Git Community Book

The open Git resource pulled together by the whole community

AUTHORS

Thank these guys:

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

MAINTAINER / EDITOR

Bug this guy:

Scott Chacon (schacon@gmail.com)

Chapter 1

Введение

ДОБРО ПОЖАЛОВАТЬ В GIT

Добро пожаловать в Git. Git - это быстрая, распределенная система контроля версий.

Эта книга предназначена для тех кто впервые столкнулся с Git. Она поможет вам легко и быстро освоить Git.

Вначале этой книги вы найдете описание того как, как Git хранит свои данные. Это поможет вам понять почему Git так отличается от других систем контроля версий. Эта глава займет у вас около 20 минут.

Затем будет глава **Начинающий пользователь Git** - где вы узнаете команды, которыми вы будете пользоваться 90% своего времени. Эта глава даст вам достаточно хорошую базу, чтобы уверенно

пользоваться Git. В большинстве случаев этого будет вполне достаточно. На прочтение этой главы вам понадобится около 30 минут.

Следующая глава **Средний пользователь Git** - здесь более сложный материал, и вы узнаете дополнительные возмжности уже известных вам из первой главы команд. В большинстве случаев это технические приемы и команды которые сделают вашу работу с Git более удобной.

После того как вы одолели предыдущие главы, мы приступим к главе **Продвинутый пользователь Git**. Большинство пользователей не часто используют команды из этой части, но они могут быть очень полезными в определенных ситуациях. После изучения этой главы ваши знания Git покроют все необходимое в повседневной практике; теперь вы мастер Git!

Теперь когда вы уже знаете Git, мы приступим к главе **Работа с Git**. Здесь мы пройдемся по тому как встроить Git в скрипты, инстументы развертывания приложений, с редакторами и т.д. Эта глава главным образом предназначена для того чтобы помочь вам интегрировать Git в ваше рабочее окружение.

В самом конце, вы можете прочитать серию статей о **низко-уровневой документации** которая возможно поможет энтузиастам Git, тем кто хочет изучить внутреннее устройство и протоколы Git.

Отзывы и Сотрудничество

Если вы вдруг заметите ошибку или захотите помочь с книгой, вы можете отправить мне сообщение на мой эл.почту schacon@gmail.com, или вы можете склонировать исходный код этой книги с http://github.com/schacon/gitbook и прислать мне патч или отправить pull запрос.

Ссылки

Большая часть материала для этой книги собрана из различных источников.
Если вы хотите прочитать оригинальные статьи, то проследуйте по ссылкам ниже и поблагодарите их авторов:

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

Примечание: Материал местами сложный и не просто было подобрать термины так что если кто желает поучаствовать в переводе всегда пожалуйств. Русскоязычную версию книги можно найти на https://github.com/uleming/gitbook

1. для термина pull не удалось удалось найти устоявшийся аналог в русском языке

ОБЪЕКТНАЯ МОДЕЛЬ GIT

SHA

Вся информация требуемая чтобы представить историю проекта хранится в особым образом организованных файлах. Все файлы хранят ссылаются друг на друга с помощью 40-значного "имени объекта" и это имя выглядит так:

```
6ff87c4664981e4397625791c8ea3bbb5f2279a3
```

Вы увидите эти 40-значные строки повсюду в Git. В каждом случае имя вычисляется как SHA1 значение содержимого объекта. SHA1 хэш это криптографическая хэш-функция. Для нас это значит то, что практически нереально найти два разных объекта с одинаковым именем. Это дает огромную выгоду; такую как:

- Git может быстро определить идентичны ли два объекта или нет, просто сравнивая их имена.
- Так как имена объектов вычисляются одинаково во всех репозиториях, то объекты с одинаковым содержимым в двух репозиториях всегда будут хранится под одинаковыми именами.
- Git может находить ошибки когда читает объект, для этого нужно просто сравнить хэш значение содержимого объекта с его именем.

Объекты

Каждый объект состоит из трех частей - **тип**, **размер** , **содержимое**. Размер это просто объем содержимого, а содержимое зависит от типа объекта. Существуют 4 разных типа объекта: "блоб", "дерево", "коммит", и "таг".

- **"блоб"** используется чтобы хранить содержимое файла - обычно это просто файл.
- **"дерево"** это что то вроде директории - оно ссылается на группу других деревьев и/или блобов (т.е. файлов и директорий)
- **"коммит"** указывает на отдельное дерево, от по сути отмечает дерево фиксируя в истории каким образом оно выглядет в момент выполнения коммита. Он содержит метаинфомацию фиксируя момент времени и автора изменений внесенных с последднего коммита, указатель на предыдущий коммит, и т.д.
- **"таг"** это способ маркировать некоторым образом определенный комит. Обычно это используется чтобы маркировать(по сути дать какое либо легко запоминающеся имя) определенные комиты как специфические, чтобы впоследствии было легче их найти.

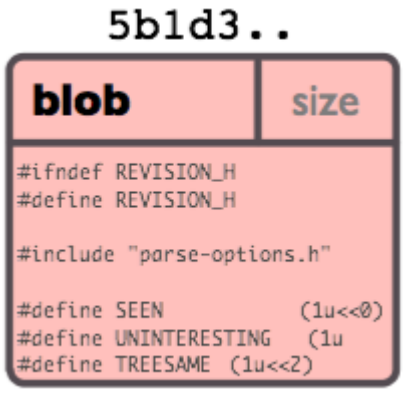
Почти все в Git построено вокруг манипуляций этой простой структурой состоящей из четырех различных типов объектов. Это что-то вроде своеобразной файловой-системы надстроенной над файловой-системой компьютера.

Различия с SVN

Важное замечание: в отличии от других распространенных систем контроля версий с которым вы возможно знакомы. Subversion, CVS, Perforce, Mercurial другие подобные им используют Delta Storage Systems (приемы на базе дельта алгоритмов http://ru.wikipedia.org/wiki/Дельта-кодирование) - они хранят разницу между двумя следующими друг за другом комитами. Git не делает этого - он хранит снапшот своего рода точный снимок всех файлов и директорий, состояния всего дерева в момент коммита. Очень важно понимать эту концепцию когда испльзуете Git.

Объект типа блоб

Блоб обычно хранит содержимое файла.



Вы можете использовать git show чтобы исследовать содержимое блоба. Предположим у нас есть SHA-значение блоба, таким образом чтобы просмотреть его содержимое выполните выполнить:

```
$ 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.
...
```

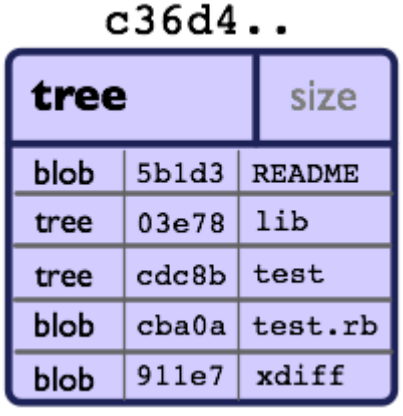
Объект "блоб" это всего лишь некоторая порция бинарных данных. Он ни на что не ссылается у него нет каких либо атрибутов, нет даже имени файла.

Поскольку блоб полностью определяется его собственным содержимым, то если два файла в директории или даже в разных версиях репозитория имеют одинаковое содержимое, они будут разделять один и тот

же блоб объект. Объект полностью независит от его расположения в дереве каталогов, и переименование файла не изменит объект с которым этом файл связан.

Объект дерево

Дерево это простой объект который заключает в себе группу указателей на блобы и другие деревья - обычно представляет содержимое директорий или поддиректорий.



Команда git show более общая, и также может быть использована чтобы исследовать дерево объектов, но git ls-tree даст вам больше подробностей. Предположим у нас есть SHA значение дерева, тогда мы можем исследовать его следующим образом:

```
$ 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
...
```

Как вы можете видеть, объект дерево содержит список записей. Каждая запись состоит из вида, типа объекта, SHA1 значения, и имени соответственно. Записи отсортированы по имени. Так выглядит содержимое одной директории дерева.

Ссылка на объект в дереве может быть как блобом (файлом по сути) так и деревом (поддиректорией). Поскольку имена всех объектов, деревьев и блобов, совпадает с SHA хэш-значением их содержимого, то SHA значения двух деревьев будут идентичны только если их содержимое (включая, рекурсивно, содержимое всех поддиректорий) идентично.

Это свойство позволяет git быстро найти отличия между двумя родственными объектами типа дерево, так как git может игнорировать объекты с одинаковыми именами.

Замечание: деревья могут также содержать записи коммитов. Более продробно от этом в секции **Подмодули**.)

Отметьте для себя, что все файлы имеют права 644 или 755: фактически git обращает внимание только на бит исполнения.

Объекты коммит

Объект "коммит" связывает физическое состояние дерева с описпнием того каким образом мы пришли к этому и почему.

Вы можете использовать параметр --pretty=raw с git show или git log чтобы исследовать коммит:

```
$ 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>
```

Как вы можете это видеть, коммит определяется:

- **дерево**: SHA1 имя объекта дерево (как определено ниже), представляющее содержимое директории в определенный момент времени.
- **родитель(и)**: SHA1 имя некоторого числа коммитов которые представляют собой предыдущий шаг(и) в истории проекта. Пример выше имеет одного родителя; хотя коммиты слияния могут иметь более чем одного родителя. Коммит без родителей называется "root (корневой)" коммит, и представляет собой начальное состояние проекта. Каждый проект должен иметь по крайней мере один корневой коммит. Проект может также иметь множество корней, однако это не общий случай (и не обязательно хорошая идея).
- **автор**: Имя разработчика ответственного за эти изменения, вместе с датой.
- **коммитер**: имя разработчика который создал этот коммит, вместе с датой этого события. Оно(имя) может отличаться от имени автора; например в случае, если автор написал патч и отправил его по эл.почте другому разработчику который наложил патч и выполнил коммит.
- **комментарий** описывающий этот коммит.

Заметьте что коммит сам по себе не содержит никакой информации о том что изменилось; все изменения вычисляются при сравнении содержимого дерева на которое ссылается создаваемый коммит и дерева на которое ссылается его родитель. Git не пытается явно регистрировать переименования файлов хотя может идентифицировать случаи где существование одниковых файловых данных в измененном пути предложит переименовать. (Просмотрите, например параметр -M к команде git diff).

Коммит обычно создается git commit. Эта команда создает коммит - родитель которого текущая ветка HEAD, и чье дерево взято из содержимого сохраненного в данный момент в индексе.

Объектная модель

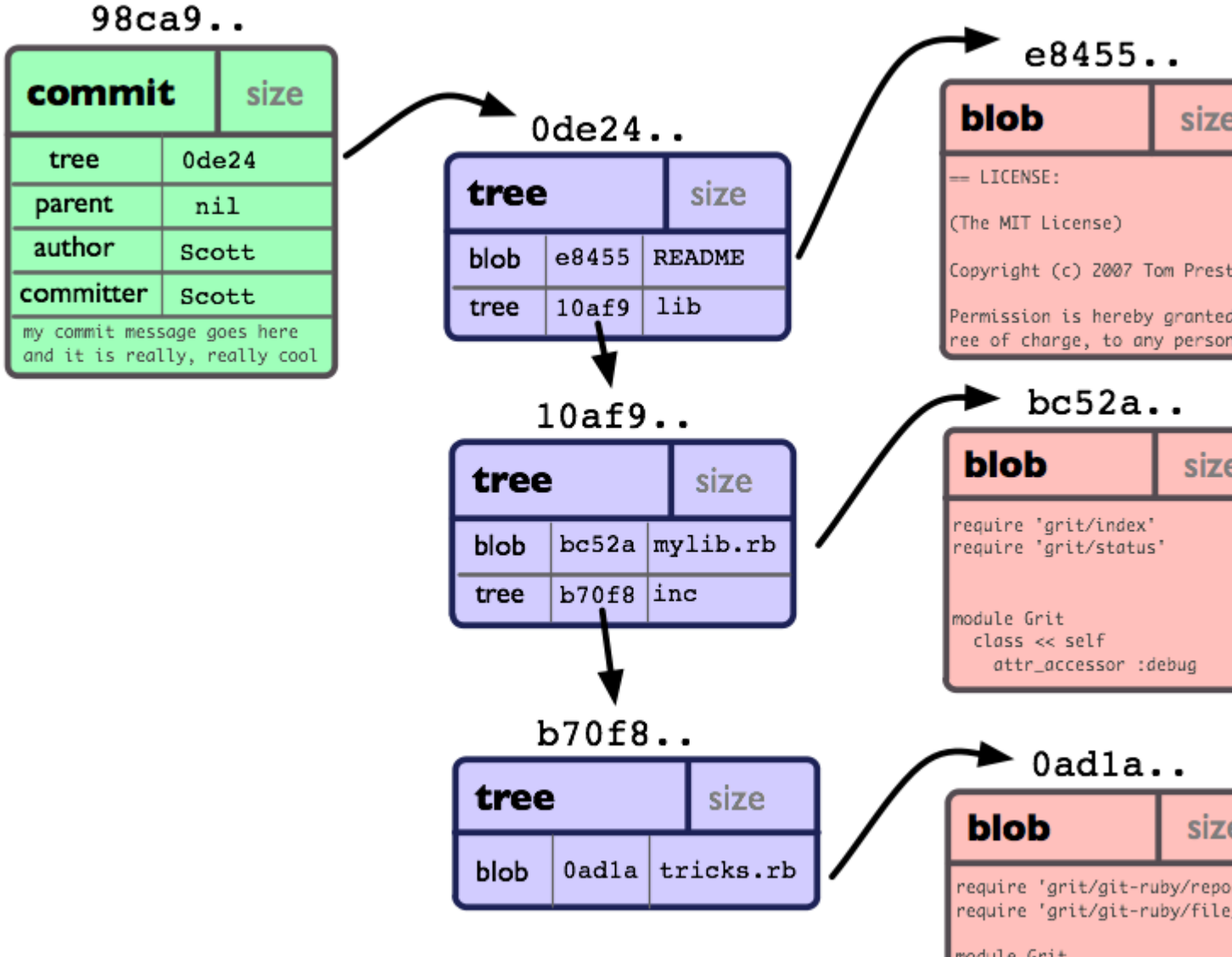
Теперь когда мы рассмотрели 3 главных объекта (блоб, дерево и коммит), давайте теперь посмотрим как они объединяются.

Если у нас есть простой проект со след. структурой директории:

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

2 directories, 3 files
```

И мы выполнили коммит всего этого в репозиторий Git, это будет выглядеть след. образом:


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

Вы можете видеть что мы создали объект **дерево** для каждой директории (включая корневую) и объект **блоб** для каждого файла. Затем, мы имеем объект **коммит** указывающий на кореневую директорию, и мы можем отследить как наш проект выглядел в момент коммита.

Объект таг

Объект таг содержит имя объекта (называетмого просто 'объект'), тип объекта, имя тага, или разработчика ("таггер") который создал таг, и сообщение, которое может содержить подпись. Это можно увидеть выполнив git cat-file:

```
$ git cat-file tag v1.5.0
object 437b1b20df4b356c9342dac8d38849f24ef44f27
type commit
tag v1.5.0
```

```
tagger Junio C Haemano <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-----
```

Просмотрите документацию команды git tag чтобы изучить как создавать и проверять объекты таг. (Заметьте что git tag может также использоваться чтобы создавать "легковесные таги", которые не являеются объектами таг вообще, это просто ссылки чьи имена начинаются с "refs/tags/").

ДИРЕКТОРИЯ GIT И РАБОЧАЯ ДИРЕКТОРИЯ

Директория Git

'Директория git' это директория в которой хранится вся история Git и мета-информация вашего проекта - включая все объекты (коммиты, деревья, блобы, таги), все указатели на различные ветви и многое другое.

На каждый проект имеется только одна директория Git (в отличие SVN или CVS, где она в каждой поддиректории), и это директория (по умолчанию но не обязадельно) '.git' в корне вашего проекта. Если вы посмотрите на содержимое этой директории то увидите все ваши важные файлы:

```
$>tree -L 1
.
|-- HEAD         # указатель на вашу активную ветку
```

```
|-- config       # ваши персональные настройки
|-- description  # описание проекта
|-- hooks/       # pre/post action hooks (скрипты (далее хуки) которые могут вызываться git командами)
|-- index        # индексный файл (смотрите в след.главе)
|-- logs/        # история веток проекта (где они располагались)
|-- objects/     # ваши объекты (коммиты, деревья, блобы, таги)
`-- refs/        # указатели на ваши ветки разработки
```

(Также там могут быть и другие файлы/директории, но они не так важны в данный момент)

Рабочая директория

'Рабочая директория' Git это директория которая содержит в себе то с чем вы работаете или то что вы извлекли из истории проекта в данный момент. Файлы в этой директории часто удаляются или изменяются Git-ом когда вы переключаетесь между ветками - не переживайте это нормально. Вся история вашего проекта хранится в директории Git; рабочая директория это просто временное место где вы можете модифицировать файлы, а затем выполнить коммит.

Замечания: Коммит это фиксация изменений в истории проекта

ИНДЕКС GIT

Индекс Git - используется как промежуточная ступень между вашей рабочей директорией и репозиторием. Вы можете использовать индекс чтобы собрать набор изменений, которые впоследствии вы хотите закоммитить вместе. Когда вы выполняете коммит, в действительности в этот коммит идут данные из индекса, а не из рабочей директории.

Как просмотреть индекс

Самый быстрый способ увидеть что индексе, можно с помощью команды git status. Когда вы выполните команду git status, то увидите какие файлы попали в индекс, какие модифицированы но не в индексе, и какие в данный момент вообще неотслеживаемые git.

```
$>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
```

Если вы удалили полностью индекс, в общем вы не потеряли никакой информации поскольку у вас есть еще имя дерева которое он описывает.

Теперь вы достаточно хорошо должны понимать основы того, что Git делает за сценой, и чем он отличается от других систем контроля версий. Не волнуйтесь если вы не полностью поняли прочитанный материал; мы вернемся к этим темам в следующих главах. Теперь вы готовы приступить к установке, конфигурированию, и использованию Git.

Chapter 2

Первый раз

УСТАНОВКА GIT

Установка из исходников

Кратко, на Unix системах, вы можете скачать исходный код Git со страницы закачки Git Download Page, и затем выполнить следующие команды:

```
$ make prefix=/usr all ;# от своего имени
$ make prefix=/usr install ;# как суперпользователь (root)
```

Вам потребуются следующие библиотеки: expat curl zlib openssl хотя они уже будут в системе, за исключением возможно *expat*,

Линукс

Если вы пользователь Линукс, скорее всего лучший способ установить Git это использовать вашу родную систему управления пакетами:

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

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

Если это не сработает, то можете скачать готовый установочный *.deb или* .rpm пакет отсюда:

RPM Packages

Stable Debs

Если вы предпочитаете установить Git из исходниов на Linux систему, то обратитесь к этой статье:

Article: Installing Git on Ubuntu

Mac 10.4

На обоих Mac 10.4 and 10.5, вы можете установить Git через MacPorts, Если у вас уже установлен MacPorts. Если нет, то установите его отсюда here.

Как только вы установили MacPorts, все что вам нужно сделать это выполнить:

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

Если вы предпочитаете уставить из исходников, эти статьи вам помогут:

Article: Installing Git on Tiger

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

Mac 10.5

На Leopard, вы также можете установить через MacPorts, но здесь у вас есть дополнительный выбор, использовать инсталятор, который вы можете скачать отсюда: Git OSX Installer

Если вы предпочитаете устанавливать Git из исходников, эти статью помогут вам:

Article: Installing Git on OSX Leopard

Article: Installing Git on OS 10.5

Windows

На Windows, установить Git очень легко. Просто скачайте и установите msysGit инсталятор.

Просмотрите главу *Git на Windows* где есть скринкаст демонстрирующий установку и использование Git на Windows.

УСТАНОВКА И ИНИЦИАЛИЗАЦИЯ

Конфигурирование Git

Первое что вам нужно сделать это настроить свое имя и эл.почту в Git, эта информация будет использоваться когда вы выполняете коммиты.

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

Эти две команды добавят след.строчки в особый файл в вашей домашней директории. Этот файл будет использован для всех ваших проектов. По умолчанию этот файл *~/.gitconfig*, и его содержимое будет выглядеть след.образом:

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

Если вы желаете переопределить эти значения для какого либо определенного проекта (например чтобы использовать ваш рабочую не персональную эл.почту), то вы можете выполнить команду *git config* без параметра *--global* когда находитесь внутри этого проекта (Git также определяет возможность вручную указать файл конфигурации в параметрах запуска команды git). Это добавит секцию [user], такую же как и было показано выше, в файл *.git/config* в вашей корневой директории проекта.

Примечание: В Windows следует выполнить также git config --global core.autocrlf true git config --global core.safecrlf true

Chapter 3

Начинающий пользователь Git

КАК ПОЛУЧИТЬ GIT РЕПОЗИТОРИЙ

Теперь когда Git уже установлен и сконфигурирован, нам нужен Git репозиторий. Этого можно добиться двумя способами. Первый - мы можем *клонировать* существующий репозиторий. Второй - мы можем *инициализировать* репозиторий, либо в уже существующей директории исходников, либо в пустой директории.

Клонирование репозитория

Для того чтобы получить копию проекта, вам нужно знать Git URL проекта - это расположение проекта. Git может работать со многими различными протоколами, URL может начинаться с ssh://, http(s)://, git://, или с имени пользователя (в этом случае git предположит что нужно использовать ssh). Некоторые репозитории

могут быть доступны на нескольких протоколах. Для примера, исходный код самого Git может быть склонирован или через git:// протокол:

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

или через http:

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

Протокол git:// более быстрый и эффективный, но иногда необходимо использовать http, если вы позади корпоративного или персонального фаервола. В любом случае в итоге у вас появится директория с именем 'git' которая содержит весь исходный код Git и историю - по существу это полная копия того что есть на сервере.

По умолчанию Git даст имя новой директории следующим образом. URL клонируемого репозитория заканчивается на '.git', все что идет до этого окончания будет использоваться как имя для новой директории. (т.е. *git clone http://git.kernel.org/linux/kernel/git/torvalds/linux-2.6.git* результат будет новая директория 'linux-2.6')

Инициализация нового репозитория

Предположим у вас есть архив тарболл(tarball) с именем project.tar.gz, где исходники вашего проекта. Вы можете поместить проект в git след. образом:

```
$ tar xzf project.tar.gz    #распаковать проект
$ cd project                #перейти в директорию проекта
$ git init                  #инициализировать git для проекта
```

Вы увидите вывод команды Git

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

Теперь у вас есть проинициализированная рабочая директория -- и вы возможно заметили что появилась новая директория ".git".

gitcast:c1_init

ОБЫЧНЫЙ РАБОЧИЙ ПРОЦЕСС

Модифицируйте некоторые файлы, а затем добавьте их в индекс:

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

Теперь вы готовы выполнить коммит. Вы можете узнать что именно будет добавлено в коммит используя git diff с параметром --cached:

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

(Без --cached, git diff покажет вам все изменения, которые вы сделали но не добавили еще в индекс.) Вы также можете получить краткое описание сложившейся ситуации с помощью git status:

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

Если вам необходимо сделать еще какие-либо изменения, делайте это сейчас, и затем добавьте измененные файлы в индекс. В заключении выполните коммит чтобы закрепить изменения след.образом:

```
$ git commit
```

Вы увидите приглашение ввести сообщение описывающее изменения, а затем и будет записана новая версия проекта.

Как альтернатива, вместо предварительного выполнения команды `git add`, вы можете использовать следующую команду

```
$ git commit -a
```

эта команда будет автоматически определять измененные файлы (но не новые файлы), добавлять их в индекс, и затем коммитить, и все это за один шаг.

Замечание (как описывать коммиты): Несмотря на то что это необязательно, хорошая привычка начинать сообщение-описание коммита с одной короткой строки (менее чем 50 символов) подытоживающей проделанные изменения, затем оставить пустую строку и затем дать более развернутое описание. Инструменты, которые отсылают эл.сообщение о сделанном коммите, обычно используют первую строку как Тему: в эл.письма и подробное описание в теле эл.письма.

Git отслеживает содержимое не файлов

Во многих системах контроля версий есть команда "add", которая говорит системе начать отслеживать изменения нового файла. В Git команда "add" более простая но вместе с тем более мощная: `git add` используется как для новых так и для модифицированных файлов, и в обоих случаях она делает снапшот

указываемых файлов и помещает их в индекс, после чего эти файлы будут включены в следующий коммит.

gitcast:c2_normal_workflow

УПРАВЛЕНИЕ ВЕТВЛЕНИЕМ И СЛИЯНИЕМ

Один git репозиторий может заключать в себе множество ветвей разработки. Чтобы создать новое ответвление под именем "experimental", выполните команду

```
$ git branch experimental
```

Теперь если вы выполните

```
$ git branch
```

то получите список всех существующих ветвей:

```
  experimental
* master
```

Ветка "experimental" это та, которую вы только что создали, а ветка "master" это ветка по умолчанию которая создается автоматически. Звездочка указывает в какой ветке вы в данный момент находитесь; наберите

```
$ git checkout experimental
```

чтобы переключиться в ветку experimental. Теперь отредактируйте файл, выполните комит, и переключитесь обратно в главную ветку "master":

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

Убедитесь, что сделанные изменения невидимы, поскольку они были сделаны в ветке experimental, а вы сейчас в главной ветке "master".

Вы можете сделать другое изменение в ветке "master", затем выполнить коммит:

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

на этом этапе две ветки разошлись, поскольку в каждой из них различные изменения. Чтобы включить изменения в ветке experimental в master, выполните

```
$ git merge experimental
```

Если изменения не конфликтуют, то вы закончили. Если же существуют какие либо конфликты, то в проблемных файлах останутся заметки которые можно увидеть выполнив;

```
$ git diff
```

Как только вы отредактировали файлы вызывающие конфликты выполните,

```
$ git commit -a
```

это выполнит коммит результат слияния. В заключении,

```
$ gitk
```

покажет наглядное графическое представление истории.

Теперь вы можете удалить ветку experimental командой

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

Эта команда гарантирует что изменения в ветке experimental уже в текущей активной ветке.(Прим. переводчика: если вы попытаетесь удалить ветку которую вы не слили со своей рабочей git выведет предупреждение и попросит вас выполнить след.команду $ git branch -D experimental)

Если вы отрабатываете в ветке сумашедшие идеи, и уже пожалели об этой ветке, вы всегла можете удалить ветку выполнив

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

Ветки это легко и просто, и это хороший способ попробовать что то новое.

Как сливать ветки

Вы можете объединить две разошедшиеся ветки разработки используя git merge:

```
$ git merge branchname
```

сливает изменения сделанные в ветке "branchname" в активную(рабочую) ветку. Если присутствуют конфликты -- например один и тот же файл модифицирован разными способами в удаленной и локальной ветках -- то вы будете предупреждены; вывод будет выглядеть след. образом:

```
$ 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.
```

Отметки конфликтов останутся в проблемных файлах, и после того как вы исправите их вручную, вы можете обновить индекс и выполнить git commit, как вы обычно это делаете когда изменяете файл.

Если вы просмотрите результирующий коммит используя gitk, вы увидите что у него два родителя: один указывает на последний коммит активной ветки, а другой на последний коммит другой ветки.

Исправление конфликтов при слиянии

Когда слияние не происходит автоматически, git оставляет индекс и рабочее дерево в особом состоянии которое дает всю информацию необходимую чтобы разрешить конфликт.

Файлы с конфликтами отмечаются в индексе особым образом, так что до тех пор пока вы не исправите проблему и не обновите индекс, выполненить git commit не удастся:

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

Также, git status перечислит эти файлы как "unmerged", а файлы с конфликтами будут иметь добавленные отметки, и выглядеть след.образом:

```
<<<<<<< HEAD:file.txt
Hello world
=======
```

```
Goodbye
>>>>>>> 77976da35a11db4580b80ae27e8d65caf5208086:file.txt
```

Все что вам нужно это отредактировать файлы, исправить конфликты, а затем выполнить

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

Заметьте что сообщение-описание коммита уже будет содержить некоторую информацию о слиянии. Обычно вы можете оставить это сообщение-описание, а также можете добавить свои дополнительные комментарии, если пожелаете.

Теперь вы знаете все что вам нужно чтобы выполнить простое слияние. Но git предоставляет больше информации, чтобы помочь разрешить конфликты:

Отменить слияние

Если в процессе слияния и исправления конфликтов,вы застряли и решили сдаться и выбросить все к черту, то вы всегда можете вернуться с состоянию pre-merge (такое же как и было до того как вы запустили слияние) выполнив

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

Если вы уже выполнили коммит после слияния, и вы хотите сбросить его,

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

Как бы там ни было, последняя команда может быть опасной в некоторых случаях --никогда не сбрасывайте коммит если этот коммит возможно уже был использован в слиянии в другую ветку, если вы это сделаете то это запутает последующие слияния.

fast-forwarding слияния

Есть один специальный случай не упомянутый выше, который несколько иной по сути. Обычно, результаты слияния это коммит у которого два родителя, один на каждую ветку разработки.

Однако в некоторых случаях когда активная ветка не отклонилась от другой -- и каждый коммит в активной ветке уже содержится в другой -- то git просто выполняет "fast forward"; голова активной ветки перемещается вперед и указывается на голову сливаемой ветки, без создания каких-либо новых коммитов.

gitcast:c6-branch-merge

ПРОСМОТР ИСТОРИИ - GIT LOG

Команда git log может показать список коммитов. Сама по себе она показывает все коммиты достижимые из родительского коммита; но вы можете также сделать более определенный запрос:

```
$ git log v2.5..        # коммиты с (но не дистижимые) v2.5
$ git log test..master  # коммит достижимый из ветки master но не из test
$ git log master..test  # коммит достижимый из ветки test но не из master
$ git log master...test # коммит достижимый или из test или master, но не
                        # из обоих
$ git log --since="2 weeks ago" # коммиты начиная с 2 недель назад
$ git log Makefile      # коммиты которые модифицировали Makefile
```

```
$ git log fs/          # коммиты которые модифицировали любой из файлов в
                       # поддиректории fs/
$ git log -S'foo()'    # коммиты которые добавили или удалили любые                    # файловые данны
$ git log --no-merges  # не показывать коммиты слияния
```

Конечно вы можете комбинировать их; следующая команда найдет коммиты начиная с v2.5 которые затрагивают файл Makefile или любой файл в директории fs/:

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

Git log покажет список из всех коммитов, начиная с наиболее свежего(по дате) коммита, который удовлетворяет условиям заданным в аргументах команды.

```
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.
```

Вы также можете попросить git log показать патчи:

```
$ 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):
```

Статистика в логах

Если вы передадите параметр `--stat` в 'git log', он покажет вам которые файлы изменились в этом коммите и как много строк кода было добавлено и удалено из каждого из них.

```
$ 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(-)
```

Форматирование вывода log

Вы можете форматировать вывод log так как это удобно. Параметр '--pretty' может принимать множество предопределенных значений, таких как 'oneline' (в одну линию):

```
$ 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
```

или вы можете получить 'short' (кратко) форматирование:

```
$ 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
```

Далее список возможных вариантов 'medium', 'full', 'fuller', 'email' или 'raw'. Тут лучше поэкспериментировать, чтобы выяснить какой наиболее вам подходит. Если ни один из них не удовалетворяет вашим потребностям вы можете создать свой собственный формат задав параметр след.образом '--pretty=format' (просмотрите документацию git log чтобы узнать все форматирующие параметры).

```
$ 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:
```

Другая интересная вещь которую вы можете сделать - это визуализировать граф коммитов ипользуя параметр '--graph', след.образом:

```
$ 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
```

Это даст вас очень легкое для восприятия ASCII представление истории коммитов.

Упорядочивание Log

Вы также можете сортировать вывод в различном ввиде. Заметьте что git log начинает с самого свежего коммита,затем спускается по родительской линии; так как история git может содержать множество независимых ветвей разработки, определенный порядок коммитов может быть произвольным.

Если вам хочется изменить порядок вывода особым образом, вы можете добавить параметр урорядочивающий вывод команды git log.

По умолчанию, коммиты показываются в обратном хронологическом порядке.

Как бы там ни было, вы можете определить '--topo-order', который выведет коммиты в хронологическом порядке (т.е. коммиты потомки будут выведены перед их родителями). Если мы просмотрим git log для репозитария Grit в --topo-order, то увидим что все линии разработки сгруппированы вместе.

```
$ 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.
```

Вы можете также использовать '--date-order', который изначально упорядочивает коммиты по дате коммита. Этот параметр похож на --topo-order в том смысле что он располагает родителей позади потомков, но тем не менее вывод упорядочен по времени коммита. Вы можете видеть что линии разработки не сгруппированы здесь вместе, так что они скачут вокруг в процессе паралельной разработки:

```
$ 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
```

В заключении, вы можете изменить порядок вывода на обратный используя параметр '--reverse'.

gitcast:c4-git-log

СРАВНЕНИЕ КОММИТОВ - GIT DIFF

Вы можете генерировать diff между любыми двумя версиями вашего проекта используя git diff:

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

Это сгенерирует diff между последними коммитами двух ветвей разработки. Если вы предпочитаете найти diff от их общего предка к ветке test, вы можете использовать три точки вместо двух:

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

git diff это невероятно полезный инструмент для нахождения изменений между любыми двумя точками в истории вашего проекта, или чтобы увидеть что другие разработчики пытаются вносить в новые ветви. и т.д.

Что вы будете коммитить

Вы будете обычно использовать git diff для нахождения различий между последним коммитом, вашем индексом, и вашей рабочей директории. Это просто сделать выполнив

```
$ git diff
```

Это покажет изменения в вашей рабочей директории которые еще не были добавлены в индекс для последующего коммита. Если вы хотите видеть что в индексе готово для коммита то выполните

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

что покажет вам различия между индексом и вашим последним коммитом; то что вы бы закоммитили, если выполнили коммит командой "git commit" без параметра "-a". В заключении вы можете выполнить

```
$ git diff HEAD
```

что покажет изменения в рабочей директории от последнего коммита; то что вы бы закоммитили если выполнили команду "git commit -a".

Больше параметров Diff

Если вы хотите увидеть как ваша рабочая директория отличается от состояния проекта в другой ветке, то выполните команду

```
$ git diff test
```

Это покажет вам что именно отличается между вашей рабочей директорией и снапшотом в ветке 'test'. Вы также можете ограничить сравнение определенным файлом или поддиректорией добавив *path limiter* (*ограничитель пути*):

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

Эта команда покажет вам изменения между вашей рабочей директорией и последним коммитом (или, если быть более точным, концом текущей ветки, ограничивая сравнение файлами в поддиректории 'lib'.

Если вы не хотите видеть весь патч, вы можете добавить параметр '--stat', которые ограничит вывод списком файлов с изменениями и с кратким текстовым графическим описанием сколько строк изменилось в каждом файле..

```
$>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(-)
```

Иногда полезно увидеть общие изменения чтобы освежить память.

РАСПРЕДЕЛЕННЫЙ РАБОЧИЙ ПРОЦЕСС

Предположим, что Алиса начала новый проект с git репозиторием в директории /home/alice/project, и что Боб, у которого есть своя домашняя директории на той же машине, хочеть помочь с проектом.

Боб начнет работу с:

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

Это создаст новую директорию "myrepo" содержащую копию репозитория Алисы. Клон эквивалентен в основании оригиналу проекта, и обладает своей собственной копией истории оригинального проекта.

Затем Боб вносит некоторые изменения и выполняет коммит:

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

Когда он готов, он говорит Алисе выполнить pull изменений из репозитория в /home/bob/myrepo. Она совершает это выполнив:

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

Это сливает изменения из ветки "master" Боба в активную ветку Алисы. Если Алиса внесла изменения в свою ветку в тоже самое время, тогда возможно ей придется вручную исправить конфликты если таковые появятся. (Заметьте аргумент "master" в команде выше в действительности не требуется, так как он будет использоваться по умолчанию.)

Команда "pull" таким образом выполняет два действия: она вытягивает изменения из удаленного репозитория, и сливает их в активную ветку.

Когда вы работаете в маленькой группе, это обычная практика работать с одим и тем же репозиторием снова и снова. Определив сокращение 'remote(удаленный)' репозитория вы можете упростить себе работу:

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

С этим Алиса, может выполнять первую операцию одной командой "git fetch" без сливания их в свою ветку:

```
$ git fetch bob
```

В отличие от полной формы, когда Алиса извлекает изменения из удаленного репозитория Боба используя сокращение с помощью `git remote`, то что было вытянуто, хранится в удаленной отслеживающей ветке, в нашем случае это `bob/master`. Так после этого:

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

покажет список всех изменений которые Боб сделал с того времени как он ответвился от ветки "master" Алисы.

После проверки этих изменений, Алиса может слить изменения в свою ветку master:

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

Этот `merge` может также быть выполнено с помощью 'вытягивания из ее собственной удаленной отслеживаемой ветки', слею.образом:

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

Заметьте что команда git pull всегда сливает в активную ветку, не обращая внимания на другие аргументы в командной строке.

Позже Боб, может обновить свой репозиторий включив последние обновления Алисы, выполнив

```
$ git pull
```

Заметьте что ему не требуется указывать путь к репозиторию Алисы; когда Боб клонирует репозиторий Алисы, git хранит местоположение ее репозитория в настройках репозитория, и это расположние используется для извлечения:

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

(Полные настойки созданы командой git-clone. Их можно просмотреть с помощью "git config -l", страница справочника git config описывает значения каждого параметра.)

Git также держит чистую копию ветки "master" Алисы под именем "origin/master":

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

Если Боб позже решит работать с другой машины, он все еще может выполнить клонирование и вытянуть данные используя ssh протокол:

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

Кроме того, у git есть свой родной протокол, или можно использовать rsync или http; просмотрите git pull чтобы получить больше подробностей.

Git может также быть использован в режиме CVS, центрального репозитория в который различные пользователи добавляют изменения; просмотрите git push и gitcvs-migration.

Публичные git репозитории

Другой способ передать изменения в проект это попросить мантейнера проекта вытянуть изменения из вашего репозитория используя git pull. Это способ получить обновления из "main" репозитория, но он хорошо работает и в другом направлении.

Если вы и мантейнер оба имеете учетные записи на одной машине, тогда вы можете просто вытянуть изменения из репозиториев друг друга напрямую; команды которые принимают URLы репозиториев как аргумент также примут имя локальной директории:

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

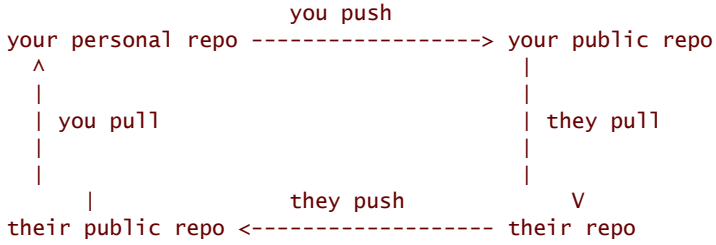
или ssh URL:

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

Для проектов с малым количеством разработчиков, или для синхронизации нескольких частных репозиториев, это именно то что вам нужно.

Как бы так ни было, более общий способ сделать это открыть отдельный публичный репозиторий (обычно на отдельной машине) для других, из которого они могли бы вытягивать изменения. Обычно это более удобный способ, и он позволяет вам отделить личную работу от публичной ее части.

Вы продолжите делать вашу обычную работу в вашем личном репозитории, но периодически будете вставлять изменения из вашего личного репозитория в ваш публичный репозиторий, позволяя другим разработчикам вытягивать из этого репозитория. Так поток изменений, в ситуации где есть один разработчик с публичным репозиторием, выглядит так:



Внесение изменений в публичный репозиторий

Заметьте что экспорт через http или git позволяет другим мантейнерам получать ваши изменения, но у них нет доступа на запись. Для этого вам нужно обновить публичный репозиторий и включить туда последние обновления созданные в вашем частном репозитории.

Простейший способ сделать это - использовать git push и ssh; чтобы обновить удаленную ветку по имени "master" с последним состоянием вашей ветки под именем "master", выполните

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

или просто

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

Как и с git-fetch, git-push будет жаловаться если это не результат fast forward; просмотрите след.секцию чтобы узнать как решить эту проблему.

Заметьте что цель "push" обычно пустой репозиторий. Вы также можете выполнить push в репозиторий который имеет извлеченное рабочее дерево, но рабочее дерево не обновится выполнением push. Этот способ ведет к неопределенным результатам если ветка в которую вы выполняете push - активная извлеченная ветка!

Также как и с git-fetch, вы возможно также установите параметры конфигурации для безопасной печати; так например после

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

вы должны быть способны выполнить push выше след.образом

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

Просмотрите объяснение remote..url, branch..remote, and remote..push параметры в git config для получения подробностей.

Что делать если выполнение push завершилось неудачей

Если результат выполнения push не будет fast-forward удаленной ветки, то ошибка будет выглядеть так:

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

Это может случиться, например если вы:

```
- использовали `git-reset --hard` чтобы удалить уже опубликованные коммиты или
- использовали `git-commit --amend` чтобы заменить уже опубликованные коммиты или
- использовали `git-rebase` чтобы переопределить любой уже опубликованный коммиты.
```

Вы также можете заставить git-push выполнить обновление в любом случае если перед именем ветки поставите символ плюс:

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

Обычно всякий раз когда голова ветки в публичном репозитории модифицируется, оно модифицируется так чтобы указывать на потомка коммита на который она указывало до этого. Принуждая выполнение push в такой ситуации, вы нарушаете это соглашение.

Тем не менее, это общая практика для разработчиков которым нужен простой способ опубликовать серию патчей, и это допустимый компромис пока другие разработчики предупреждены каким образом вы намереваетесь управлять веткой.

Также возможны такие ситуации когда push завершается неудачно в случае если другие разработчики имеют права на выполнение push в тот же репозиторий. В этом случае правильное решение - это

повторить push после первого обновления вашей работы или выполнив pull или выполнив fetch с последующим rebase.; Просмотрите след. секцию и gitcvs-migration чтобы получить больше подробностей.

gitcast:c8-dist-workflow

ТАГИ GIT

Легковесные Таги

Мы можем создавать таг чтобы сослаться на определенный коммит выполняя git tag без каких-либо аргументов.

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

После этого, мы можем использовать stable-1 чтобы ссылаться на коммит 1b2e1d63ff.

Это создает легковесный таг, обычно это ветка которая никогда не двигается. Если вам хочется включить коментарий в таг, а также возможно вставить криптографическую подпись, тогда ты можем создать "таг объект*.

Таговые объекты

Если в команду передан один из параметров **-a**, **-s**, или **-u** , то эта команда создает таговый объект, и требует сообщение-описание тага. Если только не переданы параметры -m или -F , то запускается редактор для пользователя чтобы он мог ввести сообщение-описание тага.

Когда это происходит, новый объект добавляется в базу данных объектов Git и таговая ссылка указывает на этот *таговый объект*, лучше чем сам коммит. Польза от этого, то что вы можете подписать таг, и вы можете проверить позже что это правильный коммит. Вы можете создать таговый объект след.образом:

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

Вообще в действительности возможно поставить таг на любой объект, но таггинг объектов типа коммит более общий. (В исходниках ядра Linux, первый таговый объект ссылается на дерево, ранее чем на коммит)

Подписанные таги

Если у вас есть установленный GPG ключ, вы можете создать подписанный таг легко. Первое, что вам нужно это установить ваш id ключа в ваш *.git/config* или *~.gitconfig* файл.

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

Вы также можете установить это выполнив

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

Теперь вы можете создать подписанный таг просто заменив **-a** на **-s**.

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

Если у вас нет ключа GPG в вашем конфигурационном файле, вы можете выполнить то же самое след.путем:

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

Chapter 4

Средний уровень использования

ИГНОРИРОВАНИЕ ФАЙЛОВ

Проект часто создает файлы которые вы не хотите отслеживать с помощью git. Это обычно включает файлы генерируемые процессом сборки или временные файлы созданные вашим редактором. Конечно, понятие неотслеживаемые файлы git-ом означает что они не будут обрабатываться при выполнении `git add`. Но это быстро начинает раздражать, когда вокруг лежат неотслеживаемые файлы; например они делают `git add .` и `git commit -a` практически бесполезными, и они могут содержаться в выводе команды "`git status`".

Вы можете указать git игнорировать определенные файлы создав файл .gitignore на самом верхнем уровне рабочей директории, добавив в его содержимое что то вроде:

```
# Lines starting with '#' are considered comments.
# Ignore any file named foo.txt.
```

```
foo.txt
# Ignore (generated) html files,
*.html
# except foo.html which is maintained by hand.
!foo.html
# Ignore objects and archives.
*.[oa]
```

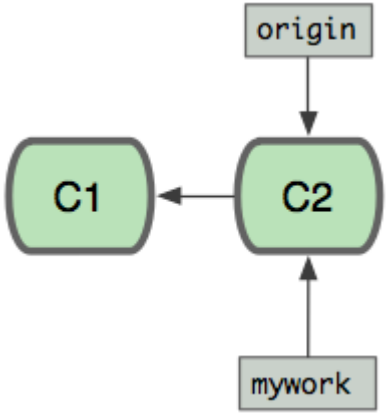
Просмотрите gitignore для детального объяснения синтаксиса. Вы можете также расположить .gitignore файлы в другой директории в вашем рабочем дереве, и их правила будут распространяться только на эти директории и их поддиректории. Файлы .gitignore как и любые другие файлы могут быть добавлены в репозиторий (выполните `git add .gitignore` и `git commit` как обычно), что удобно когда исключающие паттерны (такие как паттерны сравнивающие выходные файлы сборки)имеют смысл для других пользователей, которые склонируют ваш репозиторий.

Если вы хотите чтобы исключающие паттерны действовали только на определенные репозитории (вместо каждого репозитория для заданного проекта), то вы возможно положите их в файл в вашем репозитории под именем .git/info/exclude, или в любой файл определенный переменной настройки `core.excludesfile`. Некоторые команды git могут также брать исключающие шаблоны прямо из командной строки. Просмотрите gitignore для получения подробностей.

РЕБАЗИРОВАНИЕ

Предположим вы создаете ветку "mywork" на удаленной-отслеживаемой ветке "origin".

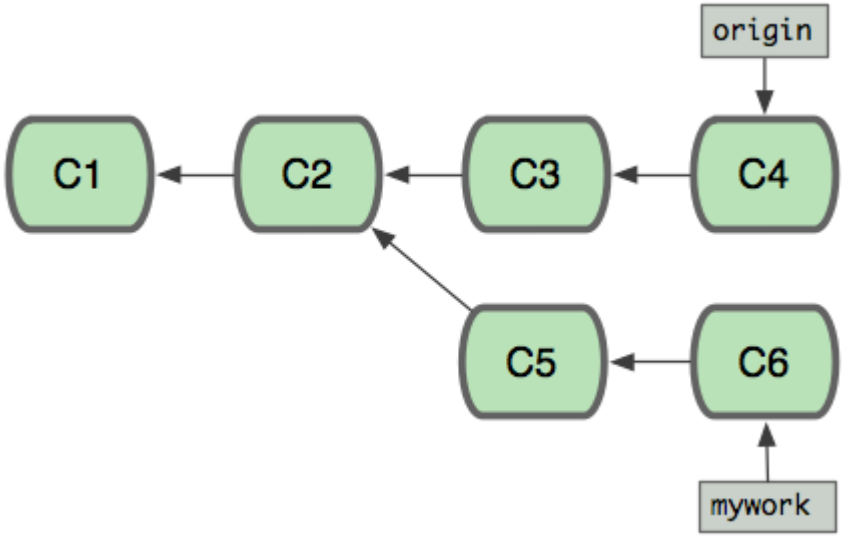
```
$ git checkout -b mywork origin
```



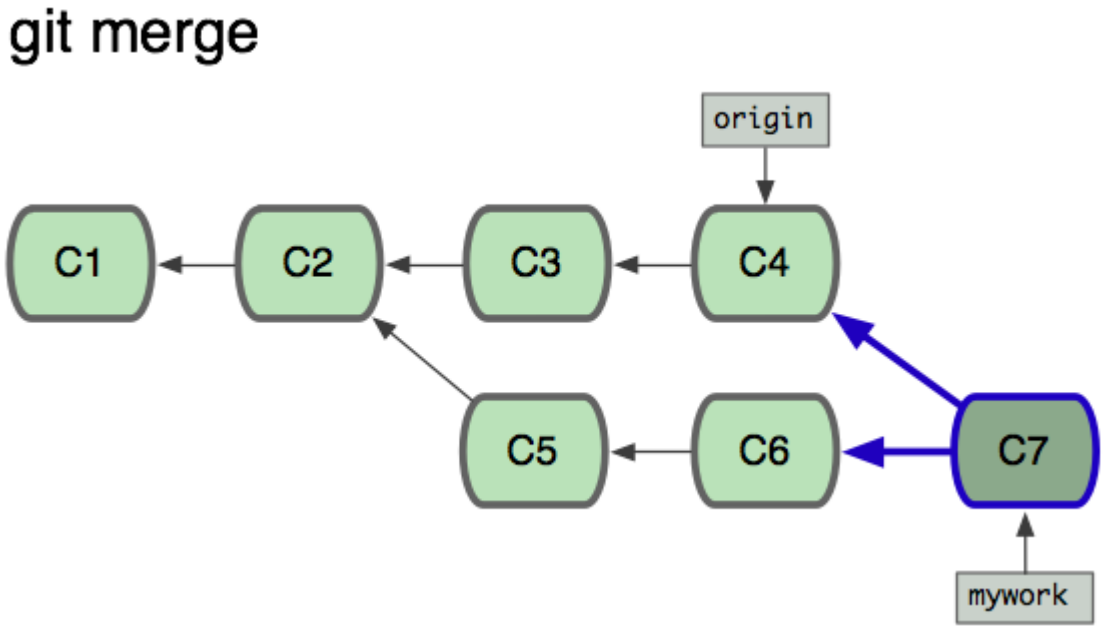
Терерь вы проделываете некоторую работу, создающую два новых коммита.

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

Тем временем, некто еще проделываает некоторую работу создающую два новых коммита в ветке origin. Это означает обе 'origin' и 'mywork' продивинулись вперед, что означает что ветки отклонились друг от друга.



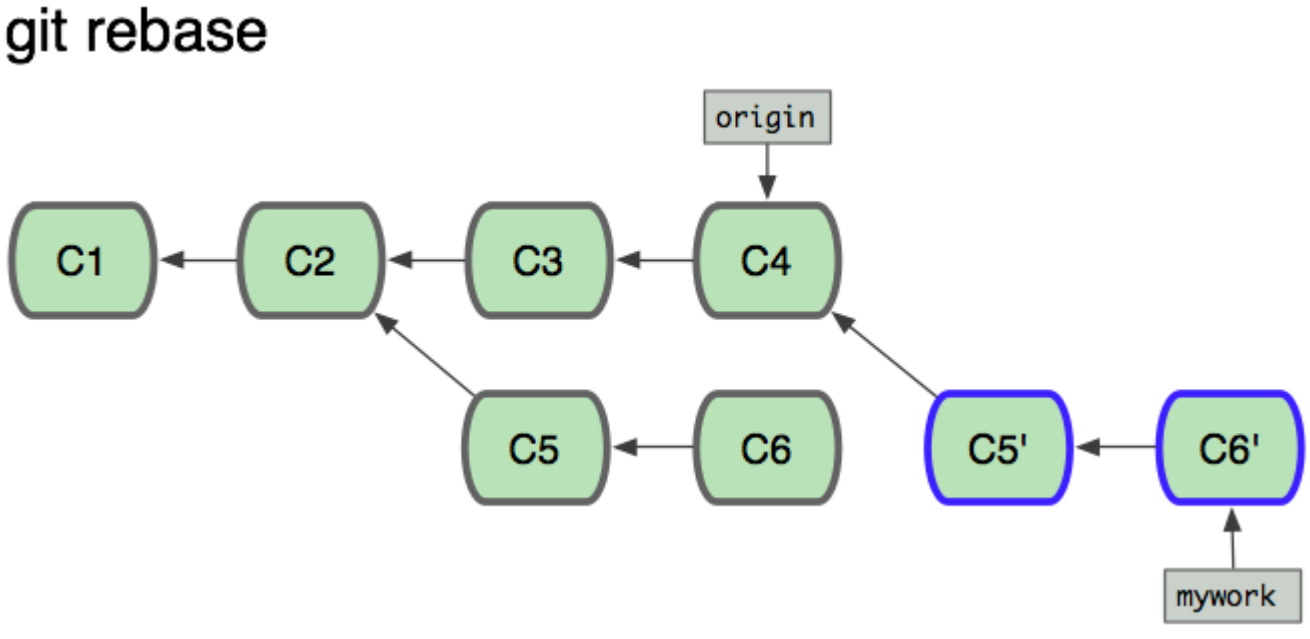
В этот момент, вы можете выполнить "pull" чтобы слить ваши изменения обратно, в результате отразуется новый слитый коммит, след. образом:



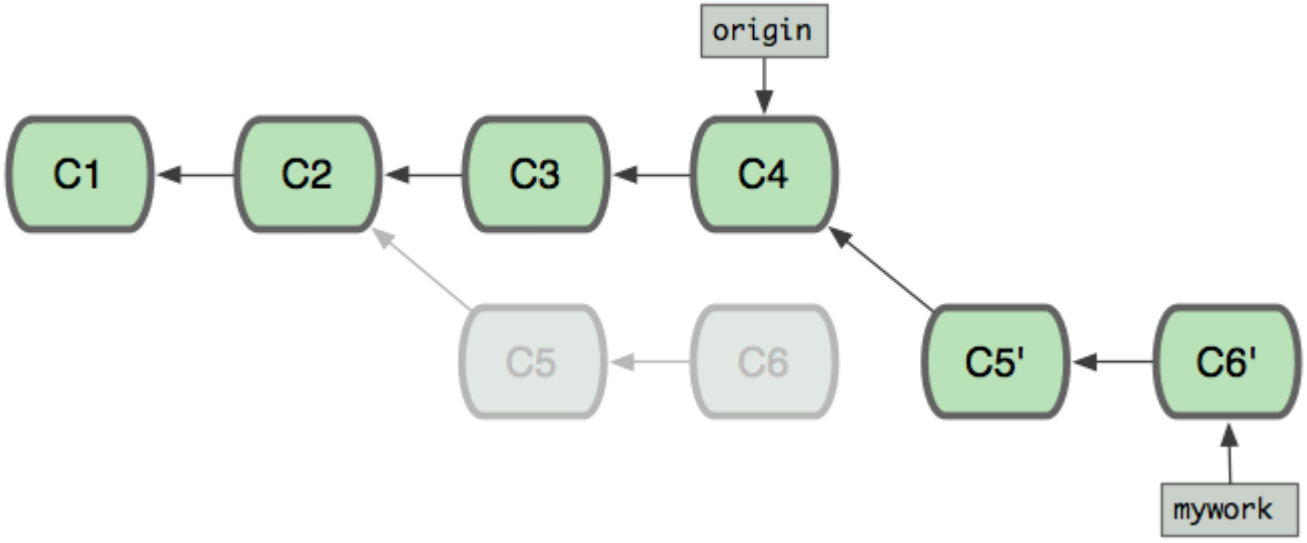
Тем не менее, если вы предпочитаете держить историю в mywork как простую серию коммитов без каких либо слияний, вы можете вместо этого использовать git rebase:

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

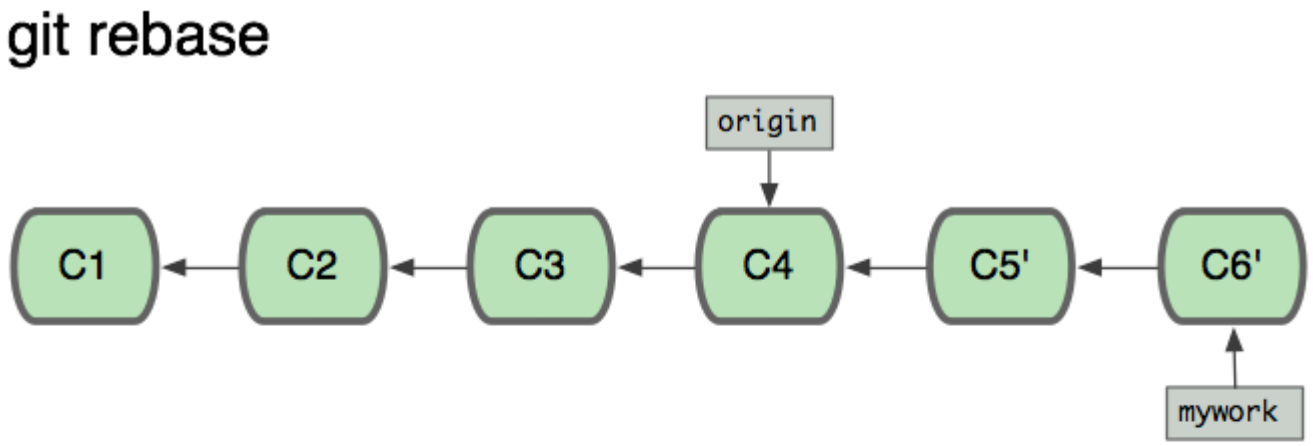
Это удалит каждый ваш коммит из mywork, временно сохранив их как патчи (в директории под именем ".git/rebase"), и обновит mywork до точки в последнней версии origin, затем применит каждый сохраненный ранее патч с новому mywork.

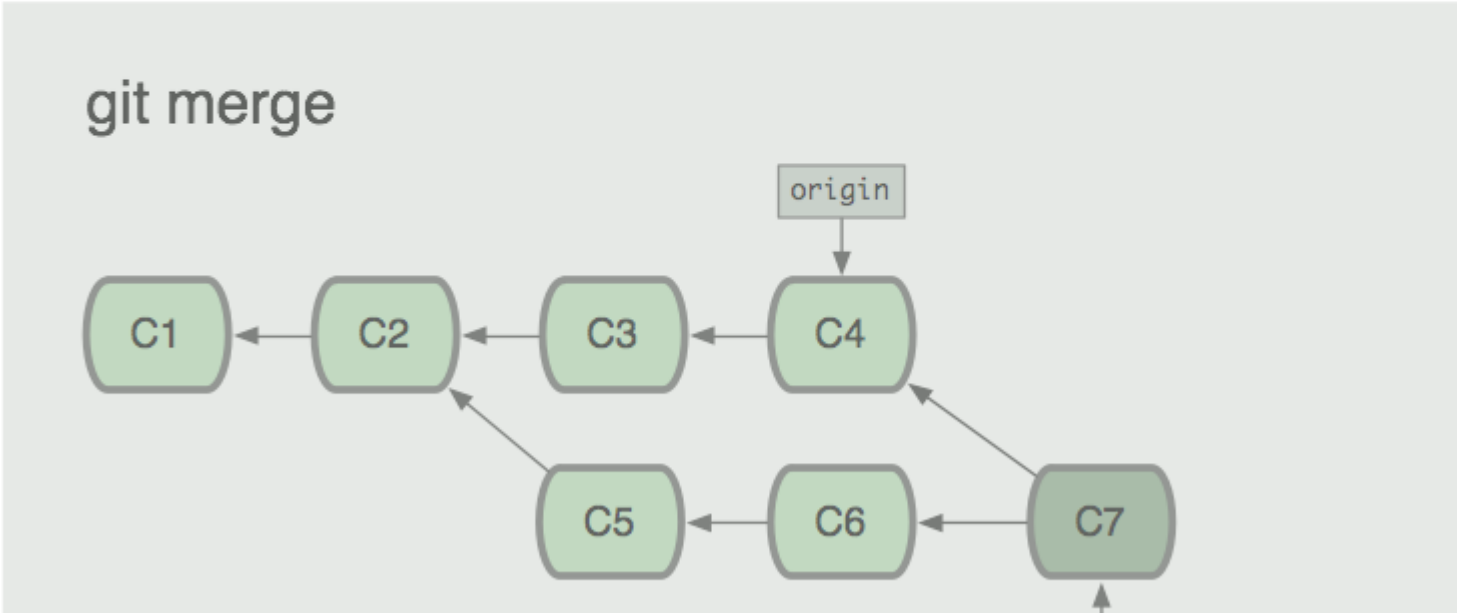


Как только ссылка ('mywork') обновлена и указывает на новые созданные коммит объекты, ваши ранние коммиты будут отменены. Они будут скорее всего удалены если вы выполните сборку мусора. (просмотрите git gc)



Таким образом мы теперь можем взглянуть на разницу в вашей истории между выполнением ребазирования и слияния:


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


git merge
origin
C1
C2
C3
C4
C5
C6
C7

В процессе ребазирования, возможно обнаружатся конфликты. В этом случае он остановится и позволит вам устранить конфликты; после чего, выполните "git-add" чтобы обновить индекс этого содержимого, и затем, вместо выполнения git-commit, просто запустите

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

и git продолжит налагать оставшиеся патчи.

В любой момент вы можете использовать параметр `--abort` чтобы оборвать этот процесс и вернуть mywork к тому состоянию в котором она была до того как вы начали процесс ребазирования:

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

gitcast:c7-rebase

ИНТЕРАКТИВНОЕ РЕБАЗИРОВАНИЯ

Вы также можете выполнять ребазирование интерактивно. Это часто используется чтобы переписать ваши коммит-объекты перед тем как выполнить их push. Это простой способ разделить, слить или переупорядочить коммиты перед тем как расшарить их с другими. Вы также можете использовать его чтобы чистить коммиты которые вы вытянули у кого либо и затем применить их на локальном репозитории..

Если у вас есть некоторые коммиты которые вы хотели бы модифицировать во время выполнения ребазирования, вы можете вызвать интерактивный диалог передав параметр '-i' или '--interactive' в команду 'git rebase'.

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

Это вызовет интерактивный режим выполнения операции ребазирования на все коммиты которые вы сделали с вашего последнего момента выполнения push (или слияния из репозитория origin).

Чтобы увидеть эти коммиты заранее, вы можете выполнить команду git log след. образом:

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

Как только вы выполнили команду 'rebase -i', тут же откроется ваш редактор по умолчанию с текстом похожим на что то в виде:

```
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.
#
```

Это означает что было сделано пять коммитов с последнего выполнения вами push и это даст вам одну линию на каждый коммит следующего формата:

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

Теперь, вы можете изменить действие (по умолчанию 'pick'-выбрать) на 'edit'-редактировать или 'squash'-сдавить или просто оставте это как 'pick'. Вы можете также переупорядочить коммиты просто передвигая линии так как вам этого хочется. Затем, когда вы выйдете из редактора, git попытается уложить коммиты тем образом каким вы это определили и выполнить действия определенные вами.

Если определен 'pick', то он просто попробует применить патч и сохранить коммит с тем же сообщением-описанием как и было до этого.

Если определено 'squash', то он будет комбинировать это коммит с предыдущим чтобы создать новый коммит. Вы опять попадете в редактор чтобы объединить оба сообщение-описания обоих коммитов сложив их вместе. Таким образом если вы выйдете из редактора с этим:

```
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
```

То вам нужно создать одно сообщение-описание коммита из этого:

```
# 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.
```

Как только вы отредактировали это в одно сообщение-описание и вышли из редактора, коммит будет сохранен с вашим новым сообщением-описанием.

Если определен 'edit', то процесс будет идти также, но приостановится перед тем как двигаться к следующему вас выбросит в коммандую строку и вы сможете изменить коммит, или изменить содержимое коммита некоторым образом.

Если вы хотите разделить коммит, например, вы определите 'edit' для этого коммита:

```
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
```

И затем когда вы попадете в командную строку, вы вернетесь к этому коммиту и создадите два (или более) новых. Давайте положим 21d80a5 изменил два файла , file1 и file2, и вы хотите разделить их в разные коммиты. Вы можете сделать это после того как выполнение перебазирования выкинуло вас в командную строку:

```
$ 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
```

И теперь вместо 5 коммитов, у вас их 6.

Последняя вещь которую интерактивное выполнение ребазирования может делать для вас - это сбрасывать коммиты. Если вместо того чтобы выбрать 'pick', 'squash' или 'edit' для линии коммита, вы просто удалите линию, это удалить коммит из истории.

ИНТЕРАКТИВНОЕ ДОБАВЛЕНИЕ

Интерактивное Добавление это весьма удобный способ работы с визуализацией индекса Git. Чтобы запустить его, просто напечатайте 'git add -i'. Git автоматически покажет вам ваши измененные файлы и их статус.

```
$>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>
```

В этом случае, мы можем видеть: что у нас есть 5 измененных файлов которые еще не добавлены в индекс (т.е. их статус unstaged), и даже как много строк было добавлено или удалено в каждый файл. Затем покажется интерактивное меню описывающее что мы можем сделать в этом режиме.

Если мы хотим добавить эти файлы в индекс, мы можем напечатать '2' или 'u' что бы перейти в режим обновления. Затем Я могу определить какие файлы поместить в индекс (добавить в индекс) напечатав цифру по числу файлов (в нашем случае, 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>>
```

Если Я нажму клавишу enter, Я попаду назад в главное меню где Я могу увидеть что статус файла изменился:

```
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
```

Теперь мы можем видеть что первые четыре файла проиндексированы а последний все еще нет. Это по сути краткий способ увидеть ту же информацию которую можно получить выполнив 'git status':

```
$ 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
#
```

Есть несколько полезных вещей которые мы можем сделать, включая удаление файлов из индекса (3: reverse - откат), добавление неослеживаемых файлов (4:add untracked - добавление неослеживаемых), и просмотр изменений (6: diff - изменения). Эти значения прямо соотвутствуют тому что они делают. Как бы там ни было есть одна замечательная команда, которая производит индексирует патчи (5: patch).

Если вы нажмете '5' или 'p' в меню, git покажет вам ваши diff(изменения) патч за патчем и спросит если вы желаете проиндексировать каждый патч. Этим способом вы можете фактически проидексировать для

коммита часть отредактированного файла. Если вы отредакталировали файл и хотите выполнить коммит только той части что изменили пропустив неизмененную часть, или выполнить коммит документации или отделить значительные изменения от просто добавленных или удаленных пробелов, вы можете выполнять 'git add -i' чтобы легко проделать это.

Здесь Я проиндексировал некоторые изменения в файле book_index_template.html, но не все:

```
           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
```

Когда вы окончили фиксацию изменений в индексе используя 'git add -i', просто завершите работу нажав (7: quit - выход) и затем выполните 'git commit' чтобы закоммитить индексированные изменения. Запомните **не** запускать 'git commit -a', так это сотрет все сделанные вами аккуратные изменения которые вы только что проделали и просто закоммитит все сразу.

gitcast:c3_add_interactive

ЗАМОРОЗКА КОДА

Во время работы над чем сложным, вы нашли какой то баг который в принципе очевидный и простой но не относится к той работе которой вы занимаетесь в данный момент, и вы хотите исправить его перед тем как продолжить работу. Вы можете использовать git stash чтобы заморозить текущее состояние вашей

работы, и позже, после того как исправите баг, (или опционально после исправления этого бага в другой ветке и потом возвращения назад), разморозить изменения.

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

Эта команда заморозит ваши изменения в `stash`, и сбросит вашу рабочее дерево и индекс на совпадающее с кончиком вашей текущей ветки. Затем вы можете приступить к исправлению.

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

После чего, вы можете вернуться над чем вы работали выполнив `git stash apply`:

```
$ git stash apply
```

Очередь заморозки

Вы также можете испльзовать заморозку чтобы создвать очереди замороженных изменений.
Если вы выполните 'git stash list' вы можете увидеть спискок из замороженных состояний сохраненных вами:

```
$>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
```

Затем вы можете приложить их индивидуально с помощью 'git stash apply stash@{1}'. Вы можете очистить список выполнив 'git stash clear'.

Примечание: stash - переводится как тайник но наверное лучше сказать что заморозить

GIT TREEISHES

Существует несколько способов сослаться на определенный коммит или дерево, несколько иных чем набор полного 40-значного sha значения. В Git, на них ссылаются как на 'treeish'.

Частичный Sha

Если sha-значение вашего коммита '`980e3ccdaac54a0d4de358f3fe5d718027d96aae`', git узнает его по любому из следующих идентификаторов:

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

Пока частичное sha-значение уникально - git не спутает его с другим (что весьма маловероятно если вы используете по крайней мере 5 символов), git дополнит частичное sha значение для вас.

Имена ветвей, тагов или удаленок

Вы всегда можете использовать имя ветки, тага или удаленки вместо sha значения, так как они просто указатели. Если ваша ветка master на коммите 980e3 и вы только что выполнили ее push в ветку origin и поставили таг 'v1.0', то все следующие указатели эквивалентны:

```
980e3ccdaac54a0d4de358f3fe5d718027d96aae
origin/master
refs/remotes/origin/master
master
refs/heads/master
```

```
v1.0
refs/tags/v1.0
```

Это означает содержимое вывода след.команд будет одинаковым:

```
$ git log master

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

Спецификация даты

Журнал ссылок (The Ref Log) который ведет git позволяет вам проделывать некоторые похожие вещи локально, например такие как:

```
master@{yesterday}

master@{1 month ago}
```

Это сокращение для 'туда где голова ветки master была вчера', и т.д. Заметьте что в этом формате результатом может быть разные sha значения на разных компьютерах, даже если ветка master в данный момент указывает на одно место.

Спецификация порядковая

Этот формат выдаст вам N-ое предыдущее значение определенной ссылки. Например:

```
master@{5}
```

выдаст вам 5-ое предшествующее значение ссылки головы ветки master.

Родитель морковь

Это выдаст N-ого родителя определенного коммита. Этот формат полезен только в случае слияния коммитов - объекты коммит которые имеют более одного прямого родителя.

```
master^2
```

Спецификация тильда

Спека тильда выдаст вам N-ого прародителя объекта коммита. Например,

```
master~2
```

наш результат первый родитель первого родителя коммита на который указывает master. Это эквивалентно:

```
master^^
```

Вы также можете продолжать это. След. спеки укажут на тот же самый коммит:

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

Указатель дерева

Этот неоднозначный коммит из дерева на которое он указывает. Если вы хотите sha значение на которое указывает этот коммит, вы можете добавить '{tree}' спеку в его конец.

```
master^{tree}
```

Спека блоб

Если вы желаете получить sha значение определенного блоба, то вы можете добавить путь блоба к концу treeish, след.образом:

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

Интервал

В заключении, вы можете определить интервал коммитов с помощью спеки интервал. Это даст вам все коммиты между 7b593b5 и 51bea1 (где 51bea1 наиболее свежий), исключая 7b593b5 но включая 51bea1:

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

Это включит каждый коммит начиная *с* 7b593b:

```
7b593b..
```

ОТСЛЕЖИВАЮЩИЕ ВЕТКИ

'Отслеживающая ветка' в Git это локальная ветка которая соединена с удаленной веткой. Когда вы выполняете push и pull на эту ветку, это автоматически выполняет push и pull для удаленной ветки с которой она соединена.

Используйте это если вы всегда выполняете pull из той же главной ветки в новую ветку, и если вы не хотите явно использовать "git pull ".

Команда 'git clone' автоматически конфигурирует ветку 'master' как отслеживающую ветку для 'origin/master' - ветка master в клонируемом репозитории.

Вы можете создавать отслеживающие ветки вручную добавляя параметр '--track' к команде 'branch' в Git.

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

Затем когда вы выполните:

```
$ git pull experimental
```

то это автоматически вытянет из ветки 'origin' и сольет 'origin/experimental' с вашей локальной веткой 'experimental' branch.

Подобно этому, когда вы выполните push в origin, то это выполнит push на который ваша ветка 'experimental' указывает к ветке origins ветки 'experimental', без обязательного определения ее.

ПОИСК С GIT GREP

Выполнить поиск файлов содержащих слово или фразу в Git очень просто с помощью команды git grep. Это возможно проделать и с помощью обычной Unix команды 'grep', но с помощью 'git grep' вы можете искать также в предыдущих версиях проекта без его извлечения в рабочую директорию.

Например, если Я захотел увидеть каждое место где был использован вызов 'xmmap' в моем репозитории git.git, Я могу выполнить это:

```
$ 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,
```

Если Я захотел увидеть также номер линии каждого совпадения, я могу добавить параметр '-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,
```

Если мы заинтересованы и в имени файла, мы можем передать параметр '--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
```

Мы также можем увидеть как много одинаковых линий в каждом файле с помощью параметра '-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
```

Теперь, если Я хочу увидеть где это было использовано в определенной версии git, Я могу добавить ссылку таг в конец:

```
$ 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
```

Мы можем видеть что есть некоторые отличия между текущими строками и строками в версии 1.5.0, одно из которых это то что xmmap теперь используется в wrapper.c в отличии от версии v1.5.0 где это было не так.

Мы также можем комбинировать условия поиска в grep. Положим мы захотели найти где в нашем репозитории определена SORT_DIRENT:

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

Мы также можем поискать каждый файл который удовлетворяет *обоим* условиям поиска, но отобразив каждую строку которая имеет условие *или* в этих файлах:

```
$ 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)
```

Мы также можем поискать строки которые содержут один термин и оба двух других, например мы захотели увидеть где мы определили постоянные которые имеют и тот и другой PATH или MAX в имени:

```
$ 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)
...
```

ОТМЕНА В GIT - СБРОС, ИЗВЛЕЧЕНИЕ И ОТКАТ

Git имеет множество способов исправить ошибку в процессе вашей работы. Выбор подходящего способа зависит от того провели вы коммит с ошибкой или нет, и если вы включили ошибку в коммит, то расшарили вы ошибочный коммит с кем еще либо.

Исправление незакоммиченных ошибок

Если вы испортили рабочее дерево, но не выполнили еще коммит, вы можете вернуть все рабочее дерево к состоянию на момент последнего коммита с помощью

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

Это отбросит все сделанные изменения которые вы возможно добавили в индекс git, а также все другие изменения в вашей рабочем дереве. Другими словами, результат этого - вывод команд "git diff" и "git diff --cached" будет пустым.

Если вы просто хотите восстановить только один единственный файл, предположим hello.rb, то выполните git checkout вместо

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

Первая команда восстановит hello.rb до версии хранящейся в индексе, и команда "git diff hello.rb" не покажет отличий. Вторая команда восстановит hello.rb до версии в ревизии HEAD, таким образом обе команды "git diff hello.rb" и "git diff --cached hello.rb" не покажут отличий.

Исправление закоммиченных ошибок

Если вы сделали коммит а позже вам захотелось чтоб его и не было, то есть два фундаментально разных способа осуществить это:

1. Вы создаете новый коммит в котором нет тех нежелательных частей присутствующих в старом коммите. Это правильный способ если ваша ошибка уже была опубликована.

2. Вы можете вернуться назад и изменить ваш старый коммит. Никогда не делайте этого если вы уже опубликовали вашу историю; git не предполагает изменения в **истории** проекта, и не может правильно выполнять повтор слияния из ветки, история которой была изменена.

Исправление ошибки новым коммитом

Создание нового коммита который восстанавливает предыдущее состояние очень просто; нужно только передать команду git revert ссылку на ошибочный коммит; например чтобы восстановить наиболее свежий коммит:

```
$ git revert HEAD
```

Это создаст новый коммит который отменяет изменения в HEAD. Вам предоставится шанс отредактировать сообщение-описание нового коммита.

Вы также можете восстановить предыдущие изменения, например, от слеж к прошлому:

```
$ git revert HEAD^
```

В этом случае git попробует отменить прошлые изменения, в то время как любые изменения сделанные потом остануться нетронутыми. Если более свежие изменения перекрывают с изменениями которые должны быть восстановлены, то git пропросит вас исправить конфликты вручную, таким же образом как в случае слияний.

Исправление ошибок изменяя коммит

Если вы только что выполнили коммит но осознали что вам нужно в нем что то исправить, последнии версии git commit поддерживают флаг **--amend** который указывает git заменит коммит HEAD новым коммитом, основываясь на текущем содержании индекса. Это дает вам удобный случай добавить файлы которые вы забыли добавить или исправить опечатки в сообщении-описании коммита, до того как вы выполните push и сделаете свои изменения доступными всему миру.

Если вы нашли ошибку в давнем коммите, но который еще не опубликовали, используйте git rebase в интерактивном режиме, с "git rebase -i" отмечающий изменения которые требуют коррекции с **edit**. Это позволит вам изменить коммит во время процесса rebase.

СОПРОВОЖДЕНИЕ GIT

Обеспечение хорошей производительности

На больших репоозиториях, git зависит от способа сжатия информации истории, чтобы не занимать много дискового пространства или памяти.

Это сжатие не выполнятеся автоматически. Поэтому вы должны иногда выполнять git gc:

```
$ git gc
```

чтобы заново сжать архив. Это может занять длительное время, так что вы возможно предпочтете выполнять git-gc тогда когда не заняты чем то другим.

Обеспечение достоверности

Команда git fsck выполняет некоторое количество проверок целостности репозитория, и составляет отчет если будут найдены какие либо проблемы. Это возможно займет некоторое время. Наиболее общее предупреждение в значительной степени будут о подвешенных объектах:

```
$ 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
...
```

Подвешенные объекты это не проблема. Самое худшее что они могут сделать это просто занять немного дополнительного дискового пространства. Иногда они могут быть вашим последним способом восстановить потерянную работу.

УСТАНОВКА ПУБЛИЧНОГО РЕПОЗИТОРИЯ

Предположим ваш персональный репозиторий находится в директории ~/proj. Сначала мы создадим клон репозитория и скажем демону git (git-daemon) что он предназначен быть публичным:

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

Результирующая директория proj.git содержит "пустой" репозиторий git-- это только содержимое директории ".git", без каких либо файлов.

Затем скопируем proj.git на сервер где вы планируете хостить публичный репозиторий. Вы можете использовать scp, rsync, или что нибудь более привычное или удобное.

Экспорт репозитория git через протокол git

Это наиболее предпочтительный способ.

Если кто либо администрирует сервер, то они должны сказать вам в какую директорию поместить репозиторий, и что появится URL git://.

Иначе, все что вам нужно сделать это запустить git daemon; он будет слушать порт 9418. По умолчанию, он разрешит доступ в любую директорию которая выглядит как репозиторий git и содержит волшебный файл git-daemon-export-ok. Передача путей директорий как аргументы в git-daemon ограничит экспорт до этих путей.

Вы также можете запустить git-daemon как сервис inetd; просмотрите документацию по git daemon чтобы получить больше подробностей. (Обратите особое внимание на секцию примеров.)

Экспорт репозитория git через http

Протокол git дает лучшую производительность и надежность, но на хосте с установленным вебсервером, экспорт http будет проще установить.

Все что вам нужно сделать это положить новый созданный пустой репозиторий git в директорию которая доступна вебсерверу, и скорректировать немного чтобы дать веб клиентам эктра информацию которая им нужна:

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

(Для объяснения последних двух строк, просмотрите git update-server-info и githooks.)

Объявите URL proj.git. Потом любой может склонировать или вытянуть репозиторий, просто выполнив команду:

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

КОНФИГУРИРОВАНИЕ ЧАСТНОГО РЕПОЗИТОРИЯ

Если вам потребуется сконфигурировать частный репозиторий и вы предпочитаете расположить его локально нежели чем на выделенном хосте, то у вас есть несколько вариантов.

Доступ в репозиторий через SSH

Обычно, простейшее решение это использовать Git через SSH. Если пользователи уже имеют ssh аккаунт на машине, вы можете разместить git репозиторий в таком месте куда они имеют доступ. Таким образом они могут легко получить доступ к нему через ssh логин. Например, скажем у вас есть репозиторий которым вы хотите управлять. Вы можете экспортировать его как простой репозиторий и затем безопасно скопировать(используя scp) его на ваш сервер след.образом:

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

Затем кто-нибудь еще имеющий аккаунт ssh на myserver.com может склонировать репозиторий выполнив:

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

Что просто пригласит их ввести ssh пароль или использовать их публичный ключ, мы исходим из того что ssh у этих пользователей сконфигурирован.

Множественный доступ пользователей с помощью Gitosis

Если вы не хотите настраивать отдельные аккаунты для каждого пользователя, вы можете использовать инструмент Gitosis. В gitosis, существует файл authorized_keys который содержит публичные ключи всех

авторизированных на доступ к репозиторию пользователей, и потом все они используют пользователя 'git' чтобы выполнять push и pull.

Установка и Конфигурация Gitosis

Chapter 5

Продвинутый уровень Git

СОЗДАНИЕ НОВЫХ ПУСТЫХ ВЕТВЕЙ

Иногда, вам возможно понадобятся ветки в вашем репозитории, которые не расшаривают предка с вашим обычным кодом. Пример такого случая сгенерированная документация или что нибудь в этом роде. Если вам требуется создать новую ветку которая не содержит ваш текущий код как родителя, вы можете это проделать след.образом:

```
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

ИЗМЕНЕНИЕ ВАШЕЙ ИСТОРИИ

Интерактивное выполнение rebase это хороший способ изменить отдельные коммиты.

git filter-branch хороший способ чтобы редактировать коммиты все вместе.

ПРОДВИНУТАЯ РАБОТА С ВЕТКАМИ И СЛИЯНИЕМ

Получение помощи решения конфликтов во время слияния

Все изменения которые git способен слить автоматически уже добавлены в индекс, так что git diff показывает только конфликты. Эта команда использует необычный синтаксис:

```
$ 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
```

Отменить коммит который будет выполнен после того как мы разрешим этот конфликт будет иметь двух родителей вместо обычного одного: один родитель будет HEAD, конец текущей ветки; другой будет конец другой ветки, которая сохранится временно в MERGE_HEAD.

Во время слияния, индекс содержит три версии каждого файла. Каждый их этих "проиндексированных файлов" представляет разную версию файла:

```
$ git show :1:file.txt  # файл которой потомок обоих файлов из сливаемых ветвей
$ git show :2:file.txt  # версия из HEAD.
$ git show :3:file.txt  # версия и сливаемой ветки MERGE_HEAD.
```

Когда вы запросите git diff показать конфликты, он запускает diff в три пути, между конфликтовавшем результом слияния в рабочем дереве с индексом 2 и 3 чтобы показать только те куски кода, содержимое которых приходит из обоих ветвей, смешанным (другими словами, когда результаты слияния целиком приходят только из индекса 2, то эта часть не конфликтующая и не показывается. То же самое для индекса 3).

Команда diff выше показывает отличия между версией рабочего дерева fie.txt и версиями индексов 2 и 3. Таким образом вместо того чтобы предварять каждую строку одним "+" или "-", теперь используются две колонки: первая колонка используется для отличий между первым родителем и копией рабочей директории и вторая для отличий между вторым родителем и копией рабочего дерева. (Просмотрите секцию "COMBINED DIFF FORMAT" git diff-files для получения дополнительных подробностей формата.)

После улаживания конфликта очевидный способ (но перед обновлением индекса), diff будет выглядеть след.образом:

```
$ 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
```

Это показывает что наша исправленная версия удалила "Hello world" из первого родителя, удалила "Goodbye" из второго родителя, и добавила "Goodbye world", которая до этого отсутствовала в обоих.

Некоторые особые параметры diff позволяют выполнять diff рабочей директории от любых других этих индексов:

```
$ git diff -1 file.txt      # diff against stage 1
$ git diff --base file.txt  # same as the above
$ git diff -2 file.txt      # diff against stage 2
$ git diff --ours file.txt  # same as the above
$ git diff -3 file.txt      # diff against stage 3
$ git diff --theirs file.txt    # same as the above.
```

Команды git log и gitk также предлагают специальную помощь для слияний:

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

Это покажет все коммиты которые существуют только в HEAD или в MERGE_HEAD, и которые затрагивают неслитый файл.

Вы можете также использовать git mergetool, которая позволяет вам сливать неслитые файлы используя внешние инструменты такие как emacs или kdiff3.

Каждый раз как только вы разрешили конфликты в файле, обновите индекс:

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

и тогда разные версии этого файла в индексе будут "разрушены", после чего git-diff не будет более (по умолчанию) показывать diff-ы для этого файла.

Множественное слияние

Вы можете сливать несколько веток одновременно просто перечисляя их в git merge. Например,

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

что эквивалентно:

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

Поддерево

Есть ситуации где вы захотите включить содержимое из стороннего проекта в свой собственный. Вы можете просто выполнить pull из другого проекта если нет конфликтов.

Проблемный случай это когда есть конфиликтующие файлы. Потенциальный кандидат это Makefiles и другие стандартные файлы. Вы можете слить эти файлы но скорее всего не захотите этого делать. Лучшее решение для этой проблемы это слить проект как его собственную поддиректорию. Это не поддерживается стратегией рекурсивного слияния, но просто выполнить pull не сработает.

То что вам нужно это стратегия слияния поддерева, которая поможет вам в таких ситуациях.

В этом примере, давайте скажем у вас есть репозиторий в /path/to/B (но это также может быть и URL если вы пожелаете). Вы хотите слить ветку master этого репозитория в поддиректорию dir-B в вашей текущей ветке.

Здесь последовательность команд которая вам нужна:

```
$ 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)
```

Выгода от использования поддерева слияния это то что оно требует меньше административной заботы от пользователей вашего репозитория. Это работает со старшими (до Git v1.5.2) клиентами и у вас есть код сразу после клонирования.

Как бы там ни было если вы используете подмодули, то вы можете выбрать не перемещать объекты подмодулей. Это проблема возможна со слиянием поддерева.

Также, в случае если вы сделали изменения в другом проекте, проще предоставить изменения если вы используете подмодули.

(from Using Subtree Merge)

ПОИСК ПРОБЛЕМ - GIT BISECT

Предположим версия 2.6.18 вашего проекта работала, но версия в ветке master падает. Иногда самый лучший способ найти причину этой регрессии(проблемы) это выполнить брут-форс поиск сквозь всю историю проекта, чтобы найти тот коммит который вызывает крах приложения. Команда git bisect поможет вам в этом:

```
$ 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
```

Если вы выполните "git branch" в этот момент, вы увидите что git переместил вас временно в новую ветку с имененм "bisect". Эта ветка указывает на коммит (коммит с id 65934...) который можно достичь из ветки "master" но не из v2.6.18. Скомпилируйте и протестируйте его, и узнайте падает ли программа. Положим что да, тогда:

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

Извлекает старую версию. Продолжайте в этом духе, уведомляя git на каждом этапе что версия которую вы пробуете нормальная или проблемная, и заметьте что число повторных просмотров оставшихся для тестирования уменьшается приблизительно наполовину каждый раз.

После приблизительно 13 тестов (в этом случае), вы получите в результате id коммита-виновника аварий. Вы можете затем исследовать коммит с помощью git show, чтобы найти того кто создал его, и написать отчет об ошибке с id коммита. В заключении выполните

```
$ git bisect reset
```

чтобы вернуть вас в ветку в которой вы были до этого и удалить временную ветку "bisect".

Заметьте что версия которую git-bisect извлекает для вас на каждом этапе просто предположение, и вы свободны попробовать другую версию если вы думаете это хорошая идея. Например, иногда вы приземлитесь на коммит который ломает что то неопределенное; выполните

```
$ git bisect visualize
```

которая выполнит gitk и укажет выбранный ею коммит маркером который говорит "bisect". Выберите безопасно выглядящий коммит рядом, отметьте id этого коммита, и извлеките его выполнив:

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

затем протестируйте, выполните "bisect good" или "bisect bad" как более подходящее, и продолжите.

ПОИСК НЕИСПРАВНОСТЕЙ - GIT BLAME

Команда git blame это действительно полезная команда, чтобы найти того кто изменил определенную часть файла. Вы просто выполните 'git blame [filename]' и получите вывод всех данных файла включая sha значение последнего коммита, дату и автора каждой строки файла.

```
$ 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
...
```

Часто полезно увидеть кто последним изменил файл, если в файле изменения ломают сборку приложения.

Вы также можете определить для этой команды начальную и конечную строки:

```
$>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 И ЭЛ.ПОЧТА

Отправка патчей в проект

Если вы только что сделали несколько изменений, простейший способ передать их это оправить их как патчи по эл.почте:

Первое выполните git format-patch; например:

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

это снерерирует группу файлов в текущей директории, один файл для каждого патча в текущей ветке но не в origin/HEAD.

Затем вы можете импортировать их в вашу почтовую программу и послать их вручную. Если у вас слишком большое количество патчей, предпочтительнее будет выполнить скрипт git send-email чтобы автоматизировать этот процесс. Проконсультируйтесь с почтой рассылки вашего проекта чтобы сначала определить каким образом мантейнеры предпочитают получать патчи.

Импортирование патчей в проект

Git также предоставляет инструмент называющийся git am (am здесь означает "apply mailbox"), для импортирования таких групп патчей полученных по эл.почте. Просто сохраните все сообщения содержащие патчи, по порядку, в единый файл, скажем "patches.mbox", и затем выполните

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

Git наложит каждый патч по порядку; если будут найдены какие-либо конфликты, то процесс остановится, и вы можете вручную исправить конфликты и выполнить слияние. (Параметр "-3" указывает git выполнить слияние; если вы предпочитаете просто прервать операцию оставив ваше дерево и индекс нетронутыми, вы можете пропустить этот параметр.)

Как только индекс обновлен результатами исправления конфликтов, вместо создания нового коммита просто выполните

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

и git создаст коммит для вас и вы сможете продолжить накладывать оставшиеся патчи из эл.почты.

Окончательный результат будет группой коммитов, по одному коммиту на каждый патч из эл.почты, вместе с авторством и логом сообщением коммита, каждое будет взято из сообщения содержащего каждый патч.

НАСТРОЙКА GIT

git config

Изменение вашего редактора

Это изменит редактор вызываемый git по умолчанию

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

Добавление алиасов

```
$ 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
```

Добавление цветов

Просмотрите все параметры документации color.* git config

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

Или, вы можете установить их все с помощью параметра color.ui:

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

Коммит шаблон

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

Форматирование лога

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

Другие настройки

Существуют также и другие интересные настройки для операций упаковки, gc-ing(сборщик мусора git), слияния, удаленнок, ветвей, http transport, diffs, страниц, пробелов и многие другие. Если вы хотите настроить их, просмотрите документацию git config.

ХУКИ GIT

Хуки это короткие скрипты которые вы можете разместить в директории $GIT_DIR/hooks и которые будут запускаться по определенному событию. Когда выполняется git-init, то набор хуков идущих в комплекте с git копируется в директорию hooks нового репозитория, но по умолчанию они деактивированы. Чтобы активировать хуки, их следует переименовать, удалив их расширение .sample.

applypatch-msg

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

Этот хук вызывается скриптом git-am. Он принимает единственный параметр - имя файла, который содержит лог сообщение коммита. Выход с ненулевым статусом прерывает процесс выполнения команды git-am до наложения патча.

Хуку разрешается редактировать файл сообщение, и он может быть использован чтобы составить сообщение в особом формате, стандартном для проекта(если таковой формат имеется). Он также может быть использован чтобы отказаться от коммита после проверки файла сообщения. Хук по умолчанию applypatch-msg, когда разблокирован, запускает хук commit-msg, если последний разблокирован.

pre-applypatch

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

Этот хук вызывается git-am. Он не берет аргументы, и вызывается после того как патч уже наложен, но до того как будет выполнен коммит. Если он завершается с ненулевым статусом, тогда коммит рабочего дерева не будет выполнен после наложения патча.

Он может быть использован для проверки текущего рабочего дерева и может отказать в выполнении коммита если оно не пройдет определенные тесты. По умолчанию pre-applypatch хук, когда разблокирован, выполняет хук pre-commit, если последний разблокирован.

post-applypatch

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

Этот хук вызывается 'git-am'. Он не берет параметров, и вызывается после того как наложен патч и выполнен коммит.

Назначение хука главным образом уведомление, и не может влиять на последствия выполнения 'git-am'.

pre-commit

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

Этот хук вызывается 'git-commit', и его вызов может быть отменен с помощью параметра `\--no-verify`. Он не берет параметров, и активируется до получения предложения лог-сообщения коммита и выполнения коммита. Завершение скрипта с ненулевым статусом прервет выполнение 'git-commit'.

Дефолтовый 'pre-commit' хук, когда разблокирован, ловит ввод строк с пробелами в окончании и прерывает коммит когда такая строка найдена.

Все хуки 'git-commit' вызываются с переменной окружения `GIT_EDITOR=:` если команда не вызывает редактор чтобы модифицировать сообщение-описпние коммита.

Здесь пример Ruby скрипта который выполняет тесты RSpec перед тем как позволить выполнить коммит.

```ruby
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
```

Этот хук активируется 'git-commit' сразу после подготовки дефолтного лога сообщения, и перед тем как откроется редактор.

Он берет от одного до трех параметров. Первый это имя файла содержащего сообщение-описание коммита. Второй источник сообщения коммита, и может быть: 'message' (если параметри `-m` или `-F` был передан); `template` (если параметр `-t` был передан или установлен параметр настройки `commit.template`); `merge` (если коммит это слияние или существует файл `.git/MERGE_MSG`); `squash` (если существует файл `.git/SQUASH_MSG`); или `commit`, со следующим за ним SHA1 значением коммита (если передан параметр `-c`, `-C` или `\--ammend`)..

Если статус окончания работы ненулевой, выполнение 'git-commit' прервется.

Цель хука отредактировать файл сообщение на месте, и это не подавляется передачей параметра `\--no-verify`. Ненулевой статус окончания выполнения означает сбой хука и прерывание коммита. Он не должен быть использован как замена хука pre-commit.

Образец хука `prepare-commit-msg` который поставляется с комментариями git вне части `Conflicts:` сообщения описания коммита слияния.

commit-msg

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

Этот хук вызывается 'git-commit', и может быть обойден с помощью параметра `\--no-verify`. Он берет единственный параметр, имя файла который содержит предполагаемое сообщение описание коммита. Выход с ненулевым статусом прерывает выполнение 'git-commit'.

Этому хуку позоволяется редактировать файл сообщение на месте, и он может быть испльзован чтобы привести сообщение в стандартный вид для проекта(если таковой имеется). Он также может быть использован чтобы отказаться от коммита после проверки файла сообщения.

Хук по умолчания 'commit-msg', когда активирован, определяет дубликаты строк "Signed-off-by", и прерывает коммит если найдет таковую.

post-commit

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

Этот хук вызывается 'git-commit'. Он не берет параметров, о вызывается после того как коммит уже завершен.

Назначение этого хука главным образом уведомление, и он не может повлиять на результат работы 'git-commit'.

pre-rebase

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

Этот хук вызывается 'git-rebase' и может быть использован чтобы предотвратить выполнение операции ребазирования в ветке.

post-checkout

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

Этот хук вызывается когда 'git-checkout' выполняется после обновления рабочего дерева. Хук требуется три параметра: ссылка на предыдущую HEAD, ссылка на новую HEAD (которая возможно была изменена а возможно и нет), и флаг показывающий была ли операция checkout с веткой (смена ветки, flag=1) или она была проведена над файлом (получение файла из индекса, flag=0). Этот хук не может повлиять на результат 'git-checkout'.

Этот хук может быть использован чтобы выполнять проверки правильности репозитория валидность, автоматический показ отличий от предыдущей HEAD если отличается, или набор свойств метаданных рабочей директории.

post-merge

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

Этот хук вызывается 'git-merge', что происходит когда 'git-pull' закончил свою работу в локальном репозитории. Хук берет единственный аргумент, флаг статуса определяющий было или нет выполненное

слияние squash слиянием. Этот хук не может влиять на результат 'git-merge' и не выполняется если слияние потерпело неудачу вследствии конфликтов.

Этот хук может быть использован совместно с соотвествующим хуком pre-commit чтобы сохранять и восстанавливать метаданные ассоциированные с рабочим деревом в любой форме (в т.ч. разрешения/владение, ACLS и т.д.) Просмотрите contrib/hooks/setgitperms.perl чтобы на примере изучить как проделывать это.

pre-receive

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

Этот хук вызывается 'git-receive-pack' на удаленный репозиторий, что происходит когда заканчивается выполнение 'git-push' на локальном репозитории. Сразу после начала обновления refs(сслылок) на удаленном репозитории, вызывается хук pre-receive. Его статус выхода определяет успех или неудачу обновления.

Этот хук выполняется однажды для операции получения. Он не принимает аргументов, но чтобы обновить каждую ref(ссылку) он принимает в стандартный ввод строку след.формата:

SP SP LF

где `<old-value>` это старое имя объекта сохраненного в ref, `<new-value>` это имя нового объекта который будет сохранен в ref и `<ref-name>` это полное имя ref. Когда создается новая ref, `<old-value>` принимает значение 40 `0`.

Если хук завершает выполнение с ненулевым статусом, ни одна из refs(ссылок) будет обновлена. Если хук завершится с нулевым статусом, то обновление индивидуальных refs все еще можно предотвратить хуком <<update,'update'>>.

Оба стандарный вывод и стандартный вывод ошибок перенаправлены на 'git-send-pack' на другой стороне, так что вы можете просто посылать `echo` сообщения для пользователя.

Если вы написали его на Руби, вы можете получить аргументы след.образом:

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

Или в bash скрипт, аналогичный тому что ниже тоже будет работать:

```
#!/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
```

Этот хук вызывается 'git-receive-pack' на удаленном репозитории, что происходит когда выполнение 'git-push' завершилось на локальном репозитории. Перед самым обновлением ссылок на удаленном репозитории, вызывается хук обновления. Его статус окончания выполнения определяет успешно или неудачно обновилась ссылка.

Этот хук выполняется один раз для обновления каждой ссылки, и он принимает три параметра:

- имя обновляемой ссылки,
- имя старого объекта хранящегося в ссылке,
- и новое имя объекта которое сохранится в ссылке.

Нулевой выход из хука обновления позволяет обновиться ссылке. Выход с ненулевым статусом предотвратит обновление этой ссылки хуком 'git-receive-pack'

Этот хук может быть использован чтобы предотвратить 'forced(вынужденное)' обновление определенных ссылок, с помощью проверки что имени объекта соотвествует объект коммит, который является потомоком объекта коммита названного по имени старого объекта. Это для того чтобы заставить работать правило "только fast-forwarding".

Он также может быть использован что логгировать старый..новый статус. Как бы там ни было, он не знает все множество ветвей, и закончит посылая одно эл.сообщение на каждую обновленную ссылку, когда используется напрямую. Хук <<post-receive,'post-receive'>> лучше подходит для этого.

Другое использование предложеное в списке рассылки, использовать этот хук чтобы реализовать контроль доступа, который будет лучше гранулирован чем тот что используется на основе групп в файловой системе.

Оба стандартный вывод и стандартные вывод ошибок перенаправлены с другого конца в 'git-send-pack', так что вы легко можете `посылать echo` сообщения для пользователя.

Хук 'update' по умолчанию, когда активирован--и с включенным конфигурационным параметром `hooks.allowunannotated`--предотвращает выполнение push неаннотированных тагов.

post-receive

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

Этот хук вызывается 'git-receive-pack' на удаленном репозитории, что происходит когда завершается 'git-push' на локальном репозитории. Он выполняется на удаленном репозитории один раз после того как все ссылки будут обновлены.

Этот хук выполняется один раз для операции получения. Он не принимает аргументов, но получает туже информацию что и хук <<pre-receive,'pre-receive'>> на его стандартный ввод.

Этот хук не влияет на результат 'git-receive-pack', так как он вызывается после того как работа фактически уже окончена.

Он замещает хук <<post-update,'post-update'>> таким образом, он получает оба старые и новые значения всех ссылок в добавок к их именам.

Оба стандартный вывод и стандартный вывод ошибок перенаправлены с другой стороны на 'git-send-pack', так что вы можете просто посылать `echo` сообщения пользователю.

Дефолтовый хук 'post-receive' пуст, но существует скрипт образец `post-receive-email` в директории `contrib/hooks` идущий по умолчанию в комплекте с git, который реализует отпраление коммитов по эл.почте.

post-update

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

Этот хук вызвается 'git-receive-pack' на удаленном репозитории, и происходит когда завершается выполнение 'git-push' на локальном репозитории. Он выполняется на удаленном репозитории один раз после того как все ссылки будут обновлены.

Он берет переменное число параметров, каждый из которых это имя ссылки которая фактически была обновлена.

Это хук главным образом назначается для уведомлений, и не может повлиять на результат 'git-receive-pack'.

Хук 'post-update' может сообщить для каких HEAD было выполнена операция push, но он не знает каковы их оригинальные и обновленные значения, так что это не самое лучшее место чтобы логировать. Хук <<post-receive,'post-receive'>> получает оба оригинальные и обновленные значения ссылок. Вам лучше использовать его если они вам нужны.

Когда активирован, хук 'post-update' по умолчанию запускает 'git-update-server-info' чтобы поддерживать информацию обновленной использованную транспотными протоколами (e.g., HTTP). Если вы публикуете репозитоириий git который доступен по протоколу HTTP, то возможно вы должны разрешить этот хук.

Оба стандартный вывод и стандартный вывод ошибок перенаправлены на 'git-send-pack' с другого конца, так что вы можете просто посылать `echo` сообщения для пользователя.

pre-auto-gc

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

Это хук вызывается 'git-gc --auto'. Он не принимает параметров, и выход с ненулевым статусом из этого скрипта вызывает прекращение работы 'git-gc --auto'.

References

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

Недопереведено

ВОСТАНОВЛЕНИЕ ПОВРЕЖДЕННЫХ ОБЪЕКТОВ

Recovering Lost Commits Blog Post

Recovering Corrupted Blobs by Linus

ПОДМОДУЛИ

Большие проекты часто составлены из маленьких, самодостаточных модулей. Например, дерево исходных кодов дистрибутива "Встроенный Linux" включает каждую часть програмного обеспечения обычного дистрибутива с некоторыми изменениями; сборка видео проигрывателя возможно потребует только определенную, рабочую версию библиотеки декомпресии; несколько независимых программ возможно пользуются совместно одним наборов скриптов для сборки.

С централизованной системой контроля версий это часто достигается включением каждого модуля в единый репозиторий. Разработчики могут извлекать все модули или только модули необходимые им для работы в данный момент. Они могут даже формлять модифицированные файлы в нескольких модулях в единый коммит, в то время как вокруг перемещаются файлы или обновляется API и переводы..

Git не позволяет выполнять частичное извлечение, и дублирование этого подхода в Git заставит разработчиков поддерживать локальные копии модулей которые они не трогают. Коммиты в таких огромных репозитория будут намного медленнее чем вы ожидаете, так как Git должен будет просканировать каждую директорию на наличие изменений. Если модули имеют длительную локальную историю то клонирование может занять вечность.

Распределенные системы контроля версий имеют одну положительную сторону, они могут быть гораздо легче интегрированы с внешними источниками. В централизованной модели, один произвольный снапшот внешнего проекта экспортирован из его собственной системы контроля версий и затем импортирован в локальную систему контроля версий в ветку vendor. Вся история скрыта. С распределенной системой контроля версий вы модете склонировать всю внешнюю историю и даже больше просто следуя разработке и пере-слиянию локальных изменений.

Поддержка в Git подмодулей позволяет репозиторию содержать, как поддиректорию, извлеченный внешний проект. Подмодули поддерживают свою подлинность; поддержка подмодулей хранит

расположение репозитория подмодуля и ID коммита, так что другие разработчики которые клонируют существующий проект ("superproject")могут легко клонировать все подмодули при извлечении. Частичные извлечения superproject возможны: вам можете указать Git клонировать ничего, часть или все подмодули.

Команда git submodule доступна начиная с версии Git 1.5.3. Пользователи с Git 1.5.2 могут поискать коммиты подмодулей в репозитории и вручную извлечь их; ранние версии Git не узнают подмодули вообще..

Чтобы увидеть как работает поддержка подмодулей, создайте (для примера) 4 образца репозиториев которые могут быть использованы позже как подмодуль:

```
$ 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
```

Теперь создайте superproject и добавте все подмодули:

```
$ mkdir super
$ cd super
$ git init
$ for i in a b c d
do
```

```
    git submodule add ~/git/$i $i
done
```

Замечание: Не используйте здесь локальные URL если вы планируете опубликовать ваш superproject!

Посмотрим какие файлы создал `git-submodule`:

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

Команда `git-submodule add` выполняет пару вещей:

- Она клонирует подмодули в текущую директорию и по умолчанию извлекает ветку master.
- Она добавляет путь клона подмодуля к файлу gitmodules и добавляет этот файл в индекс, подготавливая все к коммиту.
- Она добавляет текущий ID коммита подмодуля в индекс, подготавлива все к коммиту.

Выполним коммит superproject:

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

Теперь склонируем superproject:

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

Директории подмодуля здесь, но они пустые:

```
$ ls -a a
.  ..
```

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

Замечание: Имена объектов коммит показанные выше будут отличаться от ваших, но они должны совпадать с именами объектов коммит HEAD ваших репозиториев. Вы можете проверить это выполнив `git ls-remote ../git/a`.

Снос подмодулей это двух шаговый процесс. Сначала выполните `git submodule init` чтобы добавить URL репозитория подмодуля в `.git/config`:

```
$ git submodule init
```

Теперь используйте `git-submodule update` чтобы склонировать репозитории и извлечь коммиты определенные в superproject:

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

Одна существенная разница между `git-submodule update` и `git-submodule add` это то что `git-submodule update` извлекает определенный коммит, чем кончик ветки. Это как извлечение тага: голова отделена, так что вы не работает на ветке.

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

Если вы хотите сделать изменения внутри подмодуля и у вас есть отсоединенная голова, тогда вы должны создать или извлечь ветку, сделать изменения, опубликовать изменения внутри подмодуля, и затем обновить superproject чтобы он указывал на новый коммит:

```
$ git checkout master
```

или

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

затем

```
$ 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
```

Вы должны выполнить `git submodule update` после `git pull` если вам также нужно обновить подмодули.

Заблуждения с подмодулями

Всегда публикуйте изменения подмодуля перед тем как публиковать изменения в superproject который ссылается на него. Если вы забыли опубликовать изменения подмодуля, другие не смогут склонировать репозиторий:

```
$ 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'
```

Если вы индексирете обновленный подмодель для ручного коммита, будьте осторожны не добавляйте в конце слэш когда определяете путь. Если слэш присутсвует в конце, Git будет полагать что вы удаляете подмодуль и проверяете содержимое этой дироктории в содержащий репозиторий.

```
$ 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
#
```

Чтобы исправить индекс после выполнения этой операции, откатите изменения и затем добавте подмодуль без слэша в конце пути..

```
$ 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
#
```

Вы также не должны перематывать ветки в подмодуле кроме коммитов которые были записаны в любом суперпроекте.

Это не безопасно выполнять `git submodule update` если вы сделали и закоммитили изменения в подмодуле без извлечения первоначально ветки. Они будут молча перезаписаны.:

```
$ 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
```

ЗАМЕЧАНИЕ: Изменения все еще видимы в reflog подмодуля.

Это не тот случай если вы не выполнили коммит для ваших изменений.

gitcast:c11-git-submodules

Chapter 6

Работа с Git

GIT НА WINDOWS

(mSysGit)

gitcast:c10-windows-git

РАЗВЕРТЫВАНИЕ С GIT

Capistrano и Git

GitHub Guide on Deploying with Cap

Git and Capistrano Screencast

ИНТЕГРАЦИЯ С SUBVERSION

SCM МИГРАЦИЯ

Вы приняли решение сменить вашу текущей систему контроля версий и сконвертировать весь проект в Git. Как проделать это быстро и безболезненно?

Импорт Subversion

Git поставляется со скриптом git-svn который имеет команду клонирования которая импортирует репозиторий subversion в новый git репозиторий. Также существует бесплатная утилита на GitHub которая может это сделать.

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

Это даст вам новый Git репозиторий со всей историей оригинального репозитория Subversion. Это занимает большое количество времени, обычно она начинается с версии 1 и извлекает и выполняет коммиты локально на каждый один снапшот.

Импорт Perforce

В contrib/fast-import вы найдете скрипт git-p4, это Python скрипт который может импортировать для вас репозиторий Perforce.

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

Импорт других

Существуют другие SCM которые перечислены выше в Git Survey, найдите документацию по импорту для них.

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

ГРАФИЧЕСКИЙ GIT

Для Git существует пара популярных GUI которые могут читать и/или манипулировать репозиториями Git .

GUI в комплекте

Git поставляется с двумя основными GUI программами написанными на Tcl/Tk. Gitk это обозреватель репозитория и инструмент визуализации истории коммитов.

gitk

git gui это инструмент который поможет вам наглядно изобразить процессы индексирования, такие как добавление, удаление и выполнение коммита. Он не позволит вам выполнить все то что доступно командной строке, но в большинстве базовых операций будет вполне достаточно.

git gui

Сторонние аналогичные проекты

For Mac users, there are GitX and GitNub

For Linux or other Qt users, there is QGit

HOSTED GIT

github

repoorcz

Примечание: переводить нечего

ALTERNATIVE USES

ContentDistribution

TicGit

Примечание: переводить нечего

SCRIPTING AND GIT

Ruby and Git

grit

jgit + jruby

PHP and Git

Python and Git

pygit

Perl and Git

perlgit

Примечание: нужно обратиться к документации соотвествующей библиотеки

GIT AND EDITORS

textmate

eclipse

netbeans

Примечание: подержка git в эклипс (www.eclipse.org/egit/) и нетбинс (www.nbgit.org) уже есть

Chapter 7

Internals and Plumbing

КАК GIT ХРАНИТ ОБЪЕКТЫ

Эта глава более подробно описывает как Git физически хранит объекты.

Все объекты хранятся в сжатом виде по имени их sha значения. Они содержат тип объекта, размер и содержимое в формате gzipped.

Существуют два формата в которых Git хранит объекты - свободные и сжатые.

Свободные Objects

Свободные объекты это простейший формат. Это просто сжатые данные сохраненные в файл на диске. Каждый объект записывается в отдельный файл.

Если sha значение вашего объекта `ab04d884140f7b0cf8bbf86d6883869f16a46f65`, то файл будет храниться по след. пути:

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

Git отсекает два первых символа и использует их как поддиректорию, таким образом никогда не бывает очень много объектов в одной директории. Имя файла в действительности состоит из оставшихся 38 символов.

Простейший способ описать в точности как хранятся данные, это показать реализацию на Ruby хранилища объекта:

```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
```

Сжатые объекты

Другой формат хранения объектов это пакфайлы, Так как Git хранит каждую версию файла как отдельный объект, то это способ будет не очень эффективным. Представьте что у вас есть файл в несколько тысяч строк и изменяется всего лишь одна строка. Git сохранит второй файл не полностью, что было бы расточительством дискового пространства.

Чтобы сохранить это пространство, Git использует пакфайлы. Это формат, в котором Git сохранит только измененную часть второго файла, и указатель на первый оригинальный файл.

Когда объекты записываются на диск, чаще это свободный формат, так как этот формат менее затратный. Тем не менее, со временем вам потребуется сохранить дисковое простанство упаковывая объекты - это выполняется командой git gc. Эта команда использует специальный алгоритм чтобы определить какие файлы похожи в наибольшей степени. Могут существовать и составные пакфайлы, они могут быть перепакованы если это необходимо (git repack) или легко распакованы обратно в свободный формат (git unpack-objects).

Git также запишет индекс файл для каждого пакфайла который значительно меньше и содержит смещения в пакфайле, чтобы можно было еще быстрее найти определенные объекты по их sha значению.

Подробности реализации пакофайлов могут быть найдены в главе Пакфайлы которая будет несколько позже.

ПРОСМОТР ОБЪЕКТОВ GIT

Мы можем запросить git о определенном объекте с помощью команды cat-file. Заметьте что вы можете сократить sha до нескольких символов чтобы не печатать все 40 шестнадцатиричных цифр:

```
$ 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
```

Дерево может ссылаться на один или более объектов, каждому из которых соотвествует файл. Вдобавак, дерево может также ссылаться на другие объекты дерево, таким образом и образуется иерархия директорий. Вы можете исследовать содержимое любого дерева используя команду ls-tree (помните что вполне достаточно набрать начальное значение SHA1):

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

Таким образом мы увидим что это дерево имеет один файл в нем. SHA1 значение это ссылка на файл с данными:

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

Блоб это просто файл с данными, который мы можем также просмотреть с помощью:

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

Заметьте что это старый файл с данными; таким образом объект который git наименовал в ответ на начальное дерево было дерево со снапшотом состояния директории которое было записано первым коммитом.

Все эти объекты хранятся под их SHA1 именами в директории 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
```

и содержимое этих файлов это просто сжатые данные плюс заголовок идентифицирующий их длину и их тип. Тип это блоб, дерево или коммит или таг.

Простейший коммит для поиска это коммит HEAD, который можно найти из .git/HEAD:

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

Вы можете видеть, что это говорит нам в какой ветке мы находимся в данный момент, и это нам известно по имени файла в директории .git, который сам содержит SHA1 имя ссылающееся на объект коммит, который мы можем просмотреть с помощью 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
```

Объект дерево здесь ссылается на новое состояние дерева:

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

и объект родитель ссылается на предыдущий коммит:

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

GIT СПРАВОЧНИК

Ветви, удаленно-отслеживаемые ветки, и таги все ссылаются на коммиты. Все ссылки именуются со слэш разделением имени пути начинающегося с "refs"; имена которыми мы пользуемся до сих пор фактически сокращения:

```
- The branch "test" is short for "refs/heads/test".
- The tag "v2.6.18" is short for "refs/tags/v2.6.18".
- "origin/master" is short for "refs/remotes/origin/master".
```

Полное имя иногда полезно, если например, существует ветка и таг с одинаковым именем.

(Только что созданные ссылки в действительности хранятся в директории .git/refs, по пути заданному по их имени. Как бы там ни было, для большей эффективности они могут быть также упакованы вместе в единый файл; просмотрите git pack-refs).

Другое полезное сокращение, ссылку на "HEAD" репозитория можно получить просто используя имя этого репозитория. Например имя "origin" обычно сокращение для HEAD ветки в репозитории "origin".

Для полного списка путей которые git проверяет на ссылки, и порядок который он использует чтобы решать что выбрать когда есть несколько ссылок с одним сокращенным именем, просмотрите секцию "SPECIFYING REVISIONS" git rev-parse.

Отобразить коммиты уникальные для данной ветки

Предположим вам хочется увидеть все коммиты достижимые из головы ветки с именем "master", но не из любой другой головы в вашем репозитории.

Мы можем получить список HEAD(ветвей) в нашем репозитории с помощью 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
```

Мы можем получить просто имена HEAD ветвей, и удалить "master", с помощью стандартных утилит cut и 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
```

И затем мы можем запросить просмотреть все коммиты достижимые из ветки master но не из других HEAD ветвей:

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

Очевидно, что тут возможны бесконечные вариации; например чтобы увидеть все коммиты достижимые из некоторой HEAD ветки но не из какого либо тага в репозитории:

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

(Просмотрите git rev-parse чтобы получить разяснение синтаксиса commit-selecting такого как `--not`.)

(!!update-ref!!)

ИНДЕКС GIT

Индекс это бинарный файл (обычно находится в .git/index) содержащий отсортированный список имен путей, каждый с разрешениями и SHA1 значениями объекта блоба; git ls-files покажет вам содержимое индекса:

```
$ 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
```

Замечание, в старой документации вы возможно встретите определение индекса "current directory cache(кэш текущей директории)" или просто "cache(кэш)". Он имеет три важдых свойства:

1. Индекс содержит всю информацию необходимую чтобы сгенерировать один (уникальный) объект дерево.

   Например, выполнение git commit сгенерирует этот объект дерево из индекса, сохранит его в базе данных объектов, и использует его как объект дерево связанный с новым коммитом.

2. Индекс разрешает быстрое сравнение между объектом дерево которое он определяет и рабочим деревом.

   Он делает это сохраняя некоторую дополнительную информацию каждой записи (такой как время последней модификации). Эти данные не отображаются выше, и не сохраняются в созданном объекте дерево, но они могут быть использованы чтобы быстро определить какие файлы в рабочей директории отличаются от тех что в индексе, и поэтому предохраняет git от постоянного чтения всех данных из таких файлов в поисках изменений.

3. Он может эффективно предствить информацию о конфликтах при слиянии между различными объектами дерево, позволяя каждому пути быть связанным с достаточной информацией о вовлеченных деревьях, и вы можете создавать слияние между ними в три способа.

   В процессе слияния, индекс может хранить множество версий одного файла (называемых "stages(заморозка)"). Третья колонка в выводе git ls-files выше это номер заморозки, и примет значения отличные от 0 для файлов с конфликтами при слиянии.

Индекс это своего рода временная область заморозки, которая заполняется деревом над котором вы в процессе работы.

ПАКФАЙЛЫ

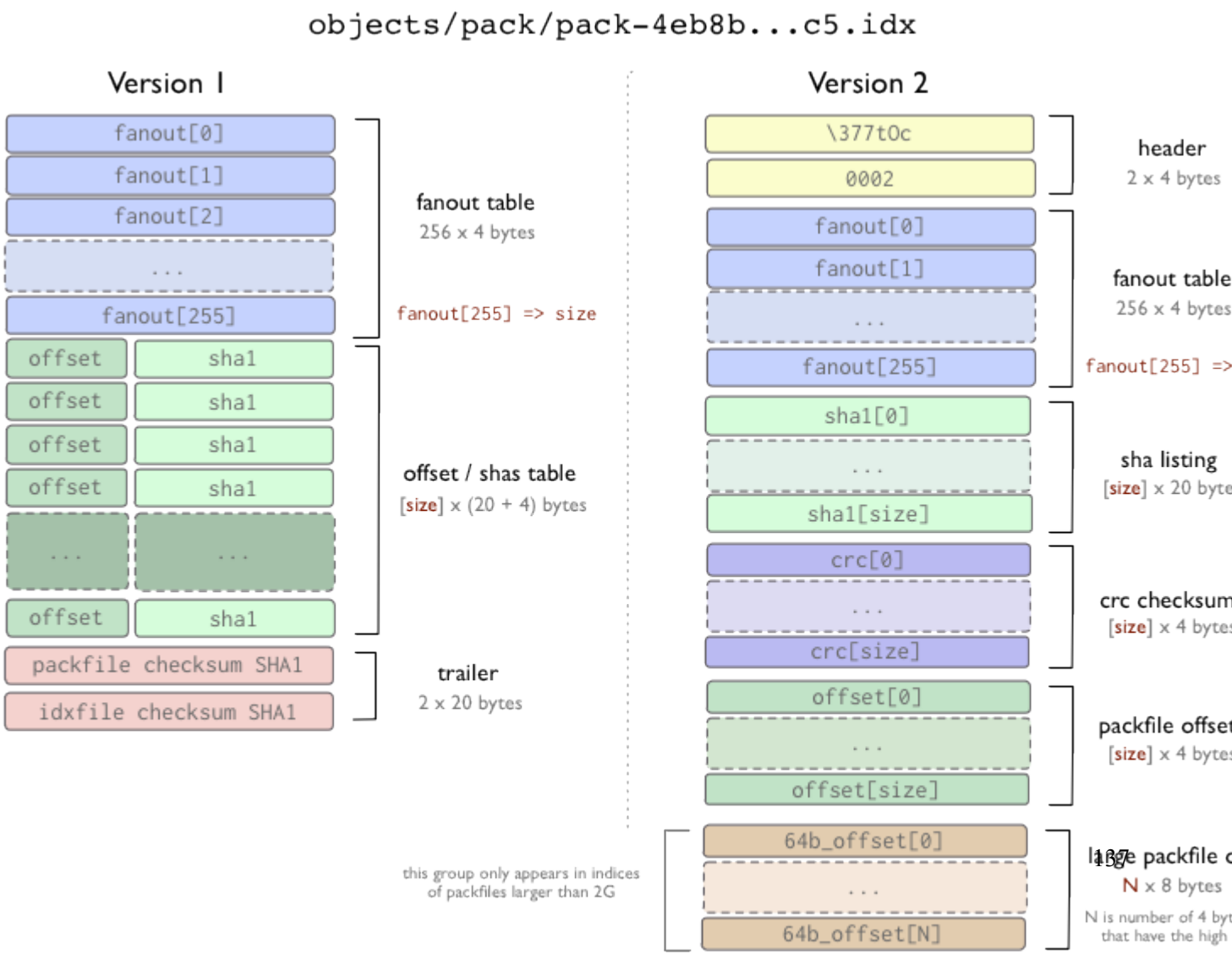
Это глава объясняет подробнее, вниз до самых битов, какой формат записи у пакфайлов и у индекса пакфайла.

Файл с индексом пакфайла

Первое, у нас есть пакфайл индекса, который в основе просто группа закладок в пакфайле.

Существуют две версии индекс пакфайла - версия один, которая по умолчанию для Git старше версии 1.6 и версия два которая по умолчанию начиная для Git с версии 1.6, но может быть прочитана Git вплоть до версии 1.5.2, и даже была портирована глубже до версии 1.4.4.5 если вы до сих пор работаете с версией из серии 1.4..

Версия два также включает CRC контрольную сумму каждого объекта и упакованные данные могут быть скопированы прямо из пакета в пакет во время операции переупакования без того чтобы поврежденные данные оказались незамеченными. Индексы в версии 2 могут также обрабатывать пакфайлы размером более 4Gb.


objects/pack/pack-4eb8b...c5.idx
Version 1
fanout[0]
fanout[1]
fanout[2]
...
fanout[255]
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 table
256 x 4 bytes
sha1[0]
sha1[size]
sha listing
[size] x 20 byte
crc[0]
crc[size]
crc checksum
[size] x 4 bytes
offset[0]
offset[size]
packfile offset
[size] x 4 bytes
64b_offset[0]
64b_offset[N]
N x 8 bytes
N is number of 4 byt
that have the high

В обоих форматах, таблица ветвления это просто способ быстро найти смещение определенного sha внутри индекс файла. Таблицы offset/sha1[] сортируются по sha[] значениям (это для того чтобы позволить бинарный поиск по этой таблице), и таблица fanout[] указывает на таблицу offset/sha1[] определенном способом (таким образом что часть последней таблицы которая охватывает все хэши которые начинаются с данного байта могут быть найдены, чтобы избежать 8 итераций бинарного поиска).

В версии 1, смещения и sha в одном пространстве, тогда как в версии 2, существуют раздельные таблицы для sha, crc и смещений. В конце обоих файлов проверочная сумма sha значений для обоих индекс и пак файлов на который он ссылается.

Обратите внимание, индексы пакфайлов *не* обязательно выделяют объекты из пакфайла, они просто использованы чтобы *быстро* получить отдельные объекты из пакета. Формат пакфайла используется в программах upload-pack и receieve-pack (push и fetch протокол), чтобы передавать объекты и индекс в этих случаях не используется, так как он может быть собран после, фактически при сканировании пакфайла.

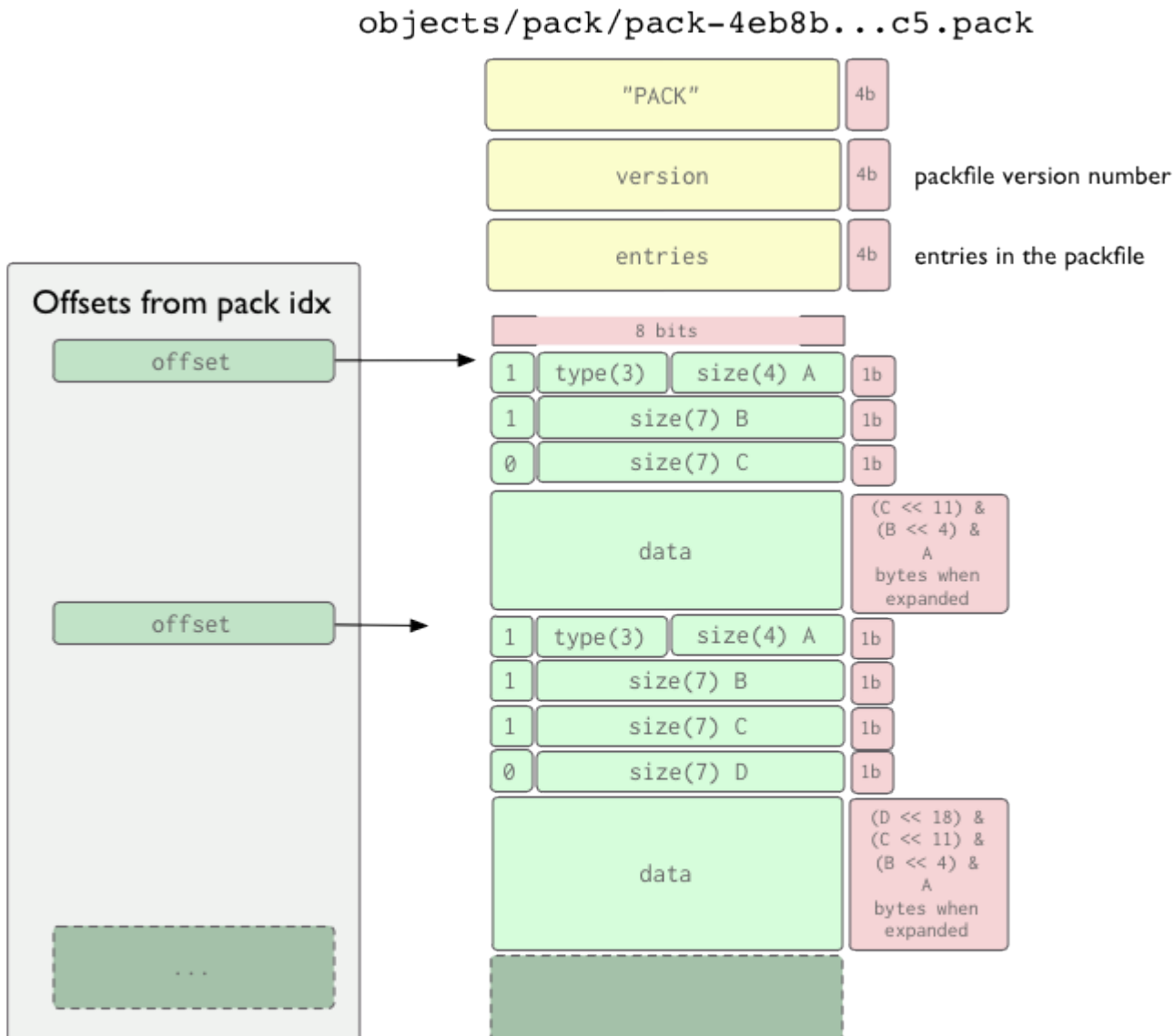
Формат пакфайла

Пакфайл сам по себе очень простой формат. Существует заголовок, группа упакованных объектов (каждый со своим заголовоком и телом) и затем в конце контрольная сумма. Первые четыре байта это строка 'PACK', которая гарантирует что вы правильно определили начало пакфайла. Затем идет еще 4 байта версии пакфайла и затем 4 байта количество записей в этом файле. В Ruby, вы могли бы прочитать данные заголовока след.образом:

```
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
```

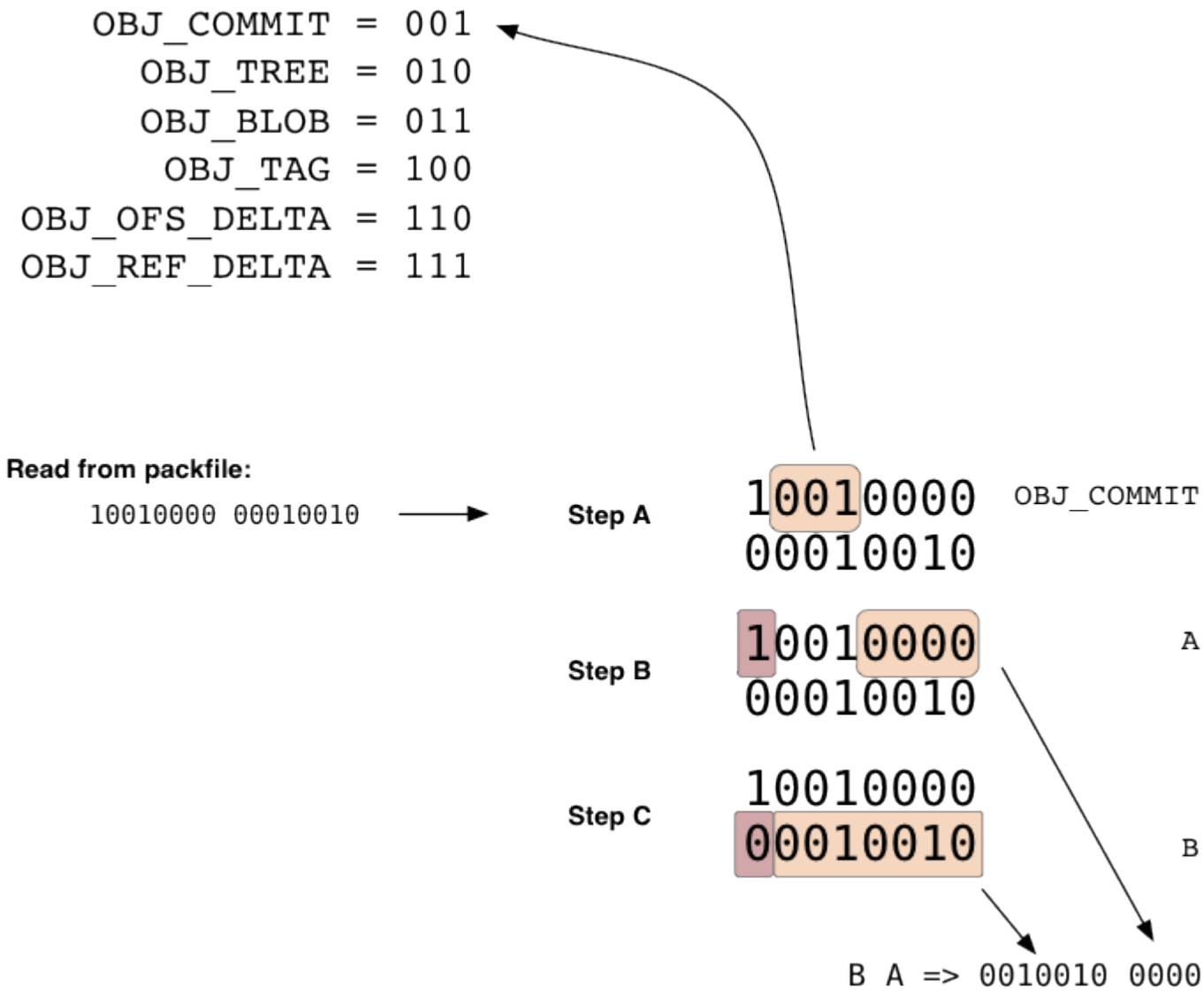
После этого, вы получаете группу упакованных объектов, в порядке их SHA значения, каждый из которых состоит из заголовка объекта и содержимого объекта. В конце пакфайла 20 байтовая сумма SHA1 всех sha значений (в отсортированном порядке) в этом пакфайле.


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

Заголовок объекта это группа из одного или более 1 байта (8 битов) кусков которые определяют тип объекта и последующие данные, и размер данных после распаковки. Каждый байт в действительности 7 битов данных, первый бит используется чтобы объявить что кусок с первым битом использованным чтобы определить если это последний кусок или что данные не начинаются до него. Если первый бит 1, вы прочитаете другой байт, иначе далее идут данные. Первые 3 бита в первом байте определяют тип данных, в соотвествии с таблицей ниже.

(В данный момент, 8 значений которые могут выразить 3 бита (0-7), 0 (000)неопределен а 5 (101) неиспользован.

Здесь мы можем увидеть пример заголовка из двух байтов, где первый определяет что следом идущие данные это коммит, и остаток первого и последние 7 битов второго определяют то что данные займут 144 байта после распаковки.


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

Это важно понять что размер определенных в заголовке данных это не размер который в действительно последует после, а размер данных *после того как они будут извлечены (распакованы)*. Поэтому смещение в индексе пакфайла так полезно, иначе вы должны распаковать каждый объект чтобы определить когда начнется следующий заголовок.

Часто то что является данными это просто zlib поток для объектов типа non-delta; для представления двух delta объектов, порция данных содержит нечто что идентифицирует от какого базового объекта это дельта представление зависит, и дельта накладывается на базовый объект чтобы восстановить объект. `ref-delta` использует 20-байтный хэш базового объекта в начале данных, в то время как `ofs-delta` хранит смещение в пределах того же пакфайла чтобы идентифицировать базовый объект. В других случаях, реализатор должен следовать двум важным ограничениям:

- дельта представление должно быть основано на некотором другом объекте того же пакфайла;

- базовый объект должен быть того же нижележащего типа (блоб, дерево, коммит или таг);

Примечания: Дельта кодирование алгоритм сжатия основанный на разнице предыдущего и последующих значений.

RAW GIT

Здесь мы взглянем на то как манипулировать git на более низком уровне, в случае если вы пожелаете написать инструмент генерирующий новые блобы, деревья, или коммиты искуственным способом. Если вы захотите написать скрипт который использует более низкоуровневые функции git, чтобы проделать что то новое, здесь некоторые инструменты нужные вам.

Создание блобов

Создание блобов в вашем Git репозитории и получение SHA значения несложно. Команда git hash-object это все что вам нужно. Чтобы создать объект блоб из существующего файла, просто выполните ее с параметром '-w' (который говорит ей записать блоб, а не просто вычислить SHA).

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

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

В выводе команды вы увидите SHA значение блоба который был создан..

Создание деревьев

Теперь давайте положим вы хотите создать дерево из ваших новых объектов. Команда git mktree делает это просто генерируя новый объект дерево из git ls-tree форматированного вывода. Например, если вы записали следующее в файл под именем '/tmp/tree.txt' :

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

и затем пропустили это через команду git mktree, Git запишет новое дерево в базу данных объектов и выдаст вам новое sha значение этого дерева.

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

Затем, мы можем взять это и сделать это поддиректорией еще одного другого дерева, и так далее. Если мы хотим создать новое дерево с этим как поддеревом, мы просто созданим новый файл (/tmp/newtree.txt) с нашим новым SHA как дерево в нем:

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

и затем выполним git mk-tree снова:

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

И мы теперь имеем искуственную структуру директорий в Git, которая выглядит следующим образом:

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

1 directory, 3 files
```

причем это структура никогда в действительности не существовала на диске. Плюс, у нас есть SHA (`5bac6559`) которое указывает на нее.

Перестроить деревья

Мы также можем проделать манипуляции с объединением деревьев в новую структуру используя файл индекс. Как простейший пример, давайте возьмем дерево которое мы только что создали и сделаем

новое дерево которое имеет две копии нашего `5bac6559` дерева в нем используя временный файл индекс. (Вы можете проделать это сбросив переменную окружения GIT_INDEX_FILE или в командной строке)

Первое, мы читаем дерево в наш индекс файл под новым префиксом используя команду git read-tree, и затем запишем содержимое индекса как дерево используя команду 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
```

Теперь мы видим что мы создали новое дерево просто манипулируя индексом. Вы также можете делать интересные операции слияния и тому подобное во временном индексе. Просмотрите документацию git read-tree чтобы получить больше подробностей.

Создание коммитов

Теперь когда у нас есть SHA дерева, мы можем создать объект коммит которое на него указывает. Мы можем проделать это используя команду git commit-tree. Большинство данных которые потом войдут в коммит должны быть установленны как переменные окружения, так что вам потребуется установить их:

```
GIT_AUTHOR_NAME
GIT_AUTHOR_EMAIL
GIT_AUTHOR_DATE
GIT_COMMITTER_NAME
```

```
GIT_COMMITTER_EMAIL
GIT_COMMITTER_DATE
```

Затем вам потребуется записать ваше сообщение-описание коммита в файл или каким либо образом передать эту информацию в команду через STDIN. Затем, вы можете создать ваш коммит основываясь на sha дерева которое есть нас.

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

Если вы хотите определить один или более родителей коммита, просто добавьте sha значения в командную строку вместе с параметром '-p' перед каждым из них. SHA нового объекта коммита будет выведено как результат работы команды..

Обновление ссылки ветки

Теперь у нас есть SHA значение нового объекта коммит, и мы можем обновить ветку чтобы она указывала него если мы так хотим. Давайте скажем что мы хотим обновить нашу ветку 'master' так чтобы она указывала на новый коммит который мы только что создали - мы используем команду git update-ref для этого:

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

ПРОТОКОЛЫ ПЕРЕДАЧИ

Здесь мы узнаем как клиенты и серверы общаются друг с другом и обмениваются данными Git между собой.

Извлечение данных через HTTP

Извлечение через http/s URL заставит Git использовать немного глупый протокол. В этом случае, вся логика ложится на сторону клиента. Сервер не требует специальной настройки - любой статический вебсервер подойдет для этой работы если директори git которую вы извлекает доступна вебсерверу.

Чтобы это работало, вам нужно выполнять единственную команду на репозитории сервера каждый раз если что то обновилось, команда выглядит след.образом git update-server-info, которая обновляет файлы objects/info/packs и info/refs чтобы составить список какие ссылки и пакфайлы досупны, так как вы не можете изменить этот список через http. Когда эта команда выполняется, файл objects/info/packs выглядит приблизительно след.образом:

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

Так что если извлечение не может найти свободный(loosy) файл, то оно может попробовать эти пакфайлы. Файл info/refs приблизительно будет выглядеть след.образом:

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

Потом когда вы извлекаете из этого репозитория, то процесс начнет с этих refs и пройдет объекты коммит до тех пор пока клиет не получит все объекты которые ему нужны.

Например, если вы попросите извлечь ветку master, то видно что ветка master указывает на `32aae7ae` а ваша ветка master указывает на `ab04d88`, так что вам нужно `32aae7ae`. Вы извлекаете этот объект

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

И это выглядит след.образом:

```
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
```

Теперь это извлекает дерево `aa176fb8`:

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

которе выглядет след.образом:

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

И теперь это извлекает эти объекты:

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

В действительности это происходит с помощью Curl, и может открываться множество паралельных потоков чтобы ускорить процесс. Когда процесс заканчивает рекурсию дерева указанного коммитом, оно извлекает следующего родителя.

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

Теперь в этом случае, коммит который вернулся выглядет след.образом:

```
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
```

и можем видеть что родитель, `ab04d88` это куда наша ветка master в данный момент указывает. Теперь, мы рекурсивно извлекаем это дерево и затем останавливаемся, так как мы знаем что у нас есть все что идет до этой точки. Вы можете заставить Git дважды проверить, то что у нас уже есть все, с помощью параметра '--recover'. Просмотрите документацию git http-fetch чтобы получить больше подробностей.

Если извлечение одного из свободных объектов потерпит неудачу, Git будет скачивать индексы пакфайлов пытаясь найти sha значение которое ему нужно, и затем скачает этот пакфайл.

Это важно если вы используете git сервер который обслуживает репозитории этим способом чтобы реализовать хуки post-recieve которые запускаются командой 'git update-server-info' каждый раз или произойдет путаница.

Извлечение данных с помощью выгрузки пакетов

В более умных протоколах, извлечение объектов более эффективно. Сокет открыт, или через ssh или другой порт 9418 (в случае протокола git://), и команда git fetch-pack на клиенте начнет сообщаться с форком процесса git upload-pack на сервере..

Затем сервер сообщит клиенту которое SHA значение он имеет для каждой ref, и клиент определит что ему нужно и ответит списком SHA значений которые ему нужныи которые у него уже есть.

В этот момент, сервер сгенерирует пакфайл со всеми объектами которые нужны клиенту и передаст их клиенту.

Давайте взглянем на пример.

Клиент соединяется и посылает заголовок запроса. Команда клон

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

производит следующий запрос:

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

Первые 4 байта содержат 16ое длину строки (включая 4 байта длины строки и символ окончания строки если таковой имеется). Следующее это команды и их аргументы. Затем идет нулевой байт и затем данные о хосте. Запрос заканчивается нулевым байтом.

Запрос обрабатывается и конвертируется в вызов git-upload-pack:

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

Это немедленно возвращает информацию репозитория:

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

Каждая строка начинается с 4 байт строки длины объявления в hex. Эта часть завершается строкой с длиной объявления 0000.

Это отсылается назад клиенту. Клиент отвечает другим запросом:

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

Это оправлено чтобы открыть процесс git-upload-pack который затем создат поток и вернет его назад как заключительный ответ:

```
"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"
```

Просмотрите предыдущую главу Пакфайл чтобы получить описание формата данных пакфайла в ответе.

Выполнение push

Выполнение push через git и ssh протоколы похоже, но проще. По существу происходит то что клиент запрашивает экземпляр receive-pack, который запускается если клиент имеет доступ, затем сервер возвращает все sha значения заголовков ссылок, которые у него есть опять и клиент генерирует пакфайл всего что требуется серверу (обычно только если то что на сервере это прямой предок того для чего и выполняется push) и посылает этот пакфайл в исходящий поток, где сервер или сохраняет его на диске и строит индекс для него, или распаковывает его (если там не много объектов)

Это весь процесс выполняется с помощью команды git send-pack на клиенте, которая вызывается git push и git receive-packкомандой на стороне сервера, которая вызывается процессом соединения ssh или демоном git (если это открытый push сервер).

ГЛОССАРИЙ

Здесь даны определения некоторых терминов использованых в среде Git.
Все эти термины полностью могут быть скопированы из Git Glossary.

alternate object database

> Via the alternates mechanism, a repository
>
> ```
> can inherit part of its object database
> from another object database, which is called "alternate".
> ```

bare repository

A bare repository is normally an appropriately

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

объект типа блоб

Untyped object, e.g. the contents of a file.

ветка

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

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

кэш

устаревшее для: индекс

chain

A list of objects, where each object in the list contains

```
a reference to its successor (for example, the successor of a
commit could be one of its parents).
```

changeset

BitKeeper/cvsps speak for "commit". Since git does not

```
store changes, but states, it really does not make sense to use the term
"changesets" with git.
```

checkout

The action of updating all or part of the

```
working tree with a tree object
or blob from the
object database, and updating the
index and HEAD if the whole working tree has
been pointed at a new branch.
```

cherry-picking

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

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

clean

A working tree is clean, if it

```
corresponds to the revision referenced by the current
head. Also see "dirty".
```

commit

As a noun: A single point in the

```
git history; the entire history of a project is represented as a
set of interrelated commits.  The word "commit" is often
used by git in the same places other revision control systems
use the words "revision" or "version".  Also used as a short
hand for commit object.
```

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

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

commit object

An object which contains the information about a

```
particular revision, such as parents, committer,
author, date and the tree object which corresponds
to the top directory of the stored
revision.
```

core git

Fundamental data structures and utilities of git. Exposes only limited

```
source code management tools.
```

DAG

Directed acyclic graph. The commit objects form a

```
directed acyclic graph, because they have parents (directed), and the
graph of commit objects is acyclic (there is no chain
which begins and ends with the same object).
```

dangling object

An unreachable object which is not

```
reachable even from other unreachable objects; a
dangling object has no references to it from any
reference or object in the repository.
```

detached HEAD

Normally the HEAD stores the name of a

```
branch.  However, git also allows you to check out
an arbitrary commit that isn't necessarily the tip of any
particular branch.  In this case HEAD is said to be "detached".
```

dircache

You are *waaaaay* behind. See index.

directory

The list you get with "ls" :-)

dirty

A working tree is said to be "dirty" if

```
it contains modifications which have not been committed to the current
branch.
```

ent

Favorite synonym to "tree-ish" by some total geeks. See

```
`http://en.wikipedia.org/wiki/Ent_(Middle-earth)` for an in-depth
explanation. Avoid this term, not to confuse people.
```

evil merge

An evil merge is a merge that introduces changes that

```
do not appear in any parent.
```

fast forward

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

```
revision and you are "merging" another
branch's changes that happen to be a descendant of what
you have. In such these cases, you do not make a new merge
commit but instead just update to his
revision. This will happen frequently on a
tracking branch of a remote
repository.
```

fetch

Fetching a branch means to get the

```
branch's head ref from a remote
repository, to find out which objects are
missing from the local object database,
and to get them, too.  See also git fetch.
```

file system

Linus Torvalds originally designed git to be a user space file system,

```
i.e. the infrastructure to hold files and directories. That ensured the
efficiency and speed of git.
```

git archive

Synonym for repository (for arch people).

grafts

Grafts enables two otherwise different lines of development to be joined

```
together by recording fake ancestry information for commits. This way
you can make git pretend the set of parents a commit has
is different from what was recorded when the commit was
created. Configured via the `.git/info/grafts` file.
```

hash

In git's context, synonym to object name.

head

A named reference to the commit at the tip of a

```
branch.  Heads are stored in
`$GIT_DIR/refs/heads/`, except when using packed refs. (See
git pack-refs.)
```

HEAD

The current branch. In more detail: Your working tree is normally derived

```
from the state of the tree referred to by HEAD.  HEAD is a reference to one
of the heads in your repository, except when using a detached HEAD, in which
case it may reference an arbitrary commit.
```

head ref

A synonym for head.

hook

During the normal execution of several git commands, call-outs are made

```
to optional scripts that allow a developer to add functionality or
checking. Typically, the hooks allow for a command to be pre-verified
and potentially aborted, and allow for a post-notification after the
operation is done. The hook scripts are found in the
`$GIT_DIR/hooks/` directory, and are enabled by simply
removing the `.sample` suffix from the filename. In earlier versions
of git you had to make them executable.
```

index

A collection of files with stat information, whose contents are stored

```
as objects. The index is a stored version of your
working tree. Truth be told, it can also contain a second, and even
a third version of a working tree, which are used
when merging.
```

index entry

The information regarding a particular file, stored in the

```
index. An index entry can be unmerged, if a
merge was started, but not yet finished (i.e. if
the index contains multiple versions of that file).
```

master

The default development branch. Whenever you

```
create a git repository, a branch named
"master" is created, and becomes the active branch. In most
cases, this contains the local development, though that is
purely by convention and is not required.
```

merge

As a verb: To bring the contents of another

```
branch (possibly from an external
repository) into the current branch.  In the
case where the merged-in branch is from a different repository,
this is done by first fetching the remote branch
and then merging the result into the current branch.  This
combination of fetch and merge operations is called a
pull.  Merging is performed by an automatic process
that identifies changes made since the branches diverged, and
then applies all those changes together.  In cases where changes
conflict, manual intervention may be required to complete the
merge.
```

As a noun: unless it is a fast forward, a

```
successful merge results in the creation of a new commit
representing the result of the merge, and having as
parents the tips of the merged branches.
This commit is referred to as a "merge commit", or sometimes just a
"merge".
```

object

The unit of storage in git. It is uniquely identified by the

```
SHA1> of its contents. Consequently, an
object can not be changed.
```

object database

Stores a set of "objects", and an individual object is

```
identified by its object name. The objects usually
live in `$GIT_DIR/objects/`.
```

object identifier

Synonym for object name.

object name

The unique identifier of an object. The hash

```
of the object's contents using the Secure Hash Algorithm
1 and usually represented by the 40 character hexadecimal encoding of
the hash of the object.
```

object type

One of the identifiers "commit", "tree", "tag" or "blob" describing the

```
type of an object.
```

octopus

To merge more than two branches. Also denotes an

```
intelligent predator.
```

origin

The default upstream repository. Most projects have

```
at least one upstream project which they track. By default
'origin' is used for that purpose. New upstream updates
will be fetched into remote tracking branches named
origin/name-of-upstream-branch, which you can see using
"`git branch -r`".
```

pack

A set of objects which have been compressed into one file (to save space

```
or to transmit them efficiently).
```

pack index

The list of identifiers, and other information, of the objects in a

```
pack, to assist in efficiently accessing the contents of a
pack.
```

parent

A commit object contains a (possibly empty) list

```
of the logical predecessor(s) in the line of development, i.e. its
parents.
```

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

Cute name for 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 a branch means to fetch it and

```
merge it.  See also 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

To reapply a series of changes from a branch to a

```
different base, and reset the head of that branch
to the result.
```

ref

A 40-byte hex representation of a SHA1 or a name that

```
denotes a particular object. These may be stored in
`$GIT_DIR/refs/`.
```

reflog

A reflog shows the local "history" of a ref. In other words,

```
it can tell you what the 3rd last revision in _this_ repository
was, and what was the current state in _this_ repository,
yesterday 9:14pm.  See git reflog for details.
```

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 <src>:<dst>, 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

A collection of refs together with an

```
object database containing all objects
which are reachable from the refs, possibly
accompanied by meta data from one or more porcelains. A
repository can share an object database with other repositories
via alternates mechanism.
```

resolve

The action of fixing up manually what a failed automatic

```
merge left behind.
```

revision

A particular state of files and directories which was stored in the

```
object database. It is referenced by a
commit object.
```

rewind

To throw away part of the development, i.e. to assign the

```
head to an earlier revision.
```

SCM

Source code management (tool).

SHA1

Synonym for object name.

shallow repository

A shallow repository has an incomplete

```
history some of whose commits have parents cauterized away (in other
words, git is told to pretend that these commits do not have the
parents, even though they are recorded in the commit
object). This is sometimes useful when you are interested only in the
recent history of a project even though the real history recorded in the
upstream is much larger. A shallow repository
is created by giving the `--depth` option to git clone, and
its history can be later deepened with git fetch.
```

symref

Symbolic reference: instead of containing the SHA1

```
id itself, it is of the format 'ref: refs/some/thing' and when
referenced, it recursively dereferences to this reference.
'HEAD' is a prime example of a symref. Symbolic
references are manipulated with the git symbolic-ref
command.
```

tag

A ref pointing to a tag or

```
commit object. In contrast to a head,
a tag is not changed by a commit. Tags (not
tag objects) are stored in `$GIT_DIR/refs/tags/`. A
git tag has nothing to do with a Lisp tag (which would be
called an object type in git's context). A
tag is most typically used to mark a particular point in the
commit ancestry chain.
```

tag object

An object containing a ref pointing to

```
another object, which can contain a message just like a
commit object. It can also contain a (PGP)
signature, in which case it is called a "signed tag object".
```

topic branch

A regular git branch that is used by a developer to

```
identify a conceptual line of development. Since branches are very easy
and inexpensive, it is often desirable to have several small branches
that each contain very well defined concepts or small incremental yet
related changes.
```

tracking branch

A regular git branch that is used to follow changes from

```
another repository. A tracking
branch should not contain direct modifications or have local commits
made to it. A tracking branch can usually be
identified as the right-hand-side ref in a Pull:
refspec.
```

tree

Either a working tree, or a tree object together with the dependent

```
blob and tree objects (i.e. a stored representation of a working tree).
```

tree object

An object containing a list of file names and modes along

```
with refs to the associated blob and/or tree objects. A
tree is equivalent to a directory.
```

tree-ish

A ref pointing to either a commit object, a tree object, or a tag

```
object pointing to a tag or commit or tree object.
```

unmerged index

An index which contains unmerged

```
index entries.
```

unreachable object

An object which is not reachable from a

```
branch, tag, or any other reference.
```

working tree

The tree of actual checked out files. The working tree is

```
normally equal to the HEAD plus any local changes
that you have made but not yet committed.
```