CI 工作流详细问答文档

> 本文档详细解释 `.github/workflows/ci.yml` 的每个部分，适合初学者理解 GitHub Actions CI/CD 流程。

📋 目录

基础概念

Q1: 什么是 CI/CD?

CI (Continuous Integration - 持续集成):

- 每次代码提交后，自动编译、测试
- 及早发现问题

CD (Continuous Delivery/Deployment - 持续交付/部署):

- 自动打包、发布软件
- 自动部署到服务器

我们的 CI 工作流同时包含 CI 和 CD。

Q2: GitHub Actions 是什么?

GitHub 提供的 **免费 CI/CD 服务**:

- 代码推送时自动运行脚本
- 在 GitHub 服务器上执行
- 支持 Linux、Windows、macOS

工作流文件位置: `.github/workflows/*.yml`

Q3: 这个 CI 工作流做什么?

```
代码提交 →
  检查 DCO 签名 →
  多版本构建测试 (JDK 17/21/25) →
  生成 zip 包 →
  发布到 GitHub Releases →
  更新 GitHub Pages 网站
```

工作流触发

Q4: 什么时候会运行 CI?

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

三种触发方式:

1. **push**：任何分支有新提交时
2. **pull_request**：有人提交 PR 时
3. **workflow_dispatch**：手动点击 "Run workflow" 按钮

示例:

```
git push origin main           # ✅ 触发 CI
git push origin feature/test   # ✅ 触发 CI
# 创建 PR                      # ✅ 触发 CI
# 在 Actions 页面点击按钮       # ✅ 触发 CI
```

Q5: `concurrency` 是什么意思?

```
concurrency:
  group: ${{ github.workflow }}
  cancel-in-progress: false
```

并发控制:

- `group`: 同一个 workflow 为一组
- `cancel-in-progress: false`: 不取消正在运行的任务

行为:

- 如果 CI 正在运行，新的推送会**排队等待**
- 不会取消之前的构建

为什么这样设置?

- 构建时间短（~10 分钟）
- 保留完整的构建历史
- 避免构建被中断

Q6: `env` 环境变量有什么用?

```
env:
  output_file_name: "ruyisdk-eclipse-plugins.site.zip"
```

定义全局变量，所有 job 都可以使用:

- 集中管理文件名

- 修改一处，全局生效

使用方式：`${{ env.output_file_name }}`

Job 详解

Q7: 有哪些 Job？按什么顺序执行？

```
1. dco (DCO 检查)
   ↓
2. config (配置) + build_and_upload (构建上传) [并行]
   ↓
3. publish_releases (发布 Release) [条件执行]
   ↓
4. publish_github_pages (发布网站) [条件执行]
```

依赖关系：

- `build_and_upload` 需要 `dco` 成功
- `publish_releases` 需要 `config` + `build_and_upload` 成功
- `publish_github_pages` 需要所有前面的都成功

Job 1: DCO 检查

Q8: 什么是 DCO?

DCO (Developer Certificate of Origin)：

- 开发者来源证书
- 确认贡献者拥有代码版权
- 通过 `git commit -s` 签名

检查内容：

```
git commit -s -m "fix: bug"
               ↑
         Signed-off-by: Your Name <email>
```

Q9: DCO 检查的代码详解

```yaml
- name: Run christophebedard/dco-check
  if: ${{ github.ref_type != 'tag' }}
  uses: christophebedard/dco-check@7b0205d...
  with:
    args: "--verbose"
```

逐行解释：

代码	含义
`if: ${{ github.ref_type != 'tag' }}`	如果不是 tag，才检查（tag 不产生新提交）
`uses: christophebedard/dco-check@...`	使用第三方 Action（固定到 SHA 保证安全）
`args: "--verbose"`	详细输出检查结果
`GITHUB_TOKEN`	GitHub 自动提供的认证 token

为什么使用完整 SHA 而不是版本号?

- 安全！防止 Action 被恶意修改
- 如：`@7b0205d...` 是 `v0.5.0` 的 commit SHA

Job 2: 配置

Q10: config job 做什么?

目的：计算一些条件，供后续 job 使用

```
outputs:
  datetime: ...                    # 时间戳
  make_continuous_release: ...     # 是否发布 continuous release
  make_versioning_release: ...     # 是否发布版本 release
```

为什么单独一个 job?

- 避免重复计算
- 其他 job 通过 `needs.config.outputs.xxx` 引用

Q11: 如何判断是否发布 Release?

```
echo 'continuous=${{ github.ref_type == 'tag' || (github.ref_type == 'branch' && github.ref_name == 'main') }}' >> "$GITHUB_OUTPUT"
echo 'versioning=${{ github.ref_type == 'tag' }}' >> "$GITHUB_OUTPUT"
```

条件表格:

场景	`github.ref_type`	`github.ref_name`	continuous	versioning
推送到 main	`branch`	`main`	✅ true	❌ false
推送到 feature 分支	`branch`	`feature/xxx`	❌ false	❌ false
创建 tag v0.0.5	`tag`	`v0.0.5`	✅ true	✅ true

翻译成人话:

- **continuous release**：main 分支更新 或 创建 tag 时发布

- **versioning release**：只在创建 tag 时发布

Q12: 时间戳有什么用?

```
echo "datetime=$(TZ='Asia/Shanghai' date '+%Y-%m-%d %H:%M:%S %z')" >>
"$GITHUB_OUTPUT"
```

生成示例：`2025-11-08 16:30:00 +0800`

用途：

- 显示在网站上："最后更新: 2025-11-08 16:30:00"
- 显示在 Release Notes 中

为什么用上海时区?

- 项目主要用户在中国
- 方便查看发布时间

Job 3: 构建和上传

Q13: 为什么要在多个 JDK 版本上构建?

```yaml
strategy:
  matrix:
    java_version: [17, 21, 25]
```

Matrix 构建：并行运行 3 个任务

- ✅ JDK 17 构建测试
- ✅ JDK 21 构建测试
- ✅ JDK 25 构建测试

目的：

1. **兼容性测试**：确保在所有 JDK 版本上都能编译通过
2. **早发现问题**：如果某个 JDK 版本有语法问题，立即知道
3. **质量保证**：支持多 LTS 版本

实际构建产物只用一个（JDK 21），其他只做测试!

Q14: 构建步骤详解

```yaml
- name: Checkout
  uses: actions/checkout@v5.0.0
  with:
    fetch-depth: 0
```

`fetch-depth: 0`：

- 获取完整的 Git 历史
- Tycho 需要 Git 信息生成版本号时间戳

```yaml
- name: Set up JDK ${{ matrix.java_version }}
  uses: actions/setup-java@v5.0.0
  with:
    distribution: 'temurin'
    java-version: ${{ matrix.java_version }}
    cache: 'maven'
```

逐项解释:

- `distribution: 'temurin'`：使用 Eclipse Temurin JDK（开源）
- `java-version: ${{ matrix.java_version }}`：安装当前矩阵的 JDK 版本
- `cache: 'maven'`：缓存 Maven 依赖，加速构建

```yaml
- name: Build
  run: |
    set -ex
    mvn clean verify
```

`set -ex`:

- `set -e`：任何命令失败立即退出
- `set -x`：打印每条执行的命令（方便调试）

Q15: 为什么只有 JDK 21 打包?

```yaml
- name: Package for Release
  if: ${{ env.java_version_for_release == matrix.java_version }}
```

条件判断:

- `java_version_for_release: 21`（第 58 行定义）
- 只有当前 matrix 是 JDK 21 时执行

为什么:

- JDK 17/25 只做**测试**
- JDK 21 用于**正式发布**
- 避免重复打包（节省时间和空间）

Q16: 打包步骤详解

```
input_dir_path="$PWD/sites/repository/target/repository/"
output_file_path="$(mktemp -d)/${{ env.output_file_name }}"
pushd "$input_dir_path"
  echo 'output_file_list<<EOF' >> $GITHUB_OUTPUT
    find . -type f >> $GITHUB_OUTPUT
  echo 'EOF' >> $GITHUB_OUTPUT
  find . -type f -print0 | xargs -0 zip -9 "$output_file_path"
popd
```

逐步解释:

1. **定义路径**:

   ```
   input_dir_path="$PWD/sites/repository/target/repository/"  # P2 仓库目录
   output_file_path="$(mktemp -d)/ruyisdk-eclipse-plugins.site.zip"  # 临时目录中的 zip
   ```

2. **进入 P2 仓库目录**:

   ```
   pushd "$input_dir_path"  # 类似 cd，但会记住之前的位置
   ```

3. **记录文件列表**（用于 Release Notes）：

   ```
   echo 'output_file_list<<EOF' >> $GITHUB_OUTPUT  # 开始多行输出
   find . -type f >> $GITHUB_OUTPUT                # 列出所有文件
   echo 'EOF' >> $GITHUB_OUTPUT                    # 结束多行输出
   ```

   示例输出:

   ```
   ./artifacts.jar
   ./content.jar
   ./features/org.ruyisdk.feature_0.0.5.jar
   ./plugins/org.ruyisdk.core_0.0.5.jar
   ...
   ```

4. **压缩成 zip**:

   ```
   find . -type f -print0 | xargs -0 zip -9 "$output_file_path"
   ```

   逐部分解释:

   - `find . -type f`：找到所有文件
   - `-print0`：用 null 分隔（支持空格文件名）
   - `xargs -0`：读取 null 分隔的输入
   - `zip -9`：最高压缩率（9）

5. **返回原目录**:

   ```
   popd  # 回到之前的目录
   ```

Q17: `GITHUB_OUTPUT` 是什么?

作用：在 job 的不同 step 之间传递数据

写入方式：

```
echo "变量名=值" >> $GITHUB_OUTPUT
```

读取方式：

```
${{ steps.步骤id.outputs.变量名 }}
```

示例：

```
# Step 1: 写入
- id: packaging
  run: echo "output_file_path=/tmp/xxx.zip" >> $GITHUB_OUTPUT

# Step 2: 读取
- run: echo ${{ steps.packaging.outputs.output_file_path }}
```

Q18: 为什么要上传 Artifact?

```
- name: Upload Artifact for Release
  uses: actions/upload-artifact@v5.0.0
  with:
    name: ${{ env.output_file_name }}
    path: ${{ steps.packaging.outputs.output_file_path }}
```

Artifact：GitHub Actions 的临时文件存储

作用：

1. 在不同 job 之间传递文件
2. 构建产物可以下载查看
3. 保留 30 天（默认）

流程：

```
build_and_upload (上传 zip)
    ↓
publish_releases (下载 zip 并发布)
    ↓
publish_github_pages (下载 zip 并解压到网站)
```

为什么不直接传递文件?

- 不同 job 运行在不同的服务器上
- 需要通过 Artifact 机制共享文件

Job 4: 发布 Releases

Q19: 什么时候发布 Release?

```
if: ${{ needs.config.outputs.make_continuous_release == 'true' }}
```

条件:

- ✅ 推送到 main 分支：发布 **Continuous Release**
- ✅ 创建 tag：发布 **Continuous Release** + **Versioning Release**
- ❌ 推送到 feature 分支：不发布

Q20: Continuous Release 和 Versioning Release 有什么区别?

类型	触发条件	Tag	Latest	Prerelease	用途
Continuous	main 更新或 tag	`continuous`	❌	✅	开发测试版
Versioning	创建 tag	`v0.0.5` 等	✅	❌	正式版本

举例:

```
git push origin main              # → Continuous Release (覆盖旧的)
git tag v0.0.5 && git push --tags # → Continuous + Versioning Release
```

Continuous Release:

- 总是最新的 main 分支构建
- 自动覆盖旧版本
- 标记为"预发布"

Versioning Release:

- 正式版本（如 v0.0.5）
- 永久保留
- 标记为"最新版本"

Q21: 为什么要删除旧的 Continuous Release?

```
- name: Delete Old Continuous Releases
  run: |
    if gh release view 'continuous'; then
      gh release delete 'continuous' --cleanup-tag --yes

      while gh release view 'continuous'; do
        echo 'the release is still there'
        sleep 5
      done
    fi
```

原因:

- Continuous Release 只保留**最新的一个**
- 避免积累过多测试版本

流程:

1. 检查是否存在 `continuous` release
2. 如果存在，删除它（包括 tag）
3. 等待删除完成（GitHub API 异步）
4. 创建新的 `continuous` release

`gh` **命令**：GitHub CLI 工具

- `gh release view`：查看 release
- `gh release delete`：删除 release
- `--cleanup-tag`：同时删除 tag
- `--yes`：不要确认提示

Q22: Release Notes 如何生成?

```
- name: Prepare Release Notes
  run: |
    for f in $(find '.github/resources/release/' -name '*.md'); do
      perl -i.bak -pe 's/%FILE_LIST%/$ENV{"output_file_list"}/g' "$f" && rm
"$f.bak"
    done
```

工作原理:

1. **模板文件**:
   - `release_note_continuous.md`
   - `release_note_versioning.md`
2. **模板中的占位符**:

   ```
   ## 包含文件
   %FILE_LIST%
   ```

3. **Perl 替换**:

   ```
   s/%FILE_LIST%/$ENV{"output_file_list"}/g
   ```

   - 将 `%FILE_LIST%` 替换为实际的文件列表
   - `$ENV{"output_file_list"}`：环境变量中的文件列表
4. **生成结果**:

```
## 包含文件
./artifacts.jar
./content.jar
./features/org.ruyisdk.feature_0.0.5.jar
...
```

Q23: 如何发布 Release?

```yaml
- name: Publish a Continuous Release
  uses: softprops/action-gh-release@6da8fa93...
  with:
    name: Continuous
    tag_name: continuous
    make_latest: false
    prerelease: true
    generate_release_notes: false
    body_path: .github/resources/release/release_note_continuous.md
    files: |
      ${{ env.output_file_name }}
```

参数详解:

参数	值	含义
`name`	Continuous	Release 显示名称
`tag_name`	continuous	Git tag 名称
`make_latest`	false	不标记为"最新版本"
`prerelease`	true	标记为"预发布版"
`generate_release_notes`	false	不自动生成（使用自定义）
`body_path`	...md	Release 说明文件路径
`files`	*.zip	附加的文件

Versioning Release 的区别:

```yaml
make_latest: true           # ✅ 标记为最新
prerelease: false           # ❌ 不是预发布
generate_release_notes: true # ✅ 自动生成 changelog
```

Job 5: 发布 GitHub Pages

Q24: GitHub Pages 是什么?

免费的静态网站托管:

- 每个仓库一个网站
- URL: `https://用户名.github.io/仓库名/`
- 用于托管文档、下载页面、P2 仓库

我们的用途:

- 托管 P2 更新站点（Eclipse 安装源）
- 托管网站首页（中英文）

Q25: 什么时候更新 GitHub Pages?

```
if: ${{ needs.config.outputs.make_versioning_release == 'true' }}
```

只在创建 tag 时更新!

原因:

- GitHub Pages 是**正式发布渠道**
- 不应该频繁更新（避免用户安装不稳定版本）
- main 分支的测试版本只发布到 Releases

Q26: GitHub Pages 发布流程详解

```
- name: Extract Artifact Content and Prepare Index Page
  run: |
    output_dir_path='.github/resources/website/'
    pushd "$output_dir_path"
      unzip '下载的zip'
      for f in 'index.html' 'index.zh_Hans.html'; do
        sed -i "s!%PUBLISH_DATE%!$datetime!g" "$f"
        perl -i.bak -pe 's/%FILE_LIST%/$ENV{"output_file_list"}/g' "$f" && rm
"$f.bak"
      done
    popd
```

步骤分解:

1. **解压 P2 仓库到网站目录**:

```
.github/resources/website/
├── index.html           # 首页（英文）
├── index.zh_Hans.html   # 首页（中文）
├── favicon.ico          # 网站图标
├── ruyi-logo-720.svg    # Logo
├── artifacts.jar        # ← 解压的 P2 仓库
├── content.jar
├── features/
└── plugins/
```

2. **替换网页中的占位符**:
   - `%PUBLISH_DATE%` → 实际发布时间
   - `%FILE_LIST%` → 文件列表
3. **结果**:
   - 用户访问 `https://xxx.github.io/...` 看到漂亮的首页
   - Eclipse 访问同一个 URL 可以安装插件

```yaml
- name: Upload static files as artifact
  uses: actions/upload-pages-artifact@v4.0.0
  with:
    path: ${{ steps.extraction.outputs.output_dir_path }}
```

上传到 GitHub Pages 专用 Artifact

```yaml
- name: Deploy to GitHub Pages
  id: deployment
  uses: actions/deploy-pages@v4.0.5
```

实际部署:

- GitHub 将 Artifact 发布到 Pages
- 几分钟后网站更新

权限要求:

```yaml
permissions:
  pages: write      # 写入 Pages
  id-token: write   # OIDC 认证
```

发布流程总览

Q27: 完整的发布流程是怎样的?

场景 1：推送到 feature 分支

```
git push origin feature/test
    ↓
✅ DCO 检查
✅ JDK 17/21/25 构建测试
❌ 不发布任何东西
```

结果：只做 CI 检查，不发布

场景 2：推送到 main 分支

```
git push origin main
    ↓
✅ DCO 检查
✅ JDK 17/21/25 构建测试
✅ 生成 zip 包 (JDK 21)
✅ 发布 Continuous Release (覆盖旧的)
    - Tag: continuous
    - 附件: ruyisdk-eclipse-plugins.site.zip
❌ 不更新 GitHub Pages
```

结果：开发者可以从 Releases 下载最新测试版

场景 3：创建 tag

```
git tag v0.0.5 && git push --tags
    ↓
✅ DCO 检查（跳过，tag 没有新提交）
✅ JDK 17/21/25 构建测试
✅ 生成 zip 包 (JDK 21)
✅ 发布 Continuous Release
✅ 发布 Versioning Release
    - Tag: v0.0.5
    - 标记为"最新版本"
    - 附件: ruyisdk-eclipse-plugins.site.zip
✅ 更新 GitHub Pages
    - 解压 zip 到网站目录
    - 更新首页时间戳和文件列表
    - 部署到 https://xxx.github.io/...
```

结果：正式版本发布，用户可以在线/离线安装

常见问题

Q28: 为什么 CI 失败了?

常见原因:

1. **DCO 检查失败**:

   ```
   ❌ Commit xxx is missing Signed-off-by line
   ```

   解决: 使用 `git commit -s` 或 `git commit --amend -s`

2. **编译失败**:

   ```
   ❌ Compilation failure in JDK 25
   ```

   解决: 修复代码，确保兼容所有 JDK 版本

3. **权限问题**:

   ```
   ❌ Permission denied to deploy to GitHub Pages
   ```

   解决: 检查仓库设置 → Pages → Source

Q29: 如何手动触发 CI?

1. 访问: `https://github.com/your-repo/actions`
2. 点击左侧 "CI" 工作流
3. 点击右上角 "Run workflow"
4. 选择分支
5. 点击绿色 "Run workflow" 按钮

Q30: CI 运行多长时间?

超时设置:

Job	超时时间	实际耗时
dco	默认	~10 秒
config	1 分钟	~5 秒
build_and_upload	10 分钟	~3-5 分钟（每个 JDK）
publish_releases	10 分钟	~30 秒
publish_github_pages	10 分钟	~1 分钟

总计: 约 **10-15 分钟**（并行构建）

Q31: 如何调试 CI 失败?

方法 1：查看日志

1. 点击失败的 job
2. 展开每个 step
3. 查看红色错误信息

方法 2：本地复现

```
# 运行和 CI 相同的命令
mvn clean verify

# 使用特定 JDK 版本测试
export JAVA_HOME=/path/to/jdk-17
mvn clean verify
```

方法 3：添加调试输出

```
- name: Debug
  run: |
    echo "Current directory: $PWD"
    ls -la
    env | sort
```

Q32: `needs` 关键字的作用?

```
publish_releases:
  needs:
    - config
    - build_and_upload
```

声明依赖关系:

- `publish_releases` 必须等待 `config` 和 `build_and_upload` 完成
- 如果依赖失败，这个 job 会被跳过

访问输出:

```
${{ needs.config.outputs.datetime }}
${{ needs.build_and_upload.outputs.output_file_list }}
```

Q33: 为什么要用 `permissions`?

```
permissions:
  pages: write
  id-token: write
```

GitHub 安全机制:

- 默认 token 权限很低
- 需要显式声明需要的权限

权限类型：

- `pages: write`：允许部署到 GitHub Pages
- `id-token: write`：允许 OIDC 认证（Pages v4 需要）
- `contents: write`：允许创建 Release

Q34: 如果只想构建不发布怎么办？

推送到 feature 分支即可：

```
git checkout -b feature/test
git push origin feature/test
```

结果：

- ✅ 运行 DCO 检查
- ✅ 运行多版本构建测试
- ❌ 不发布 Release
- ❌ 不更新 GitHub Pages

Q35: 如何理解整个工作流？

用流程图理解：

```
[ 任何推送/PR ]
    ↓
[ DCO 检查 ] ——————————→ 失败 → ❌ 停止
    ↓ 成功
[ Config 计算条件 ] + [ 多版本构建 (17/21/25) ] ——→ 失败 → ❌ 停止
    ↓ 成功                        ↓ 成功 (JDK 21)
    ↓                      [ 打包 zip ]
    ↓                           ↓
    └————————→ [ 判断：是 main 或 tag 吗？ ]
                        ↓ Yes                  ↓ No
               [ 发布 Continuous ]          结束 ✅
                        ↓
               [ 判断：是 tag 吗？ ]
                        ↓ Yes                ↓ No
    [ 发布 Versioning ] + [ 更新 Pages ]     结束 ✅
                        ↓
                   结束 ✅
```

高级主题

Q36: 为什么 CI 中 checkout 的是别人的仓库?

```
repository: 'pzhlkj6612/forked-ruyisdk-eclipse-plugins'
ref: 'chore/clean-pomless-builds'
```

这是 PR #5 中的配置，用于测试!

合并后应该改为:

```
# 不指定 repository，使用当前仓库
# 不指定 ref，使用触发 CI 的分支
```

或直接删除这两行（使用默认值）。

Q37: 如何修改 CI 配置?

修改文件: `.github/workflows/ci.yml`

常见修改:

1. **添加新的 JDK 版本测试**:

   ```
   matrix:
     java_version: [17, 21, 25, 27]  # 添加 27
   ```

2. **修改发布条件**（如发布所有分支）:

   ```
   make_continuous_release: ${{ github.ref_type == 'branch' }}
   ```

3. **调整超时时间**:

   ```
   timeout-minutes: 20  # 从 10 改为 20
   ```

测试修改:

- 推送到 feature 分支测试
- 确认无误后合并到 main

Q38: 如何查看历史构建?

1. 访问: `https://github.com/your-repo/actions`
2. 点击具体的 workflow run
3. 查看每个 job 的日志
4. 可以下载 Artifacts

保留时间:

- Workflow 日志: **90 天**

- Artifacts：**30 天**（可配置）

Q39: Matrix 构建的好处是什么?

```yaml
strategy:
  matrix:
    java_version: [17, 21, 25]
```

优势：

1. **并行执行**：3 个版本同时构建（节省时间）
2. **全面测试**：确保代码在所有版本都能工作
3. **早期发现问题**：某个 JDK 版本的问题立即暴露

结果显示：

```
✅ Build and Upload (JDK 17)
✅ Build and Upload (JDK 21)
❌ Build and Upload (JDK 25) ← 一眼看出是 JDK 25 的问题
```

Q40: 如何优化 CI 速度?

当前已有的优化：

1. **Maven 缓存**：

   ```yaml
   cache: 'maven'  # 缓存依赖，不用每次下载
   ```

2. **并发控制**：

   ```yaml
   cancel-in-progress: false  # 不取消，避免重复构建
   ```

3. **条件执行**：

   ```yaml
   if: ${{ condition }}  # 不必要的 job 不运行
   ```

可能的改进：

- 使用自托管 runner（更快的网络）
- 缓存 Tycho 本地仓库
- 只在 main/tag 时做多版本测试

总结

Q41: 这个 CI 的核心价值是什么?

1. ✅ **质量保证**：多 JDK 版本测试
2. ✅ **自动化**：推送即构建、发布
3. ✅ **规范化**：强制 DCO 签名
4. ✅ **用户友好**：提供在线和离线安装
5. ✅ **可追溯**：完整的构建历史和 Release 版本

Q42: 我该如何使用这个 CI?

作为开发者:

```
# 1. 开发功能
git checkout -b feature/new-feature

# 2. 提交时签名
git commit -s -m "feat: add xxx"

# 3. 推送
git push origin feature/new-feature
```

CI 会自动:

- ✅ 检查签名
- ✅ 在 3 个 JDK 版本测试
- ✅ 告诉您是否有问题

作为维护者（发布新版本）:

```
# 1. 合并到 main
git checkout main
git merge feature/new-feature
git push origin main
    → 触发 Continuous Release

# 2. 准备好正式发布时
git tag v0.0.5
git push --tags
    → 触发 Versioning Release + GitHub Pages 更新
```

参考资料

- GitHub Actions 官方文档
- Tycho 官方文档
- Eclipse P2 仓库格式
- DCO 说明

文档版本：基于 ci.yml（PR #5）
最后更新：2025-11-08