GIT 速查表（可用 GitHub 桌面版代替）

VX：UFO1692129467

- Git:开放源码的跨平台分布式版本控制系统（VCS）.

  ：跟踪、存储、恢复版本、历史版本、以及团队协作的分支。

  ：对服务器的依赖程度较小，速度快、成本低、占用空间小。

- CMD 命令窗口：
- 检查安装是否成功
  - $ git version
  - $ git help
- 初次运行 Git 前的配置

  配置用户信息:

  git config --global user.name “用户名”

  git config --global user.email “邮箱”

  git config --global core.editor vim

检查配置信息

  git config --global --list

  git config user.name

用户配置命令别名

  $ git config --global alias.别名 “命令全称"

  Eq: $ git config --global alias.st “status"。git st

查看用户已配置的别名信息

  $ git config —global —list

删除用户配置别名方法

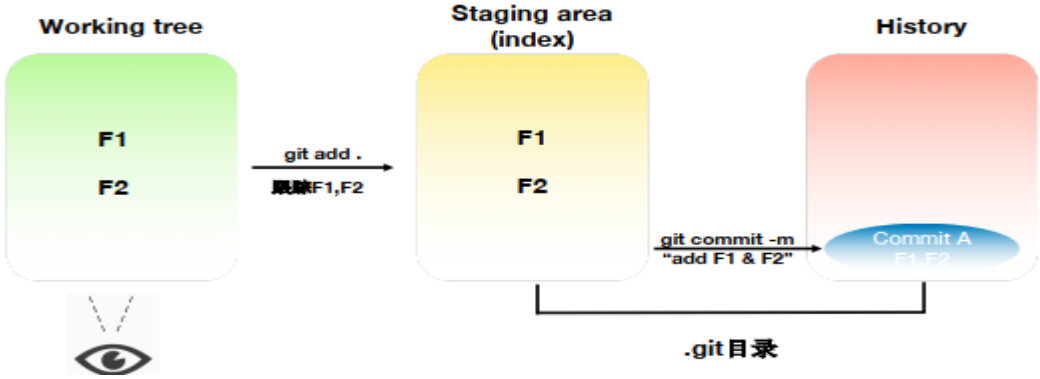
  • 修改配置文件

  $ git config --global —edit

  • 修改配置文件

  $ git config --global --unset alias.别名

- **Git的三个区域**



- 对某文件夹<<右键<<git bush here

- 显示所有的提交
  $ git log
  $ git log --all --graph --oneline
  $ git config --global alias.graph "log --all --graph --oneline"
  显示特定文件随时间的变化
  $ git log -p <file>
  • 谁在< file >中更改了内容和时间?
  $ git blame <file>
- Git diff :用来描述工作区、暂存区、版本区之间的差别，代码，下面是图形可视化。
  Git diff-tool：Helix Visual Merge Tool（P4Merge）是一种三向合并和并排文件比较工具。可以**可视化**您的合并，获取全面的文件历史记录并比较各种图像文件。
  - 安装合并工具：p4merge
  • https://www.perforce.com/
  • 点击 Downloads
  • 找到 Helix Visual Merge Tool，并点击链接
  • 选择操作系统，并下载安装
  Git diff-tool
  - 配置 diff-tool
  - $ git config —global diff.tool p4merge
  - $ git config —global difftool.p4merge.path
  /Applications/p4merge.app/Contents/MacOS/p4merge（这是安装目录）
  - $ git config —global difftool.prompt false
  - $ git config —global —list
- Git init:初始化
- 跟踪文件到暂存区：git add 【文件名.后缀】
- 提交到历史区：git commit -m【文件名.后缀】: 尽量“xxx“添加说明之后再进行提交。
  Git diff ： 暂存区与工作区的区别。
  Git diff --staged: 暂存区与历史版本区的区别。
  Git status :查看当前的三个区域的状态。
  Git rm【文件名.后缀】:完全删除放到历史区的文件
  Git checkout 【文件名.后缀】:抛弃暂存区的修改。
  Git reset HEAD 【文件名.后缀】:将文件从暂存区中删除。
  Git checkout [commit 参数] 【文件名.后缀】:将文件回滚到某一个版本上，需要知道的是 commit 的参数，一般先用 git graph 查找。会出现分离指针头情况。
  或者：1.git reset --hard HEAD^，回滚到上个版本
  2.git reset --hard HEAD^~2，回滚到前两个版本
- 修改最近的提交
- 将提交后修改的内容提交到最近的提交中
  $ git commit —amend
- 显示特定文件随时间的变化
  $ git log -p <file>
- 谁在< file >中更改了内容和时间?
  $ git blame <file>
- 忽略文件

- 忽略没有被跟踪的文件
    - 创建.gitignore 文件
    - 将文件名按照规则写入.gitignore 文件：eq:　*.bug　　dev/
    - 忽略已经被跟踪的文件
    - 创建.gitignore 文件
    - git rm --cached -- a.out
    - 将文件名按照规则写入.gitignore 文件
- 可下载相应的.gitignore 文件

- 分支操作:

  Eq:主线 master ,分支 develop.

  HEAD 指向当前分支，而不是提交

  $ Git branch :结果中的*表示的是当前分支所在。

  新建分支

  - $ git branch <branch name>
  - $ git branch /$ graph 查看分支
  - 切换分支
  - $ git checkout <branch name>
  - 新建并切换到分支
  - $ git checkout -b <branch name>

  分支合并

  - Fast Forward 合并
  - 切换到 mater 分支
  - $ git diff .. <branch name>
  - $ git merge <branch name>

  Git graph

- Git 先进行一个合并提交
  - 然后 Git 再与另一个分支合并
  - 称之为“3-way merge”
  - 合并时，只需:

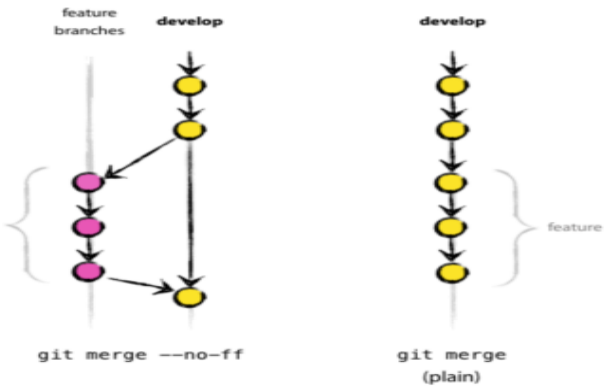
  $ git merge <branch name>

- **合并的冲突处理**
  - 手动处理
  - 打开冲突的文件
  - 删除冲突显示标示
  - 确认文件内容
  - 提交修改后的文件
  - 使用配置的合并工具解决冲突
  - **$ git mergetool**
- Git merge-tool
- - **配置 merge-tool**
- - $ git config —global merge.tool p4merge
- - $ git config —global mergetool.p4merge.path　粘贴复制之前，重新填写--global
- /Applications/p4merge.app/Contents/MacOS/p4merge　　地址不是这个地址，
- - $ git config —global mergetool.prompt false

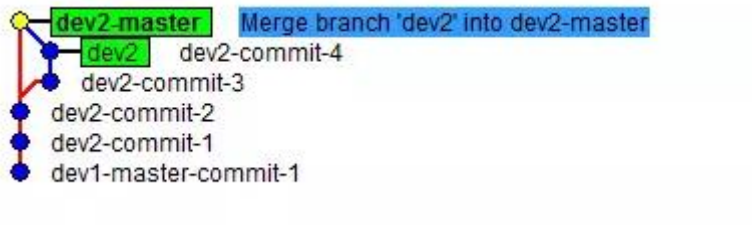
- $ git config --global mergetool.keepBackup false
- $ git config —global —list

关于分支的合并：

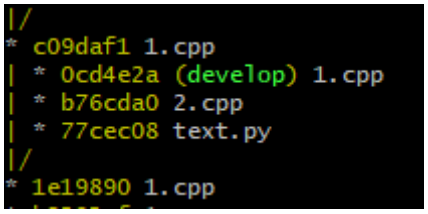
- git merge –no-ff 可以保存你之前的分支历史。能够更好的查看 merge 历史，以及 branch 状态。
  git merge 则不会显示分支，只保留单条分支记录：默认就是-ff。即 fast-forward
- 不进行 fast-forward 合并
  - $ Git merge --no-ff



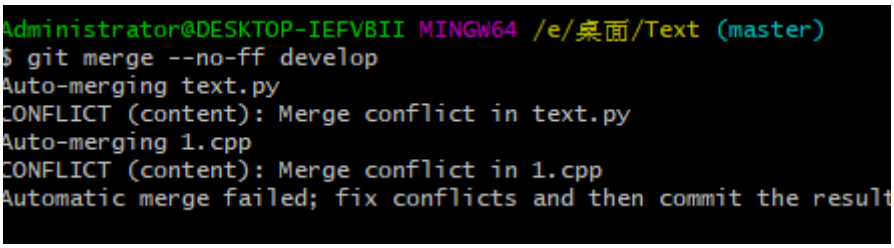
（参考：https://blog.csdn.net/qq_40999917/article/details/103316870）



一点定要会看图，git graph:dev2 一定要与 master 合口， 下图 develop 就没合口



一顿 ADD、commit 之后若出现：



说明冲突，则调用 git mergetool.选择最终版本

- **分离指针头**：当回滚到某版本时（git checkout [commit ID]）时，出现了 HEAD 分离，不再与分支（eq:master）（正常的话每当提交的时候，HEAD 跟随着分支一起移动）。同时前进，而出现，当在分离指针头工作时，通过提交相关的 add、commit 时，

返回 master 会造成文件不同，分离出来的修改可能随时间的变化会被 GIT 删掉，所以如下操作：（参考：https://blog.csdn.net/start_mao/article/details/94722393）

- 分离指针头(Detached HEAD)状态
  - 处理方法
  - 检出 master 分支

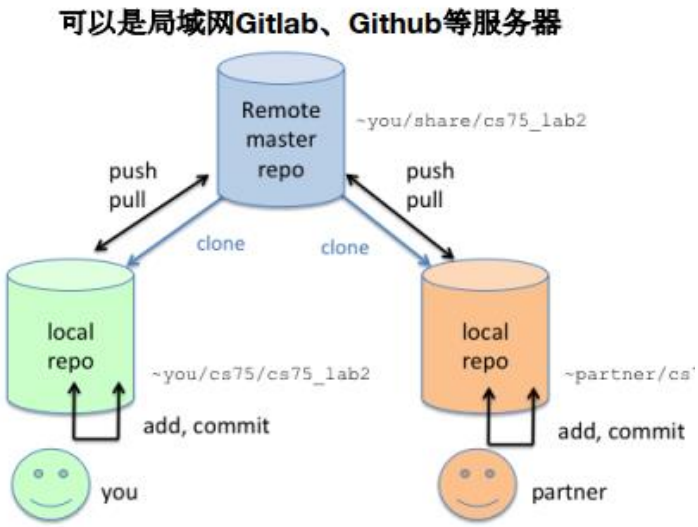
  $ git checkout master
  - 创建 stage branch

  $ git branch stage

  $ git checkout stage
- 保存当前修改不提交，切换到另一分支
  - $ git stash
  - $ git stash list
  - $ git stash list -p
  - $ git stash apply
  - $ git diff
  - $ git add
  - $ git commit
  - $ git stash pop

GITHUB

- Git 的远程仓库：Gitlab 或者 GitHub：私有、开源的托管平台



- 代码仓库（云端存储代码）、版本管理、查找源码。
- **Git 远程仓库到本地**
- - 将远程仓库克隆到本地

  $ git clone <git address>
- - 远程仓库的相关命令

  获得远程仓库信息

  $ git remote
- 获得详细远程仓库信息

$ git remote -v

- 查看远程仓库分支

  $ git branch -r

- Git 远程仓库到本地

  将远程仓库最新内容拉到本地

  $ git fetch <origin> (可在该命令前后分别用$ git status / $ graph)

- 合并远程仓库的分支

  $ git merge <origin/master>

- 将远程仓库最新内容拉到本地，并合并远程仓库的分支

  $ git pull = $ git fetch + $ git merge

  $ git fetch + $ git merge 更好一些

  Git clone

  - 当使用 clone with ssh 时，需要将 ssh 的公钥加入到 Github 中
    - $ ssh-keygen
    - 在 Github 中增加 id_rsa.pub 中的内容

- **Git 本地到远程仓库**

- - 本地改动并提交

  $ git add . and $ git commit

  $ git fetch origin(多人合作项目) and $ git status/ $ graph

- - 将本地改动推到远程仓库

  $ git push origin master

- - 确认用户名和邮箱，否则需要输入 Github 用户名和密码

  $ git config --local user.name "username at github"

  $ git config --local user.email "email at GitHub"

  （需要确认邮箱, —local 指仅将更改当前仓库下的用户名和邮箱）

- **Git 在远程仓库创建分支**

- - 在本地创建分支 branch-2

- - 本地修改并提交

  - $ git push origin branch-2 命令将此分支推到远程仓库的分支 branch-2
  - $ git branch -a / $git graph

- 增加远程仓库地址

  - $ git remote add upstream <remote name>
  - $ git remote -v

- • 获取 upstream 远程仓库信息

  - $ git fetch upstream

- • 同步到 upstream 远程仓库信息，并推送到自己的远程仓库

  - $ git merge upsteam/master
  - $ git push origin master

- • 将自己的修改提交到 upstream

  - pull request

- 增加远程仓库地址

  - $ git remote add upstream <remote name>
  - $ git remote -v

- • 获取新加的远程仓库信息

- $ git fetch upstream

- • 同步到新加的远程仓库信息，并推送到自己的远程仓库
  - $ git merge upsteam/master
  - $ git push origin master
- • 删除远程仓库地址
  - $ git remote remove <remote name>
  - $ git remote -v
- 使用 git config --show-origin --get credential.helper 可以找出你自己电脑的提交代码时使用的账号策略：.gitconfig 文件的位置，不一定和账号密码的是同一个 gitconfig 文件
  - • 然后你需要在.gitconfig 中修改这一行 ： [credential] helper = osxkeychain。
  - • 更改为 helper = store
  - • 保存文件
- Git 标签
  - 标签（tag）是 git 版本库的一个标记，指向某个 commit 的指针。
  - 如果你达到一个重要的阶段，并希望永远记住那个特别的提交快照，你可以使用 git tag 给它打上标签
  - 标签主要用于发布版本的管理，一个版本发布之后，我们可以为 git 打上 v.1.0.1 v.1.0.2 ...这样的标签。
- Git 创建标签
- - 切换到要创建标签的分支
  $ git checkout <branch-name>
- - 创建标签
  - • 创建轻量级的（lightweight）标签
  $ git tag <tag-name>
- • 创建含附注的（annotated）标签
  $ git tag -a <tag-name> -m “tag message”
- • 附注标签，实际上是存储在仓库中的一个独立对象，它有自身的校验和信息，包含着
  标签的名字，电子邮件地址和日期，以及标签说明，标签本身也允许使用 GNU Privacy Guard (GPG) 来签署或验证。建议使用含附注型的标签，以便保留相关信息。临时性加注标签，或者不需要旁注额外信息，则使用轻量级标签
- 显示标签
  - • $ git tag
  - • $ git tat -l <tag-name>
  - • $ git show <tag-name>
- - 将标签推送到远程仓库
- • 推送一个标签
  $ git push origin <tag-name>
- • 推送所有标签
  $ git push origin —tags
  $ git push —tags
- 删除标签
  - • 删除本地标签

$ git tag -d <tag-name>
$ git tag -delete <tag-name>
$ git tag -d <tag-name1> <tag-name2> …

- • 删除远程仓库的标签
  $ git push origin -d <tag-name>
  $ git push origin -delete <tag-name>
  $ git push origin -d <tag-name1> <tag-name2>…
- 检出标签
  - 无法检出标签
- - 可以从标签出创建一个分支，并检出分支
  $ git checkout -b <branch-name> <tag-name>
- • 在历史 commit 出创建标签
  $ git tag <tag-name> <reference of commit>

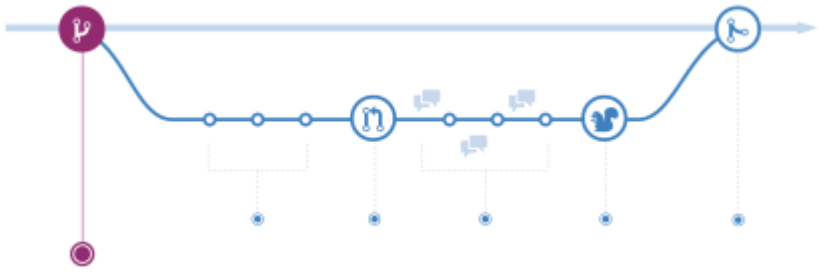
Git 工作流

- Git 满足整个研发过程中（开发、测试、发布、维护），代码的版本控制。
- 这就需要我们在做分支管理的时候，去决策需要使用多少分支，每个分支分别需要做什么事情，以及什么时候创建/合并分支
- Git flow:

- **Git Flow**
  - 最早诞生、并得到广泛采用的一种工作流程，使用长期分支、临时分支和标签来进行工作
  - 长期分支
    - 主分支（master branch），发布代码
    - 开发分支（develop branch），功能代码
  - 临时分支
    - 功能分支（feature branch）
    - 预发分支（release branch）
    - 补丁分支（hotfix branch）

- 功能分支（feature branch）
  - 为开发新功能或者修改非紧急的 bug 而新建的分支 始于开发分支
  - 合并到开发分支，并删除该功能分支
  命名原则：除了 master, develop, release*,hotfix*等的其它任何名称
- 预发分支（release branch）
  - 为发布版本新版本而建立的分支，方便在发布前修改版本中的 bug
  - 始于开发分支
  - 合并到开发分支和主分支，并删除该功能分支
  - 命名原则：release*

- 补丁分支（hotfix branch）
  - 为修复先前发布版本的小 bug 而建立的分支
  - 始于开发分支
  - 合并到开发分支和主分支，并删除该功能分支
  - 命名原则：hotfix*
- - 标签(Tags)
  - 用于表示已发布的版本
- 优点：清晰可控
  - 缺点：
  - 相对复杂，需要同时维护两个长期分支
  - 大多数工具都将 master 分支当作默认分支，可是开发是在 develop 分支进行的，这导致经常要切换分支，非常烦人
  - 这个模式是基于"版本发布"的，目标是一段时间以后产出一个新版本。但是，很多网站项目是"持续发布"，代码一有变动，就部署一次。这时，master 分支和 develop 分支的差别不大，没必要维护两个长期分支
- GitHub flow：适合持续发布的版本控制。



只有一个长期分支 master，使用起来相对简单。
- 流程
- 根据需求，从 master 拉出新分支，不区分功能分支或补丁分支。
- 新分支开发完成后，或者需要讨论的时候，就向 master 发起一个 pull request（简称 PR）
- Pull Request 既是一个通知，让别人注意到你的请求，又是一种对话机制，大家一起评审和讨论你的代码。对话过程中，你还可以不断提交代码。
- 评审通过后，合并进 master，重新部署后，删除拉出的分支
- 优点：简单，对于"持续发布"的产品，可以说是最合适的流程
- 问题在于它的假设：master 分支的更新与产品的发布是一致的。也就是说，master 分支的最新代码，默认就是当前的线上代码。可是，有些时候并非如此，代码合并进入 master 分支，并不代表它就能立刻发布。此时，master 分支就会与刚发布的版本不一致，需要新建 production 分支来跟踪发布的版本。

Git 常用的命令：

- 创建
  - 克隆现有的存储库

git clone https://github.com/xfsundlmu/test2.git
  - 创建新的本地存储库

$ git init

- 本地变化

• 查看工作目录中的文件状态

$ git status

- 比较工作目录中跟踪文件和暂存区文件的差别

$ git diff

• 比较暂存区文件和提交版本文件的差别

$ git diff —staged

• 将所有当前更改添加到下一次提交

$ git add .

• 提交更改

$ git commit -m “commit message”

• 抛弃更改

$ git checkout .

- 更改最后提交，但不修改发布的提交

$ git commit —amend -m “commit message”

- 提交历史

• 显示所有提交，从最新开始

$ git log

• 显示特定文件随时间的变化

$ git log -p <file>

• 谁在< file >中更改了内容和时间?

$ git blame <file>

- 分支和标签

• 列出所有现有分支

$ git branch -av

• 创建分支

$ git branch <new-branch>

• 切换分支

$ git checkout <branch-name>

• 创建并分支

$ git checkout -b <branch-name>

- 合并分支

$ git merge <branch>

$ git merge <branch> —no-ff

• 删除本地分支

$ git branch -d <branch-name>

• 提交标签

$ git tag <tag-name>

- 更新和发布

• 列出所有当前配置的远程主机

$ git remote -v

- 显示有关远程

$ git remote show <remote>

• 从< Remote >下载所有更改，但不要集成到 Head 中

$ git fetch <remote>

• 合并远程分支

$ git merge <remote> <branch>

• 下载更改并直接合并远程分支

$ git pull <remote> <branch>

• 在远程上发布本地更改

$ git push <remote> <branch>

- 删除远程上的分支

  $ git branch -dr <remote/branch>

  • 发布标签

  $ git push —tags

Git 建议

- 修复两个不同的 bug 应该产生两个单独的提交。如果出了什么问题就可以很方便地把它们退回去。
- 经常提交使您的承诺保持较小，并且再次帮助您仅提交相关的更改。
- 只在相应功能完成时提交代码
- 在提交之前测试代码

  - 提交前通过测试尽可能保证代码的功能和质量

  • 编写良好的提交消息

  - 提交消息应尽可能简洁地表明提交的动机、与原版本的不同等

  • 版本控制不是备份系统

  - 重要的版本还是要备份
- 尽可能使用分支

  - 分支是 Git 最强大的特性之一

  - 分支是帮助你避免混淆不同发展方向的完美工具. 您应该在开发工作流程中广泛使

  用分支：用于新特性、bug 修复、想法等

  • 就工作流达成一致

  - git 允许您从许多不同的工作流中选择：长时间运行的分支、主题 bran-ch、合并或

  重基、git-flow…。您选择哪一个取决于以下几个因素：你的项目，你的整体开发和部署工作流程，（也许最重要的）是你和你的队友的个人喜好。无论你选择工作，只要确保达成一个共同的工作流程，每个人都遵循这个工作流程。

  • 使用帮助和文档

  - $ git help <command>