

GIT 学习手册

笔记本： git

创建时间： 2015/8/24 14:20

更新时间： 2015/9/10 9:51

1. 获取与创建项目

你得先有一个 Git 仓库，才能用它进行操作。仓库是 Git 存放你要保存的快照的数据的地方。

拥有一个 Git 仓库的途径有两种。在已有的目录中，初始化一个新的，其一。比如一个新的项目，或者一个已存在的项目，但该项目尚未有版本控制。如果你想要复制一份别人的项目，或者与别人合作某个项目，也可以从一个公开的 Git 仓库克隆，其二。本章将对两者都做介绍。

Git使用前配置

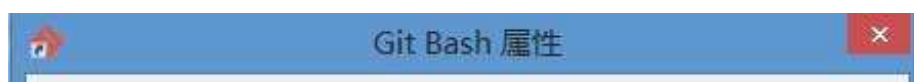
如果设置了，在输入命令示界面可以很方便的使用复制和粘贴（用左键选取要复制的，点右键直接就可以复制，粘贴时只需点一下右键。）设置方法：Git Bash快捷图标（桌面图标）右键属性-选项，把快速编辑模式勾上就可以，如下图：

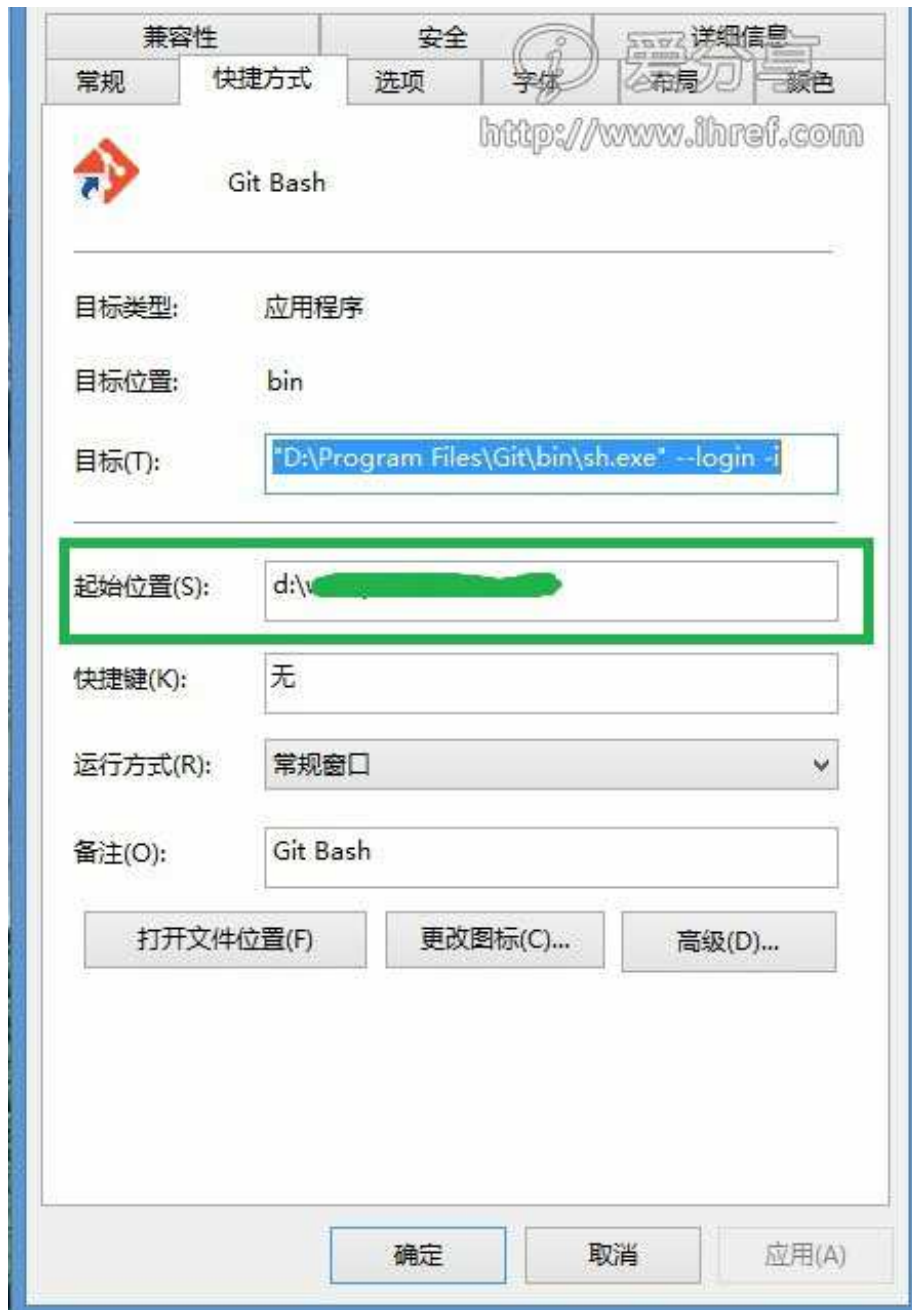




设置Git本地项目开发库默认路径

如果设置了，就不用每次打开Git再cd打开目录了。方法：右键Git Bash快捷图标（桌面图标）属性，找到快捷方式-起始位置，把你的项目地址放在这里就可以了。如下图：





配置本地用户和邮箱

用户名邮箱作用：我们需要设置一个用户名 和 邮箱, 这是用来上传本地仓库到GitHub中, 在GitHub中显示代码上传者;

使用命令：

```
git config --global user.name "HanShuliang" //设置用户名
```

```
git config --global user.email "13241153187@163.com" //设置邮箱
```

```
Administrator@XRDPTJ9ILK6IWRA ~  
$ git config --global user.name "HanShuliang"  
Administrator@XRDPTJ9ILK6IWRA ~  
$ git config --global user.email "13241153187@163.com"
```

到此Git客户端已安装及GitHub配置完成，现在可以从GitHub传输代码了。

git init 将一个目录初始化为 Git 仓库

在目录中执行 `git init`，就可以创建一个 Git 仓库了。比如，我们恰好有个目录，里头有些许文件，如下：

```
$ cd konnichiwa$ ls  
README  hello.rb
```

在这个项目里头，我们会用各种编程语言写 “Hello World” 实例。到目前为止，我们只有 Ruby 的，不过，这才刚上路嘛。为了开始用 Git 对这个项目作版本控制，我们执行一下 `git init`。

```
$ git init  
Initialized empty Git repository in /opt/konnichiwa/.git/  
# 在 /opt/konnichiwa/.git 目录初始化空 Git 仓库完毕。
```

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

```
$ ls -a  
.      ..      .git    README  hello.rb
```

恭喜，现在你就有了一个 Git 仓库的架子，可以开始快照你的项目了。

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

git clone 复制一个 Git 仓库，以上下其手

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

想复制而又与别人合作。一个项目，项目名又复杂。一个项目，管理代码，代码与别人合作。项目。执行 `git clone [url]`，[url] 为你想要复制的项目，就可以了。

```
$ git clone git://github.com/schacon/simplegit.git
Initialized empty Git repository in /private/tmp/simplegit/.git/
remote: Counting objects: 100, done.
remote: Compressing objects: 100% (86/86), done.
remote: Total 100 (delta 35), reused 0 (delta 0)
Receiving objects: 100% (100/100), 9.51 KiB, done.
Resolving deltas: 100% (35/35), done.
$ cd simplegit/$ ls
README  Rakefile lib
```

上述操作将复制该项目的全部记录，让你本地拥有这些。并且该操作将拷贝该项目的主分支，使你能够查看代码，或编辑、修改。进到该目录中，你会看到 `.git` 子目录。所有的项目数据都存在那里。

```
$ ls -a
.      ..      .git      README  Rakefile lib
HEAD   description info      packed-refs
branches  hooks      logs      refs
config    index      objects
```

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

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

2. 基本快照

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

这里有个重要的概念，Git 有一个叫做“索引”的东东，有点像是你的快照的缓存区。这就使你能够从更改的文件中创建出一系列组织良好的快照，而不是一次提交所有的更改。

简而言之，使用 `git add` 添加需要追踪的新文件和待提交的更改，然后使用 `git status` 和

`git diff` 查看有何改动，最后用 `git commit` 将你的快照记录。这就是你要用的基本流程，绝大部分时候都是这样的。

git add 添加文件到缓存

在 Git 中，在提交你修改的文件之前，你需要把它们添加到缓存。如果该文件是新创建的，你可以执行 `git add` 将该文件添加到缓存，但是，即使该文件已经被追踪了——也就是说，曾经提交过了——你仍然需要执行 `git add` 将新更改的文件添加到缓存去。让我们看几个例子：

回到我们的 Hello World 示例，初始化该项目之后，我们就要用 `git add` 将我们的文件添加进去了。我们可以用 `git status` 看看我们的项目的当前状态。

```
$ git status -s?? README
?? hello.rb
```

我们有俩尚未被追踪的文件，得添加一下。

```
$ git add README hello.rb
```

现在再执行 `git status`，就可以看到这两文件已经加上去了。

```
$ git status -sA README
A hello.rb
```

新项目中，添加所有文件很普遍，可以在当前工作目录执行命令：`git add .`。因为 Git 会递归地将你执行命令时所在的目录中的所有文件添加上去，所以如果你将当前的工作目录作为参数，它就会追踪那儿的所有文件了。如此，`git add .` 就和 `git add README hello.rb` 有一样的效果。此外，效果一致的还有 `git add *`，不过那只是因为我们这还没有子目录，不需要递归地添加新文件。

好了，现在我们改个文件，再跑一下 `git status`，有点古怪。

```
$ vim README$ git status -sAM README
A hello.rb
```

“AM” 状态的意思是，这个文件在我们将它添加到缓存之后又有改动。这意味着如果我们现在提交快照，我们记录的将是上次跑 `git add` 的时候的文件版本，而不是现在在磁盘中的这个。Git 并不认为磁盘中的文件与你想快照的文件必须是一致的——（如果你需要它们一致，）得用 `git add` 命令告诉它。

一言以蔽之，当你要将你的修改包含在即将提交的快照里的时候，执行 `git add`。任何你没有添加的改动都不会被包含在内——这意味着你可以比绝大多数其他源代码版本控制系统更精确地归置你的快照。

请查看《Pro Git》中 `git add` 的“-p” 参数，以了解更多关于提交文件的灵活性的例子。

git status 查看你的文件在工作目录与缓存的状态

正如你在 `git add` 小节中所看到的，你可以执行 `git status` 命令查看你的代码在缓存与当前工作目录的状态。我演示该命令的时候加了 `-s` 参数，以获得简短的结果输出。若没有这个标记，命令 `git status` 将告诉你更多的提示与上下文欣喜。以下便是同样状态下，有跟没有 `-s` 参数的输出对比。简短的输出如下：

```
$ git status -s
AM README
A  hello.rb
```

而同样的状态，详细的输出看起来是这样的：

```
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#   new file:   README
#   new file:   hello.rb
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#
```

```
└─ y /
#
# modified:   README
#
```

你很容易发现简短的输出看起来很紧凑。而详细输出则很有帮助，提示你可以用何种命令完成你接下来可能要做的事情。

Git 还会告诉你在你上次提交之后，有哪些文件被删除、修改或者存入缓存了。

```
$ git status -sM README
D hello.rb
```

你可以看到，在简短输出中，有两栏。第一栏是缓存的，第二栏则是工作目录的。所以假设你临时提交了 README 文件，然后又改了些，并且没有执行 `git add`，你会看到这个：

```
$ git status -sMM README
D hello.rb
```

一言以蔽之，执行 `git status` 以查看在你上次提交之后有啥被修改或者临时提交了，从而决定自己是否需要提交一次快照，同时也能知道有什么改变被记录进去了。

git diff 显示已写入缓存与已修改但尚未写入缓存的改动的区别

`git diff` 有两个主要的应用场景。我们将在此介绍其一，在 **检阅与对照** 一章中，我们将介绍其二。我们这里介绍的方式是用此命令描述已临时提交的或者已修改但尚未提交的改动。

git diff #尚未缓存的改动

如果没有其他参数，`git diff` 会以规范化的 diff 格式（一个补丁）显示自从你上次提交快照之后尚未缓存的所有更改。

```
$ vim hello.rb$ git status -s
M hello.rb
$ git diffdiff --git a/hello.rb b/hello.rb
index d62ac43..8d15d50 100644
--- a/hello.rb
+++ b/hello.rb@@ -1,7 +1,7 @@
class HelloWorld
```



```

def self.hello
-   puts "hello world"+   puts "hola mundo"
end

end

```

所以，`git status` 显示你上次提交更新至后所更改或者写入缓存的改动，而 `git diff` 一行一行地显示这些改动具体是啥。通常执行完 `git status` 之后接着跑一下 `git diff` 是个好习惯。

git diff --cached #查看已缓存的改动

`git diff --cached` 命令会告诉你有哪些内容已经写入缓存了。也就是说，此命令显示的是接下来要写入快照的内容。所以，如果你将上述示例中的 `hello.rb` 写入缓存，因为 `git diff` 显示的是尚未缓存的改动，所以在此执行它不会显示任何信息。

```

$ git status -s
M hello.rb
$ git add hello.rb $ git status -sM hello.rb
$ git diff$

```

如果你想看看已缓存的改动，你需要执行的是 `git diff --cached`。

```

$ git status -sM hello.rb
$ git diff$ $ git diff --cacheddiff --git a/hello.rb b/hello.rb
index d62ac43..8d15d50 100644
--- a/hello.rb
+++ b/hello.rb@@ -1,7 +1,7 @@
class HelloWorld

    def self.hello
-   puts "hello world"+   puts "hola mundo"
end

end

```

git diff HEAD 查看已缓存的与未缓存的所有改动

如果你想一并查看已缓存的与未缓存的改动，可以执行 `git diff HEAD` —— 也就是说你要看到的是工作目录与上一次提交的更新的区别，无视缓存。假设我们又改了些 `ruby.rb` 的内

容，那缓存的与未缓存的改动我们就都有了。以上三个 `diff` 命令的结果如下：

```
$ vim hello.rb $ git diffdiff --git a/hello.rb b/hello.rb
index 4f40006..2ae9ba4 100644
--- a/hello.rb
+++ b/hello.rb@@ -1,7 +1,7 @@
class HelloWorld

+ # says hello
  def self.hello
    puts "hola mundo"
  end

end

$ git diff --cacheddiff --git a/hello.rb b/hello.rb
index 2aabb6e..4f40006 100644
--- a/hello.rb
+++ b/hello.rb@@ -1,7 +1,7 @@
class HelloWorld

  def self.hello
-   puts "hello world"+   puts "hola mundo"
  end

end

$ git diff HEADdiff --git a/hello.rb b/hello.rb
index 2aabb6e..2ae9ba4 100644
--- a/hello.rb
+++ b/hello.rb@@ -1,7 +1,8 @@
class HelloWorld

+ # says hello
  def self.hello
-   puts "hello world"+   puts "hola mundo"
  end

end
```

git diff -stat 显示摘要而非整个 diff

如果我们不想要看整个 diff 输出，但是又想比 `git status` 详细点，就可以用 `--stat` 选项。该选项使它显示摘要而非全文。上文示例在使用 `--stat` 选项时，输出如下：

```
^ . . . . .
```

```
$ git status -sMM hello.rb
$ git diff --stat
hello.rb |    1 +
1 files changed, 1 insertions(+), 0 deletions(-)
$ git diff --cached --stat
hello.rb |    2 +-
1 files changed, 1 insertions(+), 1 deletions(-)
$ git diff HEAD --stat
hello.rb |    3 ++-
1 files changed, 2 insertions(+), 1 deletions(-)
```

你还可以在上述命令后面制定一个目录，从而只查看特定文件或子目录的 `diff` 输出。

简而言之，执行 `git diff` 来查看执行 `git status` 的结果的详细信息 —— 一行一行地显示这些文件是如何被修改或写入缓存的。

git commit 记录缓存内容的快照

现在你使用 `git add` 命令将想要快照的内容写入了缓存，执行 `git commit` 就将它实际存储快照了。Git 为你的每一个提交都记录你的名字与电子邮箱地址，所以第一步是告诉 Git 这些都是啥。

```
$ git config --global user.name 'Your Name'$ git config --global user.emai
l you@somedomain.com
```

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

```
$ git add hello.rb $ git status -sM hello.rb

$ git commit -m 'my hola mundo changes'
[master 68aa034] my hola mundo changes
1 files changed, 2 insertions(+), 1 deletions(-)
```

现在我们已经记录了快照。如果我们再执行 `git status`，会看到我们有一个“干净的工作目录”。这意味着我们在最近一次提交之后，没有做任何改动 —— 在我们的项目中没有未快照的工作。

```
$ git status
```

```
➤ git status
# On branch master
nothing to commit (working directory clean)
```

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

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

在此，你在文件头部添加实际的提交信息。以“#”开头的行都会被无视——Git 将 `git status` 的输出结果放在那儿以提示你都改了、缓存了啥。

通常，撰写良好的提交信息是很重要的。以开放源代码项目为例，多多少少以以下格式写你的提示消息是个不成文的规定：

简短的关于改动的总结（25个字或者更少）

如果有必要，更详细的解释文字。约 36 字时换行。在某些情况下，第一行会被作为电子邮件的开头，而剩余的则会作为邮件内容。将小结从内容隔开的空行是至关重要的（除非你没有内容）；

如果这两个待在一起，有些 git 工具会犯迷糊。

空行之后是更多的段落。

- 列表也可以
- 通常使用连字符（-）或者星号（*）来标记列表，前面有个空格，在列表项之间有空行，不过这些约定也会有些变化。

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
```

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   hello.rb
#
~
~
~
".git/COMMIT_EDITMSG" 25L, 884C written
```

提交注解是很重要的。因为 Git 很大一部分能耐就是它在组织本地提交和与他人分享的弹性，它很给力地能够让你为逻辑独立的改变写三到四条提交注解，以便你的工作被同仁审阅。因为提交与推送改动是有区别的，请务必花时间将各个逻辑独立的改动放到另外一个提交，并附上一份良好的提交注解，以使与你合作的人能够方便地了解你所做的，以及你为何要这么做。

git commit -a 自动将在提交前将已记录、修改的文件放入缓存区

如果你觉得 `git add` 提交缓存的流程太过繁琐，Git 也允许你用 `-a` 选项跳过这一步。基本上这句话的意思就是，为任何已有记录的文件执行 `git add` —— 也就是说，任何在你最近的提交中已经存在，并且之后被修改的文件。这让你能够用更 Subversion 方式的流程，修改些文件，然后想要快照所有所做的改动的时候执行 `git commit -a`。不过你仍然需要执行 `git add` 来添加新文件，就像 Subversion 一样。

```
$ vim hello.rb$ git status -s
 M hello.rb
$ git commit -m 'changes to hello file'
# On branch master
# Changed but not updated:
#
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
# modified:   hello.rb
#
no changes added to commit (use "git add" and/or "git commit -a")$ git commit -am 'changes to hello file'
[master 78b2670] changes to hello file
 1 files changed, 2 insertions(+), 1 deletions(-)
```

注意 如果你不经常改动，直接执行 `git commit`，Git 会直接给出 `git status` 命令的输出

注意，如未作任何改动，且执行了 `git commit`，Git 会显示由 `git status` 即有的输出，提醒你啥也没缓存。我已将该消息中的重要部分高亮，它说没有添加需要提交的缓存。如果你使用 `-a`，它会缓存并提交每个改动（不含新文件）。

现在你就完成了整个快照的流程——改些文件，然后用 `git add` 将要提交的改动提交到缓存，用 `git status` 和 `git diff` 看看你都改了啥，最后 `git commit` 永久地保存快照。

简而言之，执行 `git commit` 记录缓存区的快照。如果需要的话，这个快照可以用来做比较、共享以及恢复。

git reset HEAD 取消缓存已缓存的内容

`git reset` 可能是人类写的最费解的命令了。我用 Git 有些年头了，甚至还写了本书，但有的时候还是会搞不清它会做什么。所以，我只说三个明确的，通常有用的调用。请你跟我一样尽管用它——因为它可以很有用。

在此例中，我们可以用它来将不小心缓存的东东取消缓存。假设你修改了两个文件，想要将它们记录到两个不同的提交中去。你应该缓存并提交一个，再缓存并提交另外一个。如果你不小心两个都缓存了，那要如何才能取消缓存呢？你可以用 `git reset HEAD -- file`。技术上说，在这里你不需要使用 `--`——它用来告诉 Git 这时你已经不再列选项，剩下的是文件路径了。不过养成使用它分隔选项与路径的习惯很重要，即使在你可能并不需要的时候。

好，让我们看看取消缓存是什么样子的。这里我们有两个最近提交之后又有所改动的文件。我们将两个都缓存，并取消缓存其中一个。

```
$ git status -s
M README
M hello.rb
$ git add . $ git status -sM README
M hello.rb
$ git reset HEAD -- hello.rb
Unstaged changes after reset:
M hello.rb
$ git status -sM README
M hello.rb
```

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

`hello.rb`。

如果你好奇，它实际的操作是将该文件在“索引”中的校验和重置为最近一次提交中的值。`git add` 会计算一个文件的校验和，将它添加到“索引”中，而 `git reset HEAD` 将它改写回原先的，从而取消缓存操作。

如果你想直接执行 `git unstage`，你可以在 Git 中配置个别名。执行

`git config --global alias.unstage "reset HEAD"` 即可。一旦执行完它，你就可以直接用

`git unstage [file]` 作为代替了。

如果你忘了取消缓存的命令，Git 的常规 `git status` 输出的提示会很有帮助。例如，在你有已缓存的文件时，如果你不带 `-s` 执行 `git status`，它将告诉你怎样取消缓存：

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   README
#   modified:   hello.rb
#
```

简而言之，执行 `git reset HEAD` 以取消之前 `git add` 添加，但不希望包含在下一提交快照中的缓存。

git rm 将文件从缓存区移除

`git rm` 会将条目从缓存区中移除。这与 `git reset HEAD` 将条目取消缓存是有区别的。“取消缓存”的意思就是将缓存区恢复为我们做出修改之前的样子。在另一方面，`git rm` 则将该文件彻底从缓存区踢出，因此它不再下一个提交快照之内，进而有效地删除它。

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

git mv `git rm --cached orig; mv orig new; git add new`

不像绝大多数其他版本控制系统，Git 并不记录记录文件重命名。它反而只记录快照，并对

比快照以找到有啥文件可能被重命名了。如果一个文件从更新中删除了，而在下次快照中新添加的另一个文件的内容与它很相似，Git 就知道这极有可能是个重命名。因此，虽然有 `git mv` 命令，但它有点多余——它做得所有事情就是 `git rm --cached`，重命名磁盘上的文件，然后再执行 `git add` 把新文件添加到缓存区。你并不需要用它，不过如果觉得这样容易些，尽管用吧。

我自己并不使用此命令的普通形式——删除文件。通常直接从硬盘删除文件，然后执行 `git commit -a` 会简单些。它会自动将删除的文件从索引中移除。

简而言之，执行 `git rm` 来删除 Git 追踪的文件。它还会删除你的工作目录中的相应文件。

3. 分支与合并

分支是我最喜欢的 Git 特性之一。如果你用过其他版本控制系统，把你所知的分支给忘记，倒可能更有帮助些——事实上，以我们使用分支的方式，把 Git 的分支看作 上下文 反而更合适。当你检出分支时，你可以在两三个不同的分支之间来回切换。

简而言之，你可以执行 `git branch (branchname)` 来创建分支，使用 `git checkout (branchname)` 命令切换到该分支，在该分支的上下文环境中，提交快照等，之后可以很容易地来回切换。当你切换分支的时候，Git 会用该分支的最后提交的快照替换你的工作目录的内容，所以多个分支不需要多个目录。使用 `git merge` 来合并分支。你可以多次合并到统一分支，也可以选择合并之后直接删除被并入的分支。

`git branch` 列出、创建与管理工作上下文 `git checkout` 切换到新的分支上下文

`git branch` 命令是 Git 中的通用分支管理工具，可以通过它完成多项任务。我们先说你会用到的最多的命令——列出分支、创建分支和删除分支。我们还会介绍用来切换分支的 `git checkout` 命令。

`git branch` 列出可用的分支

没有参数时，`git branch` 会列出你在本地的分支。你所在的分支的行首会有个星号作标记。如果你开启了**彩色模式**，当前分支会用绿色显示。

```
$ git branch
```



```
* master
```

此例的意思就是，我们有一个叫做“master”的分支，并且该分支是当前分支。当你执行 `git init` 的时候，缺省情况下 Git 就会为你创建“master”分支。但是这名字一点特殊意味都没有——事实上你并不非得要一个叫做“master”的分支。不过由于它是缺省分支名的缘故，绝大部分项目都有这个分支。

git branch (branchname) 创建新分支

我们动手创建一个分支，并切换过去。执行 `git branch (branchname)` 即可。

```
$ git branch testing
$ git branch
* master
  testing
```

现在我们可以看到，有了一个新分支。当你以此方式在上次提交更新之后创建了新分支，如果后来又有更新提交，然后又切换到了“testing”分支，Git 将还原你的工作目录到你创建分支时候的样子——你可以把它看作一个记录你当前进度的书签。让我们实际运用看看——我们用 `git checkout (branch)` 切换到我们要修改的分支。

```
$ ls
README  hello.rb
$ echo 'test content' > test.txt
$ echo 'more content' > more.txt
$ git add *.txt
$ git commit -m 'added two files'

[master 8bd6d8b] added two files
 2 files changed, 2 insertions(+), 0 deletions(-)
 create mode 100644 more.txt
 create mode 100644 test.txt
$ ls
README  hello.rb more.txt test.txt
$ git checkout testing
Switched to branch 'testing'
$ ls
README  hello.rb
```

当我们切换到“测试”分支的时候，我们添加的新文件被移除了。切换回“master”分支的时候，它们又重新出现了。

```
✓ ls
README  hello.rb
$ git checkout master
Switched to branch 'master'
$ ls
README  hello.rb more.txt test.txt
```

git checkout -b (branchname) 创建新分支，并立即切换到它

通常情况下，你会更希望立即切换到新分支，从而在该分支中操作，然后当此分支的开发日趋稳定时，将它合并到稳定版本的分支（例如“master”）中去。执行

`git branch newbranch; git checkout newbranch` 也很简单，不过 Git 还为你提供了快捷方式：`git checkout -b newbranch`。

```
$ git branch
* master
$ ls
README  hello.rb more.txt test.txt
$ git checkout -b removals
Switched to a new branch 'removals'
$ git rm more.txt
rm 'more.txt'
$ git rm test.txt
rm 'test.txt'
$ ls
README  hello.rb
$ git commit -am 'removed useless files'

[removals 8f7c949] removed useless files
 2 files changed, 0 insertions(+), 2 deletions(-)
 delete mode 100644 more.txt
 delete mode 100644 test.txt
$ git checkout master
Switched to branch 'master'
$ ls
README  hello.rb more.txt test.txt
```

如你所见，我们创建了一个分支，在该分支的上下文中移除了一些文件，然后切换回我们的主分支，那些文件又回来了。使用分支将工作切分开来，从而让我们能够在不同上下文中做事，并来回切换。

创建新分支，在其中完成一部分工作，完成之后将它合并到主分支并删除。你会觉得这很方便，因为这么做很快很容易。如此，当你觉得这部分工作并不靠谱，舍弃它很容易。并

且，如果你必须回到稳定分支做些事情，也可以很方便地这个独立分支的工作先去在一边，完成要事之后再切换回来。

git branch -d (branchname) 删除分支

假设我们要删除一个分支（比如上例中的“testing”分支，该分支没啥特殊的内容了），可以执行 `git branch -d (branch)` 把它删掉。

```
$ git branch
* master
  testing
$ git branch -d testing
Deleted branch testing (was 78b2670).
$ git branch
* master
```

简而言之 使用 `git branch` 列出现有的分支、创建新分支以及删除不必要或者已合并的分支。

git merge 将分支合并到你的当前分支

一旦某分支有了独立内容，你终究会希望将它合并回到你的主分支。你可以使用 `git merge` 命令将任何分支合并到当前分支中去。我们那上例中的“removals”分支为例。假设我们创建了一个分支，移除了一些文件，并将它提交到该分支，其实该分支是与我们的主分支

（也就是“master”）独立开来的。要想将这些移除操作包含在主分支中，你可以将“removals”分支合并回去。

```
$ git branch
* master
  removals
$ ls
README  hello.rb  more.txt  test.txt
$ git merge removals
Updating 8bd6d8b..8f7c949
Fast-forward
 more.txt | 1 -
 test.txt | 1 -
 2 files changed, 0 insertions(+), 2 deletions(-)
 delete mode 100644 more.txt
 delete mode 100644 test.txt
$ ls README  hello.rb
```

更多复杂合并

当然，合并并不仅仅是简单的文件添加、移除的操作，Git 也会合并修改——事实上，它很会合并修改。举例，我们看看在某分支中编辑某个文件，然后在另一个分支中把它的名字改掉再做些修改，最后将这俩分支合并起来。你觉得会变成一坨 shi？我们试试看。

```
$ git branch
* master
$ cat hello.rb
class HelloWorld
  def self.hello
    puts "Hello World"
  end
end

HelloWorld.hello
```

首先，我们创建一个叫做“change_class”的分支，切换过去，从而将重命名类等操作独立出来。我们将类名从“HelloWorld”改为“HiWorld”。

```
$ git checkout -b change_class
M hello.rb

Switched to a new branch 'change_class'
$ vim hello.rb $ head -1 hello.rb
class HiWorld
$ git commit -am 'changed the class name'
[change_class 3467b0a] changed the class name
1 files changed, 2 insertions(+), 4 deletions(-)
```

然后，将重命名类操作提交到“change_class”分支中。现在，假如切换回“master”分支我们可以看到类名恢复到了我们切换到“change_class”分支之前的样子。现在，再做些修改（即代码中的输出），同时将文件名从 `hello.rb` 改为 `ruby.rb`。

```
$ git checkout master
Switched to branch 'master'
$ git mv hello.rb ruby.rb $ vim ruby.rb $ git diffdiff --git a/ruby.rb b/ruby.rb
index 2aabb6e..bf64b17 100644
--- a/ruby.rb
```

```

+++ b/ruby.rb@@ -1,7 +1,7 @@
class HelloWorld

  def self.hello
-   puts "Hello World"+   puts "Hello World from Ruby"
  end

end

$ git commit -am 'added from ruby'
[master b7ae93b] added from ruby
1 files changed, 1 insertions(+), 1 deletions(-)
rename hello.rb => ruby.rb (65%)

```

现在这些改变已经记录到我的 “master” 分支了。请注意，这里类名还是 “HelloWorld”，而不是 “HiWorld”。然后我想将类名的改变合并过来，我把 “change_class” 分支合并过来就行了。但是，我已经将文件名都改掉了，Git 知道该怎么办？

```

$ git branch
  change_class
* master

$ git merge change_class
Renaming hello.rb => ruby.rb

Auto-merging ruby.rb
Merge made by recursive.
  ruby.rb | 6 ++----
  1 files changed, 2 insertions(+), 4 deletions(-)

$ cat ruby.rb
class HiWorld
  def self.hello
    puts "Hello World from Ruby"
  end
end

HiWorld.hello

```

不错，它就是发现了。请注意，在这部操作，我没有遇到合并冲突，并且文件已经重命名、类名也换掉了。挺酷。

合并冲突

那么，Git 合并很有魔力，我们再也不用处理合并冲突了，对吗？不太确切。不同分支中修

改了相同区块的代码，电脑自己猜个延伸与的情况下，冲突就摆住我们面前了。我们有有两个分支中改了同一行代码的例子。

```
$ git branch
* master

$ git checkout -b fix_readme
Switched to a new branch 'fix_readme'

$ vim README $ git commit -am 'fixed readme title'
[fix_readme 3ac015d] fixed readme title
1 files changed, 1 insertions(+), 1 deletions(-)
```

我们在某分支中修改了 README 文件中的一行，并提交。我们再看 “master” 分支中对同个文件的同一行内容作不同的修改。

```
$ git checkout master
Switched to branch 'master'

$ vim README $ git commit -am 'fixed readme title differently'
[master 3cbb6aa] fixed readme title differently
1 files changed, 1 insertions(+), 1 deletions(-)
```

有意思的来了 —— 我们将前一个分支合并到 “master” 分支，一个合并冲突就出现了。

```
$ git merge fix_readme
Auto-merging README
CONFLICT (content): Merge conflict in README
Automatic merge failed; fix conflicts and then commit the result.

$ cat README
<<<<<<< HEAD
Many Hello World Examples
=====
Hello World Lang Examples
>>>>>>> fix_readme

This project has examples of hello world in
nearly every programming language.
```

你可以看到，Git 在产生合并冲突的地方插入了标准的与 Subversion 很像的合并冲突标记。轮到我们去解决这些冲突了。在这里我们就手动把它解决。如果你要 Git 打开一个图形化的合并工具，可以看看 **git 合并工具**（比如 kdiff3、emerge、p4merge 等）。

```
$ vim README      here I'm fixing the conflict
$ git diff
diff --cc README
```

```

y vim README here I'm fixing the conflict git diff --cc README
index 9103e27,69cad1a..0000000
--- a/README
+++ b/README@@@ -1,4 -1,4 +1,4 @@@- Many Hello World Examples
-Hello World Lang Examples++Many Hello World Lang Examples

```

This project has examples of hello world in

在 Git 中，处理合并冲突的时候有个很酷的提示。如果你执行 `git diff`，就像我演示的这样，它会告诉你冲突的两方，和你是如何解决的。现在是时候把它标记为已解决了。在 Git 中，我们可以用 `git add` —— 要告诉 Git 文件冲突已经解决，你必须把它写入缓存区。

```

$ git status -s
UU README
$ git add README $ git status -s
M README
$ git commit
[master 8d585ea] Merge branch 'fix_readme'

```

现在我们成功解决了合并中的冲突，并提交了结果

简而言之 使用 `git merge` 将另一个分支并入当前的分支中去。Git 会自动以最佳方式将两个不同快照中独特的工作合并到一个新快照中去。

git log 显示一个分支中提交的更改记录

到目前为止，我们已经提交快照到项目中，在不同的各自分离的上下文中切换，但假如我们忘了自己是如何到目前这一步的那该怎么办？或者假如我们想知道此分支与彼分支到底有啥区别？Git 提供了一个告诉你使你达成当前快照的所有提交消息的工具，叫做 `git log`。

要理解日志（log）命令，你需要了解当执行 `git commit` 以存储一个快照的时候，都有啥信息被保存了。除了文件详单、提交消息和提交者的信息，Git 还保存了你的此次提交所基于的快照。也就是，假如你克隆了一个项目，你是在什么快照的基础上做的修改而得到新保存的快照的？这有益于为项目进程提供上下文，使 Git 能够弄明白谁做了什么改动。如果 Git 有你的快照所基于的快照的话，它就能自动判断你都改变了什么。而新提交所基于的提交，被称作新提交的“父亲”。

某分支的按时间排序的“父亲”列表，当你在该分支时，可以执行 `git log` 以查看。例如，如果我们在本章中操作的 Hello World 项目中执行 `git log`，我们可以看到已提交的消息。

```
$ git log
commit 8d585ea6faf99facd39b55d6f6a3b3f481ad0d3d
Merge: 3cbb6aa 3ac015d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jun 4 12:59:47 2010 +0200

    Merge branch 'fix_readme'

Conflicts:
    README

commit 3cbb6aae5c0cbd711c098e113ae436801371c95e
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jun 4 12:58:53 2010 +0200

    fixed readme title differently

commit 3ac015da8ade34d4c7ebeffa2053fcac33fb495b
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jun 4 12:58:36 2010 +0200

    fixed readme title

commit 558151a95567ba4181bab5746bc8f34bd87143d6
Merge: b7ae93b 3467b0a
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jun 4 12:37:05 2010 +0200

    Merge branch 'change_class'
...
```

我们可以用 `--oneline` 选项来查看历史记录的最简洁的版本。

```
$ git log --oneline
8d585ea Merge branch 'fix_readme'
3cbb6aa fixed readme title differently
3ac015d fixed readme title
558151a Merge branch 'change_class'
b7ae93b added from ruby
```



```
3467b0a changed the class name
17f4acf first commit
```

这告诉我们的是，此项目的开发历史。如果提交消息描述性很好，这就能为我们提供关于有啥改动被应用、或者影响了当前快照的状态、以及这快照里头有啥。

我们还可以用它的十分有帮助的 `--graph` 选项，查看历史中什么时候出现了分支、合并。以下为相同的命令，开启了拓扑图选项：

```
$ git log --oneline --graph
*    8d585ea Merge branch 'fix_readme'
|\
| * 3ac015d fixed readme title
* | 3cbb6aa fixed readme title differently
|/
*    558151a Merge branch 'change_class'
|\
| * 3467b0a changed the class name
* | b7ae93b added from ruby
|/
* 17f4acf first commit
```

现在我们可以更清楚明了地看到何时工作分叉、又何时归并。这对查看发生了什么、应用了什么改变很有帮助，并且极大地帮助你管理你的分支。让我们创建一个分支，在里头做些事情，然后切回到主分支，也做点事情，然后看看 `log` 命令是如何帮助我们理清这两分支上都发生了啥的。

首先我们创建一个分支，来添加 Erlang 编程语言的 Hello World 示例——我们想要在一个分支里头做这个，以避免让可能还不能工作的代码弄乱我们的稳定分支。这样就可以切来切去，片叶不沾身。

```
$ git checkout -b erlang
Switched to a new branch 'erlang'
$ vim erlang_hw.erl $ git add erlang_hw.erl $ git commit -m 'added erlang'
[erlang ab5ab4c] added erlang
1 files changed, 5 insertions(+), 0 deletions(-)
create mode 100644 erlang_hw.erl
```

由于我们玩函数式编程很开心，以至于沉迷其中，又在“erlang”分支中添加了一个 Haskell 的示例程序。

```
$ vim haskell.hs$ git add haskell.hs $ git commit -m 'added haskell'
[erlang 1834130] added haskell
1 files changed, 4 insertions(+), 0 deletions(-)
create mode 100644 haskell.hs
```


最后，我们决定还是把 Ruby 程序的类名改回原先的样子。与其创建另一个分支，我们可以返回主分支，改变它，然后直接提交。

```
$ git checkout master
Switched to branch 'master'
$ ls
README  ruby.rb
$ vim ruby.rb $ git commit -am 'reverted to old class name'
[master 594f90b] reverted to old class name
1 files changed, 2 insertions(+), 2 deletions(-)
```

现在假设我们有段时间不做这个项目了，我们做别的去了。当我们回来的时候，我们想知道“erlang”分支都是啥，而主分支的进度又是怎样。仅仅看分支的名字，我们是从无从知道自己还在里面有 Haskell 的改动的，但是用 `git log` 我们就可以。如果你在命令行中提供一个分支名字，它就会显示该分支历史中“可及”的提交，即从该分支创立起可追溯的影响了最终的快照的提交。

```
$ git log --oneline erlang1834130 added haskell
ab5ab4c added erlang
8d585ea Merge branch 'fix_readme'
3cbb6aa fixed readme title differently
3ac015d fixed readme title
558151a Merge branch 'change_class'
b7ae93b added from ruby
3467b0a changed the class name
17f4acf first commit
```

如此，我们很容易就看到分支里头还包括了 Haskell 代码（高亮显示了）。更酷的是，我们很容易地告诉 Git，我们只对某个分支中可及的提交感兴趣。换句话说，某分支中与其他分支相比唯一的提交。

在此例中，如果我们想要合并“erlang”分支，我们需要看当合并的时候，都有啥提交会作用到我们的快照上去。我们告诉 Git 的方式是，在不要看到的分支前放一个 。例

如，如果我们想要看“erlang”分支中但不在主分支中的提交，我们可以用

`erlang ^master`，或者反之。

```
$ git log --oneline erlang ^master
1834130 added haskell
ab5ab4c added erlang
$ git log --oneline master ^erlang
594f90b reverted to old class name
```

这为我们提供了一个良好的、简易的分支管理工具。它使我们能够非常容易地查看对某个分支唯一的提交，从而知道我们缺少什么，以及当我们要合并时，会有什么被合并进去。

简而言之 使用 `git log` 列出促成当前分支目前的快照的提交的提交历史记录。这使你能够看到项目是如何到达现在的状况的。

git tag 给历史记录中的某个重要的一点打上标签

如果你达到一个重要的阶段，并希望永远记住那个特别的提交快照，你可以使用 `git tag` 给它打上标签。该 `tag` 命令基本上会给该特殊提交打上永久的书签，从而使你在将来能够用它与其他提交比较。通常，你会在切取一个发布版本或者交付一些东西的时候打个标签。

比如说，我们想为我们的 Hello World 项目发布一个“1.0”版本。我们可以用 `git tag -a v1.0` 命令给最新一次提交打上（`HEAD`）“v1.0”的标签。`-a` 选项意为“创建一个带注解的标签”，从而使你为标签添加注解。绝大部分时候都会这么做的。不用 `-a` 选项也可以执行的，但它不会记录这标签是啥时候打的，谁打的，也不会让你添加个标签的注解。我推荐一直创建带注解的标签。

```
$ git tag -a v1.0
```

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

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

```
$ git log --oneline --decorate --graph
* 594f90b (HEAD, tag: v1.0, master) reverted to old class name
* 8d585ea Merge branch 'fix_readme'
|\
```

```
| * 3ac015d (fix_readme) fixed readme title
* | 3cbb6aa fixed readme title differently
|/
* 558151a Merge branch 'change_class'
|\
| * 3467b0a changed the class name
* | b7ae93b added from ruby
|/
* 17f4acf first commit
```

如果我们有新提交，该标签依然会待在该提交的边上，所以我们已经给那个特定快照永久打上标签，并且能够将它与未来的快照做比较。

不过我们并不需要给当前提交打标签。如果我们忘了给某个提交打标签，又将它发布了，我们可以给它追加标签。在相同的命令末尾加上提交的 SHA，执行，就可以了。例如，假设我们发布了提交 `558151a`（几个提交之前的事情了），但是那时候忘了给它打标签。我们

现在也可以：

```
$ git tag -a v0.9 558151a$ git log --oneline --decorate --graph
* 594f90b (HEAD, tag: v1.0, master) reverted to old class name
* 8d585ea Merge branch 'fix_readme'
|\
| * 3ac015d (fix_readme) fixed readme title
* | 3cbb6aa fixed readme title differently
|/
* 558151a (tag: v0.9) Merge branch 'change_class'
|\
| * 3467b0a changed the class name
* | b7ae93b added from ruby
|/
* 17f4acf first commit
```

4. 分享与更新项目

Git 并不像 Subversion 那样有个中心服务器。目前为止所有的命令都是本地执行的，更新的知识本地的数据库。要通过 Git 与其他开发者合作，你需要将数据放到一台其他开发者能够连接的服务器上。Git 实现此流程的方式是将你的数据与另一个仓库同步。在服务器与客户端之间并没有实质的区别——Git 仓库就是 Git 仓库，你可以很容易地在两者之间同

步。

一旦你有了个 Git 仓库，不管它是在你自己的服务器上，或者是由 GitHub 之类的地方提供，你都可以告诉 Git 推送你拥有的远端仓库还没有的数据，或者叫 Git 从别的仓库把差别取过来。

联网的时候你可以随时做这个，它并不需要对应一个 `commit` 或者别的什么。一般你会本地提交几次，然后从你的项目克隆自的线上的共享仓库提取数据以保持最新，将新完成的合并到你完成的工作中去，然后推送你的改动会服务器。

简而言之 使用 `git fetch` 更新你的项目，使用 `git push` 分享你的改动。你可以用 `git remote` 管理你的远程仓库。

git remote 罗列、添加和删除远端仓库别名

不像中心化的版本控制系统（客户端与服务端很不一样），Git 仓库基本上都是一致的，并且可以同步他们。这使得拥有多个远端仓库变得容易——你可以拥有一些只读的仓库，另外的一些也可写的仓库。

当你需要与远端仓库同步的时候，不需要使用它详细的链接。Git 储存了你感兴趣的远端仓库的链接的别名或者昵称。你可以使用 `git remote` 命令管理这个远端仓库列表。

git remote 列出远端别名

如果没有任何参数，Git 会列出它存储的远端仓库别名了事。默认情况下，如果你的项目是克隆的（与本地创建一个新的相反），Git 会自动将你的项目克隆自的仓库添加到列表中，并取名“origin”。如果你执行时加上 `-v` 参数，你还可以看到每个别名的实际链接地址。

```
$ git remote
origin
$ git remote -v
origin  git@github.com:github/git-reference.git (fetch)
origin  git@github.com:github/git-reference.git (push)
```

在此你看到了该链接两次，是因为 Git 允许你为每个远端仓库添加不同的推送与获取的链接，以备你读写时希望使用不同的协议。

git remote add 为你的项目添加一个新的远端仓库

如果你想添加一个远端仓库，你可以使用 `git remote add` 命令。如果你已经有一个远端仓库，你可以使用 `git remote rm` 命令来删除它。

如果你希望分享一个本地创建的仓库，或者你希望获取别人的仓库中的代码——如果你想要以任何方式与一个新仓库沟通，最简单的方式通常就是把它添加为一个远端仓库。执行 `git remote add [alias] [url]` 就可以。此命令将 `[url]` 以 `[alias]` 的别名添加为本地的远端仓库。

例如，假设我们想要与整个世界分享我们的 Hello World 程序。我们可以在一台服务器上创建一个新仓库（我以 GitHub 为例子）。它应该会给你一个链接，在这里就是 “git@github.com:schacon/hw.git”。要把它添加到我们的项目以便我们推送以及获取更新，我们可以这样：

```
$ git remote$ git remote add github git@github.com:schacon/hw.git$ git remote -v
github    git@github.com:schacon/hw.git (fetch)
github    git@github.com:schacon/hw.git (push)
```

像分支的命名一样，远端仓库的别名是强制的——就像 “master”，没有特别意义，但它广为使用，因为 `git init` 默认用它；“origin” 经常被用作远端仓库别名，就因为 `git clone` 默认用它作为克隆自的链接的别名。此例中，我决定给我的远端仓库取名 “github”，但我叫它随便什么都可以。

git remote rm 删除现存的某个别名

Git addeth and Git taketh away. 如果你需要删除一个远端——不再需要它了、项目已经没了，等等——你可以使用 `git remote rm [alias]` 把它删掉。

```
$ git remote -v
github    git@github.com:schacon/hw.git (fetch)
github    git@github.com:schacon/hw.git (push)
$ git remote add origin git://github.com/pjhyett/hw.git$ git remote -v
github    git@github.com:schacon/hw.git (fetch)
github    git@github.com:schacon/hw.git (push)
origin    git://github.com/pjhyett/hw.git (fetch)
origin    git://github.com/pjhyett/hw.git (push)
$ git remote rm origin$ git remote -v
github    git@github.com:schacon/hw.git (fetch)
github    git@github.com:schacon/hw.git (push)
```

简而言之 你可以用 `git remote` 列出你的远端仓库和那些仓库的链接。你可以使用 `git remote add` 添加新的远端仓库，用 `git remote rm` 删掉已存在的那些。

git fetch 从远端仓库下载新分支与数据 git pull 从远端仓库提取数据并尝试合并到当前分支

Git 有两个命令用来从某一远端仓库更新。 `git fetch` 会使你与另一仓库同步，提取你本地所没有的数据，为你在同步时的该远端的每一分支提供书签。这些分支被叫做“远端分支”，除了 Git 不允许你检出（切换到该分支）之外，跟本地分支没区别——你可以将它们合并到当前分支，与其他分支作比较差异，查看那些分支的历史日志，等等。同步之后你就可以在本地操作这些。

第二个会从远端服务器提取新数据的命令是 `git pull`。基本上，该命令就是在 `git fetch` 之后紧接着 `git merge` 远端分支到你所在的任意分支。我个人不太喜欢这命令——我更喜欢 `fetch` 和 `merge` 分开来做。少点魔法，少点问题。不过，如果你喜欢这主意，你可以看

一下 `git pull` 的 **官方文档**。

假设你配置好了一个远端，并且你想要提取更新，你可以首先执行 `git fetch [alias]` 告诉 Git 去获取它有你没有的数据，然后你可以执行 `git merge [alias]/[branch]` 以将服务器上的任何更新（假设有人这时候推送到服务器了）合并到你的当前分支。那么，如果我是与两三个其他人合作 Hello World 项目，并且想要将我最近连接之后的所有改动拿过来，我可以这么做：

```
$ git fetch github
remote: Counting objects: 4006, done.
remote: Compressing objects: 100% (1322/1322), done.
remote: Total 2783 (delta 1526), reused 2587 (delta 1387)
Receiving objects: 100% (2783/2783), 1.23 MiB | 10 KiB/s, done.
Resolving deltas: 100% (1526/1526), completed with 387 local objects.
From github.com:schacon/hw
   8e29b09..c7c5a10  master       -> github/master
   0709fdc..d4ccf73  c-langs    -> github/c-langs
   6684f82..ae06d2b  java       -> github/java
* [new branch]      ada         -> github/ada
* [new branch]      lisp        -> github/lisp
```

可以看到自从上一次与远端仓库同步以后，又新增或更新了五个分支。

“ada”与“lisp”分支是新的，而“master”、“clang”与“java”分支则被更新了。在此例中，我的团队在合并入主分支之前，将提议的更新推送到远端分支以审核。

你可以看到 Git 做的映射。远端仓库的主分支成为了本地的一个叫做“github/master”的

分支。这样我就可以执行 `git merge github/master` 将远端的主分支和并入我的本地主分支。或者，我可以 `git log github/master ^master` 看看该分支上的新提交。如果你的远端仓库叫做“origin”，那远端主分支就会叫做 `origin/master`。几乎所有能在本地分支上执行的命令都可以在远端分支上用。

如果你有多个远端仓库，你可以执行 `git fetch [alias]` 提取特定的远端仓库，或者执行 `git fetch --all` 告诉 Git 同步所有的远端仓库。

简而言之 执行 `git fetch [alias]` 来将你的仓库与远端仓库同步，提取所有它独有的数据到本地分支以合并或者怎样。

git push 推送你的新分支与数据到某个远端仓库

想要与他人分享你牛鼻的提交，你需要将改动推送到远端仓库。执行

`git push [alias] [branch]`，就会将你的 [branch] 分支推送成为 [alias] 远端上的 [branch] 分支。让我们试试推送我们的主分支到先前添加的“github”远端仓库上去。

```
$ git push github master
Counting objects: 25, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (25/25), done.
Writing objects: 100% (25/25), 2.43 KiB, done.
Total 25 (delta 4), reused 0 (delta 0)
To git@github.com:schacon/hw.git
 * [new branch]      master -> master
```

挺简单。现在如果有人从该仓库克隆，他会得到我提交的完完全全的一份历史记录。

如果有个像之前创建的“erlang”分支那样的主题分支，想只分享这个，该怎么办呢？你可以相应的只推送该分支。

```
$ git push github erlang
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 652 bytes, done.
Total 6 (delta 1), reused 0 (delta 0)
To git@github.com:schacon/hw.git
```



```
to git@github.com:schacon/hw.git
* [new branch]      erlang -> erlang
```

现在当人们从该仓库克隆时，他们就会得到一个“erlang”分支以查阅、合并。用这种方式，你可以推送任何分支到任何你有写权限的仓库。如果你的分支已经在该仓库中了，它会试着去更新，如果它不再，Git 会把它加上。

最后一个当你推送到远端分支时会碰到的主要问题是，其他人在此期间也推送了的情况。如果你和另一个开发者同时克隆了，又都有提交，那么当她推送后你也想推送时，默认情况下 Git 不会让你覆盖她的改动。相反的，它会在你试图推送的分支上执行 `git log`，确定它能够在你的推送分支的历史记录中看到服务器分支的当前进度。如果它在你的历史记录中看不到，它就会下结论说你过时了，并打回你的推送。你需要正式提取、合并，然后再次推送——以确定你把她的改动也考虑在内了。

当你试图推送到某个以被更新的远端分支时，会出现下面这种情况：

```
$ git push github master
To git@github.com:schacon/hw.git
 ! [rejected]      master -> master (non-fast-forward)
error: failed to push some refs to 'git@github.com:schacon/hw.git'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes before pushing again.  See the 'Note about
fast-forwards' section of 'git push --help' for details.
```

你可以修正这个问题。执行 `git fetch github; git merge github/master`，然后再推送

简而言之 执行 `git push [alias] [branch]` 将你的本地改动推送到远端仓库。如果可以的话，它会依据你的 [branch] 的样子，推送到远端的 [branch] 去。如果你上次提取、合并之后，另有人推送了，Git 服务器会拒绝你的推送，知道你是最新的为止。

5. 检查与比较

现在你有了一堆分支，短期的主题、长期的特性或者其它。怎样追踪他们呢？Git 有一组工具，可以帮助你弄明白工作是在哪儿完成的，两个分支间的区别是啥，等等。

简而言之 执行 `git log` 找到你的项目历史中的特定提交——按作者、日期、内容或者历史记录。执行 `git diff` 比较历史记录中的两个不同的点——通常是为了看看两个分支有啥区别，或者从某个版本到另一个版本，你的软件有啥变化。

git log 过滤你的提交历史记录

通过查看分支中另一分支看不到的提交记录，我们已经看到如何用 `git log` 来比较分支。

（如果你不记得了，它看起来是这样的：`git log branchA ^branchB`）。而且，你也可以用 `git log` 去寻找特定的提交。在此，我们会看到一些更为使用的 `git log` 选项，不过哪有很多。完整的清单可以看看官方文档。

git log --author 只寻找某个特定作者的提交

要过滤你的提交历史，只寻找某个特定作者的提交，你可以使用 `--author` 选项。例如，比方说我们要找 Git 源码中 Linus 提交的部分。我们可以执行类似 `git log --author=Linus` 的命令。这个查找是大小写敏感的，并且也会检索电子邮箱地址。我在此例中使用 `-[number]` 选项，以限制结果为最近 [number] 次的提交。

```
$ git log --author=Linus --oneline -5
81b50f3 Move 'builtin-*' into a 'builtin/' subdirectory
3bb7256 make "index-pack" a built-in
377d027 make "git pack-redundant" a built-in
b532581 make "git unpack-file" a built-in
112dd51 make "mktag" a built-in
```

git log --since --before 根据日期过滤提交记录

如果你要指定一个你感兴趣的日期范围以过滤你的提交，可以执行几个选项——我用 `--since` 和 `--before`，但是你也可以用 `--until` 和 `--after`。例如，如果我要看 Git 项目中三周前且在四月十八日之后的所有提交，我可以执行这个（我还用了 `--no-merges` 选项以隐藏合并提交）：

```
$ git log --oneline --before={3.weeks.ago} --after={2010-04-18} --no-merge
s
5469e2d Git 1.7.1-rc2
d43427d Documentation/remote-helpers: Fix typos and improve language
272a36b Fixup: Second argument may be any arbitrary string
b6c8d2d Documentation/remote-helpers: Add invocation section
5ce4f4e Documentation/urls: Rewrite to accomodate transport::address
00b84e9 Documentation/remote-helpers: Rewrite description
03aa87e Documentation: Describe other situations where -z affects git diff
77bc694 rebase-interactive: silence warning when no commits rewritten
```

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

git log --grep 根据提交注释过滤提交记录

你或许还想根据提交注释中的某个特定短语查找提交记录。可以用 `--grep` 选项。比如说我知道有个提交是有关使用 P4EDITOR 环境变量，又想回忆起那个改动是啥样子的——我可以用 `--grep` 选项找到该提交。

```
$ git log --grep=P4EDITOR --no-mergescommit 82cea9ffb1c4677155e3e2996d7654
2502611370
```

```
Author: Shawn Bohrer
```

```
Date: Wed Mar 12 19:03:24 2008 -0500
```

```
git-p4: Use P4EDITOR environment variable when set
```

```
Perforce allows you to set the P4EDITOR environment variable to your
preferred editor for use in perforce. Since we are displaying a
perforce changelog to the user we should use it when it is defined.
```

```
Signed-off-by: Shawn Bohrer <shawn.bohrer@gmail.com>
```

```
Signed-off-by: Simon Hausmann <simon@lst.de>
```

Git 会对所有的 `--grep` 和 `--author` 参数作逻辑或。如果你用 `--grep` 和 `--author` 时，想看的是某人写作的并且有某个特殊的注释内容的提交记录，你需要加上 `--all-match` 选项。在这些例子中，我会用上 `--format` 选项，这样我们就可以看到每个提交的作者是谁了。

如果我查找注释内容含有 “p4 depo” 的提交，我得到了三个提交：

```
$ git log --grep="p4 depo" --format="%h %an %s"
ee4fd1a Junio C Hamano Merge branch 'master' of git://repo.or.cz/git/fasti
mport
da4a660 Benjamin Sergeant git-p4 fails when cloning a p4 depo.
1cd5738 Simon Hausmann Make incremental imports easier to use by storing t
he p4 d
```

如果我加上 `--author=Hausmann` 参数，与进一步过滤上述结果到 Simon 的唯一提交相反，它会告诉我所有 Simon 的提交，或者注释中有 “p4 demo” 的提交：

```
$ git log --grep="p4 depo" --format="%h %an %s" --author="Hausmann"
```

```

cdc7e38 Simon Hausmann Make it possible to abort the submission of a change to Pe
f5f7e4a Simon Hausmann Clean up the git-p4 documentation
30b5940 Simon Hausmann git-p4: Fix import of changesets with file deletions
4c750c0 Simon Hausmann git-p4: git-p4 submit cleanups.
0e36f2d Simon Hausmann git-p4: Removed git-p4 submit --direct.
edae1e2 Simon Hausmann git-p4: Clean up git-p4 submit's log message handling.
4b61b5c Simon Hausmann git-p4: Remove --log-substitutions feature.
36ee4ee Simon Hausmann git-p4: Ensure the working directory and the index are clea
e96e400 Simon Hausmann git-p4: Fix submit user-interface.
38f9f5e Simon Hausmann git-p4: Fix direct import from perforce after fetching cha
2094714 Simon Hausmann git-p4: When skipping a patch as part of "git-p4 submit" m
1ca3d71 Simon Hausmann git-p4: Added support for automatically importing newl
...

```

不过，如果加上 `--all-match`，结果就是我想要的了：

```

$ git log --grep="p4 depo" --format="%h %an %s" --author="Hausmann" --all-match
1cd5738 Simon Hausmann Make incremental imports easier to use by storing the p4 d

```

git log -S 依据所引入的差值过滤

如果你写的提交注释都极度糟糕怎么办？或者，如果你要找某个函数是何时引入的，某些变量是在哪里开始被使用的？你可以告诉 Git 在每个提交之间的差值中查找特定字符串。

例如，如果我们想要找出哪个提交修改出了类似函数

名 “userformat_find_requirements”，我们可以执行（注意在 “-S” 与你要找的东东之间没有 “=”）：

```

$ git log -Suserformat_find_requirementscommit 5b16360330822527eac1fa84131d185ff784c9fb
Author: Johannes Gilger
Date: Tue Apr 13 22:31:12 2010 +0200

pretty: Initialize notes if %N is used

```

When using `git log --pretty='%N'` without an explicit `--show-notes`, git would segfault. This patches fixes this behaviour by loading the needed

notes datastructures if `--pretty` is used and the format contains `%N`. When `--pretty='%N'` is used together with `--no-notes`, `%N` won't be expanded.

This is an extension to a proposed patch by Jeff King.

Signed-off-by: Johannes Gilger

Signed-off-by: Junio C Hamano

git log -p 显示每个提交引入的补丁

每个提交都是项目的一个快照。由于每个提交都记录它所基于的快照，Git 能够经常对它们求差值，并以补丁形式向你展示。这意味着，对任意提交，你都可以获取该提交给项目引入补丁。你可以用 `git show [SHA]` 加上某个特定的提交 SHA 获取，或者执行 `git log -p`，它会告诉 Git 输出每个提交之后的补丁。这是个总结某一支或者两个提交之间都发生了神马的好途径。

```
$ git log -p --no-merges -2commit 594f90bdee4faf063ad07a4a6f503fdead3ef606
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jun 4 15:46:55 2010 +0200
```

```
    reverted to old class name
```

```
diff --git a/ruby.rb b/ruby.rb
index bb86f00..192151c 100644
--- a/ruby.rb
+++ b/ruby.rb@@ -1,7 +1,7 @@-class HiWorld+class HelloWorld
  def self.hello
    puts "Hello World from Ruby"
  end
end
```

```
-HiWorld.hello+HelloWorld.hello
```

```
commit 3cbb6aae5c0cbd711c098e113ae436801371c95e
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jun 4 12:58:53 2010 +0200
```

```
    fixed readme title differently
```

```
diff --git a/README b/README
index d053cc8..9103e27 100644
--- a/README
+++ b/README@@ -1,4 +1,4 @@-Hello World Examples+Many Hello World Examples
=====

This project has examples of hello world in
```

这是个总结改动，以及合并或发布之前重申一系列提交的好方式。

git log --stat 显示每个提交引入的改动的差值统计

如果 `-p` 选项对你来说太详细了，你可以用 `--stat` 总结这些改动。这是不用 `-p`，而用 `--stat` 选项时，同一份日志的输出。

```
$ git log --stat --no-merges -2commit 594f90bdee4faf063ad07a4a6f503fdead3e
f606
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jun 4 15:46:55 2010 +0200

    reverted to old class name

ruby.rb |    4 ++--
1 files changed, 2 insertions(+), 2 deletions(-)

commit 3cbb6aae5c0cbd711c098e113ae436801371c95e
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jun 4 12:58:53 2010 +0200

    fixed readme title differently

README |    2 +-
1 files changed, 1 insertions(+), 1 deletions(-)
```

同样的基本信息，但更紧凑——它仍然让你看到相对改动，和改动了哪些文件。

git diff

最后，要查看两个提交快照的绝对改动，你可以用 `git diff` 命令。这在两个主要情况中广为使用——查看两个分支彼此之间的差值，和查看自发布或者某个旧历史点之后都有啥变了。让我们看看这两情况。

你仅需执行 `git diff [version]` (或者你给该发布打的任何标签) 就可以查看自最近发布之后的改动。 例如, 如果我们想要看看自 v0.9 发布之后我们的项目改变了啥, 我们可以执行 `git diff v0.9`

```
$ git diff v0.9
diff --git a/README b/README
index d053cc8..d4173d5 100644
--- a/README
+++ b/README@@ -1,4 +1,4 @@-Hello World Examples+Many Hello World Lang Exa
mples
=====

This project has examples of hello world in
diff --git a/ruby.rb b/ruby.rb
index bb86f00..192151c 100644
--- a/ruby.rb
+++ b/ruby.rb@@ -1,7 +1,7 @@-class HiWorld+class HelloWorld
  def self.hello
    puts "Hello World from Ruby"
  end
end

-HiWorld.hello+HelloWorld.hello
```

正如 `git log`, 你可以给它加上 `--stat` 参数。

```
$ git diff v0.9 --stat
README | 2 +-
ruby.rb | 4 ++--
2 files changed, 3 insertions(+), 3 deletions(-)
```

要比较两个不同的分支, 你可以执行类似 `git diff branchA branchB` 的命令。 不过它的问题在于它会完完全全按你说的作 —— 它会直接给你个补丁文件, 该补丁能够将甲分支的最新快照变成乙分支的最新快照的样子。 这意味着如果两个分支已经产生分歧 —— 奔往两个不同方向了 —— 它会移除甲分支中引入的所有工作, 然后累加乙分支中的所有工作。 这大概不是你要的吧 —— 你想要不在甲分支中的乙分支的改动。 所以你真的需要的是两个分支叉开去时, 和最新的乙分支的差别。 所以, 如果我们的历史记录看起来像这样:

```
$ git log --graph --oneline --decorate --all
* 594f90b (HEAD, tag: v1.0, master) reverted to old class name
```

```
| * 1834130 (erlang) added haskell
| * ab5ab4c added erlang
|/
* 8d585ea Merge branch 'fix_readme'
...
```

并且，我们想要看“erlang”分支与主分支相比的查别。执行 `git diff master erlang` 会给我们错误的结果。

```
$ git diff --stat master erlang
erlang_hw.erl | 5 +++++
haskell.hs    | 4 ++++
ruby.rb       | 4 ++--
3 files changed, 11 insertions(+), 2 deletions(-)
```

你可以看到，它加上了 erlang 和 haskell 文件，这确实是在该分支中做的，但是它同时恢复了我们在主分支中改动的 ruby 文件。我们真心想要的只是“erlang”分支中的改动（添加两个文件）。我们可以通过求两个分支分歧时的共同提交与该分支的差值得到想要的结果：

```
$ git diff --stat 8d585ea erlang
erlang_hw.erl | 5 +++++
haskell.hs    | 4 ++++
2 files changed, 9 insertions(+), 0 deletions(-)
```

这才是我们在找的，但是我们可不想要每次都要找出两个分支分歧时的那次提交。幸运的是，Git 为此提供了一个快捷方式。如果你执行 `git diff master...erlang`（在分支名之间有三个半角的点），Git 就会自动找出两个分支的共同提交（也被成为合并基础），并求差值。

```
$ git diff --stat master erlang
erlang_hw.erl | 5 +++++
haskell.hs    | 4 ++++
ruby.rb       | 4 ++--
3 files changed, 11 insertions(+), 2 deletions(-)
$ git diff --stat master...erlang
erlang_hw.erl | 5 +++++
haskell.hs    | 4 ++++
2 files changed, 9 insertions(+), 0 deletions(-)
```


几乎每一次你要对比两个分支的时候，你都会想用三个点的语法，因为它通常会给你你想要的。

顺带提一句，你还可以让 Git 手工计算两次提交的合并基础（第一个共同的祖提交），即

`git merge-base` 命令：

```
$ git merge-base master erlang
8d585ea6faf99facd39b55d6f6a3b3f481ad0d3d
```

所以你执行下面这个也跟 `git diff master...erlang` 一样：

```
$ git diff --stat $(git merge-base master erlang) erlang
erlang_hw.erl |      5 +++++
haskell.hs    |      4 ++++
2 files changed, 9 insertions(+), 0 deletions(-)
```

当然，我会推荐简单点的那个。

简而言之 使用 `git diff` 查看某一支自它偏离出来起与过去某一点之间项目的改动。总是使用 `git diff branchA...branchB` 来查看 branchB 与 branchA 的相对差值，这会让事情简单点。