熟悉 GitHub

1. 熟悉GitHub页面

GitHub （github.com）是最大的 Git 版本库托管平台，包括 OpenMMLab 在内的许多开源项目使用 GitHub 网站实现代码托管、版本管理、问题追踪、社区交流等功能。即使你还不熟悉 Git 的使用，也很可能已经使用过 GitHub 来查找所需的开源软件或资料。我们首先简单熟悉一下 GitHub 的页面和基本功能。

以 MMPose （github.com/open-mmlab/mmpose）为例，一个软件项目在 GitHub 上的首页通常如下图所示：

图1 软件项目首页（Code 页面）

在页面最上方的区域展示了该软件项目的名称和基本信息，如 Watch（关注）、Star（点赞）和 Fork（使用或开发）的人数。基本信息下方是一排功能标签，包括 Code、Issues、Pull Requests、Actions、Insights 等，下面会依次介绍。

图2 基本信息和功能标签

1.1 Code页面

默认的 Code 页面，内容即为图1所示，主要包括：

- 文件结构和代码内容：该区域显示了项目中所有的文件，可以在此浏览文件结构，或点进文件浏览其内容
- 其他信息：通常包括使用许可、发布版本、贡献者等信息
- README：如果当前浏览的路径下有 README.md 文件，其内容会显示在这一区域。通常软件项目会在根目录下包含 README.md 文件作为项目介绍。

1.2 Issues 页面

Issues 页面类似用于社区沟通的论坛。使用者可以在这里创建 issue 来提出遇到的问题，报告发现的 bug，或提出意见建议等；开发者可以在 issue 下进行回复，或将 issue 指定（assign）给特定的人来解决。此外，还可以用 issue 功能发布消息，如项目的 Roadmap、开发计划等，并将这些重要信息置顶。每个 issue 都会有一个编号，如 #900, 可以用于在其他 issue 或稍后将介绍的 Pull Request 中关联这个 issue。

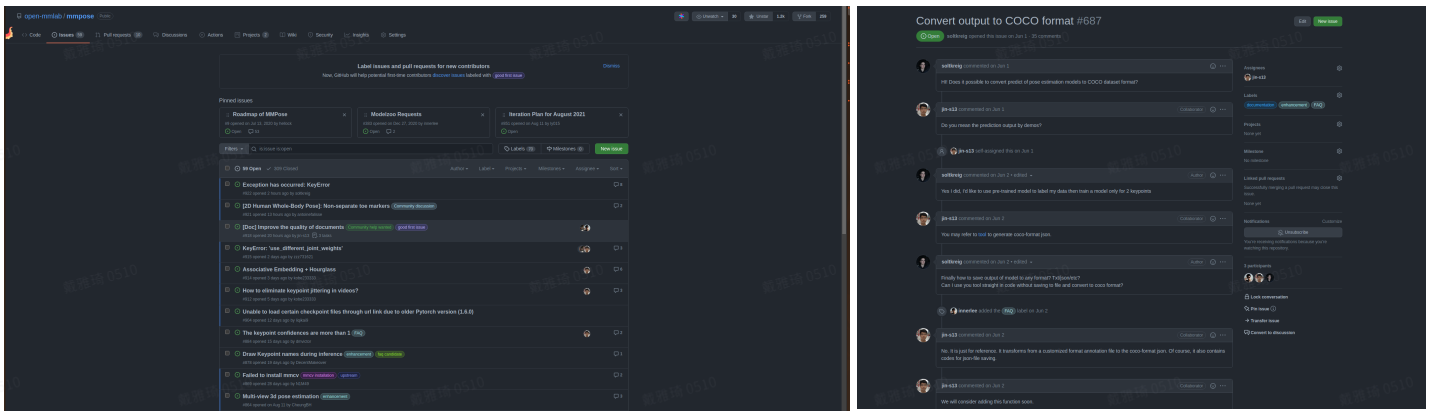


图3 Issues 页面 （左： issue 列表，右： 一个 issue 的内容）

1.3 Pull Requests 页面

Pull Requests 页面顾名思义用来浏览和管理 pull request（下简称 PR），如图4所示。通常，在 Github 上多人共同开发和维护一个项目时，会遵循一定的开发流程，这部分将在3.1中详细介绍。在此只简要说明开发流程，以引入 PR 的概念：

1. 开发者从官方代码仓库（如 open-mmlab/mmpose）fork 一份副本到自己的帐号 （如 zhang3/mmpose），并 clone 到本地；
2. 开发者在自己的代码仓库进行某项功能的开发；
3. 开发者将自己的修改推到自己的远程仓库，并向官方仓库发出申请，要求官方仓库拉取（pull）此次修改，即将此次修改加入到官方代码中。这样的请求就叫做 PR。（与之相对的还有 GitLab 平台所采用的 merge request，拓展阅读：https://stackoverflow.com/questions/22199432/pull-request-vs-merge-request）
4. 代码库的维护者或其他开发者会对 PR 进行 Review，并与作者共同进行讨论和修改。最终将修改完成的 PR 合入到官方代码仓库中。至此，一个开发项的开发周期完成。

在 PR 列表中点进某个 PR 后，可以看到其内容包括 PR 的描述信息、作者提交代码的历史、Reviewer 的意见以及和作者的往来沟通等，这部分在 3.1 中也会详细介绍。与 issue 类似，每个 PR 也会有一个编号，用来在别处引用或关联该 PR。

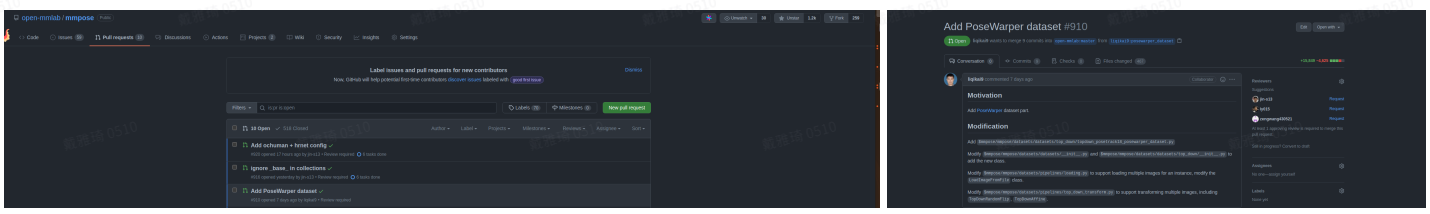


图4 Pull Requests 页面 （左： PR 列表， 右： 一个 PR 的内容）

1.4 Actions 页面

Actions 指 GitHub Actions，是 GitHub 提供的简化和方便开发流程的功能，用来在开发周期中自动触发执行特定的操作。如在 PR 被提交时，自动运行 CI； 在发布新版本时，自动编译并更新 pypi 上托管的安装包等。在 Actions 页面中，可以看到最近运行过的 action。点开其中一个action，可以看到详细信息、执行的具体操作步骤和输出的 log 等。如图5所示。

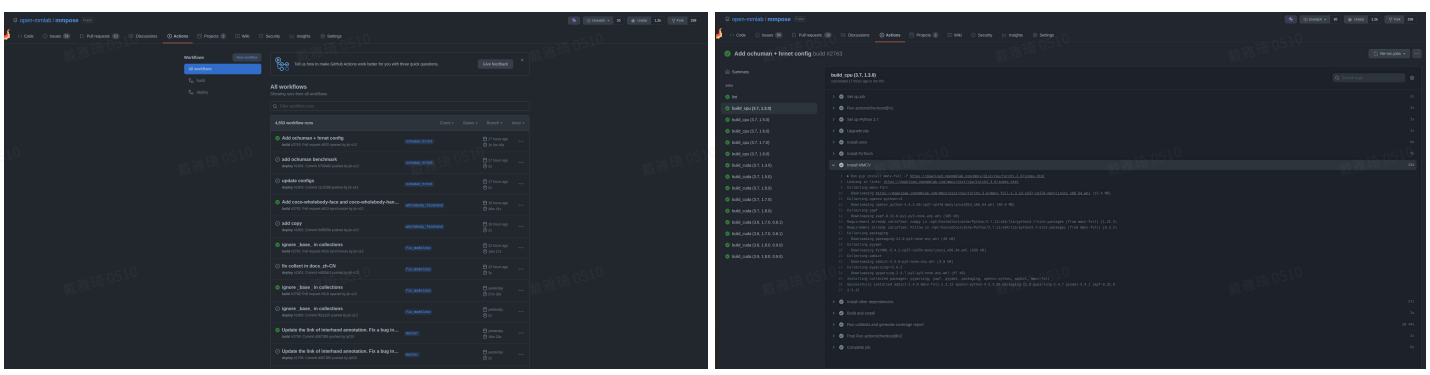


图5 Actions 页面 （左：action 列表，右：一个运行 CI 的action的详细信息）

1.5 Insights 页面

Insights 页面展示了项目的汇总信息和统计数据，以方便开发者快速了解项目的整体开发情况、社区活跃度和关注点的等。通常我们较关注的项目有：

- Pulse：项目的总览，包括最近的贡献者，PR，issue 等信息
- Contributors：代码贡献者信息
- Traffic： 项目流量信息，包括近期的访问量、clone 次数、访问者的来源以及访问量最高的项目内容等

2. 学习Git

GitHub 是基于 Git 的版本库托管网站。那么 Git 究竟是什么，又如何使用呢？ 在这部分中，我们将介绍这一强大工具的基本概念和用法。

2.1 Git 是什么

简单地说，Git 是一个版本控制系统，用来记录、管理和恢复对源代码（或任意类应的文件）所做的各种修改。与其他版本管理工具相比，Git 具有以下特点：

- 快速：Git 的大部分操作都在本地进行，而不需要访问远程数据，因而非常快速。
- 安全：Git 中所有数据记录在存储前都会计算校验和，并用校验和来引用。并且，Git 通常的操作只会向数据库中添加数据，而不会删除数据。因此，使用 Git 可以完整地记录版本信息，避免传输过程中的信息缺失或文件损坏，并且通常不会执行不可逆的操作。

- 完全分布式：即各个客户端都保留完整的代码仓库历史记录，并可以指定和若干不同的远端代码仓库进行交互（如 GitHub 上的多个远程仓库）。
- 强力支持非线性开发： Git 中的分支（branch）管理功能，可以支持大量开发者并行开发同一个软件项目。

2.2 Git 中的一些基本概念

2.2.1 工作区、暂存区和本地仓库

- 工作区（Working Directory）

当我们在本地创建一个 Git 项目，或者从 GitHub 上 clone 代码到本地后，项目所在的这个目录就是“工作区”。顾名思义，这里就是我们对项目文件进行编辑和使用的地方。如图6所示：

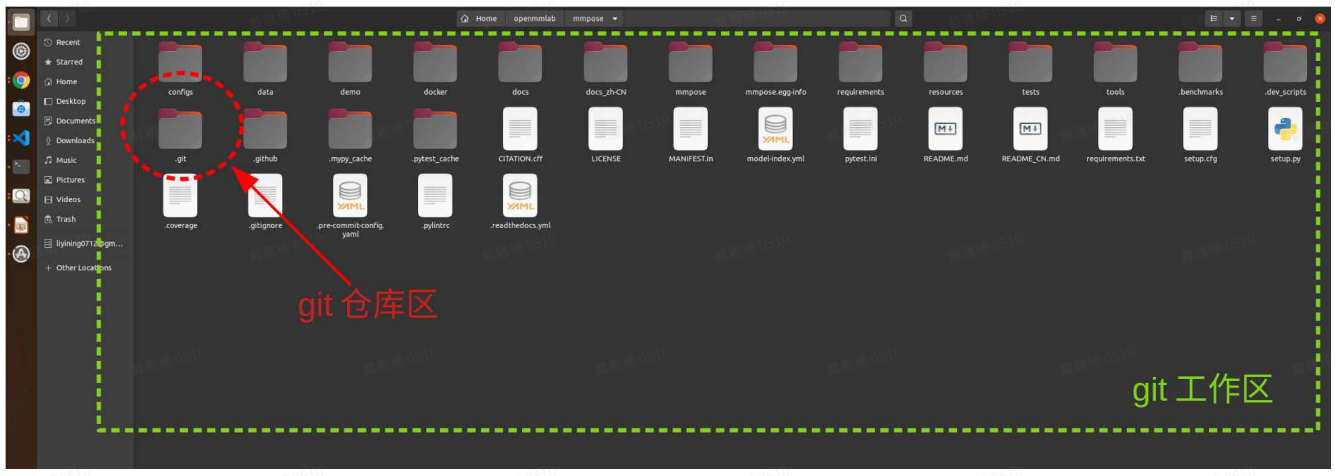


图6 MMPose 工作区和仓库区

- 仓库区 / 本地仓库（Repository）

在项目目录中，会发现一个 `.git` 隐藏目录，它不是软件代码的一部分，也不属于工作区，而是 Git 的版本仓库。简单来说，这个仓库区才是一个 Git 项目的“本体”，其中包含了所有历史版本的完整信息。而工作区正是对某个特定版本提取出来的内容。我们通常所说的仓库区，除了指 `.git` 目录这个物理位置，也指一种逻辑状态，即某个修改“记录进 Git 版本仓库”。

- 暂存区（Stage/Index）

暂存区是 Git 中独有的一个概念，它其实是位于 `.git` 目录中的一个索引文件，记录了下一次提交（commit）时将要存入仓库区的文件列表信息。当我们在工作区进行了一些修改后，并不直接将其提交进版本仓库，而是先通过 `git add` 指令将改动放入暂存区（即记录将要提交的文件索引）。在若干次 `git add` 后，再通过 `git commit` 指令将暂存区的修改提交到仓库区。这样的设计，让使用者可以更灵活地选择每次提交的内容。

这里涉及到的 git 指令，将在2.4部分详细介绍。

2.2.2 文件状态

结合2.2.1中工作区、暂存区和仓库区的概念，我们接下来介绍 Git 中的3种文件状态：

- 已跟踪（tracked）：通常，Git 会自动跟踪工作区中文件，发现哪些文件经过了编辑。当新增文件时，可以使用 `git add` 指令将文件设置为被跟踪的状态。
  - 对于一些不希望 Git 自动跟踪的文件（如本地的数据文件或临时文件等），可将其路径或匹配符加入 `.gitignore` 文件中，Git 便不会自动跟踪这些文件。可以参考 3.2.1 中的内容。
- 已修改（modified）：表示该工作区中的文件被修改，但还没有添加到暂存区。这里的修改包括内容编辑、重命名、移动位置、删除文件等。

- 已暂存（staged）：表示该文件的修改已经提交到暂存区。当使用`git commit`提交修改后，随着工作区版本更新，这些文件的状态也会变回“已跟踪”。

图7概括了文件状态的变化周期。

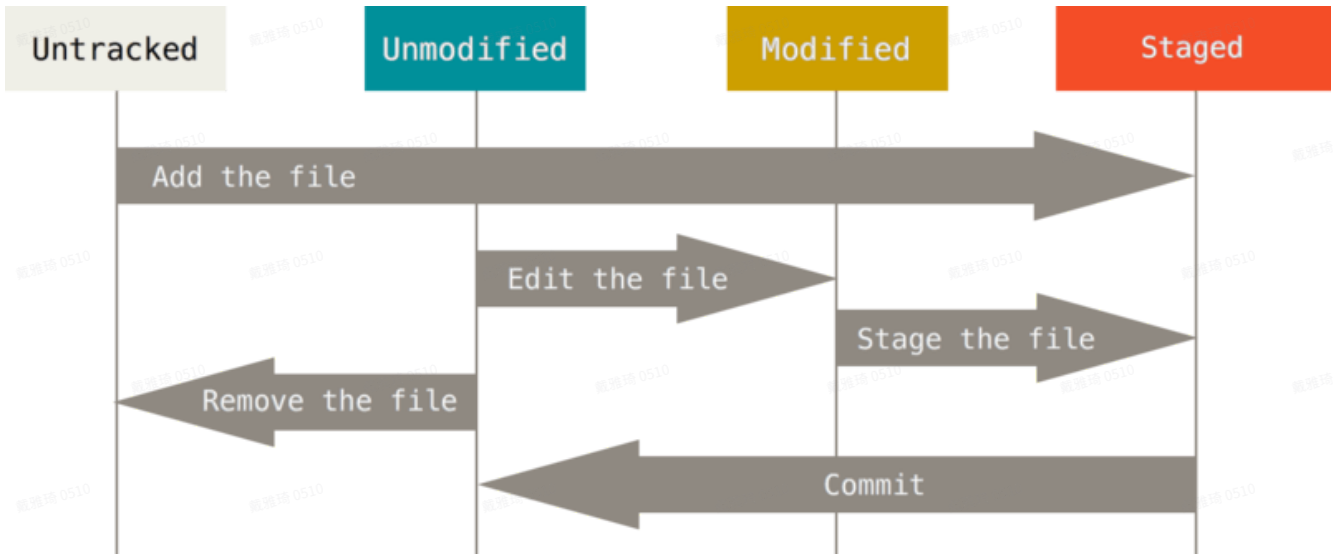


图7 Git 文件状态变化周期 （图片来源：Git - Git 是什么?）

2.2.3 分支

分支（branch）是 Git 用来支持并行开发的机制。一个分支可以简单理解为一系列提交组成的版本历史。通常一个项目会有一个主分支 master（或main）。开发者从 master 分支创建新的分支，进行功能的开发，并在开发完成后将该分支合并回 master 分支。通过这种方式，开发者们可以在多个分支上并行开发，而不会互相干扰。Git 因其独特的数据保存机制和分支模型（详见：Git - 分支简介）可以实现非常快速的分支创建和切换，因而也鼓励在开发流程中频繁使用分支与合并。

关于分支的新建、合并与解决冲突

新建和合并分支是基于 Git 的开发流程中很常用的操作。例如，当我们要在项目中开发一个功能时，通常会从当前的 master 分支新建一个分支用于开发（关于这里用到的 Git 指令，可以在 2.3 部分中介绍）：

Bash

```
# 从远程仓库 origin 拉取最新的 master
$ git fetch origin
# 从 master 分支新建 dev 分支用于开发
$ git checkout master
$ git checkout -b dev
```

在开发过程中，所有的修改都会被提交到 dev 分支。当开发完成后，我们会将 dev 分支合并进 master 分支：

Bash

```
# 切换回 master 分支
$ git checkout master

# 将 dev 分支合并到当前（master）分支
$ git merge dev
```

很多时候，在合并分支时会遇到“冲突”，即待合并的两个分支中对同一文件的同一处做了不同修改，使得 Git 无法自动合并这些修改。这时，Git 会暂停分支合并，将包含冲突的文件标识为未合并

（unmerged）状态，等待手动解决冲突。此时可以用 `git status` 指令查看存在冲突的文件：

Bash

```
# 例如，在 run.py 文件中存在冲突
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

```

3. Unmerged paths:
4. (use "git add <file>..." to mark resolution)

Bash

```

    both modified:      run.py

no changes added to commit (use "git add" and/or "git commit -a")
```

Git 会在存在冲突的文件中对应位置加入标准的冲突解决标记，方便手动解决冲突，看起来如下：

Python

```
<<<<<<< HEAD:run.py
print('OpenMMLab is an open-source project')
=======
print('OpenMMLab is AWESOME!')
>>>>>>> dev:run.py
```

这里 `HEAD` 表示当前分支（即 master）所在位置，`=======` 上面的部分就是当前分支中的内容，下面的部分则是 dev 分支中的内容。我们可以手动编辑冲突的内容，并删除冲突解决标记 （含有 `<<<<<<<`，`>>>>>>>` 和 `=======` 的行）。例如将例子中的这部分代码改为：

Python

```
print('OpenMMLab is an awesome open-source project')
```

手动解决冲突后，需要将修改过的文件添加到暂存区，然后继续分支合并即可：

Bash

```
# 添加解决冲突时的修改
$ git add run.py
# 继续分支合并
$ git merge --continue
```

如果在分支合并过程中想要中断合并，并回到合并前的状态，可以使用 `git merge --abort` 指令。

成功合并分支后，会生成一个新的提交（Commit），并将当前分支的指针 HEAD 指向该提交。

2.3 Git 基础指令

在这部分，我们介绍一些常用的 Git 指令。Git 的指令通常有着强大而丰富的功能，可以灵活地设置众多参数。这里只给出每个指令最基础和常见的用法，更多用法可以在 Git 官方教程（Git - 设置与配置）中学习，或者在命令行中查看对应指令的文档，如：

Bash

```
# 查看 git checkout 指令文档
$ man git-checkout

# 查看 git checkout 指令参数说明
$ git status -h
```

2.3.1 Git 配置

- **git config**

Bash

```
# 设置用户信息
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com

# 设置 Git 自动记录密码（token），从而无需在每次 pull 或 push 时输入
# 注意：信息会以明文存储在本地，需考虑安全性
$ git config --global credential.helper store
```

上面指令中的 `--global` 指定了配置的层级。Git 中支持3个层级的配置，分别是 system（系统）、global（全局）和 local（本地），靠后的层级会覆盖掉靠前层级的配置。`git config` 指令会将配置写入特定的文件中，因而我们也可以通过手动编辑配置文件的方式对 Git 进行配置。在 Linux 系统中，这文件的路径通常是：

- system：/etc/gitconfig（需要 sudo 权限，不推荐）git
- global： ~/.gitconfig (或者 ~/.config/git/config)
- local：正在操作的仓库的 .git 路径下的配置文件, 即 [REPO_PATH]/.git/config

2.3.2 建立仓库

- **git init**

`git init` 指令用于在一个本地的路径下创建和初始化 Git 仓库。该指令会在项目目录下创建 `.git` 目录，但不会自动关联远程仓库（如 GitHub 上的远程仓库）。

Bash

```
# 创建本地代码仓库
$ cd ~/my_project/
$ git init
```

- **git clone**

`git clone` 指令用将远程代码仓库下载到本地，以创建本地仓库。这是实际开发中更常用的一种创建本地仓库的方式。

Bash

```
# 从远程仓库创建本地仓库
$ git clone https://github.com/open-mmlab/mmpose.git
```

`git clone`中使用的远程代码库 URL，可以在其 GitHub 首页看到（如图8），其他平台如 GitLab 也类似。

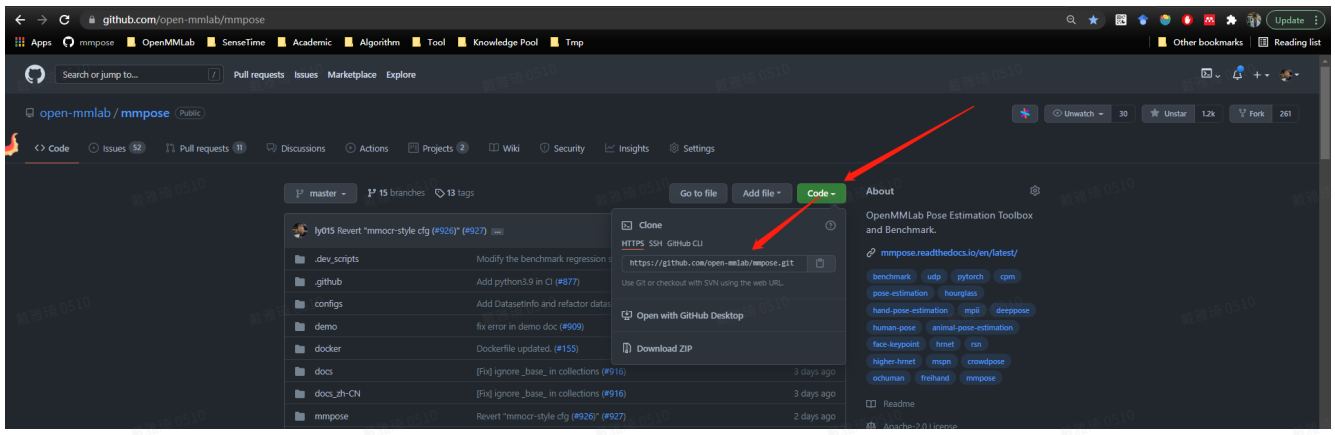


图8 GitHub 上远程仓库的URL

2.3.3 状态查询

- **git status**

`git status`指令用于查看当前本地仓库的状态，包括所在的分支、文件状态等。

Bash

```
# 查看本地仓库状态
$ git status

On branch v2.0_dataset
Your branch is up to date with 'origin/v2.0_dataset'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   README_CN.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        new_file.py
```

在上面的例子中，可以看到返回的信息包括：

- 本地处于 v2.0_dataset 分支
- 有一个新增文件`new_file.py`处于未被追踪的状态（关于文件状态，可回顾 2.2.2 文件状态）

- **git diff**

`git diff` 指令用来比较仓库中的文件在修改先后的差异。常见的用法有

Bash

```
# 查看工作区与暂存区的差异（基本用法）
$ git diff

# 查看暂存区与最近一次提交的差异
$ git diff --staged

# 比较两个提交（或分支）的差异
$ git diff <branch-a> <branch-b>
# 例如：
$ git diff master dev

# 比较分支B的最近提交与分支A、B最近的公共提交之间的差异
# 即查看分支B在分支A的基础上加入了哪些内容
$ git diff <branch-a>...<branch-b>
# 例如
$ git diff master...dev
```

2.3.4 文件操作

- **git add**

`git add` 指令用来将工作区的修改添加到暂存区。可参考本文 2.2 中关于工作区、暂存区和文件状态的介绍。

Bash

```
# 添加指定文件的修改到暂存区
$ git add <file-a> <file-b>

# 添加当前目录下的所有修改到暂存区
$ git add .
```

- **git commit**

`git commit` 指令用来将所有已添加到暂存区的修改生成一个提交对象，保存进仓库区，并将当前分支的指针（HEAD）移动到该次提交上。

Bash

```
# 提交暂存区修改
$ git commit

# 提交暂存区修改，并直接输入commit message
$ git commit -m "Some commit messages"

# 重做上一次提交，通常用于修改 commit message
$ git commit --amend

# 跳过git add，直接将工作区的修改提交
$ git commit -a
```

2.3.5 分支管理

· git branch

`git branch`指令用来进行分支管理操作，如列出有的分支、创建新分支、删除分支及重命名分支等。

Bash

```bash
# 列出所有的本地分支
$ git branch

# 列出所有的本地和远程分支
$ git branch -a

# 在当前的提交对象上创建一个新分支（但并不自动切换到新分支）
$ git branch <branch-name>

# 重命名分支,若未指定old-branch，则默认重命名当前分支
$ git branch -m [<old-branch>] <new-branch>

# 删除分支
$ git branch -D <branch-name>
```

· git checkout

`git checkout`指令主要用于切换分支。

Bash

```bash
# 切换到已有分支
$ git checkout <branch-name>

# 从当前提交创建并切换到新分支
$ git checkout -b <branch-name>
```

此外，`git checkout`指令还用于丢弃工作区中的修改（未提交到暂存区），或从其他提交对象（如其他分支，或特定commit等）中提取文件到当前工作区。

Bash

```bash
# 撤销指定文件的修改
$ git checkout -- <filename>

# 批量撤销(.py文件中的）修改
$ git checkout -- '*.py'

# 从指定分支中提取文件，添加到工作区
$ git checkout <branch-name> <filename>
```

由于`git checkout`的功能较多且分散，在较新版本的 Git 中引入了`git switch`指令，来代替`git checkout`的分支切换功能，可以参考 Git - git-switch Documentation.

· git merge

`git merge`指令用来合并分支。关于分支合并的说明，可以参考 2.2.3 部分。

```bash
# 合并指定分支到当前分支
$ git merge <branch-name>.

# 在合并过程中继续/中止
$ git merge --continue
$ git merge --abort
```

2.3.6 远程仓库管理与同步

- **git remote**

`git remote` 指令用来管理本地仓库与远程仓库的关联。

```bash
# 查看关联的远程仓库
$ git remote -v

origin  https://github.com/ly015/mmpose.git (fetch)
origin  https://github.com/ly015/mmpose.git (push)
upstream        https://github.com/open-mmlab/mmpose.git (fetch)
upstream        https://github.com/open-mmlab/mmpose.git (push)
```

在上面的例子中，可以看到已经关联的远程仓库有2个，命名分别是 origin（URL：https://github.com/ly015/mmpose.git）和 upstream（URL：https://github.com/open-mmlab/mmpose.git）。

`git remote` 除了用于查看关联仓库，还用于进行关联的管理：

```bash
# 添加关联的远程仓库
# git remote add <name> <url>
$ git remote add zhang3 https://github.com/zhang3/mmpose.git

# 重命名远程仓库
# git remote rename <old> <new>
$ git remote rename zhang3 li4

# 设置远程仓库URL
# git remote set-url <name> <url>
$ git remote set-url li4 https://github.com/li4/mmpose.git

# 删除关联远程仓库
# git remote remove <name>
$ git remote remove li4

```

- **git fetch**

`git fetch` 指令用于与一个远程的仓库交互，并且将远程仓库中有而本地仓库中没有的所有信息拉取下来，然后存储在本地仓库中。这个指令不会修改本地工作区、暂存区的内容，也不会修改本地分支的信息。例如，从远程仓库 origin 拉取的 master 分支会在本地存储为 **origin/master**, 而不会与原有的本地分支 **master** 相混淆。

Bash

```bash
# 从远程仓库（默认远程仓库为 origin）拉取信息
$ git fetch [<remote-name>]
```

git push

`git push` 指令用于将本地仓库的分支推送到远程仓库，即比较本地分支与远程仓库中对应关联的分支，并将差异同步到远程仓库。该指令需要有远程仓库的写权限，因此这通常是需要验证的。

Bash

```bash
# 推送当前分支到默认远程仓库
$ git push

# 推送本地分支到指定远程仓库的指定分支。若 <local-branch-name> 和 <remote-branch-name>
# 之一未指定，则默认两者相同；若两者均未指定，则默认将当前本地分支推送到同名远程分支
$ git push <remote-name> <local-branch-name>:<remote-branch-name>

# 在指定远程仓库创建与本地分支关联的远程分支，并推送。
# 通常用于将本地创建的分支第一次推送到远程仓库
$ git push --set-upstream <branch-name>
```

git pull

`git pull` 指令用于拉取远程分支到本地，即比较本地分支与远程仓库中对应关联的分支，并将差异合入本地分支。通常 `git pull` 指令可以看作是 `git fetch` + `git merge` 。如同一般的分支合并，如果远程关联分支和本地仓库中的提交历史存在冲突，则需要解决冲突后才能继续合并。

Bash

```bash
# 拉取当前本地分支关联的远程分支
$ git pull

# 拉取指定远程仓库的指定分支到当前本地分支
$ git pull <remote-name> <branch-name>
```

2.4 Git 进阶指令

git mv

`git mv` 指令用于重命名或移动文件、目录或链接。与 `mv` 不同的是， `git mv` 会自动将这一修改添加到暂存区。

Bash

```bash
# 移动文件或目录
$ git mv <source> <destination>

# 将文件或目录移动到指定路径下
$ git mv <source> ... <destination-directory>
```

git rm

`git rm` 指令用于从工作区或暂存区移除文件。`git rm` 会自动将这一修改添加到暂存区（即将该文件从已追踪的文件清单中移除），类似 `git mv`。如果我们只想把文件从暂存区移除（即让 Git 不再跟踪该文件），但仍保留在工作区，则可以使用 `--cached` 选项。

Bash

```
# 从工作区和暂存区移除文件
$ git rm <filename>
# 仅从暂存区移除文件
$ git rm --cached <filename>
```

git stash

`git stash` 指令用来临时保存一些还没有提交的工作，以便在分支上不需要提交未完成工作就可以清理工作目录。例如，我们正在dev分支编辑 `run.py` 文件，临时需要切换到master分支跑一个测试任务。在切换分支前，需要清理工作区和暂存区，但我们并不希望将未完成的修改提交到仓库，此时可以用 `git stash` 指令临时贮藏当前的工作，并在完成临时任务后回复这些工作：

Bash

```
# 临时贮藏 dev 分支当前的工作
$ git add run.py
$ git stash # or "git stash push"
# 此时，工作区和暂存区会被清理，恢复到最近一次提交的状态

# 切换到 master 分支，完成临时任务
$ git checkout master
$ python run.py

# 切换回 dev 分支，回复之前贮存的工作
$ git checkout dev
$ git stash pop
```

`git stash` 会将贮藏的修改存放在一个栈上，因此可以多次使用 `git stash` 贮藏修改。栈的状态可以用 `git stash list` 查看，例如。

Bash

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
```

`git stash` 的常见用法汇总如下

Bash

```
# 贮藏当前分支的修改
$ git stash [push]

# 显示贮藏栈情况
$ git stash list

# 应用一个的贮藏的工作，即将其恢复到工作区，但不删除栈中的内容
$ git stash apply  # 应用最近一个贮藏
$ git stash apply stash@{2}  # 应用指定的贮藏

# 从栈中弹出最近一个贮藏，并应用
$ git stash pop

```

- **git reset**

`git reset`指令用来执行撤销操作。例如，它可以将当前分支的 HEAD 指针移动到指定的提交上，从而实现撤销到该提交之后的所有提交。根据指令参数，这种撤销可以是只移动 HEAD 指针，而不改变工作区内容（被撤销的提交内容会保留在工作区，变成已修改，未提交的状态）；也可以直接修改工作区的内容。此外，`git reset`还可以用来撤销提交到暂存区的内容。一些例子如下：

Bash

```
# 基本用法
$ git reset [<commit>]  # 将 HEAD 指针移动到指定的提交（默认为最近一次提交），并重置暂存区
$ git reset [--mixed]  # 重置 HEAD 指针及暂存区
$ git reset --soft  # 仅重置 HEAD 指针
$ git reset --hard  # 重置 HEAD 指针、暂存区及工作区

# 撤销提交的例子
$ git reset HEAD~2  # 撤销当前分支最近的2次提交，但不改变工作区内容
$ git reset HEAD~2 --hard  # 撤销当前分支最近的2次提交，并将工作区内容回复到撤销后的状态
```

由于`git reset --hard`指令会直接修改工作区，有可能会导致数据丢失（Git 会真正销毁数据的仅有的几个操作之一），所以在这样使用前一定要确保自己知道其后果。

- **git cherry-pick**

`git cherry-pick`指令用来获得在单个提交中引入的变更，然后尝试将作为一c个新的提交引入到你当前分支上。 这通常用来从另一个分支合并单个或几个提交，而不是整个分支。例如，当前有 main 和 dev 两个分支，提交历史如下：

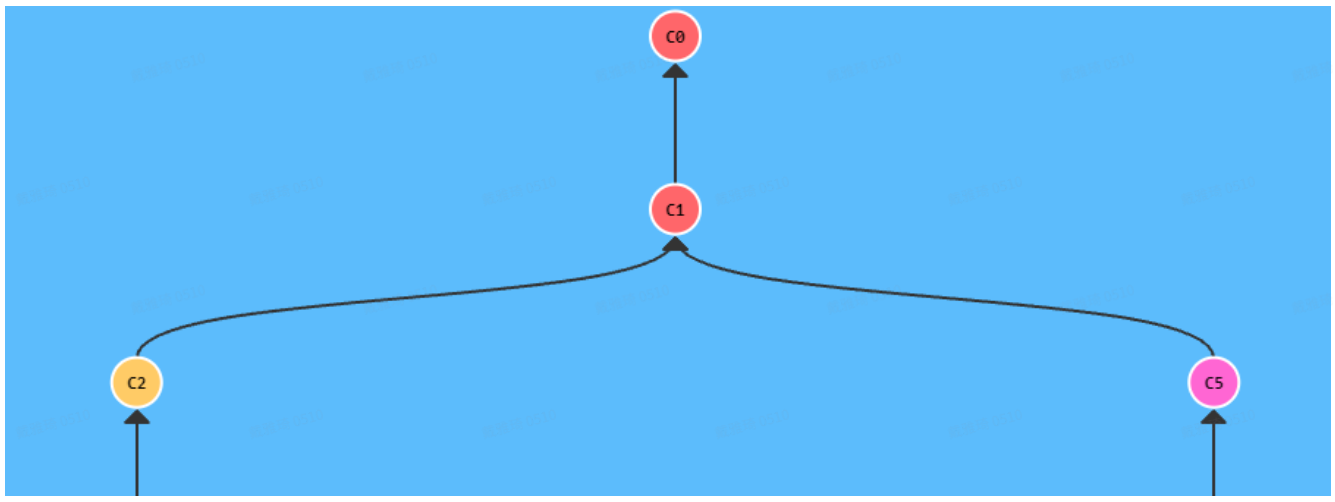


图9a git cherry-pick 示意（操作前）

如果我们只希望将 dev 分支中的 C3 提交合并到 main 分支中，可以使用以下方式：

Bash

```
$ git checkout main
$ git cherry-pick C3
```

此时，提交历史会变为：

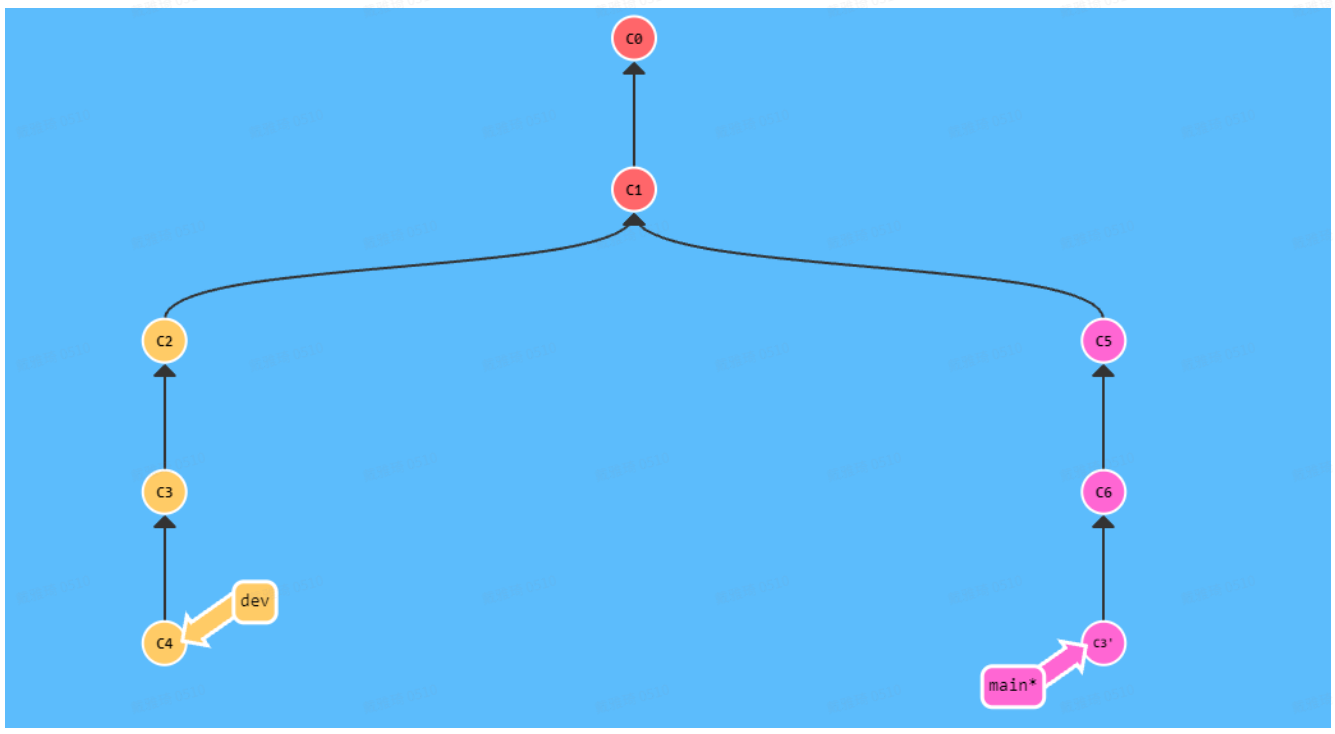


图9b git cherry-pick 示意（操作后）

`git cherry-pick` 指令的常见用法如下：

```bash
# 合并单个提交到当前分支
$ git cherry-pick <commit>
# 合并两个提交到当前分支
$ git cherry-pick <commit1> <commit2>

# 当合并遇到冲突时，可能需要手动解决，与 merge 中类似
$ git cherry-pick --continue  # 继续 cherry-pick
$ git cherry-pick --abort  # 中止 cherry-pick
```

- **git revert**

`git revert` 指令用来撤销之前的提交。与 `git reset` 指令移动 `HEAD` 指针的方式，`git revert` 会创建一个与之前提交的变更完全相反的新提交（本质上是一个特殊的 cherry-pick），从而实现撤销：

```bash
# 撤销单个或多个提交
$ git revert <commit1> [<commit2> ...]

# 同样，当遇到冲突时，解决的方式与 merge 中类似
$ git revert (--continue | --abort)
```

下图中可以更直观地对比 `git reset` 和 `git revert` 的区别：

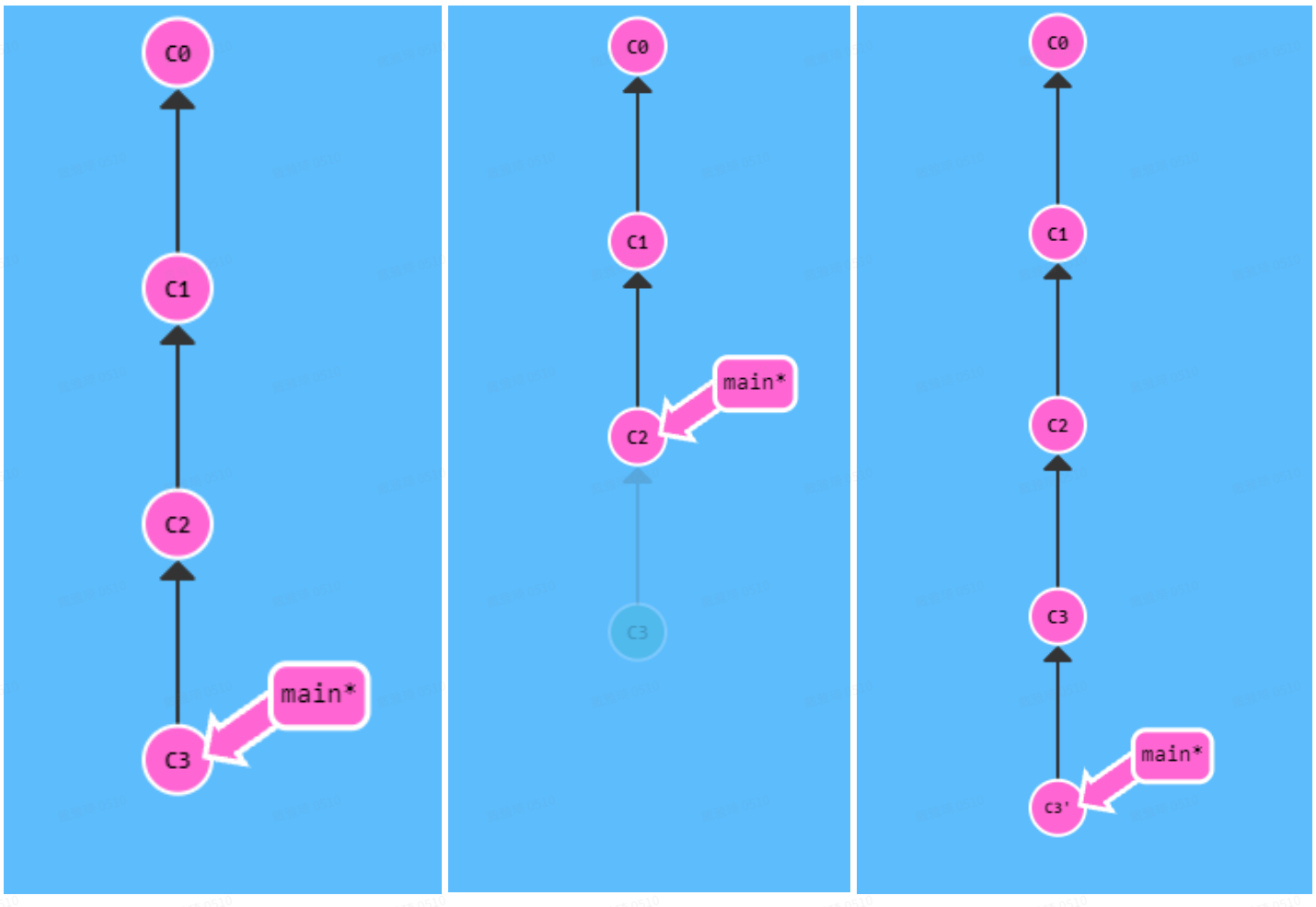


图10 git revert 与 git reset 对比

左：初始状态

中：执行 `git reset C2`

右：执行 `git revert C3`

- **git rebase**

`git rebase` 指令采用“变基”的方式整合不同分支的修改。回顾之前介绍的 `git merge` 指令，其工作方式是在当前分支创建一个新提交，在其中合并目标分支和当前分支的修改。与之不同，`git rebase` 会首先寻找当前分支和目标分支的“分叉点”，并尝试将当前分支在“分叉点“之后的部分”转移“到目标分支的末尾。下图中对比了使用 `git rebase` 和 `git merge` 合并分支的不同：

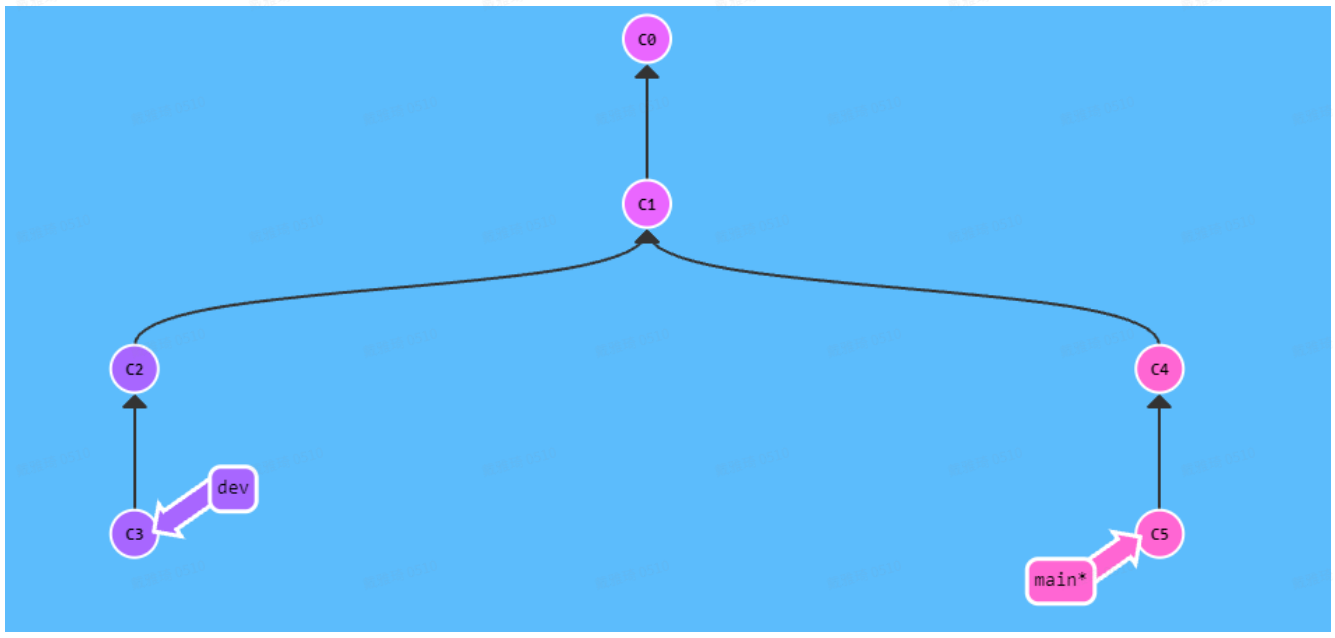


图11 git rebase 与 git merge 对比：初始状态

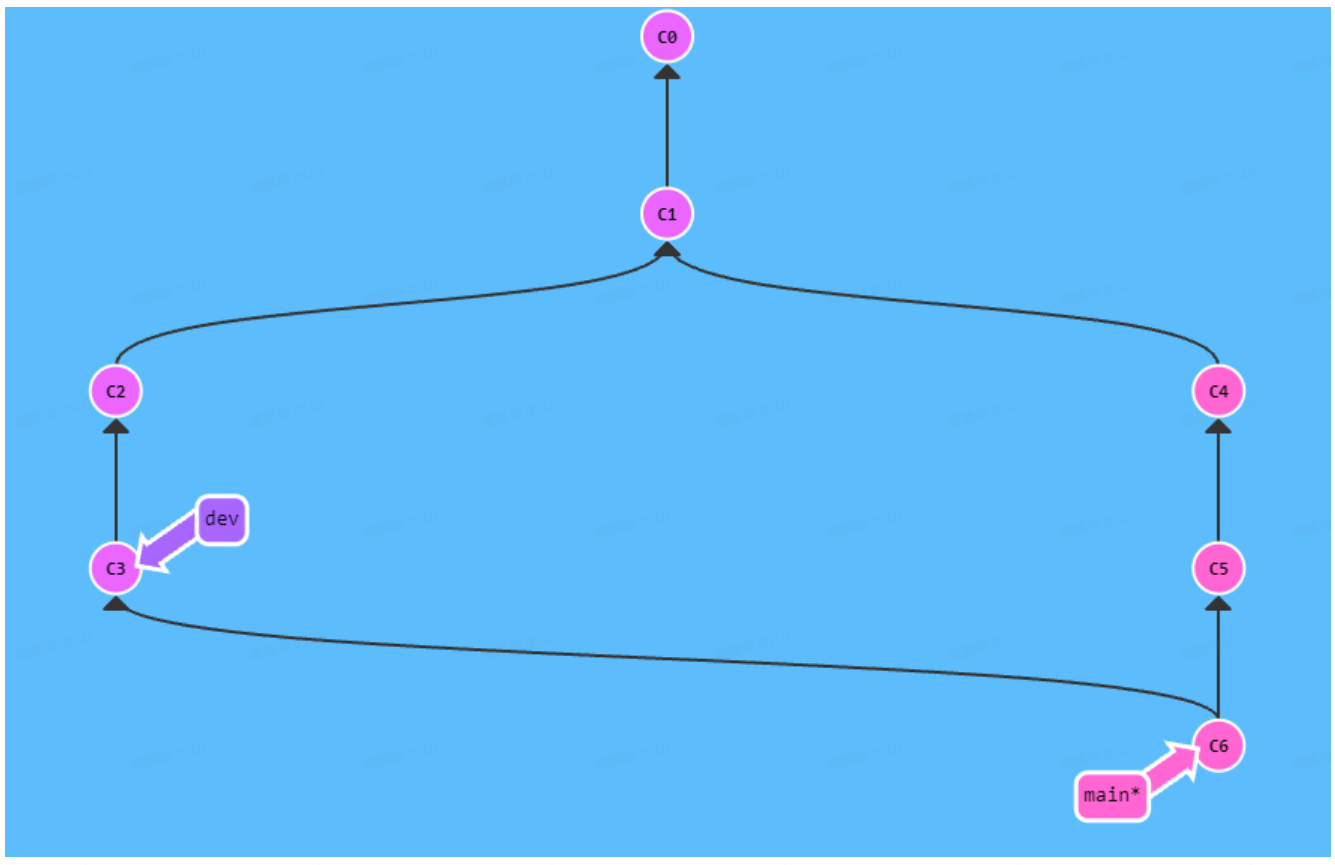


图11b git rebase 与 git merge 对比：在 main 分支（当前分支）上合并 dev 分支（目标分支）

```
$ git checkout main && git merge dev
```

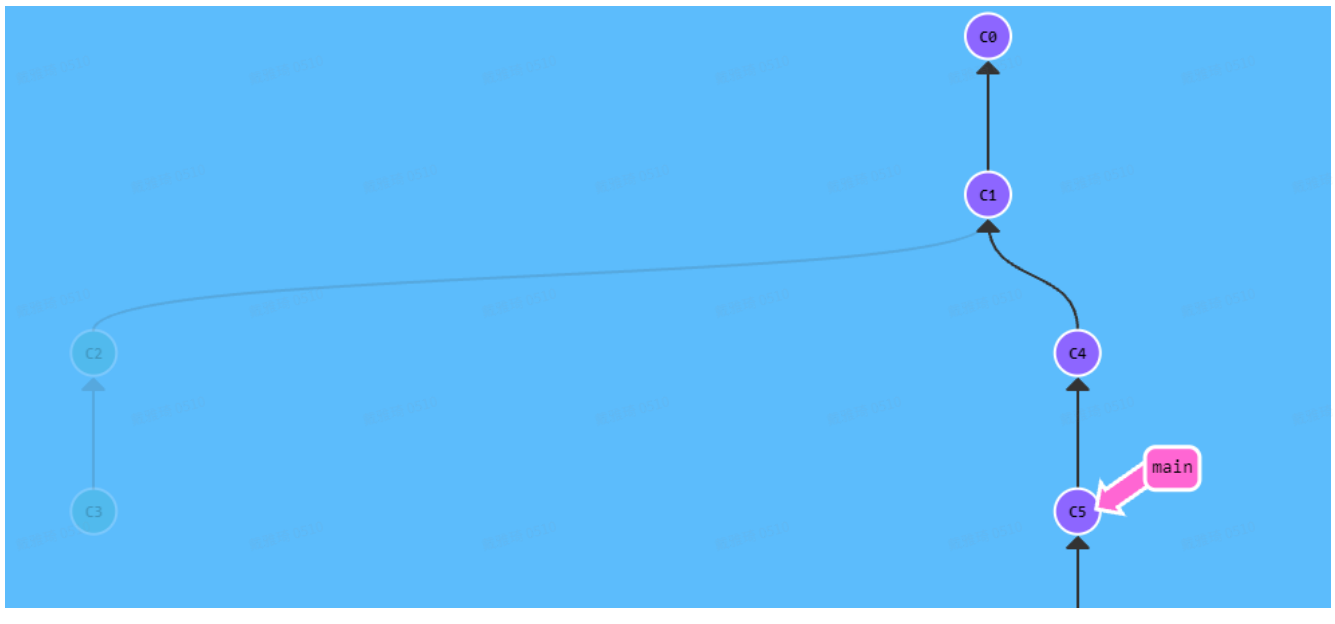


图11c git rebase 与 git merge 对比：将 dev 分支（当前分支）rebase 到 main 分支（目标分支）

`$ git checkout dev && git rebase main`

在上图中可以看到，`git rebase` 和 `git merge` 都可以实现分支的整合，但 `git rebase` 可以避免提交历史中出现分叉，保持较为清晰的提交历史。因此在实际项目开发中，通常会在将开发分支合入主分支前，先将其 rebase 到最新的主分支（在 3.1 部分中可以看到这一步）。

`git rebase` 指令的常见用法如下：

Bash

```
# 将当前分支 rebase 到目标分支 upstream-branch
$ git rebase <upstream-branch>

# 将指定分支 rebase 到目标分支 upstream-branch
$ git rebase <upstream-branch> <branch-name>
$ git rebase --onto <upstream-branch> <branch-name>~1  # 另一种方式

# 使用 -i 标志可以交互式的方式进行 rebase，使操作者可以手动选择对每个 commit 的处理方式。
# 一个常见用法是用这种方式合并当前分支的最近若干个 commit，例如：
$ git rebase -i HEAD~3
```

与 merge 类似，在 rebase 的过程中也可能会遇到冲突，需要手动解决。解决冲突的过程也与 merge 中相似（使用 `--continue`，`--abort` 等标志）。需要注意的是，由于 rebase 的工作方式是先切换到目标分支，然后依次添加当前分支中的提交，因此在处理冲突时，`HEAD` 指向的是目标分支，这一点与 merge 稍有不同（但在实际开发中，通常 merge 时把主分支作为当前分支，而 rebase 时把主分支作为目标分支，所以 `HEAD` 都会指向主分支 master（或 main）分支，所以不至于混淆）。

5. Git 实战

3.1 常规开发流程

在这一部分，我们以在 MMPose 算法库中完成一个功能开发的过程为例，介绍基于 Git 的常规开发流程。

3.1.1 准备

- **Fork 源代码仓库**

在 MMPose 的 Github 页面（https://github.com/open-mmlab/mmpose）点击 Fork 按键，在自己的 Github 帐号下创建一个副本仓库（如：https://github.com/ly015/mmpose）。

- **创建本地仓库并关联远程仓库**

到本地，并配制远程仓库地址。通常，我们会关联两个远程仓库： origin 为自己 fork 的副本仓库，upstream 为官方仓库

Bash

```bash
# clone 副本仓库。此时会自动关联远程仓库 origin
$ git clone https://github.com/ly015/mmpose.git
# 添加远程仓库 upstream
$ cd mmpose
$ git remote add upstream https://github.com/open-mmlab/mmpose.git
```

- **配制 Pre-commit Hook**

参考 Contributing to OpenMMLab 中的说明，在本地仓库配制 Pre-commit Hook。 Pre-commit Hook 是 Git 支持的钩子函数的一种，通常用于在提交修改（Commit）前自动完成代码风格检查等工作。

以上准备工作只是第一次提 PR 之前需要进行，后续开发工作，在每个 PR 开发中都需要做。

3.1.2 开发

- **创建开发分支**

通常我们会为每个开发项创建一个独立的分支，以避免耦合带来的混乱。分支的命名最好能简洁地体现开发项的内容。

Bash

```bash
# 拉取官方远程仓库
$ git fetch upstream

# 从最新的 master 分支创建开发分支
$ git checkout upstream/master
$ git checkout -b feature
# 上面两行指令等价与下面的指令
# git checkout -b feature --no-track upstream/master
```

- **完成功能开发**

在开发过程中，建议多使用 `git add` 和 `git commit` 及时提交修改。

- **将本地修改推送到远程仓库**

在完成开发并将本地修改都提交到本地仓库后，使用 `git push` 指令将本地修改同步到远程仓库。注意，我们通常不会将修改推送到官方远程仓库，而是推送到自己 fork 的远程仓库，再通过向官方仓库提 PR 的方式向其贡献代码。（这样不需要官方远程仓库的推送权限，并且有利于维护官方仓库的内容整洁）

Bash

```
# 首次推送，在远程仓库中建立关联分支
$ git push --set-upstream origin feature

# 非首次推送
$ git push

# 如果在本地进行了 merge，rebase 等操作，导致本地修改无法通过 fast-forwarded 方式
# 合入关联远程分支时，可以使用 -f 或 --force 参数强制更新远程分支（注意这是个风险操作）
$ git push -f
```

3.1.3 提交 PR

- **开始创建 PR**

当我们将本地的修改推送到自己的远程仓库后，需要向官方仓库提交一个 PR，要求官方仓库将这一修改合入到主分支（或其他指定分支）。创建 PR 可以在浏览器中操作。如果是近期推送的修改，在官方仓库或自己仓库的页面上都会出现提示，点击“Compare & pull request” 即可（如下图红色部分）。如果没有看到这个提示，也可以在自己仓库的页面上切换到开发分支，点击“Contribute”→“Open pull request”即可（如下图蓝色部分）。

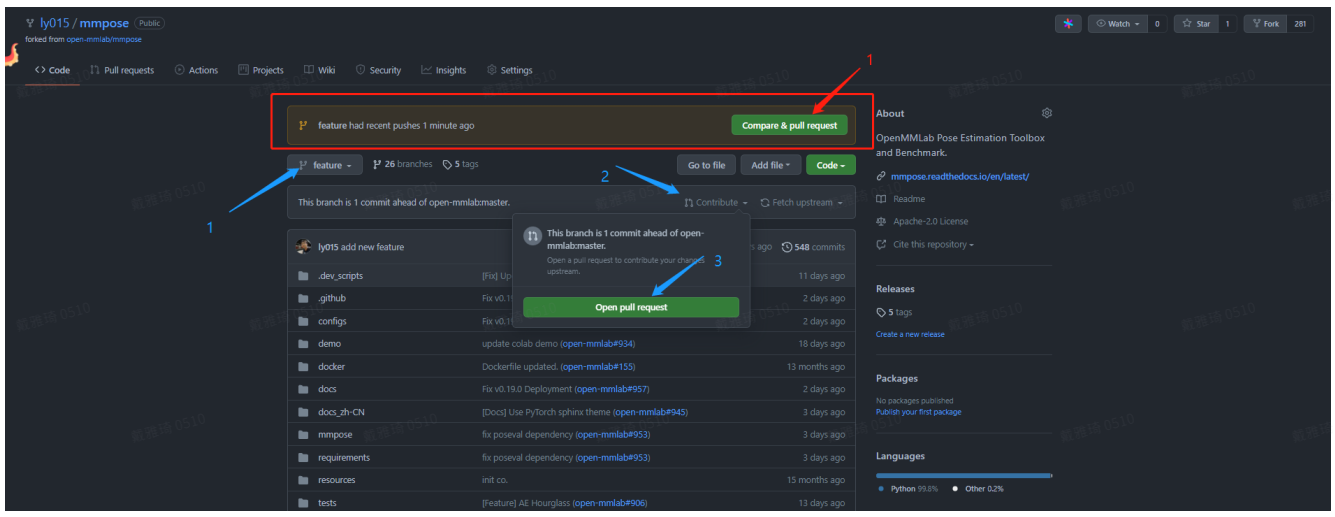


图12 开始创建 PR

- **提交 PR**

完成上一步后，会进入创建 PR 页面，如下图所示。这里我们需要填写 PR 相关的信息。红色部分是分支信息，默认将当前分支合入官方主分支，一般不用修改。绿色部分是 PR title 和 PR message，用来说明这个 PR 的内容。这部分非常重要，必须认真填写。

PR title 一般格式为 [Prefix] Short description of the pull request (Suffix)

Prefix：

- 新增功能 [Feature]
- 修 bug [Fix]
- 文档相关 [Docs]
- 开发中 [WIP] (work in process 暂时不会被review)

PR message 的主要修改内容，结果，以及对其他部分的影响，通常我们的代码库会准备 PR 模板，只需要按照模板填入对应的内容即可。另外，PR message 可以关联相关的 issue 和 PR，通过 fixes/resolves issue ID 可以在 PR merge 的时候 close issue。

蓝色部分是其他信息，在这里可以指定 reviewer 来 review 这个 PR。完成所有信息后，点击下方的“Create pull request”即可完成 PR 提交。

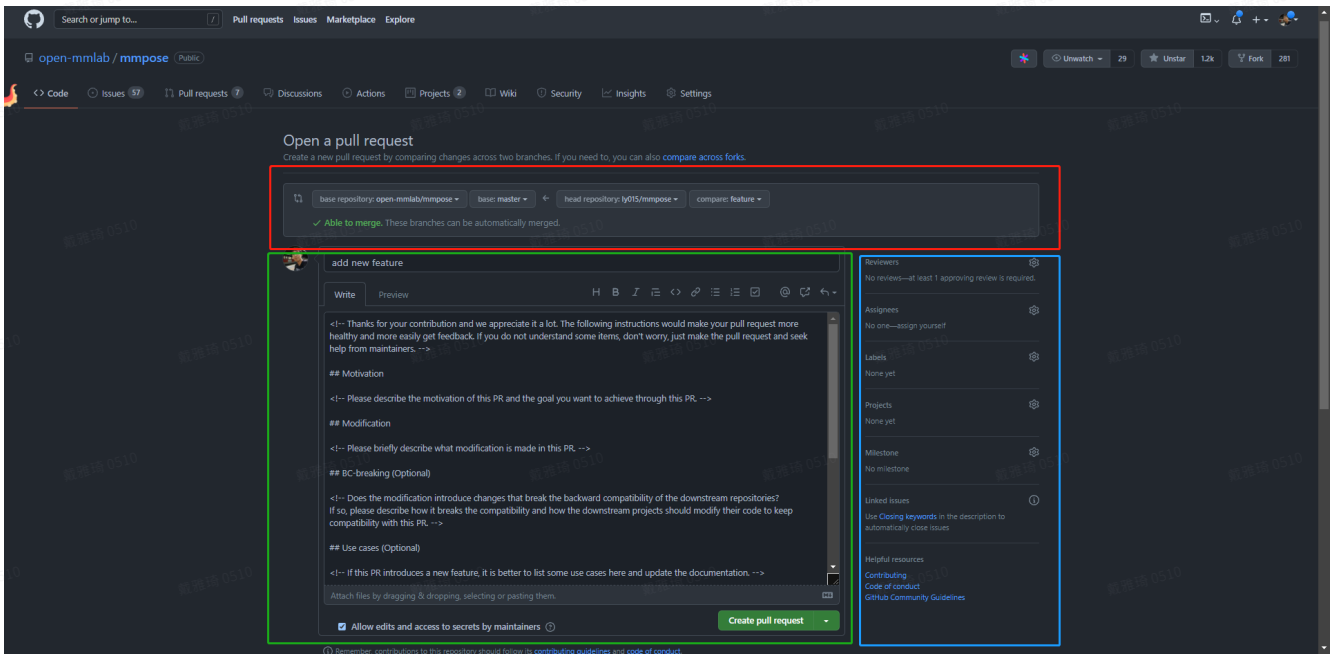


图13 创建 PR

- **查看 CI 状态**

PR 创建后会自动触发 CI（OpenMMLab 的算法库都基于 GitHub Actions 配置了 CI），完成代码格式、单元测试等检查工作。在 PR 页面可以查看 CI 的运行状态和结果，如下图所示。如果 CI 中有失败的项，可以点击“Details” 查看详细情况，修改后推送到自己的仓库，PR 也会随之更新。

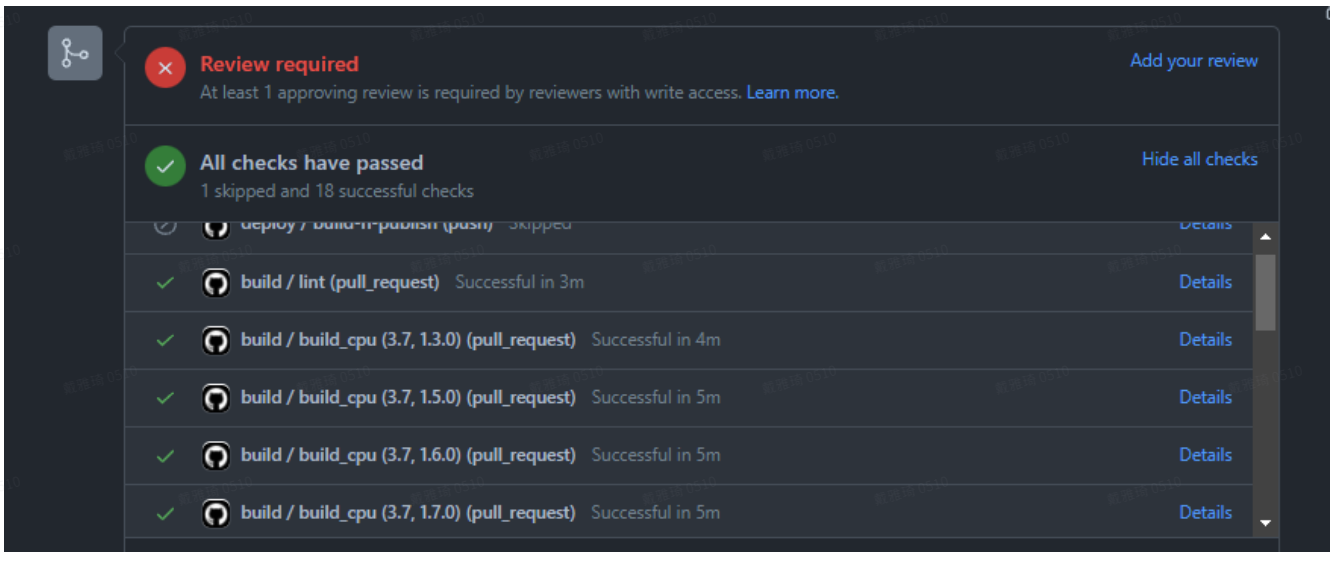


图13 CI 运行状态和结果

3.2 一些常用的 Git 技巧

3.2.1 使用 gitignore文件

`gitignore` 文件用来设置不希望 Git 进行跟踪和管理的文件路径，例如编译或运行时生成的临时文件、较大的数据文件等。`gitignore` 文件是一个文本文件，通常路径是项目目录下的 `.gitignore`，写法规范如下：

- 每一行指定一个要忽略的文件路径，除了空行和以 “#” 开头的注释行
- 文件路径可以使用 glob 模式匹配的写法
- 路径以 “/” 结尾表示要忽略的是一个目录
- 在文件路径前加入取反符号 “!” ，表示这是一个不应忽略的特例

以下是一个例子：

Plain Text

```
# 可使用"#"符号添加注释

# 忽略 "MANIFEST" 这个文件
MANIFEST

# 忽略所有 .pkl 文件
*.pkl
# 但是不忽略 a.pkl 这个文件
!a.pkl

# 忽略 data 目录下的所有文件
data/

# 忽略 model 目录下的所有 .pth 文件
model/**/*.pth
```

通常项目里已经提供了 `gitignore` 文件，必要时可以对其进行修改。 `gitignore` 文件本身也属于项目内容，需要添加修改 (`git add`)、提交修改(`git commit`)、开 PR 合入等。

3.2.2 配置 git mergetool

在合并分支时，经常需要手动解决冲突，这里可以选用一些可视化的工具帮助我们提高效率，例如 VS Code 就有较好的可视化功能，可以协助解决冲突。除此之外我们也可以借助第三方工具，这里我们以 meld 这个工具为例，介绍 Git 中 mergetool 的配置和使用。

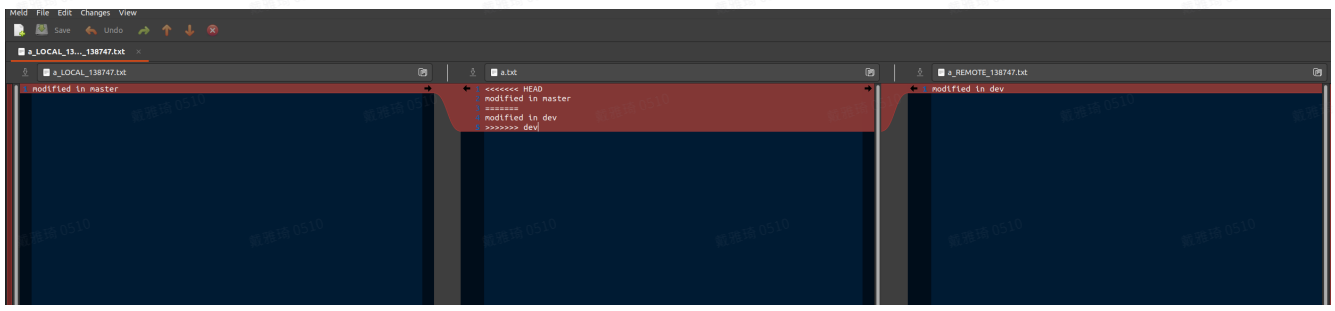
首先需要在本地安装 meld，可以参考其官网的文档。安装完成后，可以在 Git 配置文件（见 Git config 说明）中写入如下内容：

Plain Text

```
# 使用 meld 作为默认 difftool
[diff]
    tool = meld
[difftool "meld"]
    cmd = meld "$LOCAL" "$REMOTE"

# 使用 meld 作为默认 mergetool
[merge]
    tool = meld
[mergetool "meld"]
    cmd = meld "$LOCAL" "$MERGED" "$REMOTE" --output "$MERGED"
```

完成这些配制后，当需要解决冲突时，Git 会自动启动 meld，界面如下图所示。可以看到界面分成三列，对应与上面设置中的L11，分别是本地分支内容、合并后内容和目标（远程）分支内容。在这个界面中可以清晰对比两个分支的不同，还可以三列之间的箭头状按钮选快速选择内容添加到对应文件中。这里要注意，LOCAL 和 REMOTE 对应的分支，在 merge 和 rebase 两种情况下是不同的，这在 git-rebase 说明 中做过介绍。



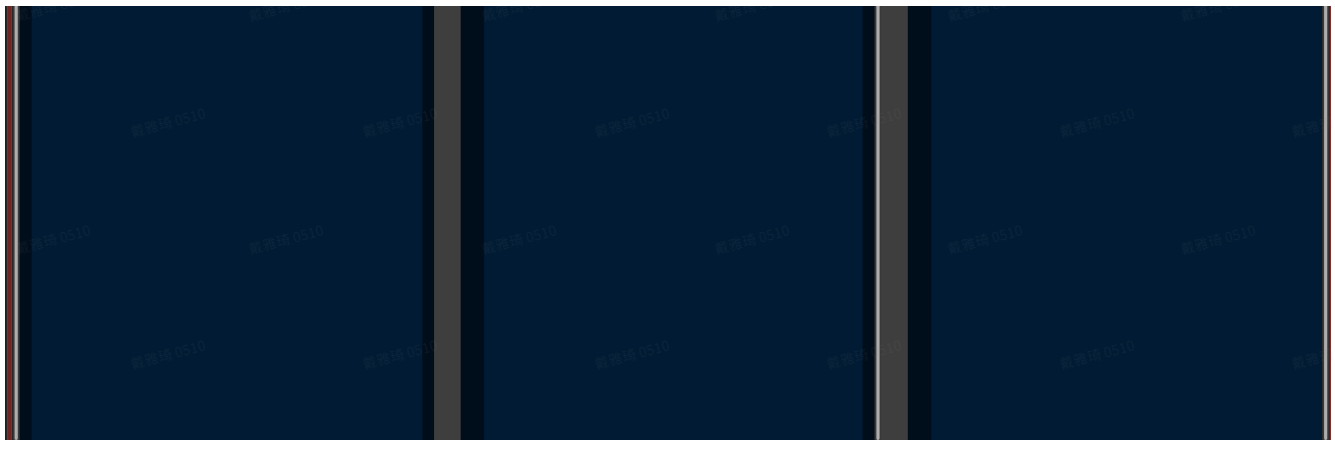


图15 使用 meld 进行可视化分支合并

由于我们同时配制了 "difftool"，所以在运行 `git diff` 指令时，也会启动 meld，用左右两列来对比文件差异。

3.2.3 添加自定义 pre-commit hook

在 3.1.1 中我们提到，Git 提供了 pre-commit hook 机制来帮助开发者自动完成格式检查等工作。一些常用的格式格式检查三方库，都提供了可以直接调用的 hook，可以在 `.pre-commit-config.yaml` 文件进行配制。除此之外，我们也可以在项目中添加本地代码脚本作为 pre-commit hook 以实现特定功能。

以 MMPose 中自动给代码文件添加版权信息的功能为例。我们首先添加了实现该功能的脚本 `.dev_scripts/github/update_copyright.py`，该脚本以一个文件列表作为输入参数，检查并添加版权信息到这些文件中。如果所有输入文件均已包含版权信息而未作修改，则正常返回 0，否则返回 1。如果该 hook 在提交时对代码做了修改，就会中止这次提交，待开发者添加这些修改后重新提交。最后，我们将该脚本添加到 `.pre-commit-config.yaml` 中即可：

YAML

```yaml
- repo: local
    hooks:
      - id: update-copyright
        name: update-copyright
        description: Add OpenMMLab copyright header to files
        # specify entry point
        entry: .dev_scripts/github/update_copyright.py
        # specify hook executor
        language: python
        # specify files that this hook will check
        files: ^(demo|docs|docs_zh-CN|mmpose|tests|tools|\.dev_scripts)/.*\.(py|c|cpp|cu|sh)$
        exclude: ^demo/mm(detection|tracking)_cfg/.*$
```

关于 Git hook 的详细介绍可以参考 Git - Git Hooks

3.2.4 使用 ssh 和远程仓库通信

与远程仓库通信可以使用 https 或 ssh 两种方式。在 2.3.6 中主要基于 https 方式进行了介绍，这里我们简单介绍 ssh 方式。在 clone 远程仓库到本地时，可以选择 ssh 地址，如下图所示：

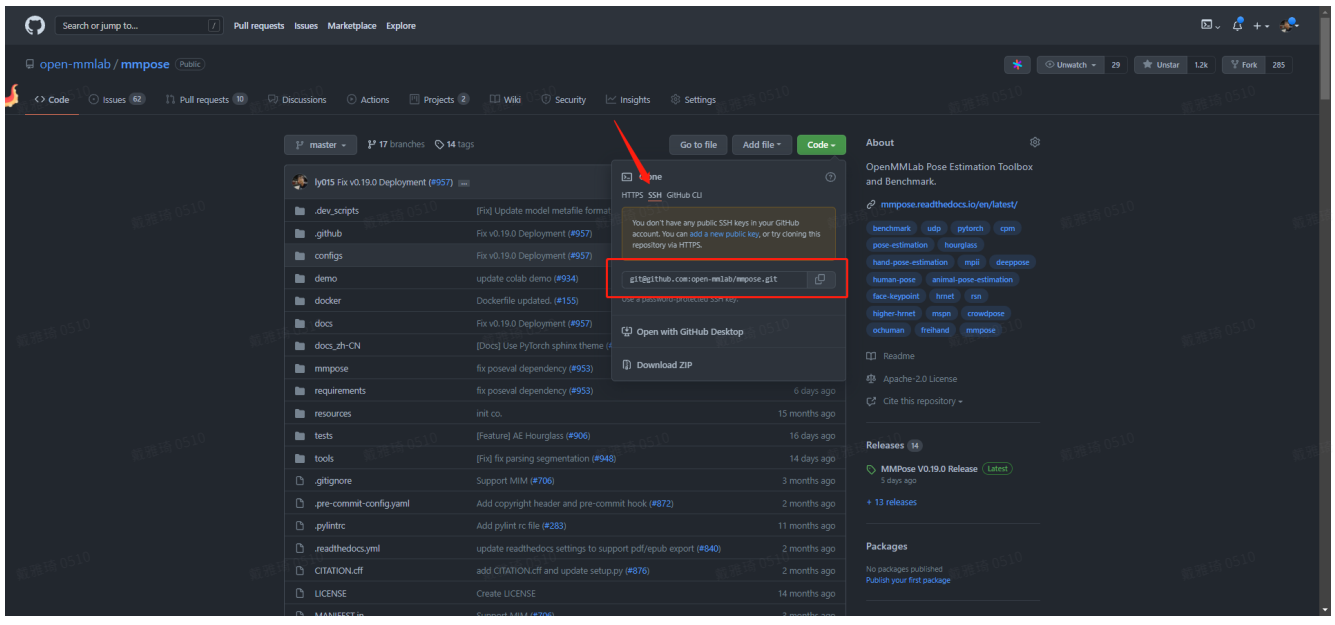


图16 使用 ssh 通信的远程仓库地址

使用 ssh 方式的仓库同步操作与使用 https 方式没有区别（如`git clone`，`git pull`，`git push`等）。ssh 方式需要配置公钥-私钥对，并将公钥提交到 GitHub 上，用以在通信时进行身份验证，不需要每次输入账号密码。配置 ssh 秘钥的具体的做法可以参考这里的文档：About SSH - GitHub Docs

3.3 FAQ

3.3.1 关于 Git 操作

- **如何修改 commit 但不产生新的提交**

请参考 2.3.4 中`git commit --amend`的用法。请注意，严格来说这个操作是用一个新的提交替换了原来的提交。

- **如何撤销 commit 但仍需保留修改**

请参考 2.4 中`git reset --mixed`或`git reset --soft`的用法。

- **如何合并多个 commit**

请参考 2.4 中`git rebase -i HEAD~n`的用法，这里举一个简单的例子。假如当前分支历史中有3个提交，如下：

Bash

```
$ git log
commit 7eea97ea4cd9ff50ccf7ce146c4b440a98c85c51 (HEAD -> master)
Author: ly015 <liyining0712@gmail.com>
Date:   Sat Oct 9 18:39:45 2021 +0800

    Add c.txt

commit 7d97740b91137ec8049c83726ba0c950090cb8f3
Author: ly015 <liyining0712@gmail.com>
Date:   Sat Oct 9 18:39:27 2021 +0800

    Add b.txt

commit 7f836e4b528a565b42d41c021197242b87a038f4
Author: ly015 <liyining0712@gmail.com>
Date:   Sat Oct 9 18:39:03 2021 +0800

    Add a.txt
```

使用`git rebase`对后2个提交进行合并：

Bash

```
$ git rebase -i HEAD~2
```

此时会出现交互界面，如下图左。可以看到相关的 commit 被逐行列出，并且每个 commit 前面有一个指令（pick），这个指令是可以按照需求修改的，所有可选项都在下方的注释中有说明。例如，我们现在希望合并2个 commit， 则可以将第一个 7d97740 的指令设置为 reword，将第二个 7eea97e 的指令设置为 fixup，如下图中，然后保存退出。由于第一个指令是 reword，接下来会跳出重新输入 commit message 的界面，我们将原本的 “Add b.txt” 改成 “ Add b.txt and c.txt ”， 并保存退出，如下图右。

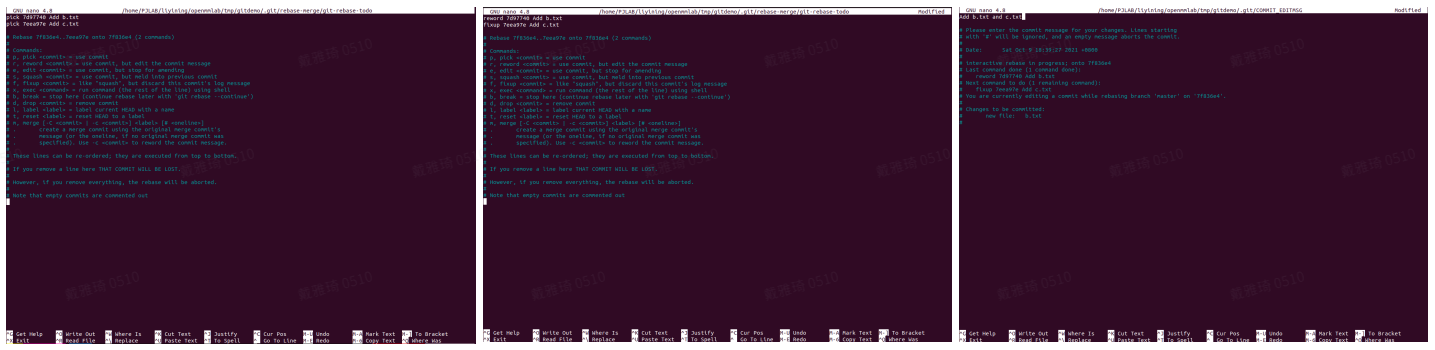


图17 使用`git rebase`合并多个 commit

此时我们再查看分支提交历史，会发现之前的2个 commit 已经被合并：

Bash

```
$ git log
commit a321f6787a3cb2f27f70f5f7877e158a2aea7e0d (HEAD -> master)
Author: ly015 <liyining0712@gmail.com>
Date:   Sat Oct 9 18:39:27 2021 +0800

    Add b.txt and c.txt

commit 7f836e4b528a565b42d41c021197242b87a038f4
Author: ly015 <liyining0712@gmail.com>
Date:   Sat Oct 9 18:39:03 2021 +0800

    Add a.txt
```

- **如何删除 add 或者 commit 之后的大文件**

在开发中会遇到不慎将一些文件错误添加到 Git 仓库中的情况（比如未及时更新 `gitignore` 文件就可能会导致这种情况）。尤其是以一些大文件（如数据文件或临时文件），如果被错误包含在项目仓库中，会使仓库变得很大而严重影响和远程仓库通信的效率（如 `git clone`，`git push`，`git pull` 等操作会变得很慢）。

如果只是通过 `git add` 添加到了暂存区，只需要用前面介绍过的 `git rm --cached` 或 `git reset` 清理暂存区即可。

但如果已经通过 `git commit` 提交到了仓库，情况会变得比较棘手，因为 Git 仓库会记录完整的提交历史，即使撤销这次提交，或者在后续提交中移除这些错误添加的文件，它们依旧会存在与仓库中。这种情况也不是无法解决的，这篇文档 Git - 维护与数据恢复 中给出了一个清除历史提交中错误引入的大文件的完整例子，大家可以仔细学习并 follow 其步骤。

3.3.2 关于 PR

- **如何往源算法库中其他人提的 PR 中提交代码**

在创建 PR 时，通常会默认勾选 “Allow edits and access to secrets by maintainers”，这将允许其他具有官方仓库 Write 权限的用户提交修改到这个 PR 对应的分支。这在多人合作开发时会用到。

当需要向别人的 PR 中提交代码时，需要将作者的远程仓库地址添加到 remote 中（参考 2.3.6 git remote），然后拉取 PR 对应的分支。在本地完成修改后，再推送到作者的远程仓库即可。

在将 PR 拉取到本地时，除了从作者的远程仓库拉取，还可以用以下简单的指令：

Bash

```
# 拉取 PR#222 到本地并为其创建一个本地分支
$ git fetch upstream pull/222/head:<branchname>
# 切换到 PR 分支
$ git checkout <branchname>
```

- **如何手动运行 CI**

可以在 PR 页面的 Checks 标签页，点击右上角的 “Re-run all jobs” 按键，即可重新运行 CI。

3.4 关于 GitLab

3.4.1 什么是 GitLab

GitLab 是一个基于 Git 的开源的代码仓库管理工具。商汤使用 GitLab 搭建了内部的代码托管服务，如 https://gitlab.sh.sensetime.com/。 GitLab的远程仓库页面如下图所示。其页面内容和使用方式都和 Github 类似。

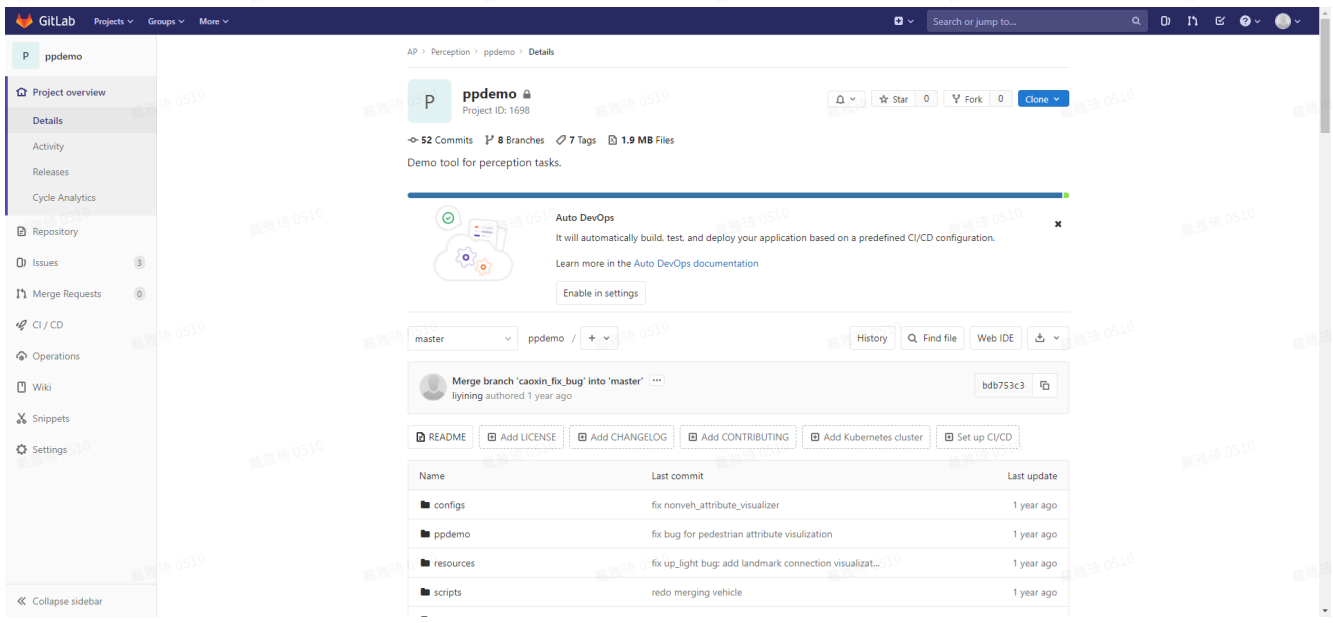


图18 GitLab 项目页面

4 参考资料

- Git 官方教程：https://git-scm.com/book/zh/v2
- 2个 Git 沙盘：
  - https://learngitbranching.js.org/
  - https://git-school.github.io/visualizing-git/
- 廖雪峰 Git 教程：https://www.liaoxuefeng.com/wiki/896043488029600