目　录

前言

基础知识

- 安装git
- 创建版本库
- 版本回退
- 工作区和暂存区
- 管理修改
- 撤销修改
- 删除文件

远程仓库

- 添加远程仓库
- 从远程仓库克隆

分支管理

- 创建与合并分支
- 解决冲突
- 分支管理策略
- Bug分支
- Feature分支
- 多人协作
- Rebase变基

标签管理

- 创建标签
- 操作标签

扩展知识

- merge与no-ff merge
- fatal: refusing to merge unrelated histories
- git merge origin master与merge origin/master
- Git撤回已经推送至远程仓库的提交

速查宝典

其他命令

前言

Git 是一个开源的分布式版本控制系统，用于敏捷高效地处理任何或小或大的项目。

Git 是 Linus Torvalds 为了帮助管理 Linux 内核开发而开发的一个开放源码的版本控制软件。

Git 与常用的版本控制工具 CVS, Subversion 等不同，它采用了分布式版本库的方式，不必服务器端软件支持。

本手册转载自：https://www.rumosky.wiki/docs/learngit 作者：如默

基础知识

说明

之前自己在使用 `git` 的时候出现了很多的问题，每次百度或者Google之后，找到的文章总是支离破碎，或者直接复制粘贴官方文档里的内容，没有很详细的指导新人如何使用 `git` ，教程里的命令介绍也很不全面。所以自己写一些笔记，方便遗忘的时候查询。

> 基础知识包含git安装、创建版本库、版本回退等基础内容，继续阅读请点击下一节目录

手册目录

安装git

前言

`git` 是全版本的软件，下载安装时请选择对应的系统即可

在Windows上安装

前往官网下载对应客户端：https://git-scm.com/downloads

下载完成后默认安装即可，（默认会将git添加到系统环境变量）

在Mac上安装

如果你正在使用 `Mac` 做开发，有两种安装 `Git` 的方法。

一是安装 `homebrew` ，然后通过 `homebrew` 安装 `Git` ，具体方法请参考 `homebrew` 文档：https://brew.sh/index_zh-cn

第二种方法更简单，也是推荐的方法，就是直接从 `AppStore` 安装 `Xcode` ， `Xcode` 集成了 `Git` ，不过默认没有安装，你需要运行 `Xcode` ，选择菜单 `Xcode->Preferences` ，在弹出窗口中找到 `Downloads` ，选择 `Command Line Tools` ，点 `Install` 就可以完成安装了。

在Linux上安装

首先，试着在终端里面输入 `git` ，查看是否已安装，若出现下列内容，则说明没有安装 `git`

```
$ git
The program 'git' is currently not installed. You can install it by typing:
sudo apt-get install git
```

所以，Ubuntu系统使用：

```
sudo apt-get install git
```

CentOS使用：

```
sudo yum install git -y
```

其他版本Linux需要使用源码编译安装
先去官网下载最新版本压缩包：https://github.com/git/git/releases
解压之后执行下列命令：

```
git clone https://github.com/git/git.git
./configure
make
sudo make install
```

配置

安装好git之后，需要配置一下才可以正常使用，打开命令行，执行：

```
git config --global user.name "Your Name"
git config --global user.email "email@example.com"
```

上述命令里的name和Email是你注册GitHub时使用的name和Email

> 注意git config命令的–global参数，用了这个参数，表示你这台机器上所有的Git仓库都会使用这个配置，当然也可以对某个仓库指定不同的用户名和Email地址。

创建版本库

结论

初始化本地仓库：

```
git init
```

添加文件到仓库：

```
# 添加单个文件
git add <file>

# 添加多个文件
git add file1 file2 ...

# 添加全部已修改文件
git add .
```

提交文件到仓库：

```
git commit -m "说明"
```

说明

什么是版本库呢？版本库又名仓库，英文名 `repository` ，你可以简单理解成一个目录，这个目录里面的所有文件都可以被 `Git` 管理起来，每个文件的修改、删除， `Git` 都能跟踪，以便任何时刻都可以追踪历史，或者在将来某个时刻可以还原。

步骤

首先，选择一个合适的地方，创建一个空目录，在该目录下执行:

```
git init
```

此时一个仓库就创建好了，在该目录下会生成一个 `.git` 隐藏文件，这个文件是 `git` 的配置文件，请勿随意修改

其次，添加一个文件到仓库，新建readme.txt，执行：

```
git add readme.txt
```

然后把文件提交到仓库，执行：

```
git commit -m "你的注释"
```

结果如下：

```
$ git commit -m "第一次提交"
[master (root-commit) 6999761] 第一次提交
```

```
1 file changed, 1 insertion(+)
create mode 100644 readme.txt
```

其中， `-m` 后面输入的是本次提交的说明，可以输入任意内容，当然最好是有意义的，这样你就能从历史记录里方便地找到改动记录。

`git commit` 命令执行成功后会告诉你， `1 file changed` ：1个文件被改动（我们新添加的readme.txt文件）； `1 insertions` ：插入了一行内容（readme.txt有一行内容）。

版本回退

结论

git工作区状态：

```
git status
```

查看全部修改内容：

```
git diff
```

查看指定文件修改内容：

```
git diff <file>
```

回退到指定版本

```
git reset --hard commit_id
```

回退到上一个版本

```
git reset --hard HEAD^
```

回退到上上一个版本

```
git reset --hard HEAD^^
```

回退到上n个版本

```
git reset --hard HEAD~n
```

查看详细提交历史

```
git log
```

查看简化提交历史

```
git log --pretty=oneline
```

查看分支合并图

```
git log --graph
```

查看命令历史

```
git reflog
```

说明

我们多次修改文件，如果不小心删除了某些东西，可以使用版本回退来实现复原

步骤

继续上一节的内容，修改readme.txt，增加了一行内容，执行：

```
git status
```

结果如下：

```
$ git status
On branch master
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.txt

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

`git status` 命令可以让我们时刻掌握仓库当前的状态，上面的命令输出告诉我们，`readme.txt` 被修改过了，但还没有提交修改。

此时，如果想查看具体我们修改了 `readme.txt` 的哪一部分内容，执行：

```
git diff
```

结果如下：

```
$ git diff
diff --git a/readme.txt b/readme.txt
index cc3b095..6b77a0b 100644
--- a/readme.txt
+++ b/readme.txt
@@ -1,2 +1,3 @@
 git is very famous!
-第一次追加内容
\ No newline at end of file
+第一次追加内容^M
+第二次追加内容
\ No newline at end of file
```

知道修改了什么内容，就可以放心提交了，依次执行 `git add` `git commit -m ""` 即可，提交完毕，执行git status查看状态，显示如下：

```
$ git status
On branch master
nothing to commit, working tree clean
```

好了，经过多次提交之后，如果想退回到某个版本，先执行：

```
git log
```

结果如下：

```
$ git log
commit f09d57ce853e850551e8802b9a4be3643ba894c0 (HEAD -> master)
Author: rumosky <rumosky@163.com>
Date:   Sun Nov 3 16:02:23 2019 +0800

    第三次追加内容

commit c3b8908ddddd8364ac8b2681b56e948885e49b1d
Author: rumosky <rumosky@163.com>
Date:   Sun Nov 3 16:00:36 2019 +0800

    第二次追加内容

commit a82d91a6bb97b1acc158d98bc1f82697df938e3b
Author: rumosky <rumosky@163.com>
Date:   Sun Nov 3 15:49:55 2019 +0800

    第一次追加内容

commit 69997611303057230d8fa50c81681bd823644553
Author: rumosky <rumosky@163.com>
Date:   Sun Nov 3 15:28:27 2019 +0800

    第一次提交
```

可以看到有四次提交，其中，`commit`后面的一串字符是`commit_id`，若觉得日志内容很长，可以添加参数`--pretty=oneline`，结果如下：

```
$ git log --pretty=oneline
f09d57ce853e850551e8802b9a4be3643ba894c0 (HEAD -> master) 第三次追加内容
c3b8908ddddd8364ac8b2681b56e948885e49b1d 第二次追加内容
a82d91a6bb97b1acc158d98bc1f82697df938e3b 第一次追加内容
69997611303057230d8fa50c81681bd823644553 第一次提交
```

回到上一个版本，结果如下：

```
$ git reset --hard HEAD^
HEAD is now at c3b8908 第二次追加内容
```

现在，我们回退到了上一个版本，但是如果我们后悔了，想恢复到新版本怎么办？
没事，此时，先执行git reflog找到最新版的commit_id，结果如下：

```
$ git reflog
c3b8908 (HEAD -> master) HEAD@{0}: reset: moving to HEAD^
f09d57c HEAD@{1}: commit: 第三次追加内容
c3b8908 (HEAD -> master) HEAD@{2}: commit: 第二次追加内容
a82d91a HEAD@{3}: commit: 第一次追加内容
6999761 HEAD@{4}: commit (initial): 第一次提交
```

第三次追加内容`commit_id`是`f09d57c`，执行回退命令，结果如下：

```
$ git reset --hard f09d57c
HEAD is now at f09d57c 第三次追加内容
```

此时，查看readme文件，发现已经恢复了：

```
$ cat readme.txt
git is very famous!
第一次追加内容
第二次追加内容
第三次追加内容
```

> commit_id没有必要全部输入，至少输入前四位就可以找到该commit

工作区和暂存区

结论

工作区 `Working Directory`
就是你在电脑里能看到的目录，在创建版本库时新建的那个目录

版本库 `Repository`
工作区有一个隐藏目录 `.git` ，这个不算工作区，而是 `Git` 的版本库。

`Git` 的版本库里存了很多东西，其中最重要的就是称为 `stage` （或者叫 `index` ）的暂存区

说明

前面讲了我们把文件往Git版本库里添加的时候，是分两步执行的：

第一步是用 `git add` 把文件添加进去，实际上就是把文件修改添加到暂存区；

第二步是用 `git commit` 提交更改，实际上就是把暂存区的所有内容提交到当前分支。

因为我们创建 `Git` 版本库时， `Git` 自动为我们创建了唯一一个 `maste` r分支，所以，现在， `git commit` 就是往 `master` 分支上提交更改。

你可以简单理解为，需要提交的文件修改通通放到暂存区，然后，一次性提交暂存区的所有修改。

管理修改

结论

Git跟踪并管理的是修改，而非文件

Git只能追踪文本文件的改动，比如TXT文件，网页，所有的程序代码等等，比如在第5行加了一个单词“Linux”，在第8行删了一个单词“Windows”。

图片、视频这些二进制文件，虽然也能由版本控制系统管理，但没法跟踪文件的变化，只能把二进制文件每次改动串起来，也就是只知道图片从100KB改成了120KB，但到底改了啥，版本控制系统不知道，也没法知道。

说明

需要注意的是，如果你按照下述方式提交：

```
第一次修改->add->第二次修改->commit
```

那么，第二次修改的内容不会被提交

按照下述方式提交，则两次修改都会被提交

```
第一次修改->add->第二次修改->add->commit
```

所以，没有add的内容，即使commit之后也不会被提交

撤销修改

结论

丢弃工作区的修改（未提交至暂存区）

```
# 丢弃指定文件的修改
git checkout -- file
git restore <file>

# 丢弃所有文件的修改
git checkout -- .
git restore .
```

丢弃已添加到暂存区的修改

```
# 丢弃指定文件的修改
git reset HEAD <file>
git restore --staged <file>

# 丢弃所有文件的修改
git reset HEAD .
git restore --staged .
```

删除文件

结论

删除未添加到暂存区的文件

```
#显示将要删除的文件和目录
git clean -n

#删除文件和目录
git clean -df

#删除文件
git clean -f
git rm <file>
```

远程仓库

添加远程仓库

结论

关联远程仓库

```
git remote add origin <url>
# 其中origin是默认的远程仓库名，也可以自行修改
# url可以是ssh链接，也可以是http链接，推荐使用ssh，安全高速
```

删除远程仓库

```
git remote rm origin
```

查看远程仓库

```
git remote -v
```

推送提交到远程仓库

```
git push origin master
# 一般用于非首次推送
```

```
git push -u origin master
# -u参数是将本地master分支与远程仓库master分支关联起来，一般用于第一次推送代码到远程库
```

说明

现在的情景是，你的本地仓库已经有了，但是你必须要有一个远程仓库，才可以使得自己的代码可以让别人来协作开发，也可以作为一个本地仓库的备份。

从远程仓库克隆

结论

克隆远程仓库到本地

```
git clone url
# url可以是ssh或http，建议使用原生ssh链接，高速安全
```

说明

之前讲的内容都是先有本地库，后有远程库，然后再关联远程库。

而一般大多数情形都是先有远程库，然后克隆远程库到本地，再进行工作。

分支管理

说明

分支就像玄幻小说里面的分身，让一个分身去学习 `JavaScript` ，另一个分身去学习 `Git` ，然后合并两个分身之后，这两个分身对你的本体没有任何影响，平时互不干扰，只有在某个特定的时间，两个分身合并了，你就学会了 `Git` 和 `JavaScript` ！

好了，言归正传，分支在实际中有什么用呢？假设你准备开发一个新功能，但是需要两周才能完成，第一周你写了50%的代码，如果立刻提交，由于代码还没写完，不完整的代码库会导致别人不能干活了。如果等代码全部写完再一次提交，又存在丢失每天进度的巨大风险。

现在有了分支，就不用怕了。你创建了一个属于你自己的分支，别人看不到，还继续在原来的分支上正常工作，而你在自己的分支上干活，想提交就提交，直到开发完毕后，再一次性合并到原来的分支上，这样，既安全，又不影响别人工作。

那么有的人可能会问了，创建、切换分支会不会很慢，如果文件非常多的话。

这个就不用担心了，Git的分支是与众不同的，无论创建、切换和删除分支，Git在1秒钟之内就能完成！无论你的版本库是1个文件还是1万个文件。

建议

1. 多提交，少推送。多人协作时，推送会频繁地带来合并冲突的问题，影响效率。因此，尽量多使用提交命令，减少合并的使用，这样会节省很多时间。
2. 使用Git流 `Git Flow` ，即多分支结构。
3. 使用分支，保持主分支的整洁。这是我强烈推荐的一点，在分支进行提交，然后切到主分支更新 `git pull —rebase` ，再合并分支、推送。这样的流程会避免交叉合并的情况出现（不会出现共同祖先节点为多个的情况）。事实上，git合并操作让很多人感到不知所措的原因就是各种原因所产生的交叉合并问题，从而造成在合并的过程中丢失某些代码。保持主分支的整洁能够避免交叉合并的情况出现。
4. 禁用 `fast-forward` 模式。在拉取代码的时候使用 `rebase` 参数（前提是保持主分支的整洁）、合并的时候使用 `--no-ff` 参数禁用 `fast-forward` 模式，这样做既能保证节点的清晰，又避免了交叉合并的情况出现。

创建与合并分支

结论

查看分支

```
git branch
```

创建分支

```
git branch <name>
```

切换分支

```
git checkout <name>
git switch <name>
# switch是2.23版本新增命令
```

创建并切换到该分支

```
git checkout -b <name>
git switch -c <name>
```

合并指定分支到当前分支

```
git merge <name>
```

删除本地已合并分支

```
git branch -d <name>
```

删除远程分支

```
git push <远程仓库名> --delete <远程分支名>
```

推送本地分支到远程仓库并在远程仓库创建新分支

```
git push <远程仓库名> <本地分支名>:<远程分支名>
```

说明

在版本回退里，你已经知道，每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。截止到目前，只有一条时间线，在Git里，这个分支叫主分支，即master分支。HEAD严格来说不是指向提交，而是指向master，master才是指向提交的，所以，HEAD指向的就是当前分支。

所以创建分支、切换分支、删除分支只是对相对应的指针进行操作，所以速度才会非常快。

switch

我们注意到切换分支使用 `git checkout <branch>` ，而前面讲过的撤销修改则是 `git checkout -- <file>` ，同一个命令，有两种作用，确实有点令人迷惑。

实际上，切换分支这个动作，用 `switch` 更科学。因此，最新版本的Git提供了新的git switch命令来切换分支，使用新的git switch命令，比git checkout要更容易理解。

git: ‘switch’ is not a git command.

找不到 `switch` 命令是因为这个命令是 `2.23` 版本发布的，在此之前的版本都没有，需要升级到最新版

首先，查看当前版本，执行：

```
git --version
```

2.17.1版本之前，升级命令：

```
git update
```

2.17.1之后的版本，升级命令：

```
git update-git-for-windows
```

解决冲突

结论

当Git无法自动合并分支时，就必须首先解决冲突。解决冲突后，再提交，合并完成。

解决冲突就是把Git合并失败的文件手动编辑为我们希望的内容，再提交。

查看分支合并图：

```
git log --graph
```

> 冲突的产生一般都是这两种情况：
>
> - 远程仓库的代码落后于本地仓库
> - 远程仓库的代码远超本地仓库
>
>> 在你还未提交代码的时候，你的同事已经提交了代码，就会导致远程仓库代码领先于你的代码

说明

冲突是如何表示的

当产生合并冲突时，该部分会以 `<<<<<<<` ， `=======` 和 `>>>>>>>` 表示。在 `=======` 之前的部分是当前分支这边的情况，在 `=======` 之后的部分是传入分支的情况。

如何解决冲突

在看到冲突以后，你可以选择以下两种方式：

- 决定不合并。这时，唯一要做的就是重置 `index` 到 `HEAD` 节点。 `git merge --abort` 用于这种情况。
- 解决冲突。 `Git` 会标记冲突的地方，解决完冲突的地方后使用 `git add` 加入到 `index` 中，然后使用 `git commit` 产生合并节点。

你可以用以下工具来解决冲突:

- 使用合并工具。 `git mergetool` 将会调用一个可视化的合并工具来处理冲突合并。
- 查看差异。 `git diff` 将会显示三路差异（三路合并中所采用的三路比较算法）。
- 查看每个分支的差异。 `git log --merge -p <path>` 将会显示 `HEAD` 版本和 `MERGE_HEAD` 版本的差异。
- 查看合并前的版本。 `git show :1:文件名` 显示共同祖先的版本， `git show :2:文件名` 显示当前分支的 `HEAD` 版本， `git show :3:文件名` 显示对方分支的 `MERGE_HEAD` 版本。

分支管理策略

结论

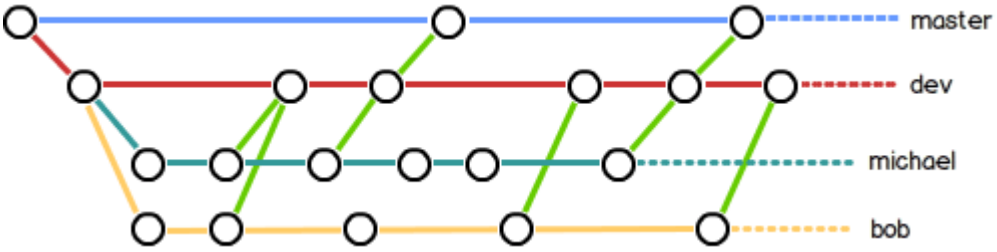
在实际开发中，我们应该按照几个基本原则进行分支管理：

首先，`master`分支应该是非常稳定的，也就是仅用来发布新版本，平时不能在上面干活；

那在哪干活呢？干活都在`dev`分支上，也就是说，`dev`分支是不稳定的，到某个时候，比如`1.0`版本发布时，再把`dev`分支合并到`master`上，在`master`分支发布`1.0`版本；

你和你的小伙伴们每个人都在`dev`分支上干活，每个人都有自己的分支，时不时地往`dev`分支上合并就可以了。

所以，团队合作的分支看起来就像这样：



说明

`Git`分支十分强大，在团队开发中应该充分应用。

通常，合并分支时，如果可能，Git会用`Fast forward`模式，但这种模式下，删除分支后，会丢掉分支信息。

如果要强制禁用`Fast forward`模式，`Git`就会在`merge`时生成一个新的`commit`，这样，从分支历史上就可以看出分支信息。

合并分支时，加上`--no-ff`参数就可以用普通模式合并，合并后的历史有分支，能看出来曾经做过合并，而`fast forward`合并就看不出来曾经做过合并。

Bug分支

结论

暂存工作区状态

```
git stash
```

查看暂存的工作区状态

```
git stash list
```

恢复全部暂存状态，但不删除暂存内容

```
git stash apply
```

恢复指定暂存状态，但不删除暂存内容

```
git stash apply stash@{<id>}
```

删除暂存内容

```
git stash drop
```

恢复暂存状态，同时删除暂存内容

```
git stash pop
```

复制一个特定的提交到当前分支

```
git cherry-pick <commit_id>
```

说明

修复bug时，我们会通过创建新的bug分支进行修复，然后合并，最后删除；

当手头工作没有完成时，先把工作现场 `git stash` 一下，然后去修复bug，修复后，再 `git stash pop` ，回到工作现场；

在master分支上修复的bug，想要合并到当前dev分支，可以用 `git cherry-pick <commit_id>` 命令，把bug提交的修改“复制”到当前分支，避免重复劳动。

Feature分支

结论

强制删除分支（会丢失分支上的修改）

```
git branch -D <name>
```

说明

开发一个新feature，最好新建一个分支；

如果要丢弃一个没有被合并过的分支，可以通过 `git branch -D <name>` 强行删除。

多人协作

结论

查看远程仓库信息

```
git remote
```

查看远程仓库详细信息

```
git remote -v
```

与远程仓库代码同步

```
git pull
# git pull = git fetch + git merge
```

在本地创建和远程分支对应的分支

```
git checkout -b branch-name origin/branch-name
git switch -c branch-name origin/branch-name
```

将本地分支与远程仓库关联

```
git branch --set-upstream-to <branch-name> origin/<branch-name>
```

推送本地分支到远程仓库

```
git push origin <branch-name>
```

说明

多人协作的工作模式通常是这样：

1. 首先，可以试图用 `git push origin <branch-name>` 推送自己的修改；
2. 如果推送失败，则因为远程分支比你的本地更新，需要先用 `git pull` 试图合并；
3. 如果合并有冲突，则解决冲突，并在本地提交；
4. 没有冲突或者解决掉冲突后，再用 `git push origin <branch-name>` 推送就能成功！
5. 如果 `git pull` 提示 `no tracking information` ，则说明本地分支和远程分支的链接关系没有创建。

这就是多人协作的工作模式，一旦熟悉了，就非常简单。

> - 本地新建的分支如果不推送到远程，对其他人就是不可见的；
> - 在本地创建和远程分支对应的分支，本地和远程分支的名称最好一致；

> - 从远程抓取分支，如果有冲突，要先处理冲突。

问题

在这里记录多人协作时可能会遇到的问题

描述

> 本地仓库有文件，远程仓库也有文件，但是这两个仓库文件不一致。这时，将本地仓库与远程仓库关联起来，执行 `git branch --set-upstream-to <branch-name> origin/<branch-name>` ，提示错误： `error: the requested upstream branch 'origin/master' does not exist"`

解决办法：

若直接执行 `git pull` 会提示： `refusing to merge unrelated histories`

正确做法：

```
git pull origin master --allow-unrelated-histories
git branch --set-upstream-to=origin/master master
git push origin master
```

Rebase变基

结论

变基（衍合）

```
git rebase <branch>
```

放弃变基

```
git rebase --abort
```

解决冲突之后继续变基

```
git rebase --continue
```

说明

rebase操作可以把本地未push的分叉提交历史整理成直线；

rebase的目的是使得我们在查看历史提交的变化时更容易，因为分叉的提交需要三方对比。

标签管理

说明

发布一个版本时，我们通常先在版本库中打一个标签（tag），这样，就唯一确定了打标签时刻的版本。将来无论什么时候，取某个标签的版本，就是把那个打标签的时刻的历史版本取出来。所以，标签也是版本库的一个快照。

Git的标签虽然是版本库的快照，但其实它就是指向某个commit的指针（跟分支很像对不对？但是分支可以移动，标签不能移动），所以，创建和删除标签都是瞬间完成的。

Git有commit，为什么还要引入tag？

“请把上周一的那个版本打包发布，commit号是6a5819e...”

“一串乱七八糟的数字不好找！”

如果换一个办法：

“请把上周一的那个版本打包发布，版本号是v1.2”

“好的，按照tag v1.2查找commit就行！”

所以，tag就是一个让人容易记住的有意义的名字，它跟某个commit绑在一起。

创建标签

结论

新建标签（指向最新的commit_id）

```
git tag <tag_name>
```

新建标签（指向特定commit_id）

```
git tag <tag_name> <commit_id>
```

查看所有标签

```
git tag
```

显示某个标签的详细信息

```
git show <tag_name>
```

新建带有说明的标签

```
git tag -a <tag_name> -m "说明" <commit_id>
```

操作标签

结论

删除指定本地标签

```
git tag -d <tag_name>
```

删除指定远程标签

```
git push origin :refs/tags/<tag_name>
```

推送一个本地标签

```
git push origin <tag_name>
```

推送全部未推送过的本地标签

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

扩展知识

说明

此文档记录一些git扩展内容，不定期更新，感兴趣的可以试试

merge与no-ff merge

git 的 merge 与 no-ff merge 的不同之处

通常，合并分支时，如果可能，`Git` 会用 `Fast forward` 模式，但这种模式下，删除分支后，会丢掉分支信息。如果要强制禁用 `Fast forward` 模式，`Git` 就会在 `merge` 时生成一个新的 `commit`，这样，从分支历史上就可以看出分支信息。

证明

> 仓库只有一个文件readme.txt，仅有一个主分支master

1. 新建分支 `dev1`，修改 `readme.txt`，然后在dev1分支下提交

```
git switch -c dev1
git add readme.txt
git commit -m "dev1 branch commit"
```

2. 回到 `master` 分支，执行 `merge` 即 `git merge dev1`

```
git switch master
git merge dev1
```

3. 删除分支

```
git branch -d dev1
```

4. 查看日志

```
git log --graph --pretty=oneline --abbrev-commit
```

5. 新建分支 `dev2`，修改 `readme.txt`，然后提交

```
git switch -c dev2
git add readme.txt
git commit -m "dev2 branch commit"
```

6. 回到 `master` 分支，执行 `merge`

```
git switch master
git merge --no-ff -m "dev2 merged with mo-ff" dev2
```

7. 删除分支

```
git branch -d dev2
```

8. 查看日志即

```
git log --graph --pretty=oneline --abbrev-commit
```

9. 比较两次合并，可以看出不同之处，`no-ff`的模式会记录分支历史

fatal: refusing to merge unrelated histories

说明

有时在pull或merge时会出现下述错误：

```
fatal: refusing to merge unrelated histories
```

解决办法：

```
git merge origin/master --allow-unrelated-histories
```

结果如下：

```
$ git merge origin/master --allow-unrelated-histories
Already up to date!
Merge made by the 'recursive' strategy.
```

只需要在命令的最后面添加 `--allow-unrelated-histories` ，这句话是告诉Git允许不相关历史合并。

git merge origin master与merge origin/master

git merge origin master和git merge origin/master的区别

```
git merge origin master
# 将origin merge 到 master 上

git merge origin/master
# 将origin上的master分支 merge 到当前 branch 上
```

Git撤回已经推送至远程仓库的提交

说明

我们在工作时，经常会遇到已经提交远程仓库，但是又不是我想要的版本，要撤下来。这时可以使用下列命令：

```
# 不删除工作区的修改，仅撤销commit
git reset --soft <commit_id>

# 删除工作区的修改，撤销commit
git reset --hard <commit_id>
```

注意

`git reset --soft` 表示只是改变了HEAD的指向，本地代码不会变化，我们使用 `git status` 依然可以看到，同时也可以 `git commit` 提交。

`git reset --hard` 后者直接回改变本地源码，不仅仅指向变化了，代码也回到了那个版本时的代码，所以使用是一定要小心，想清楚。

速查宝典

中文版

速查表

> 创建

1. 克隆现有仓库

```
git clone ssh://user@domain.com/repo.git
```

2. 创建一个新的本地仓库

```
git init
```

> **本地修改**

1. 查看工作目录中已更改的文件，即查看git状态

```
git status
```

2. 查看跟踪文件的更改，即远程仓库与本地仓库文件的不同

```
git diff
```

3. 将所有当前更改添加到下一次提交，即全部提交

```
git add .
```

4. 将某个文件的更改添加到下一次提交，即部分提交

```
git add -p <file>
```

5. 提交跟踪文件中的所有本地更改

```
git commit -a
```

6. 提交先前进行的更改

```
git commit
```

7. 更改最后一次提交（不要修改已发布的提交）

```
git commit --amend
```

提交历史

1. 显示所有提交，从最新的提交开始

```
git log
```

2. 显示特定文件随时间的修改

```
git log -p <file>
```

3. 查看特定文件什么时间被什么人修改了什么内容

```
git blame <file>
```

分支和标签

1. 列出所有现有分支

```
git branch -av
```

2. 切换当前分支

```
git checkout <branch>
```

3. 基于当前head指针创建新分支

```
git branch <new-branch>
```

4. 基于当前远程分支创建一个新跟踪分支

```
git checkout --track <remote/branch>
```

5. 删除一个本地分支

```
git branch -d <branch>
```

6. 将一次提交标记为一个标签

```
git tag <tag-name>
```

更新和推送

1. 列出当前仓库关联的所有远程仓库

```
git remote -v
```

2. 显示远程仓库的详细信息

```
git remote show <remote>
```

3. 关联一个新的远程仓库到本地仓库

```
git remote add <shortname> <url>
```

4. 仅拉取远程仓库所有更改，不合并到本地仓库

```
git fetch <remote>
```

5. 拉取远程仓库所有更改，并合并到本地仓库

```
git pull <remote> <branch>
```

6. 推送本地仓库修改到远程仓库

```
git push <remote> <branch>
```

7. 删除远程仓库的一个分支

```
git branch -dr <remote/branch>
```

8. 推送所有本地仓库标签到远程仓库

```
git push --tags
```

> **合并和变基**

1. 合并指定分支到当前分支

```
git merge <branch>
```

2. 变基

```
git rebase <branch>
```

3. 放弃变基

```
git rebase --abort
```

4. 解决冲突之后继续变基

```
git rebase --continue
```

5. 运行合并冲突解决工具来解决合并冲突

```
git mergetool
# 合并冲突解决工具需要配置，是图形化工具
```

6. 使用编辑器手动解决冲突，并（在解决之后）将文件标记为已解决

```
git add <resolved-file>
git rm <resolved-file>
```

> **撤销**

1. 放弃工作目录中的所有本地更改

```
git reset --hard HEAD
```

2. 放弃特定文件中的本地更改

```
git checkout HEAD <file>
```

3. 还原提交

```
git revert <commit>
```

4. 将你的HEAD指针重置为上一次提交...并放弃此后的所有更改

```
git reset --hard <commit>
```

5. ...并将所有更改保留为未提交的更改

```
git reset <commit>
```

6. ...并保留未提交的本地更改

```
git reset --keep <commit>
```

版本控制最佳做法

> **提交相关更改**

提交应该是相关更改的封装。 例如，修复两个不同的 `BUG` 应产生两个单独的提交。 较小的提交可使其他开发人员更容易理解更改，并在出现问题时将其回滚。 借助暂存区和仅暂存文件部分的能力等工具， `Git` 可以轻松创建非常精细的提交。

> **经常提交**

经常提交会使你的提交变小，并且帮助你仅提交相关的更改。 而且，它使你可以更频繁地与他人共享代码。 这样，每个人都可以更轻松地定期同步整合代码更改，并避免合并冲突。 相比之下，很少的大量提交和共享代码，会导致很难解决冲突。

> **不要提交未完成的代码**

你应该只能在代码完成后提交。 但这并不意味着你在提交之前必须先完成一个完整的大型功能。 恰恰相反：将功能的实现分成逻辑块，并记住要尽可能早的频繁的提交。 但是，不要只想在一天结束离开办公室之前在仓库中提交一些代码。 如果你只是因为需要干净的工作区（切换分支，同步更改等）而打算提交未完成的代码，请考虑改用 `git stash` 。

> **提交代码之前一定要先测试**

在你还没有确认工作完成之前，忍住提交代码的冲动。彻底的测试代码，确保其没有副作用（据我们所知），虽然在本地存储库中提交半生不熟的东西只需要你原谅自己，但是当涉及到向他人推送/共享代码时，测试代码就更加重要了。

> **写”好”的提交注释**

提交注释的开头应该要用简短的总结语句（最多50个字符最为开头总结），后续正文与开头用空白行分开。注释的正文应该包含下面问题的详细答案：

本次修改代码的原因是什么？
本次代码与之前的实现有何不同?

> **版本控制不是备份系统**

在远程服务器上备份文件是版本控制系统的一个很好的副作用。但是你不应该把你的 `VCS` （版本控制系统）当成一个备份系统来使用。在进行版本控制时，你应该注意语义上的提交(参见相关章节)——你不应该只是填入文件。

> **多使用分支**

分支是 `Git` 最强大的功能之一，这并非偶然，因为从 `Git` 创建的第一天开始快速简单的分支就是它的中心要求。分支是帮助你避免混淆不同开发线的完美工具。你应该在开发工作流中广泛使用分支，例如，增加一个新的功能；修复 `BUG` ，新的想法等。

> **商定工作流程**

`Git` 允许你从许多不同的工作流中进行选择：长时间运行的分支、主题分支、合并或变基、git流.....你选择哪一个取决于几个因素：你的项目、你的整体开发和部署工作流程，以及(可能最重要的)你和你的队友的个人偏好。无论你选择如何工作，只要确保大家都同意一个通用的工作流程就可以了。

> **帮助与文档**

在命令行获取帮助

```
git help <command>
```

> **免费在线资源**

https://www.git-scm.com/docs
http://rogerdudler.github.io/git-guide/index.zh.html
https://www.git-tower.com/learn/git/ebook/cn/command-line/introduction

---

英文版（原版）

CHEAT SHEET

> **CREATE**

1. Clone an existing repository

```
git clone ssh://user@domain.com/repo.git
```

2. Create a new local repository

```
git init
```

LOCAL CHANGES

1. Changed files in your working directory

```
git status
```

2. Changes to tracked files

```
git diff
```

3. Add all current changes to the next commit

```
git add .
```

4. Add some changes in to the next commit

```
git add -p <file>
```

5. Commit all local changes in tracked files

```
git commit -a
```

6. Commit previously staged changes

```
git commit
```

7. Change the last commit
   Don‘t amend published commits!

```
git commit --amend
```

COMMIT HISTORY

1. Show all commits, starting with newest

```
git log
```

2. Show changes over time for a specific file

```
git log -p <file>
```

3. Who changed what and when in

```
git blame <file>
```

> **BRANCHES & TAGS**

1. List all existing branches

```
git branch -av
```

2. Switch HEAD branch

```
git checkout <branch>
```

3. Create a new branch based on your current HEAD

```
git branch <new-branch>
```

4. Create a new tracking branch based on a remote branch

```
git checkout --track <remote/branch>
```

5. Delete a local branch

```
git branch -d <branch>
```

6. Mark the current commit with a tag

```
git tag <tag-name>
```

> **UPDATE & PUBLISH**

1. List all currently configured remotes

```
git remote -v
```

2. Show information about a remote

```
git remote show <remote>
```

3. Add new remote repository, named

```
git remote add <shortname> <url>
```

4. Download all changes from , but don‘t integrate into HEAD

```
git fetch <remote>
```

5. Download changes and directly merge/integrate into HEAD

```
git pull <remote> <branch>
```

6. Publish local changes on a remote

```
git push <remote> <branch>
```

7. Delete a branch on the remote

```
git branch -dr <remote/branch>
```

8. Publish your tags

```
git push --tags
```

> **MERGE & REBASE**

1. Merge into your current HEAD

```
git merge <branch>
```

2. Rebase your current HEAD onto , Don‘t rebase published commits!

```
git rebase <branch>
```

3. Abort a rebase

```
git rebase --abort
```

4. Continue a rebase after resolving conflicts

```
git rebase --continue
```

5. Use your configured merge tool to solve conflicts

```
git mergetool
```

6. Use your editor to manually solve conflicts and (after resolving) mark file as resolved

```
git add <resolved-file>
git rm <resolved-file>
```

> **UNDO**

1. Discard all local changes in your working directory

```
git reset --hard HEAD
```

2. Discard local changes in a specific file

```
git checkout HEAD <file>
```

3. Revert a commit(by producing a new commit with contrary changes)

```
git revert <commit>
```

4. Reset your HEAD pointer to a previous commit...and discard all changes since then

```
git reset --hard <commit>
```

5. ...and preserve all changes as unstaged changes

```
git reset <commit>
```

6. ...and preserve uncommitted local changes

```
git reset --keep <commit>
```

VERSION CONTROL BEST PRACTICES

COMMIT RELATED CHANGES

A commit should be a wrapper for related changes. For example, fixing two different bugs should produce two separate commits. Small commits make it easier for other developers to understand the changes and roll them back if something went wrong. With tools like the staging area and the ability to stage only parts of a file, Git makes it easy to create very granular commits.

COMMIT OFTEN

Committing often keeps your commits small and, again, helps you commit only related changes. Moreover, it allows you to share your code more frequently with others. That way it‘s easier for everyone to integrate changes regularly and avoid having merge conflicts. Having few large commits and sharing them rarely, in contrast, makes it hard to solve conflicts.

DON‘T COMMIT HALF-DONE WORK

You should only commit code when it‘s completed. This doesn‘t mean you have to complete a whole, large feature before committing. Quite the contrary: split the feature‘s implementation into logical chunks and remember to commit early and often. But don‘t commit just to have something in the repository before leaving the office at the end of the day. If you‘re tempted to commit just because you need a clean working copy (to check out a branch, pull in changes, etc.) consider using Git‘s Stash feature instead.

TEST CODE BEFORE YOU COMMIT

Resist the temptation to commit some-thing that you «think» is completed. Test it thoroughly to make sure it really is completed and has no side effects (as far as one can tell). While committing halfbaked things in your

local repository only requires you to forgive yourself, having your code tested is even more important when it comes to pushing/sharing your code with others.

WRITE GOOD COMMIT MESSAGES

Begin your message with a short summary of your changes (up to 50 characters as a gui-deline). Separate it from the following body by including a blank line. The body of your message should provide detailed answers to the following questions:
› What was the motivation for the change?
› How does it differ from the previous implementation?

VERSION CONTROL IS NOT A BACKUP SYSTEM

Having your files backed up on a remote server is a nice side effect of having a version control system. But you should not use your VCS like it was a backup system. When doing version control, you should pay attention to committing semantically (see «related chan-ges») - you shouldn‘t just cram in files.

USE BRANCHES

Branching is one of Git‘s most powerful features - and this is not by accident: quick and easy branching was a central requirement from day one. Branches are the perfect tool to help you avoid mixing up different lines of development. You should use branches extensively in your development workflows: for new features, bug fixes, ideas...

AGREE ON A WORKFLOW

Git lets you pick from a lot of different work-flows: long-running branches, topic branches, merge or rebase, git-flow... Which one you choose depends on a couple of factors: your project, your overall development and deployment workflows and (maybe most importantly) on your and your teammates‘ personal preferences. However you choose to work, just make sure to agree on a common workflow that everyone follows.

HELP & DOCUMENTATION

Get help on the command line

```
git help <command>
```

FREE ONLINE RESOURCES

https://www.git-scm.com/docs
http://rogerdudler.github.io/git-guide/index.html
https://www.git-tower.com/learn/git/ebook/en/command-line/introduction

其他命令

总结

显示颜色提示

```
git config --global color.ui true
```

查看全部Git配置

```
git config --list
```