代码托管

git.nju.edu.cn

- 已开启的高级功能
- Git镜像教程
- CI/CD
  - CI/CD 使用指南
  - 自动化构建 Docker镜像实践



Gravatar

Reply by email

Advanced Search

Container Registry

Shared Runners

- Linux Docker

Git镜像教程
NJU Git可以设置从其他代码托管平台如 GitHub、GitLab、Gitee等拉取、推送代码仓库（镜像）。教程见： https://mp.weixin.qq.com/s/EjAedt6A3PvuASGlFCfyWQ

CI/CD

□□□□ CI/CD□□□ git.nju.edu.cn□□□□□□□□□□□□□□

CI/CD

CI/CD 使用指南

先选方式

需求	方式	runner tag
编译 / 构建 / 打包 / 测试 / 部署等日常作业	常规 job	不用写（公共 runner）
只构建并推送镜像	kaniko	不用写（公共 runner）
`docker run` / `compose` 等 Docker in Docker	DinD	`dind`（公共 runner）

`tags` 怎么填：

- 用公共（共享）runner：不写 `tags`（常规 job、kaniko 就是这种；也可显式写 `tags: [docker]`）。
- 用公共 DinD：必须写 `tags: [dind]`——只有带这个 tag 的 runner 提供 job 内 Docker。
- 用你自己的 runner（自建本地 runner）：写你注册时设的 tag，例如 `tags: [myrunner]`。
- 注意：项目里有自己的 runner 且开了 "Run untagged jobs" 时，不写 tags 的 job 会优先被它接走（通常正是你想要的）。

常规 job

> 编译、构建产物、打包、测试、代码检查、部署……都属于「常规 job」：在 job 容器里跑命令、不构建镜像。它们的差别只是 stage 和 script 里跑什么；套路是一个 job = 一个镜像 + 一串命令 + （可选）产物。

- 特点：最快、最省资源、隔离最简单（没有 Docker daemon）。
- 不能 `docker build`；要产出镜像请看下面 kaniko / DinD 两节。
- 任意公共镜像都可用（Docker Hub / gcr.io / ghcr.io / quay.io / nvcr.io / registry.gitlab.com / registry.k8s.io）。

编译 / 构建产物

```
build:
  stage: build
  image: node:lts
  script:
    - npm ci
    - npm run build
  artifacts:
    paths: [dist/]
```

□□

```
test:
  stage: test
  image: node:lts
  needs: [build]
  script:
    - npm ci
    - npm test -- --reporter=junit --outputFile=report.xml
  artifacts:
    when: always
    reports:
      junit: report.xml
```

□□□ / □□□

```
lint:
  stage: test
  image: node:lts
  script:
    - npm ci
    - npx eslint .
```

□□□

```
package:
  stage: package
```

```
  image: alpine:latest
  needs: [build]
  script:
    - tar -czf myapp.tar.gz -C dist .
  artifacts:
    paths: [myapp.tar.gz]
```

> □□ `package` □□□□□ stage □□□□□□□□□□ `stages: [build, test, package, deploy]` □ GitLab □□□□ `.pre / build / test / deploy / .post` □□

□□□ deployment job□

□□□□□□□ `rsync` □ `scp` □ `sftp` □ WebDAV□□□□□□ `kubectl` □□□□□□□ ..... □□□□□□□□□□□□□□□□□□□□□□□□ `script` □□□□□□□□□□□

> □□□□□□□□□□□□□□□□ **Token□□□ GitLab □ CI/CD** □□□□□□□□□□□ **.gitlab-ci.yml** □□□□□□□

```
deploy:
  stage: deploy
  image: alpine:latest
  variables:
    GIT_STRATEGY: none
  environment:
    name: production
    url: https://myapp.example.com
  resource_group: production
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
      when: manual
  before_script:
    - apk add --no-cache openssh-client rsync
    - mkdir -p ~/.ssh && chmod 700 ~/.ssh
    - install -m 600 "$SSH_PRIVATE_KEY" ~/.ssh/id_ed25519
    - install -m 644 "$SSH_KNOWN_HOSTS" ~/.ssh/known_hosts
  script:
```

```
    - rsync -az --delete -e "ssh -i ~/.ssh/id_ed25519" dist/ deploy@my-server:/srv/myapp/
    - ssh -i ~/.ssh/id_ed25519 deploy@my-server 'cd /srv/myapp && docker compose up -d'
```

敏感信息存哪里（ Settings → CI/CD → Variables）：

变量	类型	说明
`SSH_PRIVATE_KEY`	File	部署专用私钥（单独生成，不要用个人密钥）。 File 类型下 `$SSH_PRIVATE_KEY` 是临时文件路径，用 `install -m 600 "$SSH_PRIVATE_KEY" ~/.ssh/id_ed25519` 复制，私钥内容不会出现在日志里
`SSH_KNOWN_HOSTS`	Variable 或 File	目标机公钥指纹（ `ssh-keyscan my-server` ）；不要用 `StrictHostKeyChecking=no`
密码 / Token	Variable	勾 Masked；生产再勾 Protected（只在受保护分支的 job 里可见）

要点：

- 私钥只存在于 job 的临时容器里（权限 600），job 结束随容器销毁；仓库里永远没有它。
- 目标机不要开密码登录，使用最小权限的专用部署账号。
- 生产发布建议限定 protected branch + `when: manual` + `environment` + `resource_group`。
- 拉私有仓库代码用 deploy token / deploy key，不要用个人 PAT。

---

kaniko

在 job 里构建镜像并推送到 registry，不需要 Docker daemon。

- 只能构建并推送镜像，不能 `docker run` / `compose`；不支持 `RUN --mount=type=secret` ；不支持跨架构构建。
- 不需要特权、不需要 DinD、任意 runner 都能跑。
- 缓存: `--cache-repo` 指向本项目 registry。
- 关于 tag: `debug` 是浮动 tag（自动跟随上游最新版），不要用 `latest`（那是 scratch 镜像，没有 shell， CI 脚本根本跑不起来）。
- 上游 `GoogleContainerTools/kaniko` 已归档（ 2025-06，冻结在 v1.24.0）； `martizih/kaniko` 属于仍活跃的替代分支 `osscontainertools/kaniko`。

```
build-image:
  stage: build
  image:
    name: martizih/kaniko:debug        # 浮动 tag，始终带 shell；等价 ghcr.io/osscontainertools/kaniko:debug
```

```
    entrypoint: [""]
  script:
    - mkdir -p /kaniko/.docker
    - |
      cat > /kaniko/.docker/config.json <<CONF

{"auths":{"${CI_REGISTRY}":{"username":"${CI_REGISTRY_USER}","password":"${CI_REGISTRY_PASSWOR
D}"}}}
      CONF
    - /kaniko/executor
      --context "${CI_PROJECT_DIR}"
      --dockerfile "${CI_PROJECT_DIR}/Dockerfile"
      --destination "${CI_REGISTRY_IMAGE}:${CI_COMMIT_SHORT_SHA}"
      --destination "${CI_REGISTRY_IMAGE}:latest"
      --snapshot-mode=redo
      --compressed-caching=false
      --cache=true
      --cache-repo="${CI_REGISTRY_IMAGE}/cache"
```

DinD

□ job □□□□ Docker□ build / run / compose□

- □□ `privileged`□ runner □□□□□□□□□□□□□□□ user-namespace □□□□
- □□ `docker build` / `run` / `compose` □□□□□□□ `docker buildx` □□□□□□□ `binfmt_misc` □□□□□□□□ multi-arch / QEMU □□□□□□□□□□□□□□□□□□□□□□□□□□□□
- □□ daemon □□ job □□□□□□□□□□□ job □□□□□□□□ registry □□ /artifacts □□□□□ docker □□□□□

```
build-and-test:
  stage: test
  tags: [dind]                    # ← □□□□□□□□□□□□ Docker daemon □ runner
  image: docker:29-cli
  services:
    - name: docker:dind
      alias: docker
  variables:
    DOCKER_HOST: tcp://docker:2375
```

```
    DOCKER_TLS_CERTDIR: ""          # 明文 2375，仅 job 内部网络
  script:
    - docker build -t myapp:ci .
    - docker run --rm myapp:ci ./run-tests.sh
    - docker compose up -d --build && docker compose ps
```

自建本地 runner

什么时候用：不方便用共享 runner（需要特殊硬件、本地数据、自定义镜像环境），或想把构建放在自己的机器上。

在 GitLab 上创建 runner

- 位置二选一：项目 → Settings → CI/CD → Runners → New project runner。
- 勾选 Tags，例如 `myrunner` —— 这是 job 匹配 runner 的唯一依据；不建议勾 "Run untagged jobs"（会跟共享 runner 抢任务）。
- 创建后复制 runner authentication token（`glrt-...`），只显示一次，等同凭据，不要提交进仓库）。

在目标机器上安装注册 runner

安装与注册方式很多（Docker、deb/rpm 包、二进制、Kubernetes 等），请按你的环境选择，参考 GitLab 官方文档：

- 安装：https://docs.gitlab.com/runner/install/
- 注册：https://docs.gitlab.com/runner/register/

注册时 `--url` 填 `https://git.nju.edu.cn/`，`--token` 填上一步的 `glrt-...`，`--executor` 按需。

使用与验证

- 只有写了 `tags: [myrunner]` 的 job 才会派到你的 runner。
- GitLab → Runners 页面应显示 online；跑一条最小 job 验证即可。

命名与凭据
推送目标统一用 ${CI_REGISTRY_IMAGE}（ = reg.nju.edu.cn/<namespace>/<project>
），不要硬编码仓库地址。
拉取本项目 /组内私有镜像用 $CI_JOB_TOKEN；外部 registry 凭据放 Masked + Protected 的 CI
变量。
凭据不要写进 .gitlab-ci.yml，也不要 --build-arg SECRET=... 烧进镜像层（ dind 可用 RUN -- mount=type=secret， kaniko 不支持）。
Docker Hub 上游图片可走 Dependency Proxy（需要改镜像名）。

常见问题
忘了写 tags: [dind] → job 会被派到没有 Docker daemon 的 runner。
忘了写 services: [docker:dind] → 连不上 tcp://docker:2375。
用 martizih/kaniko:latest → 那是 scratch 镜像，没有 shell，CI 脚本根本跑不起来；请用 martizih/kaniko:debug。
把凭据写进 .gitlab-ci.yml，或用 --build-arg SECRET=... 烧进镜像层 →
凭据会永久留在仓库或镜像历史里；请一律使用 CI/CD 变量（ DinD 里可用 RUN -- mount=type=secret， kaniko 不支持这种写法）。

CI/CD

自动化构建 Docker镜像实践

> 一个通过 CI/CD自动构建 Docker镜像并用于 HPC集群计算的简单示例。

示例代码仓库: https://git.nju.edu.cn/escience/singularity-example

容器化在集群上使用时有很大的优势，主要体现在:

- 容器内模拟的 root权限，无需权限即可任意安装自己想要的程序包
- 版本可控，自定义性强
- 可复用性强，一处构建，多处部署

因此，对于一些编译特别复杂但早已有 Docker镜像的计算软件，或者自己写的配置环境麻烦的软件，考虑 Singularity是个不错的选择。

此外， eScience中心的多个服务也涵盖了 一整套工作流 。所以全程无需出校访问 在校园内网 即可实现。

本文将通过一个简单但常见的 Conda虚拟环境下 Python Numpy库的镜像构建来演示如何使用这一功能。

通过 Dockerfile 构建自己的镜像

通过 Dockerfile，可以事先构建一个 Docker镜像。

```
FROM continuumio/miniconda3:22.11.1

# 使用南大镜像站conda源
COPY .condarc /root/.condarc

# 创建环境
RUN conda create -n my-env python=3.10 numpy

# 激活环境
SHELL ["/bin/bash", "--login", "-c"]
RUN conda init bash
RUN echo "source activate my-env" > ~/.bashrc
```

```
ENV PATH /opt/conda/envs/my-env/bin:$PATH
```

此处 `.condarc` 是修改为了南京大学镜像站的软件源以保证速度。如果要安装其他的包，把 `numpy` 改成你要的包即可，同时还可以控制 Python版本。

如若本地有 Docker，可通过 `docker` 本地尝试构建：

```
docker build -t escience/conda-numpy .
```

> Docker也可以使用 Docker缓存 `docker.nju.edu.cn` 。

构建完毕后，本地跑一下相关的脚本以测试：

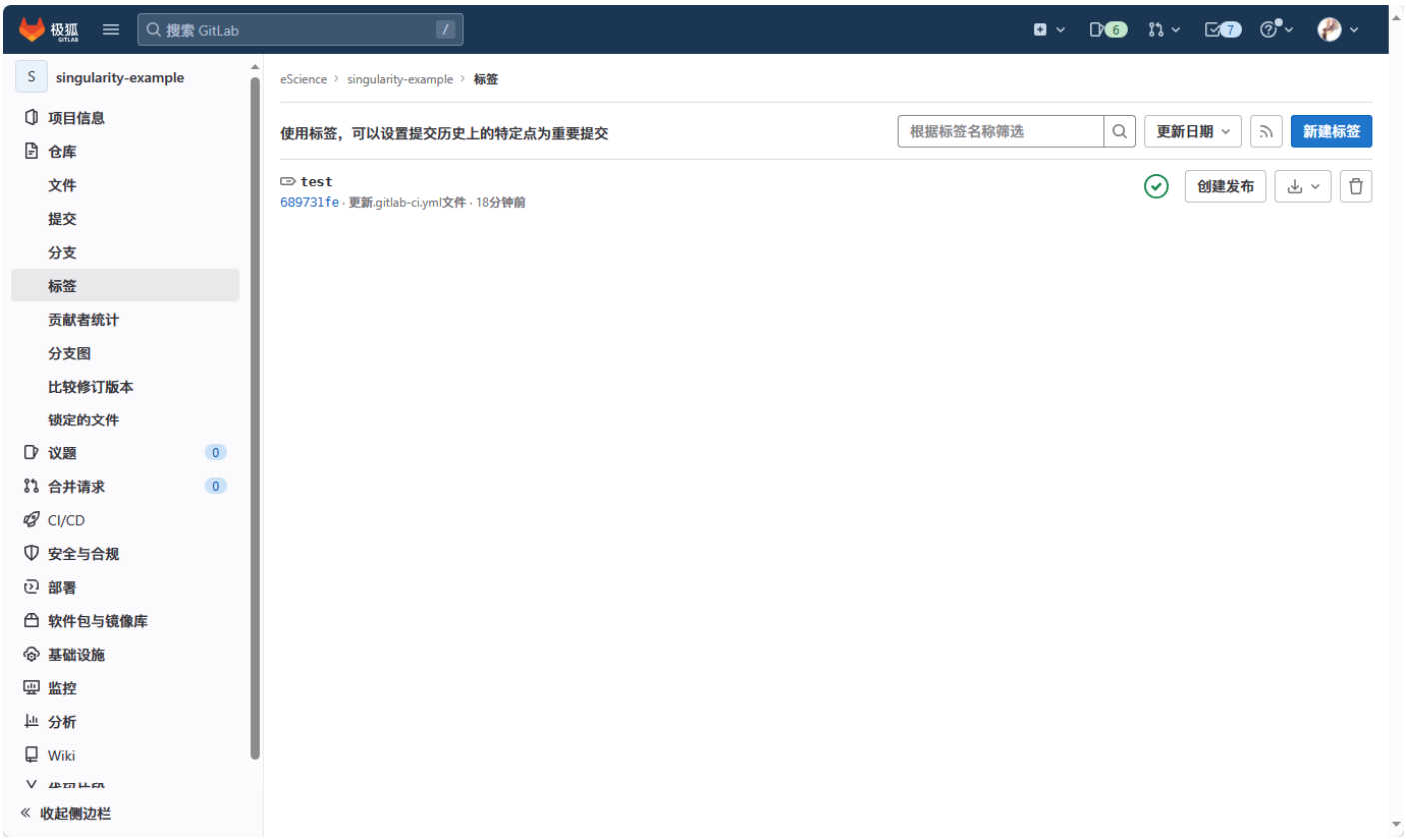
```
docker run -i escience/conda-numpy python < test.py
```

设置 CI/CD

CI/CD是代码托管服务 `git.nju.edu.cn` 的一个自动化工具，在代码仓库下的 `.gitlab-ci.yml` 文件中配置 CI/CD，可以让服务器按照设定执行自动构建、编译、测试、集成、部署等等。

CI/CD需要一个基础镜像来运行，这个镜像 `gcr.nju.edu.cn` 也有缓存服务。文件内容如下：

```
# 执行顺序
stages:
  - build
  - test
# 自动构建镜像
build:
  stage: build
  image:
    name: martizih/kaniko:debug
    entrypoint: [""]
  script:
    - /kaniko/executor
      --context "${CI_PROJECT_DIR}"
      --dockerfile "${CI_PROJECT_DIR}/Dockerfile"
      --destination "${CI_REGISTRY_IMAGE}:${CI_COMMIT_TAG}"
  rules: # 只在推送标签时触发
```





```
    - if: $CI_COMMIT_TAG
# 使用文件来测试
test:
  stage: test
  image:
    name: ${CI_REGISTRY_IMAGE}:${CI_COMMIT_TAG}
  script:
    - python "${CI_PROJECT_DIR}/test.py"
    rules:
    - if: $CI_COMMIT_TAG
```

推送代码到仓库后，为某个提交创建标签，例如此处我们创建为 `test`，这也会成为镜像的标签。

触发流水线任务后，如果通过，在 “CI/CD”-“流水线 ” 可以看到作业情况：

在自动构建成功后，即可在 “软件包与镜像库 ”-“容器镜像库 ” 中查看到此镜像。

> 当然，这里你也不一定必须用 Kaniko 的构建方案，此处是为了实现自动化。你也可以本地构建完毕后通过 `docker push` 命令直接推送至 `reg.nju.edu.cn` 来托管。此外，Singularity有其自己的定义文件（类似 Dockerfile），可以用类似的方式一步到位而不需要单独构建一个 Docker镜像。

在集群上使用

`hpc.nju.edu.cn` 唐楼集群的简单基本使用见《校内用户超算集群申请与基本使用简明指南》一文。

登录集群通过此镜像来构建 Singularity镜像:

```
singularity build conda-numpy.sif docker://reg.nju.edu.cn/escience/singularity-example:test
```

```
INFO:    Starting build...
Getting image source signatures
Copying blob 3389e9eb8624 done
Copying blob 3f4ca61aafcd done
Copying blob baee49be4542 done
Copying blob 7b4354700ca4 done
Copying blob 69a5d9e1ecd6 done
Copying blob 8a7a15cee421 done
Copying blob 5f4e24b7e321 done
Copying config 51871566f8 done
Writing manifest to image destination
Storing signatures
2023/08/29 21:46:55  info unpack layer: sha256:3f4ca61aafcd4fc07267a105067db35c0f0ac630e1970f3cd0c7bf552780e985
2023/08/29 21:46:57  info unpack layer: sha256:69a5d9e1ecd6566da53d0978004bdf37dddfaba1d8a6117966f397b41cbbc529
2023/08/29 21:46:59  info unpack layer: sha256:7b4354700ca480732ead22a553cc45916dc5466709ca64d964c4647b5b9343e9
2023/08/29 21:47:02  info unpack layer: sha256:baee49be454261f20f9770566da694b7e7845cf7d279cc2421c6b3eed68c012c
2023/08/29 21:47:02  info unpack layer: sha256:8a7a15cee4219b244df17b401664119dfd5b7e52a5659107f1ee3ec210722373
2023/08/29 21:47:20  info unpack layer: sha256:3389e9eb8624f21330c272ba23defc59ba31d46273e38cde475e7256edea80cb
2023/08/29 21:47:20  info unpack layer: sha256:5f4e24b7e32113995e0417eb5c4de5cbef9be70a4a3841a19cdb2fa1f3f12a34
INFO:    Creating SIF file...
INFO:    Build complete: conda-numpy.sif
```

虽然镜像较大，但是可以看见速度仍然很快，这正是开头 全程校园内网 带来的好处。

由于示例是一个 `numpy` 的环境，后续我们对应地使用作业脚本即可。创建一个 `job.lsf` 文件：

```
#BSUB -q 6140ib
#BSUB -n 1

module load singularity/latest

SINGULARITY="singularity run --env MKL_NUM_THREADS=$LSB_DJOB_NUMPROC conda-numpy.sif"
${SINGULARITY} python test.py
```

然后执行

```
bsub < job.lsf
```

通过 `bjobs` 查看任务运行情况，完成后通过 `bpeek`
指令可以查看输出，可以看到，任务成功提交并正确执行了。

总结

通过 CI/CD自动构建 Docker镜像并用于 HPC集群计算的简单示例，我们尝试将 eScience
的下列服务进行了结合：

- 缓存 /镜像加速: `docker.nju.edu.cn`，`gcr.nju.edu.cn`，`mirror.nju.edu.cn`
- CI/CD自动构建: `git.nju.edu.cn`，`reg.nju.edu.cn`
- 计算集群: `hpc.nju.edu.cn`

在这个过程中，这些服务的灵活使用有助于从版本控制、环境搭建、迁移部署等诸多方面建立可靠的工作流程
，最终得以节省时间、便捷开发，提升创新协同的质量。