Внутреннее устройство Git (git)

[<< Предыдущая] [ИНДЕКС] [Исправить] [src / Печать] [Следующая >>]

Ключевые слова: git, (найти похожие документы)

From: Damir Shayhutdinov <lost404@gmail.com.>
Date: Mon, 25 Sep 2010 17:02:14 +0000 (UTC)
Subject: Внутреннее устройство Git

Оригинал: http://los-t.livejournal.com/tag/git%20guts

Где-то около месяца назад я полностью прочитал man git, и немножко поэкспериментировал с внутренностями git, чтобы понять, как же там все устроено внутри.

Оказалось, внутренне все сделано просто и элегантно. В отличие от сложно-бинарных репозиториев subversion или недорепозиториев CVS, в git все кристалльно просто, и доступно даже на самом низком уровне.

Репозиторий git - это просто коллекция т.н. объектов, объединенных ссылками друг на друга. Каждый объект - это некий файл специального формата. У каждого объекта есть "имя", которое вычисляется как SHA1 хеш содержимого объекта и записываемся как шестнадцатеричное представление этого хеша. Длина хеша равна 20 байтам, так что шестнадцатеричное представление содержит 40 букв и цифр.

Для тех, кто не знает что такое хеш и зачем он нужен - поясняю буквально на пальцах.

Давайте отвлечемся от компьютеров и посмотрим на современную криминалистику. Как известно, преступники часто оставляют на месте преступления свои отпечатки пальцев. Эти отпечатки пальцев представляют собой комбинацию углов, завитков, спиралей и т.д. Эксперт, имея схему отпечатков, может просмотреть картотеку накопленных полицией/милицией отпечатков пальцев преступников, и найти совпадение. Существенное в этом методе то, что по минимальной информации - отпечаткам пальцев, удается (или не удается) найти преступника среди миллионов остальных людей.

У отпечатков пальцев есть три интересных свойства, которые и помогают

провести опознание. Первое - что у одного и того же человека отпечатки пальцев в течение жизни практически не меняются. Второе - у разных людей

отпечатки пальцев разные (даже у неразличимых близнецов). Третье - их очень легко получить и найти - нужно всего лишь чернила и бумагу. Вернемся к нашим баранам. Хеш-функция - это своеобразный "отпечаток пальца" файла и обладает всеми вышеперечисленными свойствами:

1. Если содержимое двух файлов совпадает - то их хеши тоже совпадают.

2. Если содержимое двух файлов различны - то их хеши тоже различны (за исключением случаев коллизий, о которых я расскажу ниже).

3. Вычислить хеш-функцию SHA1 можно очень быстро (фактически, современный процессор при этом не нагружается, узкое место тут - чтение файла с диска).

Возможны случаи, когда у двух разных файлов хеш-функция одинаковая. Такие случаи называются коллизиями. Количество коллизий у любой хеш-функции бесконечно - ведь она позволяет любое количество информации преобразовать в фиксированное, конечное количество байт. Если бы людей было бы бесконечное количество - то среди них бы тоже бы наблюдались коллизии по отпечаткам пальцев или по любым другим методам опознания. Поэтому различные хеш-функции отличаются сложностью возникновения коллизий (или их целенаправленного подбора). Хеш-функция SHA1 считается достаточно сложной для подбора или случайного возникновения коллизии, так как пока неизвестно алгоритма подбора коллизии, кроме как методом перебора, а криптологические особенности алгоритма уменьшают вероятность случайного возникновения коллизий.

Кстати, вычислить эту сумму для прозвольного файла в системе может специальаная утилита sha1sum, входящая в coreutils.

Коллекция объектов git - это "картотека", куда заносятся все объекты. Они упорядочены по именам, которые являются также SHA1 отпечатками объектов. Это имя позволяет быстро и однозначно идентифицировать объект, а также проверить его целостность - если объект повредился, его хеш не совпадет с именем.

Расположены все объекты в каталоге .git/objects. Для того чтобы не сваливать все объекты в одну директорию, git отделяет первые два символа имени объекта и создает поддиректорию с таким именем в .git/objects.

Вот например, содержимое базы некоторого git-репозитория

```
.git/objects/a4/b7fce097055c3cbd6879db9625f9a3890cc409
.git/objects/8c/3c7fbcd903744b20fd7567a1fcefa99133b5bc
.git/objects/e9/65047ad7c57865823c7d992b1d046ea66edf78
```

То есть в ней хранятся три объекта:

1. a4b7fce097055c3cbd6879db9625f9a3890cc409
2. 8c3c7fbcd903744b20fd7567a1fcefa99133b5bc
3. e965047ad7c57865823c7d992b1d046ea66edf78

Сделано это для ускорения поиска объекта по его имени. Несложно догадаться, что такой нехитрый прием ускоряет поиск в 256 раз. Добавлю напоследок, что объекты в git бывают четырех типов - blob, tree, commit и tag.

Часть 2

Чтож, если ваш мозг не был зохаван Ктулху во время чтения предыдущего поста, то вы уже знаете, что репозиторий git представляет собой картотеку различных объектов, разложенную по именам-хешам. Объекты бывают четырех типов - blob, tree, commit, tag.

Объекты типа blob(Binary Large Object) - это основа репозитория. Это маленькие безымянные герои. Безымянные в прямом смысле - это просто содержимое, без имени. Если вы добавляете в git файл tutorial.txt с содержимым "Hello, world!", то это результирующий blob-объект будет содержать строчку "Hello, world!" и ни слова о tutorial.txt. Это очень похоже на иноды (inodes), используемые в файловых системах, если вы понимаете о чем я.

Новый blob-объект создается из содержимого файла с помощью команды git-hash-object.

Если вызывать ее без параметров, только с именем файла - то она выведет SHA1 хеш blob-объекта, который будет создан из этого файла. Если же вызвать ее с параметром -w, то соответствующий blob-объект будет записан в базу под именем, соответствующим этому хешу.

Если объект с таким именем уже существует в базе - то он не будет перезаписан. Вспомните, что имя является "отпечатком пальца" объекта, достаточно уникальным. Значит, если у двух объектов одинаковые имена, то у них одинаковые содержимые. Поэтому git не будет перезаписывать blob.

Например, если в гит положить десять абсолютно одинаковых файлов весом

6 мегабайт, то реально в базе будет занято только 6 мегабайт, а не 60. Это из-за того что blob-объекты не содержат никакой информации об именах файлов, из которых они сделаны, поэтому они идентичны.
Вот пример создания blob-объекта:

```
$ mkdir ~/tmp/gitguts
$ cd ~/tmp/gitguts
$ git-init
Initialized empty Git repository in .git/

$ echo "Hello, World\!" > tutorial.txt
$ git-hash-object -w tutorial.txt
8ab686eafeb1f44702738c8b0f24f2567c36da6d

$ find .git/objects -type f
.git/objects/8a/b686eafeb1f44702738c8b0f24f2567c36da6d
```

Как видно, вызов git-hash-object с параметром -w действительно создал и сохранил в базе новый объект типа blob, содержащий строчку "Hello, world!"

Посмотреть, что внутри объекта-blob можно с помощью команды git-cat-file

```
$ git-cat-file blob 8ab686eafeb1f44702738c8b0f24f2567c36da6d
Hello, World!
```

Этот новосозданный объект пролежит в базе до тех пор, пока не будет вызван "уборщик мусора" (git-prune или git-gc --prune). Этот трудяга проверяет "прописку" всех объектов, и если на объект не имеется никаких ссылок, то он удаляется из базы. На этот объект мы еще не сделали никаких ссылок, так что при сборке мусора он просто исчезает из базы.

```
$ git-prune
$ find .git/objects -type f
.git/objects/info/packs
```

Сами объекты-blob не могут иметь никаких ссылок. Вместо этого, на них ссылается другой объект - дерево(tree). О них мы поговорим в следующий раз.

Часть 3

В первой части я уже упоминал, что репозиторий git представляет собой картотеку объектов, объединенных ссылками друг на друга.
Из четырех типов объектов в git (blob, tree, commit, tag) только blob-ы не могут содержать ссылки. Все остальные объекты, по сути, являются просто ссылками либо на blob-ы, либо на другие ссылки.
Мы уже знаем, что blob-ы включают в себя только содержание файла, но не его имя, или режимы доступа. Вся информация об именах содержится в объектах-деревьях (tree). Фактически, деревья аналогичны понятию "каталог" в файловой системе, так же как blob-ы аналогичны понятию inode.

Объекты-деревья могут хранить внутри себя как ссылки на blob-ы, так и ссылки на другие объекты-деревья. В результате можно построить иерархию деревьев, аналогичную иерархии каталогов и файлов.

Объект-дерево представляет собой список элементов, состоящих из четырех полей:

1. mode (режим доступа) - представляет собой права UNIX на объект-ссылку, плюс несколько дополнительных битов, позволяющих хранить в гите символические ссылки. Записывается в виде шести цифр, из которых первые три описывают тип объекта, а оставшиеся - права UNIX. Правда мне ни разу не удалось увидеть, чтобы значение третьей цифры было отлично от нуля, так что я не знаю что она означает.
   Первая цифра - 1 для файлов и символических, 0 для директорий.
   Вторая цифра - 0 для файлов, 2 для символических ссылок, 4 - для директорий

2. Тип объекта, на который ссылается элемент списка. Может быть blob или tree.

3. SHA1 хеш объекта. Собственно, это и является ссылкой, так как однозначно определяет объект в репозитории git.

4. имя объекта. Имя файла для blob-ов, имя директории для tree.

Объект-дерево после создания получает свое имя-хеш, и может быть после этого включен в другие деревья.
Создать новый объект-дерево можно с нуля, используя команду git-mktree. Ей на вход (stdin) надо передать текстовый список, в котором каждая строчка описывает один элемент. Первые три поля должны быть разделены пробелами, а последнее - имя объекта - должно быть отделено табом.

Вот пример:

```
$ mkdir ~/tmp/gitgut3
```

```
$ cd ~/tmp/gitgut3
$ git-init
Initialized empty Git repository in .git/

$ echo "File1" > file1
$ echo "File2" > file2

$ git-hash-object -w file1
03f128cf48cb203d938805e9f3e13b808d1773e9

$ git-hash-object -w file2
b973e639605e63466ea5ba09b04a545f16946ca8

$ echo -e "100640 blob 03f128cf48cb203d938805e9f3e13b808d1773e9\tfile1
100640 blob b973e639605e63466ea5ba09b04a545f16946ca8\tfile2" | git-mktree

b2efb2a7e48025c4d185080412a6ba1121ee6c59
```

Как видно из примера, команде git-mktree нужно подать на стандартный вход содержимое создавамого объекта-дерева, что я и сделал командой echo.

Полученный объект-дерево теперь присутствует в базе:

```
$ ls .git/objects/b2/efb2a7e48025c4d185080412a6ba1121ee6c59
.git/objects/b2/efb2a7e48025c4d185080412a6ba1121ee6c59
```

Его содержимое можно посмотреть, используя команду git-ls-tree

```
$ git-ls-tree b2efb2a7e48025c4d185080412a6ba1121ee6c59
100640 blob 03f128cf48cb203d938805e9f3e13b808d1773e9    file1
100640 blob b973e639605e63466ea5ba09b04a545f16946ca8    file2
```

Чтобы далеко не уходить, покажу, чем же полезно, что деревья являются именно объектами, с именами-хешами.

Например, если у двух объектов-деревьев одинаковое имя-хеш, что это означает? Что внутренности этих деревьев совпадают! А так как внутренности деревьев - это ссылки на объекты, то это означает что два дерева ссылаются на одни и те же объекты. Которые в свою очередь тоже могут быть деревьями или блобами. Таким образом имя-хеш дерева на самом деле идентифицирует не только "файлы в директории", но и все файлы во всех поддиректориях этой директории - одно имя для всех иерархии!

Это свойство позволяет git-у очень быстро производить сравнение деревьев со сколь угодно сложной иерархией, уровнями вложенности и т.д. без чтения собственно содержимого - blob-ов или tree.
Например, я создаю новое дерево, которое отличается от старого дерева b2efb2a7e4... тем, что в содержимое file2 была добавлена дополнительная строчка, а файл file1 переименован в file3.

```
$ echo Secondline >> file2
$ git-hash-object -w file2
4dd2746869211aedfec0f07afb12a879c09569e7

$ echo -e "100640 blob 03f128cf48cb203d938805e9f3e13b808d1773e9\tfile3
100640 blob 4dd2746869211aedfec0f07afb12a879c09569e7\tfile2" | git-mktree

493a5292de0b743e77aa190921da56d33599b59e

$ git-ls-tree 493a5292de0b743e77aa190921da56d33599b59e

100640 blob 4dd2746869211aedfec0f07afb12a879c09569e7    file2
100640 blob 03f128cf48cb203d938805e9f3e13b808d1773e9    file3
```

Давайте посмотрим, как git может легко вычислить разницу между этими деревьями. используя только объекты-деревья.

Для этого сохраним выводы git-ls-tree для каждого дерева в отдельный файл и натравим на них команду diff -u.

```
$ git-ls-tree b2efb2a7e48025c4d185080412a6ba1121ee6c59 > tree1
$ git-ls-tree 493a5292de0b743e77aa190921da56d33599b59e > tree2
$ diff -u tree1 tree2
--- tree1       2007-08-14 14:55:06 +0400
+++ tree2       2007-08-14 14:55:30 +0400
@@ -1,2 +1,2 @@
-100640 blob 03f128cf48cb203d938805e9f3e13b808d1773e9   file1
-100640 blob b973e639605e63466ea5ba09b04a545f16946ca8   file2
+100640 blob 4dd2746869211aedfec0f07afb12a879c09569e7   file2
+100640 blob 03f128cf48cb203d938805e9f3e13b808d1773e9   file3
```

Итак, видно, что по сравнению с деревом 1 в дереве два исчез file1, у file2 изменился SHA1 хеш, и добавился новый file3.

Также можно заметить, что у удаленного файла file1 и добавленного файла file3 одинаковый SHA1 хеш - отсюда можно сделать вывод, что было произведено переименование из file1 в file3 без изменения содержимого.

Точно такую же работу производит и git, точнее его команда git-diff-tree. Она выводит разницу между двумя деревьями в читабельном для человека виде.

```
        $ git-diff-tree b2efb2a7e48025c4d185080412a6ba1121ee6c59
493a5292de0b743e77aa190921da56d33599b59e
        :100644 000000 03f128cf48cb203d938805e9f3e13b808d1773e9
0000000000000000000000000000000000000000 D      file1
        :100644 100644 b973e639605e63466ea5ba09b04a545f16946ca8 4dd2746869211aedfec0f07a
        fb12a879c09569e7 M      file2:000000 100644
0000000000000000000000000000000000000000 03f128cf48cb203d938805e9
        f3e13b808d1773e9 A      file3
```

Если git-diff-tree вызывать с ключом -p, то она сгенерирует патч, который будучи применен к tree1, приведет его к tree2.

```
        git-diff-tree -p b2efb2a7e48025c4d185080412a6ba1121ee6c59
493a5292de0b743e77aa190921da56d33599b59e
        diff --git a/file1 b/file1
        deleted file mode 100644
        index 03f128c..0000000
        --- a/file1
        +++ /dev/null
        @@ -1 +0,0 @@
        -File1
        diff --git a/file2 b/file2
        index b973e63..4dd2746 100644
        --- a/file2
        +++ b/file2
        @@ -1 +1,2 @@
         File2
        +Secondline
        diff --git a/file3 b/file3
        new file mode 100644
        index 0000000..03f128c
        --- /dev/null
        +++ b/file3
        @@ -0,0 +1 @@
        +File1
```

Как видно по патчу, git-diff-tree не учел, что файл file1 был переименован в file3, и сгенерировал патч так, как будто file1 удалили, и file3 добавили заново.

Но как мы знаем, blob-ы у file1 и file3 совпадают - поэтому можно точно

сказать что было переименование. Для того, чтобы git-diff-tree стал обращать на это внимание, ему надо передать ключик -M (detect renames).

Тогда он сгенерирует особый патч-переименование. К сожалению, стандартная команда patch не может прикладывать такие патчи-переименования, так что потребуется прикладывать этот патч к дереву с помощью команды git-apply.

```
        $ git-diff-tree -M -p b2efb2a7e48025c4d185080412a6ba1121ee6c59
493a5292de0b743e77aa190921da56d33599b59e
        diff --git a/file2 b/file2
        index b973e63..4dd2746 100644
        --- a/file2
        +++ b/file2
        @@ -1 +1,2 @@
         File2
        +Secondline
        diff --git a/file1 b/file3
        similarity index 100%
        rename from file1
        rename to file3
```

Итак, объекты-деревья служат для объединения blob-ов и других деревьев в иерархию, аналогичную файловой системе. Деревья хранят биты доступа, хеши содержимого и имена объектов, поэтому между двумя деревьями может быть разница только по этим параметрам.

Такие параметры как времена создания, изменения и доступа файла, а также создатель или владелец файла, в деревьях не записываются. Некоторое подобие такой информации хранят объекты-commit'ы, о которых я расскажу в следующий раз.

А пока вам домашнее задание: создайте пустое объект-дерево (вообще без файлов) и запостите сюда его SHA1

Часть 4

Git не был бы системой контроля версий, если бы не позволял хранить историю изменений деревьев.

Для хранения истории в git используются специальные объекты-commit'ы. Каждому коммиту соответствует ровно одно дерево. Коммиты также хранят информацию о "предках" этого дерева - то есть ссылки на т.н. родительские коммиты. Можно считать, что коммит указывает, из каких деревьев (их может быть несколько) произошло текущее дерево, а также

кто в этом виноват (автор коммита) и по какой причине (сообщение коммита).

У самого первого коммита в репозитории не может быть предков. Он считается начальным коммитом, и считается что до него ничего не было. В репозитории git обычно бывает только один начальный коммит, а все остальные происходят из него. Можно считать начальный коммит Адамом и Евой :) У большинства коммитов предок всего один, поэтому часто история коммитов линейна. Авраам родил Исаака, Исаак родил Иакова, Иаков родил Иуду и т.д. :)

Бывает, что несколько коммитов происходят от одного предка. В этом месте в истории появляется "развилка" - история начинает делиться на ветви ("колена", если продолжать аналогию с Библией).
Но бывают еще коммиты, у которых несколько родителей. Это т.н. коммиты-слияния (merges), и в общем-то, количество родителей у коммита не ограничено. Это действие противоположно вышеописанной "развилке", и объединяет ранее разделенные ветви. Так, породнились бы Капулетти и Монтекки, если бы Вильяму Шекспиру захотелось бы устроить в "Ромео и Джульетте" хэппи-энд.

Но довольно лирики. Если говорить формально, то сам объект-коммит - это простой текст в строго определенном формате. У каждого коммита есть соответствующее дерево (первая строчка), далее перечисляются родители (каждый родитель на отдельной строчке), а дальше указываются "автор" коммита и время создания коммита.

После этого указывается т.н. "committer" - человек, который записал коммит в историю репозитория. Вместе с committer записывается и время, когда коммит был записан в историю. После чего оставшиеся строки занимает сообщение о коммите - произвольный текст, который указал автор при создании коммита.

Обычно поля committer и author совпадают, если автор сразу же после создания коммита записывает его в репозиторий. Но бывает и другая ситуация, когда один человек создает коммит, а другой применяет этот коммит к своему репозиторию. Тогда author и committer будут совершенно разными людьми. И committer и author указываются в формате Имя <email>, который считается стандартным форматом для задания адреса электронной почты.

Создать объект-коммит можно с помощью команды git-commit-tree.
У этой команды один обязательный параметр - SHA1 объекта-дерева, соответствующего коммиту. Также может быть несколько необязательных параметров, перечисляющих родителей коммита.

На вход (stdin) этой команде надо подать сообщение коммита. Остальные

поля (author и commiter) команда заполняет сама. Если определенным образом не сконфигурировать git, по умолчанию в качестве имени автора будет использоваться имя текущего пользователя, а в качестве адреса электронной почты - <login текущего="текущего" пользователя@имя="пользователя@имя" хоста="хоста">. Также в качестве даты создания коммита и записи его в историю, будет использоваться текущая дата.

Ну что долго объяснять, вот вам пример:

```
$ mkdir ~/tmp/gitguts4
$ cd ~/tmp/gitguts4
$ git-init
Initialized empty Git repository in .git/

$ echo "file1" > file1
$ echo "file2" > file2

$ git-hash-object -w file1
e2129701f1a4d54dc44f03c93bca0a2aec7c5449

$ git-hash-object -w file2
6c493ff740f9380390d5c9ddef4af18697ac9375

$ echo -e "10644 blob e2129701f1a4d54dc44f03c93bca0a2aec7c5449\tfile1
10644 blob 6c493ff740f9380390d5c9ddef4af18697ac9375\tfile2" | git-mktree

eaa27839f1ccaa6e087202ec96c479ee2c93b71e

$ export GIT_AUTHOR_NAME="Git Guts"
$ export GIT_AUTHOR_EMAIL="gitguts@localhost"
$ export GIT_COMMITTER_NAME="$GIT_AUTHOR_NAME"
$ export GIT_COMMITTER_EMAIL="$GIT_AUTHOR_EMAIL"

$ echo "Initial commit" | faketime -t 200001010000 git-commit-tree
eaa27839f1ccaa6e087202ec96c479ee2c93b71e

a215c9607c843ff00bc1490fb51271b6211070a2
```

Обратите внимание - я использовал задание автора и коммитера через переменные окружения, а также использовал faketime для того, чтобы задать время создания коммита и время сохранения его в репозитории. Дело в том, что если вы попытаетесь повторить мои действия, и не будете использовать переменные окружения и faketime, то в коммите будет другое время, и другие авторы/коммитеры, и вы не сможете полностью воспроизвести последующие действия, так как у коммитов будут другие

имена и другое содержимое.

Посмотреть содержимое созданного объекта можно, используя утилиту git-cat-file

```
$ git-cat-file commit a215c9607c843ff00bc1490fb51271b6211070a2
tree eaa27839f1ccaa6e087202ec96c479ee2c93b71e
author Git Guts <gitguts@localhost> 946674000 +0300
committer Git Guts <gitguts@localhost> 946674000 +0300

Initial commit
```

Ну, думаю не стоит объяснять, где что находится в этом объекта - все и так очевидно. Созданный коммит не имеет предков - то есть является сиротой. :) Давайте создадим ему потомков, чтобы было веселее. Для того, чтобы указать родителя коммита, в параметры git-commit-tree надо добавить -p <sha1 родителя>, ну например как показано в следующем примере:

```
$ echo "Abraham" | faketime -t 200001010100 git-commit-tree
eaa27839f1ccaa6e087202ec96c479ee2c93b71e \
    -p a215c9607c843ff00bc1490fb51271b6211070a2

09e01781c4c8245acd0728184d7cb8d9c7579901

$ git-cat-file commit 09e01781c4c8245acd0728184d7cb8d9c7579901
tree eaa27839f1ccaa6e087202ec96c479ee2c93b71e
parent a215c9607c843ff00bc1490fb51271b6211070a2
author Git Guts <gitguts@localhost> 946677600 +0300
committer Git Guts <gitguts@localhost> 946677600 +0300

Abraham
```

Итак, Авраам рожден :) Видите, в коммите добавилось поле parent, с указанием родительского коммита. Добавим же Исаака - сына его :) (для этого укажем в поле "родитель" SHA1 Авраама).

```
$ echo "Isaac" | faketime -t 200001010200 git-commit-tree
eaa27839f1ccaa6e087202ec96c479ee2c93b71e \
    -p 09e01781c4c8245acd0728184d7cb8d9c7579901

420a3454070a1767c3fe7107f9dc753d8ff3722c

$ git-cat-file commit 420a3454070a1767c3fe7107f9dc753d8ff3722c
tree eaa27839f1ccaa6e087202ec96c479ee2c93b71e
```

```
parent 09e01781c4c8245acd0728184d7cb8d9c7579901
author Git Guts <gitguts@localhost> 946681200 +0300
committer Git Guts <gitguts@localhost> 946681200 +0300

Isaac
```

Для простоты для всех создаваемых коммитов я указываю одно и то же дерево. В большинстве реальных случаев деревья таки будут чем-то отличаться.

Давайте теперь посмотрим на историю вновь созданного Исаака. Просмотром истории в git занимается программа-историк git-log.

```
$ PAGER=cat git-log 420a3454070a1767c3fe7107f9dc753d8ff3722c
commit 420a3454070a1767c3fe7107f9dc753d8ff3722c
Author: Git Guts <gitguts@localhost>
Date:   Sat Jan 1 02:00:00 2000 +0300

Isaac

commit 09e01781c4c8245acd0728184d7cb8d9c7579901
Author: Git Guts <gitguts@localhost>
Date:   Sat Jan 1 01:00:00 2000 +0300

Abraham

commit a215c9607c843ff00bc1490fb51271b6211070a2
Author: Git Guts <gitguts@localhost>
Date:   Sat Jan 1 00:00:00 2000 +0300

Initial commit
```

Я использовал PAGER=cat, чтобы git-log не запускал для просмотра истории команду less (ну или что там у вас поставлено вместо $PAGER), а просто тупо вываливал информацию в терминал.
Итак, по выводу истории видно, что от начального коммита произошел Авраам, а от Авраама - Исаак :)

Продолжим наши уроки Ветхого завета и продемонстрируем "развилку". У Исаака, как известно, было два сына - Исав и Иаков. Исав - старший брат, Иаков - младший. Продемонстрируем это в терминах git.

```
$ echo "Esau" | faketime -t 200001010300 git-commit-tree
eaa27839f1ccaa6e087202ec96c479ee2c93b71e \
    -p 420a3454070a1767c3fe7107f9dc753d8ff3722c
```

```
        de10f1828d215892dcebd00c4f7738141bfd0df7

        $ echo "Jakob" | faketime -t 200001010400 git-commit-tree
eaa27839f1ccaa6e087202ec96c479ee2c93b71e \
                -p 420a3454070a1767c3fe7107f9dc753d8ff3722c

        f77f5c2466a3f8674d3ec8785b13a910d32e5a75
```

Вот, таким вот макаром были рождены эти два брата. Для того, чтобы отобразить их отношения, обычного текстового формата недостаточно. Поэтому будем использовать графическую программу gitk. В качестве параметров я перечислил SHA1-имена братьев.

```
        $ gitk de10f1828d215892dcebd00c4f7738141bfd0df7
f77f5c2466a3f8674d3ec8785b13a910d32e5a75
```

Результат работы можно увидеть вот тут:

Исав и Иаков

Как видно, налицо развилочка. В дальнейшем каждая ветвь может получить отдельное развитие.

Часть 5

Для тех, кто раньше работал только с CVS или CVS++ (ну то есть Subversion), концепция коммитов-слияний (merge) может оказаться не очень понятной. Так что я решил обратиться к классике для иллюстрации слияний.

Помните, Николай Васильевич Гоголь, "Женитьба"... Если кто позабыл, я напомню монолог Агафьи Тихоновны (полный текст см. тут: http://az.lib.ru/g/gogolx_n_w/text_0080.shtml).

Право, такое затруднение -- выбор! Если бы еще один, два человека, а то четыре. Как хочешь, так и выбирай. Никанор Иванович недурен, хотя, конечно, худощав; Иван Кузьмич тоже недурен. Да если сказать правду. Иван Павлович тоже хоть и толст, а ведь очень видный мужчина.

Прошу покорно, как тут быть? Балтазар Балтазарыч опять мужчина с достоинствами. Уж как трудно решиться, так просто рассказать нельзя, как трудно! Если бы губы Никанора Ивановича да приставить к носу Ивана Кузьмича, да взять сколько-нибудь развязности, какая у Балтазара Балтазарыча, да, пожалуй, прибавить к этому еще дородности

Ивана Павловича -- я бы тогда тотчас же решилась. А теперь поди подумай! просто голова даже стала болеть. Бедная Агафья Тихоновна. Ведь в то доисторическое время еще не было современных систем контроля версий, разве что CVS, который был придуман еще во времена динозавров. А ведь задача создания идеального жениха из лучших качеств четырех претендентов - типичная задача слияния! В нижеприведенном примере я намеренно не буду использовать встроенные в git автоматические системы слияния, чтобы показать внутреннюю кухню. В жизни все будет гораздо проще.

Итак, начнем с создания репозитория и инициализации переменных окружения:

```
$ mkdir ~/tmp/gitguts5
$ cd ~/tmp/gitguts5
$ git init-init
Initialized empty Git repository in .git/

$ export GIT_AUTHOR_NAME="Git Guts"
$ export GIT_AUTHOR_EMAIL="gitguts@localhost"
$ export GIT_COMMITTER_NAME="$GIT_AUTHOR_NAME"
$ export GIT_COMMITTER_EMAIL="$GIT_AUTHOR_EMAIL"
```

Теперь создадим файл-заготовку, который мы будем использовать для заполнения наших деревьев - перечень человеческих достоинств, которые ценит Агафья Тихоновна:

```
$ echo -e "Губы\nНос\nРазвязность\nДородность" > virtues-template
$ cat virtues-template

Губы
Нос
Развязность
Дородность
```

Обращаю внимание что текст в файле virtues-template записан в системной кодировке. Для того, чтобы была воспроизводимость всех проделанных действий, перед занесением в git я буду переводить текст из системной кодировки в utf-8. На самом деле git не предъявляет никаких требований к кодировке, но тем не менее я бы рекомендовал держать коммиты либо в ASCII (то есть писать их по английски), либо в utf-8, если вы хотите чтобы ваши коммиты читал кто-нибудь вне России.

Итак, следующим этапом будет создание начального коммита. Его дерево будет состоять из одного файла - virtues, который будет аналогичен

файлу virtues-template, только переведен в utf-8 для воспроизводимости. Почему начальный коммит должен быть именно таким - я объясню позже. Итак, создание начального коммита (ничего нового для тех, кто внимательно читал предыдущие выпуски):

```
$ iconv -t utf-8 < virtues-template > virtues
$ git-hash-object -w virtues

111f008f40b32148b325098b0b3ad1fe46df0aef

$ echo -e "100644 blob 111f008f40b32148b325098b0b3ad1fe46df0aef\tvirtues" | git-mktree

f387e3ef43d001f614ef1a5a8c6ac4a0996c7c3c

$ echo "Обычный человек" | iconv -t utf-8 | faketime -t 200001010000 git-commit-tree f387e3ef43d001f614ef1a5a8c6ac4a0996c7c3c

6173ad1924d1221b82fe940e96eca4ec914b4b6c
```

Итак, у нас есть начальный коммит (без предков), с сообщением "Обычный человек". Зачем? Потому что именно так работает автоматическое слияние. Ему нужен "общий предок" всех сливаемых коммитов, чтобы понять, что у них общее, а что - различается.

Теперь давайте создадим коммиты, соответствующие женихам Агафьи Тихоновны: Никанор Иваныч, Иван Кузьмич, Балтазар Балтазарыч и Иван Павлович. Отличаться эти коммиты будут тем, что вместо

```
 Губы
 Нос
 Развязность
 Дородность
```

будет

```
 Губы Никанора Иваныча
 Нос Никанора Иваныча
 Развязность Никанора Иваныча
 Дородность Никанора Иваныча
```

Ну, вы надеюсь поняли. Добавлять "Никанора Иваныча" в конце каждой строчки мы будем с помощью простейшего скрипта на sed, вот иллюстрация:

```
$ sed 's/$/ Никанора Иваныча/' virtues-template
```

```
 Губы Никанора Иваныча
 Нос Никанора Иваныча
 Развязность Никанора Иваныча
 Дородность Никанора Иваныча
```

Итак, создадим эти четыре коммита:
Никанор Иваныч:

```
$ PARENT="6173ad1924d1221b82fe940e96eca4ec914b4b6c"
$ sed 's/$/ Никанора Иваныча/' virtues-template | iconv -t utf-8 > virtues-NI
$ git-hash-object -w virtues-NI

929db472b24b02eb991257c26376609e4da6966b

$ echo -e "100644 blob 929db472b24b02eb991257c26376609e4da6966b\tvirtues" | git-
mktree

0ade4416fb17c0eb8037265a2e0405db102164eb

$ echo "Никанор Иваныч" | iconv -t utf-8 | faketime -t 200001010100 git-commit-
tree \
    0ade4416fb17c0eb8037265a2e0405db102164eb -p $PARENT

f683f1e38e0339885c5ff31ed3efa6f5060c57b3
```

Иван Кузьмич:

```
$ sed 's/$/ Ивана Кузьмича/' virtues-template | iconv -t utf-8 > virtues-IK
$ git-hash-object -w virtues-IK

b4bd4d3eae566ac8d58a5a4dc8dccf06a8a8602c

$ echo -e "100644 blob b4bd4d3eae566ac8d58a5a4dc8dccf06a8a8602c\tvirtues" | git-
mktree

f7509f166ee816355654e1fd8b21bfa616272d38

$ echo "Иван Кузьмич" | iconv -t utf-8 | faketime -t 200001010100 git-commit-
tree \
    f7509f166ee816355654e1fd8b21bfa616272d38 -p $PARENT

ff7a5afbdf16e8ade231e1adec6e9a44838c44d0
```

Балтазар Балтазарыч:

```
        $ sed 's/$/ Балтазар Балтазарыча/' virtues-template | iconv -t utf-8 > virtues-
BB

        $ git-hash-object -w virtues-BB

        66d2a243ba12d21ba95ce44e757681a4d4e05428

        $ echo -e "100644 blob 66d2a243ba12d21ba95ce44e757681a4d4e05428\tvirtues" | git-
mktree

        f56b93f223725f10602f0c404114671ed04ad743

        $ echo "Балтазар Балтазарыч" | iconv -t utf-8 | faketime -t 200001010100 git-
commit-tree \
                f56b93f223725f10602f0c404114671ed04ad743 -p $PARENT

        c89d03e1e07c2a2fdb52bc85615bed628b4de202
```

Иван Павлович:

```
        $ sed 's/$/ Ивана Павловича/' virtues-template | iconv -t utf-8 > virtues-IP
        $ git-hash-object -w virtues-IP

        9c9c6c6f479e13ce061e82863c17e3bc03ce8960

        $ echo -e "100644 blob 9c9c6c6f479e13ce061e82863c17e3bc03ce8960\tvirtues" | git-
mktree

        3d2459538e8ff3809d557758649a5a9c9393c124

        $ echo "Иван Павлович" | iconv -t utf-8 | faketime -t 200001010100 git-commit-
tree \
            3d2459538e8ff3809d557758649a5a9c9393c124 -p $PARENT

        2762e87bf446e3f886996d8e984b69a6204b4305
```

Дерево этих коммитов будет выглядеть в gitk примерно так:

```
        gitk 2762e87bf446e3f886996d8e984b69a6204b4305\
            c89d03e1e07c2a2fdb52bc85615bed628b4de202\
            ff7a5afbdf16e8ade231e1adec6e9a44838c44d0\
            f683f1e38e0339885c5ff31ed3efa6f5060c57b3

           27,19 КБ
```

Каждый из женихов отличается от общего предка - "Обычного человека" персонализированным набором качеств.

Агафья Тихоновна хотела бы создать идеального жениха, скомбинировав эти персонализированные отличия. В этом нам поможет слияние.

В классическом случае операция слияния - это

1. Формирование нового дерева, которое каким-то образом включает в себя изменения, произошедшие в сливаемых ветках со времени их общего предка.
2. Формирование нового коммита с этим деревом, в качестве предков которого указаны все сливаемые коммиты

Автоматическая система слияния git в многих случаях может сама "слить" ветки, без участия пользователя. Например, если изменения в сливаемых ветках затрагивают разные файлы, или один и тот же файл, но изменяемые строчки не пересекаются. Новое дерево в таком случае формируется автоматически.

В нашем же запущенном случае в каждом коммите-женихе все строчки изначального "Обычного человека" заменены - поэтому при слиянии получается конфликт. Например, чьи губы должны быть у результата слияния - Никанора Иваныча или Балтазара Балтазарыча? Или может Ивана Павловича?

В таких ситуациях единственное решение принять должен человек. В нашем случае - Агафья Тихоновна. Благодаря Гоголю Агафья уже разрешила все конфликты слияния, постановив, что у идеального жениха должно быть:
  * Губы Никанора Иваныча
  * Нос Ивана Кузьмича
  * Развязность Балтазара Балтазарыча
  * Дородность Ивана Павловича

Вот с таким вот идеальным деревом мы и создадим коммит-слияние:

```
$ echo "Губы Никанора Иваныча" > ideal-template
$ echo "Нос Ивана Кузьмича" >> ideal-template
$ echo "Развязность Балтазара Балтазарыча" >> ideal-template
$ echo "Дородность Ивана Павловича" >> ideal-template
$ cat ideal-template


Губы Никанора Иваныча
Нос Ивана Кузьмича
```

```
        Развязность Балтазара Балтазарыча
        Дородность Ивана Павловича

        $ iconv -t utf-8 <ideal-template >ideal
        $ git-hash-object -w ideal

        aaad89b8229eab40cde73cd3afe05cfb689f8a85

        $ echo -e "100644 blob aaad89b8229eab40cde73cd3afe05cfb689f8a85\tvirtues" | git-
mktree

        3bb4ea25e93d5962d6a568330aea334161d55009

        $ echo "Идеальный жених Агафьи Тихоновны" | iconv -t utf-8 | faketime -t
200001010200 \
            git-commit-tree 3bb4ea25e93d5962d6a568330aea334161d55009\
            -p 2762e87bf446e3f886996d8e984b69a6204b4305\
            -p c89d03e1e07c2a2fdb52bc85615bed628b4de202\
            -p ff7a5afbdf16e8ade231e1adec6e9a44838c44d0\
            -p f683f1e38e0339885c5ff31ed3efa6f5060c57b3

        31e839af8dbd1315ceaa9dbbcc2c2c71ff91d797
```

Как видно, от обычных коммитов с одним предком, коммит-слияние отличается лишь тем, что у него несколько предков, каждый указан как -p <SHA1>

Посмотрим же на результат в gitk:

```
gitk 31e839af8dbd1315ceaa9dbbcc2c2c71ff91d797
```

34,36 КБ

Как видно, коммит-слияние в gitk графически отображается как соединение всех веток в одну точку. В классическом случае (без использования всяческих хаков или низкоуровневых команд), когда git видит коммит-слияние, он считает что все изменения, которые были в сливаемых ветках, в точке слияния были согласованы, и все конфликты поправлены.

Если в дальнейшем сливаемые ветки будут развиваться дальше по отдельности, то при очередном слиянии git будет считать коммит-слияние общим предком, и конфликтовать будут только изменения, произошедшие после коммита-слияния.

Итак, подведем итоги:

Коммит-слияние с технической точки зрения ненамного сложнее обычного

коммита. Главной проблемой при слияниях является "Право, такое затруднение -- выбор!", говоря словами Агафьи Тихоновны. Во многих случаях этот выбор может сделать сам git, предоставляя несколько стратегий автоматического слияния. Но в сложных случаях без помощи человека в решении конфликтов не обойтись.

Обзор стратегий автоматического слияния я пожалуй оставлю на потом, а в следующем выпуске расскажу о текстовых ссылках (refs), которые значительно облегчают работу с git. Именно они, а не SHA1 имена объектов, используются для повседневной работы в git. Stay tuned!

Часть 6

SHA1-имена объектов как уникальные идентификаторы - это конечно удобно. Для роботов. Люди как-то привыкли называть друг друга по коротким именам, а не по кодам ДНК.

Символьные имена объектов в git называются ссылка (references), и хранятся в каталоге .git/refs.
Делятся они на три типа:

1. Теги (tags) - символьные имена любых объектов из базы, которые не меняются со временем. Расположены в .git/refs/tags/

2. Ветки (heads, branches) - символьные имена объектов-коммитов, которые меняются при добавлении нового коммита в цепочку. Расположены в .git/refs/heads/

3. Удаленные ветки (remotes) - ветки специального вида, которые предназначены для слежения за ветками (heads) в других репозитариях. Лежат в .git/refs/remotes/

Кроме этого, есть несколько специальных ссылок, которые по историческим соображениям лежат вне каталога .git/refs и их названия пишутся В РЕГИСТРЕ БЛОНДИНОК. Из всех БЛОНДИНОЧНЫХ ссылок для пользователей наиболее важными являются HEAD, ORIG_HEAD и MERGE_HEAD.

HEAD - это особая ссылка, она показывает на коммит, который соответствует рабочей копии. Если быть точным, это не просто ссылка на коммит, это ссылка на "текущую ветку".

Хватит теории, пора иллюстрировать. Создаем простой репозитарий и попробуем те самые высокоуровневые инструменты, которыми раньше не пользовались.

```
$ mkdir ~/tmp/gitguts6
$ cd ~/tmp/gitguts6
$ git-init

Initialized empty Git repository in .git/

$ git-mktree </dev/null

4b825dc642cb6eb9a060e54bf8d69288fbee4904

$ TREE=4b825dc642cb6eb9a060e54bf8d69288fbee4904
$ export GIT_AUTHOR_NAME="Git Guts"
$ export GIT_AUTHOR_EMAIL="gitguts@localhost"
$ export GIT_COMMITTER_NAME="$GIT_AUTHOR_NAME"
$ export GIT_COMMITTER_EMAIL="$GIT_AUTHOR_EMAIL"
$ echo "Первый коммит" | iconv -t utf-8 | faketime -t 200001010000 git-commit-tr

ee $TREE
e678a27ffe7b84211f09b0e397b1c6e287aee392
```

Итак, был создан новый коммит с пустым деревом. Теперь создадим символьное имя для этого коммита - пусть это будет имя "refs/heads/master". Для этого надо лишь создать обычный файл .git/refs/heads/master и записать в него SHA1-имя коммита. Примерно так:

```
$ echo e678a27ffe7b84211f09b0e397b1c6e287aee392 > .git/refs/heads/master
$ ln -sf refs/heads/master .git/HEAD
```

Как видно из примера, я создал ссылку-ветку refs/heads/master, после чего сделал интересную операцию - символическую ссылку .git/HEAD на эту ветку.

Новую созданную ветку может показать команда git-branch:

```
$ git-branch
* master
```

Звездочка около master означает что сейчас ссылка HEAD указывает именно на эту ветку.

Чтобы увидеть, как сменой ссылки HEAD git может "переключаться" между ветками, создадим вторую ветку, указывающую на тот же коммит:

```
$ echo e678a27ffe7b84211f09b0e397b1c6e287aee392 > .git/refs/heads/other
$ git-branch
* master
  other
$ ln -sf refs/heads/other .git/HEAD
$ git-branch
  master
* other
```

Мы видим как простым переставлением символьной ссылки .git/HEAD выбирается "текущая" ветка.

После того, как было создано символьное имя объекта, можно получить по нему SHA1-имя с помощью команды git-rev-parse:

```
$ git-rev-parse refs/heads/master

e678a27ffe7b84211f09b0e397b1c6e287aee392

$ git-rev-parse refs/heads/other

e678a27ffe7b84211f09b0e397b1c6e287aee392
```

Вместо refs/heads/master можно использовать heads/master или master. Также работают БЛОНДИНИСТЫЕ ссылки типа HEAD:

```
$ git-rev-parse HEAD

e678a27ffe7b84211f09b0e397b1c6e287aee392
```

Используя git-rev-parse и задание ссылок, можно произвести и простую операцию "коммит в ветку", с которой обычно начинается знакомство с git. Вот эта операция, пошагово:

```
$ PARENT=`git-rev-parse HEAD` # SHA1 текущего коммита
```

создаем новый коммит, используя в качестве родителя коммит HEAD

```
$ echo "Коммит в ветку other" | iconv -t utf-8 | faketime -t 200001010100 git-commit-tree $TREE -p $PARENT

283f22289f768361b854a78f1764dc7f1bd9b822
```

переставляем ссылку HEAD на новый коммит

```
$ echo 283f22289f768361b854a78f1764dc7f1bd9b822 > .git/HEAD
```

смотрим, по прежнему ли мы на ветке other?

```
$ git-branch

  master
* other
```

Заметили магию? Из-за того, что .git/HEAD - символическая ссылка на .git/refs/heads/other, запись SHA1 нового коммита в .git/HEAD на самом деле записывает новый коммит в refs/heads/other, затирая предыдущее значение. Теперь ссылка refs/heads/other указывает на новый коммит.

Вот схема:
Было:

```
HEAD -> refs/heads/other -> старый коммит
```

Стало:

```
HEAD -> refs/heads/other -> новый коммит -> старый коммит
```

То, что раньше пришлось делать вручную - теперь делается через механизм веток и HEAD! Для того, чтобы добавить новый коммит в ветку - надо просто повторить вышеуказанную процедуру. Можно даже сделать это в одну строчку, и при этом совсем избежать указаний SHA1.

магическая строчка, коммитящая в ветку

```
$ echo "Еще один коммит в ветку other" | iconv -t utf-8 | faketime -t
200001010200 \
    git-commit-tree $TREE -p `git-rev-parse HEAD` > .git/HEAD
```

git-log без указания коммита показывает и сторию HEAD

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

afd309cb9fe66dc314ed54c272a2d26a1b7a01be Еще один коммит в ветку other
283f22289f768361b854a78f1764dc7f1bd9b822 Коммит в ветку other
e678a27ffe7b84211f09b0e397b1c6e287aee392 Первый коммит
```

Вот так - в ветке other теперь было создано уже три коммита.
Если же теперь переставить ссылку HEAD на refs/heads/master, точно
такой же процедурой можно добавлять коммиты в ветку master:

```
$ ln -sf refs/heads/master .git/HEAD
$ git-branch
* master
  other

$ echo "Теперь коммит в ветку master" | iconv -t utf-8 | faketime -t
200001010300 \
    git-commit-tree $TREE -p `git-rev-parse HEAD` > .git/HEAD

$ PAGER=cat git-log --pretty=oneline

22339820c0dd6758be9cd940db0306d4020f7c9f Теперь коммит в ветку master
e678a27ffe7b84211f09b0e397b1c6e287aee392 Первый коммит
```

Ну и под занавес - посмотрите как эти ветки выглядят в gitk.

```
$ gitk --all
```

Параметр --all говорит gitk показывать все символьные ссылки, а не
только те, которые доступны из HEAD. Поэтому мы увидим все две ветки,
которые были созданы.

Часть 7

Ветки в git - как ветки деревьев, постоянно обновляются и растут. Одно
и то же символьное имя ветки (refs/heads/foo) может указывать на разные
коммиты в разные моменты времени. В отличие от веток, теги (tags) -
специально созданы для неизменяющихся по времени ссылок.

Символьные имена для тегов лежат в .git/refs/tags. Каждому имени тега
может соответствовать один объект в базе git. Это может быть любой из
ранее перечисленных типов объектов - блобы, деревья, коммиты.
Символьное имя, указывающее на блоб, дерево или коммит, в терминологии
git называется легковесным (lightweight) тегом. Легковесный он потому
что кроме SHA1-имени, никакой другой информации не записывается.
Такие легковесные теги можно создавать путем записи SHA1-имени объекта
в файл в директории .git/refs/tags/<имя тега>.

Настоящие теги - тяжеловесные или аннотированные (annotated), состоят из двух частей. Первая часть - это объект базы git специального типа (tag). В этот объект записываются следующие данные:

* SHA1 объекта, на который указывает аннотированный тег.
* Тип этого объекта (blob, tree, commit или tag) (да, бывают теги указывающие на теги!)
* Символьное имя тега
* Дата и время создания тега
* Имя и e-mail создателя тега (в таком же формате как имя автора коммита)
* Кусок произвольных данных на усмотрение создателя тега

После чего объект-тег записывается в базу git, и в .git/refs/tags/<имя тега> пишется SHA1 объекта-тега.

В тот самый кусок произвольных данных могут быть записано сообщение тега (по смыслу аналогичное сообщению коммита), а также в него можно внедрить GPG-подпись объекта. Такой тег будет называться подписанным (tag).

Вот тут и проявляется магия git - создавая подписанный тег на определенный коммит, на самом деле подписывается и сам коммит, и вся его история, и все деревья, составляющие историю, и все блобы, "висящие на ветках этих деревьев". То есть все, на что можно "дотянуться" по ссылкам от коммита.

Ладно, хватит теории, давайте перейдем к практике.
Обычные, легковесные теги, как я уже говорил раньше, можно создавать просто записывая SHA1-имя объекта в файл в директории refs/tags/.
Однако правильней создавать их через утилиту git-tag <имя тега> [<имя объекта>]

Если имя объекта не указывать, то по умолчанию тег будет указывать на тот же коммит, на который указывает ссылка HEAD.
Сначала создадим объект на который будет указывать тег. Для иллюстрации я создаю простейший blob, хотя обычно теги указывают на объекты-коммиты.

```
$ mkdir ~/tmp/gitguts7
$ cd ~/tmp/gitguts7
$ git-init
$ export GIT_AUTHOR_NAME="Git Guts"
$ export GIT_AUTHOR_EMAIL="gitguts@localhost"
$ export GIT_COMMITTER_NAME="$GIT_AUTHOR_NAME"
$ export GIT_COMMITTER_EMAIL="$GIT_AUTHOR_EMAIL"
$ echo "Testing blobs" > blobtest
```

```
$ git-hash-object -w blobtest

717c935c292fee3dca4c2e5f335f27b657895368
```

Теперь создаем легковесный тег

```
$ git-tag lighttag 717c935c292fee3dca4c2e5f335f27b657895368
$ cat .git/refs/tags/lighttag

717c935c292fee3dca4c2e5f335f27b657895368
```

Как видно, по содержанию легковесные теги ничем не отличаются от бранчей - это обычные файлы с SHA1 объекта внутри.
Кстати, команда git-tag без параметров (или git-tag -l) выведет список тегов.

Теперь создадим аннотированный тег (с помощью git-tag -a). Для создания аннотированного тега необходимо указывать практически то же, что и для создания коммита - то есть имя автора тега, дату создания и сообщение.

Ну и чтобы получилось одно и то же время, я опять воспользуюсь программой faketime. В отличие от git-commit-tree, команда git-tag более высокоуровневая, и сообщение для тега можно задавать прямо в командной строке, используя параметр -m.

```
$ faketime -t 200001010000 git-tag -m 'Test annotated tag' -a annotated_tag
lighttag
```

Заметьте, вместо использования SHA1 blob-а, я использовал ранее заданное имя lighttag, которое указывало на этот blob. В этом и весь смысл тегов - давать символьные имена объектам из базы.
Теперь давайте посмотрим, что же получилось в итоге

```
$ git-rev-parse annotated_tag

40f93cdf3db19ab20109c81f113a7ccb8b921827

$ git-cat-file tag annotated_tag

object 717c935c292fee3dca4c2e5f335f27b657895368
type blob
tag annotated_tag
tagger Git Guts <gitguts@localhost> 946674000 +0300
```

```
        Test annotated tag
```

Первая команда (git-rev-parse), позволяет посмотреть, каков SHA1 самого объекта-тега. Вторая команда распечатывает содержимое объекта-тега. В нем можно увидеть SHA1 блоба (первая строчка), тип объекта (вторая строчка), символьное имя (третья строчка), информация об авторе и времени создания тега (четвертая строчка), а ниже - сообщение тега.

Команда создания подписанного тега очень похожа на команду создания обычного тега, просто вместо параметра -a надо передать параметр -s. К сожалению именно эта часть не будет воспроизводиться у читателей, так как у каждого должен быть свой собственный GPG-ключ для подписи.

Приведу лишь результаты выполнения команды:

```
        $ faketime -t 200001010000 git-tag -m 'Test annotated tag' -s signed_tag
lighttag
        $ git-cat-file tag signed_tag

        object 717c935c292fee3dca4c2e5f335f27b657895368
        type blob
        tag signed_tag
        tagger Git Guts <gitguts@localhost> 946674000 +0300

        Test annotated tag
        -----BEGIN PGP SIGNATURE-----
        Version: GnuPG v1.4.9 (GNU/Linux)

        iEYEABECAAYFAkf47VMACgkQ8SRmhxtswwQ53wCdHIGaU1ulxud4cUxWVp2pjU1d
        358AnAu0Xlti6ZhCSfp9/YToFd//ipcS
        =BQ9s
        -----END PGP SIGNATURE-----
```

Как видно, тут к сообщению добавилась подпись, созданная при помощи моего ключа.

Проверить, каким ключом был подписан коммит, можно с помощью git-tag -v

```
        $ git-tag -v signed_tag

        object 717c935c292fee3dca4c2e5f335f27b657895368
        type blob
        tag signed_tag
        tagger Git Guts <gitguts@localhost> 946674000 +0300
```

```
Test annotated tag
gpg: Подпись создана Вск 06 Апр 2008 19:33:39 MSD ключом DSA с ID 1B6CC304
gpg: Действительная подпись от "Damir Shayhutdinov <damir@altlinux.ru>"
```

Вот так!
Удалять теги можно с помощью git-tag -d, это я оставляю на
самостоятельную работу.

[<< Предыдущая] [ИНДЕКС] [Исправить] [src / Печать] [Следующая >>]

Обсуждение

[RSS]

- 1, Аноним (-), 23:27, 23/04/2011 [ответить] +/-

Спасибо, классно описано!

- 3, Gorlum (??), 11:24, 30/07/2015 [ответить] +/-

> А пока вам домашнее задание: создайте пустое объект-дерево (вообще без
файлов) и запостите сюда его SHA1

4b825dc642cb6eb9a060e54bf8d69288fbee4904
правильно ?

- 4, Аноним (-), 13:44, 02/10/2015 [ответить] +/-

> Мы видим как простым переставлением символьной ссылки .git/HEAD
> выбирается "текущая" ветка.

Это не соответствует действительности.

После
$ ln -sf refs/heads/master .git/HEAD
команда
git-branch
не покажет текущую ветку а покажет
* (no branch)
либо
* (HEAD detached from 41cdde7)

где 41cdde7 - это коммит - ближайший общий предок с какой-нибудь веткой.

Чтоб переключить ветки .git/HEAD должен быть обычным файлом (не ссылкой),
и нужно сделать
echo "ref: refs/heads/master" > .git/HEAD

> P.S.
> На момент написания оригинальной статьи (2008 год)
> может .git/HEAD и был ссылкой, но сейчас нет.

Добавить комментарий

Имя:

E-Mail:

Заголовок: Внтутреннее устройство Git (git

Текст:

Отправить