工作区 working directory

版本库 repository

缓存区 stage(或者index)

第一分支 master

指针 HEAD



窗体顶端

# Git 基础

### 直接记录快照，而非差异比较

Git 和其它版本控制系统（包括 Subversion 和近似工具）的主要差别在于 Git 对待数据的方法。 概念上来区分，其它大部分系统以文件变更列表的方式存储信息。 这类系统（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。



**其它版本控制系统.存储每个文件与初始版本的差异.**

Git 不按照以上方式对待或保存数据。 反之，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**。



**Git. 存储项目随时间改变的快照.**

这是 Git 与几乎所有其它版本控制系统的重要区别。 因此 Git 重新考虑了以前每一代版本控制系统延续下来的诸多方面。 Git 更像是一个小型的文件系统，提供了许多以此为基础构建的超强工具，而不只是一个简单的 VCS。 稍后我们在[Git 分支](https://git-scm.com/book/zh/v2/ch00/_git_branching)讨论 Git 分支管理时，将探究这种方式对待数据所能获得的益处。

### 近乎所有操作都是本地执行

在 Git 中的绝大多数操作都只需要访问本地文件和资源，一般不需要来自网络上其它计算机的信息。 如果你习惯于所有操作都有网络延时开销的集中式版本控制系统，Git 在这方面会让你感到速度之神赐给了 Git 超凡的能量。 因为你在本地磁盘上就有项目的完整历史，所以大部分操作看起来瞬间完成。

举个例子，要浏览项目的历史，Git 不需外连到服务器去获取历史，然后再显示出来——它只需直接从本地数据库中读取。 你能立即看到项目历史。 如果你想查看当前版本与一个月前的版本之间引入的修改，Git 会查找到一个月前的文件做一次本地的差异计算，而不是由远程服务器处理或从远程服务器拉回旧版本文件再来本地处理。

这也意味着你离线或者没有 VPN 时，几乎可以进行任何操作。 如你在飞机或火车上想做些工作，你能愉快地提交，直到有网络连接时再上传。 如你回家后 VPN 客户端不正常，你仍能工作。 使用其它系统，做到如此是不可能或很费力的。 比如，用 Perforce，你没有连接服务器时几乎不能做什么事；用 Subversion 和 CVS，你能修改文件，但不能向数据库提交修改（因为你的本地数据库离线了）。 这看起来不是大问题，但是你可能会惊喜地发现它带来的巨大的不同。

### Git 保证完整性

Git 中所有数据在存储前都计算校验和，然后以校验和来引用。 这意味着不可能在 Git 不知情时更改任何文件内容或目录内容。 这个功能建构在 Git 底层，是构成 Git 哲学不可或缺的部分。 若你在传送过程中丢失信息或损坏文件，Git 就能发现。

Git 用以计算校验和的机制叫做 SHA-1 散列（hash，哈希）。 这是一个由 40 个十六进制字符（0-9 和 a-f）组成字符串，基于 Git 中文件的内容或目录结构计算出来。 SHA-1 哈希看起来是这样：

24b9da6552252987aa493b52f8696cd6d3b00373

Git 中使用这种哈希值的情况很多，你将经常看到这种哈希值。 实际上，Git 数据库中保存的信息都是以文件内容的哈希值来索引，而不是文件名。

### Git 一般只添加数据

你执行的 Git 操作，几乎只往 Git 数据库中增加数据。 很难让 Git 执行任何不可逆操作，或者让它以任何方式清除数据。 同别的 VCS 一样，未提交更新时有可能丢失或弄乱修改的内容；但是一旦你提交快照到 Git 中，就难以再丢失数据，特别是如果你定期的推送数据库到其它仓库的话。

这使得我们使用 Git 成为一个安心愉悦的过程，因为我们深知可以尽情做各种尝试，而没有把事情弄糟的危险。 更深度探讨 Git 如何保存数据及恢复丢失数据的话题，请参考[撤消操作](https://git-scm.com/book/zh/v2/ch00/_undoing)。

### 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。

已修改（modified）：已修改表示修改了文件，但还没保存到数据库中。

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。

由此引入 Git 项目的三个工作区域的概念：Git 仓库、工作目录以及暂存区域。



工作目录 暂存区域 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。

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。

**基本的 Git 工作流程如下：**

1. 在工作目录中修改文件。
2. 暂存文件，将文件的快照放入暂存区域。
3. 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。

如果 Git 目录中保存着的特定版本文件，就属于已提交状态。 如果作了修改并已放入暂存区域，就属于已暂存状态。 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。

### 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 "chenyue"

$ git config --global user.email [qq14487710000@163.com](mailto:qq14487710000@163.com)

### 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。

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

$ git config user.name

John Doe

## 获取帮助

若你使用 Git 时需要获取帮助，有三种方法可以找到 Git 命令的使用手册：

$ 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

有两种取得 Git 项目仓库的方法。 第一种是在现有项目或目录下导入所有文件到 Git 中； 第二种是从一个服务器克隆一个现有的 Git 仓库。

### 在现有目录中初始化仓库

如果你打算使用 Git 来对现有的项目进行管理，你只需要进入该项目目录并输入：

$ git init

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

### 克隆现有的仓库

如果你想获得一份已经存在了的 Git 仓库的拷贝，比如说，你想为某个开源项目贡献自己的一份力，这时就要用到 git clone 命令。Git 克隆的是该 Git 仓库服务器上的几乎所有数据，而不是仅仅复制完成你的工作所需要文件。 当你执行 git clone 命令的时候，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。。

git clone [url] 。

比如，要克隆 Git 的可链接库 libgit2，可以用下面的命令：

$ git clone https://github.com/libgit2/libgit2

这会在当前目录下创建一个名为 “libgit2” 的目录。

如果你想在克隆远程仓库的时候，自定义本地仓库的名字，你可以使用如下命令：

$ git clone https://github.com/libgit2/libgit2 mylibgit

这将执行与上一个命令相同的操作，不过在本地创建的仓库名字变为 mylibgit。

Git 支持多种数据传输协议。 上面的例子使用的是 https:// 协议，不过你也可以使用 git:// 协议或者使用 SSH 传输协议，比如 user@server:path/to/repo.git 。

记录每次更新到仓库

请记住，你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。

已跟踪的文件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。 工作目录中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。 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。

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

### 检查当前文件状态

要查看哪些文件处于什么状态，可以用 git status 命令。 如果在克隆仓库后立即使用此命令，会看到类似这样的输出：

$ git status

### 状态简览

git status 命令的输出十分详细，但其用语有些繁琐。 如果你使用 git status -s 命令或 git status --short 命令，你将得到一种更为紧凑的格式输出。 运行 git status -s ，状态报告输出如下：

$ git status -s

### 忽略文件

一般我们总会有些文件无需纳入 Git 的管理，也不希望它们总出现在未跟踪文件列表。 通常都是些自动生成的文件，比如日志文件，或者编译过程中创建的临时文件等。 在这种情况下，我们可以创建一个名为 .gitignore 的文件，列出要忽略的文件模式。 来看一个实际的例子：

$ cat .gitignore

\*.[oa]

\*~

### 查看已暂存和未暂存的修改

要查看尚未暂存的文件更新了哪些部分，不加参数直接输入 git diff：

$ git diff

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。

若要查看已暂存的将要添加到下次提交里的内容，可以用

$ git diff --cached

（Git 1.6.1 及更高版本还允许使用 git diff --staged，效果是相同的，但更好记些。）

### 提交更新

现在的暂存区域已经准备妥当可以提交了。

$ git commit

你也可以在 commit 命令后添加 -m 选项，将提交信息与命令放在同一行，如下所示：

$ git commit -m "Story 182: Fix benchmarks for speed"

给 git commit 加上 -a 选项，Git 就会自动把所有已经跟踪过的文件暂存起来一并提交，从而跳过 git add 步骤

$ git commit -a -m “added new benchmarks”

### 移除文件

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 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。

下一次提交时，该文件就不再纳入版本管理了。 如果删除之前修改过并且已经放到暂存区域的话，则必须要用强制删除选项 -f（译注：即 force 的首字母）。 这是一种安全特性，用于防止误删还没有添加到快照的数据，这样的数据不能被 Git 恢复。

另外一种情况是，我们想把文件从 Git 仓库中删除（亦即从暂存区域移除），但仍然希望保留在当前工作目录中。 换句话说，你想让文件保留在磁盘，但是并不想让 Git 继续跟踪。 当你忘记添加 .gitignore 文件，不小心把一个很大的日志文件或一堆 .a 这样的编译生成文件添加到暂存区时，这一做法尤其有用。 为达到这一目的，使用 --cached 选项：

$ git rm --cached README

git rm 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。 比方说：

$ git rm log/\\*.log

注意到星号 \* 之前的反斜杠 \， 因为 Git 有它自己的文件模式扩展匹配方式，所以我们不用 shell 来帮忙展开。 此命令删除 log/ 目录下扩展名为 .log 的所有文件。 类似的比如：

$ git rm \\*~

该命令为删除以 ~ 结尾的所有文件。

### 移动文件

不像其它的 VCS 系统，Git 并不显式跟踪文件移动操作。 如果在 Git 中重命名了某个文件，仓库中存储的元数据并不会体现出这是一次改名操作。 不过 Git 非常聪明，它会推断出究竟发生了什么，至于具体是如何做到的，我们稍后再谈。

既然如此，当你看到 Git 的 mv 命令时一定会困惑不已。 要在 Git 中对文件改名，可以这么做：

$ git mv file\_from file\_to

它会恰如预期般正常工作。 实际上，即便此时查看状态信息，也会明白无误地看到关于重命名操作的说明：

$ git mv README.md README

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

其实，运行 git mv 就相当于运行了下面三条命令：

$ mv README.md README

$ git rm README.md

$ git add README

如此分开操作，Git 也会意识到这是一次改名，所以不管何种方式结果都一样。 两者唯一的区别是，mv 是一条命令而另一种方式需要三条命令，直接用 git mv 轻便得多。 不过有时候用其他工具批处理改名的话，要记得在提交前删除老的文件名，再添加新的文件名。

git config –-global user.name “\*\*name”

git config –-global user.email “\*\*email”

mkdir \*\*filename//make directory 创建文件夹

touch \*\*filename 创建文件

cd \*\*filename //change directory 切换文件目录

pwd \*\*filename //print working dir 显示文件目录

ls –a //list

git init 将当前目录当成版本库

git add \*\*filename 添加文件到缓存区

git commit –m “\*\*comment” 提交文件到版本库

git status 查看当前缓存区状态

git diff 查看修改

git diff HEAD **--** \*\*filename

看工作区和版本库与最新版本的区别

git log –-pretty=oneline 显示当前日志

git reflog 显示历史日志

git reset –-hard \*\*commit\_id(HEAD^)（HEAD~10)

退回到对应版本号

git reset HEAD \*\*filename 撤销缓存区的修改（unstage）

git checkout -- \*\*filename

将工作区退回到最近一次的add或者commit时的状态

git rm \*\*filename //remove 删除文件并添加到缓存区

$ ssh-keygen -t rsa -C "youremail@example.com"

创建GitHub SSH KEY

.ssh/id\_rsa.pub公钥内容在GitHub中add SHH key

在GitHub中Create New Repository创建新的Git仓库

$ git remote add origin https://github.com/chenyue1993/learngit.git

将本地仓库关联到github仓库

origin:远程仓库默认名称

chneyue1993:Github账户名

$ git push -u origin master

Counting objects: 19, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (19/19), done.

Writing objects: 100% (19/19), 13.73 KiB, done.

Total 23 (delta 6), reused 0 (delta 0)

To git@github.com:michaelliao/learngit.git

\* [new branch] master -> master

分支

建立分支 git branch <name>

切换分支 git checkout <name>

创建并切换 git checkout –b <name>

合并分支（注意当前是那个分支，是否需要切换分支）

git merge <name> 当前合并了name

git rebase <name> 当前复制到name

Git远程操作详解

作者： [阮一峰](http://www.ruanyifeng.com/)

日期： [2014年6月12日](http://www.ruanyifeng.com/blog/2014/06/)

[Git](http://zh.wikipedia.org/wiki/Git)是目前最流行的[版本管理系统](http://www.ruanyifeng.com/blog/2008/12/a_visual_guide_to_version_control.html)，学会Git几乎成了开发者的必备技能。

Git有很多优势，其中之一就是远程操作非常简便。本文详细介绍5个Git命令，它们的概念和用法，理解了这些内容，你就会完全掌握Git远程操作。

* git clone
* git remote
* git fetch
* git pull
* git push

本文针对初级用户，从最简单的讲起，但是需要读者对Git的基本用法有所了解。同时，本文覆盖了上面5个命令的几乎所有的常用用法，所以对于熟练用户也有参考价值。

## 一、git clone

远程操作的第一步，通常是从远程主机克隆一个版本库，这时就要用到git clone命令。

$ git clone <版本库的网址>

比如，克隆jQuery的版本库。

$ git clone https://github.com/jquery/jquery.git

该命令会在本地主机生成一个目录，与远程主机的版本库同名。如果要指定不同的目录名，可以将目录名作为git clone命令的第二个参数。

$ git clone <版本库的网址> <本地目录名>

git clone支持多种协议，除了HTTP(s)以外，还支持SSH、Git、本地文件协议等，下面是一些例子。

$ git clone http[s]://example.com/path/to/repo.git/

$ git clone ssh://example.com/path/to/repo.git/

$ git clone git://example.com/path/to/repo.git/

$ git clone /opt/git/project.git

$ git clone file:///opt/git/project.git

$ git clone ftp[s]://example.com/path/to/repo.git/

$ git clone rsync://example.com/path/to/repo.git/

SSH协议还有另一种写法。

$ git clone [user@]example.com:path/to/repo.git/

通常来说，Git协议下载速度最快，SSH协议用于需要用户认证的场合。各种协议优劣的详细讨论请参考[官方文档](http://git-scm.com/book/en/Git-on-the-Server-The-Protocols)。

## 二、git remote

为了便于管理，Git要求每个远程主机都必须指定一个主机名。git remote命令就用于管理主机名。

不带选项的时候，git remote命令列出所有远程主机。

$ git remote

origin

使用-v选项，可以参看远程主机的网址。

$ git remote -v

origin [git@github](mailto:git@github).com:jquery/jquery.git (fetch)

origin [git@github](mailto:git@github).com:jquery/jquery.git (push)

上面命令表示，当前只有一台远程主机，叫做origin，以及它的网址。

克隆版本库的时候，所使用的远程主机自动被Git命名为origin。如果想用其他的主机名，需要用git clone命令的-o选项指定。

$ git clone -o jQuery https://github.com/jquery/jquery.git

$ git remote

jQuery

上面命令表示，克隆的时候，指定远程主机叫做jQuery。

git remote show命令加上主机名，可以查看该主机的详细信息。

$ git remote show <主机名>

git remote add命令用于添加远程主机。

$ git remote add <主机名> <网址>

git remote rm命令用于删除远程主机。

$ git remote rm <主机名>

git remote rename命令用于远程主机的改名。

$ git remote rename <原主机名> <新主机名>

## 三、git fetch

一旦远程主机的版本库有了更新（Git术语叫做commit），需要将这些更新取回本地，这时就要用到git fetch命令。

$ git fetch <远程主机名>

上面命令将某个远程主机的更新，全部取回本地。

git fetch命令通常用来查看其他人的进程，因为它取回的代码对你本地的开发代码没有影响。

默认情况下，git fetch取回所有分支（branch）的更新。如果只想取回特定分支的更新，可以指定分支名。

$ git fetch <远程主机名> <分支名>

比如，取回origin主机的master分支。

$ git fetch origin master

所取回的更新，在本地主机上要用"远程主机名/分支名"的形式读取。比如origin主机的master，就要用origin/master读取。

git branch命令的-r选项，可以用来查看远程分支，-a选项查看所有分支。

$ git branch -r

origin/master

$ git branch -a

\* master

remotes/origin/master

上面命令表示，本地主机的当前分支是master，远程分支是origin/master。

取回远程主机的更新以后，可以在它的基础上，使用git checkout命令创建一个新的分支。

$ git checkout -b newBrach origin/master

上面命令表示，在origin/master的基础上，创建一个新分支。

此外，也可以使用git merge命令或者git rebase命令，在本地分支上合并远程分支。

$ git merge origin/master

# 或者

$ git rebase origin/master

上面命令表示在当前分支上，合并origin/master。

## 四、git pull

git pull命令的作用是，取回远程主机某个分支的更新，再与本地的指定分支合并。它的完整格式稍稍有点复杂。

$ git pull <远程主机名> <远程分支名>:<本地分支名>

比如，取回origin主机的next分支，与本地的master分支合并，需要写成下面这样。

$ git pull origin next:master

如果远程分支是与当前分支合并，则冒号后面的部分可以省略。

$ git pull origin next

上面命令表示，取回origin/next分支，再与当前分支合并。实质上，这等同于先做git fetch，再做git merge。

$ git fetch origin

$ git merge origin/next

在某些场合，Git会自动在本地分支与远程分支之间，建立一种追踪关系（tracking）。比如，在git clone的时候，所有本地分支默认与远程主机的同名分支，建立追踪关系，也就是说，本地的master分支自动"追踪"origin/master分支。

Git也允许手动建立追踪关系。

git branch --set-upstream master origin/next

上面命令指定master分支追踪origin/next分支。

如果当前分支与远程分支存在追踪关系，git pull就可以省略远程分支名。

$ git pull origin

上面命令表示，本地的当前分支自动与对应的origin主机"追踪分支"（remote-tracking branch）进行合并。

如果当前分支只有一个追踪分支，连远程主机名都可以省略。

$ git pull

上面命令表示，当前分支自动与唯一一个追踪分支进行合并。

如果合并需要采用rebase模式，可以使用--rebase选项。

$ git pull --rebase <远程主机名> <远程分支名>:<本地分支名>

如果远程主机删除了某个分支，默认情况下，git pull 不会在拉取远程分支的时候，删除对应的本地分支。这是为了防止，由于其他人操作了远程主机，导致git pull不知不觉删除了本地分支。

但是，你可以改变这个行为，加上参数 -p 就会在本地删除远程已经删除的分支。

$ git pull -p

# 等同于下面的命令

$ git fetch --prune origin

$ git fetch -p

## 五、git push

git push命令用于将本地分支的更新，推送到远程主机。它的格式与git pull命令相仿。

$ git push <远程主机名> <本地分支名>:<远程分支名>

注意，分支推送顺序的写法是<来源地>:<目的地>，所以git pull是<远程分支>:<本地分支>，而git push是<本地分支>:<远程分支>。

如果省略远程分支名，则表示将本地分支推送与之存在"追踪关系"的远程分支（通常两者同名），如果该远程分支不存在，则会被新建。

$ git push origin master

上面命令表示，将本地的master分支推送到origin主机的master分支。如果后者不存在，则会被新建。

如果省略本地分支名，则表示删除指定的远程分支，因为这等同于推送一个空的本地分支到远程分支。

$ git push origin :master

# 等同于

$ git push origin --delete master

上面命令表示删除origin主机的master分支。

如果当前分支与远程分支之间存在追踪关系，则本地分支和远程分支都可以省略。

$ git push origin

上面命令表示，将当前分支推送到origin主机的对应分支。

如果当前分支只有一个追踪分支，那么主机名都可以省略。

$ git push

如果当前分支与多个主机存在追踪关系，则可以使用-u选项指定一个默认主机，这样后面就可以不加任何参数使用git push。

$ git push -u origin master

上面命令将本地的master分支推送到origin主机，同时指定origin为默认主机，后面就可以不加任何参数使用git push了。

不带任何参数的git push，默认只推送当前分支，这叫做simple方式。此外，还有一种matching方式，会推送所有有对应的远程分支的本地分支。Git 2.0版本之前，默认采用matching方法，现在改为默认采用simple方式。如果要修改这个设置，可以采用git config命令。

$ git config --global push.default matching

# 或者

$ git config --global push.default simple

还有一种情况，就是不管是否存在对应的远程分支，将本地的所有分支都推送到远程主机，这时需要使用--all选项。

$ git push --all origin

上面命令表示，将所有本地分支都推送到origin主机。

如果远程主机的版本比本地版本更新，推送时Git会报错，要求先在本地做git pull合并差异，然后再推送到远程主机。这时，如果你一定要推送，可以使用--force选项。

$ git push --force origin

上面命令使用--force选项，结果导致远程主机上更新的版本被覆盖。除非你很确定要这样做，否则应该尽量避免使用--force选项。

最后，git push不会推送标签（tag），除非使用--tags选项。

$ git push origin --tags