

Git Tricks

Git 介绍

Git 是什么

- Git 是目前世界上最先进的分布式代码版本控制系统
- Git 由 Linux 之父 Linus Torvalds 于2005年为了帮助Linux内核代码管理而创造，第一个可用版本仅花了10天时间
 - 起因是另一个商用的代码版本控制软件BitKeeper由于被Linux社区的大佬尝试破解而终止了与Linux的合作
- Git 这个单词的意思是蠢货
 - Linus 对此的解释:
 - I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'Git'.
 - 我是个任性的杂种，我把所有我做的项目以我自己命名。先是'Linux'，这次是'Git'.
 - 类似乔布斯的 "Stay hungry. Stay foolish."
- Github是一个著名的基于Git的代码托管网站，可以理解为代码专用云盘，几乎所有的著名开源项目都在Github上进行代码托管、版本管理、问题提交、Bug修复等，比如：
 - Linux - 著名的开源操作系统，Ubuntu、Debian、麒麟、安卓都可以看成是Linux的分支
 - PyTorch - 著名的深度学习库
 - OpenCV - 著名的图像处理库
 - PCL - 著名的点云处理库
 - ROS - 著名的机器人操作系统
 - eCAL - 我们公司目前使用的消息中间件
 - ...
- Git (分布式) VS SVN (集中式)

Git (分布式)	SVN (集中式)
分布式，支持离线操作	集中式，需要联网操作
Git 命令与概念复杂，上手困难	SVN 上手相对容易
Git 对分支的支持更完善	SVN 对分支的支持比较简陋

- Git 学习相关资源
 - [Git官网](#)
 - [廖雪峰Git教程](#)
 - [看完这篇还不会用Git, 那我就哭了!](#)
 - [通过游戏理解Git](#)

- Git VSCode 插件
 - Git Graph: 一个可视化Git分支的工具

Git 能做什么


- 分布式代码托管
 - 本地库
 - 工作笔记本 Ubuntu 系统
 - 第二台工作笔记本 Windows 系统
 - TX2 测试平台 Ubuntu Arm版
 - Xavier 测试平台 Ubuntu Arm版
 - ...
 - 远程库
 - Gitlab
 - Github
 - Gitee
 - ...
 - 防止代码遗失、电脑损坏
- 代码历史记录
 - 增量式代码修改记录
 - 每一行的代码修改对于Git来说就是一次旧代码的删除操作与新代码的增加操作
 - 删除操作与增加操作都会保存在 Git 仓库中

```

147 163 pub_msg_radarStopListPP->mutable_time_stamp()-set_nanos(GetNowNanos());
148 - aiforce_pp_msg::radarStopListPP_radarObstacle* radar_obstacle_1 =
pub_msg_radarStopListPP->add_radarobstaclelist();
149 - radar_obstacle_1->set_sensor_id(1);
150 - radar_obstacle_1->set_x(1.0);
151 - radar_obstacle_1->set_z(1.1);
152 - aiforce_pp_msg::radarStopListPP_radarObstacle* radar_obstacle_2 =
pub_msg_radarStopListPP->add_radarobstaclelist();
153 - radar_obstacle_2->set_sensor_id(2);
154 - radar_obstacle_2->set_x(2.0);
155 - radar_obstacle_2->set_z(2.1);
164 + //aiforce::pp_msg::radarStopListPP_radarObstacle* radar_obstacle_1 =
pub_msg_radarStopListPP->add_radarobstaclelist();
165 + //radar_obstacle_1->set_sensor_id(1);
166 + //radar_obstacle_1->set_x(1.0);
167 + //radar_obstacle_1->set_z(1.1);
168 + //aiforce::pp_msg::radarStopListPP_radarObstacle* radar_obstacle_2 =
pub_msg_radarStopListPP->add_radarobstaclelist();
169 + //radar_obstacle_2->set_sensor_id(2);
170 + //radar_obstacle_2->set_x(2.0);
171 + //radar_obstacle_2->set_z(2.1);



```


- 每一次提交分配一个 Commit ID

- 

mm_test_node: conf
 由 Weipu Shan 创作于 2天前



799dc7b9

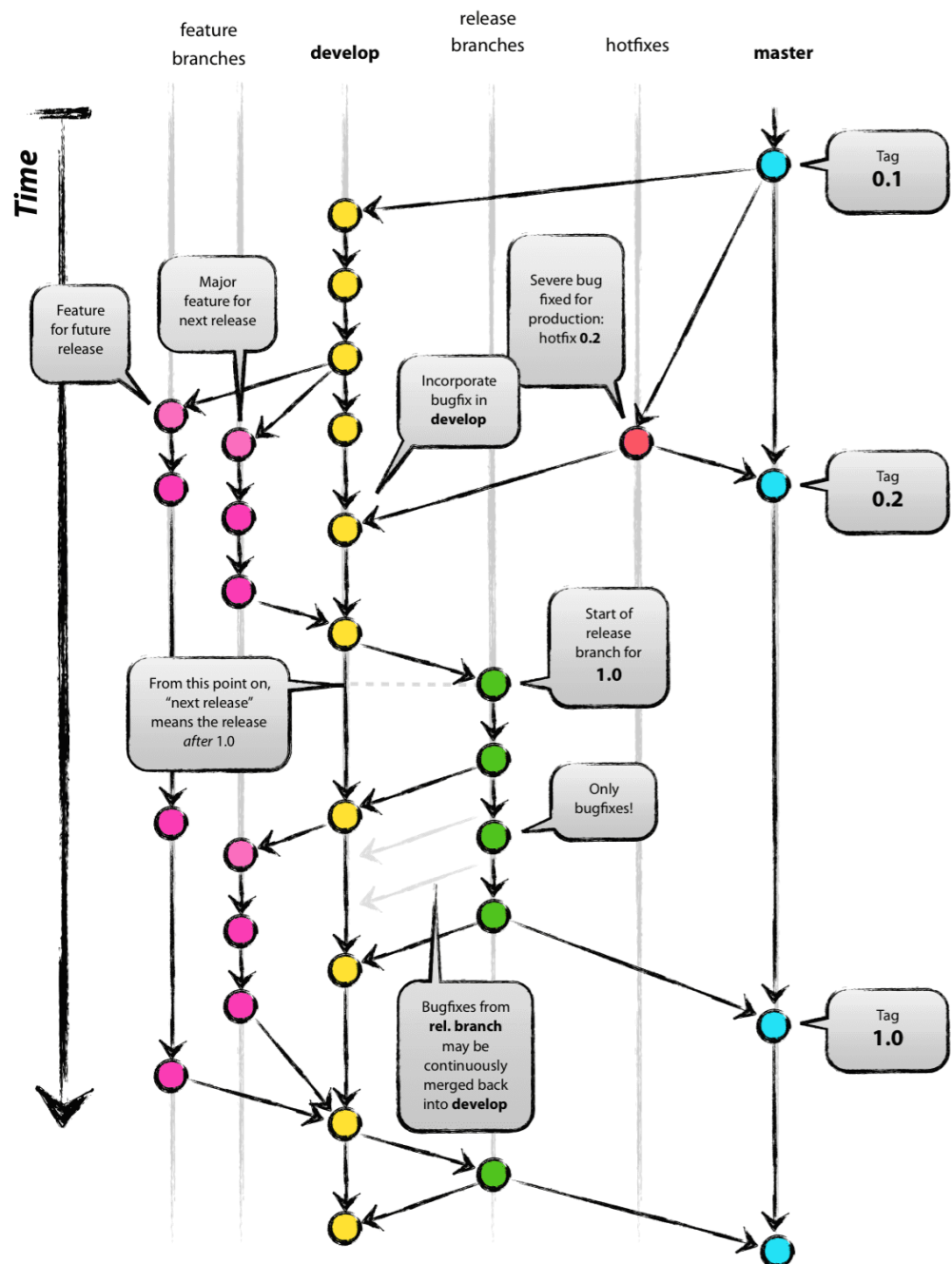


add data folder
 由 Weipu Shan 创作于 2天前

115364dd

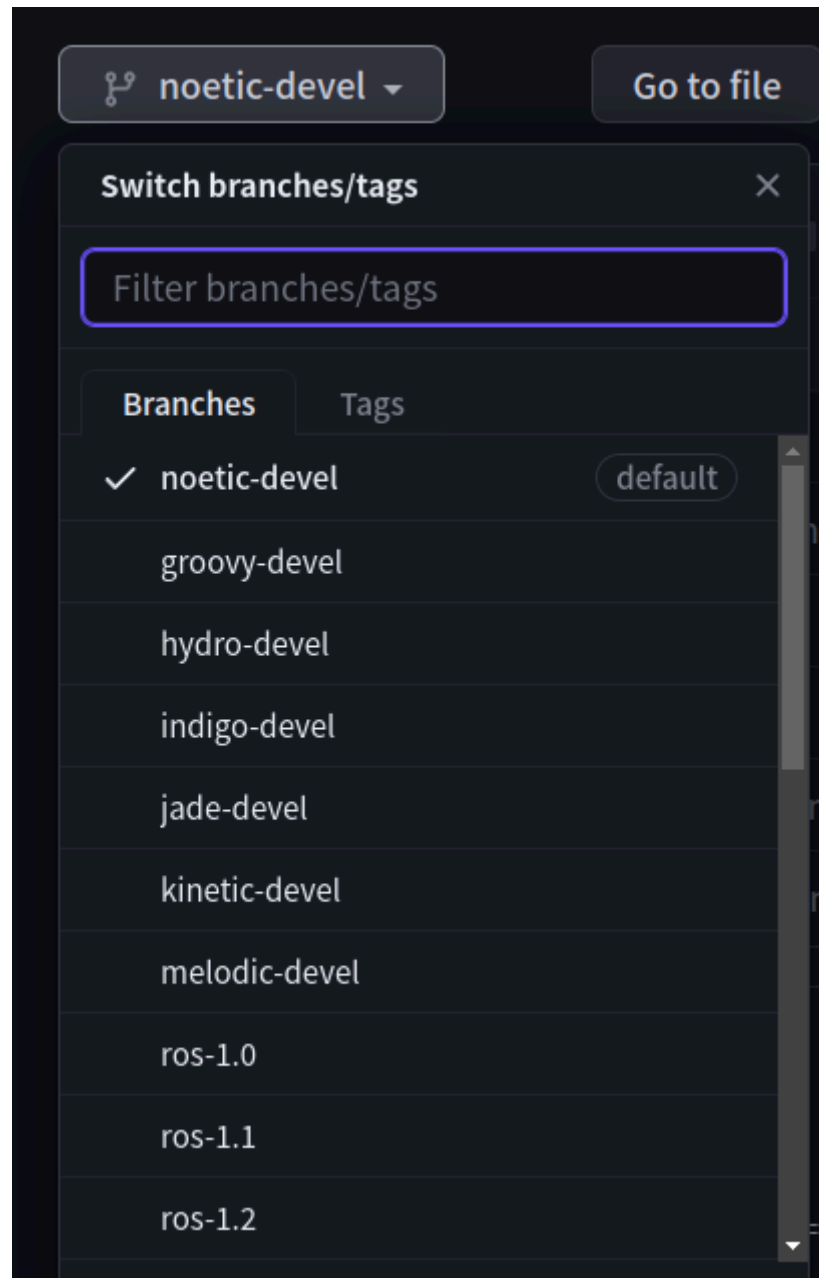
 

- 查阅历史提交记录
- 回滚历史记录
- 代码分支管理
 - 创建分支
 - master: 项目主分支
 - hotfixes: 项目重大紧急Bug修复分支
 - release: 小版本更新分支
 - develop: 开发分支
 - feature: 功能开发分支
 - ...



- 管理分支
 - 新建分支
 - 在分支中开发功能，避免产生冲突
 - 合并分支

- 将开发好的分支合并入项目主分支，合并新功能
- 删除分支
- 例子：ROS 在 Github 页面中的分支，针对不同Ubuntu版本



- 多人协作
 - 代码权限管理，小组中不同的开发者有不同的权限



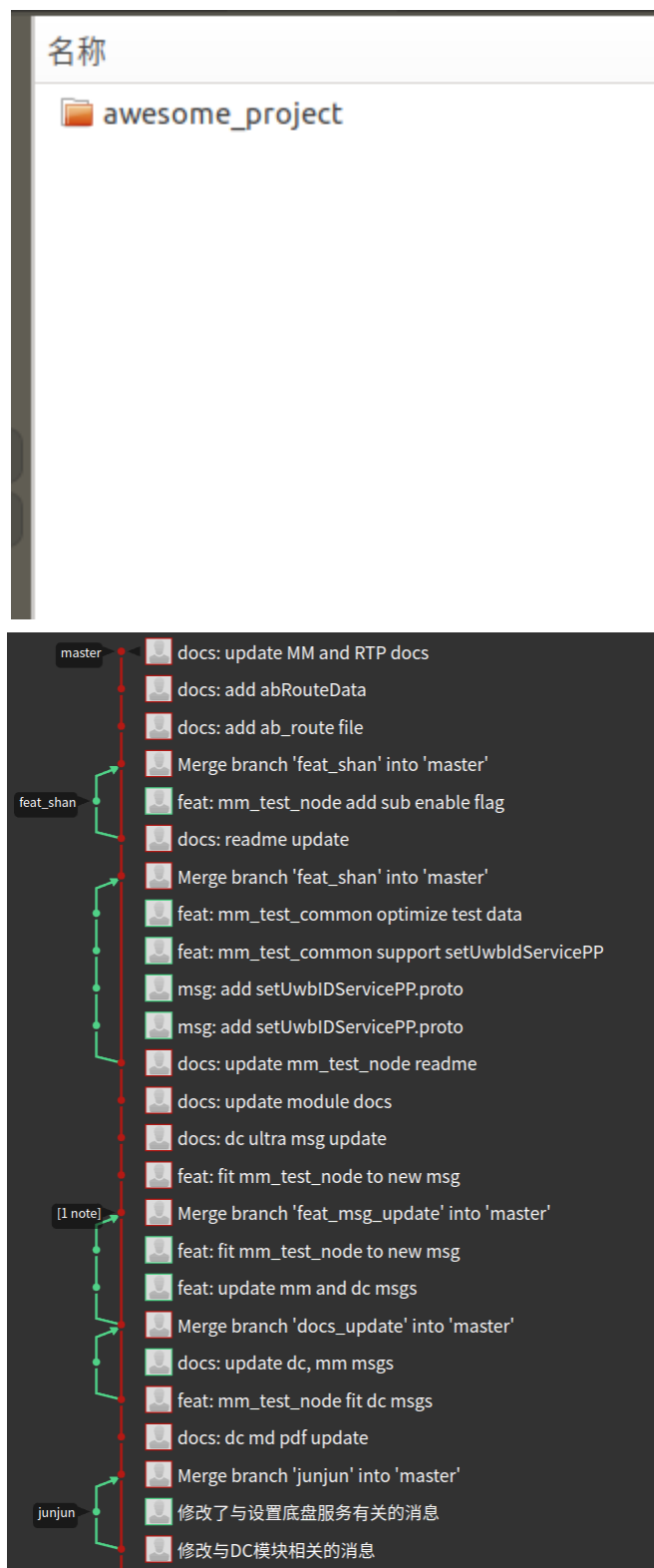
- 代码审核

- 开发者提交代码后，项目负责人审核代码质量，并将功能分支合并到主分支中
- 不同功能同时开发
 - 针对同一份代码中不同的功能模块，可以通过分支进行分别的开发，并最终合并到主分支
- 冲突解决（应该尽量避免这种情况）
 - 不同开发者对同一分支下同一文件进行了修改后，提供冲突解决方案
- Git 使用前后对比
 - 用 Git 之前
 - 混乱的不同版本文件夹
 - 多个开发者需要对同一份代码进行修改
 - 修改代码后想要回到之前版本非常困难
 - 没有时间序列上的版本控制



awesome_project_v1
awesome_project_v1.1
awesome_project_v1.1(复件)
awesome_project_v1.1(复件)(复件)
awesome_project_v1.2
awesome_project_v1(复件)
awesome_project_超级无敌终极版
awesome_project_超级无敌终极打死不改版
awesome_project_超级终极版
awesome_project_终极版

- 用 Git 之后
 - 干净的文件夹
 - 每个开发者可以在自己的分支上进行开发，之后统一合并功能
 - 通过提交记录轻松找到每次提交的代码记录



- 一个典型的Git项目可能包含的内容
 - src
 - 源代码
 - docs
 - 与项目相关的文档
 - 3rdparty
 - 本项目所需第三方依赖的库文件，安装方式，Git子模块等
 - config
 - 项目相关的配置文件

- README.md
 - Markdown格式的项目说明文档，会显示在 Github/Gitlab 项目页面
- .gitignore
 - git 项目过滤配置文件
- Git 仓库中不应该加入的内容（通过 .gitignore 过滤）
 - 代码编译生成的内容（不同环境平台编译出的内容不一样）
 - 可执行程序
 - build 文件夹
 - devel 文件夹
 - bin 文件夹
 - 开发者个人编辑器生成的配置文件（每个开发者有自己的开发工具与配置环境）
 - .vscode/
 - .idea/
 - .user
 - 数据集（体积过大，而且有专门的数据集版本管理软件）
 - 深度学习模型（体积过大，而且有专门的深度学习模型管理软件）
 - 原则上，能自动生成的，体积太大的，与代码关系不大的，以及不是项目中大家都可能需要的内容，就应该过滤，加入到 .gitignore 文件中

Git 安装

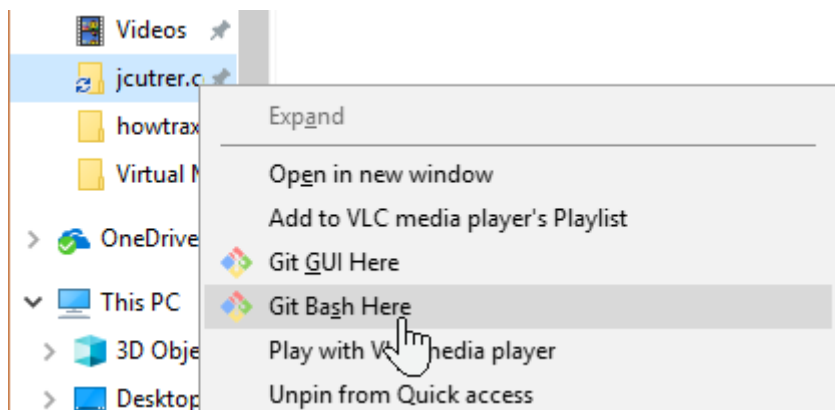
- Linux
 - 通过终端命令行输入

```
1 | sudo apt-get install git
```

- Mac
 - 通过终端命令行输入

```
1 | brew install git
```

- Windows
 - [Git 官方 Windows 下载页面](#)
 - Windows版的Git安装包附带一个 git bash 工具，类似 Linux 下的终端命令行工具
 - 在想要进行Git管理的文件夹下右键：

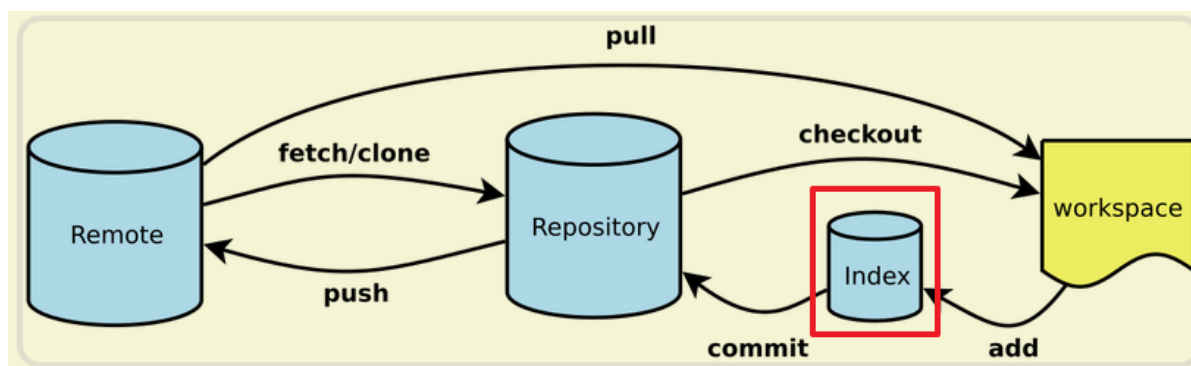


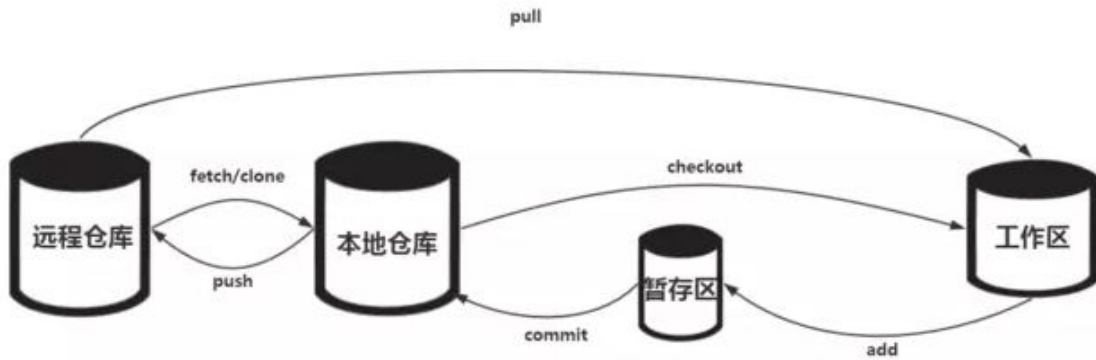
- Git Bash 的界面非常类似 Linux 的终端，也支持 ls cd mv cp rm 等常用 Linux 命令

```
MINGW64/c/Users/me/git
me@work MINGW64 ~
$ git clone https://github.com/git-for-windows/git
Cloning into 'git'...
remote: Enumerating objects: 500937, done.
remote: Counting objects: 100% (3486/3486), done.
remote: Compressing objects: 100% (1415/1415), done.
remote: Total 500937 (delta 2494), reused 2917 (delta 2071), pack-reused 497451
Receiving objects: 100% (500937/500937), 221.14 MiB | 1.86 MiB/s, done.
Resolving deltas: 100% (362274/362274), done.
Updating files: 100% (4031/4031), done.
me@work MINGW64 ~
$ cd git
me@work MINGW64 ~/git (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
me@work MINGW64 ~/git (main)
$ |
```

Git 仓库说明





- 远程库 remote repository
 - 同一个软件项目一般只有一个远程库，比如在gitlab/github上保存的库
- 本地库 local repository
 - 同一个软件项目可以有多个本地库
 - 场景1：笔记本上进行开发，之后在TX2上测试与修改
 - 场景2：同一个软件项目不同的开发人员在各自的电脑里进行开发，然后统一提交到一个远程库
- 本地库包括
 - 工作区 workspace
 - 当前对代码进行的修改会出现在工作区
 - 此时通过 `git status` 查看，会显示修改的文件名
 - 暂存区 Index/Stage
 - 将当前的修改通过 `git add` 可以添加到本地的暂存区
 - 此时通过 `git status` 查看，会显示添加到暂存区的文件名
 - 本地库 local repository
 - 通过 `git commit` 可以将当前在暂存区的修改正式提交到本地库
 - 此时通过 `git status` 查看，不会再显示修改的文件名，因为修改的文件已经提交到本地库

```

1 → TransportRobot_common git:(master) git status
2 位于分支 master
3 您的分支领先 'origin/master' 共 1 个提交。
4   （使用 "git push" 来发布您的本地提交）
5
6 无文件要提交，干净的工作区
  
```

- 通过 `git push` 可以将本地库的提交推送到远程库，此时通过 `git status` 查看，可以发现当前分支与远程库的分支一致

```

1 → TransportRobot_common git:(master) git status
2 位于分支 master
3 您的分支与上游分支 'origin/master' 一致。
4
5 无文件要提交，干净的工作区
  
```

- 通过 `git pull` 可以将远程库的更新拉取到本地库

- git pull = git fetch + git merge
 - git fetch 表示将远程库同步到本地的远程库镜像
 - git fetch 不会影响此时本地库的工作区
 - git merge 表示将本地的远程库镜像与本地库合并
 - 有冲突时需要手动解决冲突

Git 基础操作流程

1 项目文件夹中添加 .gitignore 文件

- .gitignore 可以帮助 git 过滤不需要加入到 git 仓库的内容
- 原则上git仓库中只允许储存代码以及相关说明文件或第三方库文件，体积大于1MB的文件请谨慎添加到git仓库，不建议上传到git仓库的内容比如：
 - 编译产生的文件与可执行程序
 - build 文件夹
 - bin 文件夹
 - ROS 的 devel 文件夹
 - 个人IDE的配置文件夹
 - .vscode/
 - .idea/
 - 深度学习模型
 - 数据集
 - 代码运行时自动生成的数据
- 一个可以自动生成 .gitignore 的网站，可以参考
 - <https://www.toptal.com/developers/gitignore>
- 例子：采摘机器人ROS工程中的 .gitignore 内容

```
1 # model related
2 *.engine
3 *.pth
4 *.trt
5 *.onnx
6
7 # ros related
8 devel/
9 logs/
10 build/
11 bin/
12 msg_gen/
13 srv_gen/
14 build_isolated/
15 devel_isolated/
16
```

```
17 # backup file
18 *.bak
19
20 # Generated by dynamic reconfigure
21 *.cfgc
22 /cfg/cpp/
23 /cfg/*.py
24
25 # Ignore generated docs
26 *.dox
27 *.wikidoc
28
29 # vscode stuff
30 .vscode
31
32 # eclipse stuff
33 .project
34 .cproject
35
36 ### QT related
37 *.user
38 *.user.*
```

2 上传代码到gitlab/github/gitee

- 新建空白仓库



- 注意取消勾选自述文件初始化仓库 (不然Gitlab 不会提示你接下去干什么)

新建项目 › 创建空白项目

项目名称

My awesome project

项目 URL

http://192.168.100... weipu_shan

项目标识串

my-awesome-project

想要在同一命名空间下存放几个依赖项目？请创建一个群组。

项目描述(可选)

描述格式

Project deployment target (optional)

Select the deployment target

可见性级别 ⓘ

☒ 私有

项目访问必须明确授予每个用户。 如果此项目是在一个群组中，群组成员将会获得访问权限。

☐ 内部

除外部用户外，任何登录用户均可访问该项目。

☐ 公开

该项目允许任何人访问。

项目配置

☐ 使用自述文件初始化仓库

允许您立即克隆这个项目的仓库。如果您计划推送一个现有的仓库，请跳过这个步骤。

新建项目

取消

- gitlab 页面会提示你接下来的操作。
 - **注意：**在多人共用的测试平台比如TX2等，建议将全局设置 --global 改为局部设置 --local，这样可以将设置保留在该仓库内部，不影响同一台电脑其他人的repo设置

Git 全局设置

```
git config --global user.name "Weipu Shan"
git config --global user.email "shanweipu@aiforcetech.com"
```

创建一个新仓库

```
git clone git@192.168.100.88:weipu_shan/test_repo.git
cd test_repo
git switch -c main
touch README.md
git add README.md
git commit -m "add README"
git push -u origin main
```

推送现有文件夹

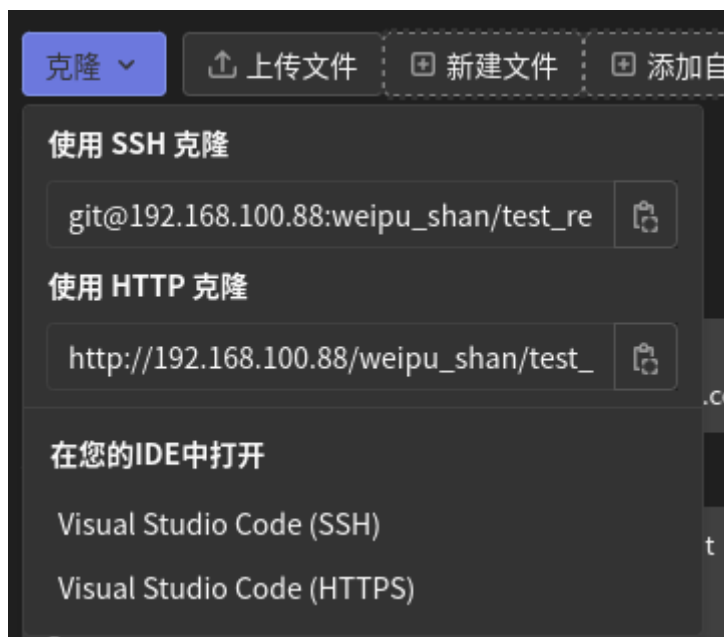
```
cd existing_folder
git init --initial-branch=main
git remote add origin git@192.168.100.88:weipu_shan/test_repo.git
git add .
git commit -m "Initial commit"
git push -u origin main
```

推送现有的 Git 仓库

```
cd existing_repo
git remote rename origin old-origin
git remote add origin git@192.168.100.88:weipu_shan/test_repo.git
git push -u origin --all
git push -u origin --tags
```

3 克隆远程库到本地

```
1 # 方法1: 通过http链接克隆
2 # 缺点: 每次提交或拉取需要输入账号密码
3 git clone [http_link]
4
5 # 方法2: 通过ssh链接克隆
6 # 需要事先将电脑的 ssh key 设置到gitlab账号中
7 # ssh-keygen -t rsa -C [email@adress]
8 # 优点: 无需输入账号密码, 方便提交, 但注意不要在公共平台中使用, 否则任何人都有可能修改你的代
   码并提交到你的gitlab仓库中
9 git clone [ssh_link]
```



4 本地库更新并提交到远程库

```
1 # 将工作区所有修改添加到暂存区
2 git add .
3
4 # 将暂存区的内容提交到本地库
5 # 提交时需要简短地说明本次提交的意义
6 # 不要每次都是"update" 或者"fix bug", 最少也要说明这次update了什么文件, 修复了什么bug
7 # 建议commit_msg的格式:
8 #   feat: 增加了什么新功能
9 #   fix: 修复了什么bug
10 #   docs: 改动了什么文档
11 #   style: 修改了代码的格式, 如注释, 缩进, 空格, 空白行等
12 #   build: 构建工具相关的改动
13 #   refactor: 重构了哪些部分的代码
14 #   test: 测试了什么内容, 造成了此次代码修改
15 #   perf: 提高性能方面的改动
16 #   other: 其他内容的改动
17 git commit -m "commit_msg"
18
19 # 将本地库推送到远程库
20 git push
```

适用场景:

- 笔记本开发完代码上传到gitlab
- 测试平台比如TX2测试后有代码修改, 将修改的内容上传到gitlab

5 远程库的修改拉取到本地库

```
1 # 方法1（常用）：通过 git pull 将远程库的更新内容直接同步到本地库
2 # 建议养成良好习惯，每次要在本地库修改代码前，先确认是否与远程库同步
3 # 如果远程库有未同步的代码，那么本地库修改了同一个代码文件后，想要上传就会引发冲突，需要手动解决
4 git pull
5
6
7 # 方法2（不常用）：通过 git fetch + git merge 的方式
8 # git fetch 将远程库同步到本地的一个远程库镜像中，不会影响本地库
9 git fetch
10 # git merge 会将远程库的内容与本地库合并，如果有冲突文件，则需要手动修改冲突后提交上传
11 git merge
12
```

6 修改冲突的流程

- 如果远程库和本地库对同一份文件有修改，就会产生冲突，导致无法将本地库的修改推送到远程库，比如笔记本上修改了代码，并推送到了gitlab，然后去TX2上调试时忘记将远程库的更新拉取到本地，这时候TX2上调试后修改的代码更新就无法推送到远程库

```
→ TransportRobot_common git:(master) git push
To 192.168.100.88:crawlerrobot/transportrobot_common.git
! [rejected]        master -> master (fetch first)
错误：无法推送一些引用到 '192.168.100.88:crawlerrobot/transportrobot_common.git'
提示：更新被拒绝，因为远程仓库包含您本地尚不存在的提交。这通常是因为另外
提示：一个仓库已向该引用进行了推送。再次推送前，您可能需要先整合远程变更
提示：（如 'git pull ...'）。
提示：详见 'git push --help' 中的 'Note about fast-forwards' 小节。
```

- 此时通过 git pull 先拉取远程库的更新到本地库，这时冲突的文件会保留冲突的部分等待用户解决

```
→ TransportRobot_common git:(master) git pull
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
展开对象中: 100% (3/3), 553 字节 | 553.00 KiB/s, 完成.
来自 192.168.100.88:crawlerrobot/transportrobot_common
65f2b36..3714bd9 master -> origin/master
自动合并 test.txt
冲突（内容）：合并冲突于 test.txt
自动合并失败，修正冲突然后提交修正的结果。
```

- 通过查看提示我们可以发现 test.txt 出现冲突，打开该文件

```
1 test
2
3 <<<<<< HEAD
4 change2
5 =====
6 change1
7 >>>>>> 3714bd9351d26b60c612a7409ee2279baa6b2e26
```

- 手动编辑该文件，随后重新 git commit + git push 即可解决冲突并同步到远程库

7 放弃本地的代码修改

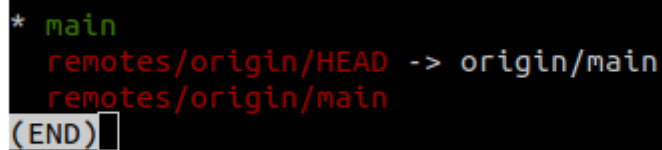
- 有时候测试平台测试后，我们并不希望保存此时的代码修改，而是希望保持本地库原来的状态，可以使用 `git stash + git stash clear` 的方式放弃此次修改

```
1 # 将当前的代码修改放入隔离区
2 git stash
3
4 # 清空隔离区
5 git stash clear
```

8 分支管理

- 查看所有分支

```
1 # 该命令会显示本地与远程库的所有分支
2 # 远程库的分支带有origin的前缀
3 # 当前分支前面会有*号
4 git branch -a
```



A terminal window with a black background and green text. It shows the output of the `git branch -a` command. The output lists several branches: `* main`, `remotes/origin/HEAD -> origin/main`, and `remotes/origin/main`. The `(END)` prompt is visible at the bottom.

- 新建分支

```
1 git branch [branch_name]
```

- 切换到分支

```
1 # 注意：切换分支前务必将当前分支工作区的内容提交到本地库
2
3 # 方法1：通过 checkout 切换分支，checkout还可以切换历史提交记录
4 git checkout [branch_name]
5
6 # 方法2：通过 switch 切换分支
7 git switch [branch_name]
```

- 新建并直接切换到分支

```
1 git checkout -b [branch_name]
```

- 上传分支到远程库


```

1  # 先确保切换到想要上传的分支
2  git checkout [branch_name]
3
4  # 修改代码，将代码提交到当前的分支
5  git add .
6  git commit -m "commit_msg"
7
8  # 情况1: 远程库还没有当前分支: push时需要设置远程分支，建议分支名称与本地一致
9  git push --set-upstream origin [branch_name]
10
11 # 情况2: 远程库已有当前分支: 直接push
12 git push

```

- 合并分支（建议通过gitlab中的merge request按钮，不建议直接合并）

```

1  # 比如要把dev分支的内容合并到主分支
2  # 首先切换到主分支
3  git checkout main
4
5  # 合并dev分支所有提交记录到main分支
6  git merge dev
7
8  # 将main分支合并后的结果push到远程库
9  git push

```

- 合并分支的部分文件

```

1  # 合并其他分支中的部分文件到当前分支
2  git checkout [branch_name] [file_name]

```

- 删除分支

```

1  # 删除本地已经完全合并的分支
2  git branch -d [branch_name]
3
4  # 强制删除本地分支，即使没有合并
5  git branch -D [branch_name]
6
7  # 删除远程分支
8  git push origin -d [branch_name]
9
10 # 当远程库中的分支被删除后，通过 git branch -a 仍然能在本地发现远程库的被删除分支
11 # 此时可以通过以下命令清除在本地的远程库镜像中的被删除分支
12 git remote prune origin

```

9 功能分支的变基操作

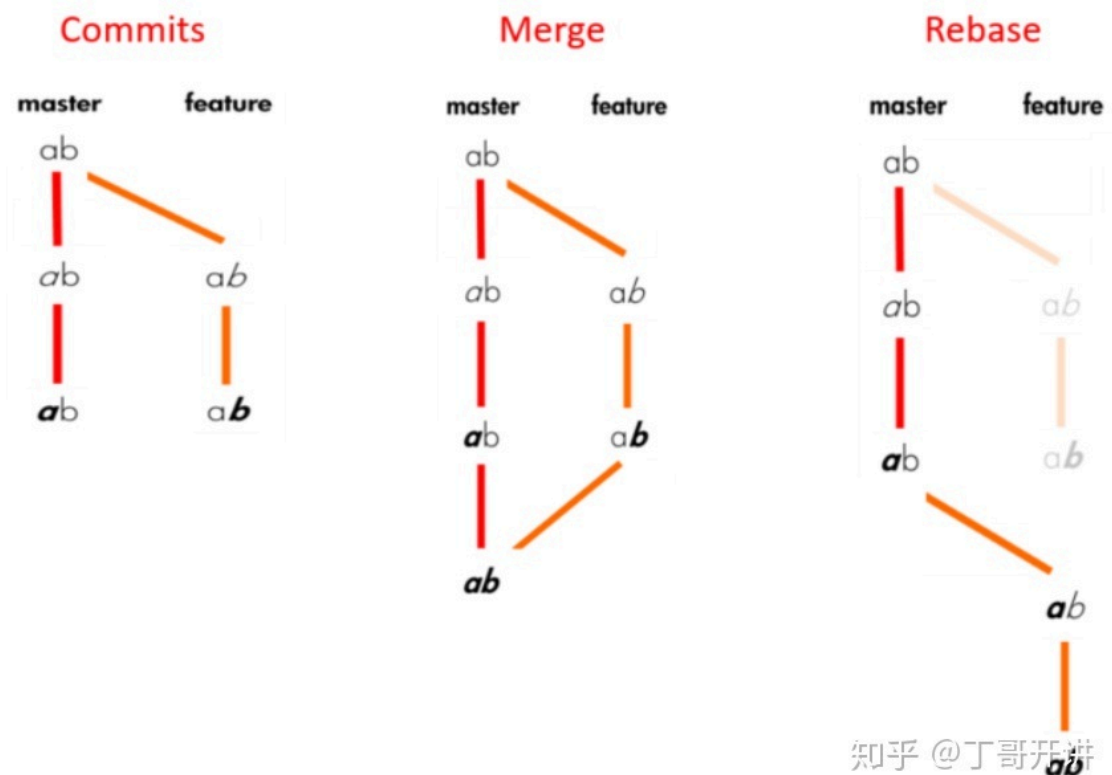
- Git rebase 操作方法

```

1  # git rebase 一般用于功能分支开发完成后，拉取主分支的变化，适应主分支
2  # 假设目前处于功能分支 feat_a 中，需要合并主分支 main 或者 master
3  git rebase [main/master]

```

- git merge vs git rebase
 - git merge
 - merge 表示合并，合并两个分支的不同之处
 - merge 用来将功能分支合并到主分支
 - merge 会保留分支的提交记录的时间节点
 - git rebase
 - rebase 表示变基，改变当前分支的基础
 - rebase 用来帮助功能分支获取主分支的变化
 - rebase 会将分支的提交记录移到主分支的最新更新之后，既改变分支提交记录的时间节点
- 建议使用方法：
 1. 开启一个功能分支
 2. 开发功能分支
 3. 功能分支开发完毕，通过 git rebase 将功能分支的所有commit移到最新的主分支之后
 4. 修复功能分支rebase之后的问题
 - git rebase master 之后，如果有冲突，会暂停rebase，提示修改有冲突的文件
 - 解决冲突文件后，通过git add [file_name] 添加修改内容
 - 无需commit，直接 git rebase --continue 继续 rebase 的过程
 5. 功能分支提交 merge request
 6. 主分支审核 功能分支提交的 merge request，完成合并，理论上这样的合并不会再出现冲突。



Git 最简工作流程

单人项目

- 克隆远程库到本地电脑

```
1 | git clone [http-link / ssh-link]
```

- 本地库获取gitlab最新的代码

```
1 | # 拉取远程库的更新
2 | git pull
```

- 本地库上传到远程库

```
1 | # 添加修改的文件到暂存区
2 | git add .
3 |
4 | # 将暂存区的内容提交到本地库
5 | # 注意提交消息尽量简短但有辨识度
6 | git commit -m "commit msg"
7 |
8 | # 将本地库推送到远程库
9 | git push
```

- 放弃本地库此次代码修改

```
1 | # 添加修改到暂存区
2 | git add .
3 |
4 | # 将暂存区中的内容移到隔离区
5 | git stash
6 |
7 | # 清空隔离区
8 | git stash clear
```

多人协作项目

1. 开启一个功能分支

```
1 | # 分支建议命名规则:
2 | #   feat_xxx: 功能分支
3 | #   bugfix_xxx: Bug修复分支
4 | #   docs_xxx: 文档更新分支
5 | #   test_xxx: 测试相关分支
6 | git checkout -b [branch_name]
```

2. 开发功能分支，在功能分支提交

```
1 # 首先确认是否在功能分支下
2 git add .
3 git commit -m "commit msg" # 注意提交消息尽量简短但有辨识度
4 git push
```

3. 功能分支开发完毕，通过 git rebase 将功能分支的所有commit移到最新的主分支之后

```
1 # 转到主分支main，拉取主分支最新更新
2 git checkout main
3 git pull
4
5 # 转到功能分支下
6 git checkout [branch_name]
7 # 确认已提交所有功能分支的修改
8 # 将功能分支变基到主分支的最新状态
9 git rebase main
```

4. 修复功能分支rebase之后的问题与冲突，确保功能完善

5. 功能分支在 gitlab 页面提交 merge request

6. 主分支审核 功能分支提交的 merge request，完成合并

- 理论上这样可以做到无冲突合并

Git 操作笔记（查询使用）

Git 使用原则

- 初始化仓库请使用 .gitignore 过滤不必要的文件
 - 代码自动生成的内容如 build bin cfg 等文件夹
 - 数据库，深度学习推理模型等大尺寸的文件
 - 个人IDE自动生成的配置文件等
- 提交时请使用简短但有辨识度的 commit_msg，建议commit_msg的格式：
 - feat: 增加了什么新功能
 - fix: 修复了什么bug
 - docs: 改动了什么文档
 - style: 修改了代码的格式，如注释，缩进，空格，空白行等
 - build: 构建工具相关的改动
 - refactor: 重构了哪些部分的代码
 - test: 测试了什么内容，造成了此次代码修改
 - perf: 提高性能方面的改动
 - other: 其他内容的改动
- 碰到不熟悉的git命令，尽量先单独测试，避免无法挽回的后果
- 多人合作时：
 - 主分支仅用来合并功能分支，不建议直接在主分支进行任何修改
 - 开发功能，修复bug都在各自的功能分支中完成，随后提交合并请求

- 功能分支开发完成后需要通过 git rebase 去适应主分支的最新变化，随后提交合并请求 (merge request)，如此可以做到无冲突合并
- 主分支合并功能分支，原则上应该无冲突合并 (conflict-free merge)

Git 帮助

```
1 # git 一般命令的帮助
2 git -h
3
4 # git 具体命令的帮助
5 git clone -h
6 git init -h
7 git add -h
8 git commit -h
9 git branch -h
10 git merge -h
11 git fetch -h
12 ...
```

Git 设置相关操作

- 生成 ssh key, 通过ssh克隆项目到本地，之后每次提交或拉取都不需要输入账号密码，**建议只在自己的电脑这样操作**

```
1 # 生成ssh key
2 cd ~
3 rm -r .ssh/
4 ssh-keygen -t rsa -C shanweipu@aiforcetech.com
5
6 # 显示ssh key, 复制到gitlab或github的设置页面中
7 cd .ssh
8 cat id_rsa.pub
```

- 设置全局的user.name, user.email

```
1 git config --global user.name "weipu Shan"
2 git config --global user.email "shanweipu@aiforcetech.com"
```

- 设置每个仓库单独的user.name, user.email

```
1 # cd to a git repo
2 # set git repo local config
3 git config --local user.name "weipu Shan"
4 git config --local user.email "shanweipu@aiforcetech.com"
```

- 修改git默认编辑器 (git 原生配置的 nano 编辑器特别难用)，常用来合并与解决冲突

```
1 # open ~/.gitconfig
2 gedit ~/.gitconfig
3
4 # put following line in file
5 [core]
6 editor = gedit
```

- 设置 VPN 端口

```
1 # 命令中的主机号（127.0.0.1）是使用的代理的主机号（自己电脑有vpn那么本机可看做访问
  github的代理主机），即填入127.0.0.1即可，否则填入代理主机 ip（就是网上找的那个ip）
2 # 命令中的端口号（7890）为代理软件（代理软件不显示端口的话，就去windows中的代理服务器
  设置中查看）或代理主机的监听IP，可以从代理服务器配置中获得，否则填入网上找的那个端口
  port
3
4 # socket5
5 git config --global http.proxy socks5 127.0.0.1:7890
6 git config --global https.proxy socks5 127.0.0.1:7890
7
8 # http
9 git config --global http.proxy 127.0.0.1:7890
10 git config --global https.proxy 127.0.0.1:7890
11
12
```

Git 查看状态与历史记录相关操作

- 查看工作区，暂存区状态, 按 q 退出

```
1 | git status
```

- 查看git 历史记录 可以获取每一次commit的版本号，按 q 退出

```
1 | git log
```

- 查看某一次commit的具体内容

```
1 | git show [commit-ID]
```

- 查看某次文件在工作区中的修改

```
1 | git diff [file path]
```

Git 提交与拉取相关操作

- 将修改添加到暂存区

```
1 # 添加一个文件到暂存区
2 git add [file_name]
3
4 # 添加全部修改的文件到暂存区
5 git add .
```

- 删除暂存区的文件

```
1 # 删除暂存区文件，但保留文件在工作区
2 git rm --cached [file_name]
3
4 # 删除暂存区以及工作区的该文件
5 git rm [file_name]
```

- 提交暂存区的文件到本地库

```
1 # 提交时需要简短地说明本次提交的意义
2 # 不要每次都是"update" 或者"fix bug"，最少也要说明这次update了什么文件，修复了什么
  bug
3 # 建议commit_msg的格式：
4 #   feat: 增加了什么新功能
5 #   fix: 修复了什么bug
6 #   docs: 改动了什么文档
7 #   style: 修改了代码的格式，如注释，缩进，空格，空白行等
8 #   build: 构建工具相关的改动
9 #   refactor: 重构了哪些部分的代码
10 #   test: 测试了什么内容，造成了此次代码修改
11 #   perf: 提高性能方面的改动
12 #   other: 其他内容的改动
13 git commit -m "commit_msg"
```

- 推送本地库到云端库

```
1 git push
```

- 拉取远程库到本地

```
1 # 方法1: 直接更新本地库的当前分支
2 git pull
3
4 # 方法2: 分开操作拉取与合并当前分支
5 # 拉取远程库到本地的镜像
6 git fetch
7 # 更新本地库当前分支
8 git merge
```

Git stash - 隔离区相关操作

- 将当前修改的操作放入隔离区

```
1 | git stash
```

- 清空隔离区

```
1 | git stash clear
```

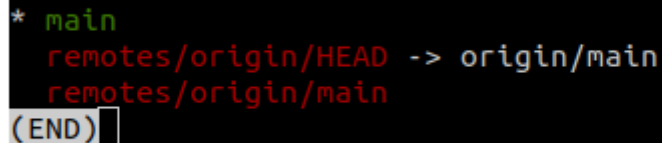
- 弹出最近一次储存到隔离区的改动

```
1 | git stash pop
```

Git branch - 分支相关操作

- 查看所有分支

```
1 | # 该命令会显示本地与远程库的所有分支
2 | # 远程库的分支带有origin的前缀
3 | # 当前分支前面会有*号
4 | git branch -a
```



```
* main
remotes/origin/HEAD -> origin/main
remotes/origin/main
(END)
```

- 新建分支

```
1 | git branch [branch_name]
```

- 切换到分支

```
1 | # 注意：切换分支前务必将当前分支工作区的内容提交到本地库
2 |
3 | # 方法1：通过 checkout 切换分支，checkout还可以切换历史提交记录
4 | git checkout [branch_name]
5 |
6 | # 方法2：通过 switch 切换分支
7 | git switch [branch_name]
```

- 新建并直接切换到分支

```
1 | git checkout -b [branch_name]
```

- 上传分支到远程库


```

1  # 先确保切换到想要上传的分支
2  git checkout [branch_name]
3
4  # 修改代码，将代码提交到当前的分支
5  git add .
6  git commit -m "commit_msg"
7
8  # 情况1: 远程库还没有当前分支: push时需要设置远程分支，建议分支名称与本地一致
9  git push --set-upstream origin [branch_name]
10
11 # 情况2: 远程库已有当前分支: 直接push
12 git push

```

- 合并分支

- 不建议使用，需要合并功能分支建议通过gitlab页面填写合并请求
- **主分支合并功能分支原则上需要无冲突合并**
- **功能分支提交合并请求前，需要 git rebase 适应主分支的最新状态，以此达到无冲突合并的效果**

```

1  # 比如要把dev分支的内容合并到主分支
2  # 首先切换到主分支
3  git checkout main
4
5  # 合并dev分支所有提交记录到main分支
6  git merge dev
7
8  # 将main分支合并后的结果push到远程库
9  git push

```

- 合并分支的部分文件

- 不建议使用，需要合并功能分支建议通过gitlab页面填写合并请求
- **主分支合并功能分支原则上需要无冲突合并**
- **功能分支提交合并请求前，需要 git rebase 适应主分支的最新状态，以此达到无冲突合并的效果**

```

1  # 合并其他分支中的部分文件到当前分支
2  git checkout [branch_name] [file_name]

```

- 删除分支

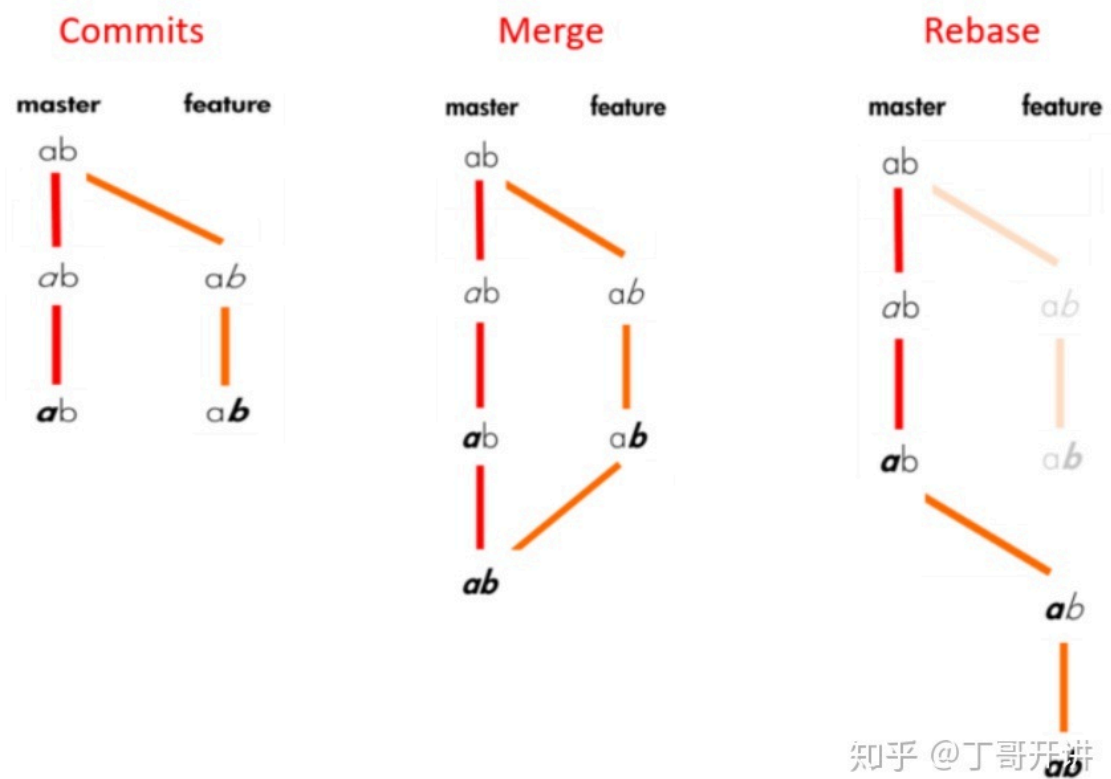
```
1  # 删除本地已经完全合并的分支
2  git branch -d [branch_name]
3
4  # 强制删除本地分支，即使没有合并
5  git branch -D [branch_name]
6
7  # 删除远程分支
8  git push origin -d [branch_name]
9
10 # 当远程库中的分支被删除后，通过 git branch -a 仍然能在本地发现远程库的被删除分支
11 # 此时可以通过以下命令清楚在本地的远程库镜像中的被删除分支
12 git remote prune origin
```

Git rebase - 变基相关操作

- Git rebase 操作方法

```
1  # git rebase 一般用于功能分支开发完成后，拉取主分支的变化，适应主分支
2  # 假设目前处于功能分支 feat_a 中，需要合并主分支 main 或者 master
3  git rebase [main/master]
```

- git merge vs git rebase
 - git merge
 - merge 表示合并，合并两个分支的不同之处
 - merge 用来将功能分支合并到主分支
 - merge 会保留分支的提交记录的时间节点
 - git rebase
 - rebase 表示变基，改变当前分支的基础
 - rebase 主要用来帮助功能分支获取主分支的变化
 - rebase 会将分支的提交记录移到主分支的最新更新之后，既改变分支提交记录的时间节点
- 建议使用方法：
 1. 开启一个功能分支
 2. 开发功能分支
 3. 功能分支开发完毕，通过 git rebase 将功能分支的所有commit移到最新的主分支之后
 4. 修复功能分支rebase之后的问题
 - git rebase master 之后，如果有冲突，会暂停rebase，提示修改有冲突的文件
 - 解决冲突文件后，通过git add [file_name] 添加修改内容
 - 无需commit，直接 git rebase --continue 继续 rebase 的过程
 5. 功能分支提交 merge request
 6. 主分支审核 功能分支提交的 merge request，完成合并，理论上这样的合并不会再出现冲突。



Git 回退版本相关操作

- 未commit, 单纯想要取消当前代码修改

```
1 # 先添加修改到暂存区
2 git add .
3
4 # 隔离当前暂存区
5 git stash
6
7 # 清空隔离区
8 git stash clear
```

- 已commit, 未push到远程库

```
1 # 撤销commit, 但保留add
2 git reset --soft [commit_ID]
3
4 # 撤销commit以及add
5 git reset --mixed [commit_ID]
```

- 已commit, 已push到远程库

```
1 # 方法1: 回退版本但保留提交记录
2 # 相当于将上一次的提交记录重新放到当前提交记录前
3 # commit_ID 为需要撤回的提交
4 git revert [commit_ID]
5
6 # 方法2: 回退版本并舍弃提交记录 (不建议这种方法)
7 # 需要 git push -f 强制提交删除commit后的记录
8 # commit_ID 为需要恢复到的提交版本
9 git reset --hard [commit_ID]
10 git push -f
```

Git submodule - 子模块相关操作

- 下载子模块

```
1 # 克隆了之后下载子模块
2 git submodule update --init --recursive
3
4 # 克隆时直接更新子模块
5 git clone --recurse-submodules [http/ssh link]
```

- 为自己的仓库加入子模块

```
1 # 进入子模块需要保存的位置, 比如 3rdparty 文件夹
2 # 添加子模块, 建议通过 http link 添加, ssh link 在主模块通过 http link clone 时无法更新
3 git submodule add [http link]
4
5 # 子模块一般不会在当前主模块内被修改, 只起到使用的作用
6 # 子模块由本身的仓库负责维护
7 # 子模块更新后, 在当前主模块的子模块文件夹内可以拉取更新
8 git submodule update
9
10 # 子模块更新后, 需要在主模块中更新子模块的注册信息, 更新该子模块加入到主模块时的 commit ID
```

Git tag 标签相关操作

- Git 标签的意义
 - tag就像是一个里程碑一个标志一个点, branch是一个新的征程一条线
 - tag是静态的, branch要向前走;
 - 稳定版本备份用tag, 新功能多人开发用branch
- 操作方法
 - 创建标签

```
1 # 在当前分支的当前提交记录上打标签
2 git tag [tag_name]
3
4 # 针对特定提交ID打标签
5 git tag [tag_name] [commit_ID]
6
7 # 创建标签时添加注释说明
8 git tag -a [tag_name] -m "tag_msg" [commit_ID]
```

- 提交标签

```
1 git push origin --tags
```

- 查看标签

```
1 # 查看所有标签
2 git tag
3
4 # 查看某个标签的注释说明
5 git show [tag_name]
```

- 删除标签

```
1 # 删除本地库的标签
2 git tag -d [tag_name]
3
4 # 删除远程库的标签
5 git push origin :refs/tags/[tag_name]
```

Git 获取当前 Commit ID

```
1 # 获取完整 commit ID
2 git rev-parse HEAD
3
4 # 获取简短 commit ID
5 git rev-parse --short HEAD
```

Git cherry-pick 合并分支中的某次提交

`cherry-pick` 是一个非常常用的Git命令，它的功能是将某个分支的某次提交应用到当前分支。这对于将特定的代码改动从一个分支移动到另一个分支非常有用。例如，你可能在开发分支上修复了一个bug，然后想要将这个修复应用到主分支，而不带入其他开发分支的改动，这时候就可以使用 `cherry-pick` 命令。

- 语法

```
1 # 该命令会将指定的提交应用到当前分支上，并创建一个新的提交
2 git cherry-pick <commit>
```

- 使用场景
 - 合并单个提交：当我们只想应用某个分支上的一个提交到当前分支时，可以使用cherry-pick命令，而不需要合并整个分支。
 - 修复bug：当我们在一个分支上修复了一个bug，并希望将这个修复应用到其他分支上时，可以使用cherry-pick命令。
 - 提取特定功能：当我们在一个分支上开发了一个新功能，并希望将该功能应用到其他分支上时，可以使用cherry-pick命令。
- 注意
 - 类似 merge, cherry-pick 也可能会造成冲突，当产生冲突时，需要手动解决冲突。

小彩蛋

通过 Gource 可视化Git仓库的提交记录

- [Gource](#)

