沉浸式学Git

极客学院出版

前言

本书简介

沉浸式学 Git 是一份强调通过实践来掌握 Git 基础用法的指南。本书包含 52 个实验，这些实验经过精心设计，篇幅皆十分短小，只需几分钟时间便可完成。对于想要快速学习 Git 的朋友而言，这是一本不可多得的好书。

关于作者

jim Jim Weirich Ruby 编程语言大师，开创了流行的构建工具 Rake。

关于译者

toy 徐小东 a.k.a toy，Linux 爱好者，曾译有《Perl 程序员应该知道的事》一书。

目录

1

设置

目的

设置 Git 以便准备开始工作。

设置姓名和 Email

如果你以前从未用过 Git，那么你需要先做一些设置。执行下列命令以便让 Git 知道你的姓名和 Email。如果你已经设置了 Git，那么你可以跳到下一小节。

```
$ git config --global user.name "Your Name"
$ git config --global user.email "your_email@whatever.com"
```

设置行尾首选项

Unix/Mac 用户：

```
$ git config --global core.autocrlf input
$ git config --global core.safecrlf true
```

Windows 用户：

```
$ git config --global core.autocrlf true
$ git config --global core.safecrlf true
```

2

再谈设置

目的

获得教程材料，并准备执行。

获得教程包

从以下地址获得 Git 教程包：

http://gitimmersion.com/git_tutorial.zip

解包

教程包有一个主目录“git_tutorial”及三个子目录：

- html：HTML 文件。让你的浏览器打开 html/index.html。
- work：空的工作目录。你可以在这里创建仓库。
- repos：预先打包的 Git 仓库，这样你可以在教程中的任何地方跳转。如果你遇到问题，那么只需复制想要的实验到你的工作目录。

3

创建项目

目的

学习如何从零开始创建 Git 仓库。

创建“Hello, World”程序

在一个空的工作目录中开始，创建一个名为“hello”的空目录，然后创建一个名为 hello.rb 且包含如下内容的文件。

```
$ mkdir hello
$ cd hello
```

文件：hello.rb

```
puts "Hello, World"
```

创建仓库

你现在有一个包含单个文件的目录。要从该目录创建 Git 仓库，执行 `git init` 命令。

```
$ git init
```

输出：

```
$ git init
Initialized empty Git repository in /Users/jim/working/git/git_immersion/auto/hello/.git/
```

添加程序到仓库

现在让我们添加“Hello, World”程序到仓库。

```
$ git add hello.rb
$ git commit -m "First Commit"
```

你应该看到：

```
$ git add hello.rb
$ git commit -m "First Commit"
[master (root-commit) 9416416] First Commit
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 hello.rb
```

4

检查状态

目的

学习如何检查仓库的状态。

检查仓库的状态

使用 `git status` 命令检查当前仓库的状态。

```
$ git status
```

你应该看到：

```
$ git status
 # On branch master

nothing to commit (working directory clean)
```

`status` 命令报告这儿没有什么要提交的。这意味着仓库具有工作目录的全部当前状态。这儿没有不同的更改要记录。

我们将继续使用 `git status` 命令来监视仓库和工作目录间的状态。

5

做更改

目的

学习如何监视工作目录的状态。

更改“Hello, World”程序

是时候更改我们的 hello 程序以便使它能从命令行传递参数。将文件更改为：

```
puts "Hello, #{ARGV.first}!"
```

检查状态

现在检查工作目录的状态。

```
$ git status
```

你应该看到：

```
$ git status
 # On branch 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:   hello.rb

 #

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

值得注意的第一件事是 Git 知道 hello.rb 文件已被修改，但 Git 还没有通知这些更改。

另外要注意的是状态信息给你接下来需要做什么的提示。如果你想要添加这些更改到仓库，那么使用 `git add` 命令。否则，使用 `git checkout` 命令放弃更改。

下一步

让我们暂存更改。

暂存更改

目的

学习如何暂存更改以用于稍后提交。

添加更改

现在告诉 Git 暂存更改，并检查状态。

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

你应该看到：

```
$ git add hello.rb
$ git status
 # On branch master

 # Changes to be committed:

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

 #

 #   modified:   hello.rb

 #
```

对 hello.rb 文件的更改已被暂存。这意味着 Git 现在知道这些更改，但还没有永久记录到仓库中。下面的提交操作将包括暂存的更改。

如果你决定不提交更改，那么 `status` 命令将提醒你使用 `git reset` 命令能取消暂存更改。

7

暂存与提交

在 Git 中分开暂存步骤是直到你需要使用源码控制处理的协调解决哲学。你可以继续对工作目录做更改，然后当你想要与源码控制交互时，Git 允许你使用精确地记录你所作的小提交来记录你的更改。

例如，假设你编辑了三个文件（a.rb、b.rb 及 c.rb）。现在你想提交所有更改，但你想要 a.rb 和 b.rb 中的更改作为单个的提交，而 c.rb 的更改与前两个文件在逻辑上不相关，那么应该分开提交。

你可以执行下列命令：

```
$ git add a.rb
$ git add b.rb
$ git commit -m "Changes for a and b"

$ git add c.rb
$ git commit -m "Unrelated change to c"
```

通过分开暂存和提交，你能够更加容易地调优每一个提交。

8

提交更改

目的

学习如何提交更改到仓库。

提交更改

好，关于暂存谈得够多了。让我们提交已暂存的内容到仓库。

当你先前使用 `git commit` 命令提交 hello.rb 文件的初始化版本到仓库时，你在命令行上的 `-m` 选项可以包含注释。`commit` 命令将允许你交互式地编辑提交的注释。现在让我们试试看。

如果你从命令行忽略 `-m` 选项，那么 Git 将带你到所选的编辑器中。编辑器按以下列表选择（使用优先级顺序）：

```
GIT_EDITOR 环境变量
core.editor 配置设置
VISUAL 环境变量
EDITOR 环境变量
```

我已将 `EDITOR` 变量设置为 emacsclient。

那么，现在提交并检查状态。

```
$ git commit
```

你应该在编辑器中看到下面的内容：

```
|
 # 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

 #
```

在第一行，输入注释：“Using ARGV”。保存文件，并退出编辑器。你应该看到：

```
git commit
Waiting for Emacs...
[master 569aa96] Using ARGV
 1 files changed, 1 insertions(+), 1 deletions(-)
```

“Waiting for Emacs…”来自发送文件到正在运行的 Emacs 程序 emacsclient，并等候关闭文件。其余的输出是标准的提交信息。

检查状态

最后，让我们再检查下状态。

```
$ git status
```

你应该看到：

```
$ git status
 # On branch master

nothing to commit (working directory clean)
```

工作目录是干净的，且准备让你继续。

9

更改而非文件

Git 聚焦于文件的更改而非文件本身。当你说 Git 添加文件时，你并非在告诉 Git 要添加文件到仓库。而是说 Git 应当对文件的当前状态做记录以便稍后提交。

我们将尝试在本次实验中探索其中的差异。

初次更改：允许默认名称

如果命令行参数未提供，更改“Hello, World”程序来接受一个默认值。

```
name = ARGV.first || "World"
puts "Hello, #{name}!"
```

添加更改

现在添加此次更改到 Git 的暂存区。

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

二次更改：添加注释

现在给“Hello, World”程序添加一行注释。

```
 # Default is "World"
name = ARGV.first || "World"
puts "Hello, #{name}!"
```

检查当前状态

```
$ git status
```

你应该看到：

```
$ git status
 # On branch master

 # Changes to be committed:

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

```
#
#   modified:   hello.rb
#
# 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:   hello.rb
#
```

注意 hello.rb 在状态中被列了两次。第一次更改已被暂存，且准备提交。第二次更改还未暂存。如果你现在提交，那么注释不会保存到仓库中。

让我们试试看。

提交

提交暂存的更改，然后重新检查状态。

```
$ git commit -m "Added a default value"
$ git status
```

你应该看到：

```
$ git commit -m "Added a default value"
[master 582495a] Added a default value
 1 files changed, 3 insertions(+), 1 deletions(-)
$ git status
 # On branch 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:   hello.rb
 #
no changes added to commit (use "git add" and/or "git commit -a")
```

`status` 命令将告诉你 hello.rb 还有未记录的更改，且不在暂存 区。

添加第二次更改

现在添加第二次更改到暂存区，然后执行 `git status`。

```
$ git add .
$ git status
```

注意：我们使用当前目录（.）作为要添加的文件。这是一种添加当前目录及其子目录下所有更改文件的习惯简写方式。但因为它添加所有东东，所以在做 `add .` 前检查状态是一个好主意，只是为了确定你没有添加不想要的文件。

我想你已经明白了 `add .` 这个技巧，但为了安全在余下的教程中我们将继续直接添加文件。

你应该看到：

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

现在第二次已经暂存，且准备提交。

提交第二次更改

```
$ git commit -m "Added a comment"
```

10

历史

目的

学习如何查看项目的历史。

获得已经做过的更改清单是 `git log` 命令的功能。

```
$ git log
```

你应该看到：

```
$ git log
commit 1f7ec5eaa8f37c2770dae3b984c55a1531fcc9e7
Author: Jim Weirich <jim (at) neo.com>
Date:   Sat Apr 13 15:20:42 2013 -0400

    Added a comment

commit 582495ae59ca91bca156a3372a72f88f6261698b
Author: Jim Weirich <jim (at) neo.com>
Date:   Sat Apr 13 15:20:42 2013 -0400

    Added a default value

commit 323e28d99a07d404c04f27eb6e415d4b8ab1d615
Author: Jim Weirich <jim (at) neo.com>
Date:   Sat Apr 13 15:20:42 2013 -0400

    Using ARGV

commit 94164160adf8faa3119b409fcfcd13d0a0eb8020
Author: Jim Weirich <jim (at) neo.com>
Date:   Sat Apr 13 15:20:42 2013 -0400

    First Commit
```

这份清单是迄今为止我们对仓库所作的总共 4 次提交。

单行历史

你可以很好的控制处理 `log` 命令要精确显示的内容。我喜欢单行格式：

```
$ git log --pretty=oneline
```

你应该看到：

```
$ git log --pretty=oneline
1f7ec5eaa8f37c2770dae3b984c55a1531fcc9e7 Added a comment
582495ae59ca91bca156a3372a72f88f6261698b Added a default value
323e28d99a07d404c04f27eb6e415d4b8ab1d615 Using ARGV
94164160adf8faa3119b409fcfcd13d0a0eb8020 First Commit
```

控制显示哪个条目

`log` 命令有许多选项用来选择显示哪个条目。玩玩下面的选项：

```
$ git log --pretty=oneline --max-count=2
$ git log --pretty=oneline --since='5 minutes ago'
$ git log --pretty=oneline --until='5 minutes ago'
$ git log --pretty=oneline --author=<your name>
$ git log --pretty=oneline --all
```

参阅 `man git-log` 了解更多细节。

更加漂亮

这是我用来复查上周所做更改的命令。如果我只想看自己所作的更改，那么我将添加 `--author=jim`。

```
$ git log --all --pretty=format:'%h %cd %s (%an)' --since='7 days ago'
```

终极日志格式

随着时间的推移，我发现在工作时最喜欢下列日志格式。

```
$ git log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short
```

它看起来像这样：

```
$ git log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short
* 1f7ec5e 2013-04-13 | Added a comment (HEAD, master) [Jim Weirich]
* 582495a 2013-04-13 | Added a default value [Jim Weirich]
* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]
* 9416416 2013-04-13 | First Commit [Jim Weirich]
```

让我们看一下细节：

- `--pretty="..."` 定义输出的格式

- `%h` 是提交 hash 的缩写
- `%d` 是提交的装饰（如分支头或标签）
- `%ad` 是创作日期
- `%s` 是注释
- `%an` 是作者姓名
- `--graph` 使用 ASCII 图形布局显示提交树
- `--date=short` 保留日期格式更好且更短

其它工具

gitx（Mac）和 gitk（任意平台）在浏览日志历史时十分有用。

11

别名

目的

学习如何设置别名及简写 Git 命令。

常用别名

`git status`、`git add`、`git commit`、`git checkout` 是非常常用的命令，因此对它们进行缩写十分有用。

添加下列内容到你的 `$HOME` 目录的 `.gitconfig` 文件中：

```
[alias]
  co = checkout
  ci = commit
  st = status
  br = branch
  hist = log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short
  type = cat-file -t
  dump = cat-file -p
```

我们已经介绍了 `commit` 和 `status` 命令。并且在上一实验中也介绍了 `log` 命令。`checkout` 命令将接下来介绍。

使用这些在 `.gitconfig` 中定义的别名，你可以通过输入 `git co` 来表示 `git checkout`。同时，`git st` 表示 `git status`，而 `git ci` 表示 `git commit`。并且，最好的是 `git hist` 将使你避免很长的 `log` 命令。

去试试新命令吧。

在 .gitconfig 文件中定义 hist 别名

在大多数介绍中，我将继续输入完整的命令。唯一的例外是，当我需要看 `git log` 的输出时，我将使用上面定义的 `hist` 别名。如果你想要遵循这里，那么在继续前设置你的 `.gitconfig` 文件。

输入与转存

我们已经添加了几个还没有介绍的命令别名。`git branch` 命令很快将介绍。`git cat-file` 命令对于浏览 Git 很有用，一会儿我们将看看。

Shell 别名（可选）

注意：本小节是为那些运行 POSIX 类 Shell 的同学写的。Windows用户及其他非 POSIX Shell 用户可以跳到下一个实验。

如果你的 Shell 支持别名或简写，那么你可以添加一些别名。下面是我使用的：

文件：.profile

```
alias gs='git status '
alias ga='git add '
alias gb='git branch '
alias gc='git commit'
alias gd='git diff'
alias go='git checkout '
alias gk='gitk --all&'
alias gx='gitx --all'

alias got='git '
alias get='git '
```

`git checkout` 的缩写 `go` 尤其好，它允许我输入：

```
$ go <branch>
```

来检出一个特定的分支。

另外，我也经常通过创建足够的别名来避免打错 Git 命令。

注意：有些 Shell 别名有点攻击性。实际上，`gs` 将与 Linux GhostScript 程序冲突。最近我开始使用 Go 编程语言，因此必须禁用上面的 `go` 别名。所以使用这些别名要小心。

12

获得旧版本

目的

学习如何检出工作目录中的先前快照。

在历史中回忆很容易。`checkout` 命令将从仓库复制任意快照到工作目录。

获得先前版本的哈希

```
$ git hist
```

注意：你记得在 `.gitconfig` 文件中定义的 `hist`，对吗？如果不是这样，那么回顾一下有关别名的实验。

```
$ git hist
* 1f7ec5e 2013-04-13 | Added a comment (HEAD, master) [Jim Weirich]
* 582495a 2013-04-13 | Added a default value [Jim Weirich]
* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]
* 9416416 2013-04-13 | First Commit [Jim Weirich]
```

检查日志输出，并找到第一个提交的哈希。它应当在 `git hist` 输出的最后一行。在下面的命令中使用哈希码（前 7 个字符就够了）。然后检查 hello.rb 文件的内容。

```
$ git checkout <hash>
$ cat hello.rb
```

注意：这儿给的是 Unix 命令，适用于 Mac 和 Linux 系统。不幸的是，Windows 用户必须换成他们的原生命令。

注意：许多命令都需要仓库中的哈希值。因为你的哈希值将与我的不同，当你看到命令中的 `<hash>` 或 `<treehash>` 时，使用你仓库的正确哈希值替换。

你应该看到：

```
$ git checkout 9416416
Note: checking out '9416416'.

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_name
```

```
HEAD is now at 9416416... First Commit
$ cat hello.rb
puts "Hello, World"
```

在 Git 中的“detached HEAD”信息意味着 HEAD (Git 跟踪当前目录应当匹配的那部分)是直接指向提交而非分支。在你没有切换到不同的分支时，这种状态只会记得已经提交的更改。一旦你检出了新的分支或标签，分离的提交将被丢弃 (因为 HEAD 已经移走)。如果你想要保存在分离状态的提交，那么你需要创建分支来记住提交。

Git 的旧版本将抱怨分离的 HEAD 状态不在本地分支。现在不用担心这种情况。

注意 hello.rb 文件是最原始的内容。

回到在 master 分支中的最新版本

```
$ git checkout master
$ cat hello.rb
```

你应该看到：

```
$ git checkout master
Previous HEAD position was 9416416... First Commit
Switched to branch 'master'
$ cat hello.rb
 # Default is "World"

name = ARGV.first || "World"

puts "Hello, #{name}!"
```

`master` 是默认分支的名称。通过名称检出分支，你能够回到该分支的最新版本。

13

给版本打标签

目的

学习何使用名称给提交打标签以便将来参考。

让我们叫 hello 程序的当前版本为 version 1 (v1)。

标记 version 1

```
$ git tag v1
```

现在你可以引用程序的当前版本为 v1。

标记先前的版本

让我们标记当前版本之前的版本为 v1-beta。首先我们需要检出先前的版本。代替查询哈希，我们将使用 `^` 来表示“v1 的父提交”。

如果使用 `v1^` 表示法遇到问题，那么你也可以试试 `v1~1`，这将引用相同的版本。该表示法意为“v1 的第一个祖先提交”。

```
$ git checkout v1^
$ cat hello.rb
```

```
$ git checkout v1^
Note: checking out 'v1^'.

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_name

HEAD is now at 582495a... Added a default value
$ cat hello.rb
name = ARGV.first || "World"

puts "Hello, #{name}!"
```

看，这是我们添加注释之前的默认值版本。让我们使它成为 v1-beta。

```
$ git tag v1-beta
```

按标签名检出

现在试试在两个标记版本之间切换。

```
$ git checkout v1
$ git checkout v1-beta
```

```
$ git checkout v1
Previous HEAD position was 582495a... Added a default value
HEAD is now at 1f7ec5e... Added a comment
$ git checkout v1-beta
Previous HEAD position was 1f7ec5e... Added a comment
HEAD is now at 582495a... Added a default value
```

使用 tag 命令查看标签

你可以使用 `git tag` 命令来看看可用的标签有什么。

```
$ git tag
```

```
$ git tag
v1
v1-beta
```

查看日志中的标签

你也可以检查日志中的标签。

```
$ git hist master --all
```

```
$ git hist master --all
* 1f7ec5e 2013-04-13 | Added a comment (v1, master) [Jim Weirich]
* 582495a 2013-04-13 | Added a default value (HEAD, v1-beta) [Jim Weirich]
* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]
* 9416416 2013-04-13 | First Commit [Jim Weirich]
```

你可以在日志输出中看到与分支名称（master）一起列出了两个标签（v1 和 v1-beta）。而且 HEAD 显示你当前检出的提交（此刻是 v1-beta）。

14

撤销本地更改

目的

学习如何还原工作目录中的更改。

检出 Master

在处理之前确认你在 master 中的最新提交上。

```
$ git checkout master
```

更改 hello.rb

有时候你修改了本地工作目录中的文件，且想要还原已经提交的内容。`checkout` 命令可以用来处理这种情况。

更改 hello.rb 让其具有错误的注释。

```
 # This is a bad comment.  We want to revert it.
name = ARGV.first || "World"
puts "Hello, #{name}!"
```

检查状态

首先，检查工作目录的状态。

```
$ git status
```

```
$ git status
 # On branch 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:   hello.rb

 #
```

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

我们看到 hello.rb 已被修改，但还没有暂存。

还原工作目录中的更改

使用 `checkout` 命令来检出 hello.rb 在仓库中的版本。

```
$ git checkout hello.rb
$ git status
$ cat hello.rb
```

```
$ git checkout hello.rb
$ git status
 # On branch master

nothing to commit (working directory clean)
$ cat hello.rb
 # Default is "World"

name = ARGV.first || "World"

puts "Hello, #{name}!"
```

`status` 命令显示在工作目录中没有未完成的更改。而且“错误的注释”也不再成为文件内容的一部分。

15

撤销暂存的更改

目的

学习如何还原已经暂存的更改。

更改文件并暂存更改

修改 hello.rb 文件来包含一个错误的注释。

```
 # This is an unwanted but staged comment

name = ARGV.first || "World"

puts "Hello, #{name}!"
```

然后去暂存它。

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

检查状态

检查你不想要的更改状态。

```
$ git status
```

```
$ git status
 # On branch master

 # Changes to be committed:

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

 #

 #   modified:   hello.rb

 #
```

`status` 输出显示更改已被暂存且准备提交。

重置暂存区

幸运的是 `status` 输出告诉我们取消暂存更改时需要做什么。

```
$ git reset HEAD hello.rb
```

```
$ git reset HEAD hello.rb
Unstaged changes after reset:
M   hello.rb
```

`reset` 命令重置 HEAD 中暂存区的内容。这将清除我们已经暂存的更改。

`reset` 命令（默认）不会更改工作目录。所以在工作目录中仍然有不想要的注释。我们可以使用之前实验中的 `checkout` 命令来从工作目录移除不想要的更改。

检出提交的版本

```
$ git checkout hello.rb
$ git status
```

```
$ git status
 # On branch master

nothing to commit (working directory clean)
```

现在我们的工作目录又变干净了。

16

撤销提交的更改

目的

学习如何还原已经提交到本地仓库的更改。

撤销提交

有时候你意识到已经提交的更改不正确并想撤销该提交。有几种方式可以处理这种问题，我们在本实验中所用的方式总是安全的。

实际上我们将通过创建新的提交来撤销原来不想要更改的提交。

更改文件并提交

更改 hello.rb 文件成下列内容：

```
 # This is an unwanted but committed change

name = ARGV.first || "World"

puts "Hello, #{name}!"
```

```
$ git add hello.rb
$ git commit -m "Oops, we didn't want this commit"
```

创建还原提交

要撤销已提交的更改，我们需要创建一个提交来移除由不想要的提交所引入的更改。

```
$ git revert HEAD
```

这将带你到编辑器中。你可以编辑默认的提交信息，或直接离开它。保存并关闭文件。你应该看到：

```
$ git revert HEAD --no-edit
[master a10293f] Revert "Oops, we didn't want this commit"
 1 files changed, 1 insertions(+), 1 deletions(-)
```

因为我们将撤销我们做的最后提交，所以我们可以使用 `HEAD` 作为还原的参数。通过简单的指定哈希值，我们可以撤销早期历史中的任意提交。

注意：命令中的 `--no-edit` 可被忽略。在不打开编辑器生成输出时需要它。

检查日志

检查日志来显示我们仓库中不想要及还原的提交。

```
$ git hist
```

```
$ git hist
* a10293f 2013-04-13 | Revert "Oops, we didn't want this commit" (HEAD, master) [Jim Weirich]
* 838742c 2013-04-13 | Oops, we didn't want this commit [Jim Weirich]
* 1f7ec5e 2013-04-13 | Added a comment (v1) [Jim Weirich]
* 582495a 2013-04-13 | Added a default value (v1-beta) [Jim Weirich]
* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]
* 9416416 2013-04-13 | First Commit [Jim Weirich]
```

这种技术将处理任何提交（虽然你可能必须解决冲突）。在公开分享的远程仓库上使用分支更加安全。

下一步

接下来，让我们看看从仓库历史中移除最近提交所用的技术。

17

从分支移除提交

目的

学习如何从分支移除最近的提交。

上一小节的 `revert` 是一个让我们撤销仓库中的任意提交的强大命令。然而，原始提交和“撤销”提交在分支历史中都可见（使用 `git log` 命令）。

我们经常做提交，并很快意识到犯了错误。如果有一个“收回”命令能允许我们假装不正确的提交从未发生过该多好啊。“收回”命令甚至还会阻止错误的提交在 `git log` 历史中的显示。这就像错误的提交从未发生过一样。

重置命令

我们已经介绍过 `reset` 命令，并用它来设置暂存区以便与特定的提交保持一致（我们在之前的实验中使用 `HEAD` 提交）。

当给定提交引用（如哈希、分支或标签名）时，`reset` 命令将：

- 重写当前分支到指向的特定提交
- 重置暂存区到匹配特定的提交（可选）
- 重置工作目录到匹配特定的提交（可选）

检查历史

让我们快速的检查我们的提交历史。

```
$ git hist
```

```
$ git hist
* a10293f 2013-04-13 | Revert "Oops, we didn't want this commit" (HEAD, master) [Jim Weirich]
* 838742c 2013-04-13 | Oops, we didn't want this commit [Jim Weirich]
* 1f7ec5e 2013-04-13 | Added a comment (v1) [Jim Weirich]
* 582495a 2013-04-13 | Added a default value (v1-beta) [Jim Weirich]
* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]
* 9416416 2013-04-13 | First Commit [Jim Weirich]
```

我们看到在该分支中的最后两个提交为“Oops”和“Revert Oops”。让我们使用 `reset` 来移除它们。

首先，标记分支

但在我们移除提交前，让我们使用一个标签来标记最新的提交以便能够再次找到它。

```
$ git tag oops
```

重置到 Oops 前

看看上面的日志历史，我们将知道标记为“v1”的提交是错误提交之前的正确提交。让我们重置分支到该位置。因为分支已经标记，所以我们可以在 `reset` 命令中使用标签名（如果它没有被标记，那么我们只能使用哈希值）。

```
$ git reset --hard v1
$ git hist
```

```
$ git reset --hard v1
HEAD is now at 1f7ec5e Added a comment
$ git hist
* 1f7ec5e 2013-04-13 | Added a comment (HEAD, v1, master) [Jim Weirich]
* 582495a 2013-04-13 | Added a default value (v1-beta) [Jim Weirich]
* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]
* 9416416 2013-04-13 | First Commit [Jim Weirich]
```

我们的 master 分支现在指到 v1 提交，并且 Oops 和 Revert Oops 提交已经不在分支中。`--hard` 参数表示应当更新工作目录以便与新的分支头保持一致。

什么也没丢

但错误的提交发生了什么？结果是提交仍然在仓库中。事实上，我们仍然能够引用它们。记得在本实验开始我们使用标签“oops”标记了还原的提交。让我们看看所有的提交。

```
$ git hist --all
```

```
$ git hist --all
* a10293f 2013-04-13 | Revert "Oops, we didn't want this commit" (oops) [Jim Weirich]
* 838742c 2013-04-13 | Oops, we didn't want this commit [Jim Weirich]
* 1f7ec5e 2013-04-13 | Added a comment (HEAD, v1, master) [Jim Weirich]
* 582495a 2013-04-13 | Added a default value (v1-beta) [Jim Weirich]
* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]
* 9416416 2013-04-13 | First Commit [Jim Weirich]
```

在这儿我们看到错误的提交并没有消失。它们仍然在仓库中。它们只是不再列到 master 分支中。如果我们没有标记它们，它们依然在仓库中，但除了使用哈希值外没有别的方法引用它们。未引用的提交保留在仓库中，一直到系统运行垃圾回收软件时。

重置的危险性

在本地分支上重置一般是安全的。任何“事故”通常都能通过重置到想要的提交来恢复。

然而，如果分支在共享的远程仓库上，那么重置可能使其他用户共享的分支混乱。

18

移除 oops 标签

目的

移除 oops 标签（料理家务）。

移除 oops 标签

oops 标签已经实现其目的。让我们移除它，以便它引用的提交被作为垃圾回收。

```
$ git tag -d oops
$ git hist --all
```

```
$ git tag -d oops
Deleted tag 'oops' (was a10293f)
$ git hist --all
* 1f7ec5e 2013-04-13 | Added a comment (HEAD, v1, master) [Jim Weirich]
* 582495a 2013-04-13 | Added a default value (v1-beta) [Jim Weirich]
* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]
* 9416416 2013-04-13 | First Commit [Jim Weirich]
```

在仓库中不再列出 oops 标签了。

19

修正提交

目的

学习如何修正现有的提交。

更改程序并提交

给程序添加作者注释。

```
 # Default is World
 # Author: Jim Weirich
name = ARGV.first || "World"
puts "Hello, #{name}!"
```

```
$ git add hello.rb
$ git commit -m "Add an author comment"
```

唉，该有 Email 啊

在你做了提交之后，你意识到任何好的作者注释都应该包含 Email 地址。更新 hello 程序来包含 Email。

```
 # Default is World
 # Author: Jim Weirich (jim@somewhere.com)
name = ARGV.first || "World"
puts "Hello, #{name}!"
```

修正先前的提交

我们真的不想因为 Email 而分开提交。让我们修正先前的提交来包含 Email 更改。

```
$ git add hello.rb
$ git commit --amend -m "Add an author/email comment"
```

```
$ git add hello.rb
$ git commit --amend -m "Add an author/email comment"
[master eb30103] Add an author/email comment
 1 files changed, 2 insertions(+), 1 deletions(-)
```

回顾历史

```
$ git hist
```

```
$ git hist
* eb30103 2013-04-13 | Add an author/email comment (HEAD, master) [Jim Weirich]
* 1f7ec5e 2013-04-13 | Added a comment (v1) [Jim Weirich]
* 582495a 2013-04-13 | Added a default value (v1-beta) [Jim Weirich]
* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]
* 9416416 2013-04-13 | First Commit [Jim Weirich]
```

我们可以看到最初的“author”提交现在消失了，而且它已经被“author/email”提交替换。通过重置分支到某个提交并重新提交新的更改，你可以实现相同的效果。

20

移动文件

目的

学习如何移动在仓库里的文件。

将 hello.rb 文件移到 lib 目录

我们现在将构建我们的小仓库结构。让我们将程序移到 lib 目录。

```
$ mkdir lib
$ git mv hello.rb lib
$ git status
```

```
$ mkdir lib
$ git mv hello.rb lib
$ git status
 # On branch master

 # Changes to be committed:

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

 #

 #   renamed:    hello.rb -> lib/hello.rb

 #
```

通过使用 Git 来移动文件，我们通知了 Git 两件事：

1. 文件 hello.rb 已被删除。
2. 文件 lib/hello.rb 已被创建。

这些信息被立即暂存并准备提交。`git status` 命令将报告文件已被移动。

移动文件另一法

关于 Git 的好事之一是你可以暂时忘掉源码控制直到准备开始提交代码。如果我们使用系统命令代替 Git 命令来移动文件会发生什么呢？

请与我们执行的命令集保持一致。虽然工作有点多，但结果是相同。

我们已经完成：

```
$ mkdir lib
$ mv hello.rb lib
$ git add lib/hello.rb
$ git rm hello.rb
```

提交新的目录

让我们提交此次移动操作。

```
$ git commit -m "Moved hello.rb to lib"
```

21

再谈结构

目的

添加另外的文件到我们的仓库。

现在添加 Rakefile

让我们添加 Rakefile 到我们的仓库。下面一个恰好。

```
 #!/usr/bin/ruby -wKU


task :default => :run

task :run do
  require './lib/hello'
end
```

添加并提交更改。

```
$ git add Rakefile
$ git commit -m "Added a Rakefile."
```

现在你应当能够使用 Rake 来执行 hello 程序了。

```
$ rake
```

```
$ rake
Hello, World!
```

22

Git 内幕：.git 目录

目的

学习有关 .git 目录结构的内容。

.git 目录

是时候做些浏览了。首先，从你的项目根目录开始……

```
$ ls -C .git
```

```
$ ls -C .git
COMMIT_EDITMSG  ORIG_HEAD   hooks       logs        rr-cache
HEAD        config      index       objects
MERGE_RR    description info        refs
```

这是全部 Git 东东所存储的魔法目录。让我们一瞥对象目录。

对象存储

```
$ ls -C .git/objects
```

```
$ ls -C .git/objects
09  1f  27  43  69  83  97  af  e4  info
0f  22  28  58  6b  94  9c  b5  e7  pack
11  24  32  59  78  96  a1  c4  eb
```

你应当看到一串包含两个字符名称的目录。目录名称是 Git 中对象存储的 sha1 哈希的开头两个字符。

深入对象存储

```
$ ls -C .git/objects/<dir>
```

```
$ ls -C .git/objects/09
6b74c56bfc6b40e754fc0725b8c70b2038b91e  9fb6f9d3a104feb32fcac22354c4d0e8a182c1
```

看看两字符目录的其中之一。你应当看到一些具有 38 个字符名称的文件。这些是 Git 中包含对象存储的文件。这些文件已被压缩和编码，所以直接查看它们的内容并没有什么用处，但我们将看一点。

配置文件

```
$ cat .git/config
```

```
$ cat .git/config
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
    ignorecase = true
[user]
    name = Jim Weirich
    email = jim (at) neo.com
```

这是项目级配置文件。在这儿的配置条目将覆盖你的主目录中 .gitconfig 文件中的配置条目，至少对此项目来说是如此。

分支与标签

```
$ ls .git/refs
$ ls .git/refs/heads
$ ls .git/refs/tags
$ cat .git/refs/tags/v1
```

```
$ ls .git/refs
heads
tags
$ ls .git/refs/heads
master
$ ls .git/refs/tags
v1
v1-beta
$ cat .git/refs/tags/v1
1f7ec5eaa8f37c2770dae3b984c55a1531fcc9e7
```

你应当认识标签子目录中的文件。每个文件都与你先前使用`git tag` 命令所创建的标签相应。它的内容是绑定到标签的提交哈希。

`heads` 目录与此相似，但它是用于分支而非标签。现在我们只有一个分支，所以你在该目录中只会看到 master。

HEAD 文件

```
$ cat .git/HEAD
```

```
$ cat .git/HEAD
ref: refs/heads/master
```

HEAD 文件包含当前分支的引用。此刻它是对于 master 的引用。

23

Git 内幕：直接处理 Git 对象

目的

浏览对象存储的结构。学习如何使用 SHA1 哈希来查找仓库中的内容。

现在让我们使用一些工具来直接探究 Git 对象。

查找最新的提交

```
$ git hist --max-count=1
```

这应当显示仓库中所做的最新提交。在你的系统中的 SHA1 哈希也许与我的不同，但应该看起来类似。

```
$ git hist --max-count=1
* 96ee164 2013-04-13 | Added a Rakefile. (HEAD, master) [Jim Weirich]
```

转存最新的提交

使用上面所列提交的 SHA1 哈希。

```
$ git cat-file -t <hash>
$ git cat-file -p <hash>
```

这儿是我的输出：

```
git cat-file -t 96ee164
commit
$ git cat-file -p 96ee164
tree 096b74c56bfc6b40e754fc0725b8c70b2038b91e
parent 0f36766e05bc55d765ec8afe288430edc69fceea
author Jim Weirich <jim (at) neo.com> 1365880844 -0400
committer Jim Weirich <jim (at) neo.com> 1365880844 -0400

Added a Rakefile.
```

注意：如果你在别名实验中定义了 `type` 和 `dump` 别名，那么你可以输入 `git type` 和 `git dump`，而不是更长的 `cat-file` 命令（我从未记住过）。

这是 `master` 分支头提交对象的转存结果。它看起来很像先前介绍的提交对象。

查找 Tree

我们可以转存提交中的目录树引用。这应当是我们项目中的文件的说明。使用上面所列“tree”那行的 SHA1 哈希。

```
$ git cat-file -p <treehash>
```

这儿是我的目录树看起来的样子……

```
$ git cat-file -p 096b74c
100644 blob 28e0e9d6ea7e25f35ec64a43f569b550e8386f90    Rakefile
040000 tree e46f374f5b36c6f02fb3e9e922b79044f754d795    lib
```

是的，我看到了 Rakefile 和 lib 目录。

转存 lib 目录

```
$ git cat-file -p <libhash>
```

```
$ git cat-file -p e46f374
100644 blob c45f26b6fdc7db6ba779fc4c385d9d24fc12cf72    hello.rb
```

这是 hello.rb 文件。

转存 hello.rb 文件

```
$ git cat-file -p <rbhash>
```

```
$ git cat-file -p c45f26b
 # Default is World

 # Author: Jim Weirich (jim@somewhere.com)

name = ARGV.first || "World"

puts "Hello, #{name}!"
```

你已经有它了。我们直接从 Git 仓库转存了提交对象、树对象、以及 blob 对象。blob、树及提交就是全部了。

浏览你自己的 Git 仓库

手动浏览你自己的 Git 仓库。看看是否能通过遵循最新提交的 SHA1 哈希引用来从第一个提交找出最初的 hello.rb 文件。

24

创建分支

目的

学习如何在仓库中创建本地分支。

是时候重写“hello world”的主要功能了。因为这可能会花一会儿时间，所以你可能想要把这些更改放到一个独立的分支，以便与 master 中的更改隔开。

创建分支

让我们叫新的分支为 `greet`。

```
$ git checkout -b greet
$ git status
```

注意：`git checkout -b <branchname>` 是 `git branch <branchname>` 及 `git checkout <branchname>` 的简写。

注意 `git status` 命令报告你在 `greet` 分支。

更改 Greet：添加 Greeter 类

文件：lib/greeter.rb

```
class Greeter
  def initialize(who)
    @who = who
  end
  def greet
    "Hello, #{@who}"
  end
end
```

```
$ git add lib/greeter.rb
$ git commit -m "Added greeter class"
```

更改 Greet：修改主程序

更新 hello.rb 文件来使用 greeter。

```
require 'greeter'

 # Default is World

name = ARGV.first || "World"

greeter = Greeter.new(name)
puts greeter.greet
```

```
$ git add lib/hello.rb
$ git commit -m "Hello uses Greeter"
```

更改 Greet：更新 Rakefile

更新 Rakefile 来使用外部的 Ruby 进程。

```
 #!/usr/bin/ruby -wKU


task :default => :run

task :run do
  ruby '-Ilib', 'lib/hello.rb'
end
```

```
$ git add Rakefile
$ git commit -m "Updated Rakefile"
```

下一步

我们现在已经有了包含 3 个新提交的 greet 新分支。接下来我们将学习如何导航及切换分支。

25

导航分支

目的

学习如何导航仓库的分支。

现在在你的项目中具有两个分支：

```
$ git hist --all
```

```
$ git hist --all
* 28917a4 2013-04-13 | Updated Rakefile (HEAD, greet) [Jim Weirich]
* 4dac415 2013-04-13 | Hello uses Greeter [Jim Weirich]
* 39347b3 2013-04-13 | Added greeter class [Jim Weirich]
* 96ee164 2013-04-13 | Added a Rakefile. (master) [Jim Weirich]
* 0f36766 2013-04-13 | Moved hello.rb to lib [Jim Weirich]
* eb30103 2013-04-13 | Add an author/email comment [Jim Weirich]
* 1f7ec5e 2013-04-13 | Added a comment (v1) [Jim Weirich]
* 582495a 2013-04-13 | Added a default value (v1-beta) [Jim Weirich]
* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]
* 9416416 2013-04-13 | First Commit [Jim Weirich]
```

切换到 master 分支

要在分支间切换，只需使用 `git checkout` 命令。

```
$ git checkout master
$ cat lib/hello.rb
```

```
$ git checkout master
Switched to branch 'master'
$ cat lib/hello.rb
 # Default is World

 # Author: Jim Weirich (jim@somewhere.com)

name = ARGV.first || "World"

puts "Hello, #{name}!"
```

你现在在 master 分支上。你可以判断 hello.rb 文件是否使用 Greeter 类。

回到 Greet 分支

```
$ git checkout greet
$ cat lib/hello.rb
```

```
$ git checkout greet
Switched to branch 'greet'
$ cat lib/hello.rb
require 'greeter'

 # Default is World

name = ARGV.first || "World"

greeter = Greeter.new(name)
puts greeter.greet
```

`lib/hello.rb` 的内容确定我们回到了 greet 分支上。

26

在 master 中更改

目的

学习如何处理包含不同（及可能有冲突）更改的多个分支。

当你更改 greet 分支时，其他人决定更新 master 分支。他们添加了一个 README。

切换到 master 分支

```
$ git checkout master
```

创建 README

```
This is the Hello World example from the git tutorial.
```

提交 README 到 master

```
$ git add README
$ git commit -m "Added README"
```

27

查看分叉的分支

目的

学习如何查看仓库中分叉的分支。

查看当前分支

在仓库中我们现在有两个分叉的分支。使用下面的 `log` 命令来查看分支及它们如何分叉。

```
$ git hist --all
```

```
$ git hist --all
* b59a8c2 2013-04-13 | Added README (HEAD, master) [Jim Weirich]
| * 28917a4 2013-04-13 | Updated Rakefile (greet) [Jim Weirich]
| * 4dac415 2013-04-13 | Hello uses Greeter [Jim Weirich]
| * 39347b3 2013-04-13 | Added greeter class [Jim Weirich]
|/
* 96ee164 2013-04-13 | Added a Rakefile. [Jim Weirich]
* 0f36766 2013-04-13 | Moved hello.rb to lib [Jim Weirich]
* eb30103 2013-04-13 | Add an author/email comment [Jim Weirich]
* 1f7ec5e 2013-04-13 | Added a comment (v1) [Jim Weirich]
* 582495a 2013-04-13 | Added a default value (v1-beta) [Jim Weirich]
* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]
* 9416416 2013-04-13 | First Commit [Jim Weirich]
```

在此我们第一次有机会看到 `git hist` 中 `--graph` 选项的效果。添加 `--graph` 到 `git hist` 使它能够使用简单的 ASCII 字符来绘制提交树。我们可以看到两个分支（greet 和 master），并且 master 分支是当前的 HEAD。两个分支的共同祖先是“Added a Rakefile”分支。

`--all` 选项确使我们看到所有分支。默认只显示当前分支。

28

合并

目的

学习如何合并两个分叉的分支以便将更改带回到单一分支。

合并分支

合并将两个分支中的更改结合在一起。让我们回到 greet 分支，并将 master 合并过来。

```
$ git checkout greet
$ git merge master
$ git hist --all
```

```
$ git checkout greet
Switched to branch 'greet'
$ git merge master
Merge made by recursive.
 README |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 README
$ git hist --all
*   844d1ed 2013-04-13 | Merge branch 'master' into greet (HEAD, greet) [Jim Weirich]
|\
| * b59a8c2 2013-04-13 | Added README (master) [Jim Weirich]
* | 28917a4 2013-04-13 | Updated Rakefile [Jim Weirich]
* | 4dac415 2013-04-13 | Hello uses Greeter [Jim Weirich]
* | 39347b3 2013-04-13 | Added greeter class [Jim Weirich]
|/
* 96ee164 2013-04-13 | Added a Rakefile. [Jim Weirich]
* 0f36766 2013-04-13 | Moved hello.rb to lib [Jim Weirich]
* eb30103 2013-04-13 | Add an author/email comment [Jim Weirich]
* 1f7ec5e 2013-04-13 | Added a comment (v1) [Jim Weirich]
* 582495a 2013-04-13 | Added a default value (v1-beta) [Jim Weirich]
* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]
* 9416416 2013-04-13 | First Commit [Jim Weirich]
```

通过周期性地合并 master 到 greet 分支，你可以选择 master 的任意更改，并保持 greet 中的更改与主干中的更改兼容。

然而，这会产生丑陋的提交图。稍后我们将看看变基替代合并这个选项。

下一步

如果 master 中的更改与 greet 中的冲突怎么办？

29

创建冲突

目的

在 master 分支中创建一个冲突更改。

切换到 master 并创建冲突

切换到 master 分支并做这个更改：

```
$ git checkout master
```

文件：lib/hello.rb

```
puts "What's your name"
my_name = gets.strip

puts "Hello, #{my_name}!"
```

```
$ git add lib/hello.rb
$ git commit -m "Made interactive"
```

查看分支

```
$ git hist --all
```

```
$ git hist --all
*   844d1ed 2013-04-13 | Merge branch 'master' into greet (greet) [Jim Weirich]
|\
* | 28917a4 2013-04-13 | Updated Rakefile [Jim Weirich]
* | 4dac415 2013-04-13 | Hello uses Greeter [Jim Weirich]
* | 39347b3 2013-04-13 | Added greeter class [Jim Weirich]
| | * 05f32c0 2013-04-13 | Made interactive (HEAD, master) [Jim Weirich]
| |/
| * b59a8c2 2013-04-13 | Added README [Jim Weirich]
|/
* 96ee164 2013-04-13 | Added a Rakefile. [Jim Weirich]
* 0f36766 2013-04-13 | Moved hello.rb to lib [Jim Weirich]
* eb30103 2013-04-13 | Add an author/email comment [Jim Weirich]
* 1f7ec5e 2013-04-13 | Added a comment (v1) [Jim Weirich]
* 582495a 2013-04-13 | Added a default value (v1-beta) [Jim Weirich]
* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]
* 9416416 2013-04-13 | First Commit [Jim Weirich]
```

master 在提交“Added README”时合并到 greet 分支，但现在 master 上有一个另外的提交还没有合并回 greet。

下一步

在 master 中的最新更改与 greet 中现有的更改冲突了。接下来我们将解决这些更改。

30

解决冲突

目的

学习如何在合并时处理冲突。

合并 master 到 greet

现在回到 greet 分支，并尝试合并新的 master。

```
$ git checkout greet
$ git merge master
```

```
$ git checkout greet
Switched to branch 'greet'
$ git merge master
Auto-merging lib/hello.rb
CONFLICT (content): Merge conflict in lib/hello.rb
Automatic merge failed; fix conflicts and then commit the result.
```

如果你打开 `lib/hello.rb`，那么你将看到：

```
<<<<<<< HEAD
require 'greeter'

 # Default is World

name = ARGV.first || "World"

greeter = Greeter.new(name)
# puts greeter.greet # Default is World


puts "What's your name"
my_name = gets.strip

puts "Hello, #{my_name}!"
>>>>>>> master
```

第一部分是当前分支（greet）头的版本。第二部分是 master 分支的版本。

修复冲突

你需要手动解决冲突。根据下列内容来修改 `lib/hello.rb`。

```
require 'greeter'

puts "What's your name"
my_name = gets.strip
```

```
greeter = Greeter.new(my_name)
puts greeter.greet
```

提交冲突解决

```
$ git add lib/hello.rb
$ git commit -m "Merged master fixed conflict."
```

```
$ git add lib/hello.rb
$ git commit -m "Merged master fixed conflict."
Recorded resolution for 'lib/hello.rb'.
[greet 25f0e8c] Merged master fixed conflict.
```

高级合并

Git 没有提供任何图形化的合并工具，但如果你想要使用第三方合并工具来处理，它将十分乐意。参阅http://onestepback.org/index.cgi/Tech/Git/UsingP4MergeWithGit.red了解 Git 使用 Perforce 合并工具的说明。

31

变基 VS 合并

目的

学习变基与合并之间的差异。

讨论

让我们来探讨下合并与变基之间的差异。为了实现这个目的， 我们需要先回到第一次合并前的仓库时间点，然后重做同样 的步骤，并且使用变基代替合并。

我们将使用 `reset` 命令来倒回分支中的时间点。

32

重置 greet 分支

目的

重置 greet 分支到第一次合并前的地方。

重置 greet 分支

让我们回到 greet 分支在合并 master 之前的时间点。我们可以重置分支到我们想要的任何提交处。重点是修改分支指针以指向提交树中的任何位置。

在本实验中，我们想要回到 greet 在与 master 合并之前的地方。我们需要找到合并前的最后一个提交。

```
$ git checkout greet
$ git hist
```

```
$ git checkout greet
Already on 'greet'
$ git hist
*   25f0e8c 2013-04-13 | Merged master fixed conflict. (HEAD, greet) [Jim Weirich]
|\
| * 05f32c0 2013-04-13 | Made interactive (master) [Jim Weirich]
* |   844d1ed 2013-04-13 | Merge branch 'master' into greet [Jim Weirich]
|\ \
| |/
| * b59a8c2 2013-04-13 | Added README [Jim Weirich]
* | 28917a4 2013-04-13 | Updated Rakefile [Jim Weirich]
* | 4dac415 2013-04-13 | Hello uses Greeter [Jim Weirich]
* | 39347b3 2013-04-13 | Added greeter class [Jim Weirich]
|/
* 96ee164 2013-04-13 | Added a Rakefile. [Jim Weirich]
* 0f36766 2013-04-13 | Moved hello.rb to lib [Jim Weirich]
* eb30103 2013-04-13 | Add an author/email comment [Jim Weirich]
* 1f7ec5e 2013-04-13 | Added a comment (v1) [Jim Weirich]
* 582495a 2013-04-13 | Added a default value (v1-beta) [Jim Weirich]
* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]
* 9416416 2013-04-13 | First Commit [Jim Weirich]
```

有点难以阅读。通过查看数据我们发现“Updated Rakefile”是 greet 分支在合并前的最后一个提交。让我们重置 greet 分支到该提交处。

```
$ git reset --hard <hash>
```

```
$ git reset --hard 28917a4
HEAD is now at 28917a4 Updated Rakefile
```

检查分支

看看 greet 分支的日志。在它的历史中，我们不再有合并提交。

```
$ git hist --all
```

```
$ git hist --all
* 05f32c0 2013-04-13 | Made interactive (master) [Jim Weirich]
* b59a8c2 2013-04-13 | Added README [Jim Weirich]
| * 28917a4 2013-04-13 | Updated Rakefile (HEAD, greet) [Jim Weirich]
| * 4dac415 2013-04-13 | Hello uses Greeter [Jim Weirich]
| * 39347b3 2013-04-13 | Added greeter class [Jim Weirich]
|/
* 96ee164 2013-04-13 | Added a Rakefile. [Jim Weirich]
* 0f36766 2013-04-13 | Moved hello.rb to lib [Jim Weirich]
* eb30103 2013-04-13 | Add an author/email comment [Jim Weirich]
* 1f7ec5e 2013-04-13 | Added a comment (v1) [Jim Weirich]
* 582495a 2013-04-13 | Added a default value (v1-beta) [Jim Weirich]
* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]
* 9416416 2013-04-13 | First Commit [Jim Weirich]
```

33

重置 master 分支

目的

重置 master 分支到冲突提交前的地方。

重置 master 分支

当我们添加交互模式到 master 分支时，我们所做的更改与 greet 分支中的更改冲突了。让我们倒回 master 分支中冲突更改前的地方。这允许我们来演示变基命令而不必担心冲突。

```
$ git checkout master
$ git hist
```

```
$ git hist
* 05f32c0 2013-04-13 | Made interactive (HEAD, master) [Jim Weirich]
* b59a8c2 2013-04-13 | Added README [Jim Weirich]
* 96ee164 2013-04-13 | Added a Rakefile. [Jim Weirich]
* 0f36766 2013-04-13 | Moved hello.rb to lib [Jim Weirich]
* eb30103 2013-04-13 | Add an author/email comment [Jim Weirich]
* 1f7ec5e 2013-04-13 | Added a comment (v1) [Jim Weirich]
* 582495a 2013-04-13 | Added a default value (v1-beta) [Jim Weirich]
* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]
* 9416416 2013-04-13 | First Commit [Jim Weirich]
```

“Added README”是冲突的交互模式之前的提交。我们将重置 master 分支到“Added README”提交处。

```
$ git reset --hard <hash>
$ git hist --all
```

回顾下日志，应当看到仓库已经回到我们合并之前的时间点。

```
$ git hist --all
* b59a8c2 2013-04-13 | Added README (HEAD, master) [Jim Weirich]
| * 28917a4 2013-04-13 | Updated Rakefile (greet) [Jim Weirich]
| * 4dac415 2013-04-13 | Hello uses Greeter [Jim Weirich]
| * 39347b3 2013-04-13 | Added greeter class [Jim Weirich]
|/
* 96ee164 2013-04-13 | Added a Rakefile. [Jim Weirich]
* 0f36766 2013-04-13 | Moved hello.rb to lib [Jim Weirich]
* eb30103 2013-04-13 | Add an author/email comment [Jim Weirich]
* 1f7ec5e 2013-04-13 | Added a comment (v1) [Jim Weirich]
* 582495a 2013-04-13 | Added a default value (v1-beta) [Jim Weirich]
* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]
* 9416416 2013-04-13 | First Commit [Jim Weirich]
```

34

变基

目的

使用 `rebase` 命令代替 `merge` 命令。

好，我们回到了第一次合并前的时间点，并且我们想要将 master 中的更改集成到 greet 分支。

这次我们将使用 `rebase` 命令代替 `merge` 命令来从 master 分支中引入更改。

```
$ git checkout greet
$ git rebase master
$ git hist
```

```
$ go greet
Switched to branch 'greet'
$
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added Greeter class
Applying: hello uses Greeter
Applying: updated Rakefile
$
$ git hist
* 2fae0b2 2013-04-13 | Updated Rakefile (HEAD, greet) [Jim Weirich]
* 1c23048 2013-04-13 | Hello uses Greeter [Jim Weirich]
* 62d7ce0 2013-04-13 | Added greeter class [Jim Weirich]
* b59a8c2 2013-04-13 | Added README (master) [Jim Weirich]
* 96ee164 2013-04-13 | Added a Rakefile. [Jim Weirich]
* 0f36766 2013-04-13 | Moved hello.rb to lib [Jim Weirich]
* eb30103 2013-04-13 | Add an author/email comment [Jim Weirich]
* 1f7ec5e 2013-04-13 | Added a comment (v1) [Jim Weirich]
* 582495a 2013-04-13 | Added a default value (v1-beta) [Jim Weirich]
* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]
* 9416416 2013-04-13 | First Commit [Jim Weirich]
```

合并 VS 变基

变基的最终结果与合并很相似。greet 分支现在包含它的全部更改以及来自 master 分支中的所有更改。然而，提交树却十分不同。greet 分支的提交树已被重写，以致 master 分支成为了其提交历史的一部分。这使提交链更加线性，且更易阅读。

何时变基，何时合并?

不要使用变基……

如果是公开且与其他人共享的分支，那么重写公开的共享分支将会搞砸团队中的其他会员。

要是提交分支的精确历史重要（因为变基将重写提交历史）。

根据上述准则，我会针对短期生命的本地分支使用变基，而对公开仓库的分支使用合并。

35

合并回 master

目的

我们已经保持 greet 分支与 master 最新（通过变基），现在让我们合并 greet 中的更改回到 master 分支。

合并 greet 到 master 中

```
$ git checkout master
$ git merge greet
```

```
$ git checkout master
Switched to branch 'master'
$
$ git merge greet
Updating b59a8c2..2fae0b2
Fast-forward
 Rakefile       |    2 +-
 lib/greeter.rb |    8 ++++++++
 lib/hello.rb   |    6 ++++--
 3 files changed, 13 insertions(+), 3 deletions(-)
 create mode 100644 lib/greeter.rb
```

因为 master 的头是 greet 分支头的直接祖先，所以 Git 可以做快进合并。当快进时，分支指针简单地前进到与 greet 分支相同的提交处。

在快进合并中从来不会冲突。

回顾日志

```
$ git hist
```

```
$ git hist
* 2fae0b2 2013-04-13 | Updated Rakefile (HEAD, master, greet) [Jim Weirich]
* 1c23048 2013-04-13 | Hello uses Greeter [Jim Weirich]
* 62d7ce0 2013-04-13 | Added greeter class [Jim Weirich]
* b59a8c2 2013-04-13 | Added README [Jim Weirich]
* 96ee164 2013-04-13 | Added a Rakefile. [Jim Weirich]
* 0f36766 2013-04-13 | Moved hello.rb to lib [Jim Weirich]
* eb30103 2013-04-13 | Add an author/email comment [Jim Weirich]
* 1f7ec5e 2013-04-13 | Added a comment (v1) [Jim Weirich]
* 582495a 2013-04-13 | Added a default value (v1-beta) [Jim Weirich]
* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]
* 9416416 2013-04-13 | First Commit [Jim Weirich]
```

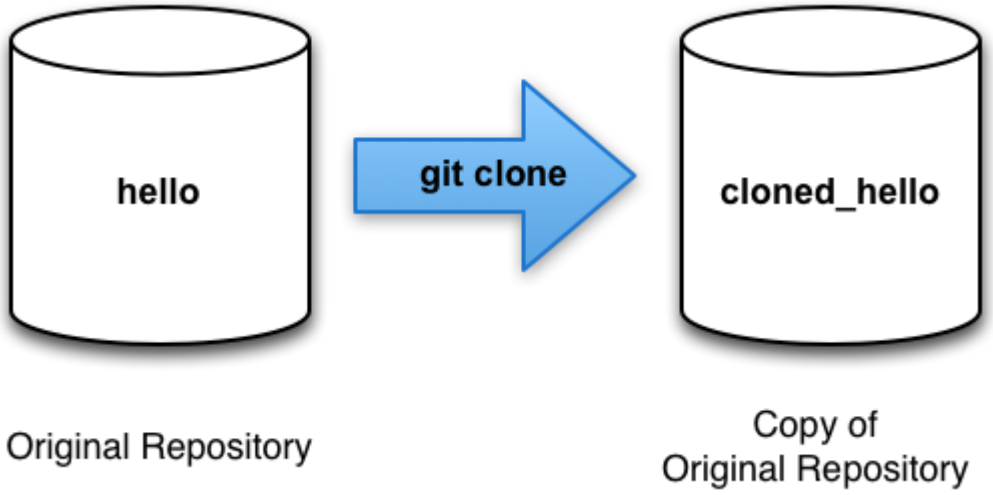
greet 和 master 分支现在相同了。

36

多个仓库

直到此时我们处理的都是单一 Git 仓库。然而，Git 也擅长处理多个仓库。这些扩展的仓库也许存储在本地，也许通过网络连接访问。

在下一节我们将创建名为“cloned_hello”的新仓库。我们将展示如何将更改从一个仓库迁移到另一个仓库，以及如何处理两个仓库之间出现的冲突。



图片 36.1 clone

现在，我们将处理本地仓库（如：存储在本机硬盘上的仓库），然而从本节学到的多数内容同样可以应用到多个仓库，无论它们是存储在本地，还是通过网络远程访问。

注意：我们将对仓库的拷贝做更改，确认在后续实验中的每个步骤你在哪个仓库。

37

克隆仓库

目的

学习如何创建仓库的拷贝。

转到工作目录

转到工作目录并创建 hello 仓库的克隆。

```
$ cd ..
$ pwd
$ ls
```

注意：现在在工作目录中。

```
$ cd ..
$ pwd
/Users/jim/working/git/git_immersion/auto
$ ls
hello
```

此刻你应当在你的工作目录，且有名为“hello”的单一仓库。

创建 hello 仓库的克隆

让我们创建仓库的克隆。

```
$ git clone hello cloned_hello
$ ls
```

```
$ git clone hello cloned_hello
Cloning into cloned_hello...
done.
$ ls
cloned_hello
hello
```

在你的工作目录中现在应当有两个仓库：原始的“hello”仓库和新克隆的“cloned_hello”仓库。

38

回顾克隆的仓库

目的

学习远程仓库上有关分支的内容。

查看克隆的仓库

让我们看一看克隆的仓库。

```
$ cd cloned_hello
$ ls
```

```
$ cd cloned_hello
$ ls
README
Rakefile
lib
```

你应当看到原始仓库的顶层目录中所有文件的列表 (README、Rakefile 及 lib)。

回顾仓库历史

```
$ git hist --all
```

```
$ git hist --all
* 2fae0b2 2013-04-13 | Updated Rakefile (HEAD, origin/master, origin/greet, origin/HEAD, master) [Jim Weirich]
* 1c23048 2013-04-13 | Hello uses Greeter [Jim Weirich]
* 62d7ce0 2013-04-13 | Added greeter class [Jim Weirich]
* b59a8c2 2013-04-13 | Added README [Jim Weirich]
* 96ee164 2013-04-13 | Added a Rakefile. [Jim Weirich]
* 0f36766 2013-04-13 | Moved hello.rb to lib [Jim Weirich]
* eb30103 2013-04-13 | Add an author/email comment [Jim Weirich]
* 1f7ec5e 2013-04-13 | Added a comment (v1) [Jim Weirich]
* 582495a 2013-04-13 | Added a default value (v1-beta) [Jim Weirich]
* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]
* 9416416 2013-04-13 | First Commit [Jim Weirich]
```

现在你应当看到新仓库的全部提交列表，而且它或多或少会匹配原始仓库的提交历史。仅有的差异体现在分支名称上。

远程分支

你应当在历史列表中看到 master 分支 (孤单的 HEAD)。而且你也将有许多奇怪的分支名称 (origin/master、origin/greet 及 origin/HEAD)。我们一会儿将讨论它们。

39

何为 Origin

目的

学习有关命名远程仓库的内容。

```
$ git remote
```

```
$ git remote
origin
```

我们看到克隆的仓库知道远程仓库名为 origin。让我们看看是否能获得有关 origin 的更多信息：

```
$ git remote show origin
```

```
$ git remote show origin
* remote origin
  Fetch URL: /Users/jim/working/git/git_immersion/auto/hello
  Push  URL: /Users/jim/working/git/git_immersion/auto/hello
  HEAD branch (remote HEAD is ambiguous, may be one of the following):
    greet
    master
  Remote branches:
    greet  tracked
    master 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)
```

现在我们看到远程仓库“origin”简直就是原始的 hello 仓库。远程仓库典型地存在于可能是中央服务器的独立机器上。正如我们可以在此处看到的，它们可以指到同一机器上的仓库。关于名称“origin”没有什么特别的，对于主中央仓库（如果有的话）使用名称“origin”只是习惯的约定而已。

40

远程分支

目的

学习有关本地与远程分支的内容。

让我们看看在克隆的仓库中可用的分支。

```
$ git branch
```

```
$ git branch
* master
```

就是这样，只有 master 分支被列出来了。greet 分支在哪儿？`git branch` 命令默认只会列出本地分支。

列出远程分支

试试执行以下命令来看全部分支：

```
$ git branch -a
```

```
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/greet
  remotes/origin/master
```

Git 具有原始仓库的全部提交，但在远程仓库中的分支不会作为本地分支。如果我们想要 greet 分支，我们需要自行创建它。一会儿我们将看看如何做到。

41

更改原始仓库

目的

对原始仓库做些更改以便我们可以尝试拉下更改。

在原始的 hello 仓库中做更改

```
$ cd ../hello
 # (You should be in the original hello repository now)
```

注意：现在在 hello 仓库中。

对 README 做下列更改：

```
This is the Hello World example from the git tutorial.
(changed in original)
```

现在添加并提交此更改。

```
$ git add README
$ git commit -m "Changed README in original repo"
```

下一步

原始的仓库现在有克隆版本中所没有的更改。接下来我们将拉下这些更改到克隆的仓库中。

42

取得更改

目的

学习如何从远程仓库拉下更改。

```
$ cd ../cloned_hello
$ git fetch
$ git hist --all
```

注意：现在在 cloned_hello 仓库中。

```
$ git fetch
From /Users/jim/working/git/git_immersion/auto/hello
   2fae0b2..2e4c559  master     -> origin/master
$ git hist --all
* 2e4c559 2013-04-13 | Changed README in original repo (origin/master, origin/HEAD) [Jim Weirich]
* 2fae0b2 2013-04-13 | Updated Rakefile (HEAD, origin/greet, master) [Jim Weirich]
* 1c23048 2013-04-13 | Hello uses Greeter [Jim Weirich]
* 62d7ce0 2013-04-13 | Added greeter class [Jim Weirich]
* b59a8c2 2013-04-13 | Added README [Jim Weirich]
* 96ee164 2013-04-13 | Added a Rakefile. [Jim Weirich]
* 0f36766 2013-04-13 | Moved hello.rb to lib [Jim Weirich]
* eb30103 2013-04-13 | Add an author/email comment [Jim Weirich]
* 1f7ec5e 2013-04-13 | Added a comment (v1) [Jim Weirich]
* 582495a 2013-04-13 | Added a default value (v1-beta) [Jim Weirich]
* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]
* 9416416 2013-04-13 | First Commit [Jim Weirich]
```

在此刻，仓库具有来自原始仓库的全部提交，但它并没有整合到克隆仓库的本地分支中。

在上面的历史中找到“Changed README in original repo”。注意提交包括“origin/master”和“origin/HEAD”。

现在看看“Updated Rakefile”提交。你将看到本地 master分支指到了此提交，并非我们取得的新提交。

`git fetch` 命令的结果将从远程仓库取得新的提交，但它不会将这些提交合并到本地分支中。

检查 README

我们可以验证克隆的 README 没有被更改。

```
$ cat README
```

```
$ cat README
This is the Hello World example from the git tutorial.
```

看，没有更改。

43

合并拉下的更改

目的

学习将拉下的更改合并到当前分支及工作目录。

将取得的更改合并到本地 master

```
$ git merge origin/master
```

```
$ git merge origin/master
Updating 2fae0b2..2e4c559
Fast-forward
 README |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

再次检查 README

我们应当看到现在更改了。

```
$ cat README
```

```
$ cat README
This is the Hello World example from the git tutorial.
(changed in original)
```

这些是更改。即使 `git fetch` 不合并更改，然而我们仍然可以手动从远程仓库合并更改。

下一步

接下来让我们看看将 `fetch` 和 `merge` 组合成单一命令。

44

拉下更改

目的

学习 `git pull` 等价于 `git fetch` 和 `git merge`。

讨论

我们不会创建另外的更改过程并再次拉下它，而是做你想要知道的做法：

```
$ git pull
```

等价于以下两步：

```
$ git fetch
$ git merge origin/master
```

45

添加跟踪的分支

目的

学习如何添加跟踪远程分支的本地分支。

包含 `remotes/origin` 开头的分支是来自原始仓库的分支。注意你没有叫 greet 的分支，但它知道原始仓库有 greet 分支。

添加跟踪远程分支的本地分支

```
$ git branch --track greet origin/greet
$ git branch -a
$ git hist --max-count=2
```

```
$ git branch --track greet origin/greet
Branch greet set up to track remote branch greet from origin.
$ git branch -a
  greet
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/greet
  remotes/origin/master
$ git hist --max-count=2
* 2e4c559 2013-04-13 | Changed README in original repo (HEAD, origin/master, origin/HEAD, master) [Jim Weirich]
* 2fae0b2 2013-04-13 | Updated Rakefile (origin/greet, greet) [Jim Weirich]
```

我们现在可以在分支列表和日志中看到 greet 分支。

46

裸仓库

目的

学习如何创建裸仓库。

裸仓库（没有工作目录）通常用于共享。

创建裸仓库

```
$ cd ..
$ git clone --bare hello hello.git
$ ls hello.git
```

注意：现在在工作目录中。

```
$ git clone --bare hello hello.git
Cloning into bare repository hello.git...
done.
$ ls hello.git
HEAD
config
description
hooks
info
objects
packed-refs
refs
```

结尾带 `.git` 的仓库习惯约定是裸仓库。我们可以看到在 hello.git 仓库中没有工作目录。实际上，除了非裸仓库的.git 目录外什么也没有。

47

添加远程仓库

目的

将裸仓库作为远程仓库添加到我们的原始仓库中。

让我们添加 hello.git 到我们的原始仓库。

```
$ cd hello
$ git remote add shared ../hello.git
```

注意：现在在 hello 仓库中。

48

推送更改

目的

学习如何将更改推到远程仓库。

因为裸仓库通常共享在某种网络服务器上，所以一般很难转到仓库中并拉下更改。因此，我们需要将更改推到其它的仓库中。

让我们通过创建更改来开始推送。编辑 README 并提交它。

```
This is the Hello World example from the git tutorial.
(Changed in the original and pushed to shared)
```

```
$ git checkout master
$ git add README
$ git commit -m "Added shared comment to readme"
```

现在推送更改到共享的仓库。

```
$ git push shared master
```

`shared` 是接收我们推送更改的仓库名称。（记住，在上一实验中我们已经将它添加为远程分支。）

```
$ git push shared master
To ../hello.git
   2e4c559..3923dd5  master -> master
```

注意：我们必须明确地指定接收推送的分支名称 master。它可以设置为自动化，但我从未记住实现的命令。选择“Git Remote Branch”gem 更易管理远程分支。

49

拉下共享的更改

目的

学习如何从共享的仓库拉下更改。

迅速跳过克隆仓库，且让我们拉下刚才推送到共享仓库的更改。

```
$ cd ../cloned_hello
```

注意：现在在 cloned_hello 仓库中。

继续处理……

```
$ git remote add shared ../hello.git
$ git branch --track shared master
$ git pull shared master
$ cat README
```

50

托管你的 Git 仓库

目的

学习如何设置 Git 服务器以共享仓库。

有很多方式来通过网络共享 Git 仓库。此处是“急就章”方式。

启动 Git 服务器

```
 # (From the work directory)

$ git daemon --verbose --export-all --base-path=.
```

现在，在分开的终端窗口中，转到你的工作目录。

```
 # (From the work directory)

$ git clone git://localhost/hello.git network_hello
$ cd network_hello
$ ls
```

你应当看到 hello 项目的拷贝。

推送到 Git daemon

如果你想要推送到 Git daemon 仓库，添加 `--enable=receive-pack` 到 `git daemon` 命令。小心，因为此服务器没有授权，任何人都能推送到你的仓库。

51

共享仓库

目的

学习通过 WiFi 共享仓库。

看看你的邻居是否在运行 `git daemon`。交换 IP 地址，并看看是否能够从其他人的仓库拉下更改。

注意：gitjour gem 在共享 ad-hoc 仓库时真的有用。

52

高级/将来的主题

这里是一些你可能想要自己研究的主题：

- 还原提交的更改
- 跨操作系统行尾处理
- 远程服务器
- 协议
- SSH 设置
- 远程分支管理
- 查找有 Bug 的提交 (git bisect)
- 工作流
- 非命令行工具 (gitx、gitk、magit)
- 处理 GitHub