

第6章 使用Github Actions实现自动化

很多敏捷方法的使用者认为工程实践不如管理和团队实践重要。但是像持续集成(CI)、持续交付(CD)和基础设施即代码(IaC)等工程能力是实现更频繁、更稳定和更低风险发布的支撑因素(Humble, J., & Farley, D. 2010)。这些做法降低了部署的难度，从而让开发者减少加班，使工作更加轻松。

从本质上讲，这些做法都与自动化有关，即让计算机执行重复任务，使人们可以专注于解决重要问题和进行一些创造性工作。

“计算机执行重复任务，人们解决问题。”

Forsgren, N., Humble, J., & Kim, G. 2018

自动化对企业文化和人们的工作方式有很大的影响，因为许多习惯是为了避免手动、重复的任务而养成的，尤其是一些非常容易出错的任务。本章将介绍GitHub自动化引擎——GitHub Actions，该引擎具有包括CI/CD在内的多种功能。

本章将包含以下话题：

- GitHub Actions概述
- 工作流、管道和操作
- YAML基础
- 工作流语法
- 使用密钥工作
- 动手实践——第一个工作流
- 动手实践——第一个操作
- GitHub Marketplace

GitHub Actions概述

GitHub Actions是GitHub原生自动化引擎，不仅是提交代码进行版本控制，它还允许用户在GitHub中的任何事件上都可以运行工作流。GitHub可以在以下情况下触发的工作流：Issue状态更改或被添加到里程碑、卡片在GitHub项目中被移动、仓库被他人点亮星标或讨论中新增了评论，几乎所有事件都可以触发工作流。工作流本身是为重用而构建的，用户只需将代码提交到仓库中即可构建可重用的操作。也可以分享自己的操作到拥有约10,000个操作的GitHub Marketplace(<https://github.com/marketplace>)上。

工作流可以在Linux、macOS、Windows、ARM和容器这些主流平台的云端中执行，甚至可以在云端或自己的数据中心配置和托管运行器，而无需对外暴露端口。

GitHub Learning Lab

GitHub Learning Lab(<https://lab.github.com>) 是一门通过实操来学习Github的课程，课程通过issue和pull request不断推进，其中有一条完整的 **DevOps with GitHub Actions** 学习路线(<https://lab.github.com/githubtraining/devops-with-github-actions>)。读者也可以单独参加一些课程，比如 **GitHub Actions: Hello World** (<https://lab.github.com/githubtraining/github-actions:-helloworld>)。所有课程都是免费的，如果读者之前没接触过Github并且喜欢在实践中学习，这值得一试！

工作流、管道和操作

GitHub工作流是一个可配置的自动化过程，由不同的**作业**组成，可以在**YAML**文件中配置，并放在仓库的 `.github/workflows` 目录中。工作流可用于构建并部署软件到不同的环境或平台，在其他CI/CD系统中通常称为**管道**。

作业是在配置好的运行器上执行的工作流的一部分，运行器环境使用**runs-on**属性进行配置。默认情况下作业并行运行，不同作业可以通过依赖（使用**needs**关键字）链接在一起顺序执行，每个作业可以在指定环境中运行。**环境**是资源的逻辑分组，可以由多个工作流共享，还可以使用**保护规则**进行保护。

作业由一系列称为**步骤**的任务组成，每个步骤可以运行命令、脚本或**GitHub操作**。**操作**是工作流的可重用部分，并非所有步骤都是操作，但所有操作都作为作业中的步骤执行。

表6.1展示了工作流中的一些重要术语：

Noun	Description
Workflow	Automated process. Often referred to as a pipeline.
Job	A part of the workflow that consists of a sequence of tasks that are executed on a runner.
Runner	A virtual or physical machine or container that executes a job of a workflow. Can be cloud-hosted or self-hosted. Also referred to an agent.
Step	A single task that is executed as part of a job.
Action	A reusable step that can be used in different jobs and workflows. This can be a Docker container, JavaScript, or a composite action that consists of other steps. It can also be shared via the GitHub marketplace.
Environment	A logical group of resources that can share the same protection rules and secrets. Environments can be used in multiple workflows.

Table 6.1 – Important terms for GitHub Actions

表6.1-GitHub Actions中的一些关键术语

名词	描述
工作流	自动化流程，通常称为管道
作业	工作流的一部分，由一系列在运行器上执行的任务组成
运行器	执行作业的虚拟或物理计算机或容器，可以云托管或自托管，也称为代理
步骤	单个任务，作为作业的一部分执行

名词

描述

操作 一个可用于不同作业和工作流的可重用步骤，可以是Docker容器、JavaScript或由其他步骤组成的复合操作，可分享在GitHub Marketplace。

环境 可以共享相同保护规则和密钥的一组逻辑资源，可用于多个工作流

YAML基础

工作流是用扩展名为 `.ym`或 `.yam`的YAML文件编写的。**YAML**(即 *YAML Ain't Markup Language*)是一种优化后的可供人类直接写入和读取的数据序列化语言。它是**JSON**的严格超集，但是使用换行符和缩进代替大括号进行语法表示。与markdown一样，它也适用于pull request，因为更改总是以行为单位。以下将介绍一些可以帮助读者入门的YAML基础知识。

注释

YAML中的注释以`#`开头：

```
# A comment in YAML
```

标量类型

可以使用以下语法定义单个值：

```
key: value
```

支持很多数据类型：

```
integer: 42
float: 42.0
string: a text value
boolean: true
null value: null
datetime: 1999-12-31T23:59:43.1Z
```

注意键和值可以包含空格且不需要引号,但是可以用单引号或双引号来引用键和值：

```
'single quotes': 'have \'one quote\' as the escape pattern'
"double quotes": "have the \"backslash \" escape pattern"
```

跨越多行的字符串，例如脚本块，可以使用管道符号`|`和缩进：

```
literal_block: |
    Text blocks use 4 spaces as indentation. The entire
    block is assigned to the key 'literal_block' and keeps
    line breaks and empty lines.

    The block continuous until the next element.
```

集合类型

嵌套数组类型（也称映射）经常用于工作流中，使用两个缩进空格：

```
nested_type:
  key1: value1
  key2: value2
  another_nested_type:
    key1: value1
```

序列在每个项目之前使用破折号 -：

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

由于YAML是JSON的超集，还可以使用JSON语法单行表示序列和映射：

```
map: {key: value}
sequence: [item1, item2, item3]
```

以上基础足以使读者在GitHub上编辑工作流，如果了解有关YAML的更多信息，可查看 <https://yaml.org/> 。
接下来介绍工作流语法。

工作流语法

工作流文件中首先映入眼帘的是它的名称，GitHub在存储库的“操作”选项卡上会显示工作流的名称：

```
name: My first workflow
```

名称后面紧跟着触发器。

工作流触发器

触发器是指on键对应的值：

```
on: push
```

触发器可以包含多个值：

```
on: [push, pull_request]
```

触发器可能包含其他可配置的值：

```
on:
  push:
    branches:
      - main
      - release/**
  pull_request:
    types: [opened, assigned]
```

触发器可分为三类：

- Webhook事件
- 计划事件
- 手动事件

Webhooks事件几乎包含到目前为止接触到的所有事件，比如将代码推送到GitHub(push)、创建或更新拉取请求(pull_request)、创建或修改问题 (issues)等都属于Webhooks事件。完整Webhooks事件列表可见<https://docs.github.com/en/actions/reference/events-thattriggerworkflows> 。

计划事件与cron作业使用相同的语法，由五个字段组成，分别代表分钟(0-59)、小时(0-23)、日期(1-31)、月份(1-12或JAN-DEC)和星期(0-6或SUN-SAT)。表6.2展示了用户可以使用的运算符：

Operator	Description
*	Any value
,	List separator
-	Range of values
/	Step values

Table 6.2 – Operators for scheduled events

表6.2-计划事件相关运算符

运算符	描述
*	任何值
,	列表分隔符
-	取值范围

运算符	描述
/	增量值

以下是一些示例：

```
on:
  schedule:
    # Runs at every 15th minute of every day
    - cron: '*/15 * * * *'
    # Runs every hour from 9am to 5pm
    - cron: '0 9-17 * * *'
    # Runs every Friday at midnight
    - cron: '0 0 * * FRI'
    # Runs every quarter (00:00 on day 1 every 3rd month)
    - cron: '0 0 1 */3 *'
```

手动事件允许用户手动触发工作流：

```
on: workflow_dispatch
```

用户可以定义用户在启动工作流时的可选(或必需)**参数**。以下示例定义了一个名为`homedrive`的变量，可以使用 `${{ github.event.inputs.homedrive }}` 表达式在工作流中使用该变量：

```
on:
  workflow_dispatch:
    inputs:
      homedrive:
        description: 'The home drive on the machine'
        required: true
        default: '/home'
```

也可以使用GitHub API触发工作流。为此，用户必须定义一个`repository_dispatch`触发器并指定一个或多个相关事件的名称：

```
on:
  repository_dispatch:
    types: [event1, event2]
```

该工作流会在发送HTTP POST请求后触发，以下是使用`curl`命令发送HTTP POST请求的示例：

```
curl \
-X POST \
```

```
-H "Accept: application/vnd.github.v3+json" \
https://api.github.com/repos/<owner>/<repo>/dispatches \
-d '{"event_type":"event1"}'
```

以下是相应的JavaScript示例（有关JavaScript的Octokit API客户端的更多详细信息可见<https://github.com/octokit/octokit.js>）：

```
await octokit.request('POST /repos/{owner}/{repo}/dispatches',
{
  owner: '<owner>',
  repo: '<repo>',
  event_type: 'event1'
})
```

使用`repository_dispatch`触发器，用户可以在任意系统中使用任意webhook来触发工作流，这有助于工作流的自动化和集成到其他系统。

工作流作业

工作流本身在作业中进行配置，作业是映射，而不是列表，并且作业默认是并行运行的。如果用户想按照一定顺序链接它们，可以使用`needs`关键字让一个作业依赖于其他作业：

```
jobs:
  job_1:
    name: My first job
  job_2:
    name: My second job
    needs: job_1
  job_3:
    name: My third job
    needs: [job_1, job_2]
```

每个作业都在运行器上执行，运行器可以是自托管的，也可以从云端选择，云端有各种适用于不同平台的版本。如果想一直使用最新版本，用户可以使用`ubuntu-latest`、`windows-latest`或`macos-latest`。读者将在第7章中了解有关运行器的更多信息：

```
jobs:
  job_1:
    name: My first job
    runs-on: ubuntu-latest
```

如果要运行具有不同配置的工作流，可以使用**矩阵策略**。工作流将执行配置矩阵值的所有组合对应的多个作业，矩阵中的键可以是任意值，可以使用 `${{ matrix.key }}` 表达式进行引用：

```
strategy:
  matrix:
    os_version: [macos-latest, ubuntu-latest]
    node_version: [10, 12, 14]
jobs:
  job_1:
    name: My first job
    runs-on: ${{ matrix.os_version }}
    steps:
      - uses: actions/setup-node@v2
        with:
          node-version: ${{ matrix.node_version }}
```

Workflow steps

A job contains a series of steps, each step can execute a command:

```
steps:
  - name: Install Dependencies
    run: npm install
```

Statements can run multi-line scripts. If the user does not want the workflow to run in the default shell, then the shell can be configured, or other values (such as `working-directory`) can be configured:

```
- name: Clean install dependencies and build
  run: |
    npm ci
    npm run build
  working-directory: ./temp
  shell: bash
```

Table 6.3 shows the shells available in workflows:

Parameter	Description
bash	Bash shell. This is the default shell on all non-Windows platforms with a fallback to sh. When specified on Windows, the Bash shell that's included with Git is used.
pwsh	PowerShell Core. Default on the Windows platform.
python	The Python shell. Allows you to run Python scripts.
cmd	Windows only! The Windows Command Prompt.
powershell	Windows only! The classic Windows PowerShell.

Table 6.3 – Available shells in workflows

Table 6.3-Workflow available shells

参数	描述
bash	Bash shell是所有非Windows平台的默认shell，可以兼容sh。在Windows平台指定时，将使用Git中包含的Bash shell。
pwsh	PowerShell核心，Windows平台的默认shell
python	Python shell，可以运行Python脚本
cmd	Windows平台专属，Windows命令提示符
powershell	Windows平台专属，传统的Windows PowerShell

*bash*是非Windows系统上的默认shell，可以兼容*sh*，Windows上的默认shell是*cmd*。用户还可以使用语法 *command [options] {0}* 来自定义shell：

```
run: print %ENV
shell: perl {0}
```

大多数用户会重用步骤。可重用的步骤称为**GitHub操作**，可以使用*uses*关键字和以下语法引用操作：

```
{owner}/{repo}@{ref}
```

{owner}/{repo} 是GitHub上操作的目录路径。*{ref}* 指代版本：它可以是标签、分支或某个提交的哈希值，一个常见的应用是使用标签对主要版本和次要版本进行显式版本控制：

```
# Reference a version using a label
- uses: actions/checkout@v2
- uses: actions/checkout@v2.2.0
# Reference the current head of a branch
- uses: actions/checkout@main
# Reference a specific commit
- uses: actions/checkout@a81bbbf8298c0fa03ea29cdc473d45769f953
675
```

如果操作与工作流在同一仓库内，则可以使用操作的相对路径：

```
uses: ../github/actions/my-action
```

用户可以使用 *docker://{image}:{tag}* 引用存储在容器镜像仓库(如Docker Hub或GitHub Packages)中的操作：

```
uses: docker://alpine:3.8
```

上下文和表达式语法

矩阵策略中有一些表达式，表达式的语法如下：

```
${{ <expression> }}
```

表达式可以获取上下文数据并将其与运算符组合，许多对象(如matrix、github、env和runner)都可以提供上下文。举例来说，可以通过`github.sha`来获取触发工作流的提交的SHA值，可以通过`runner.os`来获取运行器的操作系统，可以通过`env`访问环境变量等。完整列表可见

<https://docs.github.com/en/actions/reference/context-and-expressionsyntaxfor-github-actions#contexts>。

用户可以使用两种语法获取上下文数据，其中第二种语法较为常见：

```
context['key']  
context.key
```

根据键的格式，在某些情况下用户可能必须使用第一种语法，比如键以数字开头或键中包含特殊字符。

表达式经常在`if`对象中使用，以便根据不同条件决定是否运行作业：

```
jobs:  
  deploy:  
    if: ${{ github.ref * 'refs/heads/main' }}  
    runs-on: ubuntu-latest  
    steps:  
      - run: echo "Deploying branch $GITHUB_REF"
```

有许多预定义的函数可供使用，比如`contains(search, item)`：

```
contains('Hello world!', 'world')  
# returns true
```

其他函数的例子有 `startsWith()` 或 `endsWith()` 等，还有一些特殊的函数可用于检查当前作业的状态：

```
steps:  
  ...  
  - name: The job has succeeded  
    if: ${{ success() }}
```

该步骤只有在所有其他步骤都成功后才会执行，表6.4展示了可用于反映当前作业状态的所有函数：

Function	Description
success()	Returns <code>true</code> if none of the previous steps have failed or been canceled.
always()	Returns <code>true</code> even if a previous step was canceled and causes the step to always be executed.
cancelled()	Returns <code>true</code> if the workflow was canceled.
failure()	Returns <code>true</code> if a previous step of the job had failed.

Table 6.4 – Special functions to check the status of the job

表6.4-用于检查作业状态的特殊函数

函数	描述
success()	如果前面的步骤都没有失败或取消，则返回true
always()	如果之前某个步骤被取消就会导致该步骤被一直执行，并返回true
cancelled()	如果工作流被取消，则返回true
failure()	如果该作业之前的某一步骤失败，则返回true

除了函数，用户还可以在上下文和函数中使用运算符，表6.5展示了一些常见的运算符：

Operator	Description
()	Logical group
!	Not
< , <=	Less than, less than, or equal to
> , >=	Greater than, greater than, or equal to
==	Equal
!=	Not equal
&&	And
	Or

Table 6.5 – Operators for expressions

表6.5-表达式中的常见运算符

运算符	描述
()	逻辑组合
!	非
<,<=	小于，小于等于
>,>=	大于，大于等于
==	等于
!=	不等于
&&	且
	或

要了解更多关于上下文对象和表达式语法的信息，可访问

<https://docs.github.com/en/actions/reference/context-and-expression-syntaxforgithub-actions>。

工作流命令

用户可以使用工作流命令在步骤中与工作流交互，工作流命令通常使用`echo`指令传递给进程，向进程发送诸如 `::set-output name={name}::{value}` 的字符串。以下示例指定了一个步骤的输出，并在另一步骤中获取它，注意如何使用步骤ID来获取某个步骤的输出变量：

```
- name: Set time
  run: |
    time=$(date)
    echo '::set-output name=MY_TIME::$time'
  id: time-gen
- name: Output time
  run: echo "It is ${ steps.time-gen.outputs.MY_TIME }"
```

另一个例子是 `::error` 命令，可以将错误消息写入日志。该命令还有一些可选配置参数，可以设置文件名、行号和列号：

```
::error file={name},line={line},col={col}::{message}
```

用户还可以输出警告和调试消息、对日志行进行分组或设置环境变量。要了解更多关于工作流命令的信息，可访问 <https://docs.github.com/en/actions/reference/workflow-commands-for-github-actions>。

使用密钥工作

所有自动化工作流的一个非常重要的环节是管理密钥。无论部署应用还是访问API接口，都需要凭证或密钥，所以它们需要被谨慎管理。

GitHub中可以在仓库层级、组织层级或某个环境中安全地存储密钥。密钥被加密后再进行存储和传输，不会在日志中显示。

对于组织层级的密钥，用户可以决定哪些仓库可以访问密钥。对于环境级别的密钥，可以决定所需的审查者：只有当工作流被他们批准时，才能访问密钥。

提示：

密钥名不区分大小写，由普通字符([a-z] 和 [A-Z])、数字([0-9])和下划线(_)组成，且不能以 `GITHUB_` 或数字开头。

推荐使用下划线(_)分隔的大写单词对密钥进行命名。

存储密钥

要存储加密的密钥，用户必须拥有仓库的管理员权限，可以通过网页或GitHub CLI创建密钥。

通过网页创建新密钥时，请到仓库页面的**Settings | Secrets**。Secrets有三个选项卡，分别是**Actions**（默认）、**Codespaces**和**Dependabot**。请点击**New repository secret**创建新密钥，接着输入密钥名和密钥值（见图6.1）：

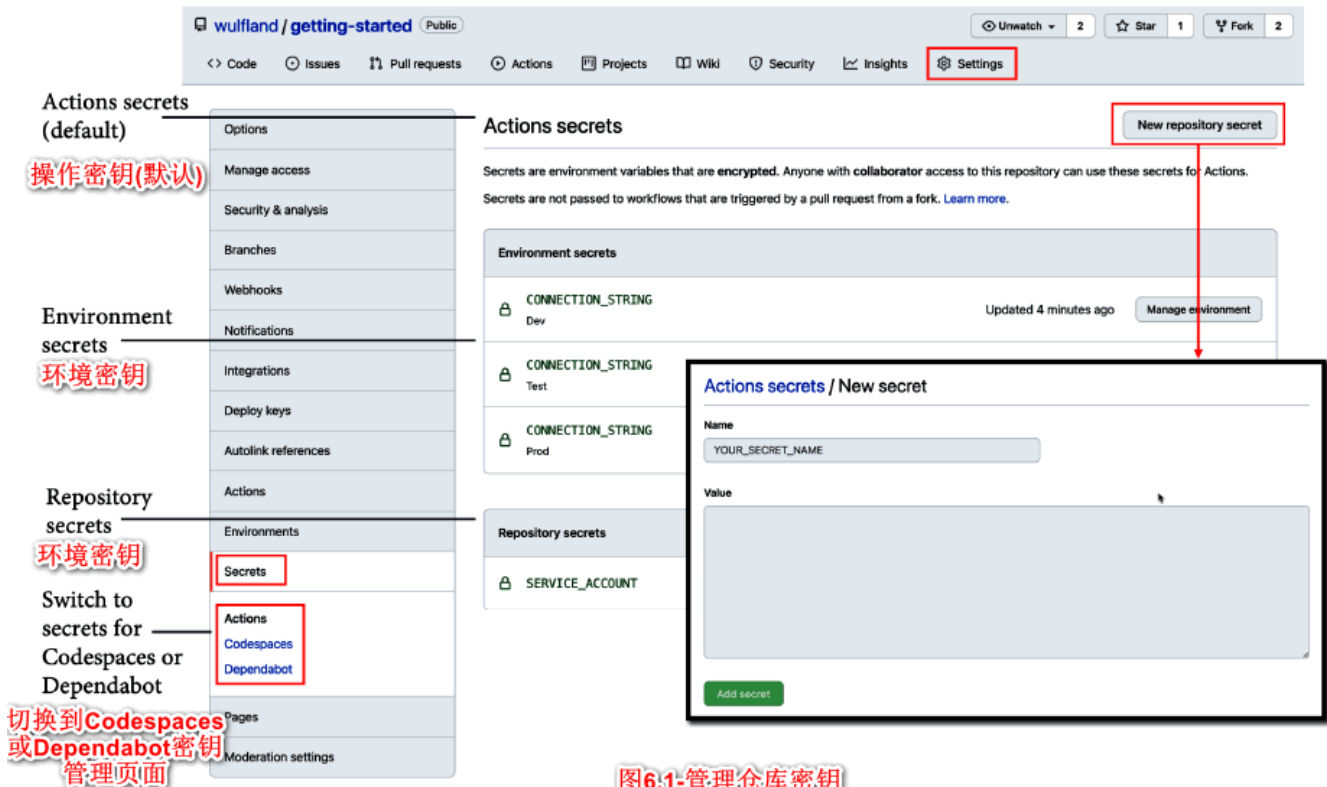


图6.1-管理仓库密钥

Figure 6.1 – Managing repository secrets

组织层级密钥的创建方法类似，在**Settings | Secrets**页面点击**New organization secret**来创建新的组织密钥，并设置该密钥的访问规则，可选访问规则如下：

- 所有仓库
- 私有仓库
- 指定仓库

当选择的访问规则是**Selected repositories**时，用户可以授予指定仓库对该密钥的访问权限。

如果使用GitHub CLI，则可以通过命令`gh secret set`创建一个新密钥：

```
$ gh secret set secret-name
```

输入以上命令后用户将得到一个新密钥。可以从文件中读取密钥，将其传输到命令中，也可以将其指定为密钥的主体（`-b` 或 `--body`）：

```
$ gh secret set secret-name < secret.txt
$ gh secret set secret-name --body secret
```

如果密钥是用于环境的，可以使用 `--env (-e)` 参数指定。

对于组织密钥，用户可以使用 `--visibility` 或 `-v` 参数将该密钥的可见性设置为 *all*、*private* 或 *selected*。如果设为 *selected*，则必须使用 `--repos` 或 `-r` 参数指定一个或多个仓库：

```
$ 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
```

获取密钥

用户可以在工作流中通过 *secrets* 上下文获取密钥，将其作为工作流文件的一个输入 (*with:~*) 或环境变量 (*env:~*) 添加到步骤中。当工作流在队列中时，会读取组织和仓库密钥，而在引用环境的作业开始时，会读取环境密钥。

注意：

GitHub会自动从日志中删除密钥，但在步骤中进行密钥相关操作时，要十分谨慎！

不同shell和环境获取环境变量的语法不同。在Bash中，通过 `$SECRET-NAME` 获取；在PowerShell中，通过 `$env:SECRET-NAME` 获取；而在cmd.exe中，通过 `%SECRET-NAME%` 获取。

以下示例展示了如何在不同shell中获取密钥并将其作为一个输入或环境变量：

```
steps:
  - name: Set secret as input
    shell: bash
    with:
      MY_SECRET: ${ secrets.secret-name }
    run: |
      dosomething "$MY_SECRET "
  - name: Set secret as environment variable
    shell: cmd
    env:
      MY_SECRET: ${ secrets.secret-name }
    run: |
      dosomething.exe "%MY_SECRET%"
```

注意：

以上示例展示了如何将密钥传递给操作。如果用户的工作流步骤是 *run:~*，也可以通过 `${ secrets.secret-name }` 直接访问密钥上下文。如果希望避免脚本注入风险，则不建议这样做。但由于只有管理员可以添加密钥，在评估工作流可读性时需要考虑这点。

GITHUB_TOKEN密钥

GITHUB_TOKEN 密钥是一种特殊的密钥，该密钥是自动创建的，可以通过 *github.token* 或 *secrets.GITHUB_TOKEN* 上下文访问。即使工作流没有将其作为输入或环境变量，也可以通过GitHub操作获取它。该令牌可用于在访问GitHub资源时进行身份验证，资源的默认权限可以设置为 *permissive* 或 *restricted*，在工作流中可以调整这些权限：

```
on: pull_request_target

permissions:
  contents: read
  pull-requests: write

jobs:
  triage:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/labeler@v2
        with:
          repo-token: ${ secrets.GITHUB_TOKEN }
```

要了解更多有关`GITHUB_TOKEN`密钥的信息，可访问
<https://docs.github.com/en/actions/reference/authentication-in-a-workflow>。

动手实践 - 第一个工作流

现在读者已经具备了足够的基础，本书将在之后的章节中深入研究运行器、环境 and 安全性。如果读者刚开始接触GitHub Actions，现在可以开始创建第一个工作流和第一个操作了！

提示：

读者可以使用GitHub的代码搜索，设置编程语言过滤条件(`language:yml`)和工作流路径过滤条件(`path:.github/workflows`)，来筛选找到一些现有的GitHub Actions工作流作为模板。

比如以下搜索请求将返回德国Corona-Warn-App项目的所有工作流程：

```
language:yml path:.github/workflows @corona-warn-app
```

具体步骤如下：

1. 进入仓库主页(链接 <https://github.com/wulfland/getting-started>)，点击右上角的**Fork**按钮进行将该仓库克隆到个人账号中。
2. 在克隆得到的仓库主页，点击**Actions**选项卡，就能看到所有可使用的工作流模板，这些模板针对仓库内代码进行了优化。在本示例中，选择.NET模板，点击**Set up this workflow**按钮：

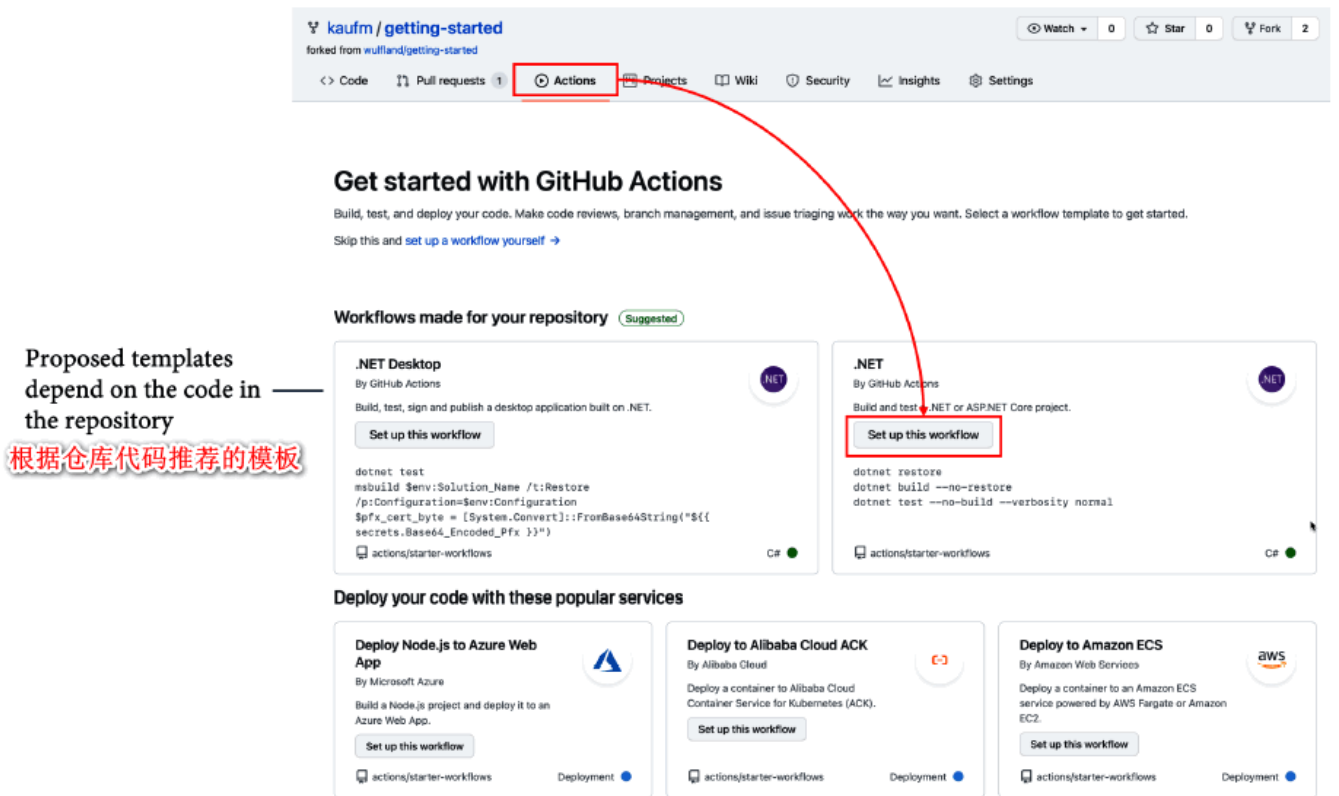


Figure 6.2 – Setting up a GitHub action for .NET

3. GitHub会在编辑器中创建并打开一个相应的工作流文件，该编辑器支持语法高亮和自动补全（按Ctrl + Space），读者也可以在右侧的Marketplace窗口内搜索操作。接着将`dotnet-version`参数设置为3.1.x并提交该工作流文件：

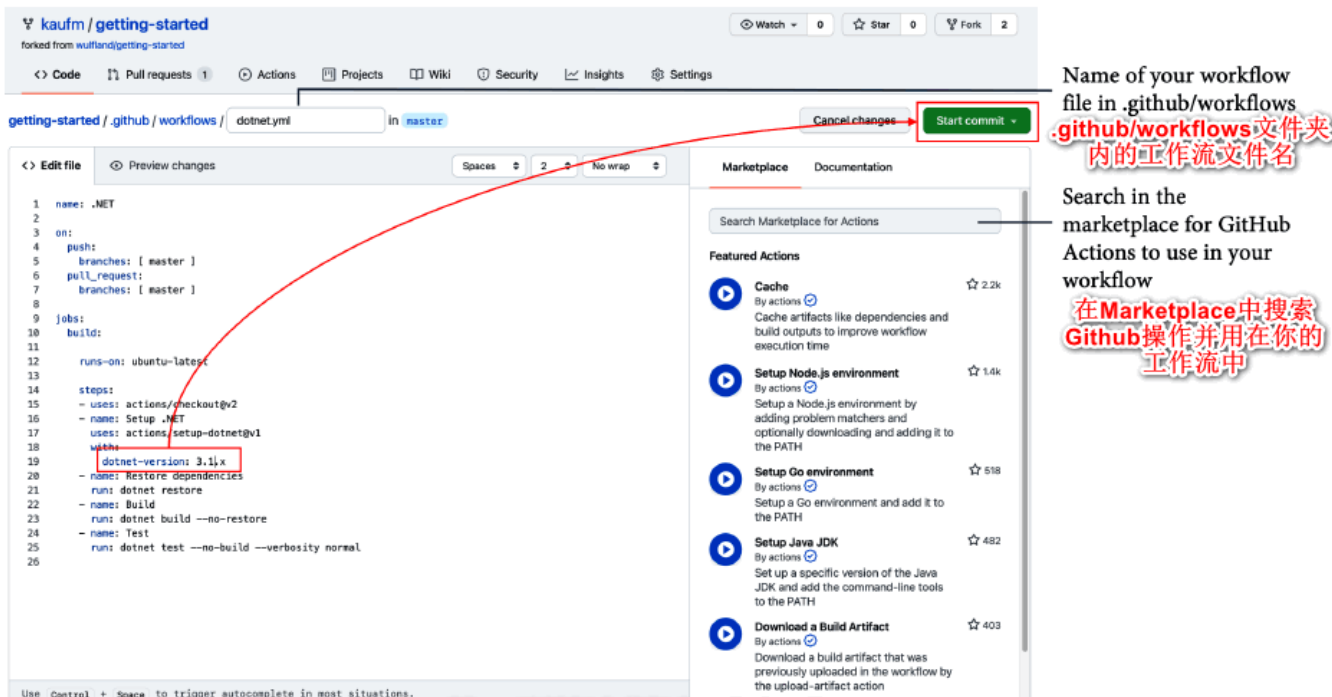


Figure 6.3 – Setting the version and committing the workflow file

4. 工作流将被自动触发，读者可以在**Actions**选项卡下找到运行的工作流。如果点击该工作流，就可以得到其中的所有作业以及一些其他关键信息：

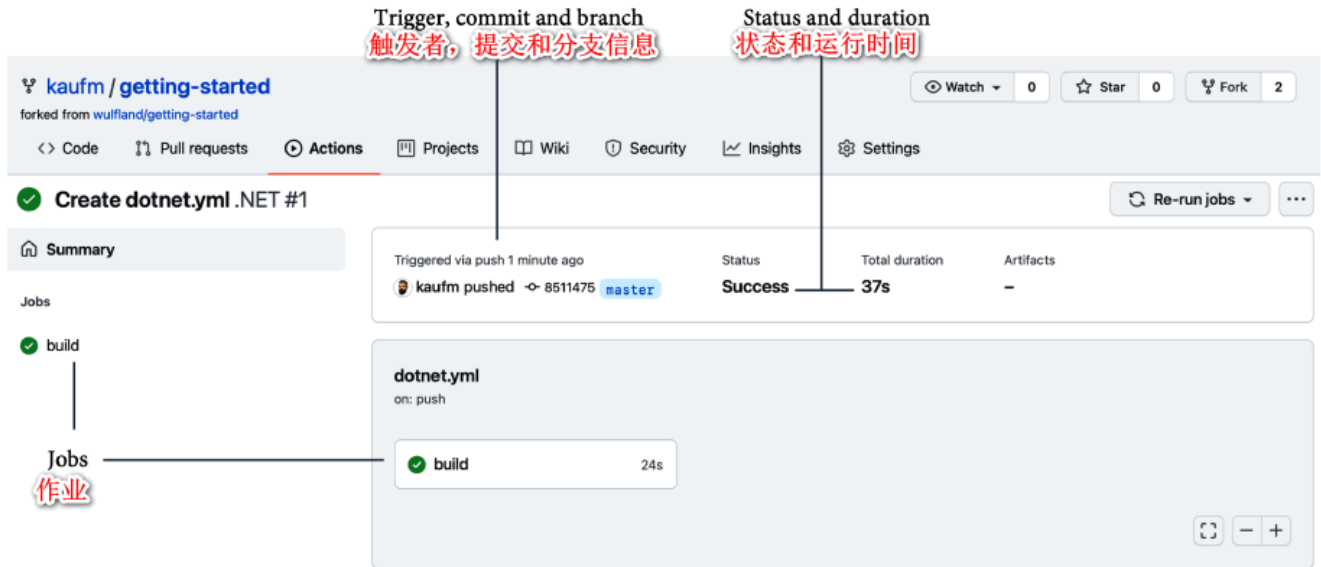


图6.4-工作流总览页面

Figure 6.4 – The workflow summary page

5. 点击作业可以查看作业内所有步骤的详细信息：

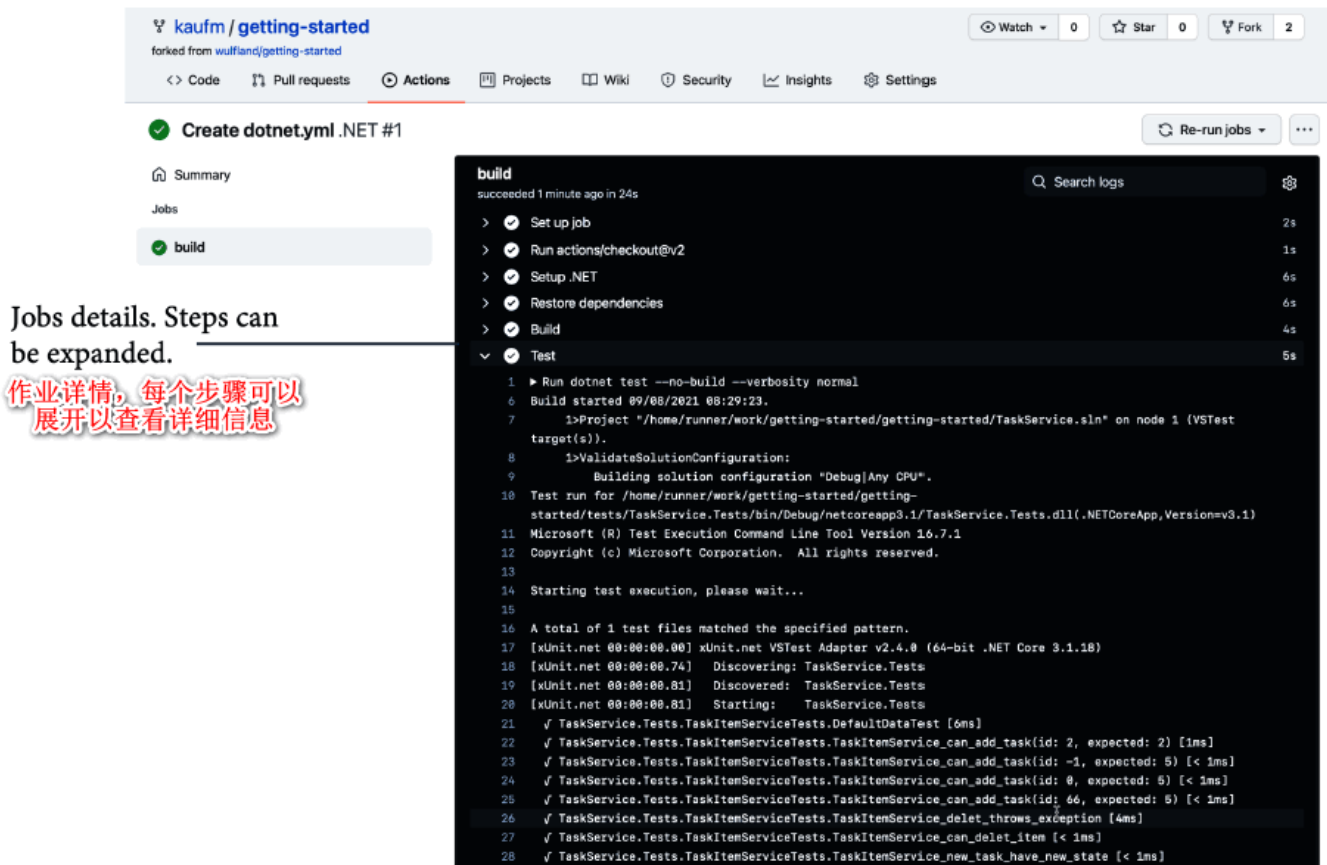


图6.5-作业和步骤详情

Figure 6.5 – Job and step details

如果读者使用的是其他语言，则可以克隆其他仓库。例如，该仓库使用的是Java和Maven(仓库链接：<https://github.com/MicrosoftDocs/pipelinesjava>)。

如果克隆了该仓库，当选择 workflow 模板时，首先向下滚动到 **Continuous integration workflows** 分类，接着点击 **More continuous integration workflows...** 按钮，最后选择 **Java with Maven**，这样 workflow 就能正常运行：

Continuous integration workflows

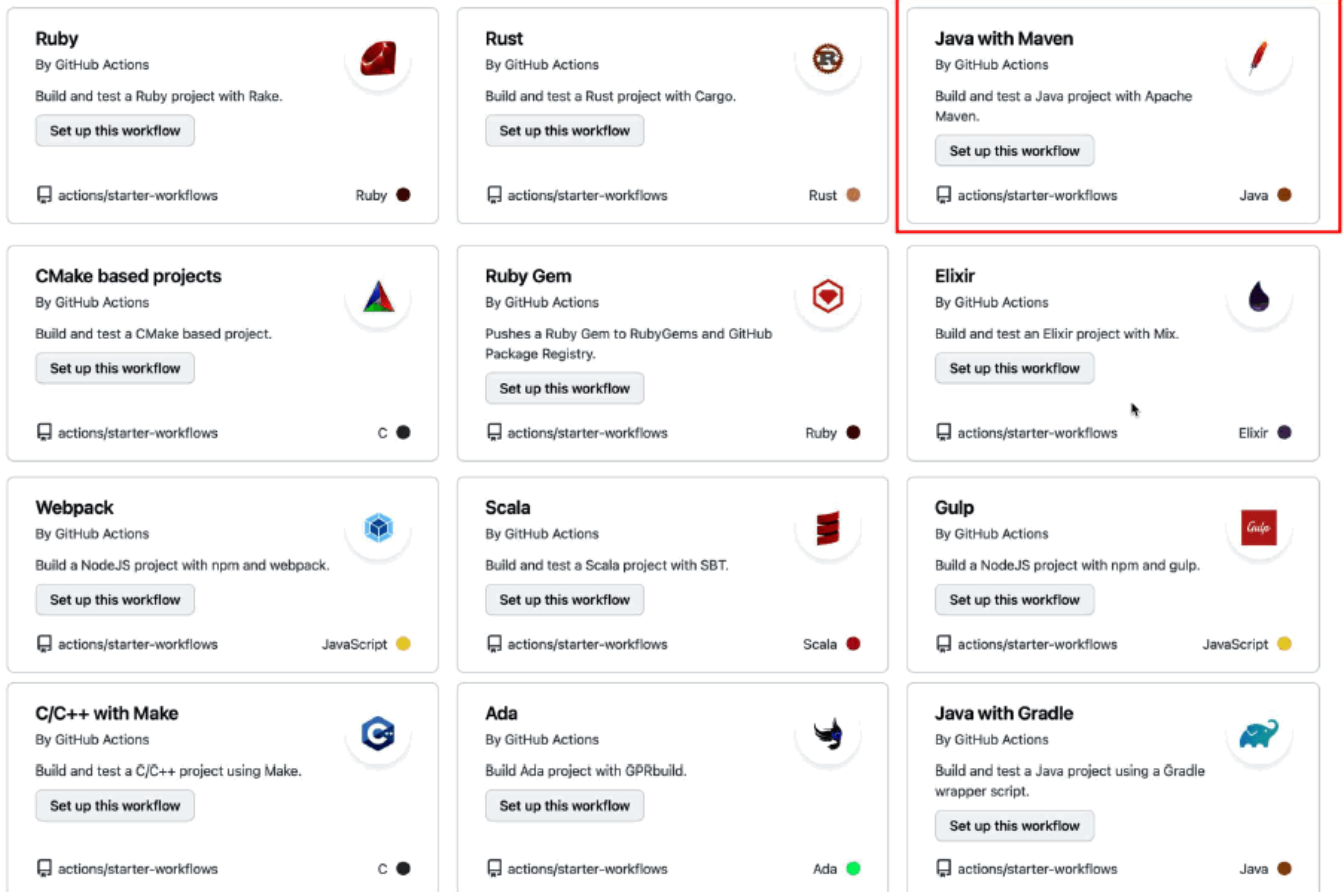


图6.6-其他持续集成模板，比如“Java with Maven”

Figure 6.6 – Other CI templates, such as "Java with Maven"

Github中有大量的模板，所以可以较容易地设置一个基本 workflow 来构建代码。

动手实践 - 第一个操作

GitHub Actions的强大之处在于它的可重用性，因此了解如何创建和使用操作十分重要。在这个动手实践中，读者将创建一个在Docker容器内运行的容器操作。

提示：

读者可以在 <https://docs.github.com/en/actions/creating-actions/creating-a-dockercontaineraction> 找到这个示例，并复制粘贴相关文本文件的内容。如果需要，还可以进入模板仓库 (仓库链接：<https://github.com/actions/container-action>) 并点击“**Use this template**”按钮，这将自动创建一个包含所有文件的仓库。

具体步骤如下：

1. 创建一个名为 *hello-world-docker-action* 的新仓库，接着将其克隆到读者个人主机上。
2. 打开终端并进入仓库目录：

```
$ cd hello-world-docker-action
```

3. 创建一个名为 *Dockerfile* 的文件，该文件不带扩展名，将以下代码添加到文件中：

```
# Container image that runs your code
FROM alpine:3.10

# Copies your code file from your action repository to
the filesystem path '/' of the container
COPY entrypoint.sh /entrypoint.sh

# Code file to execute when the docker container starts
up ('entrypoint.sh')
ENTRYPOINT ["/entrypoint.sh"]
```

在本示例中，该**Dockerfile**文件定义了一个基于Alpine Linux 3.1镜像的容器，并将 *entrypoint.sh* 文件复制到这个容器中。如果该容器被执行，就会运行*entrypoint.sh*。

4. 创建一个名为*action.yml*的新文件，文件内容如下：

```
# action.yml
name: 'Hello World'
description: 'Greet someone and record the time'
inputs:
  who-to-greet: # id of input
    description: 'Who to greet'
    required: true
    default: 'World'
outputs:
  time: # id of output
    description: 'The time we greeted you'
runs:
  using: 'docker'
  image: 'Dockerfile'
  args:
    - ${ inputs.who-to-greet }
```

*action.yml*文件定义了操作及该操作的输入和输出参数。

5. 接着创建*entrypoint.sh*脚本，该脚本将在容器中运行并可以调用其他二进制文件，往其中添加以下内容：

```
#!/bin/sh -l

echo "Hello $1"
time=$(date)
echo "::set-output name=time::$time"
```

输入参数作为参数传给脚本，并可以通过 *\$1* 访问。脚本中使用 *set-output* workflow 命令将 *time* 参数设置为当前时间。

6. 必须保证`entrypoint.sh`文件可以执行。在非Windows系统上，只需在终端中运行以下命令，然后添加并提交更改：

```
$ chmod +x entrypoint.sh
$ git add .
$ git commit -m "My first action is ready"
```

在Windows系统上，上述命令无效，但是当文件被添加到索引中时，可以将其标记为可执行文件：

```
$ git add .
$ git update-index --chmod=+x .\entrypoint.sh
$ git commit -m "My first action is ready"
```

7. 操作的版本控制使用Git标签实现。给操作添加v1版本标签并把所有更改推送到远程仓库：

```
$ git tag -a -m "My first action release" v1
$ git push --follow-tags
```

8. 操作现在可以在工作流中进行测试了。进入`getting-started`仓库(`.github/workflows/dotnet.yaml`)中的工作流目录并编辑该文件，删除`jobs`（第9行）下的所有内容，替换为：

```
hello_world_job:
  runs-on: ubuntu-latest
  name: A job to say hello
  steps:
    - name: Hello world action step
      id: hello
      uses: your-username/hello-world-action@v1
      with:
        who-to-greet: 'your-name'
    - name: Get the output time
      run: echo "The time was ${ steps.hello.outputs.time }"
```

工作流现在会调用操作（`uses`）并指向读者创建的仓库（`your-username/hello-world-action`），后面跟着标签（`@v1`）。它将读者的姓名作为一个输入参数传给该操作，并获取当前时间作为输出，然后将其输出到控制台。

9. 最后保存文件，工作流将自动运行。检查作业的详细信息，读者可以看到日志中输出的问候语和时间戳。

提示：

如果想尝试其他类型的操作，可以使用现有模板；如果想尝试JavaScript操作，可参考<https://github.com/actions/javascript-action>；如果想尝试TypeScript操作，可参考

<https://github.com/actions/typescript-action>。

复合操作更加容易，因为只需要一个action.yml文件（可参考<https://docs.github.com/en/actions/creatingactions/creating-a-composite-action>）。

处理操作是一样的，只是它们的创建方法不同。

GitHub Marketplace

用户可以使用GitHub Marketplace(<https://github.com/marketplace>)搜索想在工作流中使用的操作。由于发布操作并不难，Marketplace内已经有将近10,000个可用操作。用户可以按类别过滤操作或输入搜索条件来更快地找到合适的操作（见图6.7）：

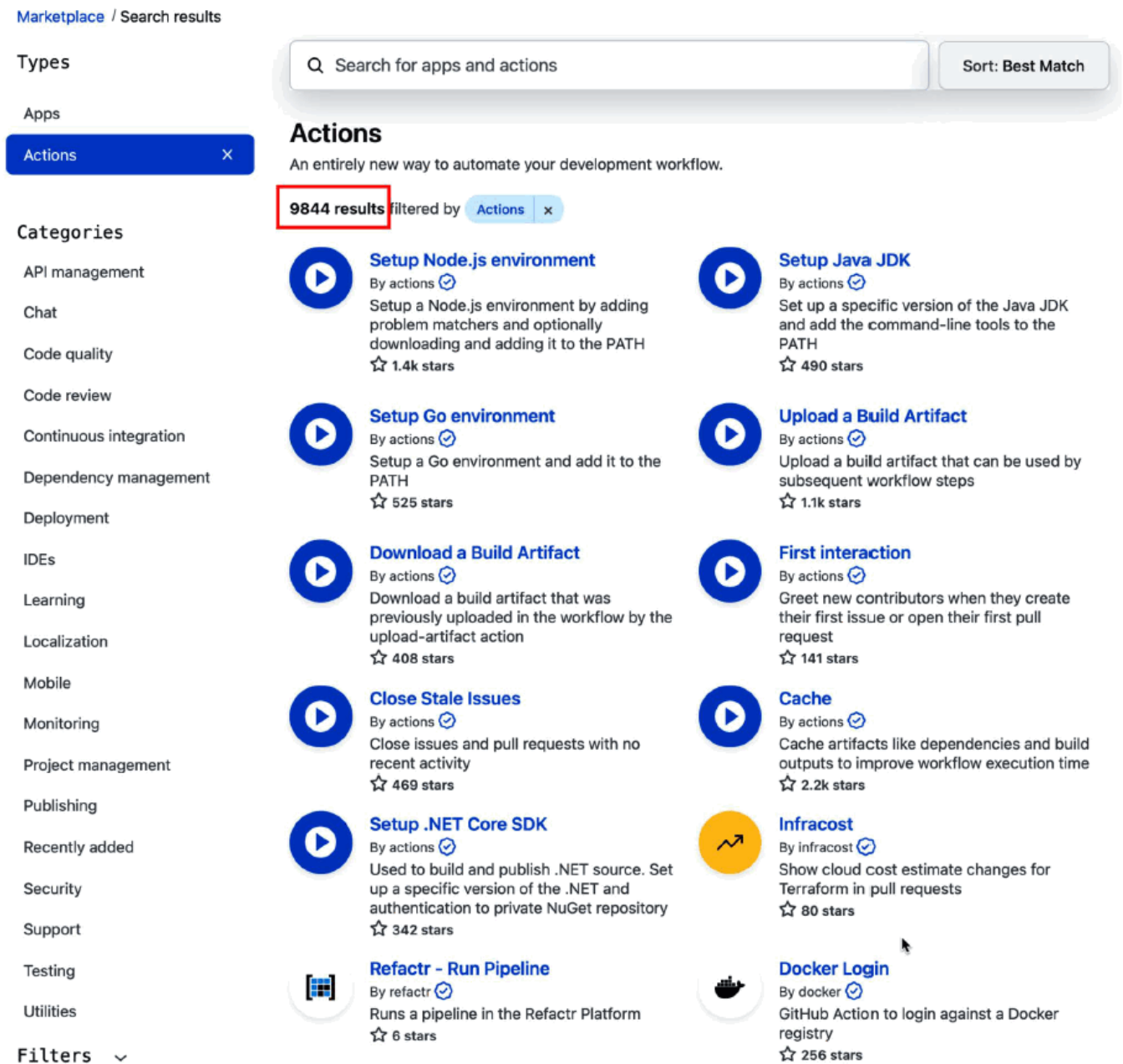


图6.7-GitHub Marketplace内有接近10,000个操作

Figure 6.7 – The marketplace contains nearly 10,000 actions

操作详情页会显示对应仓库的README文件内容和一些其他信息，用户可以查看完整的操作版本列表并了解如何使用当前版本的操作：

Marketplace / Actions / SonarCloud Scan


GitHub Action

SonarCloud Scan

v1.6 Latest version

Scan your code with SonarCloud

Using this GitHub Action, scan your code with [SonarCloud](#) to detects bugs, vulnerabilities and code smells in more than 20 programming languages!



SonarCloud is the leading product for Continuous Code Quality & Code Security online, totally free for open-source projects. It supports all major programming languages, including Java, JavaScript, TypeScript, C#, C/C++ and many more. If your code is closed source, SonarCloud also offers a paid plan to run private analyses.

Requirements

- Have an account on SonarCloud. [Sign up for free now](#) if it's not already the case!
- The repository to analyze is set up on SonarCloud. [Set it up](#) in just one click.

Usage

Project metadata, including the location to the sources to be analyzed, must be declared in the file `sonar-project.properties` in the base directory:

```
sonar.organization=<replace with your SonarCloud organization key>
sonar.projectKey=<replace with the key generated when setting up the project on SonarCloud>
# relative paths to source directories. More details and properties are described
# in https://sonarcloud.io/documentation/project-administration/narrowing-the-focus
sonar.sources=.
```

Use latest version

Verified creator

GitHub has verified that this action was created by [SonarSource](#).

[Learn more about verified Actions.](#)

Stars

☆ Star 325

Contributors

Categories

Code quality Security

Links

[SonarSource/sonarcloud-github-action](#)

[Pull requests](#) 9

[Report abuse](#)

SonarCloud Scan is not certified by GitHub. It is provided by a third-party and is governed by separate terms of service, privacy policy, and support documentation.

Switch to older versions and get information on how to use the action.

切换旧版本并了解如何使用这个操作

图6:8-Marketplace内的一个操作的详情页

Figure 6.8 – An action in the marketplace

发布操作到Marketplace上很容易。首先要确保该操作位于公开仓库中，其名称是唯一的，并且该仓库内有一个描述清楚的`README`文件。接着选择一个图标和图标颜色并将其添加到`action.yml`文件中：

```
branding:
  icon: 'award'
  color: 'green'
```

GitHub会自动检测`action.yml`文件并提供一个“Draft a release”按钮，点击此按钮会进入发布编辑页面。如果用户选择将此操作发布到GitHub Marketplace，则必须同意服务条款，并且Github将自动检查该操作是否包含所有必需的组件。在发布编辑页面可以为该发布选择合适的标签或创建一个新标签，并为其添加标题和描述信息：

ReleasesTags

Release Action

📄 action.yml

✎

✓ Everything looks good! You have all the required information.

✓ Name	Wulfland Action
✓ Description	Get started with Container actions
✓ Icon	award
✓ Color	green

📖 README

✎

✓ A README exists.

Primary Category

Code quality

Another Category — optional

Security

v2.0

🎯 Target: main

Excellent! This tag will be created from the target when you publish this release.

Release title

Tagging suggestions

It's common practice to prefix your version names with the letter v. Some good tag names might be v1.0 or v2.3.4.

If the tag isn't meant for production use, add a pre-release version after the version name. Some good pre-release versions might be v0.2-alpha or v5.9-beta.3.

Semantic versioning

If you're new to releasing software, we highly recommend reading about [semantic versioning](#).

图6.9:发布操作到Marketplace

Figure 6.9 – Publishing an action to the marketplace

最后发布操作或将其保存为草稿。

Marketplace使自动化变得简单，因为几乎所有事件都可以看作为一个操作，所以近年来它迅速发展。

总结

本章介绍了自动化的重要性，介绍了GitHub Actions——一种灵活、可扩展且适用于任何类型自动化的引擎。

下一章将学习不同的托管选项以及如何托管工作流运行器。

拓展阅读

若要了解本章相关话题的更多信息，可查阅以下参考资料：

- Humble J., & Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional.
- Forsgren, N., Humble, J., & Kim, G. (2018). *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations* (1st ed.) [E-book]. IT Revolution Press.
- YAML: <https://yaml.org/>
- GitHub Actions: <https://github.com/features/actions>、<https://docs.github.com/en/actions>
- GitHub Learning Lab: <https://lab.github.com>
- Workflow Syntax: <https://docs.github.com/en/actions/reference/workflow-syntax-for-github-actions>

- *GitHub Marketplace*: <https://github.com/marketplace>

第7章 运行工作流

本章将介绍运行工作流的不同选项，包括托管和自托管执行器，并将解释如何利用不同的托管选项处理混合云场景或硬件循环测试。同时展示如何设置、管理和扩展自托管执行器，并了解如何进行监视和故障排除。

以下是本章将涵盖的核心主题：

- 托管执行器
- 自托管执行器
- 使用运行器组管理访问
- 使用标签
- 扩展自托管执行器
- 监视和故障排除

托管执行器

在前面的章节已经使用过托管执行器(hosted runners)。托管执行器是Github托管的用于运行工作流的虚拟机。这种执行器适用于Linux、Windows和macOS操作系统。

隔离和隐私

工作流中的每个作业都在一个全新的虚拟机实例中执行，并且完全隔离。用户拥有完整的管理员访问权限（在Linux上为无密码sudo），而Windows机器上的用户账户控制（UAC）已被禁用。这意味着用户可以安装工作流中可能需要的任何工具（这仅会增加构建时间的成本）。

执行器还可以访问用户界面（UI）元素。这使用户可以在执行器内执行UI测试（例如Selenium）而无需通过另一个虚拟机进行操作。

硬件

Github托管的Linux和Windows系统的执行器运行在Standard_DS2_v2虚拟机的Microsoft Azure中。Windows和Linux虚拟机的硬件要求如下：

- 2核CPU
- 7GB RAM
- 14GB SSD 硬盘空间

MacOS版本的runner运行在Github的macOS云上，其硬件要求如下：

- 3核CPU
- 14GB RAM
- 14GB SSD硬盘空间

软件

表7.1为当前可用的镜像列表：

Virtual environment	YAML workflow label	Notes
Windows Server 2022	windows-2022	Currently in beta. The windows-latest label currently uses the Windows Server 2019 runner image and will switch to 2022 when it is out of beta.
Windows Server 2019	windows-latest or windows-2019	The windows-latest label currently points to this image.
Windows Server 2016	windows-2016	
Ubuntu 20.04	ubuntu-latest or ubuntu-20.04	The ubuntu-latest label currently points to this image.
Ubuntu 18.04	ubuntu-18.04	
Ubuntu 16.04	ubuntu-16.04	Deprecated and limited to existing customers only. Customers should migrate to Ubuntu 20.04.
macOS Big Sur 11	macos-11	
macOS Catalina 10.15	macos-latest or macos-10.15	The macos-latest label currently uses the macOS 10.15 runner image.

Table 7.1 – The currently available images for hosted runners

虚拟环境	YAML工作流标签	备注
Windows Server 2022	windows-2022	当前仍在beta阶段。windows-latest标签目前使用Windows Server 2019的runner镜像，当2022结束beta阶段后，将切换为Windows Server 2022
Windows Server 2019	windows-latest 或者 windows-2019	windows-latest标签当前指向该镜像
Windows Server 2016	windows-2016	
Ubuntu 20.04	ubuntu-latest或者 ubuntu-20.04	ubuntu-latest标签当前指向该镜像
Ubuntu 18.04	ubuntu-18.04	
Ubuntu 16.04	ubuntu-16.04	已弃用，仅限制已存在的用户使用。用户应迁移到Ubuntu 20.04
macOS Big Sur 11	macos-11	

虚拟环境	YAML工作流标签	备注
macOS Catalina 10.15	macos-latest or macos-10.15	windows-latest标签当前使用macOS 10.15的runner镜像

表7.1 托管执行器当前可用的镜像

用户可以在该网址中找到最新列表和所有包含的软件：<https://github.com/actions/virtual-environments>

如果想请求一个新的工具作为默认工具安装，用户也可以在该仓库提起issue，这个仓库还包括了执行器所有重大软件更新的公告。还可以使用Gihub仓库的“关注”功能来获取新版本创建的消息。

网络

托管执行器使用的IP地址在随时变化。用户可以通过Github的API来获取当前IP列表：

```
curl \
-H "Accept: application/vnd.github.v3+json" \
https://api.github.com/meta
```

更多信息可以查看网址：<https://docs.github.com/en/rest/reference/meta#get-github-meta-information>

通过这些信息，用户可以通过一个允许列表来阻止来自因特网的对内部资源的访问。但是请注意，所有人都可以使用托管runners和执行代码。阻止其他IP地址并不会使得资源安全。不要反对这些 IP 地址的内部系统，因为这些 IP 地址并不是以用户可以信任的方式从公共互联网访问的！这意味着必须对系统进行修补，并在适当的位置进行安全身份验证。如果情况不是这样，用户必须使用自托管的runner。

注意：

如果对Github组织或者企业账户使用IP地址允许列表，则不能使用Github托管执行器，而只能使用自托管执行器。

价格

托管执行器对于公共仓库的使用是免费的。根据用户的GitHub版本将拥有指定的存储量和每月免费构建分钟数。

GitHub edition	Storage	Minutes	Max concurrent jobs
GitHub Free	500 MB	2,000	20 (5 for macOS)
GitHub Pro	1 GB	3,000	40 (5 for macOS)
GitHub Free for organizations	500 MB	2,000	20 (5 for macOS)
GitHub Team	2 GB	3,000	60 (5 for macOS)
GitHub Enterprise Cloud	50 GB	50,000	180 (50 for macOS)

Table 7.2 – The included storage and build minutes for different GitHub editions

Github edition Github版本

Storage 存储空间

Minutes 分钟数

Max concurrent jobs 最大并发任务数量

表7-2 不同Github版本所包含的存储量和构建分钟数

如果用户通过Microsoft企业协议购买了GitHub Enterprise，则可以将Azure订阅ID连接到GitHub Enterprise帐户。这可以支付额外的GitHub Actions使用费用，以补充用户的GitHub版本中包含的内容。

在Windows和macOS执行器上运行的作业比Linux消耗更多的构建分钟数。Windows消耗的时间倍数是2，而macOS消耗的时间倍数是10。这意味着使用1,000分钟Windows系统将消耗您帐户中包含的2,000分钟，而使用1,000分钟macOS系统将消耗您帐户中包含的10,000分钟。

这是因为构建分钟数更昂贵。用户可以支付额外的分钟数，以补充GitHub版本中包含的分钟数。这些是每个操作系统的构建分钟费用：

- Linux:\$0.008
- macOS:\$0.08
- Windows:\$0.016

提示：

用户应该尽可能多地在工作流中使用 Linux，并将 macOS 和 Windows 减少到最低限度，以降低构建成本。另外，Linux 也有最好的启动性能。

额外存储的成本对所有的执行器来说是一样的——每GB 0.25美元。

如果用户是月结客户，帐户将有一个默认的支出限额：0美元。这可以防止使用额外的分钟或存储。如果使用发票支付，账户将默认有无限的支出限制。

如果用户设置的消费限额高于0美元，账户任何额外的分钟或存储将被计费，直到达到消费限额。

自托管执行器

如果用户需要比GitHub 托管执行器支持的硬件、操作系统、软件和网络访问更多的控制，可以使用自托管执行器。自托管执行器可以安装在物理机器、虚拟机或容器中。它们可以在本地或任何公共云环境中运行。

自托管运行程序允许从其他构建环境轻松迁移。如果用户已经有了自动构建，那么只需在机器上安装运行程序，代码就可以构建了。但如果构建机器仍然是手动维护的老式机器，或者机器位置远离开发人员，那么这不是一个永久性的解决方案。请记住，无论是托管在云端还是本地，构建和托管一个动态扩展环境需要专业知识，并且需要花费资金。因此，如果用户可以使用托管执行器，那么它总是更简单的选择。但是，如果需要自托管的解决方案，请确保使其具有弹性可扩展性。

注意：

拥有自己的执行器可以让用户在GitHub Enterprise Cloud内安全地构建和部署本地环境。这样，用户就可以使用混合模式运行GitHub，即可以在云端使用GitHub Enterprise，并使用托管执行器进行基本的自动化和部署到云环境，而使用自托管执行器来构建或部署托管在本地的应用程序。这比自行运行GitHub Enterprise Server和所有构建和部署的构建环境更加便宜和简单。

如果用户依赖硬件来测试软件，例如使用硬件在环测试，则必须使用自托管执行器。这是因为没有办法将硬件连接到GitHub托管执行器。

执行器软件

运行器是开源的，并可以在<https://github.com/actions/runner>上找到。它支持Linux、macOS和Windows上的x64处理器架构。它也支持Linux上的ARM64和ARM32架构。该运行器支持许多操作系统，包括Ubuntu、Red Hat Enterprise Linux 7或更高版本、Debian 9或更高版本、Windows 7/8/10和Windows Server、macOS 10.13或更高版本等。有关完整列表，请参阅文档：<https://docs.github.com/en/actions/hosting-your-own-runners/about-self-hosted-runners#supported-architectures-and-operating-systems-for-self-hosted-runners>。

运行器会自动更新，因此用户不需要管理更新。

运行器和GitHub之间的通信

运行器软件使用出站连接，在HTTPS 443端口上长轮询GitHub。它打开一个50秒的连接，如果没有收到响应，则会超时。

用户必须确保机器具有网络访问以下URL：

```
github.com
api.github.com
*.actions.githubusercontent.com
github-releases.githubusercontent.com
github-registry-files.githubusercontent.com
codeload.github.com
*.pkg.github.com
pkg-cache.githubusercontent.com
pkg-containers.githubusercontent.com
pkg-containers-az.githubusercontent.com
*.blob.core.windows.net
```

用户不必在防火墙上打开任何入站端口。所有通信都通过客户端运行。如果对 GitHub 组织或企业使用 IP 地址允许列表，必须将自托管执行器的 IP 地址范围添加到该允许列表。

在代理服务器上使用自托管执行器

如果用户需要在代理服务器上运行自托管执行器，需要注意可能会导致很多问题。执行者本身可以正常通信，但是，包管理、容器注册表以及执行器执行需要访问资源的所有内容都会产生开销。请尽可能避免使用代理服务器。如果不得不在代理服务器上运行工作流，用户可以使用以下环境变量配置执行器：

- `https_proxy`：这包括 HTTPS（端口443）流量的代理 URL。还可以包括基本身份验证（例如 `https://user:password@proxy.local`）
- `http_proxy`：这包括 HTTP（端口80）流量的代理 URL。还可以包括基本身份验证（例如 `http://user:password@proxy.local`）
- `no_proxy`：这包括应该绕过代理服务器的逗号分隔的主机列表。

如果更改环境变量，必须重新启动执行器才能使更改生效。

另一种代替使用环境变量的方法是使用 .env 文件。在执行器的应用程序文件夹中保存一个名为 .env 的文件，之后语法与环境变量相同：

```
https_proxy=http://proxy.local:8081
no_proxy=example.com,myserver.local:443
```

接下来看一下如何将自托管执行器添加到 GitHub。

添加自托管运行器到 GitHub

用户可以在 GitHub 的不同层面添加执行器：仓库、组织或企业。如果在仓库级别添加执行器，它们仅限于该单个仓库。组织级别的执行器可以处理组织中多个仓库的作业，企业级别的执行器可以分配给企业中的多个组织。安装执行器并在用户的 GitHub 实例中注册很容易。只需转到用户希望添加执行器的级别的 Settings | Actions | Runners。然后选择操作系统和处理器架构（参见图 7.1）：

Runners / Create self-managed runner

Adding a self-managed runner requires that you download, configure, and execute the GitHub Actions Runner. By downloading and configuring the GitHub Actions Runner, you agree to the [GitHub Terms of Service](#) or [GitHub Corporate Terms of Service](#), as applicable.

Runner image

☐  macOS

☒  Linux

☐  Windows

Architecture

x64

Figure 7.1 – Installing a self-hosted runner

图7.1 - 安装自托管执行器

这个脚本为用户生成了以下内容：

1. 下载和解压程序
2. 使用相应配置程序
3. 启动程序

脚本的第一部分始终创建一个名为 actions-runner 的文件夹，然后将工作目录更改为该文件夹：

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

在 Linux 和 macOS 上，使用 curl 命令下载最新的 runner 包；在 Windows 上使用 Invoke-WebRequest。

```
# Linux and macOS:
$ curl -o actions-runner-<ver>.tar.gz -L
https://github.com/actions/runner/releases/download/<ver>/actions-runner-
<ver>.tar.gz

# Windows:
$ Invoke-WebRequest -Uri
https://github.com/actions/runner/releases/download/<ver>/actions-runner-
<ver>.zip -OutFile actions-runner-<ver>.zip
```

为确保安全，验证下载压缩包的哈希值，以确保包没有被破坏：

```
# Linux and macOS:
$ echo "<hash> actions-runner-<ver>.tar.gz" | shasum -a 256 -c

# Windows:
$ if((Get-FileHash -Path actions-runner-<ver>.zip -Algorithm
SHA256).Hash.ToUpper() -ne '<hash>'.ToUpper()){ throw 'Computed checksum
did not match' }
```

然后，执行器从 ZIP/TAR 文件中被提取出来：

```
# Linux and macOS:
$ tar xzf ./actions-runner-<ver>.tar.gz
# Windows:
$ Add-Type -AssemblyName System.IO.Compression.FileSystem ;
[System.IO.Compression.ZipFile]::ExtractToDirectory("$PWD/actions-runner-
<ver>.zip", "$PWD")
```

配置是使用 config.sh / config.cmd 脚本完成的，URL 和令牌由 GitHub 自动创建：


```
# Linux and macOS:
$ ./config.sh --url https://github.com/org --token token

# Windows:
$ ./config.cmd --url https://github.com/org --token token
```

配置要求选择执行器组（默认为 Default 组）、执行器名称（默认为机器名称）以及其他标签。默认标签用于描述自托管状态，操作系统和处理器架构（例如分别是自托管，Linux 和 X64）。默认的工作文件夹是 `_work`，不应更改。在 Windows 上，用户还可以选择将 action runner 作为服务运行。在 Linux 和 macOS 上，必须在配置后使用另一个脚本安装服务：

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

如果不想以服务方式运行执行器，可以使用 `run` 脚本交互式运行它：

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

如果执行器正在运行，用户可以在“Settings|Actions|Runners”中看到其状态和标签（见图7.2）：

Runners

[New self-hosted runner](#)

Host your own runners and customize the environment used to run jobs in your GitHub Actions workflows. [Learn more about self-hosted runners.](#)



Runners				Status
 linux-runner	self-hosted	linux	x64	● Idle
 windows-runner	self-hosted	x64	windows	● Idle

Figure 7.2 – Self-hosted runners with their tags and status

图7.2 - 自托管执行器及其标签和状态

现在来学习如何从GitHub中删除这些自托管执行器。

删除自托管执行器

如果用户想重新配置或从GitHub中删除执行器，必须使用带有“删除”选项的config脚本。通过单击其名称打开执行器的详细信息，单击“Remove”按钮（参见图7.3），将为您生成脚本和令牌。

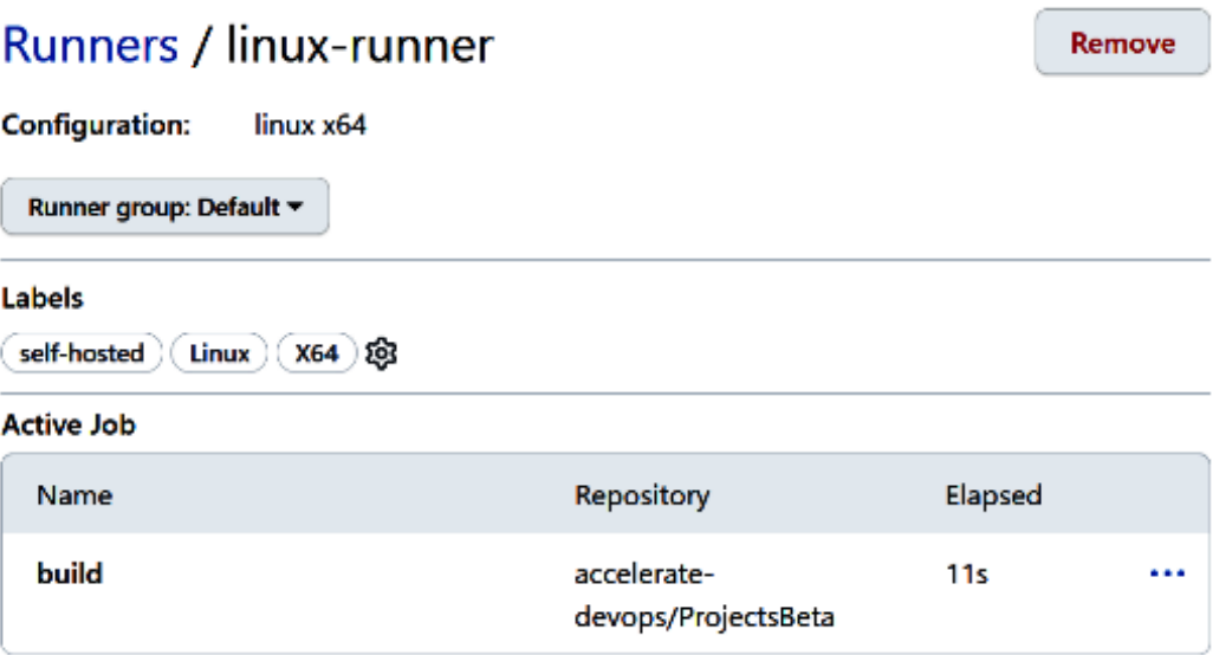


Figure 7.3 – The runner details

图7.3 - 执行器详细信息

不同操作系统的脚本如下：

```
# Linux and macOS
./config.sh remove --token <token>
# Windows
./config.cmd remove --token <token>
```

记住永远要在销毁机器之前先删除运行器！如果忘记这样做，用户可以在“Remove”对话框中使用“Force remove this runner”按钮，但这应该仅作为最后的方式。

管理访问权限

如果在组织或企业级别注册用户的执行器，则使用执行器组来控制对自托管执行器的访问。企业管理员可以配置访问策略，以控制企业中的哪些组织有访问执行器组的权限，而组织管理员可以配置访问策略，以控制组织中的哪些仓库有访问执行器组的权限。每个企业和每个组织都有一个默认的执行器组，名为Default，不能被删除。

注意：
一个执行器只能在一个执行器组中。

在进行管理访问时，打开企业级别的Policies或组织级别的Settings，并在菜单中找到 Actions|Runner Groups。用户可以创建新的执行器组或单击现有组件以调整其访问设置。根据级别是企业还是组织，用户可以允许访问特定组织或仓库（参见图 7.4）。

Runner Groups / Default

Group name

Save

Repository access

All repositories ▾

☐ Allow public repositories

Runners can be used by public repositories. Allowing self-hosted runners on public repositories and allowing workflows on public forks introduces a significant security risk [Learn more](#)

New runner



Runners	Status
 linux-runner self-hosted linux x64	● Idle
 windows-runner self-hosted x64 windows	● Idle

Figure 7.4 – Options for runner groups

图7.4 - 执行器组设置

警告

默认情况下禁用访问公共仓库。请不要更改该设置！不应该在公共存储库中使用自托管执行器！这会带来风险，因为Forks可能会在执行器上执行恶意代码。如果需要公共仓库的自托管执行器，请确保使用没有访问内部资源的临时和加固执行器。一种可能的情况是，用户需要开源项目的特殊工具，并且它在托管执行器上安装时间过长。但这是很少见的情况，用户应该尽量避免它们。

当用户注册一个新的 runner 时，需要为它输入 runner 组的名称，也可以在配置脚本中作为参数传递：

```
./config.sh --runnergroup <group>
```

现在读者已经学会了如何使用执行器组管理访问权限，接下来将学习使用标签。

使用标签

GitHub Actions 通过搜索正确的标签来将工作流匹配到用户的执行器。标签在用户注册执行器时被应用，也可以在配置脚本中作为参数传递：

```
./config.sh --labels self-hosted,x64,linux
```

用户可以在执行器的详情中稍后修改标签，并点击标签旁边的齿轮图标创建新标签（参见图 7.5）。

Runners / linux-runner

Remove

Configuration: linux x64

Runner group: Default ▾

Labels

self-hosted Linux X64 ⚙️

Active Job

Find or create a label...

Enter a new label name to create it

Unassigned labels will be removed periodically

Figure 7.5 – Creating new labels for a runner

图7.5 - 为执行器创建新标签

如果用户的工作流程有特定的需求，可以为它们创建自定义标签。自定义标签的一个例子可能是为工具（如 matLab）添加标签，或者添加必需的GPU访问标签。

所有自托管执行器默认都有自托管标签。

要在工作流中使用执行器，用户需要在标签形式中指定需求：

```
runs-on: [self-hosted, linux, X64, matlab, gpu]
```

这样，用户的工作流程就可以找到满足必要需求的相应运行器。

扩展自托管执行器

在现有构建机器上安装action runner可以轻松迁移到 GitHub。但这不是长期解决方案！如果用户无法使用托管执行器，则应该自己构建弹性扩展的构建环境。

瞬时执行器

如果用户为构建机器或容器构建了弹性扩展解决方案，应该使用瞬时执行器。这意味着使用来自空白镜像的虚拟机或 Docker 镜像并安装临时运行程序，每次运行后，一切都会被清除。这种情况下，不推荐使用弹性扩展的持久性执行器。

要配置执行器为瞬时执行期，需要向配置脚本传递以下参数：

```
$ ./config.sh --ephemeral
```

使用 GitHub Webhooks 扩展

用户可以使用 GitHub Webhooks 调整虚拟环境的配置。如果有新工作流到达队列，则 webhook 的 workflow_job 会调用 queued 操作。用户可以使用此事件创建新的构建机器并将其添加到机器池中。如果工作流运行完成，则 workflow_job 会调用 completed 操作。用户可以使用此事件进行清理和销毁机器。有关更多信息，请参阅文档 https://docs.github.com/en/developers/webhooks-and-events/webhooks/webhook-events-and-payloads#workflow_job。

现有解决方案

在 Kubernetes、AWS EC2 或 OpenShift 中构建弹性虚拟构建环境超出了本书的范围。GitHub 本身并不提供此解决方案，但如果用户希望利用它们，则有很多开源解决方案可以节省时间和精力。Johannes Nicolai (@jonico) 对所有解决方案进行了整理。用户可以在 <https://github.com/jonico/awesome-runners> 中找到相关资料。矩阵在 GitHub 页面中更容易阅读，因此读者可能更喜欢访问 <https://jonico.github.io/awesome-runners>。矩阵根据目标平台、是否具有 GitHub Enterprise 支持、自动缩放功能、清理因素等标准进行比较。

提示：

使用自定义镜像构建和运行可扩展的构建环境需要大量的时间和精力，这些时间和精力可以用于其他事情。使用托管执行器是更便宜和更可持续的解决方案。请确保真的需要自己的平台进行这样的投资。通常，还有其他选择搭载自己的执行器，例如将自己的 Docker 镜像带入 GitHub Actions 或使用机器人自动部署到本地资源。

监控和故障排除

如果用户有自托管执行器的问题，以下内容可以帮助进行故障排除。

检查运行者的状态

用户可以在“Settings|Actions|Runners”下检查执行器的状态。执行器的状态可以是空闲、活动或离线。如果执行器状态是离线，则机器可能已经关闭或未连接到网络，或者自托管执行器应用程序可能未在机器上运行。

查看应用程序日志文件

日志文件保存在运行者的根目录中的“_diag”文件夹中。用户可以在其中查看运行者应用程序日志文件。应用程序日志文件名以 Runner_ 开头，并追加 UTC 时间戳：

```
Runner_20210927-065249-utc.log
```

查看作业日志文件

作业日志文件也位于 _diag 中。每个作业都有自己的日志。应用程序日志文件名以 Worker_ 开头，也有 UTC 时间戳：

```
Worker_20210927-101349-utc.log
```

检查服务状态

如果执行器以服务的形式运行，用户可以根据您的操作系统检查服务状态。

Linux

在 Linux 上，可以从 runner 文件夹中的 .service 文件中获取服务的名称。使用 journalctl 工具监视 runner 服务的实时活动：

```
$ sudo journalctl -u $(cat ~/actions-runner/.service) -f
```

可以在 /etc/systemd/systemd/ 下检查和自定义服务的配置：

```
$ cat /etc/systemd/system/$(cat ~/actions-runner/.service)
```

macOS

在 macOS 上，可以使用 svc.sh 脚本检查服务的状态：

```
$ ./svc.sh status
```

上述脚本的输出包含服务名称和进程 ID。

要检查服务配置，请找到以下位置的文件：

```
$ cat /Users/<user_name>/Library/LaunchAgents/<service_name>
```

Windows

在 Windows 上，您可以使用 PowerShell 检索有关您的服务的信息：

```
$ Get-Service "action*"
```

使用 EventLog 监视您的服务的最近活动：

```
Get-EventLog -LogName Application -Source ActionsRunnerService
```

监控执行器更新过程

执行器会自动更新。如果更新失败，执行器将无法运行工作流。用户可以在`_diag` 目录中的 `Runner_*` 日志文件中检查它的更新活动。

案例研究

Tailwind Gears的两个试点团队在新平台上开始了他们的第一个Sprint周期。他们首先要自动化的是构建过程，以便在合并前对所有pull请求进行构建。Tailwind Gears尽量多地使用GitHub托管的执行器。大部分软件都能很好地构建。然而，一些使用旧版编译器的C代码以及在当前构建机器上安装的其他依赖项存在问题。该代码目前在两个由开发人员自行维护的本地Jenkins服务器上构建。这些服务器也连接到用于硬件在环测试的硬件上。为了方便过渡，在这些机器上安装了自托管执行器，并且构建正常运行。IT部门本来就想摆脱本地服务器，因此他们与GitHub合作伙伴一起构建了一个弹性、可扩展、基于容器的解决方案，可以运行自定义镜像，并可以访问附加硬件。

总结

本章了解了运行工作流的两种托管选项：

- GitHub托管的执行器
- 自托管的执行器

并解释了自托管执行器如何使您能够在混合云场景中运行GitHub。了解了如何设置自托管执行器以及如何找到帮助构建自己的弹性可扩展构建环境的信息。

下一章将了解如何使用GitHub Packages管理代码依赖关系。

进一步阅读

有关本章主题的更多信息，可以参考以下资源：

- 使用GitHub托管的执行器：<https://docs.github.com/en/actions/using-github-hosted-runners>
- 自托管执行器：<https://docs.github.com/en/actions/hosting-your-own-runners>
- awesome-runners-大量比较矩阵中优秀的自托管GitHub动作执行器解决方案的列表：
<https://jonico.github.io/awesome-runners>