



下载APP

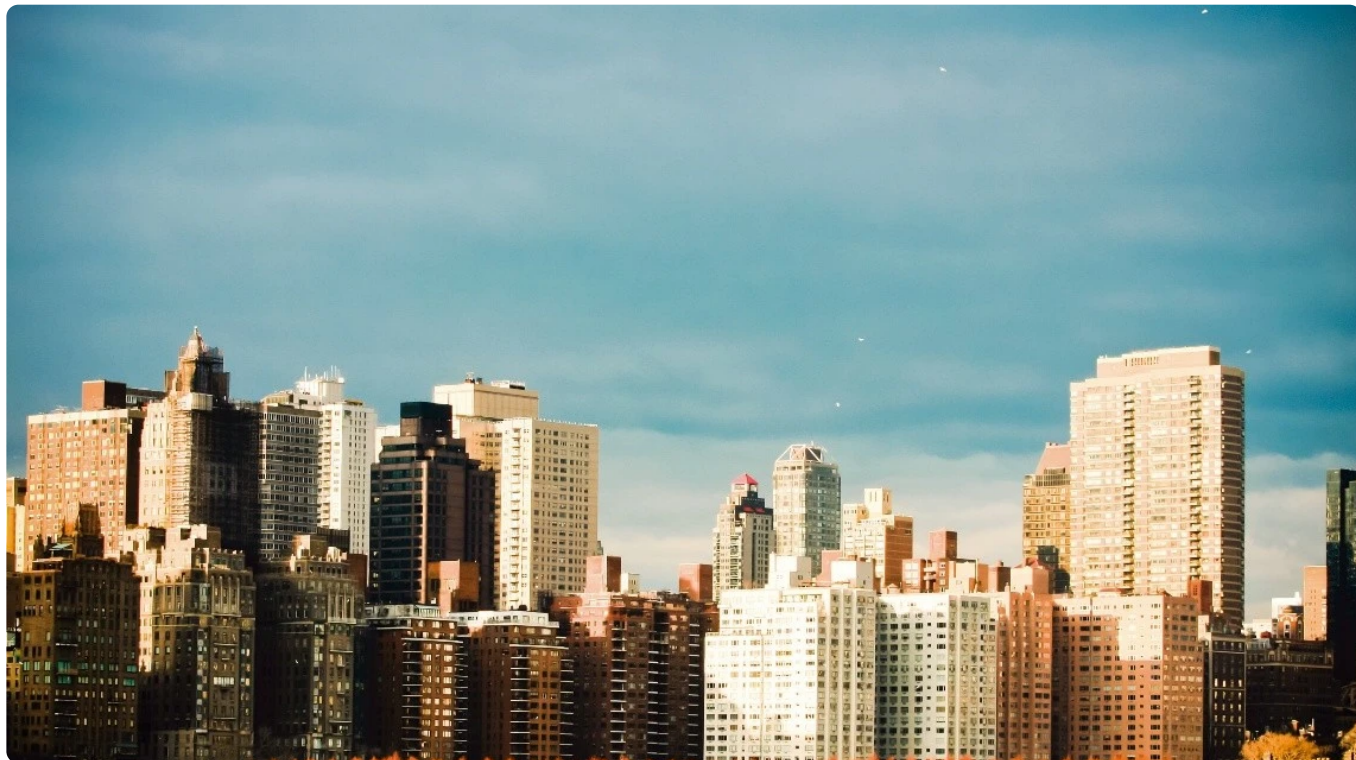


51 | 基于 GitHub Actions 的 CI 实战

2021-09-28 孔令飞

《Go 语言项目开发实战》

课程介绍 >



讲述：孔令飞

时长 12:49 大小 11.75M



你好，我是孔令飞。这是本专栏正文的最后一讲了，恭喜你坚持到了最后！

在 Go 项目开发中，我们要频繁地执行静态代码检查、测试、编译、构建等操作。如果每一步我们都手动执行，效率低不说，还容易出错。所以，我们通常借助 CI 系统来自动化执行这些操作。

当前业界有很多优秀的 CI 系统可供选择，例如 [@CircleCI](#)、[@TravisCI](#)、[@Jenkins](#)、[@CODING](#)、[@GitHub Actions](#) 等。这些系统在设计上大同小异，为了减少你的学习成本，我选择了相对来说容易实践的 GitHub Actions，来给你展示如何通过 CI 来让工作自动化。



这一讲，我会先介绍下 GitHub Actions 及其用法，再向你展示一个 CI 示例，最后给你演示下 IAM 是如何构建 CI 任务的。

GitHub Actions 的基本用法

GitHub Actions 是 GitHub 为托管在 github.com 站点的项目提供的持续集成服务，于 2018 年 10 月推出。

GitHub Actions 具有以下功能特性：

提供原子的 actions 配置和组合 actions 的 workflow 配置两种能力。

全局配置基于 [YAML 配置](#)，兼容主流 CI/CD 工具配置。

Actions/Workflows 基于 [事件触发](#)，包括 Event restrictions、Webhook events、Scheduled events、External events。

提供可供运行的托管容器服务，包括 Docker、VM，可运行 Linux、macOS、Windows 主流系统。

提供主流语言的支持，包括 Node.js、Python、Java、Ruby、PHP、Go、Rust、.NET。

提供实时日志流程，方便调试。

提供 [平台内置的 Actions](#) 与第三方提供的 Actions，开箱即用。

GitHub Actions 的基本概念

在构建持续集成任务时，我们会在任务中心完成各种操作，比如克隆代码、编译代码、运行单元测试、构建和发布镜像等。GitHub 把这些操作称为 Actions。

Actions 在很多项目中是可以共享的，GitHub 允许开发者将这些可共享的 Actions 上传到 [GitHub 的官方 Actions 市场](#)，开发者在 Actions 市场中可以搜索到他人提交的 Actions。另外，还有一个 [awesome actions](#) 的仓库，里面也有不少的 Action 可供开发者使用。如果你需要某个 Action，不必自己写复杂的脚本，直接引用他人写好的 Action 即可。整个持续集成过程，就变成了一个 Actions 的组合。

Action 其实是一个独立的脚本，可以将 Action 存放在 GitHub 代码仓库中，通过 `<userName>/<repoName>` 的语法引用 Action。例如，`actions/checkout@v2` 表示 `https://github.com/actions/checkout` 这个仓库，tag 是 v2。
`actions/checkout@v2` 也代表一个 Action，作用是安装 Go 编译环境。GitHub 官方的 Actions 都放在 github.com/actions 里面。

GitHub Actions 有一些自己的术语，下面我来介绍下。

workflow (工作流程)：一个 `.yaml` 文件对应一个 workflow，也就是一次持续集成。一个 GitHub 仓库可以包含多个 workflow，只要是在 `.github/workflow` 目录下的 `.yaml` 文件都会被 GitHub 执行。

job (任务)：一个 workflow 由一个或多个 job 构成，每个 job 代表一个持续集成任务。

step (步骤)：每个 job 由多个 step 构成，一步步完成。

action (动作)：每个 step 可以依次执行一个或多个命令 (action)。

on：一个 workflow 的触发条件，决定了当前的 workflow 在什么时候被执行。

workflow 文件介绍

GitHub Actions 配置文件存放在代码仓库的 `.github/workflows` 目录下，文件后缀为 `.yaml`，支持创建多个文件，文件名可以任意取，比如 `iam.yaml`。GitHub 只要发现 `.github/workflows` 目录里面有 `.yaml` 文件，就会自动运行该文件，如果运行过程中存在问题，会以邮件的形式通知到你。

workflow 文件的配置字段非常多，如果你想详细了解，可以查看 [官方文档](#)。这里，我来介绍一些基本的配置字段。

1. name

`name` 字段是 workflow 的名称。如果省略该字段，默认为当前 workflow 的文件名。

```
1 name: GitHub Actions Demo
```

[复制代码](#)

2. on

on字段指定触发 workflow 的条件，通常是某些事件。

```
1 on: push
```

[复制代码](#)

上面的配置意思是，push事件触发 workflow。on字段也可以是事件的数组，例如：

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

[复制代码](#)

上面的配置意思是，push事件或pull_request事件都可以触发 workflow。

想了解完整的事件列表，你可以查看 [🔗 官方文档](#)。除了代码库事件，GitHub Actions 也支持外部事件触发，或者定时运行。

3. on.<push|pull_request>.<tags|branches>

指定触发事件时，我们可以限定分支或标签。

```
1 on:
2   push:
3     branches:
4       - master
```


[复制代码](#)

上面的配置指定，只有master分支发生push事件时，才会触发 workflow。

4. jobs.<job_id>.name

workflow 文件的主体是jobs字段，表示要执行的一项或多项任务。

jobs字段里面，需要写出每一项任务的job_id，具体名称自定义。job_id里面的name字段是任务的说明。

 复制代码

```
1 jobs:
2   my_first_job:
3     name: My first job
4   my_second_job:
5     name: My second job
```

上面的代码中，jobs字段包含两项任务，job_id分别是my_first_job和my_second_job。

5. jobs.<job_id>.needs

needs字段指定当前任务的依赖关系，即运行顺序。

 复制代码

```
1 jobs:
2   job1:
3   job2:
4     needs: job1
5   job3:
6     needs: [job1, job2]
```

上面的代码中，job1必须先于job2完成，而job3等待job1和job2完成后才能运行。因此，这个 workflow 的运行顺序为：job1、job2、job3。

6. jobs.<job_id>.runs-on


runs-on字段指定运行所需要的虚拟机环境，它是必填字段。目前可用的虚拟机如下：

ubuntu-latest、ubuntu-18.04 或 ubuntu-16.04。

windows-latest、windows-2019 或 windows-2016。

macOS-latest 或 macOS-10.14。

下面的配置指定虚拟机环境为ubuntu-18.04。

 复制代码

```
1 runs-on: ubuntu-18.04
```

7. jobs.<job_id>.steps

steps字段指定每个 Job 的运行步骤，可以包含一个或多个步骤。每个步骤都可以指定下面三个字段。

jobs.<job_id>.steps.name：步骤名称。

jobs.<job_id>.steps.run：该步骤运行的命令或者 action。

jobs.<job_id>.steps.env：该步骤所需的环境变量。

下面是一个完整的 workflow 文件的范例：

 复制代码

```
1 name: Greeting from Mona
2 on: push
3
4 jobs:
5   my-job:
6     name: My Job
7     runs-on: ubuntu-latest
8     steps:
9       - name: Print a greeting
10         env:
11           MY_VAR: Hello! My name is
12           FIRST_NAME: Lingfei
13           LAST_NAME: Kong
14         run: |
15           echo $MY_VAR $FIRST_NAME $LAST_NAME.
```

上面的代码中，steps字段只包括一个步骤。该步骤先注入三个环境变量，然后执行一条 Bash 命令。

8. uses

`uses` 可以引用别人已经创建的 `actions`，就是上面说的 `actions` 市场中的 `actions`。引用格式为 `userName/repoName@version`，例如 `uses: actions/setup-go@v1`。

9. with

`with` 指定 `actions` 的输入参数。每个输入参数都是一个键 / 值对。输入参数被设置为环境变量，该变量的前缀为 `INPUT_`，并转换为大写。

这里举个例子：我们定义 `hello_world` 操作所定义的三个输入参数（`first_name`、`middle_name` 和 `last_name`），这些输入变量将被 `hello-world` 操作作为 `INPUT_FIRST_NAME`、`INPUT_MIDDLE_NAME` 和 `INPUT_LAST_NAME` 环境变量使用。

[复制代码](#)

```
1 jobs:
2   my_first_job:
3     steps:
4       - name: My first step
5         uses: actions/hello_world@master
6         with:
7           first_name: Lingfei
8           middle_name: Go
9           last_name: Kong
```

10. run

`run`指定执行的命令。可以有多个命令，例如：

[复制代码](#)

```
1 - name: Build
2   run: |
3     go mod tidy
4     go build -v -o helloci .
```

11. id

`id`是 `step` 的唯一标识。

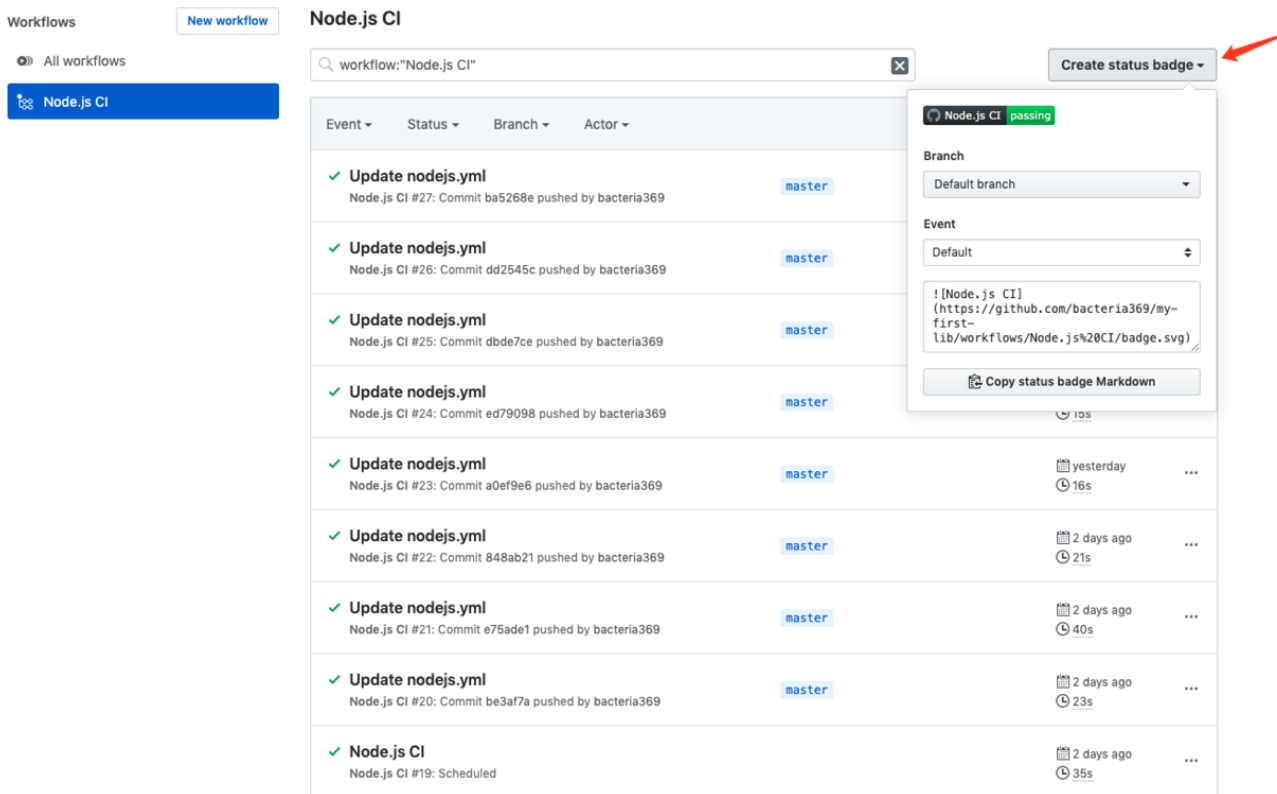
GitHub Actions 的进阶用法

上面，我介绍了 GitHub Actions 的一些基本知识，这里我再介绍下 GitHub Actions 的进阶用法。

为工作流加一个 Badge

在 action 的面板中，点击 Create status badge 就可以复制 Badge 的 Markdown 内容到 README.md 中。

之后，我们就可以直接在 README.md 中看到当前的构建结果：



Event	Status	Branch	Actor
Update nodejs.yml	✓	master	Node.js CI #27: Commit ba5268e pushed by bacteria369
Update nodejs.yml	✓	master	Node.js CI #26: Commit dd2545c pushed by bacteria369
Update nodejs.yml	✓	master	Node.js CI #25: Commit dbde7ce pushed by bacteria369
Update nodejs.yml	✓	master	Node.js CI #24: Commit ed79098 pushed by bacteria369
Update nodejs.yml	✓	master	Node.js CI #23: Commit a0ef9e6 pushed by bacteria369
Update nodejs.yml	✓	master	Node.js CI #22: Commit 848ab21 pushed by bacteria369
Update nodejs.yml	✓	master	Node.js CI #21: Commit e75ade1 pushed by bacteria369
Update nodejs.yml	✓	master	Node.js CI #20: Commit be3af7a pushed by bacteria369
Node.js CI	✓	master	Node.js CI #19: Scheduled

使用构建矩阵

如果我们想在多个系统或者多个语言版本上测试构建，就需要设置构建矩阵。例如，我们想在多个操作系统、多个 Go 版本下跑测试，可以使用如下 workflow 配置：

```
1 name: Go Test
2
3 on: [push, pull_request]
4
5 jobs:
```

[复制代码](#)


```
6   hello-ci-build:
7     name: Test with go ${ matrix.go_version } on ${ matrix.os }
8     runs-on: ${ matrix.os }
9
10    strategy:
11      matrix:
12        go_version: [1.15, 1.16]
13        os: [ubuntu-latest, macOS-latest]
14
15    steps:
16
17      - name: Set up Go ${ matrix.go_version }
18        uses: actions/setup-go@v2
19        with:
20          go-version: ${ matrix.go_version }
21          id: go
22
```

上面的 workflow 配置，通过strategy.matrix配置了该工作流程运行的环境矩阵（格式为go_version.os）：ubuntu-latest.1.15、ubuntu-latest.1.16、macOS-latest.1.15、macOS-latest.1.16。也就是说，会在 4 台不同配置的服务器上执行该 workflow。

使用 Secrets

在构建过程中，我们可能需要用到ssh或者token等敏感数据，而我們不希望这些数据直接暴露在仓库中，此时就可以使用secrets。

我们在对应项目中选择Settings-> Secrets，就可以创建secret，如下图所示：

Options

Manage access

Security & analysis

Branches

Webhooks

Notifications

Integrations

Deploy keys

Actions

Environments

Secrets

Actions

Dependabot

Pages

Actions secrets

New repository secret

Secrets are environment variables that are **encrypted**. Anyone with **collaborator** access to this repository can use these secrets for Actions.

Secrets are not passed to workflows that are triggered by a pull request from a fork. [Learn more](#).

Environment secrets

DOCKER_PASSWORD
helloci Updated 8 minutes ago [Manage environment](#)

DOCKER_USERNAME
helloci Updated 8 minutes ago [Manage environment](#)

DOCKER_USERNAME、DOCKER_PASSWORD属于helloci环境

Repository secrets

There are no secrets for this repository.

Encrypted secrets allow you to store sensitive information, such as access tokens, in your repository.

Secrets can also be created at the organization level and authorized for use in this repository.

配置文件中的使用方法如下：

复制代码

```
1 name: Go Test
2 on: [push, pull_request]
3 jobs:
4   helloci-build:
5     name: Test with go
6     runs-on: [ubuntu-latest]
7     environment:
8       name: helloci
9     steps:
10      - name: use secrets
11        env:
12          super_secret: ${ secrets.YourSecrets }
```

secret name 不区分大小写，所以如果新建 secret 的名字是 name，使用时用 secrets.name 或者 secrets.Name 都是可以的。而且，就算此时直接使用 echo 打印 secret，控制台也只会打印出*来保护 secret。

这里要注意，你的 secret 是属于某一个环境变量的，所以要指明环境的名字：environment.name。上面的 workflow 配置中的 secrets.YourSecrets 属于 helloci 环境。

使用 Artifact 保存构建产物

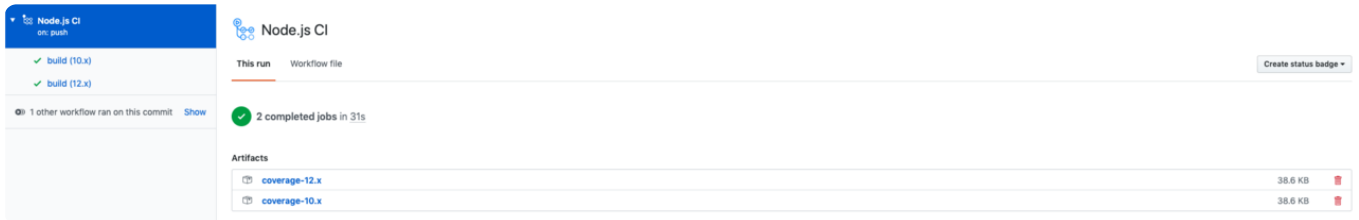
在构建过程中，我们可能需要输出一些构建产物，比如日志文件、测试结果等。这些产物可以使用 Github Actions Artifact 来存储。你可以使用 [action/upload-artifact](#) 和 [download-artifact](#) 进行构建参数的相关操作。

这里我以输出 Jest 测试报告为例来演示下如何保存 Artifact 产物。Jest 测试后的测试产物是 coverage：

[复制代码](#)

```
1 steps:
2   - run: npm ci
3   - run: npm test
4
5   - name: Collect Test Coverage File
6     uses: actions/upload-artifact@v1.0.0
7     with:
8       name: coverage-output
9       path: coverage
```

执行成功后，我们就能在对应 action 面板看到生成的 Artifact：

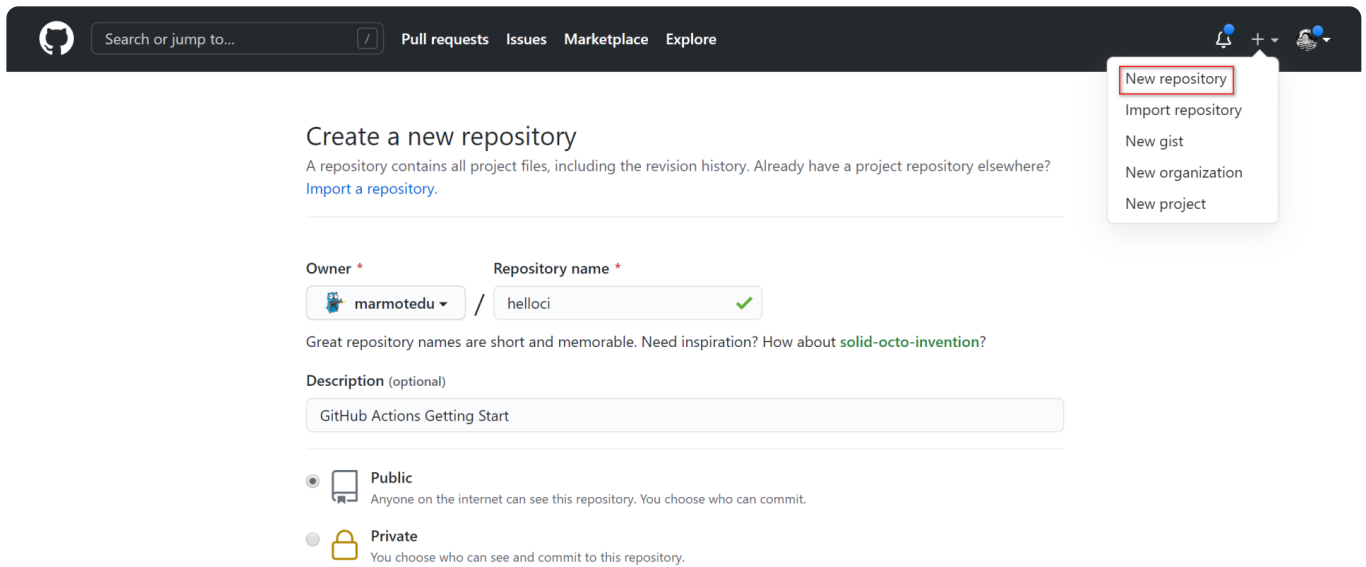


GitHub Actions 实战

上面，我介绍了 GitHub Actions 的用法，接下来我们就来实战下，看下使用 GitHub Actions 的 6 个具体步骤。

第一步，创建一个测试仓库。

登陆 [GitHub 官网](#)，点击 **New repository** 创建，如下图所示：



这里，我们创建了一个叫helloci的测试项目。

第二步，将新的仓库 clone 下来，并添加一些文件：

```
1 $ git clone https://github.com/marmotedu/helloci
```

复制代码

你可以克隆 [marmotedu/helloci](https://github.com/marmotedu/helloci)，并将里面的文件拷贝到你创建的项目仓库中。

第三步，创建 GitHub Actions workflow 配置目录：

```
1 $ mkdir -p .github/workflows
```

复制代码

第四步，创建 GitHub Actions workflow 配置。

在.github/workflows目录下新建helloci.yml文件，内容如下：

```
1 name: Go Test
2
3 on: [push, pull_request]
4
5 jobs:
6
```

复制代码

```
7   hello-ci-build:
8     name: Test with go ${ matrix.go_version } on ${ matrix.os }
9     runs-on: ${ matrix.os }
10    environment:
11      name: hello-ci
12
13    strategy:
14      matrix:
15        go_version: [1.16]
16        os: [ubuntu-latest]
17
18    steps:
19
20      - name: Set up Go ${ matrix.go_version }
21        uses: actions/setup-go@v2
22        with:
23          go-version: ${ matrix.go_version }
24          id: go
25
26      - name: Check out code into the Go module directory
27        uses: actions/checkout@v2
28
29      - name: Tidy
30        run: |
31          go mod tidy
32
33      - name: Build
34        run: |
35          go build -v -o hello-ci .
36
37      - name: Collect main.go file
38        uses: actions/upload-artifact@v1.0.0
39        with:
40          name: main-output
41          path: main.go
42
43      - name: Publish to Registry
44        uses: elgohr/Publish-Docker-GitHub-Action@master
45        with:
46          name: ccr.ccs.tencentyun.com/marmotedu/hello-ci:beta # docker image
47          username: ${ secrets.DOCKER_USERNAME } # 用户名
48          password: ${ secrets.DOCKER_PASSWORD } # 密码
49          registry: ccr.ccs.tencentyun.com # 腾讯云Registry
50          dockerfile: Dockerfile # 指定 Dockerfile 的位置
51          tag_names: true # 是否将 release 的 tag 作为 docker image 的 tag
```

上面的 workflow 文件定义了当 GitHub 仓库有 push、pull_request 事件发生时，会触发 GitHub Actions 工作流程，流程中定义了一个任务 (Job) hello-ci-build，Job 中

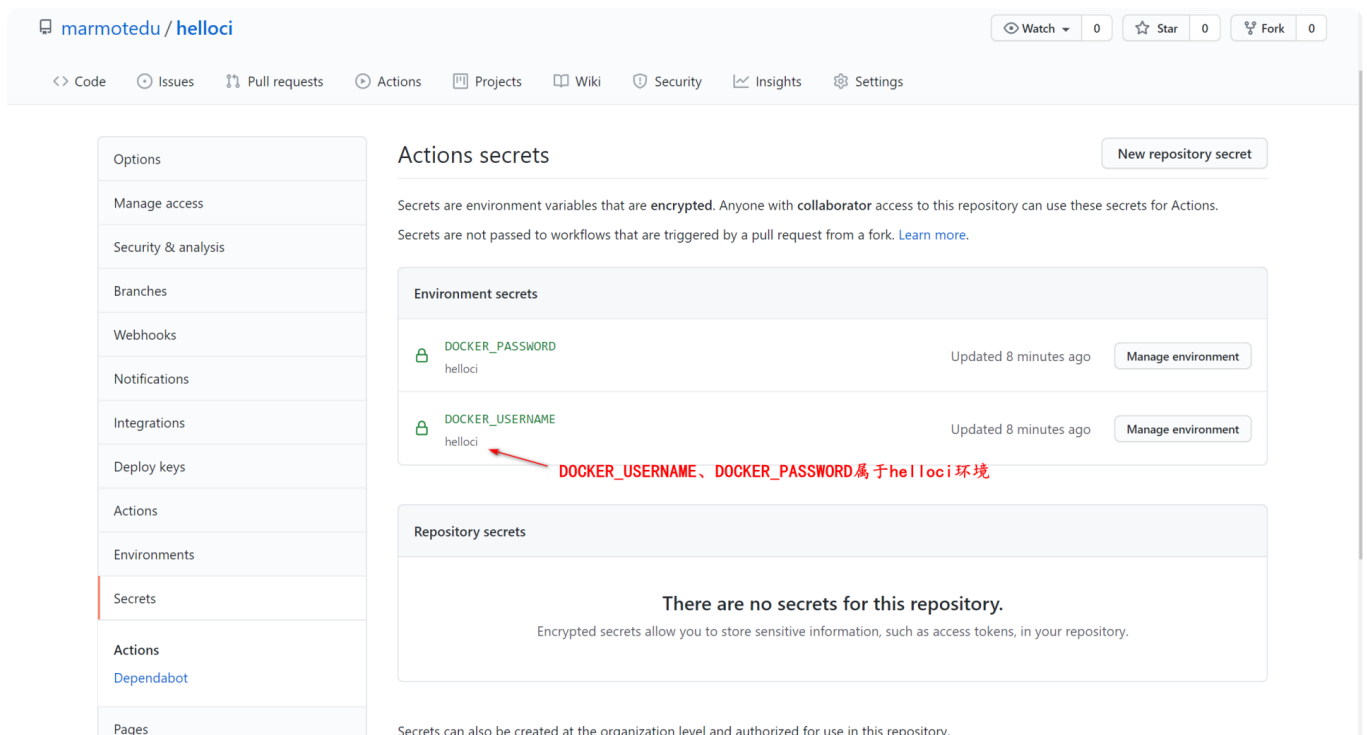
包含了多个步骤（Step），每个步骤又包含一些动作（Action）。

上面的 workflow 配置会按顺序执行下面的 6 个步骤。

1. 准备一个 Go 编译环境。
2. 从 [marmotedu/helloCI](#) 下载源码。
3. 添加或删除缺失的依赖包。
4. 编译 Go 源码。
5. 上传构建产物。
6. 构建镜像，并将镜像 push 到 `ccr.ccs.tencentyun.com/marmotedu/helloCI:beta`。

第五步，在 push 代码之前，我们需要先创建 DOCKER_USERNAME 和 DOCKER_PASSWORD secret。

其中，DOCKER_USERNAME 保存腾讯云镜像服务（CCR）的用户名，DOCKER_PASSWORD 保存 CCR 的密码。我们将这两个 secret 保存在 helloCI Environments 中，如下图所示：

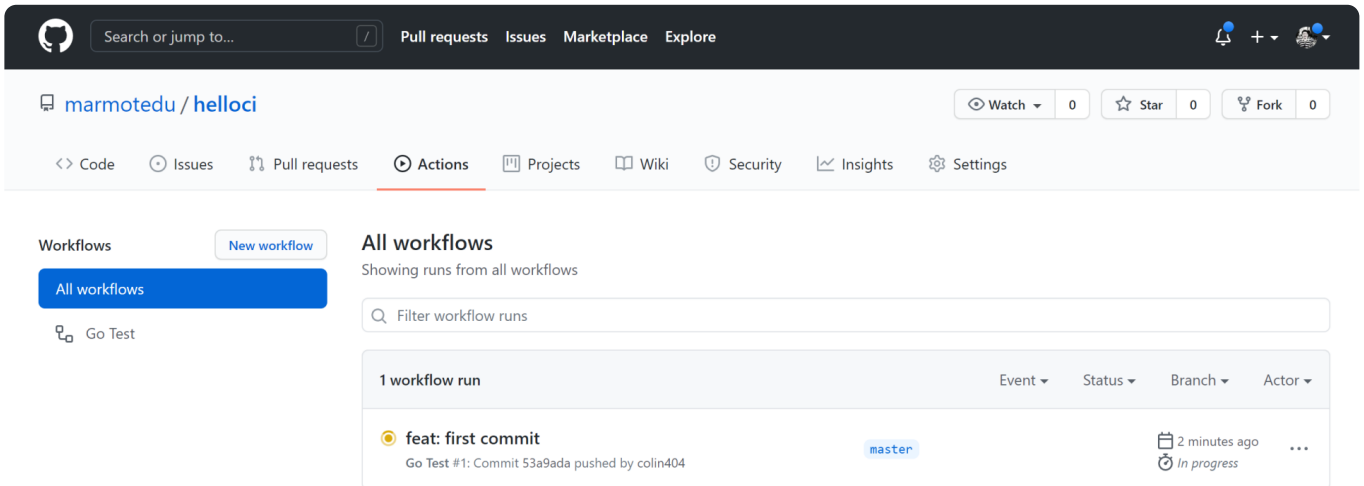


第六步，将项目 push 到 GitHub，触发 workflow 工作流：

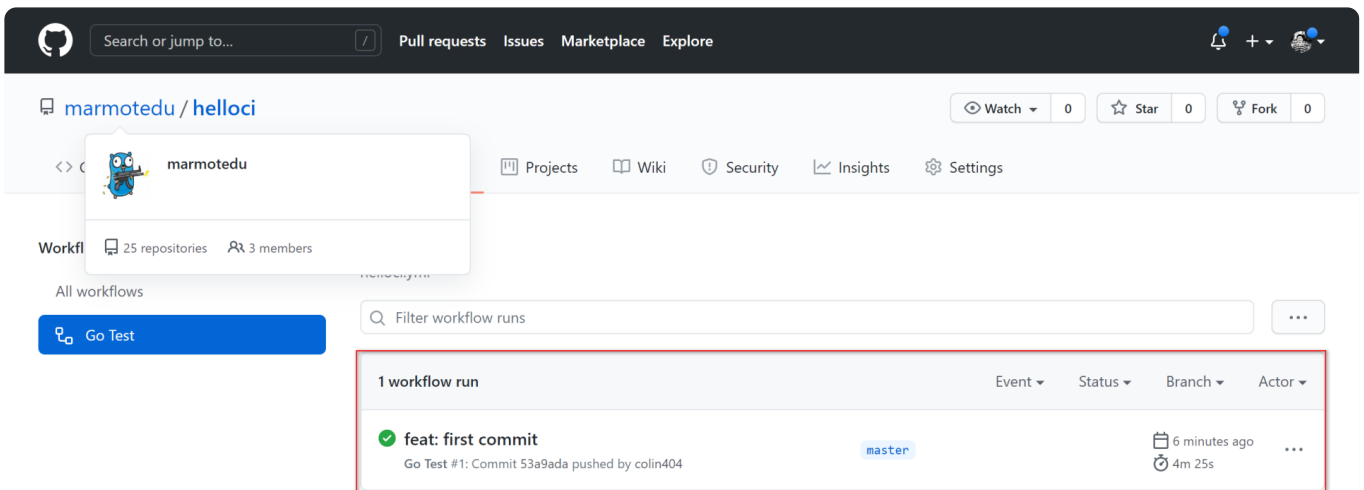
[复制代码](#)

```
1 $ git add .  
2 $ git push origin master
```

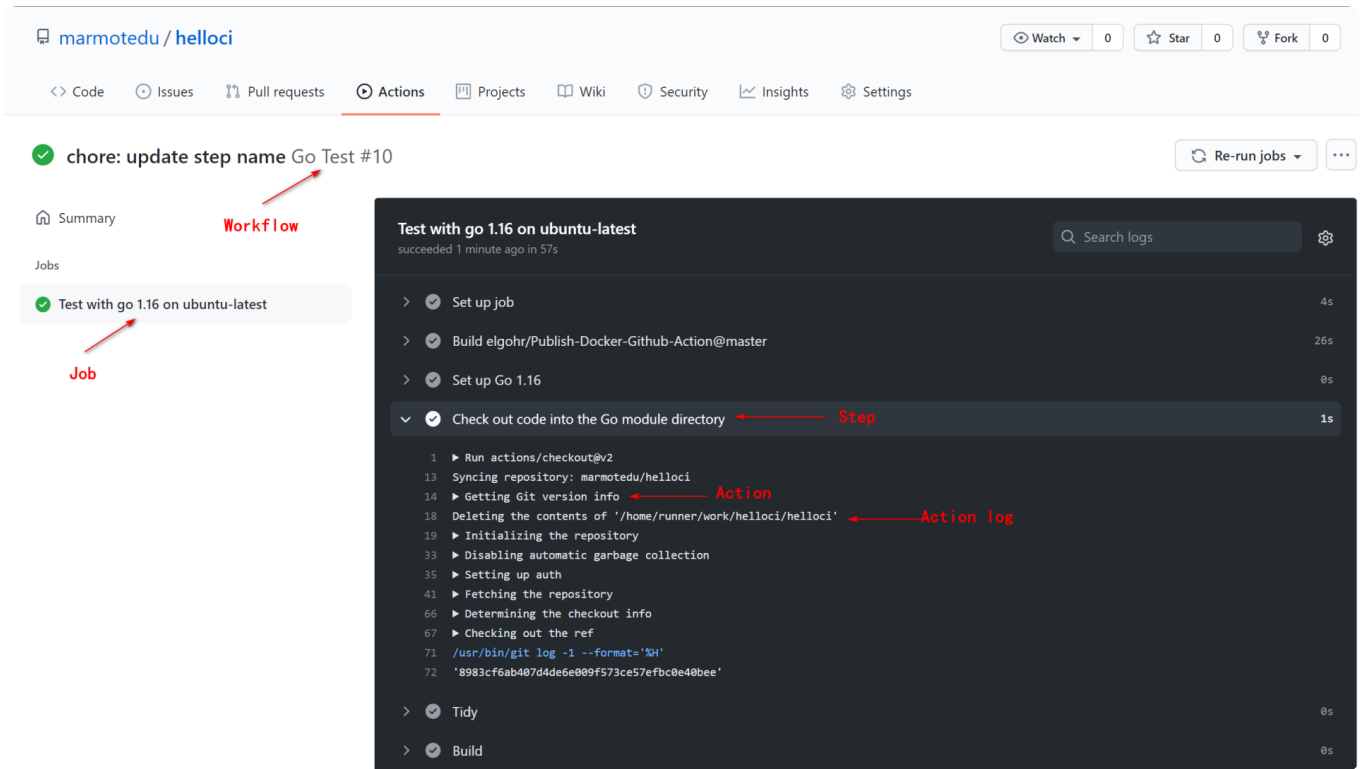
打开我们的仓库 Actions 标签页，可以发现 GitHub Actions workflow 正在执行：



等 workflow 执行完，点击 **Go Test** 进入构建详情页面，在详情页面能够看到我们的构建历史：



然后，选择其中一个构建记录，查看其运行详情（具体可参考 [chore: update step name Go Test #10](#)）：



你可以看到，Go Test 工作流程执行了 6 个 Job，每个 Job 执行了下面这些自定义 Step：

1. Set up Go 1.16。
2. Check out code into the Go module directory。
3. Tidy。
4. Build。
5. Collect main.go file。
6. Publish to Registry。

其他步骤是 GitHub Actions 自己添加的步骤：Setup Job、Post Check out code into the Go module directory、Complete job。点击每一个步骤，你都能看到它们的详细输出。

IAM GitHub Actions 实战

接下来，我们再来看下 IAM 项目的 GitHub Actions 实战。

假设 IAM 项目根目录为 `${IAM_ROOT}`，它的 workflow 配置文件为：

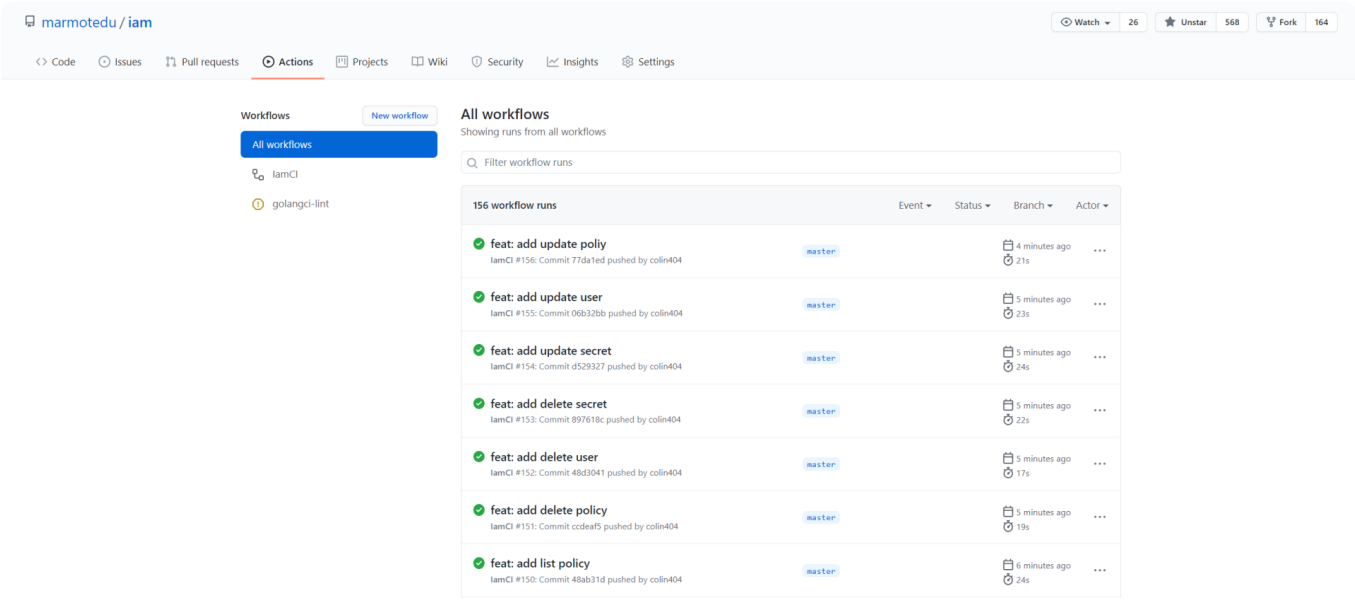
```
1 $ cat ${IAM_ROOT}/.github/workflows/iamci.yaml
2 name: IamCI
3
4 on:
5   push:
6     branches:
7       - '*'
8   pull_request:
9     types: [opened, reopened]
10
11 jobs:
12
13   iamci:
14     name: Test with go ${ matrix.go_version } on ${ matrix.os }
15     runs-on: ${ matrix.os }
16     environment:
17       name: iamci
18
19     strategy:
20       matrix:
21         go_version: [1.16]
22         os: [ubuntu-latest]
23
24     steps:
25
26       - name: Set up Go ${ matrix.go_version }
27         uses: actions/setup-go@v2
28         with:
29           go-version: ${ matrix.go_version }
30         id: go
31
32       - name: Check out code into the Go module directory
33         uses: actions/checkout@v2
34
35       - name: Run go modules Tidy
36         run: |
37           make tidy
38
39       - name: Generate all necessary files, such as error code files
40         run: |
41           make gen
42
43       - name: Check syntax and styling of go sources
44         run: |
45           make lint
46
47       - name: Run unit test and get test coverage
48         run: |
49           make cover
50
```

```
51     - name: Build source code for host platform
52       run: |
53         make build
54
55     - name: Collect Test Coverage File
56       uses: actions/upload-artifact@v1.0.0
57       with:
58         name: main-output
59         path: _output/coverage.out
60
61     - name: Set up Docker Buildx
62       uses: docker/setup-buildx-action@v1
63
64     - name: Login to DockerHub
65       uses: docker/login-action@v1
66       with:
67         username: ${ secrets.DOCKERHUB_USERNAME }
68         password: ${ secrets.DOCKERHUB_TOKEN }
69
70     - name: Build docker images for host arch and push images to registry
71       run: |
72         make push
```

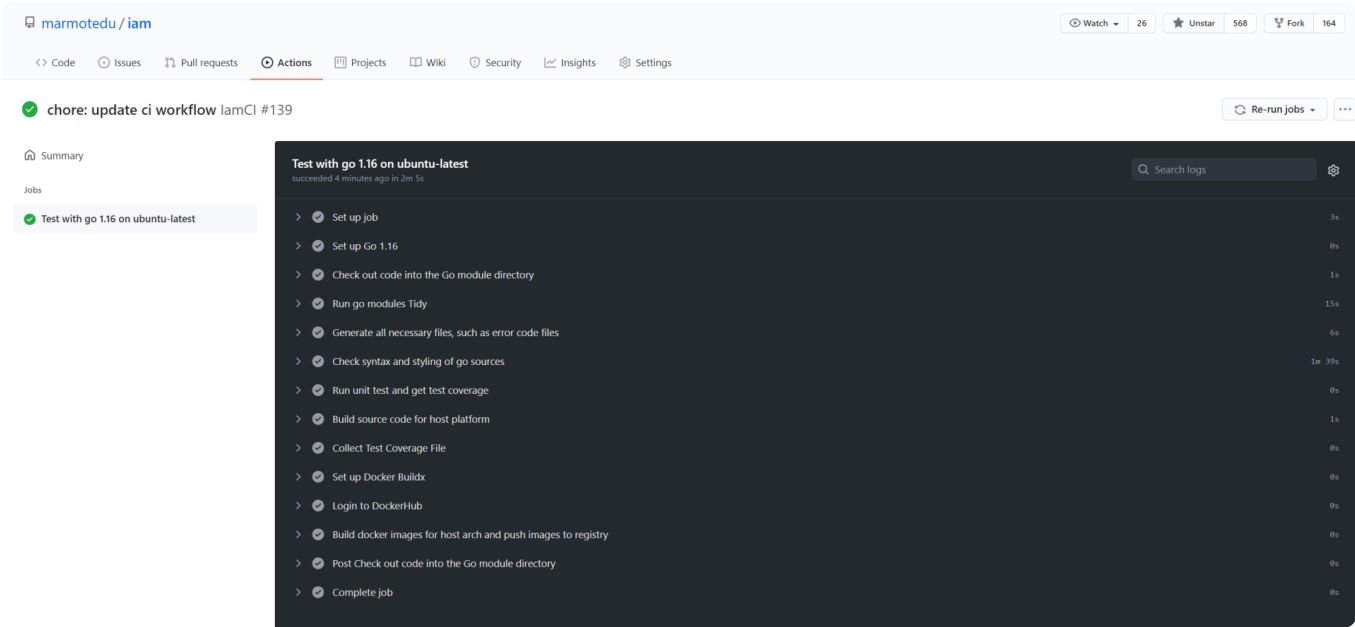
上面的 workflow 依次执行了以下步骤：

1. 设置 Go 编译环境。
2. 下载 IAM 项目源码。
3. 添加 / 删除不需要的 Go 包。
4. 生成所有的代码文件。
5. 对 IAM 源码进行静态代码检查。
6. 运行单元测试用例，并计算单元测试覆盖率是否达标。
7. 编译代码。
8. 收集构建产物_output/coverage.out。
9. 配置 Docker 构建环境。
10. 登陆 DockerHub。
11. 构建 Docker 镜像，并 push 到 DockerHub。

IamCI workflow 运行历史如下图所示：



lamCI workflow 的其中一次工作流程运行结果如下图所示：



总结

在 Go 项目开发中，我们需要通过 CI 任务来将需要频繁操作的任务自动化，这不仅可以提高开发效率，还能减少手动操作带来的失误。这一讲，我选择了最易实践的 GitHub Actions，来给你演示如何构建 CI 任务。

GitHub Actions 支持通过 push 事件来触发 CI 流程。一个 CI 流程其实就是一个 workflow，workflow 中包含多个任务，这些任务是可以并行执行的。一个任务又包含多个步骤，每一步又由多个动作组成。动作（Action）其实是一个命令 / 脚本，用来完成我们指定的任务，如编译等。

因为 GitHub Actions 内容比较多，这一讲只介绍了一些核心的知识，更详细的 GitHub Actions 教程，你可以参考 [🔗 官方中文文档](#)。


课后练习

1. 使用 CODING 实现 IAM 的 CI 任务，并思考下：GitHub Actions 和 CODING 在 CI 任务构建上，有没有本质的差异？
2. 这一讲，我们借助 GitHub Actions 实现了 CI，请你结合前面所学的知识，实现 IAM 的 CD 功能。欢迎提交 Pull Request。

这是我们这门课的最后一次练习题了，欢迎把你的思考和想法分享在留言区，也欢迎把课程分享给你的同事、朋友，我们一起交流，一起进步。

分享给需要的人，Ta 订阅后你可得 **24 元现金奖励**

 生成海报并分享

 赞 3  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 50 | 服务编排（下）：基于Helm的服务编排部署实战

下一篇 特别放送 | 给你一份清晰、可直接套用的Go编码规范

更多学习推荐

175 道 Go 工程师 大厂常考面试题

限量免费领取 



精选留言 (3)

 写留言



pedro

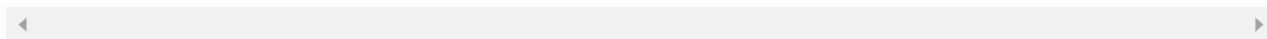
2021-09-28

最后一讲留个言，专栏基本覆盖 Go 技术栈的方方面面，还有很多工具的加餐，项目开发规范，云原生，容器等知识，物超所值。

代码质量很高，学习了很多，一路走来，多谢了~

展开 

作者回复: 感谢老哥能够坚持学习到最后



 6



Realm

2021-09-29

很专业、很系统，感谢老师的指引。

内容覆盖了编程技巧、工程化、云原生实践的经验总结，当然还有加餐鸡腿。

收获很大，谢谢老师！

展开 



1

**随风而过**

2021-09-30

整个专栏质量很高，文案虽然有些瑕疵，不影响整体专栏的专业度，专栏介绍了很多编程规范，主要还是云原生范畴内，看完整个专栏有很多反思，对go语言自我的认知有一个全新的提高(比如项目目录参杂其他语言的习惯目录结构来做是错误的，还有代码规范也会参照其他语言来组织)。

也到说再见的时候了，希望老师在出高质量的专栏，订阅破万，与君共勉。。。。

展开 ∨

