Git Community Book 中文版

The open Git resource pulled together by the whole community
社区精华 尽入囊中

AUTHORS

Thank these guys:

Alecs King (alecsk@gmail.com), Amos Waterland (apw@rossby.metr.ou.edu), Andrew Ruder (andy@aeruder.net), Andy Parkins (andyparkins@gmail.com), Arjen Laarhoven (arjen@yaph.org), Brian Hetro (whee@smaertness.net), Carl Worth (cworth@cworth.org), Christian Meder (chris@absolutegiganten.org), Dan McGee (dpmcgee@gmail.com), David Kastrup (dak@gnu.org), Dmitry V. Levin (ldv@altlinux.org), Francis Daly (francis@daoine.org), Gerrit Pape (pape@smarden.org), Greg Louis (glouis@dynamicro.ca), Gustaf Hendeby (hendeby@isy.liu.se), Horst H. von Brand (vonbrand@inf.utfsm.cl), J. Bruce Fields (bfields@fieldses.org), Jakub Narebski (jnareb@gmail.com), Jim Meyering (jim@meyering.net), Johan Herland (johan@herland.net), Johannes Schindelin (Johannes.Schindelin@gmx.de), Jon Loeliger (jdl@freescale.org), Josh Triplett (josh@freedesktop.org), Junio C Hamano (gitster@pobox.com), Linus Torvalds (torvalds@osdl.org), Lukas Sandström (lukass@etek.chalmers.se), Marcus Fritzsch (m@fritschy.de), Michael Coleman (tutufan@gmail.com), Michael Smith (msmith@cbnco.com), Mike Coleman (tutufan@gmail.com), Miklos Vajna (vmiklos@frugalware.org), Nicolas Pitre (nico@cam.org), Oliver Steele (steele@osteele.com), Paolo Ciarrocchi (paolo.ciarrocchi@gmail.com), Pavel Roskin (proski@gnu.org), Ralf Wildenhues (Ralf.Wildenhues@gmx.de), Robin Rosenberg (robin.rosenberg.lists@dewire.com), Santi Béjar (sbejar@gmail.com), Scott Chacon (schacon@gmail.com), Sergei Organov (osv@javad.com), Shawn Bohrer (shawn.bohrer@gmail.com), Shawn O. Pearce (spearce@spearce.org), Steffen Prohaska (prohaska@zib.de), Tom Prince (tom.prince@ualberta.net), William Pursell (bill.pursell@gmail.com), Yasushi SHOJI (yashi@atmark-techno.com)

英文版 MAINTAINER / EDITOR

Bug this guy:

Scott Chacon (schacon@gmail.com)

中文版 MAINTAINER / EDITOR

liuhui998 (liuhui998@gmail.com),(http://liuhui998.com)
liu Wei (liuw@liuw.name),(http://blog.liuw.name)
Wendal Chen(wendal1985@gmail.com),(http://sunfarms.net/myblog)
Jiancong Guo(guojiancong0121@gmail.com),(http://www.cbpm-gw.com)

中文版网址

http://gitbook.liuhui998.com

Chapter 1

介绍

欢迎使用GIT

欢迎来使用Git, 它是一个快速的分布式版本控制系统。

这本书的目的是为那些初学者尽快熟悉Git，提供了一个良好的起点。

此书将以介绍Git如何存储数据做为开始，让你了解它和其它版本控制系统有什么不同的背景。这大约要花你20分钟的时间。

接下来，我们会讲一些Git的**基本用法**，那些你将在90%的时间都在使用的命令。这些东东能给一个不错的使用的基础，也许这些命令就是你将使用的全部命令。这一节大约会你30分钟的时间来读。

其后，我们会讲一些稍微复杂的**Git中级用法**，这些用法也许会替换掉前面的基本用法。在你了解前面的基本用法后, 这些看起来像魔术一样的命令，你可能会用起来很爽。

如果前面的这些东东你都掌握了，我们就会讲**Git的高级用法**，这些高级用法也许大多数人很少使用，但是在特定的环境会非常有用。学习这些用法（命令）,你将能获得全面的日常Git知识； 你将能成为Git大师。

即然你学会了Git, 我们将会讲**在Git中工作**。我们将要学习 Git 配合脚本、部署工具、编辑器和其它工具一起工作。这一节将帮助你将Git 集成进你的工作环境。

最后我们会有一系列的文章：**low-level documentation**，这些可能对那些Git hacker 有用，包括Git 的内核和协议如何运作等等。

反馈与参与

如果你发现本书的中任何错误，或者你想参与进此书的编写，你可以给我写email schacon@gmail.com, 或者你也可以用git得到本书的原始 档案(source) http://github.com/schacon/gitbook, 然后给我发一个补丁(patch)或者一个pull请求

译者注:如果有哪位朋友发现中译本的错误,或者是想参加此书的翻译，也可以给我发email liuhui998@gmail.com, 或者你也可以用git得到本书的原始 档案(source) http://github.com/liuhui998/gitbook, 然后给我发一个补丁(patch)或者一个pull请求

参考

这本书由很多不同的资料汇聚起来，如果你更愿意阅读原始的文章和资料，下面提供了它们的url:

- Git User Manual

- The Git Tutorial
- The Git Tutorial pt 2
- "My Git Workflow" blog post

GIT对象模型

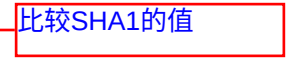
SHA

所有用来表示项目历史信息的文件,是通过一个40个字符的（40-digit）“对象名”来索引的，对象名看起来像这样:

```
6ff87c4664981e4397625791c8ea3bbb5f2279a3
```

你会在Git里到处看到这种“40个字符”字符串。每一个“对象名”都是对“对象”内容做SHA1哈希计算得来的，（SHA1是一种密码学的哈希算法）。这样就意味着两个不同内容的对象不可能有相同的“对象名”。

这样做会有几个好处：



- Git只要比较对象名，就可以很快的判断两个对象是否相同。
- 因为在每个仓库（repository）的“对象名”的计算方法都完全一样，如果同样的内容存在两个不同的仓库中，就会存在相同的“对象名”下。
- Git还可以通过检查对象内容的SHA1的哈希值和“对象名”是否相同，来判断对象内容是否正确。

对象

每个对象(object) 包括三个部分：**类型，大小和内容**。大小就是指内容的大小，内容取决于对象的类型，有四种类型的对象："blob"、"tree"、 "commit" 和"tag"。

一个指针

- **“blob”**用来存储文件数据，通常是一个文件。
- **“tree”**有点像一个目录，它管理一些**“tree”**或是 **“blob”**（就像文件和子目录）
- 一个**“commit”**只指向一个"tree"，它用来标记项目某一个特定时间点的状态。它包括一些关于时间点的元数据，如时间戳、最近一次提交的作者、指向上次提交（commits）的指针等等。
- 一个**“tag”**是来标记某一个提交(commit) 的方法。

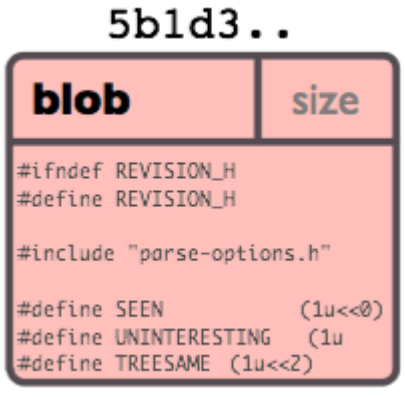
几乎所有的Git功能都是使用这四个简单的对象类型来完成的。它就像是在你本机的文件系统之上构建一个小的文件系统。

与SVN的区别

Git与你熟悉的大部分版本控制系统的差别是很大的。也许你熟悉Subversion、CVS、Perforce、Mercurial 等等，他们使用 *“增量文件系统”*（Delta Storage systems），就是说它们存储每次提交(commit)之间的差异。Git正好与之相反，它会把你的每次提交的文件的全部内容（snapshot）都会记录下来。这会是在使用Git时的一个很重要的理念。

Blob对象

一个blob通常用来存储文件的内容.



你可以使用git show命令来查看一个blob对象里的内容。假设我们现在有一个Blob对象的SHA1哈希值，我们可以通过下面的的命令来查看内容：

```
$ git show 6ff87c4664

 Note that the only valid version of the GPL as far as this project
 is concerned is _this_ particular version of the license (ie v2, not
 v2.2 or v3.x or whatever), unless explicitly otherwise stated.
...
```

一个"blob对象"就是一块二进制数据，它没有指向任何东西或有任何其它属性，甚至连文件名都没有.

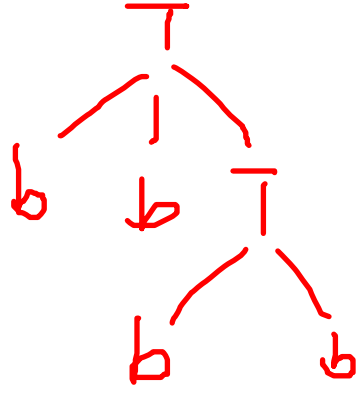
因为blob对象内容全部都是数据，如两个文件在一个目录树（或是一个版本仓库）中有同样的数据内容，那么它们将会共享同一个blob对象。Blob对象和其所对应的文件所在路径、文件名是否改被更改都完全没有关系。

Tree 对象

一个tree对象有一串(bunch)指向blob对象或是其它tree对象的指针，它一般用来表示内容之间的目录层次关系。

c36d4..

tree		size
blob	5b1d3	README
tree	03e78	lib
tree	cdc8b	test
blob	cba0a	test.rb
blob	911e7	xdiff



git show命令还可以用来查看tree对象，但是git ls-tree能让你看到更多的细节。如果我们有一个tree对象的SHA1哈希值，我们可以像下面一样来查看它：

```
$ git ls-tree fb3a8bdd0ce
100644 blob 63c918c667fa005ff12ad89437f2fdc80926e21c    .gitignore
100644 blob 5529b198e8d14decbe4ad99db3f7fb632de0439d    .mailmap
100644 blob 6ff87c4664981e4397625791c8ea3bbb5f2279a3    COPYING
040000 tree 2fb783e477100ce076f6bf57e4a6f026013dc745    Documentation
100755 blob 3c0032cec592a765692234f1cba47dfdcc3a9200    GIT-VERSION-GEN
100644 blob 289b046a443c0647624607d471289b2c7dcd470b    INSTALL
100644 blob 4eb463797adc693dc168b926b6932ff53f17d0b1    Makefile
```

```
100644 blob 548142c327a6790ff8821d67c2ee1eff7a656b52    README
...
```

就如同你所见，一个tree对象包括一串(list)条目，每一个条目包括：mode、对象类型、SHA1值 和名字(这串条目是按名字排序的)。它用来表示一个目录树的内容。

一个tree对象可以指向(reference): 一个包含文件内容的blob对象, 也可以是其它包含某个子目录内容的其它tree对象. Tree对象、blob对象和其它所有的对象一样，都用其内容的SHA1哈希值来命名的；只有当两个tree对象的内容完全相同（包括其所指向所有子对象）时，它的名字才会一样，反之亦然。这样就能让Git仅仅通过比较两个相关的tree对象的名字是否相同，来快速的判断其内容是否不同。

(注意：在submodules里，trees对象也可以指向commits对象. 请参见 **Submodules** 章节)

注意：所有的文件的mode位都是644 或 755，这意味着Git只关心文件的可执行位.

Commit对象

"commit对象"指向一个"tree对象", 并且带有相关的描述信息.

你可以用 --pretty=raw 参数来配合 git show 或 git log 去查看某个提交(commit):

```
$ git show -s --pretty=raw 2be7fcb476
commit 2be7fcb4764f2dbcee52635b91fedb1b3dcf7ab4
tree fb3a8bdd0ceddd019615af4d57a53f43d8cee2bf
parent 257a84d9d02e90447b149af58b271c19405edb6a
author Dave Watson <dwatson@mimvista.com> 1187576872 -0400
committer Junio C Hamano <gitster@pobox.com> 1187591163 -0700

    Fix misspelling of 'suppress' in docs

    Signed-off-by: Junio C Hamano <gitster@pobox.com>
```

你可以看到, 一个提交(commit)由以下的部分组成:

- 一个 **tree**　对象: tree对象的SHA1签名, 代表着目录在某一时间点的内容.
- **父对象** (parent(s)): 提交(commit)的SHA1签名代表着当前提交前一步的项目历史. 上面的那个例子就只有一个父对象; 合并的提交(merge commits)可能会有不只一个父对象. 如果一个提交没有父对象, 那么我们就叫它"根提交"(root commit), 它就代表着项目最初的一个版本(revision). 每个项目必须有至少有一个"根提交"(root commit). 一个项目可能有多个"根提交"，虽然这并不常见(这不是好的作法).
- **作者** : 做了此次修改的人的名字,　还有修改日期.
- **提交者**（committer): 实际创建提交(commit)的人的名字, 同时也带有提交日期. TA可能会和作者不是同一个人; 例如作者写一个补丁(patch)并把它用邮件发给提交者, 由他来创建提交(commit).

－**注释** 用来描述此次提交.

注意: 一个提交(commit)本身并没有包括任何信息来说明其做了哪些修改; 所有的修改(changes)都是通过与父提交(parents)的内容比较而得出的. 值得一提的是, 尽管git可以检测到文件内容不变而路径改变的情况, 但是它不会去显式(explicitly)的记录文件的更名操作.　(你可以看一下 git diff 的 -M　参数的用法)

一般用 git commit 来创建一个提交(commit), 这个提交(commit)的父对象一般是当前分支(current HEAD),　同时把存储在当前索引(index)的内容全部提交.

对象模型

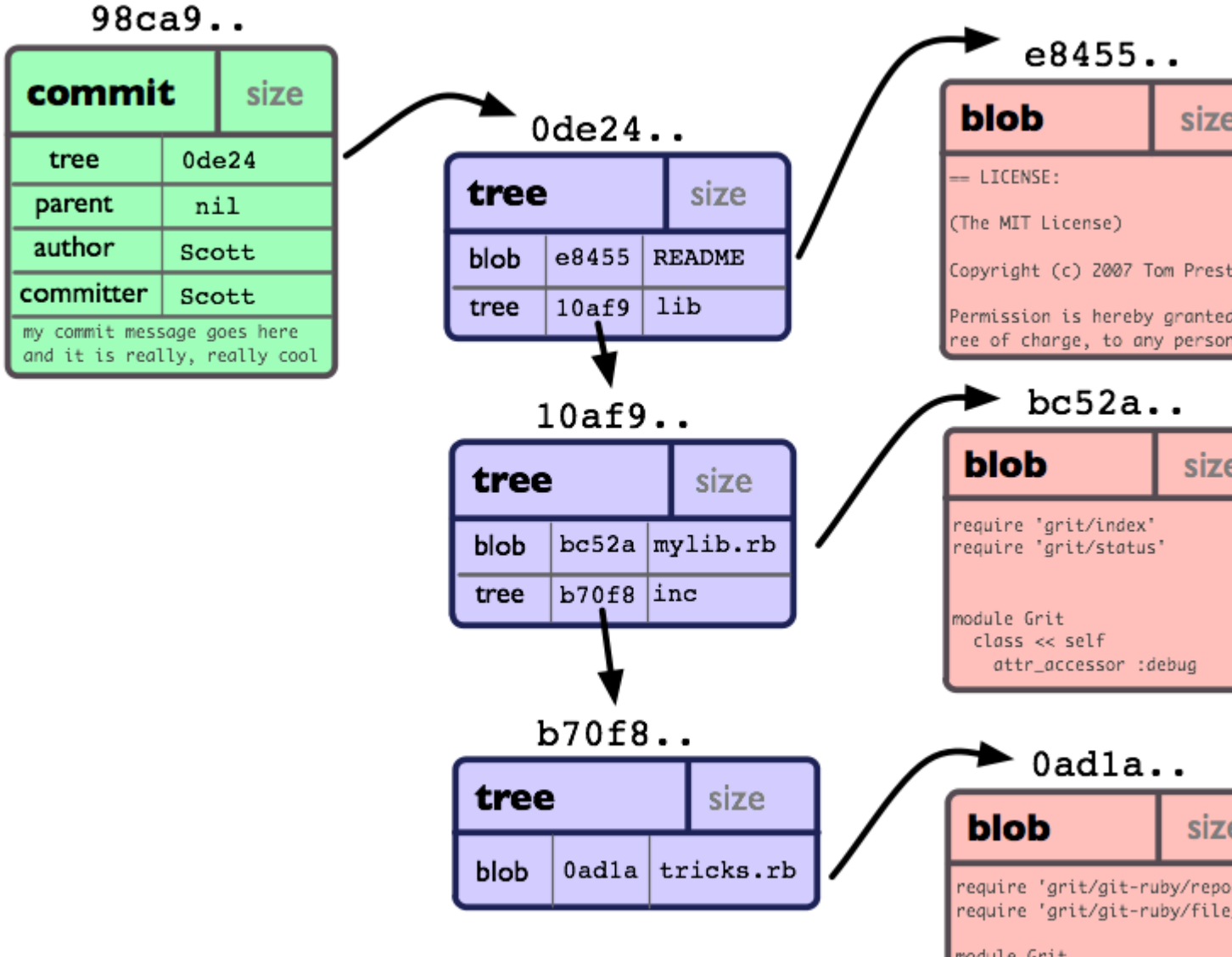
现在我们已经了解了3种主要对象类型(blob, tree 和 commit), 好现在就让我们大概了解一下它们怎么组合到一起的.

如果我们一个小项目, 有如下的目录结构:

```
$>tree
.
|-- README
`-- lib
    |-- inc
    |   `-- tricks.rb
    `-- mylib.rb

2 directories, 3 files
```

如果我们把它提交(commit)到一个Git仓库中, 在Git中它们也许看起来就如下图:


98ca9..
commit
size
tree
0de24
parent
nil
author
Scott
committer
Scott
my commit message goes here
and it is really, really cool
0de24..
tree
size
blob
e8455
README
tree
10af9
lib
e8455..
blob
size
== LICENSE:
(The MIT License)
Copyright (c) 2007 Tom Prest
Permission is hereby granted
ree of charge, to any person
10af9..
tree
size
blob
bc52a
mylib.rb
tree
b70f8
inc
bc52a..
blob
size
require 'grit/index'
require 'grit/status'
module Grit
class << self
attr_accessor :debug
b70f8..
tree
size
blob
0ad1a
tricks.rb
0ad1a..
blob
siz
require 'grit/git-ruby/repo
require 'grit/git-ruby/file

你可以看到: 每个目录都创建了 **tree对象** (包括根目录), 每个文件都创建了一个对应的 **blob对象** . 最后有一个 **commit对象** 来指向根tree对象(root of trees), 这样我们就可以追踪项目每一项提交内容.

标签对象

一个标签对象包括一个对象名(译者注:就是SHA1签名), 对象类型, 标签名, 标签创建人的名字("tagger"), 还有一条可能包含有签名(signature)的消息. 你可以用 git cat-file 命令来查看这些信息:

```
$ git cat-file tag v1.5.0
object 437b1b20df4b356c9342dac8d38849f24ef44f27
type commit
tag v1.5.0
tagger Junio C Hamano <junkio@cox.net> 1171411200 +0000

GIT 1.5.0
```

```
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.6 (GNU/Linux)

iD8DBQBF0lGqwMbZpPMRm5oRAuRiAJ9ohBLd7s2kqjkKlq1qqC57SbnmzQCdG4ui
nLE/L9aUXdWeTFPron96DLA=
=2E+0
-----END PGP SIGNATURE-----
```

点击 git tag, 可以了解如何创建和验证标签对象. (注意: git tag 同样也可以用来创建 "轻量级的标签"(lightweight tags), 但它们并不是标签对象, 而只一些以 "refs/tags/" 开头的引用罢了).

GIT目录 与 工作目录

Git目录

'Git目录'是为你的项目存储所有历史和元信息的目录 - 包括所有的对象(commits,trees,blobs,tags), 这些对象指向不同的分支.

每一个项目只能有一个'Git目录'(这和SVN,CVS的每个子目录中都有此类目录相反), 这个叫'.git'的目录在你项目的根目录下(这是默认设置,但并不是必须的). 如果你查看这个目录的内容, 你可以看所有的重要文件:

```
$>tree -L 1
.
|-- HEAD          # 这个git项目当前处在哪个分支里
|-- config        # 项目的配置信息，git config命令会改动它
|-- description   # 项目的描述信息
|-- hooks/        # 系统默认钩子脚本目录
|-- index         # 索引文件
|-- logs/         # 各个refs的历史信息
```

```
|-- objects/     # Git本地仓库的所有对象 (commits, trees, blobs, tags)
`-- refs/        # 标识你项目里的每个分支指向了哪个提交(commit)。
```

(也许现在还有其它 文件/目录 在 'Git目录' 里面, 但是现在它们并不重要)

工作目录

Git的 '工作目录' 存储着你现在签出(checkout)来用来编辑的文件. 当你在项目的不同分支间切换时, 工作目录里的文件经常会被替换和删除. 所有历史信息都保存在 'Git目录'中； 工作目录只用来临时保存签出(checkout) 文件的地方, 你可以编辑工作目录的文件直到下次提交(commit)为止.

译者注: 'Git目录' 一般就是指项目根目录下的'.git'目录.

GIT索引

Git索引是一个在你的工作目录和项目仓库间的暂存区(staging area). 有了它, 你可以把许多内容的修改一起提交(commit). 如果你创建了一个提交(commit), 那么提交的是当前索引(index)里的内容, 而不是工作目录中的内容.

查看索引

使用 git status 命令是查看索引内容的最简单办法. 你运行 git status命令, 就可以看到: 哪些文件被暂存了(就是在你的Git索引中), 哪些文件被修改了但是没有暂存, 还有哪些文件没有被跟踪(untracked).

```
$>git status
# On branch master
# Your branch is behind 'origin/master' by 11 commits, and can be fast-forwarded.
```

```
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   daemon.c
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   grep.c
#   modified:   grep.h
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   blametree
#   blametree-init
#   git-gui/git-citool
```

如果完全掌握了索引(index), 你就一般不会丢失任何信息, 只要你记得名字描述信息(name of the tree that it described)就能把它们找回来.

同时, 你最好能对Git一些基本功能的运作原理, 和它与其它版本控制系统的区别有一个清晰的理解. 如果你在这一章没有完全理解, 我们会在后面的章节重新回顾这些主题. 好了, 下面我们要去了解如何安装, 配置和使用Git.

Chapter 2

第一步

安装GIT

从源代码开始安装

如果你在一个其基于Unix的系统中，你可以从Git的官网上Git Download Page下载它的源代码,并运行像下面的几行命令,你就可以安装:

```
$ make prefix=/usr all ;# as yourself
$ make prefix=/usr install ;# 以root权限运行
```

你需一些库: expat,curl, zlib, 和 openssl; 除了expat 外，其它的可能在你的机器上都安装了。

Linux

如果你用的是Linux，你可以用你的本地包管理系统(native package management system)来安装.

```
$ yum install git-core  #译者注，在redhat等系统下用yum

$ apt-get install git-core  #译者注，在debian, ubuntu等系统下用apt-get
```

如果你用上面的命令不起作用的话，你可以从下面两个站点下载 .deb 或 .rpm 包:

RPM Packages

Stable Debs

如果你在Linux兴趣从源代码开始安装的话,下面的这篇文章也许对你有帮助: Article: Installing Git on Ubuntu

Mac 10.4

在Mac10.4和 10.5,如果你安装了MacPorts,你可以通过 MacPorts来安装Git。如果你没有安装MacPort, 你可以从 这里来安装它.

当你安装好MacPorts后，你可通过下面的命令来安装:

```
$ sudo port install git-core
```

如果你想从源代码开始安装，下面这些文章可能对你有帮助:

Article: Installing Git on Tiger

Article: Installing Git and git-svn on Tiger from source

Mac 10.5

在Leopard系统下，你也可以通过MacPorts来安装,但是你有一个新的选项:"一个漂亮的安装包", 你可以从这里来下载:Git OSX Installer

如果你想从源代码开始安装，我希望下面些资料能对你有帮助:

Article: Installing Git on OSX Leopard

Article: Installing Git on OS 10.5

Windows

在Windows下安装Git是很简单的，你只要下载msysGit就可以了。

Git on Windows 这一章有一个"screencast"来在演示如何在windows下使用Git.

安装与初始化

Git 配置

使用Git的第一件事就是设置你的名字和email,这些就是你在提交commit时的签名。

```
$ git config --global user.name "Scott Chacon"
$ git config --global user.email "schacon@gmail.com"
```

执行了上面的命令后,会在你的主目录(home directory)建立一个叫 *~/.gitconfig* 的文件. 内容一般像下面这样:

```
[user]
        name = Scott Chacon
        email = schacon@gmail.com
```

译者注:这样的设置是全局设置,会影响此用户建立的每个项目.

如果你想使项目里的某个值与前面的全局设置有区别(例如把私人邮箱地址改为工作邮箱);你可以在项目中使用git config 命令不带 *–global* 选项来设置. 这会在你项目目录下的 *.git/config* 文件增加一节[user]内容(如上所示).

Chapter 3

基本用法

获得一个GIT仓库

既然我们现在把一切都设置好了，那么我们需要一个Git仓库。有两种方法可以得到它：一种是从已有的Git仓库中 *clone* (克隆，复制)；还有一种是新建一个仓库，把未进行版本控制的文件进行版本控制。

Clone一个仓库

为了得一个项目的拷贝(copy),我们需要知道这个项目仓库的地址(Git URL). Git能在许多协议下使用，所以Git URL可能以ssh://, http(s)://, git://,或是只是以一个用户名（git 会认为这是一个ssh 地址）为前辍. 有些仓库可以通过不只一种协议来访问，例如，Git本身的源代码你既可以用 git:// 协议来访问：

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

也可以通过http 协议来访问:

```
git clone http://www.kernel.org/pub/scm/git/git.git
```

git://协议较为快速和有效,但是有时必须使用http协议,比如你公司的防火墙阻止了你的非http访问请求.如果你执行了上面两行命令中的任意一个,你会看到一个新目录: 'git',它包含所有的Git源代码和历史记录.

在默认情况下，Git会把"Git URL"里目录名的'.git'的后辍去掉,做为新克隆(clone)项目的目录名: (例如. *git clone http://git.kernel.org/linux/kernel/git/torvalds/linux-2.6.git* 会建立一个目录叫'linux-2.6')

初始化一个新的仓库

现在假设有一个叫"project.tar.gz"的压缩文件里包含了你的一些文件，你可以用下面的命令让它置于Git的版本控制管理之下.

```
$ tar xzf project.tar.gz
$ cd project
$ git init
```

Git会输出:

```
Initialized empty Git repository in .git/
```

如果你仔细观查会发现project目录下会有一个名叫".git" 的目录被创建，这意味着一个仓库被初始化了。

gitcast:cl_init

正常的工作流程

修改文件，将它们更新的内容添加到索引中.

```
$ git add file1 file2 file3
```

你现在为commit做好了准备，你可以使用 git diff 命令再加上 --cached 参数 ,看看哪些文件将被提交(commit)。

```
$ git diff --cached
```

(如果没有--cached参数，git diff 会显示当前你所有已做的但没有加入到索引里的修改.) 你也可以用git status命令来获得当前项目的一个状况:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   file1
#   modified:   file2
#   modified:   file3
#
```

如果你要做进一步的修改, 那就继续做, 做完后就把新修改的文件加入到索引中. 最后把他们提交：

```
$ git commit
```

这会提示你输入本次修改的注释，完成后就会记录一个新的项目版本.

除了用`git add` 命令，我还可以用

```
$ git commit -a
```

这会自动把所有内容被修改的文件(不包括新创建的文件)都添加到索引中，并且同时把它们提交。

这里有一个关于写commit注释的技巧和大家分享:commit注释最好以一行短句子作为开头，来简要描述一下这次commit所作的修改(最好不要超过50个字符)；然后空一行再把详细的注释写清楚。这样就可以很方便的用工具把commit注释变成email通知，第一行作为标题，剩下的部分就作email的正文.

Git跟踪的是内容不是文件

很多版本控制系统都提供了一个 "add" 命令：告诉系统开始去跟踪某一个文件的改动。但是Git里的 "add" 命令从某种程度上讲更为简单和强大. `git add` 不但是用来添加不在版本控制中的新文件，也用于添加已在版本控制中但是刚修改过的文件; 在这两种情况下, Git都会获得当前文件的快照并且把内容暂存(stage)到索引中，为下一次commit做好准备。

gitcast:c2_normal_workflow

分支与合并@基础

一个Git仓库可以维护很多开发分支。现在我们来创建一个新的叫"experimental"的分支：

```
$ git branch experimental
```

如果你运行下面这条命令：

```
$ git branch
```

你会得到当前仓库中存在的所有分支列表：

```
    experimental
  * master
```

“experimental” 分支是你刚才创建的，“master”分支是Git系统默认创建的主分支。星号(“*”)标识了你当工作在哪个分支下，输入：

```
  $ git checkout experimental
```

切换到”experimental”分支，先编辑里面的一个文件，再提交(commit)改动，最后切换回 “master”分支。

```
  (edit file)
  $ git commit -a
  $ git checkout master
```

你现在可以看一下你原来在“experimental”分支下所作的修改还在不在；因为你现在切换回了“master”分支，所以原来那些修改就不存在了。

你现在可以在“master”分支下再作一些不同的修改:

```
  (edit file)
  $ git commit -a
```

这时，两个分支就有了各自不同的修改(diverged)；我们可以通过下面的命令来合并“experimental”和“master”两个分支:

```
  $ git merge experimental
```

如果这个两个分支间的修改没有冲突(conflict), 那么合并就完成了。如有有冲突，输入下面的命令就可以查看当前有哪些文件产生了冲突:

```
$ git diff
```

当你编辑了有冲突的文件，解决了冲突后就可以提交了：

```
$ git commit -a
```

提交(commit)了合并的内容后就可查看一下:

```
$ gitk
```

执行了gitk后会有一个很漂亮的图形的显示项目的历史。

这时你就可以删除掉你的 “experimental” 分支了(如果愿意)：

```
$ git branch -d experimental
```

git branch -d只能删除那些已经被当前分支的合并的分支. 如果你要强制删除某个分支的话就用git branch –D；下面假设你要强制删除一个叫”crazy-idea”的分支：

```
$ git branch -D crazy-idea
```

分支是很轻量级且容易的，这样就很容易来尝试它。

如何合并

你可以用下面的命令来合并两个分离的分支：git merge:

```
$ git merge branchname
```

这个命令把分支"branchname"合并到了当前分支里面。如有冲突(冲突--同一个文件在远程分支和本地分支里按不同的方式被修改了）；那么命令的执行输出就像下面一样

```
$ git merge next
 100% (4/4) done
Auto-merged file.txt
CONFLICT (content): Merge conflict in file.txt
Automatic merge failed; fix conflicts and then commit the result.
```

在有问题的文件上会有冲突标记，在你手动解决完冲突后就可以把此文件添 加到索引(index)中去，用git commit命令来提交，就像平时修改了一个文件 一样。

如果你用gitk来查看commit的结果，你会看到它有两个父分支：一个指向当前 的分支，另外一个指向刚才合并进来的分支。

解决合并中的冲突

如果执行自动合并没有成功的话，git会在索引和工作树里设置一个特殊的状态， 提示你如何解决合并中出现的冲突。

有冲突(conflicts)的文件会保存在索引中，除非你解决了问题了并且更新了索引，否则执行 git commit都会失败:

```
$ git commit
file.txt: needs merge
```

如果执行 git status 会显示这些文件没有合并(unmerged),这些有冲突的文件里面会添加像下面的冲突标识符:

```
<<<<<<< HEAD:file.txt
Hello world
=======
Goodbye
>>>>>>> 77976da35a11db4580b80ae27e8d65caf5208086:file.txt
```

你所需要的做是就是编辑解决冲突，（接着把冲突标识符删掉），再执行下面的命令:

```
$ git add file.txt
$ git commit
```

注意：提交注释里已经有一些关于合并的信息了，通常是用这些默认信息，但是你可以添加一些你想要的注释。

上面这些就是你要做一个简单合并所要知道的，但是git提供更多的一些信息来 帮助解决冲突。

撤销一个合并

如果你觉得你合并后的状态是一团乱麻，想把当前的修改都放弃，你可以用下面的命令回到合并之前的状态：

```
$ git reset --hard HEAD
```

或者你已经把合并后的代码提交，但还是想把它们撤销：

```
$ git reset --hard ORIG_HEAD
```

但是刚才这条命令在某些情况会很危险，如果你把一个已经被另一个分支合并的分支给删了，那么 以后在合并相关的分支时会出错。

快速向前合并

还有一种需要特殊对待的情况，在前面没有提到。通常，一个合并会产生一个合并提交(commit), 把两个父分支里的每一行内容都合并进来。

但是，如果当前的分支和另一个分支没有内容上的差异，就是说当前分支的每一个提交(commit)都已经存在另一个分支里了，git 就会执行一个“快速向前"(fast forward)操作；git 不创建任何新的提交(commit),只是将当前分支指向合并进来的分支。

gitcast:c6-branch-merge

查看历史 －GIT日志

git log命令可以显示所有的提交(commit)。 .....

```
$ git log v2.5..        # commits since (not reachable from) v2.5
$ git log test..master  # commits reachable from master but not test
$ git log master..test  # commits reachable from test but not master
$ git log master...test # commits reachable from either test or
                        #    master, but not both
$ git log --since="2 weeks ago" # commits from the last 2 weeks
$ git log Makefile      # commits that modify Makefile
$ git log fs/           # commits that modify any file under fs/
$ git log -S'foo()'     # commits that add or remove any file data
                        # matching the string 'foo()'
$ git log --no-merges   # dont show merge commits
```

当然你也可以组合上面的命令选项；下面的命令就是找出所有从"v2.5“开 始在fs目录下的所有Makefile的修改.

```
$ git log v2.5.. Makefile fs/
```

Git会根据git log命令的参数，按时间顺序显示相关的提交(commit)。

```
commit f491239170cb1463c7c3cd970862d6de636ba787
Author: Matt McCutchen <matt@mattmccutchen.net>
Date:   Thu Aug 14 13:37:41 2008 -0400

    git format-patch documentation: clarify what --cover-letter do

commit 7950659dc9ef7f2b50b18010622299c508bfdfc3
Author: Eric Raible <raible@gmail.com>
Date:   Thu Aug 14 10:12:54 2008 -0700

    bash completion: 'git apply' should use 'fix' not 'strip'
    Bring completion up to date with the man page.
```

你也可以让git log显示补丁(patchs):

```
$ git log -p

commit da9973c6f9600d90e64aac647f3ed22dfd692f70
Author: Robert Schiele <rschiele@gmail.com>
Date:   Mon Aug 18 16:17:04 2008 +0200

    adapt git-cvsserver manpage to dash-free syntax

diff --git a/Documentation/git-cvsserver.txt b/Documentation/git-cvsserver.txt
index c2d3c90..785779e 100644
--- a/Documentation/git-cvsserver.txt
+++ b/Documentation/git-cvsserver.txt
@@ -11,7 +11,7 @@ SYNOPSIS
 SSH:
```

git show HEAD^ 查看最近一次的前一次的commit 的log
git show HEAD^^ 查看最近一次前一次的前一次的commit的log
....
git show HEAD~4 查看4次前commit的log

git show HEAD^1 查看当前HEAD的前一次commit log
git show HEAD^2 查看当前HEAD的前两次commit log

```
 [verse]
-export CVS_SERVER=git-cvsserver
+export CVS_SERVER="git cvsserver"
 'cvs' -d :ext:user@server/path/repo.git co <HEAD_name>

 pserver (/etc/inetd.conf):
```

日志统计

如果用--stat选项使用'git log',它会显示在每个提交(commit)中哪些文件被修改了, 这些文件分别添加或删除了多少行内容.

```
$ git log --stat

commit dba9194a49452b5f093b96872e19c91b50e526aa
Author: Junio C Hamano <gitster@pobox.com>
Date:   Sun Aug 17 15:44:11 2008 -0700

    Start 1.6.0.X maintenance series

 Documentation/RelNotes-1.6.0.1.txt |   15 +++++++++++++++
 RelNotes                           |    2 +-
 2 files changed, 16 insertions(+), 1 deletions(-)
```

格式化日志

你可以按你的要求来格式化日志输出。'--pretty'参数可以使用若干表现格式，如'oneline':

```
$ git log --pretty=oneline
a6b444f570558a5f31ab508dc2a24dc34773825f dammit, this is the second time this has reverted
49d77f72783e4e9f12d1bbcacc45e7a15c800240 modified index to create refs/heads if it is not
9764edd90cf9a423c9698a2f1e814f16f0111238 Add diff-lcs dependency
e1ba1e3ca83d53a2f16b39c453fad33380f8d1cc Add dependency for Open4
0f87b4d9020fff756c18323106b3fd4e2f422135 merged recent changes: * accepts relative alt pat
f0ce7d5979dfb0f415799d086e14a8d2f9653300 updated the Manifest file
```

或者你也可以使用 'short' 格式:

```
$ git log --pretty=short
commit a6b444f570558a5f31ab508dc2a24dc34773825f
Author: Scott Chacon <schacon@gmail.com>

    dammit, this is the second time this has reverted

commit 49d77f72783e4e9f12d1bbcacc45e7a15c800240
Author: Scott Chacon <schacon@gmail.com>

    modified index to create refs/heads if it is not there

commit 9764edd90cf9a423c9698a2f1e814f16f0111238
Author: Hans Engel <engel@engel.uk.to>

    Add diff-lcs dependency
```

你也可用'medium','full','fuller','email' 或'raw'. 如果这些格式不完全符合你的相求， 你也可以用'--pretty=format'参数(参见：git log)来创建你自己的"格式".

```
$ git log --pretty=format:'%h was %an, %ar, message: %s'
a6b444f was Scott Chacon, 5 days ago, message: dammit, this is the second time this has re
49d77f7 was Scott Chacon, 8 days ago, message: modified index to create refs/heads if it i
9764edd was Hans Engel, 11 days ago, message: Add diff-lcs dependency
```

```
e1ba1e3 was Hans Engel, 11 days ago, message: Add dependency for Open4
0f87b4d was Scott Chacon, 12 days ago, message: merged recent changes:
```

另一个有趣的事是：你可以用'--graph'选项来可视化你的提交图(commit graph),就像下面这样:

```
$ 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
```

它会用ASCII字符来画出一个很漂亮的提交历史(commit history)线。

日志排序

你也可以把日志记录按一些不同的顺序来显示。注意，git日志从最近的提交(commit)开始，并且从这里开始向它们父分支回溯。然而git历史可能包括多个互不关联的开发线路，这样有时提交(commits)显示出来就有点杂乱。

如果你要指定一个特定的顺序，可以为git log命令添加顺序参数(ordering option).

按默认情况，提交(commits)会按逆时间(reverse chronological)顺序显示。

但是你也可以指定'--topo-order'参数，这就会让提交(commits)按拓朴顺序来显示(就是子提交在它们的父提交前显示). 如果你用git log命令按拓朴顺序来显示git仓库的提交日志，你会看到"开发线"(development lines)都会集合在一起.

```
$ git log --pretty=format:'%h : %s' --topo-order --graph
*   4a904d7 : Merge branch 'idx2'
|\
| *   dfeffce : merged in bryces changes and fixed some testing issues
| |\
| | * 23f4ecf : Clarify how to get a full count out of Repo#commits
| | *   9d6d250 : Appropriate time-zone test fix from halorgium
| | |\
| | | * cec36f7 : Fix the to_hash test to run in US/Pacific time
| | * | decfe7b : fixed manifest and grit.rb to make correct gemspec
| | * | cd27d57 : added lib/grit/commit_stats.rb to the big list o' files
| | * | 823a9d9 : cleared out errors by adding in Grit::Git#run method
| | * |   4eb3bf0 : resolved merge conflicts, hopefully amicably
| | |\ \
| | | * | d065e76 : empty commit to push project to runcoderun
| | | * | 3fa3284 : whitespace
| | | * | d01cffd : whitespace
| | | * | 7c74272 : oops, update version here too
| | | * | 13f8cc3 : push 0.8.3
| | | * | 06bae5a : capture stderr and log it if debug is true when running commands
| | | * | 0b5bedf : update history
| | | * | d40e1f0 : some docs
| | | * | ef8a23c : update gemspec to include the newly added files to manifest
| | | * | 15dd347 : add missing files to manifest; add grit test
| | | * | 3dabb6a : allow sending debug messages to a user defined logger if provided; tes
| | | * | eac1c37 : pull out the date in this assertion and compare as xmlschemaw, to avoi
| | | * | 0a7d387 : Removed debug print.
| | | * | 4d6b69c : Fixed to close opened file description.
```

你也可以用'--date-order'参数，这样显示提交日志的顺序主要按提交日期来排序. 这个参数和'--topo-order'有一点像，没有父分支会在它们的子分支前显示，但是其它的东东还是按交时间来排序显示。你会看到"开发线"(development lines)没有集合一起，它们会像并行开发(parallel development)一样跳来跳去的：

```
$ git log --pretty=format:'%h : %s' --date-order --graph
*   4a904d7 : Merge branch 'idx2'
|\
* | 81a3e0d : updated packfile code to recognize index v2
| *   dfeffce : merged in bryces changes and fixed some testing issues
| |\
| * | c615d80 : fixed a log issue
|/ /
| * 23f4ecf : Clarify how to get a full count out of Repo#commits
| *   9d6d250 : Appropriate time-zone test fix from halorgium
| |\
| * | decfe7b : fixed manifest and grit.rb to make correct gemspec
| * | cd27d57 : added lib/grit/commit_stats.rb to the big list o' file
| * | 823a9d9 : cleared out errors by adding in Grit::Git#run method
| * |   4eb3bf0 : resolved merge conflicts, hopefully amicably
| |\ \
| * | | ba23640 : Fix CommitDb errors in test (was this the right fix?
| * | | 4d8873e : test_commit no longer fails if you're not in PDT
| * | | b3285ad : Use the appropriate method to find a first occurrenc
| * | | 44dda6c : more cleanly accept separate options for initializin
| * | | 839ba9f : needed to be able to ask Repo.new to work with a bar
| | * | d065e76 : empty commit to push project to runcoderun
* | | | 791ec6b : updated grit gemspec
* | | | 756a947 : including code from github updates
| | * | 3fa3284 : whitespace
| | * | d01cffd : whitespace
| * | | a0e4a3d : updated grit gemspec
| * | | 7569d0d : including code from github updates
```

最后，你也可以用 '--reverse'参数来逆向显示所有日志。

gitcast:c4-git-log

比较提交 - GIT DIFF

你可以用 git diff 来比较项目中任意两个版本的差异。

```
$ git diff master..test
```

上面这条命令只显示两个分支间的差异，如果你想找出‘master’,‘test’的共有 父分支和'test'分支之间的差异，你用3个‘.'来取代前面的两个'.' 。

```
$ git diff master...test
```

git diff 是一个难以置信的有用的工具，可以找出你项目上任意两点间 的改动，或是用来查看别人提交进来的新分支。

哪些内容会被提交(commit)

你通常用git diff来找你当前工作目录和上次提交与本地索引间的差异。

```
$ git diff
```

上面的命令会显示在当前的工作目录里的，没有 staged(添加到索引中)，且在下次提交时 不会被提交的修改。

如果你要看在下次提交时要提交的内容(staged,添加到索引中),你可以运行：

```
$ git diff --cached
```

上面的命令会显示你当前的索引和上次提交间的差异；这些内容在不带"-a"参数运行 "git commit"命令时就会被提交。

```
$ git diff HEAD
```

上面这条命令会显示你工作目录与上次提交时之间的所有差别，这条命令所显示的 内容都会在执行"git commit -a"命令时被提交。

更多的比较选项

如果你要查看当前的工作目录与另外一个分支的差别，你可以用下面的命令执行:

```
$ git diff test
```

这会显示你当前工作目录与另外一个叫'test'分支的差别。你也以加上路径限定符，来只 比较某一个文件或目录。

```
$ git diff HEAD -- ./lib
```

上面这条命令会显示你当前工作目录下的lib目录与上次提交之间的差别(或者更准确的 说是在当前分支)。

如果不是查看每个文件的详细差别，而是统计一下有哪些文件被改动，有多少行被改 动，就可以使用'--stat' 参数。

```
$>git diff --stat
 layout/book_index_template.html                    |    8 ++-
 text/05_Installing_Git/0_Source.markdown           |   14 ++++++
 text/05_Installing_Git/1_Linux.markdown            |   17 +++++++
 text/05_Installing_Git/2_Mac_104.markdown          |   11 +++++
 text/05_Installing_Git/3_Mac_105.markdown          |    8 ++++
 text/05_Installing_Git/4_Windows.markdown          |    7 +++
 .../1_Getting_a_Git_Repo.markdown                  |    7 +++-
 .../0_ Comparing_Commits_Git_Diff.markdown         |   45 ++++++++++++++++++++
```

```
 .../0_ Hosting_Git_gitweb_repoorcz_github.markdown |    4 +-
 9 files changed, 115 insertions(+), 6 deletions(-)
```

有时这样全局性的查看哪些文件被修改，能让你更轻轻一点。

分布式的工作流程

假设Alice现在开始了一个新项目，在/home/alice/project建了一个新的git 仓库(repository)；另一个叫Bob的工作目录也在同一台机器，他要提交代码。

Bob 执行了这样的命令:

```
$ git clone /home/alice/project myrepo
```

这就建了一个新的叫"myrepo"的目录，这个目录里包含了一份Alice的仓库的 克隆(clone). 这份克隆和原始的项目一模一样，并且拥有原始项目的历史记 录。

Bob 做了一些修改并且提交(commit)它们:

```
(edit files)
$ git commit -a
(repeat as necessary)
```

当他准备好了，他告诉Alice从仓库/home/bob/myrepo中把他的修改给拉 (pull)下来。她执行了下面几条命令:

```
$ cd /home/alice/project
$ git pull /home/bob/myrepo master
```

这就把Bob的主(master)分支合并到了Alice的当前分支里了。如果Alice在 Bob修改文件内容的同时也做了修改的话，她可能需要手工去修复冲突. (注意："master"参数在上面的命令中并不一定是必须的，因为这是一个 默认参数)

git pull命令执行两个操作: 它从远程分支(remote branch)抓取修改 的内容，然后把它合并进当前的分支。

如果你要经常操作远程分支(remote branch),你可以定义它们的缩写:

```
$ git remote add bob /home/bob/myrepo
```

这样，Alic可以用"git fetch"" 来执行"git pull"前半部分的工作， 但是这条命令并不会把抓下来的修改合并到当前分支里。

```
$ git fetch bob
```

我们用`git remote`命令建立了Bob的运程仓库的缩写，用这个(缩写) 名字我从Bob那得到所有远程分支的历史记录。在这里远程分支的名 字就叫`bob/master`.

```
$ git log -p master..bob/master
```

上面的命令把Bob从Alice的主分支(master)中签出后所做的修改全部显示出来。

当检查完修改后,Alice就可以把修改合并到她的主分支中。

```
$ git merge bob/master
```

这种合并(merge)也可以用pull来完成，就像下面的命令一样：

```
$ git pull . remotes/bob/master
```

注意：git pull 会把远程分支合并进当前的分支里，而不管你在命令 行里指定什么。

其后，Bob可以更新它的本地仓库--把Alice做的修改拉过来(pull):

```
$ git pull
```

如果Bob从Alice的仓库克隆(clone)，那么他就不需要指定Alice仓库的地 址；因为Git把Alice仓库的地址存储到Bob的仓库配库文件，这个地址就是 在git pull时使用：

```
$ git config --get remote.origin.url
/home/alice/project
```

(如果要查看git clone创建的所有配置参数，可以使用"git config -l", git config 的帮助文件里解释了每个参数的含义.)

Git同时也保存了一份最初(pristine)的Alice主分支(master)，在 "origin/master"下面。

```
$ git branch -r
  origin/master
```

如果Bob打算在另外一台主机上工作，他可以通过ssh协议来执行"clone" 和"pull"操作：

```
$ git clone alice.org:/home/alice/project myrepo
```

git有他自带的协议(native protocol),还可以使用rsync或http; 你可以点 这里 git pull 看一看更詳細的用法。

Git也可以像CVS一样来工作：有一个中心仓库，不同的用户向它推送(push) 自己所作的修改；你可以看看这里：git push gitcvs-migration.

公共Git仓库

另外一个提交修改的办法，就是告诉项目的维护者(maintainer)用 git pull 命令从你的仓库里把修改拉下来。这和从主仓库"里更新代码类似，但是是从 另外一个方向来更新的。

如果你和维护者(maintainer)都在同一台机器上有帐号，那么你们可以互相从对 方的仓库目录里直接拉(pull)所作的修改；git命令里的仓库地址也可以是本地 的某个目录名：

```
$ git clone /path/to/repository
$ git pull /path/to/other/repository
```

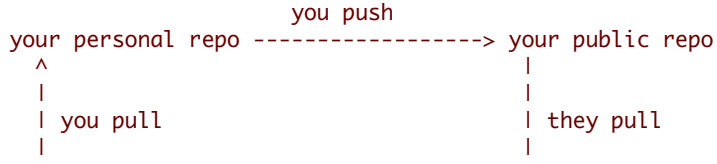
也可以是一个ssh地址：

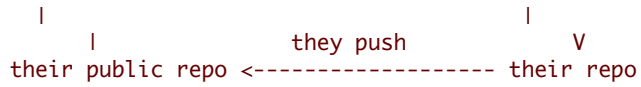
```
$ git clone ssh://yourhost/~you/repository
```

如果你的项目只有很少几个开发者，或是只需要同步很少的几个私有仓库， 上面的方法也许够你用的。

然而，更通用的作法是维护几个不同的公开仓库(public repository). 这样可以把每个人的工作进度和公开仓库清楚的分开。

你还是每天在你的本地私人仓库里工作，但是会定期的把本地的修改推(push) 到你的公开仓库中；其它开发者就可以从这个公开仓库来拉(pull)最新的代码。 如果其它开发者也有他自己的公共仓库，那么他们之间的开发流程就如下图 所示：





将修改推到一个公共仓库

通过http或是git协议，其它维护者可以抓取(fetch)你最近的修改，但是他们 没有写权限。这样，这需要将本地私有仓库的最近修改上传公共仓库中。

译者注: 通过http的WebDav协议是可以有写权限的,也有人配置了git over http.

最简单的办法就是用 git push命令 和ssh协议; 用你本地的"master" 分支去更新远程的"master"分支，执行下面的命令:

```
$ git push ssh://yourserver.com/~you/proj.git master:master
```

or just

或是:

```
$ git push ssh://yourserver.com/~you/proj.git master
```

和git-fetch命令一样git-push如果命令的执行结果不是"快速向前"(fast forward) 就会报错; 下面的章节会讲如何处理这种情况.

推(push)命令的目地仓库一般是个裸仓库(bare respository). 你也可以推到一 个签出工作目录树(checked-out working tree)的仓库，但是工作目录中内 容不会被推命令所更新。如果你把自己的分支推到一个已签出的分支里，这 会导致不可预知的后果。

在用git-fetch命令时，你也可以修改配置参数，让你少打字:)。

下面这些是例子:

```
$ cat >>.git/config <<EOF
[remote "public-repo"]
    url = ssh://yourserver.com/~you/proj.git
EOF
```

你可以用下面的命令来代替前面复杂的命令:

```
$ git push public-repo master
```

你可以点击这里: git config，查看remote..url, branch..remote, 和remote..push等选项的解释.

当推送代码失败时要怎么办

如果推送(push)结果不是"快速向前"(fast forward),那么它 可能会报像下面一样的错误：

```
error: remote 'refs/heads/master' is not an ancestor of
local  'refs/heads/master'.
Maybe you are not up-to-date and need to pull first?
error: failed to push to 'ssh://yourserver.com/~you/proj.git'
```

这种情况通常由以下的原因产生：

- 用 `git-reset --hard` 删除了一个已经发布了的一个提交，或是

- 用 `git-commit --amend` 去替换一个已经发布的提交，或是

```
- 用 `git-rebase` 去rebase一个已经发布的提交.
```

你可以强制git-push在上传修改时先更新，只要在分支名前面加一个加号。

```
$ git push ssh://yourserver.com/~you/proj.git +master
```

Normally whenever a branch head in a public repository is modified, it is modified to point to a descendant of the commit that it pointed to before. By forcing a push in this situation, you break that convention.

Nevertheless, this is a common practice for people that need a simple way to publish a work-in-progress patch series, and it is an acceptable compromise as long as you warn other developers that this is how you intend to manage the branch.

It's also possible for a push to fail in this way when other people have the right to push to the same repository. In that case, the correct solution is to retry the push after first updating your work: either by a pull, or by a fetch followed by a rebase; see the next section and gitcvs-migration for more.

gitcast:c8-dist-workflow

GIT标签

轻量级标签

我们可以用 git tag**不带任何参数创建一个标签(**tag**)指定某个提交(**commit**):**

```
$ git tag stable-1 1b2e1d63ff
```

这样，我们可以用stable-1 作为提交(commit) "1b2e1d63ff" 的代称(refer)。

前面这样创建的是一个“轻量级标签"，这种分支通常是从来不移动的。

如果你想为一个标签(tag)添加注释，或是为它添加一个签名(sign it cryptographically), 那么我们就需要创建一个 ”标签对象".

标签对象

如果有 "-a", "-s" 或是 "-u " 中间的一个命令参数被指定，那么就会创建 一个标签对象，并且需要一个标签消息(tag message). 如果没有"-m " 或是 "-F " 这些参数，那么就会启动一个编辑器来让用户输入标签消息(tag message).

译者注：大家觉得这个标签消息是不是提交注释(commit comment)比较像。

当这样的一条命令执行后，一个新的对象被添加到Git对象库中，并且标签引用就指向了 一个标签对象，而不是指向一个提交(commit). 这样做的好处就是：你可以为一个标签 打处签名(sign), 方便你以后来查验这是不是一个正确的提交(commit).

下面是一个创建标签对象的例子:

```
$ git tag -a stable-1 1b2e1d63ff
```

标签对象可以指向任何对象，但是在通常情况下是一个提交(commit). (在Linux内核代 码中，第一个标签对象是指向一个树对象(tree),而不是指向一个提交(commit)).

签名的标签

如果你配有GPG key,那么你就很容易创建签名的标签.首先你要在你的 *.git/config* *或* ~.gitconfig里配好key.

下面是示例:

```
[user]
    signingkey = <gpg-key-id>
```

你也可以用命令行来配置:

```
$ git config (--global) user.signingkey <gpg-key-id>
```

现在你可以直接用"-s" 参数来创“签名的标签”。

```
$ git tag -s stable-1 1b2e1d63ff
```

如果没有在配置文件中配GPG key,你可以用"-u“ 参数直接指定。

```
$ git tag -u <gpg-key-id> stable-1 1b2e1d63ff
```

Chapter 4

中级技能

忽略某些文件

项目中经常会生成一些Git系统不需要追踪(track)的文件。典型的是在编译生成过程中 产生的文件或是编程器生成的临时备份文件。当然，你不追踪(track)这些文件，可以 平时不用"git add"去把它们加到索引中。 但是这样会很快变成一件烦人的事，你发现 项目中到处有未追踪(untracked)的文件; 这样也使"`git add .`" 和"`git commit -a`" 变得实际上没有用处，同时"`git status`"命令的输出也会有它们。

你可以在你的顶层工作目录中添加一个叫".gitignore"的文件，来告诉Git系统要忽略 掉哪些文件，下面是文件内容的示例:

```
# 以'#' 开始的行，被视为注释.
# 忽略掉所有文件名是 foo.txt 的文件.
foo.txt
# 忽略所有生成的 html 文件,
```

```
*.html
# foo.html是手工维护的，所以例外.
!foo.html
#  忽略所有.o 和 .a文件.
*.[oa]
```

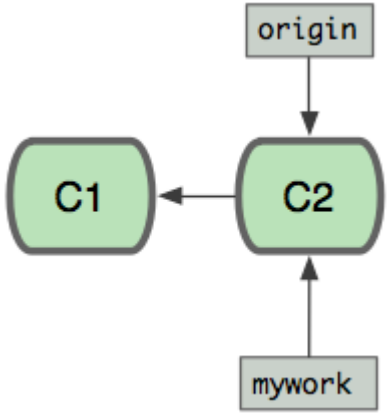
你可以点这里 gitignore 查看一下详细的语法解释. 你也可以把".gitignore" 这个文件放到工作树(working tree)里的其它目录中，这就会在它和它的子目录起忽略(ignore) 指定文件的作用。.gitignor文件同样可以像其它文件一样加到项目仓库里(直接用 `git add .gitignore` 和 `git commit`等命令), 这样项目里的其它开发者也能共享同一套忽略 文件规则。

如果你想忽略规则只对特定的仓库起作用,你可以把这些忽略规则写到你的仓库下 .git/info/exclude文件中，或是写在Git配置变量core.excludesfile中指定的 文件里。有些Git命令也可在命令行参数中指定忽略规则，你可以在这里:gitignore 查看详细的用法。

REBASE

假设你现在基于远程分支"origin"，创建一个叫"mywork"的分支。

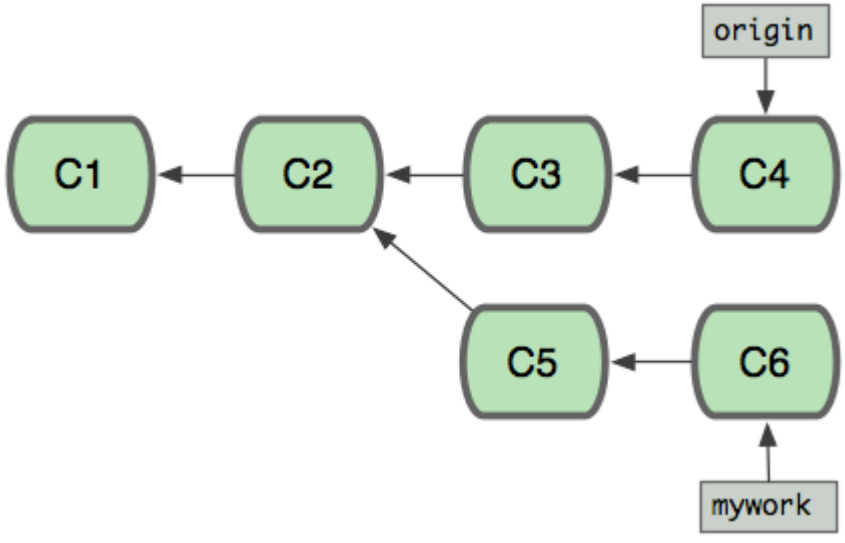
```
$ git checkout -b mywork origin
```



现在我们在这个分支做一些修改，然后生成两个提交(commit).

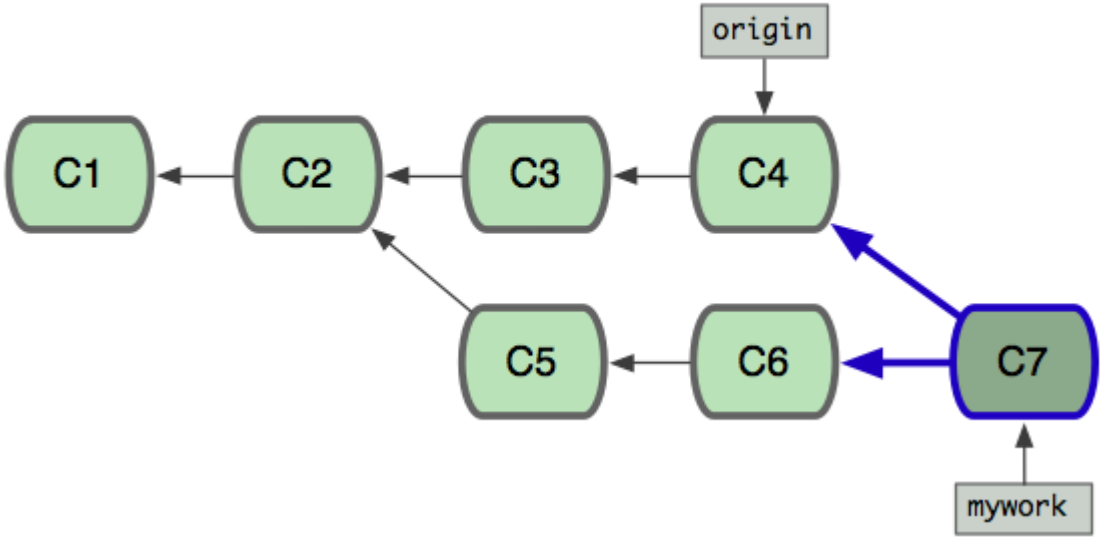
```
$ vi file.txt
$ git commit
$ vi otherfile.txt
$ git commit
...
```

但是与此同时，有些人也在"origin"分支上做了一些修改并且做了提交了. 这就意味着"origin"和"mywork"这两个分支各自"前进"了，它们之间"分叉"了。



在这里，你可以用"pull"命令把"origin"分支上的修改拉下来并且和你的修改合并； 结果看起来就像一个新的"合并的提交"(merge commit):

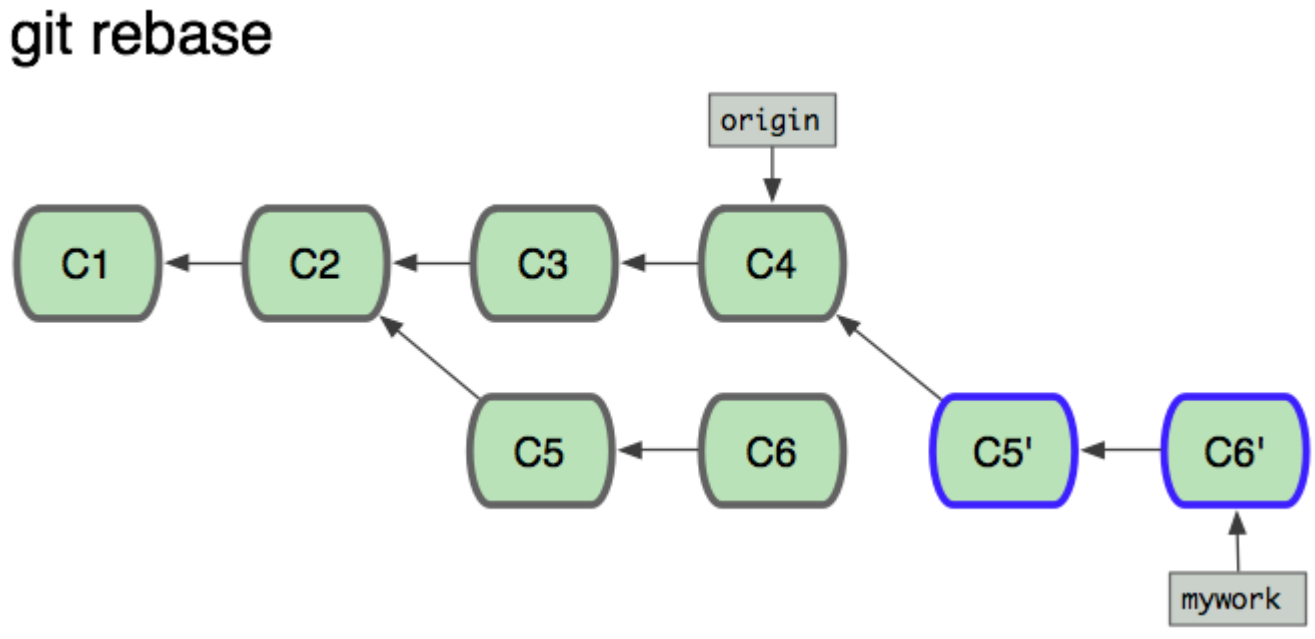
git merge



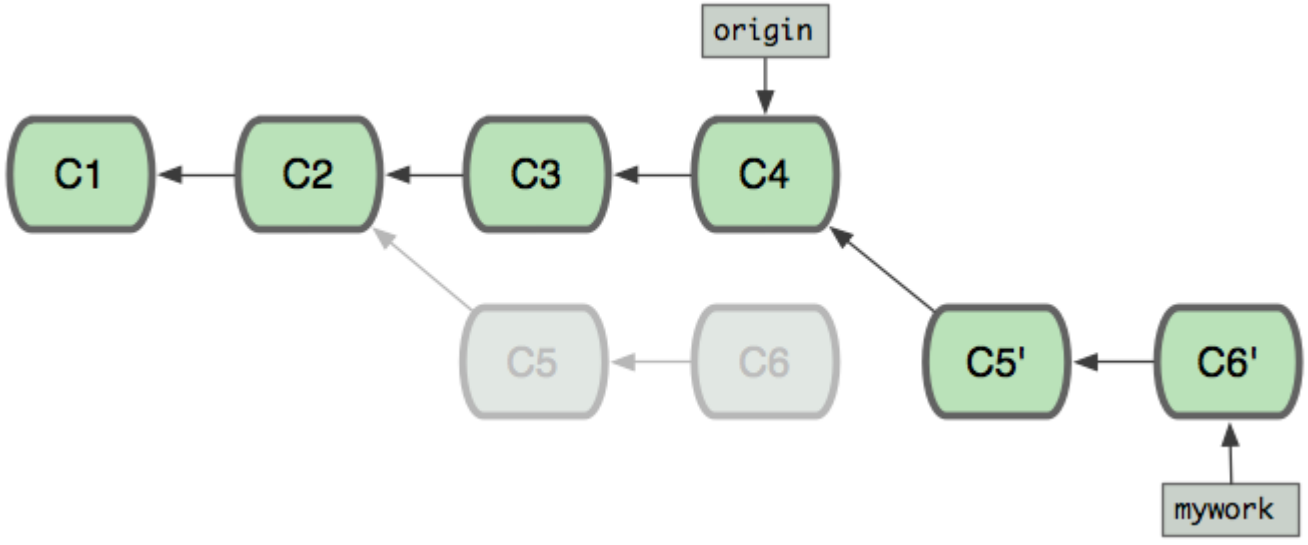
但是，如果你想让"mywork"分支历史看起来像没有经过任何合并一样，你也许可以用 git rebase:

```
$ git checkout mywork
$ git rebase origin
```

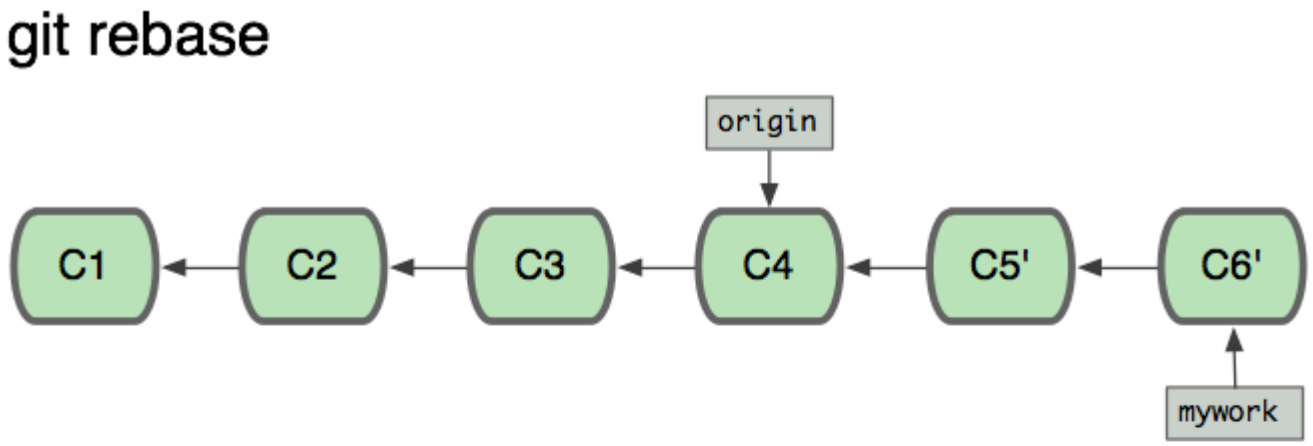
这些命令会把你的"mywork"分支里的每个提交(commit)取消掉，并且把它们临时 保存为补丁(patch)(这些补丁放到".git/rebase"目录中),然后把"mywork"分支更新 到最新的"origin"分支，最后把保存的这些补丁应用到"mywork"分支上。

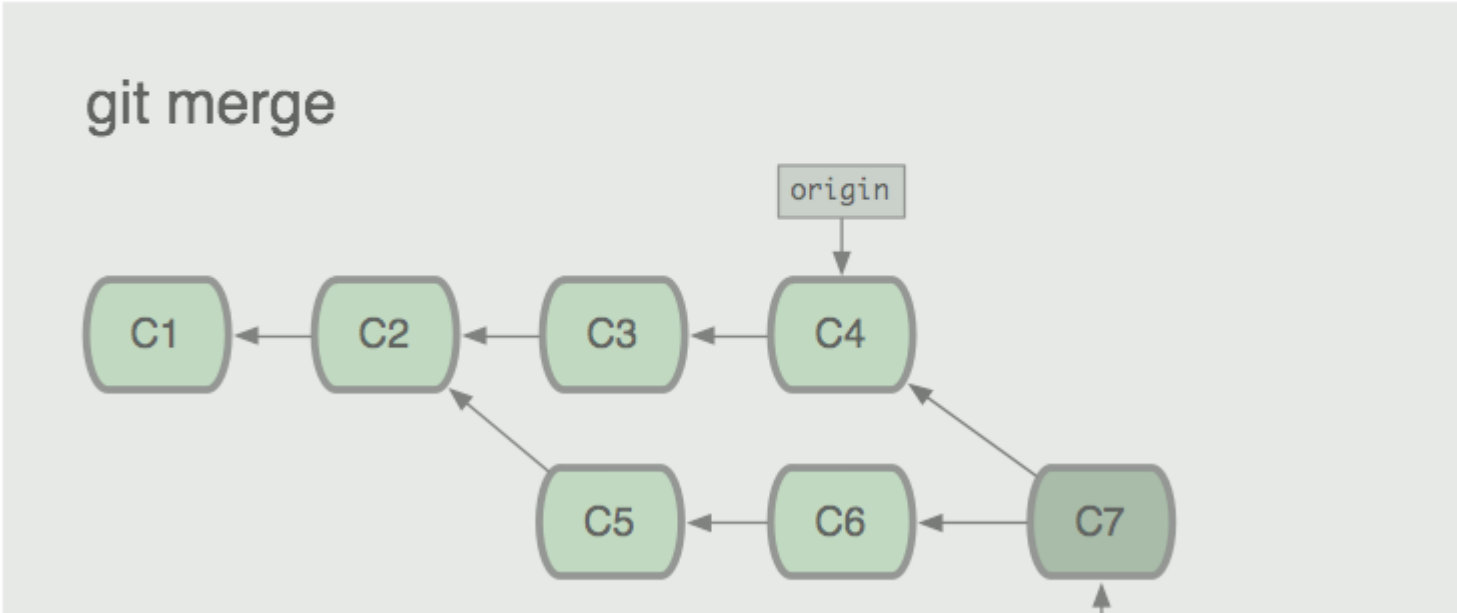


当'mywork'分支更新之后，它会指向这些新创建的提交(commit),而那些老的提交会被丢弃。 如果运行垃圾收集命令(pruning garbage collection), 这些被丢弃的提交就会删除. （请查看 git gc)



现在我们可以看一下用合并(merge)和用rebase所产生的历史的区别：


git rebase
origin
C1
C2
C3
C4
C5'
C6'
mywork


git merge
origin
C1
C2
C3
C4
C5
C6
C7

在rebase的过程中，也许会出现冲突(conflict). 在这种情况，Git会停止rebase并会让你去解决 冲突；在解决完冲突后，用"git-add"命令去更新这些内容的索引(index), 然后，你无需执行 git-commit,只要执行:

```
$ git rebase --continue
```

这样git会继续应用(apply)余下的补丁。

在任何时候，你可以用--abort参数来终止rebase的行动，并且"mywork" 分支会回到rebase开始前的状态。

```
$ git rebase --abort
```

gitcast:c7-rebase

交互式REBASE

你亦可以选择进行交互式的rebase。这种方法通常用于在向别处推送提交之前对它们进行重写。交互式rebase提供了一个简单易用的途径让你在和别人分享提交之前对你的提交进行分割、合并或者重排序。在把从其他开发者处拉取的提交应用到本地时，你也可以使用交互式rebase对它们进行清理。

如果你想在rebase的过程中对一部分提交进行修改，你可以在'git rebase'命令中加入'-i'或'--interactive'参数去调用交互模式。

```
$ git rebase -i origin/master
```

这个命令会执行交互式rebase操作，操作对象是那些自最后一次从origin仓库拉取或者向origin推送之后的所有提交。

若想查看一下将被rebase的提交，可以用如下的log命令：

```
$ git log github/master..
```

一旦运行了'rebase -i'命令，你所预设的编辑器会被调用，其中含有如下的内容：

```
pick fc62e55 added file_size
pick 9824bf4 fixed little thing
pick 21d80a5 added number to log
pick 76b9da6 added the apply command
pick c264051 Revert "added file_size" - not implemented correctly

# Rebase f408319..b04dc3d onto f408319
#
# Commands:
#  p, pick = use commit
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

这些信息表示从你上一次推送操作起有5个提交。每个提交都用一行来表示，行格式如下：

```
(action) (partial-sha) (short commit message)
```

现在你可以将操作（action）改为'edit'（使用提交，但是暂停以便进行修正）或者'squash'（使用提交，但是把它与前一提交合并），默认是'pick'（使用提交）。你可以对这些行上下移动从而对提交进行重排序。当你退出编辑器时，git会按照你指定的顺序去应用提交，并且做出相应的操作（action）。

如果指定进行'pick'操作，git会应用这个补丁，以同样的提交信息（commit message）保存提交。

如果指定进行'squash'操作，git会把这个提交和前一个提交合并成为一个新的提交。这会再次调用编辑器，你在里面合并这两个提交的提交信息。所以，如果你（在上一步）以如下的内容离开编辑器：

```
pick   fc62e55 added file_size
squash 9824bf4 fixed little thing
squash 21d80a5 added number to log
squash 76b9da6 added the apply command
squash c264051 Revert "added file_size" - not implemented correctly
```

你必须基于以下的提交信息创建一个新的提交信息：

```
# This is a combination of 5 commits.
# The first commit's message is:
added file_size

# This is the 2nd commit message:

fixed little thing

# This is the 3rd commit message:

added number to log

# This is the 4th commit message:

added the apply command

# This is the 5th commit message:

Revert "added file_size" - not implemented correctly

This reverts commit fc62e5543b195f18391886b9f663d5a7eca38e84.
```

一旦你完成对提交信息的编辑并且退出编辑器，这个新的提交及提交信息会被保存起来。

如果指定进行'edit'操作，git会完成同样的工作，但是在对下一提交进行操作之前，它会返回到命令行让你对提交进行修正，或者对提交内容进行修改。

例如你想要分割一个提交，你需要对那个提交指定'edit'操作：

```
pick   fc62e55 added file_size
pick   9824bf4 fixed little thing
edit   21d80a5 added number to log
pick   76b9da6 added the apply command
pick   c264051 Revert "added file_size" - not implemented correctly
```

你会进入到命令行，撤消（revert）该提交，然后创建两个（或者更多个）新提交。假设提交21d80a5修改了两个文件，file1和file2，你想把这两个修改放到不同的提交里。你可以在进入命令行之后进行如下的操作：

```
$ git reset HEAD^
$ git add file1
$ git commit 'first part of split commit'
$ git add file2
$ git commit 'second part of split commit'
$ git rebase --continue
```

现在你有6个提交了，而不是5个。

交互式rebase的最后一个作用是丢弃提交。如果把一行删除而不是指定'pick'、'squash'和'edit'中的任何一个，git会从历史中移除该提交。

交互式添加

交互式添加提供友好的界面去操作Git索引（index），同时亦提供了可视化索引的能力。只需简单键入'git add -i'，即可使用此功能。Git会列出所有修改过的文件及它们的状态。

```
$>git add -i
           staged     unstaged path
  1:    unchanged        +4/-0 assets/stylesheets/style.css
  2:    unchanged      +23/-11 layout/book_index_template.html
  3:    unchanged        +7/-7 layout/chapter_template.html
  4:    unchanged        +3/-3 script/pdf.rb
  5:    unchanged      +121/-0 text/14_Interactive_Rebasing/0_ Interactive_Rebasing.markdown

*** Commands ***
  1: status   2: update   3: revert   4: add untracked
  5: patch    6: diff     7: quit     8: help
What now>
```

在这个例子中，我们可以看到有5个修改过的文件还没有被加入到索引中（unstaged），甚至可以看到每个文件增加和减少的行数。紧接着是一个交互式的菜单，列出了我们可以在此模式中使用的命令。

如果我们想要暂存（stage）这些文件，我们可以键入'2'或者'u'进入更新（update）模式。然后我们可以通过键入文件的范围（本例中是1-4）来决定把哪些文件加入到索引之中。

```
What now> 2
           staged     unstaged path
  1:    unchanged        +4/-0 assets/stylesheets/style.css
  2:    unchanged      +23/-11 layout/book_index_template.html
  3:    unchanged        +7/-7 layout/chapter_template.html
  4:    unchanged        +3/-3 script/pdf.rb
  5:    unchanged      +121/-0 text/14_Interactive_Rebasing/0_ Interactive_Rebasing.markdown
```

```
Update>> 1-4
           staged     unstaged path
* 1:    unchanged        +4/-0 assets/stylesheets/style.css
* 2:    unchanged      +23/-11 layout/book_index_template.html
* 3:    unchanged        +7/-7 layout/chapter_template.html
* 4:    unchanged        +3/-3 script/pdf.rb
  5:    unchanged      +121/-0 text/14_Interactive_Rebasing/0_ Interactive_Rebasing.markdown
Update>>
```

如果键入回车，我会回到主菜单中，同时可以看到那些指定文件的状态已经发生了改变：

```
What now> status
           staged     unstaged path
  1:        +4/-0      nothing assets/stylesheets/style.css
  2:      +23/-11      nothing layout/book_index_template.html
  3:        +7/-7      nothing layout/chapter_template.html
  4:        +3/-3      nothing script/pdf.rb
  5:    unchanged      +121/-0 text/14_Interactive_Rebasing/0_ Interactive_Rebasing.markdown
```

现在我们可以看到前4个文件已经被暂存，但是最后一个没有。基本上，这是一个更加紧凑的查看状态的方式，实质上的信息与我们在命令行中运行'git status'是一致的：

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   assets/stylesheets/style.css
#   modified:   layout/book_index_template.html
#   modified:   layout/chapter_template.html
#   modified:   script/pdf.rb
#
# Changed but not updated:
```

```
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   text/14_Interactive_Rebasing/0_ Interactive_Rebasing.markdown
#
```

我们可以进行数个有用的操作，包括取消文件的暂存（3: revert），加入未跟踪的文件（4: add untracked）和查看差异（6: diff）。这些功能都很易懂。还有一个很“酷”的功能，就是暂存补丁（staging patches）（5: patch）。

如果你键入'5'或者'p'，git会一个一个补丁（一块一块）地显示出差异，然后询问是否对每一块进行暂存操作。通过这个方法，你可以暂存文件修改中的一部分。如果你编辑了一个文件，只想提交其中一部分而不包括其他未完成编辑的部分，或者把文档、空白字符从大量的修改中分开提交，你可以使用'git add -i'去相对轻松地完成任务。

这里我暂存了book_index_template.html的部分修改，而不是全部修改：

```
           staged     unstaged path
1:        +4/-0      nothing assets/stylesheets/style.css
2:       +20/-7        +3/-4 layout/book_index_template.html
3:        +7/-7      nothing layout/chapter_template.html
4:        +3/-3      nothing script/pdf.rb
5:    unchanged     +121/-0 text/14_Interactive_Rebasing/0_ Interactive_Rebasing.markdown
6:    unchanged      +85/-0 text/15_Interactive_Adding/0_ Interactive_Adding.markdown
```

当你通过'git add -i'完成对索引的改动后，你只需要退出（7: quit），然后'git commit'去提交暂存的修改。切记不要运行'git commit -a'，它会忽视你刚才辛辛苦苦做的修改而把所有东西都提交到仓库中去。

gitcast:c3_add_interactive

储藏

当你正在做一项复杂的工作时, 发现了一个和当前工作不相关但是又很讨厌的bug. 你这时想先修复bug再做手头的工作, 那么就可以用 git stash 来保存当前的工作状态, 等你修复完bug后,执行'反储藏'(unstash)操作就可以回到之前的工作里.

```
$ git stash "work in progress for foo feature"
```

上面这条命令会保存你的本地修改到储藏(stash)中, 然后将你的工作目录和索引里的内容全部重置, 回到你当前所在分支的上次提交时的状态.

好了, 你现在就可以开始你的修复工作了.

```
... edit and test ...
$ git commit -a -m "blorpl: typofix"
```

当你修复完bug后, 你可以用`git stash apply`来回复到以前的工作状态.

```
$ git stash apply
```

储藏队列

你也可多次使用'git stash'命令， 每执行一次就会把针对当前修改的‘储藏’(stash)添加到储藏队列中. 用'git stash list'命令可以查看你保存的'储藏'(stashes):

```
$>git stash list
stash@{0}: WIP on book: 51bea1d... fixed images
stash@{1}: WIP on master: 9705ae6... changed the browse code to the official repo
```

可以用类似'git stash apply stash@{1}'的命令来使用在队列中的任意一个'储藏'(stashes). 'git stash clear'则是用来清空这个队列.

GIT树名

不用40个字节长的SHA串来表示一个提交(commit)或是其它git对象, 有很多种名字表示方法. 在Git里,这些名字就叫'树名'(treeish).

译者注:我目前没有想到更好的中文名字,就先叫'树名'.

Sha短名

如果你的一个提交(commit)的sha名字是 '980e3ccdaac54a0d4de358f3fe5d718027d96aae', git会把下面的串视为等价的:

```
980e3ccdaac54a0d4de358f3fe5d718027d96aae
980e3ccdaac54a0d4
980e3cc
```

只要你的'sha短名'(Partial Sha)是不重复的(unique)，它就不会和其它名字冲突(如果你使用了5个字节以上那是很难重复的)，git也会把'sha短名'(Partial Sha)自动补全.

分支, Remote 或 标签

你可以使用分支,remote或标签名来代替SHA串名, 它们只是指向某个对象的指针. 假设你的master分支目前在提交(commit):'980e3'上, 现在把它推送(push)到origin上并把它命名为标签'v1.0', 那么下面的串都会被git视为等价的:

```
980e3ccdaac54a0d4de358f3fe5d718027d96aae
origin/master
refs/remotes/origin/master
master
refs/heads/master
v1.0
refs/tags/v1.0
```

这意味着你执行下面的两条命令会有同样的输出:

```
$ git log master

$ git log refs/tags/v1.0
```

日期标识符

The Ref Log that git keeps will allow you to do some relative stuff locally, such as:

Git的引用日志(Ref Log)可以让你做一些'相对'查询操作:

```
master@{yesterday}

master@{1 month ago}
```

上面的第一条命令是:'master分支的昨天状态(head)的缩写'. 注意: 即使在两个有相同master分支指向的仓库上执行这条命令, 但是如果这个两个仓库在不同机器上, 那么执行结果也很可能会不一样.

译者注:因为两个不同机器上的仓库的历史一般很难相同.

顺序标识符

这种格式用来表达某点前面的第N个提交(ref).

```
master@{5}
```

上面的表达式代表着master前面的第5个提交(ref).

多个父对象

这能告诉你某个提交的第N个直接父提交(parent). 这种格式在合并提交(merge commits)时特别有用, 这样就可以使提交对象(commit object)有多于一个直接父对象(direct parent).

译者注:假设master是由a和b两个分支合并的,那么 `master^1` 是指分支a, `master^2` 就是指分支b.

```
master^2
```

波浪号

波浪号用来标识一个提交对象(commit object)的第N级嫡(祖)父对象(Nth grandparent). 例如:

```
master~2
```

就代表master所指向的提交对象的第一个父对象的第一个父对象(译者:你可以理解成是嫡系爷爷:)). 它和下面的这个表达式是等价的:

```
master^^
```

你也可以把这些'标识符'(spec)叠加起来, 下面这个3个表达式都是指向同一个提交(commit):

```
master^^^^^^
master~3^~2
master~6
```

树对象指针

如果大家对第一章Git对象模型还有印象的话, 就记得提交对象(commit object)是指向一个树对象(tree object)的. 假如你要得到一个提交对象(commit object)指向的树对象(tree object)的sha串名, 你就可以在'树名'的后面加上'{tree}'来得到它:

```
master^{tree}
```

二进制标识符

如果你要某个二次制对象(blob)的sha串名,你可以在'树名'(treeish)后添加二次制对象(blob)对应的文件路径来得到它.

```
master:/path/to/file
```

区间

最后, 你可以用".."来指两个提交(commit)之间的区间. 下面的命令会给出你在"7b593b5" 和"51bea1"之间除了"7b593b5外"的所有提交(commit)(注意:51bea1是最近的提交).

```
7b593b5..51bea1
```

这会包括所有 *从* 7b593b开始的提交(commit). 译者注: 相当于 7b593b..HEAD

```
7b593b..
```

追踪分支

在Git中‘追踪分支’是用与联系本地分支和远程分支的. 如果你在’追踪分支'(Tracking Branches)上执行推送(push)或拉取(pull)时， 它会自动推送(push)或拉取(pull)到关联的远程分支上.

如果你经常要从远程仓库里拉取(pull)分支到本地,并且不想很麻烦的使用"git pull "这种格式; 那么就应当使用‘追踪分支'(Tracking Branches).

‘git clone‘命令会自动在本地建立一个'master'分支，它是'origin/master'的‘追踪分支’. 而'origin/master'就是被克隆(clone)仓库的'master'分支.

译者注: origin一般是指原始仓库地址的别名.

你可以在使用'git branch'命令时加上'--track'参数, 来手动创建一个'追踪分支'.

```
git branch --track experimental origin/experimental
```

当你运行下命令时:

```
$ git pull experimental
```

它会自动从‘origin'抓取(fetch)内容，再把远程的'origin/experimental'分支合并进(merge)本地的'experimental'分支.

当要把修改推送(push)到origin时, 它会将你本地的'experimental'分支中的修改推送到origin的'experimental'分支里, 而无需指定它(origin).

使用GIT GREP进行搜索

用git grep 命令查找Git库里面的某段文字是很方便的. 当然, 你也可以用unix下的'grep'命令进行搜索, 但是'git grep'命令能让你不用签出(checkout)历史文件, 就能查找它们.

例如, 你要看 git.git　这个仓库里每个使用'xmmap'函数的地方, 你可以运行下面的命令:

```
$ git grep xmmap
config.c:               contents = xmmap(NULL, contents_sz, PROT_READ,
diff.c:         s->data = xmmap(NULL, s->size, PROT_READ, MAP_PRIVATE, fd, 0);
git-compat-util.h:extern void *xmmap(void *start, size_t length, int prot, int fla
read-cache.c:   mmap = xmmap(NULL, mmap_size, PROT_READ | PROT_WRITE, MAP_PRIVATE,
refs.c: log_mapped = xmmap(NULL, mapsz, PROT_READ, MAP_PRIVATE, logfd, 0);
sha1_file.c:    map = xmmap(NULL, mapsz, PROT_READ, MAP_PRIVATE, fd, 0);
sha1_file.c:    idx_map = xmmap(NULL, idx_size, PROT_READ, MAP_PRIVATE, fd, 0);
sha1_file.c:                    win->base = xmmap(NULL, win->len,
sha1_file.c:                    map = xmmap(NULL, *size, PROT_READ, MAP_PRIVATE, f
sha1_file.c:            buf = xmmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);
wrapper.c:void *xmmap(void *start, size_t length,
```

如果你要显示行号, 你可以添加'-n'选项:

```
$>git grep -n xmmap
config.c:1016:          contents = xmmap(NULL, contents_sz, PROT_READ,
diff.c:1833:            s->data = xmmap(NULL, s->size, PROT_READ, MAP_PRIVATE, fd,
git-compat-util.h:291:extern void *xmmap(void *start, size_t length, int prot, int
read-cache.c:1178:      mmap = xmmap(NULL, mmap_size, PROT_READ | PROT_WRITE, MAP_
```

```
refs.c:1345:    log_mapped = xmmap(NULL, mapsz, PROT_READ, MAP_PRIVATE, logfd, 0);
sha1_file.c:377:        map = xmmap(NULL, mapsz, PROT_READ, MAP_PRIVATE, fd, 0);
sha1_file.c:479:        idx_map = xmmap(NULL, idx_size, PROT_READ, MAP_PRIVATE, fd
sha1_file.c:780:                        win->base = xmmap(NULL, win->len,
sha1_file.c:1076:                       map = xmmap(NULL, *size, PROT_READ, MAP_PR
sha1_file.c:2393:               buf = xmmap(NULL, size, PROT_READ, MAP_PRIVATE, fd
wrapper.c:89:void *xmmap(void *start, size_t length,
```

如果我们想只显示文件名, 我们可以使用'--name-onley'选项:

```
$>git grep --name-only xmmap
config.c
diff.c
git-compat-util.h
read-cache.c
refs.c
sha1_file.c
wrapper.c
```

我们可以用'-c'选项,可以查看每个文件里有多少行匹配内容(line matches):

```
$>git grep -c xmmap
config.c:1
diff.c:1
git-compat-util.h:1
read-cache.c:1
refs.c:1
sha1_file.c:5
wrapper.c:1
```

现在, 如果我们要查找git仓库里某个特定版本里的内容, 我们可以像下面一样在命令行末尾加上标签名(tag reference):

```
$ git grep xmmap v1.5.0
v1.5.0:config.c:                contents = xmmap(NULL, st.st_size, PROT_READ,
v1.5.0:diff.c:          s->data = xmmap(NULL, s->size, PROT_READ, MAP_PRIVATE, fd,
v1.5.0:git-compat-util.h:static inline void *xmmap(void *start, size_t length,
v1.5.0:read-cache.c:                    cache_mmap = xmmap(NULL, cache_mmap_size,
v1.5.0:refs.c:  log_mapped = xmmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, logfd
v1.5.0:sha1_file.c:     map = xmmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, fd,
v1.5.0:sha1_file.c:     idx_map = xmmap(NULL, idx_size, PROT_READ, MAP_PRIVATE, fd
v1.5.0:sha1_file.c:                     win->base = xmmap(NULL, win->len,
v1.5.0:sha1_file.c:     map = xmmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, fd,
v1.5.0:sha1_file.c:             buf = xmmap(NULL, size, PROT_READ, MAP_PRIVATE, fd
```

我可以看到"1.5.0版"和当前版本间一些区别: 在"1.5.0版"中, xmmap没有在wrapper.c中出现.

我们也可以组合一些搜索条件, 下面的命令就是查找我们在仓库的哪个地方定义了'SORT_DIRENT'.

```
$ git grep -e '#define' --and -e SORT_DIRENT
builtin-fsck.c:#define SORT_DIRENT 0
builtin-fsck.c:#define SORT_DIRENT 1
```

我不但可以进行"与"(*both*)条件搜索操作，也可以进行"或"(*either*)条件搜索操作.

```
$ git grep --all-match -e '#define' -e SORT_DIRENT
builtin-fsck.c:#define REACHABLE 0x0001
builtin-fsck.c:#define SEEN      0x0002
builtin-fsck.c:#define ERROR_OBJECT 01
builtin-fsck.c:#define ERROR_REACHABLE 02
builtin-fsck.c:#define SORT_DIRENT 0
builtin-fsck.c:#define DIRENT_SORT_HINT(de) 0
builtin-fsck.c:#define SORT_DIRENT 1
builtin-fsck.c:#define DIRENT_SORT_HINT(de) ((de)->d_ino)
builtin-fsck.c:#define MAX_SHA1_ENTRIES (1024)
builtin-fsck.c: if (SORT_DIRENT)
```

我们也可以查找出符合一个条件(term)且符合两个条件(terms)之一的文件行. 例如我们要找出名字中含有'PATH'或是'MAX'的常量定义:

```
$ git grep -e '#define' --and \( -e PATH -e MAX \)
abspath.c:#define MAXDEPTH 5
builtin-blame.c:#define MORE_THAN_ONE_PATH      (1u<<13)
builtin-blame.c:#define MAXSG 16
builtin-describe.c:#define MAX_TAGS     (FLAG_BITS - 1)
builtin-fetch-pack.c:#define MAX_IN_VAIN 256
builtin-fsck.c:#define MAX_SHA1_ENTRIES (1024)
...
```

译者注: 就是"与"条件搜索和"或"条件搜索可以组合使用.

GIT的撤消操作 - 重置, 签出 和 撤消

Git提供了多种修复你开发过程中的错误的方法. 方法的选择取决于你的情况: 包含有错误的文件是否提交了(commited); 如果你把它已经提交了, 那么你是否把有错误的提交已与其它人共享这也很重要.

修复未提交文件中的错误(重置)

如果你现在的工作目录(work tree)里搞的一团乱麻, 但是你现在还没有把它们提交; 你可以通过下面的命令, 让工作目录回到上次提交时的状态(last committed state):

```
$ git reset --hard HEAD
```

这条件命令会把你工作目录中所有未提交的内容清空(当然这不包括未置于版控制下的文件 untracked files). 从另一种角度来说, 这会让"git diff" 和"git diff --cached"命令的显示法都变为空.

如果你只是要恢复一个文件,如"hello.rb", 你就要使用 git checkout

```
$ git checkout -- hello.rb
```

这条命令把hello.rb从HEAD中签出并且把它恢复成未修改时的样子.

译者:上面二行和原文有出入，经验证是原文有误,所以我据正确的重写了.

修复已提交文件中的错误

如果你已经做了一个提交(commit),但是你马上后悔了, 这里有两种截然不同的方法去处理这个问题:

1. 创建一个新的提交(commit), 在新的提交里撤消老的提交所作的修改. 这种作法在你已经把代码发布的情况下十分正确.

2　你也可以去修改你的老提交(old commit). 但是如果你已经把代码发布了,那么千万别这么做; git不会处理项目的历史会改变的情况,如果一个分支的历史被改变了那以后就不能正常的合并.

创建新提交来修复错误

创建一个新的，撤消(revert)了前期修改的提交(commit)是很容易的; 只要把出错的提交(commit)的名字(reference)做为参数传给命令: git revert就可以了; 下面这条命令就演示了如何撤消最近的一个提交:

```
$ git revert HEAD
```

这样就创建了一个撤消了上次提交(HEAD)的新提交, 你就有机会来修改新提交(new commit)里的提交注释信息.

你也可撤消更早期的修改, 下面这条命令就是撤消“上上次”(next-to-last)的提交:

```
$ git revert HEAD^
```

在这种情况下,git尝试去撤消老的提交,然后留下完整的老提交前的版本. 如果你最近的修改和要撤消的修改有重叠(overlap),那么就会被要求手工解决冲突(conflicts), 就像解决合并(merge)时出现的冲突一样.

译者注: git revert 其实不会直接创建一个提交(commit), 把撤消后的文件内容放到索引(index)里,你需要再执行git commit命令，它们才会成为真正的提交(commit).

修改提交来修复错误

如果你刚刚做了某个提交(commit), 但是你又想马上修改这个提交; git commit 现在支持一个叫**--amend**的参数，它能让你修改刚才的这个提交(HEAD commit). 这项机制能让你在代码发布前,添加一些新的文件或是修改你的提交注释(commit message).

如果你在老提交(older commit)里发现一个错误, 但是现在还没有发布到代码服务器上. 你可以使用 git rebase命令的交互模式, "git rebase -i"会提示你在编辑中做相关的修改. 这样其实就是让你在rebase的过程来修改提交.

维护GIT

保证良好的性能

在大的仓库中, git靠压缩历史信息来节约磁盘和内存空间.

压缩操作并不是自动进行的, 你需要手动执行 git gc:

```
$ git gc
```

压缩操作比较耗时, 你运行git gc命令最好是在你没有其它工作的时候.

保持可靠性

git fsck 运行一些仓库的一致性检查, 如果有任何问题就会报告. 这项操作也有点耗时, 通常报的警告就是“悬空对象"(dangling objects).

```
$ git fsck
dangling commit 7281251ddd2a61e38657c827739c57015671a6b3
dangling commit 2706a059f258c6b245f298dc4ff2ccd30ec21a63
dangling commit 13472b7c4b80851a1bc551779171dcb03655e9b5
dangling blob 218761f9d90712d37a9c5e36f406f92202db07eb
dangling commit bf093535a34a4d35731aa2bd90fe6b176302f14f
dangling commit 8e4bec7f2ddaa268bef999853c25755452100f8e
dangling tree d50bb86186bf27b681d25af89d3b5b68382e4085
dangling tree b24c2473f1fd3d91352a624795be026d64c8841f
...
```

“悬空对象"(dangling objects)并不是问题, 最坏的情况只是它们多占了一些磁盘空间. 有时候它们是找回丢失的工作的最后一丝希望.

建立一个公共仓库

假设你个人的仓库在目录 ~/proj. 我们先克隆一个新的“裸仓库“,并且创建一个标志文件告诉git-daemon这是个公共仓库.

```
$ git clone --bare ~/proj proj.git
$ touch proj.git/git-daemon-export-ok
```

上面的命令创建了一个proj.git目录, 这个目录里有一个“裸git仓库" -- 即只有'.git'目录里的内容,没有任何签出(checked out)的文件.

下一步就是你把这个 proj.git 目录拷到你打算用来托管公共仓库的主机上. 你可以用scp, rsync或其它任何方式.

通过git协议导出git仓库

用git协议导出git仓库, 这是推荐的方法.

如果这台服务器上有管理员 ，TA们要告诉你把仓库放在哪一个目录中, 并且“git:// URL”除仓库目录部分外是什么.

你现在要做的是启动 git daemon; 它会监听在 9418端口. 默认情况下它会允许你访问所有的git目录(看目录中是否有git-daemon-export-ok文件). 如果以某些目录做为 git-daemon 的参数, 那么 git-daemon 会限制用户通过git协议只能访问这些目录.

你可以在inetd service模式下运行 git-daemon; 点击 git daemon 可以查看帮助信息.

通过http协议导出git仓库

git协议有不错的性能和可靠性, 但是如果主机上已经配好了一台web服务器,使用http协议(git over http)可能会更容易配置一些.

你需要把新建的"裸仓库"放到Web服务器的可访问目录里, 同时做一些调整,以便让web客户端获得它们所需的额外信息.

```
$ mv proj.git /home/you/public_html/proj.git
$ cd proj.git
$ git --bare update-server-info
$ chmod a+x hooks/post-update
```

(最后两行命令的解释可以点击这里查看: git update-server-info & githooks.)

拼好了proj.git的web URL, 任何人都可以从这个地址来克隆(clone)或拉取(pull) git仓库内容. 下面这个命令就是例子:

```
$ git clone http://yourserver.com/~you/proj.git
```

建立一个私有仓库

如果不使用第三方的代码托管服务,而是要自己在服务器上建一个网上可访问的私有代码仓库, 你有几种选择:

通过SSH协议来访问仓库

通常最简单的办法是通ssh协议访问Git(Git Over SSH). 如果你在一台机器上有了一个ssh帐号, 你只要把“git裸仓库"放到任何一个可以通过ssh访问的目录, 然后可以像ssh登录一样简单的使用它. 假设你现在有一个仓库，并且你要把它

建成可以在网上可访问的私有仓库. 你可以用下面的命令, 导出一个"裸仓库", 然后用scp命令把它们拷到你的服务器上:

```
$ git clone --bare /home/user/myrepo/.git /tmp/myrepo.git
$ scp -r /tmp/myrepo.git myserver.com:/opt/git/myrepo.git
```

如果其它人也在 myserver.com　这台服务器上有ssh帐号，那么TA也可以从这台服务器上克隆(clone)代码:

```
$ git clone myserver.com:/opt/git/myrepo.git
```

上面的命令会提示你输入ssh密码或是使用公钥(public key).

译者注1:配置ssh公钥的方法可以参考这里,这样在ssh访问时就可以不要输入命令.

译者注2:git over ssh方式对仓库有读写权限, git://协议只能读仓库.

使用Gitosis的多用户访问

如果你不想为每个用户配置不同的帐号,你可以用一个叫Gitosis的工具. 在gitosis中, 有一个叫 authorized_keys 的文件，里面包括了所有授权可以访问仓库的用户的公钥(public key), 这样每个用户就可以直接使用'git'用户来推送(push)和拉(pull)代码.

安装与配置Gitosis(英文)

译者注1: github.com就是采用这种方式来配置私有(仓库)访问.

译者注 2 : Gitosis配置(中文)

Chapter 5

高级技能

创建新的空分支

在偶尔的情况下，你可能会想要保留那些与你的代码没有共同祖先的分支。例如在这些分支上保留生成的文档或者其他一些东西。如果你需要创建一个不使用当前代码库作为父提交的分支，你可以用如下的方法创建一个空分支：

```
git symbolic-ref HEAD refs/heads/newbranch
rm .git/index
git clean -fdx
<do work>
git add your files
git commit -m 'Initial commit'
```

gitcast:c9-empty-branch

修改你的历史

交互式衍合是修改单个提交的好方法。

git filter-branch是修改大量提交的好方法。

高级分支与合并

在合并过程中得到解决冲突的协助

git会把所有可以自动合并的修改加入到索引中去, 所以git diff只会显示有冲突的部分. 它使用了一种不常见的语法:

```
$ git diff
diff --cc file.txt
index 802992c,2b60207..0000000
--- a/file.txt
+++ b/file.txt
@@@ -1,1 -1,1 +1,5 @@@
++<<<<<<< HEAD:file.txt
 +Hello world
++=======
+ Goodbye
++>>>>>>> 77976da35a11db4580b80ae27e8d65caf5208086:file.txt
```

回忆一下, 在我们解决冲突之后, 得到的提交会有两个而不是一个父提交: 一个父提交会成为HEAD, 也就是当前分支的tip; 另外一个父提交会成为另一分支的tip, 被暂时存在MERGE_HEAD.

在合并过程中, 索引中保存着每个文件的三个版本. 三个"文件暂存(file stage)"中的每一个都代表了文件的不同版本:

```
$ git show :1:file.txt  # 两个分支共同祖先中的版本.
$ git show :2:file.txt  # HEAD中的版本.
$ git show :3:file.txt  # MERGE_HEAD中的版本.
```

当你使用git diff去显示冲突时, 它在工作树(work tree), 暂存2(stage 2)和暂存3(stage 3)之间执行三路diff操作, 只显示那些两方都有的块(换句话说, 当一个块的合并结果只从暂存2中得到时, 是不会被显示出来的; 对于暂存3来说也是一样).

上面的diff结果显示了file.txt在工作树, 暂存2和暂存3中的差异. git不在每行前面加上单个'+'或者'-', 相反地, 它使用两栏去显示差异: 第一栏用于显示第一个父提交与工作目录文件拷贝的差异, 第二栏用于显示第二个父提交与工作文件拷贝的差异. (参见git diff-files中的"COMBINED DIFF FORMAT"取得此格式详细信息.)

在用直观的方法解决冲突之后(但是在更新索引之前), diff输出会变成下面的样子:

```
$ git diff
diff --cc file.txt
index 802992c,2b60207..0000000
--- a/file.txt
+++ b/file.txt
@@@ -1,1 -1,1 +1,1 @@@
- Hello world
-Goodbye
++Goodbye world
```

上面的输出显示了解决冲突后的版本删除了第一个父版本提供的"Hello world"和第二个父版本提供的"Goodbye", 然后加入了两个父版本中都没有的"Goodbye world".

一些特别diff选项允许你对比工作目录和三个暂存中任何一个的差异:

```
$ git diff -1 file.txt      # 与暂存1进行比较
$ git diff --base file.txt          # 与上相同
```

```
$ git diff -2 file.txt      # 与暂存2进行比较
$ git diff --ours file.txt          # 与上相同
$ git diff -3 file.txt      # 与暂存3进行比较
$ git diff --theirs file.txt    # 与上相同.
```

git log和gitk命令也为合并操作提供了特别的协助:

```
$ git log --merge
$ gitk --merge
```

这会显示所有那些只在HEAD或者只在MERGE_HEAD中存在的提交, 还有那些更新(touch)了未合并文件的提交.

你也可以使用git mergetool, 它允许你使用外部工具如emacs或kdiff3去合并文件.

每次你解决冲突之后, 应该更新索引:

```
$ git add file.txt
```

完成索引更新之后, git-diff(缺省地)不再显示那个文件的差异, 所以那个文件的不同暂存版本会被"折叠"起来.

多路合并

你可以一次合并多个头, 只需简单地把它们作为git merge的参数列出. 例如,

```
$ git merge scott/master rick/master tom/master
```

相当于:

```
$ git merge scott/master
$ git merge rick/master
$ git merge tom/master
```

子树

有时会出现你想在自己项目中引入其他独立开发项目的内容的情况. 在没有路径冲突的前提下, 你只需要简单地从其他项目拉取内容即可.

如果有冲突的文件, 那么就会出现问题. 可能的例子包括Makefile和其他一些标准文件名. 你可以选择合并这些冲突的文件, 但是更多的情况是你不愿意把它们合并. 一个更好解决方案是把外部项目作为一个子目录进行合并. 这种情况不被递归合并策略所支持, 所以简单的拉取是无用的.

在这种情况下, 你需要的是子树合并策略.

这下面例子中, 我们设定你有一个仓库位于/path/to/B (如果你需要的话, 也可以是一个URL). 你想要合并那个仓库的master分支到你当前仓库的dir-B子目录下.

下面就是你所需要的命令序列:

```
$ git remote add -f Bproject /path/to/B (1)
$ git merge -s ours --no-commit Bproject/master (2)
$ git read-tree --prefix=dir-B/ -u Bproject/master (3)
$ git commit -m "Merge B project as our subdirectory" (4)
$ git pull -s subtree Bproject master (5)
```

子树合并的好处就是它并没有给你仓库的用户增加太多的管理负担. 它兼容于较老(版本号小于1.5.2)的客户端, 克隆完成之后马上可以得到代码.

然而, 如果你使用子模块(submodule), 你可以选择不传输这些子模块对象. 这可能在子树合并过程中造成问题.

译者注: submodule是Git的另一种将别的仓库嵌入到本地仓库方法.

另外, 若你需要修改内嵌外部项目的内容, 使用子模块方式可以更容易地提交你的修改.

(from Using Subtree Merge)

查找问题的利器 - GIT BISECT

假设你在项目的'2.6.18'版上面工作, 但是你当前的代码(master)崩溃(crash)了. 有时解决这种问题的最好办法是: 手工逐步恢复(brute-force regression)项目历史, 找出是哪个提交(commit)导致了这个问题. 但是 linkgit:git-bisect1 可以更好帮你解决这个问题:

```
$ git bisect start
$ git bisect good v2.6.18
$ git bisect bad master
Bisecting: 3537 revisions left to test after this
[65934a9a028b88e83e2b0f8b36618fe503349f8e] BLOCK: Make USB storage depend on SCSI rather than selecting it [try
```

如果你现在运行"git branch", 会发现你现在所在的是"no branch"(译者注:这是进行git bisect的一种状态). 这时分支指向提交（commit):"69543", 此提交刚好是在"v2.6.18"和“master"中间的位置. 现在在这个分支里, 编译并测试项目代码, 查看它是否崩溃(crash). 假设它这次崩溃了, 那么运行下面的命令:

```
$ git bisect bad
Bisecting: 1769 revisions left to test after this
[7eff82c8b1511017ae605f0c99ac275a7e21b867] i2c-core: Drop useless bitmaskings
```

现在git自动签出(checkout)一个更老的版本. 继续这样做, 用"git bisect good","git bisect bad"告诉git每次签出的版本是否没有问题; 你现在可以注意一下当前的签出的版本, 你会发现git在用"二分查找(binary search)方法"签出"bad"和"good"之间的一个版本(commit or revison).

在这个项目(case)中, 经过13次尝试, 找出了导致问题的提交(guilty commit). 你可以用 git show 命令查看这个提交(commit), 找出是谁做的修改，然后写邮件给TA. 最后, 运行:

```
$ git bisect reset
```

这会到你之前(执行git bisect start之前)的状态.

注意: git-bisect 每次所选择签出的版本, 只是一个建议; 如果你有更好的想法, 也可以去试试手工选择一个不同的版本.

运行:

```
$ git bisect visualize
```

这会运行gitk, 界面上会标识出"git bisect"命令自动选择的提交(commit). 你可以选择一个相邻的提交(commit), 记住它的SHA串值, 用下面的命令把它签出来:

```
$ git reset --hard fb47ddb2db...
```

然后进行测试, 再根据测试結果执行"bisect good"或是"bisect bad"; 就这样反复执行, 直到找出问题为止.

译者注: 关于"git bisect start"后的分支状态, 译文和原文不一致. 原文是说执行"git bisect start"后会创建一个名为"bisect"的分支, 但是实际情况却是处于"no branch"的状态.

查找问题的利器 - GIT BLAME

如果你要查看文件的每个部分是谁修改的, 那么 git blame 就是不二选择. 只要运行'git blame [filename]', 你就会得到整个文件的每一行的详细修改信息:包括SHA串,日期和作者:

译者注: Git采用SHA1做为hash签名算法, 在本书中,作者为了表达方便,常常使用SHA来代指SHA1. 如果没有特别说明,本书中的SHA就是SHA1的代称.

```
$ git blame sha1_file.c
...
0fcfd160 (Linus Torvalds  2005-04-18 13:04:43 -0700    8)  */
0fcfd160 (Linus Torvalds  2005-04-18 13:04:43 -0700    9) #include "cache.h"
1f688557 (Junio C Hamano  2005-06-27 03:35:33 -0700   10) #include "delta.h"
a733cb60 (Linus Torvalds  2005-06-28 14:21:02 -0700   11) #include "pack.h"
8e440259 (Peter Eriksen   2006-04-02 14:44:09 +0200   12) #include "blob.h"
8e440259 (Peter Eriksen   2006-04-02 14:44:09 +0200   13) #include "commit.h"
8e440259 (Peter Eriksen   2006-04-02 14:44:09 +0200   14) #include "tag.h"
8e440259 (Peter Eriksen   2006-04-02 14:44:09 +0200   15) #include "tree.h"
f35a6d3b (Linus Torvalds  2007-04-09 21:20:29 -0700   16) #include "refs.h"
70f5d5d3 (Nicolas Pitre   2008-02-28 00:25:19 -0500   17) #include "pack-revindex.h"628522ec (Junio C Hamano
...
```

如果文件被修改了(reverted),或是编译(build)失败了; 这个命令就可以大展身手了.

你也可以用"-L"参数在命令(blame)中指定开始和结束行:

```
$>git blame -L 160,+10 sha1_file.c
ace1534d (Junio C Hamano 2005-05-07 00:38:04 -0700       160)}
ace1534d (Junio C Hamano 2005-05-07 00:38:04 -0700       161)
0fcfd160 (Linus Torvalds 2005-04-18 13:04:43 -0700       162)/*
0fcfd160 (Linus Torvalds 2005-04-18 13:04:43 -0700       163) * NOTE! This returns a statically allocate
```

```
790296fd (Jim Meyering   2008-01-03 15:18:07 +0100      164) * careful about using it. Do an "xstrdup()
0fcfd160 (Linus Torvalds 2005-04-18 13:04:43 -0700      165) * filename.
ace1534d (Junio C Hamano 2005-05-07 00:38:04 -0700      166) *
ace1534d (Junio C Hamano 2005-05-07 00:38:04 -0700      167) * Also note that this returns the location
ace1534d (Junio C Hamano 2005-05-07 00:38:04 -0700      168) * SHA1 file can happen from any alternate
d19938ab (Junio C Hamano 2005-05-09 17:57:56 -0700      169) * DB_ENVIRONMENT environment variable if i
```

GIT和EMAIL

向一个项目提交补丁

如果你只做了少量的改动, 最简单的提交方法就是把它们做成补丁(patch)用邮件发出去:

首先, 使用git format-patch; 例如:

```
$ git format-patch origin
```

这会在当前目录生成一系统编号的补丁文件, 每一个补丁文件都包含了当前分支和origin/HEAD之间的差异内容.

然后你可以手工把这些文件导入你的Email客户端. 但是如果你需要一次发送很多补丁, 你可能会更喜欢使用git send-email脚本去自动完成这个工作. 在发送之前, 应当先到项目的邮件列表上咨询一下项目管理者, 了解他们管理这些补丁的方式.

向一个项目中导入补丁

Git也提供了一个名为git am的工具(am是"apply mailbox"的缩写)去应用那些通过Email寄来的系列补丁. 你只需要按顺序把所有包含补丁的消息存入单个的mailbox文件, 比如说"patches.mbox", 然后运行

```
$ git am -3 patches.mbox
```

Git会按照顺序应用每一个补丁; 如果发生了冲突, git会停下来让你手工解决冲突从而完成合并. ("-3"选项会让git执行合并操作; 如果你更喜欢中止并且不改动你的工作树和索引, 你可以省略"-3"选项.)

在解决冲突和更新索引之后, 你不需要再创建一个新提交, 只需要运行

```
$ git am --resolved
```

这时git会为你创建一个提交, 然后继续应用mailbox中余下的补丁.

最后的效果是, git产生了一系列提交, 每个提交是原来mailbox中的一个补丁, 补丁中的作者信息和提交日志也一并被记录下来.

定制GIT

git config

更改你的编辑器

```
$ git config --global core.editor emacs
```

添加别名

```
$ git config --global alias.last 'cat-file commit HEAD'

$ git last
tree c85fbd1996b8e7e5eda1288b56042c0cdb91836b
```

```
parent cdc9a0a28173b6ba4aca00eb34f5aabb39980735
author Scott Chacon <schacon@gmail.com> 1220473867 -0700
committer Scott Chacon <schacon@gmail.com> 1220473867 -0700

fixed a weird formatting problem

$ git cat-file commit HEAD
tree c85fbd1996b8e7e5eda1288b56042c0cdb91836b
parent cdc9a0a28173b6ba4aca00eb34f5aabb39980735
author Scott Chacon <schacon@gmail.com> 1220473867 -0700
committer Scott Chacon <schacon@gmail.com> 1220473867 -0700

fixed a weird formatting problem
```

添加颜色

所有的color.*选项请参见git config的文档

```
$ git config color.branch auto
$ git config color.diff auto
$ git config color.interactive auto
$ git config color.status auto
```

或者你可以通过color.ui选项把颜色全部打开:

```
$ git config color.ui true
```

提交模板

```
$ git config commit.template '/etc/git-commit-template'
```

日志格式

```
$ git config format.pretty oneline
```

其他配置选项

除上面提到的选项外, 还有很多很有趣的选项去配置打包, 垃圾回收, 合并, 分支, http传输, diff, 分页, 空白字符等等的行为. 如果你需要更加深入地调教git, 请阅读git config文档.

GIT HOOKS

钩子(hooks)是一些在"$GIT-DIR/hooks"目录的脚本, 在被特定的事件(certain points)触发后被调用。当"git init"命令被调用后, 一些非常有用的示例钩子文件(hooks)被拷到新仓库的hooks目录中; 但是在默认情况下这些钩子(hooks)是不生效的。 把这些钩子文件(hooks)的".sample"文件名后缀去掉就可以使它们生效了。

applypatch-msg

```
GIT_DIR/hooks/applypatch-msg
```

当'git-am'命令执行时，这个钩子就被调用。它只有一个参数：就是存有提交消息(commit log message)的文件的名字。如果钩子的执行结果是非零，那么补丁(patch)就不会被应用(apply)。

The hook is allowed to edit the message file in place, and can be used to normalize the message into some project standard format (if the project has one). It can also be used to refuse the commit after inspecting the message file. The default applypatch-msg hook, when enabled, runs the commit-msg hook, if the latter is enabled.

这个钩子用于在其它地方编辑提交消息，并且可以把这些消息规范成项目的标准格式(如果项目些类的标准的话)。它也可以在分析(inspect)完消息文件后拒绝此次提交(commit)。在默认情况下，当 applypatch-msg 钩子被启用时。。。。

()

pre-applypatch

```
GIT_DIR/hooks/pre-applypatch
```

当'git-am'命令执行时，这个钩子就被调用。它没有参数，并且是在一个补丁(patch)被应用后还未提交(commit)前被调用。如果钩子的执行结果是非零，那么刚才应用的补丁(patch)就不会被提交。

It can be used to inspect the current working tree and refuse to make a commit if it does not pass certain test. The default pre-applypatch hook, when enabled, runs the pre-commit hook, if the latter is enabled.

它用于检查当前的工作树，当提交的补丁不能通过特定的测试就拒绝将它提交(commit)进仓库。 ()

post-applypatch

```
GIT_DIR/hooks/post-applypatch
```

This hook is invoked by 'git-am'. It takes no parameter, and is invoked after the patch is applied and a commit is made.

当'git-am'命令执行时，这个钩子就被调用。它没有参数，并且是在一个补丁(patch)被应用且在完成提交(commit)情况下被调用。

This hook is meant primarily for notification, and cannot affect the outcome of 'git-am'.

这个钩子的主要用途是通知提示(notification)，它并不会影响'git-am'的执行和输出。

pre-commit

```
GIT_DIR/hooks/pre-commit
```

这个钩子被 'git-commit' 命令调用, 而且可以通过在命令中添加`\--no-verify` 参数来跳过。这个钩子没有参数，在得到提交消息和开始提交(commit)前被调用。如果钩子执行结果是非零，那么 'git-commit' 命令就会中止执行。

译注：此钩子可以用来在提交前检查代码错误(运行类似lint的程序)。

当默认的'pre-commit'钩子开启时，如果它发现文件尾部有空白行，那么就会中止此次提交。

译注：新版的默认钩子和这里所说有所有不同。

All the 'git-commit' hooks are invoked with the environment variable `GIT_EDITOR=:` if the command will not bring up an editor to modify the commit message.

下面是一个运行 Rspec 测试的 Ruby 脚本，如果没有通过这个测试，那么不允许提交(commit)。

```
html_path = "spec_results.html"
`spec -f h:#{html_path} -f p spec` # run the spec. send progress to screen. save html results to html_path

# find out how many errors were found
html = open(html_path).read
examples = html.match(/(\d+) examples/)[0].to_i rescue 0
failures = html.match(/(\d+) failures/)[0].to_i rescue 0
pending = html.match(/(\d+) pending/)[0].to_i rescue 0
```

```
if failures.zero?
  puts "0 failures! #{examples} run, #{pending} pending"
else
  puts "\aDID NOT COMMIT YOUR FILES!"
  puts "View spec results at #{File.expand_path(html_path)}"
  puts
  puts "#{failures} failures! #{examples} run, #{pending} pending"
  exit 1
end
```

prepare-commit-msg

`GIT_DIR/hooks/prepare-commit-msg`

当'git-commit'命令执行时：在编辑器(editor)启动前，默认提交消息准备好后，这个钩子就被调用。

It takes one to three parameters. The first is the name of the file that the commit log message. The second is the source of the commit message, and can be: `message` (if a `-m` or `-F` option was given); `template` (if a `-t` option was given or the configuration option `commit.template` is set); `merge` (if the commit is a merge or a `.git/MERGE_MSG` file exists); `squash` (if a `.git/SQUASH_MSG` file exists); or `commit`, followed by a commit SHA1 (if a `-c`, `-C` or `\--amend` option was given).

它有三个参数。第一个是提交消息文件的名字。第二个是提交消息的来源，它可以是：().

如果钩子的执行結果是非零的话，那么'git-commit'命令就会被中止执行。

The purpose of the hook is to edit the message file in place, and it is not suppressed by the `\--no-verify` option. A non-zero exit means a failure of the hook and aborts the commit. It should not be used as replacement for pre-commit hook.

The sample `prepare-commit-msg` hook that comes with git comments out the `Conflicts:` part of a merge's commit message.

commit-msg

`GIT_DIR/hooks/commit-msg`

当'git-commit'命令执行时，这个钩子被调用；也可以在命令中添加`\--no-verify`参数来跳过。这个钩子有一个参数：就是被选定的提交消息文件的名字。如这个钩子的执行結果是非零，那么'git-commit'命令就会中止执行。

The hook is allowed to edit the message file in place, and can be used to normalize the message into some project standard format (if the project has one). It can also be used to refuse the commit after inspecting the message file.

这个钩子的是为提交消息更适当，可以用于规范提交消息使之符合项目的标准(如果有的话)；如果它检查完提交消息后，发现内容不符合某些标准，它也可以拒绝此次提交(commit)。

The default 'commit-msg' hook, when enabled, detects duplicate "Signed-off-by" lines, and aborts the commit if one is found.

默认的'commit-msg'钩子启用后，它后检查里面是否有重复的签名结束线(Signed-off-by lines)，如果找到它就是中止此次提交(commit)操作。

post-commit

判断条件

`GIT_DIR/hooks/post-commit`

当'git-commit'命令执行时，这个钩子就被调用。它没有参数，并且是在一个提交(commit)完成时被调用。

这个钩子的主要用途是通知提示(notification)，它并不会影响'git-commit'的执行和输出。

pre-rebase

```
GIT_DIR/hooks/pre-rebase
```

当'git-base'命令执行时，这个钩子就被调用；主要目的是阻止那不应被rebase的分支被rebase(例如，一个已经发布的分支提交就不应被rebase)。

post-checkout

```
GIT_DIR/hooks/post-checkout
```

当'git-checkout'命令更新完整个工作树(worktree)后，这个钩子就会被调用。这个钩子有三个参数：前一个HEAD的ref，新HEAD的 ref，判断一个签出是分支签出还是文件签出的标识符(分支签出＝1，文件签出＝0)。这个钩子不会影响'git-checkout'命令的输出。

这个钩子可以用于检查仓库的一致性，自动显示签出前后的代码的区别，也可以用于设置目录的元数据属性。

post-merge

```
GIT_DIR/hooks/post-merge
```

This hook is invoked by 'git-merge', which happens when a 'git-pull' is done on a local repository. The hook takes a single parameter, a status flag specifying whether or not the merge being done was a squash merge. This hook cannot affect the outcome of 'git-merge' and is not executed, if the merge failed due to conflicts.

它有一个参数：

This hook can be used in conjunction with a corresponding pre-commit hook to save and restore any form of metadata associated with the working tree (eg: permissions/ownership, ACLS, etc). See contrib/hooks/setgitperms.perl for an example of how to do this.

pre-receive

```
GIT_DIR/hooks/pre-receive
```

This hook is invoked by 'git-receive-pack' on the remote repository, which happens when a 'git-push' is done on a local repository. Just before starting to update refs on the remote repository, the pre-receive hook is invoked. Its exit status determines the success or failure of the update.

当用户在本地仓库执行'git-push'命令时，服务器上运端仓库就会对应执行'git-receive-pack'命令，而'git-receive-pack'命令会调用 pre-receive 钩子。在开始更新远程仓库上的ref之前，这个钩子被调用。钩子的执行结果(exit status)决定此次更新能否成功。

This hook executes once for the receive operation. It takes no arguments, but for each ref to be updated it receives on standard input a line of the format:

每执行一个接收(receive)操作都会调用一次这个钩子。它没有命令行参数，但是它会从标准输入(standard input)读取需要更新的ref，格式如下：

SP SP LF

译者注：SP是空格，LF是回车。

where `<old-value>` is the old object name stored in the ref, `<new-value>` is the new object name to be stored in the ref and `<ref-name>` is the full name of the ref. When creating a new ref, `<old-value>` is 40 `0`.

<old-value>是保存在ref里的老对象的名字，<new-value>是保存在ref里的新对象的名字，<ref-name>就是此次要更新的ref的全名。如果是创建一个新的ref，那么<old-value>就是由40个0组成的字符串表示。

If the hook exits with non-zero status, none of the refs will be updated. If the hook exits with zero, updating of individual refs can still be prevented by the <<update,'update'>> hook.

如果钩子的执行结果是非零，那么没有引用(ref)会被更新。如果执行结果为零，更新操作还可以被后面的<<update,'update'>> 钩子所阻止。

Both standard output and standard error output are forwarded to 'git-send-pack' on the other end, so you can simply echo messages for the user.

钩子(hook)的标准输出和标准错误输出(stdout & stderr)都会通'git-send-pack'转发给客户端(other end)，你可以把这个信息回显(echo)给用户。

If you wrote it in Ruby, you might get the args this way:

如果你用ruby,那么可以像下面的代码一样得到它们的参数。

```
rev_old, rev_new, ref = STDIN.read.split(" ")
```

Or in a bash script, something like this would work:

在bash脚本中，下面代码也可能得到参数。

```
#!/bin/sh
# <oldrev> <newrev> <refname>
# update a blame tree
while read oldrev newrev ref
```

```
do
    echo "STARTING [$oldrev $newrev $ref]"
    for path in `git diff-tree -r $oldrev..$newrev | awk '{print $6}'`
    do
      echo "git update-ref refs/blametree/$ref/$path $newrev"
      `git update-ref refs/blametree/$ref/$path $newrev`
    done
done
```

update

```
GIT_DIR/hooks/update
```

当用户在本地仓库执行'git-push'命令时，服务器上运端仓库就会对应执行'git-receive-pack'，而'git-receive-pack'会调用 update 钩子。在更新远程仓库上的ref之前，这个钩子被调用。钩子的执行结果(exit status)决定此次update能否成功。

每更新一个引用(ref)，钩子就会被调用一次，并且使用三个参数:

- the name of the ref being updated, # 要被更的ref的名字
- the old object name stored in the ref, # ref 中更新前的对象名
- and the new objectname to be stored in the ref. # ref 中更新后的对象名

如果 update hook 的执行结果是零，那么引用(ref)就会被更新。如果执行结果是非零，那么'git-receive-pack'就不会更新这个引用(ref)。

This hook can be used to prevent 'forced' update on certain refs by making sure that the object name is a commit object that is a descendant of the commit object named by the old object name. That is, to enforce a "fast forward only" policy.

这个钩子也可以用于防止强制更新某些 refs，确保old object是new object的父对象。这样也就是强制执行"fast forward only"策略。

It could also be used to log the old..new status. However, it does not know the entire set of branches, so it would end up firing one e-mail per ref when used naively, though. The <<post-receive,'post-receive'>> hook is more suited to that.

它也可以用于跟踪(log)更新详情。但是由于它不知道每次更新的ref全体集合，尽管可以傻傻的每个ref更新就发送email；但是<<post-receive,'post-receive'>>钩子更适合这种情况。

在邮件列表(mailing list)上讲了另外一种用法：用这个 update hook 实现细粒度(finer grained)权限控制。

钩子(hook)的标准输出和标准错误输出(stdout & stderr)都会通'git-send-pack'转发给客户端(other end)，你可以把这个信息回显(echo)给用户。

当默认的 update hook 被启用，且`hooks.allowunannotated`选项被打开时，那么没有注释(unannotated)的标签就不能被推送到服务器上。

post-receive

```
GIT_DIR/hooks/post-receive
```

This hook is invoked by 'git-receive-pack' on the remote repository, which happens when a 'git-push' is done on a local repository. It executes on the remote repository once after all the refs have been updated.

当用户在本地仓库执行'git-push'命令时，服务器上运端仓库就会对应执行'git-receive-pack'命令；在所有远程仓库的引用(ref)都更新后，这个钩子就会被'git-receive-pack'调用。

This hook executes once for the receive operation. It takes no arguments, but gets the same information as the <<pre-receive,'pre-receive'>> hook does on its standard input.

服务器端仓库每次执行接收(receive)操作时，这个钩子就会被调用。此钩子执行不带任何命令行参数，但是和<<pre-receive,'pre-receive'>>钩子一样从标准输入(standard input)读取信息，并且读取的信息内容也是一样的。

This hook does not affect the outcome of 'git-receive-pack', as it is called after the real work is done.

这个钩子不会影响'git-receive-pack'命令的输出，因为它是在命令执行完后被调用的。

This supersedes the <<post-update,'post-update'>> hook in that it gets both old and new values of all the refs in addition to their names.

这个钩子可以取代 <<post-update,'post-update'>>钩子；因为后者只能得到需要更新的ref的名字，而没有更新前后的对象的名字。

Both standard output and standard error output are forwarded to 'git-send-pack' on the other end, so you can simply `echo` messages for the user.

钩子(hook)的标准输出和标准错误输出(stdout & stderr)都会通'git-send-pack'转发给客户端(other end)，你可以把这个信息回显(echo)给用户。

The default 'post-receive' hook is empty, but there is a sample script `post-receive-email` provided in the `contrib/hooks` directory in git distribution, which implements sending commit emails.

默认的'post-receive'的钩子是空的，但是在git distribution `contrib/hooks` 目录里有一个名为 `post-receive-email` 的示例脚本，实实了发送commit emails的功能。

post-update

```
GIT_DIR/hooks/post-update
```

This hook is invoked by 'git-receive-pack' on the remote repository, which happens when a 'git-push' is done on a local repository. It executes on the remote repository once after all the refs have been updated.

当用户在本地仓库执行'git-push'命令时，服务器上运端仓库就会对应执行'git-receive-pack'。在所有远程仓库的引用(ref)都更新后，post-update 就会被调用。

It takes a variable number of parameters, each of which is the name of ref that was actually updated.

它的参数数目是可变的，每个参数代表实际被更新的 ref。

This hook is meant primarily for notification, and cannot affect the outcome of 'git-receive-pack'.

这个钩子的主要用途是通知提示(notification)，它并不会影响'git-receive-pack'的输出。

The 'post-update' hook can tell what are the heads that were pushed, but it does not know what their original and updated values are, so it is a poor place to do log old..new. The <<post-receive,'post-receive'>> hook does get both original and updated values of the refs. You might consider it instead if you need them.

'post-update'可以行诉我们哪些 heads 被更新了，但是它不知道head更新前后的值，所以这里不大适合记录更新详情。而<<post-receive,'post-receive'>>钩子可以得到ref(也可说是head)更新前后的值，如果你要记录更详情的话，可以考虑使用这个钩子。

When enabled, the default 'post-update' hook runs 'git-update-server-info' to keep the information used by dumb transports (e.g., HTTP) up-to-date. If you are publishing a git repository that is accessible via HTTP, you should probably enable this hook.

如果默认的'post-update'钩子启用的话，它们执行'git-update-server-info'命令去更新一些dumb协议(如http)所需要的信息。如果你的git仓库是通http协议来访问，那么你就应该开启它。

Both standard output and standard error output are forwarded to 'git-send-pack' on the other end, so you can simply echo messages for the user.

钩子(hook)的标准输出和标准错误输出(stdout & stderr)都会通'git-send-pack'转发给客户端(other end)，你可以把这个信息回显(echo)给用户。

pre-auto-gc

```
GIT_DIR/hooks/pre-auto-gc
```

当调用'git-gc --auto'命令时，这个钩子(hook)就会被调用。它没有调用参数，如果钩子的执行結果是非零的话，那么'git-gc --auto'命令就会中止执行。

参考

Git Hooks * http://probablycorey.wordpress.com/2008/03/07/git-hooks-make-me-giddy/

找回丢失的对象

译者注: 原书这里只有两个链接： Recovering Lost Commits Blog Post， Recovering Corrupted Blobs by Linus

我根据第一个链接，整理了一篇博文，并把它做为原书补充。

在玩git的过程中，常有失误的时候，有时把需要的东东给删了。 不过没有关系，git给了我们一层安全网，让们能有机会把失去的东东给找回来。

Let's go!

准备

我们先创建一个用以实验的仓库，在里面创建了若干个提交和分支。 BTW：你可以直接把下面的命令复制到shell里执行。

```
mkdir recovery;cd recovery
git init
touch file
git add file
git commit -m "First commit"
echo "Hello World" > file
git add .
git commit -m "Greetings"
git branch cool_branch
git checkout cool_branch
echo "What up world?" > cool_file
git add .
git commit -m "Now that was cool"
git checkout master
echo "What does that mean?" >> file
```

恢复已删除分支提交

现在repo里有两个branch

```
$ git branch
cool_branch
* master
```

存储当前仓库未提交的改动

```
$ git stash save "temp save"
Saved working directory and index state On master: temp save
HEAD is now at e3c9b6b Greetings
```

删除一个分支

```
$ git branch -D cool_branch
Deleted branch cool_branch (was 2e43cd5).

$ git branch
 * master
```

用`git fsck --lost-found`命令找出刚才删除的分支里面的提交对象。

```
$git fsck --lost-found
  dangling commit 2e43cd56ee4fb08664cd843cd32836b54fbf594a
```

用git show命令查看一个找到的对象的内容，看是否为我们所找的。

```
git show 2e43cd56ee4fb08664cd843cd32836b54fbf594a

  commit 2e43cd56ee4fb08664cd843cd32836b54fbf594a
  Author: liuhui <liuhui998[#]gmail.com>
  Date:   Sat Oct 23 12:53:50 2010 +0800

  Now that was cool
```

```
diff --git a/cool_file b/cool_file
new file mode 100644
index 0000000..79c2b89
--- /dev/null
+++ b/cool_file
@@ -0,0 +1 @@
+What up world?
```

这个提交对象确实是我们在前面删除的分支的内容；下面我们就要考虑一下要如何来恢复它了。

使用git rebase　进行恢复

```
$git rebase 2e43cd56ee4fb08664cd843cd32836b54fbf594a
First, rewinding head to replay your work on top of it...
Fast-forwarded master to 2e43cd56ee4fb08664cd843cd32836b54fbf594a.
```

现在我们用git log命令看一下，看看它有没有恢复:

```
$ git log

commit 2e43cd56ee4fb08664cd843cd32836b54fbf594a
Author: liuhui <liuhui998[#]gmail.com>
Date:   Sat Oct 23 12:53:50 2010 +0800

Now that was cool

commit e3c9b6b967e6e8c762b500202b146f514af2cb05
Author: liuhui <liuhui998[#]gmail.com>
Date:   Sat Oct 23 12:53:50 2010 +0800

Greetings
```

```
commit 5e90516a4a369be01b54323eb8b2660545051764
Author: liuhui <liuhui998[#]gmail.com>
Date:   Sat Oct 23 12:53:50 2010 +0800

First commit
```

提交是找回来，但是分支没有办法找回来：

```
liuhui@liuhui:~/work/test/git/recovery$ git branch
* master
```

使用git merge　进行恢复

我们把刚才的恢复的提交删除

```
$ git reset --hard HEAD^
HEAD is now at e3c9b6b Greetings
```

再把刚删的提交给找回来：

```
git fsck --lost-found
dangling commit 2e43cd56ee4fb08664cd843cd32836b54fbf594a
```

不过这回我们用是合并命令进行恢复：

```
$ git merge 2e43cd56ee4fb08664cd843cd32836b54fbf594a
Updating e3c9b6b..2e43cd5
Fast-forward
cool_file |    1 +
```

```
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 cool_file
```

git stash的恢复

前面我们用git stash把没有提交的内容进行了存储，如果这个存储不小心删了怎么办呢？

当前repo里有的存储：

```
$ git stash list
stash@{0}: On master: temp save
```

把它们清空：

```
$git stash clear
liuhui@liuhui:~/work/test/git/recovery$ git stash list
```

再用git fsck --lost-found找回来：

```
$git fsck --lost-found
dangling commit 674c0618ca7d0c251902f0953987ff71860cb067
```

用git show看一下回来的内容对不对：

```
$git show 674c0618ca7d0c251902f0953987ff71860cb067

commit 674c0618ca7d0c251902f0953987ff71860cb067
Merge: e3c9b6b 2b2b41e
Author: liuhui <liuhui998[#]gmail.com>
Date:   Sat Oct 23 13:44:49 2010 +0800
```

```
    On master: temp save

diff --cc file
index 557db03,557db03..f2a8bf3
--- a/file
+++ b/file
@@ -1,1 -1,1 +1,2 @@
  Hello World
  ++What does that mean?
```

看起来没有问题，好的，那么我就把它恢复了吧：

```
$ git merge 674c0618ca7d0c251902f0953987ff71860cb067
Merge made by recursive.
 file |    1 +
  1 files changed, 1 insertions(+), 0 deletions(-)
```

备注

这篇文章主要内容来自这里：The illustrated guide to recovering lost commits with Git,我做了一些整理的工作。

如果对于文中的一些命令不熟，可以参考Git Community Book中文版

其实这里最重要的一个命令就是：git fsck --lost-found，因为git中把commit删了后，并不是真正的删除，而是变成了悬空对象（dangling commit）。我们只要把把这悬空对象（dangling commit）找出来，用git rebase也好，用git merge也行就能把它们给恢复。

子模块

一个大项目通常由很多较小的, 自完备的模块组成. 例如, 一个嵌入式Linux发行版的代码树会包含每个进行过本地修改的软件的代码; 一个电影播放器可能需要基于一个知名解码库的特定版本完成编译; 数个独立的程序可能会共用同一个创建脚本.

在集中式版本管理系统中, 可以通过把每个模块放在一个单独的仓库中来完成上述的任务. 开发者可以把所有模块都签出(checkout), 也可以选择只签出他需要的模块. 在移动文件, 修改API和翻译时, 他们甚至可以在一个提交中跨多个模块修改文件.

Git不允许部分签出(partial checkout), 所以采用上面(集中式版本管理)的方法会强迫开发者们保留一份他们不感兴趣的模块的本地拷贝. 在签出量巨大时, 提交会慢得超过你的预期, 因为git不得不扫描每一个目录去寻找修改. 如果模块有很多本地历史, 克隆可能永远不能完成.

从好的方面看来, 分布式版本管理系统可以更好地与外部资源进行整合. 在集中化的模式中, 外部项目的一个快照从它本身的版本控制系统中被分离出来, 然后此快照作为一个提供商分支(vendor branch)导入到本地的版本控制系统中去. 快照的历史不再可见. 而分布式管理系统中, 你可以把外部项目的历史一同克隆过来, 从而更好地跟踪外部项目的开发, 便于合并本地修改.

Git的子模块(submodule)功能使得一个仓库可以用子目录的形式去包含一个外部项目的签出版本. 子模块维护它们自己的身份标记(identity); 子模块功能仅仅储存子模块仓库的位置和提交ID, 因此其他克隆父项目("superproject")的开发者可以轻松克隆所有子模块的同一版本. 对父项目的部分签出成为可能: 你可以告诉git去克隆一部分或者所有的子模块, 也可以一个都不克隆.

Git 1.5.3中加入了git submodule这个命令. Git 1.5.2版本的用户可以查找仓库的子模块并且手工签出; 更早的版本不支持子模块功能.

为说明子模块的使用方法, 创建4个用作子模块的示例仓库:

```
$ mkdir ~/git
$ cd ~/git
$ for i in a b c d
do
    mkdir $i
    cd $i
    git init
    echo "module $i" > $i.txt
    git add $i.txt
    git commit -m "Initial commit, submodule $i"
    cd ..
done
```

现在创建父项目, 加入所有的子模块:

```
$ mkdir super
$ cd super
$ git init
$ for i in a b c d
do
    git submodule add ~/git/$i $i
done
```

注意: 如果你想对外发布你的父项目, 请不要使用本地的地址!

列出`git-submodule`创建文件:

```
$ ls -a
.  ..  .git  .gitmodules  a  b  c  d
```

`git-submodule add`命令进行了如下的操作:

- 它在当前目录下克隆各个子模块, 默认签出master分支.
- 它把子模块的克隆路径加入到gitmodules文件中, 然后把这个文件加入到索引, 准备进行提交.
- 它把子模块的当前提交ID加入到索引中, 准备进行提交.

提交父项目:

```
$ git commit -m "Add submodules a, b, c and d."
```

现在克隆父项目:

```
$ cd ..
$ git clone super cloned
$ cd cloned
```

子模块的目录创建好了, 但是它们是空的:

```
$ ls -a a
.  ..
$ git submodule status
-d266b9873ad50488163457f025db7cdd9683d88b a
-e81d457da15309b4fef4249aba9b50187999670d b
-c1536a972b9affea0f16e0680ba87332dc059146 c
-d96249ff5d57de5de093e6baff9e0aafa5276a74 d
```

注意: 上面列出的提交对象的名字会和你的项目中看到的有所不同, 但是它们应该和HEAD的提交对象名字一致. 你可以运行`git ls-remote ../git/a`进行检验.

拉取子模块需要进行两步操作. 首先运行`git submodule init`, 把子模块的URL加入到`.git/config`:

```
$ git submodule init
```

现在使用git-submodule update去克隆子模块的仓库和签出父项目中指定的那个版本:

```
$ git submodule update
$ cd a
$ ls -a
.  ..  .git  a.txt
```

git-submodule update和git-submodule add的一个主要区别就是git-submodule update签出一个指定的提交, 而不是该分支的tip. 它就像签出一个标签(tag): 头指针脱离, 你不在任何一个分支上工作.

```
$ git branch
* (no branch)
master
```

如何你需要对子模块进行修改, 同时头指针又是脱离的状态, 那么你应该创建或者签出一个分支, 进行修改, 发布子模块的修改, 然后更新父项目让其引用新的提交:

```
$ git checkout master
```

或者

```
$ git checkout -b fix-up
```

然后

```
$ echo "adding a line again" >> a.txt
$ git commit -a -m "Updated the submodule from within the superproject."
$ git push
$ cd ..
```

```
$ git diff
diff --git a/a b/a
index d266b98..261dfac 160000
--- a/a
+++ b/a
@@ -1 +1 @@
-Subproject commit d266b9873ad50488163457f025db7cdd9683d88b
+Subproject commit 261dfac35cb99d380eb966e102c1197139f7fa24
$ git add a
$ git commit -m "Updated submodule a."
$ git push
```

如果你想要更新子模块, 你应该在`git pull`之后运行`git submodule update`.

子模块方式的陷阱

你应该总是在发布父项目的修改之前发布子模块修改. 如果你忘记发布子模块的修改, 其他人就无法克隆你的仓库了:

```
$ cd ~/git/super/a
$ echo i added another line to this file >> a.txt
$ git commit -a -m "doing it wrong this time"
$ cd ..
$ git add a
$ git commit -m "Updated submodule a again."
$ git push
$ cd ~/git/cloned
$ git pull
$ git submodule update
error: pathspec '261dfac35cb99d380eb966e102c1197139f7fa24' did not match any file(s) known to git.
Did you forget to 'git add'?
Unable to checkout '261dfac35cb99d380eb966e102c1197139f7fa24' in submodule path 'a'
```

如果你暂存了一个更新过的子模块, 准备进行手工提交, 注意不要在路径后面加上斜杠. 如果加上了斜杠, git会认为你想要移除那个子模块然后签出那个目录内容到父仓库.

```
$ cd ~/git/super/a
$ echo i added another line to this file >> a.txt
$ git commit -a -m "doing it wrong this time"
$ cd ..
$ git add a/
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    a
#       new file:   a/a.txt
#
# Modified submodules:
#
# * a aa5c351...0000000 (1):
#   < Initial commit, submodule a
#
```

为了修正这个错误的操作, 我们应该重置(reset)这个修改, 然后在add的时候不要加上末尾斜杠.

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

```
# Modified submodules:
#
# * a aa5c351...8d3ba36 (1):
#   > doing it wrong this time
#
```

你也不应该把子模块的分支回退到超出任何父项目中记录的提交的范围.

如果你在没有签出分支的情况下对子模块进行了修改并且提交, 运行`git submodule update`将会不安全. 你所进行的修改会在无任何提示的情况下被覆盖.

```
$ cat a.txt
module a
$ echo line added from private2 >> a.txt
$ git commit -a -m "line added inside private2"
$ cd ..
$ git submodule update
Submodule path 'a': checked out 'd266b9873ad50488163457f025db7cdd9683d88b'
$ cd a
$ cat a.txt
module a
```

注意: 这些修改在子模块的reflog中仍然可见.

如果你不想提交你的修改, 那又是另外一种情况了.

gitcast:c11-git-submodules

Chapter 6

Git生态体系

GIT 与之 WINDOWS

(mSysGit)[http://code.google.com/p/msysgit/]

gitcast:c10-windows-git

使用GIT进行系统部署

Capistrano 与 Git

GitHub Guide on Deploying with Cap

Git and Capistrano Screencast

与 SUBVERSION 集成

从其他代码管理工具迁移到GIT

你决定要把你的整个项目从原来的代码管理工具迁移到Git, 要怎么做才比较简单呢?

从Subversion导入

Git包含了一个名为git-svn的脚本, 它有一个克隆(clone)命令, 可以把一个Subversion仓库导入到一个新的Git仓库. GitHub上也有完成同样工作的免费工具.

```
$ git-svn clone http://my-project.googlecode.com/svn/trunk new-project
```

上面的命令会创建一个包含原来Subversion仓库全部历史记录的Git仓库. 通常这个操作会花相当长的时间, 因为它从第1个版本开始, 一个一个版本地签出, 然后再把这些版本进行本地提交.

从Perforce导入

在contrib/fast-import目录下, 你会找到git-p4脚本, 它会帮你导入Perforce仓库.

```
$ ~/git.git/contrib/fast-import/git-p4 clone //depot/project/main@all myproject
```

从其他管理工具导入

These are other SCMs that listed high on the Git Survey, should find import docs for them. !!TODO!!

- CVS
- Mercurial (hg)
- Bazaar-NG
- Darcs
- ClearCase

图形化的GIT

Git有不少图形化界面工具用于读取和维护仓库.

捆绑的GUI

Git自带了两个使用Tcl/Tk写成的GUI程序. Gitk是一个仓库浏览器, 也是一个历史信息可视化工具.

gitk

git gui是一个帮助你可视化索引操作的工具, 它支持add, remove和commit. 它不能取代命令行, 但是对于基本使用是足够的.

git gui

第三方项目

Mac用户可以参考 GitX and GitNub

Linux和其他一些Qt用户可以参考 QGit

GIT仓库托管

github 对于开源项目仓库是完全免费，只对私有(private)项目仓库收费。

bitbucket 支持不限数量的免费私有仓库，同时支持5个协作者；但是超过这个数量就要收钱了。

repo.or.cz

GIT的其它用法

ContentDistribution

TicGit

GIT的脚本支持

Ruby 与 Git

grit

jgit + jruby

PHP 与 Git

Python 与 Git

pygit

Perl 与 Git

perlgit

译者注:　此章的英文版原文也只是列出了大纲, 因此中文版看出起就很单薄, 以后我们尽量完善:)

GIT 与编辑器

textmate

eclipse

netbeans

Chapter 7

原理解析

GIT是如何存储对象的

这一章会详细讲解Git如何物理存储各对象.

所有的对象都以SHA值为索引用gzip格式压缩存储, 每个对象都包含了对象类型, 大小和内容.

Git中存在两种对象 - 松散对象(loose object)和打包对象(packed object).

松散对象

松散对象是一种比较简单格式. 它就是磁盘上的一个存储压缩数据的文件. 每一个对象都被写入一个单独文件中.

如果你对象的SHA值是`ab04d884140f7b0cf8bbf86d6883869f16a46f65`, 那么对应的文件会被存储在:

```
GIT_DIR/objects/ab/04d884140f7b0cf8bbf86d6883869f16a46f65
```

Git使用SHA值的前两个字符作为子目录名字, 所以一个目录中永远不会包含过多的对象. 文件名则是余下的38个字符.

可以用下面的Ruby代码说明对象数据是如何存储的:

```ruby
def put_raw_object(content, type)
  size = content.length.to_s

  header = "#{type} #{size}\0" # type(space)size(null byte)
  store = header + content

  sha1 = Digest::SHA1.hexdigest(store)
  path = @git_dir + '/' + sha1[0...2] + '/' + sha1[2..40]

  if !File.exists?(path)
    content = Zlib::Deflate.deflate(store)

    FileUtils.mkdir_p(@directory+'/'+sha1[0...2])
    File.open(path, 'w') do |f|
      f.write content
    end
  end
  return sha1
end
```

打包对象

另外一种对象存储方式是使用打包文件(packfile). 由于Git把每个文件的每个版本都作为一个单独的对象, 它的效率可能会十分的低. 设想一下在一个数千行的文件中改动一行, Git会把修改后的文件整个存储下来, 很浪费空间.

Git使用打包文件(packfile)去节省空间. 在这个格式中, Git只会保存第二个文件中改变了的部分, 然后用一个指针指向相似的那个文件(译注: 即第一个文件).

对象通常是以松散格式写到磁盘上, 因为这个格式的访问代价比较低. 然后, 你最终会需要把对象存放到打包格式中去节省磁盘空间 - 这个工作可以通过git gc来完成. 它使用一个相当复杂的启发式算法去决定哪些文件是最相似的, 然后基于此分析去计算差异. 可以存在多个打包文件, 在必要情况下, 它们可被解包(git unpack-objects)成为松散对象或者重新打包(git repack).

Git会为每一个打包文件创建一个较小的索引文件. 索引文件中包含了对象在打包文件中的偏移, 以便于通过SHA值来快速找到特定的对象.

打包文件的实现细节会在稍后的"打包文件"(Packfile)一章中讲述.

查看GIT对象

我们可以使用cat-file命令去查询特定对象的信息. 注意下面只键入了SHA值的一部分, 不必把40个字符全部键入:

```
$ git-cat-file -t 54196cc2
commit
$ git-cat-file commit 54196cc2
tree 92b8b694ffb1675e5975148e1121810081dbdffe
author J. Bruce Fields <bfields@puzzle.fieldses.org> 1143414668 -0500
committer J. Bruce Fields <bfields@puzzle.fieldses.org> 1143414668 -0500

initial commit
```

一个树(tree)对象可以引用一个或多个块(blob)对象, 每个块对象都对应一个文件. 更进一步, 树对象亦可以引用其他的树对象, 从而构成一个目录层次结构. 你可以使用ls-tree去查看树的内容:

```
$ git ls-tree 92b8b694
100644 blob 3b18e512dba79e4c8300dd08aeb37f8e728b8dad    file.txt
```

我们可以看到树中包含了一个文件. SHA值是文件内容的一个引用(译者注: 相当于指针指向对应的块对象).

```
$ git cat-file -t 3b18e512
blob
```

一个"块"(blob)即是文件的数据, 我们可以用cat-file查看其内容:

```
$ git cat-file blob 3b18e512
hello world
```

注意到文件中的数据是旧的. 初始树其实是第一次提交时记录的目录状态快照.

所有的对象都使用SHA1值作为索引存储在git目录之下:

```
$ find .git/objects/
.git/objects/
.git/objects/pack
.git/objects/info
.git/objects/3b
.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad
.git/objects/92
.git/objects/92/b8b694ffb1675e5975148e1121810081dbdffe
.git/objects/54
.git/objects/54/196cc2703dc165cbd373a65a4dcf22d50ae7f7
.git/objects/a0
.git/objects/a0/423896973644771497bdc03eb99d5281615b51
.git/objects/d0
.git/objects/d0/492b368b66bdabf2ac1fd8c92b39d3db916e59
```

```
.git/objects/c4
.git/objects/c4/d59f390b9cfd4318117afde11d601c1085f241
```

这些文件的内容其实是压缩的数据外加一个标注类型和长度的头. 类型可以是块(blob), 树(tree), 提交(commit)或者标签(tag).

最容易找到提交是HEAD提交, 我们可以在.git/HEAD中找到:

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

如你所见, 上面的输出告诉了我们现在在哪个分支之上工作. Git通过创建.git目录下的文件去标识分支(译注: 即refs/heads下面的文件, 多个分支会有多个文件). 每个文件中包含了一个提交的SHA1值, 我们可以用cat-file去查看此提交的内容(译注: 此提交即为该分支的头):

```
$ cat .git/refs/heads/master
c4d59f390b9cfd4318117afde11d601c1085f241
$ git cat-file -t c4d59f39
commit
$ git cat-file commit c4d59f39
tree d0492b368b66bdabf2ac1fd8c92b39d3db916e59
parent 54196cc2703dc165cbd373a65a4dcf22d50ae7f7
author J. Bruce Fields <bfields@puzzle.fieldses.org> 1143418702 -0500
committer J. Bruce Fields <bfields@puzzle.fieldses.org> 1143418702 -0500

add emphasis
```

这里的树对象指向了这棵树的新状态:

```
$ git ls-tree d0492b36
100644 blob a0423896973644771497bdc03eb99d5281615b51    file.txt
```

```
$ git cat-file blob a0423896
hello world!
```

父对象指向了前一个提交:

```
$ git-cat-file commit 54196cc2
tree 92b8b694ffb1675e5975148e1121810081dbdffe
author J. Bruce Fields <bfields@puzzle.fieldses.org> 1143414668 -0500
committer J. Bruce Fields <bfields@puzzle.fieldses.org> 1143414668 -0500
```

GIT引用

分支(branch), 远程跟踪分支(remote-tracking branch)以及标签(tag)都是对提交的引用. 所有的引用是用"refs"开头, 以斜杠分割的路径. 到目前为此, 我们用到的引用名称其实是它们的简写版本:

```
- 分支"test"是"refs/heads/test"的简写.
- 标签"v2.6.18"是"refs/tags/v2.6.18"的简写.
- "origin/master"是"refs/remotes/origin/master"的简写.
```

偶尔的情况下全名会比较有用, 例如你的标签和分支重名了, 你应该用全名去区分它们.

(新创建的引用会依据它们的名字存放在.git/refs目录中. 然而, 为了提高效率, 它们也可能被打包到一个文件中, 参见 git pack-refs).

另一个有用的技巧是, 仓库的名字可以代表该仓库的HEAD. 例如, "origin"是访问"origin"中的HEAD分支的一个捷径.

要了解Git查找引用路径的完全列表, 以及多个同名简写引用的优先级关系, 请参见git rev-parse中的"SPECIFYING REVISIONS".

显示某分支特有的提交

假设你想要查看在"master"分支可达(reachable)但其他任何分支不可达的提交.

我们可以使用git show-ref列出仓库中所有的头:

```
$ git show-ref --heads
bf62196b5e363d73353a9dcf094c59595f3153b7 refs/heads/core-tutorial
db768d5504c1bb46f63ee9d6e1772bd047e05bf9 refs/heads/maint
a07157ac624b2524a059a3414e99f6f44bebc1e7 refs/heads/master
24dbc180ea14dc1aebe09f14c8ecf32010690627 refs/heads/tutorial-2
1e87486ae06626c2f31eaa63d26fc0fd646c8af2 refs/heads/tutorial-fixes
```

我们可以使用cut和grep得到"分支-头"(branch-head)部分, 不需要"master":

```
$ git show-ref --heads | cut -d' ' -f2 | grep -v '^refs/heads/master'
refs/heads/core-tutorial
refs/heads/maint
refs/heads/tutorial-2
refs/heads/tutorial-fixes
```

然后我们就可以查看master中特有的提交:

```
$ gitk master --not $( git show-ref --heads | cut -d' ' -f2 |
                grep -v '^refs/heads/master' )
```

很明显上面的命令可以有无数种变种; 例如你想查看仓库中所有的分支可达但标签不可达的提交:

```
$ gitk $( git show-ref --heads ) --not  $( git show-ref --tags )
```

(git rev-parse提供了像"--not"之类的"选择提交"语法的解释.)

(!!update-ref!!)

GIT索引

索引(index)是一个存放了排好序的路径的二进制文件(通常是.git/index), 每一个条目都附带有一个块对象的SHA1值以及访问权限; git ls-files可以显示出索引的内容:

```
$ git ls-files --stage
100644 63c918c667fa005ff12ad89437f2fdc80926e21c 0   .gitignore
100644 5529b198e8d14decbe4ad99db3f7fb632de0439d 0   .mailmap
100644 6ff87c4664981e4397625791c8ea3bbb5f2279a3 0   COPYING
100644 a37b2152bd26be2c2289e1f57a292534a51a93c7 0   Documentation/.gitignore
100644 fbefe9a45b00a54b58d94d06eca48b03d40a50e0 0   Documentation/Makefile
...
100644 2511aef8d89ab52be5ec6a5e46236b4b6bcd07ea 0   xdiff/xtypes.h
100644 2ade97b2574a9f77e7ae4002a4e07a6a38e46d07 0   xdiff/xutils.c
100644 d5de8292e05e7c36c4b68857c1cf9855e3d2f70a 0   xdiff/xutils.h
```

请注意, 在一些旧的文档中, 索引可能被称为"当前目录缓存(current directory cache)"或者"缓存(cache)". 它有三个重要的属性:

1. 索引存储了生成一个(独一无二的)树对象所需要的所有信息.

   例如, 运行git commit会从索引中生成一个树对象, 把这个树对象存储在对象数据库(object database)中, 然后把它与这个提交关联起来. (译注: 回忆"查看Git对象"一章, 每一个提交都对应一个树对象.)

2. 索引使得对索引生成的树对象和工作树进行快速比较成为可能.

索引通过存储每个对象的一些额外信息(比如说最后修改时间)来完成这个工作. 这些数据没有在上面显示出来, 也没有存储在创建出来的树对象中, 但是它们可以用于快速找出当时工作目录中的文件与索引的差异, 从而让Git不必将文件的内容全部读出.

3. 索引可以有效地表示树对象合并时的冲突信息, 使得每一个路径名都有足够的信息与树对象联系起来, 从而可以对它们进行三路合并.

   在合并期间, 索引可能存储一个文件的多个版本(称为"stages"). 上面git ls-files的第三栏输出就是stage号. 在出现合并冲突时, 这个号码会是其他值, 而不是0.

因此索引实际上是一种暂存区域(temporary staging area), 它装载了你正在使用的树对象.

打包文件

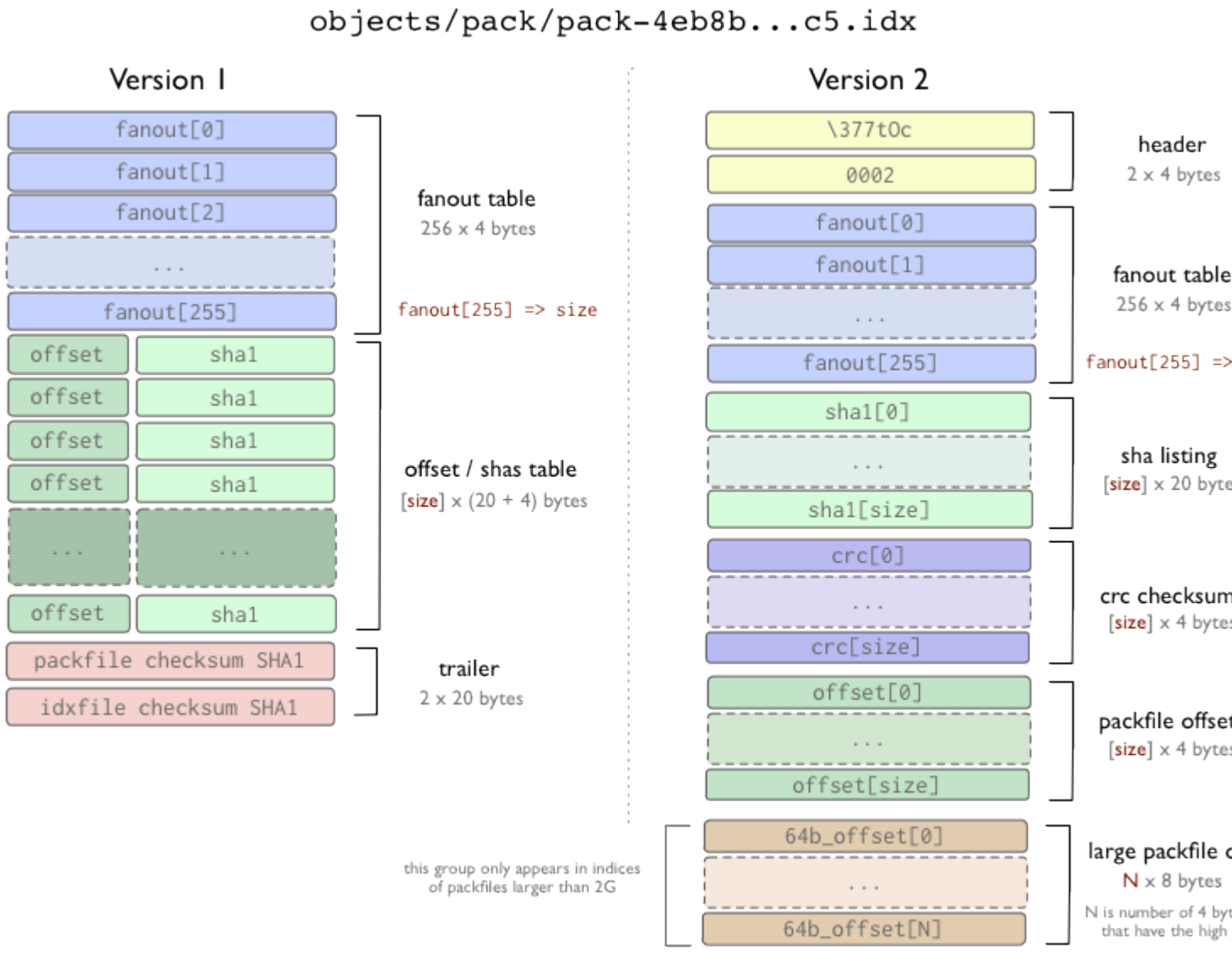
这一章将详细描述打包文件(packfile)和打包文件索引(packfile index)的格式.

打包文件索引

首先, 我们来看一下打包文件索引, 基本上它只是一系列指向打包文件内位置的书签.

打包文件索引有两个版本. 版本1的格式用于Git 1.6版本之前, 版本2的格式用于Git 1.6及以后的版本. 但是版本2可以被Git 1.5.2及以上的Git读取, 同时也被后向移植(backport)到了1.4.4.5版本.

版本2包含了每个对象的CRC校验值, 因此在重打包的过程中, 压缩过的对象可以直接进行包间拷贝(from pack to pack)而不用担心数据损坏. 版本2的打包文件索引同时亦支持大于4G的打包文件.


objects/pack/pack-4eb8b...c5.idx
Version 1
fanout[0]
fanout[1]
fanout[2]
...
fanout[255]
fanout table
256 x 4 bytes
fanout[255] => size
offset
sha1
offset / shas table
[size] x (20 + 4) bytes
packfile checksum SHA1
idxfile checksum SHA1
trailer
2 x 20 bytes
Version 2
\377tOc
0002
header
2 x 4 bytes
fanout table
256 x 4 bytes
sha1[0]
sha1[size]
sha listing
[size] x 20 byte
crc[0]
crc[size]
crc checksum
[size] x 4 bytes
offset[0]
offset[size]
packfile offset
[size] x 4 bytes
64b_offset[0]
64b_offset[N]
large packfile o
N x 8 bytes
N is number of 4 byt
that have the high
this group only appears in indices
of packfiles larger than 2G

在两个版本格式中, fanout(展开)表用于更快地查找某特定的SHA值在索引文件中的位置. offset/sha1表使用SHA1值进行排序(以便于对这个表进行二分搜索), fanout表用一种特殊的方法指向offset/sha1表(因此后一个表中包含某一特定字节开头的所有Hash的那一部分可以被轻易找到, 而不必经过二分搜索的8次迭代).

在第1版中, offset(偏移)和SHA值存在在同一位置. 但是在第2版中, SHA值, CRC值和offset被放在不同的表中. 两个版本的文件最后都是索引文件以及指向的打包文件的CRC校验值.

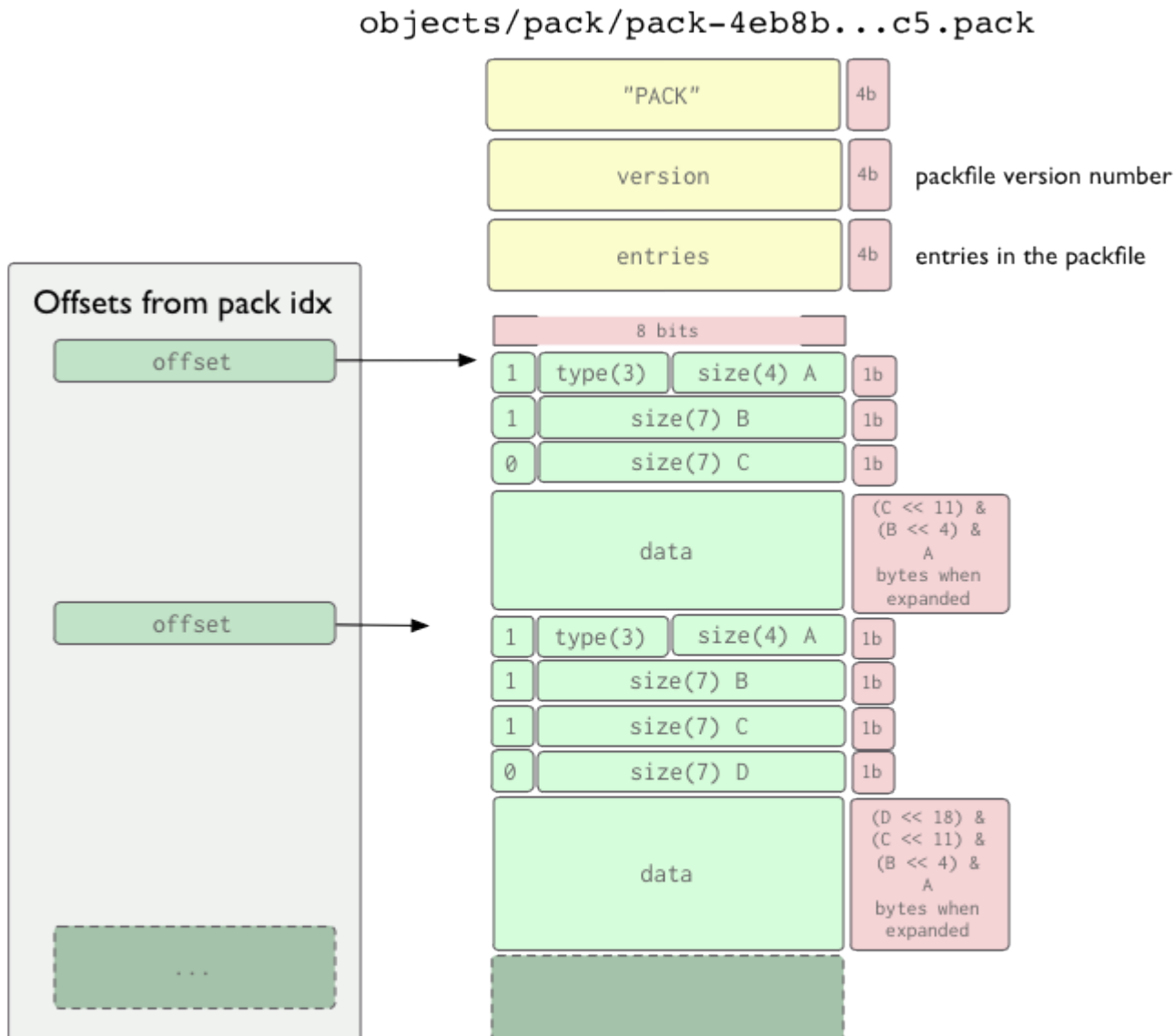
很重要的一点是, 要从打包文件中提取(extract)出一个对象, 索引文件*不是*必不可少的. 索引文件的作用是帮助用户*快速地*从打包文件中提取对象. 那些"上传打包"(upload-pack)和"取回打包"(receive-pack)程序(译注: 实现push和fetch协议的程序)使用打包文件格式(packfile format)去传输对象, 但是没有使用索引 - 索引可以在上传或者取回打包文件之后通过扫描打包文件重新建立.

打包文件格式

打包文件格式是很简单的. 它有一个头部(header)和一系列打包过的对象(每个都有自己的header和body), 还有一个校验尾部(trailer). 前4个字节是字符串'PACK', 它用于确保你找到了打包文件的起始位置. 紧接着是4个字节的打包文件版本号, 之后的4个字节指出了此文件中入口(entry)的个数. 你可以用下面Ruby程序读出打包文件的头部:

```
  def read_pack_header
    sig = @session.recv(4)
    ver = @session.recv(4).unpack("N")[0]
    entries = @session.recv(4).unpack("N")[0]
    [sig, ver, entries]
  end
```

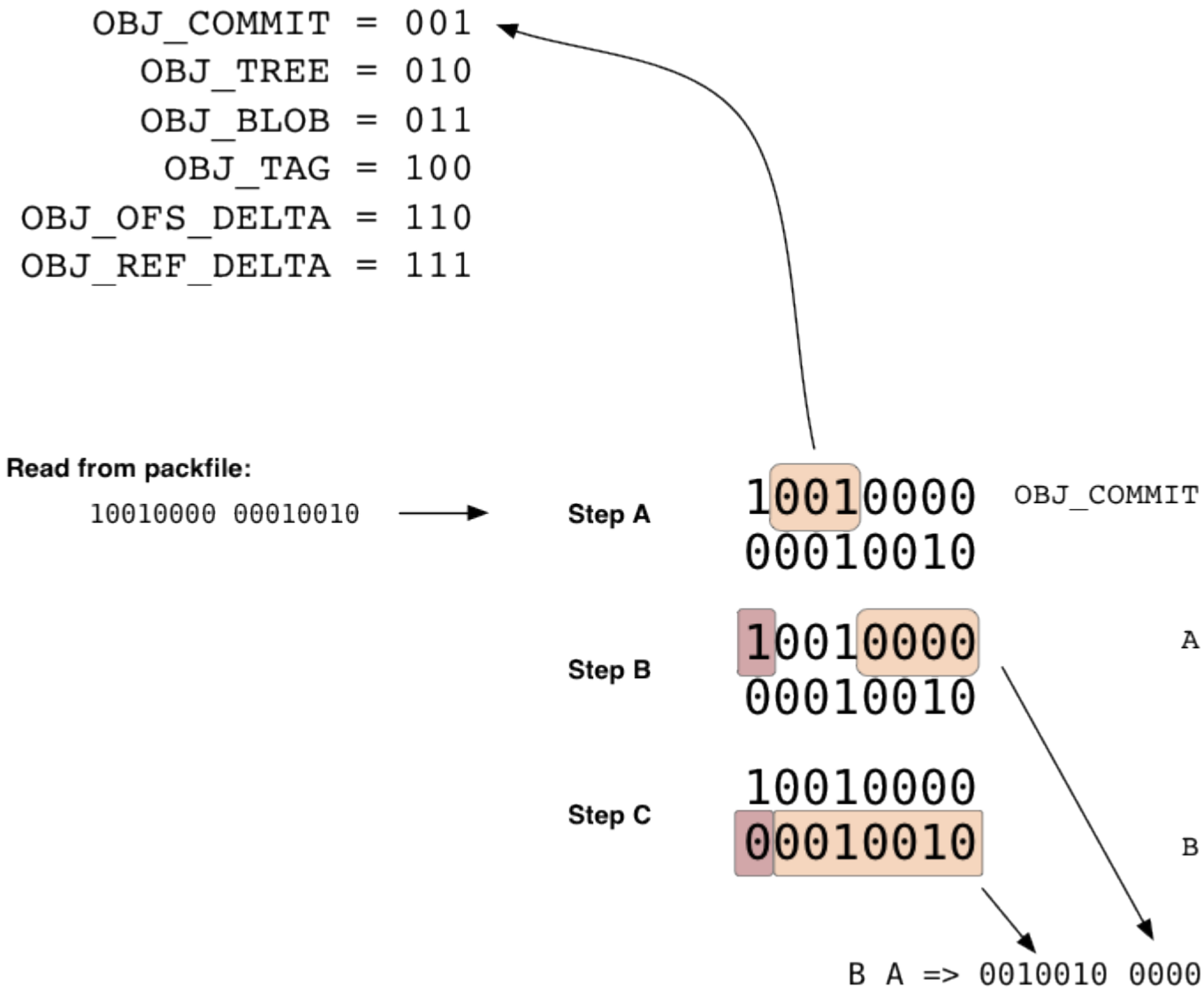
头部之后是一系列按照SHA值排序的打包对象, 每一个打包对象包含了头部和内容. 打包文件的尾部是该文件中所有(已排序)SHA值的SHA1校验值(20字节长)(译注: 即按照排序好的顺序进行迭代SHA1运算).


objects/pack/pack-4eb8b...c5.pack
"PACK"
4b
version
4b
packfile version number
entries
4b
entries in the packfile
Offsets from pack idx
offset
offset
...
8 bits
1
type(3)
size(4) A
1b
1
size(7) B
1b
0
size(7) C
1b
data
(C << 11) &
(B << 4) &
A
bytes when
expanded
1
type(3)
size(4) A
1b
1
size(7) B
1b
1
size(7) C
1b
0
size(7) D
1b
data
(D << 18) &
(C << 11) &
(B << 4) &
A
bytes when
expanded

对象头部(object header)由1个或以上的字节按序组成, 它指出了后面所跟数据的类型及展开后的尺寸. 头部的每一个字节有7位用于数据, 第1位用于说明头部是否还有后续字节. 如果第1位是'1', 你需要再读入1个字节(译注: 即下一字节仍属于头部), 否则下一字节就是数据. 第一个字节的前3位指定了数据的类型, 具体含义参见下表.

(3个位可以组合成为8个数. 在当前的使用中, 0(000)是'未定义', 5(101)目前未被使用.)

这里我们举一个由两个字节组成的头部的例子. 第1个字节的前3位说明了数据的类型是提交(commit), 余下的4位和第2个字节的7位组成的数字是144, 说明数据展开后的长度是144字节.


OBJ_COMMIT = 001
OBJ_TREE = 010
OBJ_BLOB = 011
OBJ_TAG = 100
OBJ_OFS_DELTA = 110
OBJ_REF_DELTA = 111
Read from packfile:
10010000 00010010
Step A
10010000 OBJ_COMMIT
00010010
Step B
10010000 A
00010010
Step C
10010000
00010010 B
B A => 0010010 0000

值得注意的一点是, 对象头部中包含的'尺寸'不是后面跟着的数据的长度, 而是数据*展开之后*的长度. 因此, 打包索引文件中的偏移是很有用的, 有了它你不必展开每一个对象就可以得到下一个头部的起始位置.

对于非delta对象, 数据部分就只是zlib压缩后的数据流. 对于那两种delta对象, 数据部分包含了它所依赖的基对象(base object)以及用于重构对象的delta(差异)数据. 数据的前20个字节称为`ref-delta`, 它是基对象SHA值的前20个字节. `ofs-delta`存储了基对象在同一打包文件中的偏移. 任何情况下, 有两个约束必须严格遵守:

- delta对象和基对象必须位于同一打包文件;
- delta对象和基对象的类型必须一致(即tree对tree, blob对blob, 等等).

更底层的GIT

这一章我们会学习如何在更低的层次操作Git, 以防你需要自己写一个新工具去人工生成blob(块), tree(树)或者commit(提交)对象. 如果你想使用更加底层的Git命令去写脚本, 你会需要用到以下的命令.

创建blob对象

在你的Git仓库中创建一个blob对象并且得到它的SHA值是很容易的, 使用git hash-object就足够了. 要使用一个现有的文件去创建新blob, 使用'-w'选项去运行前面提到的命令('-w'选项告诉Git要生成blob, 而不是仅仅计算SHA值).

```
$ git hash-object -w myfile.txt
6ff87c4664981e4397625791c8ea3bbb5f2279a3

$ git hash-object -w myfile2.txt
3bb0e8592a41ae3185ee32266c860714980dbed7
```

标准输出中显示的值就是创建的blob的SHA值.

创建tree对象

假设你要使用你创建的一些对象去组建一棵树, 按照git ls-tree的格式组织好输入, git mktree就可以为你生成需要的tree对象. 例如, 如果你把下面的信息写入到'/tmp/tree.txt'中:

```
100644 blob 6ff87c4664981e4397625791c8ea3bbb5f2279a3    file1
100644 blob 3bb0e8592a41ae3185ee32266c860714980dbed7    file2
```

然后通过管道把这些信息输入到git mktree中, Git会生成一个新的tree对象, 把它写入到对象数据库(object database)中, 然后返回tree对象的SHA值.

```
$ cat /tmp/tree.txt | git mk-tree
f66a66ab6a7bfe86d52a66516ace212efa00fe1f
```

然后, 我们可以把刚才生成的tree作为另外一个tree的子目录, 等等等等. 如果我们需要创建一个带子树的树对象(这个子树就是前面生成的tree对象), 只需创建一个新文件(/tmp/newtree.txt), 把前面的tree对象的SHA值写入:

```
100644 blob 6ff87c4664981e4397625791c8ea3bbb5f2279a3    file1-copy
040000 tree f66a66ab6a7bfe86d52a66516ace212efa00fe1f    our_files
```

然后再次调用git mk-tree:

```
$ cat /tmp/newtree.txt | git mk-tree
5bac6559179bd543a024d6d187692343e2d8ae83
```

现在我们有了一个人工创建的目录结构:

```
.
|-- file1-copy
`-- our_files
    |-- file1
    `-- file2

1 directory, 3 files
```

但是上面的结构并不在磁盘上存在. 另外, 我们使用SHA值去指向它(5bac6559).

重新组织树

我们也可以使用索引文件把树嵌入到新的结构中. 举个简单的例子, 我们使用一个临时索引文件创建一棵新的树, 其中包含了5bac6559这棵树的两个副本. (设置GIT_INDEX_FILE环境变量使之指向临时索引文件)

首先, 用git read-tree把树对象读入到临时索引文件中, 并给每个副本一个新的前缀; 然后再用git write-tree把索引中的内容生成一棵新的树:

```
$ export GIT_INDEX_FILE=/tmp/index
$ git read-tree --prefix=copy1/  5bac6559
$ git read-tree --prefix=copy2/  5bac6559
$ git write-tree
bb2fa6de7625322322382215d9ea78cfe76508c1

$>git ls-tree bb2fa
040000 tree 5bac6559179bd543a024d6d187692343e2d8ae83    copy1
040000 tree 5bac6559179bd543a024d6d187692343e2d8ae83    copy2
```

现在我们可以看到, 通过操纵索引文件可以得到一棵新的树. 你也可以在临时索引文件中做合并等操作 - 请参见git read-tree取得更多信息.

创建commit对象

现在我们有了一棵树的SHA值, 我们可以使用git commit-tree命令创建一个指向它的commit对象. 大部分commit对象的数据都是通过环境变量来设定的, 你需要设置下面的环境变量:

```
GIT_AUTHOR_NAME
GIT_AUTHOR_EMAIL
GIT_AUTHOR_DATE
GIT_COMMITTER_NAME
GIT_COMMITTER_EMAIL
GIT_COMMITTER_DATE
```

然后你把你的提交信息写入到一个文件中并且通过管道传送给git commit-tree, 即可得到一个commit对象.

```
$ git commit-tree bb2fa < /tmp/message
a5f85ba5875917319471dfd98dfc636c1dc65650
```

如果你需要指定一个或多个父commit对象, 只需要使用'-p'参数一个一个指定父commit对象. 同样的, 新对象的SHA值通过STDOUT返回.

更新分支的引用

现在我得拿到了新的commit对象的SHA值, 如有需要, 我们可以使用一个分支指向它. 比如说我们需要更新'master'分支的引用, 使其指向刚刚创建的新对象, 我们可以使用git update-ref去完成这个工作:

```
$ git update-ref refs/heads/master a5f85ba5875917319471dfd98dfc636c1dc65650
```

传输协议

这里我们要看一下: Git的客户端和服务器如何交互传输数据.

通过HTTP协议抓取

通过http协议的url进行的git数据抓取, 使用了一个比较傻瓜化(dumber)的协议.

使用http协议, 所有的逻辑计算(logic)都是在客户端进行. 服务器不需要特别的设置, 你只要把git目录放到一个可以访问的web目录即可.

为了能通过http访问, 当你的仓库有任何更新时, 需要运行一个命令: git update-server-info. 因为web服务器一般不允许执行列出目录中文件的操作, 所以git update-server-info命令把可用的打包文件(packfile)和引用(refs)列表更新到"objects/info/packs","info/refs"这个两个文件中. 当 git update-server-info 执行后,"objects/info/packs"文件看起来就会像下面一样:

```
P pack-ce2bd34abc3d8ebc5922dc81b2e1f30bf17c10cc.pack
P pack-7ad5f5d05f5e20025898c95296fe4b9c861246d8.pack
```

如果在通过http协议拉取数据的过程中找不到松散文件(loose file), git就会去尝试查找打包文件(packfiles). "info/refs"文件的内容看起来就下面这样:

```
184063c9b594f8968d61a686b2f6052779551613    refs/heads/development
32aae7aef7a412d62192f710f2130302997ec883    refs/heads/master
```

当你从这个仓库开始抓取(fetch)数据时, git就会从这些引用(refs)开始遍历查找所有的提交对象(commit objects), 直到客户端得到了它所有需要的所有对象为止.

例如, 你要抓取到(fetch)服务器上的"master"分支; git看到服务器上的"master"分支指向32aae7ae, 而你当前的"master"分支是指向ab04d88. 那么很明显, 你需要得到32aae7ae这个对象.

下面就是抓取时的交互过程(http协议层):

```
CONNECT http://myserver.com
GET /git/myproject.git/objects/32/aae7aef7a412d62192f710f2130302997ec883 - 200
```

然后返回信息看起来就像下面这样:

```
tree aa176fb83a47d00386be237b450fb9dfb5be251a
parent bd71cad2d597d0f1827d4a3f67bb96a646f02889
author Scott Chacon <schacon@gmail.com> 1220463037 -0700
committer Scott Chacon <schacon@gmail.com> 1220463037 -0700

added chapters on private repo setup, scm migration, raw git
```

好的那么现在它就是开始抓取树对象(tree) aa176fb8: 译者注:32aae7ae提交对象(commit object)指向的树对象(tree)是: aa176fb8.

```
GET /git/myproject.git/objects/aa/176fb83a47d00386be237b450fb9dfb5be251a - 200
```

下面这些是返回的树对象(tree)信息:

```
100644 blob 6ff87c4664981e4397625791c8ea3bbb5f2279a3    COPYING
100644 blob 97b51a6d3685b093cfb345c9e79516e5099a13fb    README
100644 blob 9d1b23b8660817e4a74006f15fae86e2a508c573    Rakefile
```

很明显, 树对象(tree)里有3个文件(blob). 好的, 我们就把它们抓下来吧:

```
GET /git/myproject.git/objects/6f/f87c4664981e4397625791c8ea3bbb5f2279a3 - 200
GET /git/myproject.git/objects/97/b51a6d3685b093cfb345c9e79516e5099a13fb - 200
GET /git/myproject.git/objects/9d/1b23b8660817e4a74006f15fae86e2a508c573 - 200
```

这些http下载操作实际上是由curl来完成的, 我们可以开多个并行的线程来加快下载速度. Git遍历完提交对象(commit)所指向的树对象(tree)后, 就会开始抓取提交对象(commit)的父对象(next parent).

```
GET /git/myproject.git/objects/bd/71cad2d597d0f1827d4a3f67bb96a646f02889 - 200
```

返回的父对象(parent commit object)信息就如下面所示:

```
tree b4cc00cf8546edd4fcf29defc3aec14de53e6cf8
parent ab04d884140f7b0cf8bbf86d6883869f16a46f65
author Scott Chacon <schacon@gmail.com> 1220421161 -0700
committer Scott Chacon <schacon@gmail.com> 1220421161 -0700

added chapters on the packfile and how git stores objects
```

我们现在可以看到ab04d88是返回的对象(commit)的父对象, 而ab04d88(commit)就是我们当前的"master"分支. 那么我们只需要得到树对象(tree):b4cc00c就可以了, 因为之前的所以的提交(commit)我们都有了. 为了保险起见, 你也可以加上'--recover'参数, 强制git反复检查我们是否拥有所有的对象. 你可以点这里: git http-fetch 查看更多信息:

如果有一个松散对象(loose object)下载失败了, git会下载打包文件索引(packfile indexes), 通过它来查找对应的sha串值，然后再下载对应的打包文件(packfile).

你一定要在git服务器的仓库里添一个"post-receive"钩子(hook), 这个钩子(hook)会在仓库更新后执行'git update-server-info; 否则仓库的相关信息就得不到更新.

通过 Upload Pack 抓取数据

对于一个聪明的协议, 抓取对象的过程(fetching objects)应当更加高效. 不管是用通过ssh协议还是git协议(git:// 协议，在9418端口上运行), 当客户端和服务器建立了一个socket连接后，客户端开始运行:git fetch-pack命令, 和服务器创建(fork)的 linkgit:git update-pack进行通讯.

服务器会告诉客户端它每个引用(ref)所有拥有的SHA串值, 而客户端会以它所需要的和所拥有SHA串值作为回应.

这里, 服务器会把客户端需要的所有对象打一个包(packfile), 然后再传送给客户端.

让我们来看一个例子.

客户端连接并且发送请求头(request header). 例如，克隆命令:

```
$ git clone git://myserver.com/project.git
```

上面的命令会产生下面的请求:

```
0032git-upload-pack /project.git\000host=myserver.com\000
```

每行的最前面的4个字节表示此行的16进行制长度(hex length) (包括这个4个字节,但不包括换行符). 下面接着的是命令和参数, 这之后是一个null字节(#body00)和主机信息. 请求的结尾是以null字节(\000)结束的.

这个请求被服务器接收并且转换成对"git-upload-pack"的命令调用.

```
$ git-upload-pack /path/to/repos/project.git
```

这条命令会马上返回仓库的信息:

```
007c74730d410fcb6603ace96f1dc55ea6196122532d HEAD\000multi_ack thin-pack side-band side-band-64k ofs-delta shall
003e7d1665144a3a975c05f1f43902ddaf084e784dbe refs/heads/debug
003d5a3f6be755bbb7deae50065988cbfa1ffa9ab68a refs/heads/dist
003e7e47fe2bd8d01d481f44d7af0531bd93d3b21c01 refs/heads/local
003f74730d410fcb6603ace96f1dc55ea6196122532d refs/heads/master
0000
```

每一行开始的头4个字节表示此行的长度(以16进制表示). 这块(section)信息以一行“0000”为结束标识符.

上面这些服务器产生的数据被发送回客户端. 然后客户端用另外一个请求做为响应:

```
0054want 74730d410fcb6603ace96f1dc55ea6196122532d multi_ack side-band-64k ofs-delta
```

p 0032want 7d1665144a3a975c05f1f43902ddaf084e784dbe

```
0032want 5a3f6be755bbb7deae50065988cbfa1ffa9ab68a
0032want 7e47fe2bd8d01d481f44d7af0531bd93d3b21c01
0032want 74730d410fcb6603ace96f1dc55ea6196122532d
00000009done
```

上面这些客户端的请求会被发送到的"git-upload-pack"进程, 这个进程会返回(streams out)最终的结果(final response):

```
"0008NAK\n"
"0023\002Counting objects: 2797, done.\n"
"002b\002Compressing objects:   0% (1/1177)   \r"
"002c\002Compressing objects:   1% (12/1177)   \r"
"002c\002Compressing objects:   2% (24/1177)   \r"
"002c\002Compressing objects:   3% (36/1177)   \r"
"002c\002Compressing objects:   4% (48/1177)   \r"
"002c\002Compressing objects:   5% (59/1177)   \r"
"002c\002Compressing objects:   6% (71/1177)   \r"
"0053\002Compressing objects:   7% (83/1177)   \rCompressing objects:   8% (95/1177)   \r"
```

```
...
"005b\002Compressing objects: 100% (1177/1177)   \rCompressing objects: 100% (1177/1177), done.\n"
"2004\001PACK\000\000\000\002\000\000\n\355\225\017x\234\235\216K\n\302"...
"2005\001\360\204{\225\376\330\345]z2673"...
...
"0037\002Total 2797 (delta 1799), reused 2360 (delta 1529)\n"
...
"<\276\255L\273s\005\001w0006\001[0000"
```

你可以查看"打包文件"(packfile)这一章, 了解响应内容中的打包文件(packfile)的格式.

推送数据

通过git和ssh协议推送数据(pushing data)是相似的, 但是更简单. 基本上是, 客户端发出一个"receive-pack"的请求, 如果客户端有访问权限, 那么服务器就返回所有引用"头"的SHA串值(all ref head shas). 客户端收到响应后, 计算出服务器需要的所有数据或对象, 再做成一个打包文件(packfile)传送给服务器. 服务器收到后要么就把它们存储到硬盘上再建立索引, 要么只把它解压(如果里面的对象不多的话).

在这整个推送数据的过程中, 客户端通过 git push 命令调用:git sendpack命令, 服务器端通过"ssh连接进程"或是"git服务器"来调用:linkgit:git-receive-pack 命令来完成整个操作.

术语表

我们把在Git里常用的一些名词做了解释列在这里。这些名词(terms)全部来自Git Glossary。

alternate object database

> Via the alternates mechanism, a repository

```
can inherit part of its object database
from another object database, which is called "alternate".
```

bare repository

A bare repository is normally an appropriately

```
named directory with a `.git` suffix that does not
have a locally checked-out copy of any of the files under
revision control. That is, all of the `git`
administrative and control files that would normally be present in the
hidden `.git` sub-directory are directly present in the
`repository.git` directory instead,
and no other files are present and checked out. Usually publishers of
public repositories make bare repositories available.
```

裸仓库

```
A bare repository is normally an appropriately
named directory with a `.git` suffix that does not
have a locally checked-out copy of any of the files under
revision control. That is, all of the `git`
administrative and control files that would normally be present in the
hidden `.git` sub-directory are directly present in the
`repository.git` directory instead,
and no other files are present and checked out. Usually publishers of
public repositories make bare repositories available.
```

blob object（二进制对象）

没有类型的数据对象。例如：一个文件的内容。

branch

A "branch" is an active line of development. The most recent

```
commit on a branch is referred to as the tip of
that branch.  The tip of the branch is referenced by a branch
head, which moves forward as additional development
is done on the branch.  A single git
repository can track an arbitrary number of
branches, but your working tree is
associated with just one of them (the "current" or "checked out"
branch), and HEAD points to that branch.
```

分支

一个“分支”是开发过程中的(active line)。。。。

cache（缓存）

索引(index)的旧称(obsolete).

chain（链表）

一串对象，其中每个对象都有指向其后继对象的引用(reference to its successor)。例如：一个提交(commit)的后继对象就是它的父对象。

changeset（修改集）

BitKeeper/cvsps 里对于提交(commit)的说法。但是 git 只存储快照(states)，不存储修改；所以这个词用在 git 里有点不大合适。

checkout（签出）

用对象仓库(object database)里的一个树对象(tree object)更新当前整个工作树(worktree)，或者一个二进制对象(blob object)更

cherry-picking

In SCM jargon, "cherry pick" means to choose a subset of

changes out of a series of changes (typically commits) and record them as a new series of changes on top of a different codebase. In GIT, this is performed by the "git cherry-pick" command to extract the change introduced by an existing commit and to record it based on the tip of the current branch as a new commit.

cherry-picking

在SCM的行话里，“cherry pick“ 意味着从一系列的修改中选出一部分修改(通常是提交)，应用到当前代码中。()

clean（干净）

如果一个工作树(working tree)中所有的修改都已提交到了当前分支里(current head)，那么就说它是干净的(clean)，反之它就是脏的(dirty)。

commit

As a verb: The action of storing a new snapshot of the project's

```
state in the git history, by creating a new commit representing the current
state of the index and advancing HEAD
to point at the new commit.
```

commit（提交）

作为名词：指向git历史的某一点的指针；整个项目的历史就由一组相互关联的提交组成的。提交(commit)在其它版本控制系统中也做"revision"或"version"。同时做为提交对象(commit object)的缩写。

作为动词：创建一新的提交(commit)来表示当前索引(index)的状态的行为，把 HEAD 指向新创建的提交，这一系列把项目在某一时间上的快照(snapshot)保存在git历史中的操作。

提交对象

一个关于特定版本信息(particular revision)的对象。包括父对象名，提交者，作者，日期和存储了此版本内容的树对象名(tree object)。

core git

Git的基本数据结构和工具，它只对外提供简单的代码管理工具。

DAG

有向无环图。众多提交对象(commit objects)组成了一个有向无环图；因为它们都有直接父对象(direct parent)，且没有一条提交线路(chain)的起点和终点都是同一个对象。

dangling object（悬空对象）

一个甚至从其它不可达对象也不可达的对象(unreachable object)；仓库里的一个悬空对象没有任何引用(reference)或是对象(object)引用它。

detached HEAD（分离的HEAD）

通常情况下HEAD里是存放当前分支的名字。然而 git 有时也允许你签出任意的一个提交(commit)，而不一定是某分支的最近的提交(the tip of any particular branch)；在这种情况下，HEAD就是处于分离的状态(detached)。 译者注：这时`.git/HEAD`中存储的就是签出的提交的SHA串值。

dircache

请参见索引(index)。

directory（目录）

执行"ls"命令所显示的结果 :-)

dirty（脏）

一个工作树里有没有提交到当前分支里修改，那么我就说它是脏的(dirty)。

ent

某些人给树名(tree-ish)起的另外一个别名，这里http://en.wikipedia.org/wiki/Ent_(Middle-earth)有更详细的解释。最好不要使用这个名词，以免让大家糊涂。

evil merge（坏的合并）

如果一次合并引入一些不存在于任何父对象(parent)中的修改，那么就称它是一个坏的合并(evil merge)。

fast forward

A fast-forward is a special type of merge where you have a

```
revision and you are "merging" another
branch's changes that happen to be a descendant of what
you have. In such these cases, you do not make a new merge
commit but instead just update to his
revision. This will happen frequently on a
tracking branch of a remote
repository.
```

快速向前

“fast-forward”是一种特殊的合并,()。 在这种情况下，并没有创建一个合并提交(merge commit)，只是更新了版本信息。 当本地分支是远端仓库(remote repository)的跟踪分支时，这种情况经常出现。

fetch（抓取）

抓取一个分支意味着：得到远端仓库(remote repository)分支的head ref，找出本地对象数据库所缺少的对象，并把它们下载下来。你可以参考一下 git fetch。

file system（文件系统）

Linus Torvalds 最初设计 git 时，是把它设计成一个在用户空间(user space)运行的文件系统；也就是一个用来保存文件和目录的 infrastructure，这样就保证了git的速度和效率。

git archive

对玩架构的人来说，这就是仓库的同义词。

grafts

Grafts enables two otherwise different lines of development to be joined

```
together by recording fake ancestry information for commits. This way
you can make git pretend the set of parents a commit has
is different from what was recorded when the commit was
created. Configured via the `.git/info/grafts` file.
```

hash（哈希）

在git里，这就是对象名(object name)的同义词。

head

指向一个分支最新提交的命名引用(named reference)。除非使用了打包引用(packed refs)，heads 一般存储在 `$GIT_DIR/refs/heads/`。 参见: git pack-refs

HEAD

当前分支。详细的讲是：你的工作树(working tree)通是从HEAD所指向的tree所派生的来的。 HEAD 必须是指向一个你仓库里的head，除非你使用分离的HEAD(detached HEAD)。

head ref

head的同义词。

hook

During the normal execution of several git commands, call-outs are made

```
to optional scripts that allow a developer to add functionality or
checking. Typically, the hooks allow for a command to be pre-verified
and potentially aborted, and allow for a post-notification after the
operation is done. The hook scripts are found in the
`$GIT_DIR/hooks/` directory, and are enabled by simply
removing the `.sample` suffix from the filename. In earlier versions
of git you had to make them executable.
```

钩子

在一些git命令的执行过程中, () 允许开发人员调用特别的脚本来添加功能或检查。

```
()
```

Typically，钩子允许对一个命令做pre-verified并且可以中止此命令的运行；同时也可在这个命令执行完后做后继的通知工作。这些钩子脚本放在`

index

A collection of files with stat information, whose contents are stored

```
as objects. The index is a stored version of your
working tree. Truth be told, it can also contain a second, and even
a third version of a working tree, which are used
when merging.
```

索引

描述项目状态信息的文件，。索引里保存的是你的工作树的版本记录。()

index entry

The information regarding a particular file, stored in the

```
index. An index entry can be unmerged, if a
merge was started, but not yet finished (i.e. if
the index contains multiple versions of that file).
```

索引条目

主分支 (master)

默认的开发分支。当你创建了一个git仓库，一个叫"master"的分支就被创建并且成为当前活动分支(active branch)。在多数情况下，这个分支里就包含有本地的开发内容。

merge

As a verb: To bring the contents of another

```
branch (possibly from an external
repository) into the current branch.  In the
case where the merged-in branch is from a different repository,
this is done by first fetching the remote branch
and then merging the result into the current branch.  This
combination of fetch and merge operations is called a
pull.  Merging is performed by an automatic process
that identifies changes made since the branches diverged, and
then applies all those changes together.  In cases where changes
```

```
conflict, manual intervention may be required to complete the
merge.
```

merge（合并）

作为动词：把另外一个分支(也许来自另外一个仓库)的内容合并进当前的分支。()

作为名词：除非合并的结果是 fast forward；那么一次成功的合并会创建一个新的提交(commit)来表示这次合并，并且把合并了的分支做为此提交(commit)的父对象。这个提交(commit)也可以表述为“合并提交”(merge commit)，或者就是"合并"(merge 名词)。

object（对象）

Git的存储单位，它以对象内容的SHA1值做为唯一对象名；因此对象内容是不能被修改的。

object database（对象仓库）

用来存储一组对象(objects)，每个对象通过对象名来区别。对象(objects)通常保存在 `$GIT_DIR/objects/`。

object identifier（对象标识符）

对象名(object name)的同义词。

object name（对象名）

一个对象的唯一标识符(unique identifier)。它是使用SHA1算法(Secure Hash Algorithm 1)给对象内容进行哈希(hash)计算，产生的一个40个字节长的16进制编码的串。

object type（对象类型）

Git有4种对象类型：提交(commit)，树(tree)，标签(tag)和二进制块(blob)。

octopus（章鱼）

一次多于两个分支的合并(merge)。也用来表示聪明的肉食动物。

origin

默认的上游仓库(upstream repository)。每个项目至少有一个它追踪(track)的上游(upstream)仓库，通常情况 origin 就是用来表示它。你可以用 " `git branch -r` " 命令查看上游仓库(upstream repository)里所有的分支，再用 origin/name-of-upstream-branch 的名字来抓取(fetch)远程追踪分支里的内容。

pack（包）

一个文件，里面有一些压缩了的对象。(用以节约空间或是提高传输效率)。

pack index（包索引）

包(pack)里的一些标识符和其它相关信息，用于帮助git快速的访问包(pack)里面的对象。

parent

A commit object contains a (possibly empty) list

```
of the logical predecessor(s) in the line of development, i.e. its
parents.
```

父对象

一个提交对象(commit object)，()。

pickaxe

The term pickaxe refers to an option to the diffcore

```
routines that help select changes that add or delete a given text
string. With the `--pickaxe-all` option, it can be used to view the full
changeset that introduced or removed, say, a
particular line of text. See git diff.
```

plumbing

core git的别名(cute name)。

porcelain

Cute name for programs and program suites depending on

```
core git, presenting a high level access to
core git. Porcelains expose more of a SCM
interface than the plumbing.
```

pull（拉）

拉(pull)一个分支意味着，把它抓取(fetch)下来并合并(merge)进当前的分支。可以参考 git pull.

push

Pushing a branch means to get the branch's

```
head ref from a remote repository,
find out if it is a direct ancestor to the branch's local
head ref, and in that case, putting all
objects, which are reachable from the local
head ref, and which are missing from the remote
repository, into the remote
object database, and updating the remote
head ref. If the remote head is not an
ancestor to the local head, the push fails.
```

推

()

reachable

All of the ancestors of a given commit are said to be

```
"reachable" from that commit. More
generally, one object is reachable from
another if we can reach the one from the other by a chain
that follows tags to whatever they tag,
commits to their parents or trees, and
trees to the trees or blobs
that they contain.
```

可达的

rebase

重新应用(reapply)当前点(branch)和另一个点(base)间的修改；并且根据rebase的结果重置当前分支的head。 译者注：这个功能可以修改历史提交。

ref（引用）

一个40字节长的SHA1串或是表示某个对象的名字。它们可能存储在 `$GIT_DIR/refs/`。

reflog

reflog用以表示本地的ref的历史记录。从另外一角度也可以说，它能行告诉你这个仓库最近的第3个版本(revision)是什么，还可以告诉你昨天晚上9点14分时你是在这个仓库的哪个分支下工作。可以参见:git reflog。

refspec

"refspec"用于描述在抓取和推的过程中，远程ref和本地ref之间的映射关系。它用冒号连接：:，前面也可以加一个加号：“+“。 例如：`git fetch $URL refs/heads/master:refs/heads/origin` 意味着：从$URL抓取主分支的 head 并把它保存到本地的origin分支的head中。`git push $URL refs/heads/master:refs/heads/to-upstream` 意味着：把我本地主分支 head 推到$URL上的 to-upstream分支里。具体可以参见：git push。

repository

A collection of refs together with an

```
object database containing all objects
which are reachable from the refs, possibly
accompanied by meta data from one or more porcelains. A
```

```
repository can share an object database with other repositories
via alternates mechanism.
```

resolve

在自动合并失败后，手工修复合并冲突的行为。

revision（版本）

对象仓库(object database)保存的文件和目录在某一特定时间点的状态；它会被一个提交对象(commit object)所引用。

rewind

丢弃某一部分开发成果。例如：把head 指向早期的版本。

SCM

源代码管理工作。

SHA1

对象名(object name)的同义词。

shallow repository

A shallow repository has an incomplete

```
history some of whose commits have parents cauterized away (in other
words, git is told to pretend that these commits do not have the
parents, even though they are recorded in the commit
object). This is sometimes useful when you are interested only in the
recent history of a project even though the real history recorded in the
upstream is much larger. A shallow repository
is created by giving the `--depth` option to git clone, and
its history can be later deepened with git fetch.
```

symref

Symbolic reference: instead of containing the SHA1

```
id itself, it is of the format 'ref: refs/some/thing' and when
referenced, it recursively dereferences to this reference.
'HEAD' is a prime example of a symref. Symbolic
references are manipulated with the git symbolic-ref
command.
```

tag（标签）

一个ref指向一个标签或提交对象。与 head 相反，标签并不会在一次提交操作后改变。标签(不是标签对象)存储在`$GIT_DIR/refs/tags/`。 一个标签通常是用来标识提交家族链(commit ancerstry chain)里的某一点。

tag object（标签对象）

一个含有指向其它对象的引用(ref)的对象，对象里包括注释消息。如果它里面可以含有一个PGP签名，那么就称为一个“签名标签对象”(signed tag object)。

topic branch

A regular git branch that is used by a developer to

```
identify a conceptual line of development. Since branches are very easy
and inexpensive, it is often desirable to have several small branches
that each contain very well defined concepts or small incremental yet
related changes.
```

tracking branch

A regular git branch that is used to follow changes from

```
another repository. A tracking
branch should not contain direct modifications or have local commits
made to it. A tracking branch can usually be
identified as the right-hand-side ref in a Pull:
```

```
refspec.
```

追踪分支

一个用以追踪(follow)另外一个仓库的修改的git分支。()

tree（树）

可以是一个工作树(working tree)，也可以是一个树对象(tree object)。

tree object（树对象）

包含有一串(list)文件名和模式(mode)，并且指向与之相关的二进制对象(blob object)和树对象(tree object)。一个树(tree)等价于一个目录。

tree-ish（树名）

一个指向的提交对象(commit object)，树对象(tree object)或是标签对象(tag object)的引用(ref)。

unmerged index（未合并索引）

一个索引中包含有未合并的索引条目(index entries)。

unreachable object（不可达对象）

从任何一个分支、标签或是其它引用(reference)做为起点都无法到达的一个对象。

working tree（工作树）

签出(checkout)用于编辑的文件目录树。 工作树一般等价于 HEAD 加本地没有提交的修改。