

01：github CI/CD的基本操作

实验一：创建并运行你的第一个 GitHub Actions 工作流 (Workflow)

实验目标

在GitHub 中配置和运行你的第一个 CI/CD 工作流 (Workflow) 。

步骤 1：了解 Runners (运行器)

在 GitHub Actions 中，**Runners** 是运行您 CI/CD 作业的服务器。

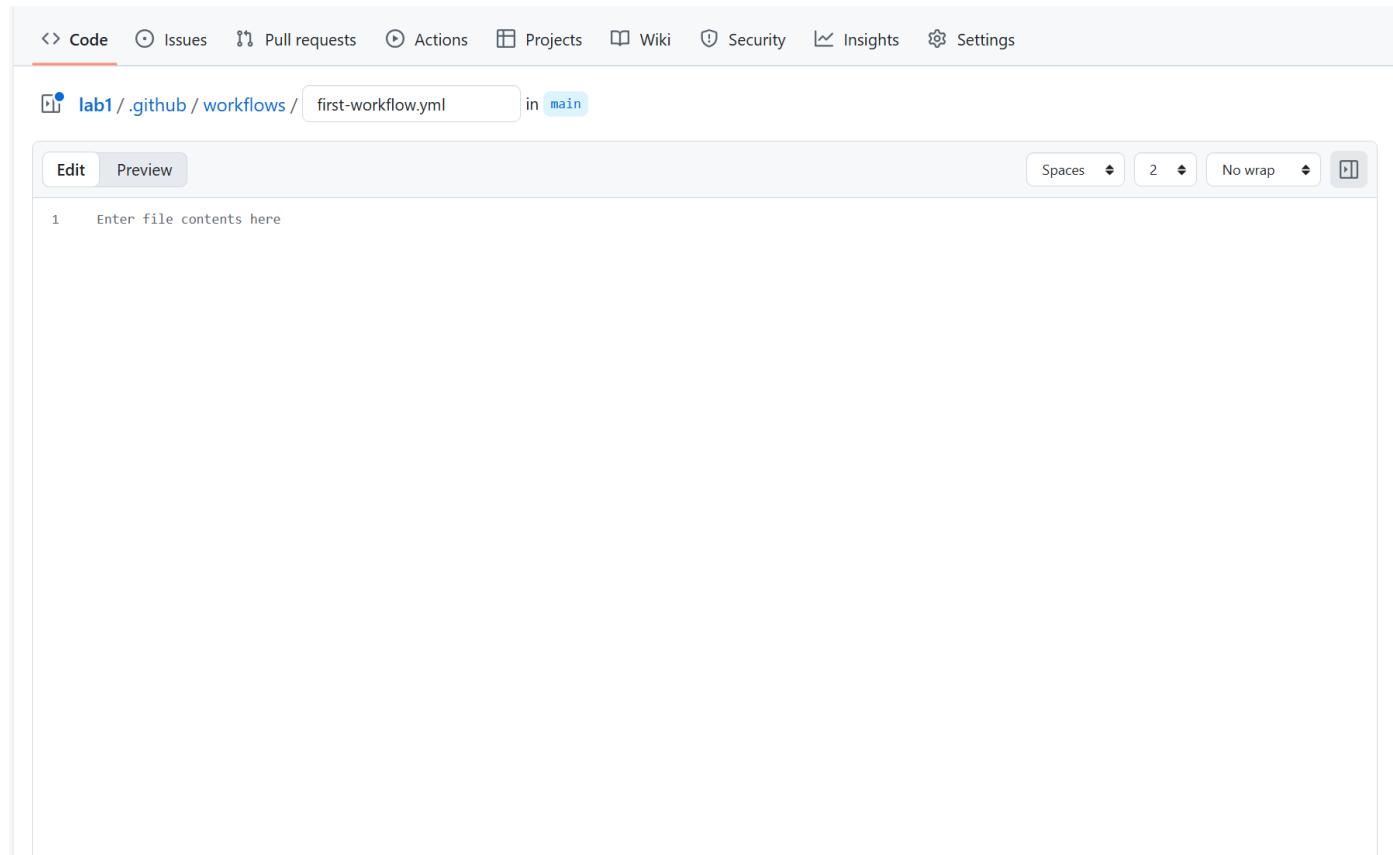
- **GitHub-hosted Runners (托管运行器)**: 如果您使用的是 [GitHub.com](#)，您可以直接使用 GitHub 提供的云端虚拟机（支持 Linux, Windows, macOS）。通常我们只需在配置中指定 `runs-on: ubuntu-latest` 即可。
- **Self-hosted Runners (自托管运行器)**: 如果您需要使用自己的机器，也可以在仓库的 `Settings > Actions > Runners` 中进行配置（本教程暂不涉及）。

步骤 2：创建 Workflow 配置文件

在 GitHub Actions 中，配置文件存放在 `.github/workflows/` 目录中，且必须是 **YAML** 格式。

操作步骤：

1. 在浏览器中打开您的 GitHub 仓库页面。
2. 点击 **Code** 选项卡。
3. 点击 **Add file** > **Create new file**。
4. 在文件名输入框中，输入完整路径：`.github/workflows/first-workflow.yml` (文件名可以是任何你喜欢的名字，但扩展名必须是 `.yml` 或 `.yaml`)。



步骤 3：编写 YAML 内容

在编辑器中粘贴以下代码。

代码块

```
1 name: First CI/CD Pipeline # 工作流名称
2
3 # 触发条件：这里指定在推送到 main 或 master 分支时触发
4 on:
5   push:
```

```

6     branches: [ "main", "master" ]
7     workflow_dispatch: # 允许手动在 Actions 页面点击按钮触发
8
9   jobs:
10    build-job:
11      runs-on: ubuntu-latest # 指定运行环境, 等同于 GitLab 的 Runner 标签
12      steps:
13        - name: Run build script
14          # ${{ github.actor }} 是 GitHub 的上下文变量, 对应 $GITLAB_USER_LOGIN
15          run: echo "Hello, ${{ github.actor }}!""
16
17    test-job1:
18      needs: build-job # 依赖 build-job, 确保顺序执行 (模拟 GitLab 的 stage 顺序)
19      runs-on: ubuntu-latest
20      steps:
21        - name: Run test script
22          run: echo "This job tests something"
23
24    test-job2:
25      needs: build-job # 同样依赖 build-job, test-job1 和 test-job2 将并行运行
26      runs-on: ubuntu-latest
27      steps:
28        - name: Run longer test script
29          run: |
30            echo "This job tests something, but takes more time than test-job1."
31            echo "After the echo commands complete, it runs the sleep command
32            for 20 seconds"
33            echo "which simulates a test that runs 20 seconds longer than test-
34            job1"
35            sleep 20
36
37    deploy-prod:
38      needs: [test-job1, test-job2] # 只有当两个测试作业都成功后才运行
39      runs-on: ubuntu-latest
40      environment: production # GitHub 的环境部署功能
41      steps:
42        - name: Deploy script
43          # ${{ github.ref_name }} 对应 $CI_COMMIT_BRANCH
44          run: echo "This job deploys something from the ${{ github.ref_name }}"
45          branch."

```

步骤 4：查看工作流和作业状态（需要截图！！！）

提交文件后，GitHub 会自动检测到 `.github/workflows/` 目录下的变更并启动工作流。

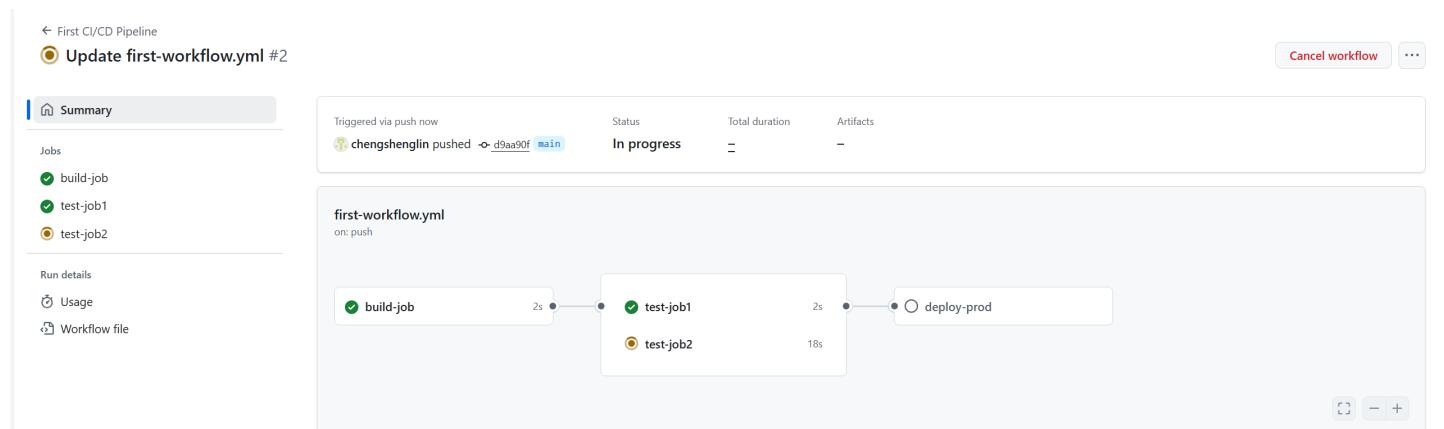
1. 在仓库顶部导航栏，点击 **Actions**。
2. 在左侧列表中，点击 **First CI/CD Pipeline**（或者所有 workflows）。
3. 你会看到一个正在运行或已完成的记录，点击它。

The screenshot shows the GitHub Actions interface. On the left, there's a sidebar with 'Actions' selected, followed by 'All workflows' (which is highlighted in blue). Below that are sections for 'First CI/CD Pipeline', 'Management' (with options for Caches, Deployments, Attestations, Runners, Usage metrics, and Performance metrics), and a 'New workflow' button. The main area is titled 'All workflows' and shows 'Showing runs from all workflows'. It displays a single workflow run for 'Update first-workflow.yml' triggered by a push to the 'main' branch. The run status is 'In progress', and it was triggered 1 minute ago. A filter bar at the top right allows searching for workflow runs.

可视化图表

点击进入后，你会看到一个可视化的流程图（Graph），展示了作业之间的依赖关系：

- `build-job` 最先运行。
- `build-job` 成功后，`test-job1` 和 `test-job2` 并行运行。
- 当测试作业全部完成后，`deploy-prod` 运行。



查看日志

点击图表中的任意作业名称（例如 `deploy-prod`），您可以查看详细日志。

您会看到 `echo` 命令输出的文本，以及系统自动填充的变量值。

The screenshot shows the GitHub Actions pipeline interface. On the left, there's a sidebar with options like 'Summary', 'Jobs' (listing 'build-job', 'test-job1', 'test-job2', and 'deploy-prod'), 'Run details', 'Usage', and 'Workflow file'. The main area displays a log for a job named 'deploy-prod' which succeeded 1 minute ago. The log is divided into sections: 'Set up job', 'Deploy script', and 'Complete job'. The 'Deploy script' section contains the following code:

```
1 Current runner version: '2.329.0'
2 ▶ Runner Image Provisioner
7 ▶ Operating System
11 ▶ Runner Image
16 ▶ GITHUB_TOKEN Permissions
20 Secret source: Actions
21 Prepare workflow directory
22 Prepare all required actions
23 Complete job name: deploy-prod

1 ▶ Run echo "This job deploys something from the main branch."
4 This job deploys something from the main branch.

1 Cleaning up orphan processes
```

实验二：构建 Docker 镜像并发布到 GitHub Packages

在上一节教程中，我们学习了 GitHub Actions 的基本语法。为了让大家掌握云计算环境中真正的 CI/CD 流程，本次作业将要求大家完成一个真实的容器化部署流水线。

实验目标

- 掌握如何在 Workflow 中使用第三方 Action（如 `actions/checkout`, `docker/build-push-action`）。
- 理解 Job 之间的依赖关系与数据传递。
- 核心考核点：**实现代码提交后，自动构建 Docker 镜像并推送到 GitHub Container Registry。

准备工作 (项目代码)

我们需要一个包含 `Dockerfile` 的简单应用。请在 GitHub 上新建一个**公开仓库**（例如命名为 `ci-cd-practice`），并在本地或直接在网页端创建以下三个文件：

1. `app.js` (简单的 Node.js 应用)

代码块

```
1 const http = require('http');
2
3 const server = http.createServer((req, res) => {
4   res.statusCode = 200;
5   res.setHeader('Content-Type', 'text/plain');
6   res.end('Hello Cloud Computing Class!');
7 });
8
9 server.listen(3000, () => {
10   console.log('Server running on port 3000');
11});
```

2. `package.json` (项目配置)

代码块

```
1 {
2   "name": "ci-cd-practice",
3   "version": "1.0.0",
4   "scripts": {
5     "test": "echo \\"Error: no test specified\\" && exit 0"
6   },
7   "dependencies": {}
8 }
```

(注：为了简化，这里 *test* 脚本直接返回成功 *exit 0*)

3. `Dockerfile` (容器构建描述)

代码块

```
1 FROM node:18-alpine
2 WORKDIR /app
```

```
3 COPY package.json ./
4 RUN npm install
5 COPY . .
6 EXPOSE 3000
7 CMD ["node", "app.js"]
```

任务要求：编写 Workflow

请在仓库中创建 `.github/workflows/docker-publish.yml` 文件。

你需要编写一个包含两个 Job 的工作流：

1. **Test Job:** 检出代码并运行 `npm test`。

2. **Build-and-Push Job:**

- * 依赖 Test Job (只有测试通过才构建)。
- * 登录 GitHub Container Registry。
- * 构建 Docker 镜像。
- * 将镜像推送到 GHCR。

参考代码 (请理解后填入)

代码块

```
1 name: Docker Build and Push
2
3 on:
4   push:
5     branches: [ "main" ]
6
7   # 这是一个重要的设置：定义 GITHUB_TOKEN 的权限
8   # 我们需要写入 Packages 的权限来推送镜像
9   permissions:
10    contents: read
```

```
11 packages: write
12
13 jobs:
14   # 第一步: 测试代码
15   test:
16     runs-on: ubuntu-latest
17     steps:
18       - name: Checkout repository
19         uses: actions/checkout@v4
20
21       - name: Setup Node.js
22         uses: actions/setup-node@v4
23         with:
24           node-version: '18'
25
26       - name: Install dependencies
27         run: npm install
28
29       - name: Run tests
30         run: npm test
31
32   # 第二步: 构建并推送 Docker 镜像
33   build-and-push:
34     needs: test # 只有测试通过才运行
35     runs-on: ubuntu-latest
36     steps:
37       - name: Checkout repository
38         uses: actions/checkout@v4
39
40       # 设置 Docker Buildx 环境 (Docker 官方推荐)
41       - name: Set up Docker Buildx
42         uses: docker/setup-buildx-action@v3
43
44       # 登录到 GitHub Container Registry (ghcr.io)
45       # ${{ secrets.GITHUB_TOKEN }} 是 GitHub 自动生成的密钥, 无需手动设置
46       - name: Log in to the Container registry
47         uses: docker/login-action@v3
48         with:
49           registry: ghcr.io
50           username: ${{ github.actor }}
51           password: ${{ secrets.GITHUB_TOKEN }}
52
53       # 构建并推送
54       - name: Build and push Docker image
55         uses: docker/build-push-action@v5
56         with:
57           context: .
```

```
58      push: true
59      # 注意：镜像名称必须全部小写
60      tags: ghcr.io/${{ github.repository_owner }}/ci-cd-practice:latest
```

作业提交与验收

截图验收三个部分

检查点 1：Actions 状态

进入你的仓库 `Actions` 标签页，应该看到名为 "Docker Build and Push" 的工作流显示绿色对勾 (Success)。

检查点 2：工作流日志

点进工作流详情，我应该能看到 `test` 和 `build-and-push` 两个作业依次执行成功。

检查点 3：产物验证 (Artifacts)

1. 进入仓库主页。
2. 点击右侧侧边栏的 `Packages`。
3. 应该能看到一个名为 `ci-cd-practice` 的 Docker 镜像包。

进阶选做实验（可选）

如果你已经完成了基础流水线的搭建，可以尝试以下任意一个实验。

选项一：实现“版本动态管理”（DevOps 核心思维）

- 目标：

目前我们的镜像 Tag 是写死的 `:latest`，这会导致旧镜像被覆盖，无法回滚。要求修改 Workflow，使得每次生成的镜像都有唯一的“身份证”。

- 任务描述：

- a. 修改 `.github/workflows/docker-publish.yml` 文件。
- b. 不再仅使用 `:latest`，而是同时使用 **Git Commit SHA**（完整哈希或前7位）作为镜像标签。
- c. 预期结果：在 Packages 页面看到类似 `ci-cd-practice:8f32c9a...` 的镜像版本，且旧版本依然保留。

提示

- 查阅 GitHub Context 文档，找到 `${{ github.sha }}` 变量。
- Docker Action 的 `tags` 参数支持多行文本（使用 `|` 符号）。能不能同时打上 `:latest` 和 `:${{ github.sha }}` 两个标签？这是生产环境的最佳实践。

选项二：添加“代码质量检查”（CI 扩展）

- 目标：

在构建镜像之前，增加一道“安检”工序。如果代码格式不规范，**直接拒绝构建**，实现 CI/CD 的“左移”原则。

- 任务描述：

- 在 `package.json` 中添加一个 lint 脚本（建议安装 `eslint`，或者简单地写一个 `exit 1` 命令来模拟检查失败）。
- 在 Workflow 的 `test` 作业中，于 `Run tests` 步骤之前增加一个新步骤 `Run Lint`。
- 发挥点：**尝试故意写一段“坏代码”（比如违反 `eslint` 规则，或在脚本中强制返回非 0 状态码），验证 Workflow 是否会像预期那样变红（失败）并停止后续的 Docker 构建。

提示

- 你需要修改 `steps` 的执行顺序。
- 理解 GitHub Actions 的**失败机制**：如果 `test` 作业中的 Lint 步骤失败了（退出码非 0），整个 `test` 作业就会标记为失败。
- 思考：**`build-and-push` 作业配置了 `needs: test`。那么当 `test` 失败时，构建还会运行吗？这正是我们想要的效果。

选项三：可视化通知（Webhooks 集成）

- 目标：

当构建成功或失败时，给常用的通讯工具（如 Discord, Slack, 飞书, 钉钉）发一条消息，建立基础的监控告警体系。

- 任务描述：

- 申请一个 Discord/Slack/钉钉的 Webhook URL（免费且无需服务器）。
- 将这个 URL 存入仓库的 **Settings > Secrets** 中（**安全实践**：千万不要直接写在代码里）。
- 在 Workflow 的最后添加一个步骤，使用 `curlimages/curl` 或社区现成的 Action 发送一条 HTTP POST 请求。
- 预期结果：**当流水线运行结束，你的手机或电脑收到了“GitHub Actions: 构建结果”的消息。

提示

- 这是云计算中“事件驱动”和“监控告警”的雏形。
- 进阶思考：**默认情况下，如果构建失败，后续步骤会自动跳过。如何配置 `if` 条件（例如 `if: always()` 或 `if: failure()`），使得即使构建失败，通知步骤也能照常运行并发送报警？

提交要求

如果你完成了以上任意一项挑战，请在提交作业时：

1. 在实验报告中注明你选择了哪个选项。
2. **提供截图证明**（例如：Packages 页面的多版本号列表、Action 失败拦截的红色日志、或者手机收到通知的截图）。