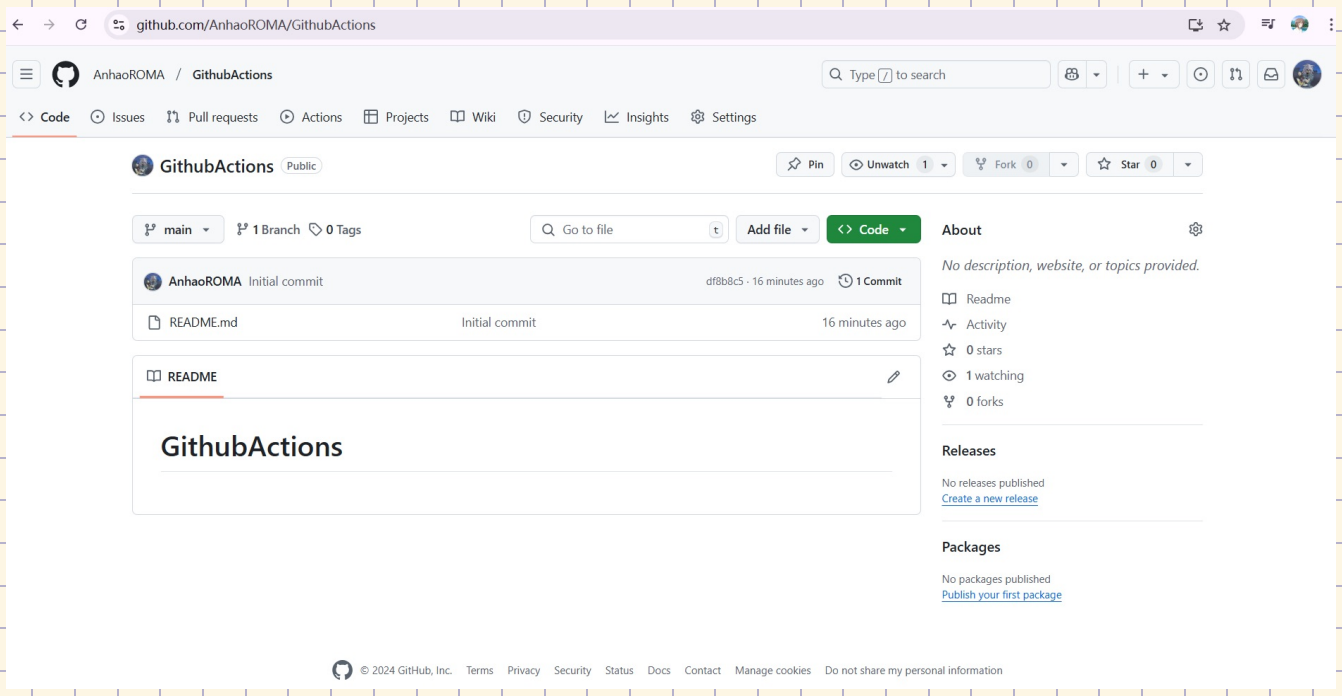
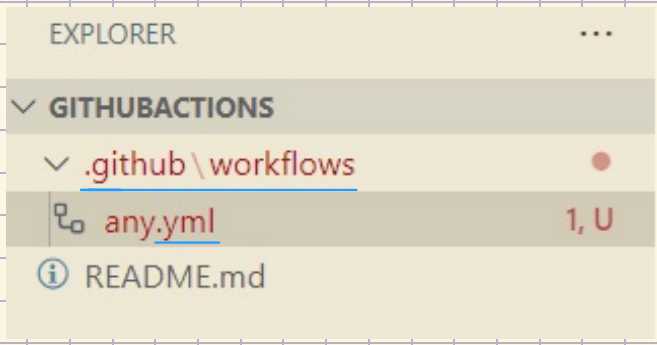
GitHub Actions

"Hello, world."

在读专业的书本之前，不妨先做个小实验。

1 在自己的 GitHub 上创建一个 repo，并克隆到本地。



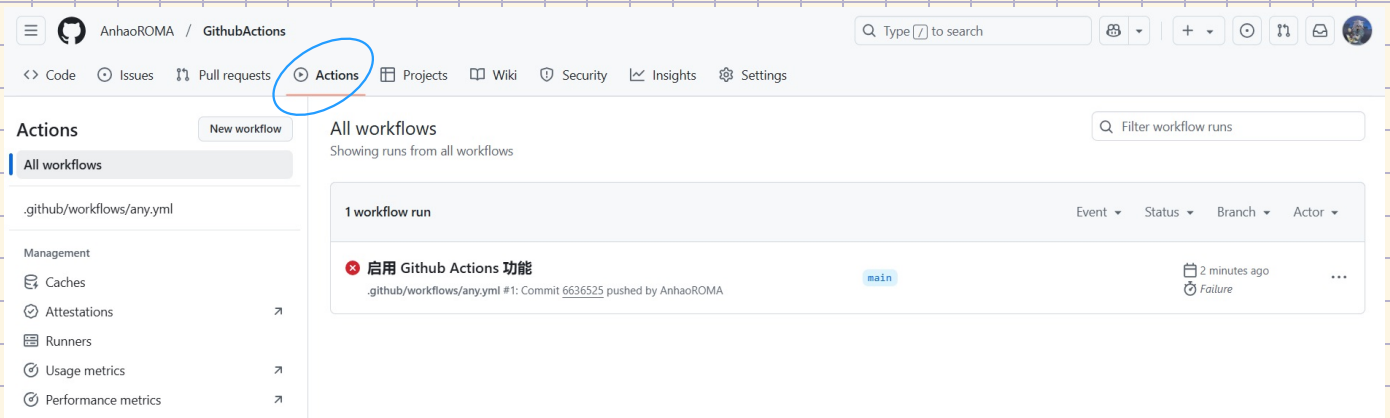


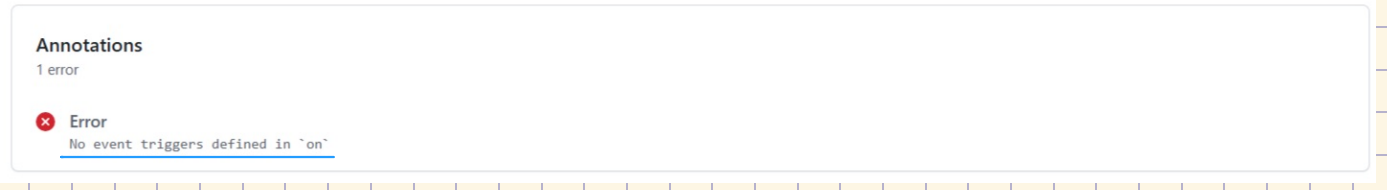
2 在本地的工作路径下创建文件 `.github/workflows/*.yml`。

```
PS E:\GithubActions> git add .github/*
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   .github/workflows/any.yml

PS E:\GithubActions> git commit -m "启用 Github Actions 功能"
[main 6636525] 启用 Github Actions 功能
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 .github/workflows/any.yml
PS E:\GithubActions> git push origin main
Enumerating objects: 6, done.
```

3 提交并推送。

4 此时在"Actions"栏发现报错。





5 根据报错信息，可知：需定义"何时(事)触发工作流"。

any.yml 1 ×

.github > workflows > any.yml

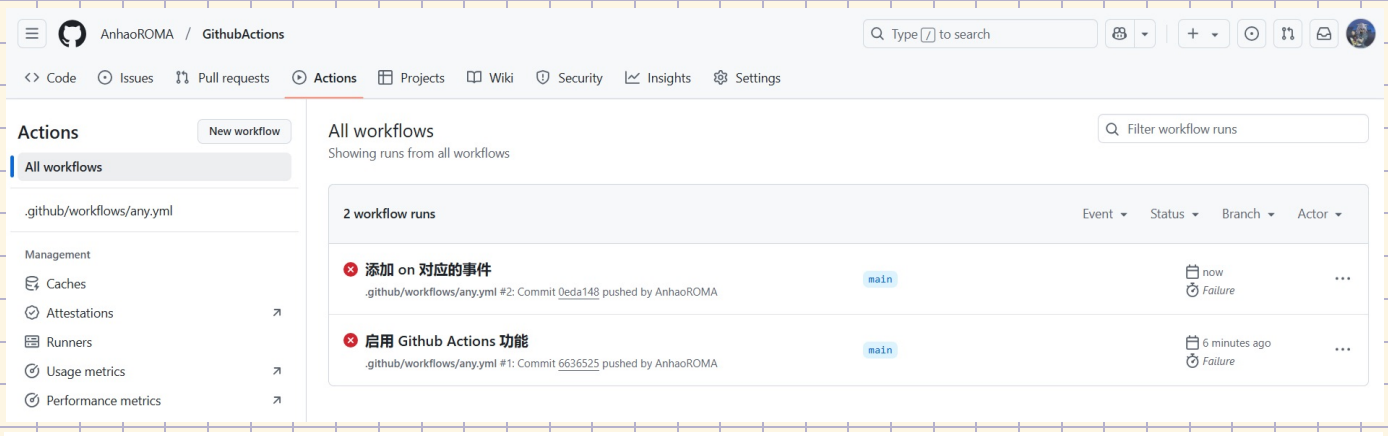
```yaml
on: push
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS E:\GithubActions> git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   .github/workflows/any.yml

no changes added to commit (use "git add" and/or "git commit -a")
PS E:\GithubActions> git add .github/workflows/any.yml
PS E:\GithubActions> git commit -m "添加 on 对应的事件"
[main 0eda148] 添加 on 对应的事件
 1 file changed, 1 insertion(+)
PS E:\GithubActions> git push origin main
Enumerating objects: 9, done.
```

6 定义"何时(事)触发工作流"：
当有内容 push 到 GitHub 仓库时，
触发工作流。

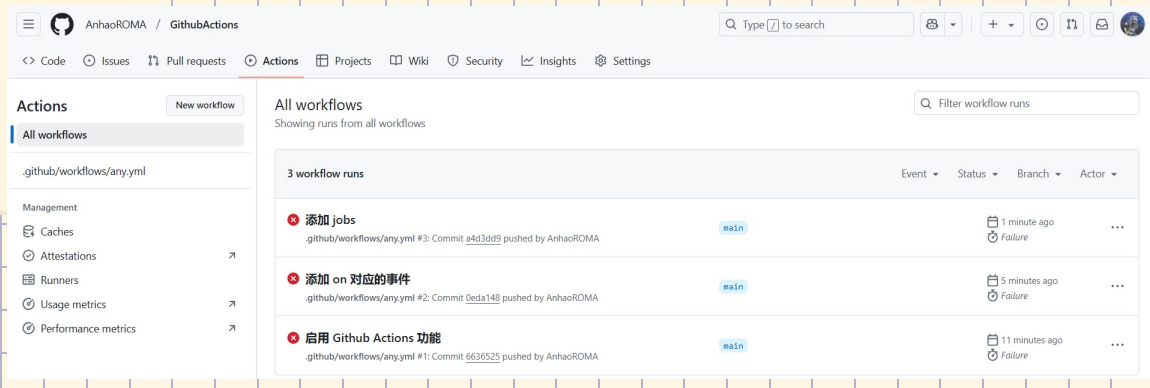
7 依旧报错。



8 报错，除了"何时(事)触发工作流"，还需要明确"干什么事情"。

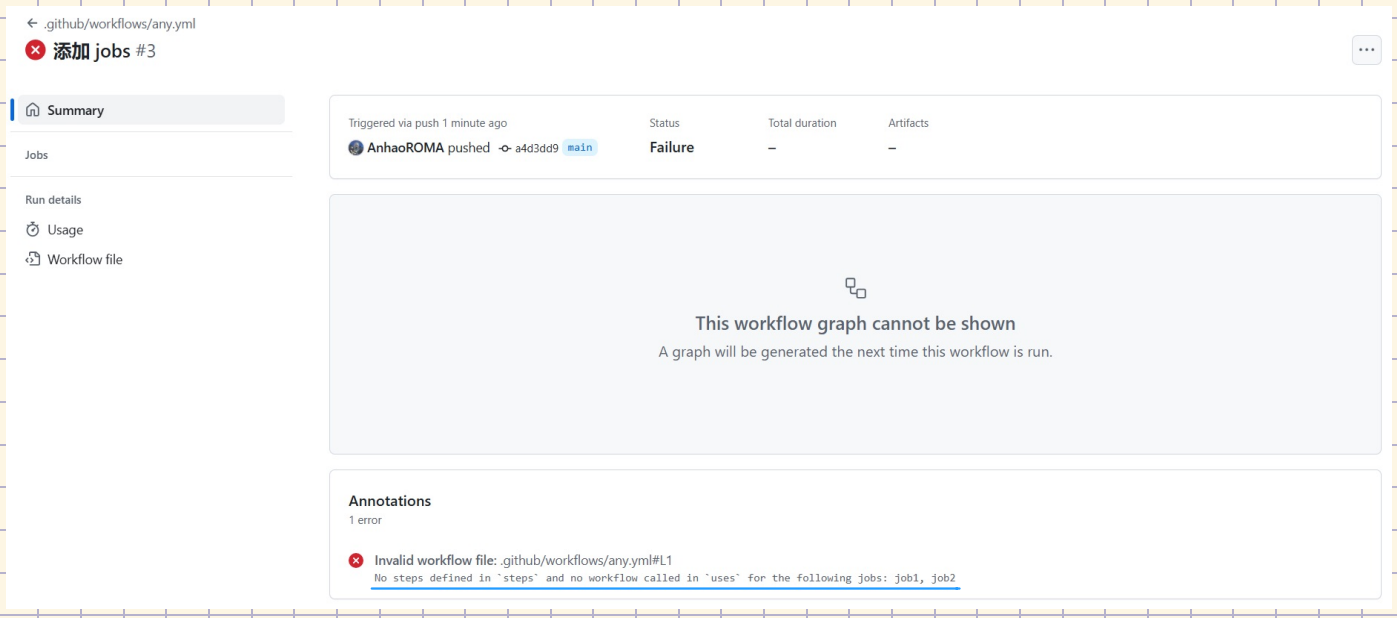
.github > workflows > any.yml

```yaml
on: push

jobs:
    job1:

    job2:
```



9 暂时仅仅留个位。

10 不出意外地收到报错。

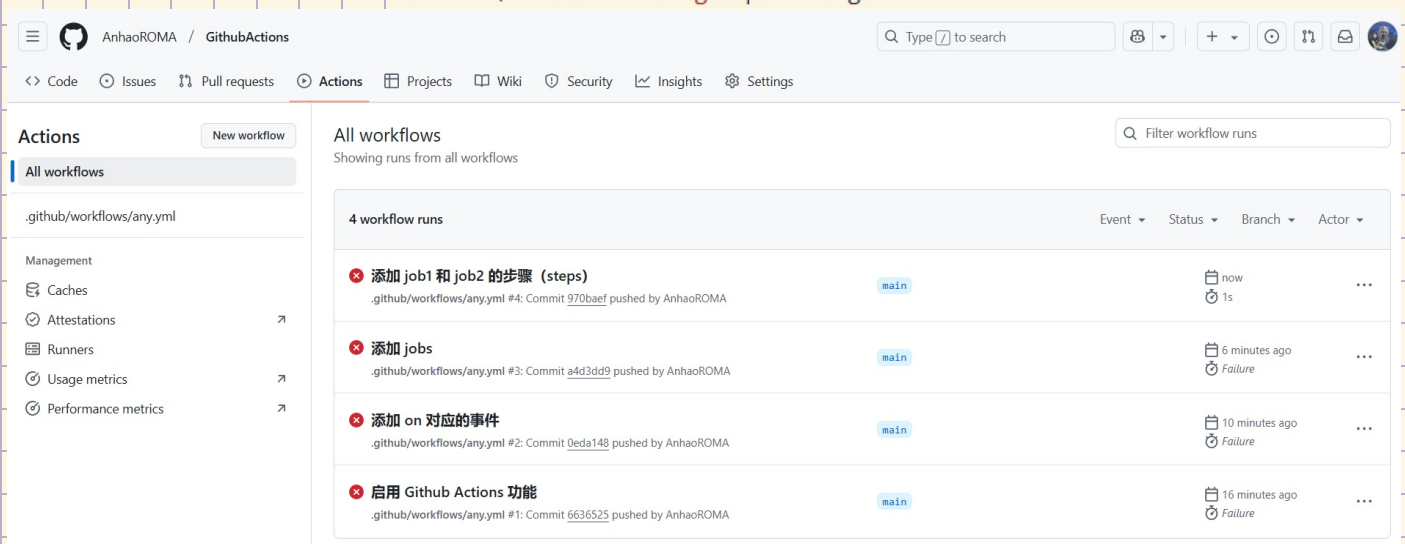


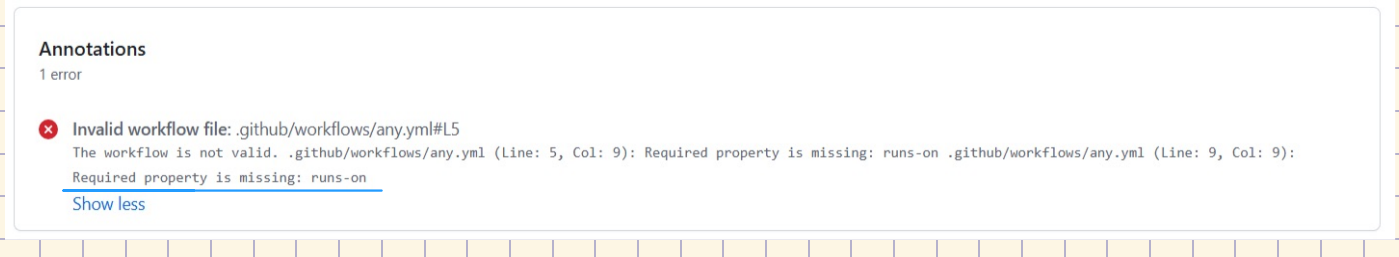
11 需在 jobs 中编写步骤。

12

pwd ; ls
echo "hello, CI/CD"

13 还是不行。





14 至少得告诉GitHub选择什么操作系统嘛！

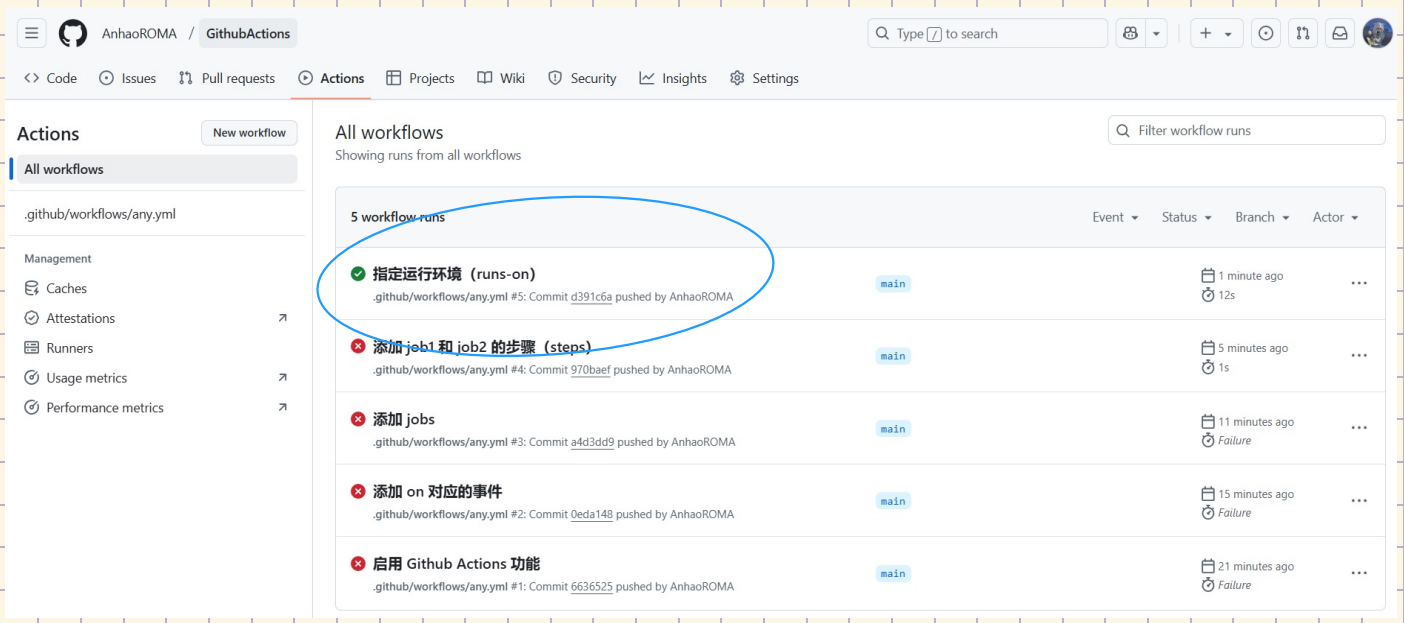
.github > workflows > any.yml

```yaml
on: push

jobs:
    job1:
        runs-on: ubuntu-latest
        steps:
        - run: pwd
        - run: ls
    job2:
        runs-on: ubuntu-latest
        steps:
        - run: echo "hello, CI/CD"
```

选择最新的Ubuntu。

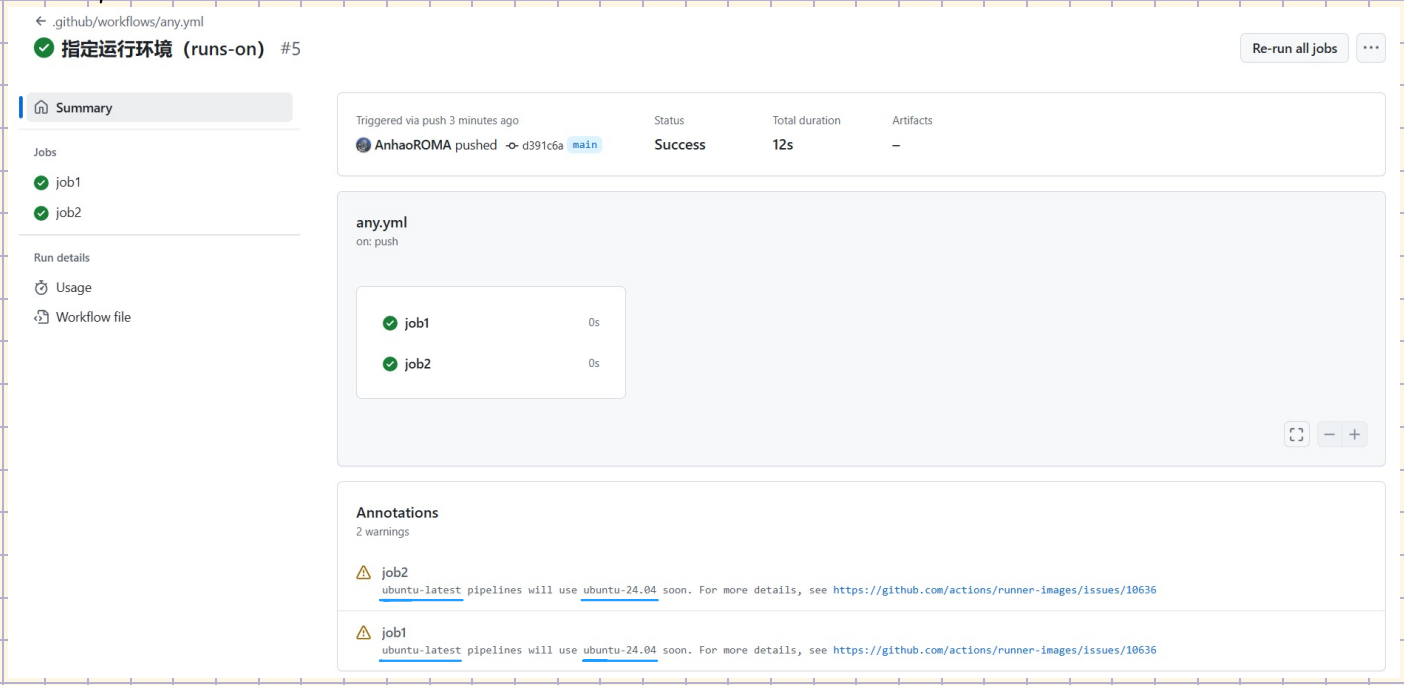
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

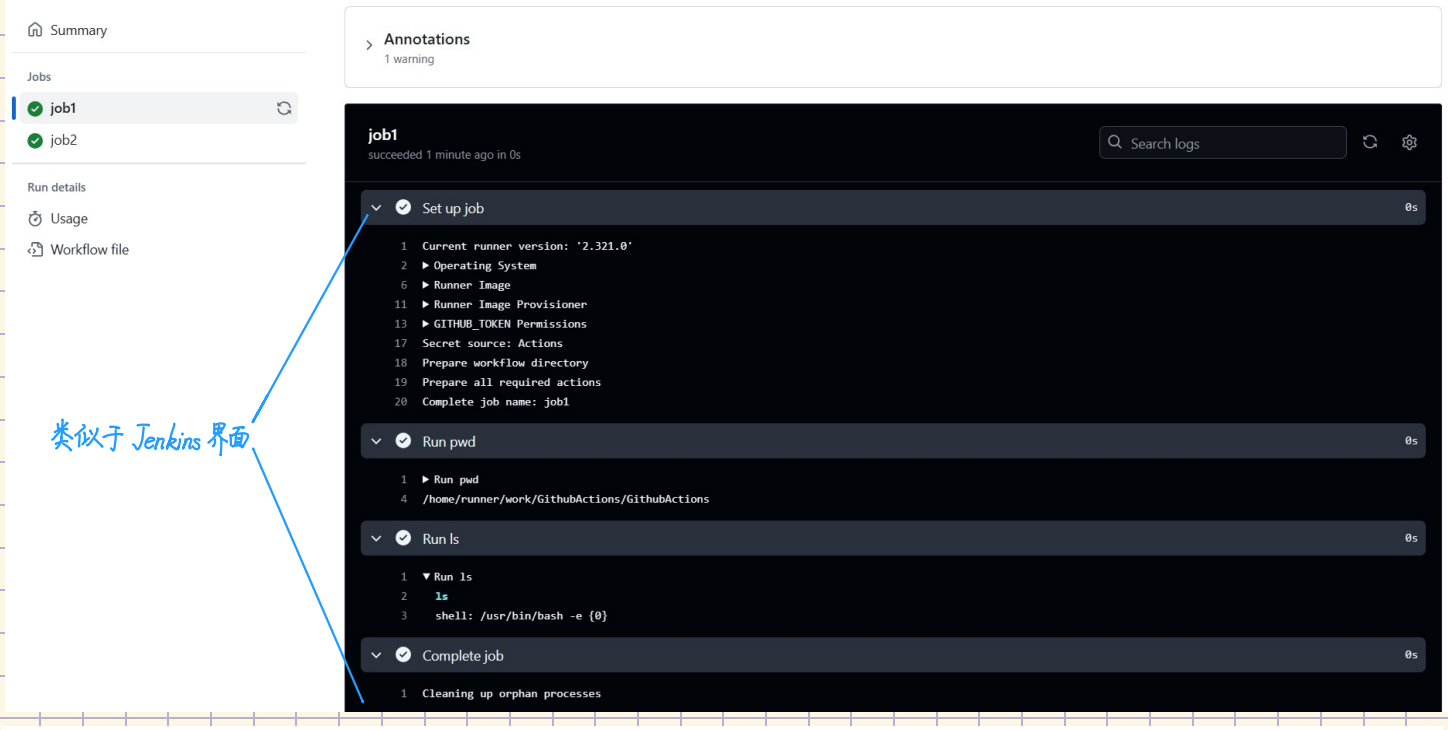
```
PS E:\GithubActions> git commit -a -m "指定运行环境（runs-on）"
[main d391c6a] 指定运行环境（runs-on）
 1 file changed, 2 insertions(+)
PS E:\GithubActions> git push origin main
Enumerating objects: 9, done.
```



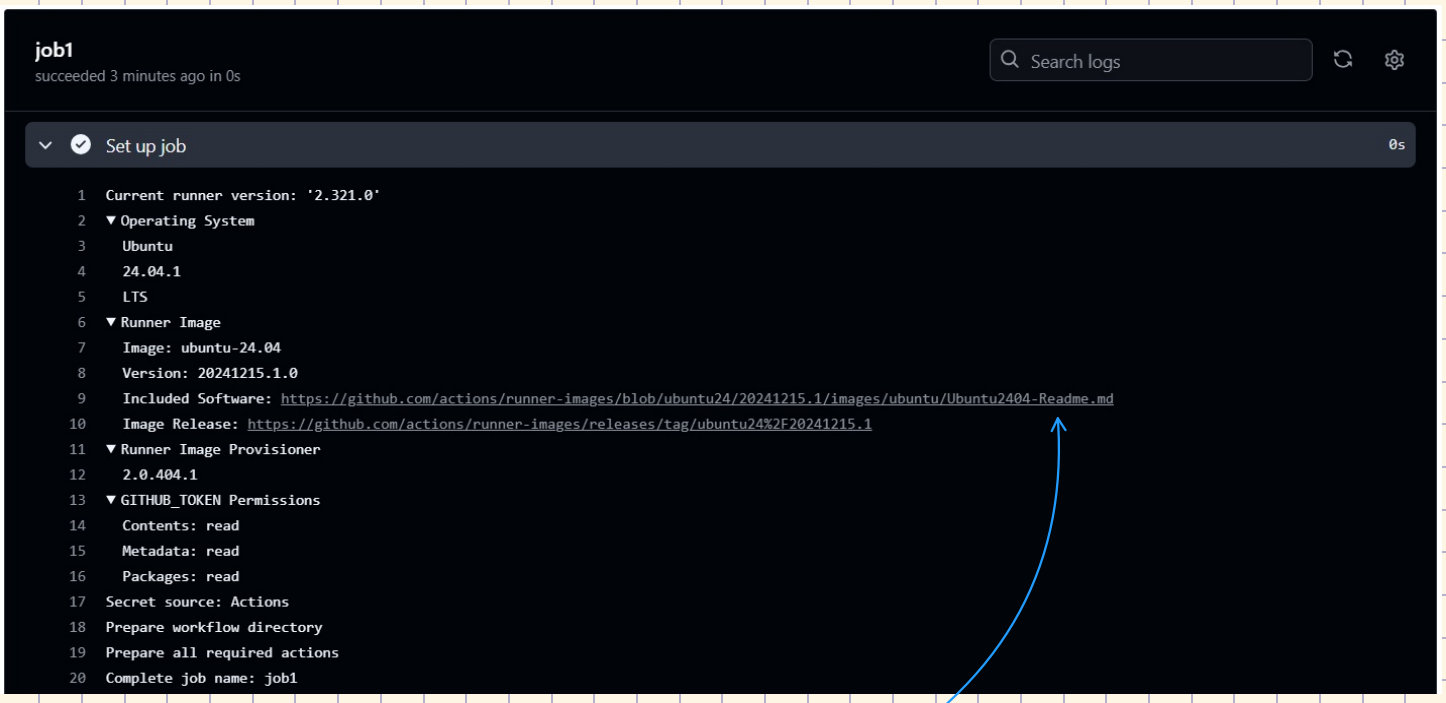
15 终于！

16 点开看看细节。





类似于 Jenkins 界面



由此可查看已预装软件目录

17 在这个项目中写一个 Go 程序并生成对应的可执行文件。

因为 .exe 文件是 Windows 系统的可执行文件后缀名，

所以 runs-on : windows-latest 。

any.yml × main.go

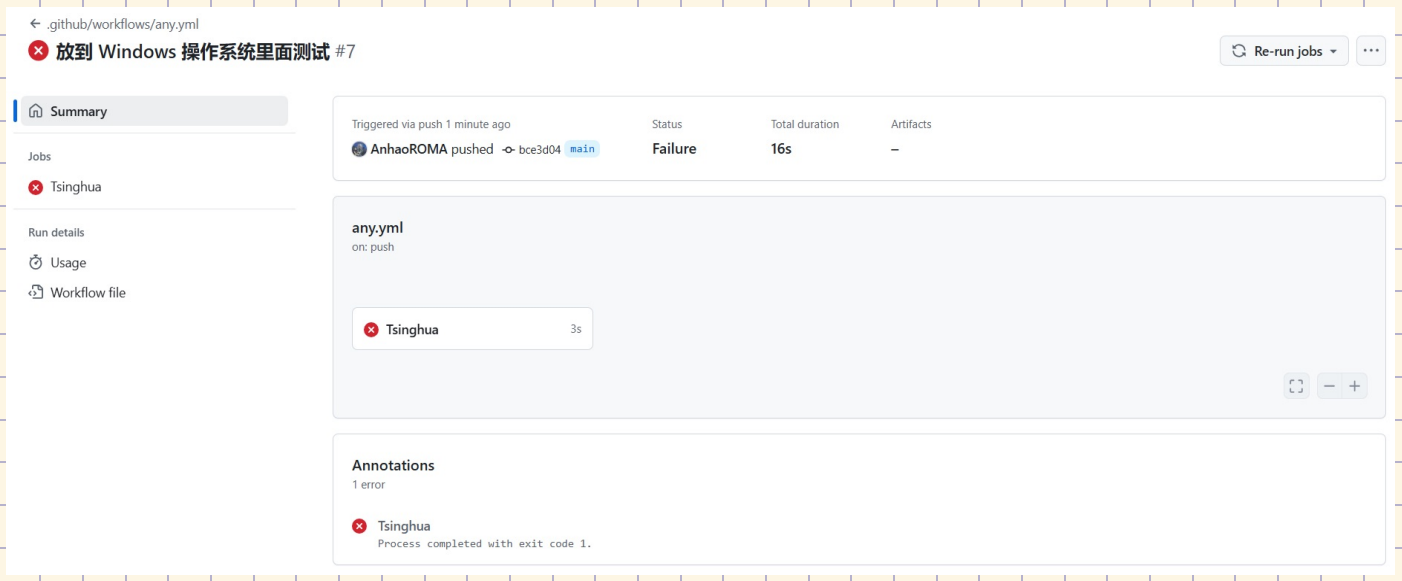
.github > workflows > any.yml

```yaml
on: push

jobs:
    Tsinghua:
        runs-on: windows-latest
        steps:
        - run: dir
        - run: ./main.exe
```

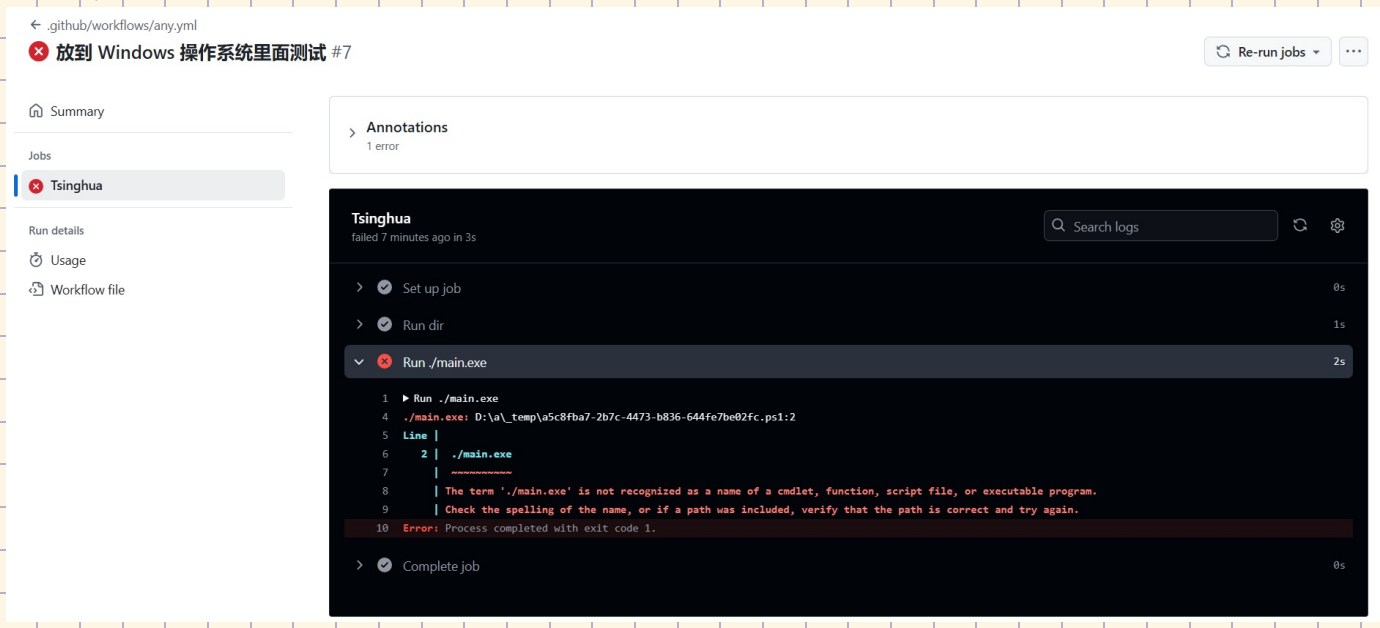
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
no changes added to commit (use "git add" and/or "git commit -a")
PS E:\GithubActions> git add .github/workflows/any.yml
PS E:\GithubActions> git commit -m "放到 Windows 操作系统里面测试"
[main bce3d04] 放到 Windows 操作系统里面测试
 1 file changed, 4 insertions(+), 9 deletions(-)
PS E:\GithubActions> git push origin main
```

18 看样子没什么问题，然而...



19 点开看看。



原因：当前路径下没有 main.exe 文件。

解决方法也很显然：把 GitHub 仓库中的文件拷贝到操作系统里面。

20 使用一个现成的"代码"。

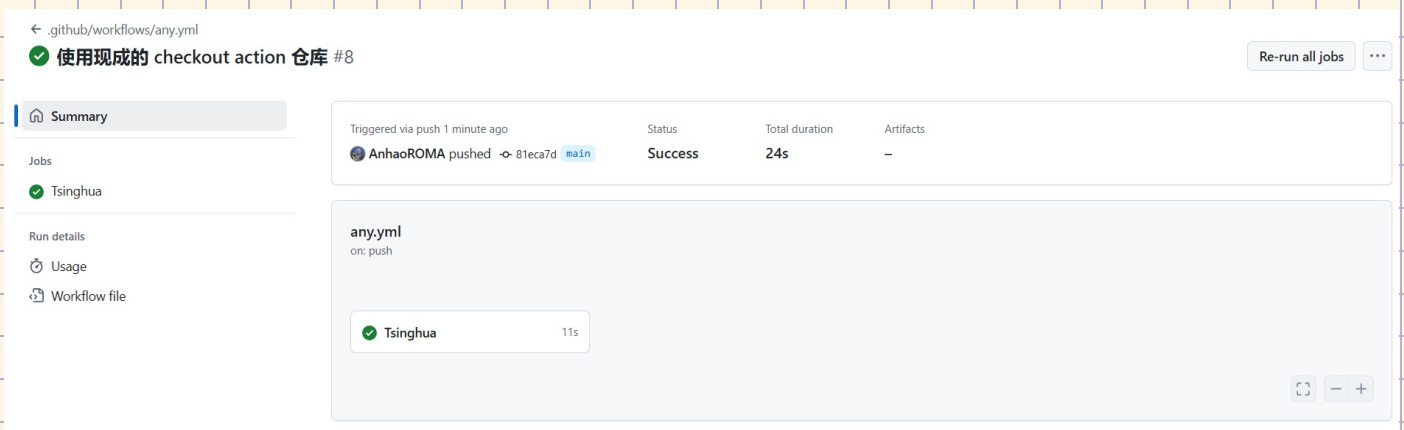
.github > workflows > any.yml

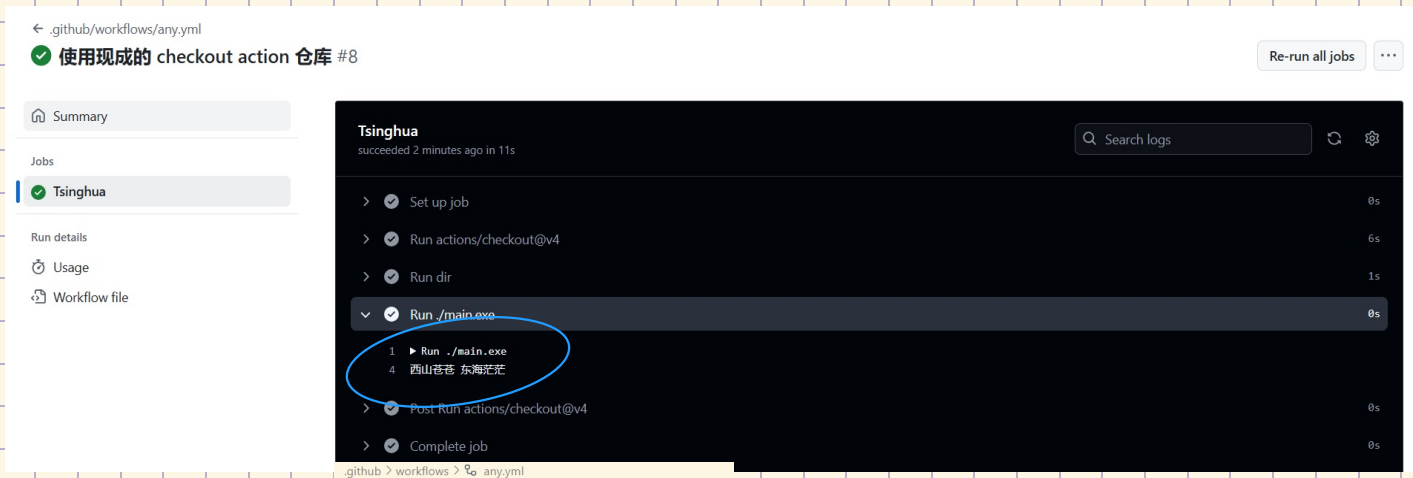
```yaml
on: push

jobs:
    Tsinghua:
        runs-on: windows-latest
        steps:
        - uses: actions/checkout@v4
        - run: dir
        - run: ./main.exe
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS E:\GithubActions> git commit -a -m "使用现成的 checkout action 仓库"
[main 81eca7d] 使用现成的 checkout action 仓库
 1 file changed, 1 insertion(+)
PS E:\GithubActions> git push origin main
```

21 成功了。





22 感觉有点乱...
不妨给每个步骤起上名字吧。

```yaml
on: push

jobs:
    Tsinghua:
        name: 清华大学

        runs-on: windows-latest

        steps:
        - name: 拷贝内容
          uses: actions/checkout@v4

        - name: 运行程序
          run: ./main.exe
```

job 的名字

step 的名字

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS E:\GithubActions> git commit -a -m "给每个步骤取名字"
[main cc559be] 给每个步骤取名字
 1 file changed, 9 insertions(+), 3 deletions(-)
PS E:\GithubActions> git push origin main
```

这样就好看多了。

23 使用 | 运算符可以将多条步骤组合在一起。

```
.github > workflows > any.yml
1   on: push
2
3   jobs:
4       Tsinghua:
5           name: 清华大学
6
7           runs-on: windows-latest
8
9           steps:
10          - name: 拷贝内容
11            uses: actions/checkout@v4
12
13          - name: 运行两次程序
14            run: |
15              ./main.exe
16              ./main.exe
```


```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS E:\GithubActions> git commit -a -m "多个步骤合到一起"
```

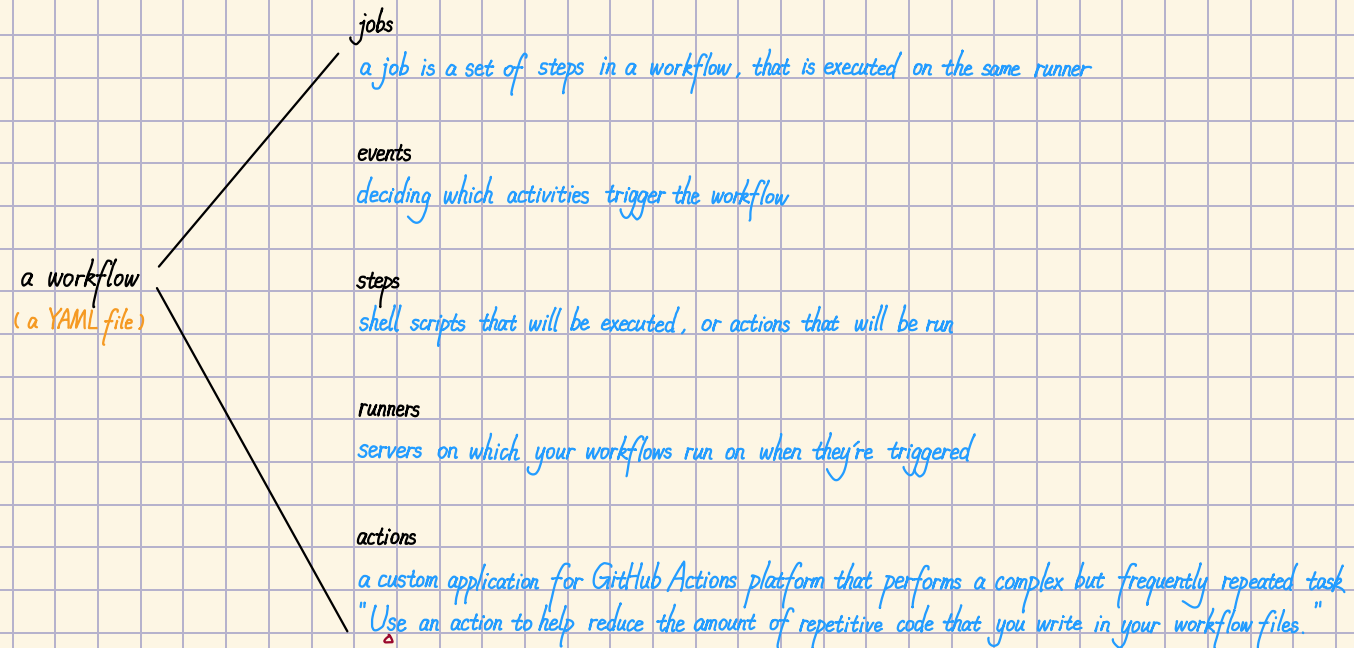
24 还可以给整个工作流起名字。

Getting Started with Workflows

GitHub Actions' workflow features allow users to configure various deployment and delivery pipelines to support different technologies.

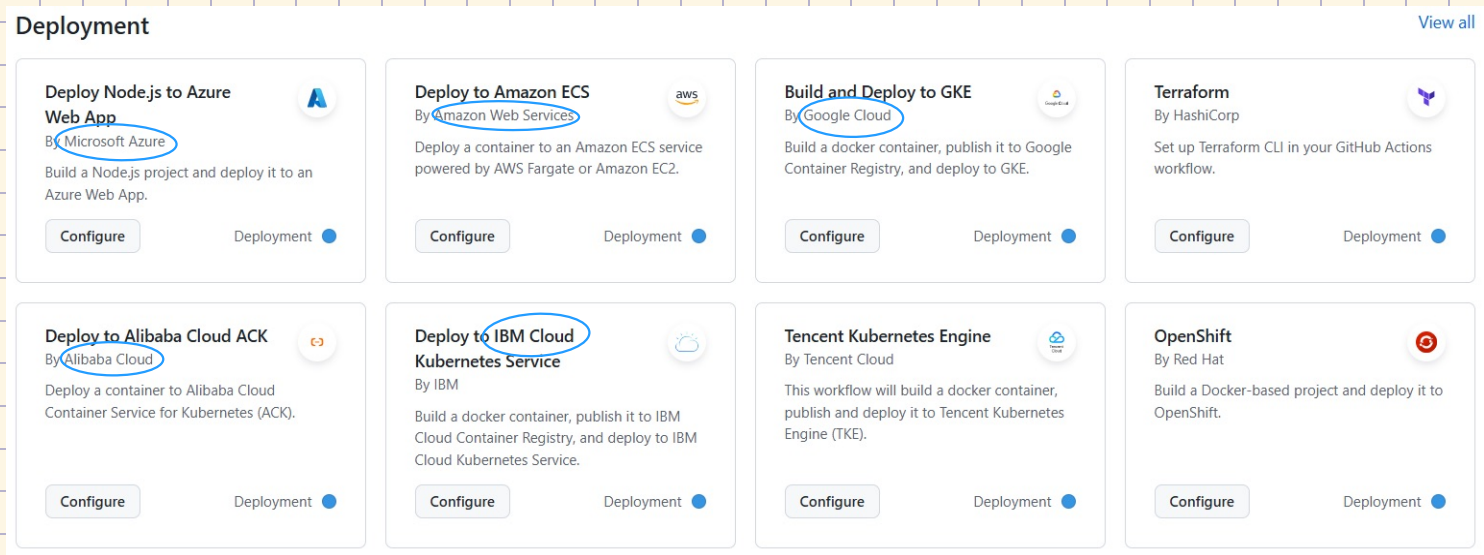
This chapter helps me understand GitHub Actions workflows by discussing the components that are important for configuring build and deployment pipelines.

GitHub Actions workflows are configured using preconfigured workflow templates or marketplace actions or self-defined actions.



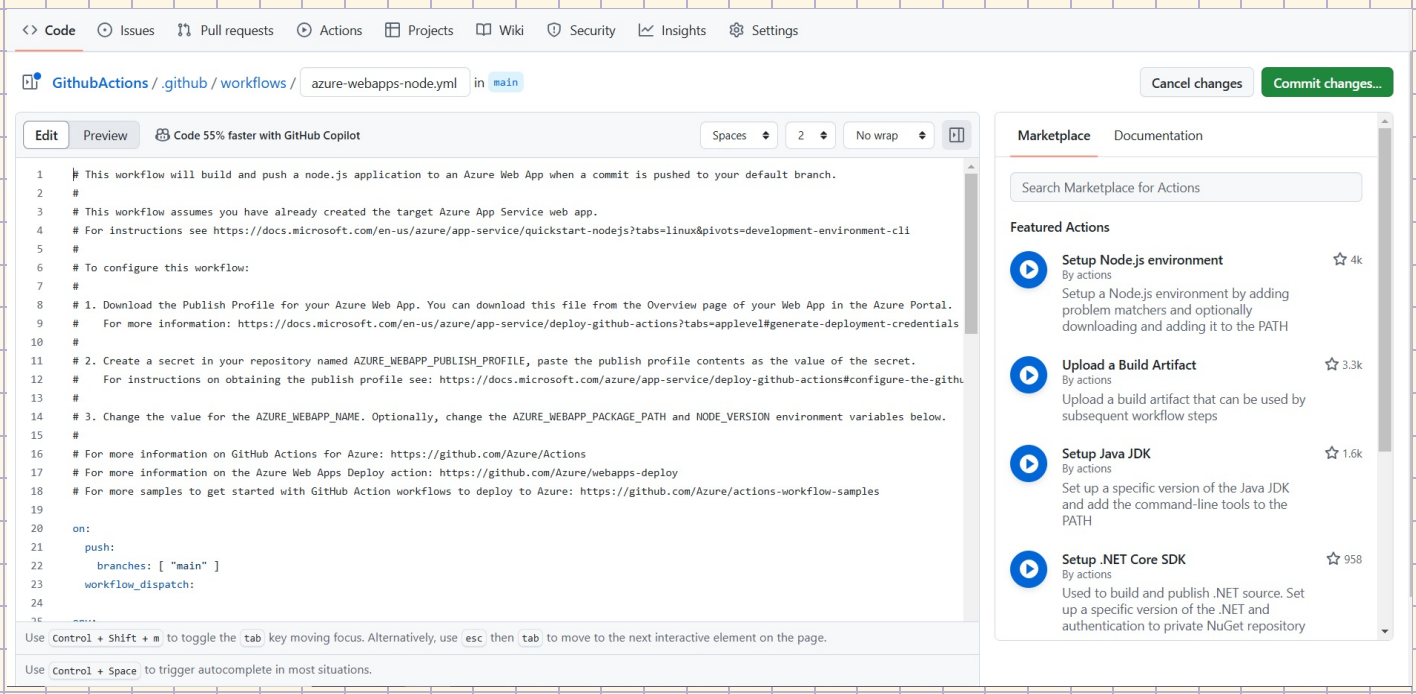
Using Preconfigured Workflow Templates

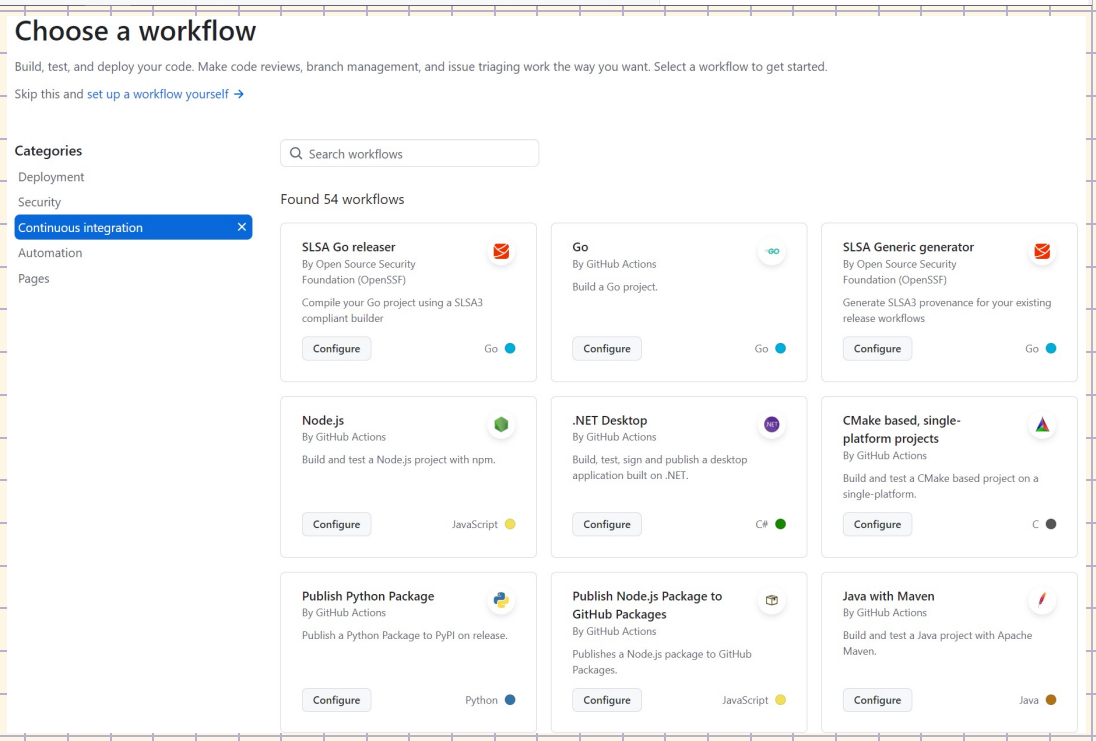
GitHub has multiple predefined workflow templates to create automated build and deployment processes.



There are deployment workflow templates for all the main cloud platforms.

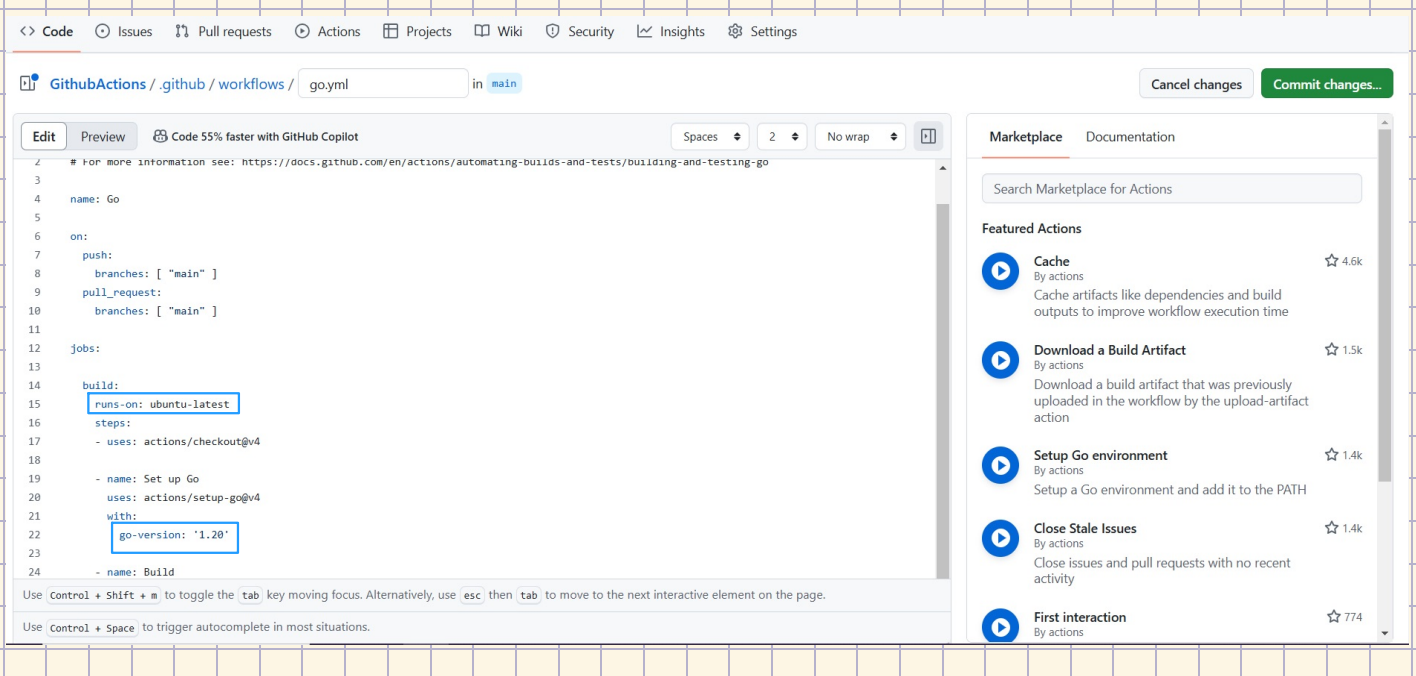
After clicking buttons, a template workflow YAML file is opened, which I can edit to fit my requirements.





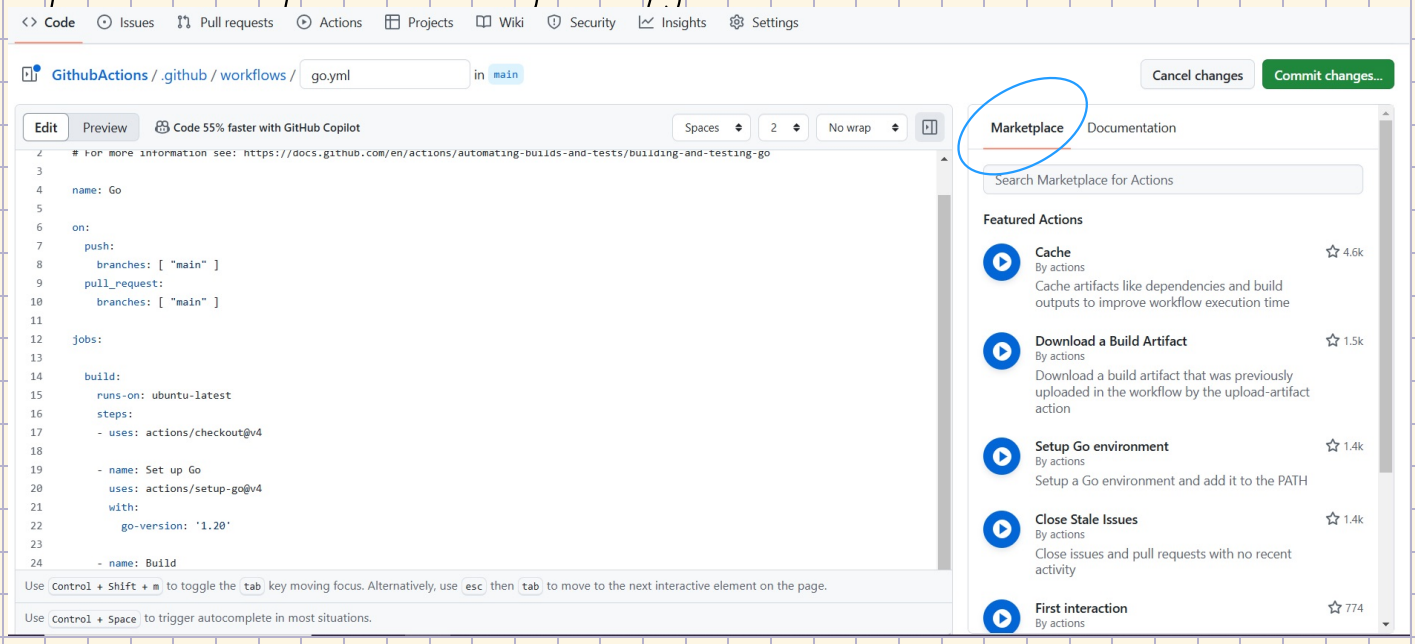
In addition to continuous deployment workflow templates, there are continuous integration workflow templates to build applications.

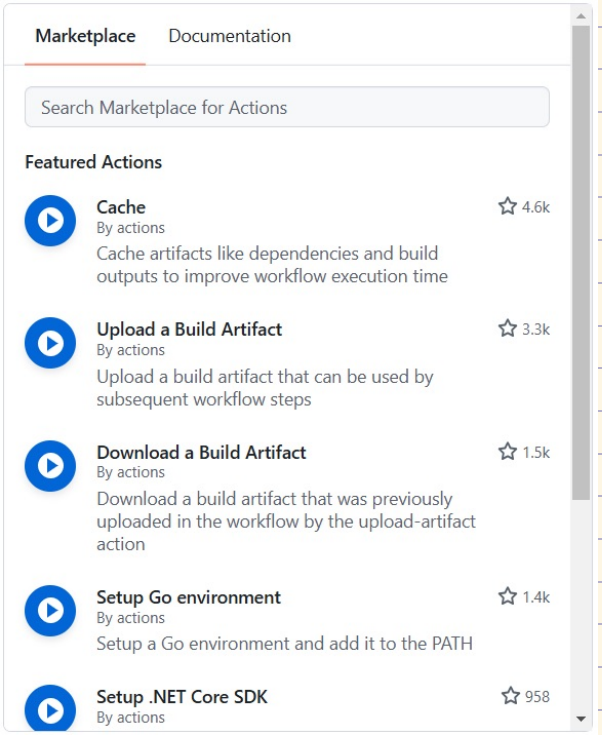
Like the deployment workflow templates, an integration workflow is also a YAML file consisting of basic build steps that I can edit according to my requirements.



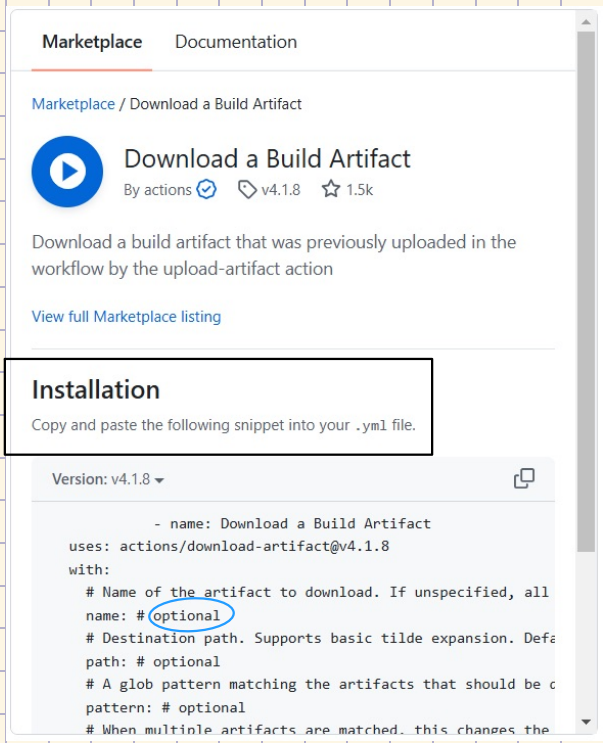
Using Marketplace Actions

One place to access Marketplace actions is the workflow editor page.





Then you can select the actions that need to be added to the workflow.

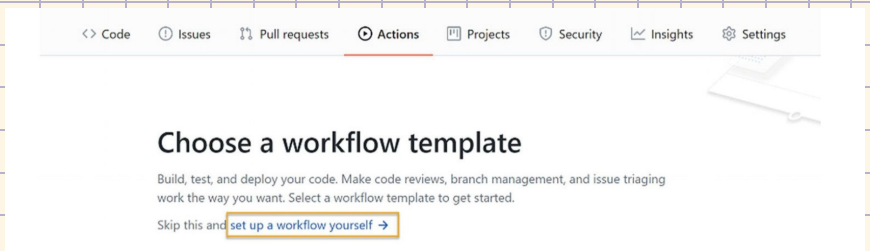


Then copy & paste the relevant scripts under the Installation section into my workflow.

Using Self-defined Actions

If you have a comprehensive understanding of all components of workflows, then you can write that yaml file from scratch!

Understanding the Structure of a Workflow



to start with an empty workflow, without using templates

Then fill the yaml file as below. Explanation is attached as comments.

Code | Blame | 32 lines (27 loc) · 1.02 KB | Code 55% faster with GitHub Copilot

```yaml
# This is a basic workflow to help me get started with Github Actions.
name: CI

# To control when the action will run:
#   to trigger the workflow on push or pull request, but only the master branch.
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

# A workflow is made up of one or more jobs that can run sequentially or in parallel.
jobs:
  # This workflow contains a single job called "build".
  build:
    # The runner (that this job will run on) runs on Ubuntu.
    runs-on: ubuntu-latest

    # Steps represent a sequence of tasks that will be executed, as part of the job.
    steps:
      # Checks-out your repository under $GITHUB_WORKSPACE, so your job(s) can access it.
      - uses: actions/checkout@v4

      # To run a single command using the runner's shell.
      - name: to run a single command
        run: echo ".- -. .... .- ---"

      # To run a set of commands using the runner's shell.
      - name: to run a set of commands
        run: |
          echo "amy"
          echo "fran"
```

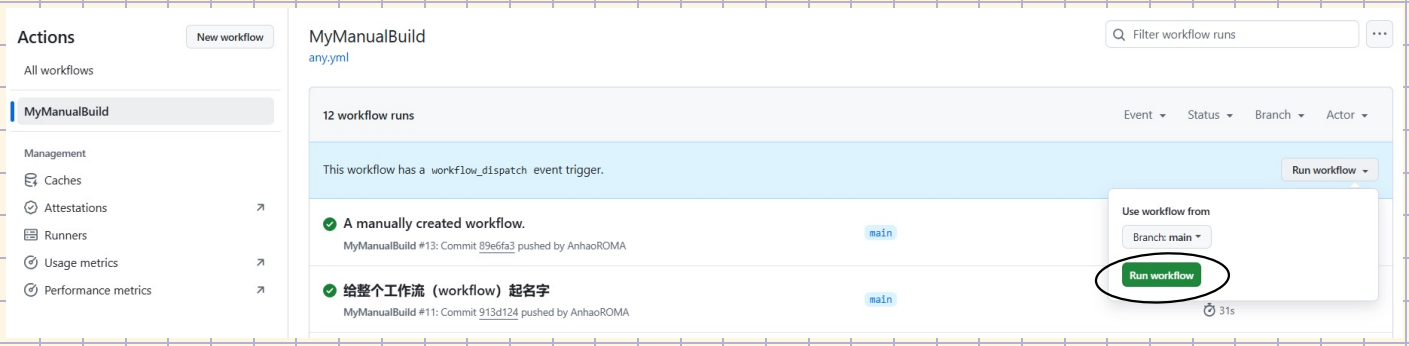
Workflow files should be saved under `.github/workflows/` in the repository root.

I can define the exact triggering condition for each workflow, for example, event triggers, schedule triggers and manual triggers.

For instance, to enable a manual trigger, a `workflow_dispatch` event should be activated in my workflow:

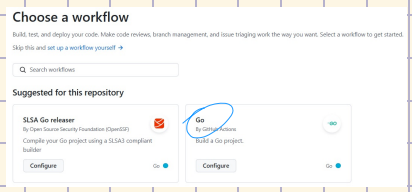
```
name: MyManualBuild
on: [ workflow_dispatch ]
```

This enables the Run workflow button.



Another important component is the runner.
A runner is a machine or container that executes the workflow. There are 2 types of runners: GitHub-hosted runners and self-hosted runners.

A workflow can have one or more jobs.
By default, jobs run in parallel. Hence, if I need to run jobs one after another, dependency `needs` to be specified.



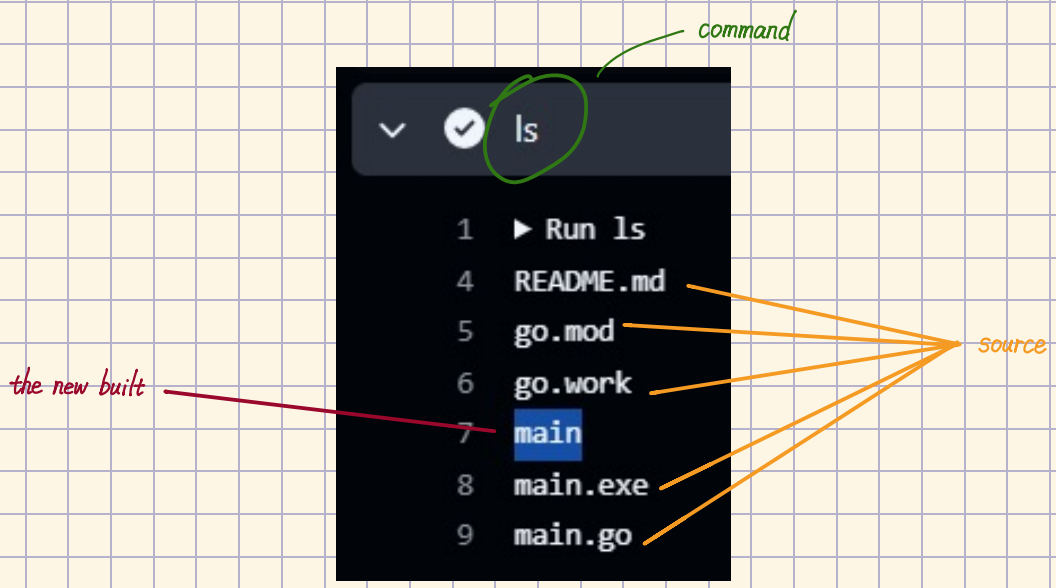
Experiment

Let me build a Go application in use of continuous integration.
I will use a Go workflow template, and modify the YAML file according to my requirements.

```yaml
# This workflow will build a golang project
# For more information see: https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-go

name: Go

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:

  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4

    - name: Set up Go
      uses: actions/setup-go@v4
      with:
        go-version: '1.20'

    - name: Build
      run: go build -v ./...

    - name: Test
      run: go test -v ./...

```

Source code should be downloaded to the runner as the first step before building the code. Therefore, the checkout action downloads the source.

Go version is defined after `with`.



More following behaviors are "publishing files" and "uploading published files as an artifact to the build pipeline", if you are dealing with .Net Core code.

Variables, Secrets, Tokens

In any platform or tool facilitating the implementation of CI/CD, it is essential to have a mechanism to configure variables in the pipelines. This chapter explores the options for GitHub Actions variables. In GitHub Actions, I can define custom variables in the scope of a workflow, job, or step.

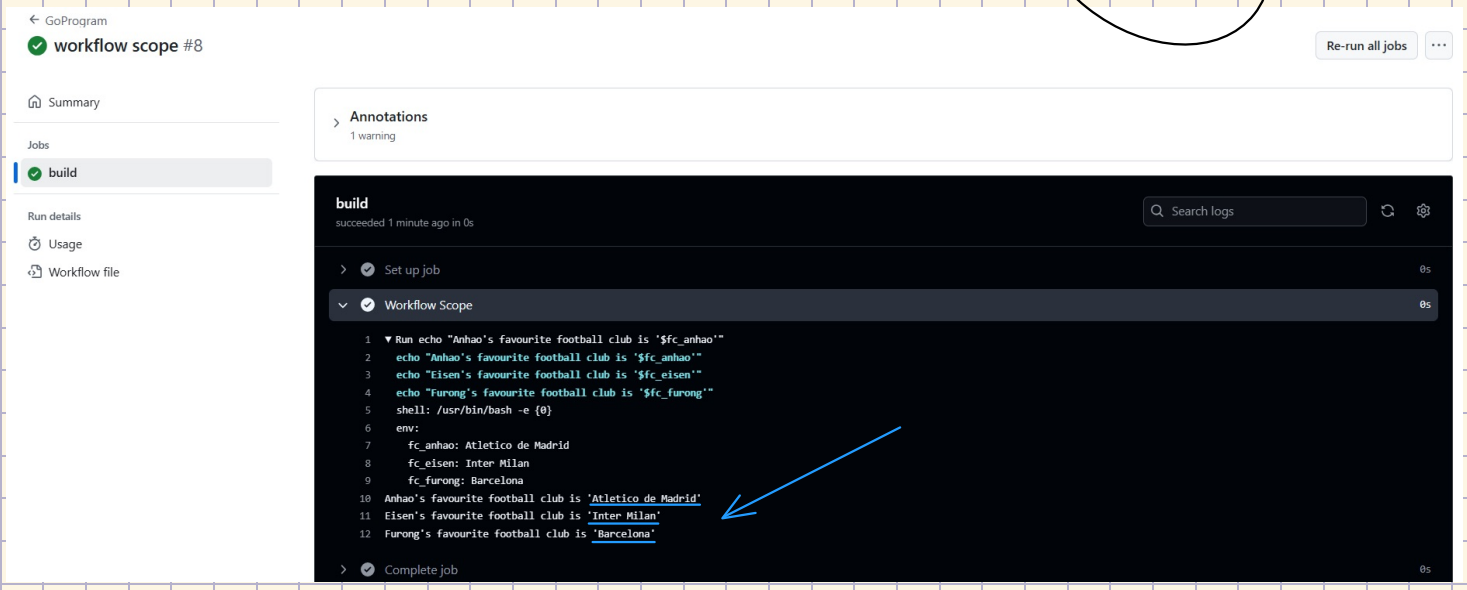
Variables in Workflow Scope

.github > workflows > go.yml

```yaml
name: GoProgram

on:
  push:
    branches: [ "main" ]

env:
  fc_anhao: "Atletico de Madrid"
  fc_eisen: "Inter Milan"
  fc_furong: "Barcelona"

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Workflow Scope
        run: |
          echo "Anhao's favourite football club is '$fc_anhao'"
          echo "Eisen's favourite football club is '$fc_eisen'"
          echo "Furong's favourite football club is '$fc_furong'"
```

To Define

To Use



Naming Considerations for Variables

1. GITHUB_* ✗
   The "GITHUB_" prefix is reserved for GitHub.

2. Case Sensitivity ✓
   GitHub Actions variables are case-sensitive.

3. *_PATH !

The variables defined to point to a filesystem location should contain the "_PATH" suffix.
(Though, the HOME and GITHUB_WORKSPACE default variables do not use this convention.)

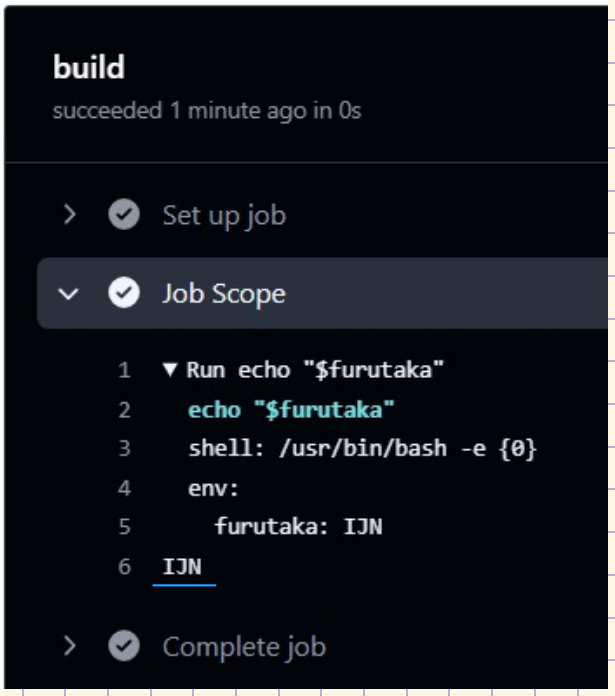
4. Special Characters !

Even though there are no syntactical errors caused by using special characters in the middle of a variable name, it is better to avoid them at all costs, other than "_".
[a-zA-Z_][a-zA-Z_0-9]

Variables in Job Scope

.github > workflows > go.yml

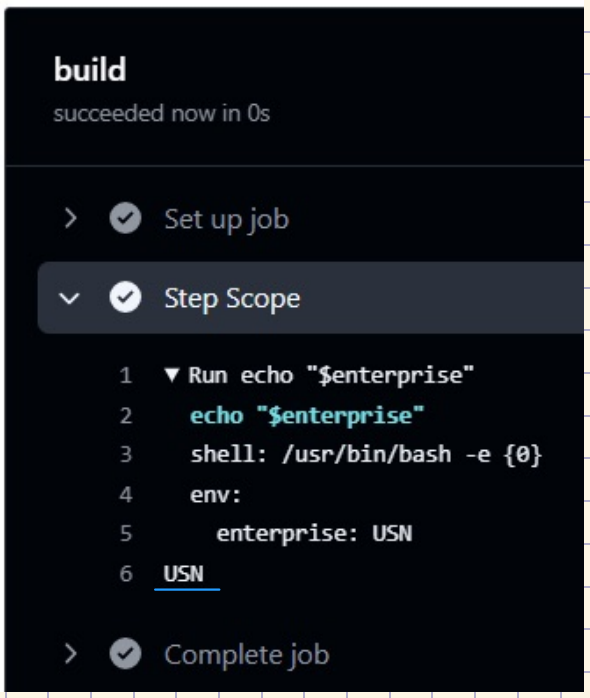
```yaml
name: GoProgram

on:
  push:
    branches: [ "main" ]

jobs:
  build:
    runs-on: ubuntu-latest

    env:
      furutaka: "IJN"

    steps:
      - name: Job Scope
        run: echo "$furutaka"
```



Variables in Step Scope

.github > workflows > go.yml

```yaml
name: GoProgram

on:
  push:
    branches: [ "main" ]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Step Scope
        env:
          enterprise: "USN"
        run: echo "$enterprise"
```



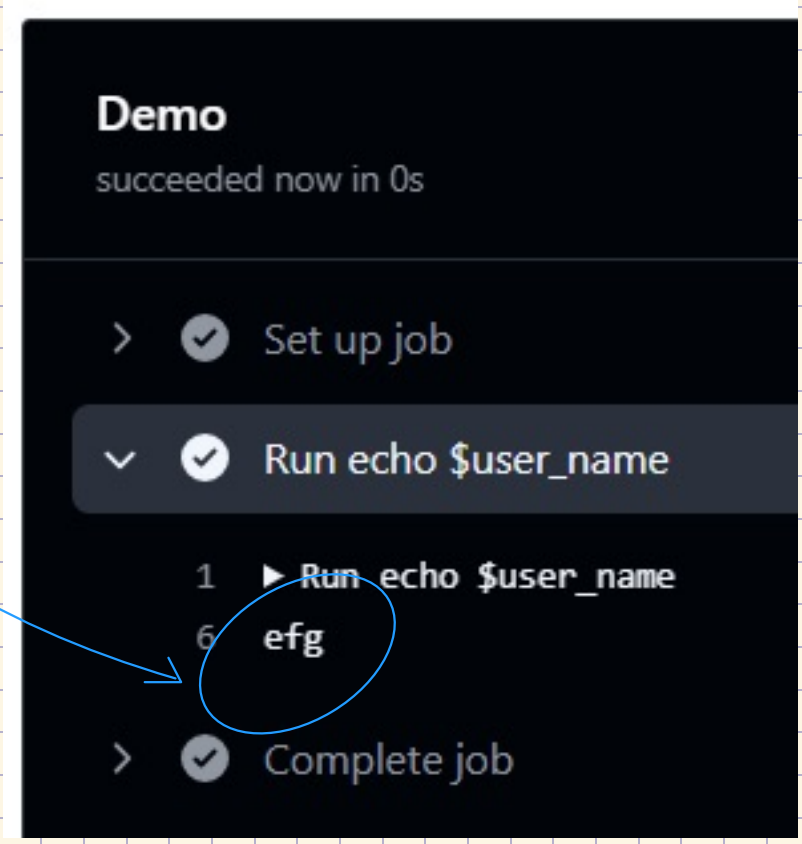
Override

.github > workflows > go.yml

```yaml
name: GoProgram

on: push

env:
  user_name: "abc"

jobs:
  Demo:
    runs-on: ubuntu-latest
    env:
      user_name: "efg"
    steps:
    - run: echo $user_name
```

user_name = "abc"

user_name = "efg"



Default Variables

A GitHub Actions workflow has a set of default variables.

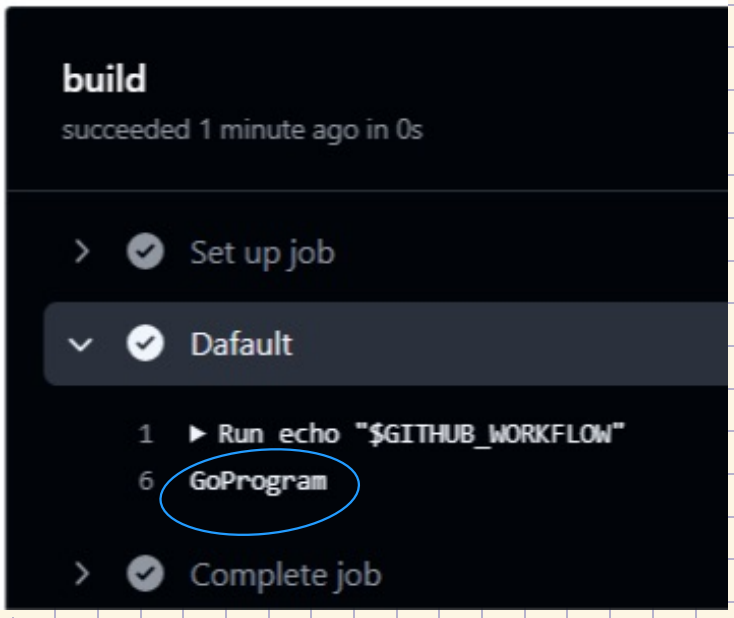
https://docs.github.com/en/actions/writing-workflows/choosing-what-your-workflow-does/store-information-in-variables#default-environment-variables

official document

using

.github > workflows > go.yml

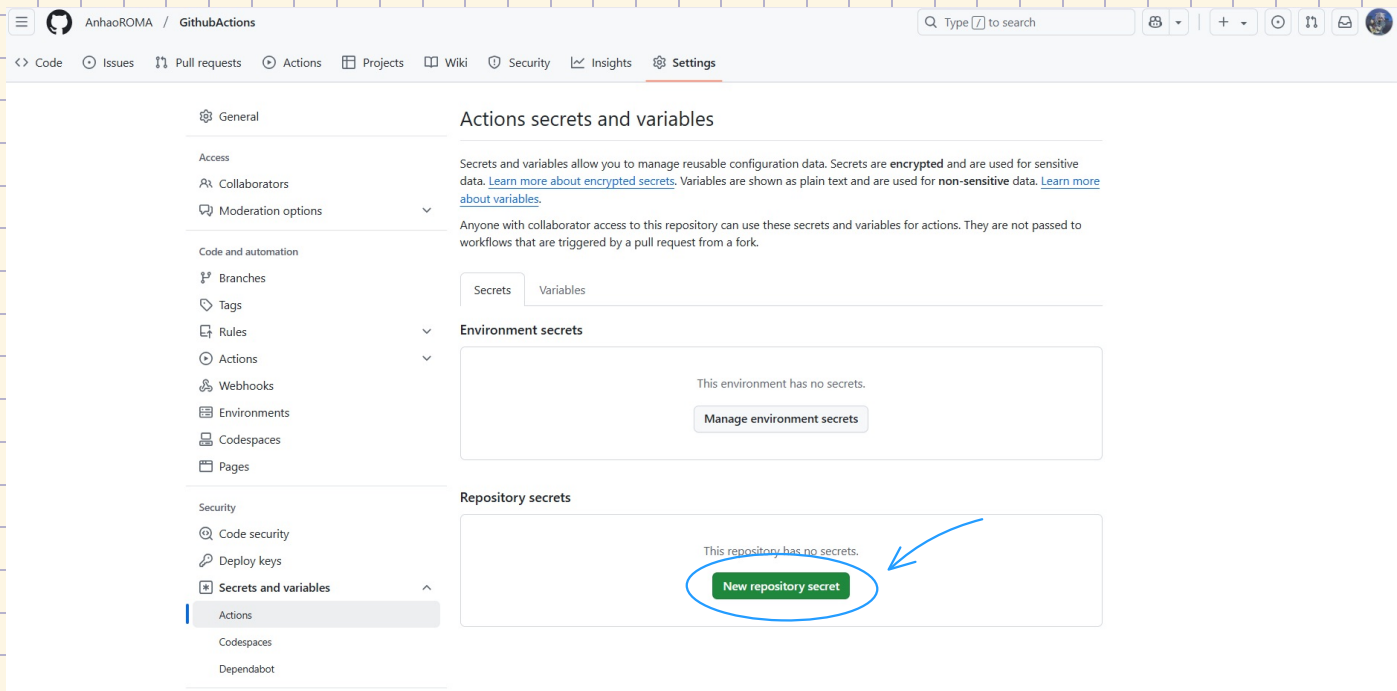
```yaml
name: GoProgram

on:
  push:
    branches: [ "main" ]

env:
  user_name: "paul"

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Dafault
        run: echo "$GITHUB_WORKFLOW"
```

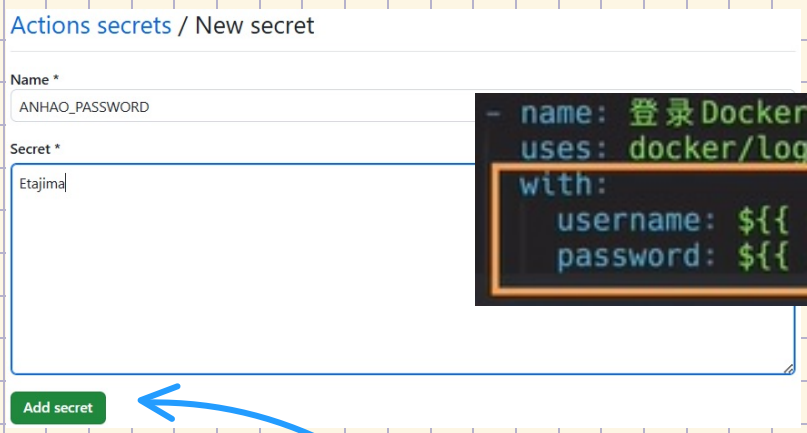


The ability to keep secret values is an essential feature in any CI/CD pipeline implementation tool, because some parameters/variables are sensitive information that cannot be stored openly. While programmatically allowing access to third parties may be necessary.

Next, the rest of this chapter explores the options for keeping secrets in GitHub Actions, as well as using them to provide programmatical access to GitHub.

Repo-Level Secrets



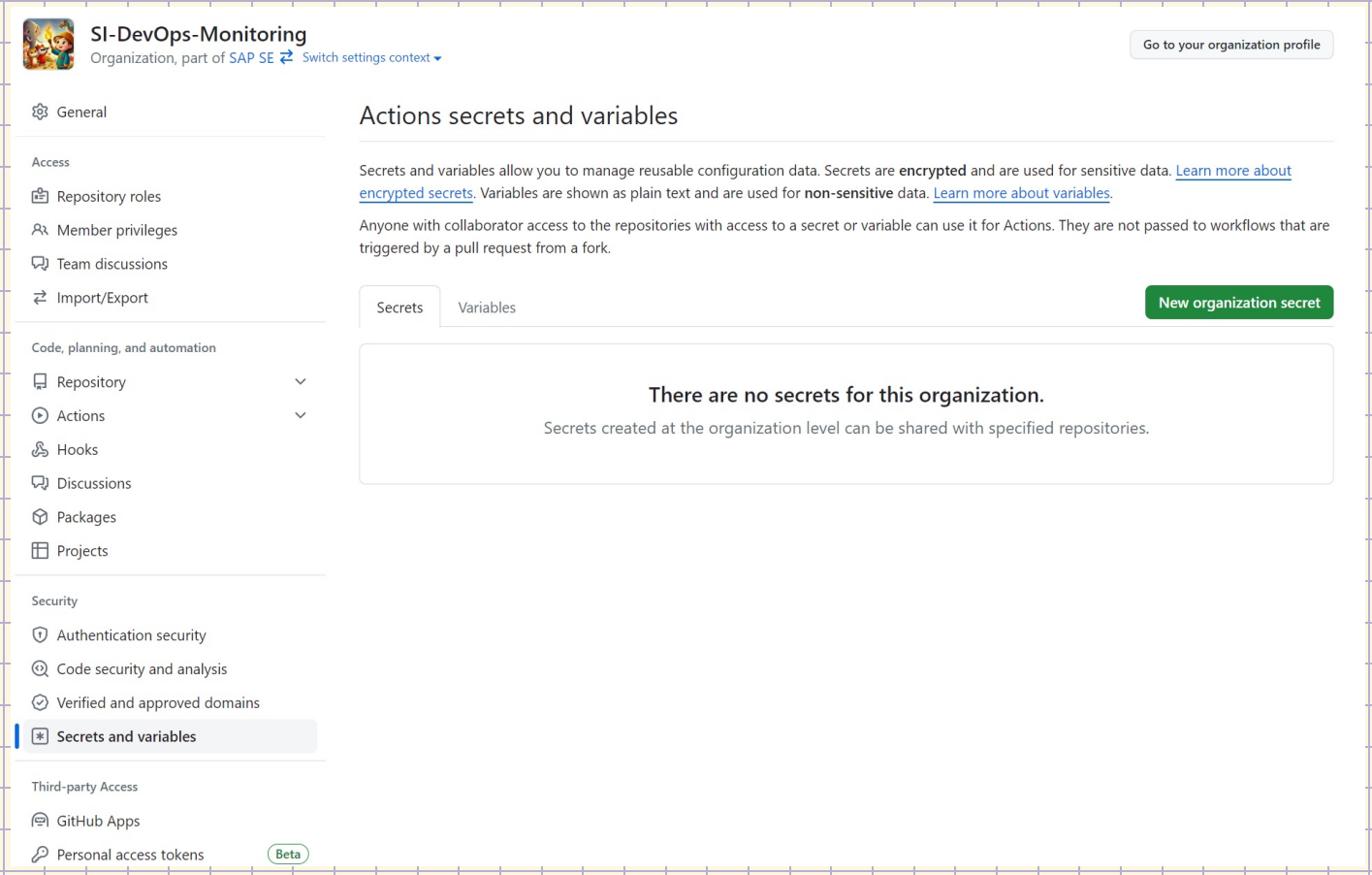



```yaml
- name: 登录DockerHub🔐
  uses: docker/login-action@v3
  with:
    username: ${{ secrets.DOCKER_HUB_USERNAME }}
    password: ${{ secrets.DOCKER_HUB_TOKEN }}
```

To Define

To Use

Organization-Level Secrets



Rules

1. To use a secret in the workflow, you need collaborator permission.
2. The secrets created in a GitHub repo are not available to the repo's folks.
3. Once a secret is created, the value cannot be seen again, but it can be utilized in the workflow.
   If required, you can either remove or update the secret to a new value.
4. Organization secrets are available to private repositories with the paid plans.
   Organization secrets are available in public repositories through workflows.
5. GitHub always redacts the secrets printed in workflow logs.
   However, you should take care not to accidently print the secrets to logs.

Naming Considerations for Secrets

1. Alphanumeric + Underscore
   [a-zA-Z_][a-zA-Z_0-9]
2. Case Sensitivity ✓
3. Unique
   Secret names must be unique at the repo or organization level.
   If you define a secret name at the organization level and use the same secret name in the organization's repos, precedence is given to the repo-level secrets.
4. GITHUB_* ✗

GitHub_Token

Case:

to create an issue if a workflow fails

```yaml
name: GoProgram

on: push

jobs:
  FailJobIssueDemo:
    permissions: write-all
    runs-on: ubuntu-latest
    steps:
    - name: Step is going to pass
      run: echo Passing step
    - name: Step is going to fail
      run: exit 1
    - name: Step To run on failure
      if: ${{ failure() }}
      run: |
        curl \
        --request POST \
        --url https://api.github.com/repos/${{ github.repository }}/issues \
        --header 'authorization: Bearer ${{ secrets.GITHUB_TOKEN }}' \
        --header 'content-type: application/json' \
        --data '{
          "title": "Issue created due to workflow failure: ${{ github.run_id }}",
          "body": "This issue was automatically created by the workflow **${{ github.workflow }}**."
        }'
```

Artifacts

Storing Content in Artifacts

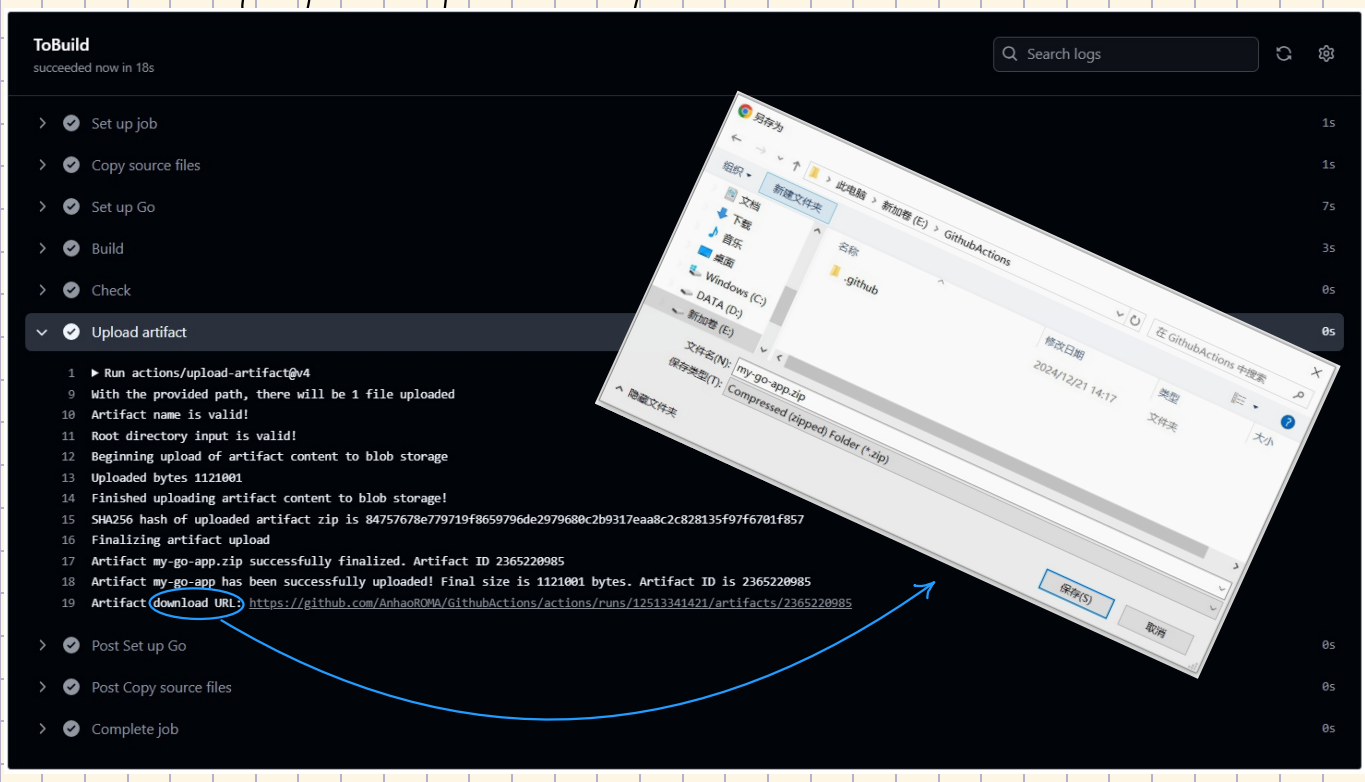
When I execute a build or test run in a GitHub Actions workflow, it will generate binaries or test results as the output of the workflow. These items may be stored for next jobs in the same workflow, or across workflows. GitHub storage is utilized to store artifacts.

.github > workflows > go.yml

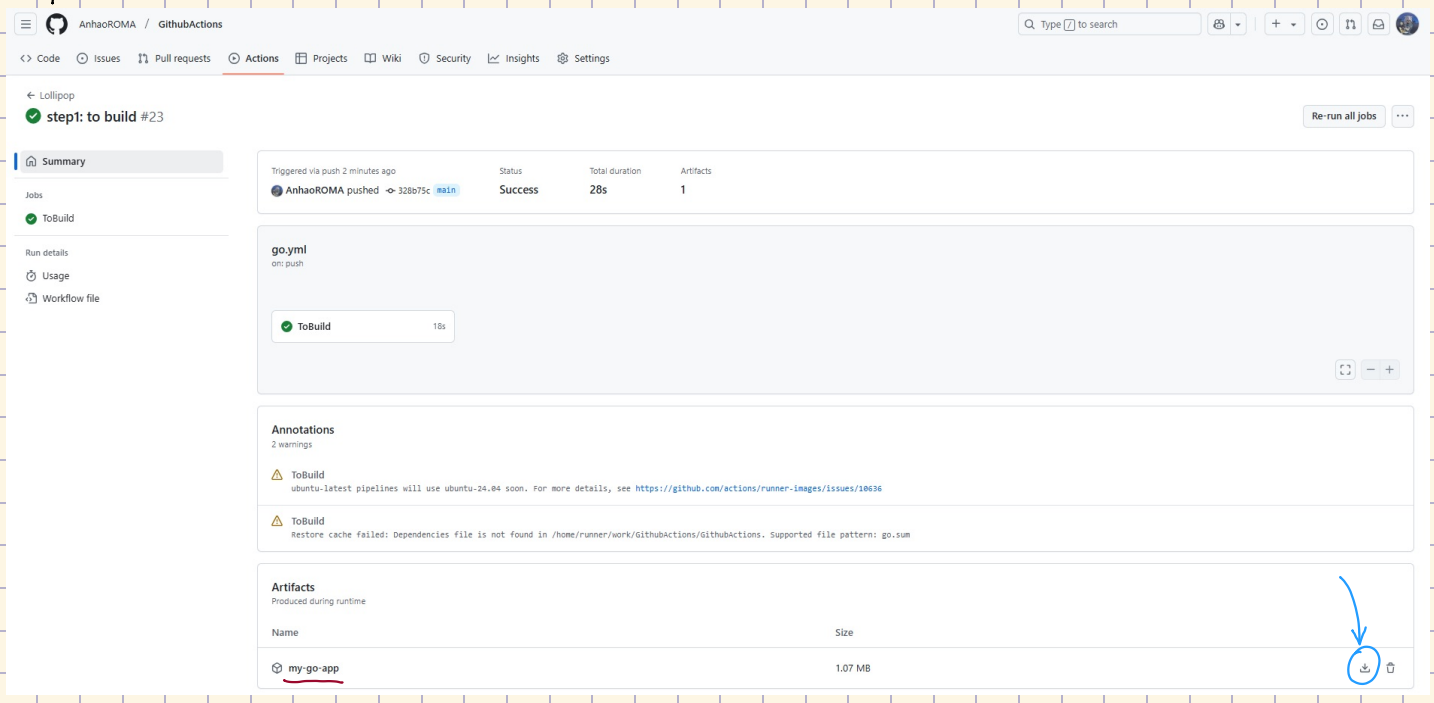
```yaml
name: Lollipop

on: push

jobs:
  ToBuild:
    runs-on: ubuntu-latest
    steps:

    - name: Copy source files
      uses: actions/checkout@v4

    - name: Set up Go
      uses: actions/setup-go@v4
      with:
        go-version: '1.20'

    - name: Build
      run: go build main.go

    - name: Check
      run: ls

    - name: Upload artifact
      uses: actions/upload-artifact@v4
      with:
        name: my-go-app
        path: ./main     # Replace with the actual name of your compiled binary
```

to upload an artifact, the default retention period is 90 days

You can download artifacts from the workflow once it is completed.



Or from browser :



Within the Same Workflow

Let's look at an example scenario where artifacts must be passed to another jobs in the same workflow : build steps are done on an Ubuntu runner and the built binaries will be deployed to MacOs.

Append the following snippet to the former :

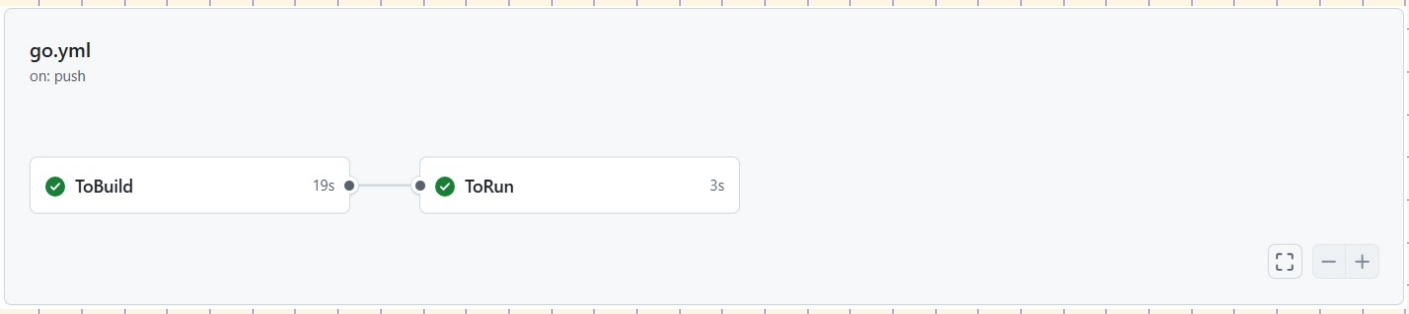
```
31  ToRun:
32    runs-on: ubuntu-latest
33    needs: ToBuild
34    steps:
35      - name: Download artifact
36        uses: actions/download-artifact@v4
37        with:
38          name: my-go-app
39      - name: Draw
40        run: |
41          chmod 777 main
42          ./main
```

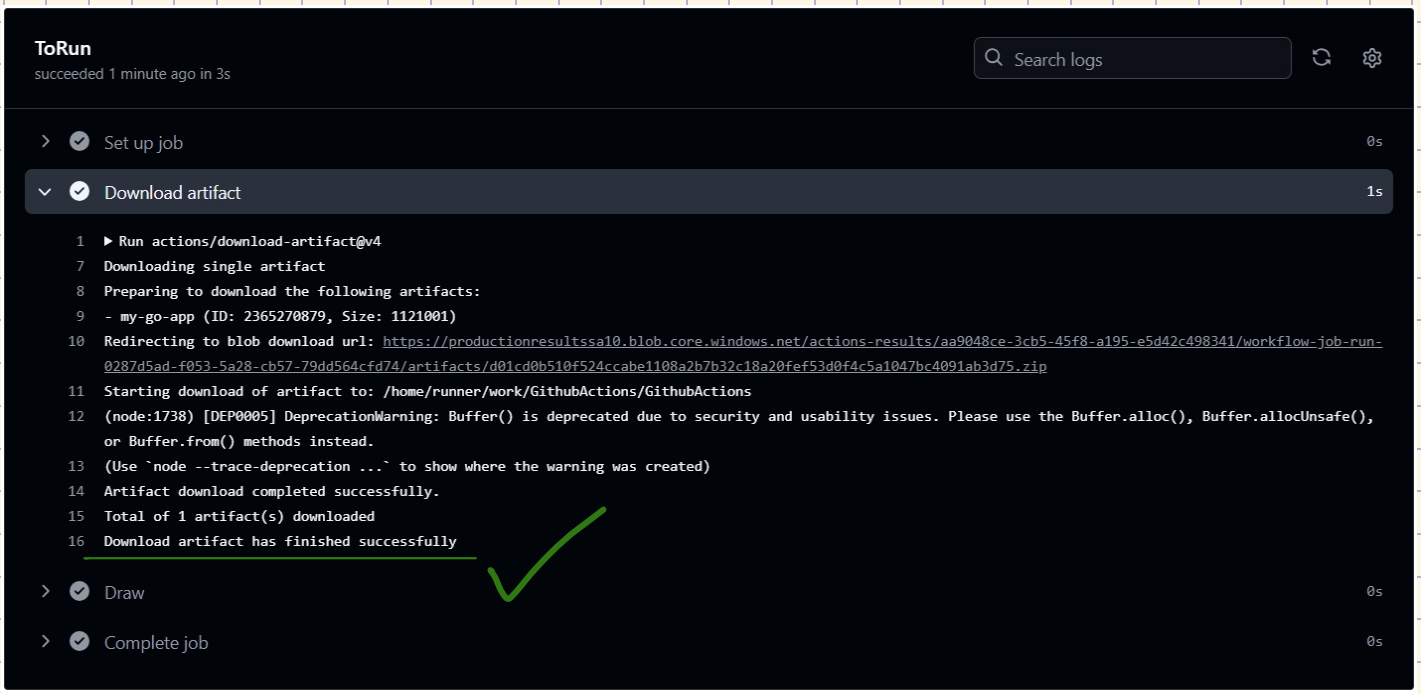
waits job "ToBuild" to finish first

corresponding

optionally a path to download artifacts

Succeeded !



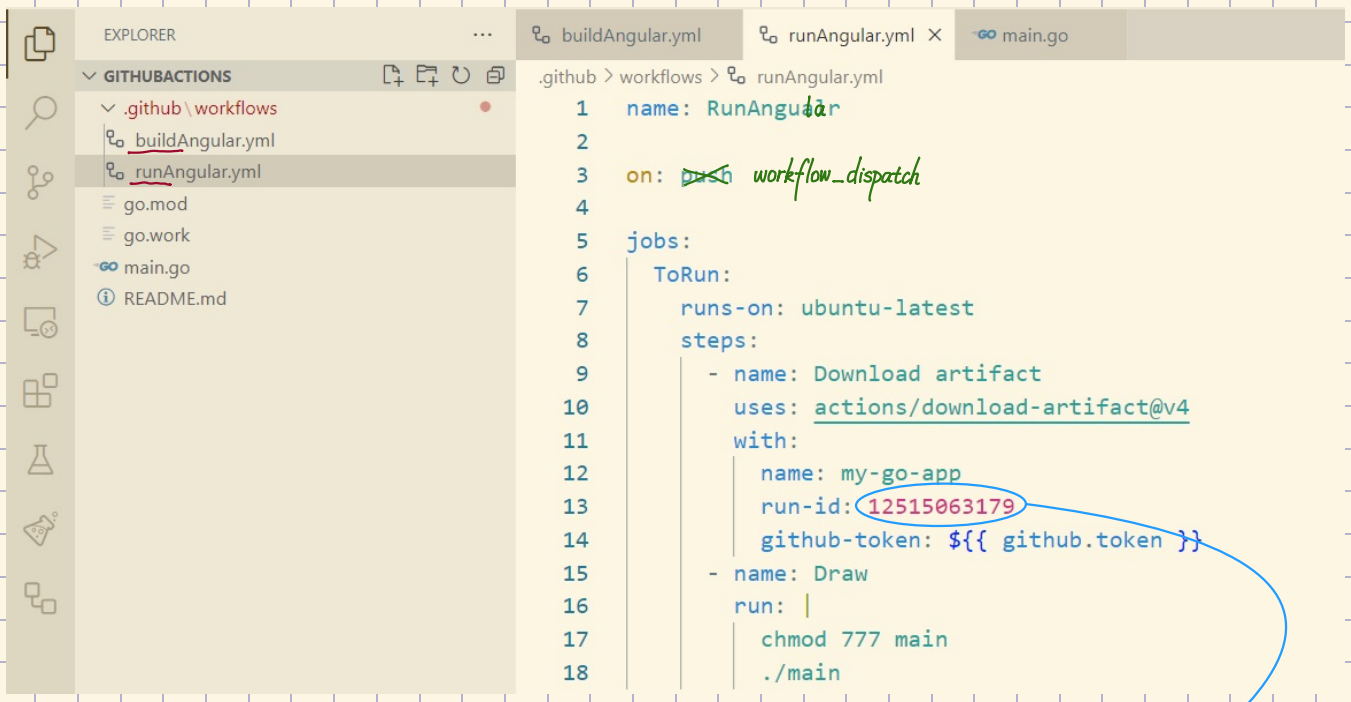


One thing to note is that **the version of** actions/upload-artifact **and** actions/download-artifact **must be the same** (@v3 or @v4).
https://stackoverflow.com/questions/78238187/unable-to-find-any-artifacts-for-the-associated-workflow-github-actions

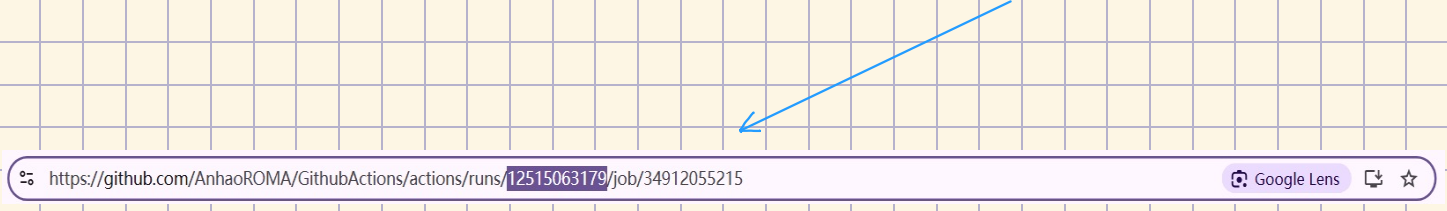
Across Different Workflows

I can implement this via REST API,
https://docs.github.com/en/rest/actions?apiVersion=2022-11-28#artifacts
but I will not use this method this time.

To make a new workflow (yaml file).



can retrieve from the build-angular



Supposed that:

- build-angular is on push
- run-angular is on workflow_dispatch

, so run-angular is manually started, using the artifact(s) generated by build-angular.

Reference:

https://github.com/marketplace/actions/download-workflow-artifact

https://stackoverflow.com/questions/60355925/share-artifacts-between-workflows-github-actions

"Persisted data in one job can be passed to another subsequent jobs, which may be running on different operating systems."

Caching Dependencies

"Reusable files can be cached, which considerably reduces the execution time of a GitHub Actions workflow."

When jobs are executed in GitHub-hosted runners, they always run in a fresh and clean virtual environment. But a clean environment demands downloading all required dependencies in each job run. As a solution, I can use GitHub's capabilities to cache dependencies.

may include files (utilized by package and dependency management tool such as npm, yarn, Gradle)

However, don't cache sensitive values in public repositories because forked repos can obtain cached information.

Prepare a Go program and a workflow.

```go
package main

import (
	"fmt"
	"os"
	"time"
)

func pathExists(path string) (bool, error) {
	_, err := os.Stat(path)
	if err == nil {
		return true, nil
	}
	if os.IsNotExist(err) {
		return false, nil
	}
	return false, err
}

func main() {
	ifExist, err := pathExists("/tmp/temp/1.txt")
	if err != nil {
		fmt.Println("Failed!")
	}
	if !ifExist {
		time.Sleep(15 * time.Second)
	}
}
```

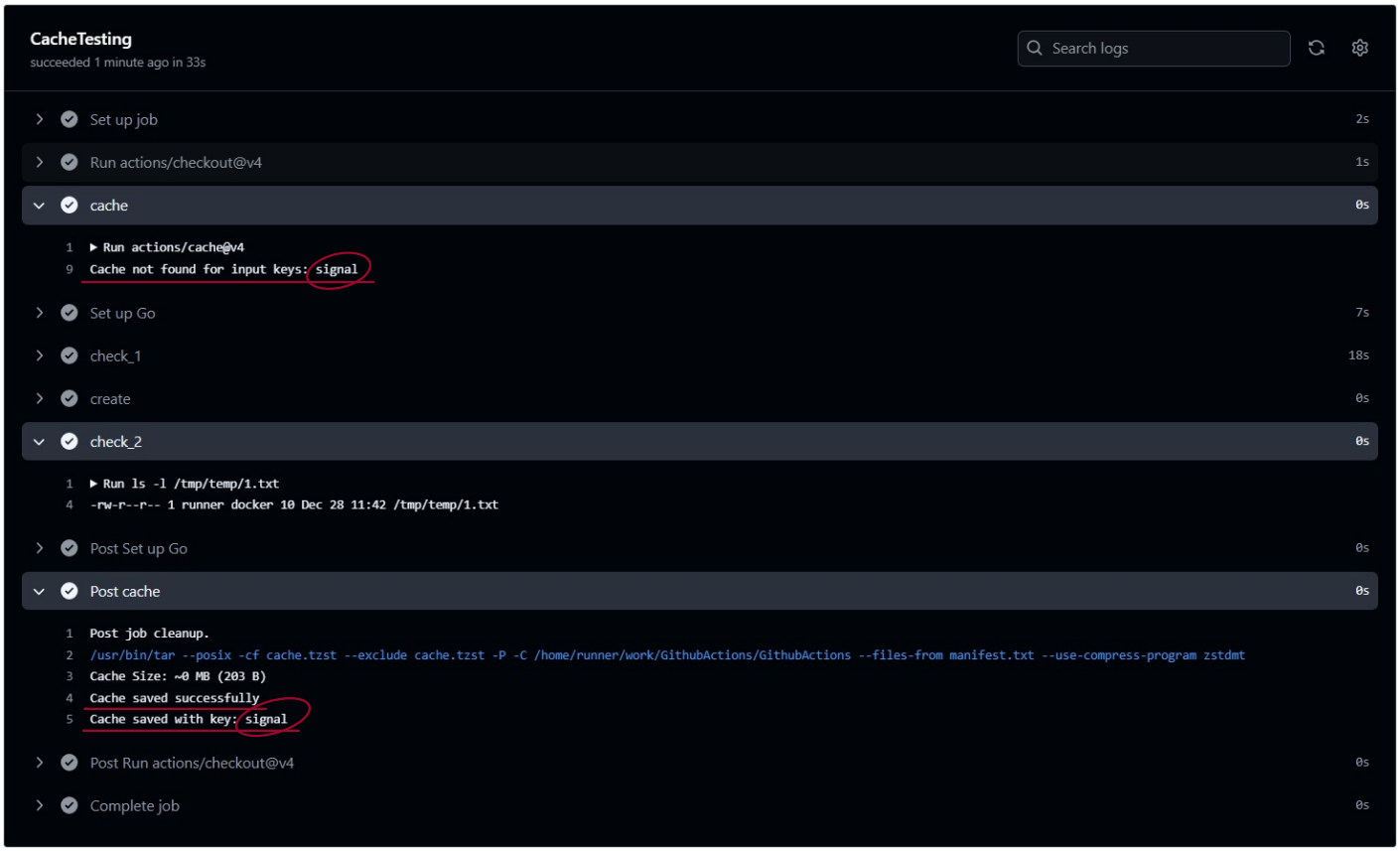
```yaml
name: Cache

on: push

jobs:
  CacheTesting:
    runs-on: ubuntu-latest
    steps:

    - uses: actions/checkout@v4

    - name: cache
      uses: actions/cache@v4
      with:
        path: /tmp/temp/1.txt
        key: signal

    - name: Set up Go
      uses: actions/setup-go@v4
      with:
        go-version: '1.20'

    - name: check_1
      run:
        go run main.go

    - name: create
      run: |
        mkdir -p /tmp/temp/
        echo "---------" > /tmp/temp/1.txt

    - name: check_2
      run: ls -l /tmp/temp/1.txt
```

The path(s) on the runner to cache or restore.

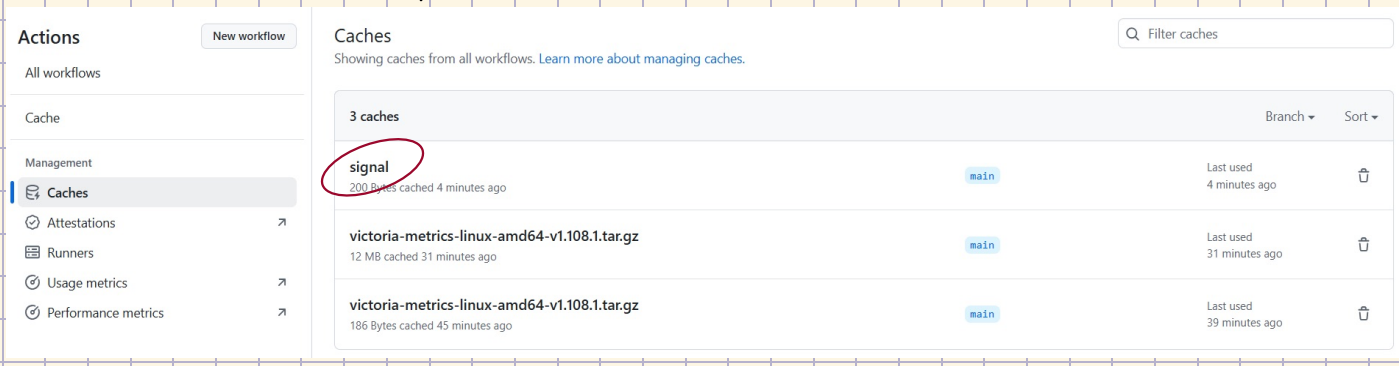
The key created when saving a cache and the key used to search for a cache.

如果该文件不存在，则停顿15秒。

When I executed the workflow for the first time, there is no cache available in the repo, so the file /tmp/temp/1.txt will be "cached".

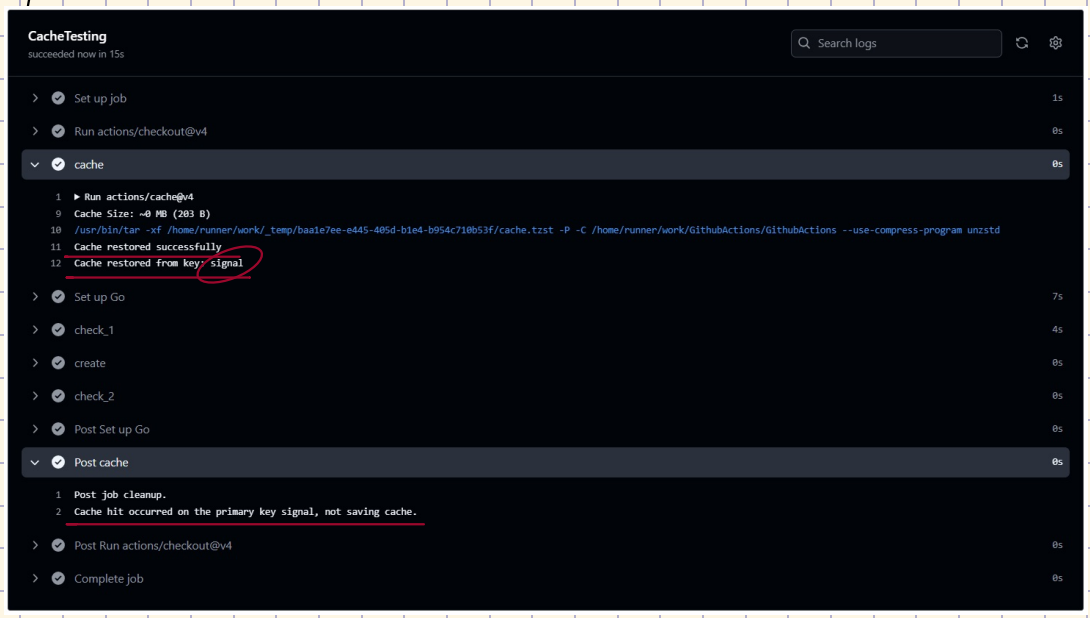


And I can find the cache in the repo.



Since there is no /tmp/temp/1.txt when running "check_1", the total run time is 33s (has a 15s pause).

In subsequent runs, the cached file(s) will be used.



Since the cache is available, the pipeline doesn't save the cache again.

Since there is /tmp/temp/1.txt when running "check-1", the total run time is 17s (no 15s pause).

GitHub's policy is to remove cached files not accessed for seven days.

I can create many caches, but there is a 5 GB size limit for all caches in the repository.

Using Self-Hosted Runners

GitHub provides hosted runners, or in other words, Windows, Linux, and MacOS machines, as workflow runners.

Hosted runner information can be found at: https://github.com/actions/runner-images/tree/main/images .
A VM runner-supported software list is specified in the readme.md file in each repo folder.

You may have specific software needs to build and deploy your applications. You may want to deploy to an on-promises environment utilizing GitHub Actions workflows. To cater to your needs, you can set up your (virtual) machines as runners for GitHub Actions.

Self-hosted runners can be added at different levels in GitHub.

- Repository level
- Organization level
- Enterprise level

However, be careful not to add a self-hosted runner to a public repository because folks in your public repos can execute malicious code by utilizing your runner.