

Git 入门指南

2019/12/30 David.

目录

contents

PART 1

Git 是什么

PART 2

Git 简明指南

PART 3

Git 常用指令

PART 4

使用 Git 一般开发规范

PART 01

- Git 是什么？
- Git 的特点
- Git 的三种状态
- Git 的工作区域
- Git 能做什么？
- Git 的不足

Git 是什么？

- Git 是什么
 - Git 是一个开源的分布式版本控制系统
 - Git 是一种目录内容管理系统
 - Git 是一种树型历史存储系统
 - Git 是一种单纯的文件异动追踪者

Git 特点

- 直接记录快照，而非差异比较
 - Git 更像是把数据看作是对小型文件系统的一组快照。每次你提交更新，或在 Git 中保存项目状态时，它主要对当时的全部文件制作一个快照并保存这个快照的索引。
 - 为了高效，如果文件没有修改，Git 不再重新存储该文件，而是只保留一个链接指向之前存储的文件。Git 对待数据更像是一个快照流。
- 近乎所有操作都是本地执行
 - 在 Git 中的绝大多数操作都只需要访问本地文件和资源，一般不需要来自网络上其它计算器的信息。

Git 特点

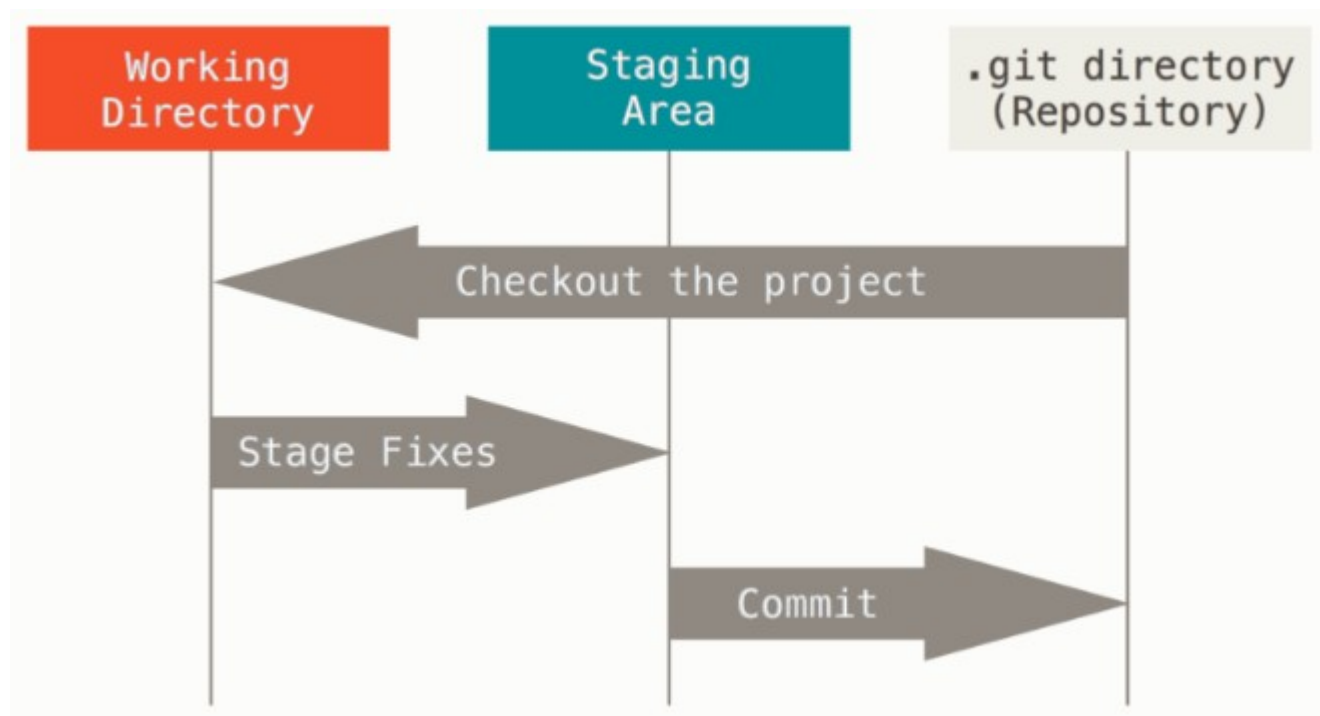
- Git 保证完整性
 - Git 中所有数据在存储前都计算校验和，然后以校验和来引用。这意味着不可能在 Git 不知情时更改任何文件或目录内容。
- Git 一般只添加数据
 - 你执行的 Git 操作，几乎只往 Git 数据库中增加数据。很难让 Git 执行任何不可逆操作，或者让它以任何方式清除数据。

Git 三种状态

- 你的文件可能处于其中之一：
 - 已修改（`modified`）：表示修改了文件，但还没保存到数据库中。
 - 已暂存（`staged`）：表示对一个已修改文件的当前版本做了标记，使之包含在下次提交的快照中。
 - 已提交（`committed`）：表示数据已经安全的保存在本地数据库中。

Git 工作区域

- Git 项目的三个工作区域的概念：
 - 工作目录、暂存区域以及 Git 仓库。



Git 工作区域

- Git 仓库目录是 Git 用来保存项目的元数据和对象数据库的地方。这是 Git 中最重要的部分，从其它计算器克隆仓库时，拷贝的就是这里的数据。
- 工作目录是对项目的某个版本独立提取出来的内容。这些从 Git 仓库的压缩数据库中提取出来的文件，放在磁盘上供你使用或修改。
- 暂存区域是一个文件，保存了下次将提交的文件列表信息，一般在 Git 仓库目录中。有时候也被称作？索引？，不过一般说法还是叫暂存区域。

Git 工作区域

- 基本的 Git 工作流程如下：
 - 在工作目录中修改文件。
 - 暂存文件，将文件的快照放入暂存区域。
 - 提交更新，找到暂存区域的文件，将快照永久性存储到 Git 仓库目录。
- 如果 Git 目录中保存着特定版本的文件，就属于已提交状态。
- 如果作了修改并已放入暂存区域，就属于已暂存状态。
- 如果自上次取出后，作了修改但还没有放到暂存区域，就是已修改状态。

Git 能做什么？

- 项目备份
- 项目开发的分工 / 合并 /
- 记录开发的历史轨迹 (code and document only)
- 有条件的时光隧道 (历史版本间切换 / 版本追踪)
- 管理五大主轴：
 - Feature (bugfix)
 - Develop
 - Release
 - Hot-fix
 - Master

Git 不足

- 无单元测试
- 主要用于管理代码，也可以管理其它静态资源
- 无法强制：依规定处理 code 的 commit, push 及必要的批注
- 无法强制：依规定做 release, hot-fix 管理
- 无法管制编号的合法性
- 无法管制流程的合法性
- Git 只是个团队开发的合作工具， 依靠制度、纪律、使用习惯等

The background of the entire slide is composed of numerous concentric circles in a light gray color, creating a ripple effect that draws the eye towards the center.

PART

02

- Git 简明指南—补充说明

Git 简明指南

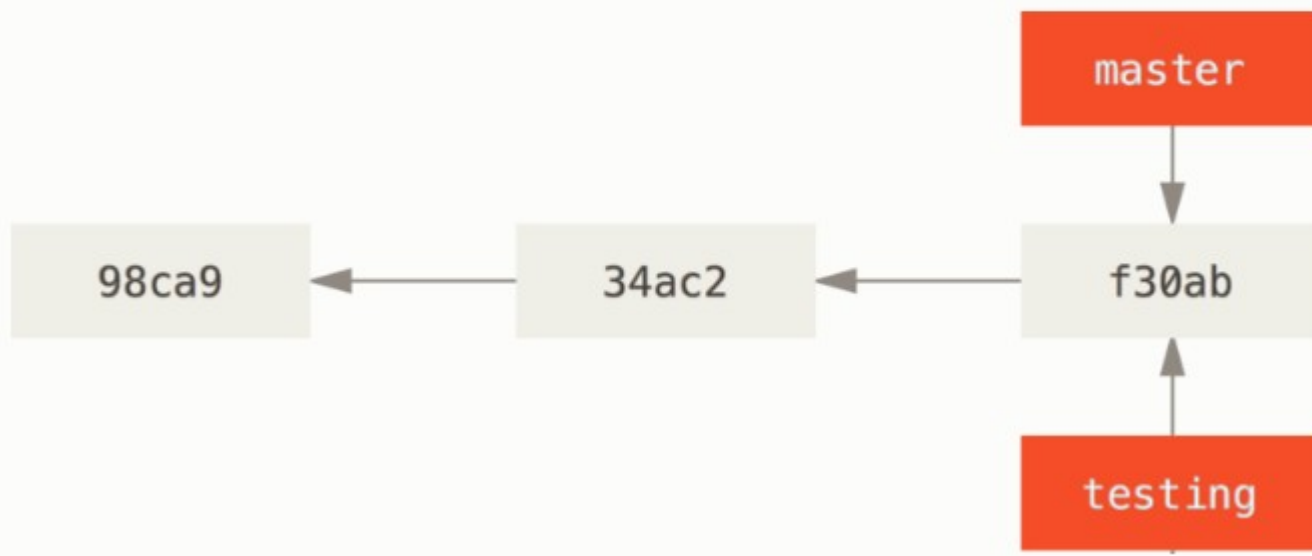
- Git 简明指南（罗杰 • 杜德勒）
 - 可访问：<https://rogerdudler.github.io/git-guide/index.zh.html>

补充说明 - 分支

- 分支
 - Git 创建分支
 - `git branch <branch name>`
 - 它只是为你创建了一个可以移动的新的指针。

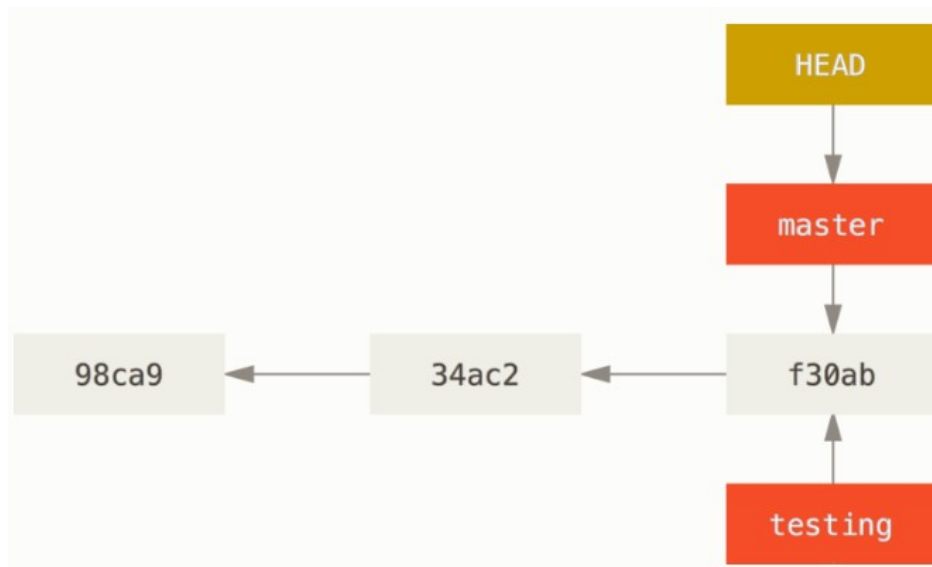
```
$ git branch testing
```

这会在当前所在的提交对象上创建一个指针。



补充说明 - 分支

- Git 又是怎么知道当前在哪一个分支上呢？
 - 它有一个名为 HEAD 的特殊指针。
 - 在 Git 中，它是一个指针，指向当前所在的本地分支（将 HEAD 想象为当前分支的别名）。
 - 在本例中，仍然在 master 分支上。因为 git branch 命令仅仅创建一个新分支，并不会自动切换到新分支中去。

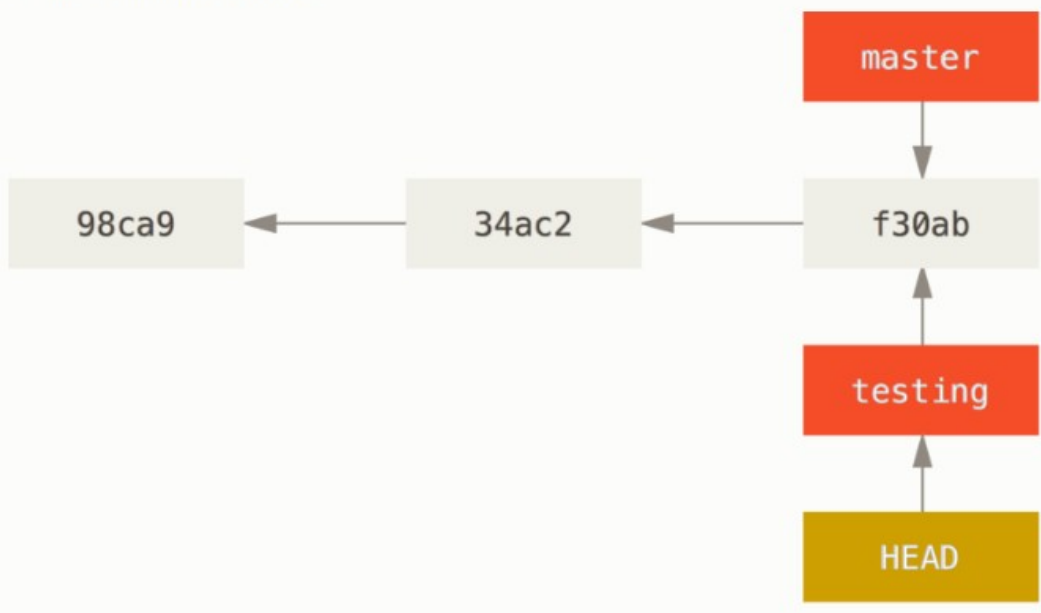


补充说明 - 分支

- 分支切换
 - 使用 `git checkout` 命令，切换到一个已存在的分支。

```
$ git checkout testing
```

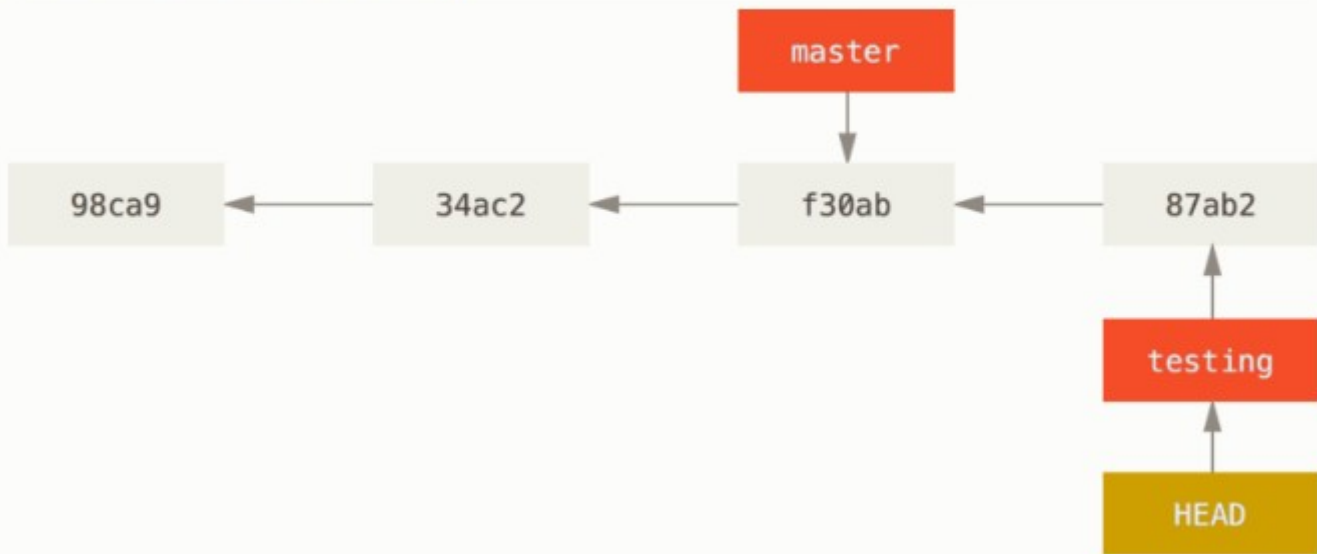
这样 HEAD 就指向 testing 分支了。



补充说明 - 分支

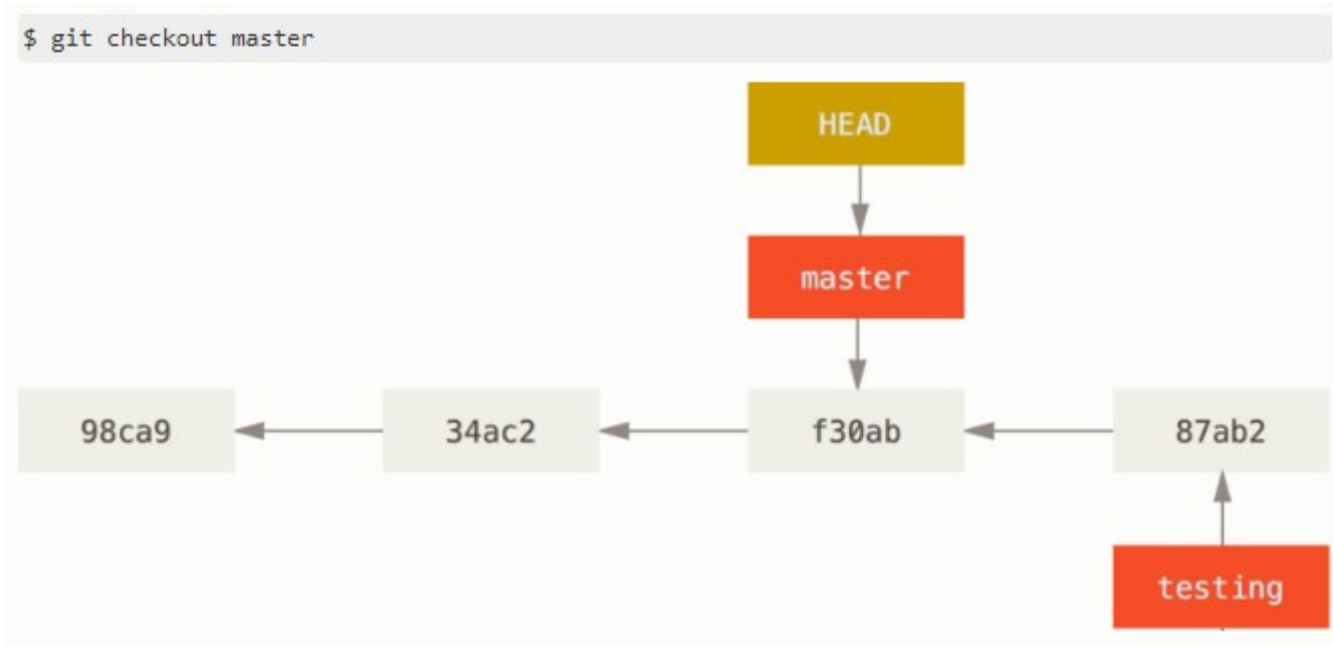
- 现在不妨修改文件再提交一次：
 - HEAD 分支随着提交操作自动向前移动
 - 如图所示，testing 分支向前移动了，但是 master 分支却没有，它仍然指向运行 git checkout 时所指的对象。

```
$ vim test.rb  
$ git commit -a -m 'made a change'
```



补充说明 - 分支

- 切换回 master 分支看看：
 - 这条命令做了两件事。一是使 HEAD 指回 master 分支，二是将工作目录恢复成 master 分支所指向的快照内容。也就是说，你现在做修改的话，项目将始于一个较旧的版本。

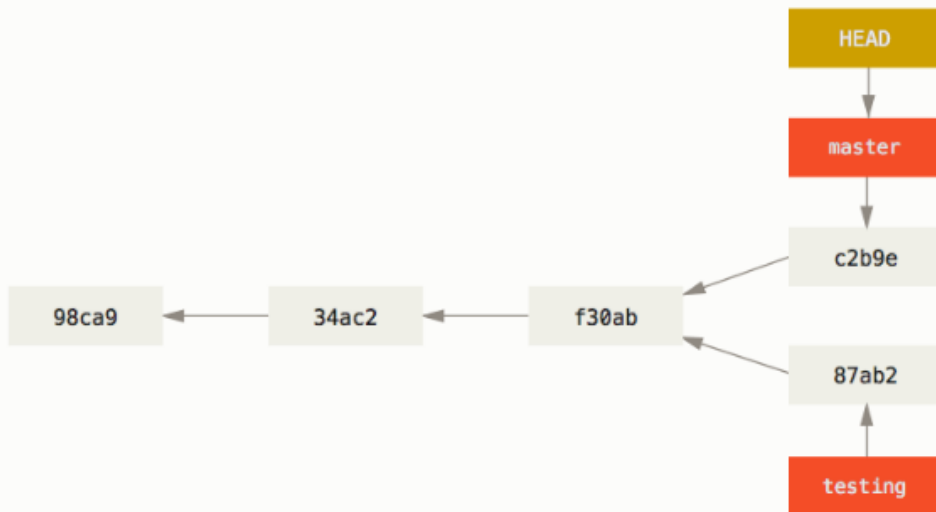


补充说明 - 分支

- 再稍微做些修改并提交：

```
$ vim test.rb  
$ git commit -a -m 'made other changes'
```

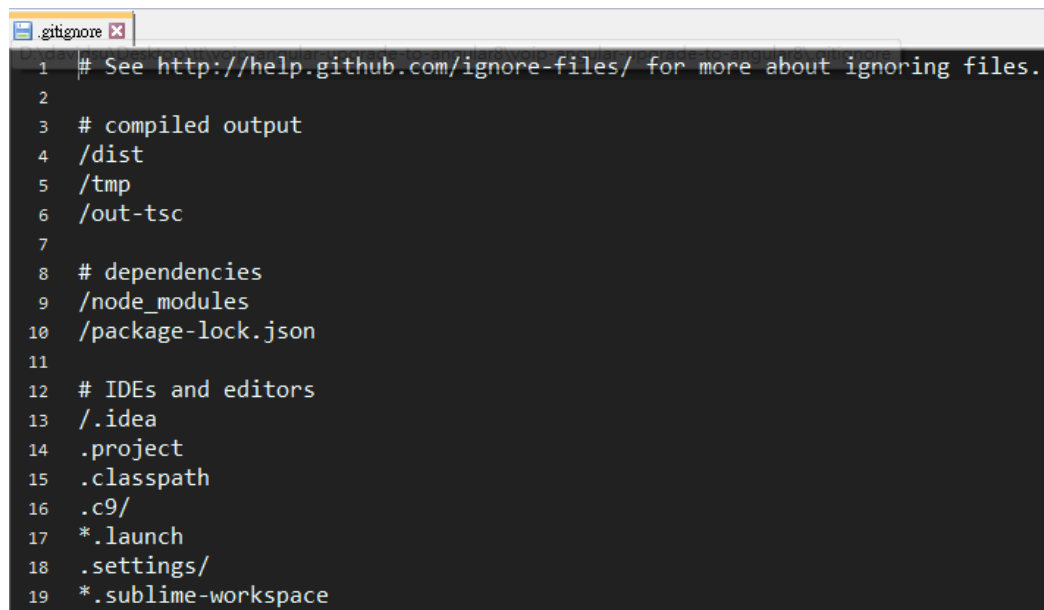
现在，这个项目的提交历史已经产生了分叉（参见 [项目分叉历史](#)）。因为刚才你创建了一个新分支，并切换过去进行了一些工作，随后又切换回 master 分支进行了另外一些工作。上述两次改动针对的是不同分支：你可以在不同分支间不断地来回切换和工作，并在时机成熟时将它们合并起来。而所有这些工作，你需要的命令只有 `branch`、`checkout` 和 `commit`。



Ref : [https://git-scm.com/book/zh/v2/Git-%E5%88%86%E6%94%AF-%E5%88%86%E6%94%AF%E7%AE%E8%80%E4%BB%8B](https://git-scm.com/book/zh/v2/Git-%E5%88%86%E6%94%AF%E5%88%86%E6%94%AF%E7%AE%E8%80%E4%BB%8B)

补充说明 - 管理例外

- 不需要 git 管理的例外
 - 有部分项目的内容，例如 nodejs 的依赖包，Java 的依赖包等，不需要 git 管理，只保留在本机，也不用上传到 gitlab 远程，就可以添加到例外中
 - 创建 .gitignore 文件，添加项目的文件夹或文件名
 - 示例：

A screenshot of a code editor window titled ".gitignore". The editor shows a list of file patterns to be ignored, with line numbers 1 through 19 on the left. The patterns include comments about where to find more information, compiled output directories, dependencies, IDEs and editors, and other project-specific files.

```
1 # See http://help.github.com/ignore-files/ for more about ignoring files.
2
3 # compiled output
4 /dist
5 /tmp
6 /out-tsc
7
8 # dependencies
9 /node_modules
10 /package-lock.json
11
12 # IDEs and editors
13 /.idea
14 .project
15 .classpath
16 .c9/
17 *.launch
18 .settings/
19 *.sublime-workspace
```

The background of the entire slide consists of numerous concentric circles in a light gray color, centered on the page. These circles vary in radius, creating a ripple effect that fills the entire frame.

PART

03

• Git 常用指令

git 常用指令:

- 配置使用 Git 的账号密码:
 - `git config --global user.name "Your Name"`
 - `git config --global user.email email@example.com`
- 初始化一个 Git 仓库:
 - `git init`
- 添加文件到 Git 仓库, 分两步:
 - 添加到暂存区:
 - `git add <file>` // 注意, 可反复多次使用, 添加多个文件;
 - 提交到仓库:
 - `git commit -m <message>`

git 常用指令:

- 查看工作区的状态:
 - `git status`
- 可以查看修改内容:
 - `git diff`
- 关联一个远程库:
 - `git remote add origin git@server-name:path/repo-name.git`
- 关联后, 使用命令第一次推送 master 分支的所有内容:
 - `git push -u origin master`
- 此后, 每次本地提交后, 推送最新修改;
 - `git push origin master`

git 常用指令:

- 要克隆一个仓库，先必须知道仓库的地址，再使用：
 - `git clone git@server-name:path/repo-name.git`
- 查看分支：
 - `git branch`
- 创建分支：
 - `git branch <name>`
- 切换分支：
 - `git checkout <name>`
- 创建 + 切换分支：
 - `git checkout -b <name>`

git 常用指令:

- 合并某分支到当前分支:
 - `git merge <name>`
- 删除分支:
 - `git branch -d <name>`
- 看到分支合并图:
 - `git log -graph`
- 查看远程库信息:
 - `git remote -v`
- 从本地推送分支:
 - `git push origin branch-name`

git 常用指令:

- 抓取远程的新提交:
 - `git pull`
- 在本地创建和远程分支对应的分支:
 - `git checkout -b branch-name origin/branch-name` // 本地和远程分支的名称最好一致;
- 建立本地分支和远程分支的关联:
 - `git branch --set-upstream branch-name origin/branch-name`

git 常用指令:

- 撤销 (undoing)
 - git checkout
 - Checkout 特定版本或分支
 - git reset
 - 将目录还原至特定的版本状态
 - 使用 `--force` 强制还原
 - git revert
 - 还原 commit 动作
 - 已 commit 的对象不做删除, 只加上 patch 作为修补
 - Reverts can themselves be reverted!
 - Git 不删除任何已经 committed 的对象

The background of the slide features a series of concentric circles in a light gray color, centered on the page. The circles vary in opacity, creating a subtle, hypnotic effect.

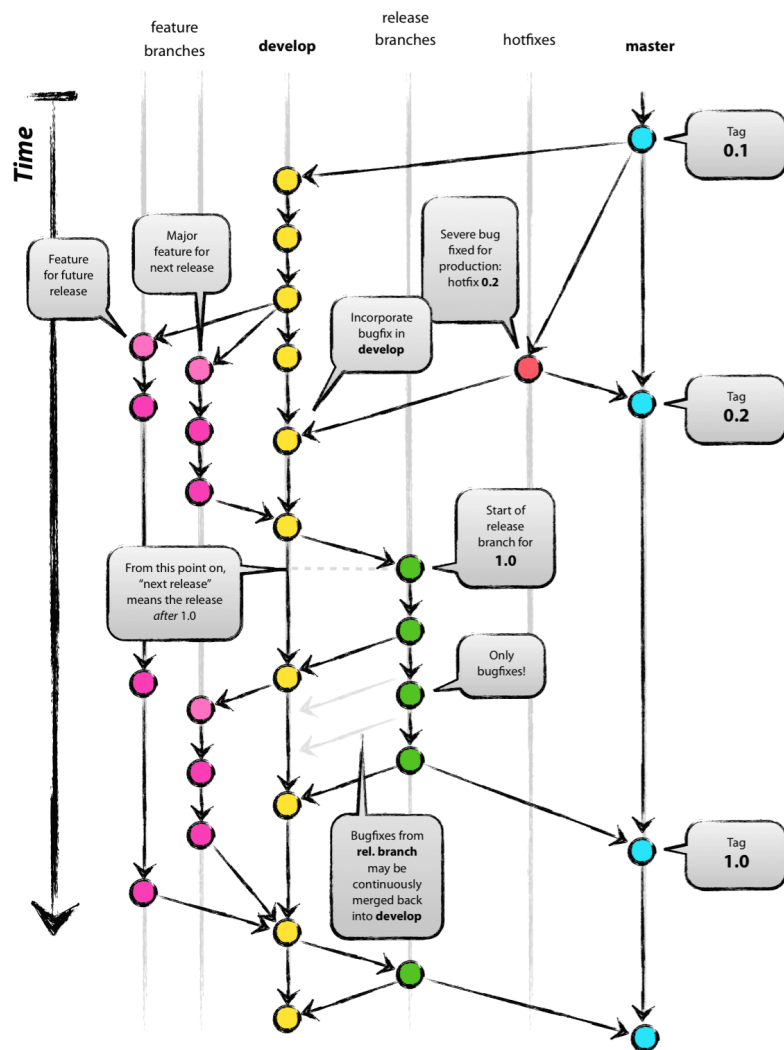
PART

04

- 使用 Git 一般开发规范
- git 可视化工具

使用 Git 一般开发规范

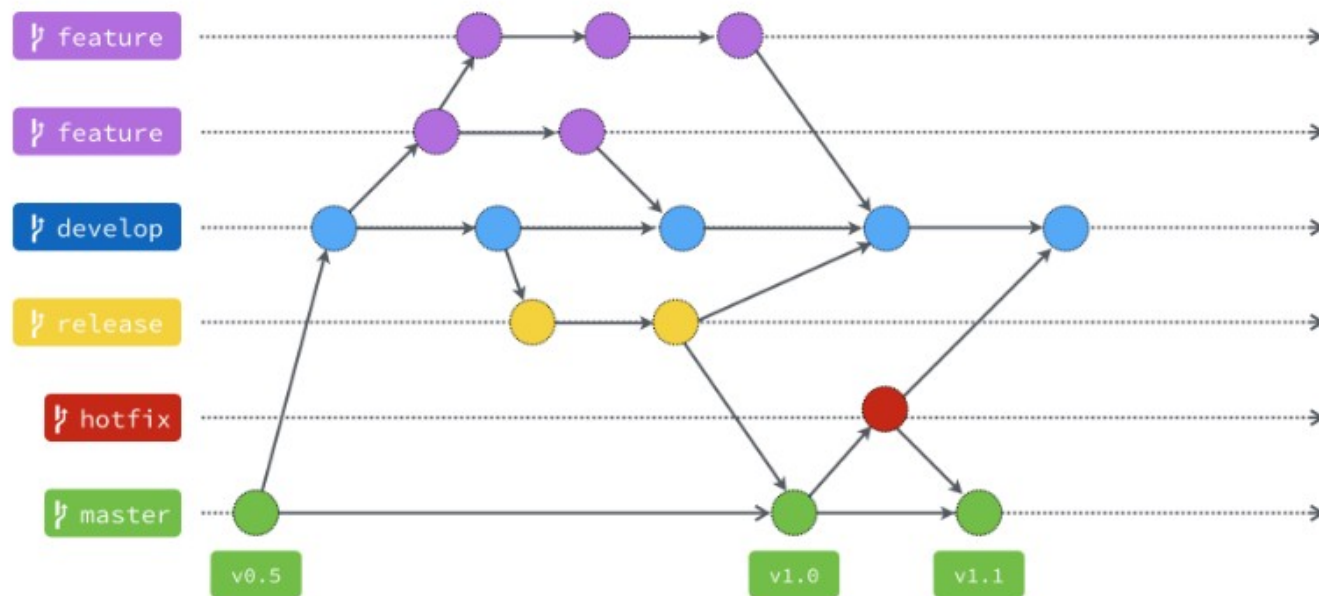
• Git Flow



Ref : <https://nvie.com/posts/a-successful-git-branching-model/>

使用 Git 一般开发规范

- 五大分支



Ref : <https://gitbook.tw/chapters/gitflow/why-need-git-flow.html>

使用 Git 一般开发规范

- Master 分支

- 主要是用来放稳定、随时可上线的版本。这个分支的来源只能从别的分支合并过来，开发者不会直接 Commit 到这个分支。因为是稳定版本，所以通常也会在这个分支上的 Commit 上打上版本号标签。

- Hotfix 分支

- 当在线产品发生紧急问题的时候，会从 Master 分支开一个 Hotfix 分支出来进行修复，Hotfix 分支修复完成之后，会合并回 Master 分支，也会同时合并一份到 Develop 分支。
- 为什么要合并回 Develop 分支？如果不这么做，等到时候 Develop 分支完成并且合并回 Master 分支的时候，那个问题就又再次出现了。
- 那为什么一开始不从 Develop 分支切出来修？因为 Develop 分支的功能可能尚在开发中，这时候硬是要从这里切出去修再合并回 Master 分支，只会造成更大的灾难。

使用 Git 一般开发规范

- Release 分支

- 当认为 Develop 分支够成熟了，就可以把 Develop 分支合并到 Release 分支，在这边进行算是上线前的最后测试。测试完成后，Release 分支将会同时合并到 Master 以及 Develop 这两个分支上。Master 分支是上线版本，而合并回 Develop 分支的目的，是因为可能在 Release 分支上还会测到并修正一些问题，所以需要跟 Develop 分支同步，免得之后的版本又再度出现同样的问题。

- Develop 分支

- 这个分支主要是所有开发的基础分支，当要新增功能的时候，所有的 Feature 分支都是从这个分支切出去的。而 Feature 分支的功能完成后，也都会合并回来这个分支。

- Feature 分支

- 当要开始新增功能的时候，就是使用 Feature 分支的时候了。Feature 分支都是从 Develop 分支来的，完成之后会再并回 Develop 分支。

使用 Git 一般开发规范

- 运作机制
 - 任何一个 master 分支上的 commit 都是可以被布署的
 - 当有新的功能或修正要作，记得一定要从 master 开新的分支出来
 - Hotfix 绝不允许新功能
 - 不允许在 Master 上做 commit，必须由 release 及 hotfix 用 merge 到 Master
 - Develop 主要由 feature merge 回来的，所以存在一定的 bug，这些 bug 由 feature 持续解决后 merge 回来，因此 原则上 develop 不做 commit。

使用 Git 一般开发规范

- 运作机制（续）
 - Feature 及 release 都是由 develop 分支出来的
 - 当 develop 中版本很稳定时，就分出 release
 - 当 develop 中需要新功能时就分出 feature
 - Release 后会进行测试及必要最后 debug，然后再 merge 回 Develop 及 Master
 - Hotfix 是由 Master 分出来的紧急修护分支，处理完后 merge 回 Master 及 Develop

使用 Git 一般开发规范

- 制度面的问题
 - commit 及 push 规范
 - reverse 及 conflict 规范
 - 决定 Git flow 的分支数，
 - 规范 new feature/release/master/hotfix 的范围
- Permission 规划（参考 gitlab）
 - 角色分配
 - 角色职司

Action	Guest	Reporter	Developer	Maintainer	Owner
Download project	✓ (1)	✓	✓	✓	✓
Leave comments	✓ (1)	✓	✓	✓	✓
View Insights charts ?	✓	✓	✓	✓	✓
View approved/blacklisted licenses ?	✓	✓	✓	✓	✓
View license management reports ?	✓ (1)	✓	✓	✓	✓
View Security reports ?	✓ (1)	✓	✓	✓	✓
View Dependency list ?	✓ (1)	✓	✓	✓	✓

使用 Git 一般开发规范

- 编码系统
 - Folder
 - Code
 - Document
- 批注标准
 - Commit 批注
 - Tag 批注
- 减少上线后的痛楚
 - 做过练习，该准备的制度讲好了？再上
 - 较频繁的 merge，少些 conflict（即多沟通）

使用 Git 一般开发规范

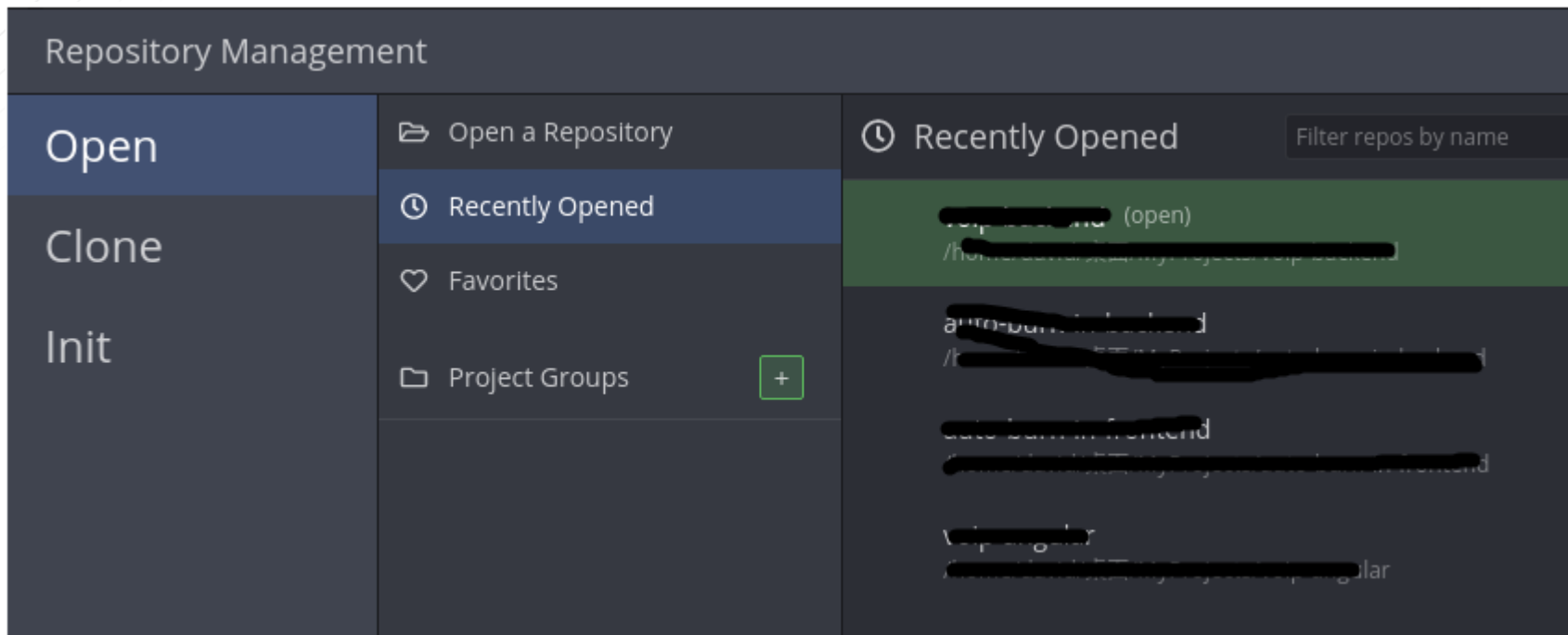
- 上线后的经验
 - feature branch 不要太长寿，最多最多，不要超过一个 scrum sprint (约 30 days)
 - 万一内容太多，就回头跟 PO 商量，这个需求太大了，必须修剪
 - 质量好的自动测试可以消除你对频繁回主线的疑虑
 - 进版前一定要开 release，预产线的测试版本一定要是上线的同一个版本
 - 承上，改一行批注也不行！
 - hotfix 跟 release 开启的同时就要决定版本号，这能帮你确认版本
 - 经常存在的 branch 只有两个：develop 与 master
 - Commit 可能以 file 来做，也可能以一个 Folder 或
 - 数个 Folder 来做，如何取舍，看分工的重迭程度。

git 可视化工具

- 推荐几个
 - GitKraken (windows & mac & linux) (用免费版即可)
 - 官网: <https://www.gitkraken.com/>
 - TortoiseGit (windows)
 - 官网: <https://tortoisegit.org/>
 - SourceTree (windows & mac)
 - 官网: <https://www.sourcetreeapp.com/>
 - 更多参考:
 - <https://git-scm.com/downloads/guis>

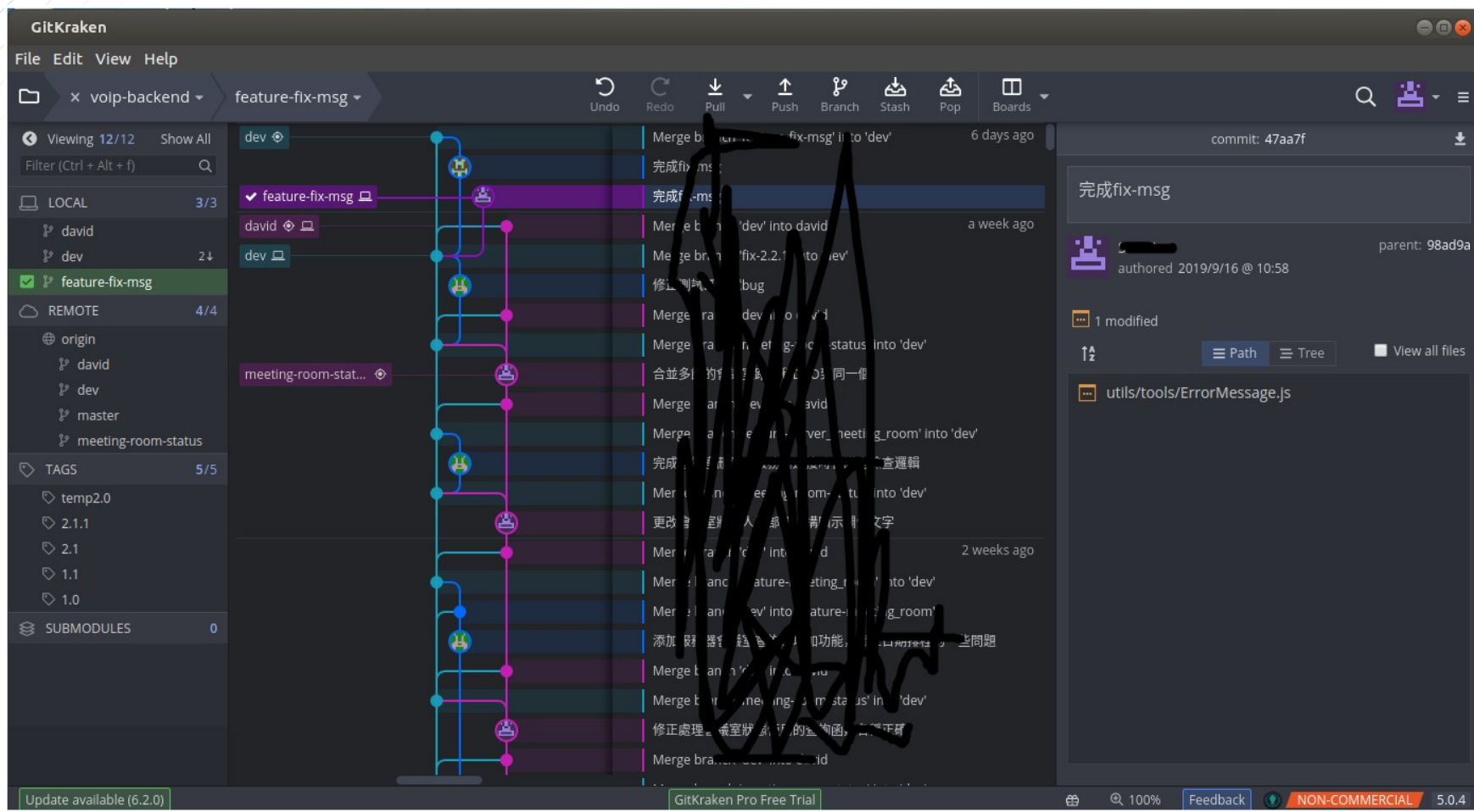
git 可视化工具—GitKraken 截图

- 关联 git 仓库（本地已有、远程克隆、新建）



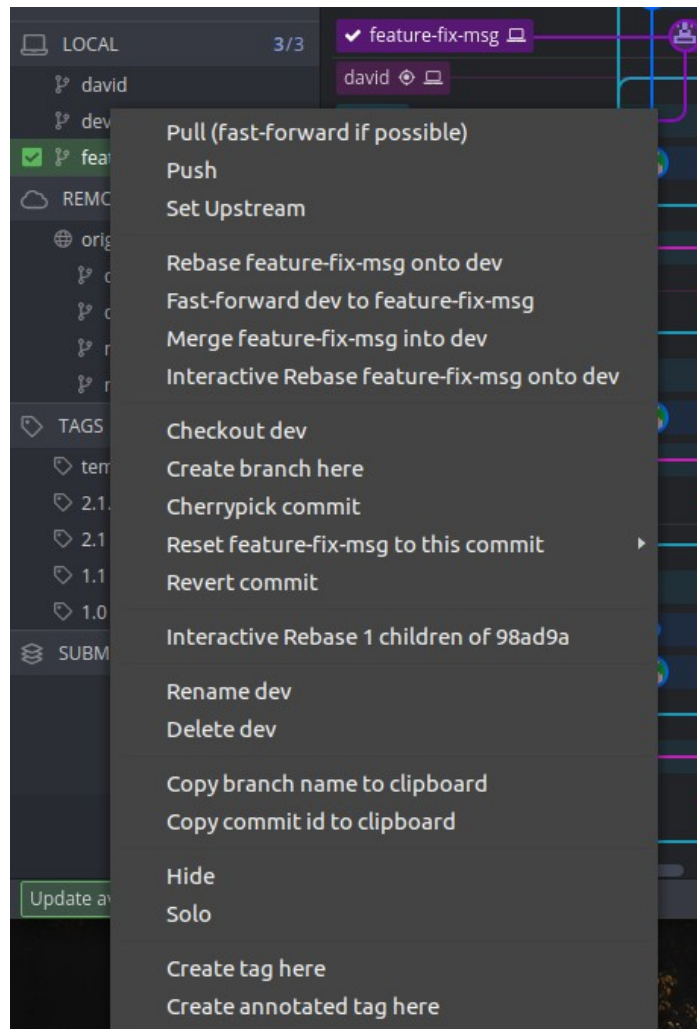
git 可视化工具—GitKraken 截图

- 本地 git 管理主页面



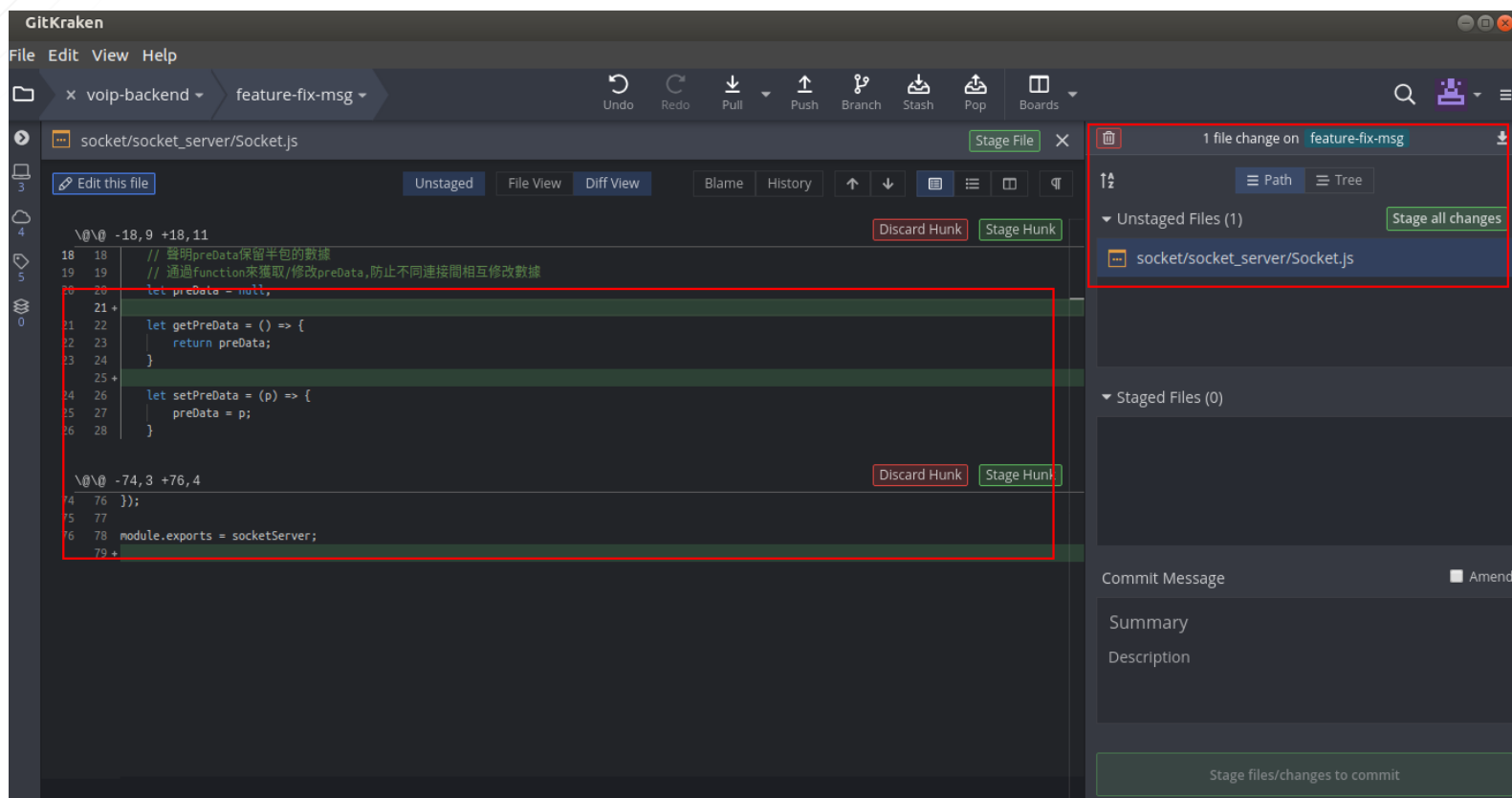
git 可视化工具—GitKraken 截图

- 本地分支管理
 - 右键直接点击对应功能



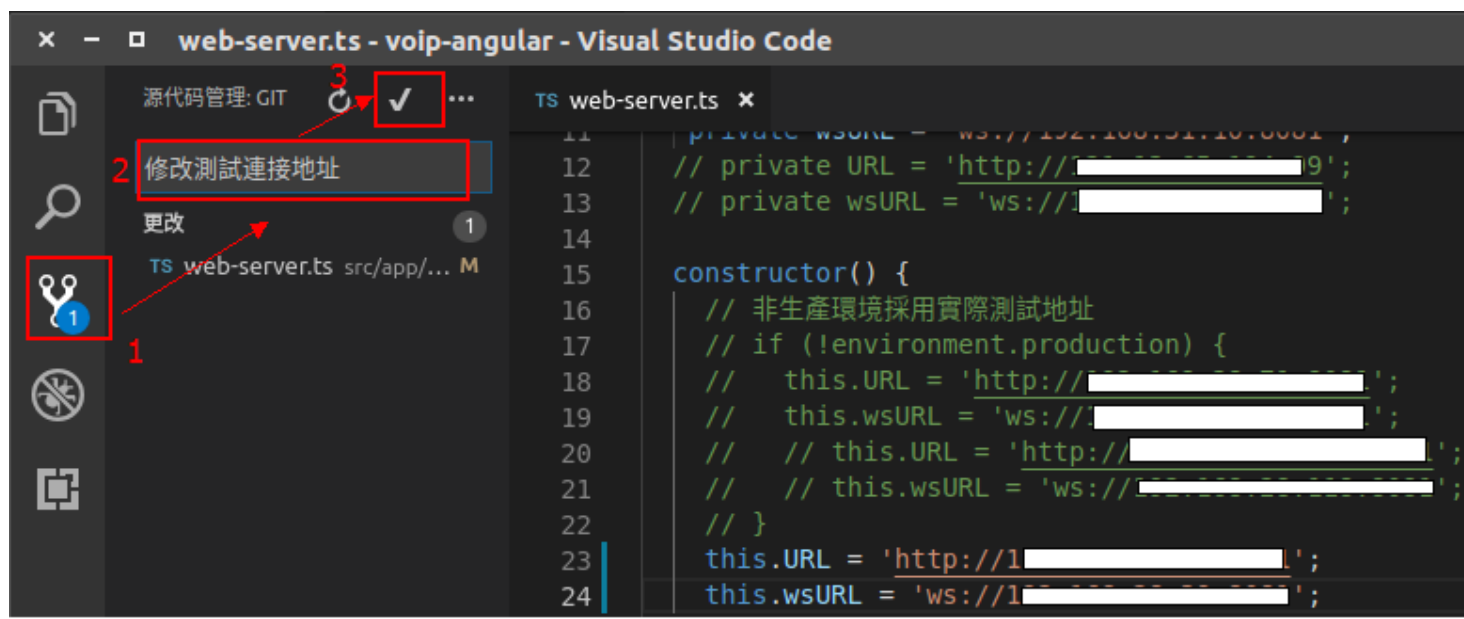
git 可视化工具—GitKraken 截图

- 代码异动实时检测，显示变化，可视化异动信息添加



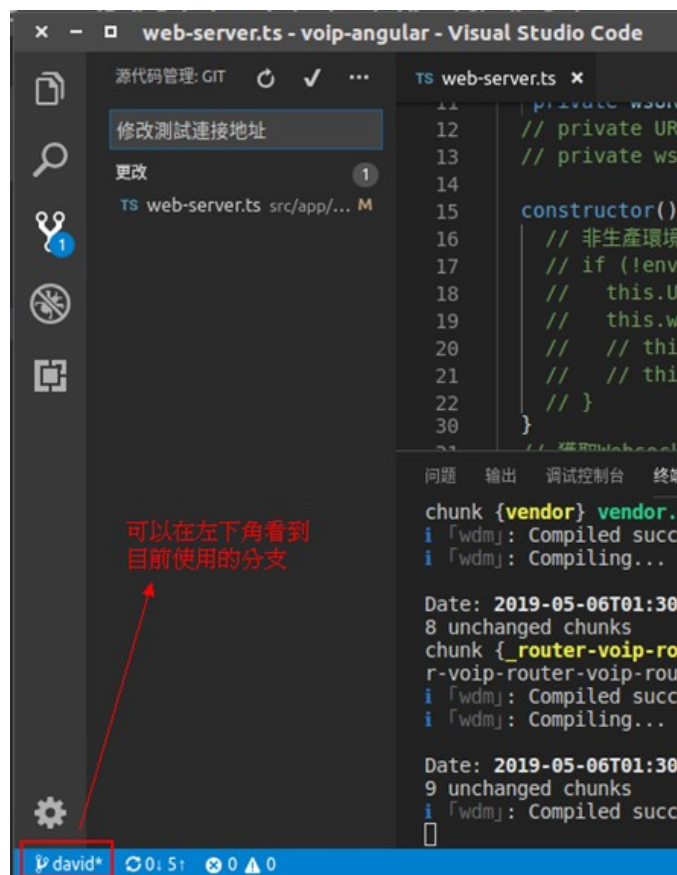
在 VS Code 中使用

- 现在大部分的开发工具都支持 Git 的集成，以 VS code 为例
- 在 VS Code 中使用 Git（假设 Git 已安装）
 - 1 提交修改内容到本地仓库
 - 因为 clone 的项目已有 git 相关仓库
 - 点击源代码管理图标，添加 commit 内容，点击勾符号，完成 git 的 commit。



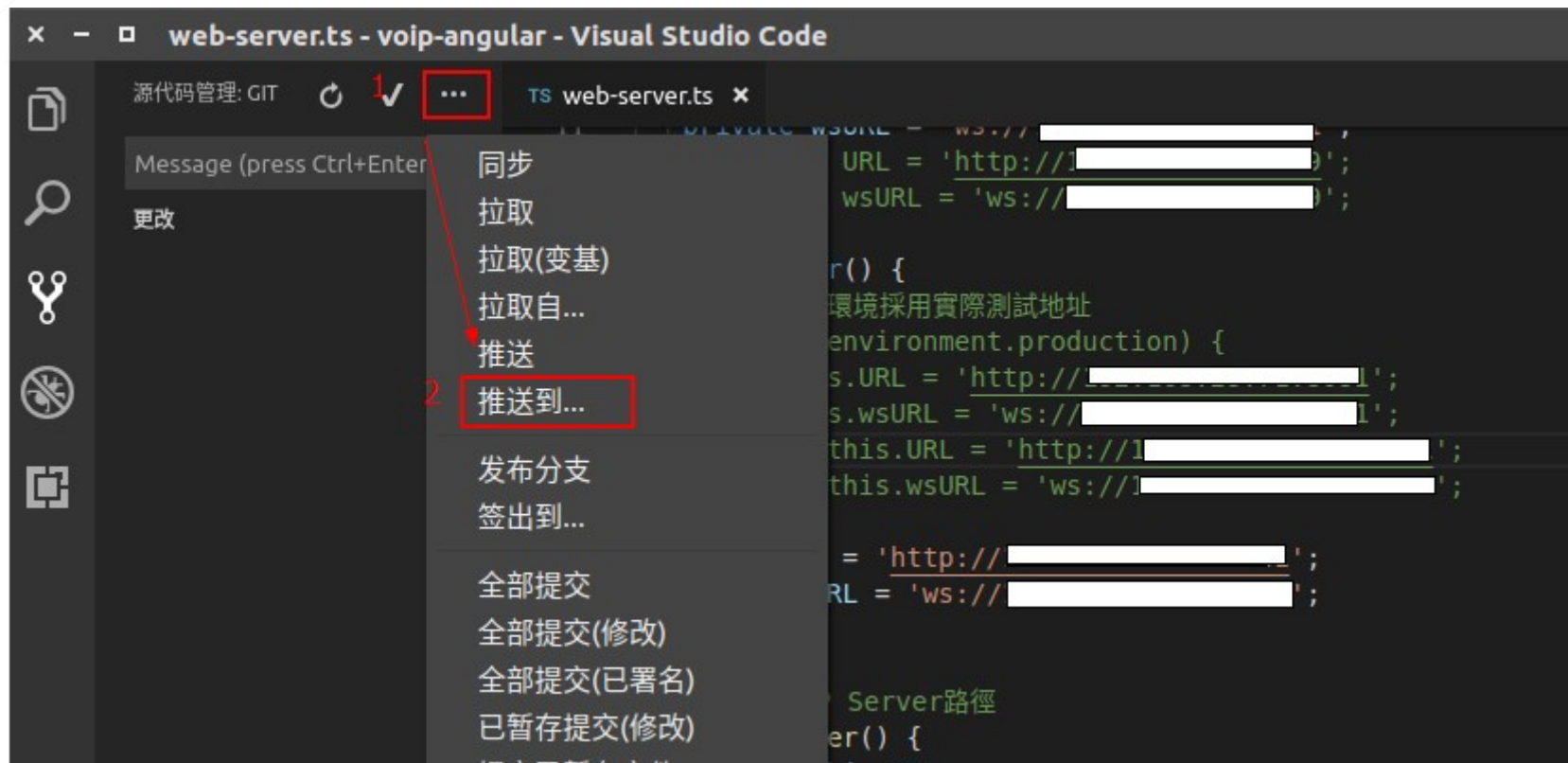
在 VS Code 中使用

- VSCode 左下角可以查看到当前使用的本地 git 分支



在 VS Code 中使用

- 2 push 到远程分支
 - 源代码管理 -> 省略号 -> 推送到 ->



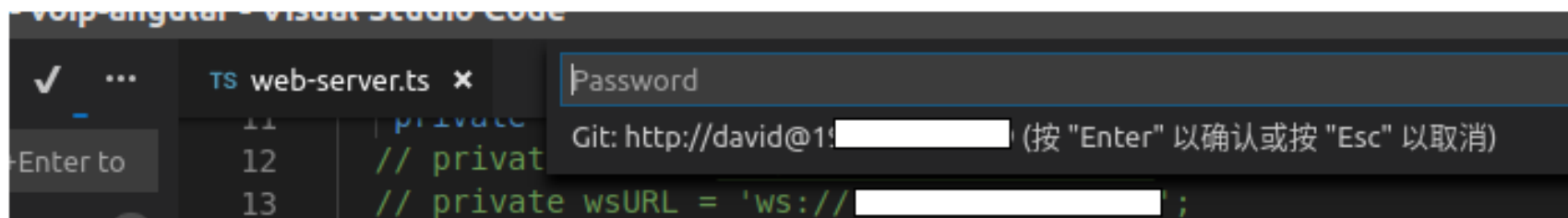
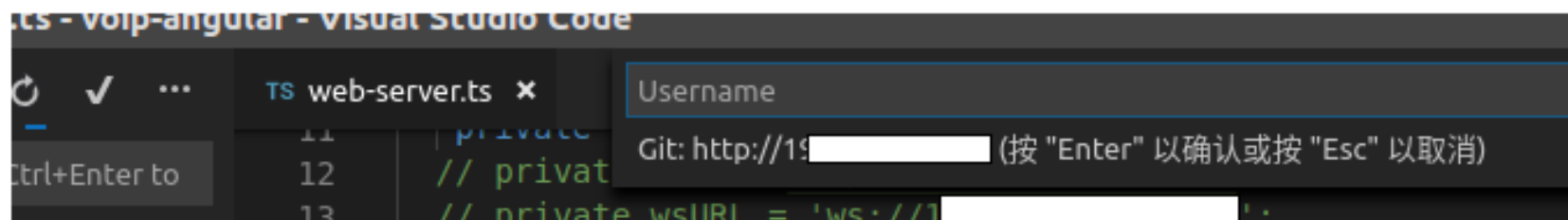
在 VS Code 中使用

- 选择需要推送到远程的地址



在 VS Code 中使用

- 输入账号密码
 - 注意：默认推送的分支名和本地分支名一致
 - 如果未报错，就 push 成功了

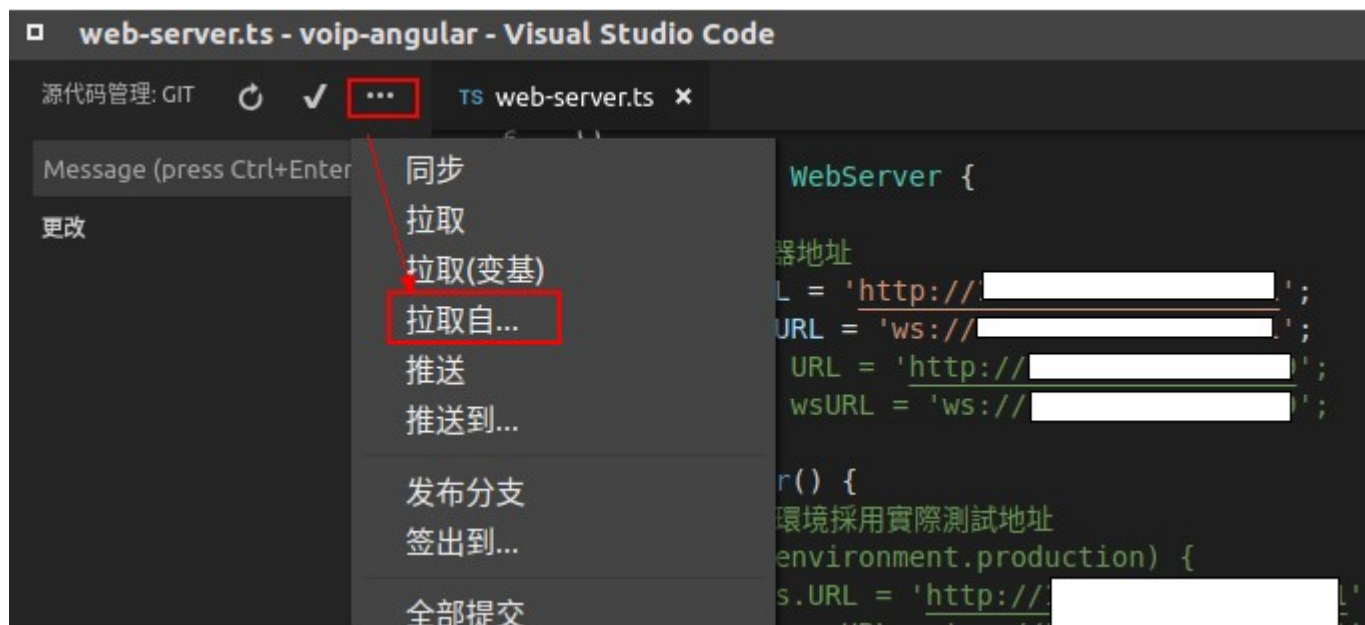


在 VS Code 中使用

- 3 远程合并分支到分支
 - 此部分在操作远程分支，和直接使用指令情况是一样的，不再重复。

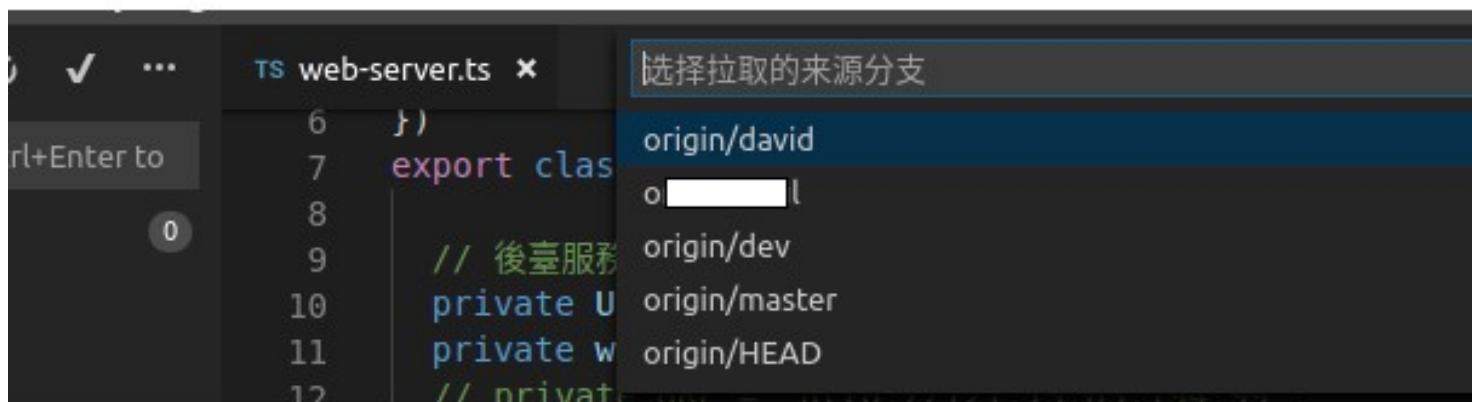
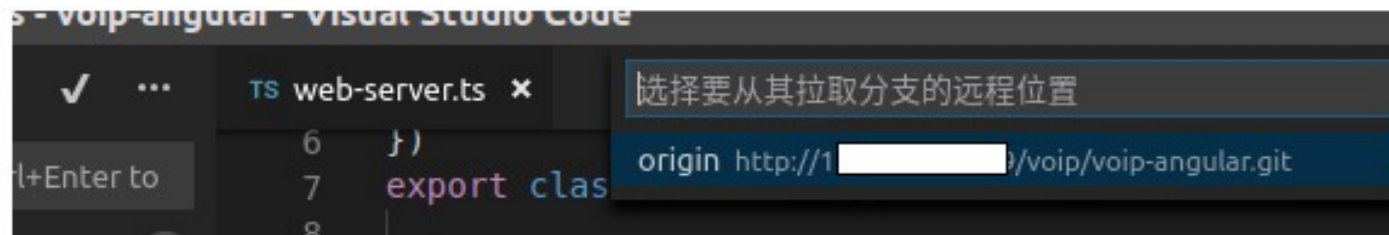
在 VS Code 中使用

- 4 pull 远程分支到本地分支
 - 源代码管理 -> 省略号 -> 拉取自... ->



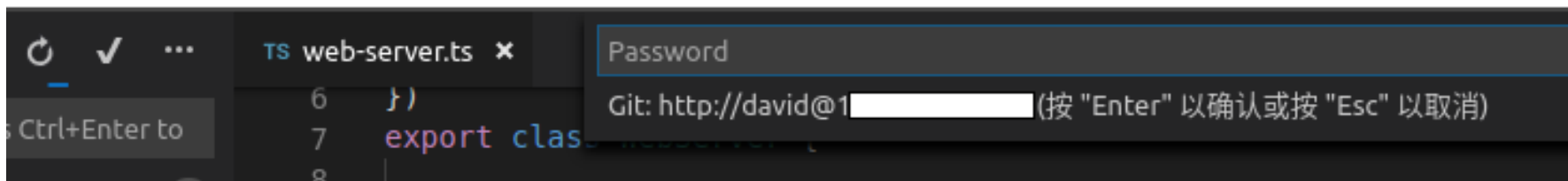
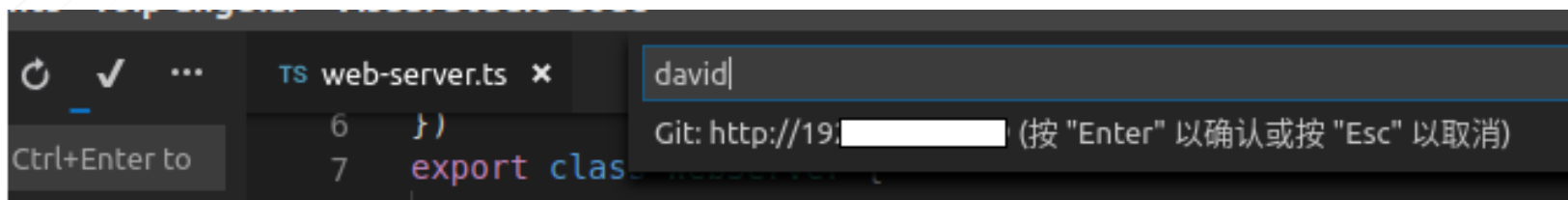
在 VS Code 中使用

- 选择要拉取 (pull) 的远程分支地址和分支名称



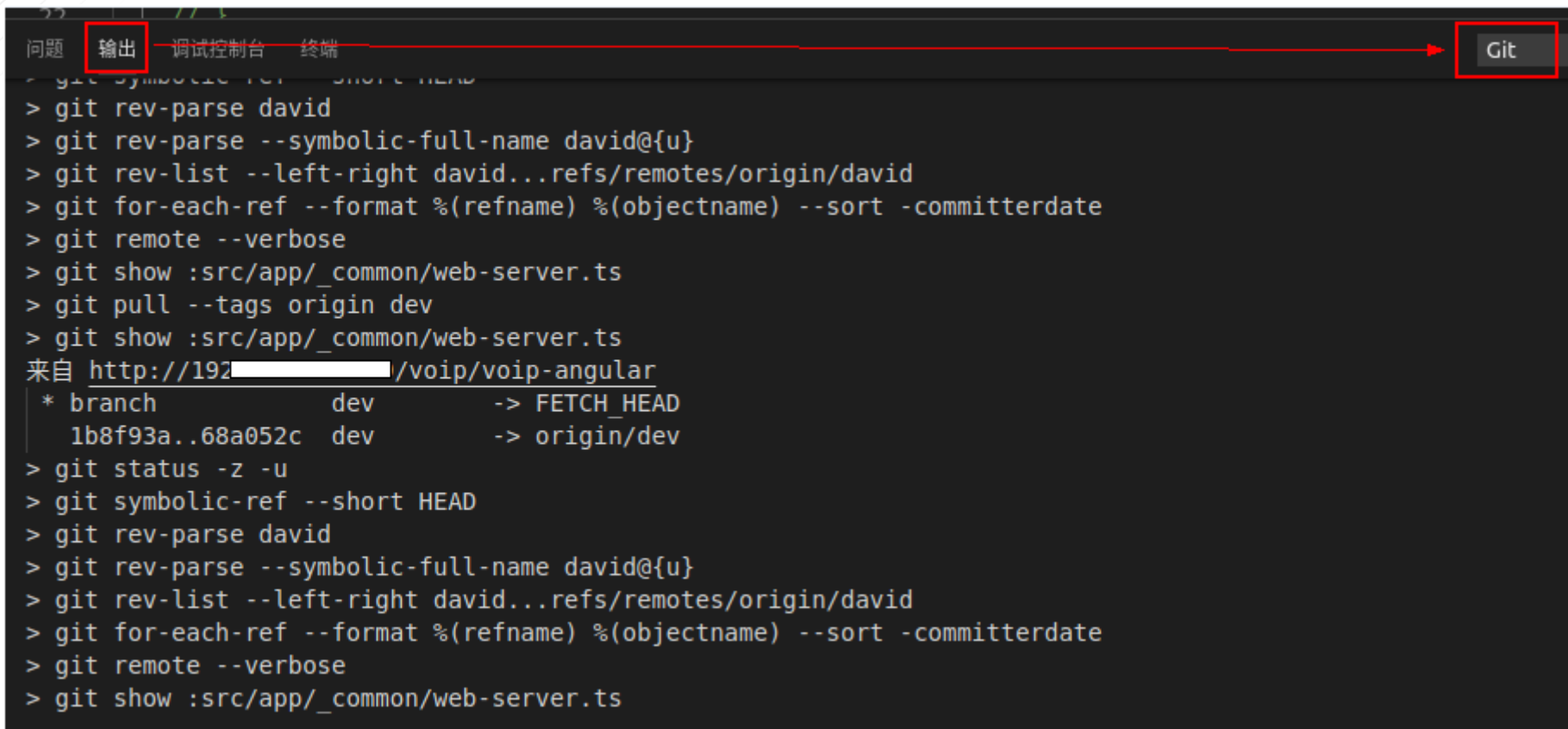
在 VS Code 中使用

- 输入账号和密码
- 没有出现错误信息，则 pull 成功



在 VS Code 中使用

- 在 VS Code 的 git 输出窗口，可以看到之前的操作直接使用 git 指令会是哪些：



The screenshot shows the VS Code interface with the 'Git' output window active. The '输出' (Output) tab is selected, and the 'Git' panel is visible on the right. The output window displays a series of git commands and their results, including repository status, remote information, and file content.

```
> git symbolic-ref --short HEAD
> git rev-parse david
> git rev-parse --symbolic-full-name david@{u}
> git rev-list --left-right david...refs/remotes/origin/david
> git for-each-ref --format %(refname) %(objectname) --sort -committerdate
> git remote --verbose
> git show :src/app/_common/web-server.ts
> git pull --tags origin dev
> git show :src/app/_common/web-server.ts
来自 http://192.168.1.100/voip/voip-angular
* branch          dev          -> FETCH_HEAD
  1b8f93a..68a052c dev          -> origin/dev
> git status -z -u
> git symbolic-ref --short HEAD
> git rev-parse david
> git rev-parse --symbolic-full-name david@{u}
> git rev-list --left-right david...refs/remotes/origin/david
> git for-each-ref --format %(refname) %(objectname) --sort -committerdate
> git remote --verbose
> git show :src/app/_common/web-server.ts
```

总结

主要了解 git 是什么，怎么用，规范化的问题。