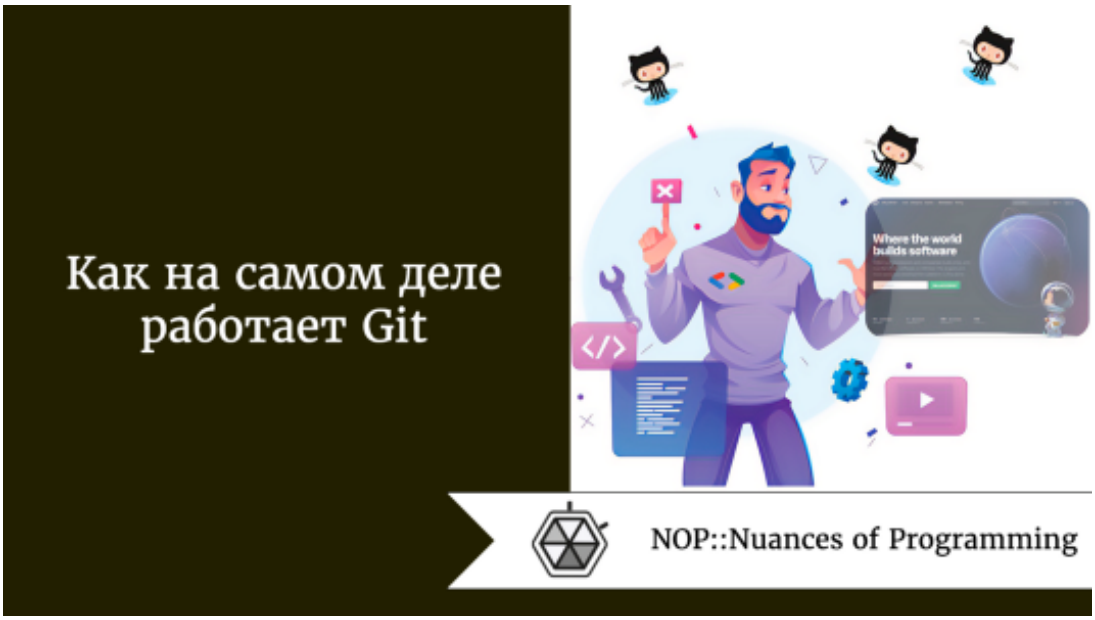
Как на самом деле работает Git

28.02.2021



В этой статье мы на реальном примере погрузимся во внутренние процессы Git. Если у вас еще не открыт терминал, то сделайте это, пристегните ремни и поехали!

Инициализируем Git-репозиторий

Вам наверняка уже доводилось инициализировать пустой репозиторий с помощью команды `git init`, но задумывались ли вы: что именно делает эта команда?

Создадим пустую папку, а в ней пустой проект Git. Вот как описывает `git init` официальная документация Git:

> *"Эта команда создает пустой репозиторий Git — каталог `.git` с подкаталогами `objects`, `refs/heads`, `refs/tags` и файлами шаблонов. Также создается начальный файл `HEAD`, который ссылается на HEAD основной ветви".*

Если мы проверим содержимое папки, то увидим следующую структуру:

```
$ tree -L 1 .git/
.git/
├── HEAD
├── config
├── description
├── hooks
├── info
├── objects
└── refs
```

Некоторые объекты отсюда мы рассмотрим позже.

Git — хранилище данных по типу "ключ-значение"

В сущности, Git — это контентно-адресуемая файловая система. А если проще — то база данных "ключ-значение". Вы помещаете любое содержимое в репозиторий, и Git возвращает вам уникальный идентификатор (ключ), которым потом можно воспользоваться для извлечения этого содержимого.

Для хранения значений в базе данных Git применяет команду `hash-object`:

> *"Вычисляет значение ID для объекта указанного типа с содержимым именованного файла (который может находиться вне рабочего дерева) и при необходимости записывает полученный объект в базу данных объектов. Сообщает ID объекта в стандартный вывод. Если <type> не указан, то по умолчанию используется значение blob".*

"Blob" — не что иное, как последовательность байтов. Большой двоичный объект (так расшифровывается blob) содержит точные данные в виде файла, но располагается в хранилище данных Git "ключ-значение», в то время как "настоящий" файл хранится в файловой системе.

Создадим такой объект:

```
$ echo hello | git hash-object --stdin -w
ce013625030ba8dba906f756967f9e9ca394464a
```

Мы воспользовались флагом `-w`, чтобы действительно записать объект в базу данных объектов, а не только отобразить его (достигается флагом `--stdin`).

Значение "hello" — это "значение" в хранилище данных Git, а хэш, возвращаемый функцией `hash-object`, это ключ. Теперь нам доступна противоположная операция — прочитать значение по его ключу с помощью команды `git cat-file`:

```
$ git cat-file -p ce013625030ba8dba906f756967f9e9ca394464a
hello
```

Можно проверить тип с помощью флага `-t`:

```
$ git cat-file -t ce013625030ba8dba906f756967f9e9ca394464a
blob
```

`git hash-object` размещает данные в папке `.git/objects/` (она же — база данных объектов). Убедимся в этом:

```
$ tree .git/objects/
.git/objects/
├── ce
│   └── 013625030ba8dba906f756967f9e9ca394464a
├── info
└── pack
```

Хэш-суффикс (в каталоге `ce`) такой же, как и тот, который мы получили из функции `hash-object`, но префикс здесь другой. Почему? Дело в том, что имя родительской папки содержит первые два символа нашего ключа. А это уже из-за того, что некоторые файловые системы ограничивают количество возможных подкаталогов. Введение промежуточного слоя смягчает эту проблему.

Сохраним еще один объект:

```
$ echo world | git hash-object --stdin -w
cc628ccd10742baea8241c5924df992b5c019f71
```

Как и ожидалось, теперь внутри `.git/objects/` есть два каталога:

```
$ tree .git/objects/
.git/objects/
├── cc
│   └── 628ccd10742baea8241c5924df992b5c019f71
├── ce
│   └── 013625030ba8dba906f756967f9e9ca394464a
├── info
└── pack
```

И опять же, каталог `cc`, содержащий префикс ключа, содержит остальную часть ключа в имени файла.

Древовидные объекты

Следующий объект Git, который мы рассмотрим, — дерево. Этот тип объекта решает проблему хранения имени файла и позволяет хранить группу файлов вместе.

Древовидный объект содержит записи. Каждая запись — это SHA-1 большого двоичного объекта (blob) или поддерева с соответствующим режимом, типом и именем файла. Определение `git-mktree` в документации гласит:

> *"Считывает данные стандартного ввода в нерекурсивном формате вывода `ls-tree` и создает древовидный объект. `mktree` нормализует порядок записей внутри дерева, поэтому предварительная сортировка входных данных не требуется. Имя построенного древовидного объекта записывается в стандартный вывод".*

Если вам интересно, что представляет собой формат `ls-tree`, то он выглядит так:

```
<mode> SP <type> SP <object> TAB <file>
```

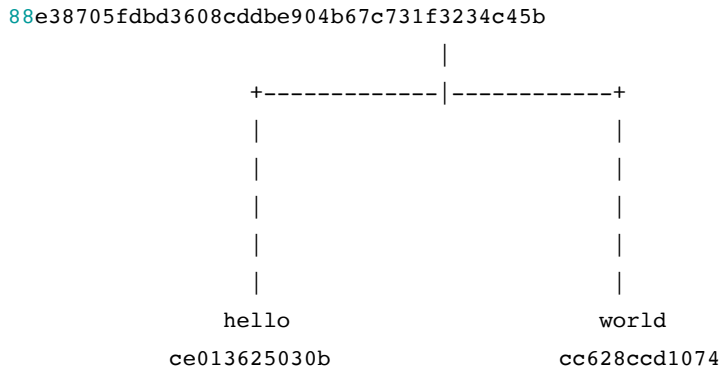
Давайте теперь свяжем два blob:

```
$ printf '%s %s %s\t%s\n' \
    100644 blob ce013625030ba8dba906f756967f9e9ca394464a hello.txt \
    100644 blob cc628ccd10742baea8241c5924df992b5c019f71 world.txt |
```

```
  git mktree
88e38705fdbd3608cddbe904b67c731f3234c45b
```

`mktree` возвращает ключ для вновь созданного древовидного объекта.

На этом этапе наше дерево визуализируется следующим образом:



Посмотрим на содержимое дерева:

```
$ git cat-file -p 88e38705fdbd3608cddbe904b67c731f3234c45b
100644 blob ce013625030ba8dba906f756967f9e9ca394464a hello.txt
100644 blob cc628ccd10742baea8241c5924df992b5c019f71 world.txt
```

И конечно, содержимое `.git/objects` обновилось соответственно:

```
$ tree .git/objects/
.git/objects/
├── 88
│   └── e38705fdbd3608cddbe904b67c731f3234c45b
├── cc
│   └── 628ccd10742baea8241c5924df992b5c019f71
├── ce
│   └── 013625030ba8dba906f756967f9e9ca394464a
├── info
└── pack
```

До сих пор мы еще не обновляли индекс. Для этого воспользуемся командой `git-read-tree`:

"Считывает информацию о дереве, заданную <tree-ish>, в индекс, но фактически не обновляет ни один из файлов, которые "кэширует" (см.: git-checkout-index[1])".

```
$ git read-tree 88e38705fdbd3608cddbe904b67c731f3234c45b
$ git ls-files -s
100644 ce013625030ba8dba906f756967f9e9ca394464a 0 hello.txt
100644 cc628ccd10742baea8241c5924df992b5c019f71 0 world.txt
```

Обратите внимание — в нашей файловой системе все еще нет файлов, так как значения пишутся непосредственно в хранилище данных Git. Чтобы "проверить"

файлы, используем команду `git-checkout-index`, которая копирует файлы из индекса в рабочее дерево:

```
git checkout-index 0 -a
```

`-a` означает “все”. Теперь у нас появилась возможность увидеть файлы:

```
$ ls
hello.txt world.txt
$ cat hello.txt
hello
$ cat world.txt
world
```

Бонус

`git-hash-object` выводит не такой SHA, как `openssl` SHA-1. В чем дело? В том, что для вычислений SHA-1 применяется следующая формула:

```
sha1("blob " + filesize + "\0" + data)
```

Поэтому чтобы получить тот же SHA-1, нужно проделать вот такой трюк:

```
# без вывода новой строки
$ echo -n hello | git hash-object --stdin -w
b6fc4c620b67d95f953a5c1c1230aaab5db5a1b0
$ printf 'blob 5\0hello' | openssl sha1
b6fc4c620b67d95f953a5c1c1230aaab5db5a1b0
```

Вывод

По ходу статьи мы сохранили два файла непосредственно в хранилище Git. Файлы еще не были видны нашей локальной файловой системе. Мы создали дерево и связали с ним два “больших двоичных объекта”, а затем перенесли файлы в рабочий каталог при помощи команды `git-checkout-index`.

Читайте также:

- Работа с GitHub Actions на маркетплейсе
- Реализация GitHub Action в контейнере Docker
- В гостях у GitHub Package Registry

Читайте нас в Telegram, VK и Яндекс.Дзен

Перевод статьи Maroun Maroun: How Git Really Works