

2022

CQG

Под капотом Git

По материалам workshop
«A Plumber's Guide to Git»

<https://alexwlchan.net/a-plumbers-guide-to-git/>

Copyright (c) 2012-2019 Alex Chan

Git commands: Porcelain & Plumbing



~ 110 команд (плюс опции):

- ❑ **Porcelain** – высокоуровневые команды, которые мы в основном и используем:

add, commit, merge, push, pull, rebase,...

- ❑ **Plumbing** – низкоуровневые команды:

hash-object, commit-tree, write-tree,...



Git под капотом



Папка .git

↑ Name	Ext	Size
↑ ..		<DIR>
hooks		18,848
info		701,778
logs		107,096
objects		373,640,482
refs		8,929
COMMIT_EDITMSG		541
COMMITMESSAGE		0
config		1,405
description		73
FETCH_HEAD		165,648
HEAD		53
index		1,618,298
ORIG_HEAD		41
packed-refs		615,112



Git под капотом. Объекты.



В Git единицей хранения данных является **объект**.

По содержанию объекта однозначно вычисляется соответствующий 40-символьный хеш **sha1**. Например, такой:

a37f3f668f09c61b7c12e857328f587c311e5d1d

Хеш, полученный из объекта, является именем файла в папке `.git/objects`, содержащего этот объект:

`.git/objects/a3/7f3f668f09c61b7c12e857328f587c311e5d1d`

(точнее, первые два символа хэша используются как имя папки в `.git/objects`, а остальные 38 – как имя файла в этой папке).



Git под капотом. Объекты.



В Git единицей хранения данных является **объект**.

По содержанию объекта однозначно вычисляется соответствующий 40-символьный хеш **sha1**. Например, такой:

a37f3f668f09c61b7c12e857328f587c311e5d1d

Хеш, полученный из объекта, является именем файла в папке `.git/objects`, содержащего этот объект:

`.git/objects/a3/7f3f668f09c61b7c12e857328f587c311e5d1d`

Поэтому структуру `.git` часто называют «контентно-адресуемой файловой системой»



Git под капотом. Объекты.



Объекты могут быть нескольких типов.

В объектах Git хранит почти всё:

- содержимое файлов пользователя
- их иерархию и имена (дерево)
- КОММИТЫ

Для экономии дискового пространства содержимое всех объектов дополнительно сжимается с помощью zlib.

Сначала объекты всех типов сохраняются как обычные файлы в папке `.git/objects`, а после `git gc` упаковывает их (если будет нужно) в `.pack`-файлы.



Git под капотом. Объекты.



Папка .git/objects

↑ ..	<DIR>
0c	<DIR>
0e	<DIR>
11	<DIR>
13	<DIR>
1a	<DIR>
22	<DIR>
24	<DIR>
2b	<DIR>

Папка .git/objects/0e

↑ ..	<DIR>
32a760dd6266ddc3117e4d7bfd5f65f18e22cf	1,410
924207c1eb3e1de666a4a69e34d258a451e302	171
a89b1db7e8efc48de32830ef4dda9f06cb7f72	93,502



Git под капотом



```
mkdir plumbing-repo  
cd plumbing-repo
```

```
git init
```

```
echo "An awesome aardvark admires the Alps" > animals.txt
```

```
git hash-object -w animals.txt
```

```
a37f3f668f09c61b7c12e857328f587c311e5d1d
```

В папке `.git/objects` появился файл!

```
.git/objects/a3/7f3f668f09c61b7c12e857328f587c311e5d1d
```


Git под капотом



```
git hash-object -w animals.txt
```

```
a37f3f668f09c61b7c12e857328f587c311e5d1d
```

```
.git/objects/a3/7f3f668f09c61b7c12e857328f587c311e5d1d
```

```
git cat-file -p a37f3f668f09c61b7c12e857328f587c311e5d1d
```

```
An awesome aardvark admires the Alps
```

Мы говорили, что объекты бывают разных типов.
Какого типа этот объект?

```
git cat-file -t a37f3f668f09c61b7c12e857328f587c311e5d1d  
blob
```

Git под капотом



```
git hash-object -w animals.txt
```

```
a37f3f668f09c61b7c12e857328f587c311e5d1d
```

```
.git/objects/a3/7f3f668f09c61b7c12e857328f587c311e5d1d
```

```
git cat-file -p a37f3f668f09c61b7c12e857328f587c311e5d1d
```

```
An awesome aardvark admires the Alps
```

Сотрем файл `plumbing_repo/animals.txt` и выполним команду

```
git cat-file -p a37f3f668f09c61b7c12e857328f587c311e5d1d >  
animals.txt
```

Git под капотом. Blobs.



```
git hash-object -w animals.txt
```

```
a37f3f668f09c61b7c12e857328f587c311e5d1d
```

```
.git/objects/a3/7f3f668f09c61b7c12e857328f587c311e5d1d
```

```
git cat-file -p a37f3f668f09c61b7c12e857328f587c311e5d1d
```

```
An awesome aardvark admires the Alps
```

Сотрем файл `plumbing_repo/animals.txt` и выполним команду

```
git cat-file -p a37f3f668f09c61b7c12e857328f587c311e5d1d >  
animals.txt
```

Снова появился `animals.txt`, с тем же содержимым:

```
An awesome aardvark admires the Alps
```

Git под капотом. Blobs.



```
echo "Big blue basilisks bawl in the basement" > animals.txt
```

```
git hash-object -w animals.txt
```

```
b13311e04762c322493e8562e6ce145a899ce570
```

Папка `.git/objects` содержит уже два объекта типа blob – старое содержание `animals.txt` и новое содержание `animals.txt`:

```
.git/objects/a3/7f3f668f09c61b7c12e857328f587c311e5d1d
```

```
.git/objects/b1/3311e04762c322493e8562e6ce145a899ce570
```

Git под капотом. Blobs.



```
echo "Big blue basilisks bawl in the basement" > animals.txt
```

```
git hash-object -w animals.txt
```

```
b13311e04762c322493e8562e6ce145a899ce570
```

```
.git/objects/a3/7f3f668f09c61b7c12e857328f587c311e5d1d
```

```
.git/objects/b1/3311e04762c322493e8562e6ce145a899ce570
```

Мы можем снова удалить файл **animals.txt** и выполнить

```
git cat-file -p b13311e04762c322493e8562e6ce145a899ce570 >  
animals.txt
```

А можно сохранить объект в файл с другим именем:

```
git cat-file -p b13311e04762c322493e8562e6ce145a899ce570 >  
alliteration.txt
```


Git под капотом. Blobs.



```
echo "Clueless cuttlefish crowd the curious crab" >  
c_creatures.txt
```

```
copy c_creatures.txt sea_creatures.txt
```

```
git hash-object -w c_creatures.txt  
ce289881a996b911f167be82c87cbfa5c6560653
```

```
git hash-object -w sea_creatures.txt  
ce289881a996b911f167be82c87cbfa5c6560653
```

Мы видим, что sha1-хеш одинаков у содержания обоих файлов – а это значит, что в `.git/objects` будет только один blob-объект для обоих файлов!

Git под капотом. Blobs.



a37f ... 5d1d —————> "An awesome aardvark admires the Alps"

b133 ... e570 —————> "Big blue basilisks bawl in the basement"

ce28 ... 0653 —————> "Clueless cuttlefish crowd the curious crab"

Только содержимое. Ничего об именах файлов.

Если мы хотим сохранять несколько файлов, а то и папок, нам надо их **имена** и в целом знать **структуру дерева файлов**.

Займемся этим.

Git под капотом. Blobs and ...



Stage (или Index) – промежуточная область («чистилище»), куда из рабочей папки помещаются файлы для будущей фиксации в репозитории.

В porcelain git это делается командой **git add**. После этого вторая porcelain команда **git commit** переносит текущее состояние stage (index) в локальный репозиторий.

Что при этом происходит «под умывальником»?

Git под капотом. Blobs and ...



Что при этом происходит «под умывальником»?

Во-первых, файл добавляется в индекс низкоуровневой командой

```
git update-index --add animals.txt
```

После этого другой plumbing-командой `ls-files` можно убедиться, что этот файл добавлен в индекс:

```
git ls-files  
animals.txt
```

Но это все в stage (index) области, ее можно модифицировать, и предыдущее состояние stage будет перетерто. Как навсегда сохранить содержание stage в репозитории?

Git под капотом. Blobs and ...



Но это все в stage (index) области, ее можно модифицировать, и предыдущее состояние stage будет перетерто. Как навсегда сохранить содержание stage в репозитории?

Выполним еще одну plumbing-команду:

```
git write-tree
```

```
dc6b8ea09fb7573a335c5fb953b49b85bb6ca985
```

Она вернула хеш! Это намекает на то, что у нас появился еще один git-объект. Посмотрим в `.git/objects`:

```
.git/objects/a3/7f3f668f09c61b7c12e857328f587c311e5d1d  
.git/objects/b1/3311e04762c322493e8562e6ce145a899ce570  
.git/objects/ce/289881a996b911f167be82c87cbfa5c6560653  
.git/objects/dc/6b8ea09fb7573a335c5fb953b49b85bb6ca985
```


Git под капотом. Blobs and ...



Что это за объект? Помните, для объектов, которые сохраняли содержание файла (но не имя!), мы инспектировали объект plumbing-командой `cat-file`?

```
git cat-file -p b13311e04762c322493e8562e6ce145a899ce570
```

```
Big blue basilisks bawl in the basement
```

Сделаем то же самое для нового объекта с хешем `dc6b8ea`, который мы создали, когда выполнили `write-tree`:

```
git cat-file -p dc6b8ea09fb7573a335c5fb953b49b85bb6ca985
```

```
100644 blob b13311e04762c322493e8562e6ce145a899ce570 animals.txt
```

Объект `dc6b8ea`, как оказалось, содержит **описание дерева файлов в виде списка**. В нашем случае файл один, поэтому строка одна.

Точнее говоря, такой объект содержит список ссылок на другие git-объекты, плюс некая мета-информация об этих объектах.

Git под капотом. Blobs and ...



```
git cat-file -p dc6b8ea09fb7573a335c5fb953b49b85bb6ca985
100644 blob b13311e04762c322493e8562e6ce145a899ce570 animals.txt
```

Здесь

100644	права доступа для файла (Git различает 100644 (non-executable) и 100755 (executable))
blob	тип объекта (поговорим ниже)
b13...570	хэш объекта, хранящего содержимое файла
animals.txt	имя файла

Этого достаточно, чтобы восстановить файл:

- ✓ мы знаем его имя
- ✓ мы знаем, где взять его содержимое (в каком объекте)
- ✓ мы знаем, какие права доступа у него должны быть

Git под капотом. Blobs and trees.



```
git cat-file -p dc6b8ea09fb7573a335c5fb953b49b85bb6ca985
100644 blob b13311e04762c322493e8562e6ce145a899ce570 animals.txt
```

Каков тип этого объекта? Снова воспользуемся plumbing-командой:

```
git cat-file -t dc6b8ea09fb7573a335c5fb953b49b85bb6ca985
tree
```

```
git cat-file -t b13311e04762c322493e8562e6ce145a899ce570
blob
```

Стало быть, объект **tree** содержит ссылку на объект типа **blob**, где хранится содержание файла.



Git под капотом. Blobs and trees.



Если бы мы добавили в индекс из рабочей папки `plumbing-repo` рядом с файлом `animals.txt` вложенную папку `plumbing-repo/underwater` с двумя файлами `d.txt` и `e.txt` внутри и сохранили бы такое дерево из индекса в репозиторий

```
git write-tree
```

```
11e2f923d36175b185cfa9dcc34ea068dc2a363c
```

мы бы увидели такую картину:

```
git cat-file -p 11e2f923d36175b185cfa9dcc34ea068dc2a363c
```

```
100644 blob b13311e04762c322493e8562e6ce145a899ce570 animals.txt
040000 tree 8972388aa2e995eb4fa0247ccc4e69144f7175b9 underwater
```

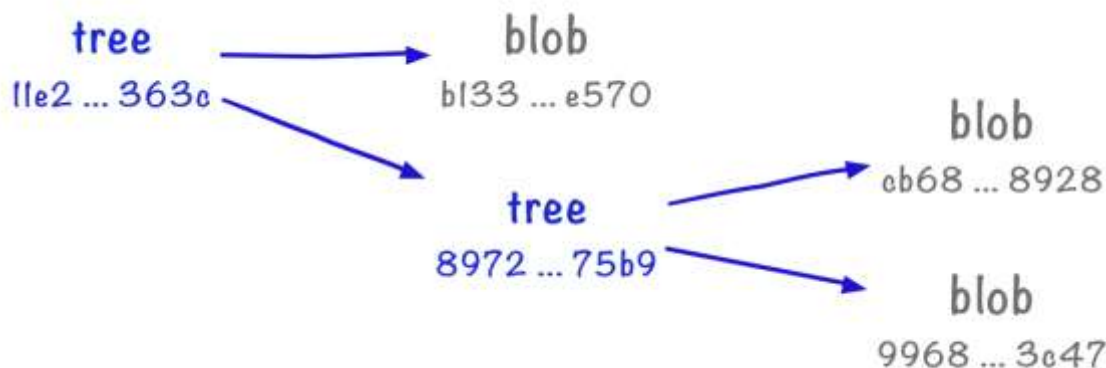
```
git cat-file -p 8972388aa2e995eb4fa0247ccc4e69144f7175b9
```

```
100644 blob cb68066907dd99eb75642bdbd449e1647cc78928 d.txt
100644 blob 9968b7362a7c97e237c74276d65b68ca20e03c47 e.txt
```

Git под капотом. Blobs and trees.



Это похоже на обычную файловую систему, правда?



Имея такой корневой объект типа `tree`, мы можем восстановить из репозитория всю структуру рабочей папки: объекты `tree` содержат информацию:

- ❑ об именах файлов (и ссылки на объекты типа `blob` с содержанием каждого файла)
- ❑ об именах вложенных папок (и ссылки на объекты типа `tree` для каждой такой папки)

Git под капотом. Blobs and trees.



Мы научились сохранять информацию о некотором состоянии дерева файлов проекта в репозиторий.

Один корневой объект типа `tree` хранит информацию о состоянии всех файлов, за которыми следит `git`, - «слепок», или «snapshot» - в один момент времени.

Другой корневой объект типа `tree` хранит такой «слепок» в другой момент времени.

Обратите внимание – объект `tree` хранит НЕ разницу между «слепокми», а полное самостоятельное описание всех файлов вашего проекта в тот момент, когда дерево было записано в репозиторий.

Git под капотом. Blobs, trees, **commits**.



Но как нам понять, какой объект `tree` нам нужен в данный момент?

Как понять, какой объект `tree` соответствует самому последнему состоянию системы, а какой соответствует, например, моменту, когда в прошлом июле был пофикшен некий конкретный баг?

Нам нужно снабдить каждый объект `tree` дополнительной мета-информацией о том, что это за дерево, к какому моменту оно относится.

Эта мета-информация для конкретного `tree` сохраняется отдельно от `tree`, в маленьком объекте третьего типа: **`commit`**.

Git под капотом. Blobs, trees, commits.



Продолжаем использовать низкоуровневые команды:

```
echo Initial commit | git commit-tree 11e2f92...  
65b080f1fe43e6e39b72dd79bda4953f7213658b
```

Мы снова получили от гита какой-то новый хэш, стало быть, созданся новый объект в папке `.git/objects`. Вот он:

```
.git/objects/11/e2f923d36175b185cfa9dcc34ea068dc2a363c  
.git/objects/65/b080f1fe43e6e39b72dd79bda4953f7213658b  
.git/objects/89/72388aa2e995eb4fa0247ccc4e69144f7175b9  
.git/objects/99/68b7362a7c97e237c74276d65b68ca20e03c47  
.git/objects/a3/7f3f668f09c61b7c12e857328f587c311e5d1d  
.git/objects/b1/3311e04762c322493e8562e6ce145a899ce570  
.git/objects/cb/68066907dd99eb75642bdbd449e1647cc78928  
.git/objects/ce/289881a996b911f167be82c87cbfa5c6560653  
.git/objects/dc/6b8ea09fb7573a335c5fb953b49b85bb6ca985
```

Git под капотом. Blobs, trees, commits.



Спросим его тип:

```
git cat-file -t 65b080f1fe43e6e39b72dd79bda4953f7213658b  
commit
```

Выведем его содержимое на экран:

```
git cat-file -p 65b080f1fe43e6e39b72dd79bda4953f7213658b  
tree 11e2f923d36175b185cfa9dcc34ea068dc2a363c  
author Alex Chan <alex@alexwlchan.net> 1520806168 +0000  
committer Alex Chan <alex@alexwlchan.net> 1520806168 +0000
```

Initial commit

Git под капотом. Blobs, trees, commits.



Нам осталось только научиться создавать цепочки коммитов, и мы научимся управлять историей проекта!

Давайте сделаем кое-какие изменения в рабочей папке и сохраним новое состояние в репозитории с помощью нового коммита:

```
git update-index --add c_creatures.txt
```

```
git write-tree
```

```
f999222f82d1ffe7233a8d86d72f27d5b92478ac
```

```
echo Adding c_creatures.txt | git commit-tree f999222 -p 65b080  
fd9274dbef2276ba8dc501be85a48fbfe6fc3e31
```

Параметр `-p 65b080` указывает, что прямым предком создаваемого коммита будет считаться коммит с хешем `65b080...`, то есть наш первый «Initial commit».

Git под капотом. Blobs, trees, commits.



```
echo Adding c_creatures.txt | git commit-tree f999222 -p 65b080
fd9274dbef2276ba8dc501be85a48fbfe6fc3e31
```

Посмотрим на вновь созданный коммит:

```
git cat-file -p fd9274dbef2276ba8dc501be85a48fbfe6fc3e31
tree f999222f82d1ffe7233a8d86d72f27d5b92478ac
parent 65b080f1fe43e6e39b72dd79bda4953f7213658b
author Alex Chan <alex@alexwlchan.net> 1520806875 +0000
committer Alex Chan <alex@alexwlchan.net> 1520806875 +0000
```

```
Adding c_creatures.txt
```

Git под капотом. Blobs, trees, commits.



Итак, у нас есть два коммита – начальный и последующий, для которого начальный является «родительским».

Отступим на секунду от plumbing, и воспользуемся porcelain-командой `log`, чтобы посмотреть историю, предшествующую нашему последнему коммиту:

```
git log fd9274dbef2276ba8dc501be85a48fbfe6fc3e31
commit fd9274dbef2276ba8dc501be85a48fbfe6fc3e31
Author: Alex Chan <alex@alexwlchan.net>
Date:    Sun Mar 11 22:21:15 2018 +0000
```

```
    Adding c_creatures.txt
```

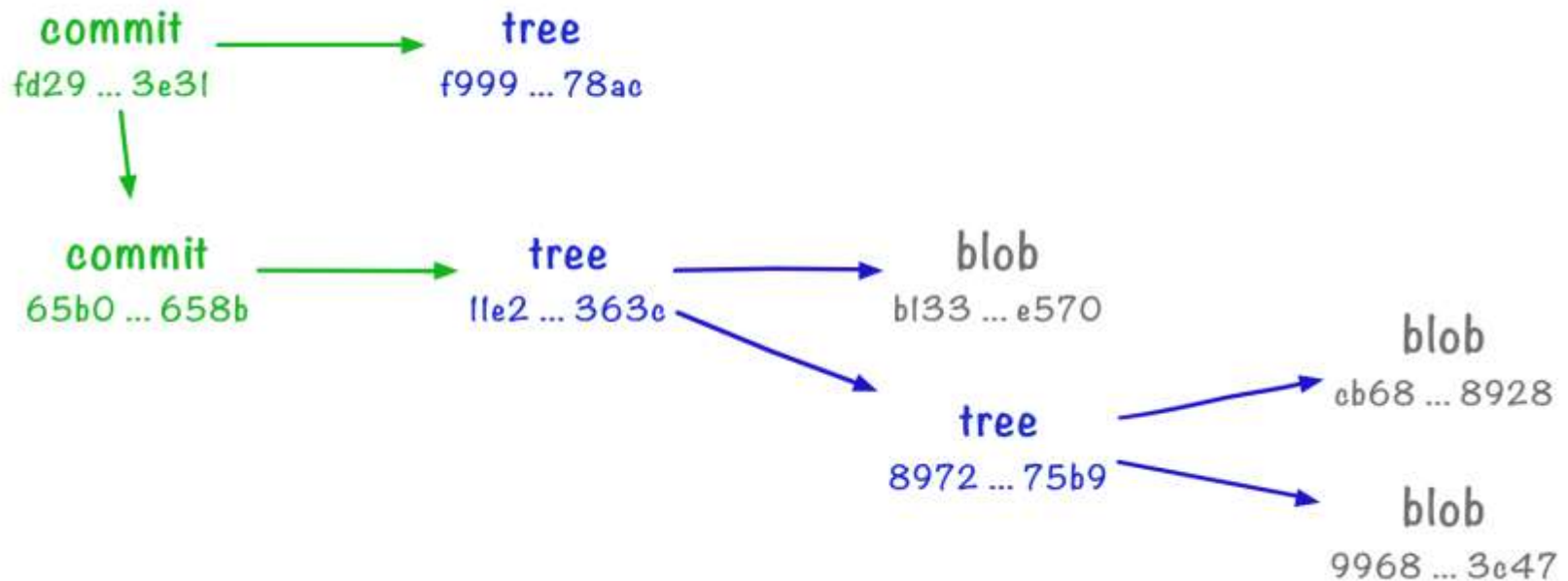
```
commit 65b080f1fe43e6e39b72dd79bda4953f7213658b
Author: Alex Chan <alex@alexwlchan.net>
Date:    Sun Mar 11 22:09:28 2018 +0000
```

```
    initial commit
```

Git под капотом. Blobs, trees, commits.



Что мы получили в итоге?



Git под капотом. Refs and branches



Рядом с папкой `.git/objects` лежит папка `.git/refs`.

До сих пор мы ее не замечали. Ее создал `git init`, в ней нет файлов пока, только две вложенные папки,

`.git/refs/heads` и
`.git/refs/tags`,

тоже пока пустые.

Git под капотом. Refs and branches



Что такое «ссылка» («reference») в git?

Это обычный именованный текстовый файл, внутри которого записана одна строка – хэш какого-то коммита.

Это позволяет ссылаться на коммиты не по их хэшу, а по «человеческому» имени.

Хэш и именованная ссылка абсолютно взаимозаменяемы при использовании в командах git.

Git под капотом. Refs and branches



Давайте создадим именованную ссылку на последний (второй) коммит в нашей «истории».

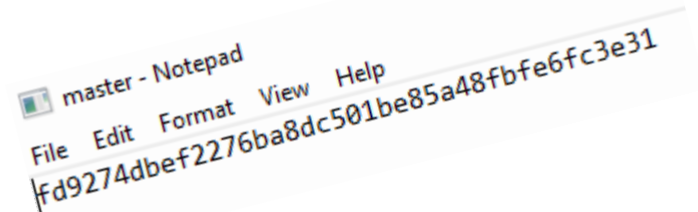
```
git update-ref refs/heads/master fd9274dbef227...
```

В папке `.git/refs/head` появился файл `master`

```
.git/refs/heads/master
```

и вот его содержимое:

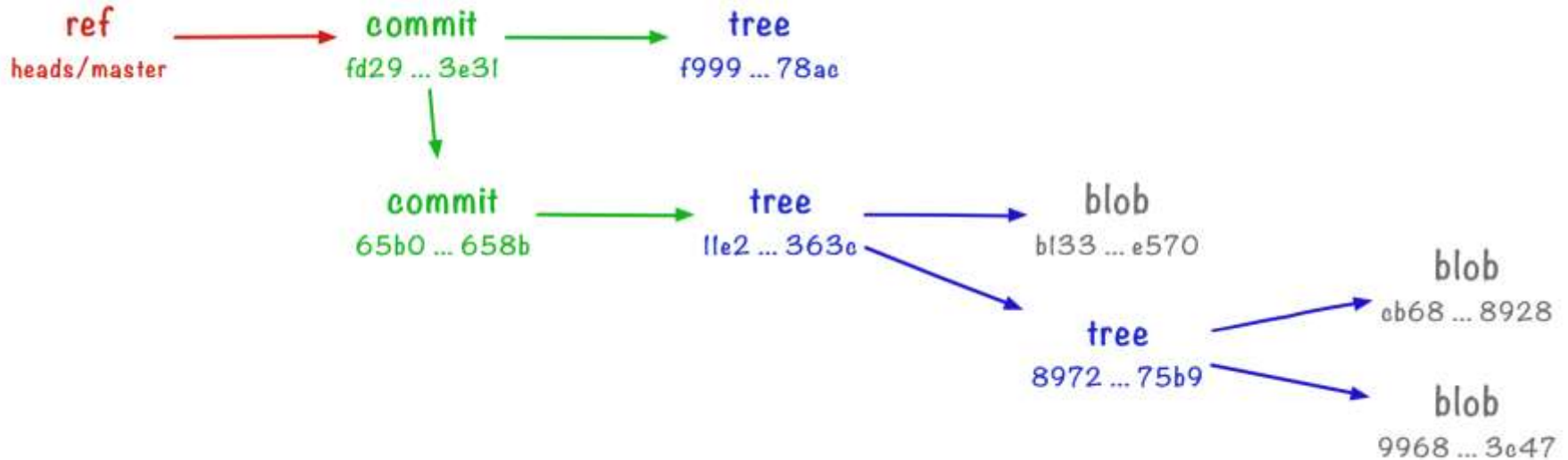
```
fd9274dbef2276ba8dc501be85a48fbfe6fc3e31.
```



Git под капотом. Refs and branches



Мы имеем такую картину:



Теперь на коммит `fd9274dbef2276ba8dc501be85a48fbfe6fc3e31` мы можем ссылаться, используя не `fd9274dbef2276ba8dc501be85a48fbfe6fc3e31`, и даже не `fd9274`, а слово `master`.

Git под капотом. Refs and branches



Проверим, что слово `master` теперь можно подставить в команду `git` ВМЕСТО ЭТОГО ХЭША:

```
git cat-file -p master
```

```
tree f999222f82d1ffe7233a8d86d72f27d5b92478ac
parent 65b080f1fe43e6e39b72dd79bda4953f7213658b
author Alex Chan <alex@alexwlchan.net> 1520806875 +0000
committer Alex Chan <alex@alexwlchan.net> 1520806875 +0000
```

```
Adding c_creatures.txt
```

Ну и конечно, зная имя ссылки (то есть, как мы выяснили, просто имя текстового файла с хэшем), можно попросить у `git` вернуть соответствующий хэш:

```
git rev-parse master
```

```
fd9274dbef2276ba8dc501be85a48fbfe6fc3e31
```

Git под капотом. Refs and branches



Ссылку такого вида, как `master` в нашем примере, сохраненную в папке `.git/refs/heads`, мы и называем `branch` («ветка»).

То есть создание «ветки» в `git` – это создание файла размером 41 байт (хэш плюс перенос строки).

И все! Почти мгновенно и практически бесплатно.

И мы только что с помощью команды

```
git update-ref refs/heads/master fd9274d
```

создали наш первый бранч!

Git под капотом. Refs and branches



Добавим еще один коммит к нашей истории, и посмотрим, что будет со ссылкой `master` в папке `.git/refs/heads`.

```
echo "Flying foxes feel fantastic but frightening" > foxes.txt
```

```
git update-index --add foxes.txt
```

```
git write-tree
```

```
c08523d153f6415cda07ea27948830407f243a37
```

```
echo Add foxes.txt | git commit-tree c08523d -p master
```

```
b023d92829d5d076dc31de5cca92cf0bd5ae8f8e
```

Мы получили новый коммит `b023d92`.

Git под капотом. Refs and branches



Посмотрим на его историю в компактном виде, снова возвращаясь к высокоуровневой команде `log`.

```
git log --oneline b023d92
b023d92 Add foxes.txt
fd9274d Adding c_creatures.txt
65b080f Initial commit
```

А что у нас с `master`?

```
git log --oneline master
fd9274d Adding c_creatures.txt
65b080f Initial commit
```

Вопреки ожиданиям, в мире низкоуровневых команд `master` сам собой не сдвинется на новый коммит. Его придется проапдейтить самим:

Git под капотом. Refs and branches



```
git log --oneline master
fd9274d Adding c_creatures.txt
65b080f Initial commit
```

Вопреки ожиданиями в мире низкоуровневых команд `master` сам собой не сдвинется на новый коммит. Его придется проапдейтить самим:

```
git update-ref refs/heads/master b023d92
```

```
git log --oneline master
b023d92 Add foxes.txt
fd9274d Adding c_creatures.txt
65b080f Initial commit
```

Git под капотом. Refs and branches



Создадим еще один бранч в нашем репозитории. Помним, что создание бранча – это просто создание именованного файла, содержащего хэш какого-то коммита. Пусть это будет самый первый коммит.

```
git update-ref refs/heads/dev 65b080f
```

Теперь в папке `.git/refs/heads` у нас два файла:

```
.git/refs/heads/dev  
.git/refs/heads/master
```

Git под капотом. Refs and branches



```
.git/refs/heads/dev  
.git/refs/heads/master
```

У Git можно спросить список всех бранчей porcelain-командой

git branch

```
dev  
* master
```

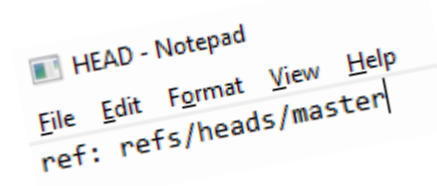
Звездочка * обозначает «текущий бранч». Это означает, что высокоуровневые команды будут добавлять коммиты, и вообще работать по умолчанию именно с ним.

Git под капотом. Refs and branches



Откуда Git узнает, что текущий бранч сейчас именно master? Он посмотрит в специальный текстовый файл `.git/HEAD`, и увидит там строку

```
ref: refs/heads/master
```



Итого, как в сказке, в файле `.git/HEAD` лежит ссылка на файл `.git/refs/heads/master`, в котором лежит ссылка на файл `.git/objects/b0/23d92829d5d076dc31de5cca92cf0bd5ae8f8e`, где лежит метаданная о последнем коммите в этой ветке и ссылка на другой файл в `.git/objects`, с tree-объектом, описывающем последний snapshot дерева файлов проекта путем указания на файлы `.git/objects`, хранящие tree-объекты с описанием подкаталогов проекта и blob-объекты с содержимым файлов проекта на этот момент.

Easy!

Git под капотом. Refs and branches



Мы можем сказать гиту, что текущий бранч теперь должен быть dev, изменив файл HEAD с помощью plumbing-команды `symbolic-ref`:

```
git symbolic-ref HEAD refs/heads/dev
```

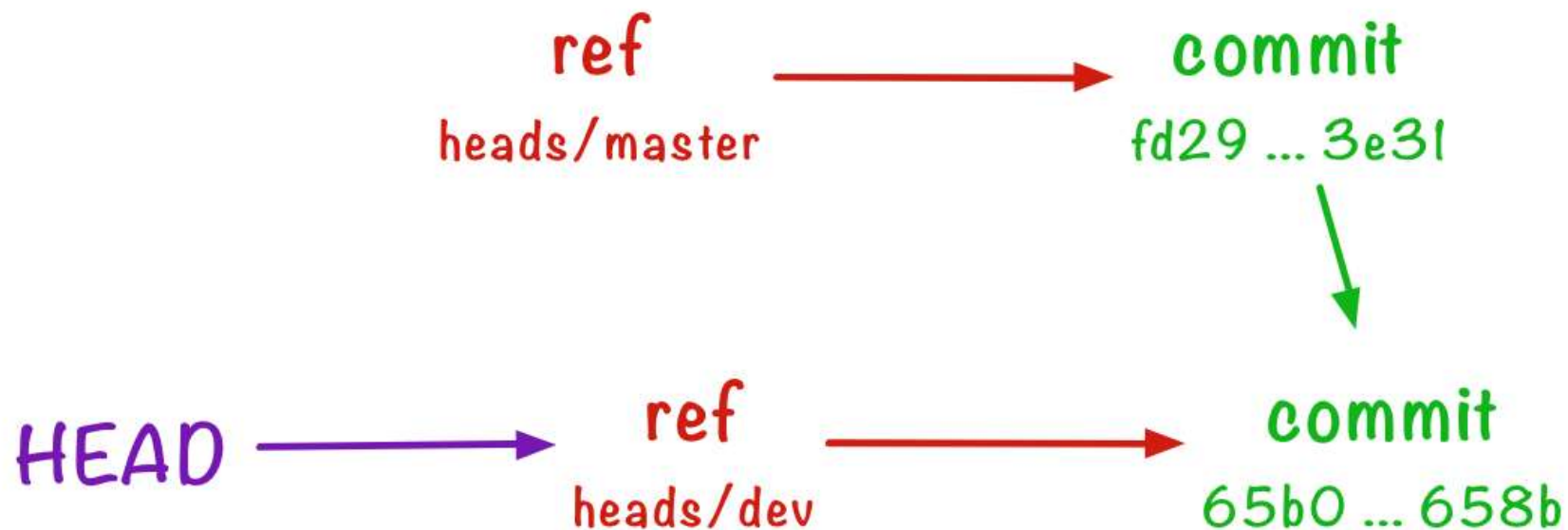
```
git branch
```

```
* dev  
master
```

Git под капотом. Refs and branches



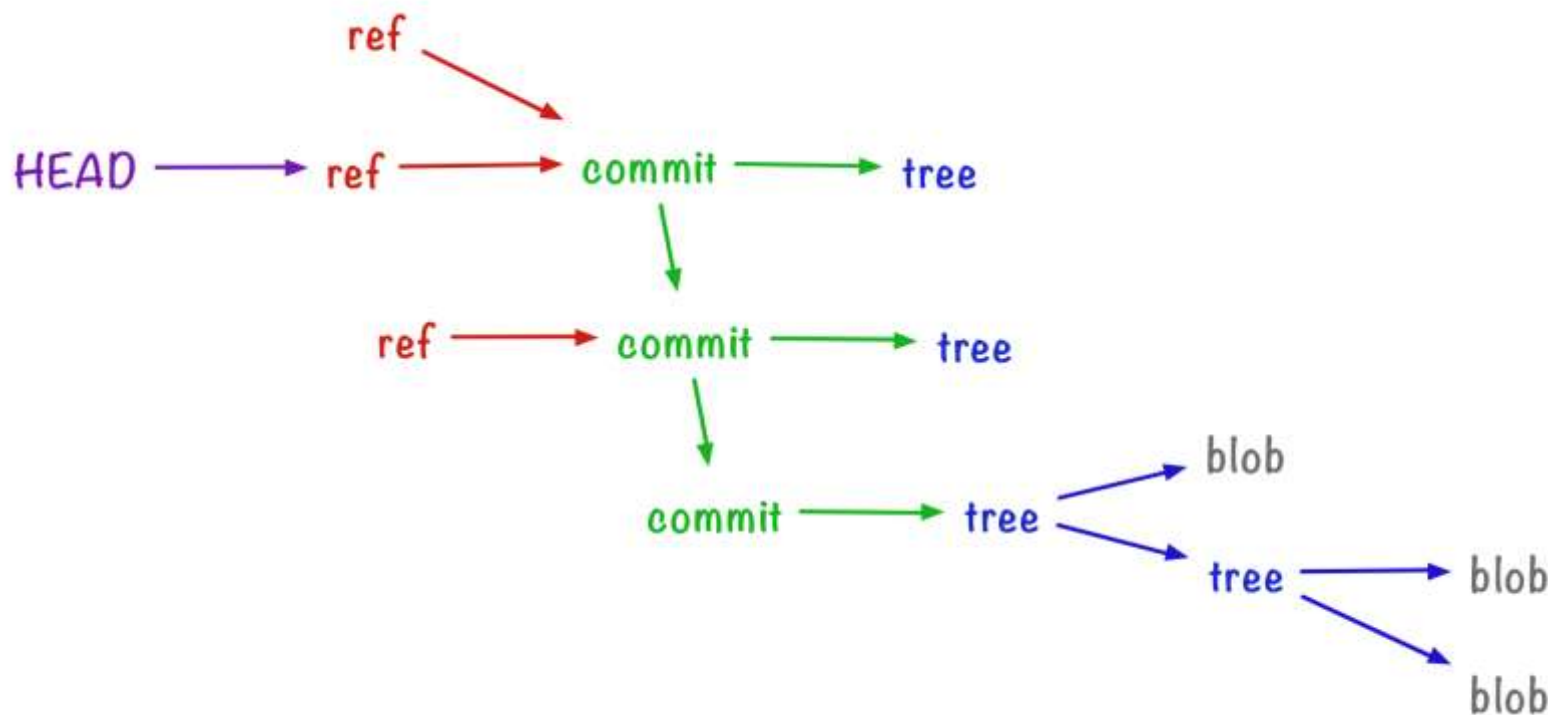
На картинке эта структура выглядит вот так:



Git под капотом. Refs and branches



А в целом вся структура Git выглядит вот так:



Reference



A Plumber's Guide to Git (English)

<https://alexwlchan.net/a-plumbers-guide-to-git/>



Alex Chan

@alexwlchan



Q&A