**Git 教程**

由 一水寒 创建，Alma 最后一次修改 2015-09-14

**Git 教程**



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。

**Git 与 SVN 区别**

GIT不仅仅是个版本控制系统，它也是个内容管理系统(CMS),工作管理系统等。

如果你是一个具有使用SVN背景的人，你需要做一定的思想转换，来适应GIT提供的一些概念和特征。

Git 与 SVN 区别点：

* 1、GIT是分布式的，SVN不是：这是GIT和其它非分布式的版本控制系统，例如SVN，CVS等，最核心的区别。
* 2、GIT把内容按元数据方式存储，而SVN是按文件：所有的资源控制系统都是把文件的元信息隐藏在一个类似.svn,.cvs等的文件夹里。
* 3、GIT分支和SVN的分支不同：分支在SVN中一点不特别，就是版本库中的另外的一个目录。
* 4、GIT没有一个全局的版本号，而SVN有：目前为止这是跟SVN相比GIT缺少的最大的一个特征。
* 5、GIT的内容完整性要优于SVN：GIT的内容存储使用的是SHA-1哈希算法。这能确保代码内容的完整性，确保在遇到磁盘故障和网络问题时降低对版本库的破坏。

**Git 快速入门**

本站也提供来Git快速入门版本，你可以点击 [Git简明指南](https://www.w3cschool.cn/manual/git-guide/) 查看。

入门后建议通过本站详细学习 Git 教程。

Git 完整命令手册地址：[http://git-scm.com/docs](https://git-scm.com/docs)

PDF 版命令手册：[github-git-cheat-sheet.pdf](https://www.w3cschool.cn/statics/demosource/github-git-cheat-sheet.pdf)

**相关文章推荐**

* 1、[Git 五分钟教程](https://www.w3cschool.cn/w3cnote/git-five-minutes-tutorial.html)
* 2、[Git GUI使用方法](https://www.w3cschool.cn/w3cnote/git-gui-window.html)
* 3、[Github 简明教程](https://www.w3cschool.cn/w3cnote/git-guide.html)
* 4、[互联网组织的未来：剖析GitHub员工的任性之源](https://www.w3cschool.cn/w3cnote/internet-organization-github.html)

# Git 安装配置

由 一水寒 创建，Alma 最后一次修改 2017-08-01 16:52:27

## Git 安装配置

在使用Git前我们需要先安装 Git。Git 目前支持 Linux/Unix、Solaris、Mac和 Windows 平台上运行。

Git 各平台安装包下载地址为：[http://git-scm.com/downloads](https://git-scm.com/downloads)

## Linux 平台上安装

Git 的工作需要调用 curl，zlib，openssl，expat，libiconv 等库的代码，所以需要先安装这些依赖工具。

在有 yum 的系统上（比如 Fedora）或者有 apt-get 的系统上（比如 Debian 体系），可以用下面的命令安装：

各 Linux 系统可以很简单多使用其安装包管理工具进行安装：

### Debian/Ubuntu

Debian/Ubuntu Git 安装命令为：

$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \

libz-dev libssl-dev

$ apt-get install git-core

$ git --version

git version 1.8.1.2

### Centos/RedHat

如果你使用的系统是 Centos/RedHat 安装命令为：

$ yum install curl-devel expat-devel gettext-devel \

openssl-devel zlib-devel

$ yum -y install git-core

$ git --version

git version 1.7.1

## Windows 平台上安装

在 Windows 平台上安装 Git 同样轻松，有个叫做 msysGit 的项目提供了安装包，可以到 GitHub 的页面上下载 exe 安装文件并运行：

安装包下载地址：[http://msysgit.github.io/](https://msysgit.github.io/)



完成安装之后，就可以使用命令行的 git 工具（已经自带了 ssh 客户端）了，另外还有一个图形界面的 Git 项目管理工具。

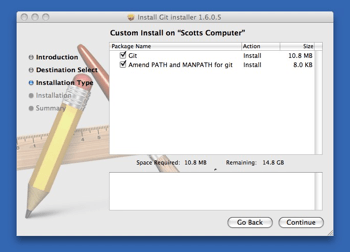
在开始菜单里找到"Git"->"Git Bash"，会弹出 Git 命令窗口，你可以在该窗口进行 Git 操作。

## Mac 平台上安装

在 Mac 平台上安装 Git 最容易的当属使用图形化的 Git 安装工具，下载地址为：

[http://sourceforge.net/projects/git-osx-installer/](https://sourceforge.net/projects/git-osx-installer/)

安装界面如下所示：



## Git 配置

Git 提供了一个叫做 git config 的工具，专门用来配置或读取相应的工作环境变量。

这些环境变量，决定了 Git 在各个环节的具体工作方式和行为。这些变量可以存放在以下三个不同的地方：

* /etc/gitconfig 文件：系统中对所有用户都普遍适用的配置。若使用 git config 时用 --system选项，读写的就是这个文件。
* ~/.gitconfig 文件：用户目录下的配置文件只适用于该用户。若使用 git config 时用 --global选项，读写的就是这个文件。
* 当前项目的 Git 目录中的配置文件（也就是工作目录中的 .git/config 文件）：这里的配置仅仅针对当前项目有效。每一个级别的配置都会覆盖上层的相同配置，所以 .git/config 里的配置会覆盖 /etc/gitconfig 中的同名变量。

在 Windows 系统上，Git 会找寻用户主目录下的 .gitconfig 文件。主目录即 $HOME 变量指定的目录，一般都是 C:\Documents and Settings\$USER。

此外，Git 还会尝试找寻 /etc/gitconfig 文件，只不过看当初 Git 装在什么目录，就以此作为根目录来定位。

### 用户信息

配置个人的用户名称和电子邮件地址：

$ git config --global user.name "w3c"

$ git config --global user.email w3c@w3cschool.cn

如果用了 --global 选项，那么更改的配置文件就是位于你用户主目录下的那个，以后你所有的项目都会默认使用这里配置的用户信息。

如果要在某个特定的项目中使用其他名字或者电邮，只要去掉 --global 选项重新配置即可，新的设定保存在当前项目的 .git/config 文件里。

### 文本编辑器

设置Git默认使用的文本编辑器, 一般可能会是 Vi 或者 Vim。如果你有其他偏好，比如 Emacs 的话，可以重新设置：:

$ git config --global core.editor emacs

### 差异分析工具

还有一个比较常用的是，在解决合并冲突时使用哪种差异分析工具。比如要改用 vimdiff 的话：

$ git config --global merge.tool vimdiff

Git 可以理解 kdiff3，tkdiff，meld，xxdiff，emerge，vimdiff，gvimdiff，ecmerge，和 opendiff 等合并工具的输出信息。

当然，你也可以指定使用自己开发的工具，具体怎么做可以参阅第七章。

### 查看配置信息

要检查已有的配置信息，可以使用 git config --list 命令：

$ git config --list

user.name=Scott Chacon

user.email=schacon@gmail.com

color.status=auto

color.branch=auto

color.interactive=auto

color.diff=auto

...

有时候会看到重复的变量名，那就说明它们来自不同的配置文件（比如 /etc/gitconfig 和 ~/.gitconfig），不过最终 Git 实际采用的是最后一个。

也可以直接查阅某个环境变量的设定，只要把特定的名字跟在后面即可，像这样：

$ git config user.name

Scott Chacon

**Git 工作流程**

由 一水寒 创建，Alma 最后一次修改 2017-08-01 16:52:36

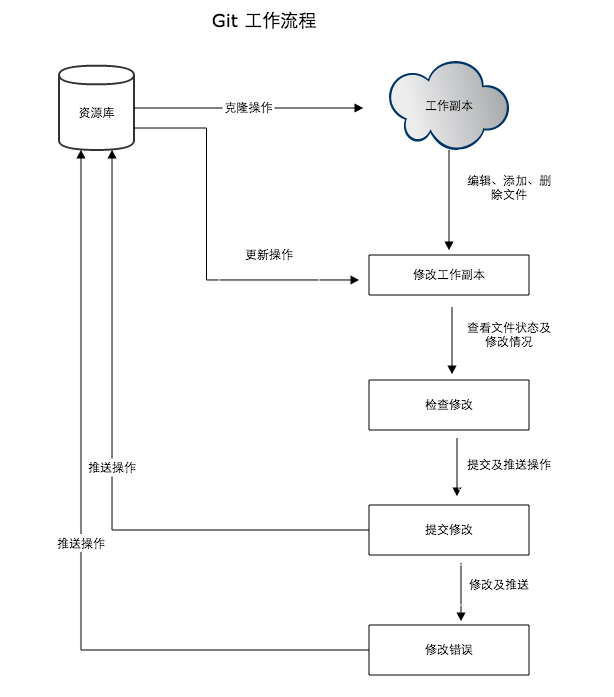
**Git 工作流程**

本章节我们将为大家介绍 Git 的工作流程。

一般工作流程如下：

* 克隆 Git 资源作为工作目录。
* 在克隆的资源上添加或修改文件。
* 如果其他人修改了，你可以更新资源。
* 在提交前查看修改。
* 提交修改。
* 在修改完成后，如果发现错误，可以撤回提交并再次修改并提交。

下图展示了 Git 的工作流程：



**Git 工作区、暂存区和版本库**

由 一水寒 创建，Alma 最后一次修改 2017-08-01 16:52:41

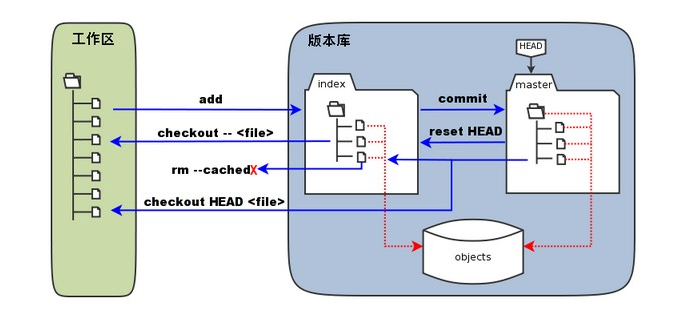
**Git 工作区、暂存区和版本库**

**基本概念**

我们先来理解下Git 工作区、暂存区和版本库概念

* **工作区：**就是你在电脑里能看到的目录。
* **暂存区：**英文叫stage, 或index。一般存放在"git目录"下的index文件（.git/index）中，所以我们把暂存区有时也叫作索引（index）。
* **版本库：**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。

下面这个图展示了工作区、版本库中的暂存区和版本库之间的关系：



图中左侧为工作区，右侧为版本库。在版本库中标记为 "index" 的区域是暂存区（stage, index），标记为 "master" 的是 master 分支所代表的目录树。

图中我们可以看出此时 "HEAD" 实际是指向 master 分支的一个"游标"。所以图示的命令中出现 HEAD 的地方可以用 master 来替换。

图中的 objects 标识的区域为 Git 的对象库，实际位于 ".git/objects" 目录下，里面包含了创建的各种对象及内容。

当对工作区修改（或新增）的文件执行 "git add" 命令时，暂存区的目录树被更新，同时工作区修改（或新增）的文件内容被写入到对象库中的一个新的对象中，而该对象的ID被记录在暂存区的文件索引中。

当执行提交操作（git commit）时，暂存区的目录树写到版本库（对象库）中，master 分支会做相应的更新。即 master 指向的目录树就是提交时暂存区的目录树。

当执行 "git reset HEAD" 命令时，暂存区的目录树会被重写，被 master 分支指向的目录树所替换，但是工作区不受影响。

当执行 "git rm --cached <file>" 命令时，会直接从暂存区删除文件，工作区则不做出改变。

当执行 "git checkout ." 或者 "git checkout -- <file>" 命令时，会用暂存区全部或指定的文件替换工作区的文件。这个操作很危险，会清除工作区中未添加到暂存区的改动。

当执行 "git checkout HEAD ." 或者 "git checkout HEAD <file>" 命令时，会用 HEAD 指向的 master 分支中的全部或者部分文件替换暂存区和以及工作区中的文件。这个命令也是极具危险性的，因为不但会清除工作区中未提交的改动，也会清除暂存区中未提交的改动。

# Git 创建仓库

由 一水寒 创建， 最后一次修改 2015-07-31 19:00:32

## Git 创建仓库

本章节我们将为大家介绍如何创建一个远程的 Git 仓库。您可以使用一个已经存在的目录作为Git仓库或创建一个空目录。

使用您当前目录作为Git仓库，我们只需使它初始化。

git init

使用我们指定目录作为Git仓库。

git init newrepo

初始化后，在当前目录下会出现一个名为 .git 的目录，所有 Git 需要的数据和资源都存放在这个目录中。

如果当前目录下有几个文件想要纳入版本控制，需要先用 git add 命令告诉 Git 开始对这些文件进行跟踪，然后提交：

$ git add \*.c

$ git add README

$ git commit -m 'initial project version'

## 从现有仓库克隆

克隆仓库的命令格式为：

git clone [url]

比如，要克隆 Ruby 语言的 Git 代码仓库 Grit，可以用下面的命令：

$ git clone git://github.com/schacon/grit.git

执行该命令后，会在当前目录下创建一个名为grit的目录，其中包含一个 .git 的目录，用于保存下载下来的所有版本记录。

如果要自己定义要新建的项目目录名称，可以在上面的命令末尾指定新的名字：

$ git clone git://github.com/schacon/grit.git mygrit

# Git 基本操作

由 一水寒 创建， 最后一次修改 2015-07-31 19:00:36

## Git 基本操作

Git 的工作就是创建和保存你的项目的快照及与之后的快照进行对比。本章将对有关创建与提交你的项目的快照的命令作介绍。

## 获取与创建项目命令

### git init

用 git init 在目录中创建新的 Git 仓库。 你可以在任何时候、任何目录中这么做，完全是本地化的。

在目录中执行 git init，就可以创建一个 Git 仓库了。比如我们创建 w3cschoolcc 项目：

$ mkdir w3cschoolcc

$ cd w3cschoolcc

$ git init

Initialized empty Git repository in /www/w3cschoolcc/.git/

# 在 /www/w3cschoolcc/.git/ 目录初始化空 Git 仓库完毕。

现在你可以看到在你的项目目录中有个 .git 的子目录。 这就是你的 Git 仓库了，所有有关你的此项目的快照数据都存放在这里。

ls -a

. .. .git

### git clone

使用 git clone 拷贝一个 Git 仓库到本地，让自己能够查看该项目，或者进行修改。

如果你需要与他人合作一个项目，或者想要复制一个项目，看看代码，你就可以克隆那个项目。 执行命令：

git clone [url]

[url] 为你想要复制的项目，就可以了。

例如我们克隆 Github 上的项目：

$ git clone git://github.com/schacon/simplegit.git

Initialized empty Git repository in /private/tmp/simplegit/.git/

remote: Counting objects: 100, done.

remote: Compressing objects: 100% (86/86), done.

remote: Total 100 (delta 35), reused 0 (delta 0)

Receiving objects: 100% (100/100), 9.51 KiB, done.

Resolving deltas: 100% (35/35), done.

$ cd simplegit/

$ ls

README Rakefile lib

上述操作将复制该项目的全部记录。

$ ls -a

. .. .git README Rakefile lib

$ cd .git

$ ls

HEAD description info packed-refs

branches hooks logs refs

config index objects

默认情况下，Git 会按照你提供的 URL 所指示的项目的名称创建你的本地项目目录。 通常就是该 URL 最后一个 / 之后的项目名称。如果你想要一个不一样的名字， 你可以在该命令后加上你想要的名称。

## 基本快照

Git 的工作就是创建和保存你的项目的快照及与之后的快照进行对比。本章将对有关创建与提交你的项目的快照的命令作介绍。

### git add

git add 命令可将该文件添加到缓存，如我们添加以下两个文件：

$ touch README

$ touch hello.php

$ ls

README hello.php

$ git status -s

?? README

?? hello.php

$

git status 命令用于查看项目的当前状态。

接下来我们执行 git add 命令来添加文件：

$ git add README hello.php

现在我们再执行 git status，就可以看到这两个文件已经加上去了。

$ git status -s

A README

A hello.php

$

新项目中，添加所有文件很普遍，可以在当前工作目录执行命令：git add .。

现在我们改个文件，再执行一下 git status：

$ vim README

$ git status -s

AM README

A hello.php

"AM" 状态的意思是，这个文件在我们将它添加到缓存之后又有改动。改动后我们在执行 git add 命令将其添加到缓存中：

$ git add .

$ git status -s

A README

A hello.php

当你要将你的修改包含在即将提交的快照里的时候，需要执行 git add。

### git status

git status 以查看在你上次提交之后是否有修改。

我演示该命令的时候加了 -s 参数，以获得简短的结果输出。如果没加该参数会详细输出内容：

$ git status

On branch master

Initial commit

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

new file: README

new file: hello.php

### git diff

执行 git diff 来查看执行 git status 的结果的详细信息。

git diff 命令显示已写入缓存与已修改但尚未写入缓存的改动的区别。git diff 有两个主要的应用场景。

* 尚未缓存的改动：git diff
* 查看已缓存的改动： git diff --cached
* 查看已缓存的与未缓存的所有改动：git diff HEAD
* 显示摘要而非整个 diff：git diff --stat

在 hello.php 文件中输入以下内容：

<?php echo 'www.w3cschool.cn'; ?>

$ git status -s

A README

AM hello.php

$ git diff

diff --git a/hello.php b/hello.php

index e69de29..d1a9166 100644

--- a/hello.php

+++ b/hello.php

@@ -0,0 +1,3 @@

+<?php +echo 'www.w3cschool.cn'; +?>

git status显示你上次提交更新至后所更改或者写入缓存的改动， 而 git diff 一行一行地显示这些改动具体是啥。

接下来我们来查看下 git diff --cached 的执行效果：

$ git add hello.php

$ git status -s

A README

A hello.php

$ git diff --cached

diff --git a/README b/README

new file mode 100644

index 0000000..704cce7

--- /dev/null

+++ b/README

@@ -0,0 +1 @@

+w3cschool.cn

diff --git a/hello.php b/hello.php

new file mode 100644

index 0000000..d1a9166

--- /dev/null

+++ b/hello.php

@@ -0,0 +1,3 @@

+<?php +echo 'www.w3cschool.cn'; +?>

### git commit

使用 git add 命令将想要快照的内容写入了缓存， 而执行 git commit 记录缓存区的快照。

Git 为你的每一个提交都记录你的名字与电子邮箱地址，所以第一步需要配置用户名和邮箱地址。

$ git config --global user.name 'w3cschool'

$ git config --global user.email w3c@w3cschool.cn

接下来我们写入缓存，并提交对 hello.php 的所有改动。在首个例子中，我们使用 -m 选项以在命令行中提供提交注释。

$ git add hello.php

$ git status -s

A README

A hello.php

$ git commit -m 'test comment from w3cschool.cn'

[master (root-commit) 85fc7e7] test comment from w3cschool.cn

2 files changed, 4 insertions(+)

create mode 100644 README

create mode 100644 hello.php

现在我们已经记录了快照。如果我们再执行 git status:

$ git status

# On branch master

nothing to commit (working directory clean)

以上输出说明我们在最近一次提交之后，没有做任何改动，是一个"干净的工作目录"。

如果你没有设置 -m 选项，Git 会尝试为你打开一个编辑器以填写提交信息。 如果 Git 在你对它的配置中找不到相关信息，默认会打开 vim。屏幕会像这样：

# Please enter the commit message for your changes. Lines starting # with '#' will be ignored, and an empty message aborts the commit. # On branch master # Changes to be committed: # (use "git reset HEAD ..." to unstage) # # modified: hello.php # ~ ~ ".git/COMMIT\_EDITMSG" 9L, 257C

如果你觉得 git add 提交缓存的流程太过繁琐，Git 也允许你用 -a 选项跳过这一步。命令格式如下：

git commit -a

如：

$ git commit -am 'changes to hello file'

[master 78b2670] changes to hello file

1 files changed, 2 insertions(+), 1 deletions(-)

### git reset HEAD

git reset HEAD 命令用于取消缓存已缓存的内容。

这里我们有两个最近提交之后又有所改动的文件。我们将两个都缓存，并取消缓存其中一个。

$ git status -s

M README

M hello.php

$ git add .

$ git status -s

M README

M hello.pp

$ git reset HEAD -- hello.php

Unstaged changes after reset:

M hello.php

$ git status -s

M README

M hello.php

现在你执行 git commit 将只记录 README 文件的改动，并不含现在并不在缓存中的 hello.rb。

### git rm

git rm 将文件从缓存区中移除。

如我们删除 hello.php文件：

$ git rm hello.php

rm 'hello.php'

$ ls

README

默认情况下，git rm file 会将文件从缓存区和你的硬盘中（工作目录）删除。 如果要在工作目录中留着该文件，可以使用命令：

git rm --cached。

### git mv

git mv 命令做得所有事情就是 git rm --cached， 重命名磁盘上的文件，然后再执行 git add 把新文件添加到缓存区。因此，虽然有 git mv 命令，但它有点多余 。

# Git 分支管理

由 一水寒 创建， 最后一次修改 2015-07-31 19:00:38

## Git 分支管理

几乎每一种版本控制系统都以某种形式支持分支。使用分支意味着你可以从开发主线上分离开来，然后在不影响主线的同时继续工作。

有人把 Git 的分支模型称为"必杀技特性"，而正是因为它，将 Git 从版本控制系统家族里区分出来。

创建分支命令：

git branch (branchname)

切换分支命令:

git checkout (branchname)

当你切换分支的时候，Git 会用该分支的最后提交的快照替换你的工作目录的内容， 所以多个分支不需要多个目录。

合并分支命令:

git merge

你可以多次合并到统一分支， 也可以选择在合并之后直接删除被并入的分支。

## Git 分支管理

### 列出分支

列出分支基本命令：

git branch

没有参数时，git branch 会列出你在本地的分支。

$ git branch

\* master

此例的意思就是，我们有一个叫做"master"的分支，并且该分支是当前分支。

当你执行 git init 的时候，缺省情况下 Git 就会为你创建"master"分支。

如果我们要手动创建一个分支，并切换过去。执行 git branch (branchname) 即可。

$ git branch testing

$ git branch

\* master

testing

现在我们可以看到，有了一个新分支 testing。

当你以此方式在上次提交更新之后创建了新分支，如果后来又有更新提交， 然后又切换到了"testing"分支，Git 将还原你的工作目录到你创建分支时候的样子

接下来我们将演示如何切换分支，我们用 git checkout (branch) 切换到我们要修改的分支。

$ ls

README

$ echo 'w3cschool.cn' > test.txt

$ git add .

$ git commit -m 'add test.txt'

[master 048598f] add test.txt

2 files changed, 1 insertion(+), 3 deletions(-)

delete mode 100644 hello.php

create mode 100644 test.txt

$ ls

README test.txt

$ git checkout testing

Switched to branch 'testing'

$ ls

README hello.php

当我们切换到"testing"分支的时候，我们添加的新文件test.txt被移除了, 原来被删除的文件hello.php文件又出现了。切换回"master"分支的时候，它们有重新出现了。

$ git checkout master

Switched to branch 'master'

$ ls

README test.txt

我们也可以使用 git checkout -b (branchname) 命令来创建新分支并立即切换到该分支下，从而在该分支中操作。

$ git checkout -b newtest

Switched to a new branch 'newtest'

$ git rm test2.txt

rm 'test2.txt'

$ ls

README test.txt

$ git commit -am 'removed test2.txt'

[newtest 556f0a0] removed test2.txt

1 file changed, 1 deletion(-)

delete mode 100644 test2.txt

$ git checkout master

Switched to branch 'master'

$ ls

README test.txt test2.txt

如你所见，我们创建了一个分支，在该分支的上下文中移除了一些文件，然后切换回我们的主分支，那些文件又回来了。

使用分支将工作切分开来，从而让我们能够在不同上下文中做事，并来回切换。

### 删除分支

删除分支命令：

git branch -d (branchname)

例如我们要删除"testing"分支：

$ git branch

\* master

testing

$ git branch -d testing

Deleted branch testing (was 85fc7e7).

$ git branch

\* master

### 分支合并

一旦某分支有了独立内容，你终究会希望将它合并回到你的主分支。 你可以使用以下命令将任何分支合并到当前分支中去：

git merge

$ git branch

\* master

newtest

$ ls

README test.txt test2.txt

$ git merge newtest

Updating 2e082b7..556f0a0

Fast-forward

test2.txt | 1 -

1 file changed, 1 deletion(-)

delete mode 100644 test2.txt

$ ls

README test.txt

以上实例中我们将 newtest 分支合并到主分支去，test2.txt 文件被删除。

### 合并冲突

合并并不仅仅是简单的文件添加、移除的操作，Git 也会合并修改。

$ git branch

\* master

$ cat test.txt

w3cschool.cn

首先，我们创建一个叫做"change\_site"的分支，切换过去，我们将内容改为 www.w3cschool.cn 。

$ git checkout -b change\_site

Switched to a new branch 'change\_site'

$ vim test.txt

$ head -1 test.txt

www.w3cschool.cn

$ git commit -am 'changed the site'

[change\_site d7e7346] changed the site

1 file changed, 1 insertion(+), 1 deletion(-)

将修改的内容提交到 "change\_site" 分支中。 现在，假如切换回 "master" 分支我们可以看内容恢复到我们修改前的，我们再次修改test.txt文件。

$ git checkout master

Switched to branch 'master'

$ head -1 test.txt

w3cschool.cn

$ vim test.txt

$ cat test.txt

w3cschool.cn

新增加一行

$ git diff

diff --git a/test.txt b/test.txt

index 704cce7..f84c2a4 100644

--- a/test.txt

+++ b/test.txt

@@ -1 +1,2 @@

w3cschool.cn

+新增加一行

$ git commit -am '新增加一行'

[master 14b4dca] 新增加一行

1 file changed, 1 insertion(+)

现在这些改变已经记录到我的 "master" 分支了。接下来我们将 "change\_site" 分支合并过来。

$ git merge change\_site

Auto-merging test.txt

CONFLICT (content): Merge conflict in test.txt

Automatic merge failed; fix conflicts and then commit the result.

$ cat test.txt

<<<<<<< HEAD w3cschool.cn 新增加一行 ======= www.w3cschool.cn >>>>>>> change\_site

我们将前一个分支合并到 "master" 分支，一个合并冲突就出现了，接下来我们需要手动去修改它。

$ vim test.txt

$ cat test.txt

www.w3cschool.cn

新增加一行

$ git diff

diff --cc test.txt

index f84c2a4,bccb7c2..0000000

--- a/test.txt

+++ b/test.txt

@@@ -1,2 -1,1 +1,2 @@@

- w3cschool.cn

+ www.w3cschool.cn

+新增加一行

在 Git 中，我们可以用 git add 要告诉 Git 文件冲突已经解决

$ git status -s

UU test.txt

$ git add test.txt

$ git status -s

M test.txt

$ git commit

[master 88afe0e] Merge branch 'change\_site'

现在我们成功解决了合并中的冲突，并提交了结果。

# Git 查看提交历史

由 一水寒 创建， 最后一次修改 2015-07-31 19:00:41

## Git 查看提交历史

在使用 Git 提交了若干更新之后，又或者克隆了某个项目，想回顾下提交历史，我们可以使用 git log 命令查看。

针对我们前一章节的操作，使用 git log 命令列出历史提交记录如下：

$ git log

commit 88afe0e02adcdfea6844bb627de97da21eb10af1

Merge: 14b4dca d7e7346

Author: w3cschool <w3c@w3cschool.cn>

Date: Sun Mar 1 15:03:42 2015 +0800

Merge branch 'change\_site'

Conflicts:

test.txt

commit 14b4dcadbdc847207651d5a9fae0d315057f346e

Author: w3cschool <w3c@w3cschool.cn>

Date: Sun Mar 1 14:53:15 2015 +0800

新增加一行

commit d7e734640da06055e107eaf29cf350b3f1de1c2c

Author: w3cschool <w3c@w3cschool.cn>

Date: Sun Mar 1 14:48:57 2015 +0800

changed the site

commit 556f0a0637978097b82287ac665a717623b21f3f

Author: w3cschool <w3c@w3cschool.cn>

Date: Sun Mar 1 14:40:34 2015 +0800

removed test2.txt

我们可以用 --oneline 选项来查看历史记录的简洁的版本。

$ git log --oneline

88afe0e Merge branch 'change\_site'

14b4dca 新增加一行

d7e7346 changed the site

556f0a0 removed test2.txt

2e082b7 add test2.txt

048598f add test.txt

85fc7e7 test comment from w3cschool.cn

这告诉我们的是，此项目的开发历史。

我们还可以用 --graph 选项，查看历史中什么时候出现了分支、合并。以下为相同的命令，开启了拓扑图选项：

$ git log --oneline --graph

\* 88afe0e Merge branch 'change\_site'

|\

| \* d7e7346 changed the site

\* | 14b4dca 新增加一行

|/

\* 556f0a0 removed test2.txt

\* 2e082b7 add test2.txt

\* 048598f add test.txt

\* 85fc7e7 test comment from w3cschool.cn

现在我们可以更清楚明了地看到何时工作分叉、又何时归并。

你也可以用 '--reverse'参数来逆向显示所有日志。

$ git log --reverse --oneline

85fc7e7 test comment from w3cschool.cn

048598f add test.txt

2e082b7 add test2.txt

556f0a0 removed test2.txt

d7e7346 changed the site

14b4dca 新增加一行

88afe0e Merge branch 'change\_site'

如果只想查找指定用户的提交日志可以使用命令：git log --author , 例如，比方说我们要找 Git 源码中 Linus 提交的部分：

$ git log --author=Linus --oneline -5

81b50f3 Move 'builtin-\*' into a 'builtin/' subdirectory

3bb7256 make "index-pack" a built-in

377d027 make "git pack-redundant" a built-in

b532581 make "git unpack-file" a built-in

112dd51 make "mktag" a built-in

如果你要指定日期，可以执行几个选项：--since 和 --before，但是你也可以用 --until 和 --after。

例如，如果我要看 Git 项目中三周前且在四月十八日之后的所有提交，我可以执行这个（我还用了 --no-merges 选项以隐藏合并提交）：

$ git log --oneline --before={3.weeks.ago} --after={2010-04-18} --no-merges

5469e2d Git 1.7.1-rc2

d43427d Documentation/remote-helpers: Fix typos and improve language

272a36b Fixup: Second argument may be any arbitrary string

b6c8d2d Documentation/remote-helpers: Add invocation section

5ce4f4e Documentation/urls: Rewrite to accomodate transport::address

00b84e9 Documentation/remote-helpers: Rewrite description

03aa87e Documentation: Describe other situations where -z affects git diff

77bc694 rebase-interactive: silence warning when no commits rewritten

636db2c t3301: add tests to use --format="%N"

更多 git log 命令可查看：[http://git-scm.com/docs/git-log](https://git-scm.com/docs/git-log)

# Git 标签

由 一水寒 创建， 最后一次修改 2015-07-31 19:00:43

## Git 标签

如果你达到一个重要的阶段，并希望永远记住那个特别的提交快照，你可以使用 git tag 给它打上标签。

比如说，我们想为我们的 w3cschoolcc 项目发布一个"1.0"版本。 我们可以用 git tag -a v1.0 命令给最新一次提交打上（HEAD）"v1.0"的标签。

-a 选项意为"创建一个带注解的标签"。 不用 -a 选项也可以执行的，但它不会记录这标签是啥时候打的，谁打的，也不会让你添加个标签的注解。 我推荐一直创建带注解的标签。

$ git tag -a v1.0

当你执行 git tag -a 命令时，Git 会打开你的编辑器，让你写一句标签注解，就像你给提交写注解一样。

现在，注意当我们执行 git log --decorate 时，我们可以看到我们的标签了：

$ git log --oneline --decorate --graph

\* 88afe0e (HEAD, tag: v1.0, master) Merge branch 'change\_site'

|\

| \* d7e7346 (change\_site) changed the site

\* | 14b4dca 新增加一行

|/

\* 556f0a0 removed test2.txt

\* 2e082b7 add test2.txt

\* 048598f add test.txt

\* 85fc7e7 test comment from w3cschool.cn

如果我们忘了给某个提交打标签，又将它发布了，我们可以给它追加标签。

例如，假设我们发布了提交 85fc7e7(上面实例最后一行)，但是那时候忘了给它打标签。 我们现在也可以：

$ git tag -a v0.9 85fc7e7

$ git log --oneline --decorate --graph

\* 88afe0e (HEAD, tag: v1.0, master) Merge branch 'change\_site'

|\

| \* d7e7346 (change\_site) changed the site

\* | 14b4dca 新增加一行

|/

\* 556f0a0 removed test2.txt

\* 2e082b7 add test2.txt

\* 048598f add test.txt

\* 85fc7e7 (tag: v0.9) test comment from w3cschool.cn

如果我们要查看所有标签可以使用以下命令：

$ git tag

v0.9

v1.0

指定标签信息命令：

git tag -a <tagname> -m "w3cschool.cn标签"

PGP签名标签命令：

git tag -s <tagname> -m "w3cschool.cn标签"

# Git 远程仓库

由 一水寒 创建，Alma 最后一次修改 2017-08-01 16:53:17

## Git 远程仓库

Git 并不像 SVN 那样有个中心服务器。

目前我们使用到的 Git 命令都是在本地执行，如果你想通过 Git 分享你的代码或者与其他开发人员合作。 你就需要将数据放到一台其他开发人员能够连接的服务器上。

本例使用了 Github 作为远程仓库，你可以先阅读我们的 [Github 简明教程。](https://www.w3cschool.cn/w3cnote/git-guide.html)

## 添加远程库

要添加一个新的远程仓库，可以指定一个简单的名字，以便将来引用,命令格式如下：

git remote add [shortname] [url]

本例以Github为例作为远程仓库，如果你没有Github可以在官网[https://github.com/](https://github.com/" \t "_blank)注册。

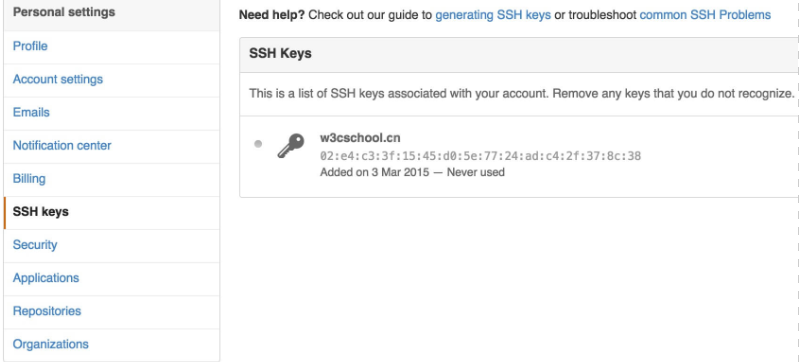
由于你的本地Git仓库和GitHub仓库之间的传输是通过SSH加密的，所以我们需要配置验证信息：

使用以下命令生成SSH Key：

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

后面的your\_email@youremail.com改为你在github上注册的邮箱，之后会要求确认路径和输入密码，我们这使用默认的一路回车就行。成功的话会在~/下生成.ssh文件夹，进去，打开id\_rsa.pub，复制里面的key。

回到github上，进入 Account Settings（账户配置），左边选择SSH Keys，Add SSH Key,title随便填，粘贴在你电脑上生成的key。



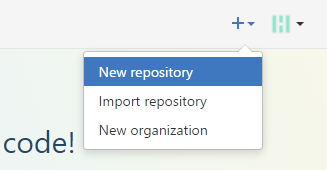
为了验证是否成功，输入以下命令：

$ ssh -T git@github.com

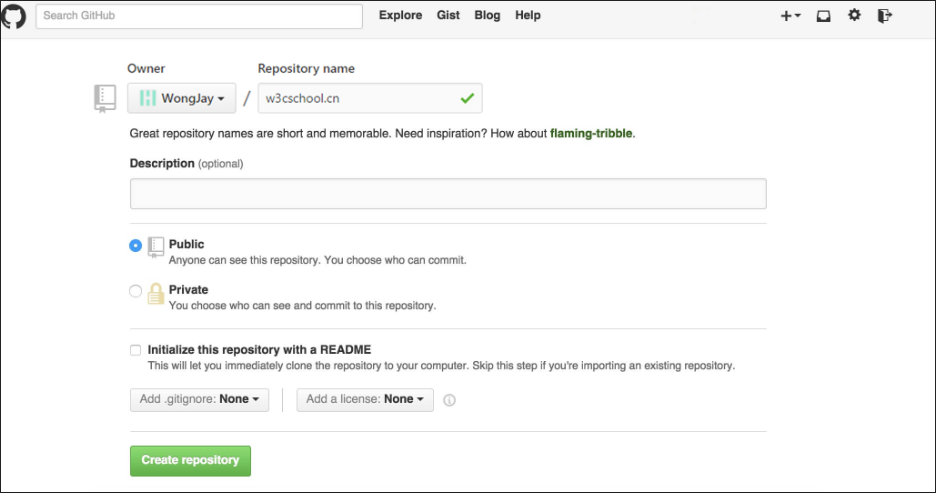
Hi WongJay! You've successfully authenticated, but GitHub does not provide shell access.

以下命令说明我们已成功连上 Github。

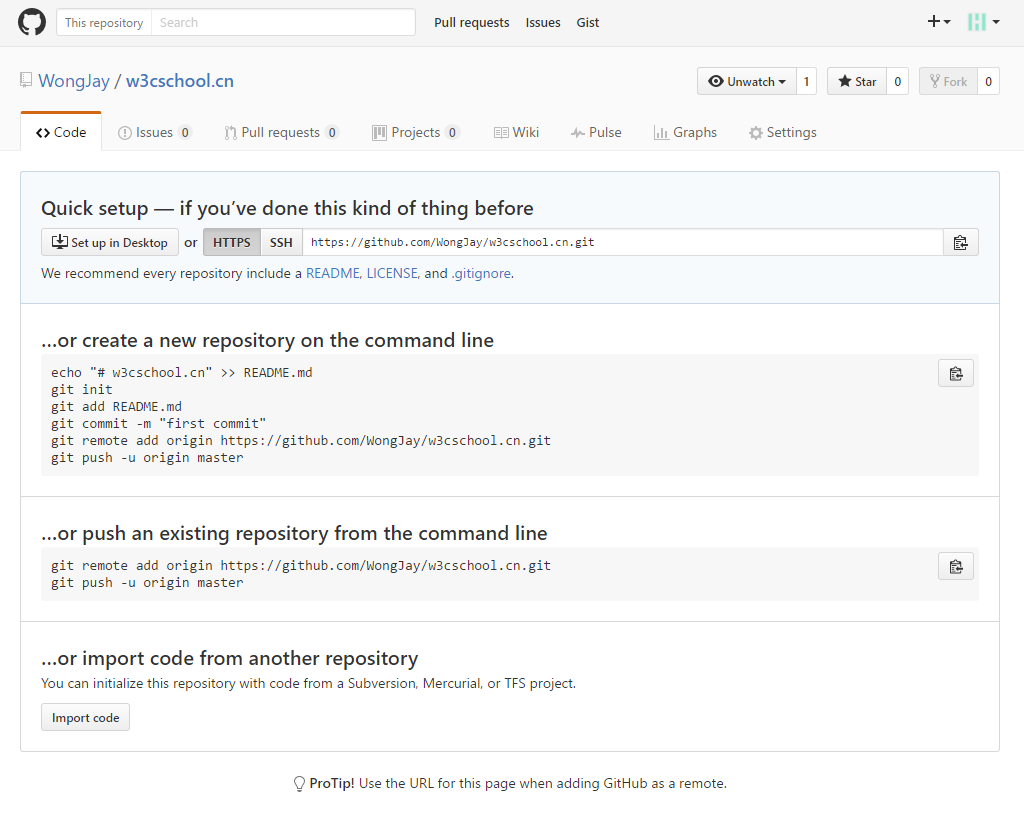
之后登录后点击" New repository " 如下图所示：



之后在在Repository name 填入 w3cschool.cn(远程仓库名) ，其他保持默认设置，点击"Create repository"按钮，就成功地创建了一个新的Git仓库：



创建成功后，显示如下信息：



以上信息告诉我们可以从这个仓库克隆出新的仓库，也可以把本地仓库的内容推送到GitHub仓库。

现在，我们根据GitHub的提示，在本地的仓库下运行命令：

$ ls

README

W3Cschool教程测试.txt

test.txt

$ git remote add origin git@github.com:WongJay/w3cschool.cn.git

$ git push -u origin master

Counting objects: 21, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (15/15), done.

Writing objects: 100% (21/21), 1.73 KiB | 0 bytes/s, done.

Total 21 (delta 4), reused 0 (delta 0)

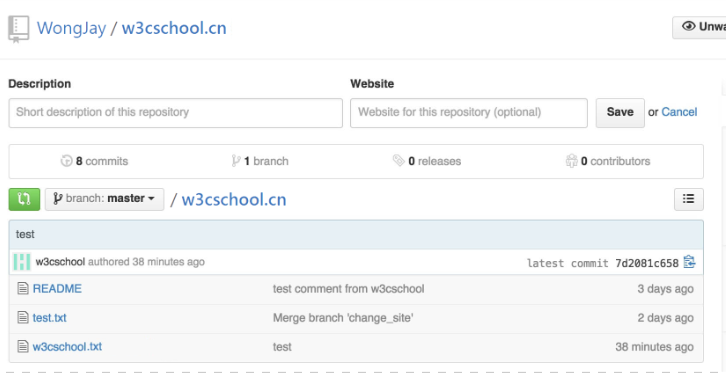
To git@github.com:WongJay/w3cschool.cn.git

\* [new branch] master -> master

Branch master set up to track remote branch master from origin.

以下命令请根据你在Github成功创建新仓库的地方复制，而不是根据我提供的命令，因为我们的Github用户名不一样，仓库名也不一样。

接下来我们返回 Github 创建的仓库，就可以看到文件已上传到Github上：



## 查看当前的远程库

要查看当前配置有哪些远程仓库，可以用命令：

git remote

$ git remote

origin

$ git remote -v

origin git@github.com:WongJay/w3cschool.cn.git (fetch)

origin git@github.com:WongJay/w3cschool.cn.git (push)

执行时加上 -v 参数，你还可以看到每个别名的实际链接地址。

## 提取远程仓库

Git 有两个命令用来提取远程仓库的更新。

1、从远程仓库下载新分支与数据：

git fetch

该命令执行完后需要执行git merge 远程分支到你所在的分支。

2、从远端仓库提取数据并尝试合并到当前分支：

git pull

该命令就是在执行 git fetch 之后紧接着执行 git merge 远程分支到你所在的任意分支。

假设你配置好了一个远程仓库，并且你想要提取更新的数据，你可以首先执行 **git fetch [alias]** 告诉 Git 去获取它有你没有的数据，然后你可以执行 **git merge [alias]/[branch]** 以将服务器上的任何更新（假设有人这时候推送到服务器了）合并到你的当前分支。

接下来我们在 Github 上点击"w3cschoolW3Cschool教程测试.txt" 并在线修改它。之后我们在本地更新修改。

$ git fetch origin

Warning: Permanently added the RSA host key for IP address '192.30.252.128' to the list of known hosts.

remote: Counting objects: 3, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0

Unpacking objects: 100% (3/3), done.

From github.com:WongJay/w3cschool.cn

7d2081c..f5f3dd5 master -> origin/master

以上信息"7d2081c..f5f3dd5 master -> origin/master" 说明 master 分支已被更新，我们可以使用以下命令将更新同步到本地：

$ git merge origin/master

Updating 7d2081c..f5f3dd5

Fast-forward

"w3cschool\350\217\234\351\270\237\346\225\231\347\250\213\346\265\213\350\257\225.txt" | 1 +

1 file changed, 1 insertion(+)

## 推送到远程仓库

推送你的新分支与数据到某个远端仓库命令:

git push [alias] [branch]

以上命令将你的 [branch] 分支推送成为 [alias] 远程仓库上的 [branch] 分支，实例如下。

$ git merge origin/master

Updating 7d2081c..f5f3dd5

Fast-forward

"w3cschool\350\217\234\351\270\237\346\225\231\347\250\213\346\265\213\350\257\225.txt" | 1 +

1 file changed, 1 insertion(+)

bogon:w3cschoolcc WongJay$ vim w3cschoolW3Cschool教程测试.txt

bogon:w3cschoolcc WongJay$ git push origin master

Everything up-to-date

## 删除远程仓库

删除远程仓库你可以使用命令：

git remote rm [别名]

$ git remote -v

origin git@github.com:WongJay/w3cschool.cn.git (fetch)

origin git@github.com:WongJay/w3cschool.cn.git (push)

$ git remote add origin2 git@github.com:WongJay/w3cschool.cn.git

$ git remote -v

origin git@github.com:WongJay/w3cschool.cn.git (fetch)

origin git@github.com:WongJay/w3cschool.cn.git (push)

origin2 git@github.com:WongJay/w3cschool.cn.git (fetch)

origin2 git@github.com:WongJay/w3cschool.cn.git (push)

$ git remote rm origin2

$ git remote -v

origin git@github.com:WongJay/w3cschool.cn.git (fetch)

origin git@github.com:WongJay/w3cschool.cn.git (push)

# Git 服务器搭建

由 一水寒 创建，Alma 最后一次修改 2017-08-01 16:53:26

## Git 服务器搭建

上一章节中我们远程仓库使用了 Github，Github 公开的项目是免费的，但是如果你不想让其他人看到你的项目就需要收费。

这时我们就需要自己搭建一台Git服务器作为私有仓库使用。

接下来我们将以 Centos 为例搭建 Git 服务器。

### 1、安装Git

$ yum install curl-devel expat-devel gettext-devel openssl-devel zlib-devel perl-devel

$ yum install git

接下来我们 创建一个git用户组和用户，用来运行git服务：

$ groupadd git

$ adduser git -g git

### 2、创建证书登录

收集所有需要登录的用户的公钥，公钥位于id\_rsa.pub文件中，把我们的公钥导入到/home/git/.ssh/authorized\_keys文件里，一行一个。

如果没有该文件创建它：

$ cd /home/git/

$ mkdir .ssh

$ chmod 700 .ssh

$ touch .ssh/authorized\_keys

$ chmod 600 .ssh/authorized\_keys

### 3、初始化Git仓库

首先我们选定一个目录作为Git仓库，假定是/home/gitrepo/w3cschoolcn.git，在/home/gitrepo目录下输入命令：

$ cd /home

$ mkdir gitrepo

$ chown git:git gitrepo/

$ cd gitrepo

$ git init --bare w3cschoolcn.git

Initialized empty Git repository in /home/gitrepo/w3cschoolcn.git/

以上命令Git创建一个空仓库，服务器上的Git仓库通常都以.git结尾。然后，把仓库所属用户改为git：

$ chown -R git:git w3cschoolcn.git

### 4、克隆仓库

$ git clone git@192.168.45.4:/home/gitrepo/w3cschoolcn.git

Cloning into 'w3cschoolcn'...

warning: You appear to have cloned an empty repository.

Checking connectivity... done.

192.168.45.4 为 Git 所在服务器 ip ，你需要将其修改为你自己的 Git 服务 ip。

这样我们的 Git 服务器安装就完成了，接下来我们可以禁用 git 用户通过shell登录，可以通过编辑/etc/passwd文件完成。找到类似下面的一行：

git:x:503:503::/home/git:/bin/bash

改为：

git:x:503:503::/home/git:/sbin/nologin

# Git 常用命令速查表

由 一水寒 创建，Alma 最后一次修改 2017-08-01 16:53:33

**Git 常用命令速查表**



# Git 分支策略

由 一水寒 创建，Loen 最后一次修改 2018-02-07 11:33:44

## Git 分支策略

本文将展示我一年前在自己的项目中成功运用的开发模型。我一直打算把这些东西写出来，但总是没有抽出时间，现在终于写好了。这里介绍的不是任何项目的细节，而是有关分支的策略以及对发布的管理。



在我的演示中，所有的操作都是通过 git 完成的。

# 为什么选择 git ？

为了了断 git 和中心源代码控制系统的比较和争论，请移步这里看看 [链接1](http://whygitisbetterthanx.com/) [链接2](https://git.wiki.kernel.org/index.php/GitSvnComparsion)。作为一个开发者，我喜欢 git 超过其它任何现有的工具。Git 真正改变了开发者对于合并和分支的认识。在传统的 CVS/SVN 里，合并/分支总是有点令人害怕的(“注意合并冲突，它们会搞死你的”)。

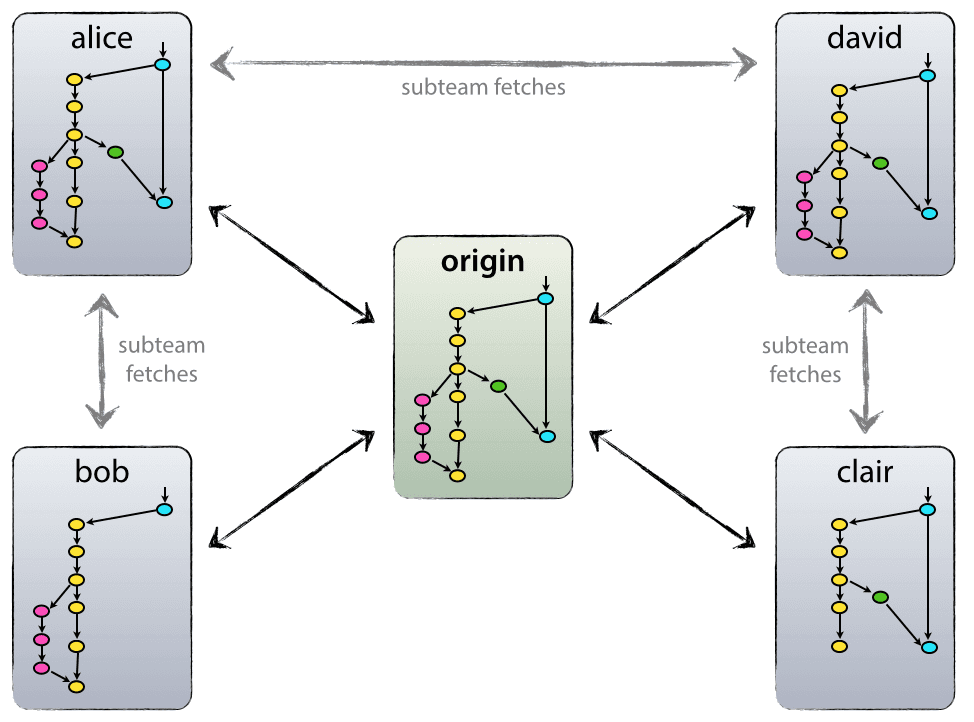
但是 git 中的这些操作是如此的简单有效，它们真正作为你每天工作流程的一部分。比如，在 CVS/SVN 的书籍里，分支和合并总是最后一个章节的讨论重点(对于高级用户)，而在每一本 git 的书里 [链接1](http://book.git-scm.com/) [链接2](http://pragprog.com/titles/tsgit/pragmatic-version-control-using-git) [链接3](http://github.com/progit/progit)，这些内容已经被包含在第三章(基础)里了。

因为它的简单直接和重复性，分支和合并不再令人害怕。版本控制工具比其它任何东西都支持分支/合并。

有关工具就介绍到这里，我们现在进入开发模型这个正题。我要展现的模型本质上无外乎是一个流程的集合，每个团队成员都有必要遵守这些流程，来达到管理软件开发流程的目的。

# 分散但也集中

我们的分支模型中使用良好的代码库的设置方式，是围绕一个真实的中心代码库的。注意，这里的代码库仅仅被看做是一个中心代码库(因为 git 是 DVCS，即分散版本控制系统，从技术层面看，是没有所谓的中心代码库的)。我们习惯于把这个中心代码库命名为 origin，这同时也是所有 git 用户的习惯。



每一位开发者都向 origin 这个中心结点 pull 和 push。但是除此之外，每一位开发者也可以向其它结点 pull 改变形成子团队。比如，对于两个以上开发者同时开发一项大的新特性来说，为了不必过早向 origin 推送开发进度，这就非常有用。在上面的这个例子中，Alice 和 Bob、Alice 和 David、Clair 和 David 都是这样的子团队。

从技术角度，这无非意味着 Alice 定义一个名为 Bob 的 git remote，指向 Bob 的代码库，反之亦然。

# 主分支



该开发模型的核心基本和现有的模型是一样的。中心代码库永远维持着两个主要的分支：

* master
* develop

在 origin 上的 master 分支和每个 git 用户的保持一致。而和 master 分支并行的另一个分支叫做 develop。

我们认为 origin/master 是其 HEAD 源代码总是代表了生产环境准备就绪的状态的主分支。

我们认为 origin/develop 是其 HEAD 源代码总是代表了最后一次交付的可以赶上下一次发布的状态的主分支。有人也把它叫做“集成分支”。该源代码还被作为了 nightly build 自动化任务的来源。

每当 develop 分支到达一个稳定的阶段，可以对外发布时，所有的改变都会被合并到master 分支，并打一个发布版本的 tag。具体操作方法我们稍后讨论。

因此，每次改动被合并到 master 的时候，这就是一个真正的新的发布产品。我们建议对此进行严格的控制，因此理论上我们可以为每次 master 分支的提交都挂一个钩子脚本，向生产环境自动化构建并发布我们的软件。

# 支持型分支

我们的开发模型里，紧接着 master 和 develop 主分支的，是多种多样的支持型分支。它们的目的是帮助团队成员并行处理每次追踪特性、准备发布、快速修复线上问题等开发任务。和之前的主分支不同，这些分支的生命周期都是有限的，它们最终都会被删除掉。

我们可能会用到的不同类型的分支有：

* feature 分支
* release 分支
* hotfix 分支

每一种分支都有一个特别的目的，并且有严格的规则，诸如哪些分支是它们的起始分支、哪些分支必须是它们合并的目标等。我们快速把它们过一遍。

这些“特殊”的分支在技术上是没有任何特殊的。分支的类型取决于我们如何运用它们。它们完完全全都是普通而又平凡的 git 分支。

## feature 分支



* 可能派发自：develop
* 必须合并回：develop
* 分支命名规范：除了 master、develop、release-\* 或 hotfix-\* 的任何名字

Feature 分支(有时也被称作 topic 分支)用来开发包括即将发布或远期发布的新的特性。当我们开始开发一个特性的时候，发布合并的目标可能还不太确定。Feature 分支的生命周期会和新特性的开发周期保持同步，但是最终会合并回 develop (恩，下次发布的时候把这个新特性带上)或被抛弃(真是一次杯具的尝试啊)。

Feature 分支通常仅存在于开发者的代码库中，并不出现在 origin 里。

### 创建一个 feature 分支

当开始一个新特性的时候，从 develop 分支派发出一个分支

$ git checkout -b myfeature develop

Switched to a new branch "myfeature"

### 把完成的特性合并回 develop

完成的特性可以合并回 develop 分支并赶上下一次发布：

$ git checkout develop

Switched to a new branch "develop"

$ git merge --no-ff myfeature

Updating ea1b82a..05e9557

(Summary of changes)

$ git branch -d myfeature

Deleted branch myfeature (was 05e9557)

$ git push origin develop

-no-ff 标记使得合并操作总是产生一次新的提交，哪怕合并操作可以快速完成。这个标记避免将 feature 分支和团队协作的所有提交的历史信息混在主分支的其它提交之后。比较一下：



在右边的例子里，我们不可能从 git 的历史记录中看出来哪些提交实现了这一特性——你可能不得不查看每一笔提交日志。恢复一个完整的特性(比如通过一组提交)在右边变成了一个头疼事情，而如果使用了 --no-ff 之后，就变得简单了。

是的，这会创造一些没有必要的(空的)提交记录，但是得到的是大量的好处。

不幸的是，我还没有找到一个在 git merge 时默认就把 --no-ff 标记打上的办法，但这很重要。

## release 分支

* 可能派发自：develop
* 必须合并回：develop 和 master
* 分支命名规范：release-\*

Release 分支用来支持新的生产环境发布的准备工作。允许在最后阶段产生提交点(dotting i's)和交汇点(crossing t's)。而且允许小幅度的问题修复以及准备发布时的meta数据(比如版本号、发布日期等)。在 release 分支做了上述这些工作之后，develop 分支会被“翻篇儿”，开始接收下一次发布的新特性。

我们选择(几近)完成所有预期的开发的时候，作为从 develop 派发出 release 分支的时机。最起码所有准备构建发布的功能都已经及时合并到了 develop 分支。而往后才会发布的功能则不应该合并到 develop 分支——他们必须等到 release 分支派发出去之后再做合并。

在一个 release 分支的开始，我们就赋予其一个明确的版本号。直到该分支创建之前，develop 分支上的描述都是“下一次”release 的改动，但这个“下一次”release 其实也没说清楚是 0.3 release 还是 1.0 release。而在一个 release 分支的开始时这一点就会确定。这将成为有关项目版本号晋升的一个守则。

### 创建一个 release 分支

Release 分支派发自 develop 分支。比如，我们当前的生产环境发布的版本是 1.1.5，马上有一个 release 要发布了。develop 分支已经为“下一次”release 做好了准备，并且我们已经决定把新的版本号定为 1.2 (而不是 1.1.6 或 2.0)。所以我们派发一个 release 分支并以新的版本号为其命名：

$ git checkout -b release-1.2 develop

Switched to a new branch "release-1.2"

$ ./bump-version.sh 1.2

Files modified successfully, version bumped to 1.2.

$ git commit -a -m "Bumped version number to 1.2"

[release-1.2 74d9424] Bumped version number to 1.2

1 files changed, 1 insertions(+), 1 deletions(-)

创建好并切换到新的分支之后，我们完成对版本号的晋升。这里的 bump-version.sh是一个虚构的用来改变代码库中某些文件以反映新版本的 shell 脚本。(当然你也可以手动完成这些改变——重点是有些文件发生了改变)然后，晋升了的版本号会被提交。

这个新的分支会存在一段时间，直到它确实发布出去了为止。期间可能会有 bug 修复(这比在 develop 做更合理)。但我们严格禁止在此开发庞大的新特性，它们应该合并到 develop 分支，并放入下次发布。

### 完成一个 release 分支

当 release 分支真正发布成功之后，还有些事情需要收尾。首先，release 分支会被合并到 master (别忘了，master 上的每一次提交都代表一个真正的新的发布)；然后，为master 上的这次提交打一个 tag，以便作为版本历史的重要参考；最后，还要把 release 分支产生的改动合并回 develop，以便后续的发布同样包含对这些 bug 的修复。

前两部在 git 下是这样操作的：

$ git checkout master

Switched to branch 'master'

$ git merge --no-ff release-1.2

Merge made by recursive

(Summary of changes)

$ git tag -a 1.2

现在发布工作已经完成了，同时 tag 也打好了，用在未来做参考。

补充：你也可以通过 -s 或 -u <key> 标记打 tag。

为了保留 release 分支里的改动记录，我们需要把这些改动合并回 develop。git 操作如下：

$ git checkout develop

Switched to branch 'develop'

$ git merge --no-ff release-1.2

Merge made by recursive.

(Summary of changes)

这一步有可能导致冲突的发生(只是有理论上的可能性，因为我们已经改变了版本号)，一旦发现，解决冲突然后提交就好了。

现在我们真正完成了一个 release 分支，该把它删掉了，因为它的使命已经完成了：

$ git branch -d release-1.2

Deleted branch release-1.2 (was ff452fe).

## hotfix 分支



* 可能派发自：master
* 必须合并回：develop 和 master
* 分支命名规范：hotfix-\*

Hotfix 分支和 release 分支非常类似，因为他们都意味着会产生一个新的生产环境的发布，尽管 hotfix 分支不是先前就计划好的。他们在实时的生产环境版本出现意外需要快速响应时，从 master 分支相应的 tag 被派发。

我们这样做的根本原因，是为了让团队其中一个人来快速修复生产环境的问题，其他成员可以按工作计划继续工作下去而不受太大影响。

### 创建一个 hotfix 分支

Hotfix 分支创建自 master 分支。例如，假设 1.2 版本是目前的生产环境且出现了一个严重的 bug，但是目前的 develop 并不足够稳定。那么我们可以派发出一个 hotfix 分支来开始我们的修复工作：

$ git checkout -b hotfix-1.2.1 master

Switched to a new branch "hotfix-1.2.1"

$ ./bump-version.sh 1.2.1

Files modified successfully, version bumped to 1.2.1.

$ git commit -a -m "Bumped version number to 1.2.1"

[hotfix-1.2.1 41e61bb] Bumped version number to 1.2.1

1 files changed, 1 insertions(+), 1 deletions(-)

别忘了在派发出分支之后晋升版本号！

然后，修复 bug，提交改动。通过一个或多个提交都可以。

$ git commit -m "Fixed severe production problem"

[hotfix-1.2.1 abbe5d6] Fixed severe production problem

5 files changed, 32 insertions(+), 17 deletions(-)

### 完成一个 hotfix 分支

当我们完成之后，对 bug 的修复需要合并回 master，同时也需要合并回 develop，以保证接下来的发布也都已经解决了这个 bug。这和 release 分支的完成方式是完全一样的。

首先，更新 master 并为本次发布打一个 tag：

$ git checkout master

Switched to branch 'master'

$ git merge --no-ff hotfix-1.2.1

Merge made by recursive

(Summary of changes)

$ git tag -a 1.2.1

补充：你也可以通过 -s 或 -u <key> 标记打 tag。

然后，把已修复的 bug 合并到 develop：

$ git checkout develop

Switched to branch 'develop'

$ git merge --no-ff hotfix-1.2.1

Merge made by recursive

(Summary of changes)

这个规矩的一个额外之处是：如果此时已经存在了一个 release 分支，那么 hotfix 的改变需要合并到这个 release 分支，而不是 develop 分支。因为把对 bug 的修复合并回 release 分支之后，release 分支最终还是会合并回 develop 分支的。(如果在 develop 分支中立刻需要对这个 bug 的修复，且等不及 release 分支合并回来，则你还是可以直接合并回 develop 分支的，这是绝对没问题的)

最后，删掉这个临时的分支：

$ git branch -d hotfix-1.2.1

Deleted branch hotfix-1.2.1 (was abbe5d6).

# 摘要

其实这个分支模型里没有什么新奇的东西。文章开头的那张大图对我们的项目来说非常有用。它非常易于团队成员理解这个优雅有效的模型，并在团队内部达成共识。

这里还有一份那张大图的 [高清PDF版本](http://nvie.com/files/Git-branching-model.pdf)，你可以把它当做手册放在手边快速浏览。

补充：还有，如果你们需要的话，这里还有一份 [Keynote 版本](http://github.com/downloads/nvie/gitflow/Git-branching-model-src.key.zip)

本文译自：[A successful Git branching model » nvie.com](http://nvie.com/posts/a-successful-git-branching-model/)

# Git clone命令用法

由 一水寒 创建，Alma 最后一次修改 2018-05-14 11:02:16

git clone是git中常用的命令，其作用是将存储库克隆到新目录中。那么在git中，git clone具体该如何用呢?

**更快更轻松的学习Git,请点击**[**Git微课**](https://www.w3cschool.cn/minicourse/play/gitcourse)

git clone命令的作用是将存储库克隆到新目录中，为克隆的存储库中的每个分支创建远程跟踪分支(使用git branch -r可见)，并从克隆检出的存储库作为当前活动分支的初始分支。

在克隆之后，没有参数的普通git提取将更新所有远程跟踪分支，并且没有参数的git pull将另外将远程主分支合并到当前主分支(如果有的话)。

此默认配置通过在refs/remotes/origin下创建对远程分支头的引用，并通过初始化remote.origin.url和remote.origin.fetch配置变量来实现。

执行远程操作的第一步，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 http://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。

**示例**

以下是所支持协议的一些示例：

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

$ git clone http://git.oschina.net/yiibai/sample.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。

**应用场景示例**

从上游克隆下来：

$ git clone git://git.kernel.org/pub/scm/.../linux.git mydir

$ cd mydir

$ make # 执行代码或其它命令

在当前目录中使用克隆，而无需检出：

$ git clone -l -s -n . ../copy

$ cd ../copy

$ git show-branch

从现有本地目录借用从上游克隆：

$ git clone --reference /git/linux.git

git://git.kernel.org/pub/scm/.../linux.git

mydir

$ cd mydir

创建一个裸存储库以将您的更改发布给公众：

$ git clone --bare -l /home/proj/.git /pub/scm/proj.git

以上就是Git clone命令的一些常见用法，希望对大家有所帮助。想要了解更多Git命令的用法，可点击：[Git常用命令速查表](https://www.w3cschool.cn/git/git-cheat-sheet.html)

# Git push命令用法

由 一水寒 创建，Alma 最后一次修改 2018-05-14 11:14:11

git push是Git中常用的命令，其作用是将本地分支的更新推送到远程主机。

**git push格式：**

git push的格式和git pull类似：

$ 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常见用法：**

$ git push origin master

该命令的作用是将本地的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，结果导致在远程主机产生一个”非直进式”的合并(non-fast-forward merge)。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

有时候当远程xxx分支被删掉了后，用git branch -a 你还可以看到本地还有remote/origin/xxx这个分支，那么你可以使用git fetch -p 这个命令可以帮你同步最新的远程分支，并删掉本地被删了的远程分支。

以上就是Git push命令的一些常见用法，希望对大家有所帮助。想要了解更多Git命令，可点击：[Git常用命令速查表](https://www.w3cschool.cn/git/git-cheat-sheet.html)

# Git merge命令用法

由 一水寒 创建，Alma 最后一次修改 2018-05-14 13:53:31

git merge是在Git中使用比较频繁的一个命令，其主要用于将两个或两个以上的开发历史加入(合并)一起。本文就为大家带来git merge命令的常见用法。

**git merge三种语法：**

git merge [-n] [--stat] [--no-commit] [--squash] [--[no-]edit]

[-s <strategy>] [-X <strategy-option>] [-S[<keyid>]]

[--[no-]allow-unrelated-histories]

[--[no-]rerere-autoupdate] [-m <msg>] [<commit>…?]

git merge --abort

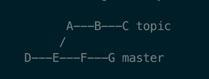
git merge --continue

**git merge用途**

1、用于git-pull中，来整合另一代码仓库中的变化（即：git pull = git fetch + git merge）

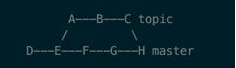
2、用于从一个分支到另一个分支的合并

假设下图中的历史节点存在，并且当前所在的分支为“master”：



那么git merge topic命令将会把在master分支上二者共同的节点（E节点）之后分离的节点（即topic分支的A B C节点）重现在master分支上，直到topic分支当前的commit节点（C节点），并位于master分支的顶部。并且沿着master分支和topic分支创建一个记录合并结果的新节点，该节点带有用户描述合并变化的信息。

即下图中的H节点，C节点和G节点都是H节点的父节点。



**git merge <msg> HEAD <commit>...命令**

该命令的存在是由于历史原因，在新版本中不应该使用它，应该使用git merge -m <msg> <commit>....进行替代

**git merge --abort命令**

该命令仅仅在合并后导致冲突时才使用。git merge --abort将会抛弃合并过程并且尝试重建合并前的状态。但是，当合并开始时如果存在未commit的文件，git merge --abort在某些情况下将无法重现合并前的状态。（特别是这些未commit的文件在合并的过程中将会被修改时）

警告：

运行git-merge时含有大量的未commit文件很容易让你陷入困境，这将使你在冲突中难以回退。因此非常不鼓励在使用git-merge时存在未commit的文件，建议使用git-stash命令将这些未commit文件暂存起来，并在解决冲突以后使用git stash pop把这些未commit文件还原出来。

以上就是Git merge命令的一些常见用法，希望对大家有所帮助。想要了解更多Git命令，可点击：[Git常用命令速查表](https://www.w3cschool.cn/git/git-cheat-sheet.html)