉取请求。问题匹配器可以使用正则表达式模式，从结果文件中读取发现的问题，并用它们注释拉取请求。通过工作流命令 add-matcher 和 remove-matcher 激活和停用，工作流命令可以在工作流程步骤和操作中使用，用于与工作流程和运行器机器通信。也可以用于向工作流程日志写入消息，将值传递给其他步骤或操作，设置环境变量，或写入调试消息。

工作流程命令使用具有特定格式的 echo 命令：

```
echo "::workflow-command param1={data},param2={data}::{command value}"
```

如果使用 JavaScript，工具包（https://github.com/actions/toolkit）提供了许多封装器，可以用来替代 echo 写入标准输出。在后续部分中，我将介绍一些有用的写入工作流程日志和注释文件的工作流程命令示例。

问题匹配器通过以下工作流程命令添加：

```
echo "::add-matcher::$GITHUB_ACTION_PATH/actionlint-matcher.json"
```

该命令采用结果文件的路径，可以使用 remove-matcher 并传入所有者来禁用匹配：

```
echo "::remove-matcher owner= actionlint::"
```

为了访问拉取请求，工作流使用 GITHUB_TOKEN，这是一个由 GitHub 自动创建的特殊令牌，可通过 github 上下文（github.token）或 secrets 上下文（secrets.GITHUB_TOKEN）访问。即使工作流没有将其作为输入或环境变量提供，GitHub Action 也可以访问该令牌。该令牌可用于在访问 GitHub 资源时对工作流进行身份验证，默认权限可以设置为宽松（读写）或受限（只读），但可以在工作流程中调整权限。可以在工作流程日志的 **Set up job | GITHUB_ TOKEN Permissions**（设置作业 | GITHUB_TOKEN 权限）下查看工作流权限。请记住，最佳实践是始终是显式设置工作流所需的权限。

所有其他权限将自动设置为无。权限可以针对单个作业或整个工作流程设置。

我们示例的情况为，给出了工作流作业读取内容和写入拉取请求的权限：

```
permissions:
  contents: read
  pull-requests: write
```

2.4.4 There’s more…

本示例，我们通过简单地添加 pull_request 触发器到工作流程，将工作流程代码检查作为拉取请求的检查。然而，即使检查失败，仍然能够将更改合并回主分支。为了防止具有代码检查错误的工作流程被合并，可以启用分支保护（https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/managing-protected-branches/about-protected-branches）或创建规则集（https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/managing-rulesets/about-rulesets）。与 codeowners（https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-code-owners） 结合使用，可以确保只有没有代码检查错误，并且需要在团队或个人对工作流审查后，才能合并回主分支。

2.5. 将消息写入日志

基于现有结果文件，问题匹配器所能做到的，也可以通过使用工作流命令将单个警告或错误事件写入日志来实现。在这个示例中，我们将向工作流程添加一些输出，并对工作流文件进行注释。

2.5.1 Getting ready

确保仍然打开着上一个示例中的拉取请求。只需使用 VS Code 添加其他更改，推送将自动触发工作流程。

2.5.2 How to do it…

1. 在 VS Code 中的 new-workflow 分支中打开.github/workflows/DevelopInBranch.yml，并在检出操作之前直接添加以下代码：

```
- run: |
  echo "::debug::This is a debug message."
  echo "::notice::This is a notice message."
  echo "::warning::This is a warning message."
  echo "::error::This is an error message."
```

   这将会将不同类型的消息写入工作流日志和工作流摘要。
2. 提交并推送更改。这将自动触发一个新的工作流运行。
3. 打开工作流日志并检查输出。它应该看起来像图 2.19：

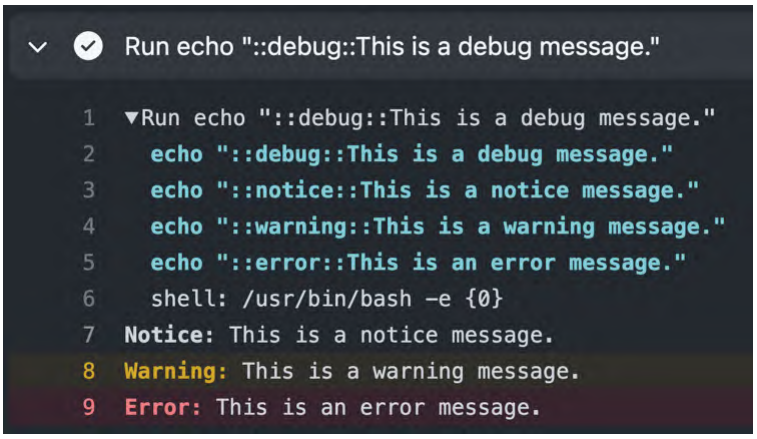


图 2.19 – 将消息写入工作流日志

   请注意，调试消息是不可见的。检查工作流摘要，看看它是否包含了我们的消息，以及来自代码检查的错误（结果应该看起来像图 2.20）：

图 2.20 – 摘要中的工作流注释

4. 为了看到调试消息的作用，可以使用启用调试日志重新运行工作流作业。在工作流摘要中，点击 **Re-run jobs | Re-run all jobs**（重新运行作业 | 重新运行所有作业），选择 **Enable debug logging**（启用调试日志），然后点击 **Re-run jobs**（重新运行作业）（如图 2.21 所示）

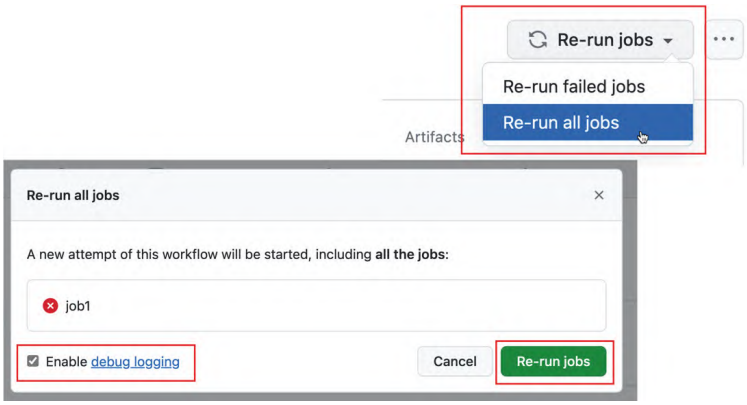


图 2.21 – 启用调试日志记录后重新运行作业

5. 再次检查工作流程日志，看看其他的消息（如图 2.22 所示）：

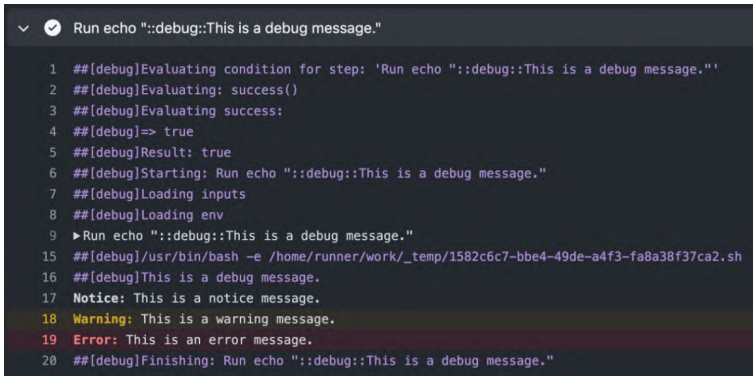


图 2.22 – 启用调试日志的工作流程日志

6. 但是，请注意，notice、warning 和 error 不只是写入日志。我们可以使用它们来注释文件。将以下代码添加到工作流中：

```
- run: |
  echo "::notice
  ↪  file=.github/workflows/DevelopInBranch.yml,line=19,col=11,endColumn=51::There is a
  ↪  debug message that is not always visible!"
  echo "::warning file=.github/workflows/DevelopInBranch.yml,line=19,endline=21::A lot of
  ↪  messages"
  echo "::error title=Script
  ↪  Injection,file=.github/workflows/DevelopInBranch.yml,line=13,col=37,endColumn=68::Potentialscript
  ↪  injection"
```

这将在第 19 行添加一个 notice 注释，在第 19 到 21 行添加一个警告，并在第 13 行的 37 到 68 列添加一个错误。如果行号和缩进不同，请调整这些值！

7. 提交并推送更改。打开拉取请求，并在 **Files changed**（文件更改）选项卡中查看注释（见图 2.23）：

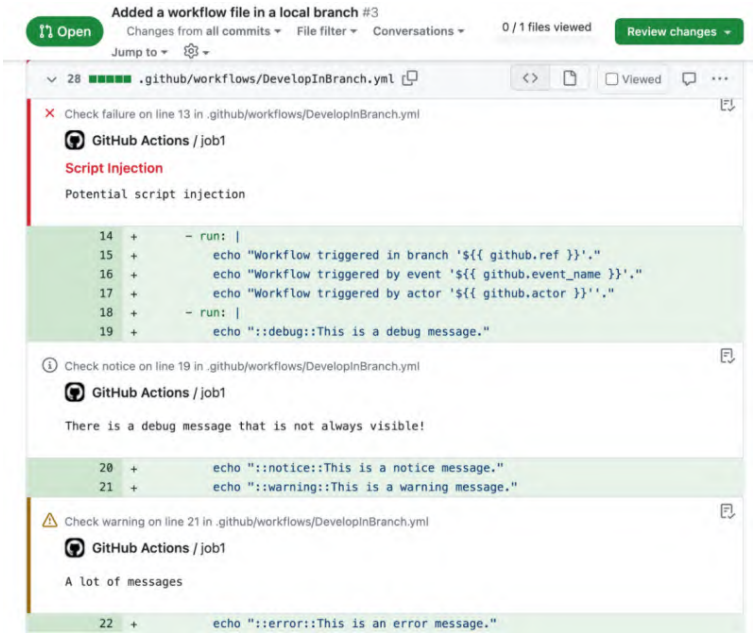


图 2.23 – 拉取请求更改中的工作流注释

2.5.3 How it works…

与匹配器的工作方式相同，可以创建警告和错误消息，并输出到日志中。这些消息将创建一个注释，该注释可以将消息与库中的特定文件关联起来。可选地，消息也可以指定文件中的位置：

```
::notice file={name},line={line},endLine={el},title={title}::{message}
::warning
file={name},line={line},endLine={el},title={title}::{message}
::error file={name},line={line},endLine={el},title={title}::{message}
```

参数如下：

- 标题（Title）：消息的自定义标题
- 文件（File）：引发错误或警告的文件名
- 列（Col）：列/字符编号，从 1 开始
- 结束列（EndColumn）：结束列编号
- 行（Line）：文件中的行号，从 1 开始
- 结束行（EndLine）：结束行号

唯一不能注释文件的命令是调试消息，此工作流命令仅接受消息作为参数。

2.6. 启用调试记录

之前的示例中，可以使用启用了调试日志的选项重新运行失败的任务或所有任务，也可以在库级别启用或禁用调试日志。

2.6.1 How to do it…

我们可以向库添加一个名为 ACTIONS_STEP_DEBUG 的变量，并将其值设置为 true 或 false 来启用或禁用调试日志。这将向工作流日志添加非常详细的输出，以及所有调试消息，并且这些内容将从所有操作中显示出来。

可以使用网页、GitHub CLI 或 VS Code 配置该变量。要使用网页设置变量，请转到 **Settings | Secrets and variables | Actions**（设置 | 秘钥和变量 | 操作）并选择 **Variables**（变量）选项卡（/settings/variables/actions）。单击 **New repository variable**（新库变量）（这将重定向到 /settings/variables/actions/new），输入 ACTIONS_STEP_DEBUG 作为名称，true 作为值，然后单击 **Add variable**（添加变量）。

要使用 CLI 设置，只需执行以下命令：

```
$ gh variable set ACTIONS_STEP_DEBUG --body true
```

如果想在 VS Code 中设置该变量，只需打开 Actions 扩展，导航到 **Variables | Repository Variables**（变量 | 库变量），单击 + 符号（见图 2.24），输入 ACTIONS_STEP_DEBUG，然后按 [Enter]；输入 true，然后再次按 [Enter]。在 VS Code 中，使用更新选项更改变量也非常方便：

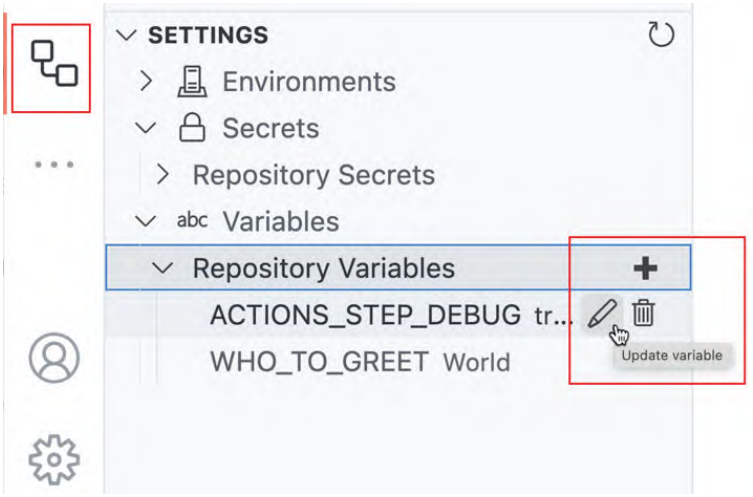


图 2.24 – 在 VS Code 中将步骤调试设置为 true

运行工作流，并检查详细的输出。

2.6.2 There’s more…

也可以将变量 ACTIONS_RUNNER_DEBUG 设置为 true 来激活运行器的附加日志。运行器调试日志将包含在工作流程的日志存档中，可以从工作流程作业日志中下载该存档。如果想了解更多关于监控和故障排除的信息，可以参考https://docs.github.com/en/actions/monitoring-andtroubleshooting-workflows/enabling-debug-logging。

2.7. 本地运行工作流

每次提交工作流并在服务器上运行可能是一个缓慢的过程，特别是对于复杂的工作流。这个示例中，将学习如何使用 act（https://github.com/nektos/act）在本地运行工作流程。

2.7.1 Getting ready

Act 依赖于 Docker 来运行工作流程，请确保 Docker 已经良好的在本地环境中运行。

可以使用不同的包管理器安装 act（https://github.com/nektos/act#installationthrough-package-managers），只需选择适合环境的包管理器，并按照说明进行操作即可。

首次运行 act 时，它会要求使用者选择一个用作默认值的 Docker 镜像，并把该信息保存到 /.actrc。有不同的镜像可供选择，有小镜像（node:16-buster-slim）只支持 NodeJS 而不支持更多。大镜像的大小超过 18 GB。鉴于现今的磁盘空间和互联网，使用大镜像会获得最佳结果。为了运行当前的工作流，可以选择中等大小的镜像（见图 2.25）：

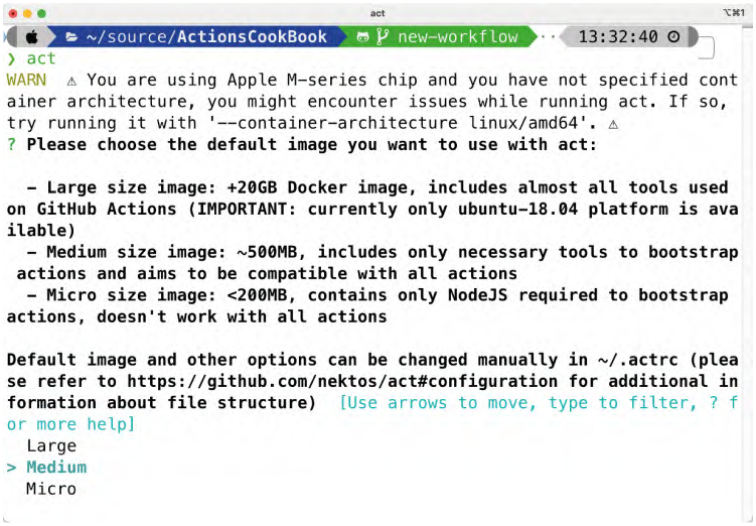


图 2.25 – 首次运行 act 时选择容器镜像

2.7.2 How to do it…

在当前库中打开命令提示符，查看新的工作流分支。act 使用工作流程触发器来执行工作流程，并且默认为 push。由于当前工作流只有 pull_request 触发器，所以必须指定。使用 -l 选项，act 将列出库中所有具有相应触发器的工作流程和作业。执行以下命令：

```
$ act pull_request -l
```

检查工作流程和作业。要进行“模拟运行”[2]，可以使用 -n 选项：

```
$ act pull_request -n
```

[2]dry run：是指在不实际执行操作的前提下，通过模拟流程或系统来验证其逻辑、流程或代码的正确性。

因为未执行工作流的检查，工作流成功完成。要在容器中真正执行工作流，请运行以下命令：

```
$ act pull_request
```

工作流将执行并在检查步骤中失败，就像在服务器上的 pull request 中一样。结果应该类似于图 2.26：

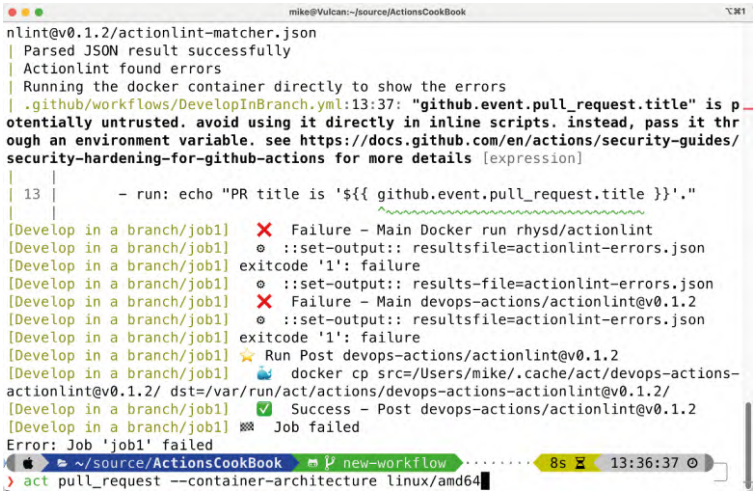


图 2.26 – 工作流检查与服务器上相同的错误失败

我认为在将更改推送到服务器之前，能够在本地运行工作流程是非常强大的。然而，根据工作流，结果可能不是 100% 可靠的，还可能需要使用超过 20 GB 的大 Docker 镜像。

2.7.3 How it works…

act 使用 docker 容器在本地运行工作流，从.github/workflows/ 读取 GitHub Actions，并确定需要运行的操作集。它使用 Docker API 来拉取或构建在工作流文件中定义的必要镜像，并最终根据定义的依赖关系确定执行路径。当确定了执行路径，可使用 Docker API 来为每个操作运行基于先前准备的镜像的容器，环境变量和文件系统都配置都可为与 GitHub 提供的相匹配。

如果你的工作流程使用 GITHUB_TOKEN，那么你必须提供一个个人访问令牌 (PAT)；act 将使用它与 GitHub 通信：

```
$ act -s GITHUB_TOKEN=[insert personal access token]
```

可以使用 GitHub CLI 的 gh auth token 自动将令牌从 CLI 传递给 act：

```
$ act -s GITHUB_TOKEN="$(gh auth token)"
```

2.7.4 There’s more…

act 的问题在于默认 GitHub 托管运行器的镜像非常巨大。为了获得良好的本地性能，不可能包含这些运行器上安装的所有工具。对于 90% 的工作流程来说这非必须，因为操作在 NodeJS

中运行或带来自己的容器，尤其是在 run 步骤的自定义脚本中使用命令行工具时，这是一个问题。

act 使用自定义镜像作为工作流作业运行得很好，可以像这样为作业分配自定义 Docker 镜像，而非依赖 GitHub 托管运行器的工具：

```
jobs:
  container-test-job:
    runs-on: ubuntu-latest
    container:
      image: custom-image:latest
```

这样，本地执行和服务器上的执行基本上是一样的。这也是一个不错的选择，如果必须长期保持构建环境，可以在这里了解更多关于在容器中运行作业的信息：https://docs.github.com/en/actions/using-jobs/running-jobs-in-a-container。

第 3 章 构建 GitHub Actions

已经介绍了如何编写工作流程，并且已经使用了市场上的一些操作，是时候完全理解操作是什么，以及它们是如何工作的了。本章中，我将解释不同类型的操作，以便了解如何自行编写操作：

主要内容有：

- 创建 Docker 容器操作
- 添加输出参数和使用作业摘要
- 创建一个 TypeScript 操作
- 创建一个组合操作
- 将操作分享到市场上
- 开发自定义操作的最佳实践

将了解如何将参数传递给操作，并在后续的工作流程步骤中使用输出参数。还将介绍如何写入工作流程日志，从操作内部注释文件中的更改，以及如何创建丰富的作业摘要。

3.1. 环境要求

接下来的示例中，我将使用 VS Code。需要 VS Code 和本地 git 客户端来进行操作。

如果想在本地运行我们作为操作提供的 Docker 镜像，还需要在安装 Docker。当然，也可以使用 GitHub Codespaces。

对于 TypeScript 操作，需要安装一个相对较新的 Node.js 版本。如果使用版本管理器，如 nodenv 或 nvm，可以在库的根目录运行 nodenv install 来安装 package.json 中指定的版本。否则，20.x 或更高版本应该可以工作。请查看 https://github.com/actions/typescript-action 的 README.md 文件，以获取更新的要求。如果不想安装 Node.js，只需要在 GitHub Codespaces 中跟随相应示例的操作即可。

3.2. 创建 Docker 容器操作

本示例中，将使用 Dockerfile 创建 Docker 容器，并在持续集成（CI）工作流程中使用它。每次更改内容时，工作流程都会运行该操作。

3.2.1 Getting ready

创建一个名为 DockerActionRecipe 的新库。将其设为公开，并用 README 文件初始化（见图 3.1）：

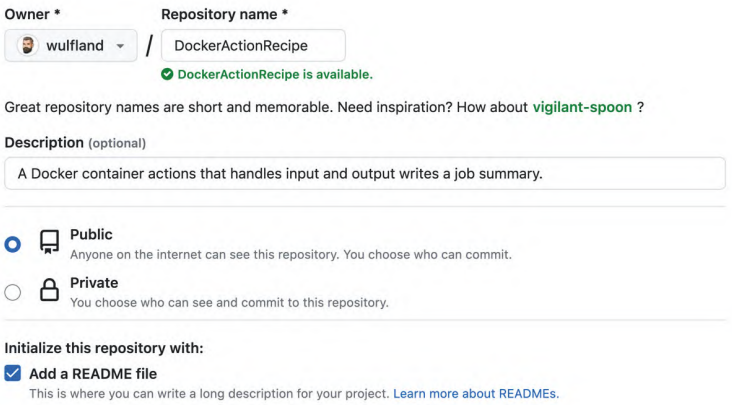


图 3.1 – 为 Docker 容器操作创建一个新库

在本地克隆该库，并在 VS Code 或 GitHub Codespaces 中打开它。

3.2.2 How to do it…

1. 在库的根目录下创建一个名为 Dockerfile 的新文件，并向文件中添加以下内容：

```
# Container image that runs your code
FROM alpine:latest
CMD echo "Hello World"
```

这将基于最新的 Alpine 镜像创建一个镜像，并添加一个将“Hello World”输出到控制台的层。

2. 使用以下命令在本地运行 Docker 容器：

```
$ docker run $(docker build -q .)
```

它将创建一个镜像（docker build）并运行它（docker run），应该能够在控制台上看到 Hello World。

3. 为了更灵活，创建一个名为 entrypoint.sh 的新文件，并添加以下内容：

```
#!/bin/sh -l
echo "Hello World"
```

4. 现在，调整 Dockerfile，使其执行脚本而不是直接写入控制台。将脚本文件复制到容器的根目录，然后将其作为入口点：

```
FROM alpine:3.10
COPY entrypoint.sh /entrypoint.sh
RUN chmod +x entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]
```

5. 请注意，这里添加了 chmod +x entrypoint.sh 命令来使脚本可执行。否则，本地运行容器时会失败，并显示一条消息：exec: /entrypoint.sh": permission denied(权限被拒绝)。

在所有类 Unix 的系统中，可以在本地运行 `chmod +x entrypoint.sh`，并在提交到 git 时附加该属性。在 Windows 上，可以使用 git 来设置文件权限：

```
$ git add entrypoint.sh
$ git update-index --chmod=+x entrypoint.sh
```

再次运行 Docker 容器，应该会再次看到 Hello World——这次是从脚本文件输出：

```
$ docker run $(docker build -q .)
```

6. 在这个操作中，我们想要使用一个输入参数，这就是要对脚本参数化的原因。将单词 World 替换为传递给 Docker 的参数（$@ 代表所有参数）：

```
#!/bin/sh -l
echo "Hello $@"
```

尝试再次在本地运行容器，并传入一些单词。容器将进行输出，如下所示：

```
$ docker run $(docker build -q .) foo bar
> Hello foo bar
```

7. 接下来，向库中添加一个名为 action.yml 的新文件，并添加以下输入：

```
name: 'Docker Action Recipe'
description: 'Greet someone'
inputs:
  who-to-greet:
    description: 'Who to greet'
    required: true
    default: 'World'
  runs:
    using: 'docker'
    image: 'Dockerfile'
    args:
      - ${{ inputs.who-to-greet }}
```

8. 此时，操作已经准备好了。为了测试，需要添加一个名为.github/workflows/ci.yml 的本地工作流文件，将在每次推送时运行。它会下载库，并使用自定义输入参数执行相应的操作：

```
name: Action CI

on: [push]

jobs:
  ci:
```

```
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4.1.1
      - name: Run my own container action
        uses: ./
        with:
          who-to-greet: '@wulfland'
```

引用本地操作

请注意，我们是通过本地路径./ 引用操作的——这就是为什么必须首先使用检出操作。工作流程将使用与工作流程运行时相同的版本，也可以通过指定 <owner>/DockerActionRecipe@main 来正常引用操作——就像从另一个库引用它一样。

9. 提交并推送所有更改。推送触发器将自动运行工作流，可以检查操作的输出。它应该看起来像图 3.2 所示：

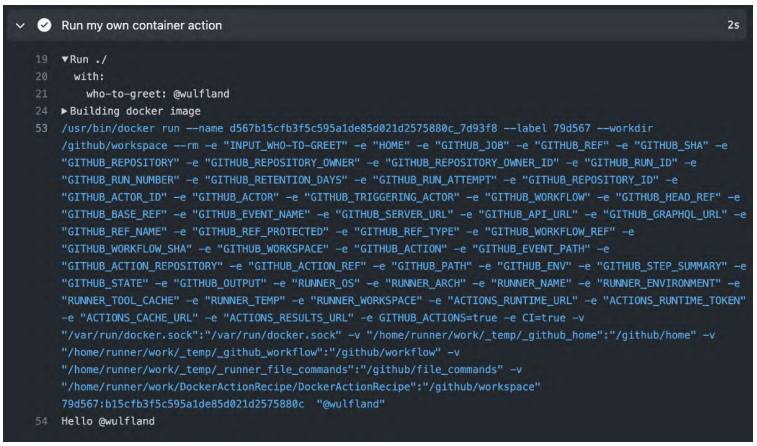


图 3.2 – 工作流操作的输出

通过 Docker 守护程序和传递给操作的参数，来检查操作的输出。

3.2.3 How it works…

有三种不同类型的操作：

- Docker 容器操作
- JavaScript 操作
- 复合操作

Docker 容器操作仅运行在 Linux 上，而 JavaScript 操作和复合操作可以在没有平台限制。所有操作都由一个名为 action.yml(或 action.yaml) 的文件定义，该文件包含定义操作的元数据。这个文件名不能变，所以一个操作必须位于自己的库或文件夹中。action.yml 文件中的 runs 部分定义了操作的类型。

Docker 容器操作包含它们所有的依赖项在容器中，因此一致。允许使用其他语言开发操作——唯一的限制是必须运行在 Linux 上。由于检索或构建镜像和启动容器所需的时间，Docker 容器操作比 JavaScript 操作慢。

Docker 容器操作可以引用容器注册表中的镜像，例如：Docker Hub 或 GitHub Packages，或者可以在运行时构建相应的 Dockerfile。这时，必须在 action.yml 文件中将 Dockerfile 指定为镜像名称。

3.2.4 There’s more…

容器操作非常强大，可以使用任何语言进行编写。可以向工作流返回输出参数，向工作流日志写入消息，在拉取请求中注释文件，并编写作业摘要。在下一个简短的示例中，我们将添加输出参数并写入作业摘要。

3.3. 添加输出参数并使用作业摘要

在这个示例中，我们将向操作添加一个输出参数，该参数可以在后续步骤中使用，并且将向工作流作业摘要写入相应内容。

3.3.1 Getting ready

您必须完成之前的示例才能继续这个示例。

3.3.2 How to do it…

1. 打开 action.yml 文件，并在 inputs 部分的下方（runs 部分之前）添加以下代码：

```
outputs:
  answer:
    description: 'The answer to everything (always 42)'
```

这定义了一个 ID 为 answer 的输出。

2. 接下来，打开 entrypoint.sh，并在文件末尾添加以下内容：

```
echo "answer=42" >> $GITHUB_OUTPUT
```

这将把 answer 的输出值设置为 42。

3. 现在，在 entrypoint.sh 末尾添加以下内容，以向步骤摘要写入一些 Markdown 和 HTML 内容：

```
echo "### Hello $@! :rocket:" >> $GITHUB_STEP_SUMMARY
echo "<h3> The answer from Deep Thought is 42 :robot:</h3>" >>
$GITHUB_STEP_SUMMARY
```

4. 在提交更改之前，我们必须调整工作流文件.github/workflows/ci.yml，以便它能够使用输出参数。为执行我们操作的步骤添加一个 ID，命名为 my-action，如下所示：

```
- name: Run my own container action
  id: my-action
  uses: ./
  with:
    who-to-greet: '@wulfland'
```

添加另一个步骤，将结果输出到工作流日志：

```
- name: Output the answer
  run: echo "The answer is ${{ steps.my-action.outputs.answer }}"
```

5. 当 Docker 容器操作返回意外结果时，为了使 CI 构建失败，必须添加一个新的步骤，只有当结果不是预期时才执行。返回非零值（例如，exit 1）将向工作流程指示该步骤已失败。可以使用文件注释来指示错误的位置（就像我们在第 2 章中所做的那样）：

```
- name: Test the container
  if: ${{ steps.my-action.outputs.answer != 42 }}
  run: |
    echo "::error file=entrypoint.sh,line=4,title=Error in container::The answer was not
    ↪  expected"
  exit 1
```

6. 提交并推送所有更改。构建将自动运行并应该成功。检查工作流日志，并确保输出参数已正确传递到下一步（见图 3.3）：

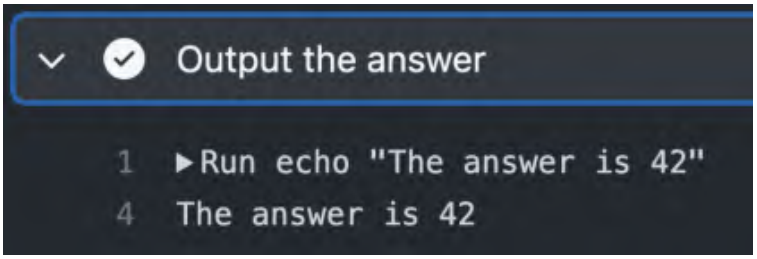


图 3.3 – Docker 容器操作的输出

同时，在工作流摘要页面上查看作业摘要，它已经渲染了 Markdown/HTML(见图 3.4)：

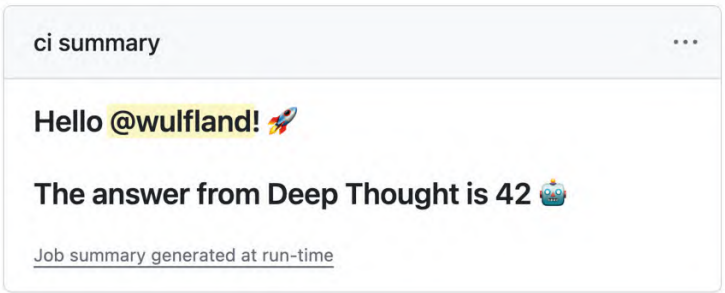


图 3.4 – 工作流摘要页面上的作业摘要

7. 最后，我们希望确保当返回意外值时，我们的 CI 构建会失败。打开 entrypoint.sh 并将 42 更改为其他值（例如，7）。
   切换到另一个分支，提交并推送，然后创建一个新的拉取请求：

```
$ git switch -c fail-ci-build
$ git commit -m "Fail CI build"
$ git push -u origin fail-ci-build
$ gh pr create --fill
```

该拉取请求的检查将失败，并且它将注释相应的文件（见图 3.5）：

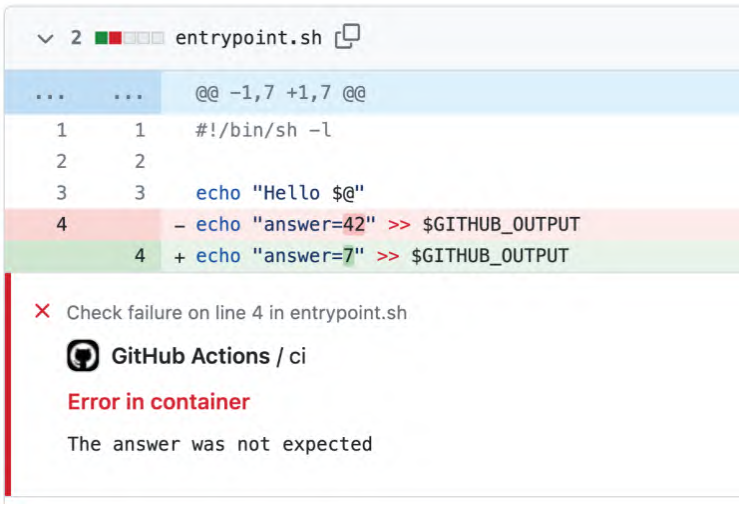


图 3.5 – 如果输出意外，CI 构建拉取请求将失败

请注意 GitHub 如何标记更改并，注释确切的行。

3.3.3 How it works…

让我们了解代码是如何工作的。

环境文件

将输出值传递给后续步骤和作业是通过将“名称-值”对管道传递到环境文件来实现的– 即 $GITHUB_OUTPUT：

```
echo "{name}={value}" >> "$GITHUB_OUTPUT"
```

>> 操作符将名称-值对追加到文件的末尾。文件的路径和名称存储在 $GITHUB_OUTPUT 环境变量中。可以通过步骤上下文（steps context）中步骤的 outputs 属性来访问该输出：

```
"${{ steps.<step-id>.outputs.<name> }}"
```

输出是 Unicode 字符串，大小不能超过 1 MB。一次工作流运行中所有输出的总大小，不能超过 50 MB。

环境文件的另一个用例是，为作业中的后续步骤设置环境变量。相应环境文件的路径存储在 $GITHUB_ENV 中，只需将一个“名称-值”对追加到文件末尾，如下所示：

```
echo "{environment_variable_name}={value}" >> "$GITHUB_ENV"
```

请注意，名称区分大小写！以下是一个完整的示例，展示了如何在一步中设置环境变量，并在后续步骤中使用 env 上下文进行访问：

```
steps:
  - name: Set the value
    id: step_one
    run: |
      echo "action_state=yellow" >> "$GITHUB_ENV"

  - run: |
      echo "${{ env.action_state }}" # This will output 'yellow'
```

作业摘要

可以为工作流中的每个作业设置自定义 Markdown，渲染后的 Markdown 将在工作流程运行的摘要页面上显示。作业摘要可用于显示内容（例如：测试或代码覆盖率结果），以便查看工作流程运行结果时，无需进入日志或外部系统。作业摘要支持 GitHub 风格的 Markdown，但由于 Markdown 是 HTML，因此也可以将 HTML 输出到作业摘要文件中。这个示例中，我的 GitHub 用户名 @wulfland 是一个链接到我个人资料并带有预览的链接，并且所有 GitHub 表情符号都受支持。

通过将 Markdown 追加到以下文件，可以将步骤结果添加到作业摘要中：

```
echo "{markdown content}" >> $GITHUB_STEP_SUMMARY
```

各个步骤是隔离的，并且限制为 1 MiB(1.04858 MB)，以便单个步骤的畸形 Markdown 不会破坏后续步骤的 Markdown 渲染。只有 20 个步骤可以写入摘要，之后的步骤的输出都不可见。

在下一个示例中，我们将使用工具包向作业摘要写入更复杂的内容。

有关环境文件和作业摘要的完整参考，请参阅 https://docs.github.com/en/actions/using-workflows/workflow-commands-for-githubactions?tool=bash#environment-files。

表达式和条件执行

在第 1 章和第 2 章中，使用了表达式（${{ ... }}）从上下文对象中输出值。本示例中，我们使用了一个带有步骤的 if 属性的表达式来有条件地执行：

```
- name: Test the container
  if: ${{ steps.my-action.outputs.answer != 42 }}
```

此步骤仅在步骤上下文中，需要有 ID 为 my-action 操作的输出答案不等于 42 时才会执行。

if 属性也存在于作业中，以有条件地执行这些作业。

当您在 if 属性中使用表达式时，可以省略 ${{ }} 表达式语法，GitHub Actions 会自动将 if 条件评估为表达式。

对于条件执行，表达式必须返回 true 或 false。要编写表达式并将上下文与静态值进行比较，可以使用表 3.1 中提供的运算符：

运算符	描述
( )	逻辑分组
[ ]	索引
.	属性解引用
!	非
<, <=	小于，小于或等于
>, >=	大于，大于或等于
==	等于
!=	不等于
&&	与
||	或

表 3.1 — 表达式运算符

GitHub 提供了一组内置函数，可以在表达式中使用，帮助搜索字符串、格式化输出或处理数组。请参阅表 3.2 以获取可用函数的列表：

函数	描述
contains(search, item)	如果 search 包含 item，则返回 true。 例如： contains('Hello world', 'llo') 返回 true contains(github.event.issue.labels.*.name, 'bug') 如果与事件相关的议题包含标签 bug，则返回 true。
startsWith(search, iten)	当 search 以 item 开头时，返回 true。
endsWith(search, item)	当 search 以 item 结尾时，返回 true。
format(string, v0, v1, ...)	替换字符串中的值。 例如： format('Hello {0} {1} {2}', 'Mona', 'the', 'Octocat') returns 'Hello Mona the Octocat'.

join(array, optS)	数组中的所有值将连接成一个字符串。如果提供了可选的分隔符 optS，其将插入到连接的值之间。如果没有提供分隔符，则默认使用逗号作为分隔符。
toJSON(value)	返回值的格式化 JSON 表示。
fromJSON(value)	为 value 返回一个 JSON 对象或 JSON 数据类型。
hashFiles(path)	为匹配路径模式的一组文件返回唯一的哈希值。

表 3.2 – GitHub 中表达式的内置函数

3.3.4 There’s more…

还有一些特殊函数用于检查当前作业的状态。下面的示例中，最后一步只有在作业的先前步骤失败时才会执行——返回一个非零值：

```
steps:
  - run: exit 1
  - name: The job has failed
    if: ${{ failure() }}
```

有关用于检查作业状态的可用函数列表，请参阅表 3.3：

函数	描述
success( )	如果之前的步骤都没有失败或取消，则返回 true。
always( )	即使之前的某个步骤取消，也返回 true，并确保该步骤必定执行。
cancelled( )	仅当工作流取消时返回 true。
failure( )	如果作业的前一个步骤失败，则返回 true。

这些函数还可以帮助有条件地执行某些步骤并执行（例如：清理操作）。要了解更多关于条件执行的表达式，请参阅：https://docs.github.com/en/actions/learn-github-actions/expressionsandhttps://ocs.github.com/en/actions/using-jobs/using-conditions-to-controljob-execution。

3.4. 创建 TypeScript 操作

这个示例中，将从模板创建一个基本的 TypeScript 操作，构建和发布，并在工作流中使用。

3.4.1 Getting ready

确保安装了相对较新版本的 Node.js(https://nodejs.org/en/download)。请按照以下步骤操作：

1. 前往 https://github.com/actions/typescript-action，然后点击右上角的 **Use this template | Create a new repository**(使用此模板 | 创建新库)(如图 3.6 所示):

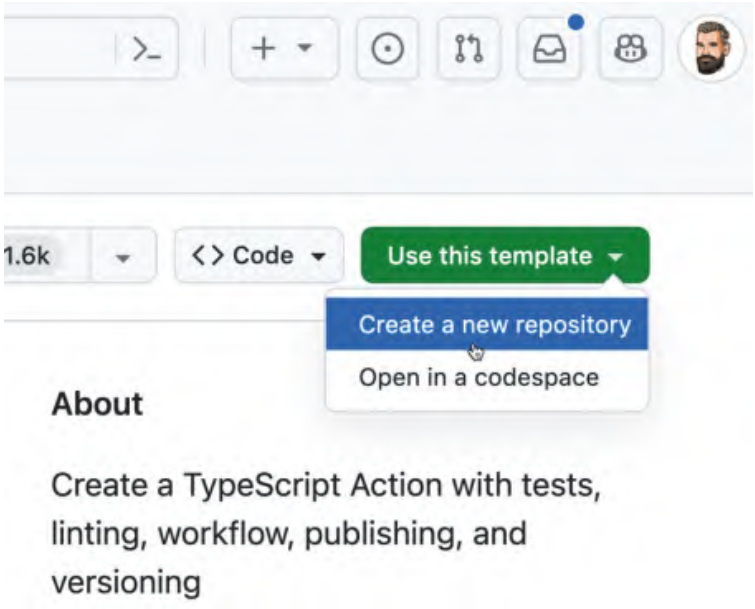


图 3.6 – 使用 typescript-action 模板创建新库

2. 选择您的 GitHub 帐户作为所有者，将其命名为 TypeScriptActionRecipe，将其可见性保留为 Public(公开)，然后单击 **Create repository**(创建库)。
3. 本地克隆该库，并在 VS Code 中打开该文件夹。

3.4.2 How to do it…

1. 打开终端，并转到库的根目录。安装所有必要的依赖项:

```
$ npm install
```

该库包含一些单元测试，运行它们以检查一切是否正常:

```
$ npm test
```

2. 打开 action.yml 文件，并更新 name(名称)、description(描述) 和 author(作者)的元数据信息:

```
name: 'TypeScript Action Recipe'
description: 'Waites for some milliseconds, writes an awesome job summary to the workflow
↪  output and returns the current date and time.'
author: 'Michael Kaufmann'
```

暂时忽略品牌信息 (branding) – 但请注意，该操作具有定义的输入参数(milliseconds)和输出参数(time):

```
# Define your inputs here.
inputs:
  milliseconds:
    description: 'Your input description here'
    required: true
    default: '1000'

# Define your outputs here.
outputs:
  time:
    description: 'Your output description here'
```

3. 打开 src/main.ts 文件并找到 run 函数：

```
export async function run(): Promise<void> {
```

这里使用了工具包（@actions/core；参见 https://github.com/actions/toolkit）来读取输入参数：

```
const ms: string = core.getInput('milliseconds')
```

还使用工具包来写入调试消息。这类似于我们在第 2 章中使用的 echo "::debug::{debug message}" 工作流命令：

```
core.debug(`Waiting ${ms} milliseconds ...`)
```

对于设置输出参数也是如此。这相当于写入 GITHUB_OUTPUT 环境文件：

```
echo "answer=42" >> $GITHUB_OUTPUT
```

使用工具包的代码为：

```
core.setOutput('time', new Date().toTimeString())
```

4. 下一步是在 run 函数中的 core.setOutput 下面编写作业摘要。从 core.summary 开始，并注意可以使用自动完成功能，来了解所有可用的功能和语法。添加一个 <h2> 标题：

```
// Write an advanced job summary
core.summary
  .addHeading('Advanced Job Summary', 'h2')
```

core.summary 对象具有流式接口（fluent interface），所以可以直接将新方法链接到前一个方法的输出中。接下来，添加一张图片并将其大小设置为 64x64：

```
.addImage(
  'https://octodex.github.com/images/droidtocat.png',
  'Droidtocat',
    {
    width: '64',
    height: '64'
    }
  )
```

添加一个带有一些数据的表格：

```
 .addTable([
   [
     { data: 'File', header: true },
     { data: 'Result', header: true }
   ],
   ['foo.js', 'Pass ☑'],
   ['bar.js', 'Fail ❌'],
   ['test.js', 'Pass ☑']
 ])
```

以及一个简单的链接：

```
.addLink('My custom link', 'https://writeabout.net'
```

需要使用 `write` 函数来完成摘要，以将缓冲区写入环境文件：

```
.write()
```

可以在此处查看代码：`https://github.com/wulfland/TypeScriptActionRecipe`。

5. 打包 TypeScript 以便分发。这是一个重要步骤，每次修改`.ts` 文件时都必须执行！

```
$ npm run bundle
```

6. 提交更改。这将自动触发一些工作流程——行时，可以使用这些时间来查看它们在做什么。`.github/workflows/linter.yml` 在对 `main` 的所有推送和拉取请求上运行，其会对您的代码库进行检查：

```
- name: Lint Code Base
  id: super-linter
  uses: super-linter/super-linter/slim@v5
  env:
    DEFAULT_BRANCH: main
    GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
    TYPESCRIPT_DEFAULT_STYLE: prettier
    VALIDATE_JSCPD: false
```

如果这失败了，可能忘记在提交更改之前运行 npm run bundle，还可以在本地运行 prettier 以确保已遵守了检查标准：

```
$ npx prettier . --check
$ npx prettier . --write
```

.github/workflows/codeql-analysis.yml 工作流也将在每次推送和对 main 的拉取请求上运行——但也会每周运行一次，将扫描代码以查找安全漏洞。

Check dist(.github/workflows/check-dist.yml) 是一个简单的工作流，将运行 npm run bundle 并将输出与您的 dist 文件夹进行比较。如果忘记使用小型简单脚本来打包您的 TypeScript，它将失败：

```
  - name: Compare Expected and Actual Directories
    id: diff
    run: |
      if [ "$(git diff --ignore-space-at-eol --text dist/ | wc -l)" -gt "0" ]; then
        echo "Detected uncommitted changes after build. See status below:"
        git diff --ignore-space-at-eol --text dist/
        exit 1
      fi
```

后一个是在 CI 构建 (.github/workflows/ci.yml)。它有两个作业：一个安装所有依赖项并运行单元测试，而另一个执行您的操作并使用输出，就像第 2 章中所做的那样：

```
- name: Test Local Action
  id: test-action
  uses: ./
  with:
    milliseconds: 1000

- name: Print Output
  id: output
  run: echo "${{ steps.test-action.outputs.time }}"
```

test-typescript 作业将失败，因为没有调整单元测试，但第二个作业应该成功。此时，可以检查工作摘要，其应该看起来像图 3.6 中所示的那样：

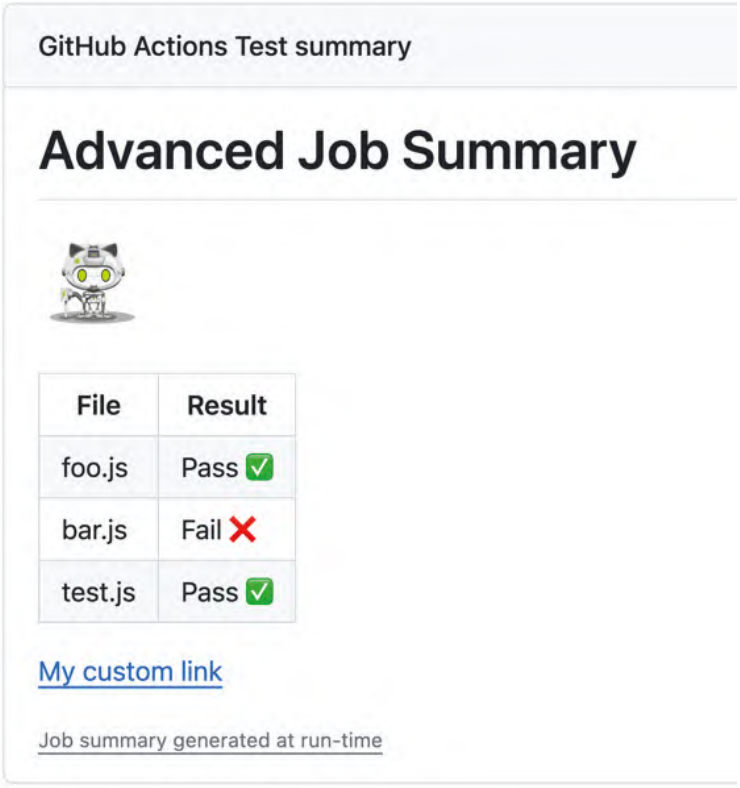


图 3.7 – 使用工具包创建的工作摘要

同时，在工作流程日志中检查输出参数的值（见图 3.8）：

图 3.8 – 工作流程日志中 TypeScript 操作的输出

修复单元测试

我没有在这本书中包含调整单元测试的示例，这本书是关于 GitHub Actions 而非 TypeScript。但想要修复测试，可以查看：https://github.com/wulfland/TypeScriptActionRecipe/blob/main/__tests__/main.test.ts。

7. 回到 main.ts 的末尾，如果遇到错误，操作将会失败。其通过使用 core.setFailed，而非返回非零值来实现：

```
} catch (error) {
  // Fail the workflow run if an error occurs
  if (error instanceof Error) core.setFailed(error.message)
```

```
}
```

8. 创建一个名为 fail-ci-build 的新分支并切换：

```
$ git switch -c fail-ci-build
```

在包含 core.setFailed 的行之前，向 catch 块中添加以下代码块：

```
core.error('Something bad happened', {
    title: 'Bad Error',
    file: '.github/workflows/ci.yml',
    startLine: 59,
    startColumn: 11,
    endColumn: 23
  })
```

在.github/workflows/ci.yml 中，修改 milliseconds 参数为无法转换为整数的值：

```
- name: Test Local Action
  id: test-action
  uses: ./
  with:
    milliseconds: xxx
```

打包 TypeScript，提交更改，并创建一个拉取请求：

```
$ npm run bundle
$ git add .
$ git commit -m "Fail CI build"
$ git push -set-upstream origin fail-ci-build
$ gh pr create --fill
```

9. 检查创建的拉取请求，并注意日志中来自 core.setFailed 和 core.error 的输出（见图 3.9）：

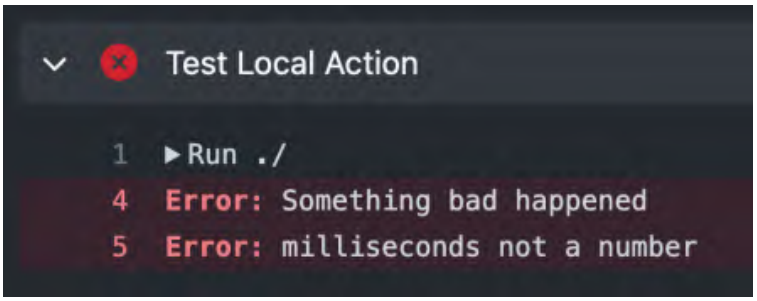


图 3.9 – 工作流程日志中失败操作的输出

同时，请注意注释在工作流程文件中显示（见图 3.10）：

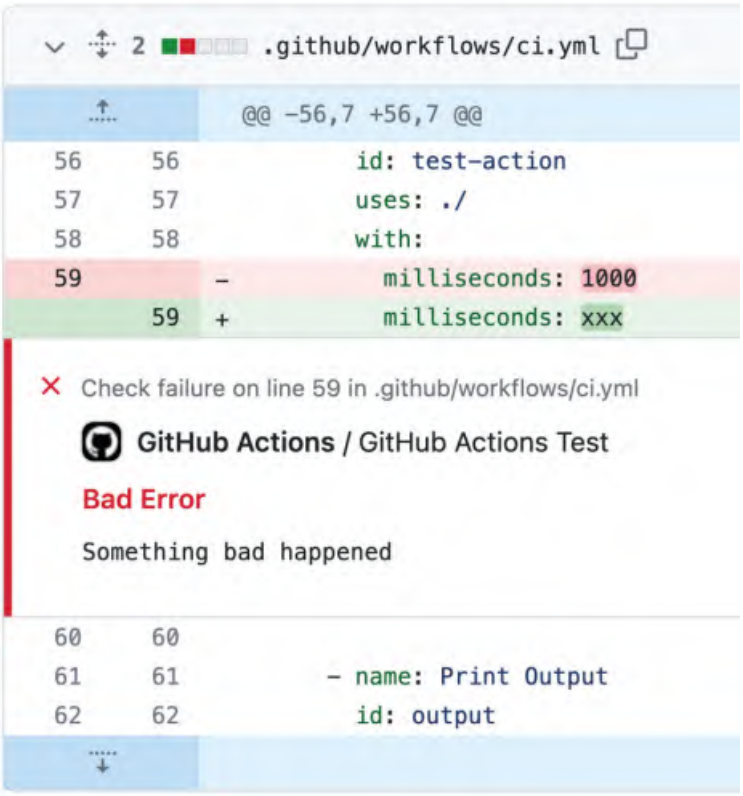


图 3.10 – 来自工具包的注释

围绕 TypeScript 和工具包的工具使得开发 GitHub Actions 变得更加便捷，并且在编写高质量的动作时提供了巨大的帮助

3.4.3 How it works…

TypeScript 操作与容器操作没有太大区别 – 它们只是在 Node.js 环境中运行，而非 Docker 容器中。TypeScript 只是在 JavaScript 之上的一层，如果运行 npm run bundle，会转译成 JavaScript（进入 /dist 文件夹）。如果你是 TypeScript 新手，所有这些工具可能看起来令人不知所措——但这些工具也使得入门变得相当容易。VS Code 中的自动完成和 IntelliSense、自动检查和格式化、带有模拟的单元测试——有很多工具有助于编写良好的代码。

3.4.4 There’s more…

我们只是接触了工具包（https://github.com/actions/toolkit），并了解了其一些功能。它还可以在使用 GitHub REST 或 GraphQL API、OICD 令牌等功能时给予我们帮助。如果想要在 GitHub Actions 中做更多事情，了解一点 TypeScript 很有必要，从而能更好借助工具包的力量。

3.5. 创建复合操作

除了 Docker 容器操作和 JavaScript/TypeScript 操作之外，第三种类型的操作是复合操作。复合操作是其他操作的包装器。这个示例中，将创建一个简单的复合操作，并在工作流程中进行使用——一次使用 bash 脚本，一次使用 GitHub 脚本。

3.5.1 Getting ready

创建一个名为 CompositeActionRecipe 的新库，将其设为公开，并用 README 文件初始化。在本地克隆该库，并在 VS Code 中打开，或在 GitHub Codespaces 中打开。

3.5.2 How to do it…

1. 在库的根目录下添加一个名为 action.yml 的新文件。添加一个名称和描述：

```
name: 'Composite Action Recipe'
description: 'Greets the user and returns 42.'
```

2. 添加一个名为 who-to-greet 的输入和一个名为 answer 的输出。注意，需要步骤 ID 来访问输出。我们将在下一步中添加：

```
inputs:
  who-to-greet:
    description: 'Who to greet'
    required: true
    default: 'World'
outputs:
  answer:
    description: "Answer to life, the universe, and everything"
    value: ${{ steps.deep-thought.outputs.answer }}
```

3. 接下来，添加 runs 部分，将 using 设置为 composite，并添加一个执行 bash 脚本的步骤。在复合操作中，必须指定 shell，且不能依赖于默认的 shell：

```
runs:
  using: "composite"
  steps:
    - name: Awesome bash script action
      id: deep-thought
      shell: bash
      run: |
        echo "Hello '${{ inputs.who-to-greet }}'."
        echo "answer=42" >> $GITHUB_OUTPUT
        echo "So long, and thanks for all the fish."
```

使用输入向工作流日志写入问候消息，并且以与在容器操作中相同的方式将输出参数设置为42。唯一的区别是，直接在工作流程运行器上执行脚本，而非在容器中。

4. 添加一个新的工作流程文件名为.github/workflows/ci.yml，并配置它在 push 和 pull_request 上运行：

```
name: CI Workflow
on: [push, pull_request]
```

添加一个作业，它检出库并带有输入参数运行复合操作。给步骤一个 ID 以便访问输出，像这样：

```
jobs:
  ci-job:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4.1.1
      - name: Run my own composite action
        id: my-action
        uses: ./
        with:
          who-to-greet: '@wulfland'
```

然后，将输出写入控制台：

```
      - name: Output the answer
        run: echo "The answer is ${{ steps.my-action.outputs.answer }}"
```

这些概念应该已经非常熟悉了。

5. 提交并推送你的更改。这将触发工作流；可以检查工作流日志，看看操作如何执行的。

3.5.3 How it works…

复合操作是步骤和其他操作的包装器，可以使用它们来捆绑多个运行命令或操作——无论自定义的，还是来自市场——并且可以为组织中的用户提供其他操作的默认值。

复合操作在 action.yml 文件的 runs 部分中有步骤——就像正常工作流中会有的那样。可以使用 inputs 上下文来访问输入参数。输出参数可以使用步骤上下文中的 outputs 上下文来访问。

3.5.4 There’s more…

复合操作直接在工作流运行器上执行，每个运行器都安装了 Node.js，所以也可以在复合操作中运行 JavaScript 和 TypeScript。我们将使用一个名为 github-script 的操作来利用工具包在复合操作中的力量，也可以直接在工作流中使用该操作。

3.6. 在复合操作中使用 github-script 为 issue 添加评论

在这个示例中，我们将使用 github-script 操作在复合操作展示工具包的强大功能。

3.6.1 How to do it…

1. 从 action.yml 文件中删除带有 bash 脚本的 run 步骤，并替换为 github-script 操作：

```
- name: Awesome github script action
  uses: actions/github-script@v6
  with:
    script: |
```

2. 使用工具包读取输入参数，向日志写入问候语，并设置输出参数：

```
var whoToGreet = core.getInput('who-to-greet')
core.notice(`Hello ${whoToGreet}`)
core.setOutput('answer', 42)
```

3. 现在，我们想要添加一些功能。如果工作流是由一个 issue 事件触发的，则想要在该 issue 添加一个评论。检查触发工作流程的事件实际上是 issue，并且使用了 github.rest.issues 来创建一个评论：

```
 if (context.eventName === 'issues') {
  github.rest.issues.createComment({
    issue_number: context.issue.number,
    owner: context.repo.owner,
    repo: context.repo.repo,
    body: '👌 Thanks for reporting!'
  })
 }
```

4. 在 CI 工作流中，添加一个触发器，当打开一个新问题时执行工作流程：

```
on:
  issues:
    types: [opened]
```

5. 提交并推送更改。
6. 在库的 **Issues | New issue**(问题 | 新问题)(issues/new) 下创建一个新 issue，给它一个标题，然后保存它。工作流将运行，并在该 issue 上添加一个评论，如图 3.11 所示：

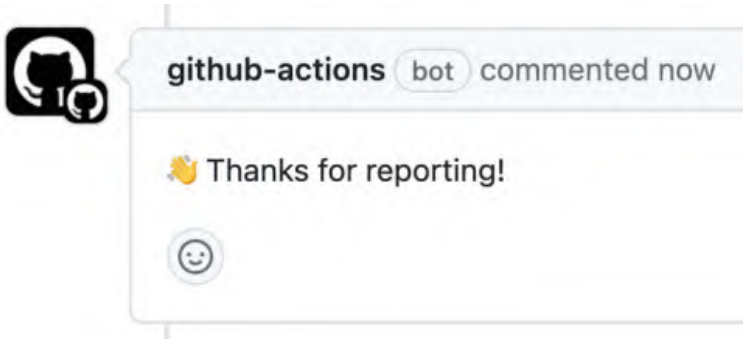


图 3.11 – 使用 github-script 操作对新 issue 添加评论

github-script 操作是使用工具包的力量快速原型设计事物的方式，而且不需要创建单独操作。

3.6.2 How it works…

github-script 操作使得在工作流程或复合操作中快速编写使用 GitHub API 和工作流程运行上下文的脚本变得容易。它使用一个名为 script 的输入，其中包含异步函数调用的主体。该操作将提供以下参数：

- github：一个预认证的 octokit/rest.js 客户端，带有分页插件
- context：一个包含工作流运行上下文的对象
- core：对 @actions/core 包的引用。
- glob：对 @actions/glob 包的引用。
- io：对 @actions/io 包的引用。
- exec：对 @actions/exec 包的引用。
- fetch：对 node-fetch 包的引用。
- require：一个围绕正常 Node.js require 的代理包装器，以启用相对于当前工作目录的相对路径，并要求 npm 包安装在当前工作目录中

由于脚本只是一个函数体，这些值将已经定义，因此不需要导入，并且可以直接像示例中使用 core、context 和 github 那样使用。

要了解更多关于 github-script 的信息，请访问 https://github.com/actions/github-script。

3.6.3 There’s more…

在 YAML 文件中的编辑和调试体验并不好，可以像这样在操作中使用脚本文件：

```
with:
  script: |
    const script = require('./path/to/script.js')
    await script({github, context, core})
```

这种方式，可以将 github-script 操作的简单性与更好的 JavaScript 编写体验结合起来。

github-script 操作是快速尝试某些事物，并创建集成概念验证的好方法。通过复合操作，可以逐渐将其放入易于共享的构建块中，也是打包可重用功能的简单方法。随着解决方案的发展，可能需要考虑将其移动到 TypeScript 或 Docker 容器操作中，以便更好地维护。

3.7. 将操作共享到市场

GitHub Actions 的强大之处在于社区——分享即是关爱。这就是为什么 GitHub 市场在赋能基于社区的流程中发挥着重要作用。在这个示例中，将为我们自定义的操作添加品牌和其他元数据，并在市场中分享。

3.7.1 Getting ready

我将使用之前创建的 Docker 容器操作，来完成这个指南——也可以使用 TypeScript 操作或复合操作，这无关紧要。只要操作位于自己的公共库中，它就可以工作。

3.7.2 How to do it…

1. 在浏览器中导航到库的根目录。GitHub 将检测到库包含一个 action.yml 文件，并且会在蓝色横幅中提议发布一个版本（见图 3.12）：

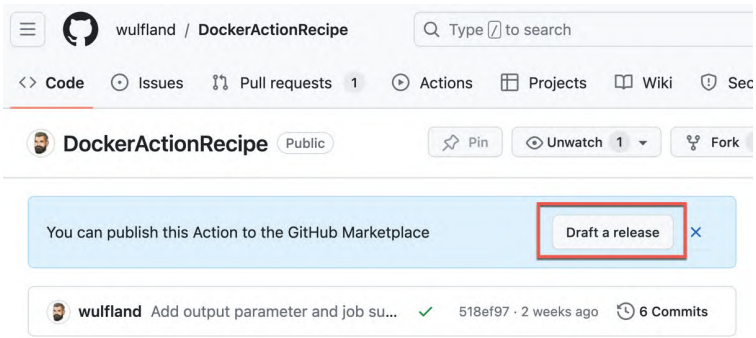


图 3.12 – 起草一个版本以将操作发布到市场

这与你转到 **Releases**(发布) 并点击 **Draft a new release**(起草新版本)(/releases/new) 相同。

2. 在对话框中，GitHub 将显示一些警告，以帮助改进市场列表（见图 3.13）：

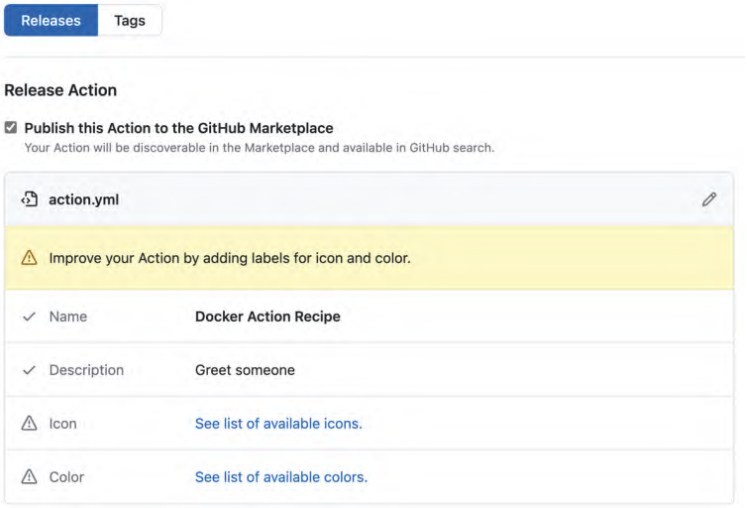


图 3.13 – 发布操作到市场的指导

点击可用图标和颜色的链接，各选一个。我选择铃铛（bell）和紫色（purple）。

3. 在 VS Code 中打开你的 action.yml 文件，并添加品牌和作者信息：

```
name: 'Docker Action Recipe'
description: 'Greet someone'
branding:
  icon: bell
  color: purple
author: 'Michael Kaufmann'
```

4. 对于市场列表来说，一个好的 README.md 文件很重要。添加一个用于输入、输出和使用示例的部分。可以参考 https://github.com/wulfland/DockerActionRecipe 获取建议。
5. 提交并推送更改。
6. 返回你的浏览器并刷新新版本窗口。现在检查应该显示没有警告，并且看起来像图 3.14 所示的样子：

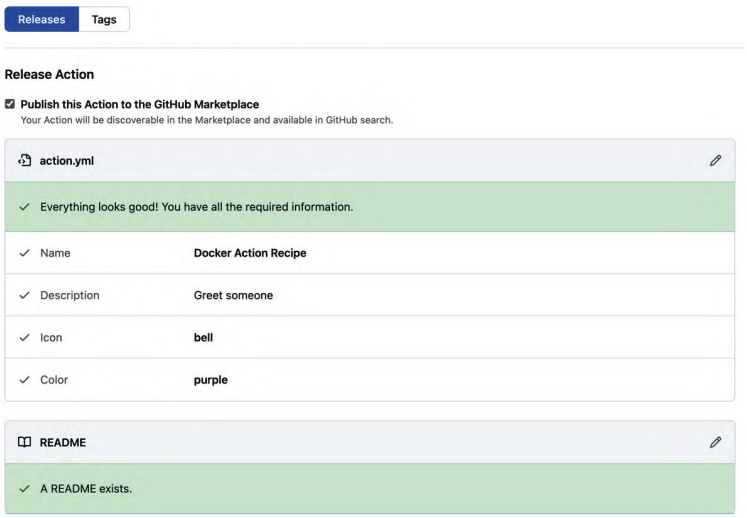


图 3.14 – 检查成功，有唯一的名称、品牌、描述和一个 README 文件

7. 点击 **Choose a tag**(选择标签)，输入 v1.0，然后点击 **Create new tag**(创建新标签)(见图 3.15):

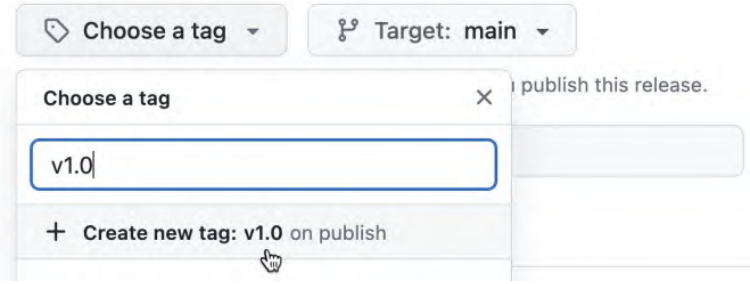


图 3.15 – 创建或选择一个标签来创建一个版本

8. 将 v1.0 添加为版本标题。请注意，可以为版本自动生成发布说明，只有在处理拉取请求时，结果才会非常好，也可以手动添加描述。
9. 点击 **Publish release**(发布版本)，将看到市场 (Marketplace) 和最新 (Latest) 标签 (见图 3.16):

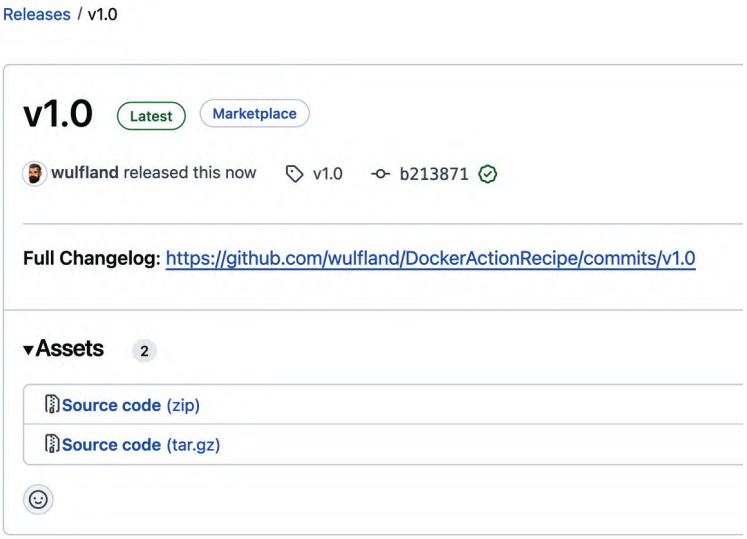


图 3.16 – 库中版本信息

10. 点击市场标签，这将转到市场列表。
11. 注意在操作名称旁边，有一个图标，颜色是在 action.yml 文件中的品牌部分指定的。库的 README.md 文件将涵盖列表的最大部分。在右侧，有一个按钮可以取消列出该操作 (从市场中移除)，一个版本选择器，以及重要链接。第一个链接将转回到你的库 (见图 3.17 列表区域):

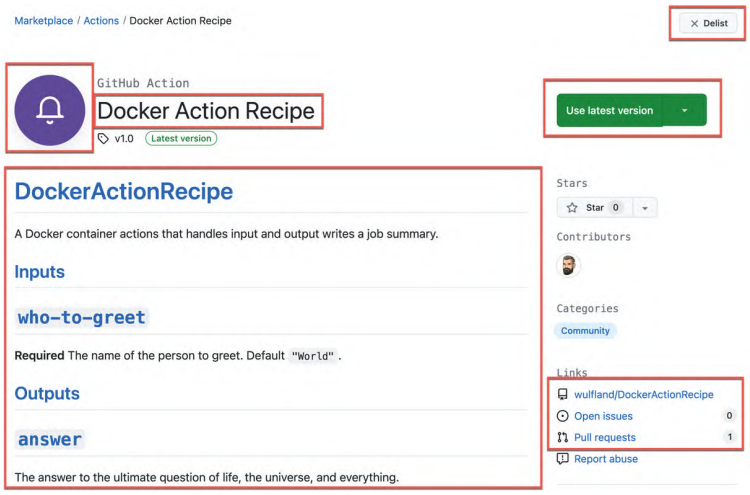


图 3.17 – 市场列表

12. 返回到库并通过重复相同的步骤创建一个名为 v1.1 的新版本。这次对话框有一个标记作为最新版本（见图 3.18）。如果忘记这一点，GitHub 将不会标记该版本为最新——无论选择的标签版本号是什么：

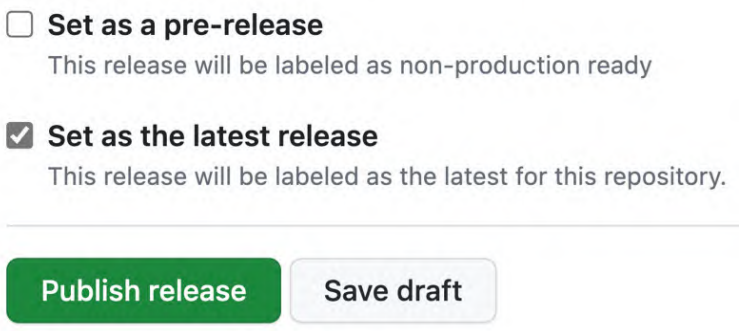


图 3.18 – 在创建或编辑版本时，必须手动设置一个版本为最新

13. 最后，在浏览器中创建一个新的工作流文件或编辑现有的工作流。在右侧，在市场窗口中输入操作的名称。操作应该立即被找到。请注意 GitHub 从版本自动创建的安装说明（见图 3.19）：

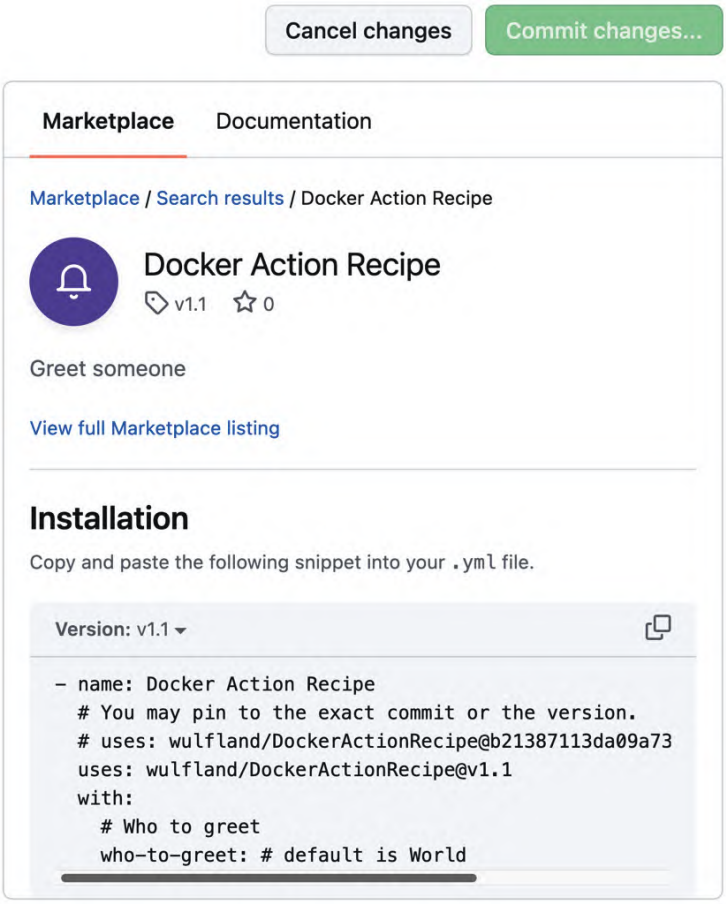


图 3.17 – 工作流编辑器中的列表

14. 如果不想在市场中保留，请取消列出操作，如图 3.17 所示。

3.7.3 How it works…

GitHub 市场是建立在 GitHub 版本之上（https://docs.github.com/en/repositories/releasing-projects-on-github），而 GitHub 版本又是建立在 git 标签之上的。标签可以是任何文本——但鼓励使用语义化版本控制来为版本赋予意义。

语义化版本控制是为软件指定版本号的正式约定，由具有不同含义的不同部分组成。语义化版本号的示例是 1.0.0 或 1.5.99-beta。格式如下：

```
<major>.<minor>.<patch>-<pre>
```

来仔细看看：

- 主版本号：一个数字标识符，如果版本不向后兼容并且有破坏性更改，则该标识符会增加。必须谨慎处理对新主版本的更新！主版本号为零表示初始开发。
- 次版本号：一个数字标识符，如果添加了新功能但版本与先前版本向后兼容，并且如果你需要新功能，可以更新而不会破坏任何东西，则该标识符会增加。
- 修订版本号：一个数字标识符，如果你发布了向后兼容的 bug 修复，则该标识符会增加。应该始终安装新的补丁。

- 预发布版本：使用连字符附加的文本标识符。标识符必须仅使用 ASCII 字母数字字符和连字符（[0-9A-Za-z-]）。文本越长，预发布版本越小（意味着 -alpha < -beta < -rc）。预发布版本始终小于正常版本（1.0.0-alpha < 1.0.0）。

在 GitHub 发布中，最佳实践是在语义化版本前缀加上 v（例如，v1.0、v1.0.1 等）。有关语义化版本的完整规范，请参见 https://semver.org/。

3.7.4 There's more…

使用标签进行版本控制会带来一些风险，对库具有写权限的人都可以修改标签，作为 GitHub 操作的维护者，鼓励在分支保护规则之外使用标签保护规则（https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/managing-repository-settings/configuring-tag-protection-rules）。例如，对 v* 的标签保护规则，将防止没有库管理员权限的人修改以 v 开头的标签。

如果想要自动化发布创建语义化版本和自动创建良好的发布说明的过程，可以使用约定式提交（Conventional Commits）（https://www.conventionalcommits.org）。约定式提交在每个提交前添加一个前缀，指示它是功能还是修复，以及它是否破坏性的。可以将此与 GitVersion（详见 https://gitversion.net/docs/）结合使用，为发布自动创建语义化版本。

第 4 章 工作流的运行时

在本章中,你将了解 GitHub Actions 的不同运行时选项。你将学习如何使用不同的 GitHub 托管的运行器和如何设置自托管的运行器。

主要内容有:

- 设置自托管运行器
- 自动扩展自托管运行器
- 使用 Actions Runner Controller (ARC) 在 Kubernetes 上扩展自托管运行器
- 运行器和运行器组
- GitHub 托管的运行器
- 设置大型运行器
- 管理和自动扩展临时运行器
- GitHub 托管和自托管运行器的安全性

4.1. 环境要求

对于本章, 需要 Docker 和 Visual Studio Code, 也可以使用 GitHub Codespaces 作为替代。对于使用 ARC 在 Kubernetes 上扩展自托管运行器的示例, 需要一个 Kubernetes 集群, 或者使用带有 Azure CLI 的 Azure 订阅来设置一个集群。对于关于运行器组的示例, 需要一个付费的 GitHub 组织的团队或企业计划。

4.2. 设置自托管运行器

到目前为止, 只使用了 ubuntu-latest 标签来运行我们的作业, 可在 GitHub 托管的最新 Ubuntu 镜像上运行工作流。但还有 macOS 和 Windows 上的运行器, 以及不同的配置, 可以在任何平台上托管自己的运行器。在第一个示例中, 将在一个 Linux Docker 容器中设置一个自托管运行器。这样, 在工作流运行之后, 将很容易扩展并清理资源。

4.2.1 Getting ready

需要为这个示例安装 Docker, 还需要知道处理器的架构。如果不知道, 只需运行 docker info 并查找"Architecture" 即可:

```
$ docker info | grep Architecture
```

4.2.2 How to do it…

1. 可以只用在上一个示例中使用的库, 或可以创建一个新库, 也可以使用在第 1 章中创建的 GitHubActionsCookbook 库。转到 **Settings | Actions | Runners**(设置 | 操

作 | 运行器)(/settings/actions/runners) 并点击 **New self-hosted**(新自托管运行器)(见图 4.1):

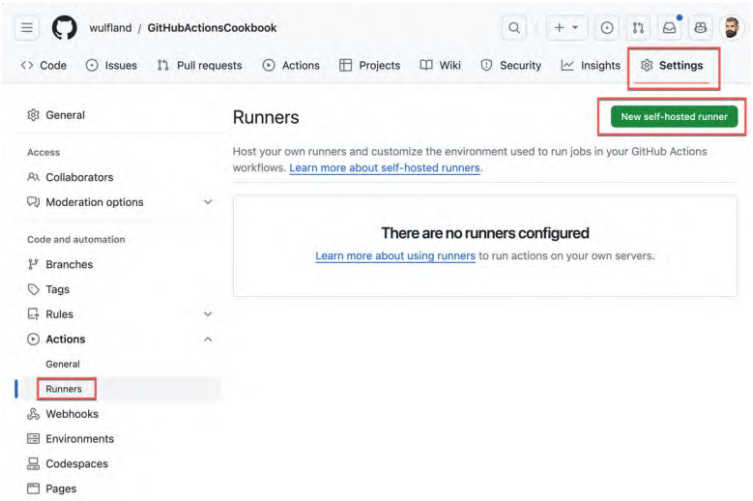


图 4.1 – 可以通过进入库的 **Settings**(设置) 区域来添加自托管运行器

这将重定向你到 /settings/actions/runners/new。为运行器镜像选择 Linux，并根据 Docker 环境设置架构属性。

注意，每个平台和处理器架构的不同脚本。可以复制整个脚本来在虚拟机上安装，因为在 Docker 容器中，所以需要增加一些步骤；也可以复制单行 (见图 4.2):

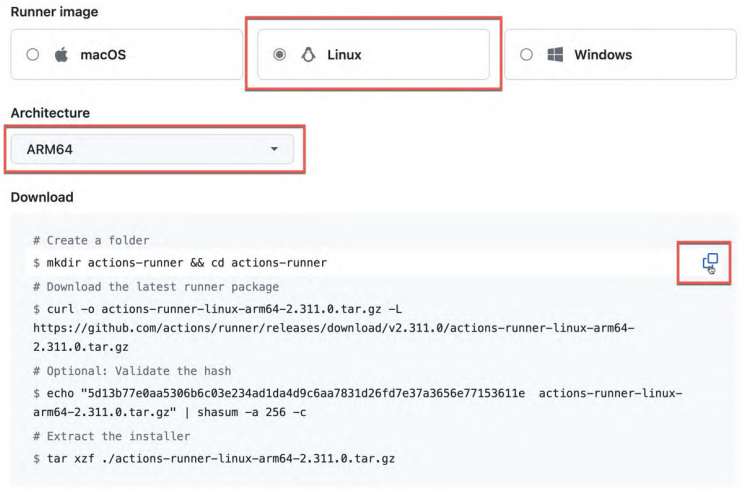


图 4.2 – 不同平台上安装自托管运行器的脚本

2. 最新版本的 Ubuntu 容器中启动一个控制台:

```
$ docker run -it ubuntu:latest /bin/bash
```

3. 运行脚本的第一行。这将创建一个运行器文件夹并切换到该目录:

```
$ mkdir actions-runner && cd actions-runner
```

4. 为了下载运行器二进制文件，必须在容器中安装 curl:

```
$ apt-get -y update; apt-get -y install curl
```

5. 现在，执行从脚本中下载最新运行器包的那一行。只需从浏览器复制并粘贴到控制台即可：

```
$ curl -o actions-runner-{version}.tar.gz -L https://{URL}.tar.gz
```

6. 使用脚本中的 `tar` 命令解压包：

```
tar xzf ./actions-runner-{version}.tar.gz
```

7. 通过执行以下脚本来安装运行器所需的所有依赖项：

```
$ ./bin/installdependencies.sh
```

8. 在配置运行器之前，必须允许它以 `root` 用户身份运行，容器默认会以 `root` 用户身份运行。我们可以通过将 `RUNNER_ALLOW_RUNASROOT` 环境变量设置为一个非零值来实现：

```
$ export RUNNER_ALLOW_RUNASROOT="1"
```

9. 现在，我们可以运行配置脚本。如果执行其他步骤花费了太长时间，可能需要在浏览器中刷新页面，令牌只在短时间内有效。复制并执行包含令牌的那一行：

```
./config.sh --url https://github.com/{OWNER}/{REPO} --token {TOKEN}
```

按回车键并接受所有默认值。

执行脚本后，可以导航回 **Settings | Actions | Runners**（设置 | 操作 | 运行器）以查看新注册的运行器。会看到它仍然离线，这是因为还没有启动运行器进程（见图 4.3）：

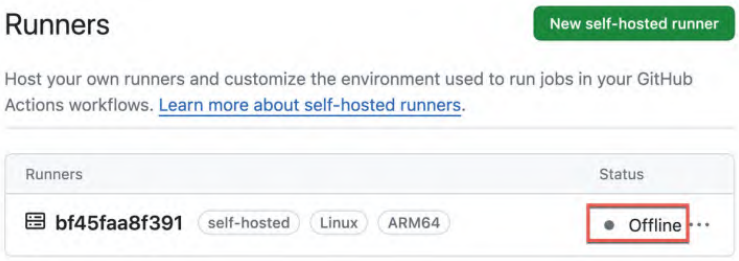


图 4.3 – 配置好的但未运行的运行器，会显示状态为离线

10. 使用以下脚本来启动运行器：

```
$ ./run.sh
```

运行器将变为空闲状态，则意味着正在等待执行工作流。

11. 创建一个使用自托管标签的简单工作流程。`bash` 应该在所有平台上都可用，所以可以省略为 `Linux` 添加的标签，但也可以通过运行 `[self-hosted, Linux]` 进行实现：

```
name: Self-Hosted

on: [workflow_dispatch]

jobs:
  main:
    runs-on: self-hosted
    steps:
      - name: Output environment
      shell: bash
      run: |-
        echo "Runner Name: '${{ runner.name }}'"
        echo "Runner OS: '${{ runner.os }}'"
        echo "Runner ARCH: '${{ runner.arch }}'"
```

12. 执行工作流程并监控 Docker 容器，以查看它如何执行工作流程，可以根据需要重复此步骤。只要容器正在运行，就会执行所有带有匹配标签的工作流。
13. 如果现在终止容器，运行器将保持在 GitHub 上离线状态。想要删除，请导航回 **Settings | Actions | Runners**（设置 | 操作 | 运行器），并从运行器右侧的菜单中选择 **Remove runner**（删除运行器）（见图 4.4）：

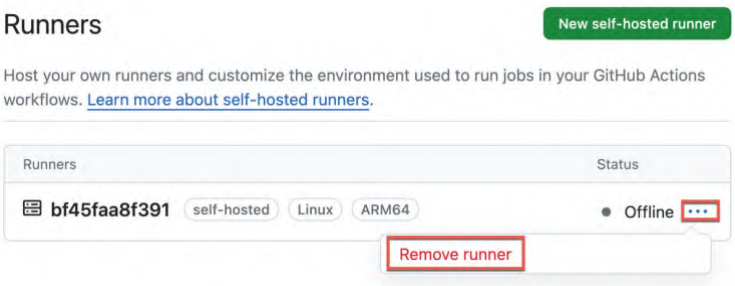


图 4.4 – 从 GitHub 中移除运行器

运行对话框提供的脚本来删除运行器：

```
$ ./config.sh remove --token {TOKEN}
```

现在，运行器将从 GitHub 中删除。

14. 还可以配置运行器，使其仅运行一个作业然后自行注销，这在容器中非常有意义。为此，可以在生成新令牌后，在配置步骤中添加 `--ephemeral` 开关：

```
./config.sh --url {URL} --token {TOKEN} --ephemeral
```

15. 再次运行工作流程，会看到运行器在执行后将自动删除。
16. 接下来，可将我们学到的所有内容放入一个 Dockerfile 中（可以使用：https://github.com/wulfland/GitHubActionsCookbook/blob/main/SelfHostedRunner/Dockerfile提供的文件）。这样，可以创建一个可重用的 Docker 镜像，将在每次运行时自动注册、等待作业、执行作业，然后终止。

我们为了简单起见，从 ubuntu:latest 继承。可以轻松地将其替换为包含所有构建工具的基础镜像：

```
FROM ubuntu:latest
```

设置将用于连接的变量。将 TOKEN 和 RUNNER_NAME 留空，这些值将在容器启动时提供，而不是在镜像创建期间，并设置正确的 URL、平台和版本：

```
ENV TOKEN=
ENV RUNNER_NAME=
ENV RUNNER_URL="https://github.com/{owner}/{repo}"
ENV GH_RUNNER_PLATFORM="linux-arm64"
ENV GH_RUNNER_VERSION="2.311.0"
ENV LABELS="self-hosted,ARM64,Linux"
ENV RUNNER_GROUP="Default"
```

安装缺失的软件包之前，必须将 DEBIAN_FRONTEND 设置为 noninteractive，以确保在 Docker 镜像构建过程中操作系统不会提示用户输入：

```
ARG DEBIAN_FRONTEND=noninteractive
```

为了在 Docker 镜像中减少层数，最好将整个脚本合并为一个 RUN 命令。该脚本更新包管理器及其所有包，安装所有依赖项，添加容器将以其下运行的 docker 用户（不希望容器以 root 用户运行），下载相应的包并解压，将所有者更改为 docker 用户并执行 installdependencies.sh 脚本：

```
RUN apt-get -y update && \
  apt-get upgrade -y && \
  useradd -m docker && \
  apt-get install -y --no-install-recommends curl ca-certificates && \
  mkdir -p /opt/hostedtoolcache /home/docker/actions-runner && \
  curl -L https://github.com/actions/runner/releases/download/v${GH_RUNNER_VERSION}/
  actions-runner-${GH_RUNNER_PLATFORM}-${GH_RUNNER_VERSION}.tar.gz  -o
  ↪  /home/docker/actions-runner/actionsrunner.tar.gz && tar xzf
  ↪  /home/docker/actions-runner/actions-runner.tar.gz -C /home/docker/actions-runner &&
  ↪  chown -R docker /home/docker &&
  ↪  /home/docker/actions-runner/bin/installdependencies.sh
```

以 docker 用户身份运行容器，并将工作目录设置为该用户主目录下的 actions-runner：

```
USER docker
WORKDIR /home/docker/actions-runner
```

如果容器将要运行，必须检查是否提供了 TOKEN 和 RUNNER_NAME。然后，使用所有参数（包括 --ephemeral）运行配置脚本，然后运行 run.sh 脚本：

```
CMD if [ -z "$TOKEN" ]; then echo 'TOKEN is not set'; exit 1; fi
&& \
  if [ -z "$RUNNER_NAME" ]; then echo 'RUNNER_NAME is not set';
exit 1; fi && \
  ./config.sh --url "${RUNNER_URL}" --token "${TOKEN}"
--name "${RUNNER_NAME}" --work "_work" --labels "${LABELS}"
--runnergroup "${RUNNER_GROUP}" --unattended --ephemeral && \
  ./run.sh
```

17. 进入包含 Dockerfile 的文件夹，并根据该文件创建一个 Docker 镜像：

```
$ docker build -t simple-ubuntu-runner .
```

18. 现在，使用 docker run 运行镜像的任意多个实例。-d（--detached）选项将在后台以分离模式运行容器，这不会阻塞控制台，但也不会在终端接收输入或显示输出。--rm 选项将在容器退出时删除它。使用 -e 选项传递 RUNNER_NAME 和 TOKEN 的参数。需要注意的时，这些名称区分大小写！

```
$ docker run -d --rm -e RUNNER_NAME=Runner1 -e TOKEN={TOKEN} simple-ubuntu-runner
```

能在库 **Settings**（设置）区域看到运行器，如图 4.5 所示：

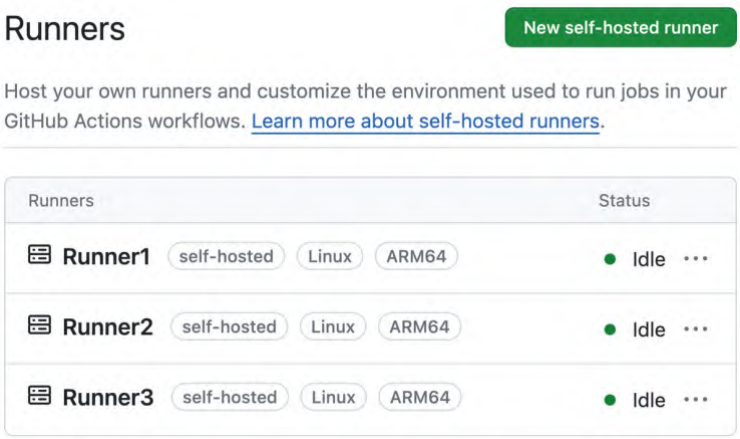


图 4.5 – 通过多次运行相同的 Docker 镜像来获得临时运行器，这些运行器将等待传入的任务

根据创建的容器数量，多次启动工作流程，并在执行后清理容器。

4.2.3 How it works…

来了解一下流程的工作原理。

自托管运行器应用程序

自托管运行器是通过安装开源的运行器应用程序（https://github.com/actions/runner）创建， 创建的。该应用程序基于.NET Core 运行时，可以在许多操作系统和处理器架构上运

行。它可以在 macOS 11(Big Sur) 或更高版本、Windows(7 到 10 和 Server 2012 R2 到 20222)、以及许多 Linux 发行版 (Red Hat Enterprise 7 或更高版本、Fedora 29 或更高版本、Ubuntu 16.04 或更高版本，等等) 上运行。它还可以在 x64、ARM64 和 ARM32 上运行。同时，它还可以在 x64、ARM64 和 ARM32 上运行。有关支持的操作系统的最新列表，请参阅 https://docs.github.com/en/actions/hosting-your-own-runners/managing-self-hosted-runners/about-self-hosted-runners#supported-architectures-and-operating-systems-for-self-hosted-runners。可以使用运行器应用程序文件夹中的 bin/installdependencies.sh 脚本来安装.NET Core 运行时所需的所有库。

如果要运行基于 Docker 的操作，必须使用 Linux 镜像。Windows 和 macOS 不支持运行基于 Docker 的操作!

GitHub 的身份验证

使用通过 GitHub UI 生成配置令牌来连接运行器到 GitHub。该令牌仅有效 1 小时，并且只能使用该令牌来安装运行器。也可以通过向 https://api.github.com/repos/OWNER/REPO/actions/runners/registration-token(或对于组织级别的运行器,https://api.github.com/orgs/ORG/actions/runners/registration-to 发送 POST 请求，按需通过 REST API 创建安装令牌。

以下是使用个人访问令牌 (PAT，personal access token) 接收令牌的示例:

```
$ curl -L \
> -X POST \
> -H "Accept: application/vnd.github+json" \
> -H "Authorization: Bearer <YOUR-PAT>" \
> -H "X-GitHub-Api-Version: 2022-11-28" \
> https://api.github.com/repos/{OWNER}/{REPO}/actions/runners/registration-token
```

结果还包含到期日期。如果想在变量中使用令牌，可以将结果管道传递给 jq:

```
$ TOKEN=$(<curl command> | jq .token --raw-output)
```

必须使用具有 repo 范围的 PAT 访问令牌来使用此端点，GitHub Apps 必须具有库的管理权限和组织的 organization_self_hosted_runners 权限。经过身份验证的用户必须具有对库或组织的管理访问权限，或企业的 manage_runners:enterprise 范围。该令牌仅用于注册。

在注册过程中，将从服务器接收到一个 JWT(OAuth 交换的 JSON Web 令牌)，只有权限监听队列。当工作流程运行开始时，将为构建的生命周期创建另一个预构建的、有限范围 (由工作流程定义) 的令牌。该令牌无法通过临时脚本或不可靠的代码访问 - 只能由构建代理和任务访问。OAuth 令牌交换之间代理和服务器之间的 RSA 私钥将存储在一个名为.credentials_rsaparams 的文件中，服务器持有公钥。每 50 分钟，服务器将向代理发送一个由公钥加密的新令牌。OAuth 配置存储在.credentials 文件中:

```
{
  "scheme": "OAuth",
  "data": {
    "clientId": "{CLIENT_ID}",
    "authorizationUrl":
    ↪  "https://pipelinesghubeus4.actions.githubusercontent.com/{TOKEN}/_apis/oauth2/token",
    "requireFipsCryptography": "True"
  }
```

运行应用程序作为服务

在 Windows 上，配置脚本会询问你是否要作为服务执行运行器，以便它与环境一起启动。在 Linux 上，需要使用 svc.sh 脚本自己配置服务：

```
sudo ./svc.sh install
sudo ./svc.sh start
```

网络通信

运行器应用程序使用出站 HTTPS 连接通过端口 443 与 GitHub 通信，使用具有 50 秒超时的长轮询。所以应用程序询问 GitHub 是否有工作排队等待运行器的标签，然后在连接关闭之前等待 50 秒的响应。连接关闭后，立即启动一个新连接。无需 GitHub 的入站连接或打开防火墙端口，仅在端口 443 上使用 SSL 的安全出站连接。

更新自托管运行器

自托管运行器将自动检查是否有可用的新版本运行器应用程序，并更新它。GitHub 只会更新运行器本身 - 机器的其余部分由客户管理。

清理

值得注意的是，GitHub 运行器应用程序不会在 workflow 运行后清理资源。这种行为与 GitHub 托管的运行器不同，它们为每个 workflow 运行提供临时干净的环境。如果下载库并执行构建，所有文件将保留在那里。如果想要为多个 workflow 运行使用运行器应用程序，则必须自己清理所有内容。这样，总是有一个干净的环境。

可以使用 workflow 逻辑，在库的 workflow 运行后进行清理 - 也可以使用前工作或后工作脚本来在运行器上执行此操作。要配置前工作或后工作脚本，需要将一个脚本文件保存在运行器可以访问的位置，然后使用以下名称之一和脚本的完整路径作为值配置一个环境变量：

- ACTIONS_RUNNER_HOOK_JOB_STARTED
- ACTIONS_RUNNER_HOOK_JOB_COMPLETED

或者，你可以将键值对存储在运行器应用程序目录内的.env 文件中。

4.2.4 There’s more…

在 macOS 上安装运行器与在 Linux 上安装相同。Windows 上的不同之处在于，脚本是一个 PowerShell 脚本，而不是 bash 脚本。例如，使用 Invoke-WebRequest 而不是 curl，

但所有步骤都是相同的。用于配置和启动运行器的脚本的扩展名为.cmd，而不是.sh：

```
$ ./config.cmd --url <URL> --token <TOKEN>
$ ./run.cmd
```

如果已经在 Linux 容器中成功安装了运行器，则在 Windows 上安装也不会有问题。

4.3. 自动扩展自托管运行器

在这个示例中，我们将在上一个指南的基础上进行构建，以便有一个解决方案，每次触发新工作流程时，都会自动启动一个新的临时 Docker 容器实例。这次，将使用 GitHub Webhook 进行实现。

4.3.1 Getting ready

确保机器或 GitHub Codespaces 上仍然有上一个示例创建的 simple-ubuntu-runner Docker 镜像。

4.3.2 How to do it…

1. 前往 https://github.com/settings/apps。点击 **New GitHub App**(新建 GitHub 应用)。
2. 将 **GitHub App Name** 设置为 auto-scale-runners，并将 **Homepage URL** 设置为正在使用库的 URL(参见图 4.6)：

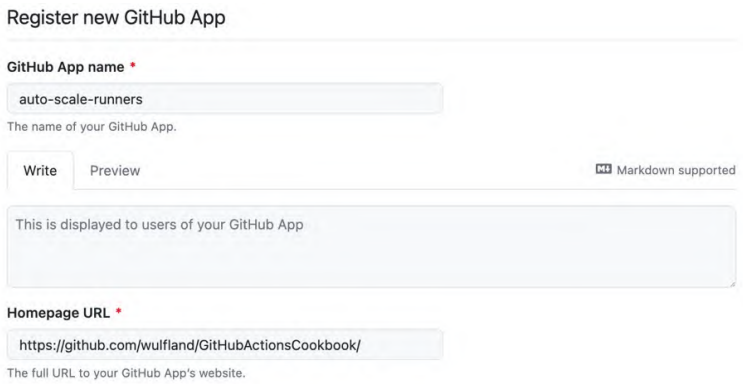


图 4.6 – 设置新应用程序的名称和 URL

3. 跳过 **Identifying and authorizing users**(识别和授权用户) 以及 **Post installation**(安装后) 部分，直接进入 Webhook 配置。
4. 打开另一个浏览器标签页，访问 https://smee.io，然后点击 **Start new channel**(启动新频道)，再复制生成的 **Webhook Proxy URL** 值。
5. 返回到创建 GitHub 应用的标签页，将复制的 **Webhook Proxy URL** 粘贴到 **Webhook URL** 字段中。在 Webhook secret 字段中设置一个能记住的字符串 (参见图 4.7)。

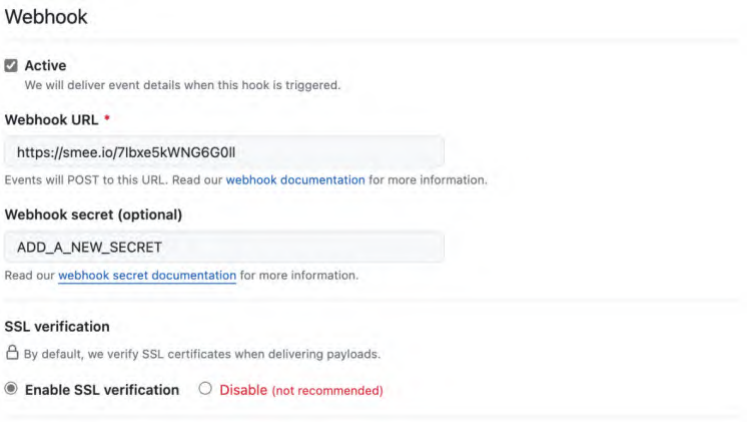


图 4.7 – 配置 Webhook

6. 在 **Permissions**(权限) 下的 **Repository permissions**(库权限) 中,将 **Actions**(操作) 设置为 **Read-only**(只读), 并将 **Administration**(管理) 设置为 **Read and write**(读写)(参见图 4.8)。

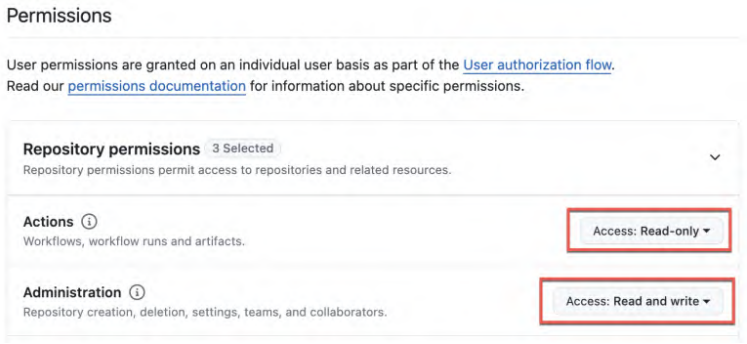


图 4.8 – 为应用程序配置库权限

7. 在 **Subscribe to events**(订阅事件) 部分, 勾选 **Workflow job**(工作流作业)(参见图 4.9)。

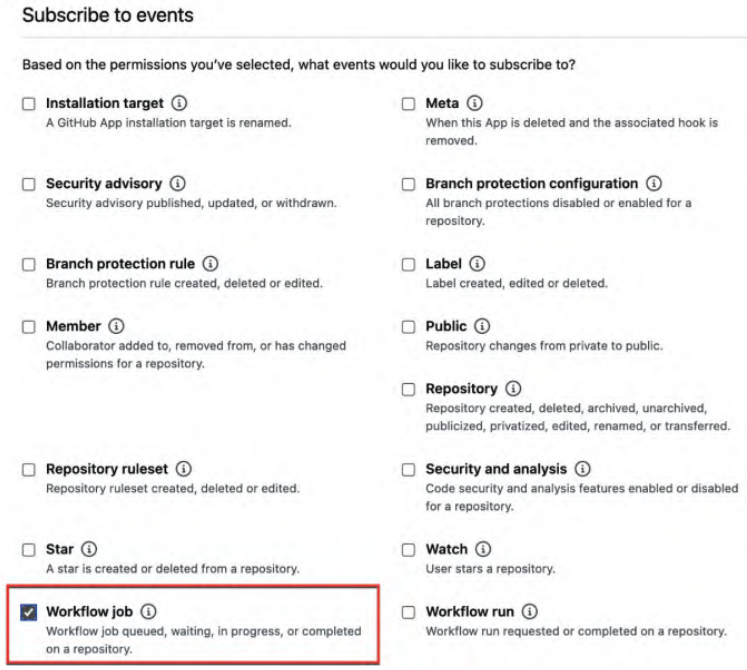


图 4.9 – 订阅工作流作业 webhook

8. 在 **Where can this GitHub App be installed**(此 GitHub 应用可以安装在哪里)下，选择 **Only on this account**(仅在此账户上)，然后点击 **Create GitHub App**(创建 GitHub 应用)。
9. 在您新创建的应用中，点击 **Generate a private key**(生成私钥)。私钥将会自动下载，将其移动到库中。
10. 从应用的 **General**(通用) 标签页中复制 **App ID** 值 (参见图 4.10)。

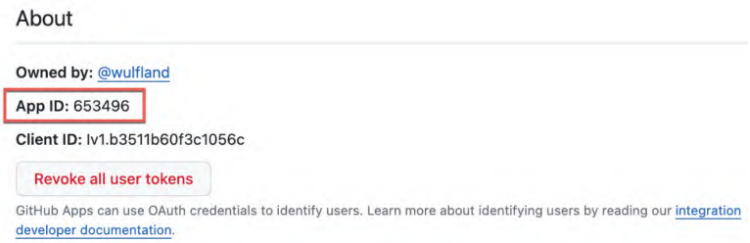


图 4.10 – 获取应用程序 ID 值

11. 在库中，创建一个名为.env 的新文件，并添加具有相应值的 APP_ID、WEBHOOK_SECRET 和 PRIVATE_KEY_PATH 变量：

```
APP_ID="653496"
WEBHOOK_SECRET="YOUR_SECRET"
PRIVATE_KEY_PATH="auto-scale-runners.2023-11-26.private-key.pem"
```

后将在我们的应用程序中，会使用这些环境变量来对 GitHub 进行身份验证。

12. 将.env 文件添加到.gitignore 文件中，以避免意外提交：

```
$ echo ".env" >> .gitignore
```

13. 在 github 的应用中，选择“安装应用”，然后单击安装 (请参见图 4.11)：

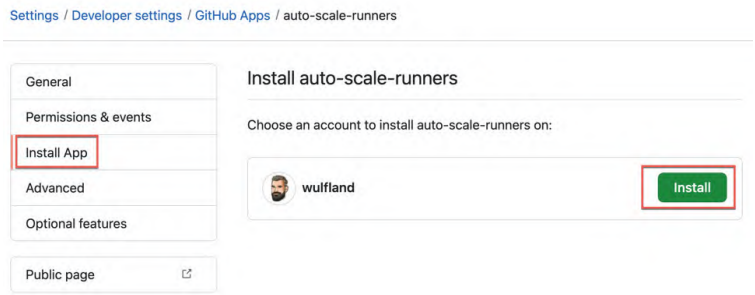


图 4.11 – 安装应用程序

14. 在出现的对话框中选择库，然后单击 **Install**(安装)(请参见图 4.12)：

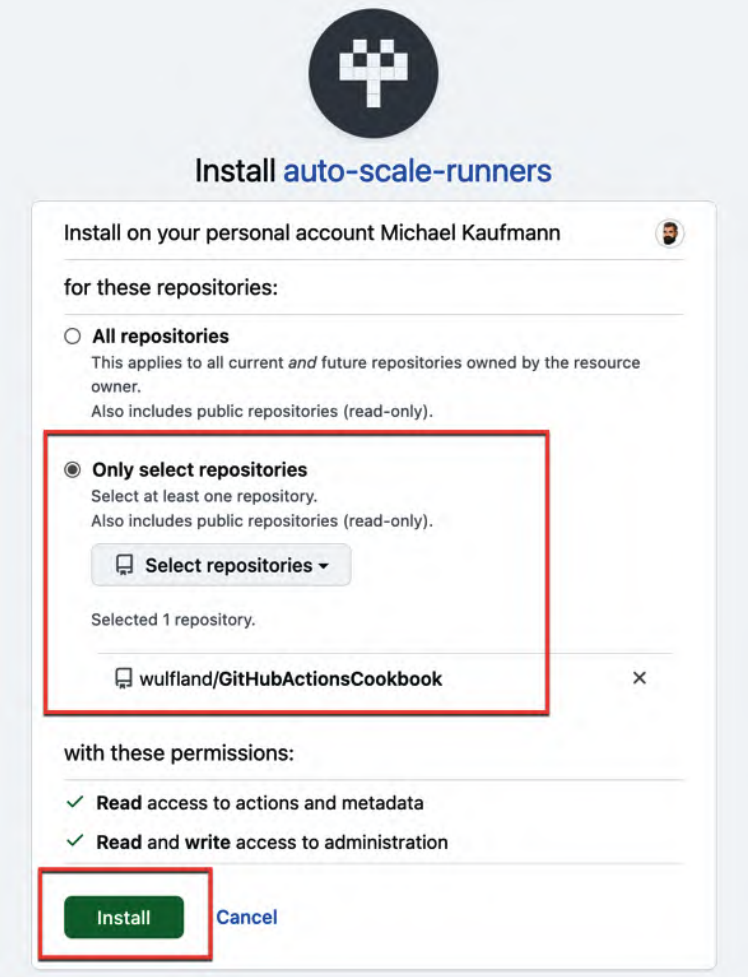


图 4.12 – 在库中安装应用程序

接下来，将创建一个服务器，当它从 webhook 接收到有效负载时，该服务器将运行，转到库并初始化：

```
$ npm init -yes
```

添加代码中使用的依赖项：

```
$ npm install octokit
$ npm install dotenv
$ npm install smee-client --save-dev
```

将 node_modules 文件夹添加到.gitignore：

```
$ echo "node_modules" >> .gitignore
```

15. 创建一个名为 app.js 的新文件，可以从这里复制内容：https://github.com/wulfland/GitHubActionsCookbook/blob/main/SelfHostedRunner/auto-scale/app.js。
16. 添加必要的依赖项

```
import dotenv from "dotenv";
import {App} from "octokit";
import {createNodeMiddleware} from "@octokit/webhooks";
import fs from "fs";
import http from "http";
import { exec } from 'child_process';
```

17. 接下来，使用 dotenv 包从之前创建的.env 文件中读取环境变量：

```
dotenv.config();
const appId = process.env.APP_ID;
const webhookSecret = process.env.WEBHOOK_SECRET;
const privateKeyPath = process.env.PRIVATE_KEY_PATH;
```

18. 接下来，从相应路径加载私钥：

```
const privateKey = fs.readFileSync(privateKeyPath, "utf8");
```

19. 创建 octokit 的 app 类的一个新实例：

```
const app = new App({
  appId: appId,
  privateKey: privateKey,
  webhooks: {
    secret: webhookSecret
  },
});
```

然后，为 workflow_job.queued 事件注册一个事件处理程序：

```
app.webhooks.on("workflow_job.queued", handleNewQueuedJobsRequestOpened);
```

20. 调用 GitHub API 以接收一个可以注册运行器的新令牌。我们需要这个令牌，以便可以将其传递给容器。必须使用 workflow_job.id 为运行器创建名称：

```
const response = await octokit.request('POST
↪  /repos/{owner}/{repo}/actions/runners/registration-token', { owner:
↪  payload.repository.owner.login,
  repo: payload.repository.name,
  headers: {
    'X-GitHub-Api-Version': '2022-11-28'
  }
});

const token = response.data.token;
const runner_name = `Runner_${payload.workflow_job.id}`;
```

21. 然后，创建一个新的 Docker 容器实例，并传入令牌和名称：

```
exec(`docker run -d --rm -e RUNNER_NAME=${runner_name} -e
  TOKEN=${token} simple-ubuntu-runner`, (error, stdout, stderr) =>
  {
  if (error) {
    console.error(`exec error: ${error}`);
    return;
  }
  console.log(`stdout: ${stdout}`);
  console.error(`stderr: ${stderr}`);
});
```

我在这里跳过了错误处理部分，因为它与此无关。

22. 在文件的最后一部分，必须创建一个在端口 3000 上监听的开发服务器：

```
const port = 3000;
const host = 'localhost';
const path = "/api/webhook";
const localWebhookUrl = `http://${host}:${port}${path}`;
const middleware = createNodeMiddleware(app.webhooks, {path});
http.createServer(middleware).listen(port, () => {
console.log(`Server is listening for events at: ${localWebhookUrl}`);
console.log('Press Ctrl + C to quit.')
});
```

23. 在 package.json 文件中，添加一个名为 type 的顶层条目，并将其设置为 module。然后，添加一个名为 server 的脚本来运行应用程序

```
"type": "module",
"scripts": {
  "server": "node app.js"
},
```

24. 我们准备好了！打开一个新的终端，并使用你的频道 URL（第 4 步）启动一个 smee 客户端：

```
$ npx smee -u https://smee.io/{ID} -t http://localhost:3000/api/webhook
```

在库的终端中，运行应用程序：

```
$ npm run server
```

为自托管工作流启动一个新的工作流：

```
$ gh workflow run Self-Hosted
```

注意 smee 客户端如何接收从 GitHub 转发来的 webhook，以及服务器如何处理它，并启动一个新容器来执行工作流。

4.3.3 How it works…

GitHub 应用程序提供了一个简单的方式来对 GitHub 进行身份验证并注册 webhook。在这个指南中，我们注册了 workflow_job webhook，并设置了 queued 动作类型，以便每次有新的工作流程排队时，我们都可以启动一个新的运行器。如果有更大的镜像需要更长时间来加载，也可以有一组现有的运行器，并在作业排队时仍然加载一个新的运行器。

启动的临时运行器不一定是执行你作业的那个运行器。

由于我们需要一个 GitHub 可以访问的端点，所以使用了 smee.io 作为代理，并在事件发生时转发有效载荷。这不适用于生产环境，只是提供了一个方便的方式来在本地或 GitHub Codespace 中开发，而无需公开可用的入站端口。对于生产用途，应该将应用程序托管在 Web 服务器上。

4.3.4 There’s more…

这个指南旨在提供创建自托管运行器扩展解决方案的基本构建模块。使用临时运行器和 webhook，很容易自动化这个过程。但如果需要一个更可扩展、更成熟的解决方案，应该考虑使用 Kubernetes 来实现。

4.4. 使用 ARC 通过 Kubernetes 实现自托管运行器的扩展

Kubernetes 非常强大，但也相当复杂。本示例中，我将只专注于让你开始使用 Kubernetes 扩展自托管运行器。

ARC 是一个 Kubernetes 运营商，能编排和扩展自托管运行器的工作负载。这是一个开源项目，但现在已得到 GitHub 的完全支持。

4.4.1 Getting ready

如果已经有一个 Kubernetes 集群，可以使用它。如果没有，可以通过运行以下命令在 Azure 中创建一个新集群：

```
$ az group create --name AKSCluster -l westeurope
$ az aks create --resource-group AKSCluster \
> --name AKSCluster \
> --node-count 3 \
> --enable-addons monitoring \
> --generate-ssh-keys
$ az aks get-credentials --resource-group AKSCluster --name AKSCluster
```

确保集群中安装了 cert-manager(https://cert-manager.io/docs/installation/)，可以通过运行以下命令来完成此操作，确保为最新版本：

```
$ kubectl apply -f
↪  https://github.com/cert-manager/cert-manager/releases/download/v1.13.2/cert-manager.yaml
```

可以检查 ARC 的快速入门教程中先决条件是否有变化：https://github.com/actions/actions-runner-controller/blob/master/docs/quickstart.md。

4.4.2 How to do it…

1. 将 ARC 部署到集群中，确保版本为最新版本：

```
$ kubectl apply -f
↪  https://github.com/actions/actions-runnercontroller/releases/download/v0.23.7/
actions-runner-controller.yaml
```

2. 在 GitHub 中创建一个具有 repo 范围的 PAT。转到 https://github.com/settings/tokens/new，选择 repo，设置过期日期，然后点击生成令牌，再复制该令牌。
3. 现在，将令牌保存为 Kubernetes 中的一个秘钥：

```
$ kubectl create secret generic controller-manager \
> -n actions-runner-system \
> --from-literal=github_token=<YOUR_TOKEN>
```

4. 在库中创建一个名为 runnerdeployment.yml 的文件，并包含以下内容。将库所有者和名称替换为库的值：

```
apiVersion: actions.summerwind.dev/v1alpha1
kind: RunnerDeployment
metadata:
  name: example-runnerdeploy
spec:
  replicas: 1
  template:
    spec:
      repository: wulfland/GitHubActionsCookbook
```

5. 将 RunnerDeployment 应用到集群：

```
$ kubectl apply -f runnerdeployment.yml
```

现在，应该有一个运行器和两个正在运行的 pod：

```
$ kubectl get runners
$ kubectl get pods
```

验证否可以在 GitHub 中看到运行器（settings/actions/runners）。请注意，每次工作流程运行后，名称都会更改（见图 4.13）：

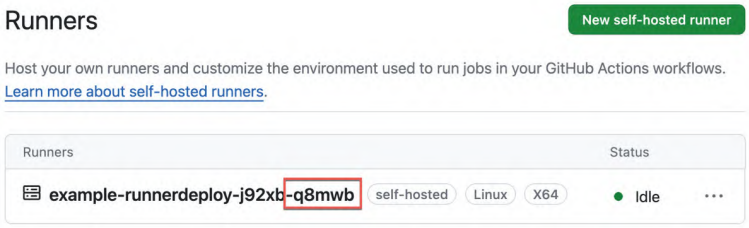


图 4.13 – GitHub 中的 ARC 运行器

从之前的配方中运行.github/workflows/self-hosted.yml 工作流程并检查输出，在 AKS 集群中执行（见图 4.14）：

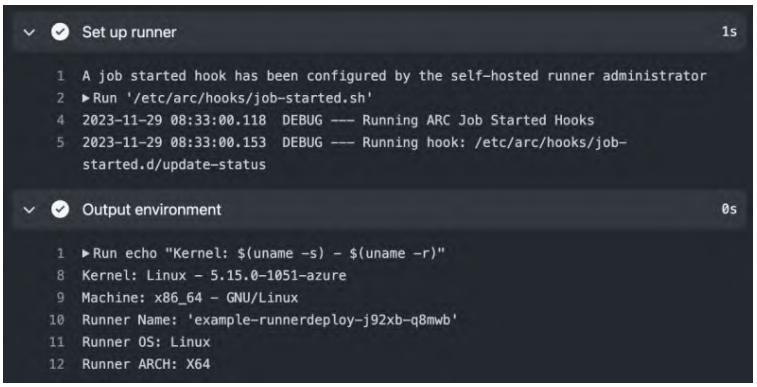


图 4.14 – 自托管工作流程在 Kubernetes 集群中由 ARC 执行

4.4.3 How it works…

ARC 运行器默认设置为临时运行，使用 Kubernetes 副本集，并在执行后自动启动一个新的容器。ARC 提供三种扩展选项：

- 定时：根据计划进行扩展和缩减
- 基于规模：执行作业的繁忙运行器的百分比
- 按需：当新的工作流程作业排队时启动实例

ARC 支持在企业、组织和库级别创建运行器。可以使用 GitHub 应用程序或 PAT 在组织和库级别进行身份验证。使用企业库时，必须使用 PAT，因为应用程序不能限定在企业范围内。

可以为不同的团队配置具有不同镜像和命名空间的扩展集，也允许限制它们之间的网络访问。

4.4.4 There’s more…

对软件供应链和构建过程的攻击构成了一个巨大的威胁。应该非常小心，特别是对于自托管运行器。永远不要在允许派生的公共库中使用，并采取措施控制在运行器上拉取的所有依赖项，以便

没有人可以执行以下操作：

- 篡改构建过程中的文件
- 逃离容器/沙盒或访问工作流程范围之外的文件
- 通过操纵依赖项缓存（缓存中毒）来损害依赖项
- 窃取数据或机密，并将数据发送到控制服务器

可以在工作流程中使用 StepSecurity 的 Harden-Runner 操作（https://github.com/stepsecurity/harden-runner），将根据策略（如出站网络出口）监控工作流：

```
steps:
  - uses: step-security/harden-runner@v2.6.1
    with:
      egress-policy: audit
```

也可以使用它来阻止流量，只允许特定的端点和端口：

```
egress-policy: block
allowed-endpoints: >
  api.nuget.org:443
  github.com:443
```

对于 ARC，不必将 Harden-Runner 操作添加到每个工作流程作业中，可以在 Kubernetes 集群上安装 ARC Harden-Runner DaemonSet。该 DaemonSet 将不断监控每个工作流程的运行，而无需将操作添加到每个工作流程中。

可以在仪表板的运行时安全选项卡下访问安全见解和运行时检测。

请注意，这不是免费软件。对于 https://github.com 上的公共库，提供免费的社区许可证；但对于私有库或 GitHub Enterprise Server（GHES），需要购买许可证（https://www.stepsecurity.io/pricing）。

4.5. 运行器与运行器组

在组织和企业级别，可按运行器组组织访问运行器。工作流与运行器的关联通过标签完成 - 但运行器组控制工作流可以访问哪些运行器。

4.5.1 Getting ready

请注意，在免费组织中，只有一个名为“默认”的运行器组，可以使用它来添加自托管运行器。要创建多个运行器组或用于 GitHub 托管运行器，需要付费的团队或企业计划。

4.5.2 How to do it…

1. 在拥有付费计划的组织中，导航到 **Settings | Actions | Runner groups**（设置 | 操作 | 运行器组）（/settings/actions/runner-groups），并点击 **New group**（新建组）。
   为该组命名。在 **Repository access**（库访问权限）下，将选择从 **All repositories**（所有库）更改为 **Selected repositories**（选定的库），并点击齿轮图标来选择一个或多个可以访问该组的库（参见图 4.15）。请注意，可以在此处允许访问公共库，但此选项默认禁用：

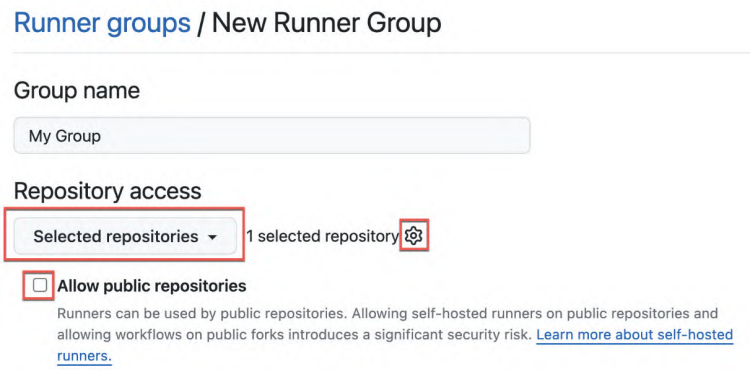


图 4.15 – 管理库对组的访问权限

2. 在 **Workflow access**（工作流访问权限）下，选择 **Selected workflows**（选定的工作流）并点击齿轮图标（参见图 4.16）：

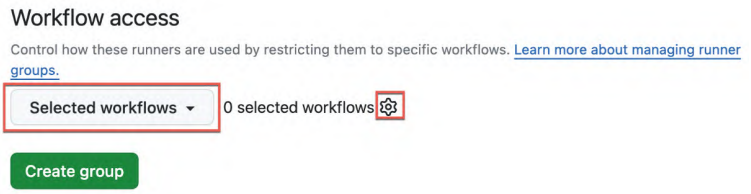


图 4.16 – 限制对某些工作流的访问

3. 请注意，在出现的对话框中，可以添加多个模式来识别工作流（见图 4.17）：

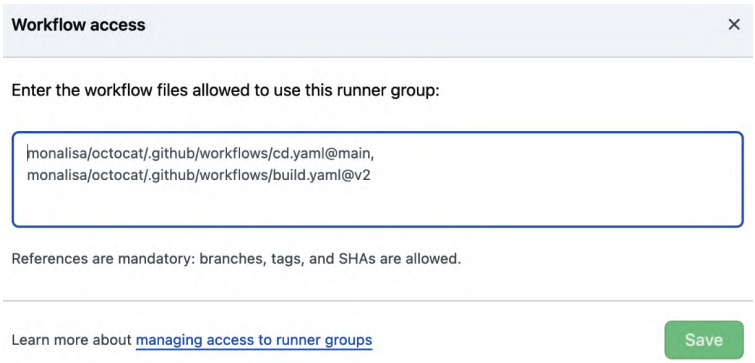


图 4.17 – 限制对工作流版本访问的语法

工作流通过指向工作流文件的路径和一个有效的 `git` 引用来指定，该引用可以是分支、标签或 SHA 值。

4. 退出对话框，将值设置回 **All workflows**（所有工作流程），然后点击 **Create group**（创建组）。现在可以向该组添加自托管运行器和 GitHub 托管运行器，点击 **New self-hosted runner**（新建自托管运行器）（见图 4.18）

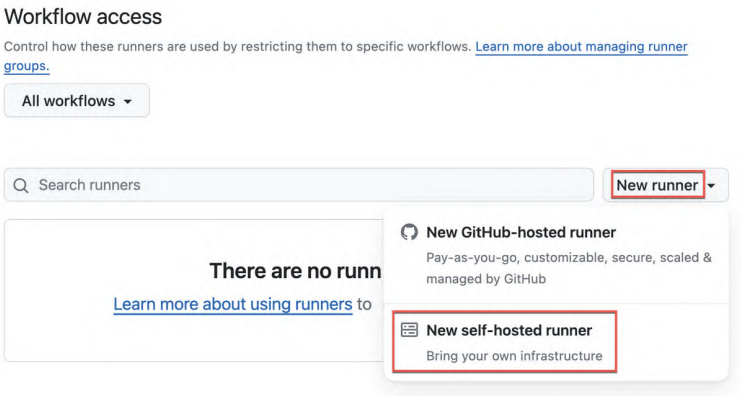


图 4.18 – 向运行器组添加自托管运行器

5. 请注意，添加运行器的对话框与设置自托管运行器配方中出现的对话框相同，可以通过覆盖 RUNNER_URL 来使用容器进行测试：

```
$ docker run -d --rm -e RUNNER_NAME=Runner_Group \
> -e TOKEN={TOKEN} \
> -e RUNNER_URL=https://github.com/{org} \
> simple-ubuntu-runner
```

这会将在默认组中创建运行器！打开运行器并将其分配给你创建的新组（见图 4.19）：

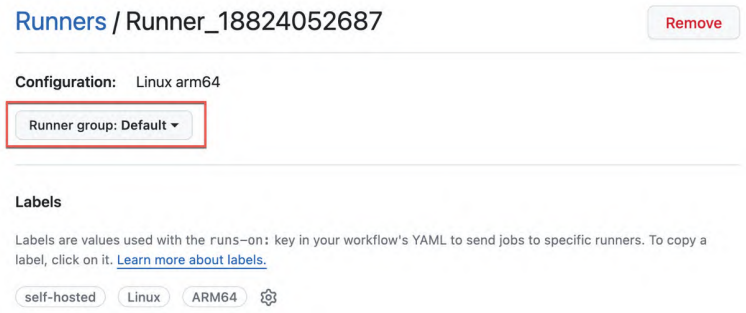


图 4.19 – 将运行器分配给运行器组

请注意，运行器只能分配到一个组。为了在设置过程中直接将运行器分配给一个组，必须将参数传递给配置脚本，这在说明中没有：

```
$ ./config.sh --url $org_or_enterprise_url --token $token --runnergroup rg-runnergroup
```

对于组织和企业管理运行器的访问权限来说，运行器组是一个重要的功能。这个功能很简单，不需要太多解释。在下一个示例中，我们将使用运行器组来添加更大的 GitHub 托管运行器。

4.6. GitHub 托管的运行器

在本章的最后一个示例中，我们将在运行器组中创建一个具有网络隔离的大型 GitHub 托管运行器。

4.6.1 Getting ready

需要在前一个配方中创建的运行器组，该组位于具有付费计划的企业或组织中！

4.6.2 How to do it…

1. 在你创建的运行器组中，点击 **New runner | New GitHub-hosted runner**（新建运行器 | 新建 GitHub 托管运行器）（见图 4.20）：

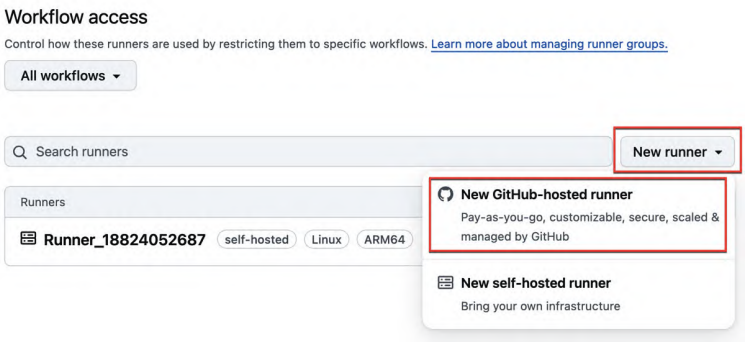


图 4.20 – 创建新的 GitHub 托管运行器

2. 给运行器一个名称。名称必须在 1 到 100 个字符之间，并且只能包含大写字母（A-Z）和小写字母（a-z）、数字（0-9）、点（.）、破折号（-）和下划线（_）。选择一个运行器镜像值（Ubuntu 或 Windows）及其对应的版本（见图 4.21）：

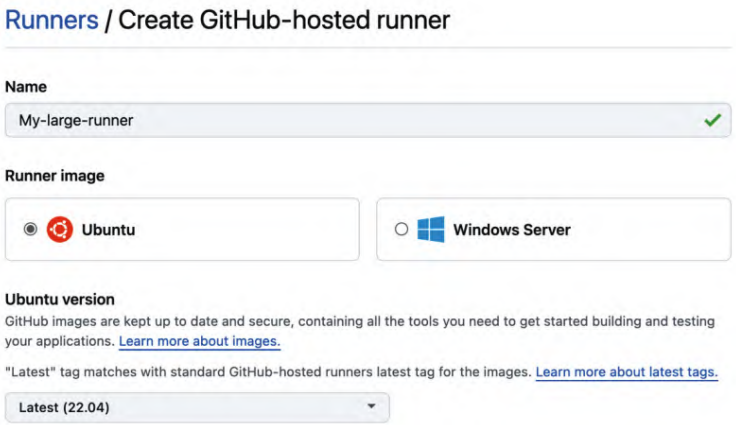


图 4.21 – 配置运行器的名称和镜像

3. 选择运行器的大小（见图 4.22），较大的运行器费用更高：

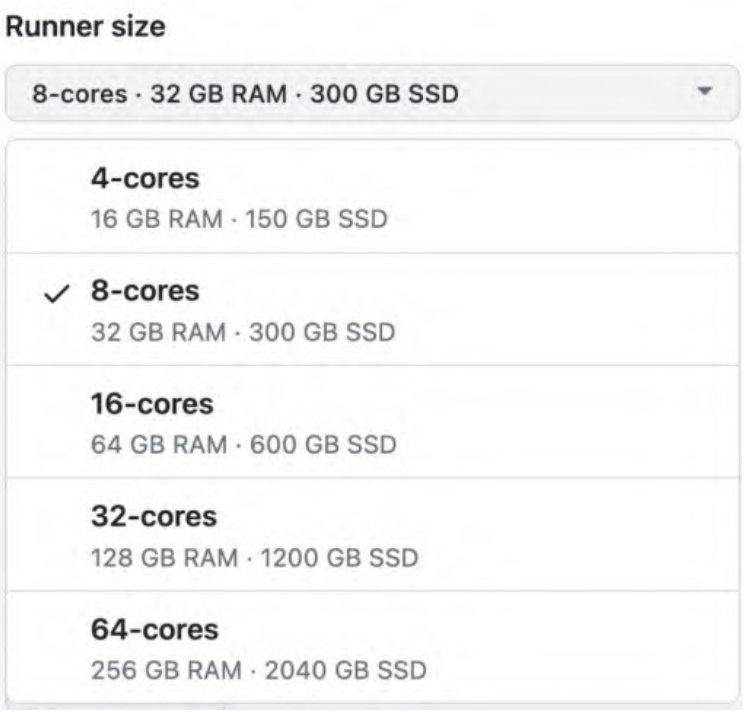


图 4.22 – 为新的运行器选择规模

4. 可以限制最大并发作业数。最大值为 500。将其保留在默认值 50，并将其保留在自动设置创建的运行器组中（见图 4.23）：

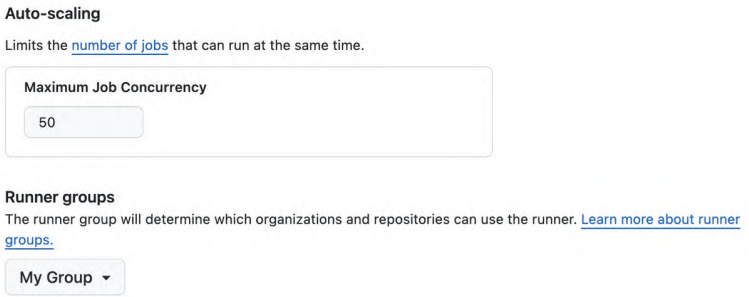


图 4.23 – 设置作业并发和运行器组

5. 可以通过为运行器分配一个唯一且静态的公共 IP 地址范围来启用网络隔离（见图 4.24）。点击创建运行器以完成该过程：

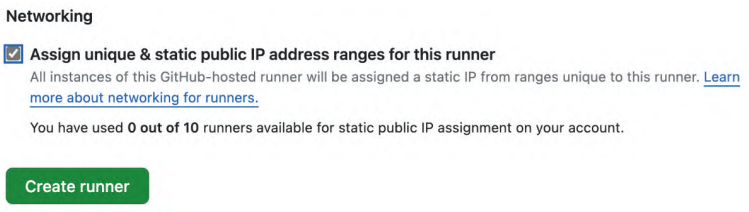


图 4.24 – 为运行器启用网络隔离

6. 请注意，供应需要一些时间。当运行器准备就绪，就可以检查与它关联的 IP 范围，此时会看到标签与运行器的名称相同（见图 4.25）。现在，可以开始在更大的运行器上执行工作流了！

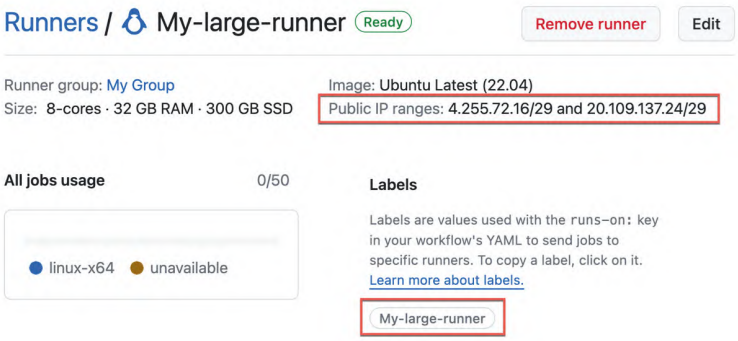


图 4.25 – 具有网络隔离的大型运行器

4.6.3 How it works…

GitHub 将提供更大的运行器，也能分配一个静态的公共 IP 范围。请注意，必须使用该运行器；如果不使用，GitHub 在一段时间后关闭。网络隔离允许运行器访问本地资源，无需公共访问。

第 5 章 使用 GitHub Actions 在 GitHub 中自动化任务

在本章中，我们将重点了解如何使用 GitHub Issues 通过 GitHub Actions 自动化 GitHub 中的常见任务，这通常称为 IssueOps。本章中，将创建一个简单的解决方案，允许管理库。这对于个人账户来说没有意义，但该解决方案应该很容易适应企业环境，其中对代码库的治理（例如：命名约定和权限）是一个重要的话题。

主要内容有：

- 创建 issue 模板
- 使用 GitHub CLI 和 GITHUB_TOKEN 访问资源
- 使用环境进行审批和检查
- 可重用的工作流和复合操作

5.1. 环境要求

如果想要了解所有细节，需要一个 GitHub 组织，也可以在 GitHub 上免费创建一个，甚至是只使用个人账户——其工作原理相同，但不太符合真实场景。可以在 Visual Studio Code 或浏览器中编写工作流程。

5.2. 创建 issue 模板

本示例中，将创建一个简单的模板，以后可以扩展它以收集 IssueOps 工作流中的用户输入。

5.2.1 Getting ready

我们将把 issue 模板添加到前几章中使用过的库中，可以本地库并在 Visual Studio Code 中打开，也可以在浏览器中完成此部分。可以按照我库中的示例（https://github.com/wulfland/GitHubActionsCookbook）进行操作。

5.2.2 How to do it…

1. 在库中创建一个名为.github/ISSUE_TEMPLATE/repo_request.yml 的文件。只要文件是 YAML 或 Markdown 文件，GitHub 就会自动将.github/ISSUE_TEMPLATE 文件夹中的文件视为 issue 模板。
2. 为模板添加名称和描述：

```
name: '📓 Repository Request'
description: 'Request a new repository.'
```

3．使用默认值预填新问题的标题：

```
title: '📄 Repository Request: '
```

4．将一个或多个标签应用于新问题：

```
labels:
  - 'repo-request'
  - 'issue-ops'
```

注意，这些标签必须存在于库中。如果不可用，请创建它们（使用 gh label list 进行检查）：

```
$ gh label create repo-request
$ gh label create issue-ops
```

还可以提供描述或明确的颜色字符串：

```
$ gh label create repo-request \
> -c=#D541D0 \
> -d="Request a new repository"
```

5．将问题分配给一个或多个用户或团队，只需使用 GitHub 用户名即可：

```
assignees:
  - wulfland
```

6．可以自动将新问题分配给一个 GitHub 项目，语法为 {owner}/{project id}。

```
projects: 'wulfland/19
```

请注意，创建问题的人需要对该项目具有写入权限。
如果没有项目，请创建一个新项目。点击右上角的 + 图标，然后选择 **New project**（新建项目）（见图 5.1）。

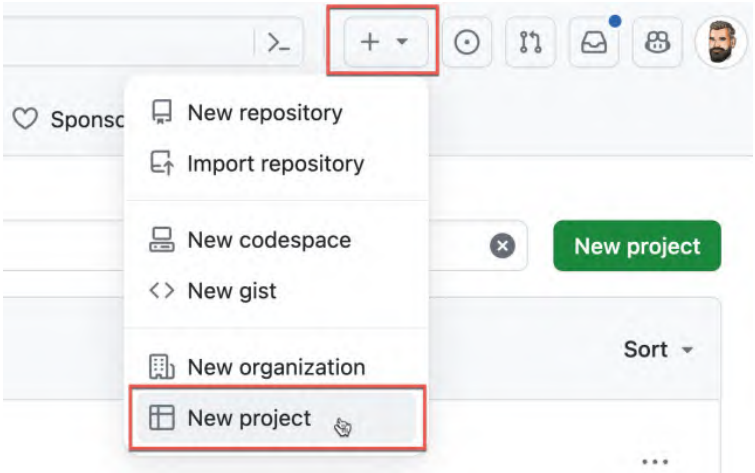


图 5.1 – 创建新项目

选择一个项目模板或从头开始。对于一个简单的管理库请求的项目，您可以直接从一个简单的 Board(看板）开始（见图 5.2)。

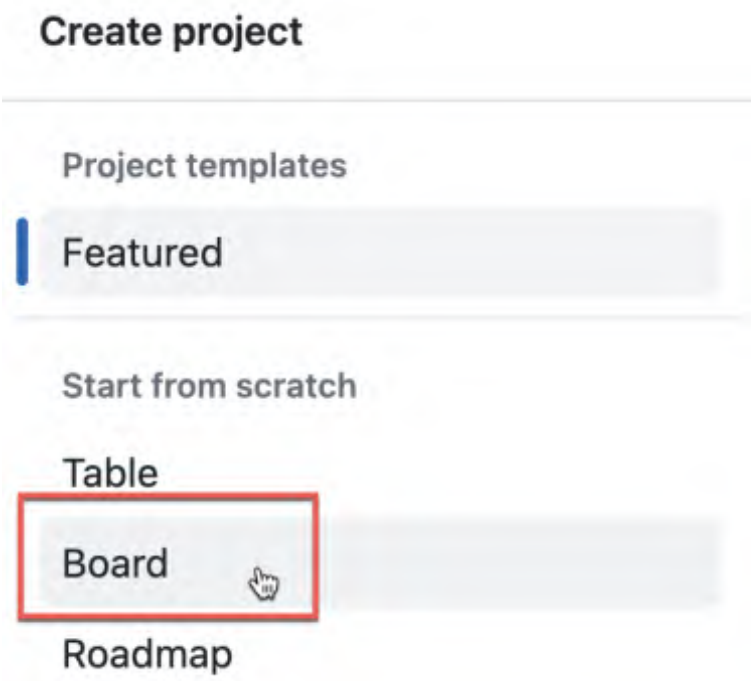


图 5.2 – 选择模板或从头开始

为项目命名——例如，Repository Requests，然后点击 **Create project**(创建项目)。从项目的 URL 中获取项目 ID: https://github.com/users/{owner}/projects/{id}。

7. 在表单正文中，可以定义不同的字段。首先，为请求的库名称添加一个简单的文本输入框，可以使字段成为必填项，并添加附加标签、默认值或占位符：

```
body:
  - type: input
    id: name
    attributes:
      label: 'Name'
      description: 'Name of the repository in lower-case and
kebab casing.'
      placeholder: 'your-name-kebab'
    validations:
      required: true
```

8. 通常，会有一个部门、区域或团队，用于权限或命名约定。添加一个简单的下拉菜单，从中选择两个示例部门：

```
- type: dropdown
  id: department
```

```
attributes:
  label: 'Department'
  description: 'Pick your department. It will be used as a   prefix for the repository
  ↪  name.'
  multiple: false
  options:
    - dep1
    - dep2
  default: 0
validations:
  required: true
```

9. 将文件提交到库。
10. 在 **Issues | New issue** 下，可以选择模板并点击 **Get started**(见图 5.3)。

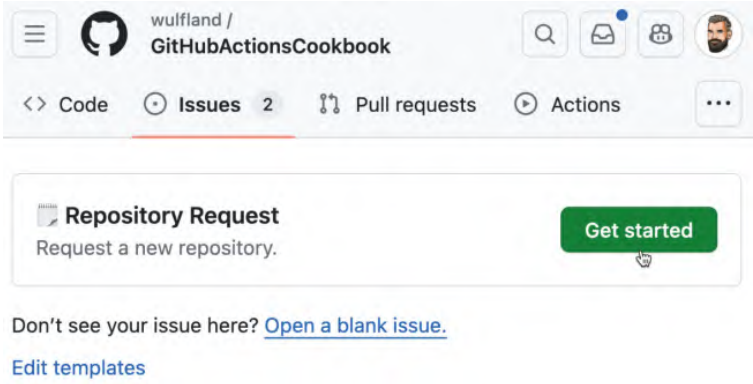


图 5.3 – 使用模板创建 issue

标签、项目和分配对象已自动设置，控件为必填字段，并设置了正确的默认值（见图 5.4)。

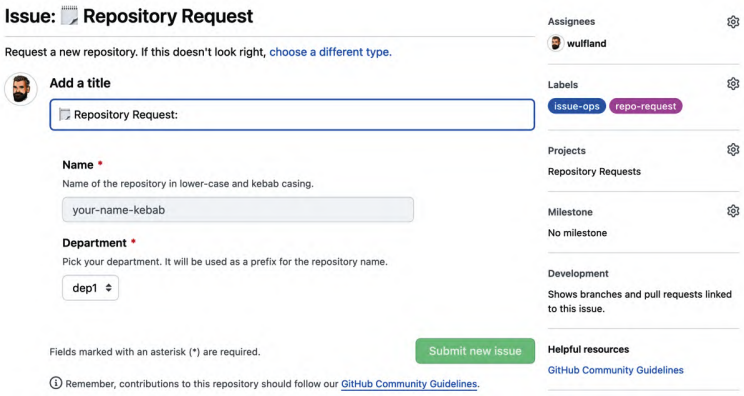


图 5.4 – 使用问题模板创建新 issue

填写表单并保存新 issue。

5.2.3 How it works…

问题（Issue）和拉取请求（Pull Request）模板，是引导用户创建问题或拉取请求的有力工具。可以通过界面生成模板，使其更易于查找（https://docs.github.com/en/communities/usi

ng-templates-to-encourage-useful-issuesand-pull-requests/syntax-for-issue-forms)。模板可以是纯 Markdown，但通过此示例中使用的新自定义模板，也可以创建包含多种表单元素（如 Markdown、文本区域、输入框、下拉菜单和复选框），还可以添加验证并设置默认值。有关 GitHub 表单模式的完整语法，请参阅 https://docs.github.com/en/communities/using-templates-toencourage-useful-issues-and-pull-requests/syntax-for-githubs-formschema。填写完表单后，数据会以 Markdown 格式添加到问题或拉取请求的正文中。模板仅在用户创建问题或拉取请求时提供支持。之后，编辑时就只是 Markdown 了。

5.2.4 There's more…

GitHub 会在创建新问题时显示.github/ISSUE_TEMPLATE 文件夹中的所有有效 Markdown 或 YAML 表单模板。但您可以配置额外的链接到外部系统，并可以配置是否允许空白问题或强制用户选择模板（请参阅图 5.5）。

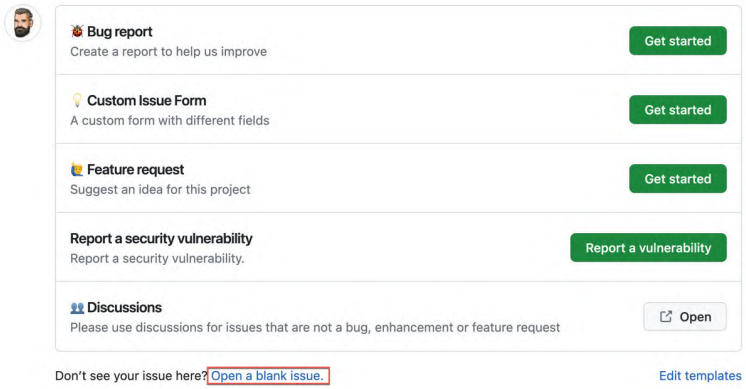


图 5.5 – 配置模板选择器表单

要配置模板选择器，请将 config.yml 文件添加到.github/ISSUE_TEMPLATE 文件夹中。将 blank_issues_enabled 设置为 true 或 false，并将额外的链接添加到 contact_links 数组中：

```
blank_issues_enabled: false
contact_links:
  - name: 👥 Discussions
    url: https://github.com/wulfland/AccelerateDevOps/discussions/new
    about: Please use discussions for issues that are not a bug, enhancement or feature
request.
```

更多详细信息，请参考官方文档:https://docs.github.com/en/communities/usingtemplates-to-encourage-useful-issues-and-pull-requests/configuringissue-templates-for-your-repository

5.3. 使用 GitHub CLI 和 GITHUB_TOKEN 访问资源

在本示例中，我们将解析上一章中的问题，并在工作流中使用 GitHub CLI 与问题进行交互。

5.3.1 Getting ready

需要上一章中的问题模板，可以在 Visual Studio Code 中创建工作流，也可以直接在 GitHub 中创建。

5.3.2 How to do it…

1. 创建一个新的工作流文件.github/workflows/issue-ops.yml，并将其命名为 issue-ops：

```
# Issue ops
name: issue-ops
```

2. 为工作流使用 issues 触发器。请注意，我们没有使用 created 或 edited 事件，而是使用了 labeled 事件。这允许用户在修改请求时重新标记问题：

```
on:
  issues:
    types: [labeled]
```

3. 添加一个 issue-ops 作业：

```
jobs:
  issue-ops:
```

我们只想在特定标签存在时运行此作业，并向作业添加如下条件：

```
if: ${{ github.event.label.name == 'repo-request' }}
```

该作业可以在最新的 Ubuntu 镜像上运行：

```
runs-on: ubuntu-latest
```

4. 要与问题进行交互，工作流需要对问题具有写入权限。要使用 GitHub CLI，还需要对内容具有读取权限：

```
permissions:
  issues: write
  contents: read
```

5. 市场中有一个操作可以帮助解析作为表单创建的问题的主体。添加 zentered/issue-forms-body-parser 并给它一个 id 属性，以便稍后访问输出：

```
steps:
  - name: Issue Forms Body Parser
    id: parse
    uses: zentered/issue-forms-body-parser@v2.0.0
```

6. 接下来，添加主脚本。设置 id 以在作业级别访问输出。还将 GH_TOKEN 环境变量设置为 GITHUB_TOKEN。此令牌将由 CLI 用于与问题进行交互：

```
- name: Repository Request Validation
  id: repo-request
  env:
    GH_TOKEN: ${{ github.token }}
  run: |
```

7. 作为第一步，使用 jq 从具有 parse ID 的步骤的输出中读取 name 和 department 的值，并将它们存储在变量中。在 JavaScript 中，可以直接访问对象，但在 bash 脚本中，必须使用 jq：

```
repo_name=$(echo '${{ steps.parse.outputs.data }}' | jq -r '.name.text')
repo_dept=$(echo '${{ steps.parse.outputs.data }}' | jq -r '.department.text')
```

将这两个变量与最终名称（我们的例子中，部门是前缀）组合起来：

```
repo_full_name=$repo_dept-$repo_name
```

将名称设置为输出，以便在以后的步骤或作业中使用：

```
echo "REPO_NAME=$repo_full_name" >> "$GITHUB_OUTPUT"
```

8. 添加一些验证逻辑。这里我将添加两个示例，可以查看工作流文件中的其余部分：https://github.com/wulfland/GitHubActionsCookbook/blob/main/.github/workflows/issue-ops.yml。
首先，设置默认消息和退出代码。默认消息由两部分组成：首先，想要提及创建问题的人员，然后根据验证的输出添加消息：

```
mention="@${{ github.event.issue.user.login }}: "
message="Requested repository '$repo_full_name' will be sent for approval."
exitcode=0
```

接下来，添加一个验证规则，即名称不能为空：

```
# shall not be empty
if [ -z $repo_full_name ]; then
  message="Repository name is empty.";
  exitcode=1;
```

```
fi;
```

此外，添加一个验证规则，即只能使用字母数字字符和连字符（如果您希望使用 kebab casing(短横线）命名法）：

```
# shall be alphanumeric and minus only
if [[ "$repo_full_name" =~ [^\-a-zA-Z0-9] ]]; then
  message="Repository name shall be alphanumeric and minus only.";
  exitcode=1;
fi;
```

9. 如果验证失败，请从问题中删除标签，并告诉用户修复问题并重新应用标签：

```
if [ $exitcode -ne 0 ]; then
  gh issue edit ${{ github.event.issue.number }} \
    --remove-label repo-request
  message=$message" Please fix the issue and try again by applying the label
  ↪  'repo-request' again to the issue.";
fi;
```

10. 最后，在问题下评论消息，并在验证失败时使作业失败：

```
gh issue comment ${{ github.event.issue.number }} \
  -b „$mention $message"
exit $exitcode
```

11. 将 REPO_NAME 输出设置为步骤中设置的输出。我们将在下一个作业中使用它来创建库：

```
outputs:
  REPO_NAME: ${{ steps.repo-request.outputs.REPO_NAME }}
```

12. 现在，使用表单模板创建一个新问题。首先使用一个无效的名称（例如 my_repo），看看它如何向问题添加评论（请参阅图 5.6）。

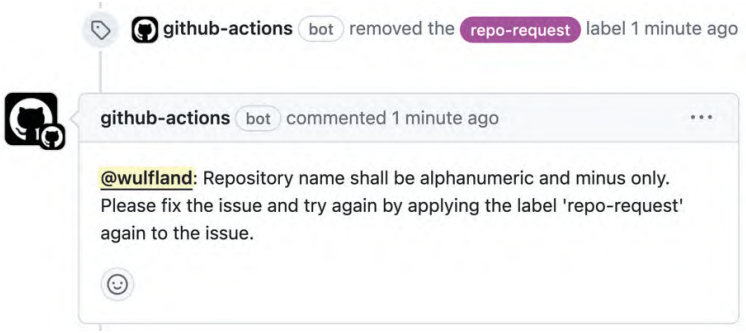


图 5.6 – 工作流与问题的交互

修复名称（例如改为 my-repo），然后再次将 repo-request 标签应用到问题上。

5.3.3 How it works…

来了解一下其工作流程。

工作流权限和 GITHUB_TOKEN

每个工作流作业开始时，GitHub 会自动创建一个唯一的 GITHUB_TOKEN，可以在工作流中使用它来与 GitHub 进行交互，也可以使用此令牌在工作流作业中进行身份验证。

可以为个人账户，以及组织和库配置默认权限——默认是只读权限。最佳实践是保持只读权限，并在工作流或作业中授予显式权限。

GITHUB_TOKEN 的权限可以配置为顶层键，以应用于工作流中的所有作业，也可以在特定作业中配置。当特定作业中添加 permissions 键时，该作业中使用 GITHUB_TOKEN 的所有操作和运行命令，都会获得指定的访问权限。

对于每个可用的范围，您可以分配以下权限之一：read(读)、write(写) 或 none(无)。

> **Note**
>
> 如果为这些范围中的一个指定了访问权限，则所有未指定的范围将自动设置为 none!

还可以一次性设置所有权限。以下设置将所有权限设置为只读：

```
permissions: read-all
```

以下将授予所有范围的写入访问权限：

```
permissions: write-all
```

最后一个将所有范围设置为 none：

```
permissions: {}
```

在我们的示例中，我们需要对问题进行写入权限，并且 CLI 需要对库的读取权限：

```
permissions:
  issues: write
  contents: read
```

所有其他权限将自动设置 none。

有关 GITHUB_TOKEN 和工作流权限的更多信息，请参阅 https://docs.github.com/en/actions/using-jobs/assigning-permissionsto-jobs。

步骤和作业输出

在第 3 章中，介绍了有关环境文件的知识。我们在本示例中使用它们来设置，将在后续作业中使用的输出，还会使用它们来访问表单解析器的数据。

表单解析器是来自市场的操作，可帮助我们访问使用 issue 模板创建的 issue 正文。

使用 id 来访问输出：

```
steps:
- name: Issue Forms Body Parser
  id: parse
  uses: zentered/issue-forms-body-parser@v2.0.0
```

在 JavaScript(即 GitHub Script 操作）中，可以直接访问 JSON 对象：

```
console.log(data.name.text);
console.log(data.department.text);
```

在 bash 中，不能像在 JavaScript 中那样直接访问 JSON 属性。需要使用一个命令行 JSON 处理器（例如 jq），来解析 JSON 字符串并访问其属性：

```
repo_name=$(echo '${{ steps.parse.outputs.data }}' | jq -r '.name.text')
```

使用 GitHub CLI 对 issue 进行评论

在第 3 章中，使用 octokit 和 REST API 对问题进行了评论。在本示例中，我们使用 GitHub CLI 来完成这个操作。

为了让 CLI 工作，必须首先检出版本库。还必须在工作流步骤中将 GH_TOKEN 环境变量设置为工作流令牌：

```
env:
  GH_TOKEN: ${{ github.token }}
```

使用 CLI 很简单——可以在评论中使用 @ 加上用户名来提及用户：

```
mention="@${{ github.event.issue.user.login }}: "
message="Requested repository '$repo_full_name' will be sent for approval."

gh issue comment ${{ github.event.issue.number }} \
  -b "$mention $message"
```

5.4. 使用环境进行审批和检查

在本示例中，我们将使用环境审批来在创建版本库之前获取审批，将使用一个 GitHub App 进行身份验证。因为版本库的创建通常发生在组织范围内，并且不能使用 GITHUB_TOKEN，所以必须使用一个 GitHub App 或具有正确范围的个人访问令牌（PAT)。

5.4.1 Getting ready

确保你已完成前面的示例，并继续在同一个库中进行操作。

5.4.2 How to do it…

1. 在库中，进入 **Settings**(设置) → **Environments**(环境)，然后点击 **New environment**(新建环境)。
2. 将环境命名为 repo-creation，然后点击 **Configure environment**(配置环境)。
3. 将自己添加为 **Required reviewer**(必需审阅者)，并且不允许管理员绕过此规则（请参阅图 5.7)。

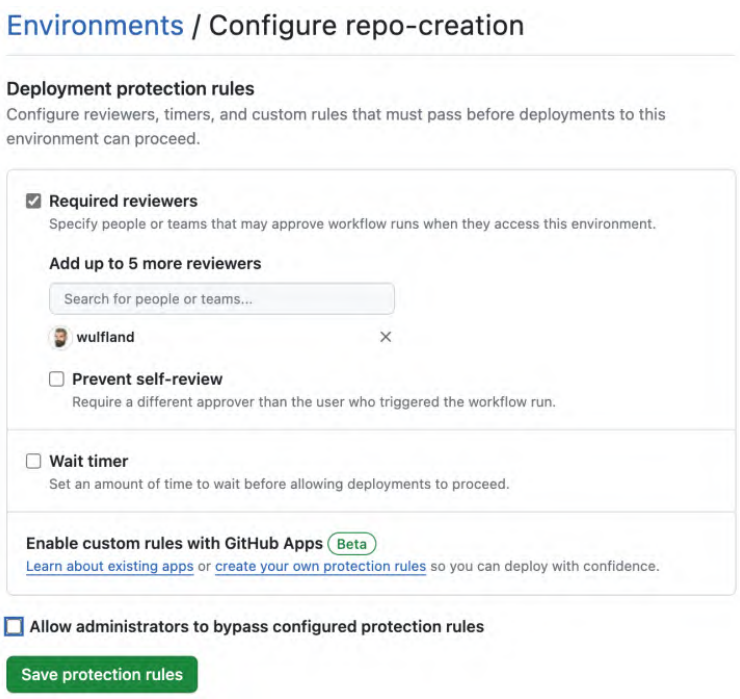


图 5.7 – 配置环境保护规则

4. 点击 **Save protection rules**(保存保护规则)。
5. 下一步是创建一个应用（App）来进行身份验证。我建议使用组织（organization）来尝试此示例，也可以使用您的个人账户。前往 https://github.com/settings/apps，然后点击 **New GitHub App**(新建 GitHub 应用)。
6. 为其指定一个名称（例如，{your username}-repo-creation)，并将 Homepage URL(主页 URL）设置为版本库的 URL。
7. 在 Repository permissions(库权限）部分，可选择以下权限：

   - Administration(管理)：Read and write(读写)
   - Contents(内容)：Read and write(读写)
   - Issues(议题)：Read and write(读写)

   在 Organization permissions(组织权限）部分，选择：

   - Administration(管理)：Read and write(读写)

8. 点击 Save(保存）应用。在新创建的应用页面中，点击 **Generate a private key**(生成私钥)，私钥将会自动下载到本地设备上。
9. 从应用的 **General**(常规）选项卡中复制 App ID。

10. 在 GitHub 上的应用中，选择 **Install App**(安装应用) 并点击 **Install**(安装)。选择组织或账户，点击 **Install**(安装)，保持 **All repositories**(所有版本库) 被选中，然后点击 **Install**(安装)。
11. 返回到版本库中的环境。添加一个新的密钥 PRIVATE_KEY，并将之前下载的密钥文件内容添加进去。另外，添加一个 APP_ID 变量，其值为应用 ID(请参阅图 5.8)。

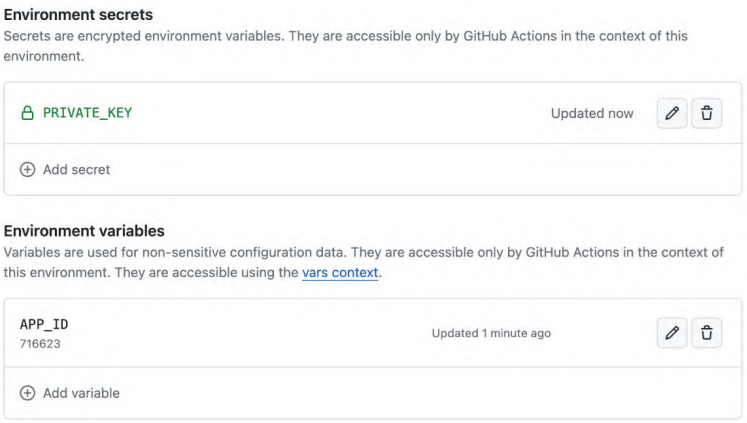


图 5.8 – 向环境添加变量和密钥

12. 编辑工作流文件，并添加一个 create-repo 作业。使该作业依赖于之前的作业 (needs: issue-ops)，并将其分配给之前创建的环境 (environment: repo-creation):

```
create-repo:
  needs: issue-ops
  runs-on: ubuntu-latest
  environment: repo-creation
```

13. 要使用 GitHub 令牌与问题进行交互，请设置工作流的权限:

```
  permissions:
    issues: write
    contents: read
```

14. 可以使用全局环境变量，这些变量可以在所有步骤中使用，将 REPO_OWNER 设置为安装了应用的组织或账户，也可以将此值保存为环境中的变量。将 REPO_NAME 设置为上一个作业的输出。USER 和 ISSUE_NUMBER 设置为上下文的值，以便于访问:

```
env:
  REPO_OWNER: ${{ vars.ORGANIZATION }}
  REPO_NAME: ${{ needs.issue-ops.outputs.REPO_NAME }}
  USER: ${{ github.event.issue.user.login }}
  ISSUE_NUMBER: ${{ github.event.issue.number }}
```

15. 要使用应用进行身份验证，可以使用 actions/create-github-app-token 操作。给它一个 ID 以便后续引用:

```
steps:
  - name: Create app token
    uses: actions/create-github-app-token@v1.6.2
    id: get-workflow-token
    with:
      app-id: ${{ vars.APP_ID }}
      private-key: ${{ secrets.PRIVATE_KEY }}
      owner: ${{ vars.ORGANIZATION }}
```

16. 创建版本库，并将 URL 设置为输出参数：

```
- name: Create repository
  id: create-repo
  env:
    GH_TOKEN: ${{ steps.get-workflow-token.outputs.token }}
  run: |
    REPO_URL=$(gh repo create $REPO_OWNER/$REPO_NAME --private --clone)
    echo "repo_url=$REPO_URL" >> "$GITHUB_OUTPUT"
    echo "Repositeory '$REPO_NAME' has been successully created: $REPO_URL"
```

17. 如果操作成功，请在 issue 下发表评论，通知用户版本库已创建：

```
- name: Notify User
  if: ${{ success() }}
  env:
    GH_TOKEN: ${{ github.token }}
    REPO_URL: ${{ steps.create-repo.outputs.repo_url }}
  run: |
    gh issue comment $ISSUE_NUMBER \
      -b "@$USER: Repository '$REPO_OWNER/$REPO_NAME' has been created successfully:
      ↪  $REPO_URL" \
      --repo ${{ github.event.repository.full_name }}
```

18. 如果发生错误，也通过在问题下发表评论通知用户：

```
- name: Handle Exception
  if: ${{ failure() }}
  env:
    GH_TOKEN: ${{ github.token }}
  run: |
    gh issue comment $ISSUE_NUMBER \
      -b "@$USER: Repository '$REPO_OWNER/$REPO_NAME' creation failed. Please contact the
      ↪  administrator."\
      --repo ${{ github.event.repository.full_name }}
```

19. 提交并推送工作流，并创建一个新的 **Repository Request** issue。可以根据通知设置收到通知，以审查工作流。该工作流如图 5.9 所示。

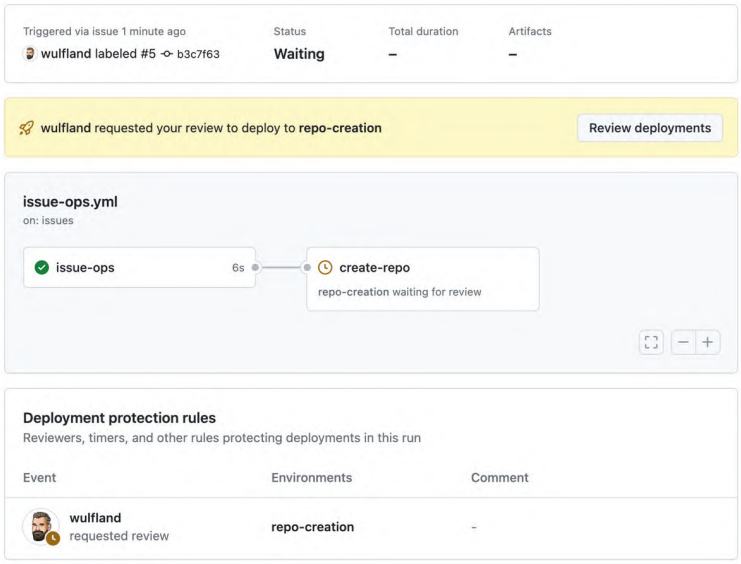


图 5.9 – 使用环境进行手动工作流审批

20. 点击 **Review deployments**(审核部署),选择环境,然后点击 **Approve and deploy**(批准并部署)(请参阅图 5.10)。

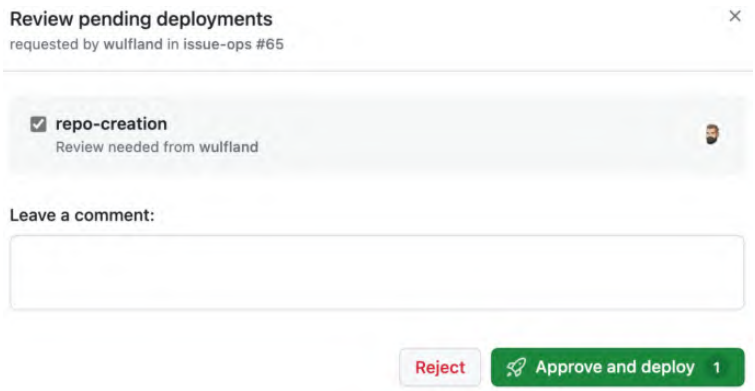


图 5.10 – 批准部署到环境

该工作流将创建版本库,并使用问题中的评论通知发起请求的用户(请参阅图 5.11)。

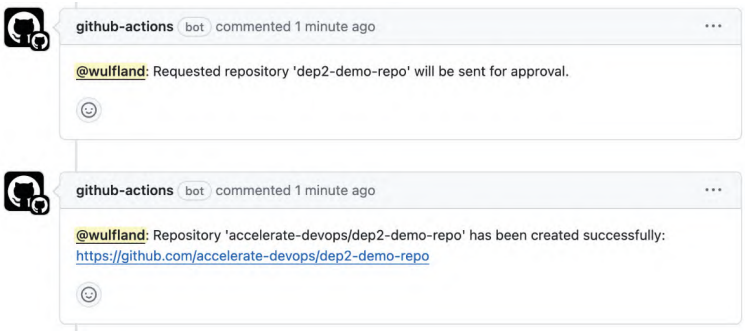


图 5.11 – 工作流使用问题评论通知用户关于版本库创建成功

检查链接是否有效,以及版本库是否正确创建。

5.4.3 How it works…

可以将工作流中的作业分配给环境，从而为该环境添加保护规则，以及特定的变量和密钥。

环境

环境通过网页界面或 API 在版本库中创建：

```
curl -L \
  -X PUT \
  -H "Accept: application/vnd.github+json" \
  -H "Authorization: Bearer <YOUR-TOKEN>" \
  -H "X-GitHub-Api-Version: 2022-11-28" \
  https://api.github.com/repos/OWNER/REPO/environments/<NAME> \
  -d '{"wait_timer":30,"prevent_self_review":false,"reviewers": [{"-type":"User","id":1},
  ↪ {"type":"Team","id":1}], "deployment_branch_policy":{"protected_branches":false,
  ↪  "custom_branch_policies":true}}'
```

大多数情况下，可以像本示例中那样使用网页界面来创建。在工作流中，环境通过其名称引用：

```
jobs:
  deployment:
    runs-on: ubuntu-latest
    environment: production
```

可以添加一个 URL，该 URL 将在工作流中显示：

```
environment:
  name: production
  url: https://writeabout.net
```

可以在工作流中创建动态环境并向其部署，可以将每个拉取请求部署到一个隔离的环境中进行测试。

可以使用不同的保护规则来保护环境：

- 必需审阅者：可以列出最多六个用户或团队作为必需审阅者，以批准引用该环境的工作流作业。审阅者必须至少拥有版本库的读取权限，只需要一个必需审阅者批准作业即可继续。
- 等待计时器：可以在作业最初触发后暂停工作流一段时间。时间（以分钟为单位）必须是一个介于 0 和 43,200(30 天）之间的整数，可以使用 API 在该时间内取消工作流。
- 部署分支和标签：使用部署分支和标签来限制哪些分支和标签可以部署到该环境。环境针对部署分支和标签的选项为无限制、仅受保护分支或选定的分支和标签。
  在后者设置中，可以添加名称模式以针对单个或一组分支或标签进行操作——例如：main 或 release/*。将环境连接到分支保护规则非常强大，可以为这些规则提供更多的保护规则——例如：强制执行代码所有者或部署到特定环境。

环境也有特定的密钥和变量，允许在同一个工作流中使用不同的配置。

还有自定义部署规则来保护环境，此功能在编写本书时仍处于公开测试阶段。自定义部署规则

基本上是 GitHub Apps，允许编写自己的集成。这使得 Datadog、Honeycomb 和 ServiceNow 等服务能够为部署提供自动审批。

要了解有关环境的更多信息，请参阅：https://docs.github.com/en/actions/deployment/targeting-different-environments/using-environments-fordeployment。

认证

可以使用 GitHub 令牌和工作流权限做很多事情。但在自动化组织级别的操作时，可能需要使用个人访问令牌（PAT）或 GitHub App。GitHub App 是推荐的方式，因其与用户无关。

我们已经在上一章中了解了 GitHub App。要在工作流中使用 GitHub App 进行身份验证，可以使用 actions/create-github-app-token 操作：

```
steps:
  - name: Create app token
    uses: actions/create-github-app-token@v1.6.2
    id: get-workflow-token
    with:
      app-id: ${{ vars.APP_ID }}
      private-key: ${{ secrets.PRIVATE_KEY }}
```

需要应用 ID 和私钥，可将其存储为环境变量和密钥。然后，通过工作流步骤的输出来访问该令牌：

```
- name: Create repository
  env:
    GH_TOKEN: ${{ steps.get-workflow-token.outputs.token }}
```

通过此操作，在工作流中使用 GitHub App 非常简单。

5.4.4 There's more…

issue 是与用户交互的好方法——可能还希望将自动化的状态存储在其他地方，可以更新 YAML 或 Markdown 文件，或调用外部系统。

也可以使用 GitHub Projects 来可视化 issue，这样 issue 就代表了自动化对象生命周期中的状态。

GitHub Projects 非常灵活，可以将其中的问题和拉取请求来自不同的版本库。所以它相当复杂，需要使用内部 ID 来引用字段。

在调整工作流之前，运行以下命令来列出用于跟踪问题的项目的字段（我们的例子中，ID 是 19，所有者是 wulfland）：

```
$ gh project field-list <ID> --owner <OWNER> --format json | jq
```

这将生成一个包含项目中所有字段的 JSON 对象。查找这些 ID。对于状态（Status）字段，还需要选项的 ID：

```
{
    "id": "PVTSSF_lAHOAFCCsc4AZoDtzgQZkEg",
    "name": "Status",
    "type": "ProjectV2SingleSelectField",
    "options": [
      {
        "id": "f75ad846",
        "name": "Request"
      },
      {
        "id": "e05aa0a3",
        "name": "Repository Created"
      },
      {
        "id": "98236657",
        "name": "Deleted"
      }
    ]
  },
```

将内部 ID 作为变量存储在环境中，如图 5.12 所示。

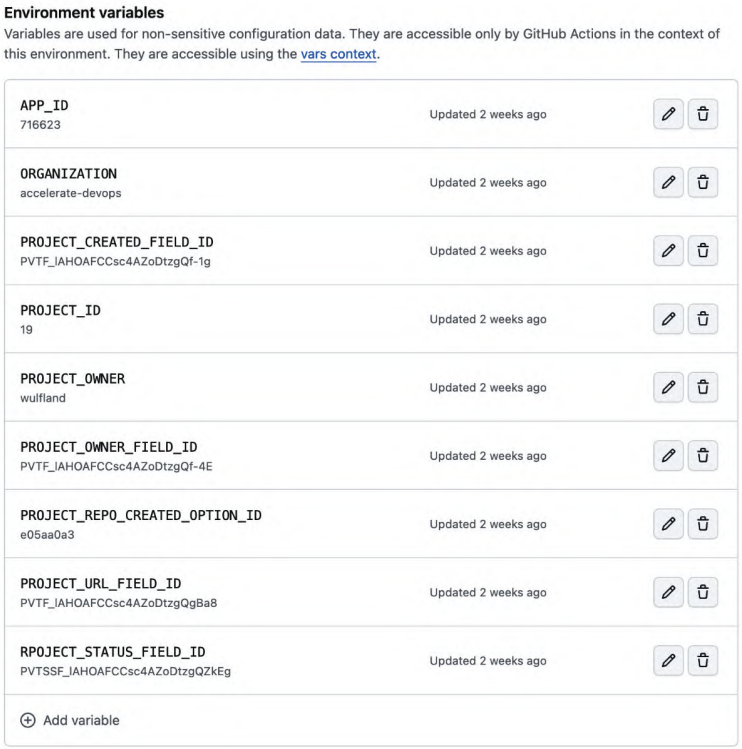


图 5.12 – 使用内部项目 ID 来引用字段和选项

在工作流中，添加一个步骤并设置环境变量：

```
- name: Update Project
  if: ${{ success() }}
```

```
  env:
    GH_TOKEN: ${{ secrets.PROJECT_TOKEN }}
    REPO_URL: ${{ steps.create-repo.outputs.repo_url }}
    PROJECTNUMBER: ${{ vars.PROJECT_ID }}
    PROJECTOWNER: ${{ vars.PROJECT_OWNER}}
```

首先，必须接收内部项目 ID，因为后续命令需要。普通的数字项目 ID 并不能在所有命令中使用：

```
run: |
  project_id=$(gh project list --owner "$PROJECTOWNER" --format json | jq -r '.projects[] |
  ↪  select(.number=='$PROJECTNUMBER') | .id')
```

接下来，获取内部 issue ID：

```
issue_id=$(gh project item-list $PROJECTNUMBER \
  --owner "$PROJECTOWNER" \
  --format json \
  | jq -r '.items[] \
  | select(.content.number=='$ISSUE_NUMBER') | .id')
```

现在，可以使用工作流中的值来更新字段，将项目的状态设置为 created 选项：

```
gh project item-edit \
  --id $issue_id \
  --field-id ${{ vars.RPOJECT_STATUS_FIELD_ID }} \
  --single-select-option-id ${{ vars.PROJECT_REPO_CREATED_OPTION_ID }} \
  --project-id $project_id
```

将创建的版本库的 URL 设置为 URL 字段：

```
gh project item-edit \
  --id $issue_id \
  --field-id ${{ vars.PROJECT_URL_FIELD_ID }} \
  --text $REPO_URL \
  --project-id $project_id
```

最后，将创建日期字段设置为当前日期：

```
gh project item-edit --id $issue_id \
  --field-id ${{ vars.PROJECT_CREATED_FIELD_ID }} \
  --date $(date +%Y-%m-%d) \
  --project-id $project_id
```

现在，可以在 Projects(项目) 中跟踪版本库请求的状态（图 5.13)。

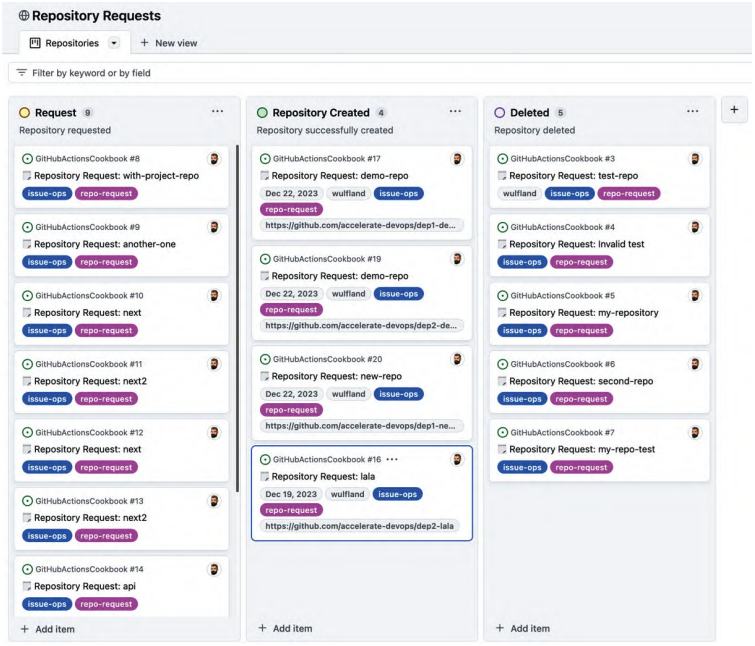


图 5.13 – 在 GitHub Projects 中跟踪 IssueOps 的状态

元数据也可以在每个单独的问题卡片上看到（请参阅图 5.14）。

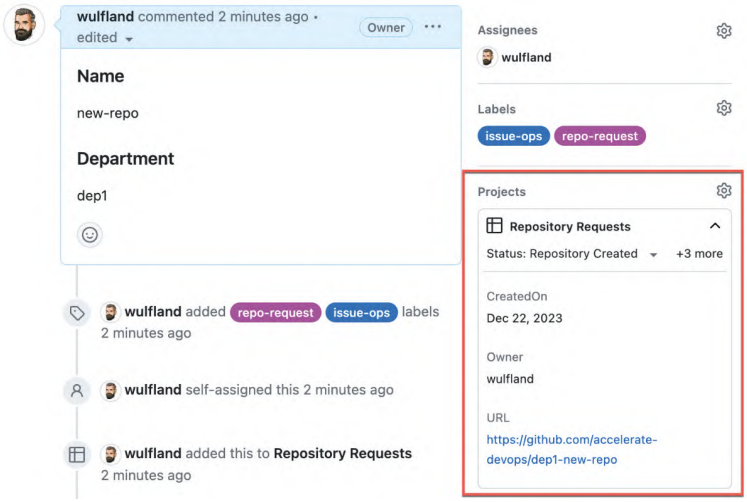


图 5.14 – GitHub Projects 卡片在 GitHub Issues 中

GitHub Projects 非常适合跟踪问题状态，但使用 GraphQL API 或 CLI 自动化它并不容易。

5.5. 可重用工作流和复合操作

如果开始使用 IssueOps 进行自动化，工作流会很快变得非常复杂。最终会在作业和步骤上使用大量的 `if` 语句。为了保持这些解决方案的可维护性，可以使用组合操作或可重用工作流，不仅可以重用功能，还可以将复杂的工作流分解为更小的部分。

我们已经在之前的章节中介绍了组合操作。在这个示例中，将使用一个可重用工作流来为 IssueOps 解决方案添加删除功能。

5.5.1 Getting ready

请确保已完成本章前面的所有示例。

5.5.2 How to do it…

1. 创建一个新的 delete-repo.yml 工作流文件。
2. 作为触发器，我们使用 workflow_call 触发器。这表明该工作流是可重用工作流，定义工作流所需的输入：

```
on:
  workflow_call:
    inputs:
      REPO_NAME:
        description: 'Repository name'
        required: true
        type: string
      ISSUE_USER:
        description: 'User who created the issue'
        required: true
        type: string
      ISSUE_NUMBER:
        description: 'Issue number'
        required: true
        type: number
```

3. 可重用工作流就是一个普通的工作流，可以有一个或多个作业，这些作业也与一个环境相关联：

```
  jobs:
    delete:
      runs-on: ubuntu-latest
      environment: repo-cleanup
      steps:
```

可以创建一个新的环境用于删除版本库，并添加 PRIVATE_KEY、APP_ID、ORGANIZATION、PROJECT_OWNER 和 REPO_OWNER。或者，为了简单起见，可以重用 repo-creation 环境。

4. 从应用获取令牌进行身份验证，就像我们在 issue-ops 工作流中所做的那样：

```
- name: Create app token
  uses: actions/create-github-app-token@v1.6.2
  id: get-workflow-token
  with:
    app-id: ${{ vars.APP_ID }}
```

```
    private-key: ${{ secrets.PRIVATE_KEY }}
    owner: ${{ vars.ORGANIZATION }}
```

5．接下来，使用提供的令牌删除版本库：

```
- name: Delete repository
  id: delete-repo
  env:
    GH_TOKEN: ${{ steps.get-workflow-token.outputs.token }}
    REPO_NAME: ${{ inputs.REPO_NAME }}
    REPO_OWNER: ${{ vars.REPO_OWNER }}
  run: |
    gh repo delete $REPO_OWNER/$REPO_NAME --yes
    echo "Repository '$REPO_NAME' has been successfully deleted."
```

6．通知用户并关闭 issue：

```
- name: Notify User
  if: ${{ success() }}
  env:
    GH_TOKEN: ${{ github.token }}
    ISSUE_NUMBER: ${{ inputs.ISSUE_NUMBER }}
    ISSUE_USER: ${{ inputs.ISSUE_USER }}
    REPO_NAME: ${{ inputs.REPO_NAME }}
    REPO_OWNER: ${{ vars.REPO_OWNER }}
  run: |
    gh issue comment $ISSUE_NUMBER \
      -b "@$ISSUE_USER: Repository '$REPO_OWNER/$REPO_NAME' has been deleted
      ↪  successfully." \
      --repo ${{ github.event.repository.full_name }}
    gh issue close $ISSUE_NUMBER \
      --repo ${{ github.event.repository.full_name }}
```

7．如果失败，也会通知用户：

```
- name: Handle Exception
  if: ${{ failure() }}
  env:
    GH_TOKEN: ${{ github.token }}
    ISSUE_NUMBER: ${{ inputs.ISSUE_NUMBER }}
    ISSUE_USER: ${{ inputs.ISSUE_USER }}
  run: |
    gh issue comment $ISSUE_NUMBER \
      -b "@$ISSUE_USER: Repository '$REPO_OWNER/$REPO_NAME' deletion failed. Please
      ↪  contact the administrator." \
      --repo ${{ github.event.repository.full_name }}
```

8．要使用这个工作流，创建一个名为 handle-issue.yml 的新工作流文件。我们让它运行在

标记了标签的问题上，并授予它对问题的写权限：

```
name: Handle Issue
on:
  issues:
    types: [labeled]
permissions:
  contents: read
  issues: write
```

9. 为了解析问题，我们添加一个通用作业，该作业使用我们在 `issue-ops` 工作流中使用的相同逻辑（在设置输出变量之前直接复制过来）：

```
jobs:
  parse-issue:
    runs-on: ubuntu-latest
    outputs:
      REPO_NAME: ${{ steps.repo-request.outputs.REPO_NAME }}
    steps:
      - name: Issue Forms Body Parser
        id: parse
        uses: zentered/issue-forms-body-parser@v2.0.0
      - name: Repository Request Validation
        id: repo-request
        env:
          GH_TOKEN: ${{ github.token }}
        run: |
          repo_name=$(echo '${{ steps.parse.outputs.data }}' | jq -r '.name.text')
          repo_dept=$(echo '${{ steps.parse.outputs.data }}' | jq -r '.department.text')
          repo_full_name=$repo_dept-$repo_name
          echo "REPO_NAME=$repo_full_name" >> "$GITHUB_OUTPUT"
```

10. 然后，添加一个将调用另一个工作流文件的作业。当应用的标签是 `delete-repo` 时，有条件地执行该作业。使用 `with` 部分传递库名称和其他参数：

```
repo-deletion:
  name: "Delete a repository"
  if: github.event.label.name == 'delete-repo'
  uses: ./.github/workflows/delete-repo.yml
  with:
    REPO_NAME: ${{ needs.parse-issue.outputs.REPO_NAME }}
    ISSUE_USER: ${{ github.event.issue.user.login }}
    ISSUE_NUMBER: ${{ github.event.issue.number }}
    needs: parse-issue
  with:
    REPO_NAME: ${{ needs.parse-issue.outputs.REPO_NAME }}
    ISSUE_USER: ${{ github.event.issue.user.login }}
```

```
    ISSUE_NUMBER: ${{ github.event.issue.number }}
  secrets: inherit
```

使用 secrets: inherit，可以允许访问父工作流中的所有密钥，而无需将它们全部指定为密钥参数。

11. 提交并推送您的文件，并将 delete-issue 标签应用于您用来测试库创建的问题。批准部署，库将删除。

请注意，可重用工作流中作业是如何嵌套在工作流作业部分中的，以及它们在设计器中的显示方式（请参阅图 5.15）。

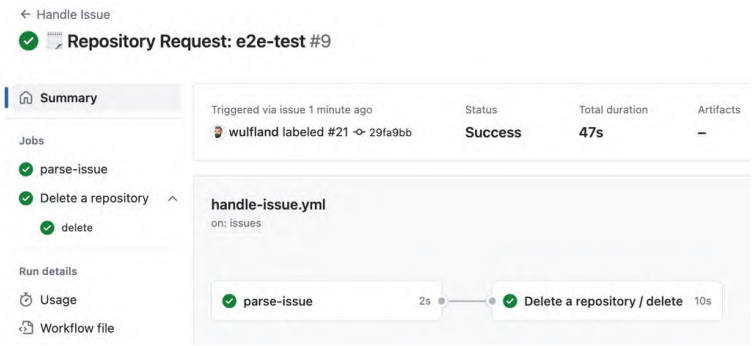


图 5.15 – 可重用工作流中的嵌套作业

此外，在删除成功评论之后，会关闭 issue。

5.5.3 How it works…

可重用工作流是结构化复杂工作流，和重用依赖多个作业或环境的更复杂功能的绝佳方式，但也存在一些限制：

- 最多可以连接 4 层嵌套的工作流。
- 单个工作流文件中最多可以调用 20 个可重用工作流，包括从顶层调用者开始的所有嵌套工作流。
- 在调用者工作流中于 env 上下文中定义的环境变量不会传递到调用的工作流中。
- 同样地，在被调用工作流中定义的 env 环境变量也无法在调用者工作流中访问。

在多个工作流中重用变量，请在组织、库或环境级别设置它们，并使用 vars 上下文引用它们。

要了解有关可重用工作流的更多信息，请参阅：https://docs.github.com/en/actions/using-workflows/reusing-workflows。

注意，我没有在删除版本库后更新项目字段。GitHub Projects 也支持可用于在问题关闭时更新字段的作业。

只需导航到项目中的 Workflows(作业)，启用 Item closed(项目关闭)，并将状态字段的值设置为所需值（请参阅图 5.16）。

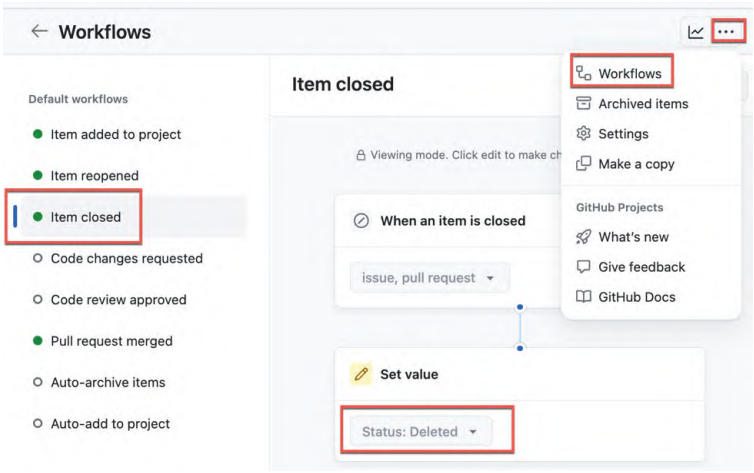


图 5.16 – 在 GitHub Projects 中配置作业

当问题关闭时，这将使状态在我们的案例中自动设置为 Deleted(已删除)。

5.5.4 There’s more…

当然，这只是一个起点。要在实际场景中采用此方案，需要将解决方案扩展到以下方面：

- 团队的生命周期
- 授予团队权限
- 为版本库设置基本配置
- 为不同解决方案类型提供不同的模板

但所使用的核心构建模块是相同的。

还需要重构当前解决方案，将其创建版本库的功能包含为一个可重用工作流，并将其包含在 handle-issue 工作流中。我在库中将其进行了保留，以尽可能降低各个步骤的复杂性，同时提供一个实际的解决方案。

第 6 章 构建并验证代码

在本章中，将介绍如何使用 GitHub Actions 来构建和验证代码、验证拉取请求中的更改，以及保持依赖项更新。还将介绍如何存储构建输出，以及如何通过缓存来优化工作流的运行。

主要内容有：

- 构建和测试代码
- 使用矩阵构建不同版本
- 通知用户有关构建和测试结果的详细信息
- 使用 CodeQL 查找安全漏洞
- 创建发布并发布包
- 版本化包
- 生成和使用软件物料清单（SBOM）
- 工作流中使用缓存

6.1. 环境要求

对于本章，需要最新版本的 NodeJS 和 Visual Studio Code，也可以使用 GitHub Codespaces。

6.2. 构建和测试代码

在示例方中，将创建一个简单的持续集成（CI）流水，该流水构建和验证代码，并集成到拉取请求验证中。我们将在后续配方中使用此代码——这也是使用一个非常简单的 JavaScript 包的原因。

6.2.1 Getting ready

我认为最好从头开始构建一个新库和 npm 包，即使不熟悉 JavaScript，也没什么问题，也可以直接使用该库：https://github.com/wulfland/package-recipe。

1. 创建一个名为 package-recipe 的新库。使用.gitignore 文件和 README 文件初始化，并选择 Node 作为.gitignore 文件的模板。
2. 将库克隆到本地，或使用 Codespaces 打开它。
3. 运行以下命令：

```
$ npm init
```

按照向导的提示操作。包的名称为 @<github-user-name>/packagerecipe。将测试命令添加为以下代码：

```
jest && make-coverage-badge
```

Jest 是我们将用来测试应用程序的测试框架，而 make-coverage-badge 将用于稍后创建一个徽章，并将其添加到 README 文件中。
可以将其余属性保留为默认值。

4. 完成后，运行以下命令：

```
$ npm install
```

5. 安装依赖项：

```
$ npm install --save-dev jest
$ npm install --save-dev make-coverage-badge
```

6. 接下来，让我们添加代码。创建一个新的 src/index.js 文件并添加以下行：

```
module.exports = function greet () {
  return 'Hello world!'
}
```

我们正在创建一个简单的包，只会返回 Hello world!。

7. 添加一个新的 __tests__/index.test.js 文件（注意双下划线）。添加以下测试：

```
describe('index.js', () => {
  it('greet function returns Hello world!', () => {
    const greet = require('../src/index')
    expect(greet()).toBe('Hello world!')
  })
})
```

这只是一个简单的测试，用于检查 index.js 是否按预期将文本写入控制台。

8. 打开 package.json 文件，并添加以下 Jest 的配置：

```
"jest": {
  "verbose": true,
  "coverageReporters": [
    "json-summary",
    [
      "text",
      {
        "file": "coverage.txt",
        "path": "./coverage"
      }
    ],
    "lcov"
  ],
```

```
  "collectCoverage": true,
  "collectCoverageFrom": [
    "./src/**"
  ]
},
```

这将添加多个代码覆盖率报告输出。

9. 执行测试：

```
$ npm run test
```

如果一切设置正确，这将执行一个测试并在 coverage 文件夹中写入报告，还会在 coverage/badge.svg 处生成一个徽章。

10. 提交文件，并将它们推送到 GitHub。
11. 在库中，转到 **Settings**（设置）并向下滚动到 **Pull Requests**（拉取请求）。检查 **Allow auto-merge**（允许自动合并）和 **Automatically delete head branches**（自动删除头分支）。

6.2.2 How to do it…

现在，已经准备好了一个包库，我们将用于下一个示例，开始添加 CI 工作流：

1. 创建一个新的_github/workflows/ci.yml 文件。
将工作流命名为 CI，并在每次拉取请求时触发以验证更改。同时，在每次推送到 main 分支时触发，以构建一个可以发布的新版本：

```
name: CI

on:
  pull_request:
  push:
    branches:
      - main
```

2. 添加一个构建作业，该作业在 ubuntu-latest 上运行并检出到库：

```
jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4
```

3. 配置工作流运行器以针对特定的 NodeJS 版本。我们可以为次要版本使用通配符，并让操作使用最新可用版本：

```
- uses: actions/setup-node@v4
  with:
    node-version: "21.x"
    check-latest: true
```

4. 构建并测试代码：

```
- name: Install dependencies
  run: npm install

- name: Run tests
  run: npm test
```

5. 提交并推送工作流到 main 分支。由于推送触发器，这应该已经触发了工作流，并且构建和测试应该成功。
6. 为了测试验证，创建一个新分支：

```
$ git switch -c fail-pr
```

7. 修改 index.js 以执行会使测试失败的更改（例如，console.log('Hello Mars!');）。添加并提交更改，然后创建一个拉取请求：

```
$ git add index.js
$ git commit
$ git push -u origin fail-pr
$ gh pr create --fill
```

拉取请求将触发工作流，并且由于 npm run test 命令将返回一个非零值，工作流将失败。

6.2.3 How it works…

再来了解如何进行验证工作的。

检出库

CI 的第一步是使用 checkout 操作（https://github.com/actions/checkout）检出到库。该操作有许多参数。例如，可以设置要拉取到运行器的 Git 记录深度。默认值为 1，这只会下载 HEAD 分支。在从 Git 生成版本号的示例中，我们将设置为 0 以下载所有分支和标签：

```
steps:
  - uses: actions/checkout@v4
    with:
      fetch-depth:0 # Default: 1
```

其他选项包括 lfs，用于确定是否应下载 Git 大文件存储（LFS）文件（默认为 false），或 submodules：

```
steps:
  - uses: actions/checkout@v4
    with:
      lfs: true # Default: false
      submodules: true # Default: false
```

对于大型单库，还可以执行稀疏检出，并仅获取库特定区域的数据。此示例仅检出.github、src 和 __tests__ 文件夹：

```
steps:
  - uses: actions/checkout@v4
    with:
      sparse-checkout: |
        .github
        src
        __tests__
```

这可以极大地减少工作流所需的时间和存储空间。

设置环境

可为不同语言进行设置操作，包括：

- Node
- Python
- Java
- Go
- .NET
- Ruby
- Elixir
- Haskell

这里，我们使用 setup-node（https://github.com/actions/setup-node）。确保设置操作在构建过程中，设置了正确的二进制文件和环境变量。所有这些操作都接受某种形式的版本参数：

```
- uses: actions/setup-node@v4
  with:
    node-version: 21
```

通配符和别名也受支持。示例包括 21.x、21.5.0、>=21.5.0、lts/Hydrogen、21-nightly、latest。如果您使用通配符，则可以设置 check-latest: true 以检查可用的最新版本。

大多数 checkout 操作还支持不同的注册表以下载依赖项。对于 node，参数是 registry-url，将使用提供的 URL 和来自 env.NODE_AUTH_TOKEN 的令牌连接到特定注册表。

Checkout 操作通常还会缓存依赖项。在许多情况下，使用相应的设置操作比自己实现更有效。

6.2.4 There's more…

验证工作流最好与分支保护和规则集结合使用——还可以添加一个检查工具来进一步提高质量。

超级代码检查工具

在第 2 章中，我们使用了一个检查工具来验证和注释拉取请求中的工作流。同样，也可以用于代码。有一个名为 superlinter(https://github.com/super-linter/super-linter) 的 GitHub 操作，它基本上将所有可用的检查工具合并到一个操作中。可以这样使用它来检查代码：

```
permissions:
  contents: read
  packages: read
  # To report GitHub Actions status checks
  statuses: write

steps:
  - name: Checkout code
    uses: actions/checkout@v4
    with:
    # super-linter needs the full git history to get the
    # list of files that changed across commits
    fetch-depth: 0

  - name: Super-linter
    uses: super-linter/super-linter@v5.7.2
    env:
      DEFAULT_BRANCH: main
      # To report GitHub Actions status checks
      GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

super-linter 的镜像相当大，可能会影响工作流的性能。如果不需要所有语言，还可以使用精简版本：

```
super-linter/super-linter/slim@[VERSION]
```

排除了 rust、dotenv、armttk、pwsh 和 C# 的 linter，并且镜像大小要小得多。在 v5 中，这将使大小从 7.35 GB 减少到 slim-v5 的 4.88 GB，并且将下载镜像的时间从大约 2 分钟减少到大约 1 分钟。

分支保护

在第 2 章中，我也介绍了分支保护。可以使用规则保护库中的一个或多个分支，并强制要求提交必须通过拉取请求合并，并且某些检查必须成功。除了手动审核外，还可以要求状态检查成功（见图 6.1）：

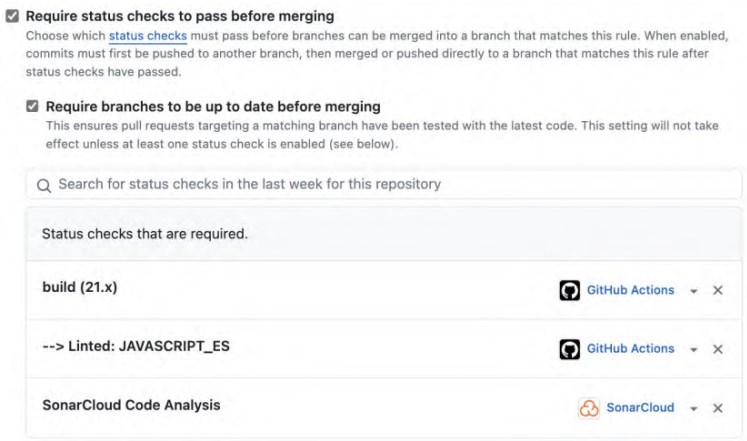


图 6.1 — 将 GitHub Action 工作流作业添加为受保护分支的状态检查

状态检查可以是一个工作流中的单个作业——或者是使用状态 API 报告状态的集成。可以为下一个示例在库中为 main 启用策略，并将状态检查策略设置为工作流的作业。有关受保护分支的更多信息，请参阅 https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/managing-protected-branches/about-protected-branches。

规则集

规则集是分支保护的更强大的继任者，具有与受保护分支相同的功能。还可以像在受保护分支中一样定义状态检查（见图 6.2）

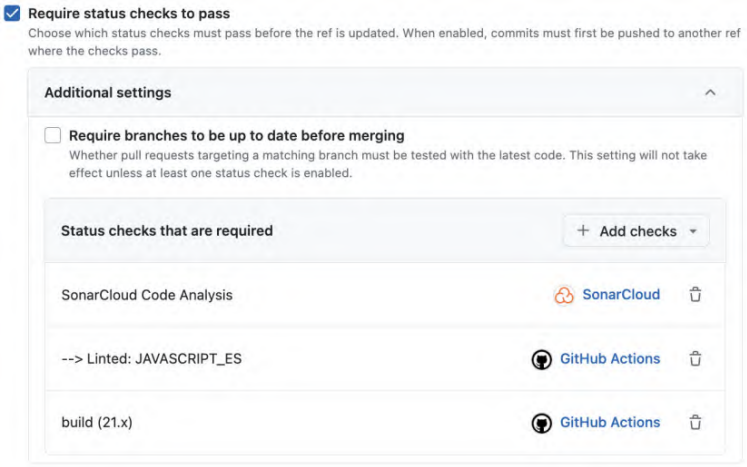


图 6.2 — 在规则集中添加 GitHub Action 工作流作业作为状态检查

规则集可以使用的大多数规则与分支保护规则相似，并且可以组合使用而无需更改现有规则。规则集比保护规则更高级，具有以下优势：

- 规则集可以在组织级别定义，并且可以针对多个库。
- 可以通过同时应用多个规则集来分层规则。
- 规则集具有状态，允许在不删除的情况下在库中激活或停用规则集。
- 只需要库的读取权限即可查看所有活动的规则集，可使得贡献者可以更清楚地了解适用的规则。
- 有一些规则在保护规则中不可用，例如：控制提交消息或作者电子邮件地址的规则。

可以在这里了解更多关于规则集的信息：https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/managing-rulesets/about-rulesets。

6.3. 使用矩阵构建不同版本

在本示例中，将为不同的版本构建和测试软件。在我们的示例中，使用的是 NodeJS 环境。

6.3.1 Getting ready

确保已从上一个示例中克隆了库，并创建一个新分支来修改工作流：

```
$ git switch -c build-matrix
```

在编辑器中打开.github/workflows/ci.yml 文件。

6.3.2 How to do it…

1. 将以下代码添加到工作流文件中：

```
strategy:
  matrix:
    node-version: ["21.x", "20.x"]
```

根据需要调整版本。

2. 在 actions/setup-node 操作中，将 node 版本设置为矩阵上下文中的对应值：

```
- uses: actions/setup-node@v4
  with:
    node-version: ${{ matrix.node-version }}
    check-latest: true
```

3. 提交并推送更改，并创建一个拉取请求：

```
$ git add .
$ git commit
$ git push -u origin build-matrix
$ gh pr create --fill
```

4. 检查工作流的输出，其将为矩阵数组中的每个条目运行一个单独的作业（见图 6.3）：

图 6.3 – 矩阵为每个条目运行一个不同的作业

5. 等待工作流完成，合并拉取请求并清理库：

```
$ gh pr merge -m
```

6.3.3 How it works…

矩阵是一种方便的方法，可以使用不同的组合来使用相同的工作流作业。可以包含一个或多个数组，这些数组可以包含许多值。矩阵将运行所有数组中所有值的所有组合，可以将矩阵视为嵌套的 `for` 循环。一个很好的例子是在不同平台上运行和测试不同版本：

```
jobs:
  example_matrix:
    strategy:
      matrix:
        os: [ubuntu-22.04, ubuntu-20.04]
        version: [10, 12, 14]
    runs-on: ${{ matrix.os }}
    steps:
      - uses: actions/setup-node@v3
      with:
        node-version: ${{ matrix.version }}
```

这样，可以重用相同的工作流逻辑，同时测试许多不同的值组合。

6.3.4 There’s more…

矩阵有一些功能。可以设置 `fail-fast` 来指示如果矩阵中的一个作业失败，是否会取消工作流，或者应该继续。可以使用 `max-parallel` 定义并行作业的数量，并且可以为某些元素包含和排除值。这是一个更加复杂的示例：

```
jobs:
  test:
    runs-on: ubuntu-latest
    continue-on-error: ${{ matrix.experimental }}
    strategy:
      fail-fast: true
      max-parallel: 2
      matrix:
        version: [5, 6, 7, 8]
        experimental: [false]
        include:
          - version: 9
            xperimental: true
```

要了解更多关于矩阵策略的信息，请访问：https://docs.github.com/en/actions/running-jobs/using-a-matrix-for-your-jobs。

6.4. 通知用户有关构建和测试结果的详细信息

本示例中，我们将使用测试结果的详细信息来装饰拉取请求和工作流摘要，还将向 README 文件添加徽章，以指示分支或发布的质量。

6.4.1 Getting ready

创建一个新分支来进行修改：

```
$ git switch -c add-badges
```

6.4.2 How to do it…

可以使用以下 URL 为工作流下载徽章：

```
https://github.com/OWNER/REPO/actions/workflows/FILE.yml/badge.svg
```

还可以通过添加查询参数（例如，?branch=main 或?event=push）按分支或事件进行过滤。我们想要一个适用于 main 分支的徽章，请将以下图像添加到您的 README 的 Markdown 中：

```
![main](https://github.com/OWNER/package-recipe/actions/workflows/ i.yml/badge.svg?branch=main)
```

徽章将使用工作流文件中的名称，并在预览中看起来像图 6.4：

图 6.4 – CI 工作流的徽章

但我们想更进一步，向 README 添加一个代码覆盖率徽章，用代码覆盖率输出来注释拉取请求，并向工作流添加摘要。

1. 要装饰拉取请求，工作流需要写入权限。由于使用矩阵构建多个版本，必须选择一个用于徽章创建的版本。将以下代码添加到构建作业中：

```
jobs:
  build:
    permissions:
      pull-requests: write
    env:
      MAIN_VERSION: "21.x"
```

2. 在工作流文件中，在 run: npm test 之后添加以下代码：

```
- name: Prepare coverage report in markdown
  uses: fingerprintjs/action-coverage-report-md@v1.0.6
  id: coverage
  with:
    textReportPath: coverage/coverage.txt"
```

使用 fingerprintjs/action-coverage-report-md 操作创建一个 Markdown 格式的报告，把该报告写入作业摘要和拉取请求。

3. 接下来，使用 marocchino/sticky-pull-request-comment 操作将 Markdown 报告写入拉取请求：

```
- name: Add coverage comment to the PR
  uses: marocchino/sticky-pull-request-comment@v2.8.0
  with:
    message: ${{ steps.coverage.outputs.markdownReport }}
```

4. 同时，将输出写入 $GITHUB_STEP_SUMMARY。由于在矩阵中运行作业，请添加一个标头以指示版本号：

```
- name: Add coverage report to the job summary
  run: |
    echo "## Code Coverage v${{ matrix.node-version }}" >> "$GITHUB_STEP_SUMMARY"
    echo "${{ steps.coverage.outputs.markdownReport }}" >> "$GITHUB_STEP_SUMMARY"
```

5. 提交并推送您的更改，然后创建一个拉取请求：

```
$ git add .
$ git commit
$ git push -u origin add-badges
$ gh pr create -fill
```

这将使用拉取请求触发器触发工作流，可以在工作流运行中检查作业摘要，应该看起来像图 6.5：

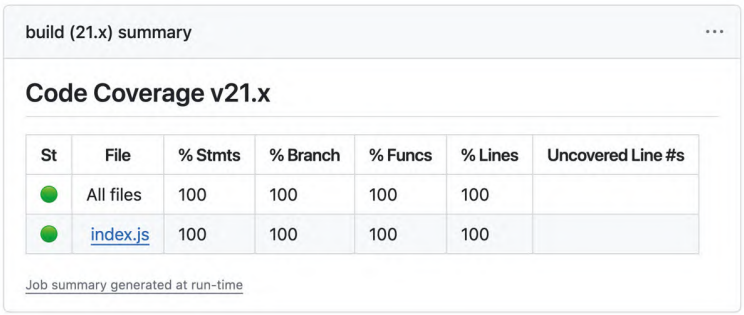


图 6.5 – 工作流摘要中显示代码覆盖率

摘要也添加为拉取请求的注释（见图 6.6）：

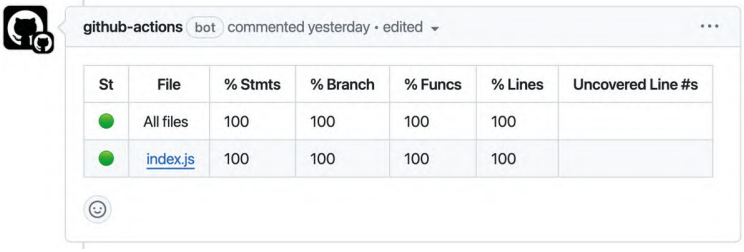


图 6.6 – 将代码覆盖率结果添加为拉取请求注释

6. 代码覆盖率徽章是在测试期间由 npm 包自动创建的。但为了添加徽章（即添加到 README），需要一个托管它的地方，为此使用 **GitHub Pages**。
   将以下代码添加到构建作业的末尾，以将工件上传到工作流。请注意，这不是普通的 actions/upload-artifact 操作——是一个用于 GitHub Pages 的特殊操作：

```
- name: Upload page artifacts
  if: ${{ matrix.node-version == env.MAIN_VERSION }}
  uses: actions/upload-pages-artifact@v3
  with:
    path: coverage
```

   由于我们只能上传一个同名工件，因此只在矩阵的 node 版本是主版本时才运行此步骤。
7. 在构建之后添加一个新的部署作业，该作业仅在推送到 main 时运行，并且依赖于构建作业：

```
deploy:
  if: ${{ github.ref == 'refs/heads/main' }}
  needs: build
  runs-on: ubuntu-latest
```

8. 创建一个名为 pages 的并发组，以便一次只将一个版本部署到 pages 环境。但不要取消正在进行的部署以部署所有版本：

```
concurrency:
  group: "pages"
  cancel-in-progress: false
```

9. 该作业需要以下权限：

```
permissions:
  contents: read
  pages: write
  id-token: write
```

10. 该作业部署到 github-pages 环境，并使用 actions/deploy-pages 操作的 URL 在工作流中显示它。该 URL 将指向包的根目录。由于我们的 index.html 文件所在的 HTML 报告在 lcov-report 文件夹中，需要将其添加到 URL 中：

```
environment:
  name: github-pages
  url: "${{ steps.deployment.outputs.page_url }}lcov-report"
```

11. 该作业只有两个简单的步骤——configure-pages 和 deploy-pages：

```
steps:
  - name: Setup Pages
    uses: actions/configure-pages@v4
  - name: Deploy to GitHub Pages
    id: deployment
    uses: actions/deploy-pages@v4
```

12. 提交并推送更改，在工作流完成后合并拉取请求：

```
$ git add .
$ git commit
$ git push
$ gh merge -m
```

13. 拉取请求合并后，一个新的工作流运行将由推送到 main 触发。完成后，可以看到包含报告的网站 URL(见图 6.7)：

图 6.7 – 检查 Pages 网站

14. GitHub 创建的网站在目录根目录中包含徽章。URL 看起来像这样：

https://{OWNER}.github.io/package-recipe/badge.svg。Markdown 中的徽章 URL 将看起来像这样：

```
[![Coverage](https://wulfland.github.io/package-recipe/badge.svg)]
```

我们希望点击徽章时可定向到网站中的 lcov-report 文件夹，需要为图像添加一个链接。将以下 Markdown 添加到 README(将 wulfland 替换为相应的 GitHub 用户名)：

```
[![Coverage](https://wulfland.github.io/package-recipe/badge.svg)]
↪ (https://wulfland.github.io/package-recipe/lcov-report)
```

徽章将如图 6.8 所示，当点击它时，将重定向到覆盖率报告：

图 6.8 – 添加测试覆盖率徽章

6.4.3 How it works…

来了解一下代码是如何工作的。

添加工作流状态徽章

可以在库中显示工作流的状态徽章，以指示工作流的状况。

```
https://github.com/OWNER/REPOSITORY/actions/workflows/WORKFLOW-FILE/badge.svg
```

通常，可显示特定分支或事件的状态，可以通过提供相应的参数（例如?branch=main 或?event=push）来实现。这样，就可以为软件的不同版本显示多个徽章。有关更多信息，请参阅 https://docs.github.com/en/actions/monitoring-and-troubleshooting-workflows/adding-aworkflow-status-badge。

GitHub Pages

GitHub Pages 是 GitHub 提供的一个静态网站托管服务，从库中获取静态文件并将其发布为网站。该网站托管在 GitHub 的 github.io 域名上，或者可以使用自己的自定义域名。

除非使用自定义域名，否则项目网站可通过以下 URL 访问：

- http(s)://<username>.github.io/<repository> 或
- http(s)://<organization>.github.io/<repository>

页面可以直接从分支部署，并且可以选择使用 Jekyll(https://github.com/jekyll/jekyll）进行预处理。这样，可以轻松地将 Markdown 文件渲染为网站——例如，用于托管博客。

可以在 https://wulfland.github.io/AccelerateDevOps/ 找到一个示例，其渲染了 https://github.com/wulfland/AccelerateDevOps/tree/main/docs 文件夹的内容。

除了从分支部署外，还可以使用自己的工作流来部署页面，这就是我们在本示例中所做的。在撰写本文时，该功能仍处于测试阶段——但在我看来，它已经可以在生产环境中使用了。

要了解更多关于 GitHub Pages 的信息，请访问：https://docs.github.com/en/pages/getting-started-with-github-pages/about-github-pages。

并发组

默认情况下，GitHub Actions 允许同一工作流中的多个作业，以及同一库中的多个工作流运行并发执行——多个步骤可以同时运行。

GitHub Actions 还可以控制工作流运行的并发性，以便确保在特定上下文中一次只运行一个运行、一个作业或一个步骤，这适合于控制同时运行多个步骤可能引起冲突，或消耗比预期更多操作时间的情况。

并发组可以有一个静态名称，但也可以使用上下文表达式将某些上下文（例如：分支）分到一个组中：

```
concurrency:
  group: ${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: true
```

cancel-in-progress 参数可用于在可用新版本时取消作业并运行该版本，有助于节省资源。如果将其设置为 false，工作流将等待前一个版本完成，然后运行并发队列中的下一个作业。这样，对于给定的组，一次只执行一个作业。

可以在这里了解有关并发组的更多信息：https://docs.github.com/en/actions/running-jobs/using-concurrency。

6.4.4 There’s more…

自动验证代码并将详细信息带到对开发人员有价值的地方非常重要。除了测试结果、代码覆盖率和检查之外，还可以集成 SonarQube 或 SonarCloud 等解决方案。它们提供了 GitHub 集成，并且对开源项目免费。只需使用 GitHub 凭据登录，就可以配置一个新项目。该项目可以创建徽章，可以将其添加到 README 中（见图 6.9）：

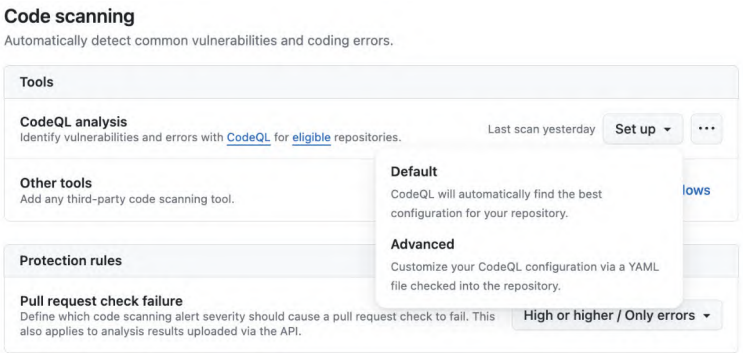


图 6.9 – 在 GitHub README 中使用 SonarCloud 徽章

Sonar 徽章提供了与拉取请求的集成，并且可以用作规则集和分支保护规则中的状态检查。质量门在拉取请求中报告（见图 6.10）：

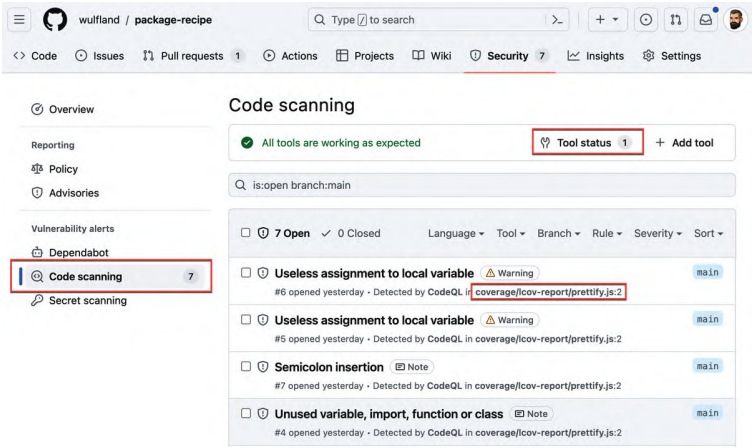


图 6.10 – SonarCloud 质量门在拉取请求中的集成

要了解有关 SonarCloud 集成 GitHub 的更多信息，请参阅https://docs.sonarsource.com/sonarcloud/getting-started/github/。

6.5. 使用 CodeQL 查找安全漏洞

这是一个简短的示例，我们将使用它来将 CodeQL 分析添加到现有的 CI 构建中。CodeQL 是 GitHub 的代码分析引擎，并且对公共库免费。

6.5.1 Getting ready

在 package-recipe 库中创建一个新分支：

```
$ git switch -c add-codeql
```

6.5.2 How to do it…

1. 打开.github/workflows/ci.yml 文件，并授予 build 作业写入安全事件的权限：

```
build:
  permissions:
    pull-requests: write
    security-events: write
```

2. 将一个 init 操作（github/codeql-action/init）添加到作业中。对于必须编译的语言，这必须放在构建过程之前。由于 JavaScript 是静态语言，可以将其添加到作业的末尾。将语言设置为 javascript-typescript，并选择 security-and-quality 查询套件：

```
    - name: Initialize CodeQL
      uses: github/codeql-action/init@v3
      with:
        languages: 'javascript-typescript'
        queries: security-and-quality
```

3．执行实际分析是在编译之后进行的。只需将以下代码添加到作业的末尾：

```
    - name: Perform CodeQL Analysis
      uses: github/codeql-action/analyze@v3
      with:
        category: "/language:javascript-typescript"
```

4．提交代码，创建一个拉取请求，并在所有检查通过后合并更改：

```
$ git add .
$ git commit
$ gh pr create --fill
$ gh pr merge -m --auto
```

6.5.3 How it works…

CodeQL 是 GitHub 的代码分析引擎，用于自动化安全和质量检查，在公共库中免费，并且对购买了 GitHub Advanced Security 许可证的企业可用。

可以分析以下语言：

- C/C++
- C#
- Go
- Java
- Kotlin(在撰写本文时处于测试阶段)
- JavaScript/TypeScript
- Python
- Ruby
- Swift(在撰写本文时处于测试阶段)

有三种方式可以分析代码：

- 默认：CodeQL 的默认设置将自动检测存中的语言，并选择支持的语言进行分析。它将应用默认的查询套件和扫描触发器。在 **Settings | Code security and analysis**(设置 | 代码安全和分析) 下，如果库中的语言受该功能支持，可以选择 **Default**(默认)(见图 6.11)：

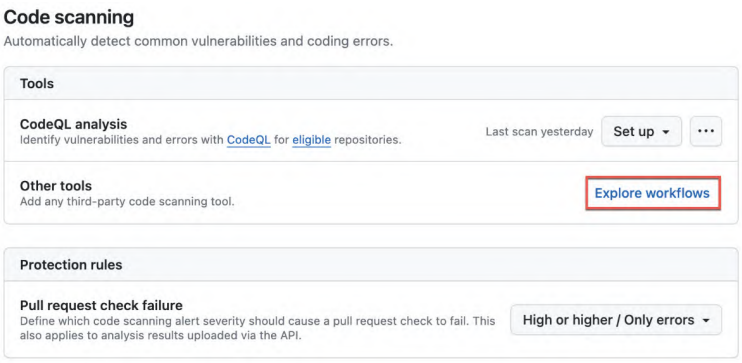


图 6.11 – 使用默认或高级模式配置代码扫描

- 高级：使用自定义工作流，并像本示例中所做的那样添加 CodeQL 分析。如果在设置（图 6.11）中点击高级,这将生成一个可自定义的工作流模板。将使用矩阵策略,并将 fail-fast 设置为 false，以分析库中检测到的所有语言。
- CLI：在外部 CI 系统中直接运行 CodeQL CLI，并将结果上传到 GitHub。

所有工具的分析结果可以在 **Security | Code scanning**（安全 | 代码扫描）下访问（见图 6.12）：

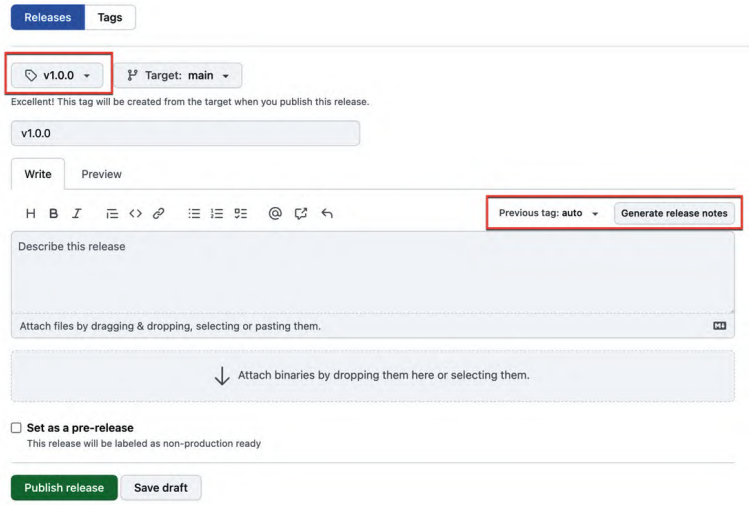


图 6.12 – 在库中查看代码分析结果

可以访问报告结果的工具的所有状态。请注意，在我们的库中，还分析了测试包生成的代码。要了解有关 CodeQL 的更多信息，请访问https://docs.github.com/en/code-security/code-scanning/introduction-to-code-scanning/about-code-scanning-with-codeql。

6.5.4 There’s more…

GitHub 支持各种其他工具——其中包括 Checkmarx、Snyk、Microsoft Defender for DevOps 和 ESLint。支持许多商业工具，也支持许多免费或开源的工具。

如果在 **Settings | Code security and analysis**（设置 | 代码安全和分析）下点击 **Explore workflows**（探索工作流）（见图 6.13），将重定向到一个新的工作流页面，该页面

按安全类别和代码扫描查询进行筛选（https://github.com/OWNER/REPO/actions/new?category=security&query=code+scanning）：

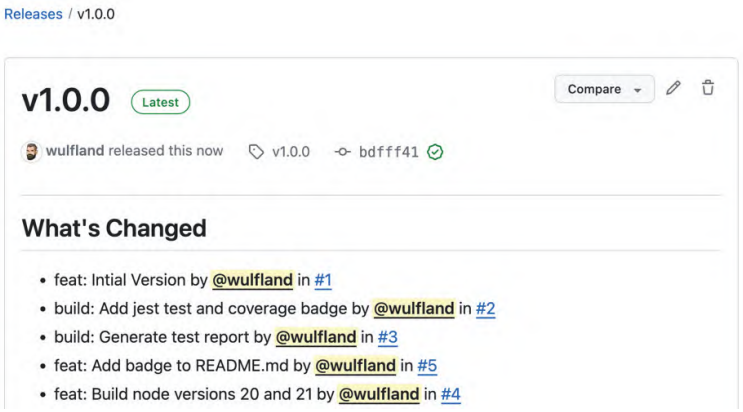


图 6.13 – 将第三方扫描工具添加到库中

可以探索所有已在市场上发布模板工作流的合作伙伴。

如果您有其他工具，只要它们支持静态分析结果交换格式（SARIF；请参阅 https://sarifweb.azurewebsites.net/）作为输出格式，可以对其进行集成。SARIF 是结构化信息标准促进组织（OASIS）批准的静态代码分析标准。

举个例子，可以使用 Checkov，这是一个支持 Terraform、Terraform plan、CloudFormation、AWS Serverless Application Model（SAM）、Kubernetes、Helm charts、Kustomize、Dockerfiles、Serverless、Bicep、OpenAPI 或 ARM 模板的 IaC（基础设施即代码）静态代码分析工具。使用 bridgecrewio/checkov-action 操作来运行分析，然后将结果上传到 GitHub：

```
- name: Checkov GitHub Action
  uses: bridgecrewio/checkov-action@v12
  with:
    output_format: sarif

- name: Upload SARIF file
  uses: github/codeql-action/upload-sarif@v3
  with:
    sarif_file: results.sarif
  if: always()
```

这样一来，只要它们支持 SARIF，就可以轻松集成市场上可用的代码扫描工具了。

6.6. 创建发布并发布软件包

本示例中，我们将创建一个工作流，以便在创建发布时发布软件包。

6.6.1 Getting ready

创建一个新的分支：

```
$ git switch -c add-release-workflow
```

6.6.2 How to do it…

1. 创建一个新的.github/workflows/release.yml 工作流文件。将工作流命名为 Release，并在创建 GitHub 发布时触发：

```
name: Release

on:
  release:
    types: [created]
```

2. 添加一个 publish 作业，并为其授予对库的读取权限和对软件包的写入权限：

```
jobs:
  publish:
    runs-on: ubuntu-latest
    permissions:
      packages: write
      contents: read
```

3. 检出代码，并使用正确的版本配置 NodeJS 环境，并将注册表设置为使用 GitHub：

```
steps:
  - uses: actions/checkout@v4

  - uses: actions/setup-node@v4
    with:
      node-version: 21.x
      registry-url: https://npm.pkg.github.com/
```

4. 构建并测试应用程序，并使用 GitHub 令牌将其发布到注册表：

```
- name: Install dependencies
  run: npm install

- name: Run tests
  run: npm test

- run: npm publish
  env:
```

```
    NODE_AUTH_TOKEN: ${{secrets.GITHUB_TOKEN}}
```

5. 提交更改，创建一个拉取请求，并在检查通过后合并更改：

```
$ git add .
$ git commit
$ gh pr create --fill
$ gh pr merge -m --auto
```

6. 拉取请求合并后，转到库的 **Code**（代码）选项卡下的 **Releases**（发布）部分，然后点击 **Draft a new Release**（草拟新发布）。
7. 点击 **Choose a tag**（选择标签），输入 v1.0.0，然后点击 **Create new tag**（创建新标签）。不要以 Markdown 格式添加发布说明正文，只需点击 **Generate release notes**（生成发布说明）（见图 6.14）：

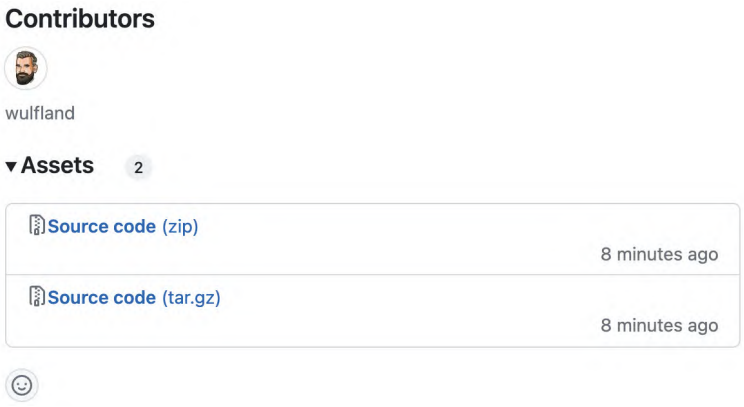


图 6.14 – 为标签草拟新发布并生成发布说明

这将根据拉取请求生成发布详情，点击 **Publish release** 来创建发布。该发布应如图 6.15 所示：

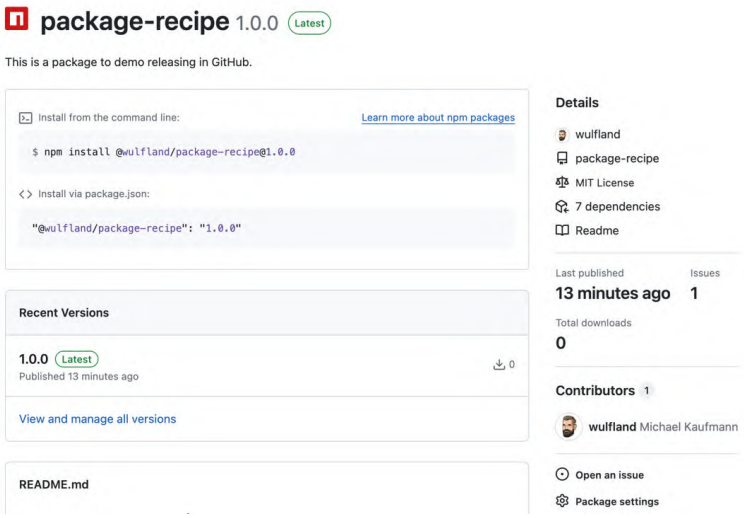


图 6.15 – GitHub 发布的详细信息

8. 默认情况下，也应包含作为.zip 和.tar 归档的源代码（见图 6.16）：

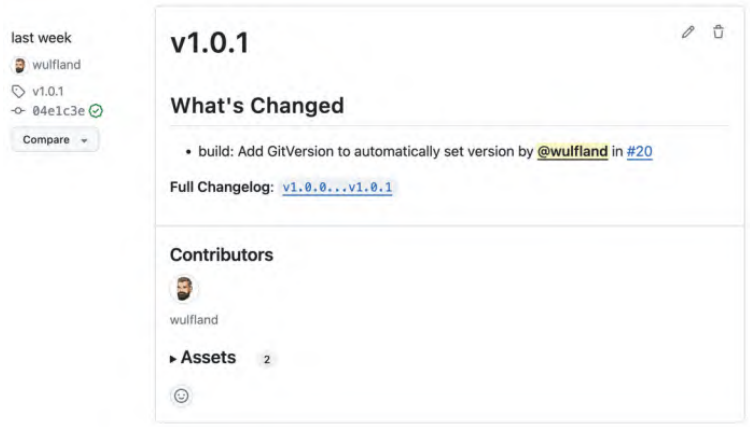


图 6.16 – 发布的资产包含作为.zip 和.tar 归档的源代码

9. 创建发布将触发工作流，并将软件包以版本 1.0.0 发布到库（请注意，这来自 package.json 文件，而非标签！）。
该软件包看起来如图 6.17 所示：

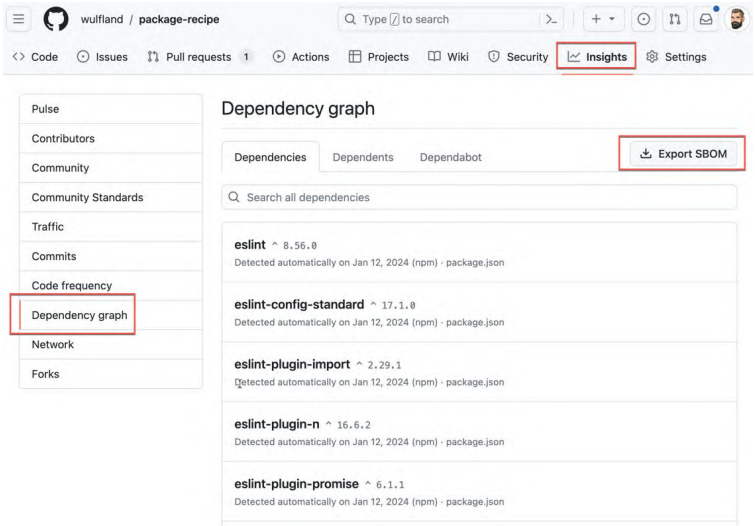


图 6.17 – 库中的软件包

包含安装说明、有关您库的信息，以及带有徽章的 README。

6.6.3 How it works…

现在，我们将看看它是如何工作的。

语义化版本控制

软件包通常使用语义化版本控制（Semantic Versioning）创建，这是一种为软件指定版本号的正式约定。它由具有不同含义的不同部分组成。语义化版本号的示例有 1.0.0 或 1.5.99-beta。其格式如下：

```
<major>.<minor>.<patch>-<pre>
```

- 主版本号（Major）：当版本不向后兼容并具有破坏性更改时，此数字标识符会增加。升级到新的主版本号必须谨慎处理！主版本号为零表示处于初始开发阶段。
- 次版本号（Minor）：当添加新功能但版本与先前版本向后兼容时，此数字标识符会增加。如果需要新功能，则可以更新而不会破坏任何内容。
- 修订版本号（Patch）：当发布向后兼容的错误修复时，此数字标识符会增加。新的修订号应始终安装。
- 预发布标识符（Pre-release）：使用连字符附加的文本标识符。标识符只能使用 ASCII 字母数字字符和连字符（[0-9A-Za-z-]）。文本越长，预发布版本越小（例如 -alpha < -beta < -rc）。预发布版本始终小于常规版本（例如 1.0.0-alpha < 1.0.0）。

有关完整规范，请参阅 https://semver.org/。

在我们的示例中，软件包的语义化版本是在 package.json 文件中设置的。在下一个示例中，将使用发布流程，来自动将软件包的版本号设置为发布版本。

发布

可以创建发布来打包软件、发布说明和供他人下载的二进制文件。

GitHub 发布是可部署的软件版本，可以将其打包并提供给更广泛的受众下载和使用，可以包含发布说明和其他可下载的二进制文件。

发布基于 Git 标签，这些标签标记了库历史中的特定点。

在接下来的示例中，将使用 GitHub 发布与标签一起，自动为软件包设置版本号，并自动添加 SBOM。有关 GitHub 发布的更多信息，请参阅 https://docs.github.com/en/repositories/releasing-projects-on-github。

6.6.4 There’s more…

软件的发布和版本控制很大程度上取决于工作流——特别是如何使用 Git 分支和标签。在示例中，我假设可以通过直接创建发布来启动发布流程。当然，也可以使用标签进行推送：

```
on:
  push:
    tags:
      - v*.**
```

该工作流将由以 v 开头的标签推送触发，可以使用它来自动创建发布：

```
gh release create ${{ github.ref_name }} --generate-notes
```

也可以在推送到发布分支时这样做：

```
on:
  push:
    branches:
      - release/*
```

6.7. 软件包的版本控制

如果创建了一个新发布，工作流将失败，因为它会尝试再次将版本 1.0.0 发布到软件包注册表。必须手动在 package.json 文件中设置版本号。

在本示例中，我们将使用 GitVersion 来自动化此过程。

6.7.1 Getting ready

切换到一个新的分支：

```
$ git switch -c add-gitversion
```

6.7.2 How to do it…

1. 为了让 GitVersion 自动确定 Git 工作流的版本，需要下载所有引用，而不仅仅是 HEAD 分支。我们通过将 fetch-depth 参数添加到 checkout 动作，并将其设置为 0 来进行实现：

```
- uses: actions/checkout@v4
  with:
    fetch-depth: 0
```

2. 以特定版本设置 GitVersion：

```
- name: Install GitVersion
  uses: gittools/actions/gitversion/setup@v0.10.2
  with:
    versionSpec: '5.x'
```

3. 执行 GitVersion 以确定版本号：

```
  - name: Determine Version
    uses: gittools/actions/gitversion/execute@v0.10.2'
```

4. 更改 npm 包的版本：

```
  - name: 'Change NPM version'
    uses: reedyuk/npm-version@1.2.2
    with:
      version: $GITVERSION_SEMVER
```

5. 提交更改，创建一个拉取请求，并在所有检查通过后进行合并：

```
$ git add .
$ git commit -m '(build): Add GitVersion to automatically set
version'
$ gh pr create --fill
$ gh pr merge -m --auto
$ gh pr checks
```

6. 等待所有检查通过，再创建发布：

```
$ gh release create v1.0.1 --generate-notes
```

将拥有一个如图 6.18 所示的新发布——这次使用 CLI 创建：

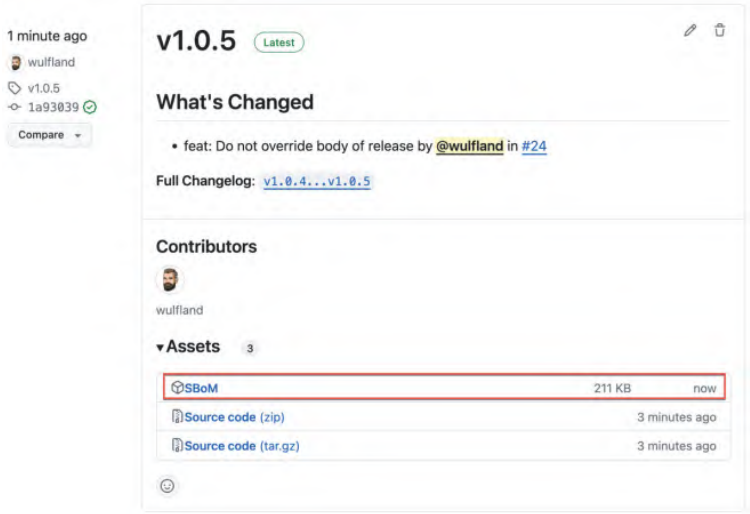


图 6.18 – 从 CLI 创建发布，这将触发工作流发布一个与发布版本相同的新软件包

该发布将触发工作流，并发布一个版本为 1.0.1 的新软件包。

6.7.3 How it works…

GitVersion(https://gitversion.net/docs/) 是一个工具，可根据您的 Git 历史记录自动生成语义化版本号。GitVersion 默认使用与 GitHub flow(https://docs.github.com/en/get-started/usinggithub/github-flow) 和 Git flow(https://nvie.com/posts/a-successfulgit-branching-model/) 兼容的配置运行。

可以运行 GitVersion init 来启动一个向导，该向导将引导您创建一个配置文件(GitVersion.yml)。在我们的示例中，使用持续交付模式——需要显式地使用标签创建一个版本。但还有其他模式，例如持续部署模式（从特定分支的每个提交创建版本）或主线模式。

gittools/actions/gitversion/execute 操作将执行 GitVersion 并将结果保存在 $GITVERSION_SEMVER 环境变量中，还可以使用该操作的输出访问版本的各个部分和配置：

```
- name: Determine Version
  id: gitversion
  uses: gittools/actions/gitversion/execute@v0
```

```
- name: Display GitVersion outputs (step output)
  run: |
    echo "Major: ${{ steps.gitversion.outputs.major }}"
    echo "Minor: ${{ steps.gitversion.outputs.minor }}"
    echo "Patch: ${{ steps.gitversion.outputs.patch }}"
```

6.7.4 There's more…

还可以使用规范提交（Conventional Commits）(https://www.conventionalcommits.org）从提交消息中自动确定语义化版本。规范提交是一种规范，提供了一组规则，用于通过描述提交消息中的特性、修复和破坏性变更来创建明确的提交历史。可以使用规范提交来创建发布说明——可以将其与 GitVersion 一起使用，以自动确定新版本是补丁、次要还是主要版本。可以通过 GitVersion.yml 配置文件完成：

```
mode: Mainline
major-version-bump-message:
"^(build|chore|ci|docs|feat|fix|perf|refactor|revert|style|test)(\\([\\w\\s-]*\\))?
↪  (!:|:.*\\n\\n((.+\\n)+\\n)?BREAKING CHANGE:\\s.+)"minor-version-bump-message:
↪  "^(feat)(\\([\\w\\s-]*\\))?:"
patch-version-bump-message:
"^(build|chore|ci|docs|fix|perf|refactor|revert|style|test)(\\([\\w\\s-]*\\))?:"
```

在 CI 构建中，可以添加一个作业，该作业从规范提交消息中确定版本，并创建一个新发布：

```
publish:
  if: ${{ github.ref == 'refs/heads/main' }}
  needs: build
  runs-on: ubuntu-latest
  permissions:
    contents: write

  steps:
    - uses: actions/checkout@v4
      with:
        fetch-depth: 0

    - name: Install GitVersion
      uses: gittools/actions/gitversion/setup@v0.10.2
      with:
        versionSpec: '5.x'

    - name: Determine Version
      uses: gittools/actions/gitversion/execute@v0.10.2
      with:
```

```
        useConfigFile: true

    - name: Create a new release
      env:
        GH_TOKEN: ${{ github.token }}
      run: |
        gh release create ${{ env.GITVERSION_SEMVER }}
--generate-notes
```

这两个工作流结合在一起，将完全自动化在每次合并到 main 分支后创建新版本的过程。

6.8. 生成和使用 SBOM(软件物料清单)

SBOM(软件物料清单)(https://www.cisa.gov/sbom) 声明了构成软件的嵌套组件清单。根据 2014 年的《网络供应链管理和国情透明法案》，美国政府被要求为其采购的任何产品获取 SBOM。

可以在 GitHub 上的 **Insights | Dependency graph | Export SBOM**(洞察 | 依赖关系图 | 导出 SBOM) 处手动导出 SBOM(见图 6.19):

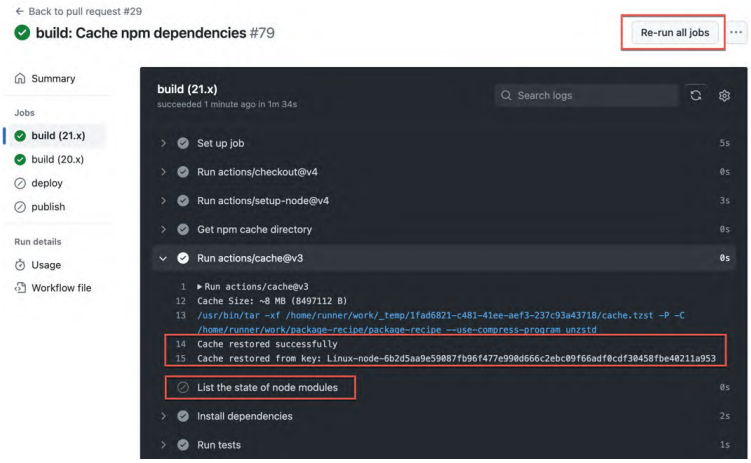


图 6.19 – 手动导出 SBOM

SBOM 是一个遵循软件包数据交换 (SPDX) 标准的 JSON 文件。

本示例中，我们将自动化从库的当前依赖项生成 SBOM 的过程，并将其作为添加到发布中。

6.8.1 Getting ready

切换到一个新分支:

```
$ git switch -c upload-sbom
```

6.8.2 How to do it…

1. 编辑.github/workflows/release.yml 文件。修改 publish 作业的权限以允许写入权限：

```
jobs:
  publish:
    runs-on: ubuntu-latest
    permissions:
      packages: write
      contents: write
```

2. 在“Publish package”步骤之后，添加以下步骤，该步骤使用 GitHub CLI 调用 API 并下载 SBOM：

```
- name: Generate SBoM
  env:
    GH_TOKEN: ${{ github.token }}
  run: |
    gh api \
      -H "Accept: application/vnd.github+json" \
      -H "X-GitHub-Api-Version: 2022-11-28" \
      /repos/wulfland/package-recipe/dependency-graph/sbom > sbom.json
```

3. 添加 svenstaro/upload-release-action 操作并将 SBOM 上传为发布的附件：

```
- name: Upload SBoM to release
  uses: svenstaro/upload-release-action@v2
  with:
    file: sbom.json
    asset_name: SBoM
    tag: ${{ github.ref }}
    overwrite: true
```

4. 提交工作流，创建一个拉取请求，并在所有检查通过后将其合并：

```
$ git add .
$ git commit
$ gh pr create --fill
$ gh pr merge -m --auto
$ gh pr checks
```

5. 等待所有检查通过，创建发布：

```
$ gh release create v1.0.5 --generate-notes
```

6. 工作流将创建一个发布并将 SBOM 附加为资产（见图 6.20）：

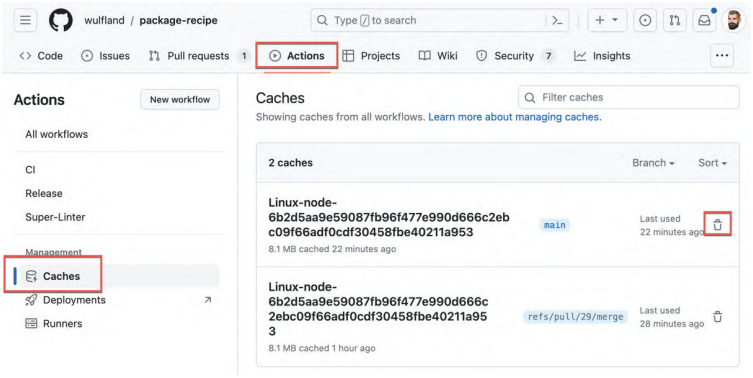


图 6.20 — 自动附加到发布中的 SBOM 资产

可以从发布中下载并检查 JSON 文件。

6.8.3 How it works…

在发布软件时，始终需要存储输出 — 有时是二进制文件，有时是文本文件。

可以使用 https://github.com/actions/ 上传工件到 GitHub 工作流程 upload-artifact 操作：

```
- uses: actions/upload-artifact@v4
  with:
    name: my-artifact
    path: path/**/[abc]rtifac?/*
```

使用此操作将测试徽章发布到 GitHub Pages。然后，可以从工作流摘要页面下载工件，或者在后续作业中使用 https://github.com/ctions/download-artifact 操作下载：

```
- uses: actions/download-artifact@v4
  with:
    name: my-artifact
```

但对于最终用户，将所有内容存储在发布中更为方便。这样，可以将所有构建输出与源代码的确切版本捆绑在一起。您可以使用 API https://docs.github.com/en/rest/releases/assets）管理发布资产：

```
gh api \
  --method POST \
  -H "Accept: application/vnd.github+json" \
  -H "X-GitHub-Api-Version: 2022-11-28" \
  --hostname HOSTNAME \
  /repos/OWNER/REPO/releases/ID/assets?name=example.zip \
  -f '@example.zip'
```

但有许多可用的操作可完成此任务。我们使用 svenstaro/upload-releaseaction 操作，也还有许多其他选项。例如，GitTools 的 GitReleaseManager(https://github.com/G

itTools/actions/blob/main/docs/examples/github/gitreleasemanager/index.md）有一整套用于与发布交互的操作。

6.8.4 There’s more…

SBOM 有不同的常见格式：

- SPDX：SPDX 是源自 Linux 基金会的 SBOM 开放标准，其起源是许可证合规性，也包含版权、安全引用和其他元数据。
  SPDX 最近被批准为 ISO/IEC 标准（ISO/IEC 5962:2021），并且它满足了国家电信和信息管理局（NTIA）的《软件物料清单的最小元素》的要求。
- CycloneDX（CDX）：CDX 是一个轻量级的开源格式，起源于开放全球应用安全项目（OWASP）社区。它优化了将 SBOM 生成集成到发布管道中。
- 软件识别（SWID）标签：SWID 是一个由各种商业软件发布者使用的 ISO/IEC 行业标准（ISO/IEC 19770-2）。支持软件库存的自动化、机器上软件漏洞的评估、缺失补丁的检测、配置清单评估的目标、软件完整性检查、安装和执行白名单/黑名单以及其他安全和操作用例。

每种格式都有不同的工具和用例。GitHub、FOSSology 和 yft 使用 SPDX。可以使用 Anchore SBOM 操作（https://github.com/marketplace/actions/anchore-sbom-action）为 Docker 或开放容器计划（OCI）容器生成 SPDX SBOM：

```
  - name: Anchore SBOM Action
    uses: anchore/sbom-action@v0.6.0
    with:
      path: .
      image: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}
      registry-username: ${{ github.actor }}
      registry-password: ${{ secrets.GITHUB_TOKEN }}
```

SBOM 正在上传工作流工件。

CDX(https://cyclonedx.org/）更专注于应用程序安全。在市场上提供了 NodeJS、.NET、Python、PHP 和 Go 的版本——但使用 CLI 或其他包管理器支持更多语言（Java、Maven、Conan 等），使用方法也很简单。

以下是.NET 操作的示例：

```
- name: CycloneDX .NET Generate SBOM
  uses: CycloneDX/gh-dotnet-generate-sbom@v1.0.1
  with:
    path: ./CycloneDX.sln
    github-bearer-token: ${{ secrets.GITHUB_TOKEN }}
```

与 Anchore 操作不同，SBOM 不会自动上传——需要手动执行此操作：

```
- name: Upload a Build Artifact
  uses: actions/upload-artifact@v2.3.1
  with:
    path: bom.xml
```

CDX 也用于 OWASP Dependency-Track(https://github.com/DependencyTrack/dependency-track) — 这是一个组件分析平台，可以将其作为容器或 Kubernetes 中运行。可以直接将 SBOM 上传到 Dependency-Track 实例中：

```
- uses: DependencyTrack/gh-upload-sbom@v1.0.0
  with:
    serverhostname: 'your-instance.org'
    apikey: ${{ secrets.DEPENDENCYTRACK_APIKEY }}
    projectname: 'Your Project Name'
    projectversion: 'main'
```

SWID 标签更多地用于软件资产管理 (SAM) 解决方案，例如 Snow(https://www.snowsoftware.com/)、Microsoft System Center 或 ServiceNow IT Operations Management(ITOM)。CDX 和 SPDX 可以在存在时使用 SWID 标签。

如果您想了解更多关于 SBOM 的信息，请访问 https://www.ntia.gov/sbom。

6.9. 工作流中使用缓存

在本示例中，我们将使用缓存来优化操作的速度。

6.9.1 Getting ready

切换到一个新分支：

```
$ git switch -c cache-npm-packages
```

6.9.2 How to do it…

1. 编辑.github/workflows/ci.yml 文件。在 setup-node 操作之后，添加以下脚本以获取正确的 npm 版本的 npm 缓存目录，并将其存储为输出变量：

   ```
   - name: Get npm cache directory
     id: npm-cache-dir
     run: echo "dir=$(npm config get cache)" >> "${GITHUB_OUTPUT}"
   ```

2. 紧接着添加实际的缓存步骤。同时，给它一个名称，并从 package-lock.json 文件的哈希值创建一个密钥：

```
- uses: actions/cache@v3
  id: npm-cache
  with:
    path: ${{ steps.npm-cache-dir.outputs.dir }}
    key: ${{ runner.os }}-node-${{ hashFiles('**/package-lock.json') }}
      restore-keys: |
        ${{ runner.os }}-node-
```

3. 下一步将仅列出依赖项，将其添加到缓存中：

```
- name: List the state of node modules
  if: ${{ steps.npm-cache.outputs.cache-hit != 'true' }}
  continue-on-error: true
  run: npm list
```

4. 提交更改并创建一个拉取请求：

```
$ git add.
$ git commit
$ gh pr create --fill
```

5. 在拉取请求中打开 CI 工作流运行，并检查添加的步骤的输出。重新运行两个作业，并注意操作如何从缓存中恢复包，以及 cache-hit 如何阻止下一步执行（见图 6.21）：

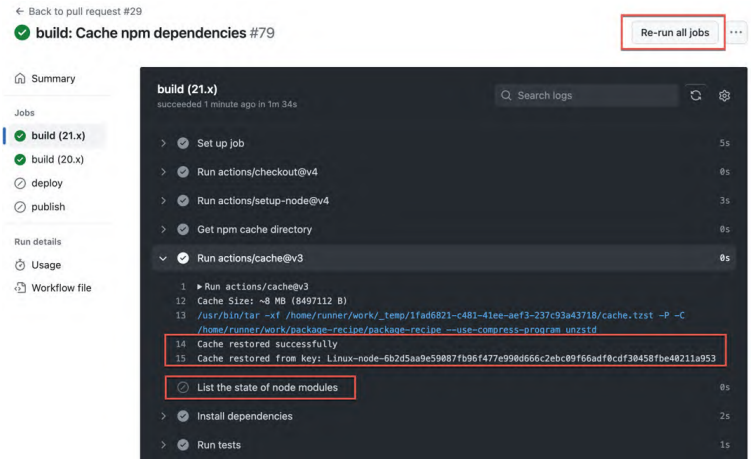


图 6.21 – 缓存成功恢复且 cache-hit 为 true

6. 合并拉取请求：

```
$ gh pr merge -s --auto
```

6.9.3 How it works…

只有在遇到性能问题时才应使用缓存。如果使用的是 setup 动作，可能不会看到显著的改进。但了解缓存的工作原理很重要，这样在需要时就可以使用该工具。

缓存操作（https://github.com/actions/cache）将信息存储在 GitHub 拥有的云存储中，并在后续运行中从该存储中检索。

假设您有一个长时间运行的操作（例如：计算质数），并且后续步骤中使用该输出。（这里我使用 sleep 来模拟长时间运行的任务）：

```
- name: Generate Prime Numbers
  run: |
    sleep 60
    echo "1 2 3..." > primes

- name: Use Prime Numbers
  run: cat primes
```

现在，可以在该步骤之前添加一个缓存步骤，并缓存 primes 文件夹：

```
- name: Cache Primes
  id: cache-primes
  uses: actions/cache@v3
  with:
    path: primes
    key: ${{ runner.os }}-primes
```

这里的键是确定缓存是否已更改所必需的唯一标识。我们的示例中，使用了 package-lock.json 文件的哈希值。对于质数，可能针对操作系统使用不同的格式。

在长时间运行的操作中，可以检查缓存是否有效，并且仅在当前键未命中时才执行：

```
- name: Generate Prime Numbers
  if: steps.cache-primes.outputs.cache-hit != 'true'
  run: |
    sleep 60
    echo "1 2 3..." > primes
```

这个工作流在第一次运行长时间操作时会执行，并将文件填充到缓存中。如果第二次运行，将从缓存中加载文件，并跳过长时间运行的步骤。

6.9.4 There’s more…

一个库最多可以存储 10 GB 的缓存数据。当达到该限制，较早的文件将根据它们上次访问的时间被移除，超过一周未被使用的缓存也会进行清理。

可以在 **Actions | Caches** 下管理缓存（见图 6.22）：

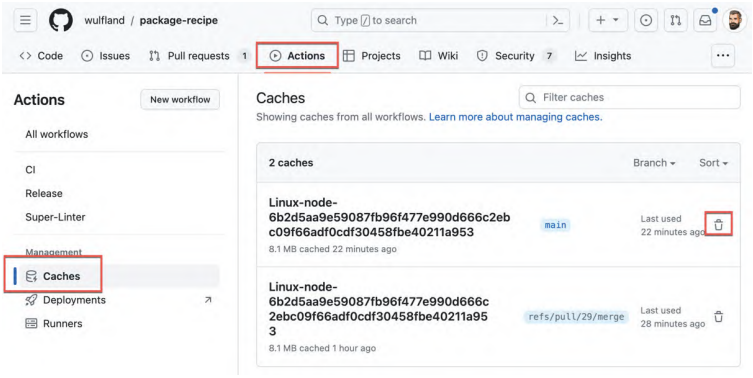


图 6.22 – 管理缓存

缓存操作（https://github.com/actions/cache）有良好的文档和针对大多数编程语言的示例。想要实现缓存操作时，也可以参考文档（https://docs.github.com/en/actions/using-workflows/caching-dependencies-to-speed-up-workflows）。

第 7 章 使用 GitHub Actions 发布软件

在本章的最后，我们将深入研究使用 GitHub Actions 进行持续部署（CD），创建一个包含简单网站的容器。该网站使用第 6 章中的软件包，并将其部署到云端的 Kubernetes 中，使用 OpenID Connect(OIDC) 来保护访问，使用环境来保护部署，并使用并发组来控制多个工作流的流程。

本章中，将使用 Microsoft Azure Kubernetes Service(AKS) 作为生产环境，也会使用其他云提供商（如 Google Cloud Platform(GCP) 上的 Google Kubernetes Engine(GKE) 或 Amazon Web Services(AWS) 上的 Elastic Container Services(ECS)）执行相同的操作。

主要内容有：

- 构建和发布容器
- 使用 OIDC(OpenID Connect) 安全地部署到云端
- 环境批准检查
- 将容器应用程序发布到 Azure Kubernetes Service(AKS)
- 自动更新依赖项

7.1. 环境要求

对于本章，需要在本地计算机上安装 Docker、Node.js 和 GitHub CLI，也可以使用 GitHub Codespaces。对于 Microsoft Azure 部分，需要一个 Azure 账户。

如果还没有，请在此处创建一个免费试用账户：https://azure.microsoft.com/en-us/free。您可以在本地使用 Azure CLI，或者直接在 Azure 门户中使用 Cloud Shell，还需要一个具有对 GitHub 软件包（https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens#creating-a-personal-access-token-classic）具有读写权限的 GitHub 个人访问令牌（PAT）。

7.2. 构建并发布容器

在本示例中，我们将容器化一个简单的 Web 应用程序，并将其推送到容器注册表。

7.2.1 Getting ready

打开库（https://github.com/wulfland/release-recipe），然后点击 **Use this template**（使用此模板）创建新的副本（直接链接：https://github.com/new?template_name=release-recipe&template_owner=wulfland）。在个人账户中创建一个新的公共库，将其命名为 release-recipe，并克隆该库。

打开 package.json 并调整作者和库 URL。

在 dependencies 下，将 package recipe 的所有者和版本调整为来自第 6 章的软件包：

```
"dependencies": {
  "@wulfland/package-recipe": "^2.0.5",
  "express": "^4.18.2"
}
```

在文件.npmrc 中替换所有者：

```
@wulfland:registry=https://npm.pkg.github.com
```

在终端中，运行以下命令：

```
$ npm login --registry https://npm.pkg.github.com
```

输入 GitHub 用户名和具有访问软件包权限的 PAT 令牌。运行以下命令：

```
$ npm install
$ npm start

> release-recipe@1.0.0 start
> node src/index.js

Server running at http://localhost:3000
```

现在，应该有一个在端口 3000 上运行的 Node.js 应用程序。打开浏览器，导航到 http://localhost:3000，并验证它是否显示 Hello World!。通过退出进程 (CTRL+C) 停止服务器。

7.2.2 How to do it…

1. 在库根目录创建一个新文件，Dockerfile。从 node 镜像继承，并选择 21-bullseye 版本。创建一个文件夹，将库内容复制到其中，并将其设置为工作文件夹：

   ```
   FROM node:21-bullseye
   RUN mkdir -p /app
   COPY . /app
   WORKDIR /app
   ```

2. 注意，需要在构建 Docker 镜像之前运行 npm install，以避免在容器中存储凭据。在容器中重建 npm 包：

   ```
   RUN npm rebuild
   ```

3. 暴露 express 网站的 3000 端口，并将 npm start 作为容器的启动命令运行：

```
EXPOSE 3000
CMD [ "npm", "start"]
```

4．接下来，本地构建容器镜像并运行：

```
$ npm install
$ docker build -t hello-world-recipe .
$ docker run -it -p 3000:3000 hello-world-recipe
```

5．再次验证网站是否在本地机器的 3000 端口上运行。

6．创建一个新的工作流文件.github/workflows/publish.yml。在拉取请求和推送到 main 分支时运行：

```
name: Publish Docker Image

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
```

7．将注册表名称和镜像名称设置为环境变量：

```
env:
  REGISTRY: 'ghcr.io'
  IMAGE_NAME: '${{ github.repository }}'
```

8．添加一个作业，为 GITHUB_TOKEN 授予写入软件包和读取内容的权限：

```
jobs:
  build-and-push-image:
    runs-on: ubuntu-latest

    permissions:
      packages: write
      contents: read
```

9．添加以下步骤。检出库并登录到 Docker 注册表：

```
- name: Checkout repository
  uses: actions/checkout@v4

- name: Log in to the Container registry
  uses: docker/login-action@v3
  with:
    registry: ${{ env.REGISTRY }}
```

```
    username: ${{ github.actor }}
    password: ${{ secrets.GITHUB_TOKEN }}
```

10. 从注册表中提取镜像的元数据，使用长 SHA 作为容器的标签。为该步骤指定一个 ID，以便稍后访问其输出：

```
- name: Extract metadata (tags, labels) for Docker
  id: meta
  uses: docker/metadata-action@v5
  with:
    images: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}
    tags: |
      type=sha,format=long
```

11. 设置 Node.js 以使用正确的版本和注册表。然后，构建和测试代码。必须将 NODE_AUTH_TOKEN 环境变量设置为 GITHUB_TOKEN，以验证到软件包注册表，并从第 6 章的软件包配方中接收 npm 软件包：

```
- uses: actions/setup-node@v4
  with:
    node-version: 21.x
    registry-url: https://npm.pkg.github.com/

- name: Build and test
  env:
    NODE_AUTH_TOKEN: ${{ secrets.GITHUB_TOKEN }}
  run: |
    npm install
    npm run test
```

12. 现在，我们准备好构建并推送 Docker 镜像。使用 meta 步骤的输出来设置标签和标签：

```
- name: Build and push Docker image
  uses: docker/build-push-action@v5
  with:
    context: .
    push: ${{ github.event_name != 'pull_request' }}
    tags: ${{ steps.meta.outputs.tags }}
    labels: ${{ steps.meta.outputs.labels }}
```

13. 提交并推送更改。工作流运行后，将在库的 **Code**（代码）选项卡右侧的 **Packages**（软件包）下找到 Docker 镜像。

7.2.3 How it works…

我使用 Express(https://expressjs.com/) 作为简单的 Web 框架来运行网站。该网站显示软件包中的内容；将在后续示例中利用这一点来自动保持依赖项的更新。代码很容易理解：

```
const express = require('express');
const greet = require('@wulfland/package-recipe/src/index')
const app = express();
const port = 3000;

app.get('/', (req, res) => {
  res.send(greet());
});

app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

这次，我省略了所有的代码检查和测试，已经在第 6 章中介绍过了。并且需要将此应用程序容器化，以便稍后将其部署到云端。

要将容器部署到云端，必须将其存储在容器注册表中。这里使用 GitHub packages，但使用特定于云的注册表方式相同。唯一区别是，需要配置一个 PAT 令牌，不能使用 GITHUB_TOKEN。

7.2.4 There’s more…

Docker meta-action(https://github.com/docker/metadata-action) 可以从 Git 引用和 GitHub 事件中提取元数据。就像第 6 章中的 GitVersion 一样，可以用于自动化您的容器版本控制，还支持语义化版本控制：

```
- name: Docker meta
  id: meta
  uses: docker/metadata-action@v5
  with:
    images: |
      ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}
    tags: |
      type=ref,event=branch
      type=ref,event=pr
      type=semver,pattern={{version}}
      type=semver,pattern={{major}}.{{minor}}
```

我们的示例中，只使用 Git SHA 以便能够部署每个提交，但可以根据工作流轻松扩展版本控制。

7.3. 使用 OIDC 安全地部署到云端

本示例中，我们将设置 Azure 中的 Kubernetes 集群，并在 Azure 中配置 OIDC，以便在不使用存储秘钥的情况下部署到集群。

7.3.1 Getting ready

确保至少有一个具有对软件包的读访问权限的 PAT 令牌。

如果熟悉 Azure 并且本地安装了 Azure CLI(https://docs.microsoft.com/cli/azure/install-azure-cli?view=azure-cli-latest)，可以直接从那里开始操作。如果是 Azure 新手或者没有安装 CLI，只需使用 Azure Cloud Shell(https://shell.azure.com) 即可。

将 PAT 令牌设置为环境变量：

```
$ export GHCR_PAT=<YOUR_PAT_TOKEN>
```

该令牌将由 Kubernetes 用于从 GitHub Package Registry 读取。打开脚本 setup-azure.sh 并调整文件顶部的 location 变量，以选择想要的 Azure 区域。可以使用 az account list-locations -o table 获取区域列表。

提交并推送更改，然后运行脚本：

```
$ git clone https://github.com/{OWNER}/release-recipe.git
$ cd release-recipe
$ chmod +x setup-azure.sh
$ ./setup-azure.sh
```

这将创建一个 Azure Kubernetes Service，并将其与 GitHub Container registry 连接。在脚本运行时，可以配置 OIDC 以从工作流访问。

7.3.2 How to do it…

1. 使用 Cloud Shell 或本地终端，并创建一个新的应用程序注册：

   ```
   $ az ad app create --display-name release-recipe
   ```

2. 然后，使用注册输出中的应用 ID 创建一个服务主体：

   ```
   $ az ad sp create --id <appId>
   ```

3. 然后，打开 Azure 门户，在 Microsoft Entra 中，在 **App registrations**(应用程序注册) 下找到 release-recipe。在 **Certificates & secrets | Federated credentials | Add credentials**(证书和机密 | 联合凭证 | 添加凭证) 下添加 OIDC 信任，填写表单，将组织设置为 GitHub 用户名，输入库名，并将实体类型选择为 **Environment**(环境)(见图 7.1)：

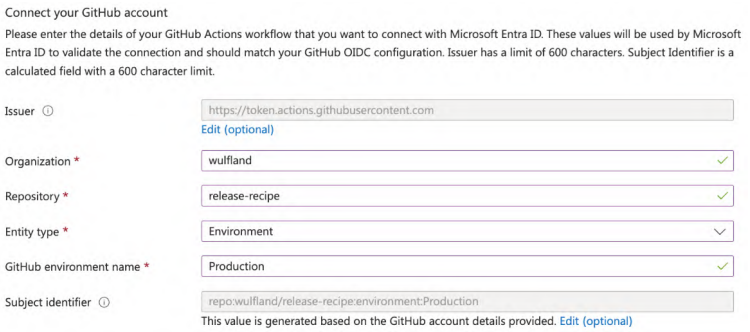


图 7.1 – 在 Microsoft Entra 中连接 GitHub 账户

为这些凭证命名并点击 **Add**(添加)。记下 release-recipe 应用程序的 **Application (client) ID**(应用程序 (客户端)ID) 和 **Directory (tenant) ID**(目录 (租户)ID)(见图 7.2)。稍后会用到这些信息:

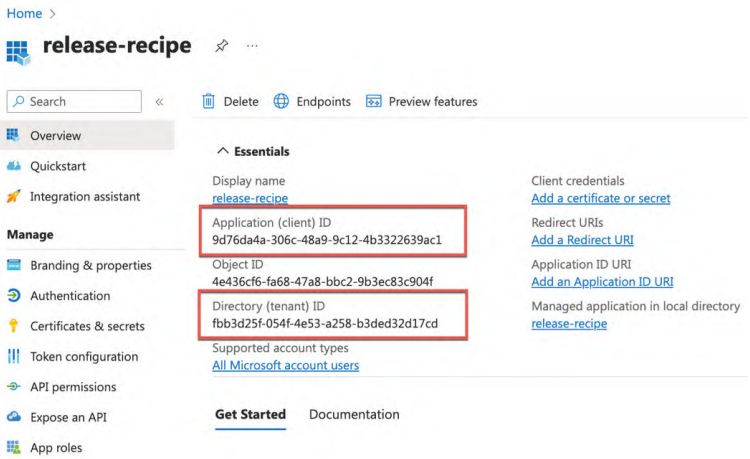


图 7.2 – 应用注册的客户端和租户 ID

4. 然后,在订阅中为服务主体分配一个角色,并在门户中打开订阅。在 **Access control (IAM) | Role assignment | Add | Add role assignment**(访问控制 (IAM) | 角色分配 | 添加 | 添加角色分配) 下,按照向导操作。选择角色——例如,**Contributor**(参与者)——再点击 **Next**(下一步)。选择 “User, group, or service principal”,然后选择之前创建的服务主体。

7.3.3 How it works…

可以使用 OIDC(而不是使用存储为机密的凭据,来连接到云提供商,如 Azure、AWS、GCP 或 HashiCorp)。OIDC 将使用短期有效的令牌进行身份验证,而非凭据。云提供商也需要在其端支持 OIDC。

使用 OIDC 时,无需将云凭据存储在 GitHub 中,可以对工作流可以访问的资源拥有更细粒度的控制,并且拥有轮换的、短期有效的令牌,这些令牌在工作流运行后将过期。图 7.3 展示了 OIDC 的工作原理概述:

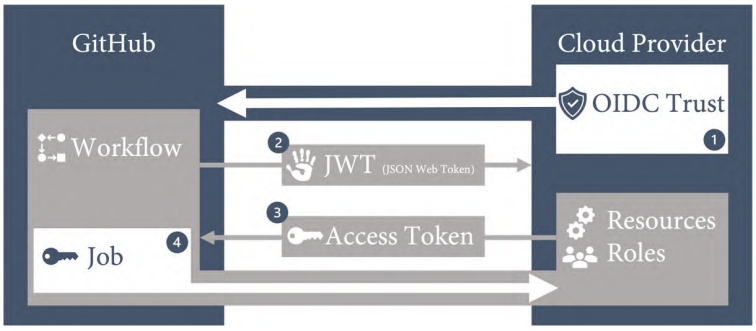


图 7.3 – OIDC 与云提供商的集成

步骤如下：

- 在云提供商和 GitHub 之间创建一个 OIDC 信任关系。将信任限制到一个组织和库，并进一步限制对环境、分支或拉取请求的访问。
- GitHub OIDC 提供程序在工作流运行期间，自动生成一个 JSON Web Token。该令牌包含多个声明，用于为特定工作流作业建立安全且可验证的身份。
- 云提供商验证声明，并提供一个仅在作业生命周期内有效的短期访问令牌。
- 使用访问令牌访问该身份有权访问的资源。

可以使用该身份直接访问资源，或者使用它从安全库（例如 Azure Key Vault 或 HashiCorp Vault）中获取凭据。通过这种方式，可以使用库安全地连接到不支持 OIDC 和自动密钥轮换的服务。

在 GitHub 上，可以在 https://docs.github.com/en/actions/deployment/security-hardening-your-deployments 找到有关为 AWS、Azure 和 GCP 配置 OIDC 的说明。

7.4. 环境审批检查

我们已经在第 5 章中使用了环境，所以这部分是重复的，所以这个示例会很简短。环境管理核心发布，接下来的示例将展示，如何使用它们在 Kubernetes 中显示我们服务的 URL。

7.4.1 Getting ready

确保手头有来自前几章的 **Application (client) ID**(应用程序（客户端)ID)、**Directory (tenant) ID**(目录（租户)ID）和 **Subscription ID**(订阅 ID)。可以通过以下方式获取订阅 ID：

```
$ az account show
```

7.4.2 How to do it…

1. 在库设置中，转到 **Environments**(环境)，点击 **New environment**(新建环境)，并创建一个新环境：Production。

2. 添加 main 作为部署分支。
3. 添加一个名为 AZURE_CLIENT_ID 的新 **Environment secret**(环境密钥)，并将其设置为 **Application (client) ID**(应用程序（客户端)ID)。
4. 添加一个名为 AZURE_TENANT_ID 的新 **Environment secret**(环境密钥)，并将其设置为 **Directory (tenant) ID**(目录（租户)ID)。
5. 添加一个名为 AZURE_SUBSCRIPTION_ID 的新 **Environment secret**(环境密钥)，并将其设置为 **Subscription ID**(订阅 ID)。
6. 添加一个名为 AZURE_CLUSTER_NAME 的新 **Environment secret**(环境密钥)，并将其设置为集群名称（如果没有修改 setup-azure.sh 脚本，则为 AKSCluster)。
7. 添加一个名为 AZURE_RESOURCE_GROUP 的新 **Environment secret**(环境密钥)，并将其设置为资源组名称（如果没有修改 setup-azure.sh 脚本，则为 AKSCluster)。

我们将在下一个示例中使用这些环境密钥，以安全地将应用程序部署到云端的 Kubernetes。

7.4.3 How it works…

环境为工作流中的作业添加了一层抽象，并且可以通过规则进行保护。有关审批检查的更多详情，请参阅第 5 章。环境也可以被 OIDC 实体信任，这就是我们将在下一个示例中使用的功能。

7.5. 将容器应用程序发布到 AKS

现在，是时候将我们的应用程序发布到 AKS 中的生产环境了。

7.5.1 Getting ready

打开文件.github/workflows/publish.yml。

7.5.2 How to do it…

1. 在工作流的顶部，添加两个更多的环境变量：

```
env:
  REGISTRY: 'ghcr.io'
  IMAGE_NAME: '${{ github.repository }}'
  APP_NAME: 'release-recipe-app'
  SERVICE_NAME: 'release-recipe-service'
```

为前一个示例中的 build-and-push-image 作业添加一个输出，以便新作业能够访问镜像名称：

```
outputs:
  image_tag: ${{ fromJSON(steps.meta.outputs.json).tags[0] }}
```

2. 向工作流添加一个名为 production 的第二个作业，该作业仅在推送到 main 时运行，并与生产环境相关联。将环境的 URL 设置为稍后添加的一个步骤的输出。该作业需要权限 id-token: write 和 content: read 以便 OIDC 能够工作：

```
production:
  if: github.ref == 'refs/heads/main' && github.event_name == 'push'
  needs: build-and-push-image
  runs-on: ubuntu-latest
  permissions:
    id-token: write
    contents: read
  environment:
    name: Production
    url: ${{ steps.get-service-url.outputs.SERVICE_URL}}
```

3. 添加步骤以签出库并使用 OIDC 登录 Azure：

```
- name: Checkout
  uses: actions/checkout@v4

- name: 'Az CLI login'
  uses: azure/login@v1
  with:
    client-id: ${{ secrets.AZURE_CLIENT_ID }}
    tenant-id: ${{ secrets.AZURE_TENANT_ID }}
    subscription-id: ${{ secrets.AZURE_SUBSCRIPTION_ID }}
```

4. 设置 Kubernetes 部署的上下文：

```
- name: 'Az CLI set AKS context'
  uses: azure/aks-set-context@v3
  with:
    cluster-name: ${{ secrets.AZURE_CLUSTER_NAME }}
    resource-group: ${{ secrets.AZURE_RESOURCE_GROUP }}
```

5. 检查文件 service.yml，其部署了一个负载均衡器，将应用程序的 3000 端口映射在 80 端口上。此外，检查 deployment.yml，其中包含应用程序的定义。要替换文件中的环境变量，可使用 envsubst。然后，将结果传递给 kubectl 并应用清单文件：

```
- name: Deploy
  env:
    IMAGE: ${{ needs.build-and-push-image.outputs.image_tag }}
  run: |-
    envsubst < service.yml | kubectl apply -f -
    envsubst < deployment.yml | kubectl apply -f -
```

6. 使用 kubectl describe service 获取 Kubernetes 中的服务 URL，并将其设置为

步骤输出，以便在环境 URL 中使用：

```
- name: 'Get Service URL'
  id: get-service-url
  run: |
    IP=$(kubectl describe service $SERVICE_NAME| grep "LoadBalancer Ingress: " | awk
    ↪  '{print $3}')
  echo "SERVICE_URL=http://$IP" >> $GITHUB_OUTPUT
```

7. 最后，我们想要检查部署是否成功。如果应用程序有一个 /health 端点，需要查询该端点，但由于应用程序非常简单，只依赖返回的状态码即可：

```
- name: 'Run smoke test'
  env:
    SERVICE_URL: ${{ steps.get-service-url.outputs.SERVICE_URL}}
  run: |
   status=`curl -s --head $SERVICE_URL | head -1 | cut -f 2 -d' '`
  if [ "$status" != "200" ]
  then
    echo "Wrong HTTP Status. Actual: '$status'"
    exit 1
  fi
```

这段代码的作用是使用 curl --head 查询网站的头部信息，-s 选项会抑制其他输出。然后，使用 head -1 获取第一行。该行看起来像 HTTP/1.1 200 OK。我们使用空格分割字符串并取第二个元素（状态码），如果状态码不是 200(OK)，则会引发异常。

8. 提交并推送更改到 main 分支。这将触发工作流；这将把容器的新版本推送到注册表，并从那里在 AKS 中发布它。按照服务的 URL(见图 7.4)，并验证网站是否正确显示：

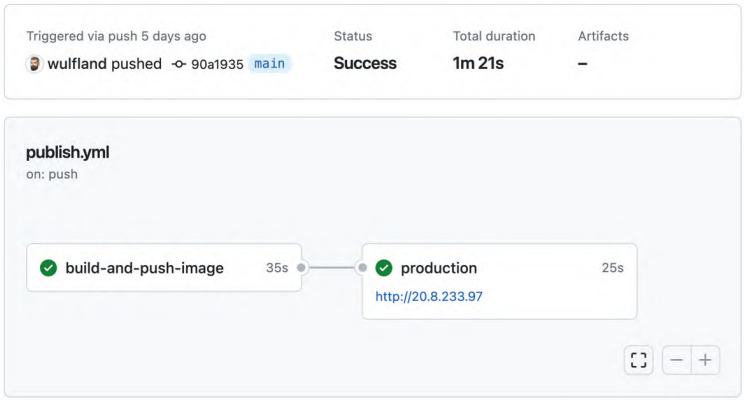


图 7.4 – 部署到动态环境

9. 将显示来自容器的 Hello World 应用程序。

7.5.3 How it works…

Azure 登录操作将使用 OIDC 身份验证 Azure，可以将此身份授予对 Azure 资源的细粒度访问权限。由于我们限制了库中对生产环境的访问，因此只有此作业可以使用应用程序来验证 Azure，也可以使用分支、标签或拉取请求进行实现。

然后使用 aks-set-context 操作来配置使用 kubectl 访问 AKS，可将实际应用程序部署到 Kubernetes。

7.5.4 There's more…

Kubernetes 可以非常快地变得非常复杂。实际上，可向集群添加 DNS 和 SSL，并使用命名空间来管理并行运行的多容器。这是一种很棒的方式，可以将每个拉取请求部署到动态环境，但这些超出了本书的范围。

如果想使用其他云提供商（而不是 Azure）来发布容器，可在这里找到有关如何部署到 AWS Elastic Container Service（ECS）的动手指南：https://github.com/wulfland/AccelerateDevOps/blob/main/ch9_release/Deploy_to_AWS_ECS.md，或者在这里找到另一个有关如何部署到 Google Kubernetes Engine（GKE）的指南：https://github.com/wulfland/AccelerateDevOps/blob/main/ch9_release/Deploy_to_GKE.md。

7.6. 自动化更新依赖项

现在，我们已经实现了从包库到发布库，再到生产环境的端到端工作流，我展示了如何使用 dependabot 结合 GitHub Actions 来自动化更新依赖项的过程。

7.6.1 Getting ready

导航到 **Settings | Code security and analysis**（设置 | 代码安全与分析），并确保 **Dependency graph**（依赖项图）已启用（见图 7.4）：

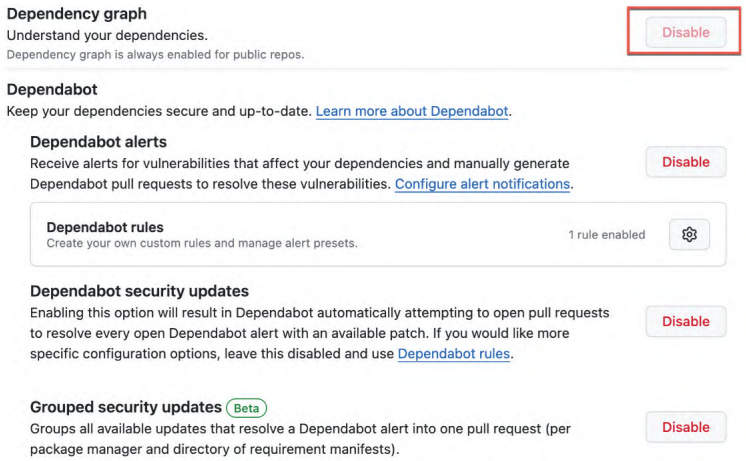


图 7.5 – 启用依赖项图和可选的 dependabot 警报

这将分析库并检测所有依赖项，可以在 **Insights | Dependency graph**（洞察 | 依赖项图）下检查这些依赖项，还可以启用 Dependabot 警报。当依赖项存在已知漏洞时，dependabot 会进行通知。Dependabot 安全更新更进一步，dependabot 会生成一个拉取请求，其中包含一个更新到非易受攻击版本的版本。为了减少拉取请求的数量，可以将更新分组在一起（此功能仍处于测试阶段）。

7.6.2 How to do it…

1. 创建一个名为 PAT 的新 dependabot 密钥，并将其设置为对软件包具有读取访问权限的 PAT 令牌：

```
$ gh secret set PAT --app dependabot
```

2. 创建一个新文件：.github/dependabot.yml。总是以 version:2 开头：

```
version: 2
```

3. 使用 PAT 配置一个新的 npm-registry，指向https://npm.pkg.github.com

```
registries:
  npm-pkg:
    type: npm-registry
    url: https://npm.pkg.github.com
    token: ${{ secrets.PAT }}
    replaces-base: true
```

4. 版本更新在 updates 下配置。添加生态系统 npm ，并指向创建的注册表的名称：

```
updates:
  - package-ecosystem: "npm"
    directory: "/"
    schedule:
      interval: "weekly"
    registries:
      - npm-pkg
```

这将检查 npm 包的每周更新，包括私有注册表中的包。

5. 可选地，添加 GitHub actions 的更新：

```
- package-ecosystem: "github-actions"
  directory: "/"
  schedule:
    interval: "weekly"
```

这将检查 GitHub actions 的更新版本。

6. 还可以将 Docker 添加为生态系统：

```
- package-ecosystem: "docker"
  directory: "/"
  schedule:
    interval: "weekly"
```

这同样适用于 Kubernetes 清单文件，dependabot 将检查清单文件中镜像标签的更新。但在我们的示例中，需要使用环境变量直接部署新版本。

7. 提交并推送该文件。然后，转到 **Insights | Dependency graph | Dependabot**（洞察 | 依赖项图 |Dependabot）。每个已配置的生态系统都有一个条目，可以通过点击右侧的链接来检查日志文件（见图 7.6）：

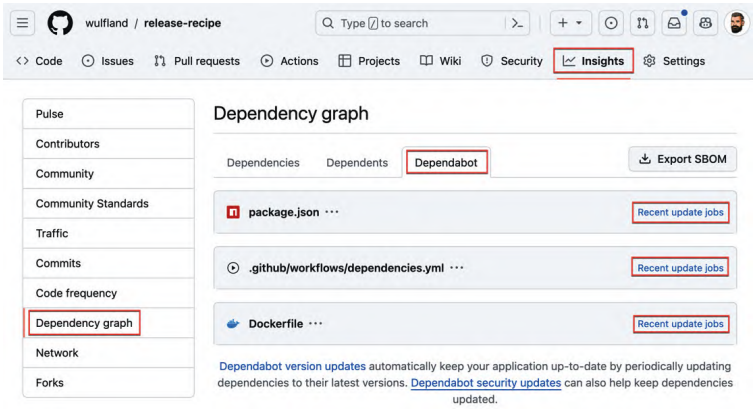


图 7.6 – 检查 dependabot 版本更新的日志

请检查日志中是否有错误。

8. 现在，转到 package-recipe 库，并创建一个带有新补丁版本的新发布。当新包版本发布，转到 **Insights | Dependency graph | Dependabot**（洞察 | 依赖项图 |Dependabot）并点击 package.json 行中的链接。点击 **Check for updates**（检查更新）以强制 dependabot 现在检查更新（见图 7.7）：

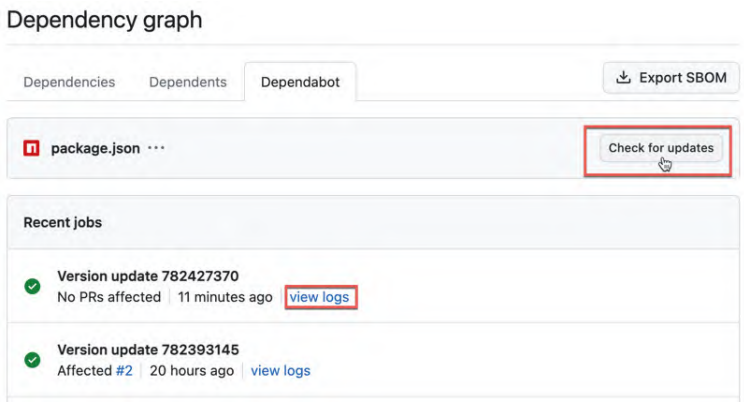


图 7.7 – 让 dependabot 现在检查更新

9. dependabot 将创建一个新的拉取请求，将版本更新到最新版本（见图 7.8）：

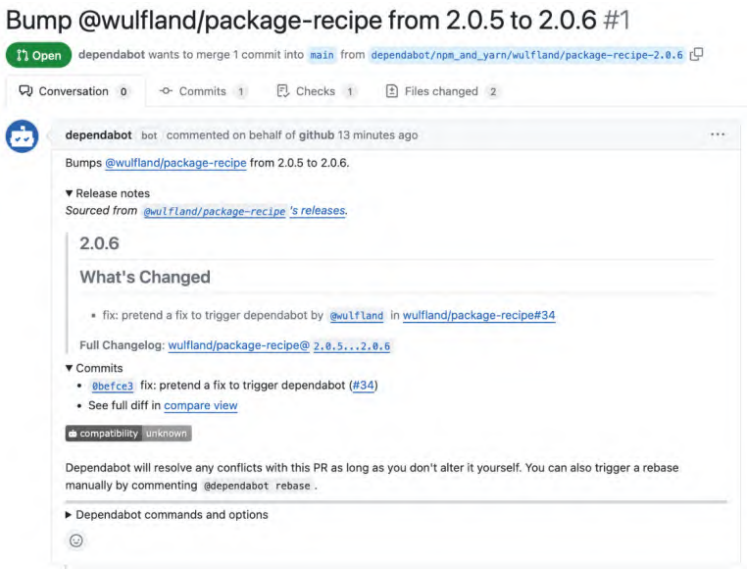


图 7.8 – 由 dependabot 版本更新创建的拉取请求

合并拉取请求或通过在拉取请求上评论 @dependabot squash and merge 让 dependabot 来完成。

10. 由于我们信任包的所有者，可以自动化这一最后步骤，并在所有检查成功后仅合并特定版本的每个拉取请求。创建一个新文件：.github/workflows/dependencies.yml。该工作流将在 pull_request_target 上运行，并且需要对拉取请求进行写入权限：

```
name: Dependabot auto-merge

on: [ pull_request_target ]

permissions:
  pull-requests: write
  contents: write
```

11. 仅当拉取请求的作者为 dependabot 时才运行作业：

```
jobs:
  dependabot:
    runs-on: ubuntu-latest
    if: ${{ github.actor == 'dependabot[bot]' }}
    steps:
```

12. 第一步是获取 dependabot 元数据：

```
- name: Dependabot metadata
  id: metadata
  uses: dependabot/fetch-metadata@v1
  with:
    github-token: "${{ secrets.GITHUB_TOKEN }}"
```

13. 接下来，添加一个基于以下元数据条件的步骤：如果依赖项名称包含您的包（将 {OWNER} 替换为相应的用户名）并且版本更新是补丁版本。如果所有检查都成功，使用 gh merge --auto 合并拉取请求：

```
- name: Enable auto-merge for all patch versions
  if: ${{contains(steps.metadata.outputs.dependency-names, '@{OWNER}/package-recipe') &&
  ↪  steps.metadata.outputs.update-type == 'version-update:semver-patch'}}
  run: gh pr merge --auto --merge "$PR_URL"
  env:
    PR_URL: ${{github.event.pull_request.html_url}}
    GITHUB_TOKEN: ${{secrets.GITHUB_TOKEN}}
```

请注意，此工作流不会触发监听 main 分支 push 触发的 publish.yml 工作流，因为合并是使用 GITHUB_TOKEN 进行的。可以使用与上一步相同的个人访问令牌（PAT），或者可以将发布逻辑移动到一个可重用的工作流中，并直接从此工作流中进行调用。

14. 现在，回到 package-recipe 库并创建一个具有新补丁版本的新发布。当新包版本发布，请转到 **Insights | Dependency graph | Dependabot**(洞察 | 依赖项图 |Dependabot)，然后单击 package.json 行中的链接。点击 **Check for updates**(检查更新）以强制 dependabot 现在检查更新（见图 7.7)。Dependabot 将创建一个拉取请求，触发工作流，并且该拉取请求将自动合并。

7.6.3 How it works…

Dependabot 可以以更少的精力使依赖项保持最新。可以使用它来自动化更新过程，并跟踪所有依赖项的最新发布。

它支持许多生态系统：

- Bundler
- Cargo
- Composer
- 开发容器（包括 GitHub Codespaces）
- Docker
- Hex
- Elm-packages
- Git 子模块
- GitHub Actions
- Go 模块
- Maven 和 Gradle
- npm
- NuGet
- pip、pipenv 和 pip-compile
- pnpm

- poetry
- pub
- Swift
- Terraform
- yarn

为了获得完整的列表，请参阅以下链接：https://docs.github.com/en/code-security/dependabot/dependabot-version-updates/about-dependabot-version-updates。

Dependabot 将为每个依赖项的每个版本更新创建拉取请求，可以在拉取请求中使用 @Dependabot 命令与 dependabot 交互，并告诉它某些事情，例如：忽略版本或重新应用更改。

dependabot.yml 文件有许多选项，可以指定允许哪些类型的更新，自定义提交消息，将更新分组在一起，忽略某些依赖项，以及添加审阅者、标签或分配人。有关配置选项的完整列表，请参阅以下链接：https://docs.github.com/en/code-security/dependabot/dependabotversion-updates/configuration-options-for-the-dependabot.yml-file。

结合工作流和 dependabot/fetch metadata 操作，这是一个非常强大的工具，可以跨许多库和团队自动化您的供应链。回看第 6 章和第 7 章，我们使用 Conventional Commits 和 GitVersion 根据约定的提交消息完全自动化语义化版本控制，可以利用 dependabot 完全自动化下游依赖项的更新。

7.6.4 There’s more…

我们的示例中,直接构建容器并将部署推送到 Kubernetes,也可以通过在 dependabot.yml 文件的 Docker package-ecosystem 元素中为包含引用 Docker 镜像标签的清单的每个目录添加一个条目,使用 dependabot 更新 Kubernetes 清单文件。Kubernetes 清单可以是普通的 Kubernetes 部署文件，也支持 Helm 图表。有关为 Kubernetes 配置 dependabot.yml 文件的更多信息，请参阅以下链接：https://docs.github.com/en/code-security/dependabot/dependabot-version-updates/configurationoptions-for-the-dependabot.yml-file#docker。

7.7. 清理

请不要忘记在完成配方后删除集群，以免产生不必要的费用。可以使用 repo 中的 destroy-azure.sh 脚本并在本地运行，或者只需在 Azure Cloud Shell(https://shell.azure.com) 中运行以下命令即可：

```
$ az group delete --resource-group AKSCluster --yes
```

只需检查在设置所有内容时，是否更改了资源组的名称。

7.8. 总结

祝贺您读完了本书。希望这些实用、动手实践、重点突出的示例能为日常工作中自动化各种任务打下基础，并帮助提高您工程团队的生产力。我努力覆盖了 GitHub Actions 中所有相关的方面，平衡了简单性和实际应用性。GitHub 是一个发展非常迅速的平台，每天都会发布很多次更新。如果考虑到来自开源社区的合作者和操作，GitHub 生态系统是巨大的，并且一直都在变化。

如果您在本书的动手实验中遇到了变化，请通过创建问题或提交拉取请求在 GitHub 上联系我，我会尝试在库中整合这些变化。

希望您喜欢这本书，也希望您能像我一样喜欢 GitHub Actions；它是我用过的最好的自动化平台。