O Livro da Comunidade Git

Um recurso Git aberto combinados juntos pela comunidade inteira.

AUTORES

Agradeça a esses caras:

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)

MANTENEDOR / EDITOR

Bug para esse cara:

Scott Chacon (schacon@gmail.com)

Capítulo 1

Introdução

BEM VINDO AO GIT

Bem vindo ao Git - o mais rápido , sistema de controle de versão distribuída.

Esse livro é um ponto de partida para as pessoas novas no Git que querem aprendê-lo tão rapidamente e facilmente quanto possível.

Esse livro iniciará introduzindo você sobre a forma como o Git armazena os dados, dando a você o contexto do porque ele é diferente do que as outras ferramentas de SCV. Essa parte será abordada em 20 minutos.

Depois cobriremos **O Uso básico do Git** - os comandos que usará em 90% do do seu tempo. Isso lhe dará uma boa base para usar o Git confortavelmente para a maioria das coisas onde poderá usá-lo. Essa seção levará 30 minutos para leitura.

Depois iremos para **Uso Intermediário do Git** - coisas que são ligeiramente mais complexas, mas podem substituir alguns dos comandos básicos que você aprendeu na primeira seção. Este mostrará principalmente truques e comandos onde se sentirá mais confortável depois que conhece os comandos básicos.

Depois que você estiver dominado, nós abordaremos **Git Avançado** - comandos que a maioria das pessoas não usam frequentemente, mas podem ser muito úteis em certas situações. Aprendendo esses comandos deverá cobrir o seu conhecimento sobre o git no dia a dia; você se tornará um mestre do Git!

Agora que você conhece o Git, nós iremos abordar **Trabalhando com Git**. Aqui nós iremos falar como usar o Git em scripts, com ferramentas de implantação, com editores e muito mais. Estas seções são principalmente para ajudar você a integrar o Git ao seu ambiente.

Finalmente, nós teremos uma série de artigos sobre **documentação de baixo nível** que pode ajudar hackers de Git que querem aprender como funcionam os protocolos e funções internas no Git.

Comentários e Contribuições

Até aqui, se você achou algum erro ou quer contribuir com o livro, você pode enviar um email para schacon@gmail.com, ou pode clonar o código-fonte deste livro em http://github.com/schacon/gitbook e me enviar um patch ou um pull-request.

Comentários e Contribuições sobre a versão Brasileira

Caso queira contribuir com a versão brasileira, você pode começar fazendo um fork do repositório no Github em http://github.com/schacon/gitbook branch pt_BR e enviar um pull-request quando fizer alguma alteração.

Créditos e Colaboradores na versão Brasileira

Aqui a lista de colaboradores que ajudaram/ajudam no desenvolvimento desta versão.

- Djalma Oliveira djalmaoliveira@gmail.com no Github http://github.com/djalmaoliveira/gitbook.
- Enderson Maia endersonmaia@gmail.com no Github http://github.com/enderson/gitbook.

Referências

Muitos dos conteúdos desse livro foram extraídos de diferentes fontes e então adicionados. Se você gostaria de ler sobre alguns dos artigos originais ou recursos, por favor visite eles e agradeça os autores.

- Git Manual do Usuário
- The Git Tutorial
- The Git Tutorial pt 2
- "My Git Workflow" blog post

Capítulo 2

O básico sobre os objetos no git

O MODELO DE OBJETOS DO GIT

O SHA

Todas as informações necessárias para representar a história do projeto são armazenados em arquivos referenciados por um "nome do objeto" de 40 dígitos que se parece com isso:

```
6ff87c4664981e4397625791c8ea3bbb5f2279a3
```

Você verá esses 40 dígitos em todo lugar no Git. Em cada caso o nome é calculado baseado no valor hash SHA1 do conteúdo do objeto. O hash SHA1 é uma função criptográfica. O que isso significa para nós é que ele é virtualmente impossível de encontrar dois objetos diferentes com o mesmo nome. Isso tem inúmeras vantagens; entre outras:

- Git pode rapidamente determinar se dois objetos são idênticos ou não, somente comparando os seus nomes.
- Visto que os nomes dos objetos são calculados da mesma forma em todo o repositório, o mesmo conteúdo armazenado em dois repositórios sempre será armazenado sobre o mesmo nome.
- Git pode detectar erros quando lê um objeto, através da checagem do nome do objeto que ainda é o hash SHA1 do seu conteúdo.

Os Objetos

Todo objeto consiste de 3 coisas - um **tipo**, um **tamanho** e **conteúdo**. O *tamanho* é simplesmente o tamanho do conteúdo, o conteúdo depende do tipo que o objeto é, e existem quatro tipos diferentes de objetos: "blob", "tree", "commit", and "tag".

- Um **"blob"** é usado para armazenar dados do arquivo - é geralmente um arquivo.
- Um **"tree"** é basicamente como um diretório - ele referencia um conjunto de outras trees e/ou blobs (ex.: arquivos e sub-diretórios)
- Um **"commit"** aponta para uma simples tree, fazendo com que o projeto se parecesse em um determinado ponto no tempo. Ele contém meta informações sobre aquele ponto no tempo, por exemplo um timestamp, o autor das modificações desde o último commit, um ponteiro para um commit anterior, etc.
- Uma **"tag"** é uma forma de marcar um commit específico de alguma forma como especial. Ele é normalmente usado para identificar certos commits como versões/revisões específicas ou alguma coisa junto a aquelas linhas.

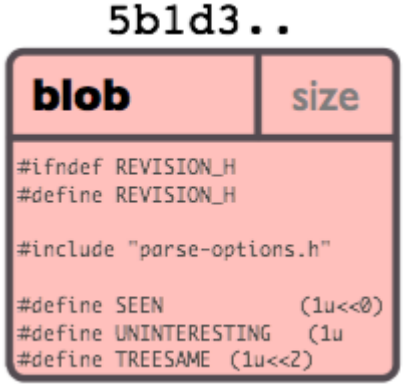
Quase tudo do Git é construído através da manipulação dessa simples estrutura de quatro diferentes tipos de objetos. Eles são organizados dentro se seu próprio sistema de arquivos que estão sobre o sistema de arquivos de sua máquina.

Diferenças do SVN

É importante notar que isso é muito diferente da maioria dos sistemas SCM com que você pode estar familiarizado. Subversion, CVS, Perforce, Mercurial e como todos eles usam sistemas *Delta Storage* - como eles armazenam as diferenças entre um commit e o próximo. Git não faz isso - ele armazena um snapshot de todos os arquivos de seu projeto como eram nessa estrutura em árvore no momento de cada commit, é claro que isso não significará (necessariamente) que seu histórico do repositório atingirá um tamanho enorme. Usando "git gc", o Git fará alguma limpeza interna, incluindo compressão de arquivos de revisão. Esse é um conceito muito importante para entender quando estiver usando Git.

Objeto Blob

Um blob geralmente armazena o conteúdo de um arquivo.



Você pode usar git show para examinar o conteúdo de qualquer blob. Supondo que temos um SHA para um blob, podemos examinar seu conteúdo assim:

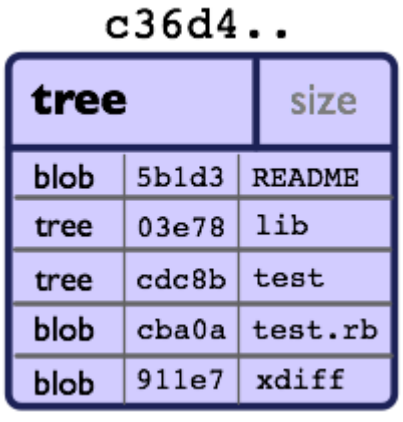
```
$ git show 6ff87c4664

 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.
...
```

Um objeto "blob" não é nada mais que um grande pedaço de dados binários. Ele não se referencia a nada ou possui atributos de qualquer tipo, nem mesmo um nome de arquivo.

Visto que o blob é inteiramente definido por esses dados, se dois arquivos em uma árvore de diretório (ou dentro de múltiplas versões diferentes desse repositório) possui o mesmo conteúdo, eles irão compartilhar o mesmo objeto blob. O objeto é totalmente independente da localização do arquivo na árvore de diretório, e renomeando esse arquivo não muda o objeto com o qual está
associado.

Objeto Tree

Um tree é um objeto simples que possui um conjunto de ponteiros para blobs e outras trees - ele geralmente representa o conteúdo de um diretório ou sub diretório.



O sempre versátil comando git show pode também ser usado para examinar objetos tree, mas git ls-tree dará a você mais detalhes. Supondo que temos um SHA para uma tree, podemos examinar ela assim:

```
$ 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
...
```

Como você pode ver, um objeto tree contém uma lista de entradas, cada uma com um modo de acesso, tipo, nome SHA1, e nome de arquivo, ordenado pelo nome de arquivo. Ele representa o conteúdo de uma simples árvore de diretório.

Um objeto referenciado por uma tree pode ser um blob, representando o conteúdo de um arquivo, ou outra tree, representando o conteúdo de um sub diretório. Visto que trees e blobs, como todos os outros objetos, são nomeados por um hash SHA1 de seus conteúdos, duas trees possui o mesmo hash SHA1 se somente se seus conteúdos (incluindo, recursivamente, o conteúdo de todos os sub-diretórios) são idênticos. Isso permite ao git determinar rapidamente as diferenças entre dois objetos tree relacionados, desde que ele possa ignorar qualquer entrada com nome de objetos idênticos.

(Nota: na presença de sub módulos, trees pode também ter commits como entradas. veja a seção **Sub Módulos**.)

Perceba que todos os arquivos possuem o modo de acesso 644 ou 755: o git na verdade somente dá atenção para o bit executável.

Objeto Commit

O objeto "commit" liga o estado físico de uma árvore com a descrição de como a conseguimos e porque.

Você pode usar a opção --pretty=raw para git show ou git log para examinar seu commit favorito:

```
$ 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>
```

Como você pode ver, um commit é definido por:

- uma **tree**: O nome SHA1 do objeto tree (como definido acima), representando o conteúdo de um diretório em um certo ponto no tempo.
- **parent(s)**: O nome SHA1 de algum número de commits que representa imediatamente o(s) passo(s) anterior(es) na história do projeto. O exemplo acima possui um parent; commits gerados por um merge podem ter mais do que um. Um commit sem nenhum parent é chamado de commit "root", e representa a versão/revisão inicial do projeto. Cada projeto deve possuir pelo menos um root. Um projeto pode também ter múltiplos roots, mesmo assim não é comum (ou necessariamente uma boa idéia).
- um **author**: O nome da pessoa responsável pela alteração, junto com uma data.
- um **committer**: O nome da pessoa que de fato criou o commit, com a data que foi feita. Ele pode ser diferente do autor, por exemplo, se o autor escreveu um patch e enviou-o para outra pessoa que usou o patch para criar o commit.
- um **comment** descrevendo esse commit.

Note que um commit não contém qualquer informação sobre o que foi alterado; todas as alterações são calculadas pela comparação dos conteúdos da tree referenciada por esse commit com as trees associadas com o seu parent. De forma particular, o git não dá atenção para arquivos renomeados explicitamente, embora possa identificar casos onde a existência do mesmo conteúdo do arquivo na alteração sugira a renomeação. (Veja, por exemplo, a opção -M para git diff).

Um commit é normalmente criado por git commit, que cria um commit no qual o parent é normalmente o HEAD atual, e do qual a tree é levada do conteúdo atualmente armazenado no index.

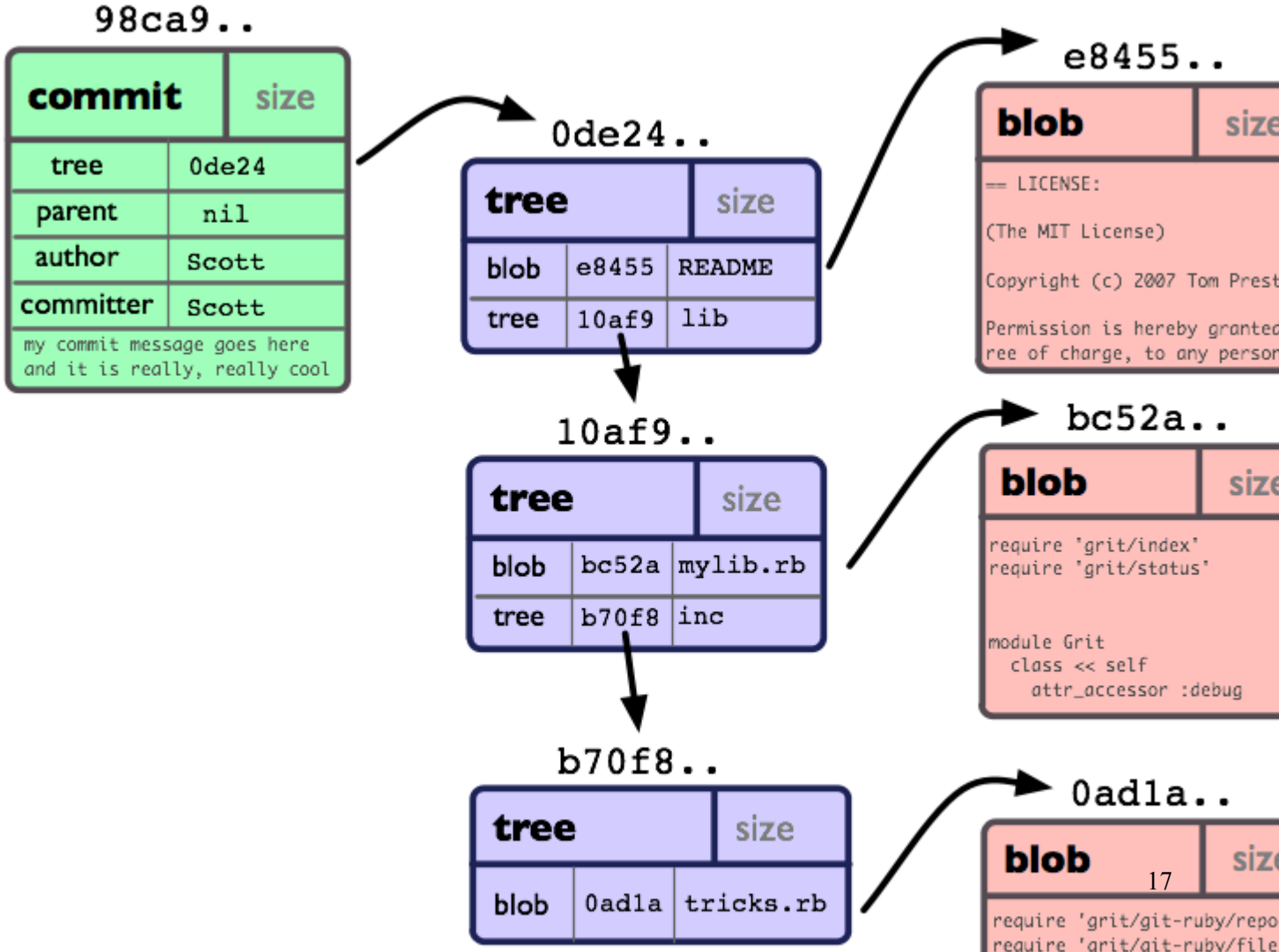
O Modelo de Objeto

Então, agora o que vimos os 3 principais tipos de objetos (blob, tree e commit), vamos dar uma rápida olhada em como eles todos se relacionam juntos.

Se tivéssemos um simples projeto com a seguinte estrutura de diretório:

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

2 diretórios, 3 arquivos
```

E comitamos ele para um repositório Git, seria representado assim:


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
require 'grit/git-ruby/repo
require 'grit/git-ruby/file

Você pode ver que temos criado um objeto **tree** para cada diretório (incluindo o root) e um objeto **blob** para cada arquivo. Então temos um objeto **commit** apontando para o root, então podemos rastreá-lo até o momento em que o nosso projeto se parecia quando foi commitado.

Objeto Tag

Um objeto tag contém um nome de objeto (chamado simplesmente de 'object'), tipo de objeto, nome da tag, o nome da pessoa ('tagger') que criou a tag, e uma mensagem, que pode conter um assinatura, como pode ser visto usando git cat-file:

```
$ 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-----
```

Veja o comando git tag para aprender como criar e verificar objetos tag. (Note que git tag pode ser também usado para criar "tags peso-leve", que não são objetos tag, mas só simples referências dos quais os nomes iniciam com "refs/tags/").

DIRETÓRIO GIT E DIRETÓRIO DE TRABALHO

O Diretório Git

O 'diretório git' é o diretório que armazena todos os históricos Git e meta informações do seu projeto - incluindo todos os objetos (commits, trees, blobs, tags), todos os ponteiros onde os diferentes branches estão e muito mais.

Existe somente um Diretório Git por projeto (o oposto de um por sub diretório como no SVN ou CVS), e que o diretório é (por padrão, embora não necessariamente) '.git' na raiz do seu projeto. Se você olha no conteúdo desse diretório, você pode ver todos os seus importantes arquivos:

```
$>tree -L 1
.
```

```
|-- HEAD         # aponta para o seu branch atual
|-- config       # suas configurações preferenciais
|-- description  # descrição do seu projeto
|-- hooks/       # pre/post action hooks
|-- index        # arquivo de index (veja a próxima seção)
|-- logs/        # um histórico de onde seus branches tem estado
|-- objects/     # seus objetos (commits, trees, blobs, tags)
`-- refs/        # ponteiros para os seus branches
```

(podem existir alguns outros arquivos/diretórios aqui mas eles não são importantes agora)

O Diretório de Trabalho

O 'diretório de trabalho' do Git é o diretório que detém o checkout atual dos arquivos sobre o qual você está trabalhando. Arquivos nesse diretório são frequentemente removidos ou renomeados pelo Git quando você troca de branches - isso é normal. Todos os seus históricos são armazenados no diretório Git; o diretório de trabalho é simplesmente um lugar temporário de checkout onde você pode modificar os arquivos até o próximo commit.

O INDEX DO GIT

O index do Git é usado como uma área de preparação entre o seu diretório de trabalho e o seu repositório. Você pode usar o index para construir um conjunto de modificações no qual você quer levar juntos para o próximo commit. Quando você cria um commit, o que é levado para o commit é o que está no index atualmente, não o que está no seu diretório de trabalho.

Visualizando o Index

A forma mais fácil para ver o que está no index é com o comando git status. Quando você roda o git status, você pode ver quais arquivos estão selecionados para o próximo commit (atualmente no seu index), quais estão modificados mas não ainda não foram selecionados, e quais estão completamente sem nenhuma seleção.

```
$>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
```

Se você descartar o index completamente, você em geral não perde qualquer informação contanto que você tenha o nome da tree que ele descreveu.

E com isso, você deveria ter um bom entendimento das coisas básicas que o Git faz por traz da cena, e porque ele é um pouco diferente da maioria dos outros sistemas SCM. Não se preocupe se você não entendeu completamente tudo até agora; iremos revisar todos esses tópicos nas próximas seções. Agora estamos prontos para irmos a instalação, configuração e uso do Git.

Capítulo 3

A Primeira vez

INSTALANDO O GIT

Instalando a partir do Código Fonte

Em resumo, em um sistema baseado em Unix, você pode baixar o código fonte do Git em Git Download Page, e então executar essas linhas :

```
$ make prefix=/usr all ;# com seu próprio usuário
$ make prefix=/usr install ;# como root
```

Você precisará dessas bibliotecas instaladas expat, curl, zlib, e openssl - embora com uma possível exceção do *expat*, esse normalmente já existe no sistema.

Linux

Se você está rodando o Linux, você pode provavelmente instalar o Git facilmente através do sistema de gerenciamento de pacotes nativo.

```
$ yum install git-core
```

```
$ apt-get install git-core
```

Se não funcionar, você pode baixar os pacotes .deb ou .rpm daqui:

RPM Packages

Stable Debs

Se você preferir instalar a partir do código fonte em um sistema Linux, este artigo pode ser útil:

Article: Installing Git on Ubuntu

Mac OS X 10.4

Em ambos Mac 10.4 e 10.5, você pode instalar o Git via MacPorts, se você tiver que instalá-lo. Se não, você pode instalá-lo a partir [daqui] (http://www.macports.org/install.php).

Uma vez instalado, tudo o que você deveria fazer é:

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

Se você preferir instalar a partir do código fonte, esses artigos podem ser úteis:

Article: Installing Git on Tiger

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

Mac 10.5 em diante

Com o Leopard, você pode também instalar através do MacPorts, mas aqui você tem opções adicionais para uso do ótimo instalador, que você pode baixar daqui: Git OSX Installer

Se você preferir instalá-lo a partir do código fonte, esses guias podem ser particularmente úteis para você:

Article: Installing Git on OSX Leopard

Article: Installing Git on OS 10.5

Esse instalador também funciona sobre Snow Leopard.

Windows

No Windows, instalar o Git é muito fácil. Simplesmente baixe e instale o pacote msysGit.

Veja no capítulo *Git no Windows* por um screencast demonstrando a instalação e uso do Git no Windows.

CONFIGURAÇÃO E INICIALIZAÇÃO

Git Config

A primeira coisa que você vai querer fazer é configurar o seu nome e o endereço de email para o Git usá-lo para assinar seus commits.

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

Isso irá configurar um arquivo em seu diretório home que pode ser usado por qualquer um dos seus projetos. Por padrão esse arquivo é *~/.gitconfig* e o conteúdo irá se parecer com isso:

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

Se você quer sobrescrever esses valores para um projeto específico (para usar um endereço de email do trabalho, por exemplo), você pode executar o comando *git config* sem a opção *--global* naquele projeto. Isso irá adicionar uma seção [user] como mostrado acima para o arquivo *.git/config* na raiz de seu projeto.

Capítulo 4

Uso Básico

CONSEGUINDO UM REPOSITÓRIO GIT

Então agora que nós já configuramos, precisamos de um repositório Git. Podemos fazer isso de duas maneiras - nós podemos *clonar* um já existente, ou podemos *inicializar* um dentro de algum projeto que ainda não tenha controle de versão , ou a partir de um diretório vazio.

Clonando um Repositório

Para que consigamos uma cópia de um projeto, você irá precisar saber qual a URL - a localização do repositório - do projeto Git. Git pode operar sobre muitos diferentes protocolos, então ele pode iniciar com ssh://, http(s)://, git://, ou só o nome de usuário (no qual o git assumirá ssh). Alguns repositórios podem ser acessados sobre mais do que um protocolo. Por exemplo, o código fonte do próprio Git pode ser clonado sobre o protocolo git:// :

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

ou sobre http:

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

O protocolo git:// é mais rápido e mais eficiente, mas algumas vezes é necessário usar http quando estão por tráz de firewalls corporativo ou que você tenha. Nesses casos você deveria então ter um novo diretório chamado 'git' que contém todos os códigos fontes do Git e o histórico - que é basicamente uma cópia do que estava no servidor.

Por padrão, o Git nomeará o novo diretório do projeto de sua clonagem de acordo com o nome do arquivo no último de nível na URL antes de '.git'. (ex.: *git clone http://git.kernel.org/linux/kernel/git/torvalds/linux-2.6.git* resultará em um novo diretório chamado 'linux-2.6')

Inicializando um Novo Repositório

Suponha que você tem um tarball chamado project.tar.gz com seu trabalho inicial. Você pode colocar ele sobre um controle de revisões do git como segue.

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

Git irá responder

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

Agora você tem um diretório de trabalho inicializado - você pode notar um novo diretório criado chamado ".git".

gitcast:c1_init

FLUXO NORMAL DE TRABALHO

Altere alguns arquivos, então adicione seus conteúdos alterados para o index:

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

Você agora está pronto para realizar o commit. Você pode ver o que será levado para commit usando git diff com a opção --cached:

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

(Sem a opção --cached, git diff mostrará a você qualquer modificação que você tem feito mas ainda não foi adicionado no index). Você pode também conseguir uma breve sumário da situação com git status:

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

Se você precisar fazer qualquer ajustes a mais, então faça-o agora, e então adicione qualquer novo conteúdo modificado no index. Finalmente, commit suas mudanças com:

```
$ git commit
```

Isso irá novamente mostrar a você uma mensagem descrevendo as mudanças, e então gravar uma nova versão do projeto.

Alternativamente, ao invés de executar `git add`, você pode usar:

```
$ git commit -a
```

que irá automaticamente avisar sobre quaisquer arquivos modificados (mas não novos), adicioná-los no index, e realizar o commit, tudo de uma vez.

Uma nota sobre as mensagens de commit: Embora não necessário, é uma boa ideia iniciar a mensagem de commit em uma linha curta (menos do que 50 caracteres) resumindo as mudanças, seguido por uma linha em branco e então, mais uma descrição profunda. Ferramentas que transformam commits em emails, por exemplo, usam a primeira linha para o Assunto: e o resto da mensagem do commit para o corpo do email.

Git percebe conteúdo não arquivos

Muitos sistemas de controle de revisões dispõem de um comando "add" que chama o sistema para iniciar a busca por mudanças em novos arquivos. O comando "add" faz algumas coisas das mais simples as mais poderosas: `git add` é usado ambos para arquivos novos e arquivos alterados, e em ambos os casos fornece um snapshot dos arquivos dados e os que estão no index, prontos para inclusão no próximo commit.

gitcast:c2_normal_workflow

BÁSICO SOBRE BRANCHING E MERGING

Um simples repositório git pode manter múltiplos branches de desenvolvimento. Para criar um novo branch chamado "experimental", use :

```
$ git branch experimental
```

Se executar agora

```
$ git branch
```

você terá uma lista de todos os branches existentes:

```
  experimental
* master
```

O branch "experimental" é o que você criou, e o branch "master" é um padrão, que foi criado automaticamente para você. O asterisco marca o branch no qual você está atualmente. Digite

```
$ git checkout experimental
```

para trocar para o branch experimental. Agora edite um arquivo, realize o commit da alteração e volte para o branch master:

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

Verifique que a alteração que você fez não está mais visível, visto que foi feito sobre o branch experimental e agora que você está de volta sobre o branch master.

Você pode fazer uma alteração diferente sobre o branch master:

```
(edite um arquivo)
$ git commit -a
```

nesse ponto os dois branches tem divergências, com diferentes modificações em cada um. Para realizar um merge das alterações feitas no branch experimental para o master, execute

```
$ git merge experimental
```

Se as mudanças não conflitarem, você terminou aqui. Se existem conflitos, marcas serão deixadas nos arquivos problemáticos mostrando os conflitos;

```
$ git diff
```

irá mostrá-los. Um vez que você editou os arquivos para resolver os conflitos

```
$ git commit -a
```

irá realizar o commit do resultado do merge. Finalmente

```
$ gitk
```

mostrará uma ótima representação gráfica resultante do histórico.

Nesse ponto você poderá apagar o branch experimental com

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

Esse comando assegura que as alterações no branch experimental já estão no no branch atual.

Se você desenvolve sobre o um branch crazy-idea, então se arrependeu, você sempre pode apagar o branch com

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

Branches são baratos e fáceis, então é uma boa maneira para testar alguma coisa.

Como realizar um merge

Você pode reunir dois diferentes branches de desenvolvimento usando git merge:

```
$ git merge branchname
```

realiza um merge com as alterações feitas no branch "branchname" no branch atual. Se existirem conflitos -- por exemplo, se o mesmo arquivo é modificado em duas diferentes formas no branch remoto e o branch local -- então você será avisado; a saída pode parecer alguma coisa com isso:

```
$ 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.
```

Marcadores dos conflitos são deixados nos arquivos problemáticos, e depois que você resolve os conflitos manualmente, você pode atualizar o index com o conteúdo e executar o git commit, como você faria normalmente quando modifica um arquivo.

Se você examinar o resultado do commit usando o gitk, você verá que ele possui dois pais: um apontando para o topo do branch atual, e um para topo do outro branch.

Resolvendo um merge

Quando um merge não é resolvido automaticamente, o git deixa o index e a árvore de trabalho em um estado especial que fornece a você todas as informações que você precisa para ajudar a resolver o merge.

Arquivos com conflitos são marcados especialmente no index, então até você resolver o problema e atualizar o index, o comando git commit irá falhar.

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

Também, git status listará esses arquivos como "unmerged", e os arquivos com conflitos terão os marcadores dos conflitos adicionados, assim:

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

Tudo que você precisa fazer é editar os arquivos para resolver os conflitos, e então

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

Veja que a mensagem de commit já estará preenchida nele para você com algumas informações sobre o merge. Normalmente você pode usá-la sem mudança nessa mensagem padrão, mas você pode adicionar um comentário adicional se desejado.

Tudo acima é o que você precisa saber para resolver um simples merge. Mas o git também provê mais informações para ajudar a resolver os conflitos:

Desfazendo um merge

Se você ficar preso e decide desistir e jogar toda a bagunça fora, você pode sempre retornar ao estado do pre-merge com

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

Ou, se você já estiver realizado o commit do merge que você quer jogar fora,

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

Contudo, esse último comando pode ser perigoso em alguns casos -- nunca jogue fora um commit se esse commit já pode ter sido realizado um merge em outro branch, que pode confundir novos merges.

Merges Fast-forward

Existe um caso especial não mencionado acima, que é tratado diferentemente. Normalmente, um merge resulta em um commit com dois pais, um de cada uma das duas linhas de desenvolvimento que foram realizados o merge.

Contudo, se o branch atual não divergiu do outro -- então cada commit presente no branch atual já está contido no outro -- então o git só realiza um "fast foward"; o HEAD do branch atual é movido para o ponto do HEAD do branch que realiza o merge, sem que qualquer novo commit seja criado.

gitcast:c6-branch-merge

REVISANDO O HISTÓRICO - GIT LOG

O comando git log pode mostrar listas de commits. Com ele são mostrados todos commits atingíveis a partir do commit pai; mas você pode também fazer requisições mais específica.

```
$ git log v2.5..        # commits desde (not reachable from) v2.5
$ git log test..master  # commits atingíveis do master mas não test
$ git log master..test  # commits atingíveis do test mas não do
$ git log master...test # commits atingível de qualquer um dos test ou
                        # master, mas não ambos
$ git log --since="2 weeks ago" # commits das 2 últimas semanas
$ git log Makefile      # commits que modificaram o Makefile
$ git log fs/           # commits que modificaram qualquer arquivo sobre fs/
$ git log -S'foo()'     # commits que adicionaram ou removeram arquivos
                        # e casam com o texto 'foo()'
$ git log --no-merges   # não mostra commits com merge
```

E é claro você pode combinar todas essas opções; vamos encontrar commits desde v2.5 que modificou o Makefile ou qualquer arquivo sobre fs/:

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

Git log mostrará uma listagem de cada commmit, com os commits mais recentes primeiro, que casa com os argumentos dados no comando.

```
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 does

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.
```

Você pode também perguntar ao git log para mostrar patches:

```
$ 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:

 [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):
```

Estatísticas do Log

Se você passar a opção `--stat` para 'git log', ele mostrará a você quais arquivos tem alterações naquele commit e quantas linhas foram adicionadas e removida de cada um.

```
$ 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(-)
```

Formatando o Log

Você também pode formatar a saída do log como queira. A opção '--pretty' pode dar um número de formatos pré determinados, como '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
```

ou você pode usar o formato 'short' :

```
$ 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
```

Você também pode usar 'medium', 'full', 'fuller', 'email' ou 'raw'. Se esses formatos não são exatamente o que você precisa, você também pode criar seu próprio formato com a opção '--pretty=format' (veja a documentação do git log para ver todas as opções de formatação).

```
$ 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:
```

Outra coisa interessante que você pode fazer é visualizar o gráfico do commit com a opção '--graph', como:

```
$ 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
```

Dará uma ótima representação em formato ASCII dos históricos dos commits.

Ordenando o Log

Você também pode visualizar as entradas do log em algumas diferentes ordens. Veja que git log inicia com os commits mais recentes e vai até os mais antigos pais; contudo, desde que o histórico do git pode conter múltiplas linhas diferentes de desenvolvimento, a ordem particular que os commits são listados podem ser de alguma forma arbitrárias.

Se você quer especificar uma certa ordem, você pode adicionar uma opção de ordenação para o comando git log.

Por padrão, os commits são mostrados em ordem cronológica reversa.

Contudo, você também pode especificar '--topo-order', que faz os commits aparecerem em order topológica (ex.: commits descendentes são mostrados antes de seus pais). Se visualizarmos o git log do repositório Grit em topo-order, você pode ver que as linhas de desenvolvimento são todas agrupadas juntas.

```
$ 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.
```

Você também pode usar '--date-order', que ordena os commits inicialmente pelas datas dos commits. Essa opção é similar ao --topo-order no sentido de que nenhum pai vem antes de todos os filhos, mas por outro lado elas são ordenadas por ordem do timestamp do commit. Você pode ver que as linhas de desenvolvimento aqui não são agrupadas juntas, que eles pulam por cima quando o desenvolvimento paralelo ocorreu:

```
$ 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
```

Finalmente, você pode reverter a ordem do log com a opção '--reverse'.

gitcast:c4-git-log

COMPARANDO COMMITS - GIT DIFF

Você pode gerar diffs entre duas versões quaisquer do seu projeto usando git diff:

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

Isso produzirá o diff entre os dois branches. Se você preferir encontrar o diff dos ancestrais comuns do test, você pode usar três pontos ao invés de dois.

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

git diff é uma ferramenta incrivelmente útil para entender as alterações que existem entre dois pontos quaisquer no histórico de seu projeto, ou para ver o que as pessoas estão tentando introduzir em novos branches, etc.

O que você levará para o commit

Você usará normalmente git diff para entender as diferenças entre seu último commit, seu index, e seu diretório de trabalho. Um uso comum é simplesmente executar

```
$ git diff
```

que mostrará a você alterações no diretório de trabalho atual que ainda não foi selecionado para o próximo commit. Se você quer ver o que *está* selecionado para o próximo commit, você pode executar

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

que mostrará a você as diferenças entre o index e o seu último commit; o que será levado para o commit se você executar "git commit" sem a opção "-a". Finalmente, você pode executar

```
$ git diff HEAD
```

que mostra as alterações no diretório de trabalho atual desde seu último commit; o que será levado para o commit se você executar "git commit -a".

Mais opções Diff

Se você quer ver como seu diretório atual difere do estado em outro branch do projeto, você pode executar algo como:

```
$ git diff test
```

Isso mostrará a você a diferença entre seu diretório de trabalho atual e o snapshot sobre o branch 'test'. Você também pode limitar a comparação para um arquivo específico ou sub diretório pela adição do *path limiter*:

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

Esse comando mostrará as alterações entre seu diretório de trabalho atual e o último commit (ou, mais exatamente, dar uma dica sobre o branch atual), limitando a comparação para os arquivos no diretório 'lib'.

Se você não quer ver o patch inteiro, você pode adicionar a opção '--stat', que limitará a saída para os arquivos que possui alteração junto com um pequeno gráfico em texto representando a quantidade de linhas alteradas em cada arquivo.

```
$>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(-)
```

As vezes fazer isso é mais fácil para visualizar tudo o que foi alterado, para refrescar sua memória.

FLUXO DE TRABALHO DISTRIBUIDO

Suponha que Alice tenha iniciado um novo projeto com um repositório git em /home/alice/project, e que Bob, que possui um diretório home na mesma máquina, quer contribuir.

Bob inicia com:

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

Isso cria um novo diretório "myrepo" contendo um clone do repositório de Alice O clone está na mesma condição de igualdade do projeto original, possuindo sua própria cópia do histórico do projeto original.

Bob então faz algumas alterações e commits dele:

```
(edite alguns arquivos)
$ git commit -a
(repita quando necessário)
```

Quando terminar, ele comunica a Alice para realizar um pull das alterações do repositório em /home/bob/myrepo. Ela faz isso com:

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

Isso realiza um merge com as alterações do branch master de Bob para o branch atual de Alice. Enquanto isso se Alice tem feito suas próprias modificações, então ela pode precisar corrigir manualmente quaisquer conflitos. (Veja que o argumento "master" no comando acima é na verdade desnecessário, já que ele é o padrão).

O comando "pull" realiza assim duas operações: ele recebe as alterações mais recentes do branch remoto, então realiza um merge dele no branch atual.

Quando você está trabalhando em um pequeno grupo muito unido, não é incomum interagir com o mesmo repositório e outro novamente. Definindo um repositório como 'remote', você pode fazer isso facilmente:

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

Com isso, Alice pode realizar a primeira operação sozinha usando o comando "git fetch" sem realizar um merge dele com o seu próprio branch, usando:

```
$ git fetch bob
```

Ao contrário da forma longa, quando Alice recebe as novas alterações de Bob usando um repositório remoto configurado com 'git remote', que foi recuperado e armazenado em um branch remoto, nesse caso 'bob/master'. Então depois disso:

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

mostra uma lista de todas as alterações que Bob fez desde quando ele criou o branch master de Alice.

Depois de examinar essas alterações, Alice poderá realizar um merge com as alterações dentro de seu branch master:

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

Esse 'merge' também pode ser feito 'realizando um pull a partir de seu próprio branch remoto registrado', dessa forma:

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

Perceba que o git pull sempre realiza o merge dentro de seu branch atual, sem levar em conta o que é dado na linha de comando.

Por fim, Bob pode atualizar seu repositório com as últimas alterações de Alice usando:

```
$ git pull
```

Veja que não foi preciso dar o caminho para o repositório de Alice; quando Bob clonou o repositório de Alice, o git armazenou a localização do repositório dela nas configurações de seu repositório, e essa localização é usada pelo pull:

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

(A configuração completa criada por git-clone é visível usando "git config -l"), e a página do manual git config explica o significado de cada opção.

Git também deixa uma cópia original do branch master de Alice sobre o nome de "origin/master":

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

Se Bob decide trabalhar a partir de um host diferente, ele ainda pode realizar clones e pulls usando o protocolo ssh:

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

Alternativamente, git possui um protocolo nativo, ou pode usar rsync or http; veja git pull para mais detalhes.

Git também pode ser usado no modo CVS, com um repositório central para onde vários usuários enviam alterações; veja git push e gitcvs-migration.

Repositórios Git Públicos

Uma outra forma de enviar alterações para um projeto é informar ao responsável por aquele projeto para realizar um pull das alterações de seu repositório usando git pull. Essa é uma forma de conseguir atualizações do repositório "principal", mas ele também funciona em outras direções.

Se o responsável pelo projeto e você possuem contas na mesma máquina, então você pode somente realizar um pull das alterações diretamente do outro repositório; comandos que aceitam URLs de repositórios como argumentos também aceitam nomes de diretórios locais:

```
$ git clone /caminho/para/repositorio
$ git pull /caminho/para/outro/repositorio
```

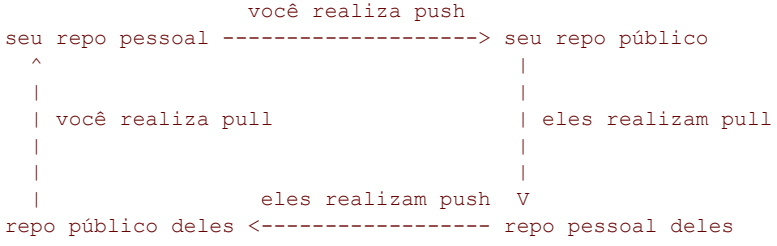
ou uma URL ssh:

```
$ git clone ssh://suamaquina/~voce/repositorio
```

Para projetos com alguns desenvolvedores, ou para a sincronização de alguns repositórios privados, isso pode ser tudo que você precisa.

Contudo, a forma mais comum de fazer isso é manter um repositório público separado (na verdade em uma máquina diferente) para os outros realizarem pull das alterações. Isso é na verdade mais conveniente, e permite claramente a você separar trabalho pessoal em progresso do trabalho visível publicamente.

Você continuará a fazer seu trabalho diário em seu repositório pessoal, mas periodicamente realizar um "push" das alterações de seu repositório pessoal para o seu repositório público, permitindo os outros desenvolvedores realizar pulls daquele repositório. Então o fluxo de alterações, na situação onde existe um outro desenvolvedor com repositório público, parece com isso:



Enviando alterações para um repositório público

Veja que exportando via http ou git, permite outros mantenedores recuperem suas últimas alterações, mas eles não tem permissão de acesso para escrita. Por isso, você precisará atualizar o repositório público com as últimas alterações criadas em seu repositório privado.

Uma forma simples de fazer isso é usando git push e ssh; para atualizar o branch remoto chamado "master" com o último estado de seu branch chamado "master", execute

```
$ git push ssh://seuservidor.com/~voce/proj.git master:master
```

ou só

```
$ git push ssh://seuservidor.com/~voce/proj.git master
```

Como o git-fetch, git-push irá reclamar se isso não resultar em um fast forward; veja a seção seguinte sobre como proceder nesse caso.

Veja que o alvo de um "push" é normalmente um repositório mínimo. Você também pode enviar para um repositório que já possui uma árvore de trabalho, mas essa árvore não será atualizada pelo push. Isso pode levar a resultados inesperados se o branch que você enviou é o branch atual!

Como com o git-fetch, você também pode ajustar as opções de configuração, então por exemplo, depois

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

você deverá estar capaz de realizar o push acima só com

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

Veja as explicações das opções remote..url, branch..remote, e remote..push em git config para mais detalhes.

O que fazer quando um push falha

Se um push não resultar em um fast forward do branch remoto, então falhará com um erro desse tipo:

```
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://seuservidor.com/~voce/proj.git'
```

Isso pode acontecer, por exemplo, se você

```
- usar 'git-reset --hard' para remover commit já publicados, ou
- usar 'git-commit --amend' para substituir commits já publicados ou
- usar 'git-rebase' para recriar qualquer commit já publicado.
```

Você pode forçar git-push para realizar a atualização precedendo o nome do branch com um sinal de +:

```
$ git push ssh://seuservidor.com/~voce/proj.git +master
```

Normalmente quando um branch head é modificado em um repositório público, ele é modificado para apontar para um descendente desse commit que ele apontou antes. Forçando um push nessa situação, você quebra aquela convenção.

Contudo, essa é uma prática comum para pessoas que precisam de uma forma simples para publicar uma série de patch de um trabalho em progresso, e é um compromisso aceitável contanto que você avise os outros desenvolvedores que é dessa forma que pretende gerenciar o branch.

Dessa forma também é possível para um push falhar quando outras pessoas tem o direito de enviar para o mesmo repositório. Nesse caso, a solução correta para tentar re-enviar depois da primeira atualização de seu

trabalho: qualquer um pull, ou um fetch seguido por um rebase; veja a próxima seção e gitcvs-migration para mais informações.

gitcast:c8-dist-workflow

TAGS NO GIT

Tags "Peso-Leve"

Nós podemos criar uma tag para referenciar um commit particular executando git tag sem nenhum argumento.

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

Depois disso, nós podemos usar a tag 'stable-1' para referenciar o commit 1b2e1d63ff.

Isso cria uma tag "peso-leve", basicamente atua como um branch, mas que nunca se altera. Se você também gostaria de incluir um comentário na tag, e possivelmente assinar criptograficamente, então em vez disso nós podemos criar um *tag object*.

Tag Objects

Se um dos **-a**, **-s**, ou **-u** é passado, o comando cria um objeto tag e solicita uma mensagem da tag. A não ser que -m ou -F seja dado, um editor é iniciado para o usuário digitar a mensagem para a tag.

Quando isso acontece, um objeto é adicionado para o banco de dados de objeto Git e a referência da tag aponta para esse *objeto tag*, em vez realizar um commit dele. A força disso é que você pode assinar a tag, então você pode verificar que este é o último commit correto. Você pode criar um objeto tag assim:

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

Na verdade é possível adicionar um tag em qualquer objeto, mas é mais comum colocar tags em objetos do tipo commit. (No código fonte do kernel do Linux, a primeiro objeto tag referencia uma árvore, em vez de um commit)

Tags Assinadas

Se você tem uma chave GPG configurada, você pode criar tags assinadas mais facilmente. Primeiro, provavelmente irá querer configurar o id de sua chave no seu arquivo *.git/config* ou *~.gitconfig*

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

Você também pode configurá-lo com

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

Agora você pode criar uma tag assinada através da substituição do **-a** pelo **-s**.

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

Se você não tem sua chave GPG no seu arquivo de configuração, você pode realizar a mesmo coisa dessa forma:

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

Capítulo 5

Uso intermediário

IGNORANDO ARQUIVOS

Um projeto frequentemente irá gerar arquivos que você 'não' quer gerenciar com o git. Isso tipicamente inclui arquivos gerados por um processo de construção ou arquivos de backup temporário feitos pelo seu editor. É claro, 'não' gerenciar arquivos com o git é apenas uma questão de 'não' chamar "'git-add'" nele. Mas isso rapidamente torna-se irritante ter esses arquivos não selecionados espalhados por ai; eles fazem o "'git add .'" e "'git commit -a'" praticamente inúteis, e eles permanecem sendo visualizados pelo "'git status'".

Você pode dizer ao git para ignorar certos arquivos através da criação do arquivo chamado .gitignore na raiz do seu diretório de trabalho, como por exemplo:

```
# Linhas que iniciam com '#' são considerados comentários
# Ignora qualquer arquivo chamado foo.txt.
foo.txt
```

```
# Ignora arquivos html (gerados),
*.html
# com exceção de foo.html que é mantido manualmente.
!foo.html
# Ignora objetos e arquivos históricos.
*.[oa]
```

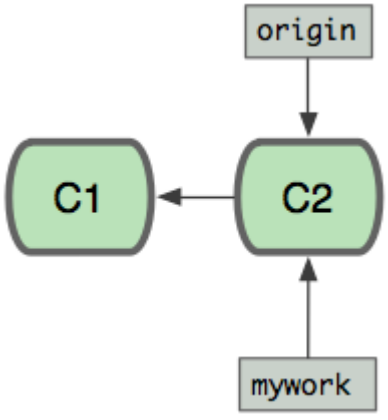
Veja gitignore para uma detalhada explicação da sintaxe. Você também pode colocar arquivos .gitignore em outros diretórios na sua árvore de trabalho e eles se aplicarão a esses diretórios e subdiretórios. Os arquivos `.gitignore` podem ser adicionados em seu repositório como todos outros arquivos (só execute `git add .gitignore` e `git commit`, como sempre) que é conveniente quando os padrões de exclusão (por exemplo os padrões que casam com arquivos construidos) também farão sentido para outros usuários que clonam seu repositório.

Se você escolhe padrões de exclusão para afetar somente certos repositórios (ao invés de cada repositório para um dado projeto), você pode por exemplo colocá-los em um arquivo em seu repositório chamado .git/info/exclude, ou em qualquer arquivo especificado pela variável `core.excludesfile`. Alguns comandos git também fornecem padrões de exclusão diretamente na linha de comando. Veja gitignore para mais detalhes.

REBASING

Suponha que você crie um branch "mywork" sobre um branch remoto "origin".

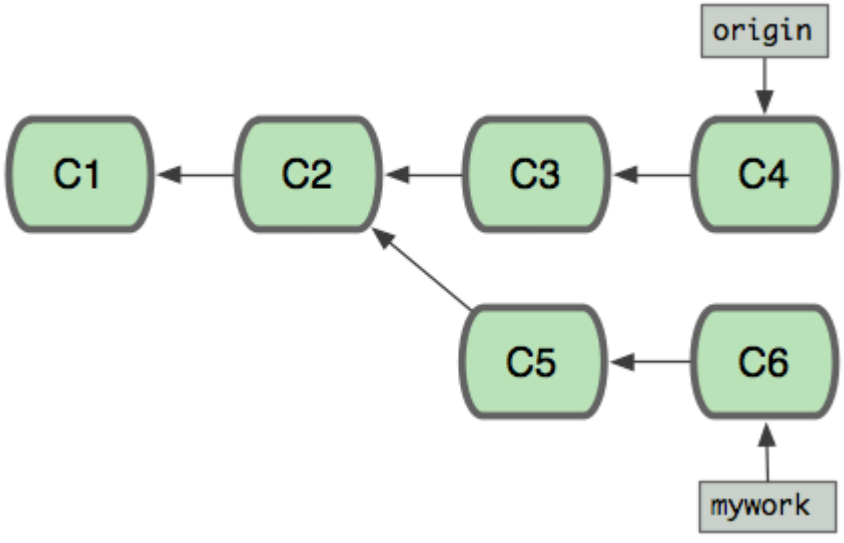
```
$ git checkout -b mywork origin
```



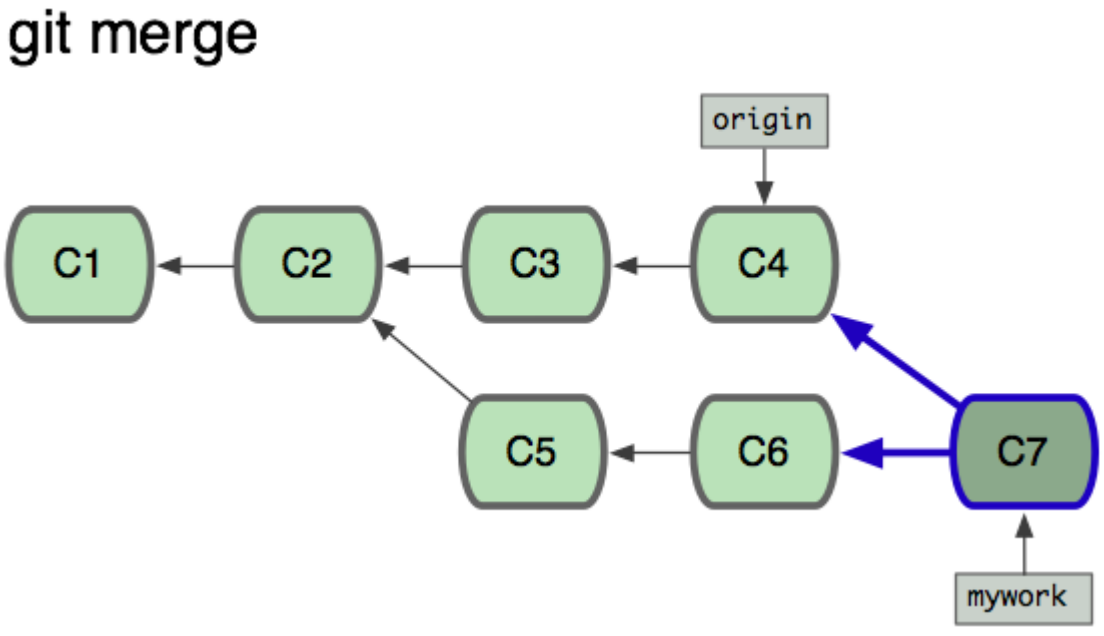
Agora você faz algum trabalho, criando dois novos commits.

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

Enquanto isso, alguém também faz algum trabalho criando dois novos commits sobre o branch origin. Nisso ambos 'origin' e 'mywork' avançam seus trabalhos, existindo divergências entre eles.



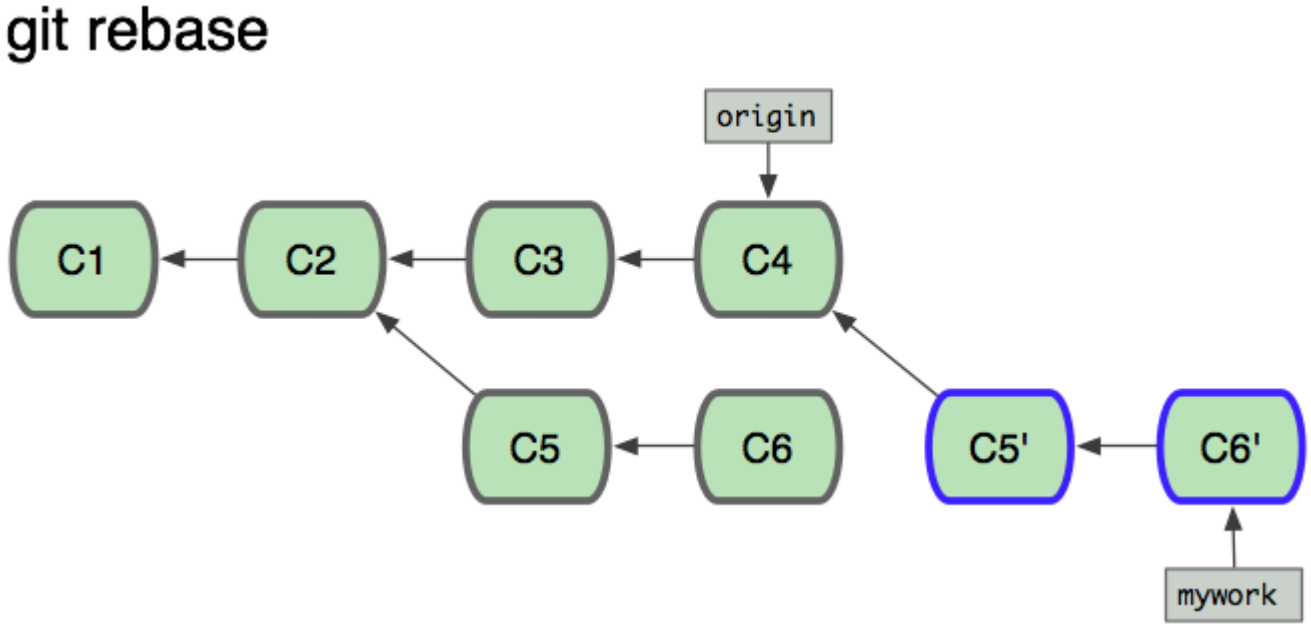
Neste ponto, você poderia usar "pull" para juntar suas alterações de volta nele; o resultado criará um novo commit através do merge, como isso:



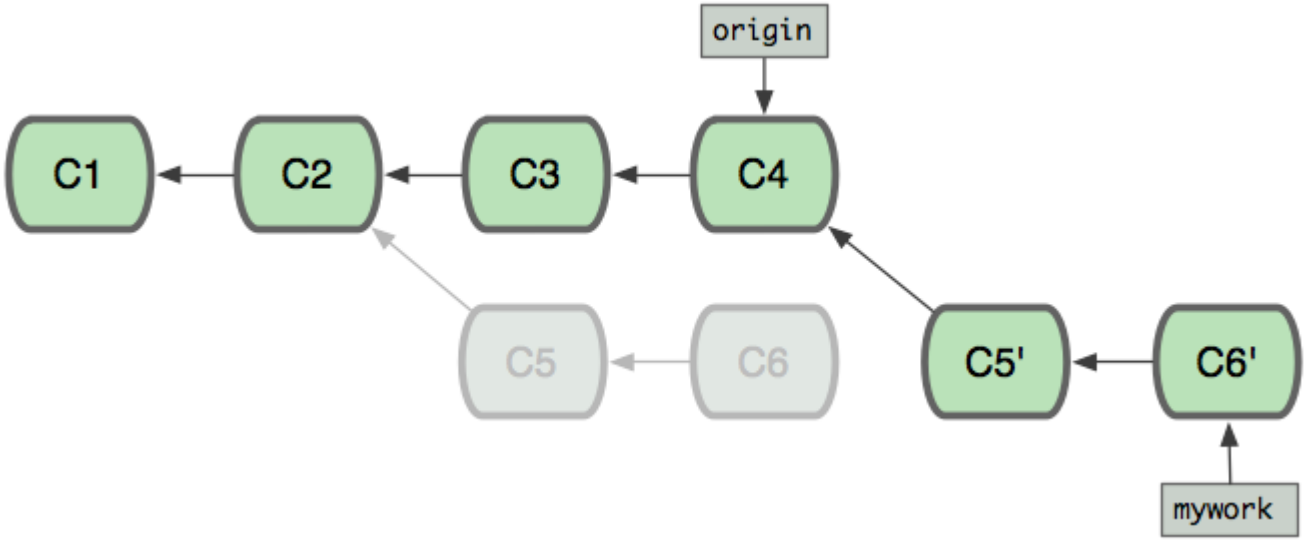
Contudo se você prefere manter o histórico em 'mywork', como uma simples série de commits sem qualquer merge, ao invés disso você pode escolher usar git rebase:

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

Isso removerá cada um dos seus commits de 'mywork', temporariamente salvando eles como patches (em um diretório chamado ".git/rebase"), atualizar 'mywork' para apontar para a última versão de 'origin', então aplicar cada um dos patches salvos para o novo 'mywork'.

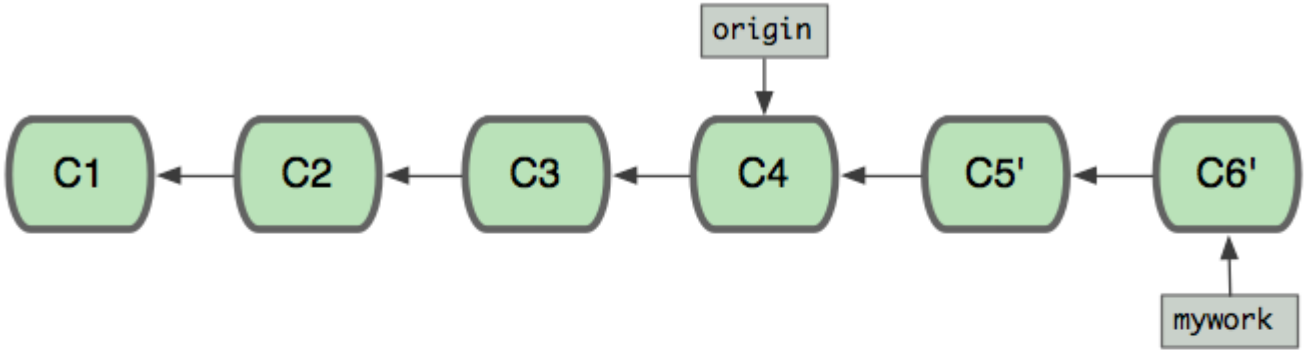


Uma vez que ('mywork') é atualizado para apontar para o mais novo objeto commit criado, seus velhos commits serão abandonados. Eles provavelmente serão removidos se você executar a coleta de lixo. (see git gc)

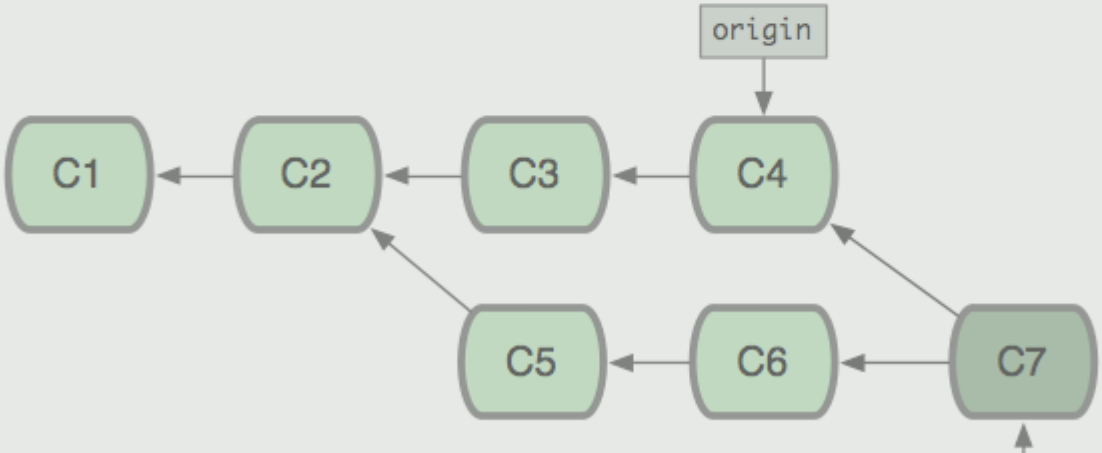


Então agora podemos ver a diferença em nosso histórico entre um merge e um rebase executado:

git rebase



git merge



No processo de rebase, ele pode descobrir alguns conflitos. Nesse caso ele interromperá e permitirá a você corrigir os conflitos; depois que corrigí-los, use "git-add" para atualizar o index com esse conteúdo, e então, ao invés de executar git-commit, só execute

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

e o git continuará aplicando o resto dos patches.

Em qualquer ponto você pode usar a opção '--abort' para esse processo e retornar 'mywork' para o estado que tinha antes de você iniciar o rebase:

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

gitcast:c7-rebase

REBASING INTERATIVO

Você pode também realizar um rebase interativamente. Isso é usado muitas vezes para re-escrever seus próprios objetos commit antes de enviá-los para algum lugar. Isso é uma forma fácil de dividir, juntar, ou re-ordenar os commits antes de compartilhá-los com os outros. Você pode também usá-lo para limpar commits que você tenha baixado de alguém quando estiver aplicando eles localmente.

Se você tem um número de commits que você gostaria de alguma maneira modificar durante o rebase, você pode invocar o modo interativo passando um '-i' ou '--interactive' para o comando 'git rebase'.

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

Isso invocará o modo de rebase interativo sobre todos os commits que você tem feito desde a última vez que você realizou um pull (ou merge de um repositório origin).

Para ver de antemão quais são os commits, você pode executar dessa forma:

```
$ git log origin/master..
```

Uma vez que você rodar o comando 'rebase -i origin/master', você será levado para o seu editor com algo parecido com isso:

```
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.
#
```

Isso significa que existem 5 commits desde o último push realizado e lhe dará uma linha por commit com o seguinte formato:

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

Agora, você pode alterar a ação (que é por padrão 'pick') para qualquer um entre 'edit' ou 'squash', ou deixá-lo como 'pick'. Você também pode re-ordenar os commits movendo as linhas como você quiser. Então, quando você sair do editor, o git tentará aplicar os commits como eles estão organizados agora e realizar a ação especificada.

Se 'pick' é especificado, ele simplesmente tentará aplicar o patch e salvar o commit com a mesma mensagem de antes.

Se 'squash' é especificado, ele combinará aquele commit com um anterior para criar um novo commit. Você cairá novamente em seu editor para juntar as mensagens de commit dos dois commits que agora são combinados. Então, se você sair do editor com isso:

```
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
```

Então você terá que criar uma única mensagem de commit dele:

```
# 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.
```

Uma vez que você tem editado a mensagem de commit e saído do editor, o commit será salvo com a sua nova mensagem.

Se 'edit' é especificado, fará a mesma coisa, mas irá parar antes de mover para o próximo commit e o levará para a linha de comando para você poder corrigir o commit, ou modificar o conteúdo do commit de alguma forma.

Se você quisesse dividir um commit, por exemplo, você especificaria 'edit' para esse commit:

```
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
```

E então quando você for levado para a linha de comando, você reverte aquele commit em dois (ou mais) novos. Digamos que o 21d80a5 modificou dois arquivos, arquivo1 e arquivo2, e você quisesse dividir eles em commits separados. Você poderia fazer isso depois que o rebase deixá-lo na linha de comando:

```
$ 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
```

E agora ao invés dos 5 commits, você terá 6.

A última coisa útil que o modo interativo do rebase pode fazer é retirar commits para você. Se ao invés de escolher 'pick', 'squash' ou 'edit' para a linha do commit, você simplesmente remove a linha e isso removerá o commit do histórico.

SELEÇÃO INTERATIVA

Seleção interativa é realmente uma ótima forma de trabalhar e visualizar o index do Git. De início, simplesmente digite 'git add -i'. O Git mostrará a você todos os arquivos modificados que você tem e seus status.

```
$>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>
```

Nesse caso, podemos ver que existem 5 arquivos modificados que não foram adicionados em nosso index ainda (sem seleção), e até mesmo a quantidade de linhas que foram adicionadas e removidas de cada um. Então ele mostra-nos um menu interativo mostrando o que podemos fazer.

Se quisermos selecionar esses arquivos, podemos digitar '2' ou 'u' para o modo de atualização. Então eu posso especificar quais arquivos eu quero selecionar (adicionar no index) através da digitação dos números que correspondem aos arquivos (nesse caso, 1-4)

```
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>>
```

Se teclar enter, serei levado de volta para o menu principal onde posso ver que arquivos que possuem o status modificado:

```
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
```

Nós podemos ver o primeiro dos quatro arquivos selecionados e o último que ainda não está. Isso é basicamente uma forma resumida para ver a mesma informação que vemos quando executamos 'git status' a partir da linha de comando:

```
$ 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
#
```

Existem várias coisas úteis que podemos fazer, incluindo deselecionar arquivos (3: revert), adicionar arquivos não selecionados (4: add untracked), e ver as diferenças (6: diff). Isso tudo de forma muito simples. Contudo, existe um comando muito legal aqui, que é selecionar patches (5: patch).

Se você digitar '5' ou 'p' no menu, git mostrará a você a diferença por pacth e perguntar se você quer selecionar cada um. Na verdade essa é uma forma que você pode selecionar para um commit partes de um arquivo editado. Se você editou um arquivo e quer somente realizar um commit de uma parte dele e não a parte

inacabada, ou commit da documentação ou espaços em branco, você pode usar 'git add -i' para fazê-lo facilmente.

Aqui eu tenho selecionado alterações para o arquivo book_index_template.html, mas não para todos eles:

```
          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
```

Quando você terminar de fazer as alterações para o index através do 'git add -i', você simplesmente sai (7: quit) e então executa 'git commit' para realizar o commit das alterações selecionadas. Lembre-se **não** execute 'git commit -a' que descartará todas as alterações que você cuidadosamente fez e simplemente realizará um commit de tudo.

gitcast:c3_add_interactive

STASHING

Enquanto você está trabalhando no meio de algo complicado, você encontra um erro não documentado mas óbvio e trivial. Você gostaria de corrigir ele antes de continuar. Você pode usar git stash para gravar o estado atual de seu trabalho, e depois corrigir o erro (ou, opcionalmente depois de fazê-lo sobre um branch diferente e então voltar), e recuperar as alterações que estava trabalhando antes do erro.

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

Esse comando gravará suas alterações para o `stash` e resetará sua árvore de trabalho e o index para o início de seu branch atual. Você então pode fazer as correções como de costume.

```
... edite e teste ...
$ git commit -a -m "blorpl: typofix"
```

Depois disso, você pode voltar para aquilo que estava trabalhando usando `git stash apply`:

```
$ git stash apply
```

Fila de Stash

Você também pode usar o stash para enfileirar stashes. Se você executar 'git stash list' você pode ver que staches você tem salvo:

```
$>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
```

Então você pode aplicar eles individualmente com 'git stash apply stash@{1}'. Você pode limpar a lista com 'git stash clear'.

GIT TREEISHES

Existem vários caminhos para referenciar um commit ou tree particular do que "cuspir" o SHA de 40 dígitos inteiro. No Git, eles são conhecidos como um 'treeish'.

SHA Parcial

Se o SHA de seu commit é '`980e3ccdaac54a0d4de358f3fe5d718027d96aae`', o git reconhecerá qualquer um desses igualmente:

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

Contanto que o SHA parcial seja único - ele não pode ser confundido com outro (que é inacreditávelmente improvável se você usa pelo menos 5 caracteres), git expandirá o SHA parcial para você.

Branch, Remote ou Tag

Você sempre pode usar um branch, remote ou tag ao invés de um SHA, desde que eles sejam de alguma forma ponteiros. Se o seu branch master é o commit 980e3 e você enviou ele para o origin(remoto) e nomeado com tag 'v1.0', então todos os seguintes são equivalentes:

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

Significa que os seguintes comandos darão um resultado idêntico:

```
$ git log master

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

Formato de Datas

O log que o git mantém permitirá a você fazer algumas coisas localmente, como:

```
git log master@{yesterday}

git log master@{1 month ago}
```

No qual é um atalho para 'onde o head do branch master estava ontem', etc. Veja que esse formato pode resultar em diferentes SHAs em diferentes computadores, mesmo se o branch master está atualmente apontando para o mesmo lugar.

Formato Ordinal

Esse formato dará a você o enésimo valor anterior de uma referência particular. Por exemplo:

```
git log master@{5}
```

dará a você o valor do quinto elemento do head master.

Carrot Parent (^)

Isso dará a você o enésimo pai de um commit particular. Esse formato é útil sobre commits criados com merges - objetos commit que possuem mais de um pai direto.

```
git log master^2
```

Formato til (~)

O ~ dará a você o quinto valor anterior do master head. Por exemplo,

```
git log master~2
```

nos dará o primeiro pai do primeiro pai do commit que o master aponta. Isso é equivalente a:

```
git log master^^
```

Você também pode continuar fazendo isso. Os seguintes formatos apontarão para o mesmo commit:

```
git log master^^^^^^
git log master~3^~2
git log master~6
```

Apontador Tree

Isso desambingua um commit da árvore para quem ele aponta. Se você quer o SHA que um commit aponta, você pode adicionar o formato '{tree}' no final dele.

```
git log master^{tree}
```

Formato Blob

Se você quer o SHA de um blob particular, você pode adicionar o caminho do blob no final do treeish, assim:

```
git log master:/path/to/file
```

Range (..)

Finalmente, você pode especificar uma faixa de commits com o formato (..). Isso dará a você todos os commits entre 7b593b5 e 51bea1 (onde 51bea1 é o mais recente), excluindo 7b593b5 mas incluindo 51bea1:

```
git log 7b593b5..51bea1
```

Isso incluirá cada commit *desde* 7b593b:

```
git log 7b593b..
```

TRACKING BRANCHES

Um 'tracking branch' no Git é um branch local que é conectado a um branch remoto. Quando você realiza um push e pull nesse branch, ele automaticamente envia e recupera do branch remoto com quem está conectado.

Use ele se você sempre realiza um pull de um mesmo branch dentro de um novo , e se você não quer usar "git pull " explicitamente.

O comando 'git clone' automaticamente configura um branch 'master' que é um branch associado com 'origin/master' - o branch master sobre um repositório clonado.

Você pode criar um tracking branch manualmente pela adição da opção '--track' sobre o comando branch no Git.

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

Então quando você executar:

```
$ git pull experimental
```

Ele irá automaticamente recuperar do 'origin' e realizará um merge de 'origin/experimental' dentro de seu branch local 'experimental'.

Dessa forma, quando você realizar um push para o origin, ele enviará para o qual seu 'experimental' aponta, sem ter que especificá-lo.

BUSCANDO COM GIT GREP

Encontrar arquivos com palavras ou frases no Git é muito fácil com o comando git grep. Isso é possível ser feito com o comando 'grep' no unix, mas com 'git grep' você também pode procurar através de versões anteriores do projeto sem ter que realizar algum checkout.

Por exemplo, se eu quisesse ver cada lugar que usou a chamada 'xmmap' no meu repositório git.git, eu poderia executar isso:

```
$ 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,
```

Se eu quisesse ver o número de linhas de cada arquivo encontrado, eu posso adicionar a opção '-n':

```
$>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,
```

Se estamos interessados somente no nome do arquivo, podemos passar a opção '--name-only':

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

Nós também poderíamos ver quantas linhas o termo foi encontrado em cada arquivo com a opção '-c':

```
$>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
```

Agora, se eu quisesse ver onde foi usado em uma versão específica do git, eu poderia adicionar uma tag de referência no final, assim:

```
$ 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
```

Nós podemos ver que existem algumas diferenças entre as linhas atuais e as linhas na versão 1.5.0, um dos quais onde o xmmap é agora usado no wrapper.c onde não estava no v1.5.0.

Podemos também combinar os termos de busca no grep. Digamos que queremos procurar onde SORT_DIRENT está definido em nosso repositório:

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

Também podemos procurar por arquivos que possuem *ambos* os termos, mas mostra cada linha que possuem *algum* dos termos nesses arquivos:

```
$ 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)
```

Podemos também procurar por linhas que possuem um dos termos e qualquer um dos dois outros termos, por exemplo, se queremos ver onde tem qualquer uma das constantes definidas PATH ou MAX:

```
$ 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)
...
```

DESFAZENDO NO GIT - RESET, CHECKOUT E REVERT

Git provê múltiplos métodos para corrigir erros quando você está desenvolvendo. Selecionar um método apropriado depende se possui ou não erros nos commits já realizados, se você realizou commits com erros, e se você compartilhou os commits com problemas com alguém.

Corrigindo erros que ainda não foram para o commit

Se você cometeu erros na sua árvore de trabalho, mas ainda não fez o commit desses erros, você pode retornar a árvore de trabalho inteira para o estado do último commit com:

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

Isso descartará qualquer alteração que você possa ter adicionado no index do git e assim como qualquer alteração que você tenha na sua árvore de trabalho. Em outras palavras, isso causa no resultado de "git diff" e "git diff --cached" que sejam ambos vazios.

Se você quer restaurar só um arquivo, digamos seu hello.rb, use git checkout:

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

O primeiro comando restaura hello.rb para a versão no index, para que o "git diff hello.rb" retorne nenhuma direfença. O segundo comando irá restaurar hello.rb da versão no HEAD, para que ambos "git diff hello.rb" e "git diff --cached hello.rb" retornem nenhuma diferença.

Corrigindo erros que foram para o commit

Se você realizou um commit e depois se arrependeu, existem dois caminhos fundamentalmente diferentes para resolver o problema:

1. Você pode criar um novo commit que desfaz qualquer coisa que foi feita pelo commit antigo. Essa é a maneira correta se seu erro já se tornou público.

2. Você pode voltar e modificar o commit antigo. Você nunca deveria fazer isso se você já tornou o histórico público; git normalmente não espera que o "histórico" de um projeto mude, e não pode realizar corretamente merges repetidos de um branch que possue o histórico alterado. Se você reescrever o histórico do repositório, qualquer pessoa que clonou o repositório terá que manualmente corrigir o problema em sua cópia, veja a seção "RECUPERANDO DE UM REBASE REMOTO" em git rebase.

Corrigindo um erro com um novo commit

Criar um novo commit que reverte um alteração mais recente é muito fácil; só passar para o comando git revert a referência para o commit ruim; por exemplo, para reverter o commit mais recente:

```
$ git revert HEAD
```

Isso criará um novo commit que desfaz as modificações no HEAD. Será dado a você a oportunidade de editar a mensagem do commit para o novo commit.

Você pode também reverter uma alteração mais recente, por exemplo, o próximo-para-último:

```
$ git revert HEAD^
```

Nesse caso o git entenderá para desfazer a alteração antiga enquanto mantém intacto qualquer alteração feita desde então. Se alterações mais recentes sobreporem com as alterações para serem revertidas, então você será questionado para corrigir manualmente os conflitos, bem na hora da resolução do merge.

Corrigindo um erro através da modificação de um commit

Se você já realizou o commit de algo mas percebe que precisa consertá-lo, versões recentes do git commit suporta uma flag **--amend** que instrui o git para substituir o commit HEAD com um novo, baseado no conteúdo atual do index. Isso dá a você uma oportunidade para adicionar arquivos que você esqueceu de adicionar ou corrigir a mensagem do commit, antes de enviar as alterações para o mundo ver.

Se você encontrar um erro em um commit antigo, mas ainda um dos que você ainda não publicou para o mundo, você pode usar git rebase em modo interativo, com "git rebase -i" fazendo a alteração que requerem correção com **edit**. Isso permitirá a você juntar o commit durante o processo de rebase.

MANUTENÇÃO NO GIT

Garantindo bom desempenho

Em grandes repositórios, git conta com a compressão para manter as informações do histórico que ocupam muito espaço no disco ou memória.

Essa compressão não é realizado automaticamente. Portanto você deveria executar ocasionalmente git gc:

```
$ git gc
```

para recomprimir o arquivo. Isso pode consumir muito tempo, então você pode preferir executar git-gc quando não estiver trabalhando.

Garantindo a confiabilidade

O comando git fsck executa várias verificações de consistência sobre o repositório, e relata algum problema. Isso pode levar algum tempo. De longe, o aviso mais comum é sobre objetos "dangling":

```
$ 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
...
```

Objetos dangling não são problemas. No pior caso eles podem ocupar um pouco de espaço extra. Eles algumas vezes podem prover um último método para recuperação do trabalho perdido.

CONFIGURANDO UM REPOSITÓRIO PÚBLICO

Assuma que seu repositório pessoal está no diretório ~/proj. Primeiro criamos um novo clone do repositório e pedimos ao git-daemon que ele seja considerado público:

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

O diretório resultante proj.git contém um repositório git "mínimo" -- ele é só o conteúdo do diretório ".git", sem qualquer arquivo dentro dele.

Depois, copie o proj.git para o servidor onde você planeja hospedar o repositório público. Você pode usar scp, rsync, ou qualquer coisa mais conveniente.

Exportando um repositório git via protocolo git

Esse é o método preferido.

Se alguém então administra o servidor, ele deverá pedir a você qual o diretório para colocar o repositório dentro, e qual URL git:// aparecerá nele.

Se não fosse assim, tudo que você precisa para fazer é iniciar git daemon; ele ouvirá a porta 9418. Por padrão, permitirá acessar qualquer diretório que se pareça com um diretório git e contém o arquivo mágico git-daemon-export-ok. Passando alguns caminhos de diretórios como argumentos para git-daemon restringirá mais ainda esses caminhos exportados.

Você pode também executar git-daemon como um serviço inetd; veja as páginas de manual do git daemon para mais detalhes. (Veja especialmente a seção de exemplos.)

Exportando um repositório git via http

O protocolo git dá melhor desempenho e confiabilidade, mas sobre host com um servidor web configurado, exportar via http pode ser mais simples de configurar.

Tudo que você precisa fazer é colocar o recém criado repositório git mínimo no diretório que está exportado pelo web server, e fazer alguns ajustes para dar os clientes webs algumas informações extras que eles precisam:

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

(Para uma explicação das últimas duas linhas, veja git update-server-info e githooks.)

Divulgue a URL do proj.git. Qualquer um então deveria ser capaz de clonar ou baixar dessa URL, por exemplo com a linha de comando:

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

CONFIGURANDO UM REPOSITÓRIO PRIVADO

Se você precisa configurar um repositório privado e quer fazê-lo localmente, em vez de usar uma solução de hospedada, você tem várias opções.

Acesso a repositório através do SSH

Geralmente, a solução mais fácil é simplesmente usar o Git sobre SSH. Se os usuários já possuem contas ssh na máquina, você pode colocar o repositório em qualquer lugar que eles tenham acesso deixando eles acessarem através de logins ssh. Por exemplo, digamos que você tem um repositório que você quer hospedar. Você pode exportá-lo como um repositório mínimo e então enviá-lo para dentro do seu servidor assim:

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

Então alguém pode clonar com uma conta ssh no servidor myserver.com via:

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

Que simplesmente solicitará suas senhas ssh ou usar suas chaves públicas, contanto que eles tenham a autenticação ssh configurada.

Acesso de Múltiplos Usuários usando Gitosis

Se você não quer configurar contas separadas para cada usuário, você pode usar uma ferramente chamada Gitosis. No Gitosis, existe um arquivo authorized_keys que contém uma chave pública para todos os autorizados a acessar o repositório, e então todos usam o usuário 'git' para realizar pushes e pulls.

Instalando e Configurando o Gitosis

Capítulo 6

Git Avançado

CRIANDO NOVOS BRANCHES VAZIOS

Ocasionalmente, você pode querer manter branches em seu repositório que não compartilha um ancestral com o seu código. Alguns exemplos disso podem ser documentações geradas ou alguma coisa nessas linhas. Se você quer criar um novo branch que não usa seu código base atual como pai, você pode criar um branch vazio assim:

```
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

MODIFICANDO SEU HISTÓRICO

Existem diversas maneiras de reescrever o histórico de um repositório. Todos eles podem causar problemas em commits que já tenham sido enviados para um repositório remoto. Se você reescrever o histórico do repositório, qualquer pessoa que clonou o repositório terá que manualmente corrigir o problema em sua cópia, veja a seção "RECUPERANDO DE UM REBASE REMOTO" em git rebase.

Um método simples é usar "git commit --amend" para modificar o último commit. Ele é útil para corrigir uma mensagem do commit, ou fazer uma simples modificação antes de enviá-lo.

Rebase interativo é uma boa maneira de modificar múltiplos commits. Commits podem ser combinados por "squashing", alterando por edição ou removendo completamente.

git filter-branch é uma boa maneira de editar commits em massa. Ele é útil quando um componente inteiro precisa ser removido de um projeto. Por exemplo removendo um sub-sistema que é licenciado sobre uma licença open-source incompatível. Ou ele pode ser usado para alterar o autor do commit sem alterar o código.

BRANCHING E MERGING AVANÇADOS

Conseguindo ajuda na resolução de conflitos durante o merge

Todas as alterações que o merge foi capaz de realizar automaticamente já estão adicionadas no arquivo index, então git diff mostrará somente os conflitos. Ele usa uma sintaxe incomum:

```
$ 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
```

Lembre-se que o commit que será realizado depois que resolvermos esses conflitos terão 2 pais ao invés de um: um pai será o HEAD, a ponta do branch atual; o outro será a ponta do outro branch, que é armazenado temporariamente no MERGE_HEAD.

Durante o merge, o index retém três versões de cada arquivo. Cada um desses três "estágios do arquivo" representam uma versão diferente do arquivo:

```
$ git show :1:file.txt  # o arquivo é o ancestral comum de ambos os branches
$ git show :2:file.txt  # a versão do HEAD.
$ git show :3:file.txt  # a versão do MERGE_HEAD.
```

Quando você pergunta ao git diff para mostrar os conflitos, ele executa um diff de três-passos entre os resultados do merge conflitantes na árvore de trabalho com o estágio 2 e 3 para mostrar somente de qual o conteúdo vem de ambos os lados, misturados (em outras palavras, quando o resultado do merge vem somente do estágio 2, que parte não está conflitando e não é mostrada. O mesmo para o estágio 3).

O diff acima mostra a diferença entre a versão da árvore de trabalho do file.txt e as versões do estágio 2 e estágio 3. Então ao invés de preceder cada linha com um simples "+" ou "-", ele agora usa duas colunas: a primeira coluna é usada para diferenciar entre o primeiro pai e a cópia do diretório de trabalho atual, e o

segundo para diferenciar entre o segungo pai e a cópia do diretório de trabalho. (Veja a seção "COMBINED DIFF FORMAT" do git diff-files para mais detalhes do formato.)

Depois da resolução dos conflitos de maneira óbvia (mas antes de atualizar o index), o diff se parecerá com isso:

```
$ 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
```

Isso mostra que nossa versão corrigida apagou "Hello world" do primeiro pai, apagou "Goodbye" do segundo pai, e adicionou "Goodbye world", que estava ausente de ambos anteriormente.

Algumas opções especiais do diff permitem diferenciar o diretório de trabalho contra qualquer estágio:

```
$ git diff -1 file.txt      # diff contra o estágio 1
$ git diff --base file.txt  # mesmo como acima
$ git diff -2 file.txt      # diff contra o estágio 2
$ git diff --ours file.txt  # mesmo como acima
$ git diff -3 file.txt      # diff contra o estágio 3
$ git diff --theirs file.txt    # mesmo como acima
```

Os comandos git log e gitk também provêm ajuda especial para merges:

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

Isso mostrará todos os commits que existem somente sobre HEAD ou sobre MERGE_HEAD, e qual tocou em um arquivo sem merge.

Você também pode usar git mergetool, que deixa você realizar o merge de arquivos sem merge usando ferramentas externas como emacs ou kdiff3.

Cada vez que você resolve os conflitos dentro do arquivo e atualiza o index:

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

os diferentes estágios daquele arquivo serão "collapsed", depois disso git-diff não mostrará (por padrão) diferenças para aquele arquivo.

Merge Múltiplos

Você pode realizar um merge de diversos heads de uma vez só através da simples listagem deles no comando git merge. Por exemplo,

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

é equivalente a :

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

Subtree

Existem situações onde você quer incluir o conteúdo em seus projetos de um projeto desenvolvido independentemente. Você só realiza um pull do outro projeto contanto que não existam conflitos nos caminhos.

O caso problemático é quando existem arquivos conflitantes. Candidatos potênciais são Makefiles e outros nomes de arquivos padrões. Você poderia realizar um merge desses arquivos mas provavelmente você não vai querer fazê-lo. Uma melhor solução para esse problema pode ser realizar um merge do projeto com o seu próprio sub-diretório. Isso não é suportado pela estratégia de merges recursivos, então realizar pulls não funcionará.

O que você quer é a estratégia de subtrees do merge, que ajuda você nessa situação.

Nesse exemplo, digamos que você tem o repositório em /path/to/B (mas ele pode ser uma URL, se quiser). Você quer realizar o merge do branch master daquele repositório para o sub-diretório dir-B em seu branch atual.

Aqui está a sequência do comando que você precisa:

```
$ 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)
```

O benefício de usar subtree merges é que ele requer menos carga administrativa dos usuários de seu repositório. Isso funciona com clientes antigos (antes de Git v1.5.2) e você possui o código correto depois do clone.

Contudo se você usa sub-módulos então você pode escolher não transferir os objetos do sub-módulo. Isso pode ser um problema com subtree merges.

Também, nesse caso de você fazer alterações para outro projeto, é mais fácil para enviar alterações se você só usa sub-módulos.

(de Using Subtree Merge)

ENCONTRANDO ERROS - GIT BISECT

Suponha uma versão 2.6.18 de seu projeto, mas a versão no "master" está defeituosa. As vezes a melhor forma de encotrar a causa é realizar uma busca usando força-bruta no histórico do projeto para encontrar o commit em particular que causou o problema. O comando git bisect pode ajudar você a fazer isso:

```
$ 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
```

Se você executar "git branch" neste momento, você verá que o git moveu você temporariamente para um novo branch chamado "bisect". Esse branch aponta para um commit (o commit 65934...) que está próximo do "master" mas não do v2.6.18. Compile e teste-o, e veja se possui erro. Assumindo que ele possui erro. Então:

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

vai para uma versão mais antiga. Continua assim, chamando o git em cada estágio se a versão que ele dá a você é boa ou ruim, e avisa que o número de revisões restante para testar é cortado aproximadamente no meio em cada vez.

Depois de 13 testes (nesse caso), ele mostrará o id do commit culpado. Você pode então examinar o commit com git show, encontrar quem escreveu ele, e enviar um email a ele sobre esse bug com o id do commit. Finalmente , execute

```
$ git bisect reset
```

para retorna ao branch onde estava antes e apagar o branch temporário "bisect".

Veja que a versão que git-bisect verifica para você em cada ponto é só uma sugestão, você está livre para tentar uma versão diferente se achar que isso é uma boa idéia. Por exemplo, ocasionalmente você pode cair em um commit que possue um erro não registrado; execute

```
$ git bisect visualize
```

que executará um gitk e marcar o commit escolhido com "bisect". Escolha um commit seguro mais próximo, veja seu id, e mova-se até ele com:

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

então teste, execute "bisect good" ou "bisect bad" de acordo, e continue.

ENCONTRANDO ERROS - GIT BLAME

O comando git blame é realmente útil para entender quem modificou que seção de um arquivo. Se você executar 'git blame [nomedoarquivo]' você conseguirá visualizar o arquivo inteiro com o último SHA do commit, data e autor para cada linha dentro do arquivo.

```
$ 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  2007-12-29 02:05:47 -0800   18) #include "sha1-lookup.h"
...
```

Isso é frequentemente útil se um arquivo possui uma linha revertida ou um erro que o danificou, para ajudar você a ver quem alterou que linha por último.

Você pode também especificar o inicio e o fim da linha para o blame:

```
$>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 E EMAIL

Enviando patches para um projeto

Se você já possui algumas alterações, a forma mais simples de fazê-lo é enviá-los como patches por email:

Primeiro, use git format-patch; por exemplo:

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

produzirá uma série numerada de arquivos no diretório atual, um para cada patch do branch atual, mas não do origin/HEAD.

Você pode então importar eles para seu cliente de email e enviá-los. Contudo, se você tem que enviar todos de uma vez, você pode preferir usar o script git send-email para automatizar o processo. Consulte primeiro a lista de email do seu projeto para determinar como eles preferem que os patches sejam manipulados.

Importando patches para o projeto

Git também provê uma ferramenta chamada git am (uma abreviação de "apply mailbox"), para importar uma série de patches recebidos. Grave todas as mensagens que contém patches, em orderm, para um arquivo mailbox simples, digamos "patches.mbox", então execute

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

Git aplicará cada patch em ordem; se algum conflito for encontrado, ele irá parar, e você pode manualmente corrigir os conflitos. (A opção "-3" informa ao git para realizar um merge; se você prefere só abortar e deixar sua árvore e index intácta, você pode omitir essa opção.)

Uma vez que o index é atualizado com o resultado da resolução do conflito, ao invés de criar um novo commit, execute

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

e o git criará o commit para você e continua aplicando o restante dos patches do mailbox.

O resultado final será uma série de commits, um para cada patch no mailbox original,cada um com o autor e a mensagem de commit trazido da mensagem contida em cada patch.

CUSTOMIZANDO O GIT

git config

Alterando o seu editor

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

Adicionando Aliases

```
$ 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
```

Adicionando Cores

Veja todas as opções de cores na documentação de git config

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

Ou, você pode configurar todos eles com a opção color.ui:

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

Commit Template

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

Log Format

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

Outras Opções de Configuração

Existem também várias opções interessantes para packing, gc-ing, merging, remotes, branches, http transport, diffs, paging, whitespace e mais. Se você quer saber mais dê uma olhada na documentação do git config.

GIT HOOKS

Hooks são pequenos scripts que você pode colocar no diretório GIT_DIR/hooks para disparar um ação em certos pontos. Quando git-init é executado, uns exemplos úteis de hooks são copiados no diretório hooks do novo repositório, mas por padrão eles são todos desativados. Para ativar um hook, renomeie ele removendo o seu sufixo .sample.

applypatch-msg

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

Esse hook é invocado pelo script `git-am`. Ele leva um simples parâmetro, o nome do arquivo que detém a mensagem de commit proposta. Saindo com um status diferente de zero faz com que `git-am` aborte antes de aplicar o patch.

Nesse hook é permitido editar o arquivo de mensagem substituindo-o, e pode ser usado para normalizar a mensagem dentro em algum formato padrão de mensagens (se o projeto tem um). Ele pode também ser usado para recusar o commit depois de inspecionar o arquivo de mensagens. O hook *applypatch-msg* padrão, quando ativado, executa o hook *commit-msg*, se este também estiver ativado.

pre-applypatch

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

Esse hook é invocado pelo `git-am`. Ele não leva nenhum parâmetro, e é invocado depois que o patch é aplicado, mas antes que um commit seja feito. Se ele sai com um status diferente de zero, então não será realizado o commit depois de aplicar esse patch.

Ele pode ser usado para inspecionar a árvore de trabalho atual e recusar realizar um commit se ele não passar em certos testes. O hook *pre-applypatch* padrão, quando ativado, executa o hook *pre-commit*, se este também estiver ativado.

post-applypatch

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

Esse hook é invocado pelo `git-am`. Ele não leva nenhum parâmetro, e é invocado depois que o patch é aplicado e um commit é feito.

Esse hook é usado essencialmente para notificações, e não pode afetar o resultado do `git-am`.

pre-commit

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

Esse hook é invocado pelo `git-commit`, e pode ser ignorado com a opção `\--no-verify`. Ele não leva nenhum parâmetro, e é invocado antes de obter a mensagem do commit proposta e realizar o commit. Saindo com status diferente de zero desse script faz com que `git-commit` seja cancelado.

O hook *pre-commit* padrão, quando ativado, captura o início das linhas com espaços vazios e cancela o commit quando alguma linha é encontrada.

Todos os hooks `git-commit` são invocados com a variável de ambiente `GIT_EDITOR=:` se o commando não carregar um editor para modificar o mensagem de commit.

Aqui é um exemplo de um script Ruby que executa testes RSpec antes de permitir um 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
```

Esse hook é invocado pelo `git-commit` depois de preparar a mensagem de log padrão, e antes de iniciar o editor.

Ele leva de um a três parâmetros. O primeiro é o nome do arquivo da mensagem de commit. O segundo é a origem da mensagem de commit, e pode ser: `message` (se a opção `-m` ou `-F` foi dada); `template` (se a opção `-t` foi dada ou a opção de configuração `commit.template` está configurada); `merge` (se o commit é um merge ou o arquivo `.git/MERGE_MSG` existe); `squash` (se o arquivo `.git/SQUASH_MSG` existe); `commit`, seguido por um id SHA1 (se a opção `-c`, `-C` ou `\--amend` foi dada).

Se o status de saída é diferente de zero, `git-commit` será cancelado.

A proposta desse hook é editar o arquivo de mensagem, e se ele não for anulado pela opção `\--no-verify`. Uma saída diferente de zero significa uma falha nesse hook e cancela o commit. Ele não deveria ser usado como substituto do hook `pre-commit`.

O exemplo do hook `prepare-commit-msg` que vem com o git comenta a parte do `Conflicts:` de uma mensagem de commit durante um merge.

commit-msg

```
GIT_DIR/hooks/commit-msg
```

Esse hook é invocado pelo `git-commit`, e pode ser anulado com a opção `\--no-verify`. Ele leva um simples parâmetro, o nome do arquivo que detém a mensagem de commit proposta. Saindo com status diferente de zero faz com que o `git-commit` aborte.

Com esse hook é permitido editar o arquivo de mensagem, e pode ser usado para normalizar a mensagem em algum formato padrão de mensagens (se o projeto tem um). Ele pode se usado para recusar o commit depois de inspecionar o arquivo de mensagem.

O hook *commit-msg* padrão, quando ativado, detecta duplicadas linhas "Signed-off-by", e cancela o commit se um for encontrada.

post-commit

```
GIT_DIR/hooks/post-commit
```

Esse hook é invocado pelo `git-commit`. Ele não leva nenhum parâmetro, e é invocado depois que o commit é feito.

Esse hook é usado essencialmente para notificações, e não pode afetar o resultado do `git-commit`.

pre-rebase

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

Esse hook é chamado pelo `git-rebase` e pode ser usado para previnir que num branch seja realizado um rebase.

post-checkout

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

Esse hook é invocado quando um `git-checkout` é executado depois de ter atualizado a árvore de trabalho. Para esse hook é dado 3 parâmetros: a referência de um HEAD anterior, a referência para um novo HEAD (que pode ou não ser mudado), e uma flag indicando se o checkout foi um branch checkout (alterando branchs, flag=1) ou o checkout de um arquivo (recuperando um arquivo do index, flag=0). Esse hook não afeta o resultado do `git-checkout`.

Esse hook pode ser usado para realizar checagens de validação no repositório, mostrar as diferenças de um HEAD anterior caso seja diferente, ou configurar as propriedades dos metadados do diretório.

post-merge

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

Esse hook é invocado pelo `git-merge`, que acontece quando um `git-pull` é feito sobre um repositório local. O hook leva um simples parâmetro, um flag de status especificando se é um merge que está sendo feito ou não era um merge do tipo squash. Esse hook não afeta o resultado do `git-merge` e não é executado, se o merge falha devido aos conflitos.

Esse hook pode ser usado em conjunto com o hook *pre-commit* correspondente para salvar e restaurar qualquer forma de metadados associados com a árvore de trabalho (ex.: permissões/donos, ACLS, etc).

pre-receive

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

Esse hook é invocado pelo `git-receive-pack` sobre o repositório remoto, que acontece quando um `git-push` é feito sobre o repositório local. Só antes de iniciar uma atualização das referências sobre o repositório remoto, o hook *pre-receive* é invocado. Seu status de saída determina o sucesso ou falha da atualização.

Esse hook executa uma vez para receber a operação. Ele não leva nenhum argumento, mas para cada referência ser atualizada ele recebe sobre a entrada padrão uma linha no formato:

SP SP LF

onde `<old-value>` é o nome do objeto antigo armazenado na referência, `<new-value>` é o nome do novo objeto para ser armazenado na referência e `<ref-name>` é o nome completo da referência. Quando criar uma nova referência, `<old-value>` é 40 `0`.

Se o hook sai com status diferente de zero, nenhum resultado será atualizado. Se o hook sai com zero, atualizações de referências individuais podem ainda ser previnidos pelo hook *update*.

Ambas as saídas e erros padrões são encaminhadas para `git-send-pack` na outra extremidade, então você pode simplesmente fazer um `echo` nas mensagens para o usuário.

Se você escreveu ele em Ruby, você pode conseguir os argumentos dessa forma:

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

Ou em um script bash, alguma coisa assim funcionaria:

```
#!/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
```

Esse hook é invocado pelo `git-receive-pack` sobre o repositório remoto, que acontece quando um `git-push` é feito sobre um repositório local. Só que antes de atualizar a referência sobre o repositório remoto, o hook *update* é invocado. Seu status de saída determina o sucesso ou falha da atualização da referência.

O hook executa uma vez para cada referência para ser atualizada, e leva 3 parâmetros:

- o nome da referência sendo atualizada,
- o nome do antigo objeto armazenado na referência,
- e o novo nome do objeto que será armazenado na referência.

Um saída zero do hook *update* permite que a referência seja atualizada. Saindo com status diferente de zero previne `git-receive-pack` de atualizar aquela referência.

Esse hook pode ser usado para previnir uma atualização 'forçada' sobre certas referências pela certeza de que o nome do objeto é um objeto commit que é um descedente do objeto commit nomeado pelo antigo nome de objeto. Que é, fazer valer somente uma política de "fast forward".

Ele também poderia ser usado para ver o status dos logs entre antigo..novo. Contudo, ele faz com que não saiba o completo conjunto de branchs, embora ele terminaria enviando um email por referência quando usado ingenuamente. O hook *post-receive* é mais adequado.

Outro uso sugerido na lista de discussão é usar esse hook para implementar controle de acesso que é mais refinado do que um baseado em grupos no sistema de arquivos.

Ambas as saídas e erros padrões são encaminhadas para `git-send-pack` na outra extremidade, então você pode simplesmente fazer um `echo` nas mensagens para o usuário.

O hook *update* padrão, quando ativado--e com a opção `hooks.allowunannotated` ligada--previne tags não definidas sejam enviadas.

post-receive

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

Esse hook é invocado pelo `git-receive-pack` sobre o repositório remoto, que acontece quando um `git-push` é feito sobre um repositório local. Ele executa sobre o repositório remoto uma vez depois de todas as referências tem sido atualizadas.

Esse hook executa uma vez para receber a operação. Ele não leva nenhum argumento , mas consegue a mesma informação quando o hook *pre-receive* faz sobre a sua entrada padrão.

Esse hook não afeta o resultado do `git-receive-pack`, quando ele é chamado depois que o trabalho real é feito.

Ele substitui o hook *post-update* no qual ele consegue ambos antigos e novos valores de todas as referências além de seus nomes.

Ambas as saídas e erros padrões são encaminhadas para `git-send-pack` na outra extremidade, então você pode simplesmente fazer um `echo` nas mensagens para o usuário.

O hook *post-receive* padrão é vazio, mas existe um script de exemplo *post-receive-email* fornecido no diretório `contrib/hooks` na distribuição do Git, que implementa o envio de emails.

post-update

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

Esse hook é invocado pelo `git-receive-pack` sobre o repositório remoto, que acontece quando um `git-push` é feito sobre o repositório local. Ele executa sobre o repositório remoto uma vez depois que todas as referências tem sido atualizadas.

Ele leva um número variável de parâmetros, cada qual é o nome da referência que na verdade foi atualizada.

Esse hook é usado essencialmente para notificações, e não pode afetar o resultado do `git-receive-pack`.

O hook *post-update* pode dizer quais são os heads que foram enviados, mas ele não sabe quais deles são valores original ou atualizados, por isso é um mau lugar para fazer ver os logs entre o antigo..novo. O hook *post-*

receive consegue ambos originais e atualizados valores das referências. Pode ser que você considere ele, por exemplo, se precisar dele.

Quando ativado, o hook *post-update* padrão executa `git-update-server-info` para manter a informação usada no transporte mudo (ex.: HTTP) atualizado. Se você está publicando um repositório git que é acessível via HTTP, você deveria provavelmente ativar esse hook.

Ambas as saídas e erros padrões são encaminhadas para `git-send-pack` na outra extremidade, então você pode simplesmente fazer um `echo` nas mensagens para o usuário.

pre-auto-gc

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

Esse hook é invocado pelo `git-gc --auto`. Ele não leva nenhum parâmetro, e a saída com o status diferente de zero desse script faz com que o `git-gc --auto` seja cancelado.

Referências

Git Hooks
Git hooks make me giddy

RECUPERANDO OBJETOS CORROMPIDOS

Recovering Lost Commits Blog Post

Recovering Corrupted Blobs by Linus

SUBMODULES

Grandes projetos muitas vezes são compostos de pequenos módulos auto-contidos. Por exemplo, uma árvore de código fonte de uma distribuição Linux embarcada incluirá cada software na distribuição com algumas modificações locais; pode ser que um player de filme precise ser construido em cima de uma específica e bem trabalhada versão de uma biblioteca de descompressão; diversos programas independentes podem compartilhar os mesmos scripts de construção.

Isso geralmente é característico em sistemas centralizados de controle de versão por incluir cada módulo em um simples repositório. Desenvolvedores podem baixar todos os módulos ou somente os módulos que eles precisam trabalhar. Eles podem até modificar arquivos pelos diversos módulos em um simples commit enquanto move coisas ou atualiza APIs e traduções.

Git não permite checkouts parciais, então duplicando essa abordagem no Git, forçará aos desenvolvedores manter uma cópia local dos módulos que eles não estão interessados. Commits em grandes checkouts será mais lento do que você poderia esperar com o Git, ele terá que buscar cada diretório por alterações. Se os módulos possuem muito histórico local, clones levarão uma eternidade.

Por outro lado, sistemas de controle de revisão distribuida podem ser muito melhor integrados com fontes externas. Em um modelo centralizado, uma simples cópia arbitrária de um projeto externo é exportado de seu próprio controle de revisão e então importado para o branch do controle de revisão local. Todo o histórico está escondido. Com controle de revisão distribuida você pode clonar o histórico externo inteiro, e muito mais facilmente seguir o desenvolvimento e realizar o re-merge das alterações locais.

O suporte a submodules no Git permite um repositório conter, como um sub-diretório, uma cópia de um projeto externo. Submodules mantém sua própria identidade; o suporte a submodule só armazena a localização do repositório do submodule e a identificação do commit, então outros desenvolvedores que clonarem o conteúdo do projeto ("superproject") podem facilmente clonar todos os submodules na mesma revisão. Checkouts parciais do superproject são possíveis: você pode chamar o Git para clonar nenhum, alguns, ou todos os submodules.

O comando git submodule está disponível desde o Git 1.5.3. Usuários com Git 1.5.2 podem procurar os commits do submodule no repositório e manualmente mover-se para eles; versões mais antigas não reconhecerão os submodules.

Para ver como o suporte a submodule funciona, crie (por exemplo) quatro repositórios de exemplo que podem ser usados depois como submodule:

```
$ 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
```

Agora crie um superproject e adicione todos os submodules:

```
$ mkdir super
$ cd super
$ git init
```

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

NOTA: Não use URLs locais aqui se você planeja publicar seu superproject!

Veja que arquivos `git-submodule` criou:

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

O comando `git-submodule` faz várias coisas:

- Ele clona o submodule sobre o diretório atual e por padrão troca para o branch master.
- Ele adiciona o caminho do clone do submodule para o arquivo gitmodules e adiciona esse arquivo no index, pronto para o commit.
- Ele adiciona a ID do commit atual do submodule no index, pronto para o commit.

Commit o superproject:

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

Agora clone o superproject:

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

Os diretórios do submodule existem, mas estão vazios:

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

NOTA: Os nomes dos objetos commit mostrado acima serão diferentes para você, mas eles deverão corresponder aos nomes dos objetos commit do HEAD em seu repositório. Você pode verificar ele executando `git ls-remote ../git/a`.

Realizar um pull dos submodules é um processo de dois passos. Primeiro execute `git submodule init` para adicionar a URL do repositório submodule para `.git/config`:

```
$ git submodule init
```

Agora use `git-submodule update` para clonar o repositório e verificar os commits especificados no superproject:

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

Uma das maiores diferenças entre `git-submodule update` e `git-submodule add` é que `git-submodule update` verifica um commit específico, ou melhor o branch atual. Isso é como mover-se para uma tag: o head é isolado, então você não trabalha sobre o branch.

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

Se você quer fazer uma alteração dentro de um submodule e você tem um head isolado, então você deverá criar ou mudar para um branch, fazer suas alterações, publicar a alteração dentro do submodule, e então atualizar o superprojetct para referenciar o novo commit:

```
$ git checkout master
```

ou

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

então:

```
$ 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
```

Você tem que executar `git submodule update` depois `git pull` se você também quer atualizar os submodules.

Armadilhas com submodules

Sempre publique a alteração do submodule antes de publicar as alterações para o superproject que referencia ele. Se você esquecer de publicar as alterações do submodule, outros não serão capazer de clonar o repositório.

```
$ 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'
```

Se você está selecionando um submodule para realizar um commit manualmente, tenha cuidado para não esquecer as barras quando especificar o path. Com as barras adicionadas, Git assumirá que você está removendo o submodule e verificando que o conteúdo do diretório contém um repositório.

```
$ 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
#
```

Para corrigir o index depois de realizar dessa operação, reset as modificações e então adicione o submodule sem a barra.

```
$ 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
#
```

Você também não deveria voltar branches em um submodule além de commits que sempre foram gravados em algum superproject.

Não é seguro executar `git submodule update` se você tem feito e realizado commit das alterações dentro do submodule sem verificar o branch primeiro. Eles serão sobrescritos silenciosamente:

```
$ 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
```

NOTA: As alterações ainda são visíveis no reflog dos submodules.

Isso não é o caso se você não realizou o commit de suas alterações.

gitcast:c11-git-submodules

Capítulo 7

Trabalhando com Git

GIT NO WINDOWS

(mSysGit)

gitcast:c10-windows-git

DEPLOYING COM GIT

Capistrano and Git

GitHub Guide on Deploying with Cap

Git and Capistrano Screencast

INTEGRAÇÃO COM SUBVERSION

MIGRAÇÃO DE UM SCM

Então você tomou a decisão de mudar de seu sistema atual e converter todo o seu projeto para o Git. Como você pode fazer isso facilmente?

Importando do Subversion

Git vem com um script chamado git-svn que tem um comando clone que importará um repositório subversion dentro de um novo repositório git. Existe também uma ferramenta grátis no GitHub que pode fazer isso para você.

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

Isso dará a você um novo repositório Git com todo o histórico do repositório Subversion original. Isso levará um bom tempo, geralmente, desde que ele inície com a versão 1 e checkouts e commits localmente a cada simples revisão um por um.

Importando do Perforce

Em contrib/fast-import você encontrará o script git-p4, que é um script em Python que importará um repositório Perforce para você.

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

Importando Outros

Existem outros SCMs que são listados no Git Survey, deveria encontrar a documentação de importação deles. !!A FAZER!!

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

GIT GRÁFICO

Git tem algumas bastante populares GUIs (Graphial User Interfaces) que podem ler e/ou manipular repositórios Git.

Bundled GUIs

Git vez com dois dos principais programas GUI escrito em Tcl/Tk. Gitk é uma ferramente de navegação de repositório e visualização do histórico de commits.

gitk

git gui é uma ferramenta que ajuda você a visualizar as operações no index, como add, remove e commit. Ele não fará tudo que pode fazer na linha de comando, mas para muitas operações básicas, ele é muito bom.

git gui

Terceira Parte Projects

Para usuários Mac existem GitX and GitNub

Para usuários Linux ou Qt existe QGit

HOSPEDAGEM GIT

github

repoorcz

USOS ALTERNATIVOS

ContentDistribution

TicGit

SCRIPTING E GIT

Ruby e Git

grit

jgit + jruby

PHP e Git

Python e Git

pygit

Perl e Git

perlgit

GIT E EDITORS

textmate

eclipse

netbeans

Capítulo 8

Internals and Plumbing

COMO O GIT ARMAZENA OBJETOS

Esse capítulo mostra em detalhes como o Git fisicamente armazenda os objetos.

Todos os objetos são armazenados pela compressão do conteúdo de acordo com os seus valores sha. Eles contém o tipo de objeto, tamanho e conteúdo no formato gzip.

Existem dois formatos que o Git mantém os objetos - objetos loose e packed.

Objetos Loose

Objetos loose é o formato mais simples. Ele é simplesmente a compressão dos dados armazenados em um simples arquivo no disco. Cada objeto é escrito em um arquivo separado.

Se o sha do seu objeto é `ab04d884140f7b0cf8bbf86d6883869f16a46f65`, então o arquivo será armazenado com o seguinte caminho:

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

Ele retira os dois primeiros caracteres e usa-o como sub-diretório, para que nunca exista muitos objetos em um diretório. O nome do arquivo na verdade é o restante dos 38 caracteres.

A forma mais fácil de descrever exatamente como os dados do objeto são armazenados é essa implementação em Ruby do armazenamento do objeto:

```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
```

Objetos Packed

O outro formato para o armazenamento de objetos é o packfile. Visto que o Git armazena cada versão do arquivo em um objeto separado, isso pode ser bastante ineficiente. Imagine tendo um arquivo com milhares de linhas e então altera uma simples linha. Git armazenará o segundo arquivo inteiramente nele, que é um grande desperdício de espaço.

Segundo as regras para a economia de espaço, Git utiliza o packfile. Esse é um formato onde o Git somente gravará a parte que foi alterada no segundo arquivo, com um apontador para o arquivo original.

Quando os objetos são escritos no disco, frequentemente é no formato loose, desde que o formato seja menos dispendioso para acessar. Contudo, finalmente você irá querer economizar espaço através do empacotamento dos objetos - isso é feito com o comando git gc. Ele usará uma heurística bastante complicada para determinar quais arquivos são provavelmente mais semelhantes e e basear os deltas dessa análise. Podem ser múltiplos packfiles, eles podem ser re-empacotados se necessário (git repack) ou desempacotados de volta em arquivos loose (git unpack-objects) com relativa facilidade.

Git também escreverá um arquivo index para cada packfile que é muito menor e contém o deslocamento dentro do packfile para rapidamente encontrar objetos específicos através do sha.

Os detalhes exatos da implementação do packfile são encontrados no capítulo Packfile um pouco mais tarde.

NAVEGANDO NOS OBJETOS GIT

Podemos perguntar ao git sobre objetos particulares com o comando cat-file. Veja que você pode encurtar os shas em somente alguns caracteres para economizar a digitação de todos os 40 digitos hexadecimais.

```
$ 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
```

Uma árvore pode referenciar um ou mais objetos "blob", cada um correspondendo a um arquivo. Além disso, uma árvore pode também referenciar para outros objetos tree, desta maneira criando uma hierarquia de diretórios. Você pode examinar o conteúdo de qualquer árvore usando ls-tree (lembre-se que uma porção inicial suficiente do SHA1 também funcionará):

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

Desse form vemos que a tree possui um arquivo dentro dela. O hash SHA1 é uma referência para aqueles arquivos de dados:

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

Um "blob" é só um arquivo de dados, que também podemos examinar com cat-file:

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

Veja que esse é um arquivo de dados antigo; então o objeto que o git nomeou dele corresponde a árvore inicial que era uma tree com o estado do diretório que foi gravado pelo primeiro commit.

Todos esses objetos são armazenados sobre seus nomes SHA1 dentro do diretório do git:

```
$ 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
```

e o conteúdo desses arquivos é só a compressão dos dados mais o header identificando seu tamanho e seu tipo. O tipo é qualquer entre blob, tree, commit ou tag.

O commit mais simples que encontra é o commit HEAD, que podemos encontrar no .git/HEAD:

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

Como você pode ver, isso nos diz qual branch estamos atualmente, e nos diz o caminho completo do arquivo sobre o diretório .git, que nele mesmo contém o nome SHA1 referindo a um objeto commit, que podemos examinar com cat-file:

```
$ 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
```

O objeto "tree" aqui se refere ao novo estado da tree:

```
$ git ls-tree d0492b36
100644 blob a0423896973644771497bdc03eb99d5281615b51    file.txt
$ git cat-file blob a0423896
hello world!
```

e o objeto "pai" se refere a um commit anterior:

```
$ 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
```

REFERÊNCIAS GIT

Branches, remote-tracking branches, e tags são todos referências para commits. Todas as referências são nomeadas, com o nome do caminho separado por barra "/" iniciando com "refs"; os nomes que usávamos até agora são na verdade atalhos:

```
- O branch "test" é abreviado de "refs/heads/test".
- A tag "v2.6.18" é abreviado de "refs/tags/v2.6.18".
- "origin/master" é abreviado de "refs/remotes/origin/master".
```

O nome completo é ocasionalmente útil se, por exemplo, se existe uma tag e um branch com o mesmo nome.

(refs recém criadas são na verdade armazenadas no diretório .git/refs, sobre o caminho formado pelo seu nome. Contudo, por razões de eficiência eles podem também ser empacotados juntos em um simples arquivo; veja git pack-refs).

Um outro atalho útil, o "HEAD" de um repositório pode ser referenciado para usar somente o nome daquele repositório. Então, por exemplo, "origin" é normalmente um atalho para o branch HEAD no repositório "origin".

Para completar a lista de caminhos que o git verifica pelas referências, e a ordem que ele usa para decidir qual escolher quando existem múltiplas referências com o mesmo atalho, veja a seção "SPECIFYING REVISIONS" do git rev-parse.

Mostrando commits únicos de um dado branch

Suponha que você gostaria de ver todos os commits alcançáveis do branch head chamado "master" mas não de qualquer outro head no seu repositório.

Podemos listar todos os heads nesse repositório com git show-ref:

```
$ 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
```

Podemos conseguir só os nomes do branch, e remover "master", com a ajuda dos utilitários padrões cut e grep:

```
$ 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
```

E então podemos ver todos os commits alcaçáveis do master mas não desse outros heads:

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

Obviamente, intermináveis variações são possíveis; por exemplo, para ver todos os commits alcançáveis de algum head mas não de qualquer tag no repositório:

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

(Veja git rev-parse para explicações da sintaxe de commit-selecting como por exemplo `--not`.)

(!!update-ref!!)

O INDEX DO GIT

O index é um arquivo binário (geralmente mantido em .git/index) contém uma lista ordenada de caminhos, cada um com permissões e o SHA1 de um objeto blob; git ls-files pode mostrar a você o conteúdo do index:

```
$ 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
```

Veja que em uma documentação mais antiga você pode ver o index ser chamado de "cache do diretório atual" ou só "cache". Ele possui três propriedades importantes:

1. O index contém todas as informações necessárias para gerar um simples (unicamente determinado) objeto tree.

   Por exemplo, executando git commit gera esse objeto tree do index, armazena ele no banco de dados de objetos, e usa ele como o objeto tree associado com o novo commit.

2. O index habilita rápidas comparações entre o objeto tree e a árvore de trabalho.

   Ele faz isso através do armazenamento de algum dado adicional para cada entrada (por exemplo a última hora modificada). Esse dado não é mostrado acima, e não está armazenado no objeto tree criado, mas ele pode ser usado para determinar rapidamente quais arquivos no diretório de trabalho diferem de qual foi armazenado no index, e dessa maneira economizar o git de ter que ler todos os dados de cada arquivo em busca de alterações.

3. Ele pode eficientemente representar informações sobre os conflitos de merge entre diferentes objetos tree, permitindo cada caminho ser associado com informação suficiente sobre as trees envolvidas que você pode criar um merge de três-passos entre eles.

   Durante um merge o index pode armazenar múltiplas versões de um simples arquivo (chamados de "estágios"). A terceira coluna na saída do git ls-files acima, é o número do estágio, e aceitará valores exceto 0 para arquivo com conflidos de merge.

O index é dessa maneira uma área ordenada de estágios temporários, que é preenchido com uma tree no qual você está, no processo de trabalho.

O PACKFILE

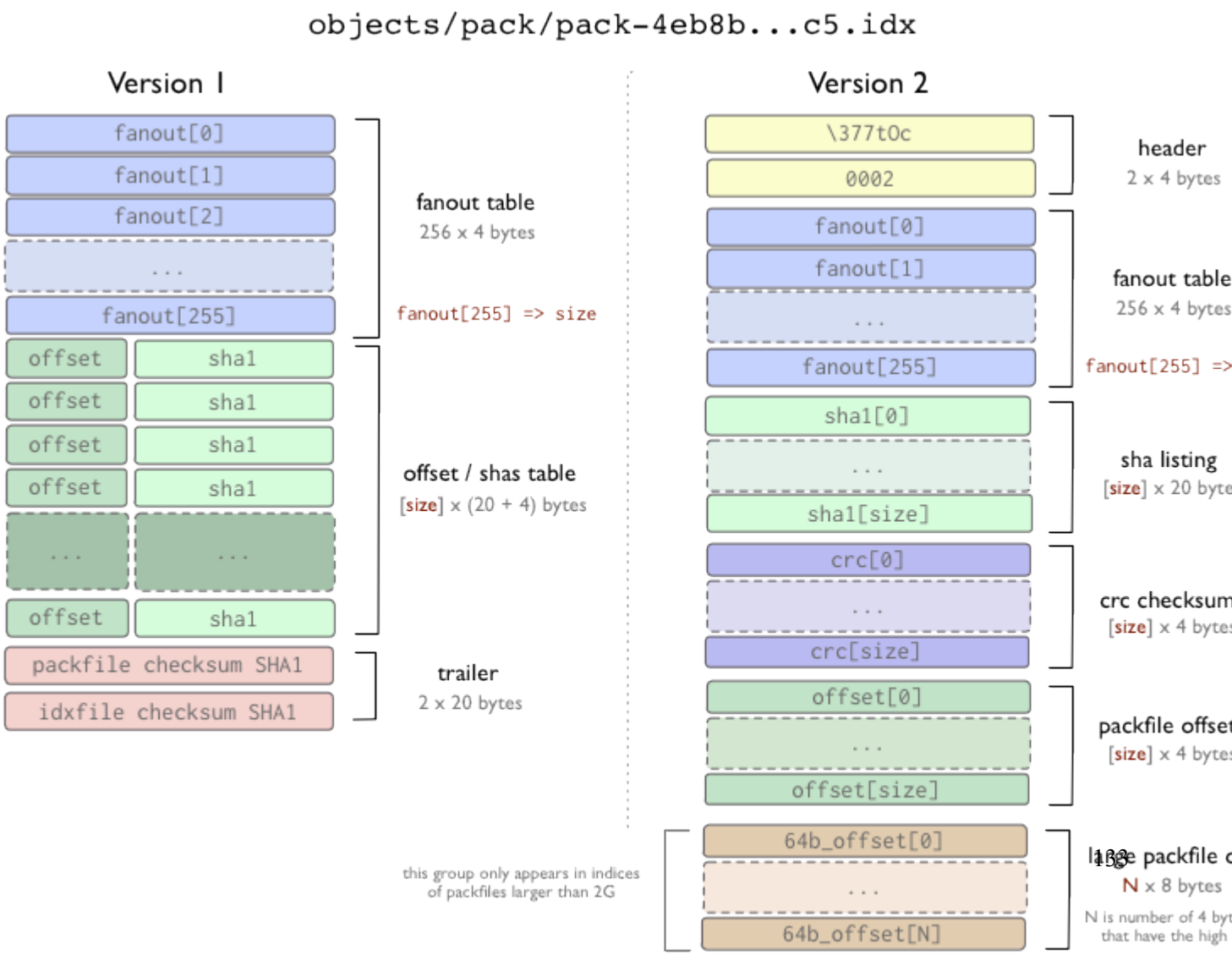
Esse capítulo explica em detalhes, a nível de bits, como os arquivos packfile e o pack index são formatados.

O Index do Packfile

Primeiro, nós temos o index do packfile, que é basicamente só uma série de bookmarks dentro do packfile.

Existem duas versões de index do packfile - versão um, que é a padrão nas versões anteriores do Git 1.6, e a versão dois, que é o padrão a partir da 1.6, mas que pode ser lida pelas versões do Git de volta a 1.5.2, e tem sido implementado de volta para a 1.4.4.5 se você está usando sobre a série 1.4.

Versão 2 também inclui uma checagem de CRC de cada objeto então dados comprimidos podem ser copiados diretamente de um packfile para outro durante o re-empacotamento sem precisar detectar corrupção de dados. Indexes versão 2 também podem manipular packfiles maiores que 4GB.


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

Em ambos formatos, a tabela fanout é simplesmente uma forma de encontrar o deslocamento de um sha particular mais rápido dentro do arquivo index. As tabelas de offset/sha1[] são ordenados por valores sha1[] (isso permite busca binária nessa tabela), e a tabela fanout[] aponta para a tabela offset/sha1[] de uma forma específica (para que parte da última tabela que cobre todos os hashes que iniciam com o byte dado pode ser encontrado para evitar 8 interações da busca binária).

Na versão 1, os offsets e shas estão no mesmo espaço, na versão 2, existem tabelas separadas para shas, CRCs e offsets. No final de ambos os arquivos estão as shas para ambos os arquivos de index e packfile que ele referencia.

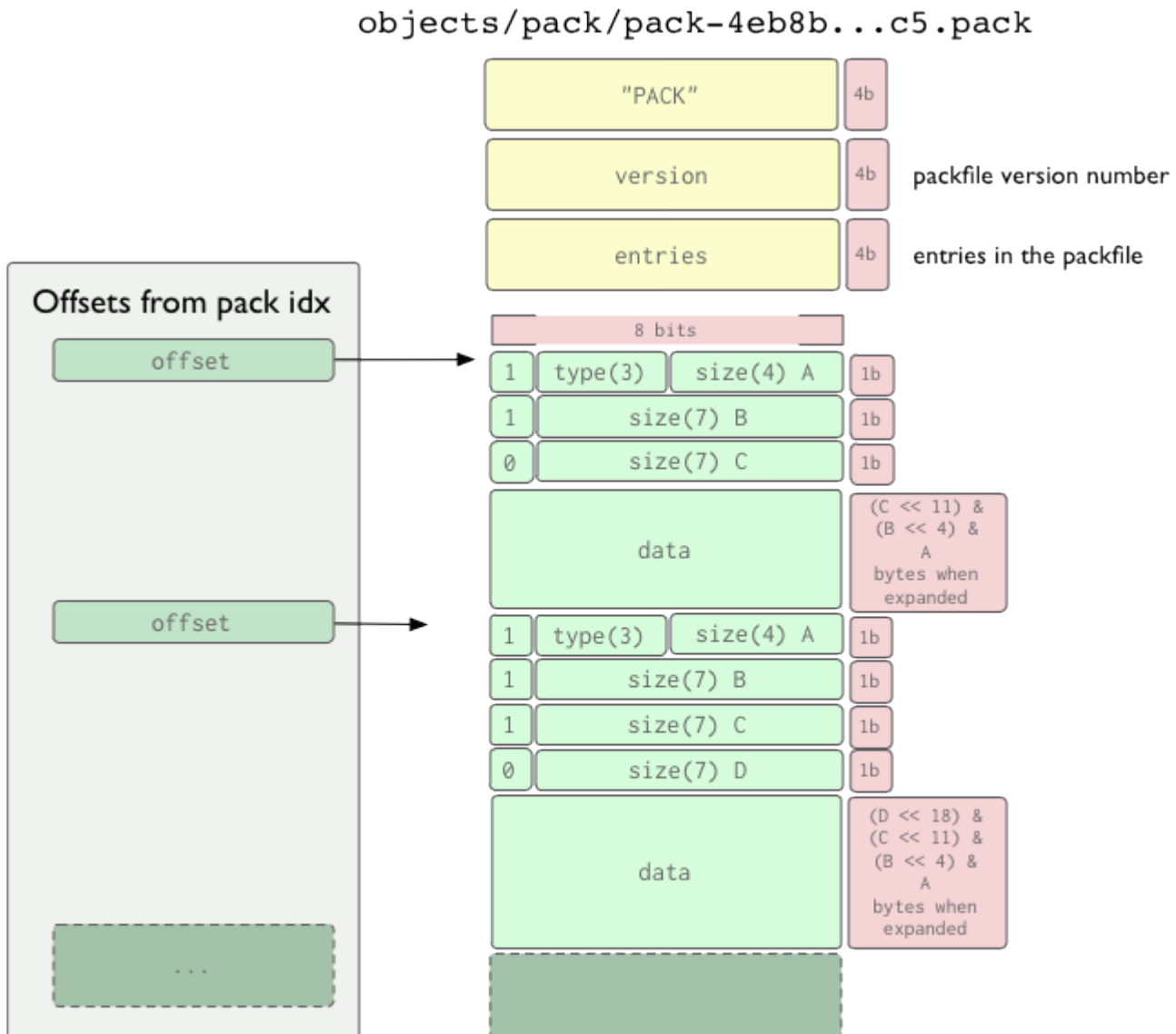
Importante, indexes de packfile *não* são necessários para extrair objetos de um packfile, eles são simplesmente usados para acessar *rapidamente* objetos individuais de um pacote.

O formato do Packfile

O packfile em si é um formato muito simples. Existe um cabeçalho, uma série de pacotes de objetos (cada um com o seu próprio cabeçalho e corpo) e o checksum. Os primeiros quatro bytes é a string 'PACK', que dá um pouco de certeza que você está conseguindo o início de um packfile corretamente. Isso é seguido por um número de versão de 4 bytes do packfile e então um número de 4 bytes de entrada nesse arquivo. Em Ruby, você pode ler os dados do cabeçalho assim:

```
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
```

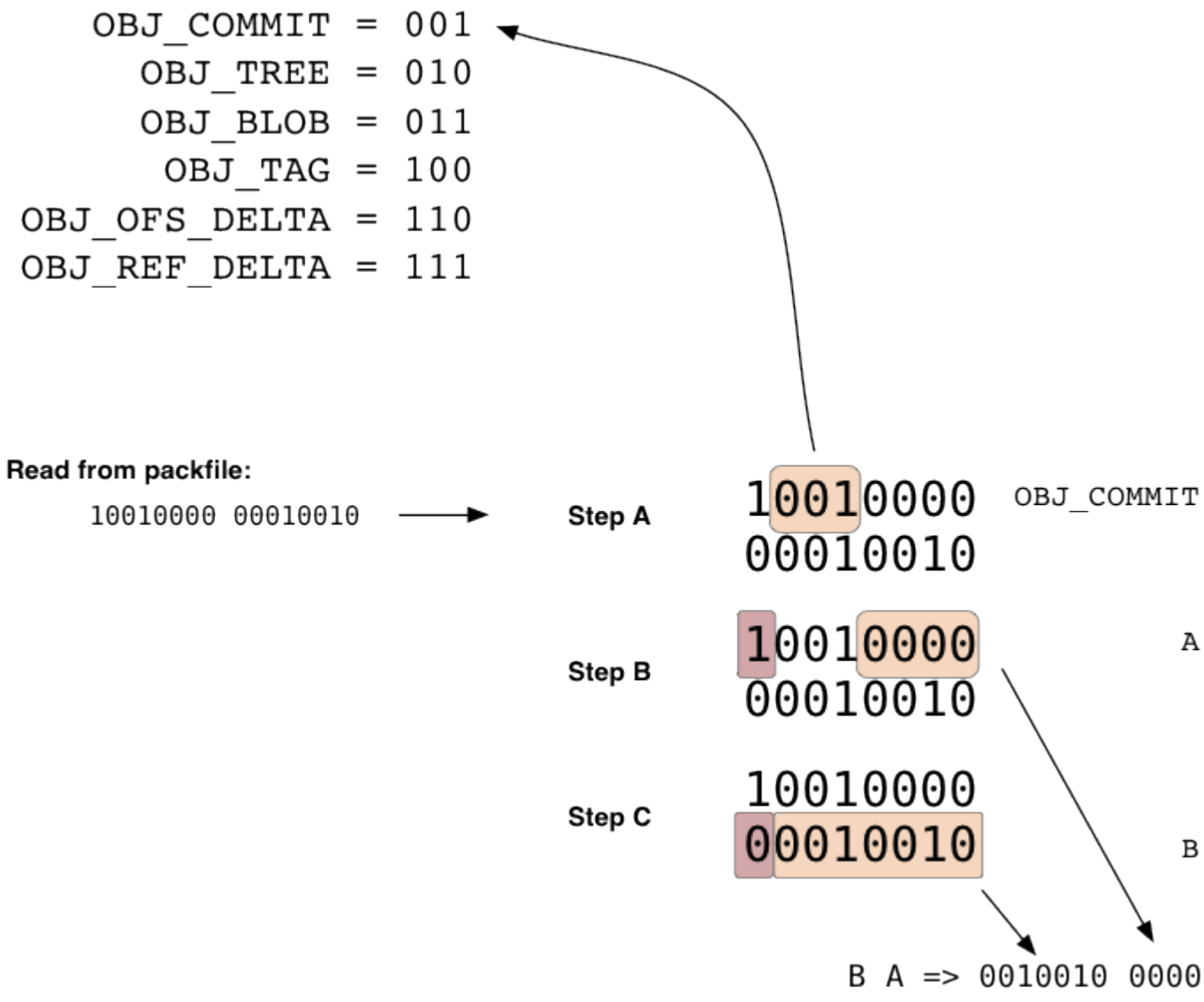
Depois que, você consegue uma série de objetos empacotados, na ordem de seus SHAs que cada um consiste de um object header e object contents. No final do packfile é uma soma SHA1 de 20 bytes de todos os SHAs (ordenados) naquele packfile.


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

O object header é uma série de um ou mais bytes (8 bits) que especifica o tipo do objeto de acordo como os dados são, com o primeiro bit existente dizendo se aquele conjunto é o último ou não antes do início dos dados. Se o primeiro bit é o 1, você lerá outro byte, senão os dados iniciariam logo depois. Os 3 primeiros bits no primeiro byte especifica o tipo do dado, de acordo com a tabela a seguir.

(Atualmente, dos 8 valores que podem ser expressados com 3 bits (0-7), 0 (000) é 'não definido' e 5 (101) é não usado ainda.)

Aqui, nós podemos ver um exemplo do cabeçalho de dois bytes, onde o primeiro especifica que o seguinte dado é um commit, e o restante do primeiro e os últimos 7 bits do segundo especifica que os dados terão 144 bytes quando expandidos.


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

Isso é importante notar que o tamanho especificado no cabeçalho não é o tamanho dos dados que na verdade segue, mas o tamanho do dado *quando expandido*. Isso é porque os offsets no index do packfile são tão úteis, senão você tem que expandir cada objeto só para dizer quando o próximo cabeçalho inicia.

A parte de dados é só uma stream zlib para tipos de objetos não-delta; para as duas representações de objetos delta, a porção de dados contém algo que identifica que objeto base essa representação do delta depende, e o delta para aplicar sobre o objeto base para ressucitar esse objeto. `ref-delta` usa um hash de 20 bytes do objeto base no início dos dados, enquanto `ofs-delta` armazena um offset dentro do mesmo packfile para identificar o objeto base. Em qualquer caso, duas importantes restrições devem ser aderidas:

delta to apply on the base object to resurrect this object. `ref-delta` uses 20-byte hash of the base object at the beginning of data, while `ofs-delta` stores an offset within the same packfile to identify the base object. In either case, two important constraints a reimplementor must adhere to are:

- representação delta deve ser baseado em algum outro objeto dentro do mesmo packfile;
- o objeto base deve ser o mesmo tipo subjacebte (blob, tree, commit ou tag);

RAW GIT

Aqui daremos uma olhada em como manipular o git em baixo nível, se você por acaso gostaria de escrever uma ferramenta que gere novos blobs, trees ou commits de uma forma mais artificial. Se você quer escrever um script que usa a estrutura de mais baixo nível do git para fazer algo novo, aqui estão algumas das ferramentas que você precisará.

Criando Blobs

Criar um blob no seu repositório Git e conseguir o SHA é muito fácil. O comando git hash-object é tudo que você precisará. Para criar um objeto blob de um arquivo existente, só executar ele com a opção '-w' (que dirá a ele para escrever o blob, não somente calcular o SHA).

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

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

A saída STDOUT do comando mostrará o SHA do blob que foi criado.

Criando Trees

Agora digamos que você quer criar um tree de seus novos objetos. O comando git mktree faz isso de maneira muito simples para gerar novos objetos tree apartir da saída formatada de git ls-tree. Por exemplo, se você escrever o seguinte para o arquivo chamado '/tmp/tree.txt':

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

e então interligar eles através do commando git mktree, o Git escreverá uma nova tree no banco de dados de objeto e devolverá o novo sha daquela tree.

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

Então, podemos pegá-lo e fazê-lo um sub-diretório de uma outra tree, e assim em diante. Se quisermos criar uma nova tree com um deles sendo uma subtree, somente criamos um novo arquivo (/tmp/newtree.txt) com o nosso novo SHA com a tree dentro dele:

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

e então usar git mk-tree novamente:

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

E agora temos uma estrutura de diretório artificial no Git se parece com isso:

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

1 diretório, 3 arquivos
```

sem que a estrutura, tenha na verdade, existido no disco. Mas, nós temos um SHA (`5bac6559`) que aponta para ele.

Rearranjando Trees

Podemos também fazer manipulação de tree através da combinação de trees em novas estruturas usando o arquivo index. Como um exemplo simples, pegamos a tree que nós já criamos e fazemos uma nova tree que

tem duas cópias de nossa tree `5bac6559` dentro dela, usando um arquivo index temporário. (Você pode fazer isso através de um reset na variável de ambiente GIT_INDEX_FILE ou na linha de comando)

Primeiro, lemos a tree dentro de nosso arquivo index sobre um novo prefixo usando o comando git read-tree, e então escrever o conteúdo do index como uma tree usando o comando git write-tree:

```
$ 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
```

Então agora podemos ver que temos criado uma nova tree somente com a manipulação do index. Você também pode fazer operações interessantes de merge e de um index temporário dessa forma - veja a documentação do git read-tree para mais informações.

Criando Commits

Agora que temos um SHA de uma tree, podemos criar um objeto commit que aponta para ele. Podemos fazer isso usando o comando git commit-tree. Muitos dos dados que vai dentro do commit tem que ser configurado como variáveis de ambiente, então você gostaria de configurar as seguintes:

```
GIT_AUTHOR_NAME
GIT_AUTHOR_EMAIL
GIT_AUTHOR_DATE
GIT_COMMITTER_NAME
```

```
GIT_COMMITTER_EMAIL
GIT_COMMITTER_DATE
```

Então você precisará escrever sua mensagem de commit para um arquivo ou de alguma forma enviá-lo para o comando através do STDIN. Então, você pode criar seu objeto commit baseado na SHA da tree que temos.

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

Se você quer especificar um ou mais commits pais, simplesmente adicione os SHAs na linha de comando com a opção '-p' antes de cada um. O SHA do novo objeto commit será retornado vi STDOUT.

Atualizando o Branch Ref

Agora que temos um novo SHA do objeto commit, podemos atualizar um branch para apontar para ele se quisermos. Digamos que queremos atualizar nosso branch 'master' para apontar para um novo commit que já criamos - usaríamos o comando git update-ref:

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

PROTOCOLOS DE TRANSFERÊNCIA

Aqui vamos examinar como os clientes e servidores falam um com o outro para transferir todos os dados do Git.

Recuperando Dados sobre HTTP

Recuperar sobre URL http/s fará o Git usar um protocolo ligeiramente simples. Nesse caso, todo a lógica está inteiramente no lado do cliente. Não requer nenhuma configuração especial no servidor - qualquer webserver estático funcionará bem se o diretório do git que você está recuperando está no caminho do webserver.

Segundo as regras, para isso funcionar você precisa excutar um simples comando sobre o repositório do servidor cada vez que alguma coisa for atualizada, mesmo assim - git update-server-info, que atualiza os arquivos objects/info/packs e info/refs para listar quais refs e packfiles estão disponíveis, desde que não possa realizar uma listagem sobre http. Quando o comando executa, o arquivo objects/info/packs se parece como algo assim:

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

Para que, se o fetch não poder encontrar um arquivo loose, ele pode tentar esses packfiles. O arquivo info/refs se parecerá assim:

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

Então quando você recuperar desse repositório, ele iniciará com esses refs e percorrerá os objetos commits até o cliente ter todos os objetos que ele precisa.

Por exemplo, se você pedir para recuperar o branch master, ele verá que o master está apontando para `32aae7ae` e que o seu master está apontando para `ab04d88`, então você precisa do `32aae7ae`. Você recupera aquele objeto

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

e ele se parecerá com isso:
and it looks like this:

```
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
```

Então agora ele recupera a tree `aa176fb8`:

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

que se parecerá com isso:

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

Então ele recupera aqueles objetos:

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

Ele na verdade faz isso com Curl, e pode abrir múltiplos threads paralelos para aumentar a velocidade desse processo. Quando ele termina recursando a tree apontado pelo commit, ele recupera o próximo pai.

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

Agora nesse caso, o commit que chega se parece com isso:

```
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
```

e podemos ver que o pai, `ab04d88` é onde nosso branch master está atualmente apontando. Então, recursivamente recuperamos essa tree e então para, desde que sabemos que temos tudo antes desse ponto. Você pode forçar o Git para realizar uma dupla checagem do que temos com a opção '--recover'. Veja git http-fetch para mais informações.

Se a recuperação de um dos objetos loose falha, o Git baixa o índice do packfile procurando pelo sha que ele precisa, então baixa esse packfile.

Isso é importante se você está executando um servidor git que serve repositórios desta forma para implementar um hook post-receive que executará o comando 'git update-server-info' cada vez ou haverá confusão.

Recuperando Dados com Upload Pack

Para protocolos espertos, recuperar objetos é muito mais eficiente. Um socket é aberto, também sobre ssh ou porta 9418 (nesse caso o protocolo git://), e o comando git fetch-pack no cliente inicia a comunicação com um processo filho no servidor do git upload-pack.

Então o servidor pedirá ao clente que SHAs ele tem para cada ref, e o cliente entende que ele precisa e responde com a lista de SHAs que ele quer e já possue.

Ness ponto, o servidor gerará um packfile com todos os objetos que o cliente precisa e iniciando a transferência para o cliente.

Vamos dar uma olhada em um exemplo.

O cliente conecta e enviar um cabeçalho de requisição. O comando clone

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

produz a seguinte requisição:

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

Os primeiros quatros bytes contém o tamanho em hexadecimal da linha (incluindo os 4 bytes e removendo caracter de nova linha se existir). Seguindo estão o comando e os argumentos. Isso é seguido por um byte null e então a informação do host. A requisição é terminada por um byte null.

A requisição é processada e transformado em uma chamada para git-upload-pack:

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

Isso imediatamente retorna a informação do repositório:

```
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
```

Cada linha inicia com uma declaração em hexadecimal de quatro bytes. A seção é terminada por uma declaração de 0000.

Isso é enviado de volta para o cliente textualmente. O cliente responde com outra requisição:

```
0054want 74730d410fcb6603ace96f1dc55ea6196122532d multi_ack side-band-64k ofs-delta
0032want 7d1665144a3a975c05f1f43902ddaf084e784dbe
0032want 5a3f6be755bbb7deae50065988cbfa1ffa9ab68a
0032want 7e47fe2bd8d01d481f44d7af0531bd93d3b21c01
0032want 74730d410fcb6603ace96f1dc55ea6196122532d
00000009done
```

É enviado para abrir o processo git-upload-pack que então envia a reposta final:

```
"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"
```

Veja o capítulo anteriormente sobre Packfile para o formato real dos dados do packfile nessa resposta.

Enviando Dados

Enviar dados sobre os protocolos git ou ssh são similares, mas simples. Basicamente o que acontece é o cliente requisitar uma instância de receive-pack, que está iniciado se o cliente tem acesso, então o servidor retorna todos os SHAs do ref heads dele tem novamente o cliente gera o packfile de tudo que o servidor precisa(geralmente somente se o que está no servidor é um ancestral direto do que é enviado) e envia esse fluxo do packfile, onde o servidor também armazena ele no disco e constroi um index para ele, ou desempacota ele (se não existe muitos objetos nele)

Esse proceso inteiro é realizado através do comando git send-pack no cliente, que é invocado pelo comando git push e o git receive-pack no lado do servidor, que é invocado pelo processo de conexão ssh ou daemon git (se ele é um servidor aberto para envio)

Capítulo 9

Glossário

GLOSSÁRIO

Aqui nós temos o significado de alguns termos usados dentro do contexto do Git. Esses termos foram retirados do Git Glossary.

alternate object database

> Através de mecanismos alternativos, um repositório pode herdar parte de seus objetos do banco de dados de outro banco de dados de objetos, que é chamado de "alternate".

bare repository

Um repositório bare é normalmente um nome apropriado para um diretório com um sufixo `.git` que não possui um cópia (checkout) local de qualquer arquivo sobre o controle de revisão. Então, todos os arquivos administrativos e de controle do `git` que normalmente estariam no sub-diretório escondido `.git` estão presentes diretamente no diretório `repository.git` por exemplo, e nenhum outro arquivo presente ou checkout. Normalmente publicadores de repositórios públicos tornam diretórios bare disponíveis.

objeto blob

Objeto sem tipo, o conteúdo de um arquivo.

branch

Um "branch" é uma linha ativa de desenvolvimento. O commit mais recente sobre um branch é referenciado como o ponto mais alto naquele branch. Esse ponto é refenciado por um HEAD, no qual é movido para cima quando algum desenvolvimento adicional é feito sobre ele. Um simples repositório do git pode conter um número arbitrário de branches, mas sua arvóre de trabalho é associada somente a um deles (o branch "atual" ou "corrente"), e o HEAD aponta para aquele branch.

cache

Termo obsoleto para: index.

chain

Uma lista de objetos, onde cada objeto em uma lista contém a referência para o seu sucessor (por exemplo, o sucessor de um commit poderia ser um dos seus pais).

changeset

Forma como BitKeeper/cvsps chama um "commit". Uma vez que o git não armazena mudanças, mas estados, realmente não faz sentido usar o termo "changesets" com o git.

checkout

A ação de atualizar todo ou parte da árvore de trabalho com um objeto tree ou blob do banco de dados de objetos, e atualizar o index e o HEAD se a árvore de trabalho inteira tem sido apontada para um novo branch.

cherry-picking

No jargão do SCM, "cherry pick" significa escolher um subconjunto de modificações fora da série de modificações (tipicamente commits) e gravá-las como um nova série de mudanças no início de um base de código diferente. No GIT, ele é realizado pelo comando `git cherry-pick` para extrair as modificações introduzidas por commit existente e gravá-las baseadas no início do branch atual como um novo commit.

clean

Um diretório de trabalho está limpo, se ele corresponde a revisão referenciada pelo head atual. Veja também "dirty".

commit

Como um substantivo: Um simples ponto no histórico do git; o histórico completo do projeto é representado como um conjunto de commits inter-relacionados. A palavra "commit" é frequentemente

usada pelo git nos mesmos locais onde outros sistemas de controle de revisão usam as palavras "revisão" or "versão". Também usado como atalho para um objeto tipo commit.

Como um verbo: A ação de armazenar um novo "ponto" do estado do projeto no histórico do git, através da criação de um novo commit representando o estado atual do index e avançando o HEAD para o ponto do novo commit.

objeto commit

Um objeto que contém a informação sobre uma revisão particular, como os pais, quem criou o commit ('committer'), autor, data e o objeto tree que corresponde ao topo do diretório da revisão armazenada.

core git

Estruturas de dados fundamentais e utilitários do git. Expoem somente limitadas ferramentas de gerenciamento de código fonte.

DAG

Gráfico acíclico direcionado. Os objetos commits formam um gráfico acíclico direcionado, por que eles tem pais (direcionados), e o gráfico dos objetos commit é acíclico (existe nenhuma corrente que inicia e termina com o mesmo objeto).

dangling object

Um objeto inalcançável que não é alcançável mesmo de outro objeto inalcançável ; um objeto "dangling" não tem referências para ele de qualquer outra referência ou objeto no repositório.

detached HEAD

Normalmente o HEAD armazena o nome de um branch. Contudo, git também permite você realizar um checkout de um commit arbitrário que não é necessariamente o topo de um branch particular. Nesse caso o HEAD é dito como ser "isolado".

dircache

Você está *muito* atrás. Veja index.

directory

A lista você consegue com um `ls` :-)

dirty

Um diretório de trabalho é dito estar "sujo" se ele contém modificações que não tem sido commitados no branch atual.

ent

Sinônimo favorito para "tree-ish" para alguns geeks. Veja `http://en.wikipedia.org/wiki/Ent_(Middle-earth)` para uma explicação mais profunda. Evite este termo, para não confundir as pessoas.

evil merge

Um merge "infernal" é um merge que introduz modificações que não aparecem em qualquer pai.

fast forward

Um "fast-forward" é um tipo especial de merge onde você tem uma revisão e realiza um merge com as modificações de outro branch que é descendente daquele que você já tem. Muitas vezes nesse caso, você não faz um novo merge, mas ao invés disso só atualiza a sua revisão. Isso acontece frequentemente sobre um branch em um repositório remoto.

fetch

Realizar um fetch significa conseguir o head do branch do repositório remoto, procurar quais objetos estão faltando no banco de dados do repositório local, e recupera-los. Veja também: `git fetch`.

file system

Linus Torvalds originalmente projetou o git para ser um sistema de arquivos no espaço do usuário, exemplo da estrutura que mantém arquivos e diretórios. Que garante a eficiência e velocidade do git.

git archive

Sinônimo para repositório (para pessoas arcaicas).

grafts

Grafts habilitam duas outras linhas de desenvolvimento diferentes para serem unidas juntas através da gravação de informações ancestrais falsas para os commits. Dessa forma você pode fazer git fingir o conjunto de pais que o commit possui sendo diferente daquele que foi gravado quando o commit foi criado. Configurado no arquivo `.git/info/grafts`.

hash

No contexto do Git, é sinônimo para o nome do objeto.

head

Uma referência nomeada para o commit no topo do branch. Heads são armazenados em `$GIT_DIR/refs/heads/`, exceto quando está usando `refs` comprimidos. (Veja git pack-refs.)

HEAD

O branch atual. Em mais detalhes: Sua árvore de trabalho é normalmente derivado do estado da árvore referida pelo HEAD. HEAD é uma referência para um dos heads em seu repositório, exceto quando está usando um HEAD desacoplado, que nesse caso ele pode referenciar um commit arbitrário.

head ref

Um sinônimo para head.

hook

Durante a execução normal de diversos comandos git, chamadas são feitas para scripts opcionais que permitem o desenvolvedor adicionar ou verificar funcionalidades. Tipicamente, os hooks permitem um comando ser pré-verificado e potencialmente abortado, e permitir que uma notificação seja lançada depois que a operação tenha sido feita. Os scripts são encontrados no diretório `$GIT_DIR/hooks/`, e são habilitados através da simples remoção do sufixo `.sample` nos nomes dos arquivos. Em versão mais antigas do git você tinha que torná-los executáveis.

index

Uma coleção de arquivos com informações de estado, no qual os conteúdos são armazenados como objetos. O index é uma versão armazenada de sua árvore de trabalho. Verdade seja dita, ele também pode conter uma segunda, e até mesmo uma terceira versão da árvore de trabalho, que são usadas quando realizam algum merge.

index entry

Uma informação relativa a um arquivo particular, armazenado no index. Uma entrada no index pode estar "umerged", se um merge foi iniciado, mas ainda não finalizado (ex.: se o index contém múltiplas versões daquele arquivo).

master

O branch padrão de desenvolvimento. Sempre que você cria um repositório git, um branch nomeado de "master" é criado, e se torna o branch ativo. Na maioria dos casos, ele contém os arquivos de desenvolvimento local, embora ele seja uma convenção e não seja necessário.

merge

Como verbo: Contruir o conteúdo de outro branch (possivelmente de um repositório externo) no branch atual. No caso onde o merge de entrada for de um repositório diferente, ele pode ser feito primeiramente recuperando o branch remoto e então realizando um merge com o resultado no branch atual. Essa combinação de operações de fetch e merge é chamado de "pull". Um merge é realizado através de um processo automático que identifica mudanças feitas desde as divergências dos branches, e então aplica todas essas alterações juntas. Nos caso onde as alterações conflitam, uma intervenção manual pode ser requisitada para completar o merge.

Como um substantivo: a não ser que seja um "fast forward", o resultado de um merge completo na criação de um novo commit representando o resultado do merge, e tendo como pais os topos dos branches que realizaram merge. Esse commit é referenciado como um "merge commit", ou as vezes só como "merge".

object

A unidade de armazenamento no git. Ele é unicamente identificado pelo SHA1 de seu conteúdo. Consequentemente, um objeto não pode ser modificado.

banco de dados de objetos

Armazena um conjunto de "objetos", e um objeto individual é identificado por seu nome de objeto. Os objetos normalmente ficam em `$GIT_DIR/objects/`.

identificador do objeto

Sinônimo para nome de objeto.

nome do objeto

O único identificador de um objeto. O hash do conteúdo de um objeto usa o "Secure Hash Algorithm 1" e normalmente é representado por uma codificação de 40 caracteres hexadecimais do hash de um objeto.

tipo de objeto

Um dos identificadores "commit", "tree", "tag" ou "blob" descrevendo o tipo de um objeto.

octopus

Para realizar um merge com mais de dois branches. Também denota um predador inteligente.

origin

O padrão para repositórios remotos. A maioria dos projetos tem pelo menos um projeto no qual eles acompanham. Por padrão 'origin' é usado para esse propósito. Novas atualizações neles serão recuperados para os branches nomeados como origin/nome-do-branch-remoto, que pode ser visto usando `git branch -r`.

pack

Um conjunto de objetos no qual foram comprimidos em um arquivo (para economizar espaço ou transmiti-los com eficiência).

pack index

Uma lista de identificadores, e outras informações, dos objetos em um pack, para eficientemente ajudar no acesso do conteúdo de um pack.

parent

Um objeto commit que contém uma (possivelmente vazia) lista de predecessores lógicos em uma linha de desenvolvimento, como exemplo seus pais.

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

Nome mais atraente para "core git".

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

Pulling um branch significa recuperá-lo e realizar um merge nele. Veja também `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.

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.

rebase

Para reaplicar uma série de mudanças de um branch para uma diferente base, e resetar o `head` desse branch para o resultado.

ref

Uma representação hexadecimal de 40 bytes de um SHA1 ou um nome que denota um objeto particular. Esses podem ser armazenados em `$GIT_DIR/refs/`.

reflog

Um `reflog` mostra um "histórico" local de `ref`. Em outras palavras, ele pode dizer a você que a terceira última revisão *nesse* repositório foi, e qual foi o estado atual *nesse* repositório, ontem às 09:14pm. Veja `git reflog` para mais detalhes.

refspec

A "refspec" is used by fetch and push to describe the mapping between remote ref and local ref. They are combined with a colon in the format :, preceded by an optional plus sign, +. For example: `git fetch $URL refs/heads/master:refs/heads/origin` means "grab the master branch head from the $URL and store it as my origin branch head". And `git push $URL refs/heads/master:refs/heads/to-upstream` means "publish my master branch head as to-upstream branch at $URL". See also git push.

repository

Uma coleção de `refs` juntas em um banco de dados de objetos contendo todos os objetos que são alcançáveis pelo `refs`, possivelmente acompanhado por meta dados de um ou mais `porcelains`. Um repositório pode compartilhar um banco de dados de objetos com outros repositórios via mecanismos alternativos.

resolve

A ação de corrigir manualmente uma falha que um merge automático causou.

revision

Um estado particular dos arquivos e diretórios no qual estão armazenados no banco de dados de objetos. Ele é referenciado por um objeto commit.

rewind

Descartar parte do desenvolvimento. Ex.: atribuir o head para uma revisão anterior.

SCM

Gerenciador de código fonte (ferramenta).

SHA1

Sinônimo para um nome de objeto.

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

Referência Simbólica: ao invés de conter uma identificação SHA1 por ele mesmo, ele está no formato 'ref: refs/alguma/coisa' e quando referenciado, ele recursivamente desreferencia para essa referência. 'HEAD' é um ótimo exemplo de um `symref`. Referências simbólicas são manipuladas com o comando `git symbolic-ref`.

tag

Uma `ref` apontando para um objeto tag ou commit. Diferente do head, uma tag não é alterável por um commit. Tags (não objetos tag) são armazenados em `$GIT_DIR/refs/tags/`. Uma tag é mais comumente usado para marcar um ponto particular de um commit dentro do projeto.

tag object

Um objeto contendo uma referência apontando para outro objeto, que pode conter uma mensagem como no objeto commit. Ele pode conter também uma assinatura (PGP), nesse caso ele é chamado de "objeto tag assinado".

topic branch

Um branch git comum que é usado pelo desenvolvedor para identificar uma linha de desenvolvimento conceitual. Como os branches são fáceis e simples, são muitas vezes desejáveis terem diversos pequenos branches onde cada um contém conceitos bem definidos ou ainda pequenas mudanças relacionadas.

tracking branch

Um branch comum que é usado para seguir as modificações de outro repositório. Um tracking branch não deveria conter modificações diretas ou ter commits locais feitos nele.

tree

Qualquer árvore de trabalho, ou uma árvore de objetos juntas com objetos blob e tree dependentes (ex.: uma representação armazenada de uma árvore de trabalho).

objeto tree

Um objeto contendo uma lista de nomes de arquivos e seus modos de acesso com refs para um blob associado e/ou objetos tree. Uma tree é equivalente a um diretório.

tree-ish

Uma referência para qualquer um objeto commit, objeto tree, ou um objeto tag, apontando para um objeto tag ou commit ou tree.

unmerged index

Um index que contém entradas no qual ainda não foram realizadas num merge.

objeto inalcançável

Um objeto no qual não é atingível a partir de um branch, tag, ou qualquer outra referência.

working tree

Uma árvore de trabalho com os arquivos do projeto corrente. Ele é normalmente equivalente ao HEAD mais qualquer alterações locais que você tenha feito mas não foi realizado commit.