# git与其它版本管理软件的区别

## 网址

https://git-scm.com/book/zh/v2/起步-Git-基础

## git特点

Git 更像是把数据看作是对小型文件系统的一组快照。每次你提交更新，或在Git 中保存项目状态时，它主要对当时的全部文件制作一个快照并保存这个快照的索引。 为了高效，如果文件没有修改，Git 不再重新存储该文件，而是只保留一个链接指向之前存储的文件。 Git 对待数据更像是一个 **快照流**。

## 其它软件管理特点

(CVS、Subversion、Perforce、Bazaar 等等)将它们保存的信息看作是一组基本文件和每个文件随时间逐步累积的差异。

# git的三种状态及三个工作区域

## 三种状态

### 已提交（committed）

已提交表示数据已经安全的保存在本地数据库中。如果 Git 目录中保存着的特定版本文件，就属于已提交状态。

### 已修改（modified）

已修改表示修改了文件，但还没保存到数据库中。如果自上次取出后，作了修改但还没有放到暂存区域，就是已修改状态。

### 已暂存（staged）

已暂存表示对一个已修改文件的当前版本做了标记，使之包含在下次提交的快照中。如果作了修改并已放入暂存区域，就属于已暂存状态。

## 三个工作区域

### 工作目录

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。比如master、dev分支

### 暂存区域

暂存区域是一个文件，保存了下次将提交的文件列表信息，一般在 Git 仓库目录中。有时候也被称作`‘索引’'，不过一般说法还是叫暂存区域。git add后添加到的地方

### Git 仓库.

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 中最重要的部分，从其它计算机克隆仓库时，拷贝的就是这里的数据。git commit后的地方。

## git基本工作流程

* 在工作目录中修改文件。
* 暂存文件，将文件的快照放入暂存区域。
* 提交更新，找到暂存区域的文件，将快照永久性存储到 Git 仓库目录。

|  |
| --- |
|  |

# git安装及配置

## windows安装git

* 在Windows上安装 Git 也有几种安装方法。 官方版本可以在 Git 官方网站下载。 打开 <http://git-scm.com/download/win>，下载会自动开始。 要注意这是一个名为 Git for Windows的项目（也叫做 msysGit），和 Git 是分别独立的项目；更多信息请访问 <http://msysgit.github.io/>。

## windows配置文件

在 Windows 系统中，Git 会查找 $HOME 目录下（一般情况下是 C:\Users\$USER）的 .gitconfig 文件。 Git 同样也会寻找 /etc/gitconfig 文件，但只限于 MSys 的根目录下，即安装 Git 时所选的目标位置。

## linux安装git

* 看最上面网址

## linux配置文件

Git 自带一个 git config 的工具来帮助设置控制 Git 外观和行为的配置变量。 这些变量存储在三个不同的位置：

* /etc/gitconfig 文件: 包含系统上每一个用户及他们仓库的通用配置。 如果使用带有 --system 选项的 git config 时，它会从此文件读写配置变量。
* ~/.gitconfig 或 ~/.config/git/config 文件：只针对当前用户。 可以传递 --global 选项让 Git 读写此文件。
* 当前使用仓库的 Git 目录中的 config 文件（就是 .git/config）：针对该仓库。

# git配置

## 用户信息

当安装完 Git 应该做的第一件事就是设置你的用户名称与邮件地址。 这样做很重要，因为每一个 Git 的提交都会使用这些信息，并且它会写入到你的每一次提交中，不可更改：

* $ git config --global user.name "John Doe"
* $ git config --global user.email [johndoe@example.com](mailto:johndoe@example.com)

再次强调，如果使用了 --global 选项，那么该命令只需要运行一次，因为之后无论你在该系统上做任何事情， Git 都会使用那些信息。 当你想针对特定项目使用不同的用户名称与邮件地址时，可以在那个项目目录下运行没有 --global 选项的命令来配置。

## 文本编辑器

既然用户信息已经设置完毕，你可以配置默认文本编辑器了，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，通常是 Vim。 如果你想使用不同的文本编辑器，例如 Emacs，可以这样做：

* $ git config --global core.editor emacs

## 检查配置信息

如果想要检查你的配置，可以使用 git config --list 命令来列出所有 Git 当时能找到的配置。

* $ git config –list

你可能会看到重复的变量名，因为 Git 会从不同的文件中读取同一个配置（例如：/etc/gitconfig 与 ~/.gitconfig）。 这种情况下，Git 会使用它找到的每一个变量的最后一个配置。

你可以通过输入 git config <key>： 来检查 Git 的某一项配置：

* $ git config user.name

## 获取帮助

* $ git help <verb>
* $ git <verb> --help
* $ man git-<verb>
* 例如，要想获得 config 命令的手册，执行$ git help config

# 获取git仓库

## 方法一：$ git init

是在现有项目或目录下导入所有文件到Git中，该命令将创建一个名为 .git 的子目录，这个子目录含有你初始化的 Git 仓库中所有的必须文件，这些文件是Git仓库的骨干。 但是，在这个时候，我们仅仅是做了一个初始化的操作，你的项目里的文件还没有被跟踪。

## 方法二：$ git clone [uri [在本地的目录名]]

从一个服务器克隆一个现有的 Git 仓库，默认配置下远程 Git 仓库中的每一个文件的每一个版本都将被拉取下来。这会在当前目录下创建一个名为 “libgit2” 的目录，并在这个目录下初始化一个 .git 文件夹，从远程仓库拉取下所有数据放入 .git 文件夹，然后从中读取最新版本的文件的拷贝。 如果你进入到这个新建的 libgit2 文件夹，你会发现所有的项目文件已经在里面了，准备就绪等待后续的开发和使用。

# git操作

## 已跟踪与未跟踪

工作目录下的每一个文件都不外乎这两种状态：已跟踪或未跟踪。

### 已跟踪：tracked files

是指那些被纳入了版本控制的文件，在上一次快照中有它们的记录，在工作一段时间后，它们的状态可能处于未修改，已修改或已放入暂存区。初次克隆某个仓库的时候，工作目录中的所有文件都属于已跟踪文件，并处于未修改状态。

### 未跟踪：Untracked files

除已跟踪文件以外的所有其它文件都属于未跟踪文件，它们既不存在于上次快照的记录中，也没有放入暂存区。

## git生命周期

编辑过某些文件之后，由于自上次提交后你对它们做了修改，Git 将它们标记为已修改文件。 我们逐步将这些修改过的文件放入暂存区，然后提交所有暂存了的修改，如此反复。所以使用 Git 时文件的生命周期如下：

|  |
| --- |
|  |

## 文件的状态变化周期：git status

* 查看文件状态：$ git status。
  + 或者git status -s 、git status --short。这两个命令可以以简单形式显示。
    - 如果文件修改并放入暂存区，会左边显示M。
    - 修改了未放入暂存区，右边显示M。
    - 修改了放入暂存区，又进行了修改，则显示MM。

## git add

* 开始跟踪新文件，或者把已跟踪的文件放到暂存区，还能用于合并时把有冲突的文件标记为已解决状态等，将这个命令理解为“添加内容到下一次提交中”而不是“将一个文件添加到项目中”要更加合适：git add 文件名/目录。

## 忽略文件

有些文件无需纳入 Git 的管理，也不希望它们总出现在未跟踪文件列表。我们可以创建一个名为 .gitignore 的文件，列出要忽略的文件模式。

* 注意windows创建.gitignore步骤：
  + 在项目根目录下面创建gitignore.txt文件
  + 把你需要排除的文件名保存到gitignore.txt文件
  + 在项目根目录下面按住Shift键并邮件然后选择“在此处打开命令窗口”
  + 执行命令 ren gitignore.txt .gitignore
* 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。
* 匹配方式
  + 所谓的 glob 模式是指 shell 所使用的简化了的正则表达式。
  + 星号（\*）匹配零个或多个任意字符；
  + [abc] 匹配任何一个列在方括号中的字符（这个例子要么匹配一个 a，要么匹配一个 b，要么匹配一个 c）；
  + 问号（?）只匹配一个任意字符；
  + 如果在方括号中使用短划线分隔两个字符，表示所有在这两个字符范围内的都可以匹配（比如 [0-9] 表示匹配所有 0 到 9 的数字）。
  + 使用两个星号（\*) 表示匹配任意中间目录，比如`a/\*\*/z` 可以匹配 a/z, a/b/z 或 `a/b/c/z`等。

## 查看已暂存和未暂存的修改：git diff

* git diff：查看所有已跟踪文件的未暂存修改。此命令比较的是工作目录中当前文件和暂存区域快照之间的差异，也就是修改之后还没有暂存起来的变化内容。显示结果上面一行为工作目录的已跟踪文件，下面一行为修改地方。
  + --cached(或git diff --staged)：若要查看已暂存的将要添加到下次提交里的内容。

## git commit

* git commit -m “此次提交的说明”：将暂存区的文件放入本地git仓库，注意如果一个已跟踪文件做了修改后git add 文件，再次做了修改，git commit 时提交的是第一次的文件(即已经存放到暂存区的文件)。

|  |
| --- |
|  |
| [哪个分支 本次提交的完整 SHA-1 校验和]  几个改变 几行增加 几行删除 |

* 对已跟踪文件跳过git add 直接commit：git commit -a -m “此次提交说明”

## 移除文件或目录：git rm

* git rm 文件名：要从 Git 中移除某个文件，就必须要从已跟踪文件清单中移除（确切地说，是从暂存区域移除），然后提交git commit，也会将文件从工作目录删除。可以用 git rm 命令完成此项工作，并连带从工作目录中删除指定的文件，这样以后就不会出现在未跟踪文件清单中了。
  + -f 文件名：如果要删除的文件已经在暂存区进行了一次git add操作(比如修改)，则要删除此文件必须用此命令。
  + --cached 文件名：将文件从git仓库删除，但依然保留在工作目录，即git不再追踪此文件。
  + 命令后面可以列出文件或者目录的名字，也可以使用 glob 模式。

## 文件移动：mv

* mv a.txt b.txt：删除a.txt 新增b.txt

## 文件改名：git mv

* git mv file\_from file\_to：底层其实是用了三条命令
  + $ mv file\_from file\_to
  + $ git rm file\_from
  + $ git add file\_to

## 查看提交历史：git log

* git log：

|  |
| --- |
| commit ca82a6dff817ec66f44342007202690a93763949  Author: Scott Chacon <schacon@gee-mail.com>  Date: Mon Mar 17 21:52:11 2008 -0700  changed the version number |
| 默认不用任何参数的话，git log 会按提交时间列出所有的更新，最近的更新排在最上面。 正如你所看到的，这个命令会列出每个提交的 SHA-1 校验和、作者的名字和电子邮件地址、提交时间以及提交说明。 |

* git log --pretty=format：可以定制要显示的记录格式，例如git log --pretty=format:"%h - %an, %ar : %s"。常用选项如下

|  |  |  |  |
| --- | --- | --- | --- |
| 选项 | 说明 | 选项 | 说明 |
| %H | 提交对象（commit）的完整哈希字串 | %an | 作者（author）的名字 |
| %h | 提交对象的简短哈希字串 | %ae | 作者的电子邮件地址 |
| %T | 树对象（tree）的完整哈希字串 | %ad | 作者修订日期（可以用 --date= 选项定制格式） |
| %t | 树对象的简短哈希字串 | %ar | 作者修订日期（可以用 --date= 选项定制格式） |
| %P | 父对象（parent）的完整哈希字串 | %cn | 提交者（committer）的名字 |
| %p | 父对象（parent）的完整哈希字串 | %ce | 提交者的电子邮件地址 |
| %cd | 提交日期 | %cr | 提交日期，按多久以前的方式显示 |
| %s | 提交说明 |  |  |

* 总结，git log常用选项：

|  |  |  |
| --- | --- | --- |
| 选项 | 说明 | 实例 |
| -p | 按补丁格式显示每个更新之间的差异。 | git log -p |
| --stat | 显示每次更新的文件修改统计信息。 | git log --stat |
| --shortstat | 只显示 --stat 中最后的行数修改添加移除统计。 | git log --shortstat |
| --name-only | 仅在提交信息后显示已修改的文件清单。 | git log --name-only |
| --name-status | 显示新增、修改、删除的文件清单。 | git log --name-status |
| --abbrev-commit | 仅显示 SHA-1 的前几个字符，而非所有的 40 个字符。 | git log --abbrev-commit |
| --relative-date | 使用较短的相对时间显示（比如，“2 weeks ago”）。 | git log --relative-date |
| --graph | 显示 ASCII 图形表示的分支合并历史。 | git log --graph |
| --pretty | 使用其他格式显示历史提交信息。可用的选项包括 oneline，short，full，fuller 和 format（后跟指定格式）。 | git log --pretty=oneline |

* git log 限制输出选项：

|  |  |  |
| --- | --- | --- |
| 选项 | 说明 | 实例 |
| -(n) | 仅显示最近的n条提交 |  |
| --since, --after | 仅显示指定时间之后的提交 |  |
| --util, --before | 仅显示指定时间之前的提交 |  |
| --author | 仅显示指定作者相关的提交 |  |
| --committer | 仅显示指定提交者相关的提交 |  |
| --grep | 仅显示含指定关键字的提交 |  |
| -s | 仅显示添加或移除了某个关键字的提交 |  |

## 撤销操作

* + git commit --amend：这个命令会将暂存区中的文件提交。 如果自上次提交以来你还未做任何修改（例如，在上次提交后马上执行了此命令），那么快照会保持不变，而你所修改的只是提交信息。$ git commit -m 'initial commit $ git add forgotten\_file $ git commit --amend最终你只会有一个提交 - 第二次提交将代替第一次提交的结果。
  + git reset HEAD 文件名：将文件取消暂存。
  + git checkout --文件名：撤销此文件在工作目录的修改，本质是拷贝了另一个文件来覆盖它。

## 远程仓库的使用

* git remote：它会列出你指定的每一个远程服务器的简写。 如果你已经克隆了自己的仓库，那么至少应该能看到 origin - 这是 Git 给你克隆的仓库服务器的默认名字。
  + -v：会显示需要读写远程仓库使用的 Git 保存的简写与其对应的 URL。
  + add <shortname> <url>：添加一个新的远程 Git 仓库，同时指定一个你可以轻松引用的简写，例如git remote add pb https://.....
  + show [remote-name]：这个命令列出了当你在特定的分支上执行 git push 会自动地推送到哪一个远程分支。 它也同样地列出了哪些远程分支不在你的本地，哪些远程分支已经从服务器上移除了，还有当你执行 git pull 时哪些分支会自动合并。
  + rename 修改前的名 修改后的名：远程仓库的重命名。
  + rm remote-name(或其简写)：想要移除一个远程仓库 - 你已经从服务器上搬走了或不再想使用某一个特定的镜像了，又或者某一个贡献者不再贡献了
* git fetch url(或shortname)：拉取远程仓库有，但本地没有的信息。执行完成后，你将会拥有那个远程仓库中所有分支的引用，可以随时合并或查看。如果你使用 clone 命令克隆了一个仓库，命令会自动将其添加为远程仓库并默认以 “origin” 为简写。 所以git fetch origin 会抓取克隆（或上一次抓取）后新推送的所有工作。
* git push [remote-name] [branch-name]：推到仓库的某个分支

## 打标签：git tag

* git tag：列出已有的标签。
  + -l ‘v1.8.5\*’：git自身标签。

### git主要的两种类型标签

#### 轻量标签(lightweight)

一个轻量标签很像一个不会改变的分支 - 它只是一个特定提交的引用。

git tag v1.4-lw：创建轻量标签。

git show v1.4-lw：显示轻量标签

#### 附注标签(annotated)

附注标签是存储在 Git 数据库中的一个完整对象。 它们是可以被校验的；其中包含打标签者的名字、电子邮件地址、日期时间；还有一个标签信息；并且可以使用 GNU Privacy Guard （GPG）签名与验证。 通常建议创建附注标签，这样你可以拥有以上所有信息。

* + git tag -a 版本名 -m ‘信息’，例如git tag -a v1.4 -m ‘my version 1.4’。注意-m，及信息必须有。

### 给过去提交打标签

git tag -a 标签名 校验和：例如git tag -a v1.2 9fceb02

### 将标签推送到远程仓库

git push 远程仓库名 标签名：例如git push origin v1.5。因为git push命令并不会传送标签到远程仓库服务器。必须显示得将标签共享到服务器。

### 推送多个标签

git push 远程仓库 --tags：例git push origin --tags，会将所有不再远程仓库服务器得标签全部传送过去。

### 删除本地标签

git tag -d 标签名：例git tag -d v1.4-ld。

### 删除远程标签

在删除本地得情况下，git push <remote> :refs/tags/<tagname>：例git push origin :refs/tags/v1.4-lw

### 检出标签，查看某个标签所指向得文件版本

git checkout <tagname>：例git checkout 2.0.0，但是有副作用，会使仓库处于“分离头指针(detached HEAD)”状态，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 v2.0.0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 v2.0.0 标签稍微有些不同，这时就应该当心了。

## git别名

作用是简化命令，比如将checkout设置为co

### 设置命令别名

* + git config --global alias.别名 命令：例1git config --global alias.co checkout。
  + 例2 git config --global alias.sts ‘status -s’，不是一个单词得话，要加单引号
  + 执行外部命令：git config --global alias.visual ‘!gitk’。想要执行外部命令要在外部命令前加!

# git分支