1. **GIT**
   1. **Git是什么**
2. Git是Linux之父Linus于2002年用C语言开发的分布式版本控制系统。

**官方教程中文完整版**

1. <https://git-scm.com/book/zh/v2>

**关于版本控制系统**

|  |  |  |
| --- | --- | --- |
| 类别 | 产品 | 缺点 |
| 本地版本控制系统VCS  Version Control Systems | RCS | 无法协同工作 |
| 集中式版本控制系统CVCS  Centralized Version Control Systems | CVS、Subversion、Perforce | 中央服务器的单点故障 |
| 分布式版本控制系统DVCS  Distributed Version Control System | Git、Mercurial、Bazaar、Darcs |  |

* 1. **安装Git**

**官方下载地址**

Git：<https://git-scm.com/downloads>

GitHub Desktop（包含Git）：<https://desktop.github.com>

**在Linux上安装Git**

Fedora：sudo yum install git

Debian/Ubuntu：sudo apt-get install git。

注：老一点的Debian/Ubuntu，要把命令改为sudo apt-get install git-core，因为以前有个软件也叫GIT（GNU Interactive Tools），结果Git就只能叫git-core了。由于Git名气实在太大，后来就把GNU Interactive Tools改成gnuit，git-core正式改为git。

如果是其他Linux版本，可以直接通过源码安装。先从Git官网下载源码，然后解压，依次输入：./config，make，sudo make install这几个命令安装就好了。

要了解更多选择，Git 官方网站上有在各种 Unix 风格的系统上安装步骤，网址为 <http://git-scm.com/download/linux>。

**在Mac OS X上安装Git**

方法一：是安装homebrew，然后通过homebrew安装Git，具体方法请参考homebrew的文档：<http://brew.sh/>。

方法二（推荐）：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。

Xcode是Apple官方IDE，功能非常强大，是开发Mac和iOS App的必选免费装备。

方法三：官网下载安装Git或GitHub Desktop。

**在Windows上安装Git**

官网下载安装Git或GitHub Desktop

**从源代码安装**

参考官方教程：1.5 [安装 Git](https://git-scm.com/book/zh/v2/%E8%B5%B7%E6%AD%A5-%E5%AE%89%E8%A3%85-Git)

* 1. **配置Git**

**三个配置级别**

|  |  |  |  |
| --- | --- | --- | --- |
| **配置级别** | **配置文件位置（Unix）** | **配置文件位置（Windows）** | **读写配置变量的git config命令参数** |
| 系统级别 | /etc/gitconfig | %Git安装目录%/etc/gitconfig | --system |
| 用户级别 | ~/.gitconfig 或  ~/.config/git/config | %USER\_HOME%/.gitconfig | --global |
| 仓库级别 | $仓库目录/.git/config | %仓库目录%/.git/config |  |

**注：范围小的配置会覆盖范围大的配置**

**配置用户名和邮件地址**

$ git config --global user.name "<Your Name>"

$ git config --global user.email ["<email\_adress>"](mailto:%22%3cemail@example.com%3e%22)（可不用引号标示？）

推荐邮件地址<Your Name>@users.noreply.github.com

**配置默认文本编辑器**

当 Git 需要你输入信息时会调用它。 如果未配置，Git 会使用操作系统默认的文本编辑器（ shell 的环境变量 $EDITOR 所指定的软件），Unix下通常是 Vim。

Unix配置emacs作为Git默认文本编辑器：git config --global core.editor emacs

Windows配置notepad++作为Git默认文本编辑器：

git config --global core.editor "'E:/Program Files/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin"

**让Git Bash显示颜色**

git config --global color.ui true

**查看配置信息**

查看所有：git config –list

查看某项：git config <key>

**获取命令用法信息**

获取所有：git –help

获取某项：git help <command>或git <command> --help

* 1. **Git基本概念**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **------------------------------------------------>** | | | | | | | | |
| 工作目录  Working Directory | 跟踪/暂存修改/  标记冲突已解决  add | 暂存区  Staging Area | 提交  commit | 版本库  Repository | 推送  push | 远程仓库  Remote/Origin | 拉取请求  pull request | 别人仓库 |
| checkout | | | fetch/clone | fork |
| pull | | | | |
| **<--------------------------------------------------** | | | | | | | | |



**使用 Git 时文件的生命周期**



**版本库**又称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最重要的部分，从其它计算机克隆仓库时，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**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。也称作“索引”。

版本库内有分支、指向第一个分支master的指针HEAD。

* 1. **常用Git命令**

|  |  |  |
| --- | --- | --- |
| **内容** | **命令** | |
| **创建版本库（repository）** | 目标目录下使用git init命令  注1）Windows系统请确保目录名（包括父目录）不包含中文  注2）创建后目标目录下会生成一个.git隐藏目录 | |
| **把文件添加到版本库** | 三种情况需要将文件添加到版本库：  1）新文件需要纳入版本管理2）已纳入版本管理的文件被修改3）合并代码冲突解决后需要标记已解决  第一步：使用命令git add <file/folder>将工作目录中的文件/目录添加到暂存区，可多次使用，或添加多个文件。  第二步：使用命令git commit -m "<message>"将暂存区文件提交到分支（未add的文件和修改将不会被提交）。  注：  1）如果git commit 不加-m参数，Git Bash会调用默认的或用户指定的文本编辑器来让用户输入message  2）git commit -v：在编辑器中编辑Git提交消息时能够看到更改的内容  3）git commit -a：跳过暂存区直接将工作区已跟踪文件的修改提交到版本库  4）git commit --amend：该命令可以修改最近一次的commit内容或信息，修改后commitID也会改变 | |
| **查看文件状态** | 查看详细的状态报告：git status  查看简洁的状态报告：git status -s或git status –short  简洁的状态报告解释：   |  |  | | --- | --- | | $ git status -s  ?? 1.txt  M 2.txt  M 3.txt  MM 4.txt  D 5.txt  D 6.txt  A 7.txt | ?? 文件未跟踪  M 文件已修改未暂存  M 文件已修改已暂存  MM 文件已修改已暂存后又修改未暂存  D 文件已删除未暂存  D 文件已删除已暂存  A 文件已暂存未提交 | | |
| **查看修改内容** | 查看工作区相对于暂存区的修改：git diff [<file>]  查看工作区相对于版本库的修改：git diff HEAD -- [<file>]  查看暂存区相对于版本库的修改：git diff --cached [<file>]或git diff --staged [<file>]（Git 1.6.1及以上）  注：Git Diff 的插件版本：  可以使用 git difftool 命令来用 Araxis ，emerge 或 vimdiff 等软件输出 diff 分析结果。 使用 git difftool --tool-help 命令来看你的系统支持哪些 Git Diff 插件。 | |
| **删除文件** | **对于未跟踪的文件：**rm <file>（慎重，没后悔药）  **对于已跟踪未提交的文件：**  git rm -f <file>：取消跟踪并删除工作区文件（-f参数用来丢弃已修改未提交的内容）  或  git rm --cached <file>：取消跟踪但保留工作区文件  **对于已提交的文件：**  第一步：  git rm [-f] <file>：取消跟踪并删除工作区文件  git rm --cached <file>：取消跟踪但保留工作区文件  第二步：git commit -m "<message>"  也可以先手动删除文件，然后git rm <file>或git add <file>，最后git commit -m "<message>" | |
| **移动/重命名文件** | git mv <file\_name> <new\_name>  注：运行 git mv README.md README就相当于运行了下面三条命令：  $ mv README.md README  $ git rm README.md  $ git add README | |
| **查看提交历史** | git log  HEAD指向的版本是当前版本。  要退出查看就输入底行命令q  查看按用户分组的提交说明历史，使用命令git shortlog  git log常用参数：   |  |  | | --- | --- | | **参数** | **说明** | | **定义显示内容** | | | 无参 | 显示哈希、作者及日期、说明 | | --pretty=medium  （等同无参） | 显示哈希、作者及日期、说明 | | --pretty=short | 显示哈希、作者、说明 | | --pretty=full | 显示哈希、作者、提交者、说明 | | --pretty=fuller | 显示哈希、作者及日期、提交者及日期、说明 | | --pretty=oneline | 显示哈希、说明于一行 | | --oneline | 显示简短哈希、说明于一行 | | -p | 额外显示内容差异 | | --stat | 额外显示各文件的和汇总的行数变化 | | --shortstat | 额外显示汇总的行数变化 | | --name-only | 显示修改的文件清单 | | --name-status | 额外显示新增、修改、删除的文件清单 | | --graph | 额外显示分支合并图 | | **定义显示格式** | | | --abbrev-commit | 指定显示简短哈希 | | --relative-date | 指定显示相对时间 | | —date | 指定日期显示格式 | | --pretty=format: | 自定义显示的内容和格式 | | **筛选提交历史** | | | -数字 | 查看最近指定次数的提交历史 | | --since/--after | 筛选指定时间点之后或指定最近时间范围之内的提交 | | --until/--before | 筛选指定时间点之前或指定最近时间范围之外的提交 | | --author | 筛选作者（模糊匹配） | | --committer | 筛选提交者（模糊匹配） | | --grep 关键字 | 筛选提交说明中的关键字 | | -S 关键字 | 筛选提交内容中的关键字 | | -G 正则表达式 | 筛选提交内容中的正则匹配字 | | --merges/--no-merges | 筛选已合并/未合并的提交 | | 文件或路径名  （无名参数放最后，多个用空格分隔） | 筛选指定文件或路径 | |  | 筛选指定分支 | |  | 筛选指定标签 | | --all-match | 显示各筛选结果的交集（鸡肋，本来就是交集） | | 参数--pretty=format的常用选项：   |  |  | | --- | --- | | **选项** | **说明** | | %H | 提交对象（commit）的完整哈希字串 | | %h | 提交对象的简短哈希字串 | | %T | 树对象（tree）的完整哈希字串 | | %t | 树对象的简短哈希字串 | | %P | 父对象（parent）的完整哈希字串 | | %p | 父对象的简短哈希字串 | | %an | 作者（author）的名字 | | %ae | 作者的电子邮件地址 | | %ad | 作者修订日期（可以用 --date= 选项定制格式） | | %ar | 作者修订的相对日期 | | %cn | 提交者（committer）的名字 | | %ce | 提交者的电子邮件地址 | | %cd | 提交日期 | | %cr | 提交的相对日期 | | %s | 提交说明 |   注：作者指实际修改人，提交者指实际提交人。  示例：  展示简洁的提交历史：  git log --pretty=format:"%h - %an, %ad : %s" --date=iso  展示酷炫的提交历史：  git log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cd) %C(bold blue)<%an>%Creset' --date=iso  筛选指定时间范围的提交历史：  git log --since "2019-12-13 22:00:00" --until "2019-12-13 23:16:00"  筛选最近两天内最近两小时外的提交历史：  git log --after 2day --before 2hour（单复数随意） |
| **查看HEAD历史** | git reflog  HEAD历史包括所有有效和无效的提交历史和重置历史 | |
| **版本回退（重置）** | **回到过去：**  回退前，可以先查看有效的提交历史，以便确定要回退到哪个版本。  git reset --soft <commit\_id>：版本库回退到指定历史  git reset [--mixed] <commit\_id>：版本库和暂存区回退到指定历史  git reset --hard <commit\_id>：版本库、暂存区和工作区都回退到指定历史  注：  回退到当前版本可用git reset [<参数>] HEAD 或git reset [<参数>] HEAD~0，  回退到上一版本可用git reset [<参数>] HEAD^ 或git reset [<参数>] HEAD~1，  以此类推。  **回到未来：**  先用git reflog查看操作历史，以便确定要回到未来的哪个版本。再用reset命令重返未来。 | |
| **版本撤销** | 如果想恢复到之前某个提交的版本，且那个版本之后提交的版本我们都不要了，就用版本回退的方法；  如果要撤销之前某个提交的版本，并且那个版本之后提交的版本我们还要，就用版本撤销的方法。  版本撤销的实质是以新的暂存来“反做”某个版本。  版本回退reset会重写提交记录，版本撤销revert不会重写提交记录。  版本撤销容易引起冲突。  git revert -n  注：-n指不要自动合并 | |
| **撤销增/删/改** | 撤销版本库未推送的增/删/改（撤销提交）：使用重置版本命令git reset --soft HEAD^  撤销暂存区未提交的增/删/改（撤销跟踪/暂存）：git restore --staged <file>或git reset HEAD <file>（旧）  撤销工作区未暂存的删/改（撤销删除/修改）：git restore [--worktree] <file>或git checkout -- <file>（旧）  注：注意git checkout -- <file>与git checkout <branch\_name>的区别（后者没有“--”，意为“切换到另一分支”）。 | |
| **把本地未push的分叉提交历史整理成直线** | git rebase | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 文件状态 | 未跟踪 | 已跟踪 | 已修改未暂存 | 已暂存未提交 | 已提交未推送 | 已推送 |
| 添加文件 | add |  |  | commit | push |  |
| 保存修改 |  |  |  |  |  |  |
| 撤销修改 |  |  |  |  |  |  |
| 删除文件 | rm |  |  | rm -f |  |  |
|  |  |  |  |  |  |  |

* 1. **忽略文件**

在Git工作区的根目录下创建一个特殊的.gitignore文件，然后把要忽略的文件名填进**去**，最后把.gitignore提交到Git，Git就会自动忽略这些文件。

GitHub为我们提供了各种配置文件模板：<https://github.com/github/gitignore>，Java的如下：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| # Compiled class file  \*.class  # Log file  \*.log  # BlueJ files  \*.ctxt  # Mobile Tools for Java (J2ME)  .mtj.tmp/  # Package Files #  \*.jar  \*.war  \*.nar  \*.ear  \*.zip  \*.tar.gz  \*.rar  # virtual machine crash logs, see http://www.java.com/en/download/help/error\_hotspot.xml  hs\_err\_pid\* | 忽略文件的原则是：  忽略操作系统自动生成的文件，比如缩略图等；  忽略编译生成的中间文件、可执行文件等；  忽略你自己的带有敏感信息的配置文件，比如存放口令的配置文件。  文件 .gitignore 的格式规范如下：  所有空行或者以 ＃ 开头的行都会被 Git 忽略。  可以使用标准的 glob 模式匹配。  匹配模式可以以（/）开头防止递归。  匹配模式可以以（/）结尾指定目录。  要忽略指定模式以外的文件或目录，可以在模式前加上惊叹号（!）取反。  所谓的 glob 模式是指 shell 所使用的简化了的正则表达式，例：   |  |  | | --- | --- | | \* | 匹配零个或多个任意字符 | | [abc] | 要么匹配 字符a，要么匹配字符 b，要么匹配字符 c | | \*.[oa] | 忽略所有以 .o 或 .a 结尾的文件 | | [0-9] | 匹配所有 0 到 9 的数字 | | a/\*\*/z | 匹配任意中间目录 | |

如果要添加被.gitignore忽略的文件到Git，使用-f参数强制添加，例如：git add -f App.class

查看某文件有没有被.gitignore忽略：git check-ignore -v <file\_name>

* 1. **连接GitHub远程仓库**

**创建并添加SSH Key**

为什么GitHub需要SSH Key呢？因为GitHub需要识别出你推送的提交确实是你推送的，而不是别人冒充的，而Git支持SSH协议，所以，GitHub只要知道了你的公钥，就可以确认只有你自己才能推送。

当然，GitHub允许你添加多个Key。假定你有若干电脑，你一会儿在公司提交，一会儿在家里提交，只要把每台电脑的Key都添加到GitHub，就可以在每台电脑上往GitHub推送了。

最后友情提示，在GitHub上免费托管的Git仓库，任何人都可以看到喔（但只有你自己才能改）。所以，不要把敏感信息放进去。

如果你不想让别人看到Git库，有两个办法，一个是交点保护费，让GitHub把公开的仓库变成私有的，这样别人就看不见了（不可读更不可写）。另一个办法是自己动手，搭一个Git服务器，因为是你自己的Git服务器，所以别人也是看不见的。这个方法我们后面会讲到的，相当简单，公司内部开发必备。

**创建SSH Key**

在用户主目录下，看看有没有.ssh目录及目录下有没有id\_rsa和id\_rsa.pub两个文件，如果已经有了，可直接跳到下一步。如果没有，打开Shell（Windows下打开Git Bash），创建SSH Key：ssh-keygen -t rsa -C "<email@example.com>"，然后一路回车，使用默认值即可，由于这个Key也不是用于军事目的，所以也无需设置密码。如果一切顺利的话，可以在用户主目录里找到.ssh目录，里面有id\_rsa和id\_rsa.pub两个文件，这两个就是SSH Key的秘钥对，id\_rsa是私钥，不能泄露出去，id\_rsa.pub是公钥，可以放心地告诉任何人。

**添加SSH Key**

登陆GitHub -> 点右上角头像 -> Settings -> SSH and GPG Keys -> New SSH key -> Title自定义，在Key文本框里粘贴id\_rsa.pub文件的内容 -> 点“Add SSH Key”-> 完成！

**创建远程库**

右上角+号 -》New repository -》填入库名例如learngit，其他默认 -》Create repository -》完成！

目前，在GitHub上的这个learngit仓库还是空的，GitHub告诉我们，可以从这个仓库克隆出新的仓库，也可以把一个已有的本地仓库与之关联，然后，把本地仓库的内容推送到GitHub仓库。

**克隆远程库**

SSH协议克隆（默认）：git clone git@<server-name>:<path>/<repo-name>.git [<local-repo-name>]

HTTPS协议克隆：git clone https://<server-name>/<path>/<repo-name>.git [<local-repo-name>]

首先，登陆GitHub，创建一个新的仓库，名字如gitskills，勾选Initialize this repository with a README

然后，在本地执行Git命令：git clone git@github.com:tongwx/gitskills.git或git clone https://github.com/tongwx/gitskills.git

1. 默认配置下Git的克隆操作是首先将远程库的所有文件的所有版本克隆到本地版本库的.git目录下，因此如果服务器挂了，可以使用任一克隆下来的用户端来重建服务器上的仓库（可能会丢失某些服务器端的挂钩设置）；其次将最新版本的文件放入工作目录中。
2. 使用https协议传输速度慢，每次推送都必须输入口令，但是在某些只开放http端口的公司内部就无法使用ssh协议而只能用https。
3. 当你从远程仓库克隆时，实际上Git自动把本地的master分支和远程的master分支对应起来了，并且，远程仓库的默认名称是origin。
4. 若要克隆时指定远程库名，使用参数-o
5. 从远程库clone后，默认只能看到本地的master分支（？），若要创建本地dev分支并关联远程dev分支：git checkout -b dev origin/dev
6. 若要关联本地分支与远程分支：git branch --set-upstream-to=origin/<origin\_branch\_name> <local\_branch\_name>

**关联远程库**

本地命令：git remote add origin git@<server-name>:<path>/<repo-name>.git

例如：git remote add origin git@github.com:tongwx/learngit.git

**查看远程库**

查看简略的远程库列表：git remote

查看详细的远程库列表：git remote -v

查看某个远程库信息：git remote show 远程库名

获得远程引用的完整列表：git ls-remote [<remote>]

如果提示fast-forwardable，说明可以推送当前分支。

**拉取自远程库**

获取：git fetch 默认只获取远程的默认分支，以及其他分支的指针。

拉取：git pull

git merge <remote>/<branch>

**推送到远程库**

本地命令：

第一次推送：git push -u origin master:[<origin\_branch\_name>]

非第一次推送：git push origin master

注1）Git默认用origin表示远程库。

注2）第一次推送master分支时，加上了-u参数，Git不但会把本地的master分支内容推送的远程新的master分支，还会把本地的master分支和远程的master分支关联起来，在以后的推送或者拉取时就可以简化命令。

注3）第一次使用Git的clone或者push命令连接GitHub时，会提示确认GitHub的Key的指纹信息是否真的来自GitHub的服务器，输入yes回车即可。其后Git会输出一个警告，告诉你已经把GitHub的Key添加到本机的一个信任列表里了。

如何避免每次输入密码：

如果你正在使用 HTTPS URL 来推送，Git 服务器会询问用户名与密码。 默认情况下它会在终端中提示服务器是否允许你进行推送。

如果不想在每一次推送时都输入用户名与密码，你可以设置一个 “credential cache”。 最简单的方式就是将其保存在内存中几分钟，可以简单地运行 git config --global credential.helper cache 来设置它。

想要了解更多关于不同验证缓存的可用选项，查看 凭证存储。

**重命名或移除远程库**

重命名远程库：git remote rename 现名 新名

移除远程库：git remote rm 远程库名

**远程库克隆别人的项目**

1. 在别人项目主页，比如bootstrap项目：<https://github.com/twbs/bootstrap> 点“Fork”，就可以克隆别人项目到自己的账户下。
2. 从自己的账户下克隆到本地：本地命令：git clone git@github.com:<自己的path>/bootstrap.git
3. 如果你希望别人接受你的修改，你就可以在GitHub上发起一个pull request。当然，对方是否接受你的pull request就不一定了。
   1. **使用码云**

和GitHub相比，码云也提供免费的Git仓库。此外，还集成了代码质量检测、项目演示、私有库等功能。对于团队协作开发，码云还提供了项目管理、代码托管、文档管理的服务，5人以下小团队免费。

码云地址：<https://gitee.com/>

上传公钥：登陆码云 -> 点右上角头像 -> 设置 -> SSH公钥 -> 添加公钥 -> 标题自定义，在公钥文本框里粘贴id\_rsa.pub文件的内容 -> 点“确定”-> 完成！

创建项目：右上角+号 -》新建仓库 -》填入库名例如learngit，其他默认 -》点“创建” -》完成！

关联本地仓库：例如：git remote add origin git@gitee.com:tongwx/learngit.git

码云同样支持Fork和pull request

**一个本地库如何同时关联GitHub和码云**

1. 步骤：
2. 1）先删除已关联的名为origin的远程库（如有）：git remote rm origin
3. 2）然后关联GitHub的远程库（注意取名为github）：例如：git remote add github git@github.com:tongwx/learngit.git
4. 3）再关联码云的远程库（注意取名为gitee）：例如：git remote add gitee git@gitee.com:tongwx/learngit.git
5. 现在就关联了两个远程库：
6. git remote -v
7. gitee git@gitee.com:tongwx/learngit.git (fetch)
8. gitee git@gitee.com:tongwx/learngit.git (push)
9. github git@github.com:tongwx/learngit.git (fetch)
10. github git@github.com:tongwx/learngit.git (push)
11. 如果要推送到GitHub，使用命令：git push github master
12. 如果要推送到码云，使用命令：git push gitee master
13. 如果本地仓库和远程仓库实际上是独立的两个仓库，会提示fatal: refusing to merge unrelated histories
14. 此时使用--allow-unrelated-histories参数来合并：git push gitee master --allow-unrelated-histories
    1. **分支管理**

**原理**

Git的指针：HEAD -> branch指针 -> commit指针 -> tree指针 -> blob文件快照

Git有个特殊指针叫HEAD，指向本地的当前分支指针。每个分支指针指向各分支的最新提交对象。每个提交对象包含一个指向不同树对象的指针，每个树对象包含指向不同文件快照的指针。

当Git里只有一条时间线即默认的master分支时，HEAD指针指向分支指针master，master指针指向分支最新的提交。

每次提交，master分支都会向前移动一步，这样，随着你不断提交，master分支的线也越来越长。

当我们创建并切换到新的分支例如dev时，Git新建了一个分支指针叫dev，指向与master指针相同指向的提交，再把HEAD指针指向dev指针，就表示当前分支在dev上。

你看，Git创建并切换一个分支很快，因为除了增加一个dev指针，改改HEAD的指向，工作区的文件都没有任何变化！

不过，从现在开始，对工作区的修改和提交就是针对dev分支了，比如新提交一次后，dev指针往前移动一步，而master指针不变。

当我们在dev上的工作完成，切换到master并把dev合并到master上时，Git把HEAD指向master指针，把master指针指向dev指针指向的当前提交，就完成了合并。

所以Git合并分支也很快！就改改指针，工作区内容也不变！

合并完分支后，甚至可以删除dev分支。删除dev分支就是把dev指针给删掉，删掉后，我们就剩下了一条master分支。

**命令**

查看分支：git branch 注：当前分支前面会标一个\*号。

创建分支：git branch <name>原理：新建一个指针指向最新提交

切换分支：git checkout <name>或者git switch <name> 注：switch为较新版本命令。原理：HEAD指向目标分支指针

创建+切换分支：git checkout -b <name>或者git switch -c <name> 注：switch为较新版本命令。

合并某分支到当前分支：git merge <name>

合并某分支到当前分支（禁用Fast forward模式）：git merge --no-ff -m "<message>" <name>

注：

**快进合并**：执行结果信息如果显示Fast-forward表示该合并是快进模式。

由于当前分支指针所指向的提交是被合并分支指针所指向提交的直接上游（两个分支的所有提交能连成一条线），Git就直接把当前分支的指针快速前进指向被合并分支的指针所指向的提交。

简言之，两个分支在一条线上（没分叉，我先你后），就可以通过指针移动进行快进合并。

快进合并模式下删除分支后会丢失历史分支信息。

**三方合并**：执行结果信息如果显示Merge made by the 'recursive' strategy表示该合并是三方合并。

由于当前分支指针所指向的提交不是被合并分支指针所指向提交的直接上游（两个分支出现分叉提交），Git就会使用两个分支的最新提交快照以及出现分支前的最后提交快照进行三方合并，把合并结果做成一个新的快照并自动创建一个新的提交（合并提交）记录合并的快照，再把两个分支的指针都前进指向这个合并提交。

简言之，两个分支不在一条线上（有分叉），就需要将两分支的最新快照和最近的共同祖先快照进行三方合并并创建新的提交，再把两个分支的指针都前进指向这个合并提交。

三方合并从分支历史上可以看出分支信息。

**合并冲突**：

如果两个分支对同一个文件的相同部分作出不同修改，将出现合并冲突，合并将暂停，等待解决冲突后手动暂存和提交合并内容。

删除已被合并的分支：git branch -d <name>原理：删除该指针

删除未被合并的分支：git branch -D <name>**（慎重）**原理：删除该指针

拉取分支：git pull

推送分支：git push origin <name>

**分支策略**

在实际开发中，我们应该按照几个基本原则进行分支管理：

干活都在dev分支上，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；

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。

如果在dev分支上干活的时候发现master分支有bug需要修复，就在master分支上新建fix分支进行修复，然后回到master分支把fix分支的修复合并进master分支，最后回到dev分支上继续干活。

**分支开发流程、命令、涉及的分支原理**

**总体流程**

1）上线某个项目，使用一个线上分支master，用来发布新版本，平时不能在上面干活。

2）为实现某个新的需求，创建一个工作分支dev，在该分支上开展工作。

3）完成需求后，切换到master分支，合并dev分支，在master分支发布版本。

如果dev分支的工作尚未完成，突然发现线上分支有个严重问题需要紧急修补，就按如下方式来处理：

1）切换到线上分支。

2）为这个紧急任务新建一个修补分支，并在其中修复它。

3）在测试通过之后，切换回线上分支，然后合并这个修补分支，最后将改动推送到远程线上分支。

4）切换回你最初工作的分支上，把线上分支合并的改动复制到工作分支，然后继续工作。或者先把工作分支的工作做完，再切换到master分支，合并dev分支，在master分支发布版本。

**具体流程解析**

|  |  |  |
| --- | --- | --- |
| **步骤** | **命令** | **涉及的分支原理** |
| 使用线上分支master发布项目 |  | HEAD指向master  master指向分支的最新提交  **工作目录文件内容与master分支所指向的快照内容一致** |
| 为实现某个新的需求，创建并切换到一个工作分支dev | git checkout -b dev | 创建dev指针指向master所指向的提交  HEAD指向dev  dev和master分别指向各自分支的最新提交（当前指向同一个提交）  **工作目录文件内容无变化，与dev分支所指向的快照内容一致** |
| 在dev分支上开展工作 |  | 随着在dev分支上不断提交，dev不断指向该分支的最新提交  fix指向的提交不断领先于master指向的提交  **工作目录文件内容不断更新** |
| 发现线上分支有问题需紧急修复 |  |  |
| 先在当前dev分支把工作现场储藏起来 | git stash | 切换分支前，必须先处理当前分支尚未暂存和提交的修改：要么丢弃修改，要么提交修改，要么储藏修改。否则工作目录的文件内容在切换分支前后会起冲突。  **工作目录将未提交的修改移至储藏区** |
| 切换到master分支 | git checkout master | HEAD指向master  master指向的提交落后于dev指向的提交  **工作目录文件内容回退到未进行dev开发时的状态** |
| 从master分支创建并切换到fix分支 | git checkout -b fix | 创建fix指针指向master所指向的提交  HEAD指向fix |
| 在fix分支上进行修复 |  | 随着在fix分支上不断提交，fix不断指向该分支的最新提交  fix指向的提交不断领先于master指向的提交  **工作目录文件内容不断更新直到fix修复完成** |
| 切换回master分支 | git checkout master | HEAD指向master  master指向的提交落后于fix指向的提交  **工作目录文件内容回退到未进行fix修复时的状态** |
| 合并fix分支 | git merge [--no-ff -m "<message>"] fix | 快进合并：master快进指向fix所指向的提交（Fast-forward）  三方合并：先自动创建一个合并提交，再把master和fix都指向这个提交  **工作目录文件内容更新到fix修复完成后的状态** |
| 删除fix分支 | git branch -d fix | 删除fix指针 |
| 切换回dev分支 | git checkout dev | HEAD指向dev  **工作目录文件内容变成dev开发时的最后一次提交状态** |
| 恢复工作现场并删除储藏 | git stash pop  （等同于：git stash apply和git stash drop）  可以多次git stash，恢复的时候，先查看储藏列表：git stash list，然后恢复指定的储藏：git stash apply <stash\_name> | **工作目录恢复储藏的修改** |
| 将master分支上的修复复制到dev分支  或者dev分支开发完成后切换到master分支将dev分支并入 | git cherry-pick <commit\_id on fix>  或：  git commit  git checkout master  git merge [--no-ff -m "<message>"] dev | **工作目录将master分支上fix修复所做的修改复制进来**  或：  **工作目录将未提交的修改提交**  **工作目录变成fix修复完成后的状态**  **工作目录将dev分支上所有提交的修改合并进来**  **（由于出现提交分叉，将进行三方合并）** |

* 1. **冲突管理**

当合并分支发生冲突时，Git用<<<<<<< 分支名，=======，>>>>>>> 分支名 标记出不同分支的内容。

需要手工修改冲突内容，然后暂存、提交之。

如果想使用图形化工具来解决冲突，可以运行 git mergetool，根据提示选择喜欢的工具来解决冲突。等你退出合并工具之后，Git 会询问刚才的合并是否成功。 如果你回答是，Git 会暂存那些文件以表明冲突已解决。你可以再次运行 git status 来确认所有的合并冲突都已被解决。如果你对结果感到满意，并且确定之前有冲突的的文件都已经暂存了，这时你可以输入 git commit 来完成合并提交。

用带参数的git log --graph可以看到分支合并图：git log --graph --pretty=oneline --abbrev-commit

* 1. **标签管理**

Git 可以给重要的提交打上标签，是提交的引用。通常在发布一个版本时打一个标签。

**打轻量标签：**

在最新提交上打标签：git tag <name>

在某个提交上打标签：git tag <name> <commit\_id>

**打附注标签**：可以创建带有说明的标签，用-a指定标签名，-m指定说明文字：git tag -a v0.1 -m "version 0.1 released" 1094adb

**查看标签**

查看标签列表：git tag

搜索标签：git tag -1 关键字

查看某个标签信息：git show <tagname>

**推送标签**

推送某个标签到远程：git push <remote> <tagname>

推送全部尚未推送的标签到远程：git push <remote> --tags

**删除标签**：先删本地：git tag -d <tagname>；再删远程（如有）：git push <remote> :refs/tags/<tagname>

**检出标签**

如果你想查看某个标签所指向的文件版本，可以使用 git checkout <tagname>命令，但这会使你的仓库处于“分离头指针（detacthed HEAD）”状态：在“分离头指针”状态下，如果你做了某些更改然后提交它们，标签不会发生变化，但你的新提交将不属于任何分支，并且将无法访问，除非确切的提交哈希。因此，如果你需要进行更改——比如说你正在修复旧版本的错误——这通常需要创建一个新分支：git checkout -b version2 <tagname>，当然，如果在这之后又进行了一次提交，version2 分支会因为这个改动向前移动，version2 分支就会和<tagname>标签稍微有些不同，这时就应该当心了。

* 1. **配置命令别名**

**在命令行配置命令别名**

git config [--global] alias.<别名> <原命令>

如果要配置别名的原命令中间有空格，需将原命令用引号（单双皆可）标示。

如果要配置外部命令的别名，需在外部命令前加个感叹号。

例：配置一键添加、提交、推送所有修改的命令别名（不知道为啥命令不能用双引号标示）：

git config --global alias.go '!git add .; git commit; git push'

**在配置文件中配置命令别名**

全局配置（当前用户）：在用户主目录下的文件.gitconfig（命令行全局配置后会生成）中的[alias]节点中配置，每行一个：别名 = 命令名。

局部配置（当前仓库）：在仓库.git目录下的文件config中的[alias]节点中配置，每行一个：别名 = 命令名。

* 1. **搭建Git服务器**

搭建Git服务器需要准备一台运行Linux的机器，强烈推荐用Ubuntu或Debian，这样，通过几条简单的apt命令就可以完成安装。

假设你已经有sudo权限的用户账号，下面，正式开始安装。

**第一步**，安装git：

$ sudo apt-get install git

**第二步**，创建一个git用户，用来运行git服务：

$ sudo adduser git

**第三步**，创建证书登录：

收集所有需要登录的用户的公钥，就是他们自己的id\_rsa.pub文件，把所有公钥导入到/home/git/.ssh/authorized\_keys文件里，一行一个。

**第四步**，初始化Git仓库：

先选定一个目录作为Git仓库，假定是/srv/sample.git，在/srv目录下输入命令：

$ sudo git init --bare sample.git

Git就会创建一个裸仓库，裸仓库没有工作区，因为服务器上的Git仓库纯粹是为了共享，所以不让用户直接登录到服务器上去改工作区，并且服务器上的Git仓库通常都以.git结尾。然后，把owner改为git：

$ sudo chown -R git:git sample.git

**第五步**，禁用shell登录：

出于安全考虑，第二步创建的git用户不允许登录shell，这可以通过编辑/etc/passwd文件完成。找到类似下面的一行：

git:x:1001:1001:,,,:/home/git:/bin/bash

改为：

git:x:1001:1001:,,,:/home/git:/usr/bin/git-shell

这样，git用户可以正常通过ssh使用git，但无法登录shell，因为我们为git用户指定的git-shell每次一登录就自动退出。

**第六步**，克隆远程仓库：

现在，可以通过git clone命令克隆远程仓库了，在各自的电脑上运行：

$ git clone git@<server>:/srv/sample.git

剩下的推送就简单了。

要方便管理公钥，用[Gitosis](https://github.com/sitaramc/gitolite)。

要像SVN那样控制权限，用[Gitolite](https://github.com/sitaramc/gitolite)。

* 1. **使用SourceTree**

SourceTree是由Atlassian开发的免费Git图形界面工具，可以操作任何Git库。

**下载**

下载地址：<https://www.sourcetreeapp.com/>

**SourceTree导入密钥**

**导入OpenSSH密钥**

SourceTree主界面-》工具-》选项-》SSH客户端选择OpenSSH后会自动加载安装Git时创建的id\_rsa密钥。

**导入PuTTY/Plink密钥**

SourceTree主界面-》工具-》创建或导入SSH密钥-》Load-》选择所有文件类型，选择%USER\_HOME%\.ssh\id\_rsa-》save private key-》保存为%USER\_HOME%\.ssh\id\_rsa.ppk-》关闭当前界面回到主界面-》工具-》选项-》SSH客户端选择PuTTY/Plink后会自动加载刚创建的id\_rsa.ppk密钥。

Git 忽略已跟踪文件的改动

git update-index --assume-unchanged

Git之本地忽略

这个分同种情况：

本地永久忽略，效果的gitignore一样，只不过不适于写到gitignore中而已，可以自己建立一个本地独享的gitignore，然后git config --global core.excludesfile  文件的绝对路径，也可以直接将本地要忽略的文件添加到.git/info/exclude中。

不过上述都是针对没有跟踪的文件来说的，如果文件已经被跟踪了你如果在本地想要忽略它的改动，就不能使用以上的方法了。通俗地讲比如一个编译Android的脚本在其它电脑上都是使用的-j32来编译的，但是你的电脑配置没有别人的好，不能开到-j32，但是这个脚本是已经跟踪过的，你修改了就会在每次的git status中看到。对于这种情况Git有一个忽略改动的方法：

$ git update-index --assume-unchanged /path/to/file       #忽略跟踪

$ git update-index --no-assume-unchanged /path/to/file  #恢复跟踪

如下所示，

Lenovo@LENOVO-PC /c/WorkSpace2/BodyBuilding (master)

$ git status

On branch master

Your branch is up-to-date with 'origin/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:   BodyBuilding.iml

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

   modified:   BodyBuilding.iml，这个文件是处于modified状态，现在要忽略他的改动，如下，

Lenovo@LENOVO-PC /c/WorkSpace2/BodyBuilding (master)

$ git update-index --assume-unchanged BodyBuilding.iml

Lenovo@LENOVO-PC /c/WorkSpace2/BodyBuilding (master)

$ git statusOn branch master

Your branch is up-to-date with 'origin/master'.

nothing to commit, working directory clean

Lenovo@LENOVO-PC /c/WorkSpace2/BodyBuilding (master)

$

好了，完成了。这个文件的改动就不会被显示出来。

=================END=================