# 创造

克隆现有存储库

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

创建一个新的本地存储库

$ git init

# 当地变化

Changed files in your working directory

$ git status

Changes to tracked files

$ git diff

将所有当前更改添加到下一个提交

$ git add .

在<file>中添加一些更改到下一次提交

$ git add -p <file>

提交跟踪文件中的所有本地更改

$ git commit -a

提交先前进行的更改

$ git commit

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

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e

$ git log -p <file>

谁更改了<file>中的内容和时间

$ git blame <file>

# 分支机构和标签

列出所有现有分支

$ git branch -av

切换HEAD分支

$ git checkout <branch>

根据您当前的HEAD创建一个新分支

$ git branch <new-branch>

基于远程分支创建一个新的跟踪分支

$ git checkout --track <remote/branch>

删除本地分支

$ git branch -d <branch>

用标签标记当前提交

$ git tag <tag-name>

# 更新和发布

列出所有当前配置的遥控器

$ git remote -v

显示有关遥控器的信息

$ git remote show <remote>

添加名为<remote>的新远程存储库

$ git remote add <shortname> <url>

从<remote>下载所有更改，但不要集成到HEAD中

$ git fetch <remote>

下载更改并直接合并/集成到HEAD中

$ git pull <remote> <branch>

在远程上发布本地更改

$ git push <remote> <branch>

删除遥控器上的分支

$ git branch -dr <remote/branch>

发布标签

$ git push --tags

# 合并与基础

将<branch>合并到当前HEAD中

$ git merge <branch>

将当前的HEAD重新设置到<branch>

不要重新发布已发布的提交！

$ git rebase <branch>

中止基准

$ git rebase --abort

解决冲突后继续进行基准

$ git rebase --continue

使用您配置的合并工具解决冲突

$ git mergetool

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>

# 撤消

丢弃工作目录中的所有本地更改

$ git reset --hard HEAD

放弃特定文件中的本地更改

$ git checkout HEAD <file>

还原提交（通过产生具有相反更改的新提交）

$ git revert <commit>

将HEAD指针重置为上一次提交

…并丢弃此后的所有更改

$ git reset --hard <commit>

…并将所有更改保留为未分阶段的更改

$ git reset <commit>

…并保留未提交的本地更改

$ git reset --keep <commit>

|  |
| --- |
|  |

# 提交相关更改

提交应该是相关更改的包装。 例如，修复两个不同的错误应产生两个单独的提交。 小型提交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。 借助暂存区等工具以及仅暂存文件部分的功能，Git使得创建非常精细的提交变得容易。

# 经常提交

提交通常会使您的提交变小，并且再次帮助您仅提交相关的更改。 而且，它使您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集成更改，避免合并冲突。 相比之下，大型提交很少且很少共享，因此很难解决冲突。

# 不要做半成品

您只应在完成代码后提交代码。 这并不意味着您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，并记住提早并经常提交。 但是，不要承诺在一天结束离开办公室之前在存储库中存放一些东西。 如果您只是因为需要干净的工作副本（来签出分支，进行更改等）而打算提交，请考虑改用Git的“隐藏”功能。

# 提交之前的测试代码

抵制诱惑，以完成您“认为”的事情。 对其进行彻底的测试，以确保它确实完成并且没有副作用（据人们所知）。 虽然只需要原谅自己在本地存储库中提交的东西，但是在与他人推送/共享代码时，测试代码就显得尤为重要。

# 写好的提交信息

首先以简短的更改摘要（最多50个字符为准则）开始您的消息。 通过包含空白行将其与后续正文分开。 邮件正文应提供以下问题的详细答案：

›改变的动机是什么？

›它与以前的实现有何不同？

使用命令式现在时态（“ change”，而不是“ changed”或“ changes”）与git merge等命令生成的消息保持一致。

# 版本控制不是备份系统

将文件备份到远程服务器上是拥有版本控制系统的一个很好的副作用。 但是，您不应像将其用作备份系统那样使用VCS。 在进行版本控制时，您应注意语义上的提交（请参阅“相关更改”）-您不应该只是在文件中塞入。

# 使用分支

分支是Git最强大的功能之一-并不是偶然的：从一开始，快速便捷的分支就成为了中心需求。 分支是完美的工具，可帮助您避免混淆不同的开发路线。 您应该在开发工作流程中广泛使用分支：有关新功能，错误修复，想法…

# 同意工作流程

Git让您从许多不同的工作流程中进行选择：长期运行的分支，主题分支，合并或重新设置，git-flow…您选择哪个取决于两个因素：项目，整体开发和部署工作流程以及（也许 最重要的是），以了解您和您的队友的个人喜好。 无论您选择工作，只要确保在每个人都遵循的通用工作流程上达成一致即可。

# 帮助和文档

在命令行获取帮助

$ git help <command>

# 免费的在线资源

[*http://www.git-tower.com/learn*](http://www.git-tower.com/learn/?utm_source=Tower+Blog&utm_medium=cheat+sheet+pdf&utm_content=german+version&utm_campaign=Tower+website)

[*http://rogerdudler.github.io/git-guide/*](http://rogerdudler.github.io/git-guide/)

[*http://www.git-scm.org/*](http://www.git-scm.org/)