# Git概述

## 1.什么是GIT

Git是一个开源的分布式版本控制系统。版本控制是一种记录一个或若干文件内容变化，以便将来查阅特定版本修订情况的系统。

## 2.Git中的相关概念

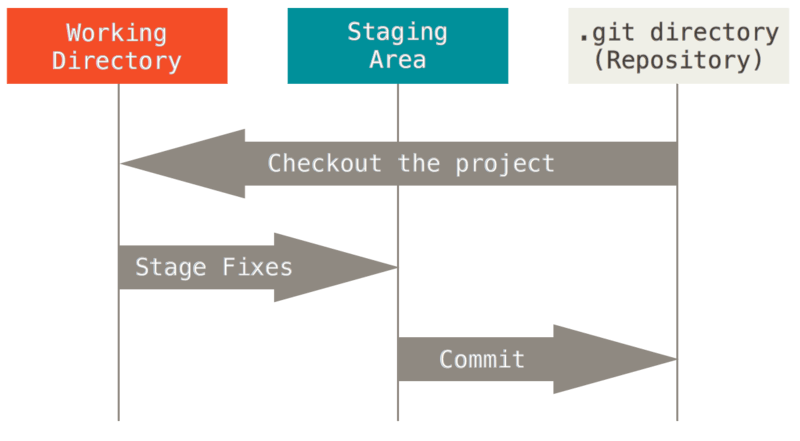
Git 有三种状态，文件可能处于其中之一： **已提交（committed）**、**已修改（modified）** 和 **已暂存（staged）**。

**已修改**表示修改了文件，但还没保存到数据库中。

**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。

**已提交**表示数据已经安全地保存在本地数据库中。

这会让我们的 Git 项目拥有三个阶段：工作区、暂存区以及 Git 目录。



**工作区**是对项目的某个版本独立提取出来的内容;

**暂存区**是一个文件，保存了下次将要提交的文件列表信息，一般在 Git 仓库目录中。

**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，复制的就是这里的数据。

## 3.基本的 Git 工作流程：

1.在工作区中修改文件。

2.将你想要下次提交的更改选择性地暂存，这样只会将更改的部分添加到暂存区。

3.提交更新，找到暂存区的文件，将快照永久性存储到 Git 目录。

如果 Git 目录中保存着特定版本的文件，就属于 **已提交** 状态。 如果文件已修改并放入暂存区，就属于 **已暂存** 状态。 如果自上次检出后，作了修改但还没有放到暂存区域，就是 **已修改** 状态。

## 4.Git 使用方式

Git有多种使用方式,可以使用原生的命令行模式，也可以使用 GUI 模式.

## 5.GIT的安装

<https://git-scm.com/>

## 6.初次运行 Git 前的配置

### 设置你的用户名和邮件地址

这一点很重要，因为每一个 Git 提交都会使用这些信息，它们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

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

## 7.Git获取帮助

$ git help <verb>

$ git <verb> --help

$ man git-<verb>

实例：$ git help config

说明：获取config命令的手册

# GIT基础

## 1．获取 Git 仓库

### **获取 Git 项目仓库的方式**：

1.将尚未进行版本控制的本地目录转换为 Git 仓库；

2.从其它服务器 **克隆** 一个已存在的 Git 仓库。

两种方式都会在你的本地机器上得到一个工作就绪的 Git 仓库。

### 在已存在目录中初始化仓库

如果你有一个尚未进行版本控制的项目目录，想要用 Git 来控制它，那么首先需要进入该项目目录中。 如果你还没这样做过，那么不同系统上的做法有些不同：

在 Linux 上：

$ cd /home/user/my\_project

在 macOS 上：

$ cd /Users/user/my\_project

在 Windows 上：

$ cd /c/user/my\_project

之后执行：

$ **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。

如果在一个已存在文件的文件夹（而非空文件夹）中进行版本控制，你应该开始追踪这些文件并进行初始提交。 可以通过 git add 命令来指定所需的文件来进行追踪，然后执行 git commit ：

$ git add \*.c

$ git add LICENSE

$ git commit -m 'initial project version'

### 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，这时就要用到 git clone 命令。

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

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。

如果你想在克隆远程仓库的时候，自定义本地仓库的名字，你可以通过额外的参数指定新的目录名：

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

这会执行与上一条命令相同的操作，但目标目录名变为了 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 。

## 2.记录每次更新到仓库

### 记录每次更新到仓库

现在我们的机器上有了一个 **真实项目** 的 Git 仓库，并从这个仓库中检出了所有文件的 **工作副本**。 通常，你会对这些文件做些修改，每当完成了一个阶段的目标，想要将记录下它时，就将它提交到仓库。

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， 它们的状态可能是未修改，已修改或已放入暂存区。简而言之，已跟踪的文件就是 Git 已经知道的文件。

工作目录中除已跟踪文件外的其它所有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，也没有被放入暂存区。 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，因为 Git 刚刚检出了它们， 而你尚未编辑过它们。

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。 在工作时，你可以选择性地将这些修改过的文件放入暂存区，然后提交所有已暂存的修改，如此反复。



### 检查当前文件状态

可以用 git status 命令查看哪些文件处于什么状态。 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

On branch master

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

nothing to commit, working directory clean

这说明你现在的工作目录相当干净。换句话说，所有已跟踪文件在上次提交后都未被更改过。 此外，上面的信息还表明，当前目录下没有出现任何处于未跟踪状态的新文件，否则 Git 会在这里列出来。 最后，该命令还显示了当前所在分支，并告诉你这个分支同远程服务器上对应的分支没有偏离。 现在，分支名是“master”，这是默认的分支名。

现在，让我们在项目下创建一个新的 README 文件。 如果之前并不存在这个文件，使用 git status 命令，你将看到一个新的未跟踪文件：

$ echo 'My Project' > README

$ git status

On branch master

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

Untracked files:

(use "git add <file>..." to include in what will be committed)

README

nothing added to commit but untracked files present (use "git add" to track)

在状态报告中可以看到新建的 README 文件出现在 Untracked files 下面。 未跟踪的文件意味着 Git 在之前的快照（提交）中没有这些文件；Git 不会自动将之纳入跟踪范围，除非你明明白白地告诉它“我需要跟踪该文件”。 这样的处理让你不必担心将生成的二进制文件或其它不想被跟踪的文件包含进来。 不过现在的例子中，我们确实想要跟踪管理 README 这个文件。

### 跟踪新文件

**使用命令 git add 开始跟踪一个文件。** 所以，要跟踪 README 文件，运行：

$ git add README

此时再运行 git status 命令，会看到 README 文件已被跟踪，并处于暂存状态：

$ git status

On branch master

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

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

new file: README

只要在 Changes to be committed 这行下面的，就说明是已暂存状态。 如果此时提交，那么该文件在你运行 git add 时的版本将被留存在后续的历史记录中。 你可能会想起之前我们使用 git init 后就运行了 git add <files> 命令，开始跟踪当前目录下的文件。 git add 命令使用文件或目录的路径作为参数；如果参数是目录的路径，该命令将递归地跟踪该目录下的所有文件。

### 暂存已修改的文件

现在我们来修改一个已被跟踪的文件。 如果你修改了一个名为 CONTRIBUTING.md 的已被跟踪的文件，然后运行 git status 命令，会看到下面内容：

$ git status

On branch master

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

Changes to be committed:

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

new file: README

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: CONTRIBUTING.md

文件 CONTRIBUTING.md 出现在 Changes not staged for commit 这行下面，说明已跟踪文件的内容发生了变化，但还没有放到暂存区。 要暂存这次更新，需要运行 git add 命令。 这是个多功能命令：可以用它开始跟踪新文件，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“精确地将内容添加到下一次提交中”而不是“将一个文件添加到项目中”要更加合适。 现在让我们运行 git add 将“CONTRIBUTING.md”放到暂存区，然后再看看 git status 的输出：

$ git add CONTRIBUTING.md

$ git status

On branch master

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

Changes to be committed:

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

new file: README

modified: CONTRIBUTING.md

现在两个文件都已暂存，下次提交时就会一并记录到仓库。 假设此时，你想要在 CONTRIBUTING.md 里再加条注释。 重新编辑存盘后，准备好提交。 不过且慢，再运行 git status 看看：

$ vim CONTRIBUTING.md

$ git status

On branch master

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

Changes to be committed:

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

new file: README

modified: CONTRIBUTING.md

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: CONTRIBUTING.md

怎么回事？ 现在 CONTRIBUTING.md 文件同时出现在暂存区和非暂存区。 这怎么可能呢？ 好吧，实际上 Git 只不过暂存了你运行 git add 命令时的版本。 如果你现在提交，CONTRIBUTING.md 的版本是你最后一次运行 git add 命令时的那个版本，而不是你运行 git commit 时，在工作目录中的当前版本。 所以，运行了 git add 之后又作了修订的文件，需要重新运行 git add 把最新版本重新暂存起来：

$ git add CONTRIBUTING.md

$ git status

On branch master

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

Changes to be committed:

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

new file: README

modified: CONTRIBUTING.md

### 状态简览

git status 命令的输出十分详细，但其用语有些繁琐。 Git 有一个选项可以帮你缩短状态命令的输出，这样可以以简洁的方式查看更改。 如果你使用 git status -s 命令或 git status --short 命令，你将得到一种格式更为紧凑的输出。

$ git status -s

M README

MM Rakefile

A lib/git.rb

M lib/simplegit.rb

?? LICENSE.txt

新添加的未跟踪文件前面有 ?? 标记，新添加到暂存区中的文件前面有 A 标记，修改过的文件前面有 M 标记。 输出中有两栏，左栏指明了暂存区的状态，右栏指明了工作区的状态。例如，上面的状态报告显示： README 文件在工作区已修改但尚未暂存，而 lib/simplegit.rb 文件已修改且已暂存。 Rakefile 文件已修改，暂存后又作了修改，因此该文件的修改中既有已暂存的部分，又有未暂存的部分。

### 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，列出要忽略的文件的模式。 来看一个实际的 .gitignore 例子：

$ cat .gitignore

\*.[oa]

\*~

第一行告诉 Git 忽略所有以 .o 或 .a 结尾的文件。一般这类对象文件和存档文件都是编译过程中出现的。 第二行告诉 Git 忽略所有名字以波浪符（~）结尾的文件，许多文本编辑软件（比如 Emacs）都用这样的文件名保存副本。 此外，你可能还需要忽略 log，tmp 或者 pid 目录，以及自动生成的文档等等。 要养成一开始就为你的新仓库设置好 .gitignore 文件的习惯，以免将来误提交这类无用的文件。

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，它会递归地应用在整个工作区中。
* 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。 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。

我们再看一个 .gitignore 文件的例子：

# 忽略所有的 .a 文件

\*.a

# 但跟踪所有的 lib.a，即便你在前面忽略了 .a 文件

!lib.a

# 只忽略当前目录下的 TODO 文件，而不忽略 subdir/TODO

/TODO

# 忽略任何目录下名为 build 的文件夹

build/

# 忽略 doc/notes.txt，但不忽略 doc/server/arch.txt

doc/\*.txt

# 忽略 doc/ 目录及其所有子目录下的 .pdf 文件

doc/\*\*/\*.pdf

|  |  |
| --- | --- |
| **Tip** | GitHub 有一个十分详细的针对数十种项目及语言的 .gitignore 文件列表， 你可以在 <https://github.com/github/gitignore> 找到它。 |
| **Note** | 在最简单的情况下，一个仓库可能只根目录下有一个 .gitignore 文件，它递归地应用到整个仓库中。 然而，子目录下也可以有额外的 .gitignore 文件。子目录中的 .gitignore 文件中的规则只作用于它所在的目录中。 （Linux 内核的源码库拥有 206 个 .gitignore 文件。）  多个 .gitignore 文件的具体细节超出了本书的范围，更多详情见 man gitignore 。 |

### 查看已暂存和未暂存的修改

如果 git status 命令的输出对于你来说过于简略，而你想知道具体修改了什么地方，可以用 git diff 命令。 稍后我们会详细介绍 git diff，你通常可能会用它来回答这两个问题：当前做的哪些更新尚未暂存？ 有哪些更新已暂存并准备好下次提交？ 虽然 git status 已经通过在相应栏下列出文件名的方式回答了这个问题，但 git diff 能通过文件补丁的格式更加具体地显示哪些行发生了改变。

假如再次修改 README 文件后暂存，然后编辑 CONTRIBUTING.md 文件后先不暂存， 运行 status 命令将会看到：

$ git status

On branch master

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

Changes to be committed:

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

modified: README

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: CONTRIBUTING.md

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

diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md

index 8ebb991..643e24f 100644

--- a/CONTRIBUTING.md

+++ b/CONTRIBUTING.md

@@ -65,7 +65,8 @@ branch directly, things can get messy.

Please include a nice description of your changes when you submit your PR;

if we have to read the whole diff to figure out why you're contributing

in the first place, you're less likely to get feedback and have your change

-merged in.

+merged in. Also, split your changes into comprehensive chunks if your patch is

+longer than a dozen lines.

If you are starting to work on a particular area, feel free to submit a PR

that highlights your work in progress (and note in the PR title that it's

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 --staged 命令。 这条命令将比对已暂存文件与最后一次提交的文件差异：

$ git diff --staged

diff --git a/README b/README

new file mode 100644

index 0000000..03902a1

--- /dev/null

+++ b/README

@@ -0,0 +1 @@

+My Project

请注意，git diff 本身只显示尚未暂存的改动，而不是自上次提交以来所做的所有改动。 所以有时候你一下子暂存了所有更新过的文件，运行 git diff 后却什么也没有，就是这个原因。

像之前说的，暂存 CONTRIBUTING.md 后再编辑，可以使用 git status 查看已被暂存的修改或未被暂存的修改。 如果我们的环境（终端输出）看起来如下：

$ git add CONTRIBUTING.md

$ echo '# test line' >> CONTRIBUTING.md

$ git status

On branch master

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

Changes to be committed:

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

modified: CONTRIBUTING.md

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: CONTRIBUTING.md

现在运行 git diff 看暂存前后的变化：

$ git diff

diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md

index 643e24f..87f08c8 100644

--- a/CONTRIBUTING.md

+++ b/CONTRIBUTING.md

@@ -119,3 +119,4 @@ at the

## Starter Projects

See our [projects list](https://github.com/libgit2/libgit2/blob/development/PROJECTS.md).

+# test line

然后用 git diff --cached 查看已经暂存起来的变化（ --staged 和 --cached 是同义词）：

$ git diff --cached

diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md

index 8ebb991..643e24f 100644

--- a/CONTRIBUTING.md

+++ b/CONTRIBUTING.md

@@ -65,7 +65,8 @@ branch directly, things can get messy.

Please include a nice description of your changes when you submit your PR;

if we have to read the whole diff to figure out why you're contributing

in the first place, you're less likely to get feedback and have your change

-merged in.

+merged in. Also, split your changes into comprehensive chunks if your patch is

+longer than a dozen lines.

If you are starting to work on a particular area, feel free to submit a PR

that highlights your work in progress (and note in the PR title that it's

|  |  |
| --- | --- |
| **Note** | **Git Diff 的插件版本**  我们使用 git diff 来分析文件差异。 但是你也可以使用图形化的工具或外部 diff 工具来比较差异。 可以使用 git difftool 命令来调用 emerge 或 vimdiff 等软件（包括商业软件）输出 diff 的分析结果。 使用 git difftool --tool-help 命令来看你的系统支持哪些 Git Diff 插件。 |

### 提交更新

现在的暂存区已经准备就绪，可以提交了。 在此之前，请务必确认还有什么已修改或新建的文件还没有 git add 过， 否则提交的时候不会记录这些尚未暂存的变化。 这些已修改但未暂存的文件只会保留在本地磁盘。 所以，每次准备提交前，先用 git status 看下，你所需要的文件是不是都已暂存起来了， 然后再运行**提交命令 git commit**：

$ git commit

这样会启动你选择的文本编辑器来输入提交说明。

|  |  |
| --- | --- |
| **Note** | 启动的编辑器是通过 Shell 的环境变量 EDITOR 指定的，一般为 vim 或 emacs。 当然也可以按照 [起步](https://git-scm.com/book/zh/v2/ch00/ch01-getting-started) 介绍的方式， 使用 git config --global core.editor 命令设置你喜欢的编辑器。 |

编辑器会显示类似下面的文本信息（本例选用 Vim 的屏显方式展示）：

# Please enter the commit message for your changes. Lines starting

# with '#' will be ignored, and an empty message aborts the commit.

# On branch master

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

#

# Changes to be committed:

# new file: README

# modified: CONTRIBUTING.md

#

~

~

~

".git/COMMIT\_EDITMSG" 9L, 283C

可以看到，默认的提交消息包含最后一次运行 git status 的输出，放在注释行里，另外开头还有一个空行，供你输入提交说明。 你完全可以去掉这些注释行，不过留着也没关系，多少能帮你回想起这次更新的内容有哪些。

|  |  |
| --- | --- |
| **Note** | 更详细的内容修改提示可以用 -v 选项查看，这会将你所作的更改的 diff 输出呈现在编辑器中，以便让你知道本次提交具体作出哪些修改。 |

退出编辑器时，Git 会丢弃注释行，用你输入的提交说明生成一次提交。

另外，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"

[master 463dc4f] Story 182: Fix benchmarks for speed

2 files changed, 2 insertions(+)

create mode 100644 README

好，现在你已经创建了第一个提交！ 可以看到，提交后它会告诉你，当前是在哪个分支（master）提交的，本次提交的完整 SHA-1 校验和是什么（463dc4f），以及在本次提交中，有多少文件修订过，多少行添加和删改过。

请记住，提交时记录的是放在暂存区域的快照。 任何还未暂存文件的仍然保持已修改状态，可以在下次提交时纳入版本管理。 每一次运行提交操作，都是对你项目作一次快照，以后可以回到这个状态，或者进行比较。

### 跳过使用暂存区域

尽管使用暂存区域的方式可以精心准备要提交的细节，但有时候这么做略显繁琐。 Git 提供了一个跳过使用暂存区域的方式， 只要在提交的时候，**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 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: CONTRIBUTING.md

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

$ git commit -a -m 'added new benchmarks'

[master 83e38c7] added new benchmarks

1 file changed, 5 insertions(+), 0 deletions(-)

看到了吗？提交之前不再需要 git add 文件“CONTRIBUTING.md”了。 这是因为 -a 选项使本次提交包含了所有修改过的文件。 这很方便，但是要小心，有时这个选项会将不需要的文件添加到提交中。

### 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。 **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。

如果只是简单地从工作目录中手工删除文件，运行 git status 时就会在 “Changes not staged for commit” 部分（也就是 **未暂存清单**）看到：

$ rm PROJECTS.md

$ git status

On branch master

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

Changes not staged for commit:

(use "git add/rm <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

deleted: PROJECTS.md

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

然后再运行 git rm 记录此次移除文件的操作：

$ git rm PROJECTS.md

rm 'PROJECTS.md'

$ git status

On branch master

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

Changes to be committed:

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

deleted: PROJECTS.md

下一次提交时，该文件就不再纳入版本管理了。 如果要删除之前修改过或已经放到暂存区的文件，则必须使用强制删除选项 -f（译注：即 force 的首字母）。 这是一种安全特性，用于防止误删尚未添加到快照的数据，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。比如：

$ 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 \\*~

该命令会删除所有名字以 ~ 结尾的文件。

### 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

Your branch is up-to-date with 'origin/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重命名，所以不管何种方式结果都一样。 两者唯一的区别在于，git mv 是一条命令而非三条命令，直接使用 git mv 方便得多。 不过在使用其他工具重命名文件时，记得在提交前 git rm 删除旧文件名，再 git add 添加新文件名。

## 3.查看提交历史

### 查看提交历史

在提交了若干更新，又或者克隆了某个项目之后，你也许想回顾下提交历史。 完成这个任务最简单而又有效的工具是**git log**命令。

我们使用一个非常简单的 “simplegit” 项目作为示例。 运行下面的命令获取该项目：

$ git clone https://github.com/schacon/simplegit-progit

当你在此项目中运行 git log 命令时，可以看到下面的输出：

$ git log

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 16:40:33 2008 -0700

removed unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 10:31:28 2008 -0700

first commit

不传入任何参数的默认情况下，git log 会按时间先后顺序列出所有的提交，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 有许多选项可以帮助你搜寻你所要找的提交， 下面我们会介绍几个最常用的选项。

其中一个比较有用的选项是 -p 或 --patch ，它会显示每次提交所引入的差异（按 **补丁** 的格式输出）。 你也可以限制显示的日志条目数量，例如使用 -2 选项来只显示最近的两次提交：

$ git log -p -2

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

diff --git a/Rakefile b/Rakefile

index a874b73..8f94139 100644

--- a/Rakefile

+++ b/Rakefile

@@ -5,7 +5,7 @@ require 'rake/gempackagetask'

spec = Gem::Specification.new do |s|

s.platform = Gem::Platform::RUBY

s.name = "simplegit"

- s.version = "0.1.0"

+ s.version = "0.1.1"

s.author = "Scott Chacon"

s.email = "schacon@gee-mail.com"

s.summary = "A simple gem for using Git in Ruby code."

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 16:40:33 2008 -0700

removed unnecessary test

diff --git a/lib/simplegit.rb b/lib/simplegit.rb

index a0a60ae..47c6340 100644

--- a/lib/simplegit.rb

+++ b/lib/simplegit.rb

@@ -18,8 +18,3 @@ class SimpleGit

end

end

-

-if $0 == \_\_FILE\_\_

- git = SimpleGit.new

- puts git.show

-end

该选项除了显示基本信息之外，还附带了每次提交的变化。 当进行代码审查，或者快速浏览某个搭档的提交所带来的变化的时候，这个参数就非常有用了。 你也可以为 git log 附带一系列的总结性选项。 比如你想看到每次提交的简略统计信息，可以使用 --stat 选项：

$ git log --stat

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

Rakefile | 2 +-

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

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 16:40:33 2008 -0700

removed unnecessary test

lib/simplegit.rb | 5 -----

1 file changed, 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 10:31:28 2008 -0700

first commit

README | 6 ++++++

Rakefile | 23 +++++++++++++++++++++++

lib/simplegit.rb | 25 +++++++++++++++++++++++++

3 files changed, 54 insertions(+)

正如你所看到的，--stat 选项在每次提交的下面列出所有被修改过的文件、有多少文件被修改了以及被修改过的文件的哪些行被移除或是添加了。 在每次提交的最后还有一个总结。

另一个非常有用的选项是 --pretty。 这个选项可以使用不同于默认格式的方式展示提交历史。 这个选项有一些内建的子选项供你使用。 比如 oneline 会将每个提交放在一行显示，在浏览大量的提交时非常有用。 另外还有 short，full 和 fuller 选项，它们展示信息的格式基本一致，但是详尽程度不一：

$ git log --pretty=oneline

ca82a6dff817ec66f44342007202690a93763949 changed the version number

085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test

a11bef06a3f659402fe7563abf99ad00de2209e6 first commit

最有意思的是 format ，可以定制记录的显示格式。 这样的输出对后期提取分析格外有用——因为你知道输出的格式不会随着 Git 的更新而发生改变：

$ git log --pretty=format:"%h - %an, %ar : %s"

ca82a6d - Scott Chacon, 6 years ago : changed the version number

085bb3b - Scott Chacon, 6 years ago : removed unnecessary test

a11bef0 - Scott Chacon, 6 years ago : first commit

[git log --pretty=format 常用的选项](https://git-scm.com/book/zh/v2/ch00/pretty_format) 列出了 format 接受的常用格式占位符的写法及其代表的意义。

| Table 1. git log --pretty=format 常用的选项 | |
| --- | --- |
| **选项** | **说明** |
| %H | 提交的完整哈希值 |
| %h | 提交的简写哈希值 |
| %T | 树的完整哈希值 |
| %t | 树的简写哈希值 |
| %P | 父提交的完整哈希值 |
| %p | 父提交的简写哈希值 |
| %an | 作者名字 |
| %ae | 作者的电子邮件地址 |
| %ad | 作者修订日期（可以用 --date=选项 来定制格式） |
| %ar | 作者修订日期，按多久以前的方式显示 |
| %cn | 提交者的名字 |
| %ce | 提交者的电子邮件地址 |
| %cd | 提交日期 |
| %cr | 提交日期（距今多长时间） |
| %s | 提交说明 |

你一定奇怪 **作者** 和 **提交者** 之间究竟有何差别， 其实作者指的是实际作出修改的人，提交者指的是最后将此工作成果提交到仓库的人。 所以，当你为某个项目发布补丁，然后某个核心成员将你的补丁并入项目时，你就是作者，而那个核心成员就是提交者。 我们会在 [分布式 Git](https://git-scm.com/book/zh/v2/ch00/ch05-distributed-git) 再详细介绍两者之间的细微差别。

当 oneline 或 format 与另一个 log 选项 --graph 结合使用时尤其有用。 这个选项添加了一些 ASCII 字符串来形象地展示你的分支、合并历史：

$ git log --pretty=format:"%h %s" --graph

\* 2d3acf9 ignore errors from SIGCHLD on trap

\* 5e3ee11 Merge branch 'master' of git://github.com/dustin/grit

|\

| \* 420eac9 Added a method for getting the current branch.

\* | 30e367c timeout code and tests

\* | 5a09431 add timeout protection to grit

\* | e1193f8 support for heads with slashes in them

|/

\* d6016bc require time for xmlschema

\* 11d191e Merge branch 'defunkt' into local

这种输出类型会在我们下一章学完分支与合并以后变得更加有趣。

以上只是简单介绍了一些 git log 命令支持的选项。 [git log 的常用选项](https://git-scm.com/book/zh/v2/ch00/log_options) 列出了我们目前涉及到的和没涉及到的选项，以及它们是如何影响 log 命令的输出的：

| Table 2. git log 的常用选项 | |
| --- | --- |
| **选项** | **说明** |
| -p | 按补丁格式显示每个提交引入的差异。 |
| --stat | 显示每次提交的文件修改统计信息。 |
| --shortstat | 只显示 --stat 中最后的行数修改添加移除统计。 |
| --name-only | 仅在提交信息后显示已修改的文件清单。 |
| --name-status | 显示新增、修改、删除的文件清单。 |
| --abbrev-commit | 仅显示 SHA-1 校验和所有 40 个字符中的前几个字符。 |
| --relative-date | 使用较短的相对时间而不是完整格式显示日期（比如“2 weeks ago”）。 |
| --graph | 在日志旁以 ASCII 图形显示分支与合并历史。 |
| --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（用来定义自己的格式）。 |
| --oneline | --pretty=oneline --abbrev-commit 合用的简写。 |

### 限制输出长度

除了定制输出格式的选项之外，git log 还有许多非常实用的限制输出长度的选项，也就是只输出一部分的提交。 之前你已经看到过 -2 选项了，它只会显示最近的两条提交， 实际上，你可以使用类似 -<n> 的选项，其中的 n 可以是任何整数，表示仅显示最近的 n 条提交。 不过实践中这个选项不是很常用，因为 Git 默认会将所有的输出传送到分页程序中，所以你一次只会看到一页的内容。

但是，类似 --since 和 --until 这种按照时间作限制的选项很有用。 例如，下面的命令会列出最近两周的所有提交：

$ git log --since=2.weeks

该命令可用的格式十分丰富——可以是类似 "2008-01-15" 的具体的某一天，也可以是类似 "2 years 1 day 3 minutes ago" 的相对日期。

还可以过滤出匹配指定条件的提交。 用 --author 选项显示指定作者的提交，用 --grep 选项搜索提交说明中的关键字。

|  |  |
| --- | --- |
| **Note** | 你可以指定多个 --author 和 --grep 搜索条件，这样会只输出匹配 **任意** --author 模式和 **任意** --grep 模式的提交。然而，如果你添加了 --all-match 选项， 则只会输出匹配 **所有** --grep 模式的提交。 |

另一个非常有用的过滤器是 -S（俗称“pickaxe”选项，取“用鹤嘴锄在土里捡石头”之意）， 它接受一个字符串参数，并且只会显示那些添加或删除了该字符串的提交。 假设你想找出添加或删除了对某一个特定函数的引用的提交，可以调用：

$ git log -S function\_name

最后一个很实用的 git log 选项是路径（path）， 如果只关心某些文件或者目录的历史提交，可以在 git log 选项的最后指定它们的路径。 因为是放在最后位置上的选项，所以用两个短划线（--）隔开之前的选项和后面限定的路径名。

在 [限制 git log 输出的选项](https://git-scm.com/book/zh/v2/ch00/limit_options) 中列出了常用的选项

| Table 3. 限制 git log 输出的选项 | |
| --- | --- |
| **选项** | **说明** |
| -<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。 |
| --until, --before | 仅显示指定时间之前的提交。 |
| --author | 仅显示作者匹配指定字符串的提交。 |
| --committer | 仅显示提交者匹配指定字符串的提交。 |
| --grep | 仅显示提交说明中包含指定字符串的提交。 |
| -S | 仅显示添加或删除内容匹配指定字符串的提交。 |

来看一个实际的例子，如果要在 Git 源码库中查看 Junio Hamano 在 2008 年 10 月其间， 除了合并提交之外的哪一个提交修改了测试文件，可以使用下面的命令：

$ git log --pretty="%h - %s" --author='Junio C Hamano' --since="2008-10-01" \

--before="2008-11-01" --no-merges -- t/

5610e3b - Fix testcase failure when extended attributes are in use

acd3b9e - Enhance hold\_lock\_file\_for\_{update,append}() API

f563754 - demonstrate breakage of detached checkout with symbolic link HEAD

d1a43f2 - reset --hard/read-tree --reset -u: remove unmerged new paths

51a94af - Fix "checkout --track -b newbranch" on detached HEAD

b0ad11e - pull: allow "git pull origin $something:$current\_branch" into an unborn branch

在近 40000 条提交中，上面的输出仅列出了符合条件的 6 条记录。

|  |  |
| --- | --- |
| **Tip** | **隐藏合并提交**  按照你代码仓库的工作流程，记录中可能有为数不少的合并提交，它们所包含的信息通常并不多。 为了避免显示的合并提交弄乱历史记录，可以为 log 加上 --no-merges 选项。 |

## 撤消操作

### 撤消操作

在任何一个阶段，你都有可能想要撤消某些操作。 这里，我们将会学习几个撤消你所做修改的基本工具。 注意，有些撤消操作是不可逆的。 这是在使用 Git 的过程中，会因为操作失误而导致之前的工作丢失的少有的几个地方之一。

**有时候我们提交完了才发现漏掉了几个文件没有添加，或者提交信息写错了。 此时，可以运行带有 --amend 选项的提交命令来重新提交：**

$ 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。

文本编辑器启动后，可以看到之前的提交信息。 编辑后保存会覆盖原来的提交信息。

例如，你提交后发现忘记了暂存某些需要的修改，可以像下面这样操作：

$ 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。

|  |  |
| --- | --- |
| **Note** | 当你在修补最后的提交时，与其说是修复旧提交，倒不如说是完全用一个 **新的提交** 替换旧的提交， 理解这一点非常重要。从效果上来说，就像是旧有的提交从未存在过一样，它并不会出现在仓库的历史中。  修补提交最明显的价值是可以稍微改进你最后的提交，而不会让“啊，忘了添加一个文件”或者 “小修补，修正笔误”这种提交信息弄乱你的仓库历史。 |

### 取消暂存的文件

接下来的两个小节演示如何操作暂存区和工作目录中已修改的文件。 这些命令在修改文件状态的同时，也会提示如何撤消操作。 例如，你已经修改了两个文件并且想要将它们作为两次独立的修改提交， 但是却意外地输入 git add \* 暂存了它们两个。如何只取消暂存两个中的一个呢？ git status 命令提示了你：

$ git add \*

$ git status

On branch master

Changes to be committed:

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

renamed: README.md -> README

modified: CONTRIBUTING.md

在 “Changes to be committed” 文字正下方，提示使用**git reset HEAD <file>... 来取消暂存。** 所以，我们可以这样来取消暂存 CONTRIBUTING.md 文件：

$ git reset HEAD CONTRIBUTING.md

Unstaged changes after reset:

M CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

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

renamed: README.md -> README

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: CONTRIBUTING.md

这个命令有点儿奇怪，但是起作用了。 CONTRIBUTING.md 文件已经是修改未暂存的状态了。

|  |  |
| --- | --- |
| **Note** | git reset 确实是个危险的命令，如果加上了 --hard 选项则更是如此。 然而在上述场景中，工作目录中的文件尚未修改，因此相对安全一些。 |

到目前为止这个神奇的调用就是你需要对 git reset 命令了解的全部。 我们将会在 [重置揭密](https://git-scm.com/book/zh/v2/ch00/_git_reset) 中了解 reset 的更多细节以及如何掌握它做一些真正有趣的事。

### 撤消对文件的修改

如果你并不想保留对 CONTRIBUTING.md 文件的修改怎么办？ 你该如何方便地撤消修改——将它还原成上次提交时的样子（或者刚克隆完的样子，或者刚把它放入工作目录时的样子）？ 幸运的是，git status 也告诉了你应该如何做。 在最后一个例子中，未暂存区域是这样：

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: CONTRIBUTING.md

它非常清楚地告诉了你**如何撤消之前所做的修改**。 让我们来按照提示执行：

$ git checkout -- CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

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

renamed: README.md -> README

可以看到那些修改已经被撤消了。

|  |  |
| --- | --- |
| **Important** | 请务必记得 git checkout -- <file> 是一个危险的命令。 你对那个文件在本地的任何修改都会消失——Git 会用最近提交的版本覆盖掉它。 除非你确实清楚不想要对那个文件的本地修改了，否则请不要使用这个命令。 |

如果你仍然想保留对那个文件做出的修改，但是现在仍然需要撤消，我们将会在 [Git 分支](https://git-scm.com/book/zh/v2/ch00/ch03-git-branching) 介绍保存进度与分支，这通常是更好的做法。

记住，在 Git 中任何 **已提交** 的东西几乎总是可以恢复的。 甚至那些被删除的分支中的提交或使用 --amend 选项覆盖的提交也可以恢复 （阅读 [数据恢复](https://git-scm.com/book/zh/v2/ch00/_data_recovery) 了解数据恢复）。 然而，任何你未提交的东西丢失后很可能再也找不到了。

## 远程仓库的使用

### 远程仓库的使用

为了能在任意 Git 项目上协作，你需要知道如何管理自己的远程仓库。 远程仓库是指托管在因特网或其他网络中的你的项目的版本库。 你可以有好几个远程仓库，通常有些仓库对你只读，有些则可以读写。 与他人协作涉及管理远程仓库以及根据需要推送或拉取数据。 管理远程仓库包括了解如何添加远程仓库、移除无效的远程仓库、管理不同的远程分支并定义它们是否被跟踪等等。 在本节中，我们将介绍一部分远程管理的技能。

### 查看远程仓库

如果想查看你已经配置的远程仓库服务器，可以运行 git remote 命令。 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：

$ git clone https://github.com/schacon/ticgit

Cloning into 'ticgit'...

remote: Reusing existing pack: 1857, done.

remote: Total 1857 (delta 0), reused 0 (delta 0)

Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.

Resolving deltas: 100% (772/772), done.

Checking connectivity... done.

$ cd ticgit

$ git remote

origin

你也可以指定选项 -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。

$ git remote -v

origin https://github.com/schacon/ticgit (fetch)

origin https://github.com/schacon/ticgit (push)

如果你的远程仓库不止一个，该命令会将它们全部列出。 例如，与几个协作者合作的，拥有多个远程仓库的仓库看起来像下面这样：

$ cd grit

$ git remote -v

bakkdoor https://github.com/bakkdoor/grit (fetch)

bakkdoor https://github.com/bakkdoor/grit (push)

cho45 https://github.com/cho45/grit (fetch)

cho45 https://github.com/cho45/grit (push)

defunkt https://github.com/defunkt/grit (fetch)

defunkt https://github.com/defunkt/grit (push)

koke git://github.com/koke/grit.git (fetch)

koke git://github.com/koke/grit.git (push)

origin git@github.com:mojombo/grit.git (fetch)

origin git@github.com:mojombo/grit.git (push)

这表示我们能非常方便地拉取其它用户的贡献。我们还可以拥有向他们推送的权限，这里暂不详述。

注意这些远程仓库使用了不同的协议。我们将会在 [在服务器上搭建 Git](https://git-scm.com/book/zh/v2/ch00/_getting_git_on_a_server) 中了解关于它们的更多信息。

### 添加远程仓库

我们在之前的章节中已经提到并展示了 git clone 命令是如何自行添加远程仓库的， 不过这里将告诉你如何自己来添加它。 运行 git remote add <shortname> <url> 添加一个新的远程 Git 仓库，同时指定一个方便使用的简写：

$ git remote

origin

$ git remote add pb https://github.com/paulboone/ticgit

$ git remote -v

origin https://github.com/schacon/ticgit (fetch)

origin https://github.com/schacon/ticgit (push)

pb https://github.com/paulboone/ticgit (fetch)

pb https://github.com/paulboone/ticgit (push)

现在你可以在命令行中使用字符串 pb 来代替整个 URL。 例如，如果你想拉取 Paul 的仓库中有但你没有的信息，可以运行 git fetch pb：

$ git fetch pb

remote: Counting objects: 43, done.

remote: Compressing objects: 100% (36/36), done.

remote: Total 43 (delta 10), reused 31 (delta 5)

Unpacking objects: 100% (43/43), done.

From https://github.com/paulboone/ticgit

\* [new branch] master -> pb/master

\* [new branch] ticgit -> pb/ticgit

现在 Paul 的 master 分支可以在本地通过 pb/master 访问到——你可以将它合并到自己的某个分支中， 或者如果你想要查看它的话，可以检出一个指向该点的本地分支。 （我们将会在 [Git 分支](https://git-scm.com/book/zh/v2/ch00/ch03-git-branching) 中详细介绍什么是分支以及如何使用分支。）

### 从远程仓库中抓取与拉取

就如刚才所见，从远程仓库中获得数据，可以执行：

$ git fetch <remote>

这个命令会访问远程仓库，从中拉取所有你还没有的数据。 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。 必须注意 git fetch 命令只会将数据下载到你的本地仓库——它并不会自动合并或修改你当前的工作。 当准备好时你必须手动将其合并入你的工作。

如果你的当前分支设置了跟踪远程分支（阅读下一节和 [Git 分支](https://git-scm.com/book/zh/v2/ch00/ch03-git-branching) 了解更多信息）， 那么可以用 git pull 命令来自动抓取后合并该远程分支到当前分支。 这或许是个更加简单舒服的工作流程。默认情况下，git clone 命令会自动设置本地 master 分支跟踪克隆的远程仓库的 master 分支（或其它名字的默认分支）。 运行 git pull 通常会从最初克隆的服务器上抓取数据并自动尝试合并到当前所在的分支。

### 推送到远程仓库

当你想分享你的项目时，必须将其推送到上游。 这个命令很简单：git push <remote> <branch>。 当你想要将 master 分支推送到 origin 服务器时（再次说明，克隆时通常会自动帮你设置好那两个名字）， 那么运行这个命令就可以将你所做的备份到服务器：

$ git push origin master

只有当你有所克隆服务器的写入权限，并且之前没有人推送过时，这条命令才能生效。 当你和其他人在同一时间克隆，他们先推送到上游然后你再推送到上游，你的推送就会毫无疑问地被拒绝。 你必须先抓取他们的工作并将其合并进你的工作后才能推送。 阅读 [Git 分支](https://git-scm.com/book/zh/v2/ch00/ch03-git-branching) 了解如何推送到远程仓库服务器的详细信息。

### 查看某个远程仓库

如果想要查看某一个远程仓库的更多信息，可以使用 git remote show <remote> 命令。 如果想以一个特定的缩写名运行这个命令，例如 origin，会得到像下面类似的信息：

$ git remote show origin

\* remote origin

Fetch URL: https://github.com/schacon/ticgit

Push URL: https://github.com/schacon/ticgit

HEAD branch: master

Remote branches:

master tracked

dev-branch tracked

Local branch configured for 'git pull':

master merges with remote master

Local ref configured for 'git push':

master pushes to master (up to date)

它同样会列出远程仓库的 URL 与跟踪分支的信息。 这些信息非常有用，它告诉你正处于 master 分支，并且如果运行 git pull， 就会抓取所有的远程引用，然后将远程 master 分支合并到本地 master 分支。 它也会列出拉取到的所有远程引用。

这是一个经常遇到的简单例子。 如果你是 Git 的重度使用者，那么还可以通过 git remote show 看到更多的信息。

$ git remote show origin

\* remote origin

URL: https://github.com/my-org/complex-project

Fetch URL: https://github.com/my-org/complex-project

Push URL: https://github.com/my-org/complex-project

HEAD branch: master

Remote branches:

master tracked

dev-branch tracked

markdown-strip tracked

issue-43 new (next fetch will store in remotes/origin)

issue-45 new (next fetch will store in remotes/origin)

refs/remotes/origin/issue-11 stale (use 'git remote prune' to remove)

Local branches configured for 'git pull':

dev-branch merges with remote dev-branch

master merges with remote master

Local refs configured for 'git push':

dev-branch pushes to dev-branch (up to date)

markdown-strip pushes to markdown-strip (up to date)

master pushes to master (up to date)

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 时哪些本地分支可以与它跟踪的远程分支自动合并。

### 远程仓库的重命名与移除

你可以运行 git remote rename 来修改一个远程仓库的简写名。 例如，想要将 pb 重命名为 paul，可以用 git remote rename 这样做：

$ git remote rename pb paul

$ git remote

origin

paul

值得注意的是这同样也会修改你所有远程跟踪的分支名字。 那些过去引用 pb/master 的现在会引用 paul/master。

如果因为一些原因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——可以使用 git remote remove 或 git remote rm ：

$ git remote remove paul

$ git remote

origin

一旦你使用这种方式删除了一个远程仓库，那么所有和这个远程仓库相关的远程跟踪分支以及配置信息也会一起被删除。

## 打标签

### 打标签

像其他版本控制系统（VCS）一样，Git 可以给仓库历史中的某一个提交打上标签，以示重要。 比较有代表性的是人们会使用这个功能来标记发布结点（ v1.0 、 v2.0 等等）。 在本节中，你将会学习如何列出已有的标签、如何创建和删除新的标签、以及不同类型的标签分别是什么。

### 列出标签

在 Git 中列出已有的标签非常简单，只需要输入 git tag （可带上可选的 -l 选项 --list）：

$ git tag

v1.0

v2.0

这个命令以字母顺序列出标签，但是它们显示的顺序并不重要。

你也可以按照特定的模式查找标签。 例如，Git 自身的源代码仓库包含标签的数量超过 500 个。 如果只对 1.8.5 系列感兴趣，可以运行：

$ git tag -l "v1.8.5\*"

v1.8.5

v1.8.5-rc0

v1.8.5-rc1

v1.8.5-rc2

v1.8.5-rc3

v1.8.5.1

v1.8.5.2

v1.8.5.3

v1.8.5.4

v1.8.5.5

|  |  |
| --- | --- |
| **Note** | **按照通配符列出标签需要 -l 或 --list 选项**  如果你只想要完整的标签列表，那么运行 git tag 就会默认假定你想要一个列表，它会直接给你列出来， 此时的 -l 或 --list 是可选的。  然而，如果你提供了一个匹配标签名的通配模式，那么 -l 或 --list 就是强制使用的。 |

### 创建标签

Git 支持两种标签：轻量标签（lightweight）与附注标签（annotated）。

轻量标签很像一个不会改变的分支——它只是某个特定提交的引用。

而附注标签是存储在 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， 此外还有一个标签信息，并且可以使用 GNU Privacy Guard （GPG）签名并验证。 通常会建议创建附注标签，这样你可以拥有以上所有信息。但是如果你只是想用一个临时的标签， 或者因为某些原因不想要保存这些信息，那么也可以用轻量标签。

### 附注标签

在 Git 中创建附注标签十分简单。 最简单的方式是当你在运行 tag 命令时指定 -a 选项：

$ git tag -a v1.4 -m "my version 1.4"

$ git tag

v0.1

v1.3

v1.4

-m 选项指定了一条将会存储在标签中的信息。 如果没有为附注标签指定一条信息，Git 会启动编辑器要求你输入信息。

通过使用 git show 命令可以看到标签信息和与之对应的提交信息：

$ git show v1.4

tag v1.4

Tagger: Ben Straub <ben@straub.cc>

Date: Sat May 3 20:19:12 2014 -0700

my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

输出显示了打标签者的信息、打标签的日期时间、附注信息，然后显示具体的提交信息。

### 轻量标签

另一种给提交打标签的方式是使用轻量标签。 轻量标签本质上是将提交校验和存储到一个文件中——没有保存任何其他信息。 创建轻量标签，不需要使用 -a、-s 或 -m 选项，只需要提供标签名字：

$ git tag v1.4-lw

$ git tag

v0.1

v1.3

v1.4

v1.4-lw

v1.5

这时，如果在标签上运行 git show，你不会看到额外的标签信息。 命令只会显示出提交信息：

$ git show v1.4-lw

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

### 后期打标签

你也可以对过去的提交打标签。 假设提交历史是这样的：

$ git log --pretty=oneline

15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'

a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support

0d52aaab4479697da7686c15f77a3d64d9165190 one more thing

6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'

0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function

4682c3261057305bdd616e23b64b0857d832627b added a todo file

166ae0c4d3f420721acbb115cc33848dfcc2121a started write support

9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile

964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo

8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme

现在，假设在 v1.2 时你忘记给项目打标签，也就是在 “updated rakefile” 提交。 你可以在之后补上标签。 要在那个提交上打标签，你需要在命令的末尾指定提交的校验和（或部分校验和）：

$ git tag -a v1.2 9fceb02

可以看到你已经在那次提交上打上标签了：

$ git tag

v0.1

v1.2

v1.3

v1.4

v1.4-lw

v1.5

$ git show v1.2

tag v1.2

Tagger: Scott Chacon <schacon@gee-mail.com>

Date: Mon Feb 9 15:32:16 2009 -0800

version 1.2

commit 9fceb02d0ae598e95dc970b74767f19372d61af8

Author: Magnus Chacon <mchacon@gee-mail.com>

Date: Sun Apr 27 20:43:35 2008 -0700

updated rakefile

...

### 共享标签

默认情况下，git push 命令并不会传送标签到远程仓库服务器上。 在创建完标签后你必须显式地推送标签到共享服务器上。 这个过程就像共享远程分支一样——你可以运行 git push origin <tagname>。

$ git push origin v1.5

Counting objects: 14, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (12/12), done.

Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.

Total 14 (delta 3), reused 0 (delta 0)

To git@github.com:schacon/simplegit.git

\* [new tag] v1.5 -> v1.5

如果想要一次性推送很多标签，也可以使用带有 --tags 选项的 git push 命令。 这将会把所有不在远程仓库服务器上的标签全部传送到那里。

$ git push origin --tags

Counting objects: 1, done.

Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.

Total 1 (delta 0), reused 0 (delta 0)

To git@github.com:schacon/simplegit.git

\* [new tag] v1.4 -> v1.4

\* [new tag] v1.4-lw -> v1.4-lw

现在，当其他人从仓库中克隆或拉取，他们也能得到你的那些标签。

|  |  |
| --- | --- |
| **Note** | **git push 推送两种标签**  使用 git push <remote> --tags 推送标签并不会区分轻量标签和附注标签， 没有简单的选项能够让你只选择推送一种标签。 |

### 删除标签

要删除掉你本地仓库上的标签，可以使用命令 git tag -d <tagname>。 例如，可以使用以下命令删除一个轻量标签：

$ git tag -d v1.4-lw

Deleted tag 'v1.4-lw' (was e7d5add)

注意上述命令并不会从任何远程仓库中移除这个标签，你必须用 git push <remote> :refs/tags/<tagname> 来更新你的远程仓库：

第一种变体是 git push <remote> :refs/tags/<tagname> ：

$ git push origin :refs/tags/v1.4-lw

To /git@github.com:schacon/simplegit.git

- [deleted] v1.4-lw

上面这种操作的含义是，将冒号前面的空值推送到远程标签名，从而高效地删除它。

第二种更直观的删除远程标签的方式是：

$ git push origin --delete <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 命令， 虽然这会使你的仓库处于“分离头指针（detached HEAD）”的状态——这个状态有些不好的副作用：

$ git checkout 2.0.0

Note: checking out '2.0.0'.

You are in 'detached HEAD' state. You can look around, make experimental

changes and commit them, and you can discard any commits you make in this

state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may

do so (now or later) by using -b with the checkout command again. Example:

git checkout -b <new-branch>

HEAD is now at 99ada87... Merge pull request #89 from schacon/appendix-final

$ git checkout 2.0-beta-0.1

Previous HEAD position was 99ada87... Merge pull request #89 from schacon/appendix-final

HEAD is now at df3f601... add atlas.json and cover image

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，除非通过确切的提交哈希才能访问。 因此，如果你需要进行更改，比如你要修复旧版本中的错误，那么通常需要创建一个新分支：

$ git checkout -b version2 v2.0.0

Switched to a new branch 'version2'

如果在这之后又进行了一次提交，version2 分支就会因为这个改动向前移动， 此时它就会和 v2.0.0 标签稍微有些不同，这时就要当心了。

## Git 别名

### Git 别名

在我们结束本章 Git 基础之前，正好有一个小技巧可以使你的 Git 体验更简单、容易、熟悉：别名。 我们不会在之后的章节中引用到或假定你使用过它们，但是你大概应该知道如何使用它们。

Git 并不会在你输入部分命令时自动推断出你想要的命令。 如果不想每次都输入完整的 Git 命令，可以通过 git config 文件来轻松地为每一个命令设置一个别名。 这里有一些例子你可以试试：

$ git config --global alias.co checkout

$ git config --global alias.br branch

$ git config --global alias.ci commit

$ git config --global alias.st status

这意味着，当要输入 git commit 时，只需要输入 git ci。 随着你继续不断地使用 Git，可能也会经常使用其他命令，所以创建别名时不要犹豫。

在创建你认为应该存在的命令时这个技术会很有用。 例如，为了解决取消暂存文件的易用性问题，可以向 Git 中添加你自己的取消暂存别名：

$ git config --global alias.unstage 'reset HEAD --'

这会使下面的两个命令等价：

$ git unstage fileA

$ git reset HEAD -- fileA

这样看起来更清楚一些。 通常也会添加一个 last 命令，像这样：

$ git config --global alias.last 'log -1 HEAD'

这样，可以轻松地看到最后一次提交：

$ git last

commit 66938dae3329c7aebe598c2246a8e6af90d04646

Author: Josh Goebel <dreamer3@example.com>

Date: Tue Aug 26 19:48:51 2008 +0800

test for current head

Signed-off-by: Scott Chacon <schacon@example.com>

可以看出，Git 只是简单地将别名替换为对应的命令。 然而，你可能想要执行外部命令，而不是一个 Git 子命令。 如果是那样的话，可以在命令前面加入 ! 符号。 如果你自己要写一些与 Git 仓库协作的工具的话，那会很有用。 我们现在演示将 git visual 定义为 gitk 的别名：

$ git config --global alias.visual '!gitk'

# Git 分支

## Git 分支 -分支简介

几乎所有的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把你的工作从开发主线上分离开来，以免影响开发主线。 在很多版本控制系统中，这是一个略微低效的过程——常常需要完全创建一个源代码目录的副本。对于大项目来说，这样的过程会耗费很多时间。

有人把 Git 的分支模型称为它的“必杀技特性”，也正因为这一特性，使得 Git 从众多版本控制系统中脱颖而出。 为何 Git 的分支模型如此出众呢？ Git 处理分支的方式可谓是难以置信的轻量，创建新分支这一操作几乎能在瞬间完成，并且在不同分支之间的切换操作也是一样便捷。 与许多其它版本控制系统不同，Git 鼓励在工作流程中频繁地使用分支与合并，哪怕一天之内进行许多次。 理解和精通这一特性，你便会意识到 Git 是如此的强大而又独特，并且从此真正改变你的开发方式。

### 分支简介

为了真正理解 Git 处理分支的方式，我们需要回顾一下 Git 是如何保存数据的。Git 保存的不是文件的变化或者差异，而是一系列不同时刻的 **快照** 。

在进行提交操作时，Git 会保存一个提交对象（commit object）。 知道了 Git 保存数据的方式，我们可以很自然的想到——该提交对象会包含一个指向暂存内容快照的指针。 但不仅仅是这样，该提交对象还包含了作者的姓名和邮箱、提交时输入的信息以及指向它的父对象的指针。 首次提交产生的提交对象没有父对象，普通提交操作产生的提交对象有一个父对象， 而由多个分支合并产生的提交对象有多个父对象，

为了更加形象地说明，我们假设现在有一个工作目录，里面包含了三个将要被暂存和提交的文件。 暂存操作会为每一个文件计算校验和（使用我们在 [起步](https://git-scm.com/book/zh/v2/ch00/ch01-getting-started) 中提到的 SHA-1 哈希算法），然后会把当前版本的文件快照保存到 Git 仓库中 （Git 使用 **blob** 对象来保存它们），最终将校验和加入到暂存区域等待提交：

$ git add README test.rb LICENSE

$ git commit -m 'The initial commit of my project'

当使用 git commit 进行提交操作时，Git 会先计算每一个子目录（本例中只有项目根目录）的校验和， 然后在 Git 仓库中这些校验和保存为树对象。随后，Git 便会创建一个提交对象， 它除了包含上面提到的那些信息外，还包含指向这个树对象（项目根目录）的指针。 如此一来，Git 就可以在需要的时候重现此次保存的快照。

现在，Git 仓库中有五个对象：三个 **blob** 对象（保存着文件快照）、一个 **树** 对象 （记录着目录结构和 blob 对象索引）以及一个 **提交** 对象（包含着指向前述树对象的指针和所有提交信息）。



Figure 9. 首次提交对象及其树结构

做些修改后再次提交，那么这次产生的提交对象会包含一个指向上次提交对象（父对象）的指针。



Figure 10. 提交对象及其父对象

Git 的分支，其实本质上仅仅是指向提交对象的可变指针。 Git 的默认分支名字是 master。 在多次提交操作之后，你其实已经有一个指向最后那个提交对象的 master 分支。 master 分支会在每次提交时自动向前移动。

|  |  |
| --- | --- |
| **Note** | Git 的 master 分支并不是一个特殊分支。 它就跟其它分支完全没有区别。 之所以几乎每一个仓库都有 master 分支，是因为 git init 命令默认创建它，并且大多数人都懒得去改动它。 |



Figure 11. 分支及其提交历史

### 分支创建

Git 是怎么创建新分支的呢？ 很简单，它只是为你创建了一个可以移动的新的指针。 比如，创建一个 testing 分支， 你需要使用 git branch 命令：

$ git branch testing

这会在当前所在的提交对象上创建一个指针。



Figure 12. 两个指向相同提交历史的分支

那么，Git 又是怎么知道当前在哪一个分支上呢？ 也很简单，它有一个名为 HEAD 的特殊指针。 请注意它和许多其它版本控制系统（如 Subversion 或 CVS）里的 HEAD 概念完全不同。 在 Git 中，它是一个指针，指向当前所在的本地分支（译注：将 HEAD 想象为当前分支的别名）。 在本例中，你仍然在 master 分支上。 因为 git branch 命令仅仅 **创建** 一个新分支，并不会自动切换到新分支中去。



Figure 13. HEAD 指向当前所在的分支

你可以简单地使用 git log 命令查看各个分支当前所指的对象。 提供这一功能的参数是 --decorate。

$ git log --oneline --decorate

f30ab (HEAD -> master, testing) add feature #32 - ability to add new formats to the central interface

34ac2 Fixed bug #1328 - stack overflow under certain conditions

98ca9 The initial commit of my project

正如你所见，当前 master 和 testing 分支均指向校验和以 f30ab 开头的提交对象。

### 分支切换

要切换到一个已存在的分支，你需要使用 git checkout 命令。 我们现在切换到新创建的 testing 分支去：

$ git checkout testing

这样 HEAD 就指向 testing 分支了。



Figure 14. HEAD 指向当前所在的分支

那么，这样的实现方式会给我们带来什么好处呢？ 现在不妨再提交一次：

$ vim test.rb

$ git commit -a -m 'made a change'



Figure 15. HEAD 分支随着提交操作自动向前移动

如图所示，你的 testing 分支向前移动了，但是 master 分支却没有，它仍然指向运行 git checkout 时所指的对象。 这就有意思了，现在我们切换回 master 分支看看：

$ git checkout master



Figure 16. 检出时 HEAD 随之移动

这条命令做了两件事。 一是使 HEAD 指回 master 分支，二是将工作目录恢复成 master 分支所指向的快照内容。 也就是说，你现在做修改的话，项目将始于一个较旧的版本。 本质上来讲，这就是忽略 testing 分支所做的修改，以便于向另一个方向进行开发。

|  |  |
| --- | --- |
| **Note** | **分支切换会改变你工作目录中的文件**  在切换分支时，一定要注意你工作目录里的文件会被改变。 如果是切换到一个较旧的分支，你的工作目录会恢复到该分支最后一次提交时的样子。 如果 Git 不能干净利落地完成这个任务，它将禁止切换分支。 |

我们不妨再稍微做些修改并提交：

$ vim test.rb

$ git commit -a -m 'made other changes'

现在，这个项目的提交历史已经产生了分叉（参见 [项目分叉历史](https://git-scm.com/book/zh/v2/ch00/divergent_history)）。 因为刚才你创建了一个新分支，并切换过去进行了一些工作，随后又切换回 master 分支进行了另外一些工作。 上述两次改动针对的是不同分支：你可以在不同分支间不断地来回切换和工作，并在时机成熟时将它们合并起来。 而所有这些工作，你需要的命令只有 branch、checkout 和 commit。



Figure 17. 项目分叉历史

你可以简单地使用 git log 命令查看分叉历史。 运行 git log --oneline --decorate --graph --all ，它会输出你的提交历史、各个分支的指向以及项目的分支分叉情况。

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

\* c2b9e (HEAD, master) made other changes

| \* 87ab2 (testing) made a change

|/

\* f30ab add feature #32 - ability to add new formats to the

\* 34ac2 fixed bug #1328 - stack overflow under certain conditions

\* 98ca9 initial commit of my project

由于 Git 的分支实质上仅是包含所指对象校验和（长度为 40 的 SHA-1 值字符串）的文件，所以它的创建和销毁都异常高效。 创建一个新分支就相当于往一个文件中写入 41 个字节（40 个字符和 1 个换行符），如此的简单能不快吗？

这与过去大多数版本控制系统形成了鲜明的对比，它们在创建分支时，将所有的项目文件都复制一遍，并保存到一个特定的目录。 完成这样繁琐的过程通常需要好几秒钟，有时甚至需要好几分钟。所需时间的长短，完全取决于项目的规模。 而在 Git 中，任何规模的项目都能在瞬间创建新分支。 同时，由于每次提交都会记录父对象，所以寻找恰当的合并基础（译注：即共同祖先）也是同样的简单和高效。 这些高效的特性使得 Git 鼓励开发人员频繁地创建和使用分支。

接下来，让我们看看你为什么应该这样做。

|  |  |
| --- | --- |
| **Note** | **创建新分支的同时切换过去**  通常我们会在创建一个新分支后立即切换过去，这可以用 git checkout -b <newbranchname> 一条命令搞定。 |

## Git 分支 - 分支的新建与合并

### 分支的新建与合并

让我们来看一个简单的分支新建与分支合并的例子，实际工作中你可能会用到类似的工作流。 你将经历如下步骤：

1. 开发某个网站。
2. 为实现某个新的用户需求，创建一个分支。
3. 在这个分支上开展工作。

正在此时，你突然接到一个电话说有个很严重的问题需要紧急修补。 你将按照如下方式来处理：

1. 切换到你的线上分支（production branch）。
2. 为这个紧急任务新建一个分支，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，最后将改动推送到线上分支。
4. 切换回你最初工作的分支上，继续工作。

### 新建分支

首先，我们假设你正在你的项目上工作，并且在 master 分支上已经有了一些提交。



Figure 18. 一个简单提交历史

现在，你已经决定要解决你的公司使用的问题追踪系统中的 #53 问题。 想要新建一个分支并同时切换到那个分支上，你可以运行一个带有 -b 参数的 git checkout 命令：

$ git checkout -b iss53

Switched to a new branch "iss53"

它是下面两条命令的简写：

$ git branch iss53

$ git checkout iss53



Figure 19. 创建一个新分支指针

你继续在 #53 问题上工作，并且做了一些提交。 在此过程中，iss53 分支在不断的向前推进，因为你已经检出到该分支 （也就是说，你的 HEAD 指针指向了 iss53 分支）

$ vim index.html

$ git commit -a -m 'added a new footer [issue 53]'



Figure 20. iss53 分支随着工作的进展向前推进

现在你接到那个电话，有个紧急问题等待你来解决。 有了 Git 的帮助，你不必把这个紧急问题和 iss53 的修改混在一起， 你也不需要花大力气来还原关于 53# 问题的修改，然后再添加关于这个紧急问题的修改，最后将这个修改提交到线上分支。 你所要做的仅仅是切换回 master 分支。

但是，在你这么做之前，要留意你的工作目录和暂存区里那些还没有被提交的修改， 它可能会和你即将检出的分支产生冲突从而阻止 Git 切换到该分支。 最好的方法是，在你切换分支之前，保持好一个干净的状态。 有一些方法可以绕过这个问题（即，暂存（stashing） 和 修补提交（commit amending））， 我们会在 [贮藏与清理](https://git-scm.com/book/zh/v2/ch00/_git_stashing) 中看到关于这两个命令的介绍。 现在，我们假设你已经把你的修改全部提交了，这时你可以切换回 master 分支了：

$ git checkout master

Switched to branch 'master'

这个时候，你的工作目录和你在开始 #53 问题之前一模一样，现在你可以专心修复紧急问题了。 请牢记：当你切换分支的时候，Git 会重置你的工作目录，使其看起来像回到了你在那个分支上最后一次提交的样子。 Git 会自动添加、删除、修改文件以确保此时你的工作目录和这个分支最后一次提交时的样子一模一样。

接下来，你要修复这个紧急问题。 我们来建立一个 hotfix 分支，在该分支上工作直到问题解决：

$ git checkout -b hotfix

Switched to a new branch 'hotfix'

$ vim index.html

$ git commit -a -m 'fixed the broken email address'

[hotfix 1fb7853] fixed the broken email address

1 file changed, 2 insertions(+)



Figure 21. 基于 master 分支的紧急问题分支 hotfix branch

你可以运行你的测试，确保你的修改是正确的，然后将 hotfix 分支合并回你的 master 分支来部署到线上。 你可以使用 git merge 命令来达到上述目的：

$ git checkout master

$ git merge hotfix

Updating f42c576..3a0874c

Fast-forward

index.html | 2 ++

1 file changed, 2 insertions(+)

在合并的时候，你应该注意到了“快进（fast-forward）”这个词。 由于你想要合并的分支 hotfix 所指向的提交 C4 是你所在的提交 C2 的直接后继， 因此 Git 会直接将指针向前移动。换句话说，当你试图合并两个分支时， 如果顺着一个分支走下去能够到达另一个分支，那么 Git 在合并两者的时候， 只会简单的将指针向前推进（指针右移），因为这种情况下的合并操作没有需要解决的分歧——这就叫做 “快进（fast-forward）”。

现在，最新的修改已经在 master 分支所指向的提交快照中，你可以着手发布该修复了。



Figure 22. master 被快进到 hotfix

关于这个紧急问题的解决方案发布之后，你准备回到被打断之前时的工作中。 然而，你应该先删除 hotfix 分支，因为你已经不再需要它了 —— master 分支已经指向了同一个位置。 你可以使用带 -d 选项的 git branch 命令来删除分支：

$ git branch -d hotfix

Deleted branch hotfix (3a0874c).

现在你可以切换回你正在工作的分支继续你的工作，也就是针对 #53 问题的那个分支（iss53 分支）。

$ git checkout iss53

Switched to branch "iss53"

$ vim index.html

$ git commit -a -m 'finished the new footer [issue 53]'

[iss53 ad82d7a] finished the new footer [issue 53]

1 file changed, 1 insertion(+)



Figure 23. 继续在 iss53 分支上的工作

你在 hotfix 分支上所做的工作并没有包含到 iss53 分支中。 如果你需要拉取 hotfix 所做的修改，你可以使用 git merge master 命令将 master 分支合并入 iss53 分支，或者你也可以等到 iss53 分支完成其使命，再将其合并回 master 分支。

### 分支的合并

假设你已经修正了 #53 问题，并且打算将你的工作合并入 master 分支。 为此，你需要合并 iss53 分支到 master 分支，这和之前你合并 hotfix 分支所做的工作差不多。 你只需要检出到你想合并入的分支，然后运行 git merge 命令：

$ git checkout master

Switched to branch 'master'

$ git merge iss53

Merge made by the 'recursive' strategy.

index.html | 1 +

1 file changed, 1 insertion(+)

这和你之前合并 hotfix 分支的时候看起来有一点不一样。 在这种情况下，你的开发历史从一个更早的地方开始分叉开来（diverged）。 因为，master 分支所在提交并不是 iss53 分支所在提交的直接祖先，Git 不得不做一些额外的工作。 出现这种情况的时候，Git 会使用两个分支的末端所指的快照（C4 和 C5）以及这两个分支的公共祖先（C2），做一个简单的三方合并。



Figure 24. 一次典型合并中所用到的三个快照

和之前将分支指针向前推进所不同的是，Git 将此次三方合并的结果做了一个新的快照并且自动创建一个新的提交指向它。 这个被称作一次合并提交，它的特别之处在于他有不止一个父提交。



Figure 25. 一个合并提交

既然你的修改已经合并进来了，就不再需要 iss53 分支了。 现在你可以在任务追踪系统中关闭此项任务，并删除这个分支。

$ git branch -d iss53

### 遇到冲突时的分支合并

有时候合并操作不会如此顺利。 如果你在两个不同的分支中，对同一个文件的同一个部分进行了不同的修改，Git 就没法干净的合并它们。 如果你对 #53 问题的修改和有关 hotfix 分支的修改都涉及到同一个文件的同一处，在合并它们的时候就会产生合并冲突：

$ git merge iss53

Auto-merging index.html

CONFLICT (content): Merge conflict in index.html

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

此时 Git 做了合并，但是没有自动地创建一个新的合并提交。 Git 会暂停下来，等待你去解决合并产生的冲突。 你可以在合并冲突后的任意时刻使用 git status 命令来查看那些因包含合并冲突而处于未合并（unmerged）状态的文件：

$ git status

On branch master

You have unmerged paths.

(fix conflicts and run "git commit")

Unmerged paths:

(use "git add <file>..." to mark resolution)

both modified: index.html

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

任何因包含合并冲突而有待解决的文件，都会以未合并状态标识出来。 Git 会在有冲突的文件中加入标准的冲突解决标记，这样你可以打开这些包含冲突的文件然后手动解决冲突。 出现冲突的文件会包含一些特殊区段，看起来像下面这个样子：

<<<<<<< HEAD:index.html

<div id="footer">contact : email.support@github.com</div>

=======

<div id="footer">

please contact us at support@github.com

</div>

>>>>>>> iss53:index.html

这表示 HEAD 所指示的版本（也就是你的 master 分支所在的位置，因为你在运行 merge 命令的时候已经检出到了这个分支）在这个区段的上半部分（======= 的上半部分），而 iss53 分支所指示的版本在 ======= 的下半部分。 为了解决冲突，你必须选择使用由 ======= 分割的两部分中的一个，或者你也可以自行合并这些内容。 例如，你可以通过把这段内容换成下面的样子来解决冲突：

<div id="footer">

please contact us at email.support@github.com

</div>

上述的冲突解决方案仅保留了其中一个分支的修改，并且 <<<<<<< , ======= , 和 >>>>>>> 这些行被完全删除了。 在你解决了所有文件里的冲突之后，对每个文件使用 git add 命令来将其标记为冲突已解决。 一旦暂存这些原本有冲突的文件，Git 就会将它们标记为冲突已解决。

**如果你想使用图形化工具来解决冲突，你可以运行 git mergetool，该命令会为你启动一个合适的可视化合并工具**，并带领你一步一步解决这些冲突：

$ git mergetool

This message is displayed because 'merge.tool' is not configured.

See 'git mergetool --tool-help' or 'git help config' for more details.

'git mergetool' will now attempt to use one of the following tools:

opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge araxis bc3 codecompare vimdiff emerge

Merging:

index.html

Normal merge conflict for 'index.html':

{local}: modified file

{remote}: modified file

Hit return to start merge resolution tool (opendiff):

如果你想使用除默认工具（在这里 Git 使用 opendiff 做为默认的合并工具，因为作者在 Mac 上运行该程序） 外的其他合并工具，你可以在 “下列工具中（one of the following tools）” 这句后面看到所有支持的合并工具。 然后输入你喜欢的工具名字就可以了。

|  |  |
| --- | --- |
| **Note** | 如果你需要更加高级的工具来解决复杂的合并冲突，我们会在 [高级合并](https://git-scm.com/book/zh/v2/ch00/_advanced_merging) 介绍更多关于分支合并的内容。 |

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：

$ git status

On branch master

All conflicts fixed but you are still merging.

(use "git commit" to conclude merge)

Changes to be committed:

modified: index.html

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。 默认情况下提交信息看起来像下面这个样子：

Merge branch 'iss53'

Conflicts:

index.html

#

# It looks like you may be committing a merge.

# If this is not correct, please remove the file

# .git/MERGE\_HEAD

# and try again.

# Please enter the commit message for your changes. Lines starting

# with '#' will be ignored, and an empty message aborts the commit.

# On branch master

# All conflicts fixed but you are still merging.

#

# Changes to be committed:

# modified: index.html

#

如果你觉得上述的信息不够充分，不能完全体现分支合并的过程，你可以修改上述信息， 添加一些细节给未来检视这个合并的读者一些帮助，告诉他们你是如何解决合并冲突的，以及理由是什么。

## Git 分支 - 分支管理

### 分支管理

现在已经创建、合并、删除了一些分支，让我们看看一些常用的分支管理工具。

git branch 命令不只是可以创建与删除分支。 如果不加任何参数运行它，会得到当前所有分支的一个列表：

$ git branch

iss53

\* master

testing

注意 master 分支前的 \* 字符：它代表现在检出的那一个分支（也就是说，当前 HEAD 指针所指向的分支）。 这意味着如果在这时候提交，master 分支将会随着新的工作向前移动。 如果需要查看每一个分支的最后一次提交，可以运行 git branch -v 命令：

$ git branch -v

iss53 93b412c fix javascript issue

\* master 7a98805 Merge branch 'iss53'

testing 782fd34 add scott to the author list in the readmes

--merged 与 --no-merged 这两个有用的选项可以过滤这个列表中已经合并或尚未合并到当前分支的分支。 如果要查看哪些分支已经合并到当前分支，可以运行 git branch --merged：

$ git branch --merged

iss53

\* master

因为之前已经合并了 iss53 分支，所以现在看到它在列表中。 在这个列表中分支名字前没有 \* 号的分支通常可以使用 git branch -d 删除掉；你已经将它们的工作整合到了另一个分支，所以并不会失去任何东西。

查看所有包含未合并工作的分支，可以运行 git branch --no-merged：

$ git branch --no-merged

testing

这里显示了其他分支。 因为它包含了还未合并的工作，尝试使用 git branch -d 命令删除它时会失败：

$ git branch -d testing

error: The branch 'testing' is not fully merged.

If you are sure you want to delete it, run 'git branch -D testing'.

如果真的想要删除分支并丢掉那些工作，如同帮助信息里所指出的，可以使用 -D 选项强制删除它。

|  |  |
| --- | --- |
| **Tip** | 上面描述的选项 --merged 和 --no-merged 会在没有给定提交或分支名作为参数时， 分别列出已合并或未合并到 **当前** 分支的分支。  你总是可以提供一个附加的参数来查看其它分支的合并状态而不必检出它们。 例如，尚未合并到 master 分支的有哪些？  $ git checkout testing  $ git branch --no-merged master  topicA  featureB |

## Git 分支 - 分支开发工作流

### 分支开发工作流

现在你已经学会新建和合并分支，那么你可以或者应该用它来做些什么呢？ 在本节，我们会介绍一些常见的利用分支进行开发的工作流程。而正是由于分支管理的便捷， 才衍生出这些典型的工作模式，你可以根据项目实际情况选择一种用用看。

### 长期分支

因为 Git 使用简单的三方合并，所以就算在一段较长的时间内，反复把一个分支合并入另一个分支，也不是什么难事。 也就是说，在整个项目开发周期的不同阶段，你可以同时拥有多个开放的分支；你可以定期地把某些主题分支合并入其他分支中。

许多使用 Git 的开发者都喜欢使用这种方式来工作，比如只在 master 分支上保留完全稳定的代码——有可能仅仅是已经发布或即将发布的代码。 他们还有一些名为 develop 或者 next 的平行分支，被用来做后续开发或者测试稳定性——这些分支不必保持绝对稳定，但是一旦达到稳定状态，它们就可以被合并入 master 分支了。 这样，在确保这些已完成的主题分支（短期分支，比如之前的 iss53 分支）能够通过所有测试，并且不会引入更多 bug 之后，就可以合并入主干分支中，等待下一次的发布。

事实上我们刚才讨论的，是随着你的提交而不断右移的指针。 稳定分支的指针总是在提交历史中落后一大截，而前沿分支的指针往往比较靠前。



Figure 26. 趋于稳定分支的线性图

通常把他们想象成流水线（work silos）可能更好理解一点，那些经过测试考验的提交会被遴选到更加稳定的流水线上去。



Figure 27. 趋于稳定分支的流水线（“silo”）视图

你可以用这种方法维护不同层次的稳定性。 一些大型项目还有一个 proposed（建议） 或 pu: proposed updates（建议更新）分支，它可能因包含一些不成熟的内容而不能进入 next 或者 master 分支。 这么做的目的是使你的分支具有不同级别的稳定性；当它们具有一定程度的稳定性后，再把它们合并入具有更高级别稳定性的分支中。 再次强调一下，使用多个长期分支的方法并非必要，但是这么做通常很有帮助，尤其是当你在一个非常庞大或者复杂的项目中工作时。

### 主题分支

主题分支对任何规模的项目都适用。 主题分支是一种短期分支，它被用来实现单一特性或其相关工作。 也许你从来没有在其他的版本控制系统（VCS）上这么做过，因为在那些版本控制系统中创建和合并分支通常很费劲。 然而，在 Git 中一天之内多次创建、使用、合并、删除分支都很常见。

你已经在上一节中你创建的 iss53 和 hotfix 主题分支中看到过这种用法。 你在上一节用到的主题分支（iss53 和 hotfix 分支）中提交了一些更新，并且在它们合并入主干分支之后，你又删除了它们。 这项技术能使你快速并且完整地进行上下文切换（context-switch）——因为你的工作被分散到不同的流水线中，在不同的流水线中每个分支都仅与其目标特性相关，因此，在做代码审查之类的工作的时候就能更加容易地看出你做了哪些改动。 你可以把做出的改动在主题分支中保留几分钟、几天甚至几个月，等它们成熟之后再合并，而不用在乎它们建立的顺序或工作进度。

考虑这样一个例子，你在 master 分支上工作到 C1，这时为了解决一个问题而新建 iss91 分支，在 iss91 分支上工作到 C4，然而对于那个问题你又有了新的想法，于是你再新建一个 iss91v2 分支试图用另一种方法解决那个问题，接着你回到 master 分支工作了一会儿，你又冒出了一个不太确定的想法，你便在 C10 的时候新建一个 dumbidea 分支，并在上面做些实验。 你的提交历史看起来像下面这个样子：



Figure 28. 拥有多个主题分支的提交历史

现在，我们假设两件事情：你决定使用第二个方案来解决那个问题，即使用在 iss91v2 分支中方案。 另外，你将 dumbidea 分支拿给你的同事看过之后，结果发现这是个惊人之举。 这时你可以抛弃 iss91 分支（即丢弃 C5 和 C6 提交），然后把另外两个分支合并入主干分支。 最终你的提交历史看起来像下面这个样子：



Figure 29. 合并了 dumbidea 和 iss91v2 分支之后的提交历史

我们将会在 [分布式 Git](https://git-scm.com/book/zh/v2/ch00/ch05-distributed-git) 中向你揭示更多有关分支工作流的细节， 因此，请确保你阅读完那个章节之后，再来决定你的下个项目要使用什么样的分支策略（branching scheme）。

请牢记，当你做这么多操作的时候，这些分支全部都存于本地。 当你新建和合并分支的时候，所有这一切都只发生在你本地的 Git 版本库中 —— 没有与服务器发生交互。

## Git 分支 - 远程分支

### 远程分支

远程引用是对远程仓库的引用（指针），包括分支、标签等等。 你可以通过 git ls-remote <remote> 来显式地获得远程引用的完整列表， 或者通过 git remote show <remote> 获得远程分支的更多信息。 然而，一个更常见的做法是利用远程跟踪分支。

远程跟踪分支是远程分支状态的引用。它们是你无法移动的本地引用。一旦你进行了网络通信， Git 就会为你移动它们以精确反映远程仓库的状态。请将它们看做书签， 这样可以提醒你该分支在远程仓库中的位置就是你最后一次连接到它们的位置。

它们以 <remote>/<branch> 的形式命名。 例如，如果你想要看你最后一次与远程仓库 origin 通信时 master 分支的状态，你可以查看 origin/master 分支。 你与同事合作解决一个问题并且他们推送了一个 iss53 分支，你可能有自己的本地 iss53 分支， 然而在服务器上的分支会以 origin/iss53 来表示。

这可能有一点儿难以理解，让我们来看一个例子。 假设你的网络里有一个在 git.ourcompany.com 的 Git 服务器。 如果你从这里克隆，Git 的 clone 命令会为你自动将其命名为 origin，拉取它的所有数据， 创建一个指向它的 master 分支的指针，并且在本地将其命名为 origin/master。 Git 也会给你一个与 origin 的 master 分支在指向同一个地方的本地 master 分支，这样你就有工作的基础。

|  |  |
| --- | --- |
| **Note** | **“origin” 并无特殊含义**  远程仓库名字 “origin” 与分支名字 “master” 一样，在 Git 中并没有任何特别的含义一样。 同时 “master” 是当你运行 git init 时默认的起始分支名字，原因仅仅是它的广泛使用， “origin” 是当你运行 git clone 时默认的远程仓库名字。 如果你运行 git clone -o booyah，那么你默认的远程分支名字将会是 booyah/master。 |



Figure 30. 克隆之后的服务器与本地仓库

如果你在本地的 master 分支做了一些工作，在同一段时间内有其他人推送提交到 git.ourcompany.com 并且更新了它的 master 分支，这就是说你们的提交历史已走向不同的方向。 即便这样，只要你保持不与 origin 服务器连接（并拉取数据），你的 origin/master 指针就不会移动。



Figure 31. 本地与远程的工作可以分叉

如果要与给定的远程仓库同步数据，运行 git fetch <remote> 命令（在本例中为 git fetch origin）。 这个命令查找 “origin” 是哪一个服务器（在本例中，它是 git.ourcompany.com）， 从中抓取本地没有的数据，并且更新本地数据库，移动 origin/master 指针到更新之后的位置。



Figure 32. git fetch 更新你的远程跟踪分支

为了演示有多个远程仓库与远程分支的情况，我们假定你有另一个内部 Git 服务器，仅服务于你的某个敏捷开发团队。 这个服务器位于 git.team1.ourcompany.com。 你可以运行 git remote add 命令添加一个新的远程仓库引用到当前的项目，这个命令我们会在 [Git 基础](https://git-scm.com/book/zh/v2/ch00/ch02-git-basics-chapter) 中详细说明。 将这个远程仓库命名为 teamone，将其作为完整 URL 的缩写。



Figure 33. 添加另一个远程仓库

现在，可以运行 git fetch teamone 来抓取远程仓库 teamone 有而本地没有的数据。 因为那台服务器上现有的数据是 origin 服务器上的一个子集， 所以 Git 并不会抓取数据而是会设置远程跟踪分支 teamone/master 指向 teamone 的 master 分支。



Figure 34. 远程跟踪分支 teamone/master

### 推送

当你想要公开分享一个分支时，需要将其推送到有写入权限的远程仓库上。 本地的分支并不会自动与远程仓库同步——你必须显式地推送想要分享的分支。 这样，你就可以把不愿意分享的内容放到私人分支上，而将需要和别人协作的内容推送到公开分支。

如果希望和别人一起在名为 serverfix 的分支上工作，你可以像推送第一个分支那样推送它。 运行 git push <remote> <branch>:

$ git push origin serverfix

Counting objects: 24, done.

Delta compression using up to 8 threads.

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

Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.

Total 24 (delta 2), reused 0 (delta 0)

To https://github.com/schacon/simplegit

\* [new branch] serverfix -> serverfix

这里有些工作被简化了。 Git 自动将 serverfix 分支名字展开为 refs/heads/serverfix:refs/heads/serverfix， 那意味着，“推送本地的 serverfix 分支来更新远程仓库上的 serverfix 分支。” 我们将会详细学习 [Git 内部原理](https://git-scm.com/book/zh/v2/ch00/ch10-git-internals) 的 refs/heads/ 部分， 但是现在可以先把它放在儿。你也可以运行 git push origin serverfix:serverfix， 它会做同样的事——也就是说“推送本地的 serverfix 分支，将其作为远程仓库的 serverfix 分支” 可以通过这种格式来推送本地分支到一个命名不相同的远程分支。 如果并不想让远程仓库上的分支叫做 serverfix，可以运行 git push origin serverfix:awesomebranch 来将本地的 serverfix 分支推送到远程仓库上的 awesomebranch 分支。

|  |  |
| --- | --- |
| **Note** | **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](https://git-scm.com/book/zh/v2/ch00/_credential_caching)。 |

下一次其他协作者从服务器上抓取数据时，他们会在本地生成一个远程分支 origin/serverfix，指向服务器的 serverfix 分支的引用：

$ git fetch origin

remote: Counting objects: 7, done.

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

remote: Total 3 (delta 0), reused 3 (delta 0)

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

From https://github.com/schacon/simplegit

\* [new branch] serverfix -> origin/serverfix

要特别注意的一点是当抓取到新的远程跟踪分支时，本地不会自动生成一份可编辑的副本（拷贝）。 换一句话说，这种情况下，不会有一个新的 serverfix 分支——只有一个不可以修改的 origin/serverfix 指针。

可以运行 git merge origin/serverfix 将这些工作合并到当前所在的分支。 如果想要在自己的 serverfix 分支上工作，可以将其建立在远程跟踪分支之上：

$ git checkout -b serverfix origin/serverfix

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

Switched to a new branch 'serverfix'

这会给你一个用于工作的本地分支，并且起点位于 origin/serverfix。

### 跟踪分支

从一个远程跟踪分支检出一个本地分支会自动创建所谓的“跟踪分支”（它跟踪的分支叫做“上游分支”）。 跟踪分支是与远程分支有直接关系的本地分支。 如果在一个跟踪分支上输入 git pull，Git 能自动地识别去哪个服务器上抓取、合并到哪个分支。

当克隆一个仓库时，它通常会自动地创建一个跟踪 origin/master 的 master 分支。 然而，如果你愿意的话可以设置其他的跟踪分支，或是一个在其他远程仓库上的跟踪分支，又或者不跟踪 master 分支。 最简单的实例就是像之前看到的那样，运行 git checkout -b <branch> <remote>/<branch>。 这是一个十分常用的操作所以 Git 提供了 --track 快捷方式：

$ git checkout --track origin/serverfix

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

Switched to a new branch 'serverfix'

由于这个操作太常用了，该捷径本身还有一个捷径。 如果你尝试检出的分支 (a) 不存在且 (b) 刚好只有一个名字与之匹配的远程分支，那么 Git 就会为你创建一个跟踪分支：

$ git checkout serverfix

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

Switched to a new branch 'serverfix'

如果想要将本地分支与远程分支设置为不同的名字，你可以轻松地使用上一个命令增加一个不同名字的本地分支：

$ git checkout -b sf origin/serverfix

Branch sf set up to track remote branch serverfix from origin.

Switched to a new branch 'sf'

现在，本地分支 sf 会自动从 origin/serverfix 拉取。

设置已有的本地分支跟踪一个刚刚拉取下来的远程分支，或者想要修改正在跟踪的上游分支， 你可以在任意时间使用 -u 或 --set-upstream-to 选项运行 git branch 来显式地设置。

$ git branch -u origin/serverfix

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

|  |  |
| --- | --- |
| **Note** | **上游快捷方式**  当设置好跟踪分支后，可以通过简写 @{upstream} 或 @{u} 来引用它的上游分支。 所以在 master 分支时并且它正在跟踪 origin/master 时，如果愿意的话可以使用 git merge @{u} 来取代 git merge origin/master。 |

如果想要查看设置的所有跟踪分支，可以使用 git branch 的 -vv 选项。 这会将所有的本地分支列出来并且包含更多的信息，如每一个分支正在跟踪哪个远程分支与本地分支是否是领先、落后或是都有。

$ git branch -vv

iss53 7e424c3 [origin/iss53: ahead 2] forgot the brackets

master 1ae2a45 [origin/master] deploying index fix

\* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this should do it

testing 5ea463a trying something new

这里可以看到 iss53 分支正在跟踪 origin/iss53 并且 “ahead” 是 2，意味着本地有两个提交还没有推送到服务器上。 也能看到 master 分支正在跟踪 origin/master 分支并且是最新的。 接下来可以看到 serverfix 分支正在跟踪 teamone 服务器上的 server-fix-good 分支并且领先 3 落后 1， 意味着服务器上有一次提交还没有合并入同时本地有三次提交还没有推送。 最后看到 testing 分支并没有跟踪任何远程分支。

需要重点注意的一点是这些数字的值来自于你从每个服务器上最后一次抓取的数据。 这个命令并没有连接服务器，它只会告诉你关于本地缓存的服务器数据。 如果想要统计最新的领先与落后数字，需要在运行此命令前抓取所有的远程仓库。 可以像这样做：

$ git fetch --all; git branch -vv

### 拉取

当 git fetch 命令从服务器上抓取本地没有的数据时，它并不会修改工作目录中的内容。 它只会获取数据然后让你自己合并。 然而，有一个命令叫作 git pull 在大多数情况下它的含义是一个 git fetch 紧接着一个 git merge 命令。 如果有一个像之前章节中演示的设置好的跟踪分支，不管它是显式地设置还是通过 clone 或 checkout 命令为你创建的，git pull 都会查找当前分支所跟踪的服务器与分支， 从服务器上抓取数据然后尝试合并入那个远程分支。

由于 git pull 的魔法经常令人困惑所以通常单独显式地使用 fetch 与 merge 命令会更好一些。

### 删除远程分支

假设你已经通过远程分支做完所有的工作了——也就是说你和你的协作者已经完成了一个特性， 并且将其合并到了远程仓库的 master 分支（或任何其他稳定代码分支）。 可以运行带有 --delete 选项的 git push 命令来删除一个远程分支。 如果想要从服务器上删除 serverfix 分支，运行下面的命令：

$ git push origin --delete serverfix

To https://github.com/schacon/simplegit

- [deleted] serverfix

基本上这个命令做的只是从服务器上移除这个指针。 Git 服务器通常会保留数据一段时间直到垃圾回收运行，所以如果不小心删除掉了，通常是很容易恢复的。

## Git 分支 - 变基

### 变基

在 Git 中整合来自不同分支的修改主要有两种方法：merge 以及 rebase。 在本节中我们将学习什么是“变基”，怎样使用“变基”，并将展示该操作的惊艳之处，以及指出在何种情况下你应避免使用它。

### 变基的基本操作

请回顾之前在 [分支的合并](https://git-scm.com/book/zh/v2/ch00/_basic_merging) 中的一个例子，你会看到开发任务分叉到两个不同分支，又各自提交了更新。



Figure 35. 分叉的提交历史

之前介绍过，整合分支最容易的方法是 merge 命令。 它会把两个分支的最新快照（C3 和 C4）以及二者最近的共同祖先（C2）进行三方合并，合并的结果是生成一个新的快照（并提交）。



Figure 36. 通过合并操作来整合分叉的历史

其实，还有一种方法：你可以提取在 C4 中引入的补丁和修改，然后在 C3 的基础上应用一次。 在 Git 中，这种操作就叫做 **变基（rebase）**。 你可以使用 rebase 命令将提交到某一分支上的所有修改都移至另一分支上，就好像“重新播放”一样。

在这个例子中，你可以检出 experiment 分支，然后将它变基到 master 分支上：

$ git checkout experiment

$ git rebase master

First, rewinding head to replay your work on top of it...

Applying: added staged command

它的原理是首先找到这两个分支（即当前分支 experiment、变基操作的目标基底分支 master） 的最近共同祖先 C2，然后对比当前分支相对于该祖先的历次提交，提取相应的修改并存为临时文件， 然后将当前分支指向目标基底 C3, 最后以此将之前另存为临时文件的修改依序应用。 （译注：写明了 commit id，以便理解，下同）



Figure 37. 将 C4 中的修改变基到 C3 上

现在回到 master 分支，进行一次快进合并。

$ git checkout master

$ git merge experiment



Figure 38. master 分支的快进合并

此时，C4' 指向的快照就和 [the merge example](https://git-scm.com/book/zh/v2/ch00/ebasing-merging-example) 中 C5 指向的快照一模一样了。 这两种整合方法的最终结果没有任何区别，但是变基使得提交历史更加整洁。 你在查看一个经过变基的分支的历史记录时会发现，尽管实际的开发工作是并行的， 但它们看上去就像是串行的一样，提交历史是一条直线没有分叉。

一般我们这样做的目的是为了确保在向远程分支推送时能保持提交历史的整洁——例如向某个其他人维护的项目贡献代码时。 在这种情况下，你首先在自己的分支里进行开发，当开发完成时你需要先将你的代码变基到 origin/master 上，然后再向主项目提交修改。 这样的话，该项目的维护者就不再需要进行整合工作，只需要快进合并便可。

请注意，无论是通过变基，还是通过三方合并，整合的最终结果所指向的快照始终是一样的，只不过提交历史不同罢了。 变基是将一系列提交按照原有次序依次应用到另一分支上，而合并是把最终结果合在一起。

### 更有趣的变基例子

在对两个分支进行变基时，所生成的“重放”并不一定要在目标分支上应用，你也可以指定另外的一个分支进行应用。 就像 [从一个主题分支里再分出一个主题分支的提交历史](https://git-scm.com/book/zh/v2/ch00/bdiag_e) 中的例子那样。 你创建了一个主题分支 server，为服务端添加了一些功能，提交了 C3 和 C4。 然后从 C3 上创建了主题分支 client，为客户端添加了一些功能，提交了 C8 和 C9。 最后，你回到 server 分支，又提交了 C10。



Figure 39. 从一个主题分支里再分出一个主题分支的提交历史

假设你希望将 client 中的修改合并到主分支并发布，但暂时并不想合并 server 中的修改， 因为它们还需要经过更全面的测试。这时，你就可以使用 git rebase 命令的 --onto 选项， 选中在 client 分支里但不在 server 分支里的修改（即 C8 和 C9），将它们在 master 分支上重放：

$ git rebase --onto master server client

以上命令的意思是：“取出 client 分支，找出它从 server 分支分歧之后的补丁， 然后把这些补丁在 master 分支上重放一遍，让 client 看起来像直接基于 master 修改一样”。这理解起来有一点复杂，不过效果非常酷。



Figure 40. 截取主题分支上的另一个主题分支，然后变基到其他分支

现在可以快进合并 master 分支了。（如图 [快进合并 master 分支，使之包含来自 client 分支的修改](https://git-scm.com/book/zh/v2/ch00/bdiag_g)）：

$ git checkout master

$ git merge client



Figure 41. 快进合并 master 分支，使之包含来自 client 分支的修改

接下来你决定将 server 分支中的修改也整合进来。 使用 git rebase <basebranch> <topicbranch> 命令可以直接将主题分支 （即本例中的 server）变基到目标分支（即 master）上。 这样做能省去你先切换到 server 分支，再对其执行变基命令的多个步骤。

$ git rebase master server

如图 [将 server 中的修改变基到 master 上](https://git-scm.com/book/zh/v2/ch00/bdiag_h) 所示，server 中的代码被“续”到了 master 后面。



Figure 42. 将 server 中的修改变基到 master 上

然后就可以快进合并主分支 master 了：

$ git checkout master

$ git merge server

至此，client 和 server 分支中的修改都已经整合到主分支里了， 你可以删除这两个分支，最终提交历史会变成图 [最终的提交历史](https://git-scm.com/book/zh/v2/ch00/bdiag_i) 中的样子：

$ git branch -d client

$ git branch -d server



Figure 43. 最终的提交历史

### 变基的风险

呃，奇妙的变基也并非完美无缺，要用它得遵守一条准则：

**如果提交存在于你的仓库之外，而别人可能基于这些提交进行开发，那么不要执行变基。**

如果你遵循这条金科玉律，就不会出差错。 否则，人民群众会仇恨你，你的朋友和家人也会嘲笑你，唾弃你。

变基操作的实质是丢弃一些现有的提交，然后相应地新建一些内容一样但实际上不同的提交。 如果你已经将提交推送至某个仓库，而其他人也已经从该仓库拉取提交并进行了后续工作，此时，如果你用 git rebase 命令重新整理了提交并再次推送，你的同伴因此将不得不再次将他们手头的工作与你的提交进行整合，如果接下来你还要拉取并整合他们修改过的提交，事情就会变得一团糟。

让我们来看一个在公开的仓库上执行变基操作所带来的问题。 假设你从一个中央服务器克隆然后在它的基础上进行了一些开发。 你的提交历史如图所示：



Figure 44. 克隆一个仓库，然后在它的基础上进行了一些开发

然后，某人又向中央服务器提交了一些修改，其中还包括一次合并。 你抓取了这些在远程分支上的修改，并将其合并到你本地的开发分支，然后你的提交历史就会变成这样：



Figure 45. 抓取别人的提交，合并到自己的开发分支

接下来，这个人又决定把合并操作回滚，改用变基；继而又用 git push --force 命令覆盖了服务器上的提交历史。 之后你从服务器抓取更新，会发现多出来一些新的提交。



Figure 46. 有人推送了经过变基的提交，并丢弃了你的本地开发所基于的一些提交

结果就是你们两人的处境都十分尴尬。 如果你执行 git pull 命令，你将合并来自两条提交历史的内容，生成一个新的合并提交，最终仓库会如图所示：



Figure 47. 你将相同的内容又合并了一次，生成了一个新的提交

此时如果你执行 git log 命令，你会发现有两个提交的作者、日期、日志居然是一样的，这会令人感到混乱。 此外，如果你将这一堆又推送到服务器上，你实际上是将那些已经被变基抛弃的提交又找了回来，这会令人感到更加混乱。 很明显对方并不想在提交历史中看到 C4 和 C6，因为之前就是他把这两个提交通过变基丢弃的。

### 用变基解决变基

如果你 **真的** 遭遇了类似的处境，Git 还有一些高级魔法可以帮到你。 如果团队中的某人强制推送并覆盖了一些你所基于的提交，你需要做的就是检查你做了哪些修改，以及他们覆盖了哪些修改。

实际上，Git 除了对整个提交计算 SHA-1 校验和以外，也对本次提交所引入的修改计算了校验和——即 “patch-id”。

如果你拉取被覆盖过的更新并将你手头的工作基于此进行变基的话，一般情况下 Git 都能成功分辨出哪些是你的修改，并把它们应用到新分支上。

举个例子，如果遇到前面提到的 [有人推送了经过变基的提交，并丢弃了你的本地开发所基于的一些提交](https://git-scm.com/book/zh/v2/ch00/_pre_merge_rebase_work) 那种情境，如果我们不是执行合并，而是执行 git rebase teamone/master, Git 将会：

* 检查哪些提交是我们的分支上独有的（C2，C3，C4，C6，C7）
* 检查其中哪些提交不是合并操作的结果（C2，C3，C4）
* 检查哪些提交在对方覆盖更新时并没有被纳入目标分支（只有 C2 和 C3，因为 C4 其实就是 C4'）
* 把查到的这些提交应用在 teamone/master 上面

从而我们将得到与 [你将相同的内容又合并了一次，生成了一个新的提交](https://git-scm.com/book/zh/v2/ch00/_merge_rebase_work) 中不同的结果，如图 [在一个被变基然后强制推送的分支上再次执行变基](https://git-scm.com/book/zh/v2/ch00/_rebase_rebase_work) 所示。



Figure 48. 在一个被变基然后强制推送的分支上再次执行变基

要想上述方案有效，还需要对方在变基时确保 C4' 和 C4 是几乎一样的。 否则变基操作将无法识别，并新建另一个类似 C4 的补丁（而这个补丁很可能无法整洁的整合入历史，因为补丁中的修改已经存在于某个地方了）。

在本例中另一种简单的方法是使用 git pull --rebase 命令而不是直接 git pull。 又或者你可以自己手动完成这个过程，先 git fetch，再 git rebase teamone/master。

如果你习惯使用 git pull ，同时又希望默认使用选项 --rebase，你可以执行这条语句 git config --global pull.rebase true 来更改 pull.rebase 的默认配置。

如果你只对不会离开你电脑的提交执行变基，那就不会有事。 如果你对已经推送过的提交执行变基，但别人没有基于它的提交，那么也不会有事。 如果你对已经推送至共用仓库的提交上执行变基命令，并因此丢失了一些别人的开发所基于的提交， 那你就有大麻烦了，你的同事也会因此鄙视你。

如果你或你的同事在某些情形下决意要这么做，请一定要通知每个人执行 git pull --rebase 命令，这样尽管不能避免伤痛，但能有所缓解。

### 变基 vs. 合并

至此，你已在实战中学习了变基和合并的用法，你一定会想问，到底哪种方式更好。 在回答这个问题之前，让我们退后一步，想讨论一下提交历史到底意味着什么。

有一种观点认为，仓库的提交历史即是 **记录实际发生过什么**。 它是针对历史的文档，本身就有价值，不能乱改。 从这个角度看来，改变提交历史是一种亵渎，你使用 **谎言** 掩盖了实际发生过的事情。 如果由合并产生的提交历史是一团糟怎么办？ 既然事实就是如此，那么这些痕迹就应该被保留下来，让后人能够查阅。

另一种观点则正好相反，他们认为提交历史是 **项目过程中发生的事**。 没人会出版一本书的第一版草稿，软件维护手册也是需要反复修订才能方便使用。 持这一观点的人会使用 rebase 及 filter-branch 等工具来编写故事，怎么方便后来的读者就怎么写。

现在，让我们回到之前的问题上来，到底合并还是变基好？希望你能明白，这并没有一个简单的答案。 Git 是一个非常强大的工具，它允许你对提交历史做许多事情，但每个团队、每个项目对此的需求并不相同。 既然你已经分别学习了两者的用法，相信你能够根据实际情况作出明智的选择。

总的原则是，只对尚未推送或分享给别人的本地修改执行变基操作清理历史， 从不对已推送至别处的提交执行变基操作，这样，你才能享受到两种方式带来的便利。

# 服务器上的 Git

## 服务器上的 Git - 协议

到目前为止，你应该已经有办法使用 Git 来完成日常工作。 然而，为了使用 Git 协作功能，你还需要有远程的 Git 仓库。 尽管在技术上你可以从个人仓库进行推送（push）和拉取（pull）来修改内容，但不鼓励使用这种方法，因为一不留心就很容易弄混其他人的进度。 此外，你希望你的合作者们即使在你的电脑未联机时亦能存取仓库 — 拥有一个更可靠的公用仓库十分有用。 因此，与他人合作的最佳方法即是建立一个你与合作者们都有权利访问，且可从那里推送和拉取资料的共用仓库。

架设一台 Git 服务器并不难。 首先，选择你希望服务器使用的通讯协议。 在本章第一节将介绍可用的协议以及各自优缺点。 下面一节将解释使用那些协议的典型设置及如何在你的服务器上运行。 最后，如果你不介意托管你的代码在其他人的服务器，且不想经历设置与维护自己服务器的麻烦，可以试试我们介绍的几个仓库托管服务。

如果你对架设自己的服务器没兴趣，可以跳到本章最后一节去看看如何申请一个代码托管服务的帐户然后继续下一章，我们会在那里讨论分布式源码控制环境的林林总总。

一个远程仓库通常只是一个裸仓库（bare repository）——即一个没有当前工作目录的仓库。 因为该仓库仅仅作为合作媒介，不需要从磁盘检查快照；存放的只有 Git 的资料。 简单的说，裸仓库就是你工程目录内的 .git 子目录内容，不包含其他资料。

### 协议

Git 可以使用四种不同的协议来传输资料：本地协议（Local），HTTP 协议，SSH（Secure Shell）协议及 Git 协议。 在此，我们将会讨论那些协议及哪些情形应该使用（或避免使用）他们。

### 本地协议

最基本的就是 **本地协议（Local protocol）** ，其中的远程版本库就是同一主机上的另一个目录。 这常见于团队每一个成员都对一个共享的文件系统（例如一个挂载的 NFS）拥有访问权，或者比较少见的多人共用同一台电脑的情况。 后者并不理想，因为你的所有代码版本库如果长存于同一台电脑，更可能发生灾难性的损失。

如果你使用共享文件系统，就可以从本地版本库克隆（clone）、推送（push）以及拉取（pull）。 像这样去克隆一个版本库或者增加一个远程到现有的项目中，使用版本库路径作为 URL。 例如，克隆一个本地版本库，可以执行如下的命令：

$ git clone /srv/git/project.git

或你可以执行这个命令：

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

如果在 URL 开头明确的指定 file://，那么 Git 的行为会略有不同。 如果仅是指定路径，Git 会尝试使用硬链接（hard link）或直接复制所需要的文件。 如果指定 file://，Git 会触发平时用于网路传输资料的进程，那样传输效率会更低。 指定 file:// 的主要目的是取得一个没有外部参考（extraneous references） 或对象（object）的干净版本库副本——通常是在从其他版本控制系统导入后或一些类似情况需要这么做 （关于维护任务可参见 [Git 内部原理](https://git-scm.com/book/zh/v2/ch00/ch10-git-internals) ）。 在此我们将使用普通路径，因为这样通常更快。

要增加一个本地版本库到现有的 Git 项目，可以执行如下的命令：

$ git remote add local\_proj /srv/git/project.git

然后，就可以通过新的远程仓库名 local\_proj 像在网络上一样从远端版本库推送和拉取更新了。

#### 优点

基于文件系统的版本库的优点是简单，并且直接使用了现有的文件权限和网络访问权限。 如果你的团队已经有共享文件系统，建立版本库会十分容易。 只需要像设置其他共享目录一样，把一个裸版本库的副本放到大家都可以访问的路径，并设置好读/写的权限，就可以了， 我们会在 [在服务器上搭建 Git](https://git-scm.com/book/zh/v2/ch00/_getting_git_on_a_server) 讨论如何导出一个裸版本库。

这也是快速从别人的工作目录中拉取更新的方法。 如果你和别人一起合作一个项目，他想让你从版本库中拉取更新时，运行类似 git pull /home/john/project 的命令比推送到服务器再抓取回来简单多了。

#### 缺点

这种方法的缺点是，通常共享文件系统比较难配置，并且比起基本的网络连接访问，这不方便从多个位置访问。 如果你想从家里推送内容，必须先挂载一个远程磁盘，相比网络连接的访问方式，配置不方便，速度也慢。

值得一提的是，如果你使用的是类似于共享挂载的文件系统时，这个方法不一定是最快的。 访问本地版本库的速度与你访问数据的速度是一样的。 在同一个服务器上，如果允许 Git 访问本地硬盘，一般的通过 NFS 访问版本库要比通过 SSH 访问慢。

最终，这个协议并不保护仓库避免意外的损坏。 每一个用户都有“远程”目录的完整 shell 权限，没有方法可以阻止他们修改或删除 Git 内部文件和损坏仓库。

### HTTP 协议

Git 通过 HTTP 通信有两种模式。 在 Git 1.6.6 版本之前只有一个方式可用，十分简单并且通常是只读模式的。 Git 1.6.6 版本引入了一种新的、更智能的协议，让 Git 可以像通过 SSH 那样智能的协商和传输数据。 之后几年，这个新的 HTTP 协议因为其简单、智能变的十分流行。 新版本的 HTTP 协议一般被称为 **智能** HTTP 协议，旧版本的一般被称为 **哑** HTTP 协议。 我们先了解一下新的智能 HTTP 协议。

#### 智能 HTTP 协议

智能 HTTP 的运行方式和 SSH 及 Git 协议类似，只是运行在标准的 HTTP/S 端口上并且可以使用各种 HTTP 验证机制， 这意味着使用起来会比 SSH 协议简单的多，比如可以使用 HTTP 协议的用户名/密码授权，免去设置 SSH 公钥。

智能 HTTP 协议或许已经是最流行的使用 Git 的方式了，它即支持像 git:// 协议一样设置匿名服务， 也可以像 SSH 协议一样提供传输时的授权和加密。 而且只用一个 URL 就可以都做到，省去了为不同的需求设置不同的 URL。 如果你要推送到一个需要授权的服务器上（一般来讲都需要），服务器会提示你输入用户名和密码。 从服务器获取数据时也一样。

事实上，类似 GitHub 的服务，你在网页上看到的 URL（比如 <https://github.com/schacon/simplegit>）， 和你在克隆、推送（如果你有权限）时使用的是一样的。

#### 哑（Dumb） HTTP 协议

如果服务器没有提供智能 HTTP 协议的服务，Git 客户端会尝试使用更简单的“哑” HTTP 协议。 哑 HTTP 协议里 web 服务器仅把裸版本库当作普通文件来对待，提供文件服务。 哑 HTTP 协议的优美之处在于设置起来简单。 基本上，只需要把一个裸版本库放在 HTTP 根目录，设置一个叫做 post-update 的挂钩就可以了 （见 [Git 钩子](https://git-scm.com/book/zh/v2/ch00/_git_hooks)）。 此时，只要能访问 web 服务器上你的版本库，就可以克隆你的版本库。 下面是设置从 HTTP 访问版本库的方法：

$ cd /var/www/htdocs/

$ git clone --bare /path/to/git\_project gitproject.git

$ cd gitproject.git

$ mv hooks/post-update.sample hooks/post-update

$ chmod a+x hooks/post-update

这样就可以了。 Git 自带的 post-update 挂钩会默认执行合适的命令（git update-server-info），来确保通过 HTTP 的获取和克隆操作正常工作。 这条命令会在你通过 SSH 向版本库推送之后被执行；然后别人就可以通过类似下面的命令来克隆：

$ git clone https://example.com/gitproject.git

这里我们用了 Apache 里设置了常用的路径 /var/www/htdocs，不过你可以使用任何静态 Web 服务器 —— 只需要把裸版本库放到正确的目录下就可以。 Git 的数据是以基本的静态文件形式提供的（详情见 [Git 内部原理](https://git-scm.com/book/zh/v2/ch00/ch10-git-internals)）。

通常的，会在可以提供读／写的智能 HTTP 服务和简单的只读的哑 HTTP 服务之间选一个。 极少会将二者混合提供服务。

#### 优点

我们将只关注智能 HTTP 协议的优点。

不同的访问方式只需要一个 URL 以及服务器只在需要授权时提示输入授权信息，这两个简便性让终端用户使用 Git 变得非常简单。 相比 SSH 协议，可以使用用户名／密码授权是一个很大的优势，这样用户就不必须在使用 Git 之前先在本地生成 SSH 密钥对再把公钥上传到服务器。 对非资深的使用者，或者系统上缺少 SSH 相关程序的使用者，HTTP 协议的可用性是主要的优势。 与 SSH 协议类似，HTTP 协议也非常快和高效。

你也可以在 HTTPS 协议上提供只读版本库的服务，如此你在传输数据的时候就可以加密数据；或者，你甚至可以让客户端使用指定的 SSL 证书。

另一个好处是 HTTPS 协议被广泛使用，一般的企业防火墙都会允许这些端口的数据通过。

#### 缺点

在一些服务器上，架设 HTTPS 协议的服务端会比 SSH 协议的棘手一些。 除了这一点，用其他协议提供 Git 服务与智能 HTTP 协议相比就几乎没有优势了。

如果你在 HTTP 上使用需授权的推送，管理凭证会比使用 SSH 密钥认证麻烦一些。 然而，你可以选择使用凭证存储工具，比如 macOS 的 Keychain 或者 Windows 的凭证管理器。 参考 [凭证存储](https://git-scm.com/book/zh/v2/ch00/_credential_caching) 如何安全地保存 HTTP 密码。

### SSH 协议

架设 Git 服务器时常用 SSH 协议作为传输协议。 因为大多数环境下服务器已经支持通过 SSH 访问 —— 即使没有也很容易架设。 SSH 协议也是一个验证授权的网络协议；并且，因为其普遍性，架设和使用都很容易。

通过 SSH 协议克隆版本库，你可以指定一个 ssh:// 的 URL：

$ git clone ssh://[user@]server/project.git

或者使用一个简短的 scp 式的写法：

$ git clone [user@]server:project.git

在上面两种情况中，如果你不指定可选的用户名，那么 Git 会使用当前登录的用的名字。

#### 优势

用 SSH 协议的优势有很多。 首先，SSH 架设相对简单 —— SSH 守护进程很常见，多数管理员都有使用经验，并且多数操作系统都包含了它及相关的管理工具。 其次，通过 SSH 访问是安全的 —— 所有传输数据都要经过授权和加密。 最后，与 HTTPS 协议、Git 协议及本地协议一样，SSH 协议很高效，在传输前也会尽量压缩数据。

#### 缺点

SSH 协议的缺点在于它不支持匿名访问 Git 仓库。 如果你使用 SSH，那么即便只是读取数据，使用者也 **必须** 通过 SSH 访问你的主机， 这使得 SSH 协议不利于开源的项目，毕竟人们可能只想把你的仓库克隆下来查看。 如果你只在公司网络使用，SSH 协议可能是你唯一要用到的协议。 如果你要同时提供匿名只读访问和 SSH 协议，那么你除了为自己推送架设 SSH 服务以外， 还得架设一个可以让其他人访问的服务。

### Git 协议

最后是 Git 协议。 这是包含在 Git 里的一个特殊的守护进程；它监听在一个特定的端口（9418），类似于 SSH 服务，但是访问无需任何授权。 要让版本库支持 Git 协议，需要先创建一个 git-daemon-export-ok 文件 —— 它是 Git 协议守护进程为这个版本库提供服务的必要条件 —— 但是除此之外没有任何安全措施。 要么谁都可以克隆这个版本库，要么谁也不能。 这意味着，通常不能通过 Git 协议推送。 由于没有授权机制，一旦你开放推送操作，意味着网络上知道这个项目 URL 的人都可以向项目推送数据。 不用说，极少会有人这么做。

#### 优点

目前，Git 协议是 Git 使用的网络传输协议里最快的。 如果你的项目有很大的访问量，或者你的项目很庞大并且不需要为写进行用户授权，架设 Git 守护进程来提供服务是不错的选择。 它使用与 SSH 相同的数据传输机制，但是省去了加密和授权的开销。

#### 缺点

Git 协议缺点是缺乏授权机制。 把 Git 协议作为访问项目版本库的唯一手段是不可取的。 一般的做法里，会同时提供 SSH 或者 HTTPS 协议的访问服务，只让少数几个开发者有推送（写）权限，其他人通过 git:// 访问只有读权限。 Git 协议也许也是最难架设的。 它要求有自己的守护进程，这就要配置 xinetd、systemd 或者其他的程序，这些工作并不简单。 它还要求防火墙开放 9418 端口，但是企业防火墙一般不会开放这个非标准端口。 而大型的企业防火墙通常会封锁这个端口。

## 服务器上的 Git - 在服务器上搭建 Git

### 在服务器上搭建 Git

现在我们将讨论如何在你自己的服务器上搭建 Git 服务来运行这些协议。

|  |  |
| --- | --- |
| **Note** | 这里我们将要演示在 Linux 服务器上进行一次基本且简化的安装所需的命令与步骤，当然在 macOS 或 Windows 服务器上同样可以运行这些服务。 事实上，在你的计算机基础架构中建立一个生产环境服务器，将不可避免的使用到不同的安全措施与操作系统工具。但是，希望你能从本节中获得一些必要的知识。 |

在开始架设 Git 服务器前，需要把现有仓库导出为裸仓库——即一个不包含当前工作目录的仓库。 这通常是很简单的。 为了通过克隆你的仓库来创建一个新的裸仓库，你需要在克隆命令后加上 --bare 选项。 按照惯例，裸仓库的目录名以 .git 结尾，就像这样：

$ git clone --bare my\_project my\_project.git

Cloning into bare repository 'my\_project.git'...

done.

现在，你的 my\_project.git 目录中应该有 Git 目录的副本了。

整体上效果大致相当于

$ cp -Rf my\_project/.git my\_project.git

虽然在配置文件中有若干不同，但是对于你的目的来说，这两种方式都是一样的。 它只取出 Git 仓库自身，不要工作目录，然后特别为它单独创建一个目录。

### 把裸仓库放到服务器上

既然你有了裸仓库的副本，剩下要做的就是把裸仓库放到服务器上并设置你的协议。 假设一个域名为 git.example.com 的服务器已经架设好，并可以通过 SSH 连接， 你想把所有的 Git 仓库放在 /srv/git 目录下。 假设服务器上存在 /srv/git/ 目录，你可以通过以下命令复制你的裸仓库来创建一个新仓库：

$ scp -r my\_project.git user@git.example.com:/srv/git

此时，其他可通过 SSH 读取此服务器上 /srv/git 目录的用户，可运行以下命令来克隆你的仓库。

$ git clone user@git.example.com:/srv/git/my\_project.git

如果一个用户，通过使用 SSH 连接到一个服务器，并且其对 /srv/git/my\_project.git 目录拥有可写权限，那么他将自动拥有推送权限。

如果到该项目目录中运行 git init 命令，并加上 --shared 选项， 那么 Git 会自动修改该仓库目录的组权限为可写。 注意，运行此命令的工程中不会摧毁任何提交、引用等内容。

$ ssh user@git.example.com

$ cd /srv/git/my\_project.git

$ git init --bare --shared

由此可见，根据现有的 Git 仓库创建一个裸仓库，然后把它放上你和协作者都有 SSH 访问权的服务器是多么容易。 现在你们已经准备好在同一项目上展开合作了。

值得注意的是，这的确是架设一个几个人拥有连接权的 Git 服务的全部—— 只要在服务器上加入可以用 SSH 登录的帐号，然后把裸仓库放在大家都有读写权限的地方。 你已经准备好了一切，无需更多。

下面的几节中，你会了解如何扩展到更复杂的设定。 这些内容包含如何避免为每一个用户建立一个账户，给仓库添加公共读取权限，架设网页界面等等。 然而，请记住这一点，如果只是和几个人在一个私有项目上合作的话，**仅仅** 是一个 SSH 服务器和裸仓库就足够了。

### 小型安装

如果设备较少或者你只想在小型开发团队里尝试 Git ，那么一切都很简单。 架设 Git 服务最复杂的地方在于用户管理。 如果需要仓库对特定的用户可读，而给另一部分用户读写权限，那么访问和许可安排就会比较困难。

#### SSH 连接

如果你有一台所有开发者都可以用 SSH 连接的服务器，架设你的第一个仓库就十分简单了， 因为你几乎什么都不用做（正如我们上一节所说的）。 如果你想在你的仓库上设置更复杂的访问控制权限，只要使用服务器操作系统的普通的文件系统权限就行了。

如果需要团队里的每个人都对仓库有写权限，又不能给每个人在服务器上建立账户，那么提供 SSH 连接就是唯一的选择了。 我们假设用来共享仓库的服务器已经安装了 SSH 服务，而且你通过它访问服务器。

有几个方法可以使你给团队每个成员提供访问权。 第一个就是给团队里的每个人创建账号，这种方法很直接但也很麻烦。 或许你不会想要为每个人运行一次 adduser（或者 useradd）并且设置临时密码。

第二个办法是在主机上建立一个 **git** 账户，让每个需要写权限的人发送一个 SSH 公钥， 然后将其加入 git 账户的 ~/.ssh/authorized\_keys 文件。 这样一来，所有人都将通过 **git** 账户访问主机。 这一点也不会影响提交的数据——访问主机用的身份不会影响提交对象的提交者信息。

另一个办法是让 SSH 服务器通过某个 LDAP 服务，或者其他已经设定好的集中授权机制，来进行授权。 只要每个用户可以获得主机的 shell 访问权限，任何 SSH 授权机制你都可视为是有效的。

## 服务器上的 Git - 生成 SSH 公钥

### 生成 SSH 公钥

如前所述，许多 Git 服务器都使用 SSH 公钥进行认证。 为了向 Git 服务器提供 SSH 公钥，如果某系统用户尚未拥有密钥，必须事先为其生成一份。 这个过程在所有操作系统上都是相似的。 首先，你需要确认自己是否已经拥有密钥。 默认情况下，用户的 SSH 密钥存储在其 ~/.ssh 目录下。 进入该目录并列出其中内容，你便可以快速确认自己是否已拥有密钥：

$ cd ~/.ssh

$ ls

authorized\_keys2 id\_dsa known\_hosts

config id\_dsa.pub

我们需要寻找一对以 id\_dsa 或 id\_rsa 命名的文件，其中一个带有 .pub 扩展名。 .pub 文件是你的公钥，另一个则是与之对应的私钥。 如果找不到这样的文件（或者根本没有 .ssh 目录），你可以通过运行 ssh-keygen 程序来创建它们。 在 Linux/macOS 系统中，ssh-keygen 随 SSH 软件包提供；在 Windows 上，该程序包含于 MSysGit 软件包中。

$ ssh-keygen -o

Generating public/private rsa key pair.

Enter file in which to save the key (/home/schacon/.ssh/id\_rsa):

Created directory '/home/schacon/.ssh'.

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

Your identification has been saved in /home/schacon/.ssh/id\_rsa.

Your public key has been saved in /home/schacon/.ssh/id\_rsa.pub.

The key fingerprint is:

d0:82:24:8e:d7:f1:bb:9b:33:53:96:93:49:da:9b:e3 schacon@mylaptop.local

首先 ssh-keygen 会确认密钥的存储位置（默认是 .ssh/id\_rsa），然后它会要求你输入两次密钥口令。 如果你不想在使用密钥时输入口令，将其留空即可。 然而，如果你使用了密码，那么请确保添加了 -o 选项，它会以比默认格式更能抗暴力破解的格式保存私钥。 你也可以用 ssh-agent 工具来避免每次都要输入密码。

现在，进行了上述操作的用户需要将各自的公钥发送给任意一个 Git 服务器管理员 （假设服务器正在使用基于公钥的 SSH 验证设置）。 他们所要做的就是复制各自的 .pub 文件内容，并将其通过邮件发送。 公钥看起来是这样的：

$ cat ~/.ssh/id\_rsa.pub

ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU

GPl+nafzlHDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h9lFX5QVkbPppSwg0cda3

Pbv7kOdJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XA

t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/En

mZ+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx

NrRFi9wrf+M7Q== schacon@mylaptop.local

关于在多种操作系统中生成 SSH 密钥的更深入教程，请参阅 GitHub 的 SSH 密钥指南 <https://docs.github.com/cn/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent>。

## 服务器上的 Git - 配置服务器

### 配置服务器

我们来看看如何配置服务器端的 SSH 访问。 本例中，我们将使用 authorized\_keys 方法来对用户进行认证。 同时我们假设你使用的操作系统是标准的 Linux 发行版，比如 Ubuntu。 首先，创建一个操作系统用户 git，并为其建立一个 .ssh 目录。

|  |  |
| --- | --- |
| **Note** | 以下操作可通过 ssh-copy-id 命令自动完成，这样就不必手动复制并安装公钥了。 |

首先，创建一个操作系统用户 git，并为其建立一个 .ssh 目录。

$ sudo adduser git

$ su git

$ cd

$ mkdir .ssh && chmod 700 .ssh

$ touch .ssh/authorized\_keys && chmod 600 .ssh/authorized\_keys

接着，我们需要为系统用户 git 的 authorized\_keys 文件添加一些开发者 SSH 公钥。 假设我们已经获得了若干受信任的公钥，并将它们保存在临时文件中。 与前文类似，这些公钥看起来是这样的：

$ cat /tmp/id\_rsa.john.pub

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4L

ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k

Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9Ez

Sdfd8AcCIicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myiv

O7TCUSBdLQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq

dAv8JggJICUvax2T9va5 gsg-keypair

将这些公钥加入系统用户 git 的 .ssh 目录下 authorized\_keys 文件的末尾：

$ cat /tmp/id\_rsa.john.pub >> ~/.ssh/authorized\_keys

$ cat /tmp/id\_rsa.josie.pub >> ~/.ssh/authorized\_keys

$ cat /tmp/id\_rsa.jessica.pub >> ~/.ssh/authorized\_keys

现在我们来为开发者新建一个空仓库。可以借助带 --bare 选项的 git init 命令来做到这一点，该命令在初始化仓库时不会创建工作目录：

$ cd /srv/git

$ mkdir project.git

$ cd project.git

$ git init --bare

Initialized empty Git repository in /srv/git/project.git/

接着，John、Josie 或者 Jessica 中的任意一人可以将他们项目的最初版本推送到这个仓库中， 他只需将此仓库设置为项目的远程仓库并向其推送分支。 请注意，每添加一个新项目，都需要有人登录服务器取得 shell，并创建一个裸仓库。 我们假定这个设置了 git 用户和 Git 仓库的服务器使用 gitserver 作为主机名。 同时，假设该服务器运行在内网，并且你已在 DNS 配置中将 gitserver 指向此服务器。 那么我们可以运行如下命令（假定 myproject 是已有项目且其中已包含文件）：

# on John's computer

$ cd myproject

$ git init

$ git add .

$ git commit -m 'initial commit'

$ git remote add origin git@gitserver:/srv/git/project.git

$ git push origin master

此时，其他开发者可以克隆此仓库，并推回各自的改动，步骤很简单：

$ git clone git@gitserver:/srv/git/project.git

$ cd project

$ vim README

$ git commit -am 'fix for the README file'

$ git push origin master

通过这种方法，你可以快速搭建一个具有读写权限、面向多个开发者的 Git 服务器。

需要注意的是，目前所有（获得授权的）开发者用户都能以系统用户 git 的身份登录服务器从而获得一个普通 shell。 如果你想对此加以限制，则需要修改 /etc/passwd 文件中（git 用户所对应）的 shell 值。

借助一个名为 git-shell 的受限 shell 工具，你可以方便地将用户 git 的活动限制在与 Git 相关的范围内。 该工具随 Git 软件包一同提供。如果将 git-shell 设置为用户 git 的登录 shell（login shell）， 那么该用户便不能获得此服务器的普通 shell 访问权限。 若要使用 git-shell，需要用它替换掉 bash 或 csh，使其成为该用户的登录 shell。 为进行上述操作，首先你必须确保 git-shell 的完整路径名已存在于 /etc/shells 文件中：

$ cat /etc/shells # see if git-shell is already in there. If not...

$ which git-shell # make sure git-shell is installed on your system.

$ sudo -e /etc/shells # and add the path to git-shell from last command

现在你可以使用 chsh <username> -s <shell> 命令修改任一系统用户的 shell：

$ sudo chsh git -s $(which git-shell)

这样，用户 git 就只能利用 SSH 连接对 Git 仓库进行推送和拉取操作，而不能登录机器并取得普通 shell。 如果试图登录，你会发现尝试被拒绝，像这样：

$ ssh git@gitserver

fatal: Interactive git shell is not enabled.

hint: ~/git-shell-commands should exist and have read and execute access.

Connection to gitserver closed.

此时，用户仍可通过 SSH 端口转发来访问任何可达的 git 服务器。 如果你想要避免它，可编辑 authorized\_keys 文件并在所有想要限制的公钥之前添加以下选项：

no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty

其结果如下：

$ cat ~/.ssh/authorized\_keys

no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty ssh-rsa

AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4LojG6rs6h

PB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4kYjh6541N

YsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9EzSdfd8AcC

IicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myivO7TCUSBd

LQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPqdAv8JggJ

ICUvax2T9va5 gsg-keypair

no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty ssh-rsa

AAAAB3NzaC1yc2EAAAADAQABAAABAQDEwENNMomTboYI+LJieaAY16qiXiH3wuvENhBG...

现在，网络相关的 Git 命令依然能够正常工作，但是开发者用户已经无法得到一个普通 shell 了。 正如输出信息所提示的，你也可以在 git 用户的主目录下建立一个目录，来对 git-shell 命令进行一定程度的自定义。 比如，你可以限制掉某些本应被服务器接受的 Git 命令，或者对刚才的 SSH 拒绝登录信息进行自定义，这样，当有开发者用户以类似方式尝试登录时，便会看到你的信息。 要了解更多有关自定义 shell 的信息，请运行 git help shell。

## 服务器上的 Git - Git 守护进程

### Git 守护进程

接下来我们将通过 “Git” 协议建立一个基于守护进程的仓库。 对于快速且无需授权的 Git 数据访问，这是一个理想之选。 请注意，因为其不包含授权服务，任何通过该协议管理的内容将在其网络上公开。

如果运行在防火墙之外的服务器上，它应该只对那些公开的只读项目服务。 如果运行在防火墙之内的服务器上，它可用于支撑大量参与人员或自动系统 （用于持续集成或编译的主机）只读访问的项目，这样可以省去逐一配置 SSH 公钥的麻烦。

无论何时，该 Git 协议都是相对容易设定的。 通常，你只需要以守护进程的形式运行该命令：

$ git daemon --reuseaddr --base-path=/srv/git/ /srv/git/

--reuseaddr 选项允许服务器在无需等待旧连接超时的情况下重启，而 --base-path 选项允许用户在未完全指定路径的条件下克隆项目， 结尾的路径将告诉 Git 守护进程从何处寻找仓库来导出。 如果有防火墙正在运行，你需要开放端口 9418 的通信权限。

你可以通过许多方式将该进程以守护进程的方式运行，这主要取决于你所使用的操作系统。

由于在现代的 Linux 发行版中，systemd 是最常见的初始化系统，因此你可以用它来达到此目的。 只要在 /etc/systemd/system/git-daemon.service 中放一个文件即可，其内容如下：

[Unit]

Description=Start Git Daemon

[Service]

ExecStart=/usr/bin/git daemon --reuseaddr --base-path=/srv/git/ /srv/git/

Restart=always

RestartSec=500ms

StandardOutput=syslog

StandardError=syslog

SyslogIdentifier=git-daemon

User=git

Group=git

[Install]

WantedBy=multi-user.target

你可能会注意这里以 git 启动的 Git 驻留程序同时使用了 Group 和 User 权限。 按需修改它并确保提供的用户在此系统上。此外，请确保 Git 二进制文件位于 /usr/bin/git，必要时可修改此路径。

最后，你需要运行 systemctl enable git-daemon 以让它在系统启动时自动运行， 这样也能让它通过 systemctl start git-daemon 启动，通过 systemctl stop git-daemon 停止。

在其他系统中，你可以使用 sysvinit 系统中的 xinetd 脚本，或者另外的方式来实现——只要你能够将其命令守护进程化并实现监控。

接下来，你需要告诉 Git 哪些仓库允许基于服务器的无授权访问。 你可以在每个仓库下创建一个名为 git-daemon-export-ok 的文件来实现。

$ cd /path/to/project.git

$ touch git-daemon-export-ok

该文件将允许 Git 提供无需授权的项目访问服务。

## 服务器上的 Git - Smart HTTP

### Smart HTTP

我们一般通过 SSH 进行授权访问，通过 git:// 进行无授权访问，但是还有一种协议可以同时实现以上两种方式的访问。 设置 Smart HTTP 一般只需要在服务器上启用一个 Git 自带的名为 git-http-backend 的 CGI 脚本。 该 CGI 脚本将会读取由 git fetch 或 git push 命令向 HTTP URL 发送的请求路径和头部信息， 来判断该客户端是否支持 HTTP 通信（不低于 1.6.6 版本的客户端支持此特性）。 如果 CGI 发现该客户端支持智能（Smart）模式，它将会以智能模式与它进行通信， 否则它将会回落到哑（Dumb）模式下（因此它可以对某些老的客户端实现向下兼容）。

在完成以上简单的安装步骤后， 我们将用 Apache 来作为 CGI 服务器。 如果你没有安装 Apache，你可以在 Linux 环境下执行如下或类似的命令来安装：

$ sudo apt-get install apache2 apache2-utils

$ a2enmod cgi alias env

该操作将会启用 mod\_cgi， mod\_alias 和 mod\_env 等 Apache 模块， 这些模块都是使该功能正常工作所必须的。

你还需要将 /srv/git 的 Unix 用户组设置为 www-data，这样 Web 服务器才能读写该仓库， 因为运行 CGI 脚本的 Apache 实例默认会以该用户的权限运行：

$ chgrp -R www-data /srv/git

接下来我们要向 Apache 配置文件添加一些内容，来让 git-http-backend 作为 Web 服务器对 /git 路径请求的处理器。

SetEnv GIT\_PROJECT\_ROOT /srv/git

SetEnv GIT\_HTTP\_EXPORT\_ALL

ScriptAlias /git/ /usr/lib/git-core/git-http-backend/

如果留空 GIT\_HTTP\_EXPORT\_ALL 这个环境变量，Git 将只对无授权客户端提供带 git-daemon-export-ok 文件的版本库，就像 Git 守护进程一样。

最后，如果想让 Apache 允许 git-http-backend 请求并实现写入操作的授权验证，使用如下授权屏蔽配置即可：

<Files "git-http-backend">

AuthType Basic

AuthName "Git Access"

AuthUserFile /srv/git/.htpasswd

Require expr !(%{QUERY\_STRING} -strmatch '\*service=git-receive-pack\*' || %{REQUEST\_URI} =~ m#/git-receive-pack$#)

Require valid-user

</Files>

这需要你创建一个包含所有合法用户密码的 .htpasswd 文件。 以下是一个添加 “schacon” 用户到此文件的例子：

$ htpasswd -c /srv/git/.htpasswd schacon

你可以通过许多方式添加 Apache 授权用户，选择使用其中一种方式即可。 以上仅仅只是我们可以找到的最简单的一个例子。 如果愿意的话，你也可以通过 SSL 运行它，以保证所有数据是在加密状态下进行传输的。

我们不想深入去讲解 Apache 配置文件，因为你可能会使用不同的 Web 服务器，或者可能有不同的授权需求。 它的主要原理是使用一个 Git 附带的，名为 git-http-backend 的 CGI。它被引用来处理协商通过 HTTP 发送和接收的数据。 它本身并不包含任何授权功能，但是授权功能可以在 Web 服务器层引用它时被轻松实现。 你可以在任何所有可以处理 CGI 的 Web 服务器上办到这点，所以随便挑一个你最熟悉的 Web 服务器试手吧。

|  |  |
| --- | --- |
| **Note** | 欲了解更多的有关配置 Apache 授权访问的信息，请通过以下链接浏览 Apache 文档： <https://httpd.apache.org/docs/current/howto/auth.html> |

## 服务器上的 Git - GitWeb

### GitWeb

如果你对项目有读写权限或只读权限，你可能需要建立起一个基于网页的简易查看器。 Git 提供了一个叫做 GitWeb 的 CGI 脚本来做这项工作。



Figure 49. GitWeb 的网页用户界面

如果你想要查看 GitWeb 如何展示你的项目，并且在服务器上安装了轻量级 Web 服务器比如 lighttpd 或 webrick， Git 提供了一个命令来让你启动一个临时的服务器。 在 Linux 系统的电脑上，lighttpd 通常已经安装了，所以你只需要在项目目录里执行 git instaweb 命令即可。 如果你使用 Mac 系统， Mac OS X Leopard 系统已经预安装了 Ruby，所以 webrick 或许是你最好的选择。 如果不想使用 lighttpd 启动 instaweb 命令，你需要在执行时加入 --httpd 参数。

$ git instaweb --httpd=webrick

[2009-02-21 10:02:21] INFO WEBrick 1.3.1

[2009-02-21 10:02:21] INFO ruby 1.8.6 (2008-03-03) [universal-darwin9.0]

这个命令启动了一个监听 1234 端口的 HTTP 服务器，并且自动打开了浏览器。 这对你来说十分方便。 当你已经完成了工作并想关闭这个服务器，你可以执行同一个命令，并加上 --stop 选项：

$ git instaweb --httpd=webrick --stop

如果你现在想为你的团队或你托管的开源项目持续的运行这个页面，你需要通过普通的 Web 服务器来设置 CGI 脚本。 一些 Linux 发行版的软件库有 gitweb 包，可以通过 apt 或 dnf 来安装，你可以先试试。 接下来我们来快速的了解一下如何手动安装 GitWeb。 首先，你需要获得 Git 的源代码，它包含了 GitWeb ，并可以生成自定义的 CGI 脚本：

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

$ cd git/

$ make GITWEB\_PROJECTROOT="/srv/git" prefix=/usr gitweb

SUBDIR gitweb

SUBDIR ../

make[2]: `GIT-VERSION-FILE' is up to date.

GEN gitweb.cgi

GEN static/gitweb.js

$ sudo cp -Rf gitweb /var/www/

需要注意的是，你需要在命令中指定 GITWEB\_PROJECTROOT 变量来让程序知道你的 Git 版本库的位置。 现在，你需要在 Apache 中使用这个 CGI 脚本，你需要为此添加一个虚拟主机：

<VirtualHost \*:80>

ServerName gitserver

DocumentRoot /var/www/gitweb

<Directory /var/www/gitweb>

Options +ExecCGI +FollowSymLinks +SymLinksIfOwnerMatch

AllowOverride All

order allow,deny

Allow from all

AddHandler cgi-script cgi

DirectoryIndex gitweb.cgi

</Directory>

</VirtualHost>

再次提醒，GitWeb 可以通过任何一个支持 CGI 或 Perl 的网络服务器架设；如果你需要的话，架设起来应该不会很困难。 现在，你可以访问 http://gitserver/ 在线查看你的版本库。

## 服务器上的 Git - GitLab

### GitLab

虽然 GitWeb 相当简单。 但如果你正在寻找一个更现代，功能更全的 Git 服务器，这里有几个开源的解决方案可供你选择安装。 因为 GitLab 是其中最出名的一个，我们将它作为示例并讨论它的安装和使用。 这比 GitWeb 要复杂的多并且需要更多的维护，但它的确是一个功能更全的选择。

### 安装

GitLab 是一个数据库支持的 web 应用，所以相比于其他 git 服务器，它的安装过程涉及到更多的东西。 幸运的是，这个过程有非常详细的文档说明和支持。

这里有一些可参考的方法帮你安装 GitLab 。 为了更快速的启动和运行，你可以下载虚拟机镜像或者在 <https://bitnami.com/stack/gitlab> 上获取一键安装包，同时调整配置使之符合你特定的环境。 Bitnami 的一个优点在于它的登录界面（通过 alt+→ 键进入）， 它会告诉你安装好的 GitLab 的 IP 地址以及默认的用户名和密码。

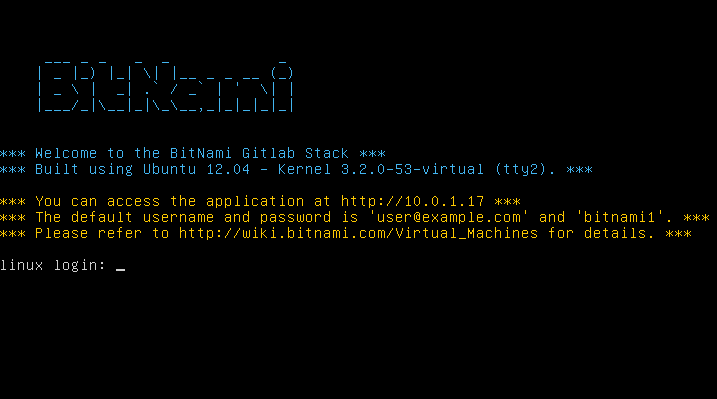


Figure 50. Bitnami GitLab 虚拟机登录界面。

无论如何，跟着 GitLab 社区版的 readme 文件一步步来，你可以在这里找到它 <https://gitlab.com/gitlab-org/gitlab-ce/tree/master> 。 在这里你将会在主菜单中找到安装 GitLab 的帮助，一个可以在 Digital Ocean 上运行的虚拟机，以及 RPM 和 DEB 包（都是测试版）。 这里还有 “非官方” 的引导让 GitLab 运行在非标准的操作系统和数据库上，一个全手动的安装脚本，以及许多其他的话题。

### 管理

GitLab 的管理界面是通过网络进入的。 将你的浏览器转到已经安装 GitLab 的 主机名或 IP 地址，然后以管理员身份登录即可。 默认的用户名是 admin@local.host，默认的密码是 5iveL!fe（你会得到类似 请登录后尽快更换密码 的提示）。 登录后，点击主栏上方靠右位置的 “Admin area” 图标进行管理。



Figure 51. GitLab 主栏的 “Admin area” 图标。

#### 使用者

GitLab 上的用户指的是对应协作者的帐号。 用户帐号没有很多复杂的地方，主要是包含登录数据的用户信息集合。 每一个用户账号都有一个 **命名空间** ，即该用户项目的逻辑集合。 如果一个叫 jane 的用户拥有一个名称是 project 的项目，那么这个项目的 url 会是 http://server/jane/project 。



Figure 52. GitLab 用户管理界面。

移除一个用户有两种方法。 “屏蔽（Blocking）” 一个用户阻止他登录 GitLab 实例，但是该用户命名空间下的所有数据仍然会被保存， 并且仍可以通过该用户提交对应的登录邮箱链接回他的个人信息页。

而另一方面，“销毁（Destroying）” 一个用户，会彻底的将他从数据库和文件系统中移除。 他命名空间下的所有项目和数据都会被删除，拥有的任何组也会被移除。 这显然是一个更永久且更具破坏力的行为，所以很少用到这种方法。

#### 组

一个 GitLab 的组是一些项目的集合，连同关于多少用户可以访问这些项目的数据。 每一个组都有一个项目命名空间（与用户一样），所以如果一个叫 training 的组拥有一个名称是 materials 的项目，那么这个项目的 url 会是 http://server/training/materials。



Figure 53. GitLab 组管理界面。

每一个组都有许多用户与之关联，每一个用户对组中的项目以及组本身的权限都有级别区分。 权限的范围从 “访客”（仅能提问题和讨论） 到 “拥有者”（完全控制组、成员和项目）。 权限的种类太多以至于难以在这里一一列举，不过在 GitLab 的管理界面上有帮助链接。

#### 项目

一个 GitLab 的项目相当于 git 的版本库。 每一个项目都属于一个用户或者一个组的单个命名空间。 如果这个项目属于一个用户，那么这个拥有者对所有可以获取这个项目的人拥有直接管理权； 如果这个项目属于一个组，那么该组中用户级别的权限也会起作用。

每一个项目都有一个可视级别，控制着谁可以看到这个项目页面和仓库。 如果一个项目是 **私有** 的，这个项目的拥有者必须明确授权从而使特定的用户可以访问。 一个 **内部** 的项目可以被所有登录的人看到，而一个 **公开** 的项目则是对所有人可见的。 注意，这种控制既包括 git fetch 的使用也包括对项目 web 用户界面的访问。

#### 钩子

GitLab 在项目和系统级别上都支持钩子程序。 对任意级别，当有相关事件发生时，GitLab 的服务器会执行一个包含描述性 JSON 数据的 HTTP 请求。 这是自动化连接你的 git 版本库和 GitLab 实例到其他的开发工具，比如 CI 服务器，聊天室，或者部署工具的一个极好方法。

### 基本用途

你想要在 GitLab 做的第一件事就是建立一个新项目。 这通过点击工具栏上的 “+” 图标完成。 你需要填写项目名称，项目所属命名空间，以及它的可视层级。 绝大多数的设定并不是永久的，可以通过设置界面重新调整。 点击 “Create Project”，你就完成了。

项目存在后，你可能会想将它与本地的 Git 版本库连接。 每一个项目都可以通过 HTTPS 或者 SSH 连接，任意两者都可以被用来配置远程 Git。 在项目主页的顶栏可以看到这个项目的 URLs。 对于一个存在的本地版本库，这个命令将会向主机位置添加一个叫 gitlab 的远程仓库：

$ git remote add gitlab https://server/namespace/project.git

如果你的本地没有版本库的副本，你可以这样做：

$ git clone https://server/namespace/project.git

web 用户界面提供了几个有用的获取版本库信息的网页。 每一个项目的主页都显示了最近的活动，并且通过顶部的链接可以使你浏览项目文件以及提交日志。

### 一起工作

在一个 GitLab 项目上一起工作的最简单方法就是赋予协作者对 git 版本库的直接 push 权限。 你可以通过项目设定的 “Members（成员）” 部分向一个项目添加协作者， 并且将这个新的协作者与一个访问级别关联（不同的访问级别在 [组](https://git-scm.com/book/zh/v2/ch00/_gitlab_groups_section) 中已简单讨论）。 通过赋予一个协作者 “Developer（开发者）” 或者更高的访问级别， 这个用户就可以毫无约束地直接向版本库或者向分支进行提交。

另外一个让合作更解耦的方法就是使用合并请求。 它的优点在于让任何能够看到这个项目的协作者在被管控的情况下对这个项目作出贡献。 可以直接访问的协作者能够简单的创建一个分支，向这个分支进行提交，也可以开启一个向 master 或者其他任何一个分支的合并请求。 对版本库没有推送权限的协作者则可以 “fork” 这个版本库（即创建属于自己的这个库的副本），向 **那个** 副本进行提交，然后从那个副本开启一个到主项目的合并请求。 这个模型使得项目拥有者完全控制着向版本库的提交，以及什么时候允许加入陌生协作者的贡献。

在 GitLab 中合并请求和问题是一个长久讨论的主要部分。 每一个合并请求都允许在提出改变的行进行讨论（它支持一个轻量级的代码审查），也允许对一个总体性话题进行讨论。 两者都可以被分配给用户，或者组织到 milestones（里程碑） 界面。

这个部分主要聚焦于在 GitLab 中与 Git 相关的特性，但是 GitLab 作为一个成熟的系统， 它提供了许多其他产品来帮助你协同工作，例如项目 wiki 与系统维护工具。 GitLab 的一个优点在于，服务器设置和运行以后，你将很少需要调整配置文件或通过 SSH 连接服务器。 绝大多数的管理和日常使用都可以在浏览器界面中完成。

## 服务器上的 Git - 第三方托管的选择

### 第三方托管的选择

如果不想设立自己的 Git 服务器，你可以选择将你的 Git 项目托管到一个外部专业的托管网站。 这带来了一些好处：一个托管网站可以用来快速建立并开始项目，且无需进行服务器维护和监控工作。 即使你在内部设立并且运行了自己的服务器，你仍然可以把你的开源代码托管在公共托管网站——这通常更有助于开源社区来发现和帮助你。

现在，有非常多的托管供你选择，每个选择都有不同的优缺点。 欲查看最新列表，请浏览 Git 维基的 GitHosting 页面 <https://git.wiki.kernel.org/index.php/GitHosting>

我们会在 [GitHub](https://git-scm.com/book/zh/v2/ch00/ch06-github) 详细讲解 GitHub，作为目前最大的 Git 托管平台， 你很可能需要与托管在 GitHub 上的项目进行交互，而且你也很可能并不想去设立你自己的 Git 服务器。

## 服务器上的 Git - 总结

你有多种远程存取 Git 仓库的选择便于与其他人合作或是分享你的工作。

运行你自己的服务器将有许多权限且允许你运行该服务于你自己的防火墙内，但如此通常需要耗费你大量的时间去设置与维护服务器。 如果你放置你的资料于托管服务器内，可轻易的设置与维护；无论如何，你必须能够保存你的代码在其他服务器，且某些组织不允许此作法。 这将直截了当的决定哪个作法或组合的方式较适合你或你的组织。

# 分布式 Git

## 分布式 Git - 分布式工作流程

你现在拥有了一个远程 Git 版本库，能为所有开发者共享代码提供服务，在一个本地工作流程下，你也已经熟悉了基本 Git 命令。你现在可以学习如何利用 Git 提供的一些分布式工作流程了。

这一章中，你将会学习如何作为贡献者或整合者，在一个分布式协作的环境中使用 Git。 你会学习为一个项目成功地贡献代码，并接触一些最佳实践方式，让你和项目的维护者能轻松地完成这个过程。另外，你也会学到如何管理有很多开发者提交贡献的项目。

### 分布式工作流程

与传统的集中式版本控制系统（CVCS）相反，Git 的分布式特性使得开发者间的协作变得更加灵活多样。 在集中式系统中，每个开发者就像是连接在集线器上的节点，彼此的工作方式大体相像。 而在 Git 中，每个开发者同时扮演着节点和集线器的角色——也就是说， 每个开发者既可以将自己的代码贡献到其他的仓库中，同时也能维护自己的公开仓库， 让其他人可以在其基础上工作并贡献代码。 由此，Git 的分布式协作可以为你的项目和团队衍生出种种不同的工作流程， 接下来的章节会介绍几种利用了 Git 的这种灵活性的常见应用方式。 我们将讨论每种方式的优点以及可能的缺点；你可以选择使用其中的某一种，或者将它们的特性混合搭配使用。

### 集中式工作流

集中式系统中通常使用的是单点协作模型——集中式工作流。 一个中心集线器，或者说 **仓库**，可以接受代码，所有人将自己的工作与之同步。 若干个开发者则作为节点，即中心仓库的消费者与中心仓库同步。



Figure 54. 集中式工作流。

这意味着如果两个开发者从中心仓库克隆代码下来，同时作了一些修改，那么只有第一个开发者可以顺利地把数据推送回共享服务器。 第二个开发者在推送修改之前，必须先将第一个人的工作合并进来，这样才不会覆盖第一个人的修改。 这和 Subversion （或任何 CVCS）中的概念一样，而且这个模式也可以很好地运用到 Git 中。

如果在公司或者团队中，你已经习惯了使用这种集中式工作流程，完全可以继续采用这种简单的模式。 只需要搭建好一个中心仓库，并给开发团队中的每个人推送数据的权限，就可以开展工作了。Git 不会让用户覆盖彼此的修改。

例如 John 和 Jessica 同时开始工作。 John 完成了他的修改并推送到服务器。 接着 Jessica 尝试提交她自己的修改，却遭到服务器拒绝。 她被告知她的修改正通过非快进式（non-fast-forward）的方式推送，只有将数据抓取下来并且合并后方能推送。 这种模式的工作流程的使用非常广泛，因为大多数人对其很熟悉也很习惯。

当然这并不局限于小团队。 利用 Git 的分支模型，通过同时在多个分支上工作的方式，即使是上百人的开发团队也可以很好地在单个项目上协作。

### 集成管理者工作流

Git 允许多个远程仓库存在，使得这样一种工作流成为可能：每个开发者拥有自己仓库的写权限和其他所有人仓库的读权限。 这种情形下通常会有个代表“官方”项目的权威的仓库。 要为这个项目做贡献，你需要从该项目克隆出一个自己的公开仓库，然后将自己的修改推送上去。 接着你可以请求官方仓库的维护者拉取更新合并到主项目。 维护者可以将你的仓库作为远程仓库添加进来，在本地测试你的变更，将其合并入他们的分支并推送回官方仓库。 这一流程的工作方式如下所示（见 [集成管理者工作流。](https://git-scm.com/book/zh/v2/ch00/wfdiag_b)）：

1. 项目维护者推送到主仓库。
2. 贡献者克隆此仓库，做出修改。
3. 贡献者将数据推送到自己的公开仓库。
4. 贡献者给维护者发送邮件，请求拉取自己的更新。
5. 维护者在自己本地的仓库中，将贡献者的仓库加为远程仓库并合并修改。
6. 维护者将合并后的修改推送到主仓库。



Figure 55. 集成管理者工作流。

这是 GitHub 和 GitLab 等集线器式（hub-based）工具最常用的工作流程。人们可以容易地将某个项目派生成为自己的公开仓库，向这个仓库推送自己的修改，并为每个人所见。 这么做最主要的优点之一是你可以持续地工作，而主仓库的维护者可以随时拉取你的修改。 贡献者不必等待维护者处理完提交的更新——每一方都可以按照自己的节奏工作。

### 主管与副主管工作流

这其实是多仓库工作流程的变种。 一般拥有数百位协作开发者的超大型项目才会用到这样的工作方式，例如著名的 Linux 内核项目。 被称为 **副主管（lieutenant）** 的各个集成管理者分别负责集成项目中的特定部分。 所有这些副主管头上还有一位称为 **主管（dictator）** 的总集成管理者负责统筹。 主管维护的仓库作为参考仓库，为所有协作者提供他们需要拉取的项目代码。 整个流程看起来是这样的（见 [主管与副主管工作流。](https://git-scm.com/book/zh/v2/ch00/wfdiag_c) ）：

1. 普通开发者在自己的主题分支上工作，并根据 master 分支进行变基。 这里是主管推送的参考仓库的 master 分支。
2. 副主管将普通开发者的主题分支合并到自己的 master 分支中。
3. 主管将所有副主管的 master 分支并入自己的 master 分支中。
4. 最后，主管将集成后的 master 分支推送到参考仓库中，以便所有其他开发者以此为基础进行变基。



Figure 56. 主管与副主管工作流。

这种工作流程并不常用，只有当项目极为庞杂，或者需要多级别管理时，才会体现出优势。 利用这种方式，项目总负责人（即主管）可以把大量分散的集成工作委托给不同的小组负责人分别处理，然后在不同时刻将大块的代码子集统筹起来，用于之后的整合。

### 工作流程总结

上面介绍了在 Git 等分布式系统中经常使用的工作流程，但是在实际的开发中，你会遇到许多可能适合你的特定工作流程的变种。 现在你应该已经清楚哪种工作流程组合可能比较适合你了，我们会给出一些如何扮演不同工作流程中主要角色的更具体的例子。 下一节我们将会学习为项目做贡献的一些常用模式。

## 分布式 Git - 向一个项目贡献

### 向一个项目贡献

描述如何向一个项目贡献的主要困难在于完成贡献有很多不同的方式。 因为 Git 非常灵活，人们可以通过不同的方式来一起工作，所以描述应该如何贡献并不是非常准确——每一个项目都有一点儿不同。 影响因素包括活跃贡献者的数量、选择的工作流程、提交权限与可能包含的外部贡献方法。

第一个影响因素是活跃贡献者的数量——积极地向这个项目贡献代码的用户数量以及他们的贡献频率。 在许多情况下，你可能会有两三个开发者一天提交几次，对于不活跃的项目可能更少。 对于大一些的公司或项目，开发者的数量可能会是上千，每天都有成百上千次提交。 这很重要，因为随着开发者越来越多，在确保你的代码能干净地应用或轻松地合并时会遇到更多问题。 提交的改动可能表现为过时的，也可能在你正在做改动或者等待改动被批准应用时被合并入的工作严重损坏。 如何保证代码始终是最新的，并且提交始终是有效的？

下一个影响因素是项目使用的工作流程。 它是中心化的吗，即每一个开发者都对主线代码有相同的写入权限？ 项目是否有一个检查所有补丁的维护者或整合者？ 是否所有的补丁是同行评审后批准的？ 你是否参与了那个过程？ 是否存在副官系统，你必须先将你的工作提交到上面？

下一个影响因素是提交权限。 是否有项目的写权限会使向项目贡献所需的流程有极大的不同。 如果没有写权限，项目会选择何种方式接受贡献的工作？ 是否甚至有一个如何贡献的规范？ 你一次贡献多少工作？ 你多久贡献一次？

所有这些问题都会影响实际如何向一个项目贡献，以及对你来说哪些工作流程更适合或者可用。 我们将会由浅入深，通过一系列用例来讲述其中的每一个方面；从这些例子应该能够建立实际中你需要的特定工作流程。

### 提交准则

在我们开始查看特定的用例前，这里有一个关于提交信息的快速说明。 有一个好的创建提交的准则并且坚持使用会让与 Git 工作和与其他人协作更容易。 Git 项目提供了一个文档，其中列举了关于创建提交到提交补丁的若干好的提示——可以在 Git 源代码中的 Documentation/SubmittingPatches 文件中阅读它。

首先，你的提交不应该包含任何空白错误。 Git 提供了一个简单的方式来检查这点——在提交前，运行 git diff --check，它将会找到可能的空白错误并将它们为你列出来。



Figure 57. git diff --check 的输出

如果在提交前运行那个命令，可以知道提交中是否包含可能会使其他开发者恼怒的空白问题。

接下来，尝试让每一个提交成为一个逻辑上的独立变更集。 如果可以，尝试让改动可以理解——不要在整个周末编码解决五个问题，然后在周一时将它们提交为一个巨大的提交。 即使在周末期间你无法提交，在周一时使用暂存区域将你的工作最少拆分为每个问题一个提交，并且为每一个提交附带一个有用的信息。 如果其中一些改动修改了同一个文件，尝试使用 git add --patch 来部分暂存文件（在 [交互式暂存](https://git-scm.com/book/zh/v2/ch00/_interactive_staging) 中有详细介绍）。 不管你做一个或五个提交，只要所有的改动都曾添加过，项目分支末端的快照就是一样的，所以尽量让你的开发者同事们在审查你的改动的时候更容易些吧。

当你之后需要时这个方法也会使拉出或还原一个变更集更容易些。 [重写历史](https://git-scm.com/book/zh/v2/ch00/_rewriting_history) 描述了重写历史与交互式暂存文件的若干有用的 Git 技巧——在将工作发送给其他人前使用这些工具来帮助生成一个干净又易懂的历史。

最后一件要牢记的事是提交信息。 有一个创建优质提交信息的习惯会使 Git 的使用与协作容易的多。 一般情况下，信息应当以少于 50 个字符（25个汉字）的单行开始且简要地描述变更，接着是一个空白行，再接着是一个更详细的解释。 Git 项目要求一个更详细的解释，包括做改动的动机和它的实现与之前行为的对比——这是一个值得遵循的好规则。 使用指令式的语气来编写提交信息，比如使用“Fix bug”而非“Fixed bug”或“Fixes bug”。 这里是一份 [最初由 Tim Pope 写的模板](https://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html)：

首字母大写的摘要（不多于 50 个字符）

如果必要的话，加入更详细的解释文字。在大概 72 个字符的时候换行。

在某些情形下，第一行被当作一封电子邮件的标题，剩下的文本作为正文。

分隔摘要与正文的空行是必须的（除非你完全省略正文），

如果你将两者混在一起，那么类似变基等工具无法正常工作。

使用指令式的语气来编写提交信息：使用“Fix bug”而非“Fixed bug”或“Fixes bug”。

此约定与 git merge 和 git revert 命令生成提交说明相同。

空行接着更进一步的段落。

- 标号也是可以的。

- 项目符号可以使用典型的连字符或星号，后跟一个空格，行之间用空行隔开，

但是可以依据不同的惯例有所不同。

- 使用悬挂式缩进

如果你所有的提交信息都遵循此模版，那么对你和与你协作的其他开发者来说事情会变得非常容易。 Git 项目有一个良好格式化的提交信息——尝试在那儿运行 git log --no-merges 来看看漂亮的格式化的项目提交历史像什么样。

|  |  |
| --- | --- |
| **Note** | **按我们说的去做，不要照着我们做的去做。**  为简单起见，本书中很多例子的提交说明并没有遵循这样良好的格式， 我们只是对 git commit 使用了 -m 选项。  简而言之，按我们说的去做，不要照着我们做的去做。 |

### 私有小型团队

你可能会遇到的最简单的配置是有一两个其他开发者的私有项目。 “私有” 在这个上下文中，意味着闭源——不可以从外面的世界中访问到。 你和其他的开发者都有仓库的推送权限。

在这个环境下，可以采用一个类似使用 Subversion 或其他集中式的系统时会使用的工作流程。 依然可以得到像离线提交、非常容易地新建分支与合并分支等高级功能，但是工作流程可以是很简单的；主要的区别是合并发生在客户端这边而不是在提交时发生在服务器那边。 让我们看看当两个开发者在一个共享仓库中一起工作时会是什么样子。 第一个开发者，John，克隆了仓库，做了改动，然后本地提交。 （为了缩短这些例子长度，协议信息已被替换为 ...。）

# John's Machine

$ git clone john@githost:simplegit.git

Cloning into 'simplegit'...

...

$ cd simplegit/

$ vim lib/simplegit.rb

$ git commit -am 'remove invalid default value'

[master 738ee87] remove invalid default value

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

第二个开发者，Jessica，做了同样的事情——克隆仓库并提交了一个改动：

# Jessica's Machine

$ git clone jessica@githost:simplegit.git

Cloning into 'simplegit'...

...

$ cd simplegit/

$ vim TODO

$ git commit -am 'add reset task'

[master fbff5bc] add reset task

1 files changed, 1 insertions(+), 0 deletions(-)

现在，Jessica 把她的工作推送到服务器上，一切正常：

# Jessica's Machine

$ git push origin master

...

To jessica@githost:simplegit.git

1edee6b..fbff5bc master -> master

上方输出信息中最后一行显示的是推送操作执行完毕后返回的一条很有用的消息。 消息的基本格式是 <oldref>..<newref> fromref -> toref ， oldref 的含义是推送前所指向的引用， newref 的含义是推送后所指向的引用， fromref 是将要被推送的本地引用的名字， toref 是将要被更新的远程引用的名字。 在后面的讨论中你还会看到类似的输出消息，所以对这条消息的含义有一些基础的了解将会帮助你理解仓库的诸多状态。 想要了解更多细节请访问文档 [git-push](https://git-scm.com/docs/git-push) 。

John 稍候也做了些改动，将它们提交到了本地仓库中，然后试着将它们推送到同一个服务器：

# John's Machine

$ git push origin master

To john@githost:simplegit.git

! [rejected] master -> master (non-fast forward)

error: failed to push some refs to 'john@githost:simplegit.git'

这时 John 会推送失败，因为之前 Jessica 已经推送了她的更改。 如果之前习惯于用 Subversion 那么理解这点特别重要，因为你会注意到两个开发者并没有编辑同一个文件。 尽管 Subversion 会对编辑的不同文件在服务器上自动进行一次合并，但 Git 要求你先在本地合并提交。 换言之，John 必须先抓取 Jessica 的上游改动并将它们合并到自己的本地仓库中，才能被允许推送。

第一步，John 抓取 Jessica 的工作（这只会 **抓取** Jessica 的上游工作，并不会将它合并到 John 的工作中）：

$ git fetch origin

...

From john@githost:simplegit

+ 049d078...fbff5bc master -> origin/master

在这个时候，John 的本地仓库看起来像这样：



Figure 58. John 的分叉历史

现在 John 可以将抓取下来的 Jessica 的工作合并到他自己的本地工作中了：

$ git merge origin/master

Merge made by the 'recursive' strategy.

TODO | 1 +

1 files changed, 1 insertions(+), 0 deletions(-)

合并进行得很顺利——John 更新后的历史现在看起来像这样：



Figure 59. 合并了 origin/master 之后 John 的仓库

此时，John 可能想要测试新的代码，以确保 Jessica 的工作没有影响他自己的工作， 当一切正常后，他就能将新合并的工作推送到服务器了：

$ git push origin master

...

To john@githost:simplegit.git

fbff5bc..72bbc59 master -> master

最终，John 的提交历史看起来像这样：



Figure 60. 推送到 origin 服务器后 John 的历史

在此期间，Jessica 新建了一个名为 issue54 的主题分支，然后在该分支上提交了三次。 她还没有抓取 John 的改动，所以她的提交历史看起来像这样：



Figure 61. Jessica 的主题分支

忽然，Jessica 发现 John 向服务器推送了一些新的工作，她想要看一下， 于是就抓取了所有服务器上的新内容：

# Jessica's Machine

$ git fetch origin

...

From jessica@githost:simplegit

fbff5bc..72bbc59 master -> origin/master

那会同时拉取 John 推送的工作。 Jessica 的历史现在看起来像这样：



Figure 62. 抓取 John 的改动后 Jessica 的历史

Jessica 认为她的主题分支已经准备好了，但她想知道需要将 John 工作的哪些合并到自己的工作中才能推送。 她运行 git log 找了出来：

$ git log --no-merges issue54..origin/master

commit 738ee872852dfaa9d6634e0dea7a324040193016

Author: John Smith <jsmith@example.com>

Date: Fri May 29 16:01:27 2009 -0700

remove invalid default value

issue54..origin/master 语法是一个日志过滤器，要求 Git 只显示所有在后面分支 （在本例中是 origin/master）但不在前面分支（在本例中是 issue54）的提交的列表。 我们将会在 [提交区间](https://git-scm.com/book/zh/v2/ch00/_commit_ranges) 中详细介绍这个语法。

目前，我们可以从输出中看到有一个 John 生成的但是 Jessica 还没有合并的提交。 如果她合并 origin/master，那个未合并的提交将会修改她的本地工作。

现在，Jessica 可以合并她的特性工作到她的 master 分支， 合并 John 的工作（origin/master）进入她的 master 分支，然后再次推送回服务器。

首先（在已经提交了所有 issue54 主题分支上的工作后），为了整合所有这些工作， 她切换回她的 master 分支。

$ git checkout master

Switched to branch 'master'

Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.

Jessica 既可以先合并 origin/master 也可以先合并 issue54 ——它们都是上游，所以顺序并没有关系。 不论她选择的顺序是什么最终的结果快照是完全一样的；只是历史会稍微有些不同。 她选择先合并 issue54：

$ git merge issue54

Updating fbff5bc..4af4298

Fast forward

README | 1 +

lib/simplegit.rb | 6 +++++-

2 files changed, 6 insertions(+), 1 deletions(-)

没有发生问题，如你所见它是一次简单的快进合并。 现在 Jessica 在本地合并了之前抓取的 origin/master 分支上 John 的工作：

$ git merge origin/master

Auto-merging lib/simplegit.rb

Merge made by the 'recursive' strategy.

lib/simplegit.rb | 2 +-

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

每一个文件都干净地合并了，Jessica 的历史现在看起来像这样：



Figure 63. 合并了 John 的改动后 Jessica 的历史

现在 origin/master 是可以从 Jessica 的 master 分支到达的， 所以她应该可以成功地推送（假设同一时间 John 并没有更多推送）：

$ git push origin master

...

To jessica@githost:simplegit.git

72bbc59..8059c15 master -> master

每一个开发者都提交了几次并成功地合并了其他人的工作。



Figure 64. 推送所有的改动回服务器后 Jessica 的历史

这是一个最简单的工作流程。 你通常会在一个主题分支上工作一会儿，当它准备好整合时就合并到你的 master 分支。 当想要共享工作时，如果有改动的话就抓取它然后合并到你自己的 master 分支， 之后推送到服务器上的 master 分支。通常顺序像这样：



Figure 65. 一个简单的多人 Git 工作流程的通常事件顺序

### 私有管理团队

在接下来的场景中，你会看到大型私有团队中贡献者的角色。 你将学到如何在这种工作环境中工作，其中小组基于特性进行协作，而这些团队的贡献将会由其他人整合。

让我们假设 John 与 Jessica 在一个特性（featureA）上工作， 同时 Jessica 与第三个开发者 Josie 在第二个特性（featureB）上工作。 在本例中，公司使用了一种整合-管理者工作流程，独立小组的工作只能被特定的工程师整合， 主仓库的 master 分支只能被那些工程师更新。 在这种情况下，所有的工作都是在基于团队的分支上完成的并且稍后会被整合者拉到一起。

因为 Jessica 在两个特性上工作，并且平行地与两个不同的开发者协作，让我们跟随她的工作流程。 假设她已经克隆了仓库，首先决定在 featureA 上工作。 她为那个特性创建了一个新分支然后在那做了一些工作：

# Jessica's Machine

$ git checkout -b featureA

Switched to a new branch 'featureA'

$ vim lib/simplegit.rb

$ git commit -am 'add limit to log function'

[featureA 3300904] add limit to log function

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

在这个时候，她需要将工作共享给 John，所以她推送了 featureA 分支的提交到服务器上。 Jessica 没有 master 分支的推送权限——只有整合者有——所以为了与 John 协作必须推送另一个分支。

$ git push -u origin featureA

...

To jessica@githost:simplegit.git

\* [new branch] featureA -> featureA

Jessica 向 John 发邮件告诉他已经推送了一些工作到 featureA 分支现在可以看一看。 当她等待 John 的反馈时，Jessica 决定与 Josie 开始在 featureB 上工作。 为了开始工作，她基于服务器的 master 分支开始了一个新分支。

# Jessica's Machine

$ git fetch origin

$ git checkout -b featureB origin/master

Switched to a new branch 'featureB'

现在，Jessica 在 featureB 分支上创建了几次提交：

$ vim lib/simplegit.rb

$ git commit -am 'made the ls-tree function recursive'

[featureB e5b0fdc] made the ls-tree function recursive

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

$ vim lib/simplegit.rb

$ git commit -am 'add ls-files'

[featureB 8512791] add ls-files

1 files changed, 5 insertions(+), 0 deletions(-)

现在 Jessica 的仓库看起来像这样：



Figure 66. Jessica 的初始提交历史

她准备好推送工作了，但是一封来自 Josie 的邮件告知一些初始的“featureB” 工作已经被推送到服务器的 featureBee 上了。 Jessica 在能够将她的工作推送到服务器前，需要将那些改动与她自己的合并。 她首先通过 git fetch 抓取了 Josie 的改动：

$ git fetch origin

...

From jessica@githost:simplegit

\* [new branch] featureBee -> origin/featureBee

假设 Jessica 还在她检出的 featureB 分支上，现在可以通过 git merge 将其合并到她做的工作中了：

$ git merge origin/featureBee

Auto-merging lib/simplegit.rb

Merge made by the 'recursive' strategy.

lib/simplegit.rb | 4 ++++

1 files changed, 4 insertions(+), 0 deletions(-)

此时，Jessica 想要将所有合并后的“featureB”推送回服务器，，但她并不想直接推送她自己的 featureB 分支。 由于 Josie 已经开启了一个上游的 featureBee 分支，因此 Jessica 想要推送到 **这个** 分支上，于是她这样做：

$ git push -u origin featureB:featureBee

...

To jessica@githost:simplegit.git

fba9af8..cd685d1 featureB -> featureBee

这称作一个 **引用规范**。 查看 [引用规范](https://git-scm.com/book/zh/v2/ch00/_refspec) 了解关于 Git 引用规范与通过它们可以做的不同的事情的详细讨论。 也要注意 -u 标记；这是 --set-upstream 的简写，该标记会为之后轻松地推送与拉取配置分支。

紧接着，John 发邮件给 Jessica 说他已经推送了一些改动到 featureA 分支并要求她去验证它们。 她运行一个 git fetch 来拉取下那些改动：

$ git fetch origin

...

From jessica@githost:simplegit

3300904..aad881d featureA -> origin/featureA

Jessica 通过比较新抓取的 featureA 分支和她同一分支的本地副本，看到了 John 的新工作日志。

$ git log featureA..origin/featureA

commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6

Author: John Smith <jsmith@example.com>

Date: Fri May 29 19:57:33 2009 -0700

changed log output to 30 from 25

如果 Jessica 觉得可以，她就能将 John 的新工作合并到她本地的 featureA 分支上：

$ git checkout featureA

Switched to branch 'featureA'

$ git merge origin/featureA

Updating 3300904..aad881d

Fast forward

lib/simplegit.rb | 10 +++++++++-

1 files changed, 9 insertions(+), 1 deletions(-)

最后，Jessica 可能想要对整个合并后的内容做一些小修改， 于是她将这些修改提交到了本地的 featureA 分支，接着将最终的结果推送回了服务器。

$ git commit -am 'small tweak'

[featureA 774b3ed] small tweak

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

$ git push

...

To jessica@githost:simplegit.git

3300904..774b3ed featureA -> featureA

Jessica 的提交历史现在看起来像这样：



Figure 67. 在一个主题分支提交后 Jessica 的历史

这时，Jessica、Josie 与 John 通知整合者服务器上的 featureA 与 featureBee 分支准备好整合到主线中了。 在整合者将这些分支合并到主线后，就能一次将这个新的合并提交抓取下来，历史看起来就会像这样：



Figure 68. 合并了 Jessica 的两个主题分支后她的历史

许多团队切换到 Git 就是看中了这种能让多个团队并行工作、并在之后合并不同工作的能力。 团队中更小一些的子小组可以通过远程分支协作而不必影响或妨碍整个团队的能力是 Git 的一个巨大优势。 在这儿看到的工作流程顺序类似这样：



Figure 69. 这种管理团队工作流程的基本顺序

### 派生的公开项目

向公开项目做贡献有一点儿不同。 因为没有权限直接更新项目的分支，你必须用其他办法将工作给维护者。 第一个例子描述在支持简单派生的 Git 托管上使用派生来做贡献。 许多托管站点支持这个功能（包括 GitHub、BitBucket、repo.or.cz 等等），许多项目维护者期望这种风格的贡献。 下一节会讨论偏好通过邮件接受贡献补丁的项目。

首先，你可能想要克隆主仓库，为计划贡献的补丁或补丁序列创建一个主题分支，然后在那儿做工作。 顺序看起来基本像这样：

$ git clone <url>

$ cd project

$ git checkout -b featureA

... work ...

$ git commit

... work ...

$ git commit

|  |  |
| --- | --- |
| **Note** | 你可以用 rebase -i 将工作压缩成一个单独的提交，或者重排提交中的工作使补丁更容易被维护者审核—— 查看 [重写历史](https://git-scm.com/book/zh/v2/ch00/_rewriting_history) 了解关于交互式变基的更多信息。 |

当你的分支工作完成后准备将其贡献回维护者，去原始项目中然后点击“Fork”按钮，创建一份自己的可写的项目派生仓库。 然后需要在本地仓库中将该仓库添加为一个新的远程仓库，在本例中称作 myfork：

$ git remote add myfork <url>

然后需要推送工作到上面。 相对于合并到主分支再推送上去，推送你正在工作的主题分支到仓库上更简单。 原因是工作如果不被接受或者是被拣选的，就不必回退你的 master 分支 （拣选操作 cherry-pick 详见 [变基与拣选工作流](https://git-scm.com/book/zh/v2/ch00/_rebase_cherry_pick)）。 如果维护者合并、变基或拣选你的工作，不管怎样你最终会通过拉取他们的仓库找回来你的工作。

在任何情况下，你都可以使用下面的命令推送你的工作：

$ git push -u myfork featureA

当工作已经被推送到你的派生仓库后，你需要通知原项目的维护者你有想要他们合并的工作。 这通常被称作一个 **拉取请求（Pull Request）**，你通常可以通过网站生成它—— GitHub 有它自己的 Pull Request 机制，我们将会在 [GitHub](https://git-scm.com/book/zh/v2/ch00/ch06-github) 介绍——也可以运行 git request-pull 命令然后将随后的输出通过电子邮件手动发送给项目维护者。

git request-pull 命令接受一个要拉取主题分支的基础分支，以及它们要拉取的 Git 仓库的 URL， 产生一个请求拉取的所有修改的摘要。 例如，Jessica 想要发送给 John 一个拉取请求，她已经在刚刚推送的分支上做了两次提交。她可以运行这个：

$ git request-pull origin/master myfork

The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:

Jessica Smith (1):

added a new function

are available in the git repository at:

git://githost/simplegit.git featureA

Jessica Smith (2):

add limit to log function

change log output to 30 from 25

lib/simplegit.rb | 10 +++++++++-

1 files changed, 9 insertions(+), 1 deletions(-)

此输出可被发送给维护者——它告诉他们工作是从哪个分支开始的、提交的摘要、以及从哪里拉取这些工作。

在一个你不是维护者的项目上，通常有一个总是跟踪 origin/master 的 master 分支会很方便，在主题分支上做工作是因为如果它们被拒绝时你可以轻松地丢弃。 如果同一时间主仓库移动了然后你的提交不再能干净地应用，那么使工作主题独立于主题分支也会使你变基（rebase）工作时更容易。 例如，你想要提供第二个特性工作到项目，不要继续在刚刚推送的主题分支上工作——从主仓库的 master 分支重新开始：

$ git checkout -b featureB origin/master

... work ...

$ git commit

$ git push myfork featureB

$ git request-pull origin/master myfork

... email generated request pull to maintainer ...

$ git fetch origin

现在，每一个特性都保存在一个贮藏库中——类似于补丁队列——可以重写、变基与修改而不会让特性互相干涉或互相依赖，像这样：



Figure 70. featureB 的初始提交历史

假设项目维护者已经拉取了一串其他补丁，然后尝试拉取你的第一个分支，但是没有干净地合并。 在这种情况下，可以尝试变基那个分支到 origin/master 的顶部，为维护者解决冲突，然后重新提交你的改动：

$ git checkout featureA

$ git rebase origin/master

$ git push -f myfork featureA

这样会重写你的历史，现在看起来像是 [featureA 工作之后的提交历史](https://git-scm.com/book/zh/v2/ch00/psp_b)



Figure 71. featureA 工作之后的提交历史

因为你将分支变基了，所以必须为推送命令指定 -f 选项，这样才能将服务器上有一个不是它的后代的提交的 featureA 分支替换掉。 一个替代的选项是推送这个新工作到服务器上的一个不同分支（可能称作 featureAv2）。

让我们看一个更有可能的情况：维护者看到了你的第二个分支上的工作并且很喜欢其中的概念，但是想要你修改一下实现的细节。 你也可以利用这次机会将工作基于项目现在的 master 分支。 你从现在的 origin/master 分支开始一个新分支，在那儿压缩 featureB 的改动，解决任何冲突，改变实现，然后推送它为一个新分支。

$ git checkout -b featureBv2 origin/master

$ git merge --squash featureB

... change implementation ...

$ git commit

$ git push myfork featureBv2

--squash 选项接受被合并的分支上的所有工作，并将其压缩至一个变更集， 使仓库变成一个真正的合并发生的状态，而不会真的生成一个合并提交。 这意味着你的未来的提交将会只有一个父提交，并允许你引入另一个分支的所有改动， 然后在记录一个新提交前做更多的改动。同样 --no-commit 选项在默认合并过程中可以用来延迟生成合并提交。

现在你可以给维护者发送一条消息，表示你已经做了要求的修改然后他们可以在你的 featureBv2 分支上找到那些改动。



Figure 72. featureBv2 工作之后的提交历史

### 通过邮件的公开项目

许多项目建立了接受补丁的流程——需要检查每一个项目的特定规则，因为它们之间有区别。 因为有几个历史悠久的、大型的项目会通过一个开发者的邮件列表接受补丁，现在我们将会通过一个例子来演示。

工作流程与之前的用例是类似的——你为工作的每一个补丁序列创建主题分支。 区别是如何提交它们到项目中。 生成每一个提交序列的电子邮件版本然后邮寄它们到开发者邮件列表，而不是派生项目然后推送到你自己的可写版本。

$ git checkout -b topicA

... work ...

$ git commit

... work ...

$ git commit

现在有两个提交要发送到邮件列表。 使用 git format-patch 来生成可以邮寄到列表的 mbox 格式的文件——它将每一个提交转换为一封电子邮件，提交信息的第一行作为主题，剩余信息与提交引入的补丁作为正文。 它有一个好处是使用 format-patch 生成的一封电子邮件应用的提交正确地保留了所有的提交信息。

$ git format-patch -M origin/master

0001-add-limit-to-log-function.patch

0002-changed-log-output-to-30-from-25.patch

format-patch 命令打印出它创建的补丁文件名字。 -M 开关告诉 Git 查找重命名。 文件最后看起来像这样：

$ cat 0001-add-limit-to-log-function.patch

From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001

From: Jessica Smith <jessica@example.com>

Date: Sun, 6 Apr 2008 10:17:23 -0700

Subject: [PATCH 1/2] add limit to log function

Limit log functionality to the first 20

---

lib/simplegit.rb | 2 +-

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

diff --git a/lib/simplegit.rb b/lib/simplegit.rb

index 76f47bc..f9815f1 100644

--- a/lib/simplegit.rb

+++ b/lib/simplegit.rb

@@ -14,7 +14,7 @@ class SimpleGit

end

def log(treeish = 'master')

- command("git log #{treeish}")

+ command("git log -n 20 #{treeish}")

end

def ls\_tree(treeish = 'master')

--

2.1.0

也可以编辑这些补丁文件为邮件列表添加更多不想要在提交信息中显示出来的信息。 如果在 --- 行与补丁开头（diff --git 行）之间添加文本，那么开发者就可以阅读它，但是应用补丁时会忽略它。

为了将其邮寄到邮件列表，你既可以将文件粘贴进电子邮件客户端，也可以通过命令行程序发送它。 粘贴文本经常会发生格式化问题，特别是那些不会合适地保留换行符与其他空白的 “更聪明的” 客户端。 幸运的是，Git 提供了一个工具帮助你通过 IMAP 发送正确格式化的补丁，这可能对你更容易些。 我们将会演示如何通过 Gmail 发送一个补丁，它正好是我们所知最好的邮件代理；可以在之前提到的 Git 源代码中的 Documentation/SubmittingPatches 文件的最下面了解一系列邮件程序的详细指令。

首先，需要在 ~/.gitconfig 文件中设置 imap 区块。 可以通过一系列的 git config 命令来分别设置每一个值，或者手动添加它们，不管怎样最后配置文件应该看起来像这样：

[imap]

folder = "[Gmail]/Drafts"

host = imaps://imap.gmail.com

user = user@gmail.com

pass = YX]8g76G\_2^sFbd

port = 993

sslverify = false

如果 IMAP 服务器不使用 SSL，最后两行可能没有必要，host 的值会是 imap:// 而不是 imaps://。 当那些设置完成后，可以使用 git imap-send 将补丁序列放在特定 IMAP 服务器的 Drafts 文件夹中：

$ cat \*.patch |git imap-send

Resolving imap.gmail.com... ok

Connecting to [74.125.142.109]:993... ok

Logging in...

sending 2 messages

100% (2/2) done

此时，你可以到 Drafts 文件夹中，修改收件人字段为想要发送补丁的邮件列表， 可能需要抄送给维护者或负责那个部分的人，然后发送。

你也可以通过一个 SMTP 服务器发送补丁。 同之前一样，你可以通过一系列的 git config 命令来分别设置选项， 或者你可以手动地将它们添加到你的 ~/.gitconfig 文件的 sendmail 区块：

[sendemail]

smtpencryption = tls

smtpserver = smtp.gmail.com

smtpuser = user@gmail.com

smtpserverport = 587

当这完成后，你可以使用 git send-email 发送你的补丁：

$ git send-email \*.patch

0001-added-limit-to-log-function.patch

0002-changed-log-output-to-30-from-25.patch

Who should the emails appear to be from? [Jessica Smith <jessica@example.com>]

Emails will be sent from: Jessica Smith <jessica@example.com>

Who should the emails be sent to? jessica@example.com

Message-ID to be used as In-Reply-To for the first email? y

然后，对于正在发送的每一个补丁，Git 会吐出这样的一串日志信息：

(mbox) Adding cc: Jessica Smith <jessica@example.com> from

\line 'From: Jessica Smith <jessica@example.com>'

OK. Log says:

Sendmail: /usr/sbin/sendmail -i jessica@example.com

From: Jessica Smith <jessica@example.com>

To: jessica@example.com

Subject: [PATCH 1/2] added limit to log function

Date: Sat, 30 May 2009 13:29:15 -0700

Message-Id: <1243715356-61726-1-git-send-email-jessica@example.com>

X-Mailer: git-send-email 1.6.2.rc1.20.g8c5b.dirty

In-Reply-To: <y>

References: <y>

Result: OK

### 总结

这个部分介绍了处理可能会遇到的几个迥然不同类型的 Git 项目的一些常见的工作流程， 介绍了帮助管理这个过程的一些新工具。 接下来，你会了解到如何在贡献的另一面工作：维护一个 Git 项目。 你将会学习如何成为一个仁慈的独裁者或整合管理者。

## 分布式 Git - 维护项目

### 维护项目

除了如何有效地参与一个项目的贡献之外，你可能也需要了解如何维护项目。 这包含接受并应用别人使用 format-patch 生成并通过电子邮件发送过来的补丁， 或对项目添加的远程版本库分支中的更改进行整合。 但无论是管理版本库，还是帮忙验证、审核收到的补丁，都需要同其他贡献者约定某种长期可持续的工作方式。

### 在主题分支中工作

如果你想向项目中整合一些新东西，最好将这些尝试局限在 **主题分支**——一种通常用来尝试新东西的临时分支中。 这样便于单独调整补丁，如果遇到无法正常工作的情况，可以先不用管，等到有时间的时候再来处理。 如果你基于你所尝试进行工作的特性为分支创建一个简单的名字，比如 ruby\_client 或者具有类似描述性的其他名字，这样即使你必须暂时抛弃它，以后回来时也不会忘记。 项目的维护者一般还会为这些分支附带命名空间，比如 sc/ruby\_client（其中 sc 是贡献该项工作的人名称的简写）。 你应该记得，可以使用如下方式基于 master 分支建立主题分支：

$ git branch sc/ruby\_client master

或者如果你同时想立刻切换到新分支上的话，可以使用 checkout -b 选项：

$ git checkout -b sc/ruby\_client master

现在你已经准备好将你收到的贡献加入到这个主题分支，并考虑是否将其合并到长期分支中去了。

### 应用来自邮件的补丁

如果你通过电子邮件收到了一个需要整合进入项目的补丁，你需要将其应用到主题分支中进行评估。 有两种应用该种补丁的方法：使用 git apply，或者使用 git am。

#### 使用 apply 命令应用补丁

如果你收到了一个使用 git diff 或 Unix diff 命令的变体（不推荐使用这种方式，具体见下一节） 创建的补丁，可以使用 git apply 命令来应用。 假设你将补丁保存在了 /tmp/patch-ruby-client.patch 中，可以这样应用补丁：

$ git apply /tmp/patch-ruby-client.patch

这会修改工作目录中的文件。 它与运行 patch -p1 命令来应用补丁几乎是等效的，但是这种方式更加严格，相对于 patch 来说，它能够接受的模糊匹配更少。 它也能够处理 git diff 格式文件所描述的文件添加、删除和重命名操作，而 patch 则不会。 最后，git apply 命令采用了一种“全部应用，否则就全部撤销（apply all or abort all）”的模型， 即补丁只有全部内容都被应用和完全不被应用两个状态，而 patch 可能会导致补丁文件被部分应用， 最后使你的工作目录保持在一个比较奇怪的状态。 总体来看，git apply 命令要比 patch 谨慎得多。 并且，它不会为你创建提交——在运行之后，你需要手动暂存并提交补丁所引入的更改。

在实际应用补丁前，你还可以使用 git apply 来检查补丁是否可以顺利应用——即对补丁运行 git apply --check 命令：

$ git apply --check 0001-seeing-if-this-helps-the-gem.patch

error: patch failed: ticgit.gemspec:1

error: ticgit.gemspec: patch does not apply

如果没有产生输出，则该补丁可以顺利应用。 如果检查失败了，该命令还会以一个非零的状态退出，所以需要时你也可以在脚本中使用它。

#### 使用 am 命令应用补丁

如果补丁的贡献者也是一个 Git 用户，并且其能熟练使用 format-patch 命令来生成补丁，这样的话你的工作会变得更加轻松，因为这种补丁中包含了作者信息和提交信息供你参考。 如果可能的话，请鼓励贡献者使用 format-patch 而不是 diff 来为你生成补丁。 而只有对老式的补丁，你才必须使用 git apply 命令。

要应用一个由 format-patch 命令生成的补丁，你应该使用 git am 命令 （该命令的名字 am 表示它“应用（Apply）一系列来自邮箱（Mailbox）的补丁”）。 从技术的角度看，git am 是为了读取 mbox 文件而构建的， mbox 是一种用来在单个文本文件中存储一个或多个电子邮件消息的简单纯文本格式。 其大致格式如下所示：

From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001

From: Jessica Smith <jessica@example.com>

Date: Sun, 6 Apr 2008 10:17:23 -0700

Subject: [PATCH 1/2] add limit to log function

Limit log functionality to the first 20

这其实就是你前面看到的 git format-patch 命令输出的开始几行， 而同时它也是有效的 mbox 电子邮件格式。 如果有人使用 git send-email 命令将补丁以电子邮件的形式发送给你， 你便可以将它下载为 mbox 格式的文件，之后将 git am 命令指向该文件，它会应用其中包含的所有补丁。 如果你所使用的邮件客户端能够同时将多封邮件保存为 mbox 格式的文件， 你甚至能够将一系列补丁打包为单个 mbox 文件，并利用 git am 命令将它们一次性全部应用。

然而，如果贡献者将 git format-patch 生成的补丁文件上传到工单系统或类似的任务处理系统， 你可以先将其保存到本地，之后通过 git am 来应用补丁：

$ git am 0001-limit-log-function.patch

Applying: add limit to log function

你会看到补丁被顺利地应用，并且为你自动创建了一个新的提交。 其中的作者信息来自于电子邮件头部的 From 和 Date 字段，提交消息则取自 Subject 和邮件正文中补丁之前的内容。 比如，应用上面那个 mbox 示例后生成的提交是这样的：

$ git log --pretty=fuller -1

commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0

Author: Jessica Smith <jessica@example.com>

AuthorDate: Sun Apr 6 10:17:23 2008 -0700

Commit: Scott Chacon <schacon@gmail.com>

CommitDate: Thu Apr 9 09:19:06 2009 -0700

add limit to log function

Limit log functionality to the first 20

其中 Commit 信息表示的是应用补丁的人和应用补丁的时间。 Author 信息则表示补丁的原作者和原本的创建时间。

但是，有时候无法顺利地应用补丁。 这也许是因为你的主分支和创建补丁的分支相差较多，也有可能是因为这个补丁依赖于其他你尚未应用的补丁。 这种情况下，git am 进程将会报错并且询问你要做什么：

$ git am 0001-seeing-if-this-helps-the-gem.patch

Applying: seeing if this helps the gem

error: patch failed: ticgit.gemspec:1

error: ticgit.gemspec: patch does not apply

Patch failed at 0001.

When you have resolved this problem run "git am --resolved".

If you would prefer to skip this patch, instead run "git am --skip".

To restore the original branch and stop patching run "git am --abort".

该命令将会在所有出现问题的文件内加入冲突标记，就和发生冲突的合并或变基操作一样。 而你解决问题的手段很大程度上也是一样的——即手动编辑那些文件来解决冲突，暂存新的文件， 之后运行 git am --resolved 继续应用下一个补丁：

$ (fix the file)

$ git add ticgit.gemspec

$ git am --resolved

Applying: seeing if this helps the gem

如果你希望 Git 能够尝试以更加智能的方式解决冲突，你可以对其传递 -3 选项来使 Git 尝试进行三方合并。 该选项默认并没有打开，因为如果用于创建补丁的提交并不在你的版本库内的话，这样做是没有用处的。 而如果你确实有那个提交的话——比如补丁是基于某个公共提交的——那么通常 -3 选项对于应用有冲突的补丁是更加明智的选择。

$ git am -3 0001-seeing-if-this-helps-the-gem.patch

Applying: seeing if this helps the gem

error: patch failed: ticgit.gemspec:1

error: ticgit.gemspec: patch does not apply

Using index info to reconstruct a base tree...

Falling back to patching base and 3-way merge...

No changes -- Patch already applied.

比如上面这种情况，如果没有 -3 选项的话，这看起来就像是存在一个冲突。 由于使用了 -3 选项，该补丁就被干净地应用了

如果你正在利用一个 mbox 文件应用多个补丁，也可以在交互模式下运行 am 命令， 这样在每个补丁之前，它会停住询问你是否要应用该补丁：

$ git am -3 -i mbox

Commit Body is:

--------------------------

seeing if this helps the gem

--------------------------

Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all

这在你保存的补丁较多时很好用，因为你可以在应用之前查看忘掉内容的补丁，并且跳过已经应用过的补丁。

当与你的特性相关的所有补丁都被应用并提交到分支中之后，你就可以选择是否以及如何将其整合到更长期的分支中去了。

### 检出远程分支

如果你的贡献者建立了自己的版本库，并且向其中推送了若干修改， 之后将版本库的 URL 和包含更改的远程分支发送给你，那么你可以将其添加为一个远程分支，并且在本地进行合并。

比如 Jessica 向你发送了一封电子邮件，内容是在她的版本库中的 ruby-client 分支中有一个很不错的新功能， 为了测试该功能，你可以将其添加为一个远程分支，并在本地检出：

$ git remote add jessica git://github.com/jessica/myproject.git

$ git fetch jessica

$ git checkout -b rubyclient jessica/ruby-client

如果她再次发邮件说另一个分支中包含另一个优秀功能，因为之前已经设置好远程分支了， 你就可以直接进行 fetch 和 checkout 操作。

这对于与他人长期合作工作来说很有用。 而对于提交补丁频率较小的贡献者，相对于每个人维护自己的服务器，不断增删远程分支的做法，使用电子邮件来接收可能会比较省时。 况且你也不会想要加入数百个只提供一两个补丁的远程分支。 然而，脚本和托管服务在一定程度上可以简化这些工作——这很大程度上依赖于你和你的贡献者开发的方式。

这种方式的另一种优点是你可以同时得到提交历史。 虽然代码合并中可能会出现问题，但是你能获知他人的工作是基于你的历史中的具体哪一个位置；所以 Git 会默认进行三方合并，不需要提供 -3 选项，你也不需要担心补丁是基于某个你无法访问的提交生成的。

对于非持续性的合作，如果你依然想要以这种方式拉取数据的话，你可以对远程版本库的 URL 调用 git pull 命令。 这会执行一个一次性的抓取，而不会将该 URL 存为远程引用：

$ git pull https://github.com/onetimeguy/project

From https://github.com/onetimeguy/project

\* branch HEAD -> FETCH\_HEAD

Merge made by the 'recursive' strategy.

### 确定引入了哪些东西

你已经有了一个包含其他人贡献的主题分支。 现在你可以决定如何处理它们了。 本节回顾了若干命令，以便于你检查若将其合并入主分支所引入的更改。

一般来说，你应该对该分支中所有 master 分支尚未包含的提交进行检查。 通过在分支名称前加入 --not 选项，你可以排除 master 分支中的提交。 这和我们之前使用的 master..contrib 格式是一样的。 假设贡献者向你发送了两个补丁，为此你创建了一个名叫 contrib 的分支并在其上应用补丁，你可以运行：

$ git log contrib --not master

commit 5b6235bd297351589efc4d73316f0a68d484f118

Author: Scott Chacon <schacon@gmail.com>

Date: Fri Oct 24 09:53:59 2008 -0700

seeing if this helps the gem

commit 7482e0d16d04bea79d0dba8988cc78df655f16a0

Author: Scott Chacon <schacon@gmail.com>

Date: Mon Oct 22 19:38:36 2008 -0700

updated the gemspec to hopefully work better

如果要查看每次提交所引入的具体修改，你应该记得可以给 git log 命令传递 -p 选项，这样它会在每次提交后面附加对应的差异（diff）。

而要查看将该主题分支与另一个分支合并的完整 diff，你可能需要使用一个有些奇怪的技巧来得到正确的结果。 你可能会想到这种方式：

$ git diff master

这个命令会输出一个 diff，但它可能并不是我们想要的。 如果在你创建主题分支之后，master 分支向前移动了，你获得的结果就会显得有些不对。 这是因为 Git 会直接将该主题分支与 master 分支的最新提交快照进行比较。 比如说你在 master 分支中向某个文件添加了一行内容，那么直接比对最新快照的结果看上去就像是你在主题分支中将这一行删除了。

如果 master 分支是你的主题分支的直接祖先，其实是没有任何问题的； 但是一旦两个分支的历史产生了分叉，上述比对产生的 diff 看上去就像是将主题分支中所有的新东西加入， 并且将 master 分支所独有的东西删除。

而你真正想要检查的东西，实际上仅仅是主题分支所添加的更改——也就是该分支与 master 分支合并所要引入的工作。 要达到此目的，你需要让 Git 对主题分支上最新的提交与该分支与 master 分支的首个公共祖先进行比较。

从技术的角度讲，你可以以手工的方式找出公共祖先，并对其显式运行 diff 命令：

$ git merge-base contrib master

36c7dba2c95e6bbb78dfa822519ecfec6e1ca649

$ git diff 36c7db

或者，更简洁的形式：

$ git diff $(git merge-base contrib master)

然而，这种做法比较麻烦，所以 Git 提供了一种比较便捷的方式：三点语法。 对于 git diff 命令来说，你可以通过把 ... 置于另一个分支名后来对该分支的最新提交与两个分支的共同祖先进行比较：

$ git diff master...contrib

该命令仅会显示自当前主题分支与 master 分支的共同祖先起，该分支中的工作。 这个语法很有用，应该牢记。

### 将贡献的工作整合进来

当主题分支中所有的工作都已经准备好整合进入更靠近主线的分支时，接下来的问题就是如何进行整合了。 此外，还有一个问题是，你想使用怎样的总体工作流来维护你的项目？ 你的选择有很多，我们会介绍其中的一部分。

#### 合并工作流

一种基本的工作流就是将所有的工作直接合并到 master 分支。 在这种情况下，master 分支包含的代码是基本稳定的。 当你完成某个主题分支的工作，或审核通过了其他人所贡献的工作时，你会将其合并进入 master 分支，之后将主题分支删除，如此反复。

举例来说，如果我们的版本库包含类似 [包含若干主题分支的提交历史。](https://git-scm.com/book/zh/v2/ch00/merwf_a) 的两个名称分别为 ruby\_client 和 php\_client 的分支， 并且我们合并完 ruby\_client 分支后，再合并 php\_client 分支，那么提交历史最后会变成 [合并主题分支之后。](https://git-scm.com/book/zh/v2/ch00/merwf_b) 的样子。



Figure 73. 包含若干主题分支的提交历史。



Figure 74. 合并主题分支之后。

这也许是最简单的工作流了，但是当项目更大，或更稳定，你对自己所引入的工作更加在意时，它可能会带来问题。

如果你的项目非常重要，你可能会使用两阶段合并循环。 在这种情况下，你会维护两个长期分支，分别是 master 和 develop，master 分支只会在一个非常稳定的版本发布时才会更新，而所有的新代码会首先整合进入 develop 分支。 你定期将这两个分支推送到公共版本库中。 每次需要合并新的主题分支时（[合并主题分支前。](https://git-scm.com/book/zh/v2/ch00/merwf_c)），你都应该合并进入 develop 分支（[合并主题分支后。](https://git-scm.com/book/zh/v2/ch00/merwf_d)）；当打标签发布的时候，你会将 master 分支快进到已经稳定的 develop 分支（[一次发布之后。](https://git-scm.com/book/zh/v2/ch00/merwf_e)）。



Figure 75. 合并主题分支前。



Figure 76. 合并主题分支后。



Figure 77. 一次发布之后。

这样当人们克隆你项目的版本库后，既可以检出 master 分支以构建最新的稳定版本并保持更新， 也可以检出包含更多前沿内容 develop 分支。 你也可以扩展这个概念，维护一个将所有工作合并到一起的整合分支。 当该分支的代码稳定并通过测试之后，将其合并进入 develop 分支； 经过一段时间，确认其稳定之后，将其以快进的形式并入 master 分支。

#### 大项目合并工作流

Git 项目包含四个长期分支：master、next，用于新工作的 pu（proposed updates）和用于维护性向后移植工作（maintenance backports）的 maint 分支。 贡献者的新工作会以类似之前所介绍的方式收入主题分支中（见 [管理复杂的一系列接收贡献的平行主题分支。](https://git-scm.com/book/zh/v2/ch00/merwf_f)）。 之后对主题分支进行测试评估，检查其是否已经能够合并，或者仍需要更多工作。 安全的主题分支会被合并入 next 分支，之后该分支会被推送使得所有人都可以尝试整合到一起的特性。



Figure 78. 管理复杂的一系列接收贡献的平行主题分支。

如果主题分支需要更多工作，它则会被并入 pu 分支。 当它们完全稳定之后，会被再次并入 master 分支。 这意味着 master 分支始终在进行快进，next 分支偶尔会被变基，而 pu 分支的变基比较频繁：



Figure 79. 将贡献的主题分支并入长期整合分支。

当主题分支最终被并入 master 分支后，便会被从版本库中删除掉。 Git 项目还有一个从上一次发布中派生出来的 maint 分支来提供向后移植过来的补丁以供发布维护更新。 因此，当你克隆 Git 的版本库之后，就会有四个可分别评估该项目开发的不同阶段的可检出的分支， 检出哪个分支，取决于你需要多新的版本，或者你想要如何进行贡献； 对于维护者来说，这套结构化的工作流能帮助它们审查新的贡献。 Git 项目的工作流是特别的。要清晰地理解它，请阅读 [Git 维护者手册](https://github.com/git/git/blob/master/Documentation/howto/maintain-git.txt)。

#### 变基与拣选工作流

为了保持线性的提交历史，有些维护者更喜欢在 master 分支上对贡献过来的工作进行变基和拣选，而不是直接将其合并。 当你完成了某个主题分支中的工作，并且决定要将其整合的时候，你可以在该分支中运行变基命令， 在当前 master 分支（或者是 develop 等分支）的基础上重新构造修改。 如果结果理想的话，你可以快进 master 分支，最后得到一个线性的项目提交历史。

另一种将引入的工作转移到其他分支的方法是拣选。 Git 中的拣选类似于对特定的某次提交的变基。 它会提取该提交的补丁，之后尝试将其重新应用到当前分支上。 这种方式在你只想引入主题分支中的某个提交，或者主题分支中只有一个提交，而你不想运行变基时很有用。 举个例子，假设你的项目提交历史类似：



Figure 80. 拣选之前的示例历史。

如果你希望将提交 e43a6 拉取到 master 分支，你可以运行：

$ git cherry-pick e43a6

Finished one cherry-pick.

[master]: created a0a41a9: "More friendly message when locking the index fails."

3 files changed, 17 insertions(+), 3 deletions(-)

这样会拉取和 e43a6 相同的更改，但是因为应用的日期不同，你会得到一个新的提交 SHA-1 值。 现在你的历史会变成这样：



Figure 81. 拣选主题分支中的一个提交后的历史。

现在你可以删除这个主题分支，并丢弃不想拉入的提交。

#### Rerere

如果你在进行大量的合并或变基，或维护一个长期的主题分支，Git 提供的一个叫做“rerere”的功能会有一些帮助。

Rerere 是“重用已记录的冲突解决方案（reuse recorded resolution）”的意思——它是一种简化冲突解决的方法。 当启用 rerere 时，Git 将会维护一些成功合并之前和之后的镜像，当 Git 发现之前已经修复过类似的冲突时， 便会使用之前的修复方案，而不需要你的干预。

这个功能包含两个部分：一个配置选项和一个命令。 其中的配置选项是 rerere.enabled，把它放在全局配置中就可以了：

$ git config --global rerere.enabled true

现在每当你进行一次需要解决冲突的合并时，解决方案都会被记录在缓存中，以备之后使用。

如果你需要和 rerere 的缓存交互，你可以使用 git rerere 命令。 当单独调用它时，Git 会检查解决方案数据库，尝试寻找一个和当前任一冲突相关的匹配项并解决冲突 （尽管当 rerere.enabled 被设置为 true 时会自动进行）。 它也有若干子命令，可用来查看记录项，删除特定解决方案和清除缓存全部内容等。 我们将在 [Rerere](https://git-scm.com/book/zh/v2/ch00/ef_rerere) 中详细探讨。

### 为发布打标签

当你决定进行一次发布时，你可能想要打一个标签，这样在之后的任何一个提交点都可以重新创建该发布。 你在 [Git 基础](https://git-scm.com/book/zh/v2/ch00/ch02-git-basics-chapter) 中已经了解了创建新标签的过程。 作为一个维护者，如果你决定要为标签签名的话，打标签的过程应该是这样子的：

$ git tag -s v1.5 -m 'my signed 1.5 tag'

You need a passphrase to unlock the secret key for

user: "Scott Chacon <schacon@gmail.com>"

1024-bit DSA key, ID F721C45A, created 2009-02-09

如果你为标签签名了，你可能会遇到分发用来签名的 PGP 公钥的问题。 Git 项目的维护者已经解决了这一问题，其方法是在版本库中以 blob 对象的形式包含他们的公钥，并添加一个直接指向该内容的标签。 要完成这一任务，首先你可以通过运行 gpg --list-keys 找出你所想要的 key：

$ gpg --list-keys

/Users/schacon/.gnupg/pubring.gpg

---------------------------------

pub 1024D/F721C45A 2009-02-09 [expires: 2010-02-09]

uid Scott Chacon <schacon@gmail.com>

sub 2048g/45D02282 2009-02-09 [expires: 2010-02-09]

之后你可以通过导出 key 并通过管道传递给 git hash-object 来直接将 key 导入到 Git 的数据库中，git hash-object 命令会向 Git 中写入一个包含其内容的新 blob 对象，并向你返回该 blob 对象的 SHA-1 值：

$ gpg -a --export F721C45A | git hash-object -w --stdin

659ef797d181633c87ec71ac3f9ba29fe5775b92

既然 Git 中已经包含你的 key 的内容了，你就可以通过指定由 hash-object 命令给出的新 SHA-1 值来创建一个直接指向它的标签：

$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92

如果你运行 git push --tags 命令，那么 maintainer-pgp-pub 标签将会被共享给所有人。 需要校验标签的人可以通过从数据库中直接拉取 blob 对象并导入到 GPG 中来导入 PGP key：

$ git show maintainer-pgp-pub | gpg --import

人们可以使用这个 key 来校验所有由你签名的标签。 另外，如果你在标签信息中包含了一些操作说明，用户可以通过运行 git show <tag> 来获取更多关于标签校验的说明。

### 生成一个构建号

Git 中不存在随每次提交递增的“v123”之类的数字序列，如果你想要为提交附上一个可读的名称， 可以对其运行 git describe 命令。作为回应，Git 将会生成一个字符串， 它由最近的标签名、自该标签之后的提交数目和你所描述的提交的部分 SHA-1 值（前缀的 g 表示 Git）构成：

$ git describe master

v1.6.2-rc1-20-g8c5b85c

这样你在导出一个快照或构建时，可以给出一个便于人们理解的命名。 实际上，如果你的 Git 是从 Git 自己的版本库克隆下来并构建的，那么 git --version 命令给出的结果是与此类似的。 如果你所描述的提交自身就有一个标签，那么它将只会输出标签名，没有后面两项信息。

默认情况下， git describe 命令需要有注解的标签（即使用 -a 或 -s 选项创建的标签）； 如果你想使用轻量标签（无注解的标签），请在命令后添加 --tags 选项。 你也可以使用这个字符串来调用 git checkout 或 git show 命令， 但是这依赖于其末尾的简短 SHA-1 值，因此不一定一直有效。 比如，最近 Linux 内核为了保证 SHA-1 值对象的唯一性，将其位数由 8 位扩展到了 10 位， 导致以前的 git describe 输出全部失效。

### 准备一次发布

现在你可以发布一个构建了。 其中一件事情就是为那些不使用 Git 的可怜包们创建一个最新的快照归档。 使用 git archive 命令完成此工作：

$ git archive master --prefix='project/' | gzip > `git describe master`.tar.gz

$ ls \*.tar.gz

v1.6.2-rc1-20-g8c5b85c.tar.gz

如果有人将这个压缩包解压，他就可以在一个 project 目录中得到你项目的最新快照。 你也可以以类似的方式创建一个 zip 压缩包，但此时你应该向 git archive 命令传递 --format=zip 选项：

$ git archive master --prefix='project/' --format=zip > `git describe master`.zip

现在你有了本次发布的一个 tar 包和一个 zip 包，可以将其上传到网站或以电子邮件的形式发送给人们。

### 制作提交简报

现在是时候通知邮件列表里那些好奇你的项目发生了什么的人了。 使用 git shortlog 命令可以快速生成一份包含从上次发布之后项目新增内容的修改日志（changelog）类文档。 它会对你给定范围内的所有提交进行总结；比如，你的上一次发布名称是 v1.0.1，那么下面的命令可以给出上次发布以来所有提交的总结：

$ git shortlog --no-merges master --not v1.0.1

Chris Wanstrath (6):

Add support for annotated tags to Grit::Tag

Add packed-refs annotated tag support.

Add Grit::Commit#to\_patch

Update version and History.txt

Remove stray `puts`

Make ls\_tree ignore nils

Tom Preston-Werner (4):

fix dates in history

dynamic version method

Version bump to 1.0.2

Regenerated gemspec for version 1.0.2

这份整洁的总结包括了自 v1.0.1 以来的所有提交，并且已经按照作者分好组，你可以通过电子邮件将其直接发送到列表中。

# GitHub

## GitHub - 账户的创建和配置

### 什么是Github

GitHub是一个面向开源及私有软件项目的托管平台。 大部分 Git 版本库都托管在 GitHub，很多开源项目使用 GitHub 实现 Git 托管。GitHub 为免费账户提供了几乎所有的功能，除了一些高级的特性。

### 账户的创建和配置

你所需要做的第一件事是创建一个免费账户。 直接访问 [https://github.com](https://github.com/)，选择一个未被占用的用户名，提供一个电子邮件地址和密码，点击写着“Sign up for GitHub”的绿色大按钮即可。



### SSH 访问

现在，你完全可以使用 https:// 协议，通过你刚刚创建的用户名和密码访问 Git 版本库。 但是，如果仅仅克隆公有项目，你甚至不需要注册——刚刚我们创建的账户是为了以后 fork 其它项目，以及推送我们自己的修改。

如果你习惯使用 SSH 远程，你需要配置一个公钥。 （如果你没有公钥，参考 [生成 SSH 公钥](https://git-scm.com/book/zh/v2/ch00/_generate_ssh_key)。） 使用窗口右上角的链接打开你的账户设置：



Figure 83. “Account settings”链接。

然后在左侧选择“SSH keys”部分。



Figure 84. “SSH keys”链接。

在这个页面点击“Add an SSH key”按钮，给你的公钥起一个名字，将你的 ~/.ssh/id\_rsa.pub （或者自定义的其它名字）公钥文件的内容粘贴到文本区，然后点击“Add key”。

|  |  |
| --- | --- |
| **Note** | 确保给你的 SSH 密钥起一个能够记得住的名字。 你可以为每一个密钥起名字（例如，“我的笔记本电脑”或者“工作账户”等），以便以后需要吊销密钥时能够方便地区分。 |

### 邮件地址

GitHub 使用用户邮件地址区分 Git 提交。 如果你在自己的提交中使用了多个邮件地址，希望 GitHub 可以正确地将它们连接起来， 你需要在管理页面的 Emails 部分添加你拥有的所有邮箱地址。



Figure 87. 添加邮件地址

在 [添加邮件地址](https://git-scm.com/book/zh/v2/ch00/_add_email_addresses) 中我们可以看到一些不同的状态。 顶部的地址是通过验证的，并且被设置为主要地址，这意味着该地址会接收到所有的通知和回复。 第二个地址是通过验证的，如果愿意的话，可以将其设置为主要地址。 最后一个地址是未通过验证的，这意味着你不能将其设置为主要地址。 当 GitHub 发现任意版本库中的任意提交信息包含了这些地址，它就会将其链接到你的账户。

### 两步验证

最后，为了额外的安全性，你绝对应当设置两步验证，简写为 “2FA”。 两步验证是一种用于降低因你的密码被盗而带来的账户风险的验证机制，现在已经变得越来越流行。 开启两步验证，GitHub 会要求你用两种不同的验证方法，这样，即使其中一个被攻破，攻击者也不能访问你的账户。

你可以在 Account settings 页面的 Security 标签页中找到 Two-factor Authentication 设置。



Figure 88. Security 标签页中的 2FA

点击“Set up two-factor authentication”按钮，会跳转到设置页面。该页面允许你选择是要在登录时使用手机 app 生成辅助码（一种“基于时间的一次性密码”），还是要 GitHub 通过 SMS 发送辅助码。

选择合适的方法后，按照提示步骤设置 2FA，你的账户会变得更安全，每次登录 GitHub 时都需要提供除密码以外的辅助码。

# 6.2 GitHub - 对项目做出贡献

## 对项目做出贡献

账户已经建立好了，现在我们来了解一些能帮助你对现有的项目做出贡献的知识。

### 派生项目

如果你想要参与某个项目，但是并没有推送权限，这时可以对这个项目进行“派生（Fork）”。 当你“派生”一个项目时，GitHub 会在你的空间中创建一个完全属于你的项目副本，且你对其具有推送权限。

|  |  |
| --- | --- |
| **Note** | 在以前，“fork”是一个贬义词，指的是某个人使开源项目向不同的方向发展，或者创建一个竞争项目，使得原项目的贡献者分裂。 在 GitHub，“fork”指的是你自己的空间中创建的项目副本，这个副本允许你以一种更开放的方式对其进行修改。 |

通过这种方式，项目的管理者不再需要忙着把用户添加到贡献者列表并给予他们推送权限。 人们可以派生这个项目，将修改推送到派生出的项目副本中，并通过创建拉取请求（Pull Request，简称 PR）来让他们的改动进入源版本库，下文我们会详细说明。 创建了拉取请求后，就会开启一个可供审查代码的板块，项目的拥有者和贡献者可以在此讨论相关修改，直到项目拥有者对其感到满意，并且认为这些修改可以被合并到版本库。

你可以通过点击项目页面右上角的“Fork”按钮，来派生这个项目。

Figure 89. “Fork”按钮

稍等片刻，你将被转到新项目页面，该项目包含可写的代码副本。

### GitHub 流程

GitHub 设计了一个以拉取请求为中心的特殊合作流程。 它基于我们在 [主题分支](https://git-scm.com/book/zh/v2/ch00/_topic_branch) 的 [Git 分支](https://git-scm.com/book/zh/v2/ch00/ch03-git-branching) 中提到的工作流程。 不管你是在一个紧密的团队中使用单独的版本库，或者使用许多的“Fork”来为一个由陌生人组成的国际企业或网络做出贡献，这种合作流程都能应付。

流程通常如下：

1. 派生一个项目
2. 从 master 分支创建一个新分支
3. 提交一些修改来改进项目
4. 将这个分支推送到 GitHub 上
5. 创建一个拉取请求
6. 讨论，根据实际情况继续修改
7. 项目的拥有者合并或关闭你的拉取请求
8. 将更新后的 master 分支同步到你的派生中

这基本和 [集成管理者工作流](https://git-scm.com/book/zh/v2/ch00/_integration_manager) 中的一体化管理流程差不多，但是团队可以使用 GitHub 提供的网页工具替代电子邮件来交流和审查修改。

# GitHub - 维护项目

## 维护项目

现在我们可以很方便地向一个项目贡献内容，来看一下另一个方面的内容：创建、维护和管理你自己的项目。

### 创建新的版本库

让我们创建一个版本库来分享我们的项目。 通过点击面板右侧的“New repository”按钮，或者顶部工具条你用户名旁边的 + 按钮来开始我们的旅程。

### 添加合作者

如果你想与他人合作，并想给他们提交的权限，你需要把他们添加为 “Collaborators”。 如果 Ben，Jeff，Louise 都在 GitHub 上注册了，你想给他们推送的权限，你可以将他们添加到你的项目。 这样做会给他们 “推送” 权限，就是说他们对项目和 Git 版本库都有读写的权限。

### 管理合并请求

现在你有一个包含一些代码的项目，可能还有几个有推送权限的合作者，下面来看当你收到合并请求时该做什么。

合并请求可以来自仓库副本的一个分支，或者同一仓库的另一个分支。 唯一的区别是 fork 过来的通常是和你不能互相推送的人，而内部的推送通常都可以互相访问。

作为例子，假设你是 “tonychacon” ，你创建了一个名为 “fade” 的 Arduino 项目.

#### 邮件通知

有人来修改了你的代码，给你发了一个合并请求。 你会收一封关于合并请求的提醒邮件，它看起来像 [新的合并请求的邮件通知.](https://git-scm.com/book/zh/v2/ch00/_email_pr)。

#### 在合并请求上进行合作

就像我们在 [GitHub 流程](https://git-scm.com/book/zh/v2/ch00/_github_flow) 中说过的，现在你可以跟开启合并请求的人进行会话。 你既可以对某些代码发表评论，也可以对整个提交或整个合并请求发表评论， 在任何地方都可以用 GitHub 风格的 Markdown。

每次有人在合并请求上发表了评论，你都会收到邮件，通知你哪里发生了改变。邮件里面包含一个链接，指向改变的位置，你可以直接在邮件中回复，相当于在合并请求上发表评论。

#### 合并请求引用

如果你正在处理 **许多** 合并请求，不想添加一堆 remote 或者每次都要做一次拉取，这里有一个可以在 GitHub 上用的小技巧。 这是有点高级的技巧，但它相当有用，我们会在 [引用规范](https://git-scm.com/book/zh/v2/ch00/_refspec) 有更多的细节说明。

实际上 GitHub 在服务器上把合并请求分支视为一种 “假分支”。 默认情况下你克隆时不会得到它们，但它们还是隐式地存在，你可以很容易地访问到它们。

为了展示这个，我们要用到一个叫做 ls-remote 的低级命令（通常被叫做“plumbing”， 我们会在 [底层命令与上层命令](https://git-scm.com/book/zh/v2/ch00/_plumbing_porcelain) 读到更多相关内容）。 这个命令在日常 Git 操作中基本不会用到，但在显示服务器上有哪些引用（reference）时很管用。

如果在我们之前用过的 “blink” 版本库上使用这个命令，我们会得到一个版本库里所有的分支，标签和其它引用（reference）的列表。

$ git ls-remote https://github.com/schacon/blink

10d539600d86723087810ec636870a504f4fee4d HEAD

10d539600d86723087810ec636870a504f4fee4d refs/heads/master

6a83107c62950be9453aac297bb0193fd743cd6e refs/pull/1/head

afe83c2d1a70674c9505cc1d8b7d380d5e076ed3 refs/pull/1/merge

3c8d735ee16296c242be7a9742ebfbc2665adec1 refs/pull/2/head

15c9f4f80973a2758462ab2066b6ad9fe8dcf03d refs/pull/2/merge

a5a7751a33b7e86c5e9bb07b26001bb17d775d1a refs/pull/4/head

31a45fc257e8433c8d8804e3e848cf61c9d3166c refs/pull/4/merge

当然，如果你在你自己的版本库或其它你想检查的远程版本库中使用 git ls-remote origin ，它会显示相似的内容。

如果版本库在 GitHub 上并且有打开的合并请求，你会得到一些以 refs/pull/ 开头的引用。 它们实际上是分支，但因为它们不在 refs/heads/ 中，所以正常情况下你克隆时不会从服务器上得到它们 ——抓取过程正常情况下会忽略它们。

每个合并请求有两个引用——其中以 /head 结尾的引用指向的提交记录与合并请求分支中的最后一个提交记录是同一个。 所以如果有人在我们的版本库中开启了一个合并请求，他们的分支叫做 bug-fix， 指向 a5a775 这个提交记录，那么在 **我们的** 版本库中我们没有 bug-fix 分支（因为那是在他们的 fork 中）， 但我们 **可以** 有一个 pull/<pr#>/head 指向 a5a775。 这意味着我们可以很容易地拉取每一个合并请求分支而不用添加一堆远程仓库。

现在，你可以像直接抓取引用一样抓取那些分支或提交。

$ git fetch origin refs/pull/958/head

From https://github.com/libgit2/libgit2

\* branch refs/pull/958/head -> FETCH\_HEAD

这告诉 Git： “连接到 origin 这个 remote，下载名字为 refs/pull/958/head 的引用。” Git 高高兴兴去执行，下载构建那个引用需要的所有内容，然后把指针指向 .git/FETCH\_HEAD 下面你想要的提交记录。 然后你可以用 git merge FETCH\_HEAD 把它合并到你想进行测试的分支，但那个合并的提交信息看起来有点怪。 然而，如果你需要审查 **一大批** 合并请求，这样操作会很麻烦。

还有一种方法可以抓取 **所有的** 合并请求，并且在你连接到远程仓库的时候保持更新。 用你最喜欢的编辑器打开 .git/config ，查找 origin 远程仓库。 看起来差不多像下面这样：

[remote "origin"]

url = https://github.com/libgit2/libgit2

fetch = +refs/heads/\*:refs/remotes/origin/\*

以 fetch = 开头的行是一个 “refspec.” 它是一种把 remote 的名称映射到你本地 .git 目录的方法。 这一条（就是上面的这一条）告诉 Git，“remote 上 refs/heads 下面的内容在我本地版本库中都放在 refs/remotes/origin 。” 你可以把这一段修改一下，添加另一个 refspec：

[remote "origin"]

url = https://github.com/libgit2/libgit2.git

fetch = +refs/heads/\*:refs/remotes/origin/\*

fetch = +refs/pull/\*/head:refs/remotes/origin/pr/\*

最后一行告诉 Git： “所有看起来像 refs/pull/123/head 的引用应该在本地版本库像 refs/remotes/origin/pr/123 一样存储” 现在，如果你保存那个文件，执行 git fetch：

$ git fetch

# …

\* [new ref] refs/pull/1/head -> origin/pr/1

\* [new ref] refs/pull/2/head -> origin/pr/2

\* [new ref] refs/pull/4/head -> origin/pr/4

# …

现在所有的合并请求在本地像分支一样展现，它们是只读的，当你执行抓取时它们也会更新。 这让在本地测试合并请求中的代码变得超级简单：

$ git checkout pr/2

Checking out files: 100% (3769/3769)， done.

Branch pr/2 set up to track remote branch pr/2 from origin.

Switched to a new branch 'pr/2'

你的鹰眼系统会发现在 refspec 的 remote 部分的结尾有个 head 。 在 GitHub 那边也有一个 refs/pull/#/merge 引用，它代表的是如果你在网站上按了 “merge” 按钮对应的提交记录。 这甚至让你可以在按按钮之前就测试这个合并。

#### 合并请求之上的合并请求

你不仅可以在主分支或者说 master 分支上开启合并请求，实际上你可以在网络上的任何一个分支上开启合并请求。 其实，你甚至可以在另一个合并请求上开启一个合并请求。

如果你看到一个合并请求在向正确的方向发展，然后你想在这个合并请求上做一些修改或者你不太确定这是个好主意，或者你没有目标分支的推送权限，你可以直接在合并请求上开启一个合并请求。

当你开启一个合并请求时，在页面的顶端有一个框框显示你要合并到哪个分支和你从哪个分支合并过来的。 如果你点击那个框框右边的 “Edit” 按钮，你不仅可以改变分支，还可以选择哪个 fork。

### 网页通知

网页通知只在 GitHub 上存在，你也只能在 GitHub 上查看。 如果你打开了这个选项并且有一个你的通知，你会在你屏幕上方的通知图标上看到一个小蓝点。

### 邮件通知

邮件通知是你处理 GitHub 通知的另一种方式。 如果你打开这个选项，每当有通知时，你会收到一封邮件。 我们在 [通过电子邮件发送的评论提醒](https://git-scm.com/book/zh/v2/ch00/_email_notification) 和 [新的合并请求的邮件通知.](https://git-scm.com/book/zh/v2/ch00/_email_pr) 看到了一些例子。 邮件也会被合适地按话题组织在一起，如果你使用一个具有会话功能的邮件客户端那会很方便。

GitHub 在发送给你的邮件头中附带了很多元数据，这对于设置过滤器和邮件规则非常有帮助。

举个例子，我们来看一看在 [新的合并请求的邮件通知.](https://git-scm.com/book/zh/v2/ch00/_email_pr) 中发给 Tony 的一封真实邮件的头部，我们会看到下面这些：

To: tonychacon/fade <fade@noreply.github.com>

Message-ID: <tonychacon/fade/pull/1@github.com>

Subject: [fade] Wait longer to see the dimming effect better (#1)

X-GitHub-Recipient: tonychacon

List-ID: tonychacon/fade <fade.tonychacon.github.com>

List-Archive: https://github.com/tonychacon/fade

List-Post: <mailto:reply+i-4XXX@reply.github.com>

List-Unsubscribe: <mailto:unsub+i-XXX@reply.github.com>，...

X-GitHub-Recipient-Address: tchacon@example.com

这里有一些有趣的东西。如果你想高亮或者转发这个项目甚至这个合并请求相关的邮件， Message-ID 中的信息会以`<user>/<project>/<type>/<id>` 的格式展现所有的数据。 例如，如果这是一个问题（issue），那么 <type> 字段就会是 “issues” 而不是 “pull” 。

List-Post 和 List-Unsubscribe 字段表示如果你的邮件客户端能够处理这些，那么你可以很容易地在列表中发贴或取消对这个相关帖子的订阅。 那会很有效率，就像在页面中点击静音按钮或在问题/合并请求页面点击 “Unsubscribe” 一样。

值得注意的是，如果你同时打开了邮件和网页通知，那么当你在邮件客户端允许加载图片的情况下阅读邮件通知时，对应的网页通知也将会同时被标记为已读。

### 特殊文件

如果你的版本库中有一些特殊文件，GitHub 会提醒你。

### README

第一个就是 README 文件，可以是几乎任何 GitHub 可以识别的格式。 例如，它可以是 README ，README.md ， README.asciidoc 。 如果 GitHub 在你的版本库中找到 README 文件，会把它在项目的首页渲染出来。

很多团队在这个文件里放版本库或项目新人需要了解的所有相关的信息。 它一般包含这些内容：

* 该项目的作用
* 如何配置与安装
* 有关如何使用和运行的例子
* 项目的许可证
* 如何向项目贡献力量

因为 GitHub 会渲染这个文件，你可以在文件里植入图片或链接让它更容易理解。

### 贡献 CONTRIBUTING

另一个 GitHub 可以识别的特殊文件是 CONTRIBUTING 。 如果你有一个任意扩展名的 CONTRIBUTING 文件，当有人开启一个合并请求时 GitHub 会显示 [开启合并请求时有 CONTRIBUTING 文件存在.](https://git-scm.com/book/zh/v2/ch00/_contrib_file)。

### 项目管理

对于一个单个项目其实没有很多管理事务要做，但也有几点有趣的。

#### 改变默认分支

如果你想用 “master” 之外的分支作为你的默认分支，其他人将默认会在这个分支上开启合并请求或进行浏览，你可以在你版本库的设置页面的 "options" 标签下修改。

#### 移交项目

如果你想把一个项目移交给 GitHub 中的另一个人或另一个组织，还是设置页面的这个 “options” 标签下有一个 “Transfer ownership” 选项可以用来干这个。

# GitHub - 管理组织

## 管理组织

除了个人帐户之外，GitHub 还提供被称为组织（Organizations）的帐户。 组织账户和个人账户一样都有一个用于存放所拥有项目的命名空间，但是许多其他的东西都是不同的。 组织帐户代表了一组共同拥有多个项目的人，同时也提供一些工具用于对成员进行分组管理。 通常，这种账户被用于开源群组（例如：“perl”或者“rails”），或者公司（例如：“google”或者“twitter”）。

### 组织的基本知识

我们可以很简单地创建一个组织，只需要点击任意 GitHub 页面右上角的“+”图标，在菜单中选择“New organization”即可。

### 团队

组织使用团队（Teams）来管理成员，团队就是组织中的一组个人账户和版本库，以及团队成员对这些版本库的访问权限。

## Git的使用步骤

1. 注册Github账号
2. 在Github上创建仓库
3. 在本地安装Git、初始化和配置Git
4. 将文件提交到github中

# 初始化和配置Git

## 1.初始化git

语法：git init;

说明:初始化git

## 2.配置Git账号信息(名称和邮箱)

1.语法：git config user.name "名称"

   说明：配置用户名

   实例：git config user.name "wangtao-luse"

 2.语法：git config user.email "邮箱"

   说明：配置邮箱

   实例：git config user.email "wangtao@163.com"

2.查看配置的信息

语法：git cofig user.name

说明：查看名称

语法：git config user.email

说明：查看邮箱;

# Git添加和提交文件

Git分为工作区，缓存区，版本库;