

ссылка на видео:

https://www.youtube.com/playlist?list=PLDyvV36pndZFHXjXuwA_NywNrVQO0aQqb
GIT - система контроля версий (как бы бд для историй проекта)

GIT INIT создаст новый репозиторий, то есть папку .git со всеми начальными настройками

```
~/project> git init  
Initialized empty Git repository in /Users/learn/project/.git/
```

При создании пустого репо в [user] ничего не будет вообще

```
~/project master> git config user.name "Ilya Kantor"  
~/project master> git config user.email iliakan@gmail.com  
~/project master> cat .git/config  
[core]  
    repositoryformatversion = 0  
    filemode = true  
    bare = false  
    logallrefupdates = true  
    ignorecase = true  
    precomposeunicode = true  
[user]  
    name = Ilya Kantor  
    email = iliakan@gmail.com
```

Настройки пользователя в терминале бывают:

1) общесистемные (на всю систему)

- /etc/gitconfig

--system

2) глобальны (на уровне конкретного пользователя)

~/.gitconfig

--global

3) локальные (на уровне проекта)

<project>/.git/config

--local

git изначально ищет настройки локально, если нет, то глобально, если тоже нет, то системно

ALIAS

`git config --global <alias> <что заменяем>`

```
~/project master> git c
git: 'c' is not a git command. See 'git --help'.

The most similar commands are
  checkout
  clone
  commit
  gc

✖ ~/project master> git config --global alias.c config

~/project master> git c --list
user.name=Ilya Kantor
user.email=iliakan@gmail.com
core.editor=/Applications/Sublime\ Text.app/Contents/SharedSupport/bin/subl -w
alias.c=config
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
core.ignorecase=true
core.precomposeunicode=true

~/project master> git config alias.sayhi '!echo "hello"; echo "from git"'

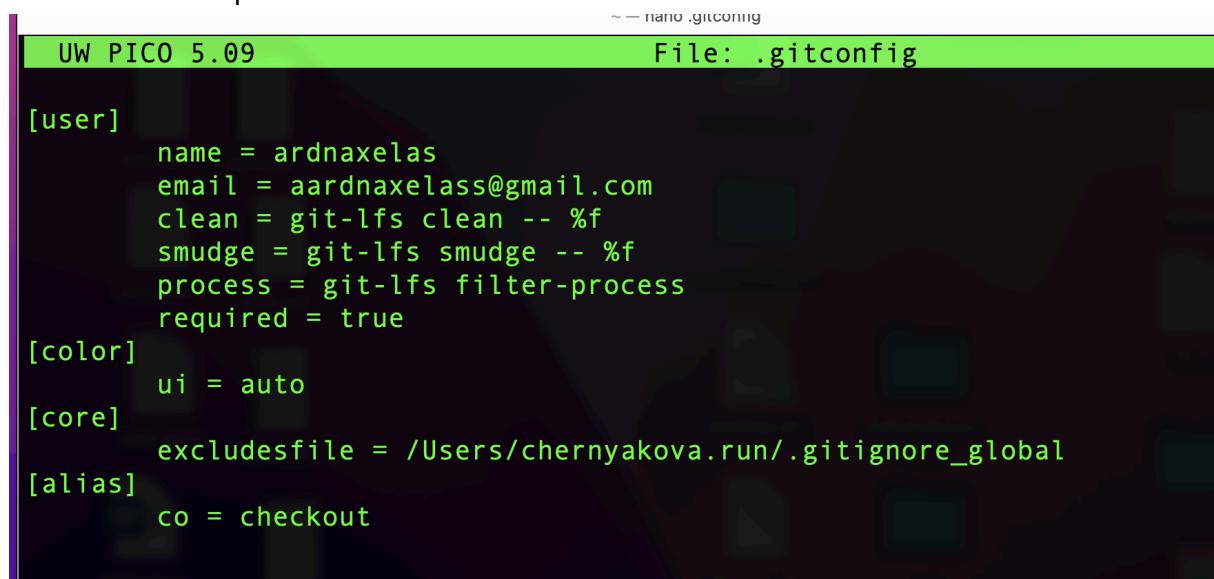
~/project master> git sayhi
hello
from git

~/project master> git config alias.sayhi '!git ...; git ...'
```

``git config --global <alias> <что заменяем>``

- **Глобальная настройка:** Эта команда применяет настройку глобально для всех репозиториев на вашем компьютере для текущего пользователя.
- **Файл конфигурации:** Настройки сохраняются в файле конфигурации, расположенному по пути `~/.gitconfig` (в домашней директории пользователя).
- **Применение:** Используйте эту команду, если хотите, чтобы настройка или алиас были доступны во всех репозиториях для текущего пользователя.

Глобальный конфиг:



The screenshot shows a terminal window titled "UW PICO 5.09" with the command "File: .gitconfig" at the top. The terminal displays the configuration file's content, which includes sections for [user], [color], [core], and [alias]. The [alias] section contains the entry "co = checkout". The terminal has a dark background with green text and a black border.

```
UW PICO 5.09                                         File: .gitconfig
~ — nano .gitconfig

[user]
    name = ardnaxelas
    email = aardnaxelass@gmail.com
    clean = git-lfs clean -- %f
    smudge = git-lfs smudge -- %f
    process = git-lfs filter-process
    required = true
[color]
    ui = auto
[core]
    excludesfile = /Users/chernyakova.run/.gitignore_global
[alias]
    co = checkout
```

``git config <alias> <что заменяем>``

- **Локальная настройка:** Эта команда применяет настройку только для текущего репозитория, в котором она выполняется.
- **Файл конфигурации:** Настройки сохраняются в файле конфигурации, расположенному по пути `<repo>/.git/config` (в директории текущего репозитория).
- **Применение:** Используйте эту команду, если хотите, чтобы настройка или алиас были доступны только в конкретном репозитории.

git help

`git help <команда>`
(например `git help config`)

при выводе информации используется удобная утилита less
ниже несколько ее функций

- 1) /<слово для поиска>
- 2) q - выход из листалки
- 3) n - вперед, shift+n - назад



git commit + частные случаи

git status - посмотреть что не проиндексировано

git add - проиндексировать файл, чтобы он стал отслеживаемым (**git add . - добавить все**)

git commit - сделать коммит (**git commit -m <коммит текст>**)

1 коммит = 1 изменения (атомарность)

Идентификатор коммита (его часть):

(по этому числу можно получить файлы на момент этого коммита)

```
~/project master •1> git commit
[master (root-commit) e740259] Create welcome page
 1 file changed, 9 insertions(+)
 create mode 100644 index.html
```

```
~/project master •1> git commit
[master (root-commit) e740259] Create welcome page
 1 file changed, 9 insertions(+)
 create mode 100644 index.html
```

первая половина числа - тип объекта (в данном случае 100 - файл).
 вторая половина - права на этот файл (гит их не сохраняет и бывает только 2 вида прав для гита: 755 - исполняемый для текущего пользователя, 644 - обычный)

`chmod 755` директория — предоставить владельцу полные **права** — `rwxr-xr-x`, а остальным пользователям **право** только на чтение и выполнение. `chmod 644` директория — предоставить **права** на чтение и запись для владельца каталога, а остальным пользователям и группам оставить только на чтение содержимого.

обычно: папка - 755, файл - 644

CHMOD

Если права поменять, то это равносильно изменению файла:

```
~/project master> chmod +x index.html

~/project master +1> git status
On branch master
Changes not staged for commit:
  modified:   index.html

no changes added to commit

~/project master +1> chmod -x index.html

~/project master> git status
On branch master
nothing to commit, working tree clean
```

GIT SHOW

МОЖНО ПОСМОТРЕТЬ ИНФОРМАЦИЮ О КОММите:

```
[chernyakova.run@MacBook-Air-Sasha-2 testRepo % git commit -m "first file"
[main (root-commit) 0e79ed6] first file
```

```
[chernyakova.run@MacBook-Air-Sasha-2 testRepo % git show 0e79
commit 0e79ed69efa963c8be65d6f8c3e315f51d87f436 (HEAD -> main)
Author: Sasha Chernyakova <alexandra13.04.2004@gmail.com>
Date:   Thu Mar 28 15:33:05 2024 +0300

    first file

diff --git a/hello.txt b/hello.txt
new file mode 100644
index 000000..ee47c1e
--- /dev/null
+++ b/hello.txt
@@ -0,0 +1 @@
+mfklls
```

Более развернутая информация:

```
[chernyakova.run@MacBook-Air-Sasha-2 testRepo % git show 0e79 --pretty=fuller
commit 0e79ed69efa963c8be65d6f8c3e315f51d87f436 (HEAD -> main)
Author: Sasha Chernyakova <alexandra13.04.2004@gmail.com>
AuthorDate: Thu Mar 28 15:33:05 2024 +0300
Commit: Sasha Chernyakova <alexandra13.04.2004@gmail.com>
CommitDate: Thu Mar 28 15:33:05 2024 +0300

    first file

diff --git a/hello.txt b/hello.txt
new file mode 100644
index 000000..ee47c1e
--- /dev/null
+++ b/hello.txt
@@ -0,0 +1 @@
+mfklls
```

Здесь предполагается что автор и коммитер это один человек, однако это может быть два разных человека

Это может быть полезно, когда есть большой проект с открытым исходным кодом, опытным разработчикам присылают улучшения/ошибки и тп. А потом разработчики могут это использовать, но указать автора:

```
~/project master> git commit --author='John Smith <john@me.com>' --date='...' █
```

ПУСТЫЕ ДИРЕКТОРИИ

git не умеет работать с пустыми каталогами и он не индексирует их и не коммитит, поэтому обычно создают файл **.gitkeep** в директории, он скрыт, а директория **не пустая**

GIT ADD

ОТМЕНА:

```
chernyakova.run@MacBook-Air-Sasha-2 testRepo % git status
On branch main
Your branch is based on 'origin/main', but the upstream is gone.
(use "git branch --unset-upstream" to fixup)

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .DS_Store
    papka/

nothing added to commit but untracked files present (use "git add" to track)
chernyakova.run@MacBook-Air-Sasha-2 testRepo % git add .
chernyakova.run@MacBook-Air-Sasha-2 testRepo % git status
On branch main
Your branch is based on 'origin/main', but the upstream is gone.
(use "git branch --unset-upstream" to fixup)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   .DS_Store
    new file:   papka/1.txt
    new file:   papka/2.txt
```

```
chernyakova.run@MacBook-Air-Sasha-2 testRepo % git rm -f .DS_Store  
rm '.DS_Store'  
chernyakova.run@MacBook-Air-Sasha-2 testRepo % git status  
On branch main  
Your branch is based on 'origin/main', but the upstream is gone.  
  (use "git branch --unset-upstream" to fixup)  
  
Changes to be committed:  
  (use "git restore --staged <file>..." to unstage)  
    new file:  papka/1.txt  
    new file:  papka/2.txt
```

```
chernyakova.run@MacBook-Air-Sasha-2 papka % git restore --staged 1.txt  
chernyakova.run@MacBook-Air-Sasha-2 papka % cd ..  
chernyakova.run@MacBook-Air-Sasha-2 testRepo % git status  
On branch main  
Your branch is based on 'origin/main', but the upstream is gone.  
  (use "git branch --unset-upstream" to fixup)  
  
Changes to be committed:  
  (use "git restore --staged <file>..." to unstage)  
    new file:  papka/2.txt  
  
Untracked files:  
  (use "git add <file>..." to include in what will be committed)  
    papka/1.txt
```

git restore - чтобы убрать файл из индекса

git ignore

можно добавить файл в gitignore, тогда git status и другие команды гита не будут его видеть.

Пример. Добавим директорию .idea в gitignore

```
~/project master •2...1> git status  
On branch master  
Changes to be committed:  
  (use "git reset HEAD <file>..." to unstage)  
  
    modified:   index.html  
    new file:   src/script.js  
  
Untracked files:  
  (use "git add <file>..." to include in what will be committed)  
  
    .gitignore  
  
~/project master •2...1> ls -a  
.          ..          .git          .gitignore      .idea          index.html      src
```

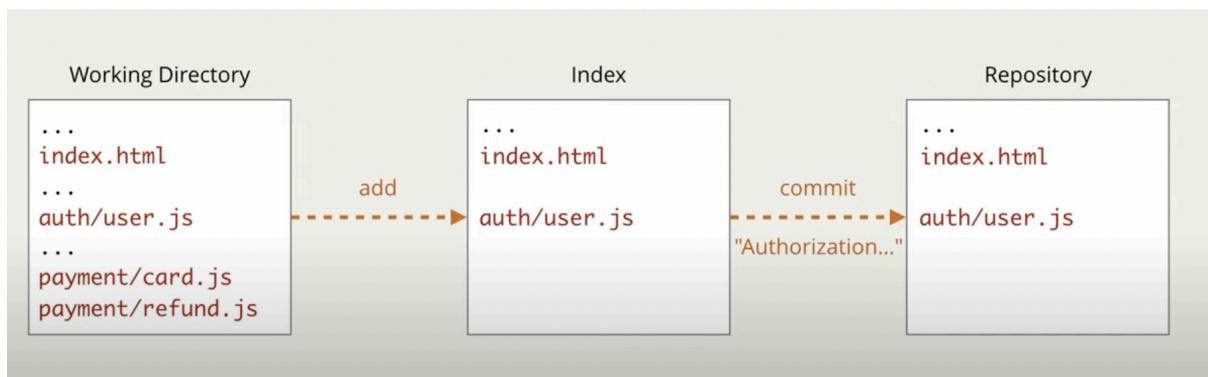
гит статус не видит этот файл, в реальности он есть

НО МОЖНО ЗАСТАВИТЬ ГИТ ДОБАВИТЬ В ИНДЕКС ДАЖЕ ТО, ЧТО В ИГНОРЕ: флаг -f. Это удобно например когда вся директория в игноре, а нужен только один файл из нее. Когда файл окажется в индексе, то всего его изменения будут отслеживаться независимо от gitignore.

```
~/project master •2...1> git add -f .idea/project.iml
```

1 коммит = 1 задача
АТОМАРНОСТЬ

Зачем нужна индексация?



Бывает что для изменения чего-то нам нужно изменить несколько файлов, которые на самом деле практически не связаны друг с другом. Тогда мы проиндексируем один и сделаем коммит, а потом другой и сделаем коммит. Двухступенчатая система как раз нужна чтобы не добавлять все сразу в один коммит, так это противоречит принципу атомарности

Также бывает что мы сделали в одном файле очень много изменений и не хотим коммитить все сразу, для этого можно индексировать выборочно (**флаг -p**):

```
~/project master +1> git add -p index.html
diff --git a/index.html b/index.html
index 7dadf8e..eb8ce1d 100644
--- a/index.html
+++ b/index.html
@@ -1,7 +1,7 @@
<!DOCTYPE html>
<html>
<head>
-    <title>Welcome</title>
+    <title>Welcome to Git!</title>
        <script src="src/script.js"></script>
</head>
<body>
Stage this hunk [y,n,q,a,d,j,J,g/,e,?]?
```

Тогда на каждый измененный фрагмент кода, будет спрашивать нужно ли проиндексировать

```
2.7 Git – Основы – Зачем нужен индекс?
<html>
<head>
-    <title>Welcome</title>
+    <title>Welcome to Git!</title>
        <script src="src/script.js"></script>
</head>
<body>
Stage this hunk [y,n,q,a,d,j,J,g/,e,?]? y
@@ -9,5 +9,7 @@
<script>
    hello();
</script>
+
...Unfinished changes...
</body>
</html>
Stage this hunk [y,n,q,a,d,K,g/,e,?]? n
```

```
~/project master •1+1> git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   index.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   index.html
```

```
<!DOCTYPE html>
<html>
<head>
    <title>Welcome to Git!</title>
    <script src="src/script.js"></script>
</head>
<body>
    <p>Welcome to Git!</p>
    <script>
        hello();
    </script>
</body>
</html>
```

```
...Unfinished changes...
```

то есть теперь у нас в индексации файл с первым изменением, а файл в рабочем изменение содержит еще второе непроиндексированное изменение

git add и git commit одной командой

```
~/project master +2> git commit -am 'Rename hello to helloGitty'  
[master de089dc] Rename hello to helloGitty  
 2 files changed, 2 insertions(+), 2 deletions(-)
```

здесь а - сразу add и commit, м - сообщение

НО!!! флаг -а работает только для тех файлов которые уже есть в индексе, то есть отслеживаемые гитом (если создать новый файл, то флаг -а не сработает)

если нужно не все файлы (опять же только для проиндексированных):

```
~/project master •1+2> git status  
On branch master  
Changes to be committed:  
  (use "git reset HEAD <file>..." to unstage)  
  
    modified:   index.html  
  
Changes not staged for commit:  
  (use "git add <file>..." to update what will be committed)  
  (use "git checkout -- <file>..." to discard changes in working directory)  
  
    modified:   .gitignore  
    modified:   src/script.js  
  
~/project master •1+2> git commit -m 'Ignore log files' .gitignore  
[master 3b588e7] Ignore log files  
 1 file changed, 1 insertion(+)
```

Если все-таки по какой-то причине нужно так сделать даже с непроиндексированными файлами, то:

```
x ~/project master •1+1...1> git config --global alias.commitall '!git add .;git commit'
```

но в данном случае это сработает только с файлами текущей директории, если хотим со всеми файлами, начиная с корня проекта, то:

```
~/project master •1+1...1> git config --global alias.commitall '!git add -A;git commit'
```

.gitignore

в корне проекта можно создать файл .gitignore, чтобы давать туда файлы/директории, которые даже при add . не будут индексироваться. но это локальный игнор.

```
~/project master •2...1> ls -a
.          ..          .git          .gitignore          .idea          index.html          src
```

можно создать глобальный игнор.

```
git config --global core.excludesfile ~/.gitignore_global
```

я это сделала в файле .gitignore_global и указал этот файл в config

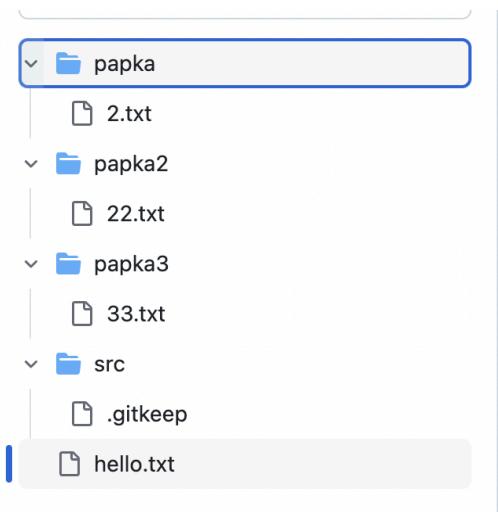
```
UW PICO 5.09                                     File: .gitignore_global

# OS generated files #
#####
.DS_Store
.DS_Store?
```

```
[user]
    name = Sasha Chernyakova
    email = alexandra13.04.2004@gmail.com
    clean = git-lfs clean -- %f
    smudge = git-lfs smudge -- %f
    process = git-lfs filter-process
    required = true
[color]
    ui = auto
[core]
    excludesfile = /Users/chernyakova.run/.gitignore_global
```

Доказательство:

```
chernyakova.run@MacBook-Air-Sasha-2 testRepo % ls -a
.
..          .DS_Store      hello.txt      papka2      src
..          .git           papka         papka3
chernyakova.run@MacBook-Air-Sasha-2 testRepo % git status
On branch first_my_branch
nothing to commit, working tree clean
```



git add -f <путь к файлу/директории которая в игноре> -
для того чтобы проиндексировать то что в игноре

УДАЛЕНИЕ ФАЙЛОВ

При удалении файла, мы также должны проиндексировать это изменение и закоммитить потом. Тогда в репозитории текущий проект не будет показывать данный файл, однако в репозитории он останется, так как присутствовал на других коммитах

Одна команда для удаления и индексации:

```
~/project master> git rm -r src = rm -r src + git add src
```

команда выше сработает только если нет никаких изменений в файле, который еще не проиндексировали. Это специально делает гит, чтобы мы случайно что-то не удалили, гит считает что если вы изменили что-то, то вряд ли захотите удалять

если есть:

- 1) сделать коммит и удалить как выше
- 2) удалить из директории рабочей и добавить в индекс даже если есть изменения после предыдущего коммита в индексе (так гит заставляет удалить чтобы случайно не удалил человек что то нужное):

```
git rm -f <файл>  
git rm -r -f <директория>
```

Также можно удалить из индекса, но оставить в рабочем каталоге (то есть на компе останется, untracked теперь):

```
git rm --cached <файл>  
git rm -r --cached <директория>
```

```
~/project master> git rm -r --cached src
rm 'src/script.js'

~/project master •1...1> git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    deleted:    src/script.js

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    src/
```

ПЕРЕИМЕНОВАНИЕ ФАЙЛОВ

при изменение имени файла гит считает, что мы удалили файл и создали новый, поэтому отображается два изменения. Но как только все добавится в индекс, файл станет отслеживаемым гитом и гит поймет что это тот же файл и выведет уже информацию о том, что файл переименован (git status)

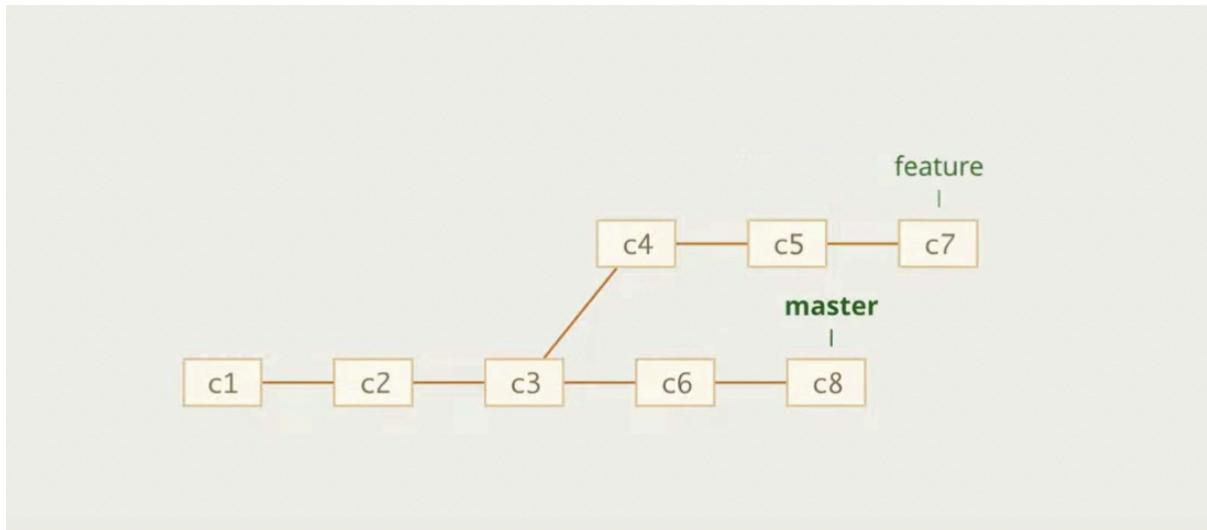
сразу переименовать и добавить в индекс:

```
~/project master> git mv index.html hello.html
```

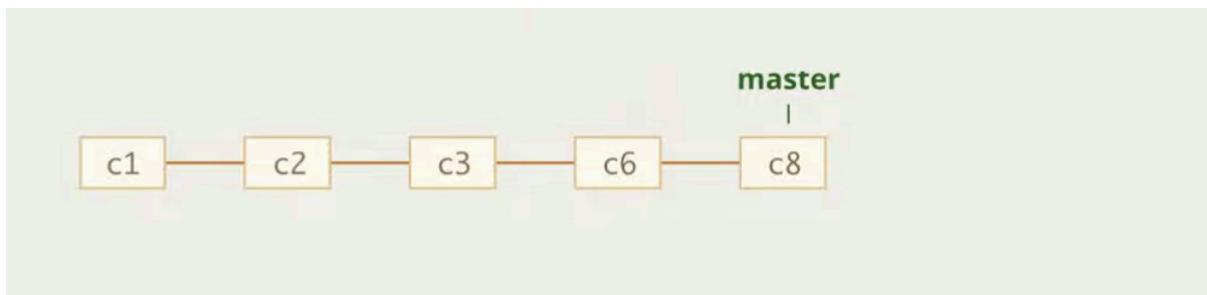
БЕТКА

Ветка - изолированный поток разработки, в котором можно делать коммиты так, что их не видно из других веток

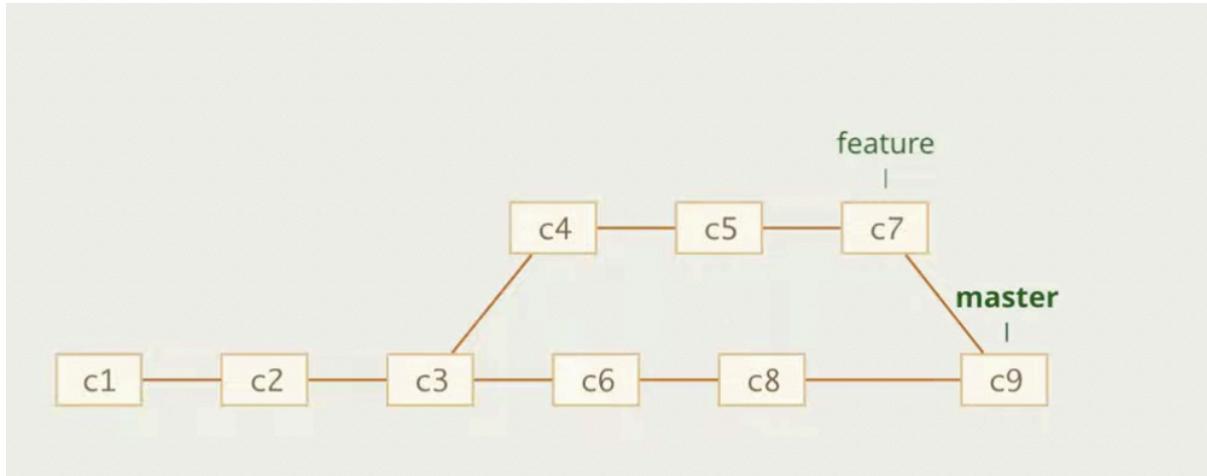
При создании проекта у нас всегда изначально есть главная ветка master, далее мы например хотим делать какую-то другую фичу и не уверены что она не помешает нашему основному стабильному проекту в main, мы создаем новую ветку, переключаемся на нее и делаем коммиты в ней. (**master - старая версия, сейчас это main**)



если нам что-то не понравилось в новой ветке feature, мы можем удалить ее без вреда всему основному стабильному проекту



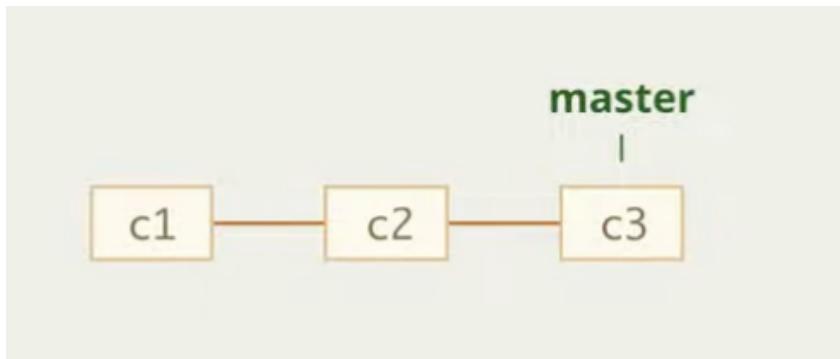
но обычно конечно ветка становится успешной и мы объединяем ее с master



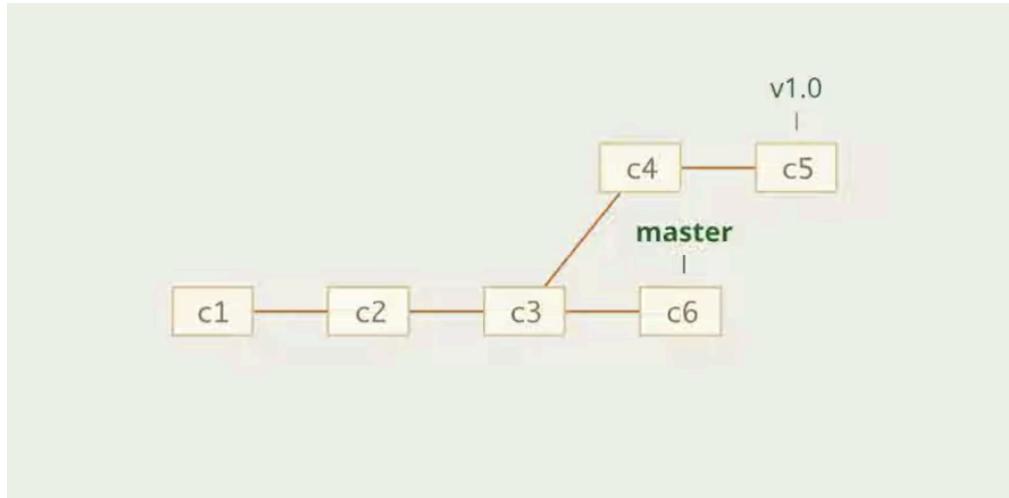
Новая функциональность = новая ветка
И интегрируем в master когда она готова

еще один распространенный подход использования веток:
поддержка одновременно нескольких версий проекта

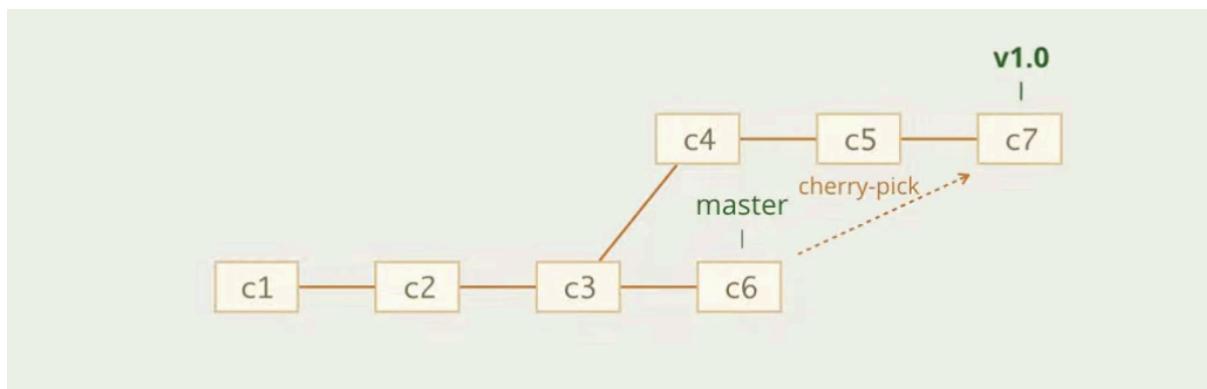
мы решаем сделать релиз:



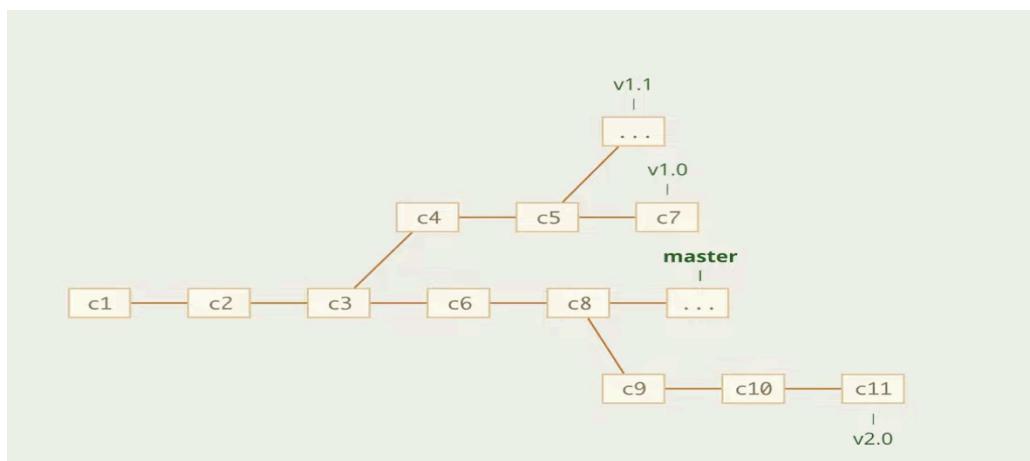
В ветке мастер мы продолжаем делать новые возможности, а в ветке v1 уже шлифуем то, что решили добавить в релиз:



Бывает такое, что ошибка существовала до разделения веток, тогда ее надо исправить и там, и там. Такая возможность есть. Тогда мы делаем изменение в одной ветке(обычно master), а потом применяем это изменение к другой ветке - **cherry-pick**. В итоге изменения применяются к двум веткам.



И т.д.



Создание ветки релиза запускает следующий цикл релиза, и с этого момента новые функции добавить больше нельзя — допускается лишь исправление багов, создание документации и решение других задач, связанных с релизом. Когда подготовка к поставке завершается, ветка `release` сливаются с `main` и ей присваивается номер версии. Кроме того, нужно выполнить ее слияние с веткой `develop`, в которой с момента создания ветки релиза могли возникнуть изменения.

РАБОТА С ВЕТКАМИ

Ветка - ссылка на коммит

`git branch` - посмотреть все ветки

`git branch -v` - посмотреть ветки с последним коммитом (она не него указывает)

после того как сделаем хотя бы 1 коммит, по умолчанию появится ветка `main`

```
chernyakova.run@MacBook-Air-Sasha-2 hello % git branch
* main
chernyakova.run@MacBook-Air-Sasha-2 hello % git branch -v
* main f7e7913 start
```

* - текущая ветка

В папке `.git` есть файл **HEAD**, для того чтобы гит репозиторий понимал, где мы находимся сейчас (хранит ссылку на текущую ветку).

```
chernyakova.run@MacBook-Air-Sasha-2 .git % cat HEAD
ref: refs/heads/main
```

`git branch -v` всегда показывает последний commit ветки, при создании новой ветки, ей присваивается последний коммит в точке ответвления:

```
chernyakova.run@MacBook-Air-Sasha-2 hello % git branch 1
chernyakova.run@MacBook-Air-Sasha-2 hello % git branch -v
  1    f7e7913 start
* main f7e7913 start
```



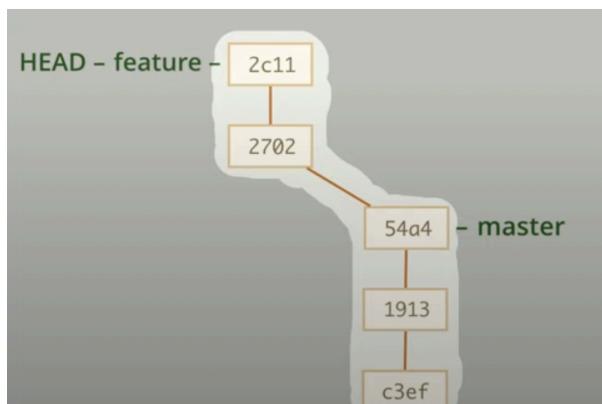
git checkout <название ветки> - переключение ветки:

```
[chernyakova.run@MacBook-Air-Sasha-2 hello % git checkout 1
Switched to branch '1'
[chernyakova.run@MacBook-Air-Sasha-2 hello % cd .git
[chernyakova.run@MacBook-Air-Sasha-2 .git % cat HEAD
ref: refs/heads/1
```

git checkout -b <название ветки> - создание и переключение ветки одной командой:

```
~/project feature> git checkout -b feature = branch + checkout
```

В общем технически ветка - ссылка на один конкретный коммит, но при этом принадлежащими ветке коммитами называют все коммиты, которые находятся на пути от вершины вниз по цепочке родителей:



до переключения на ветку master:

The screenshot shows a terminal window with a project structure on the left. The current file is index.html, which contains the following code:

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Git rules</title>
5          <script src="script.js"></script>
6      </head>
7      <body>
8          Git rules!
9
10         <script>
11             work();
12         </script>
13     </body>
14 </html>
```

Below the code, the terminal prompt is ~project feature> git checkout master

после переключения на ветку master:

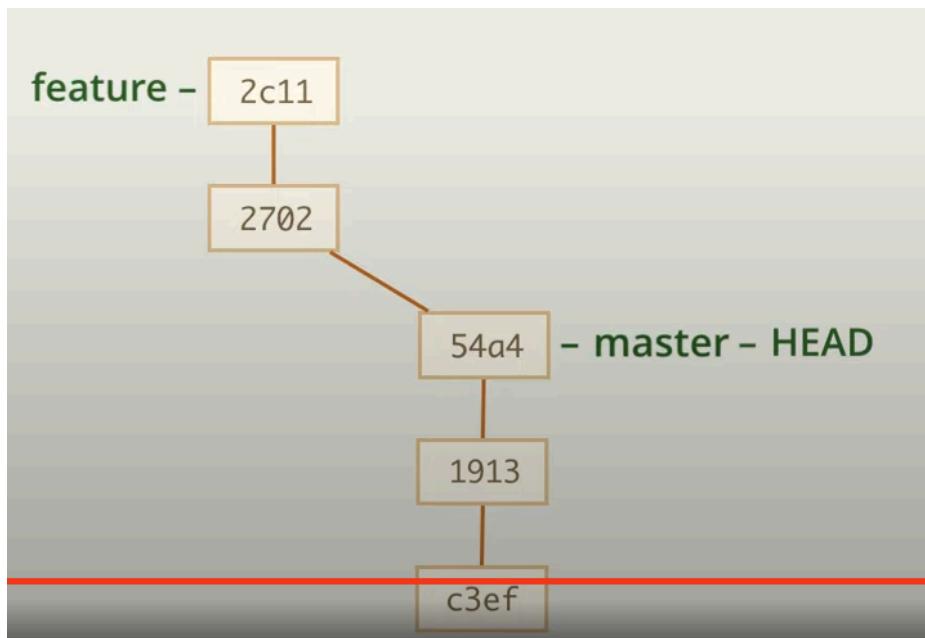
The screenshot shows a terminal window with a project structure on the left. The current file is index.html, which now contains:

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Git rules</title>
5          <script src="script.js"></script>
6      </head>
7      <body>
8          Git rules!
9
10         Let's have some fun with git.
11     </body>
12 </html>
```

Below the code, the terminal prompt is ~project feature> git checkout master
Switched to branch 'master'

при переключении на ветку main наш файл вернулся в то состояние в котором он был в своей последней точке

Также поменяется HEAD:



GIT STASH

Бывает ситуация, что мы работаем на ветке, но нам срочно нужно переключиться на другую, но на этой закоммитить мы не можем, так как еще не готово.

Тогда при команде `checkout` нам гит покажет предупреждение:

```

chernyakova.run@MacBook-Air-Sasha-2 hello % git checkout main
Switched to branch 'main'
chernyakova.run@MacBook-Air-Sasha-2 hello % git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   hello.txt

no changes added to commit (use "git add" and/or "git commit -a")
chernyakova.run@MacBook-Air-Sasha-2 hello % git checkout 1
error: Your local changes to the following files would be overwritten by checkou
t:
    hello.txt
Please commit your changes or stash them before you switch branches.
Aborting

```

если мы все равно хотим переключиться `git checkout -f <название ветки>`

```

chernyakova.run@MacBook-Air-Sasha-2 hello % git checkout -f 1
Switched to branch '1'

```

но нужно понимать, что когда мы вернемся назад на нашу ветку, изменений незакоммиченных не будет!!!!

флаг **-f** часто используют для удаления всех **незакоммиченных действий**:

```
~/project feature> git checkout -f HEAD
```

то есть мы не переключимся на другую ветку, а останемся на текущей, но вернем все к состоянию до изменений

НО!!!! Важно понимать, что очищается только то, что существует в индексе. Неотслеживаемые файлы не очищаются. С точки зрения гит это логично, так как он лезет туда куда не должен (он ведь не отслеживает).

Для этого существует отдельная команда:

git clean -f - удаляет неотслеживаемые файлы и директории

флаги:

-d - чтобы удалялись не только файлы но и директории

-x - чтобы удаляло файлы, которые игнорируются гитом через гитигнор

-f - чтобы работало

То есть для полной очистки репо (чтобы получилось как на последнем коммите) нужно использовать две команды: для удаления отслеживаемых и для удаления неотслеживаемых. (для удаления отслеживаемых команд есть много возможностей : checkout, reset и тд)

Но все-таки зачастую при переключение резко на другую ветку нам нужно сохранить наши незакоммиченные изменения - **git stash**:

наш измененный файл без коммита

```
Project
  project
    .git
    index.html
    script.js

function sayHi() {
  alert(`Hello from Git!`);
}

function work() {
  alert(`Work, work!`);
  // ...work in progress...
}

function sayBye() {
  alert(`Goodbye from Git!`);
}
```

```
~/project feature +1> g
```

после git stash сохранили изменения, но вернули файл в состояние прошлого коммита

```
Project index.html script.js
v project
> .git
index.html
script.js

1 function sayHi() {
2     alert(`Hello from Git!`);
3 }
4
5 function work() {
6     alert(`Work, work!`);
7 }
8
9 function sayBye() {
10    alert(`Goodbye from Git!`);
11 }
12

~/project feature +1> git stash
Saved working directory and index state WIP on feature: 2702040
Create work

~/project feature •1> g
```

git status показывает, что изменений нет, переключаемся на другую ветку

```
Project index.html script.js
v project
> .git
index.html
script.js

1 function sayHi() {
2     alert(`Hello from Git!`);
3 }
4
5 function work() {
6     alert(`Work, work!`);
7 }
8
9 function sayBye() {
10    alert(`Goodbye from Git!`);
11 }
12

~/project feature +1> git stash
Saved working directory and index state WIP on feature: 2702040
Create work

~/project feature •1> git status
On branch feature
nothing to commit, working tree clean

~/project feature •1> git checkout master
Switched to branch 'master'

~/project master •1> git checkout feature
Switched to branch 'feature'
```

Затем возвращаемся на нашу, где изменений нет, но после написания **git stash pop**, изменения появляются и git status уже не пустой.

```
project
  .git
  index.html
  script.js

1 function sayHi() {
2   alert(`Hello from Git!`);
3 }
4
5 function work() {
6   alert(`Work, work!`);
7   // ...work in progress...
8 }
9
10 function sayBye() {
11   alert(`Goodbye from Git!`);
12 }
13

Switched to branch 'master'

~/project master •1> git checkout feature
Switched to branch 'feature'

~/project feature •1> git stash pop
On branch feature
Changes not staged for commit:
  modified:  script.js

no changes added to commit
Dropped refs/stash@{0} (e6e64ffcbf5bfed318b389c9070c203e8bd9b28
b)

~/project feature +1>
```

Важная особенность **git stash** - изменение на самом деле не привязывается ни к какой-то ветке или коммиту, то есть мы можем сохранить изменения на одной ветке, а применить на другой. Но так лучше не делать, так как если файл итак изменен, а мы наверх накладываем изменения, то у гита могут возникнуть сложности - конфликт.

ВАЖНО!

Выше мы проводили изменения с файлами script.js, которые разные в двух ветках, поэтому нам приходилось использовать stash. Однако если файлы одинаковые, то при переключении незакоммиченные изменения останутся (так как для гита файлы не менялись и он не будет их менять):

```
~/project feature> git status
On branch feature
Changes not staged for commit:
  modified:  index.html

no changes added to commit

~/project feature +1> git checkout master
M      index.html
Switched to branch 'master'
```

здесь видно M - незакоммиченные изменения сохранились
git status покажет эти изменения

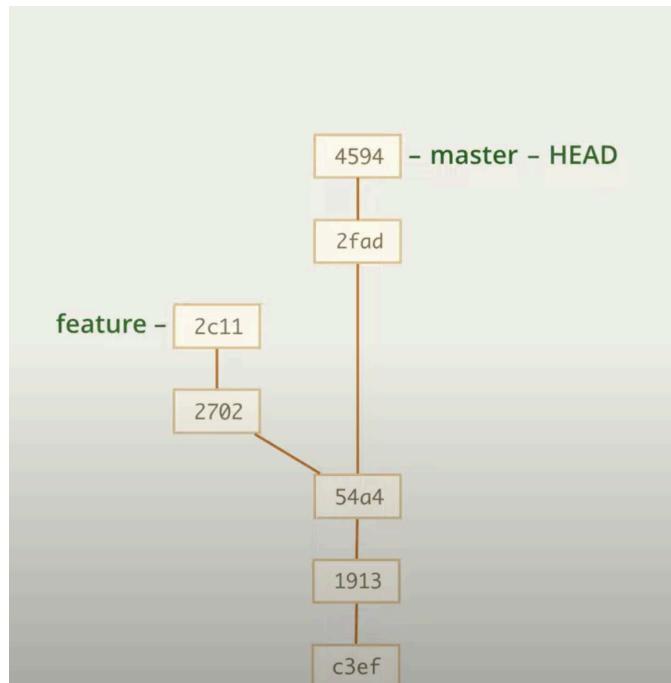
Это важно так как index.html изменится и мы можем случайно закоммитить его на другой ветке

checkout перезаписывает файлы, которые между ветками разные, поэтому ругается на изменения в них, а остальные не трогает

второй способ возможен если файлы одинаковые, то есть

- 1)обычно опасно: файлы которые я хочу перенести должны быть закоммичены в одинаковом варианте на обеих ветках
- 2)обычно полезно: сделаны изменения, а потом создана новая ветка и уже на ней закоммитить эти изменения, а предыдущая ветка останется без изменений и с чистым git status; обычно такое надо если начали делать в main, поняли что делать много и не факт что все заработает, переключились на другую ветку и там сделали коммиты и попробовали.

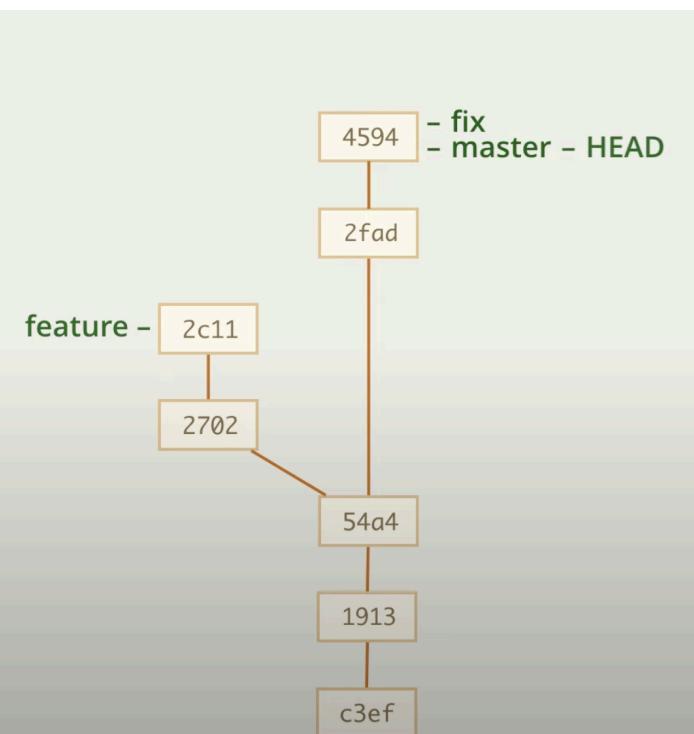
Бывает, что случайно сразу закоммитили в ветку main, хотя нужно было сначала в другую (и это тоже можно решить):



для решений нужно:

- 1) создать новую ветку на текущем коммите, куда хотели бы перенести:

```
~/project master> git branch fix
```



- 2) откатить main (или ту ветку на которой лишние коммиты, просто в данном примере main) до нужного коммита:

```
~/project master> git branch master 54a4
fatal: A branch named 'master' already exists.
```

```
x ~/project master> git branch -f master 54a4
fatal: Cannot force update the current branch.
```

```
x ~/project master> git checkout fix
Switched to branch 'fix'
```

```
~/project fix> git branch -f master 54a4
```

git branch <ветка для отката> <коммит на который откатиться> -

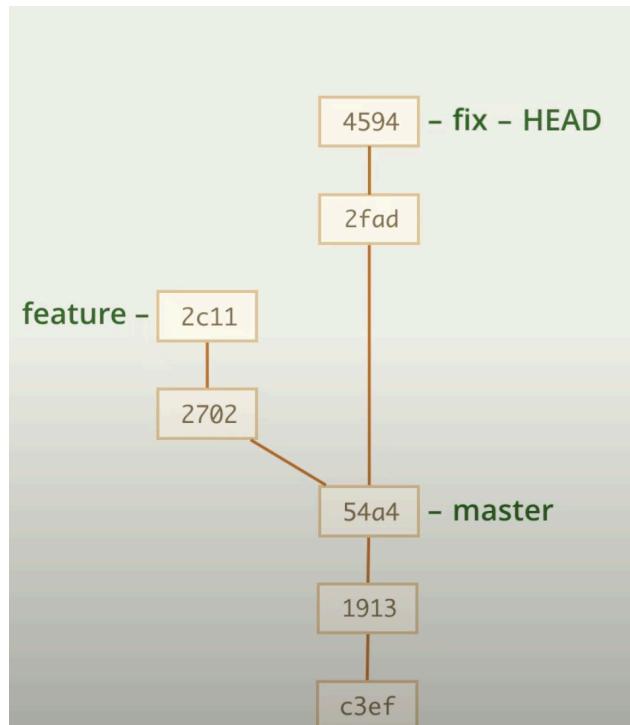
если ветка для отката не существует

git branch -f <ветка для отката> <коммит на который откатиться> -

если ветки не существует, то создает, и откатит на этот коммит (вообще для существующих веток)

На скрине ошибка, так как для отката нужно уйти с ветки, которую откатываем

Теперь:



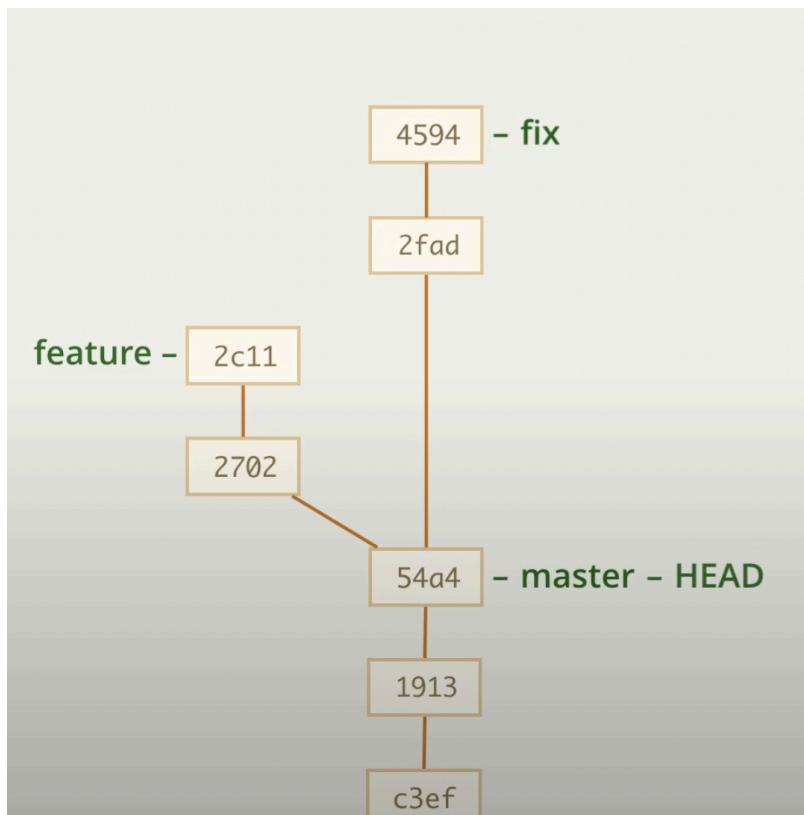
Можно вернуться назад, если передумали:

```
~/project fix> git branch -f master fix
```

указать ветку вместо идентификатора не является ошибкой, так как ветка указывает всегда на последний коммит

Сместить ветку и сразу переключиться на нее:

```
~/project fix> git checkout -B master 54a4
Switched to and reset branch 'master'
```



Переключиться можно не только на ветку, но и на коммит:

```
~/project fix> git checkout 1913
```

При этом возникает “состояние отделенной HEAD”

```
~/project fix> git checkout 1913
Note: checking out '1913'.
```

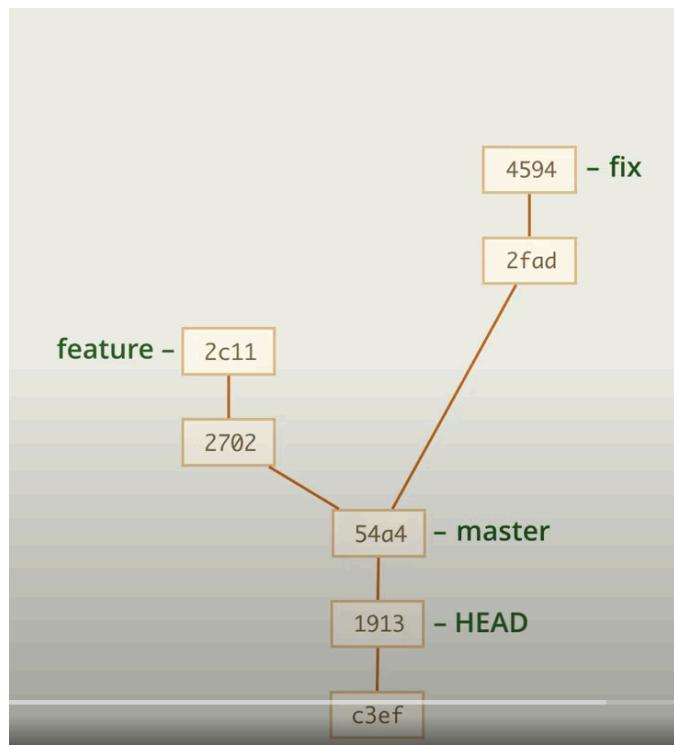
You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using -b with the checkout command again. Example:

```
git checkout -b <new-branch-name>
```

```
HEAD is now at 1913975 Create sayHi
```

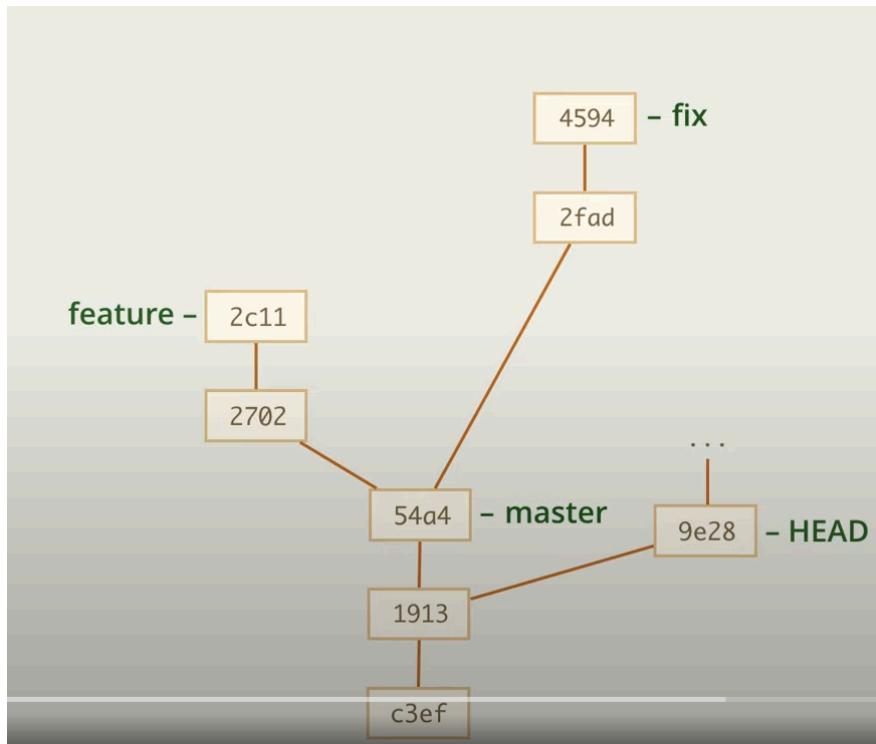
Гит как обычно передвигает HEAD и обновляет файлы проекта на момент этого коммита



НО!!!! мы перешли в особое состояние detached HEAD - HEAD указывает на коммит, а не на ветку как обычно

```
~/project :1913975> cat .git/HEAD
19139752981c7ce057bc734b708a5334dd258069
```

Далее если мы будем делать коммиты, то они будут отходить друг от друга:



особенность этих коммитов в том, что они не участвуют ни в одной ветви разработки. То есть когда мы переключимся на другую ветку, то эта цепочка коммитов будет висеть и не принадлежать ни одной ветке. Чтобы вернуться к этой цепочке, нужно знать идентификатор коммита, который легко забыть. Гит такие недостижимые коммиты со временем сам удаляет.

Поэтому при попытке переключиться на ветку, гит предупредит, что нужно эти изменения перенести на новую ветку

```
~/project :9e28e82> git checkout fix
Warning: you are leaving 1 commit behind, not connected to
any of your branches:
```

```
9e28e82 Detached HEAD demo
```

```
If you want to keep it by creating a new branch, this may be a good time
to do so with:
```

```
git branch <new-branch-name> 9e28e82
```

```
Switched to branch 'fix'
```

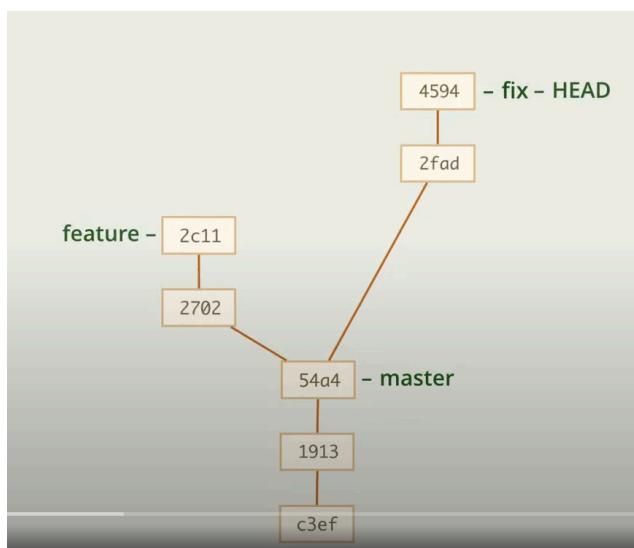
Но необязательно переносить это на новую ветку, можно на текущую с помощью команды **git cherry-pick**

```
project fix> git cherry-pick 9e28
```

Обычно, конечно, переключаются на ветку, однако переключиться на коммит может быть полезным, чтобы просмотреть файлы которые были на тот момент и протестировать

ВОССТАНОВЛЕНИЕ ПРЕДЫДУЩИХ ВЕРСИЙ ФАЙЛА

Например в ходе разработки мы понимаем, что допустили ошибку в файле и хотим вернуть его версию, которая была несколько коммитов назад.



Например хотим вернуть состояние файла index.html в коммите 4594 как в коммите 54a4 (то есть именно один файл откатить, а не в общем к коммиту). Восстановленные файлы checkout автоматически добавляется в индекс

```
~/project fix> git checkout 54a4 index.html

~/project fix •1> git status
On branch fix
Changes to be committed:
  modified:  index.html
```

Здесь index.html - путь к файлу

Также удобно использовать когда мы поэкспериментировали с файлом, а потом хотим откатить его к текущему коммиту (но именно файл, а не все файлы коммита):

```
~/project fix +1> git checkout HEAD index.html
```

Важно знать!!! Название директории в проекте может совпадать с названием ветки (на самом деле такое редко):

```
git checkout master
```

тогда появляется вопрос, как же checkout поймет что это. Он действует по такому принципу:

- 1) checkout сначала пытается найти ссылку или коммит с такими идентификатором
- 2) если не нашел, то считает это путем

то есть в нашем примере он будет считать это веткой master, а не директорией в проекте

если мы хотим получить именно путь, то нужно поставить двойной дефис перед путем

```
~/project fix> git checkout HEAD -- master
```

М:

```
4 ~/project fix> git checkout -- master
```

ПРОСМОТР КОММИТОВ

git log - посмотреть все коммиты

```
commit 2c11f12eb82247fd3fecc90c1c87d0996d0603d57 (HEAD -> feature)
демонстрация истории и старых файлов, символы ~, ^, @, поиск с ./
Author: Ilya Kantor <iliakan@gmail.com>
Date:   Tue Sep 12 22:49:07 2017 +0200

    Run work

commit 270204032166018243eb9caee83c1cbf7416aa49
Author: Ilya Kantor <iliakan@gmail.com>
Date:   Tue Sep 12 22:45:43 2017 +0200

    Create work

commit 54a4be6ff4ca63c909328ce4894c9ab0bc632c43 (master)
Author: Ilya Kantor <iliakan@gmail.com>
Date:   Tue Sep 12 15:38:14 2017 +0200

    Create sayBye

commit 19139752981c7ce057bc734b708a5334dd258069
Author: Ilya Kantor <iliakan@gmail.com>
Date:   Tue Sep 12 15:36:37 2017 +0200

    Create sayHi

commit c3ef9b94833abfc8845de1492ed81aae8985b75c
Author: Ilya Kantor <iliakan@gmail.com>
Date:   Tue Sep 12 15:32:30 2017 +0200

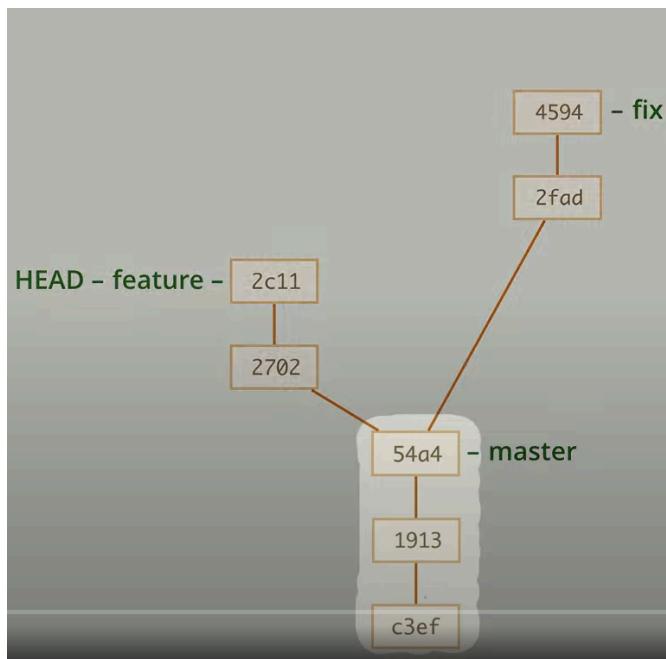
    Initial commit
```

git log --oneline - посмотреть все коммиты компактно

```
~/project feature> git log --oneline
2c11f12 (HEAD -> feature) Run work
2702040 Create work
54a4be6 (master) Create sayBye
1913975 Create sayHi
c3ef9b9 Initial commit
```

По умолчанию git log выводит коммиты, начиная от HEAD, но можно передать и любую ветку (тогда только ее коммиты выводятся)

```
~/project feature> git log master --oneline  
54a4be6 (master) Create sayBye  
1913975 Create sayHi  
c3ef9b9 Initial commit
```



git show <идентификатор коммита/ссылка(ветка)>- посмотреть подробно коммит (по умолчанию коммит из HEAD)

В жизни обычно интересуют коммиты 1-2 шага назад по текущей ветке. Чтобы не искать в логах, есть удобный синтаксис в гите:

~ - родитель коммита

~~ - родитель родителя коммита

~~~ (~3 - краткая запись)

И тд

@ - краткое обозначение HEAD

```
git show @~
```

```
просмотр истории старых файлов, символы ~, ~~, @, поиск с ./
~/project feature> git show HEAD~
commit 270204032166018243eb9caee83c1cbf7416aa49
Author: Ilya Kantor <iliakan@gmail.com>
Date:   Tue Sep 12 22:45:43 2017 +0200
```

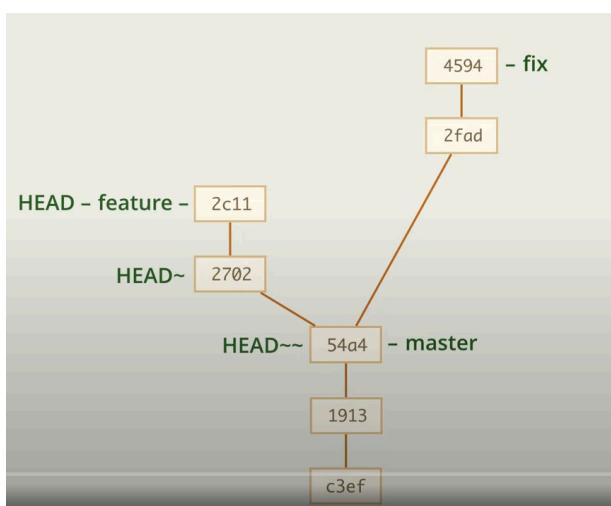
Create work

```
diff --git a/script.js b/script.js
index 935d66a..ec09669 100644
--- a/script.js
+++ b/script.js
@@ -2,6 +2,10 @@ function sayHi() {
    alert(`Hello from Git!`);
}

+function work() {
+  alert(`Work, work!`);
+
+function sayBye() {
    alert(`Goodbye from Git!`);
}
```

```
~/project feature> git show HEAD~~ --quiet
commit 54a4be6ff4ca63c909328ce4894c9ab0bc632c43 (master)
Author: Ilya Kantor <iliakan@gmail.com>
Date:   Tue Sep 12 15:38:14 2017 +0200
```

Create sayBye



не обязатель по текущей ветке смотреть, можно и по другим:

```
~$ cd /tmp  
~/tmp$ git log  
commit 19139752981c7ce057bc734b708a5334dd2580  
Author: Ilya Kantor <iliakan@gmail.com>  
Date:   Tue Sep 12 15:36:37 2017 +0200
```

Create sayHi

```
diff --git a/index.html b/index.html  
index 12f5d7d..e8e448b 100644  
--- a/index.html  
+++ b/index.html  
@@ -2,6 +2,7 @@  
 <html>  
   <head>  
     <title>Git rules</title>  
+    <script src="script.js"></script>  
   </head>  
   <body>  
     Git rules!  
diff --git a/script.js b/script.js  
new file mode 100644  
index 000000..0f67e66  
--- /dev/null  
+++ b/script.js  
@@ -0,0 +1,3 @@  
+function sayHi() {  
+  alert(`Hello from Git!`);  
+}
```

При таком выводе показываются отличия файлов

Как было рассмотрено ранее можно откатиться к версии файла:

```
'project feature> git checkout @~ index.html'
```

но тогда заменится наш текущий index.html на более старый (это не очень удобно, особенно если мы делали в нем изменения, тогда они просто пропадут)

если мы не хотим заменять файл, а хотим просто посмотреть его более старую версию, то:

```
~/project feature> git show @~:index.htm
<!DOCTYPE html>
<html>
  <head>
    <title>Git rules</title>
    <script src="script.js"></script>
  </head>
  <body>
    Git rules!

    Let's have some fun with git.
  </body>
</html>
```

Доступ к коммиту, когда не помним его идентификатор, но знаем что в описание есть конкретное слово (гит найдет самый свежий коммит который содержит это слово в описании, не обязательно на той ветке которой мы находимся):

```
~/project feature> git show :/sayBye
commit 4594f10bab02bdf034e0fab8d57b5bc09fb21594 (fix)
Author: Ilya Kantor <iliakan@gmail.com>
Date:   Thu Sep 14 08:32:33 2017 +0200
```

Run sayBye

```
diff --git a/index.html b/index.html
index fdba09e..da9736c 100644
--- a/index.html
+++ b/index.html
@@ -8,9 +8,11 @@
<script>
    sayHi();
</script>
-
+
    Git rules!

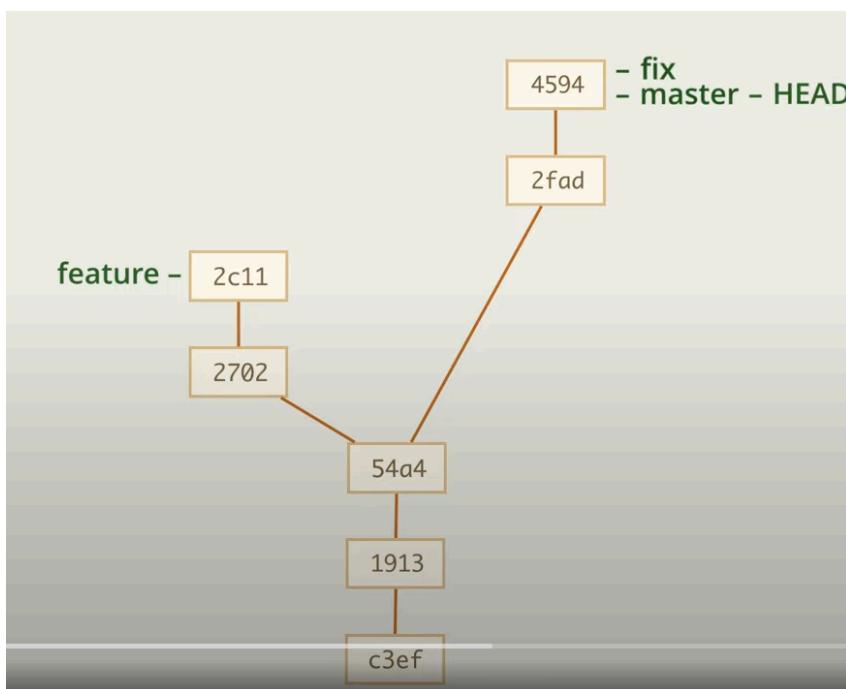
-
+    Let's have some fun with git.
+
+    <script>
+        sayBye();
+
+    </script>
</body>
</html>
```

## СЛИЯНИЕ ВЕТОК

1. Сначала переключимся на ветку, в которую будет слияние
2. укажем ветку, которую влияем

```
~/project fix> git checkout master
Switched to branch 'master'

~/project master> git merge fix
Updating 54a4be6..4594f10
Fast-forward
 index.html | 8 ++++++-+
 1 file changed, 7 insertions(+), 1 deletion(-)
```



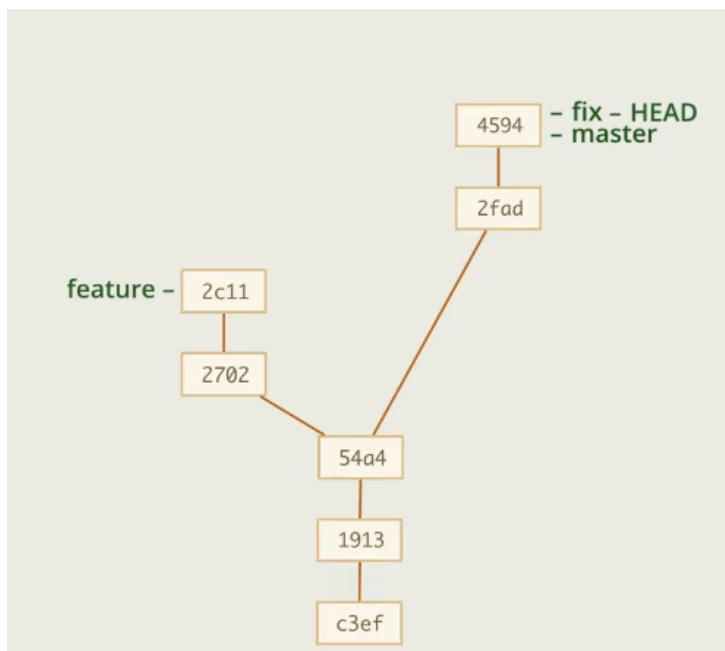
Теперь все изменения сделанные на ветке fix находятся на ветке master

Fast-forward - название алгоритма, используемого для слияния веток. Fast-forward один из простейших методов слияния - он забавный, так не изменения из ветки fix

добавляются в master, а master сдвигается так чтобы указывать на тот же коммит, что и fix

Если мы поняли, что зря произвели слияния и хотим вернуть назад, но забыли коммит, где была ветка до слияния, то и это можно решить (гит обо всем позаботился): команда merge перед переносом ветки записывает старый идентификатор в файл:

```
~/project fix> cat .git/ORIG_HEAD  
54a4be6ff4ca63c909328ce4894c9ab0bc632c43
```



Теперь можно просто перенести ветку, как было рассмотрено ранее:

```
~/project fix> git branch -f master ORIG_HEAD
```

Если переключаться на ветку, то будет ругаться. Нужно сначала уйти с этой ветки или воспользоваться командой git checkout -B (ключ создает ветку с таким названием указывающую на этот коммит и переключает на нее, если

ветка создана, то просто перемещает и переключается на нее):

```
~/project fix> git checkout master
Switched to branch 'master'

~/project master> git branch -f master fix
fatal: Cannot force update the current branch.

✖ ~/project master> git checkout -B master fix
```

В данном примере получилось, что можно было просто сдвинуть ветку, а не делать merge, но это лишь на самом простейшем примере. В реальности merge очень мощная команда и выполняет сложные слияния.

## Удаление веток

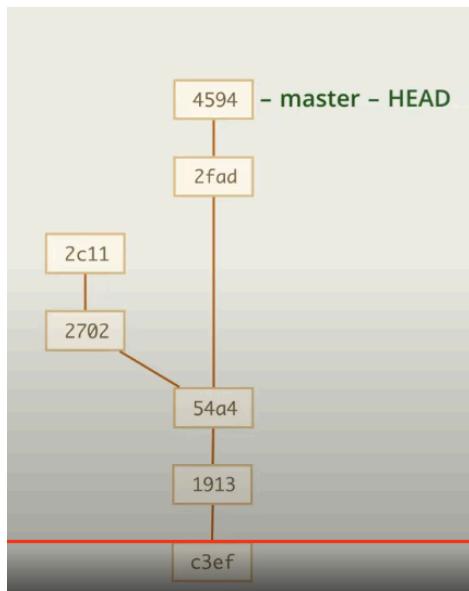
**git branch -d** - Ветка удалится только в том случае, если она была объединена с текущей (вмержена) - то есть у ветки не было отдельных коммитов и удаление прошло успешно:

```
~/project master> git branch -d fix
Deleted branch fix (was 4594f10).
```

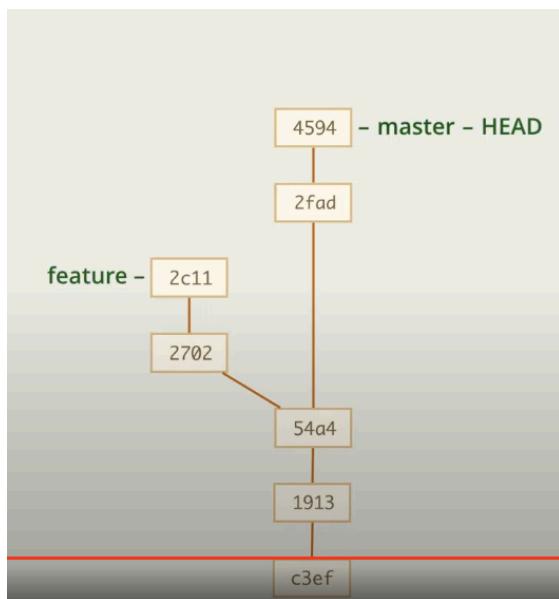
То есть если ветка не вмержена, то подразумевается, что ее удалять опасно, так как коммиты на ней будут недостижимы и со временем гит удалит их со своей базы

Если мы все-таки хотим удалить невмерженную ветку, то флаг **-D** (помнить, что удаляется только ссылка, коммиты сразу будут, позже гит их удалит):

```
x ~/project master> git branch -D feature  
Deleted branch feature (was 2c11f12).
```



**Отменить удаление ветки нельзя, но можно создать новую с таким же названием и указать ее на нужный коммит (но надо успеть пока гит не удалит недостижимые коммиты)**



Мы смогли удалить ветку и тут же восстановить , так как помнили на каком коммите она раньше была. Но в реальности редко кто помнит идентификатор коммита. Восстановить удаленную ветку без идентификатора помогут reflogs. Любое изменение ссылок записывается в файл reflogs.

Все изменения с ссылками записываются в директорию logs директории .git.

```
~/project master> cat .git/logs/HEAD
0000000000000000000000000000000000000000000000000000000000000000 c3ef9b94833abfc8845de1492ed81aae8985b75c Ilya Kantor <iliakan@gmail.com> 1505223150 +0
200    commit (initial): Initial commit
c3ef9b94833abfc8845de1492ed81aae8985b75c 19139752981c7ce057bc734b708a5334dd258069 Ilya Kantor <iliakan@gmail.com> 1505223397 +0
200    commit: Create sayHi
19139752981c7ce057bc734b708a5334dd258069 54a4be6ff4ca63c909328ce4894c9ab0bc632c43 Ilya Kantor <iliakan@gmail.com> 1505223494 +0
200    commit: Create sayBye
54a4be6ff4ca63c909328ce4894c9ab0bc632c43 54a4be6ff4ca63c909328ce4894c9ab0bc632c43 Ilya Kantor <iliakan@gmail.com> 1505223572 +0
200    checkout: moving from master to feature
54a4be6ff4ca63c909328ce4894c9ab0bc632c43 270204032166018243eb9cae83c1cbf7416aa49 Ilya Kantor <iliakan@gmail.com> 1505249143 +0
200    commit: Create work
270204032166018243eb9cae83c1cbf7416aa49 2c11f12ebe2247fd3fecc90c1c87a996d0f63d57 Ilya Kantor <iliakan@gmail.com> 1505249347 +0
200    commit: Run work
2c11f12ebe2247fd3fecc90c1c87a996d0f63d57 54a4be6ff4ca63c909328ce4894c9ab0bc632c43 Ilya Kantor <iliakan@gmail.com> 1505249722 +0
200    checkout: moving from feature to master
54a4be6ff4ca63c909328ce4894c9ab0bc632c43 54a4be6ff4ca63c909328ce4894c9ab0bc632c43 Ilya Kantor <iliakan@gmail.com> 1505292182 +0
200    checkout: moving from master to fix
54a4be6ff4ca63c909328ce4894c9ab0bc632c43 2fad3acb6d276d2d1e504a3ba7d4d75bc6d85963 Ilya Kantor <iliakan@gmail.com> 1505370447 +0
200    commit: Run sayHi
2fad3acb6d276d2d1e504a3ba7d4d75bc6d85963 4594f10bab02bdf034e0fab8d57b5bc09fb21594 Ilya Kantor <iliakan@gmail.com> 1505370753 +0
200    commit: Run sayBye
4594f10bab02bdf034e0fab8d57b5bc09fb21594 54a4be6ff4ca63c909328ce4894c9ab0bc632c43 Ilya Kantor <iliakan@gmail.com> 1505372128 +0
200    checkout: moving from fix to master
54a4be6ff4ca63c909328ce4894c9ab0bc632c43 4594f10bab02bdf034e0fab8d57b5bc09fb21594 Ilya Kantor <iliakan@gmail.com> 1505377395 +0
200    merge fix: Fast-forward
```

То есть при каждом изменении коммита, на который указывает HEAD, в тот файл добавляется новая строка. В строке показано куда указывало раньше, куда сейчас, кто, куда и зачем.

для красивого вывода используется команда **git reflog <ветка>**. (внизу самый старый коммит)

```
~/project master> git reflog
4594f10 (HEAD -> master) HEAD@{0}: merge fix: Fast-forward
54a4be6 HEAD@{1}: checkout: moving from fix to master
4594f10 (HEAD -> master) HEAD@{2}: commit: Run sayBye
2fad3ac HEAD@{3}: commit: Run sayHi
54a4be6 HEAD@{4}: checkout: moving from master to fix
54a4be6 HEAD@{5}: checkout: moving from feature to master
2c11f12 (feature) HEAD@{6}: commit: Run work
2702040 HEAD@{7}: commit: Create work
54a4be6 HEAD@{8}: checkout: moving from master to feature
54a4be6 HEAD@{9}: commit: Create sayBye
1913975 HEAD@{10}: commit: Create sayHi
c3ef9b9 HEAD@{11}: commit (initial): Initial commit
```

если ветка не указана, то подразумевается HEAD

создание ветки не отображается в логах, но когда мы на нее переходим, то меняется HEAD -> это отображается в логах

зачем нужен reflog?

1) к примеру если мы удалили ветку feature неделю назад и уже не помним какие идентификаторы там были, но нам понадобились коммиты из этой ветки, то обычная команда git log их конечно не выведет, так как они не достижимы. Но HEAD ведь когда-то стоял на этих коммитах, то по reflog мы можем их найти

для обращения к записям из рефлога можно использовать фигурные скобки:

```
+594f10 (HEAD -> master) HEAD@{0}: merge fix. fast-forward
54a4be6 HEAD@{1}: checkout: moving from fix to master
4594f10 (HEAD -> master) HEAD@{2}: commit: Run sayBye
2fad3ac HEAD@{3}: commit: Run sayHi
54a4be6 HEAD@{4}: checkout: moving from master to fix
54a4be6 HEAD@{5}: checkout: moving from feature to master
2c11f12 HEAD@{6}: commit: Run work
2702040 HEAD@{7}: commit: Create work
54a4be6 HEAD@{8}: checkout: moving from master to feature
54a4be6 HEAD@{9}: commit: Create sayBye
1913975 HEAD@{10}: commit: Create sayHi
c3ef9b9 HEAD@{11}: commit (initial): Initial commit

~/project master> git branch feature HEAD@{6}

~/project master>
```

После этого ветка feature пересоздана

У рефлога есть много ключей: дата и тд. Можно например вывести рефлоги с датой и потом восстановить ветку передав в фигурные скобки не порядковый номер, а дату. Здесь важно понимать, что рефлоге ищется HEAD с такой датой, а если такой даты нет, то с более старой максимально

приближенной. Чтобы это работало, коммит должен быть именно запись в рефлоге, а не просто в гите.

По умолчанию в рефлоге запись хранится 90 дней, а недостижимые коммиты только 30 дней. Данные параметры можно поменять, но обычно так не делают

Если делиться своими ветками (отправлять на сервер или коллеге) рефлоги будут только локально у меня.

2. еще один способ обратиться к рефлогу в совокупности с get checkout:

```
~/project master> git checkout @{-1}
```

-1 - предыдущая ветка, с которой был checkout на данную, то есть мы возвращаемся на предыдущую ветку, с которой мы ушли

В нашей ситуации это не сработало, так как мы были на ветке fix, которую удалили (принцип поиска гита-он смотрит по reflog)

```
~/project master> git checkout @{-1}
error: pathspec '@{-1}' did not match any file(s) known to git.

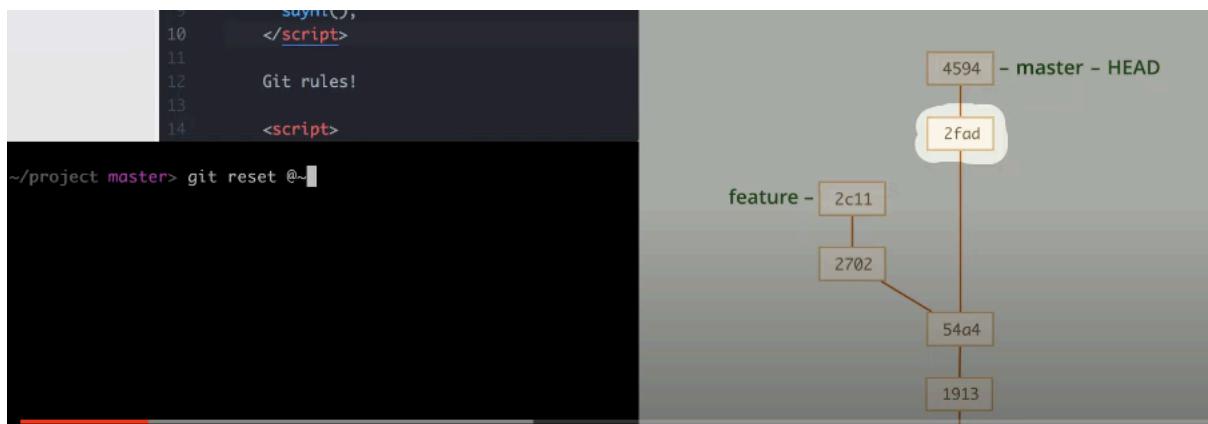
✖ ~/project master>
```

В фигурных скобках можно указать и меньшее число {-3} - на три шага назад checkout

# RESET

Бывает, что мы понимаем об ошибке, когда это уже попало в репозиторий. И хотелось бы отменить последний коммит или даже несколько.

## git reset <коммит на который хотим вернуться>



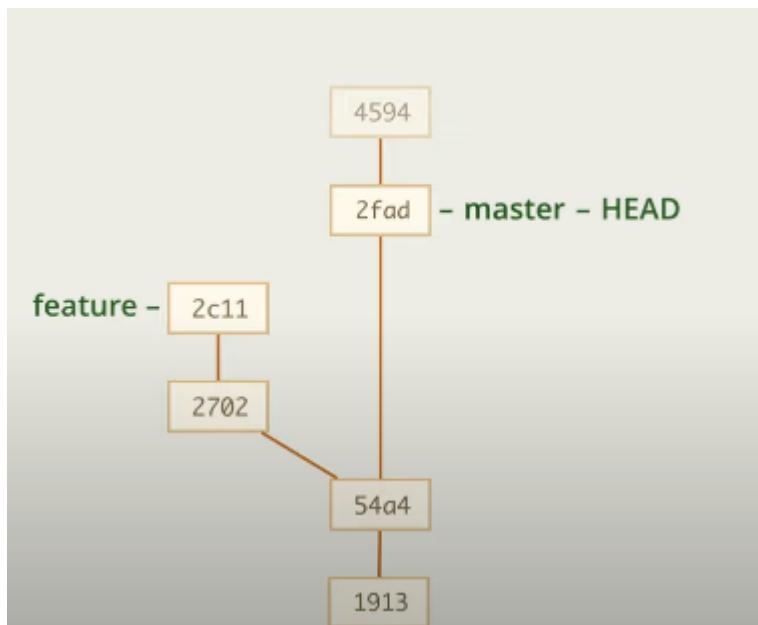
В данном примере не конкретный коммит указали, а что хотим на один коммит назад от текущего

Флаги:

**--hard** - перемещаемся на текущий коммит и обновляет директорию вместе с индексом как было в состоянии коммита, на который передвинулись

```
~/project master> git reset --hard 2fad
HEAD is now at 2fad3ac Run sayHi

~/project master> git status
On branch master
nothing to commit, working tree clean
```

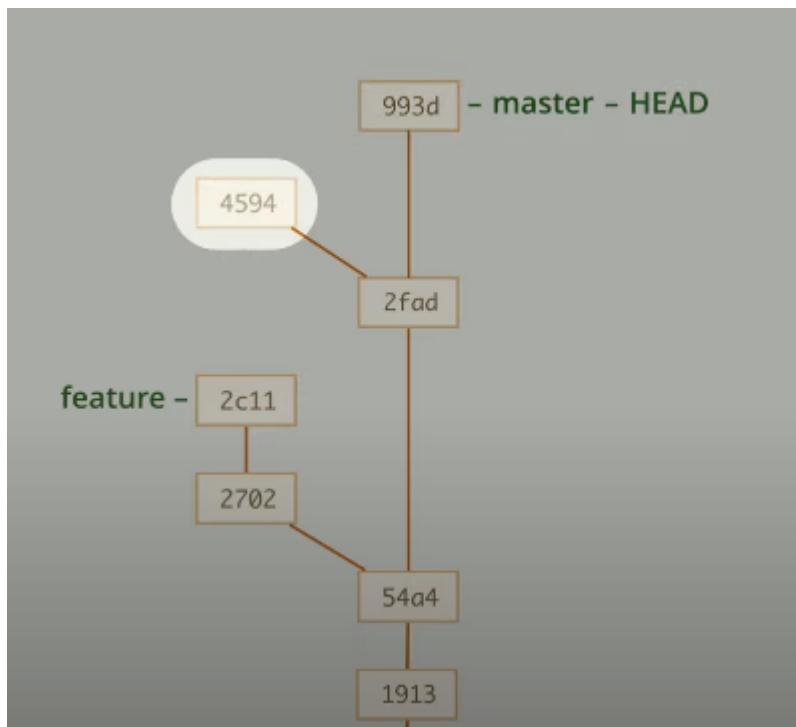


Теперь если сделать коммит, то коммиты выше как бы потеряются:

```

~/project master> git commit -am 'New stuff on master'
[master 993d1eb] New stuff on master
 1 file changed, 2 insertions(+), 2 deletions(-)

```



этот недостижимый коммит со временем гит удалить (вроде 30 дней). Таким образом мы отменили коммит последний

Можно все отменить:

git записывает предыдущее состояние HEAD в файл .git/ORIG\_HEAD:

The diagram illustrates the Git reflog and commit history. On the left, a terminal window shows the command `git reset --hard ORIG_HEAD` being run, which moves the HEAD pointer back to the commit 4594f10. On the right, a commit graph shows the following sequence of commits:

```
graph TD; 993d[993d] --> 4594[4594 - master - HEAD]; 993d --> 2fad[2fad]; 4594 --> 2fad; 2c11[2c11 - feature] --> 2702[2702]; 2702 --> 54a4[54a4]; 54a4 --> 1913[1913]
```

The commit 4594 is labeled as the master branch's HEAD. The commit 993d is labeled as ORIG\_HEAD. The commit 2c11 is labeled as feature.

Можно по стандарту узнать идентификатор через [git reflog](#)

ORIG\_HEAD это второстепенная ссылка, git ее сохраняет на всякий случай.

Если в файле были незакоммиченные изменения, то при жестком reset они пропадут.

Очистить незакоммиченные изменения из индекса и рабочей директории удобно делать с помощью reset:

```
~/project master *1> git reset --hard
```

если коммит не указан, то подразумевается HEAD

**--soft** - перенос текущей ветки на указанный коммит, но рабочую директорию и индекс не трогает.

```

3      sayHi(),
4      </script>
5
6      Git rules!
7
8      <script>
9
10     sayHi();
11
12
13
14
~/project master> git reset --soft @~

~/project master •1> git branch -v
  feature 2c11f12 Run work
* master 2fad3ac Run sayHi

~/project master •1> git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   index.html

```

так как мягкий ресет не трогает рабочую директорию и индекс, то в них остались версии файлов от коммита, с которого мы ушли

Мягкий ресет отличается от жесткого тем, что мы находясь на предыдущем состоянии уже сделали изменения и добавили в индекс, но не закоммитили. То есть все подготовленные для коммита данные сохранены.

Возьмем для примера ветку:

– А – В – С (master)

HEAD указывает на С и индекс совпадает с С.

После выполнения

```
git reset --soft B
```

HEAD будет указывать на В и изменения из коммита С будут в индексе, как будто вы их добавили командой `git add`. Если вы сейчас выполните `git commit` вы получите коммит полностью идентичный С.

**ВАЖНО: ИЗМЕНИТЬ КОММИТ НЕЛЬЗЯ, НО МОЖНО  
ОТКАТИТЬСЯ НАЗАД И ЗАМЕНИТЬ ЕГО НА ДРУГОЙ**

вернулись назад, индекс чист

```
~/project master •1> git reset --soft ORIG_HEAD  
  
~/project master> git status  
On branch master  
nothing to commit, working tree clean
```

Мягкий ресет удобно использовать чтобы немного подрапить/что-то добавить в старый коммит (так как описание повторно писать от руки не хочется, то можно использовать флаг: -с - повторить описание и данные об авторе прошлого коммита и даст возможность отредактировать его, -С - полное повторение описания прошлого коммита):

```
~/project master •2> git commit -C ORIG_HEAD  
[master 4ed8335] Run sayBye  
Date: Thu Sep 14 08:32:33 2017 +0200  
2 files changed, 5 insertions(+), 2 deletions(-)
```

Старый коммит со временем будет удален как недостижимый

То есть автор и дата автора будут одинаковы, но дата коммита будет нормальной:

```
commit 4594f10bab02bdf034e0fab8d57b5bc09fb21594  
Author: Ilya Kantor <iliakan@gmail.com>  
Date: Thu Sep 14 08:32:33 2017 +0200  
  
Run sayBye  
  
~/project master> git show --quiet  
commit 4ed833512477719439e1de863cd309266db0145c (HEAD -> master)  
)  
Author: Ilya Kantor <iliakan@gmail.com>  
Date: Thu Sep 14 08:32:33 2017 +0200  
  
Run sayBye
```

```
~/project master> git show --quiet --pretty=fuller
commit 4ed833512477719439e1de863cd309266db0145c (HEAD -> master)
)
Author: Ilya Kantor <iliakan@gmail.com>
AuthorDate: Thu Sep 14 08:32:33 2017 +0200
Commit: Ilya Kantor <iliakan@gmail.com>
CommitDate: Thu Dec 21 16:00:24 2017 +0300

Run sayBye
```

Можно указать автора, а не копировать, если принципиально (тогда только описание возьмет)

```
~/project master> git commit -C ORIG_HEAD --reset-author
```

### ИСПРАВЛЕНИЕ ПОСЛЕДНЕГО КОММИТА ПРИ ПОМОЩИ ФЛАГ amend

- 1) изменяем файл
- 2) добавляем в индекс
- 3) для замены коммита "git commit --amend"
- 4) открывается редактор для описания коммита (предварительно там написано как в последнем коммите)

с точки зрения гита сразу произошел мягкий ресет, а потом сделался новый коммит (как делали руками раньше)

также можно использовать флаг для замены автора

```
git commit --amend --no-edit
```

флаг no edit - отменяет вызов редактора

также --amend удобно использовать для замены описания коммита (например сделали ошибку)

**мягкий коммит дает возможность откатиться на много коммитов, amend только на один**

например МЯГКИМ КОММИТОМ можно объединить несколько коммитов один:

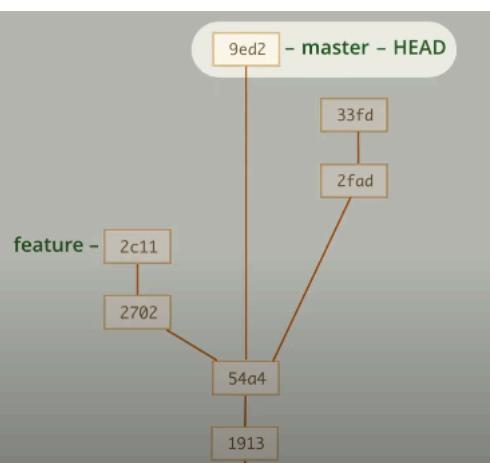
```
7      <body>
8          <script>
9              sayHi();
10             </script>
11
12     Git rules at replacing commits!
13
14     <script>
```

```
~/project master> git reset --soft @~2
```

```
~/project master •1> git branch -v
  feature 2c11f12 Run work
* master 54a4be6 Create sayBye
```

```
~/project master •1> git commit -m 'sayHi and sayBye'
[master 9ed2e71] sayHi and sayBye
 1 file changed, 8 insertions(+), 2 deletions(-)
```

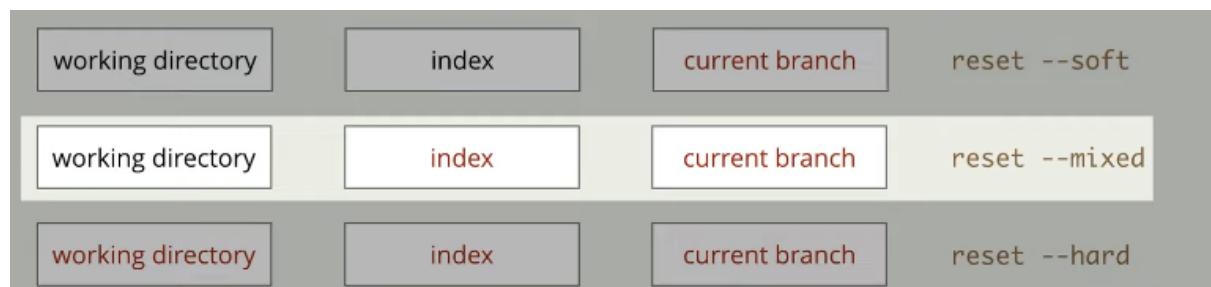
```
~/project master> █
```



теперь коммит 9ed2 это 33fd + 2fad, а те два коммита через 30 дней удаляются как недостижимые

Важно: изменения можно вносить только в последние коммиты ветки, коммит между другими коммитами исправить нельзя не меняя коммиты, которые идут за ним. Есть команда для переписывания git rebase.

флаг --mixed (смешанный reset) - он перемещает ветку и обновляет индекс на новое состояние, но при этом не трогает рабочую директорию. (можно без флага, так как это по умолчанию)



то есть отличие от мягкого ресета в том, что изменения остались в рабочей директории, но не проиндексированы

```

~/project master> git reset @~
Unstaged changes after reset:
M      index.html

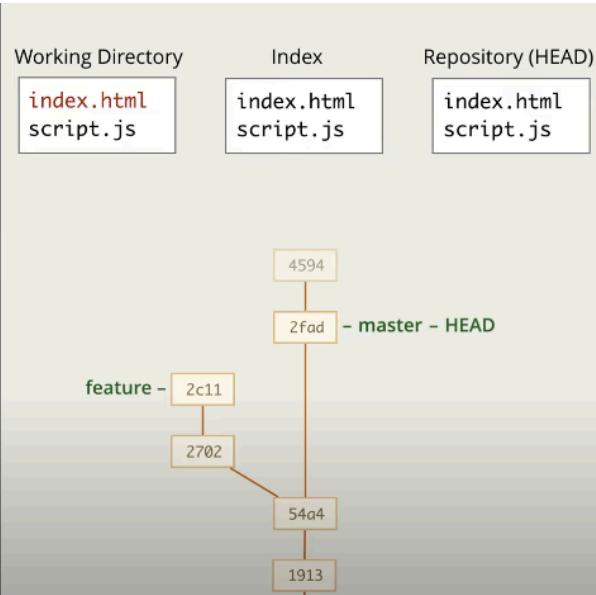
~/project master +1> git branch -v
  feature 2c11f12 Run work
* master  2fad3ac Run sayHi

~/project master +1> git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
    (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   index.html

no changes added to commit (use "git add" and/or "git commit -a")

```



важнейшее применение смешанного ресета:

1) очистка индекса:

```
~/project master> git reset HEAD
```

2) очистка определенного файла из индекса:

```

~/project master •1> git reset index.html
Unstaged changes after reset:
M      index.html

```

3) получение версии файла из другого комита (передать коммит и путь):

```
~/project master +1> git reset 54a4 index.html
```

в таком случае index.html будет обновлен как index.html в коммите 54a4. рабочая директория не меняется, только индекс. Такое используют редко, так как обычно нужно изменить не только в индексе ,но и в текущей директории. (этим как раз занимается checkout как рассмотрено было ранее)

## СРАВНЕНИЕ КОММИТОВ, ВЕТОК И НЕ ТОЛЬКО

**git diff** - для сравнения коммитов или файлов

1) **git diff <commit1> <commit2>** сравнение состояний двух

коммитов (можно передать ссылки (ветки) или идентификаторы)

флаг `--name-only` - чтобы вывести только названия разных файлов

```
~/project master> git diff master feature
```

```
diff --git a/index.html b/index.html
index da9736c..f4952f9 100644
--- a/index.html
+++ b/index.html
@@ -5,14 +5,10 @@
<script src="script.js"></script>
</head>
<body>
-  <script>
-    sayHi();
-  </script>
-
  Git rules!

  <script>
-    sayBye();
+    work();
  </script>
</body>
</html>

diff --git a/script.js b/script.js
index 935d66a..ec09669 100644
--- a/script.js
+++ b/script.js
@@ -2,6 +2,10 @@ function sayHi() {
  alert(`Hello from Git!`);
}

+function work() {
+  alert(`Work, work!`);
+}
```

Для каждого файла отдельный блок

какие файлы сравниваются: а - кодовое название для первого источника (ветка master), б - в нашем случае ветка feature

```
diff --git a/index.html b/index.html
index da9736c..f4952f9 100644
--- a/index.html
+++ b/index.html
@@ -5,14 +5,10 @@
```

Далее идут заголовки diff:

```
diff --git a/index.html b/index.html
index da9736c..f4952f9 100644
--- a/index.html
+++ b/index.html
@@ -5,14 +5,10 @@
```

100644 - тип объекта, перед ним написано какая контрольная сумма файла была и какая стала, то есть видно что разная, значит поменялся файл

после подчеркнутой строки описано: если из index.html в мастере удалить строки помеченные минусом и добавить строки помеченные плюсом, то получим index.html в feature. Подчеркнуты номера строк, к которым нужно применить изменения

```
diff --git a/index.html b/index.html
index da9736c..f4952f9 100644
--- a/index.html
+++ b/index.html
@@ -5,14 +5,10 @@
<script src="script.js"></script>
</head>
<body>
-  <script>
-    sayHi();
-  </script>
-
  Git rules!

  <script>
-    sayBye();
+    work();
  </script>
</body>
</html>
```

подчеркнутая запись означает: первый исходный файл из мастера (-) если взять в нем начиная с пятой строки 14 строк и применить к ним полученные изменения, то получится в итоговом файле (+) начиная с пятой 10 строк

если файл большой то в нем может быть много изменений, например в начале и конце, тогда не будет весь большой файл показан, а он разобьется на несколько кусков диф

```
diff --git a/index.html b/index.html
index da9736c..f4952f9 100644
--- a/index.html
+++ b/index.html
@@ -5,14 +5,10 @@
<script src="script.js"></script>
</head>
<body>
- <script>
-   sayHi();
- </script>
-
  Git rules!

  <script>
-   sayBye();
+   work();
  </script>
</body>
</html>
```

Еще есть такие заголовки, их может не быть или они могут быть неправильными, в общем не обращать на них внимание

```
@@ -5,14 +5,10 @@
<script src="script.js"></script>
</head>
<body>
- <script>
-   sayHi();
- </script>
-
  Git rules!

  <script>
-   sayBye();
+   work();
  </script>
</body>
</html>
diff --git a/script.js b/script.js
index 935d66a..ec09669 100644
--- a/script.js
+++ b/script.js
@@ -2,6 +2,10 @@ function sayHi() {
  alert(`Hello from Git!`);
}

+function work() {
+  alert(`Work, work!`);
+}
+
```

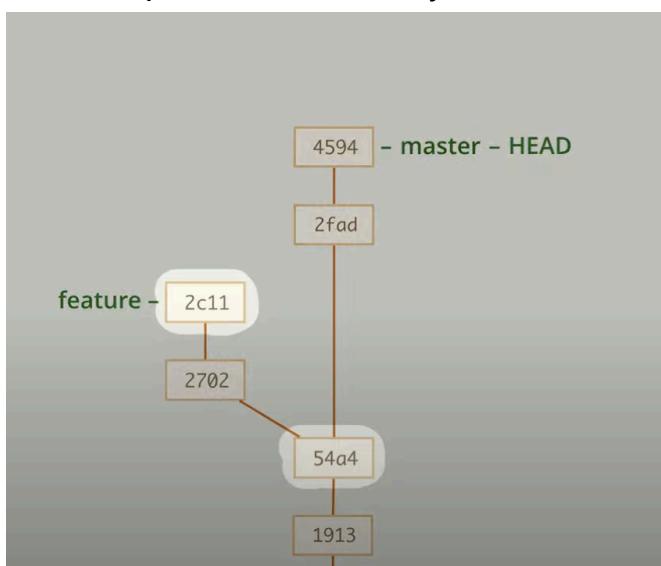
можно так вызывать с двумя точками (просто другой синтаксис):

```
~/project master> git diff master..feature
```

- 2) **git diff <1>...<2>** - что изменилось в feature с момента ее ответвления от master (здесь именно ветки передаем, не коммиты)

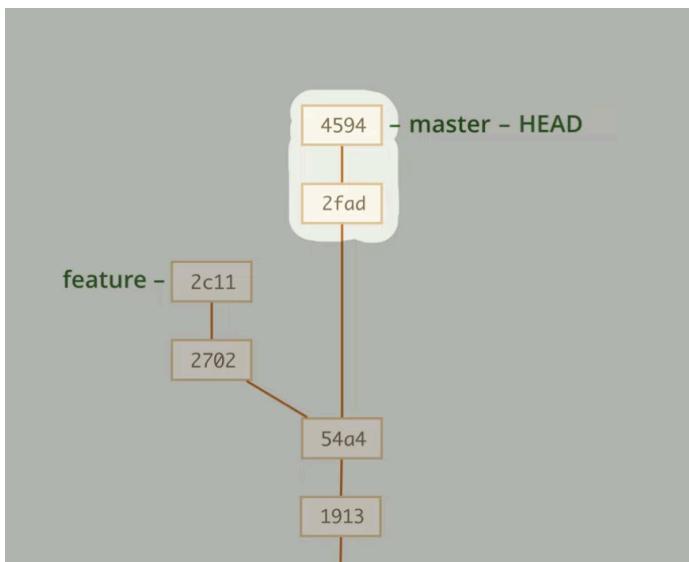
```
~/project master> git diff master...feature
```

то есть сравнение этих двух коммитов:



если хотим наоборот: узнать что произошла в ветке мастер с момента ее расхождения с веткой feature:

```
~/project master> git diff feature...master
```



- 3) **git diff** - сравнивает рабочую директорию с индексом, то есть показывает непроиндексированные изменения
- 4) **git diff <идентификатор/ссылка>** - сравнивает содержимое рабочей директории с репозиторием на момент этого коммита

```
~/project master> git diff 2fad
```

#### Частный случай с HEAD:

Посмотреть изменения в рабочей директории с момента последнего коммита:

```
~/project master> git diff HEAD
```

если в индексе нет изменений то git diff и git diff HEAD работают одинаково, а если изменения есть, то по-разному.



**ВАЖНО!!! сравниваются только отслеживаемые файлы**

**5) git diff --cached <коммит>-** покажет изменения, которые проиндексированы, но еще не в репо (коммит обычно не указывают и сравнивают с HEAD), то есть что сейчас планируем закоммитить

```
~/project master •2+1> git diff --cached
```



**6) git diff <файл>** - покажет изменения конкретного файла/директории (как и обычный git diff, который без указания коммита, этот тоже сравнивает репо с индексом)

```
~/project master •2+1> git diff NEWS
```

Если указать с путем, то сравнивает репо с директорией

```
~/project master •2+1> git diff HEAD index.html
```

можно передавать несколько путей и/или несколько файлов просто через пробел

Вообще рекомендуется перед путем ставить `-`, так как файл или директорию могут называться как ветка и двойной дефис нужен чтобы сказать гиту что это путь а не ветка

# СРАВНЕНИЕ ФАЙЛОВ

› `git diff commit1:path1 commit2:path2`

- `git diff commit1:path1 commit2:path2`
  - › `git diff master:one.html feature:two.html`

Сравнение любых двух файлов на диске, которые никак не связаны с гит

› `git diff --no-index path1 path2`

## GIT LOG

**git log - выводит коммиты, достижимые из HEAD  
(наверху самый новый)**

**флаги:**

**--pretty** - поменять формат (по умолчанию = medium)

```
~/project master> git log --pretty=oneline
4594f10bab02bdf034e0fab8d57b5bc09fb21594 (HEAD -> master) Run sayBye
2fad3acb6d276d2d1e504a3ba7d4d75bc6d85963 Run sayHi
54a4be6ff4ca63c909328ce4894c9ab0bc632c43 Create sayBye
19139752981c7ce057bc734b708a5334dd258069 Create sayHi
c3ef9b94833abfc8845de1492ed81aae8985b75c Initial commit
```

кроме oneline существует и много других

```
~/project master> git log --pretty=format:'%h %cd | %s%d [%an] '
4594f10 Thu Sep 14 08:32:33 2017 +0200 | Run sayBye (HEAD -> master) [Ilya Kantor]
2fad3ac Thu Sep 14 08:27:27 2017 +0200 | Run sayHi [Ilya Kantor]
54a4be6 Tue Sep 12 15:38:14 2017 +0200 | Create sayBye [Ilya Kantor]
1913975 Tue Sep 12 15:36:37 2017 +0200 | Create sayHi [Ilya Kantor]
c3ef9b9 Tue Sep 12 15:32:30 2017 +0200 | Initial commit [Ilya Kantor]
```

```
~/project master> git log --pretty=format:'%C(yellow)%h %C(dim green)%ad | %s%d [%an]' --date=format:'%F %R'
4594f10 2017-09-14 08:32 | Run sayBye (HEAD -> master) [Ilya Kantor]
2fad3ac 2017-09-14 08:27 | Run sayHi [Ilya Kantor]
54a4be6 2017-09-12 15:38 | Create sayBye [Ilya Kantor]
1913975 2017-09-12 15:36 | Create sayHi [Ilya Kantor]
c3ef9b9 2017-09-12 15:32 | Initial commit [Ilya Kantor]
```

вывести краткие идентификаторы:

```
~/project master> git log --pretty=oneline --abbrev-commit
4594f10 (HEAD -> master) Run sayBye
2fad3ac Run sayHi
54a4be6 Create sayBye
1913975 Create sayHi
c3ef9b9 Initial commit
```

```
~/project master> git log --oneline
4594f10 (HEAD -> master) Run sayBye
2fad3ac Run sayHi
54a4be6 Create sayBye
1913975 Create sayHi
c3ef9b9 Initial commit
```

флаг -p: добавляет к каждому коммиту diff того, что в нем было сделано

```
~/project master> git log -p
```

По умолчанию git log выводит коммиты, достижимые из HEAD, но можно задать другой коммит или ветку, тогда git log выведет коммиты, достижимые отсюда

Можно передать несколько коммитов или ссылок, тогда гитлог выведет все для них. Но тогда непонятна структура, какой коммит куда относится: поможет флаг --graph (как бы дерево консольное):

```
~/project master> git log master feature --graph
* 4594f10 2017-09-14 08:32 | (HEAD -> master) Run sayBye
* 2fad3ac 2017-09-14 08:27 | Run sayHi
| * 2c11f12 2017-09-12 22:49 | (feature) Run work
| * 2702040 2017-09-12 22:45 | Create work
|/
* 54a4be6 2017-09-12 15:38 | Create sayBye
* 1913975 2017-09-12 15:36 | Create sayHi
* c3ef9b9 2017-09-12 15:32 | Initial commit
```

коммиты достижимые из всех ссылок:

```
~/project master> git log --all --graph
```

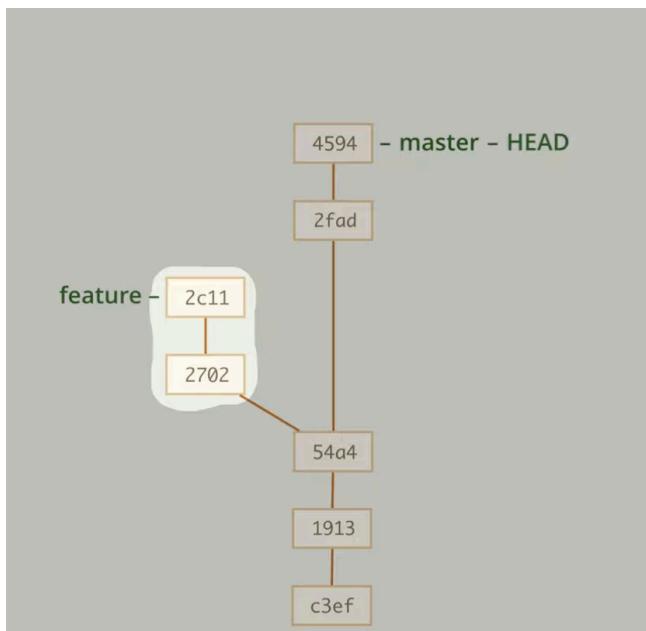
Если проект большой, то лучше использовать графические утилиты (git gui), их нужно установить. Но в реальности обычно нам не интересны все ветки, а интересна 1-2, поэтому гитлог отлично с этим справляется. + у гитлога очень много флагов

---

Очень частая задача посмотреть, какие коммиты были добавлены после ответвления ветки:

```
~/project master> git log feature ^master
2c11f12 2017-09-12 22:49 | (feature) Run work
2702040 2017-09-12 22:45 | Create work
```

то есть здесь говори что интересна ветку feature без коммитов, которые есть в master:



можно записать и так (выполняют одну и ту же функцию, просто пишутся по-разному)

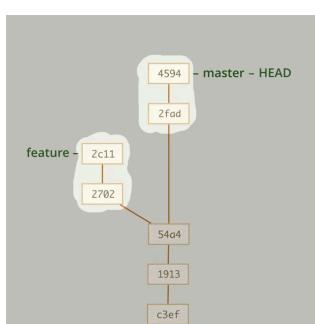
```
~/project master> git log master..feature
2c11f12 2017-09-12 22:49 | (feature) Run work
2702040 2017-09-12 22:45 | Create work
```

во всех этих выводах отсутствует пограничный коммит, на котором ветки разделяются. Если он нужен, то флаг --boundary

---

Симметрическая разность, то есть коммиты которые достижимы из master или feature, но не из обоих

```
~/project master> git log master...feature
```



`git log <файл>` - вывести все коммиты, в которых менялся этот файл  
флаг `-r` чтобы показать конкретные различия

```
~/project master> git log script.js
54a4be6 2017-09-12 15:38 | Create sayBye
1913975 2017-09-12 15:36 | Create sayHi

~/project master> git log -p script.js
54a4be6 2017-09-12 15:38 | Create sayBye
diff --git a/script.js b/script.js
index 0f67e66..935d66a 100644
--- a/script.js
+++ b/script.js
@@ -1,3 +1,7 @@
 function sayHi() {
   alert(`Hello from Git!`);
 }

+function sayBye() {
+  alert(`Goodbye from Git!`);
+}

1913975 2017-09-12 15:36 | Create sayHi
diff --git a/script.js b/script.js
new file mode 100644
index 0000000..0f67e66
--- /dev/null
+++ b/script.js
@@ -0,0 +1,3 @@
+function sayHi() {
+  alert(`Hello from Git!`);
+}
```

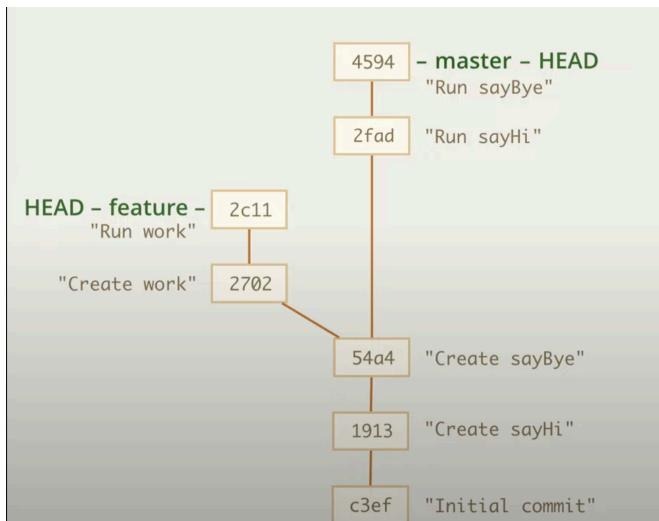
флаг `--follow`: если в процессе разработки файл был переименован, то `log` найдет предыдущее имя файла и продолжит выводить историю для него

Путь можно прибавлять к любому синтаксису `git log`:

```
~/project master> git log feature..master index.html
4594f10 2017-09-14 08:32 | (HEAD -> master) Run sayBye
2fad3ac 2017-09-14 08:27 | Run sayHi
```

и конечно можно указывать несколько файлов или директорию

## Флаги фильтры (для поиска коммитов):



--grep <слово><ветка> - поиск всех коммитов где есть это слово  
(по умолчанию ветка HEAD)

```
~/project master> git log --grep Run
4594f10 2017-09-14 08:32 | (HEAD -> master) Run sayBye
2fad3ac 2017-09-14 08:27 | Run sayHi
```

```
~/project master> git log --grep Run feature
2c11f12 2017-09-12 22:49 | (feature) Run work
```

поиск по нескольким (или то, или то):

```
~/project master> git log --grep Run --grep sayHi
4594f10 2017-09-14 08:32 | (HEAD -> master) Run sayBye
2fad3ac 2017-09-14 08:27 | Run sayHi
1913975 2017-09-12 15:36 | Create sayHi
```

если нужно не “или”, а “и то и то”, то флаг --all-match:

```
~/project master> git log --grep Run --grep sayHi --all-match
2fad3ac 2017-09-14 08:27 | Run sayHi
```

По регулярным выражениям можно искать (-P=perl необходимо, так как по умолчанию гит использует старый стандарт регулярок, а сейчас наиболее распространены перл-совместимые регулярки):

```
~/project master> git log --grep 'say(Hi|Bye)' -P  
4594f10 2017-09-14 08:32 | (HEAD -> master) Run sayBye  
2fad3ac 2017-09-14 08:27 | Run sayHi  
54a4be6 2017-09-12 15:38 | Create sayBye  
1913975 2017-09-12 15:36 | Create sayHi
```

отключить регулярные выражения: **git log -F**

**По умолчанию поиск регистрозависимый, это можно изменить флагом -i:**

```
~/project master> git log --grep sayhi -i  
2fad3ac 2017-09-14 08:27 | Run sayHi  
1913975 2017-09-12 15:36 | Create sayHi
```

Все выше мы искали по сообщениям, но в реальности нам нужен поиск по изменениям: **git log -G<регулярное выражение>** ГИТ выведет все коммиты в изменениях которых есть строка которая под него подпадает

```
~/project master> git log -GsayHi  
2fad3ac 2017-09-14 08:27 | Run sayHi  
1913975 2017-09-12 15:36 | Create sayHi
```

```
~/project master> git log -GsayHi -p  
2fad3ac 2017-09-14 08:27 | Run sayHi  
diff --git a/index.html b/index.html  
index e8e448b..fdb09e 100644  
--- a/index.html  
+++ b/index.html  
@@ -5,6 +5,10 @@  
     <script src="script.js"></script>  
   </head>  
   <body>  
+    <script>  
+      sayHi();  
+    </script>  
+  
+    Git rules!  
  
    Let's have some fun with git.  
  
1913975 2017-09-12 15:36 | Create sayHi  
diff --git a/script.js b/script.js  
new file mode 100644  
index 0000000..0f67e66  
--- /dev/null  
+++ b/script.js  
@@ -0,0 +1,3 @@  
+function sayHi() {  
+  alert(`Hello from Git!`);  
+}
```

регулярка берется в кавычки, так как в ней содержится пробел, перед скобкой стоит слэш, так как это особый символ в регулярке)

```
* ~/project master> git log -G'function sayHi()' -p  
1913975 2017-09-12 15:36 | Create sayHi  
diff --git a/script.js b/script.js  
new file mode 100644  
index 0000000..0f67e66  
--- /dev/null  
+++ b/script.js  
@@ -0,0 +1,3 @@  
+function sayHi() {  
+  alert(`Hello from Git!`);  
+}
```

**флаг -L** - вывести все коммиты, в которых были изменения с 3 по 6 строку в файле index.html

```
~/project master> git log -L 3,6:index.html  
1913975 2017-09-12 15:36 | Create sayHi
```

номер строки не очень удобно передавать, поэтому вместо номера можно передать регулярное выражение для начала и конца:

```
~/project master> git log -L '/<head>/' , '/<\head>/' :index.html

1913975 2017-09-12 15:36 | Create sayHi
diff --git a/index.html b/index.html
--- a/index.html
+++ b/index.html
@@ -3,3 +3,4 @@
 <head>
     <title>Git rules</title>
+    <script src="script.js"></script>
 </head>

c3ef9b9 2017-09-12 15:32 | Initial commit
diff --git a/index.html b/index.html
--- /dev/null
+++ b/index.html
@@ -0,0 +3,3 @@
+ <head>
+     <title>Git rules</title>
+ </head>
```

это удобно для просмотра изменений функции/блока в файле

Еще один синтаксис ищем в файле изменение определенной функции:

```
~/project master> git log -L :sayHi:script.js
```

но как же гит определит, где функция, ведь функции в каждом языке объявляются по-разному?

по умолчанию любая строка начинающаяся с буквы, подчеркивания или доллара - это объявление функции; телом функции считает весь текст до следующей функции:

```
~/project master> cat script.js
function sayHi() {
  alert(`Hello from Git!`);
}

function sayBye() {
  alert(`Goodbye from Git!`);
}
```

такое не часто используют, так видно, что потенциально много “но”

функцию удобно искать по принципу “от-до”, в “до” здесь указано, что строка научиться должна с фигурной скобки (именно начинаться, так может быть много закрывающихся скобок):

```
~/project master> git log -L '/^function sayHi/', '/^}/' :script.js
1913975 2017-09-12 15:36 | Create sayHi
diff --git a/script.js b/script.js
--- /dev/null
+++ b/script.js
@@ -0,0 +1,3 @@
+function sayHi() {
+  alert(`Hello from Git!`);
```

МОЖНО ИСКАТЬ ПО АВТОРУ, КОММИТТЕРУ, ИХ ЭМАЙЛАМ И ТД

```
~/project master> git log --author=Ilya
4594f10 2017-09-14 08:32 | (HEAD -> master) Run sayBye
2fad3ac 2017-09-14 08:27 | Run sayHi
54a4be6 2017-09-12 15:38 | Create sayBye
1913975 2017-09-12 15:36 | Create sayHi
c3ef9b9 2017-09-12 15:32 | Initial commit

~/project master> git log --author=iliakan
4594f10 2017-09-14 08:32 | (HEAD -> master) Run sayBye
2fad3ac 2017-09-14 08:27 | Run sayHi
54a4be6 2017-09-12 15:38 | Create sayBye
1913975 2017-09-12 15:36 | Create sayHi
c3ef9b9 2017-09-12 15:32 | Initial commit
```

По дате (до определенной даты, после определенной даты):

```
~/project master> git log --before '2017-09-13'  
54a4be6 2017-09-12 15:38 | Create sayBye  
1913975 2017-09-12 15:36 | Create sayHi  
c3ef9b9 2017-09-12 15:32 | Initial commit  
  
~/project master> git log --after '2017-09-13'  
4594f10 2017-09-14 08:32 | (HEAD -> master) Run sayBye  
2fad3ac 2017-09-14 08:27 | Run sayHi
```

Бывает что видим непонятный код или с ошибкой и хотим узнать, кто это написал - **git blame**:

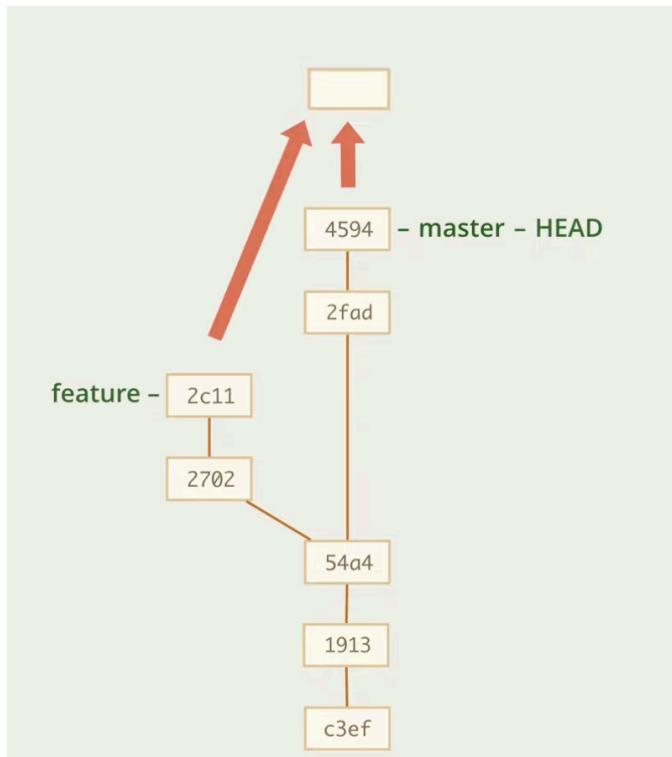
```
~/git master> git blame INSTALL
```

построчный вывод (справа строки файла, слева последний коммит который менял строку):

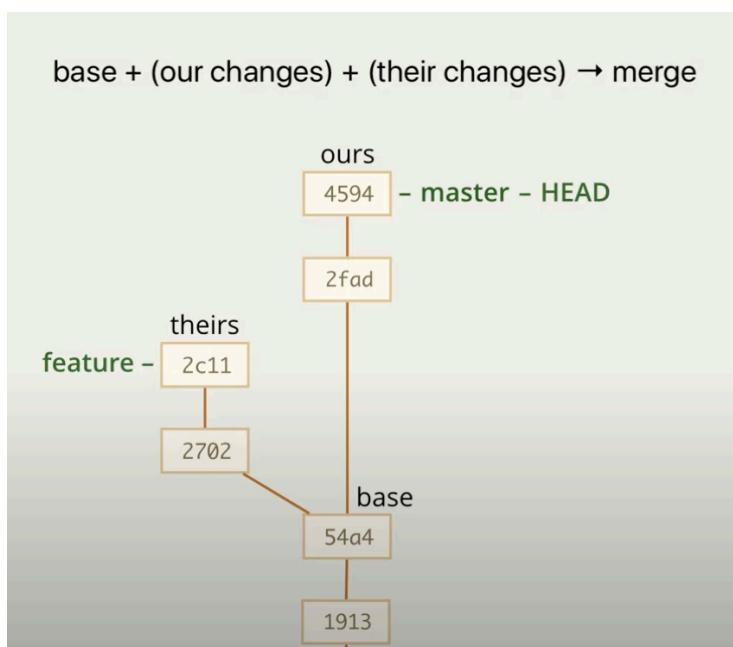
```
and that  
c538d2d34a (Linus Torvalds 2005-06-17 11:30:04 -0700 5) will install the git programs in your own ~/bin/ directory.  
If you want  
c538d2d34a (Linus Torvalds 2005-06-17 11:30:04 -0700 6) to do a global install, you can do  
c538d2d34a (Linus Torvalds 2005-06-17 11:30:04 -0700 7)  
98e79f63be (David Kastrup 2007-08-07 12:02:12 +0200 8) $ make prefix=/usr all doc info ;# as yourself  
414851a42e (Michael J Gruber 2008-09-10 10:19:34 +0200 9) # make prefix=/usr install install-doc install-html ins  
tall-info ;# as root  
c538d2d34a (Linus Torvalds 2005-06-17 11:30:04 -0700 10)  
c44922a781 (Junio C Hamano 2005-11-09 12:40:03 -0800 11) (or prefix=/usr/local, of course). Just like any program su  
ite  
c44922a781 (Junio C Hamano 2005-11-09 12:40:03 -0800 12) that uses $prefix, the built results have some paths encoded  
,  
c44922a781 (Junio C Hamano 2005-11-09 12:40:03 -0800 13) which are derived from $prefix, so "make all; make prefix=/u  
sr  
c44922a781 (Junio C Hamano 2005-11-09 12:40:03 -0800 14) install" would not work.  
c538d2d34a (Linus Torvalds 2005-06-17 11:30:04 -0700 15)  
Sbeb577db8 (Brian Gernhardt 2009-09-10 16:28:19 -0400 16) The beginning of the Makefile documents many variables that  
affect the way  
Sbeb577db8 (Brian Gernhardt 2009-09-10 16:28:19 -0400 17) git is built. You can override them either from the command  
line, or in a  
2009-09-10 16:28:19 -0400 18)
```

## GIT MERGE (ранее рассмотрен был простой случай, сейчас более реальные и сложные)

**Истинное слияние** - при слиянии образуется новый коммит



перед слиянием нужно чтобы статус был чистым



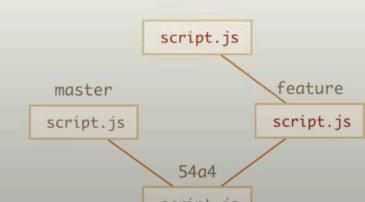
возможны ситуации, чтобы изменения конфликтуют между собой, то есть в обоих ветках поменяли файл в одном и том же месте по-разному, тогда гит выдаст сообщение об этом и скажет выбрать правильный вариант

```
~/project master> git merge feature
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

При конфликте мы автоматически попадаем в состояние “прерванное слияние”

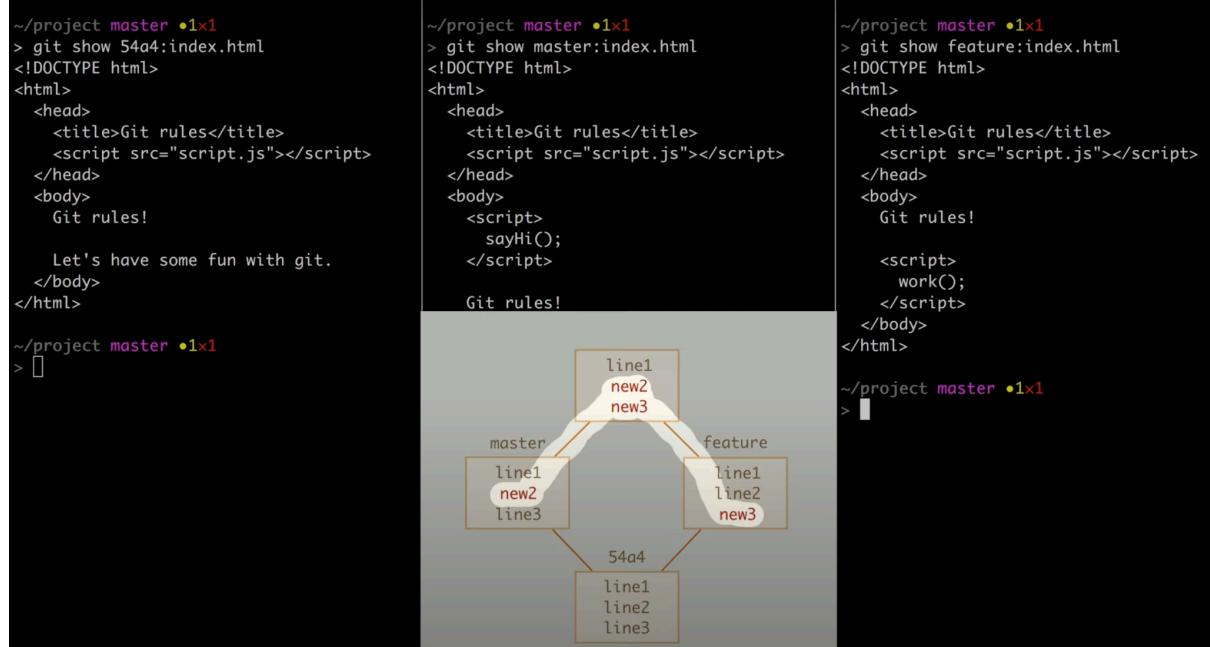
Вот объяснение правильного слияния:

|                                                                                                                                                                                                    |                                                                                                                                                                                                      |                                                                                                                                                                                                                                                    |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>~/project master •1x1 &gt; git show 54a4:script.js function sayHi() {   alert(`Hello from Git!`); }  function sayBye() {   alert(`Goodbye from Git!`); }  ~/project master •1x1 &gt; []</pre> | <pre>~/project master •1x1 &gt; git show master:script.js function sayHi() {   alert(`Hello from Git!`); }  function sayBye() {   alert(`Goodbye from Git!`); }  ~/project master •1x1 &gt; []</pre> | <pre>~/project master •1x1 &gt; git show feature:script.js function sayHi() {   alert(`Hello from Git!`); }  function work() {   alert(`Work, work!`); }  function sayBye() {   alert(`Goodbye from Git!`); }  ~/project master •1x1 &gt; []</pre> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

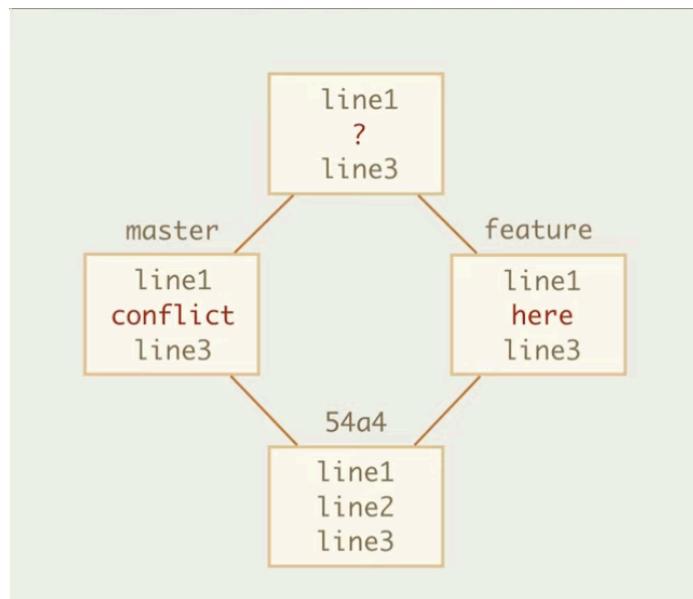


во втором столбце (master) нет изменений с первым столбцом, в третьем столбце (feature) есть изменения. Дальше гит хочет добавить в базовую ветку изменения из обеих веток: в master изменения не вносились, поэтому берутся изменения только из feature. Получается, что версия feature становится итоговой

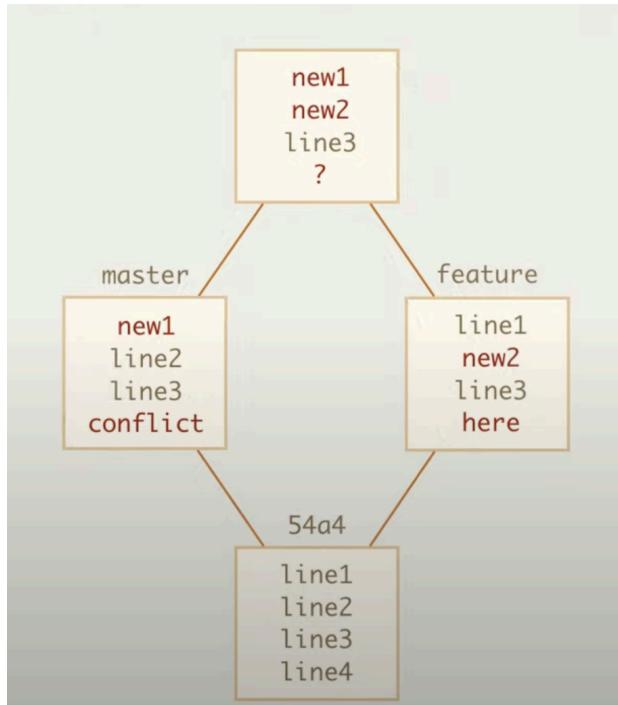
Если изменения были в двух столбцах (ветках), то это уже задача интеллектуальная, но Гит пытается ее решить. Если изменения касаются разных строк, то слияние будет успешно как показано на схеме



### КОНФЛИКТ СЛИЯНИЯ:



## Бывает что что-то удалось объединить, а что-то нет



```
~/project master •1x1
> git show 54a4:index.html
<!DOCTYPE html>
<html>
  <head>
    <title>Git rules!</title>
    <script src="script.js"></script>
  </head>
  <body>
    Git rules!
    Let's have some fun with git.
  </body>
</html>

~/project master •1x1
> git diff -U0 54a4 master index.html
diff --git a/index.html b/index.html
index e8e448b..da9736c 100644
--- a/index.html
+++ b/index.html
@@ -7,0 +8,4 @@
+  <script>
+    sayHi();
+  </script>
+
@@ -10 +14,3 @@
-  Let's have some fun with git.
+  <script>
+    sayBye();
+  </script>

~/project master •1x1
> git show feature:index.html
<!DOCTYPE html>
<html>
  <head>
    <title>Git rules!</title>
    <script src="script.js"></script>
  </head>
  <body>
    Git rules!
    <script>
      work();
    </script>
  </body>
</html>

~/project master •1x1
> git diff -U0 54a4 feature index.html
diff --git a/index.html b/index.html
index e8e448b..f4952f9 100644
--- a/index.html
+++ b/index.html
@@ -10 +10,3 @@
-  Let's have some fun with git.
+  <script>
+    work();
+  </script>
```

редактор вот так изменит файл указав на конфликт:

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>Git rules</title>
5      <script src="script.js"></script>
6    </head>
7    <body>
8      <script>
9        sayHi();
10     </script>
11
12     Git rules!
13
14     <script>
15 <<<<< HEAD
16       sayBye();
17 =====
18       work();
19 >>>>> feature
20       </script>
21     </body>
22   </html>
23
```

вытащим файл из master:

```
~/project master •1x1> git checkout --ours index.html
```

продолжим слияние, выберем один вариант и получим опять конфликт:

```
~/project master •1x1> git checkout --merge index.html
```

МОЖНО отменить слияние:

```
~/project master •1x1> git reset --hard
```

жесткий ресет прекратит слияние и очистит все незакоммиченные изменения включая конфликты несколько аккуратнее сработает:

```
~/project master •1x1> git reset --merge
```

он оставляет незакоммиченные изменения в файлах, которые не участвовали в слиянии (то есть которые в обеих ветках одинаковые), он удаляет изменения которые внесены в индекс и конфликтные, но не трогает те, где изменения только в рабочей директории

Теперь рассмотрим вариант, когда не отменяем слияние, а решаем конфликт. При решении хотелось бы посмотреть не только то что в нашей ветке и сливаемой ветке, а еще и то что было до разделения веток:

```
* ~/project master •1x1> git checkout --conflict=diff3 --merge index.html
```

```
Git rules!

<<<<< ours
<script>
  sayBye();
</script>
||||||| base
| Let's have some fun with git.
=====
<script>
  work();
</script>
>>>>> theirs
</body>
</html>
```

то что находится в `base` - это то что было до разделения

МОЖНО ПОСТАВИТЬ ЧТОБЫ ТАК ПО УМОЛЧАНИЮ ВСЕГДА ВЫВОДИЛОСЬ с base:

```
~/project master •1x1> git config --global merge.conflictStyle diff3
```

Исправлено от руки, теперь правильное объединение

```
<!DOCTYPE html>
<html>
  <head>
    <title>Git rules</title>
    <script src="script.js"></script>
  </head>
  <body>
    <script>
      sayHi();
    </script>

    Git rules!

    <script>
      work();
      sayBye();
    </script>
  </body>
</html>
```

Из-за особенностей гита, статус не будет чист, надо добавить в индекс и завершить слияние

```
~/project master •1x1> git status
On branch master
You have unmerged paths.

Changes to be committed:
  modified:  script.js

Unmerged paths:
  both modified:  index.html

~/project master •1x1> git commit
U      index.html
error: Committing is not possible because you have unmerged files.
hint: Fix them up in the work tree, and then use 'git add/rm <file>'
hint: as appropriate to mark resolution and make a commit.
fatal: Exiting because of an unresolved conflict.

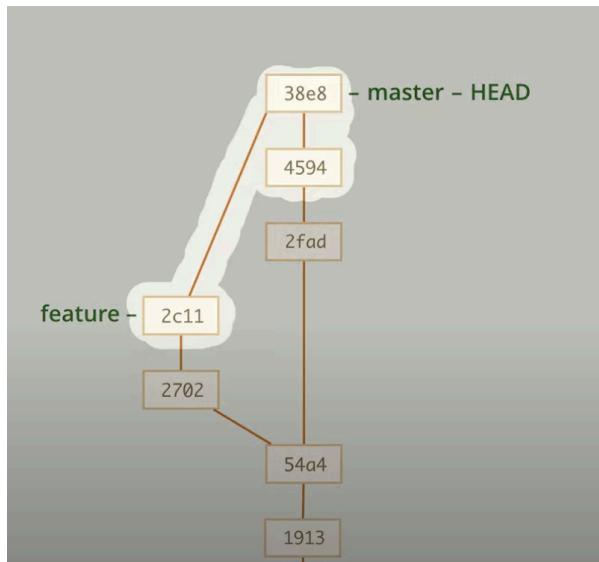
x ~/project master •1x1> git add index.html

~/project master •2> git status
On branch master
All conflicts fixed but you are still merging.

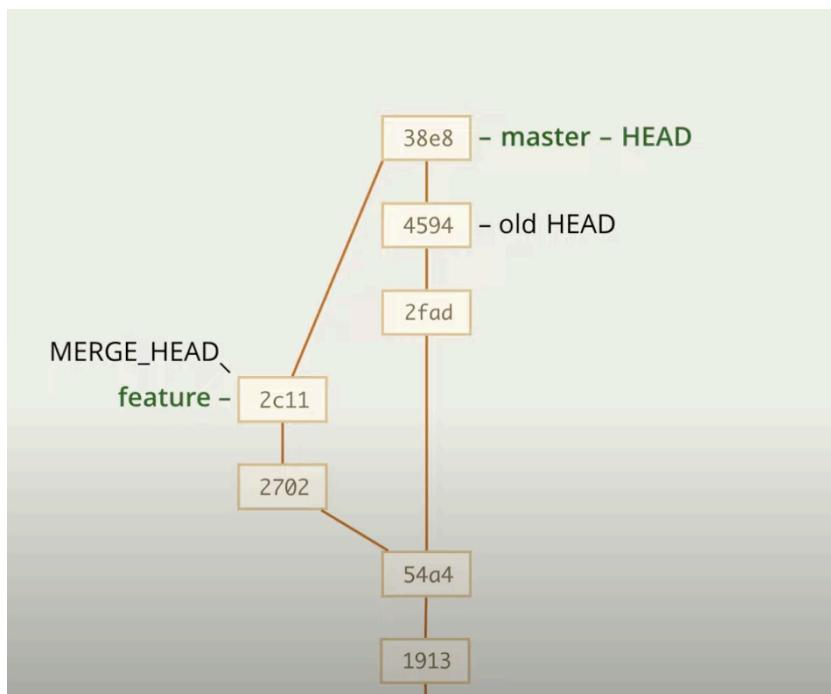
Changes to be committed:
  modified:  index.html
  modified:  script.js

~/project master •2> git merge --continue
```

Получаем особый коммит (коммит слияния), у него целых 2 родителя



При мерче гит создает файл MERGE\_HEAD - содержит идентификатор ветки, с которой мысливаем и команда коммит учитывает этот файл. Если он существует, то мы находимся в процессе слияния и при создании нового коммита соответствующий идентификатор добавляется в него как дополнительный родитель. После чего MERGE\_HEAD удаляется и слияние завершено



```
~/project master> git show
commit 38e84f1b37991a14a1398d5c5c4aa35c899c6648 (HEAD -> master)
Merge: 4594f10 2c11f12
Author: Ilya Kantor <iliakan@gmail.com>
Date:   Thu Oct 26 10:50:30 2017 +0300

    Merge branch 'feature'

diff --cc index.html
index da9736c,f4952f9..31d04b3
--- a/index.html
+++ b/index.html
@@@ -12,7 -8,7 +12,8 @@
    Git rules!

<script>
+    work();
+    sayBye();
</script>
</body>
</html>
```

здесь специальный diff, поэтому выглядит по-другому, + левее означает что не этой строки не было в первом родителе (то есть было во втором в feature), правее что не было во втором родителе (то есть было в первом в master), два плюса рядом бы означали что ни в одном родителе этого не было, то есть например добавили при решении конфликта слияния. вторая особенность такого дифа, что он показывает не все изменения, а только те, в которых были конфликты. Специальный диф предназначен, чтобы показать как разрешились конфликты

Если хотим посмотреть отличия от первого родителя:

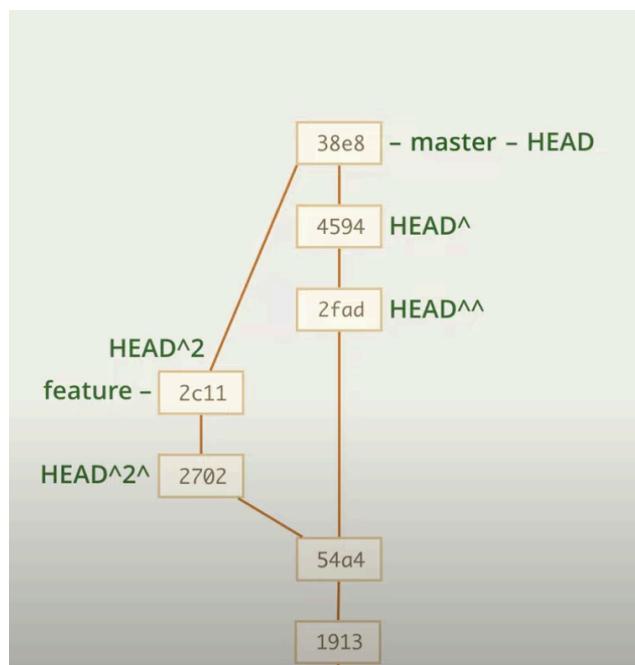
```
~/project master> git show --first-parent
```

но удобнее использовать обычный git diff  
посмотреть изменения по сравнению с предыдущей версией  
первого родителя (master)

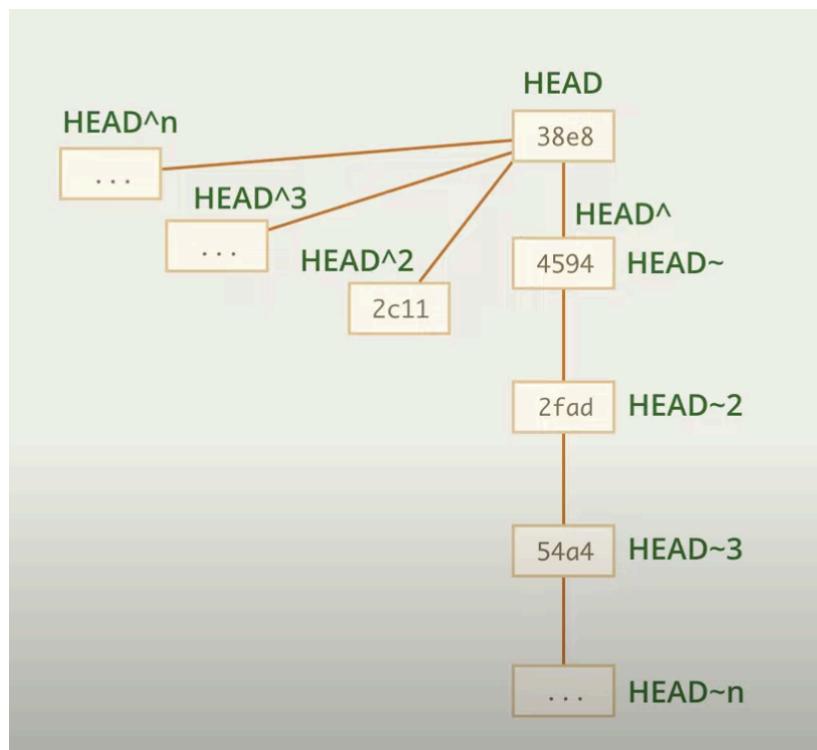
```
~/project master> git diff HEAD^
```

второго родителя:

```
~/project master> git diff HEAD^2
```



Не путать ~ и ^:



<sup>^</sup> имеет смысл для коммитов слияния, так она по сути к какому родителю перейти

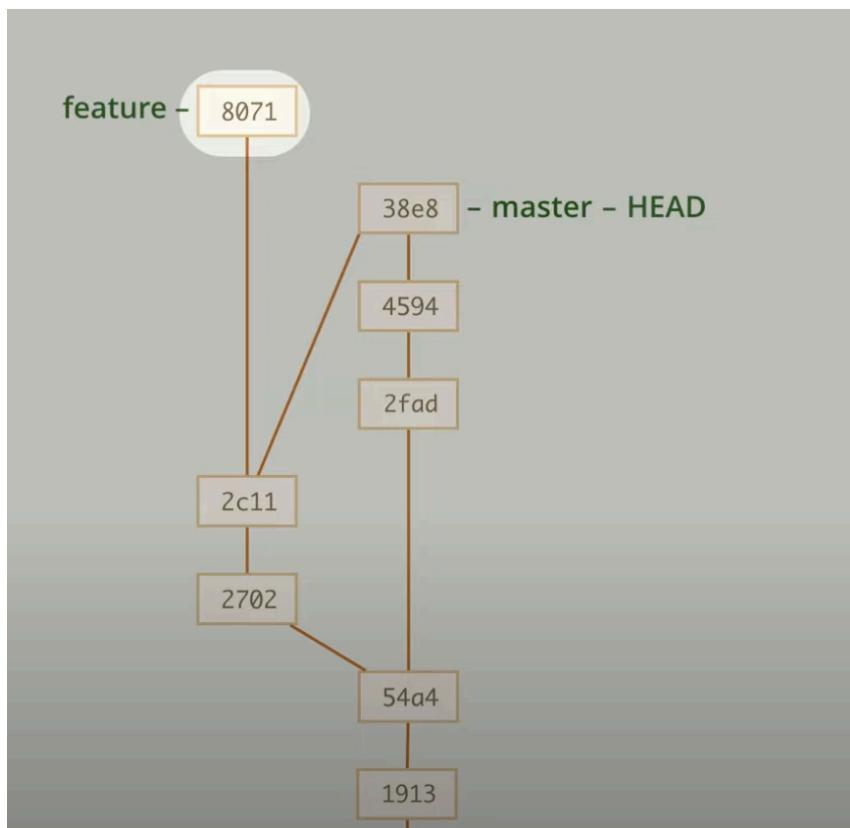
показывает ветки, которые объединены с текущей

```
~/project master> git branch --merged  
  feature  
* master
```

показывает ветки, которые не объединены с текущей

```
~/project master> git branch --no-merged
```

сделали изменения и хотим опять слить, гит знает, что до этого уже было слияние, поэтому сливать будет только коммит 8071, а не всю ветку feature.



указать сообщение при слиянии:

```
~/project master> git merge feature -m 'Improved work'
```

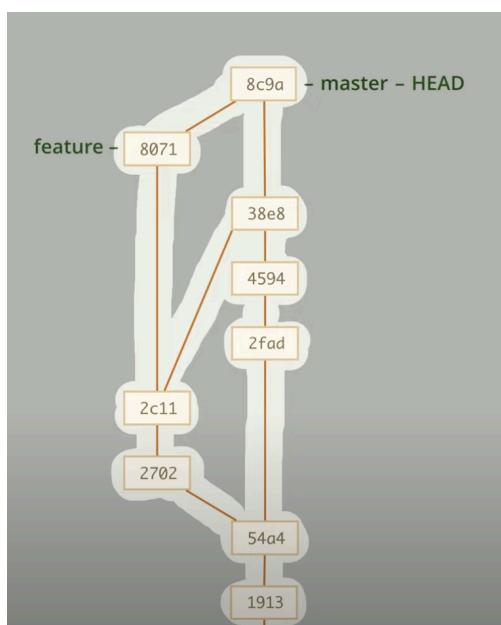
можно добавить в описание коммита слияния описание всех сливаемых коммитов с ветки feature (можно задать лимит сообщений, по умолчанию 20)

```
~/project master> git merge feature --log=5
```

это удобно, чтобы в коммите слияния подробно описать что было сделано

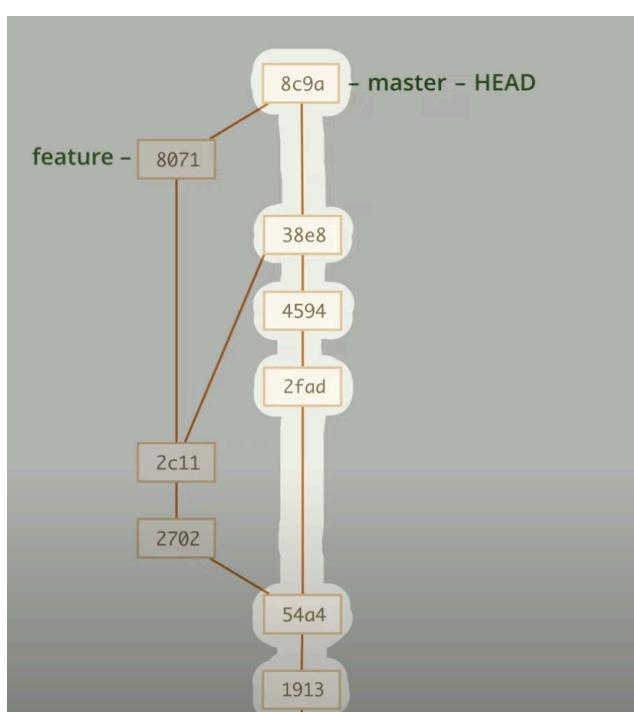
Выведем информацию об коммитах ветки master - выведутся все коммиты вообще

```
~/project master> git log master --oneline
8c9af5a (HEAD -> master) Merge branch 'feature'
807167a (feature) Work harder!
38e84f1 Merge branch 'feature'
4594f10 Run sayBye
2fad3ac Run sayHi
2c11f12 Run work
2702040 Create work
54a4be6 Create sayBye
1913975 Create sayHi
c3ef9b9 Initial commit
```



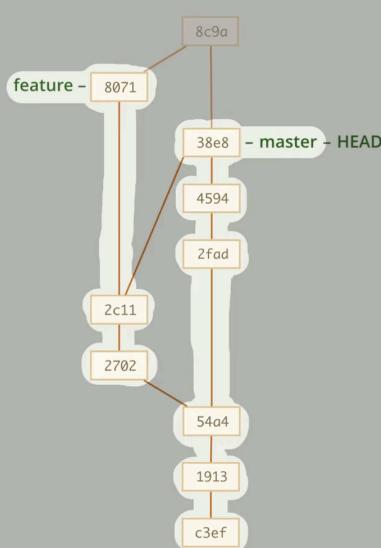
Если хотим только коммиты именно из master, то флаг  
**--first-parent**: если гит видит коммит слияния, то идет дальше  
только по первому родителю

```
~/project master> git log master --oneline --first-parent
8c9af5a (HEAD -> master) Merge branch 'feature'
38e84f1 Merge branch 'feature'
4594f10 Run sayBye
2fad3ac Run sayHi
54a4be6 Create sayBye
1913975 Create sayHi
c3ef9b9 Initial commit
```



## ОТМЕНА СЛИЯНИЯ УЖЕ ПОСЛЕ УСПЕШНОГО СЛИЯНИЯ - отмена как отмена обычного коммита

```
~/project master> git reset --hard @~  
HEAD is now at 38e84f1 Merge branch 'feature'  
~/project master>
```



и чтобы вернуть точно как раньше посмотреть в рефлоге идентификатор и передвинуть туда ветку

```
~/project master> git reflog -4  
142fc92 (HEAD -> master) HEAD@{0}: commit (merge): Merge branch  
'feature'  
4594f10 HEAD@{1}: reset: moving to @~  
38e84f1 HEAD@{2}: reset: moving to @~  
8c9af5a HEAD@{3}: merge feature: Merge made by the 'recursive'  
strategy.  
~/project master> git reset --hard @{3}
```

## СЕМАНТИЧЕСКИЕ КОНФЛИКТЫ И ИХ РАЗРЕШЕНИЕ

Например мы ответвили ветку от master и добавили в ней еще один вызов существующей функции, а в ветку master мы изменили определение функции и решили передавать ей аргументы, вызов исправили с тех местах где он был. Но на ветке ответвленной никто не знает, что определение функции поменялось и выполняют вызов функции без аргумента. Затем произвели

слияние, конфликтов не было и все успешно слилось, но теперь код не рабочий, так как в нем одна и та же функция вызывается по разному:

```
index.html      script.js
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <title>Git rules</title>
5  <script src="script.js"></script>
6  </head>
7  <body>
8  <script>
9    sayHi();
10   work();
11  </script>
12
13  Git rules!
14
15  <script>
16    work(5);
17    sayBye();
18  </script>
19  </body>
20 </html>
21
```

2686 - master - HEAD  
work - f982  
63de  
8c9a

Это семантический конфликт - изменения конфликтуют по смыслу

Решение:

Сначала отменить слияние:

```
~/project master> git reset --hard @~  
HEAD is now at 63de0f7 Work n times harder
```

Затем произвести слияние с флагом **--no-commit**, чтобы гит сделал слияние, добавил все в индекс но остановился на этапе коммита

```
~/project master> git merge work --no-commit  
Auto-merging index.html  
Automatic merge went well; stopped before committing as request  
ed
```

Мы оказались в состоянии прерванного слияния

Исправляем вызов функции, добавляем в индекс и завершаем слияния:

```
~/project master •1+1> git add index.html
```

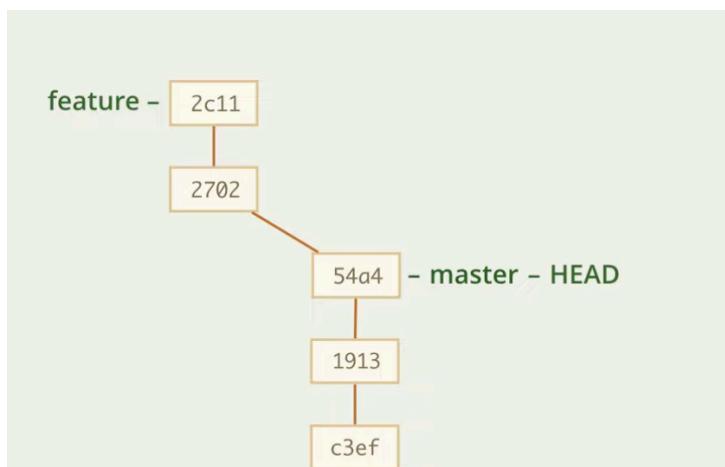
```
~/project master •1> cat .git/MERGE_HEAD  
f982755b03db743c3e2fb749ea2bb05399f0a3e5
```

```
~/project master •1> git merge --continue█
```

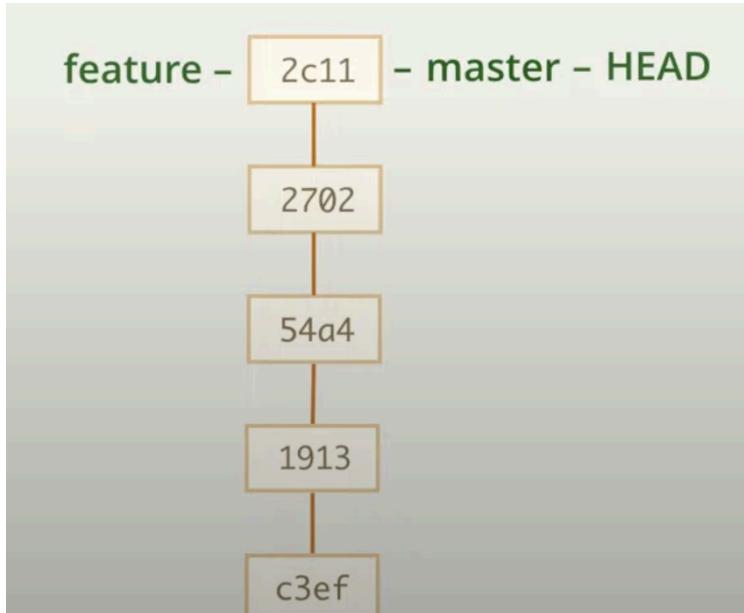
гит понимает что это прерванное слияние как раз благодаря файлу MERGE\_HEAD

---

Рассмотрим такое состояние репозитория, еще будем делать слияние, то будет по умолчанию слияние перемоткой (самый простой способ, так как после ответвления в master ничего не добавилось)



После слияния получаем такую картину:

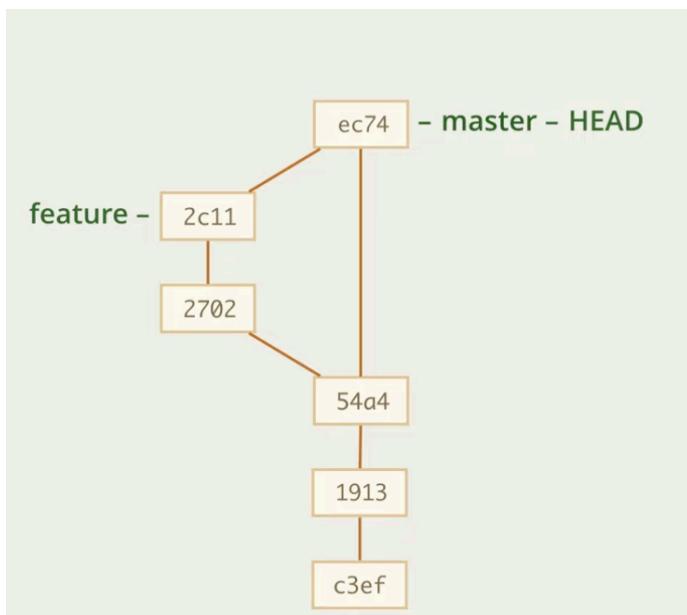


и не понятно где раньше было ответвление -> не понятно как отменить слияние (если сразу отменить, то orig\_head поможет, а если позже, то не понятно куда откатываться, придется искать в логе или рефлоге)

Решение: **флаг --no-ff (запрет на перемотку, заставит делать коммит слияния)**

```
~/project master> git merge --no-ff --no-edit feature
```

получим так как надо:



поставить такой флаг по умолчанию

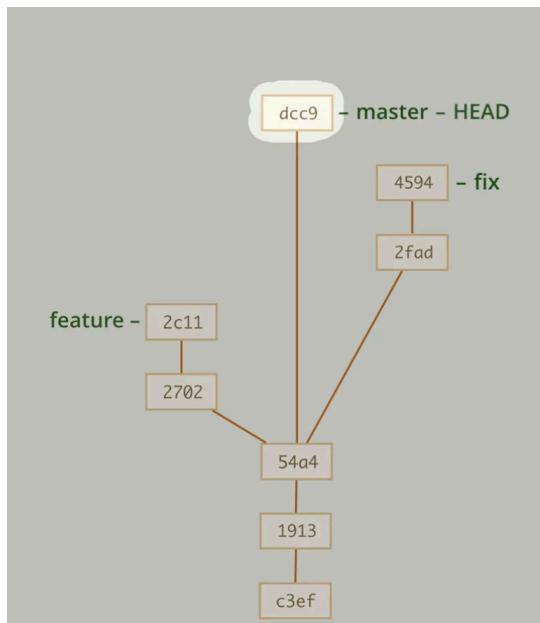
```
~/project master> git config merge.ff false
```

---

как правило история разработки для нас важна, но бывает и ситуации когда результат разработки какой-то ветки мы хотели бы интегрировать в master, но история создания такого результата нам не интересна: **git merge --squash**

```
~/project master> git merge --squash fix
Updating 54a4be6..4594f10
Fast-forward
Squash commit -- not updating HEAD
 index.html | 8 ++++++-
 1 file changed, 7 insertions(+), 1 deletion(-)
```

Эти изменения будут проиндексированы и когда мы закоммитим, мы получим коммит, в котором содержатся все изменения из fix (dcc9 содержит изменения из 2fad и 4594), а fix если она прям стремная была, что не хотим хранить историю разработки, вообще можно удалить



Нюансы работы:

```
~/project master> git merge --squash feature
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Squash commit -- not updating HEAD
Automatic merge failed; fix conflicts and then commit the result.
```

в случае если при слиянии конфликт, нам об этом сообщил как всегда. Теперь мы хотим отменить, а тут проблема:

```
* ~/project master •1x1> git merge --abort
fatal: There is no merge to abort (MERGE_HEAD missing).
```

гит сообщает, что нет слияния, которое нужно отменить. Обычное слияние вначале работы создает файл MERGE\_HEAD. При использовании флага squash MERGE\_HEAD не создается (поэтому и коммит будет обычный с одним родителем, так как нет файла, где взять второго родителя).

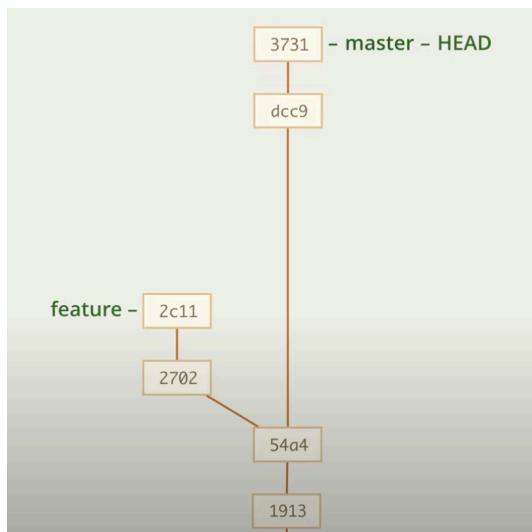
abort и continue не работают, но abort делает то же что и reset --merge, которая работает и очищает проиндексированные изменения:

```
* ~/project master •1x1> git reset --merge
```

а для завершения слияния вместо continue использовать git commit:

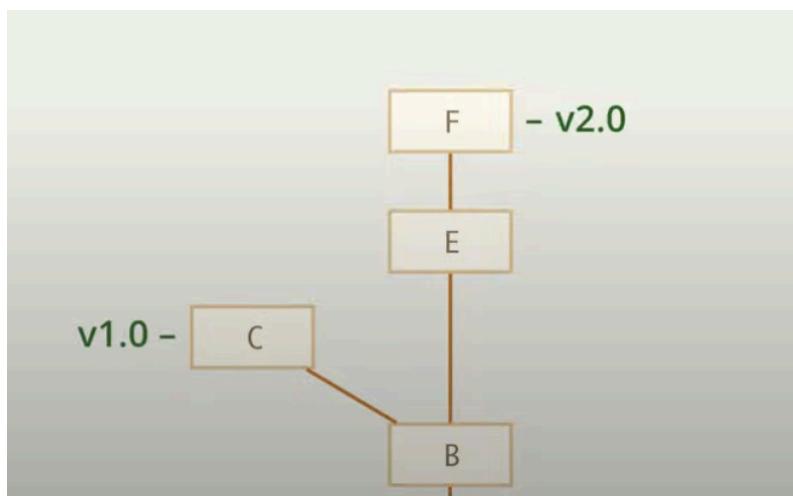
```
* ~/project master •1x1> git commit
```

отредактируем файл(решим конфликт), добавим в индекс и завершим слияние. Получим новый коммит, никак не связанный с feature.

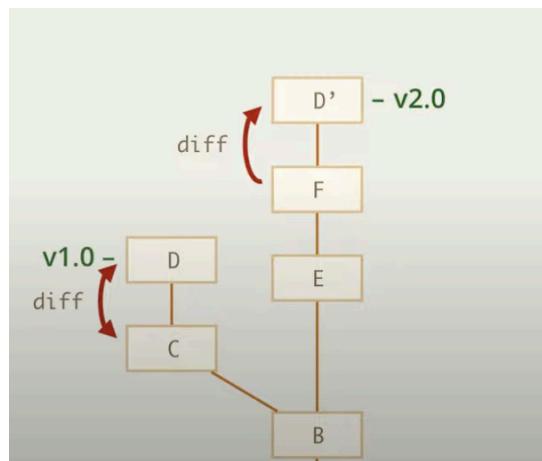


## КОПИРОВАНИЕ КОММИТОВ `cherry-pick`

Самое частое использование: когда есть две ветки и обнаружена ошибка в файле, которая есть и там и там

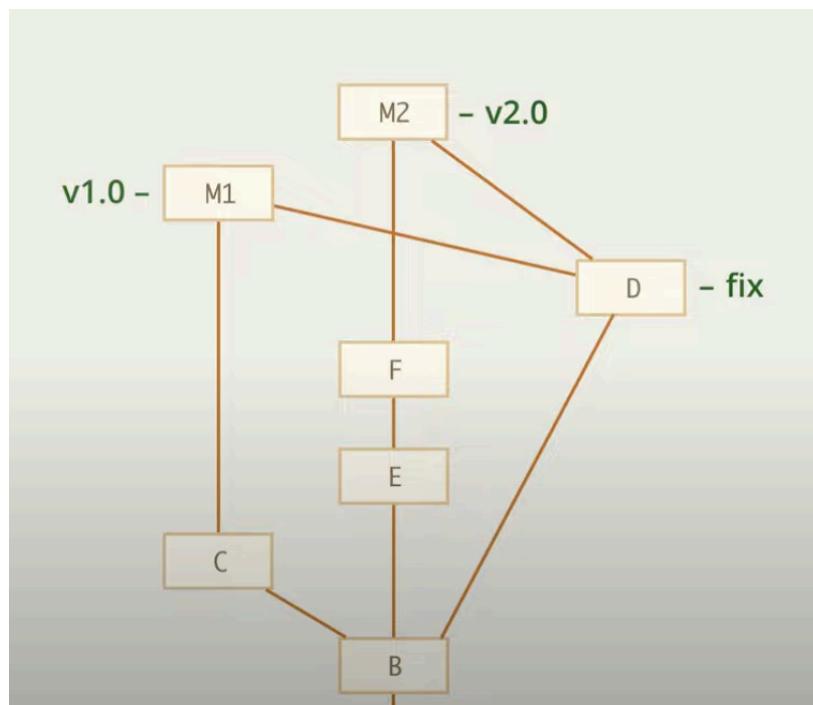


тогда можно исправить на одной ветке (коммит D) и командой `cherry-pick` скопировать этот коммит на ветку v2.0. Эта команда берет изменения, указанные в этом коммите и применяет их к текущей ветке, создавая новый коммит.



копирование коммитов, как и любым дублированием кода,  
желательно не злоупотреблять

еще один вариант: ошибка была до разделения веток, создать  
ветку fix и вмержить ее в две другие. Но это тоже не очень удобно,  
так как ветки могут вестись разными людьми и один разработчик  
мог в своей ветке уже исправить это.



После копирования коммита все будет одинаково, отличаться будут только идентификаторы (так как это все-таки разные коммиты), имя коммитера и дата коммита

```
~/project master> git show --no-decorate
commit c73b7b1c04fab7865c5337804f42e74ddbb6cac4
Author: Ilya Kantor <iliakan@gmail.com>
Date: Tue Sep 12 22:45:43 2017 +0200

Create work

diff --git a/script.js b/script.js
index 935d66a..ec09669 100644
--- a/script.js
+++ b/script.js
@@ -2,6 +2,10 @@ function sayHi() {
    alert(`Hello from Git!`);
}

+function work() {
+    alert(`Work, work!`);
+
+    function sayBye() {
        alert(`Goodbye from Git!`);
    }
}

~/project master> █
```

```
~/project master> git show 2702
commit 270204032166018243eb9caee83c1cbf7416aa49
Author: Ilya Kantor <iliakan@gmail.com>
Date: Tue Sep 12 22:45:43 2017 +0200

Create work

diff --git a/script.js b/script.js
index 935d66a..ec09669 100644
--- a/script.js
+++ b/script.js
@@ -2,6 +2,10 @@ function sayHi() {
    alert(`Hello from Git!`);

+function work() {
+    alert(`Work, work!`);
+
+    function sayBye() {
        alert(`Goodbye from Git!`);
    }
}

~/project master> █
```

cherry-pick может копировать не один коммит, а сразу несколько или даже целую ветку. Например хотим скопировать все изменения, сделанные в feature, в master:

можно перечислить через пробел все коммиты, а можно указать диапазон .. (такая запись означает добавит все коммиты feature которых нету в master):

```
~/project master> git cherry-pick master..feature
[master 0cd5f17] Create work
Date: Tue Sep 12 22:45:43 2017 +0200
1 file changed, 4 insertions(+)

error: could not apply 2c11f12... Run work
hint: after resolving the conflicts, mark the corrected paths
hint: with 'git add <paths>' or 'git rm <paths>'
hint: and commit the result with 'git commit'
```

первый коммит скопирован успешно, а второй - возникнет конфликт (как был выше при слиянии), так как по сути cherry-pick также накладывает изменения как и merge. Гит запоминает текущее состояние cherry-pick и ждет пока мы выберем одну из трех опций:

- 1) отменит скопированные коммиты и вернет все как было до cherry-pick

```
x ~/project master ✘1> git cherry-pick --abort
```

- 2) продолжит выполнение cherry-pick (сейчас в нашем случае это невозможно)

```
x ~/project master ✘1> git cherry-pick --continue
```

- 3) остановиться там где мы сейчас и сбросить запомненное состояние, то есть те коммиты, которые скопировались успешно, они остаются и команда cherry-pick как бы завершилась

```
x ~/project master ✘1> git cherry-pick --quit
```

при конфликте редактор также предложит исправить. Затем добавляем в индекс и continue.

Завершенный cherry-pick тоже можно отменить:  
жестким ресетом откатиться на нужное количество назад ( в данном случае 2 коммита)

```
~/project master> git reset --hard @~2
```

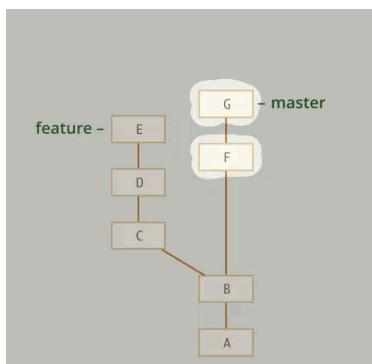
Бывает что мы хотим скопировать не один в один, а немного подредактировать - флаг -n - с ним cherry-pick не будет коммитить, только добавит изменения в рабочую директорию и индекс. Можно что-то поправить и тогда уже сделать коммит

```
~/project master> git cherry-pick -n
```

## GIT REBASE

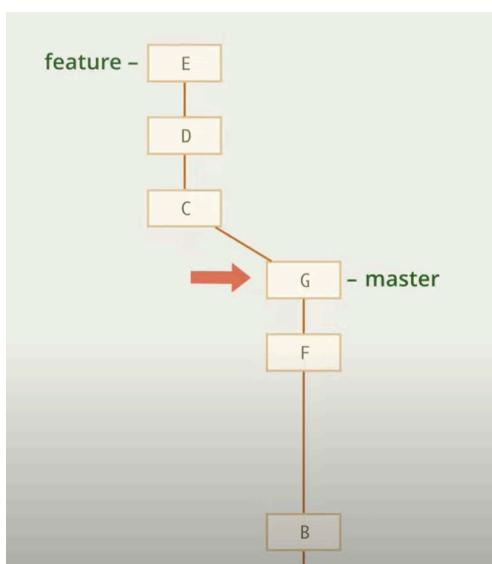
у этой команды много функций: можно переписывать историю, объединять и редактировать и менять местами коммиты.

Базовое применение - **перебазирование веток**: например мы очень хотим использовать на feature коммиты G и F, которые есть с мастер после ответвления. Можно произвести слияние веток, а можно

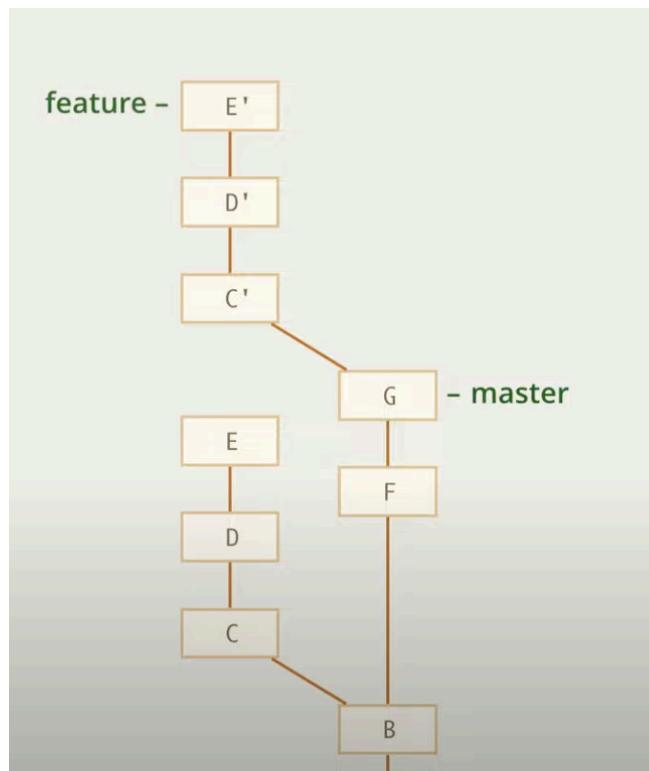


а можно перенести ветку:

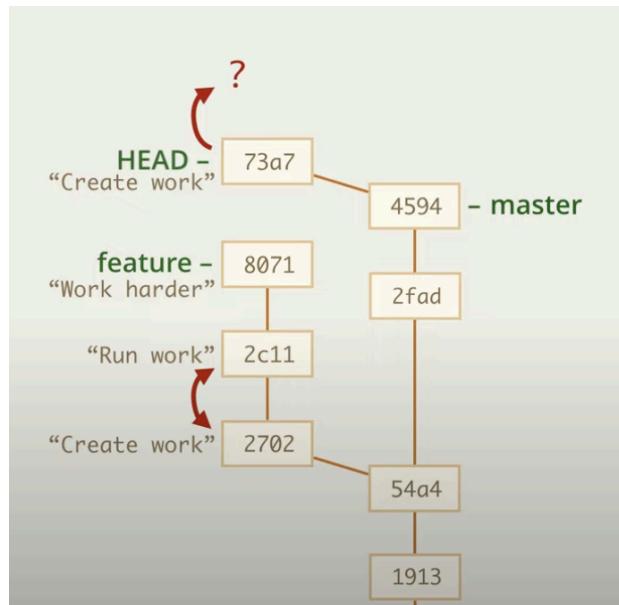
```
~/project feature> git rebase master
```



теперь как будто мы начали разработку feature с коммита G, а не В если разбираться как работает это, то значала гит как бы поочередно копирует коммиты поверх вершины мастер как cherry-pick, затем ссылка feature переносится на новосозданную цепочку коммитов:



это была теория, теперь рассмотрим на нашем реальном примере:



как и раньше у нас возникнет конфликт при переносе второго коммита:

```
~/project feature> git rebase master
First, rewinding head to replay your work on top of it...
Applying: Create work
Applying: Run work
Using index info to reconstruct a base tree...
M     index.html
Falling back to patching base and 3-way merge...
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
error: Failed to merge in the changes.
Patch failed at 0002 Run work
Use 'git am --show-current-patch' to see the failed patch

Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git
rebase --abort".
```

сейчас мы находимся в состоянии отдаленной HEAD - ссылка HEAD указывает на вершину скопированных коммитов, а не на какую-то ветку

в данном случае командой git reset --hard почистить не получится:

```
x ~/project :73a759f x1> git reset --hard
```

она удалит незакоммиченные изменения, удалит конфликт, но ссылку HEAD не передвинет

Поэтому если мы захотим отказаться от перебазирования, то нужно вызвать команду:

```
x ~/project :73a759f x1> git rebase --abort
```

Эта команда удалит служебную инфу о прогрессе rebase и вернет HEAD обратно на feature.

еще есть такая команда - она также прекращает перебазирование, удаляет служебную инфу, но не возвращает обратно HEAD

```
x ~/project :73a759f x1> git rebase --quit
```

Можно проигнорировать один коммит и тогда rebase продолжит копирование со следующего, а конфликтный пропустит.

```
x ~/project :73a759f x1> git rebase --skip
```

Возможна ситуация, что копируемый коммит не вносит никаких изменений (то есть в разных коммитах независимо исправлена одна и та же ошибка). Такие коммиты rebase пропускает (они не дают изменения и считаются пустыми).

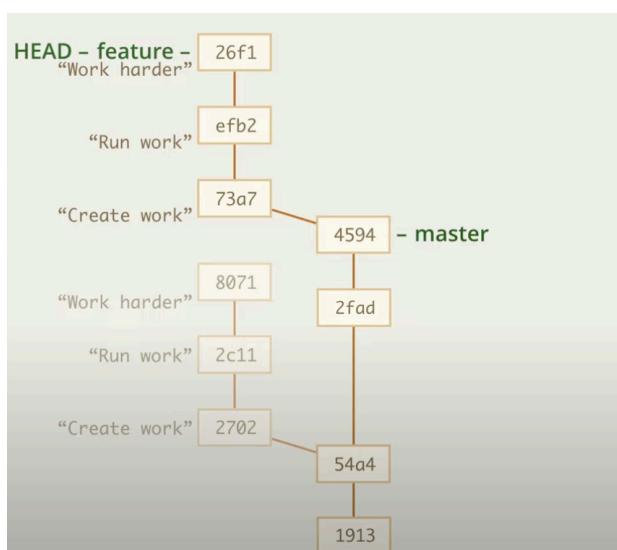
продолжить перебазирование:

```
x ~/project :73a759f x1> git rebase --continue
```

Разрешаем конфликт как раньше (на одном и том же месте в разных файлах были разные вызовы функций, отредактируем вызов этих функций друг за другом)

Проиндексируем и запустим continue

```
~/project :73a759f •1> git rebase --continue
Applying: Run work
Applying: Work harder!
```



Отменить после успешного перебазирования: просто перенести указатель feature обратно (коммит куда перенести хранится в ORIG\_HEAD):

```
~/project feature> cat .git/ORIG_HEAD  
807167a56a61bd1950cfab6cc2786abd2fa49c43
```

```
~/project feature> git reset --hard ORIG_HEAD
```

Однако ORIG\_HEAD не очень надежна в данном случае, так как она много раз перебазируется и rebase не гарантирует в итоге ее правильность. Лучше посмотреть идентификатор через reflog причем именно ветки рефлог а не HEAD, так как HEAD переносится много раз от коммита к коммиту, а ветка один раз в конце

```
~/project feature> git reflog feature
```

```
~/project feature> git reflog feature -1  
26f1b5d (HEAD -> feature) feature@{0}: rebase finished: refs/heads/feature onto 4594f10bab02bdf034e0fab8d57b5bc09fb21594
```

```
~/project feature> git show --quiet feature@{1}  
commit 807167a56a61bd1950cfab6cc2786abd2fa49c43  
Author: Ilya Kantor <iliakan@gmail.com>  
Date:   Sun Oct 29 21:57:26 2017 +0300
```

Work harder!

```
~/project feature> git reset --hard feature@{1}
```

git rebase <ветка на которую нужно сделать перебазирование>

```
~/project feature> git rebase master
```

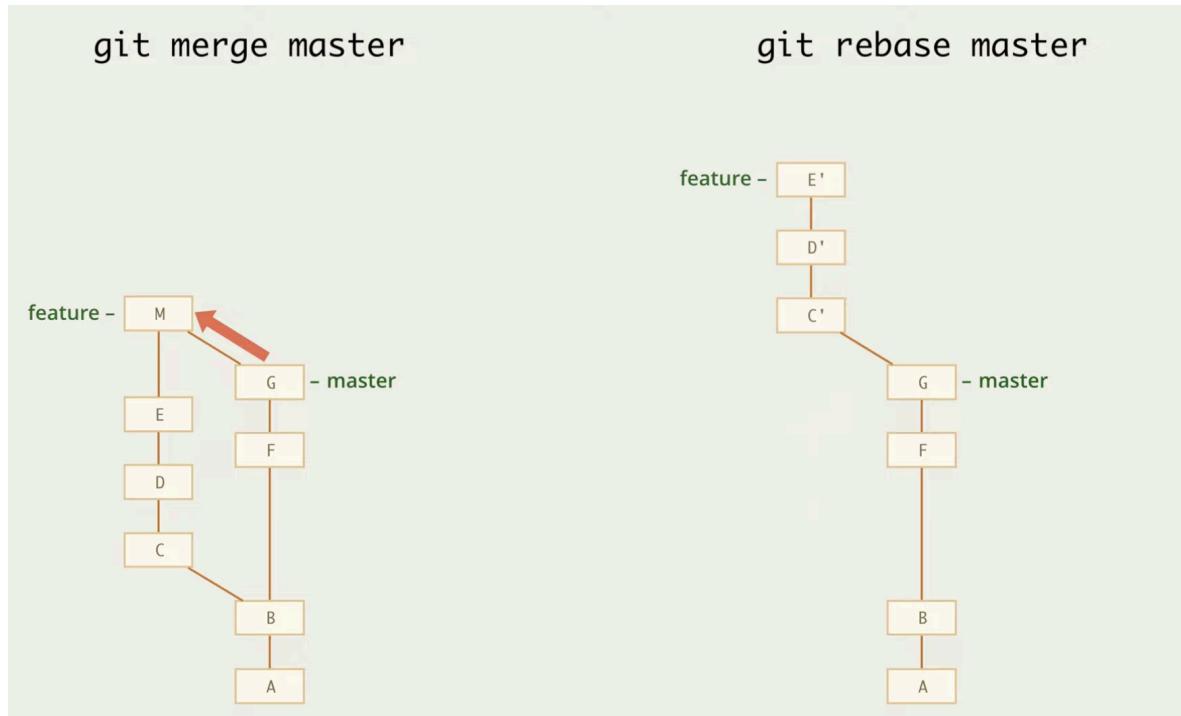
можно указать две ветки - это означает перебазировать feature на master

```
~/project feature> git rebase master feature
```

порядок можно забыть, поэтому лучше перейти на ветку и перебазировать как в первом случае

```
~/project feature> git rebase master feature = git checkout feature + git rebase master
```

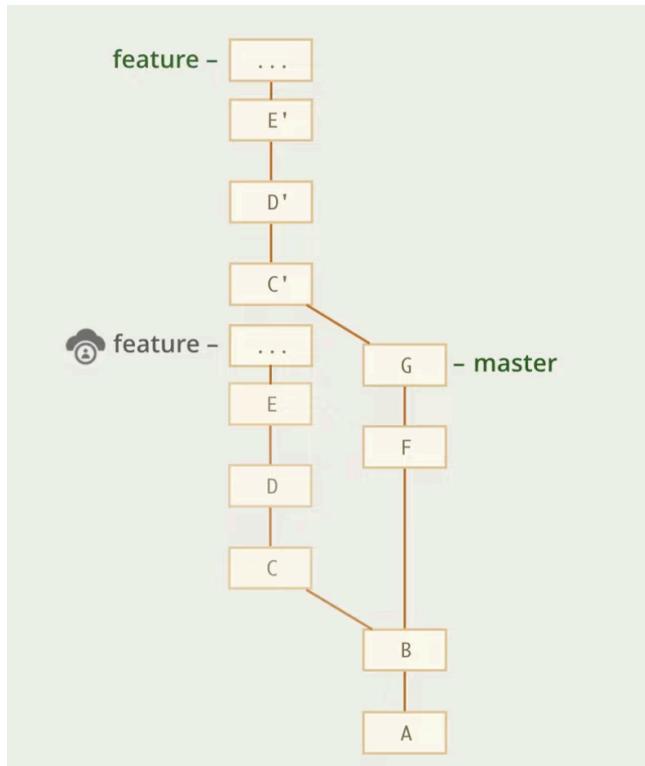
## СРАВНЕНИЕ MERGE и REBASE



результат одинаковый, но способы абсолютно разные

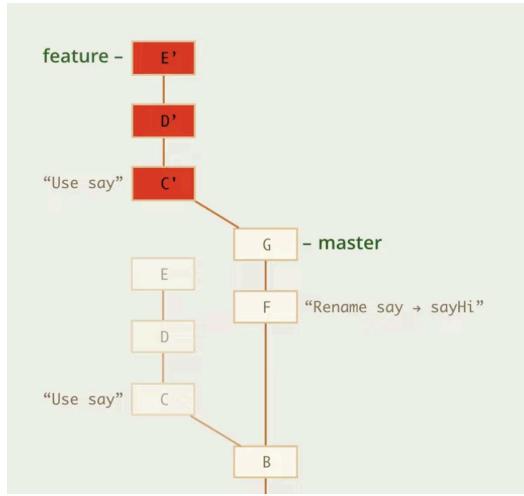
Какой же способ лучше?

+плюс перебазирования - это упрощение истории разработки  
-минус перебазирования - если с веткой работают несколько человек, то второй человек будет удивлен, особенно если он не заметил перебазирование и продолжил коммитить старую ветку.  
Восстановиться конечно можно из такой ситуации, но не приятно



если ветка не только у меня на компьютере, то можно, если она публичная и с ней работают несколько человек, то rebase делать нельзя и любые изменения истории

-по сути rebase это обман. Бывает, что коммит С хороший, а при наложении изменений С' уже сломан, при этом конфликта может даже не быть. То есть например в мастер мы переименовали функцию, а разработчик в feature вызвал эту функцию под старым именем. До перебазирования коммит со старым именем работал, а после уже не работает. А само перебазирования может успешно пройти, так как это не будет конфликтом. Причем скорее всего тогда и все коммиты дальше будут сломаны.



конечно можно сделать еще один коммит и исправить, но получится пачка битых коммитов и будет очень плохо, если мы когда откатимся на эти битые коммиты

При этом при слиянии такой проблемы не существует, если коммит слияния вдруг сломан, то сломан только он, а не коммиты feature, что были до него. И при исправлении мы будем исправлять этот один коммит слияния.

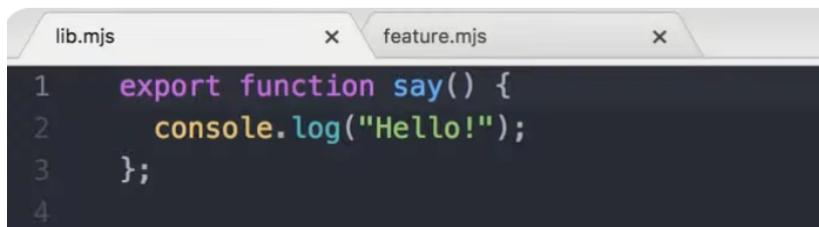
Тут помогут автоматизированные тесты:

```
git rebase -x '...' master
```

ключ -x позволяет указать произвольную команду, обычно это запуск тестов, который будет выполняться после каждого перебазированного коммита. И если тесты упадут, то они завершатся с кодом отличным от нуля, перебазировании остановится, а мы сможем исправить проблемный коммит.

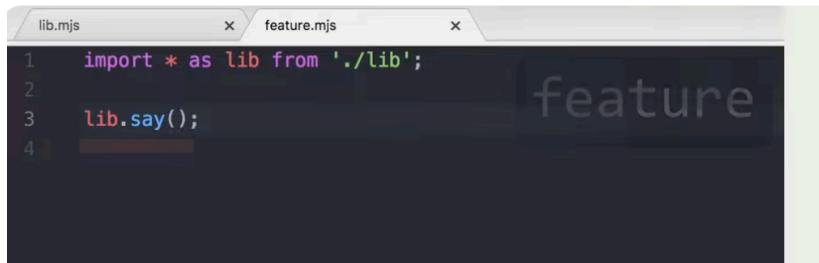
Пример:

файл тестирования функции say():



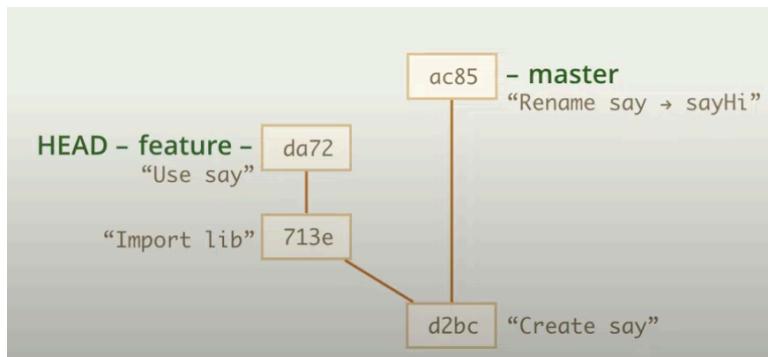
```
lib.mjs
1  export function say() {
2      console.log("Hello!");
3  };
4

feature.mjs
1  import * as lib from './lib';
2
3  lib.say();
4
```



```
feature
```

в master мы переименовали эту функцию на sayHi(), а в feature вызываем как hi()

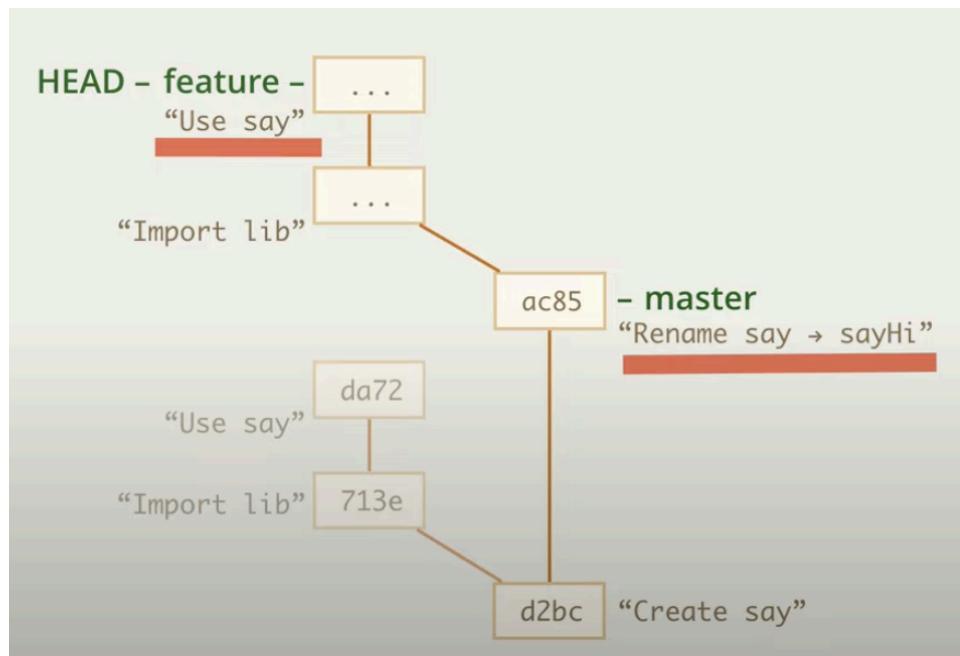


Протестируем, все окей, сделали перебазирование, оно прошло успешно, протестируем опять и ошибка, так как вызываем функцию как say(), а она переименована на sayHi()

```
~/project feature> node feature.mjs
Hello!

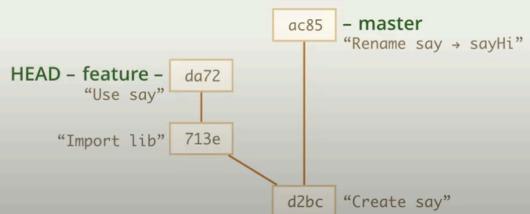
~/project feature> git rebase master
First, rewinding head to replay your work on top of it...
Applying: Import lib
Applying: Use say

~/project feature> node feature.mjs
TypeError: lib.say is not a function
    at file:///Users/learn/project/feature.mjs:3:5
    at ModuleJob.run (internal/modules/esm/ModuleJob.js:106:14)
    at <anonymous>
```



## отменяем перебазирование

```
x ~/project feature> git reset --hard ORIG_HEAD
```



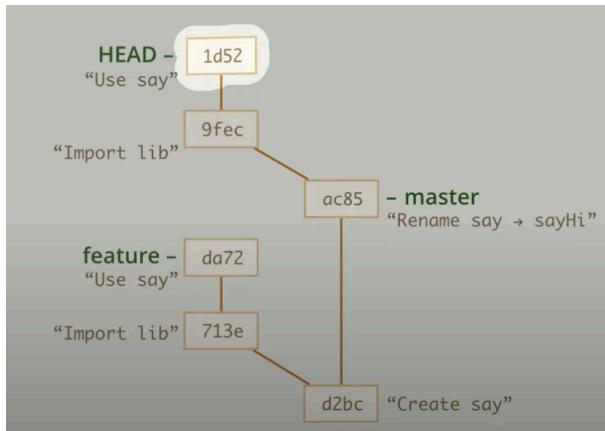
## сделаем его заново:

```
~/project feature> git rebase -x 'node feature.mjs' master
Executing: node feature.mjs
Executing: node feature.mjs
TypeError: lib.say is not a function
  at file:///Users/learn/project/feature.mjs:3:5
  at ModuleJob.run (internal/modules/esm/ModuleJob.js:106:14)
  at <anonymous>
warning: execution failed: node feature.mjs
You can fix the problem, and then run

  git rebase --continue
```

rebase вначале передвинул HEAD, затем запустил команду и первый коммит скопировался успешно, затем стала копировать второй коммит и ошибка. И теперь rebase ждет нашего решения.

Важно!!! коммит с ошибкой уже сохранен



поэтому понадобится не только исправить ошибку, но и этот ошибочный коммит заменить

изменим имя функции и проверим починился ли проект:

```
✗ ~/project :1d526e9> node feature.mjs
Hello!
```

затем добавим в индекс и заменим последний коммит:

```
~/project :1d526e9 +1> git add feature.mjs

~/project :1d526e9 •1> git commit --amend --no-edit
[detached HEAD ec2c76a] Use say
Date: Sat Apr 7 17:03:41 2018 +0300
1 file changed, 2 insertions(+)
```

продолжим перебазирование

```
~/project :ec2c76a> git rebase --continue
Successfully rebased and updated refs/heads/feature.
```

так как больше нет коммитов для перебазирования, то процесс завершился

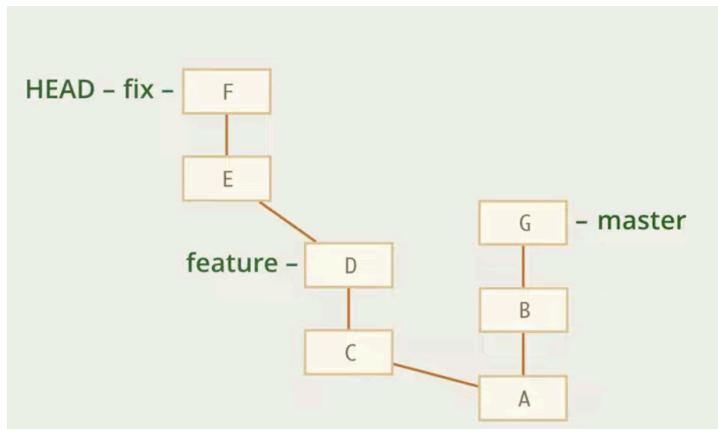
итог rebase:

Упрощает историю  
Только для приватных веток  
Возможны ошибки в коммитах

при этом слияние не всегда заменит rebase: если сделали коммиты не на той ветке и хотим перенести, то только rebase поможет а не слияние

---

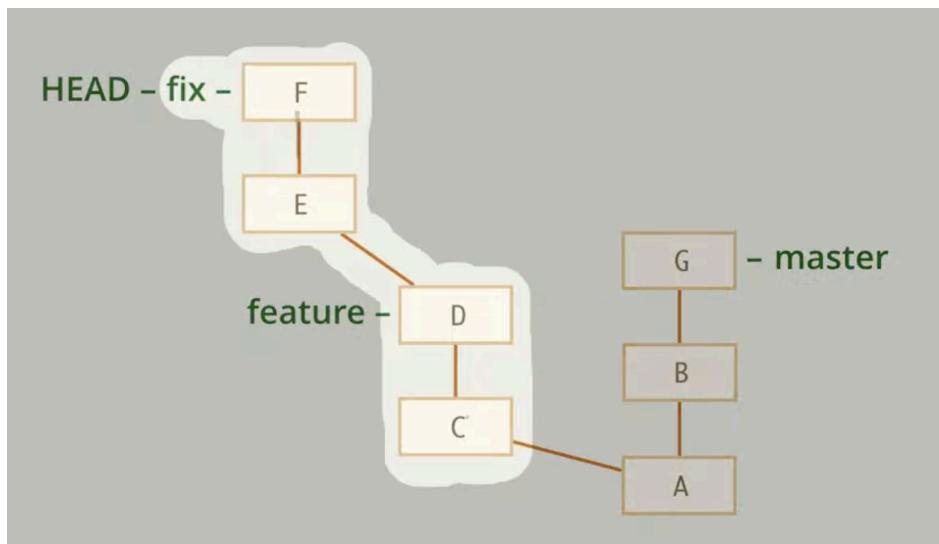
Бывает, что новые коммиты создаем не там, где нужно. Например fix создали на feature, а надо было на master и теперь хотим ее перенести



но если вы вызовем так:

```
~/project fix> git rebase master
```

то возьмутся коммиты такие (все коммиты с момента отхождения от мастер, а нам такое не нужно):

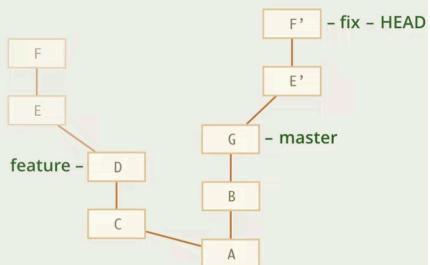


поможет флаг --onto:

```
~/project fix> git rebase --onto master feature
```

указываем его перед веткой, на которую нужно перебазировать, а после ветки пишем с какого момента (в нашем случае с feature)

```
~/project fix> git rebase --onto master feature
First, rewinding head to replay your work on top of it...
Applying: E
Applying: F
~/project fix>
```

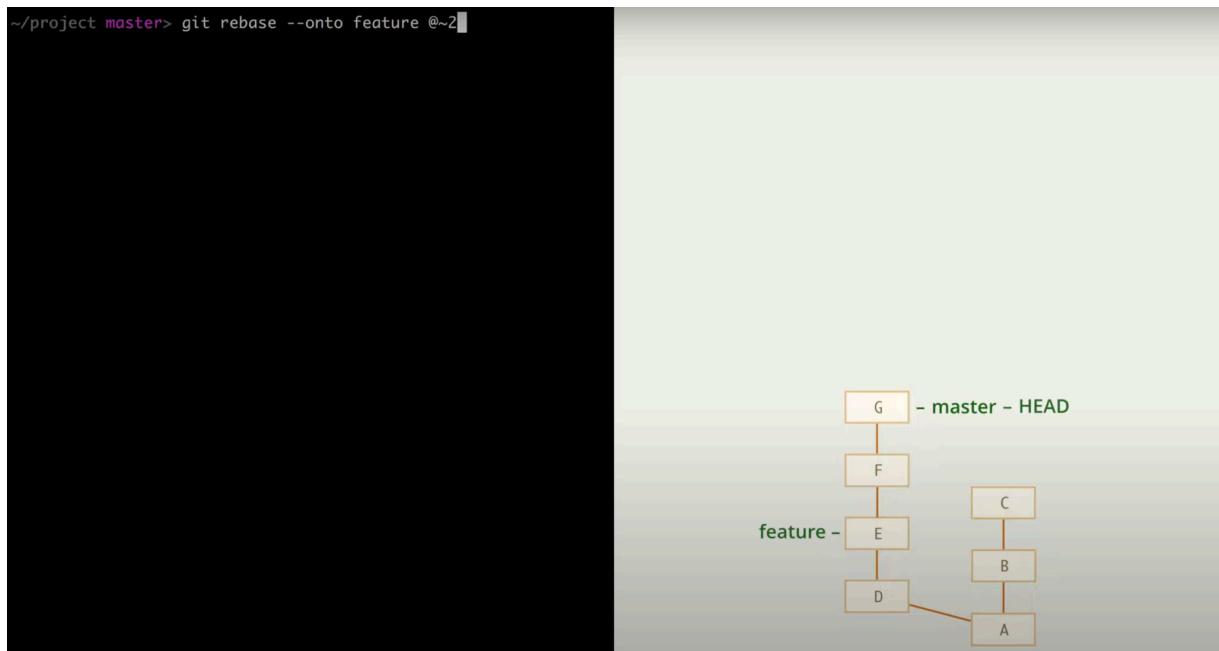


Также такую запись можно:

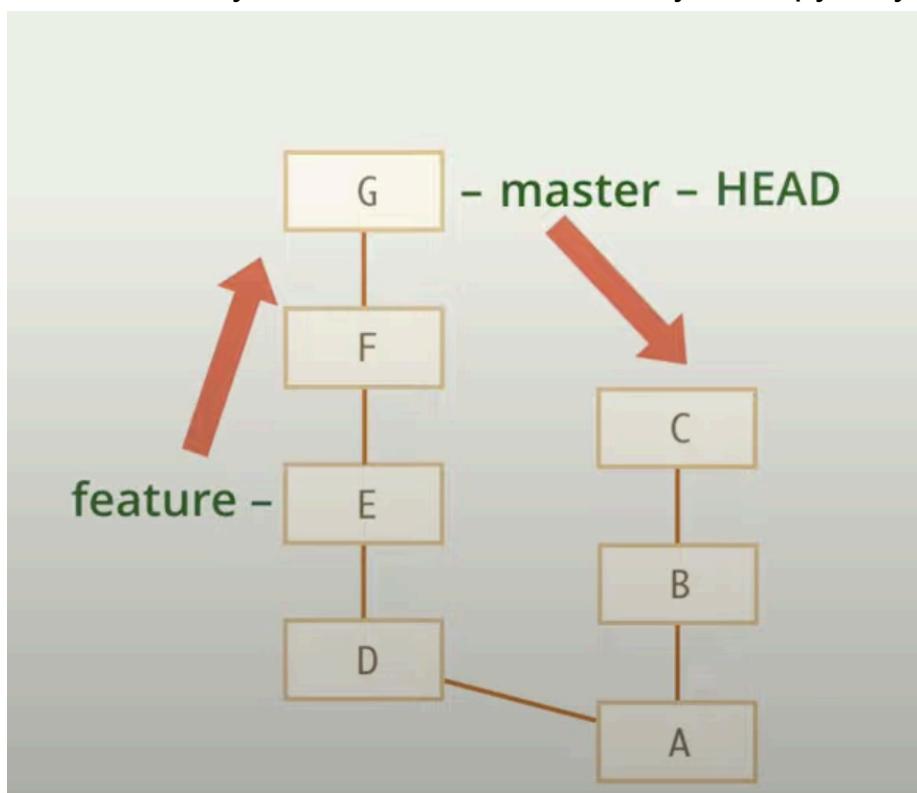
```
~/project fix> git rebase --onto master feature fix = checkout
fix + rebase --onto master feature
```

Можно для переноса пользоваться командой `cherry-pick`

Важно отличие: `rebase` переносит не только коммиты (G, F), но и ветку



НО по логике указатели должны быть тут по-другому



cherry-pick:

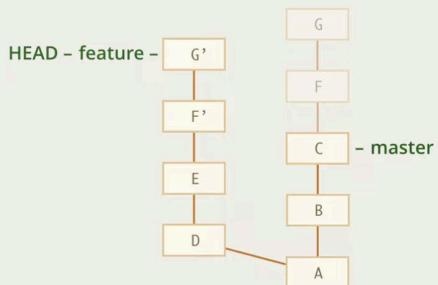
```
~/project master> git checkout feature
Switched to branch 'feature'

~/project feature> git cherry-pick master~2..master
[feature 9056cd4] F
 Date: Sun Apr 29 11:58:35 2018 +0300
 1 file changed, 2 insertions(+)

[feature 388e382] G
 Date: Sun Apr 29 11:58:36 2018 +0300
 1 file changed, 2 insertions(+)

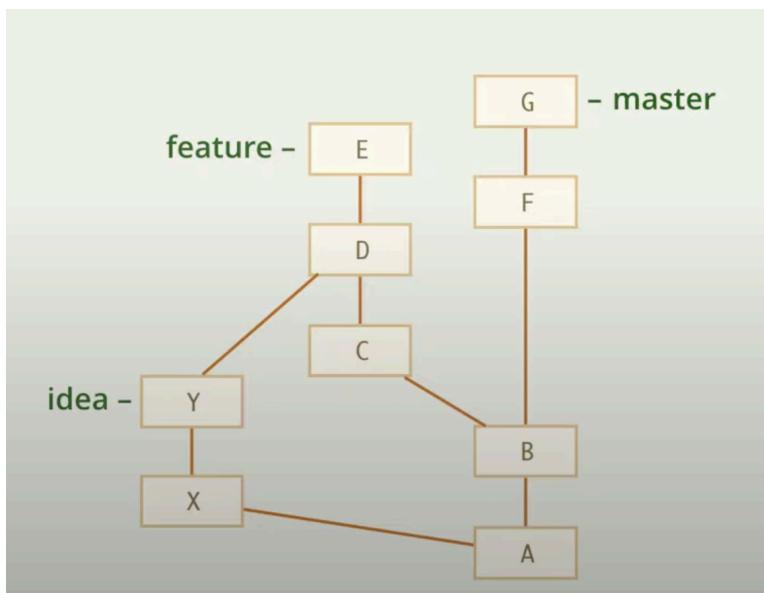
~/project feature> git branch -f master master~2

~/project feature> █
```

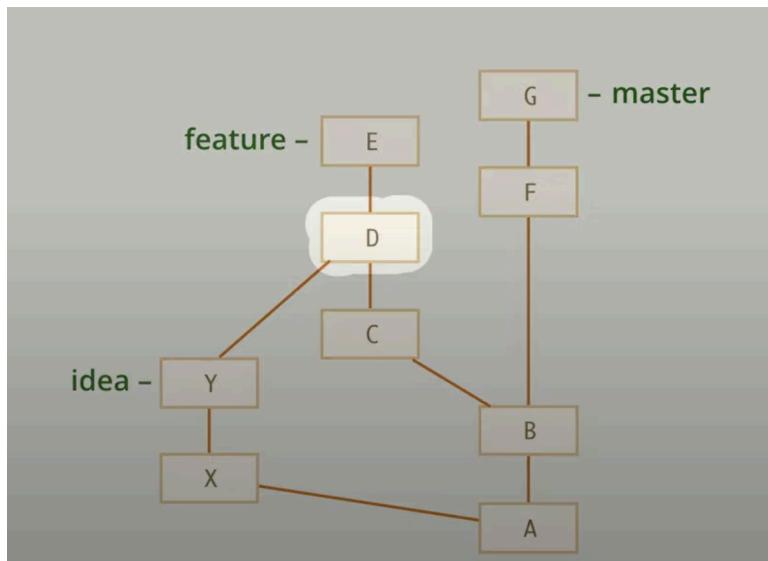


главное помнить различия этих способов и тогда они могут быть взаимозаменяемы

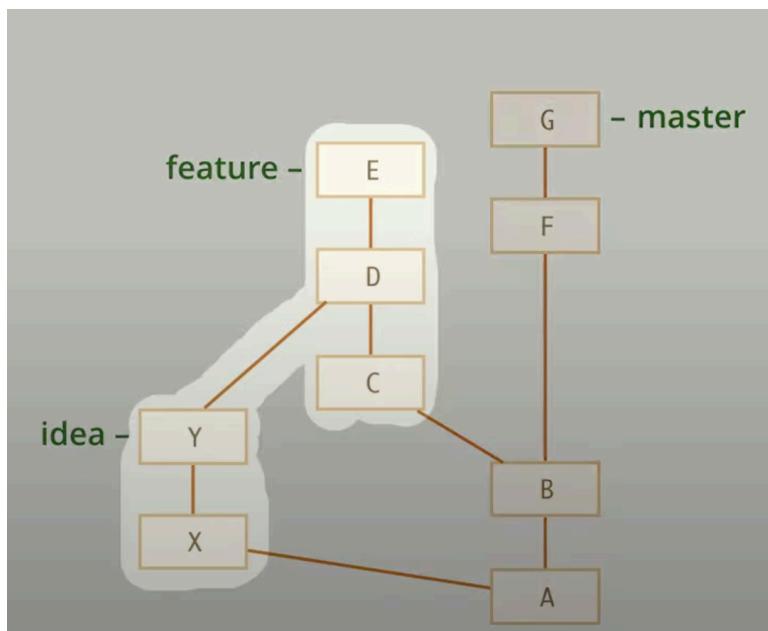
Рассмотрим более сложные варианты перебазирования, когда в ветке есть коммиты слияния: например здесь мы хотим перебазировать ветку feature, но в нее вмержена ветка idea

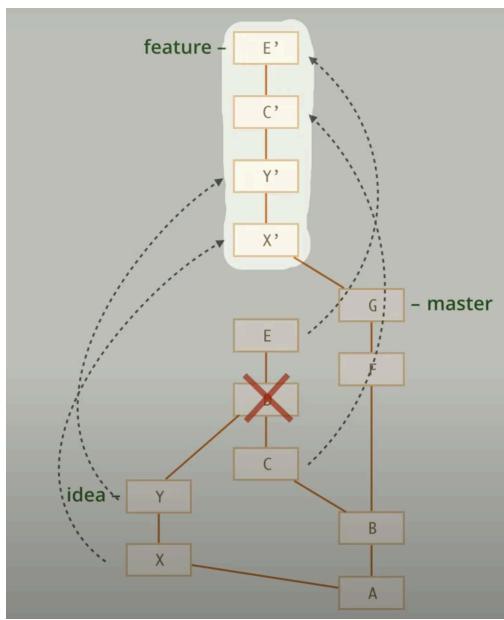


при таком перебазировании пропускаются коммиты слияния:



rebase сначала ищет все коммиты, которые есть в feature, но которых нет в master (ветке принадлежат все коммиты, которые являются предками ее вершины, в том числе и вторые родители при слиянии):

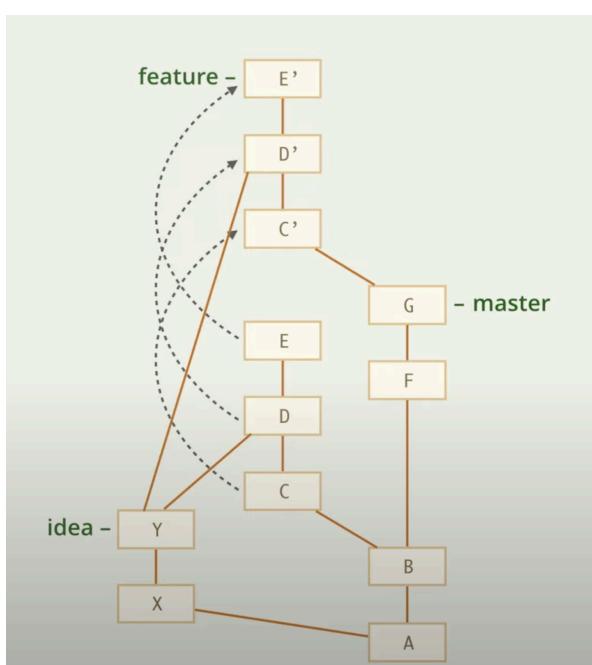




КОММИТЫ СЛИЯНИЯ пропускаются, получается линейная ветка. Если в коммите слияния мы вносили какие-то правки/изменения, то конечно это все не сохранится

с этим поможет флаг --rebase-merges:

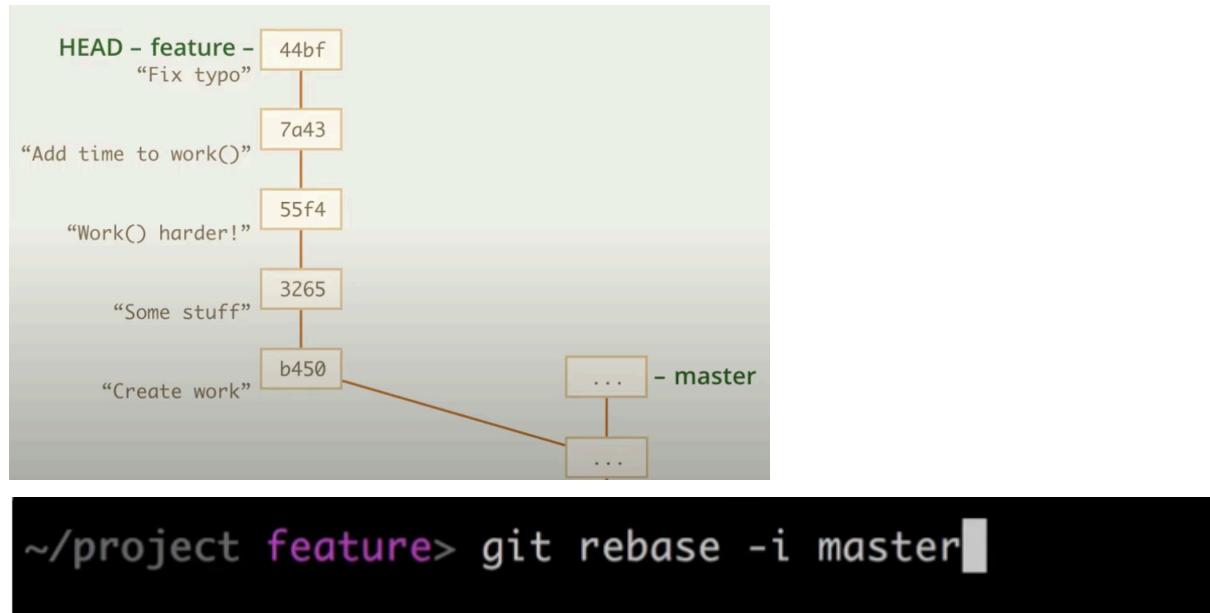
```
~/project feature> git rebase --rebase-merges master
Successfully rebased and updated refs/heads/feature.
```



КОММИТЫ ИЗ СТОРОННЕЙ ВЕТКИ не скопировались и была создана копия коммита слияния. И если был конфликт в D, то в D' он тоже скорее всего будет и его надо будет опять решить

Когда мы разрабатываем, наши коммиты не всегда оптимальны. Гит позволяет подредактировать историю с помощью так называемого интерактивного перебазирования (флаг `-i`). Обычно редактирование нужно перед публикацией ветки или чтобы показать главному разработчику и тд.

## Пример



сверху отображается список действий, который rebase хочет совершить <название действия> <идентификатор> <заголовок сообщения коммита>. Порядок такой, в каком rebase будет копировать.

```
git-rebase-todo
1 pick b4500d8 Create work
2 pick 3265b1a Some stuff
3 pick 55f49e7 Work() harder!
4 pick 7a43c0c Add time to work()
5 pick 44bf38d Fix typo
6
7 # Rebase eb9b9d3..44bf38d onto eb9b9d3 (5 commands)
8 #
9 # Commands:
10 # p, pick = use commit
11 # r, reword = use commit, but edit the commit message
12 # e, edit = use commit, but stop for amending
13 # s, squash = use commit, but meld into previous commit
14 # f, fixup = like "squash", but discard this commit's log message
15 # x, exec = run command (the rest of the line) using shell
16 # d, drop = remove commit
17 #
18 # These lines can be re-ordered; they are executed from top to bottom.
19 #
20 # If you remove a line here THAT COMMIT WILL BE LOST.
```

можно удалить коммит, тогда он скопирован не будет, можно поменять строки местами - изменится порядок копирования. Конечно, при перемещении или удалении коммитов, могут возникать конфликты.

Мы также можем указать rebase, какое именно действие выполнять (список приведен ниже):

pick - скопировать коммит

reword - скопировать коммит и изменить сообщение (отобразится потом редакторе, когда rebase будет копировать этот коммит)  
**(вызывается до копирования)**

edit - более сложное редактирование коммита, можно будет даже на 2 разбить**(вызывается после копирования)**

squash - изменение этого коммита слить с предыдущим коммитом и сообщения тоже объединить (то есть перед squash обязательно должен быть коммит, squash не может первым стоять)

fixup - изменение этого коммита слить с предыдущим, но сообщение отбросить

ехес - используется для добавления в этот список произвольных команд (то что мы пишем флага -x и он автоматически после каждого коммита ехес поставит)

drop - пропустить коммит (можно и от руки строку удалить)

вообще опасно удалять строки (можно случайно лишнего удалить), поэтому глобально можно поставить защиту от редактирования такого (только drop)

```
reword = use commit, but edit the commit message
edit = use commit, but stop for amending
squash = use commit, but merge previous commit
fixup = like squash, but discard this commit's log message
git config rebase.missingCommitsCheck warn/error
```

закрываем редактор и запустится перебазирование

появилось окошко reword, мы поменяли текст коммита.

Далее мы скопировали коммит, edit остановился и дает возможность исправить коммит:

```
~/project feature> git rebase -i master
[detached HEAD 4135e25] Create work()
 1 file changed, 3 insertions(+)
  create mode 100644 script.js
Stopped at 3265b1a... Some stuff
You can amend the commit now, with
```

```
git commit --amend
```

Once you are satisfied with your changes, run

```
git rebase --continue
```

мы решили разбить коммит на два:

- 1) передвинем HEAD на один коммит назад смешанным ресетом  
(сбрасывает индекс, но оставляет изменения в рабочей директории)
- 2) теперь закоммитим в 2 коммита

```
~/project :4135e25 +1...1> git add News.md

~/project :4135e25 •1+1> git commit -m 'Add News.md'
[detached HEAD 5a08ddb] Add News.md
 1 file changed, 1 insertion(+)
  create mode 100644 News.md

~/project :5a08ddb +1> git add index.html

~/project :5a08ddb •1> git commit -m 'Use work()'
[detached HEAD d5db80b] Use work()
 1 file changed, 4 insertions(+)
```

при желании можно даже поправить оставшийся список действий:

```
~/project :d5db80b> git rebase --edit-todo
```

```
git-rebase-todo
1 pick 55f49e7 Work() harder!
2 squash 7a43c0c Add time to work()
3 fixup 44bf38d Fix typo
4 #
5 # Commands:
6 # p, pick = use commit
7 # r, reword = use commit, but edit the commit message
8 # e, edit = use commit, but stop for amending
9 # s, squash = use commit, but meld into previous
10 # f, fixup = like "squash", but discard this commit if amending
11 # x, exec = run command (the rest of the line) under the rebase
12 # d, drop = remove commit
13 #
14 # These lines can be re-ordered; they are executed from top to bottom
15 #
16 # If you remove a line here THAT COMMIT WILL BE DROPPED!
17 #
18 # You are editing the todo file of an ongoing interactive rebase;
19 # To continue, close this editor window
```

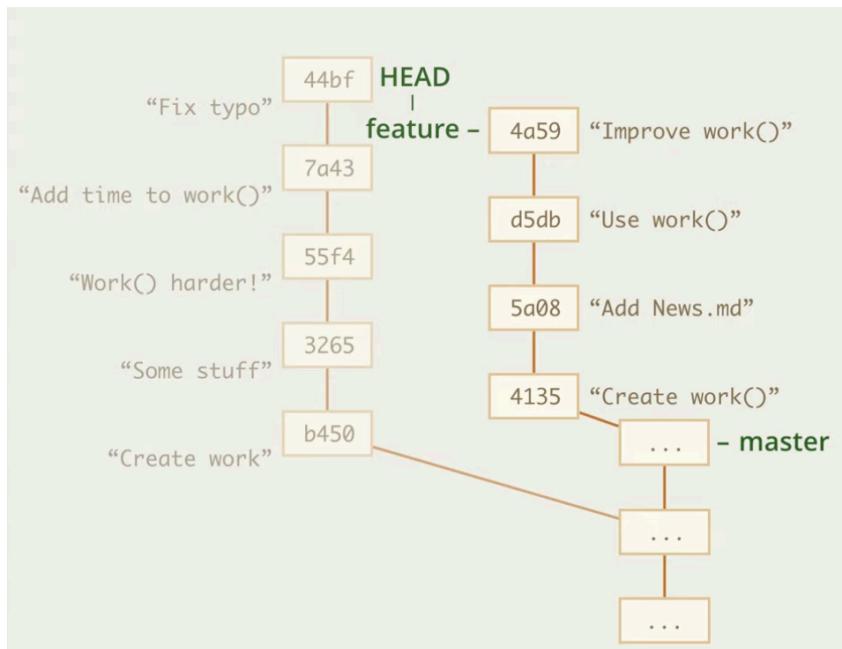
Продолжим перебазирование:

```
~/project :d5db80b> git rebase --continue
```

опять высвечивается редактор: rebase подготовил объединенный коммит и просит уточнить сообщения для него в качестве заготовки он берет сообщения всех трех коммитов (третий коммит закомментирован так как fixup отбрасывает сообщение)

```
git-rebase-todo
1 # This is a combination of 3 commits.
2 # This is the 1st commit message:
3
4 Work() harder!
5
6 # This is the commit message #2:
7
8 Add time to work()
9
10 # The commit message #3 will be skipped:
11
12 # Fix typo
13
14 # Please enter the commit message for your changes.
15 # with '#' will be ignored, and an empty message
16 #
17 # Date:      Sun May 27 15:38:17 2018 +0300
18 #
19 # interactive rebase in progress; onto eb9b9d3
20 # Last commands done (5 commands done):
21 #     squash 7a43c0c Add time to work()
22 #     fixup 44bf38d Fix typo
```

## ИТОГ:



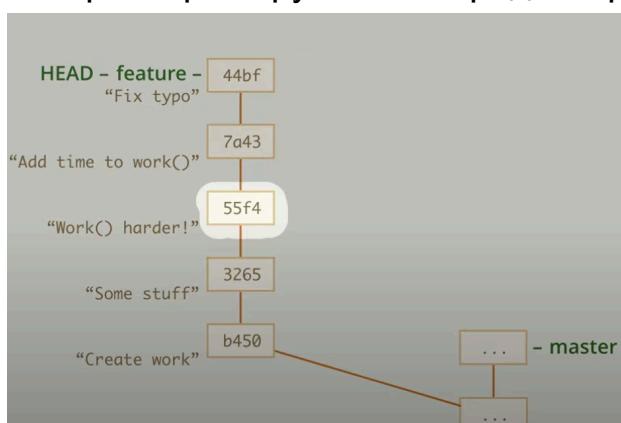
отменить перебазирование так в нашем случае не получится, так как мы делали reset (при разбивки коммита) и он ORIG\_HEAD перезаписывал

```
~/project feature> git reset --hard ORIG_HEAD
```

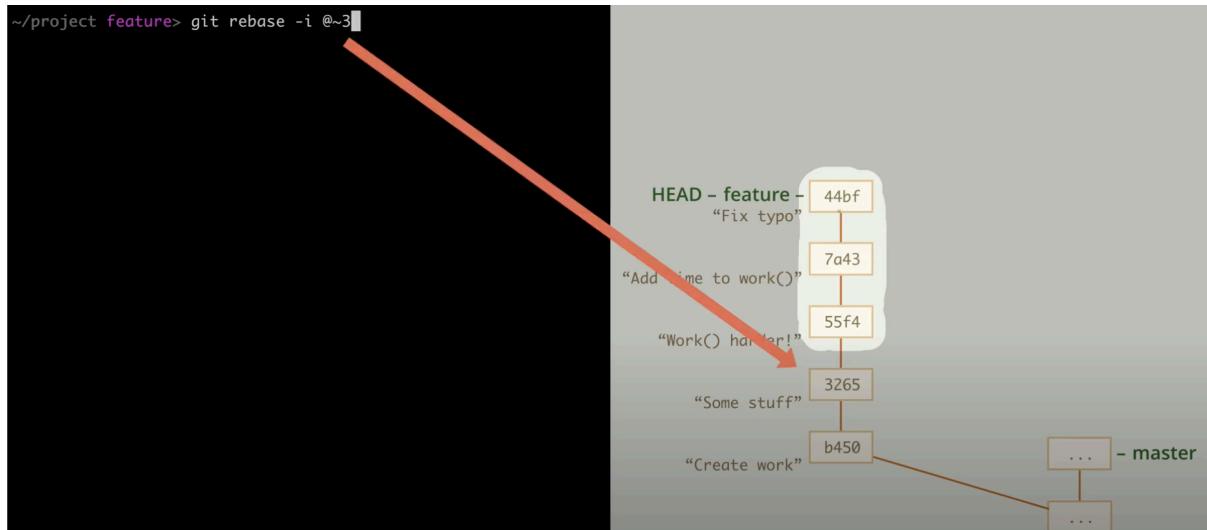
поэтому отмени через reflog:

```
~/project feature> git reset --hard feature@{1}
HEAD is now at 44bf38d Fix typo
```

Теперь к примеру хотим отредактировать коммит 55f4



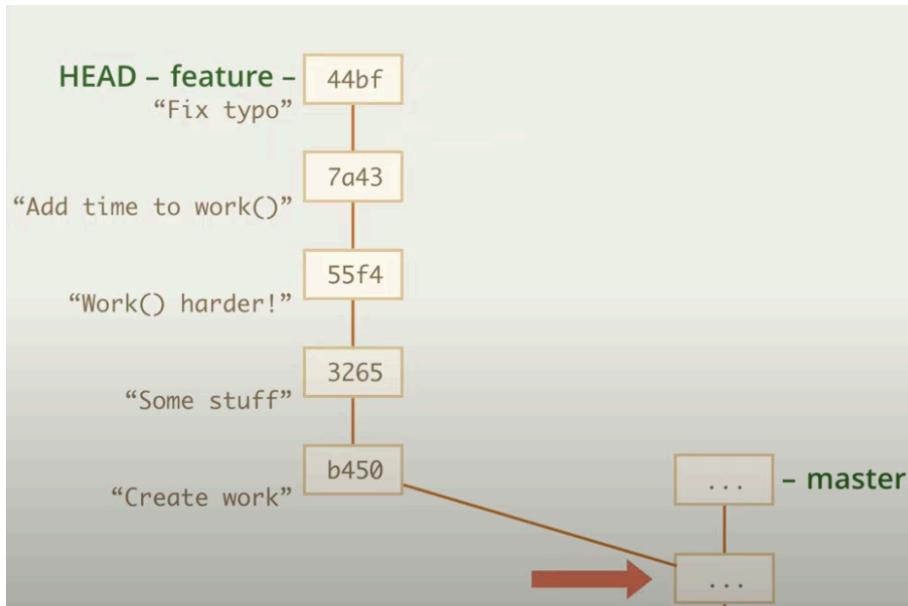
можно перебазировать ветку на саму себя, а именно на коммит который находится перед тем, на какой мы хотим изменить



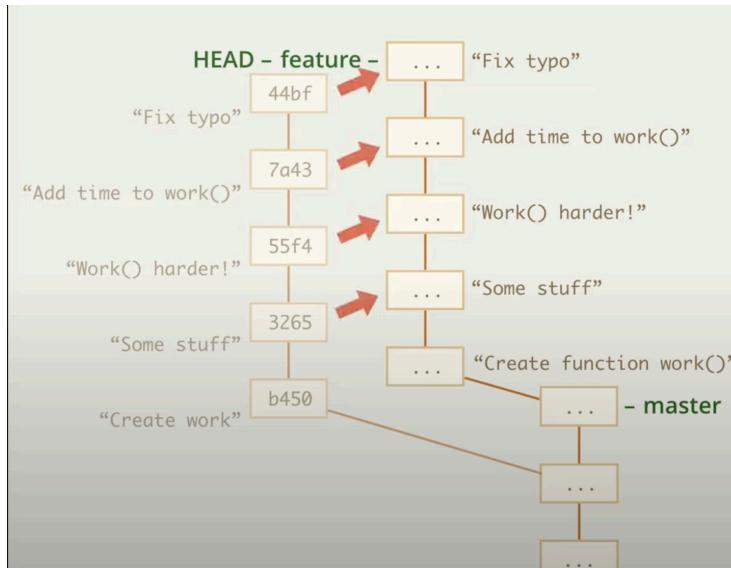
это означает что перемещению подлежат коммиты ветки feature с момента отхождения от точки перебазирования

и так как перебазирование интерактивное, то мы сможем сделать с коммитами 55f4 и выше все что захотим.

если мы хотим отредактировать историю всей ветки, то можно перебазировать на точку отхождения ветки от master

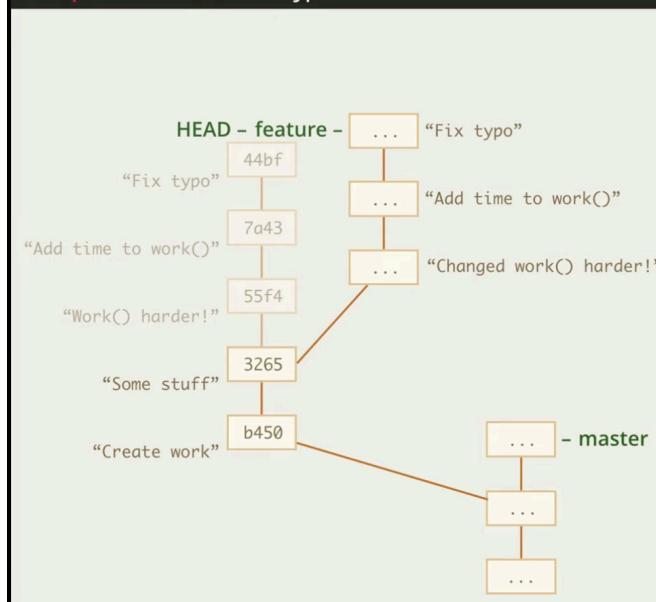


При этом нужно понимать, даже если мы меняем единственный коммит, то rebase копируют всю ветку начиная с этого измененного коммита (в данном случае мы поменяли только b450)



но если не трогать начальные коммиты ветки, а начать менять например с 55f4, то результат будет таким:

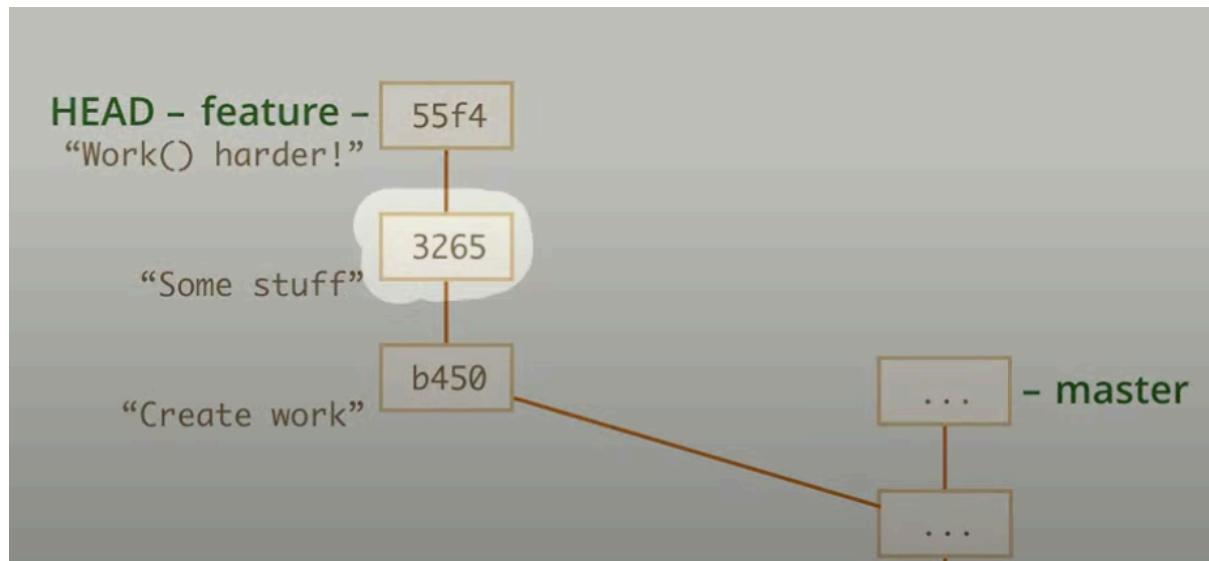
```
1 pick b450d8 Create work
2 pick 3265b1a Some stuff
3 edit 55f49e7 Work() harder!
4 pick 7a43c0c Add time to work()
5 pick 44bf38d Fix typo
```



Исправить последний коммит легко (amend), посередине ветки можно исправить интерактивным перебазированием и подредактировать коммит.

Но в rebase есть дополнительный механизм autosquash, который делает это еще удобнее

Например была допущена такая ошибка:



исправим файл News.md и создадим новый коммит:

A screenshot of a code editor showing a file named 'News.md'. The content of the file is:

```
1 Created script.js with simple work(), to be
* improved.
2
```

To the left of the editor is a 'Project' sidebar showing files: 'project', '.git', 'index.html', 'News.md' (which is selected), and 'script.js'.

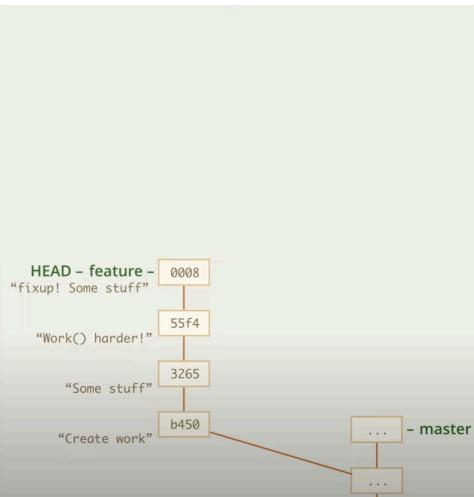
```
~/project feature +1> git commit -a --fixup=3265
```

Можно не через идентификатор:

```
~/project feature +1> git commit -a --fixup=@~
```

после этого вызовем git show (можно заметить что сообщение скопировано старого коммита, но перед ним стоит слово fixup)

```
~/project feature> git show  
commit 000860c887f45489dab2a961656943641044ad02 (HEAD -> feature  
e)  
Author: Ilya Kantor <iliakan@gmail.com>  
Date: Mon Jun 4 18:21:12 2018 +0300  
  
fixup! Some stuff  
  
diff --git a/News.md b/News.md  
index d300b42..f8ce8ab 100644  
--- a/News.md  
+++ b/News.md  
@@ -1 +1 @@  
-Created script.js with simple work, to be improved.  
+Created script.js with simple work(), to be improved.  
  
~/project feature>
```



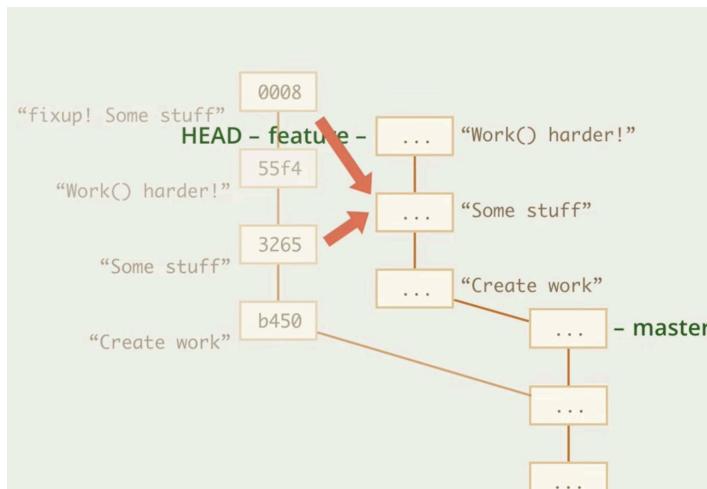
далее вызываем интерактивное перебазировани с опцией --autosquash

```
~/project feature> git rebase -i --autosquash
```

теперь исправленный коммит автоматически поставится после исправляемого коммита

```
1 pick d974724 Create function work()  
2 pick 312e6cb Some stuff  
3 fixup 000860c fixup! Some stuff  
4 pick 1592218 Work() harder!  
5  
6 # Rebase 63b8b29..000860c onto 63b8b29 (4 commands)  
7 #  
8 # Commands:
```

теперь когда закроем редактор, картина такая (два коммита объединены в один):



можно так для исправления:

```
~/project feature> git commit --fixup
```

или так:

```
~/project feature> git commit --squash
```

тогда действие с коммитом будет squash вместо fixup

## ОТМЕНА КОММИТОВ ЧЕРЕЗ REVERT

отменить коммиты можно и с помощью reset и с помощью rebase, но у двух этих приемов есть недостатки: если коммит уже отправлен коллегам, то его не так просто отменить. Здесь как раз будет полезна другая команда - git revert.



```
~/project master> git revert @
```

данная команда смотрит какие изменения сделаны в указанном коммите (в данном случае в текущем - @) и создает новый коммит с противоположными изменениями.

Вот в предыдущем коммите зеленым то что мы добавили

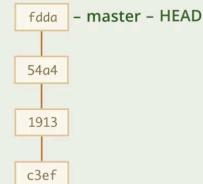
```
~/project master> git revert @
[master fddae83] Revert "Create sayBye"
 1 file changed, 4 deletions(-)

~/project master> git show @-
commit 54a4be6ff4ca63c909328ce4894c9ab0bc632c43
Author: Ilya Kantor <iliakan@gmail.com>
Date:   Tue Sep 12 15:38:14 2017 +0200

  Create sayBye

diff --git a/script.js b/script.js
index 0f67e66..935d66a 100644
--- a/script.js
+++ b/script.js
@@ -1,3 +1,7 @@
 function sayHi() {
   alert(`Hello from Git!`);
 }
+
+function sayBye() {
+  alert(`Goodbye from Git!`);
+}

~/project master> git
```



а вот после revert они красным удалены:

```
~/project master> git show
commit fddae8331b7f477662c486983db27d93876438e1 (HEAD -> master)
)
Author: Ilya Kantor <iliakan@gmail.com>
Date:   Mon May 7 14:01:57 2018 +0300

  Revert "Create sayBye"

    This reverts commit 54a4be6ff4ca63c909328ce4894c9ab0bc632c4
3.

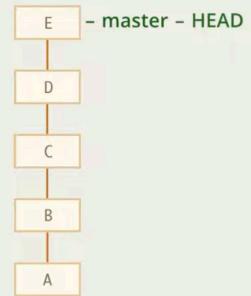
diff --git a/script.js b/script.js
index 935d66a..0f67e66 100644
--- a/script.js
+++ b/script.js
@@ -1,7 +1,3 @@
 function sayHi() {
   alert(`Hello from Git!`);
 }
-
-function sayBye() {
-  alert(`Goodbye from Git!`);
-}
```

конечно, не очень хорошо, когда в одном коммите изменения создаются, а в другом они же отменяются, это загрязняет историю, но в данном случае (если мы уже все-таки отправили этот коммит) это может быть единственным способом. Но если

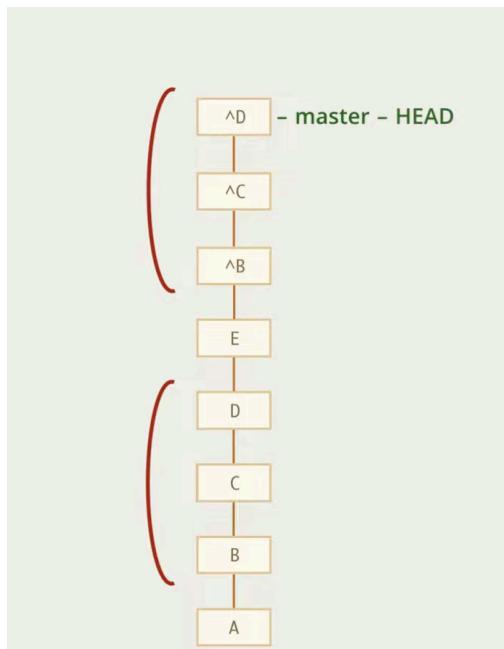
изменение было давно, могло быть как-то исправлено, то , конечно, может возникнуть конфликт. Разрешается обычным образом: поправили файл, проиндексировали и закоммитили.

можно обращать и диапазон коммитов:

```
~/project master> git revert A..D
```



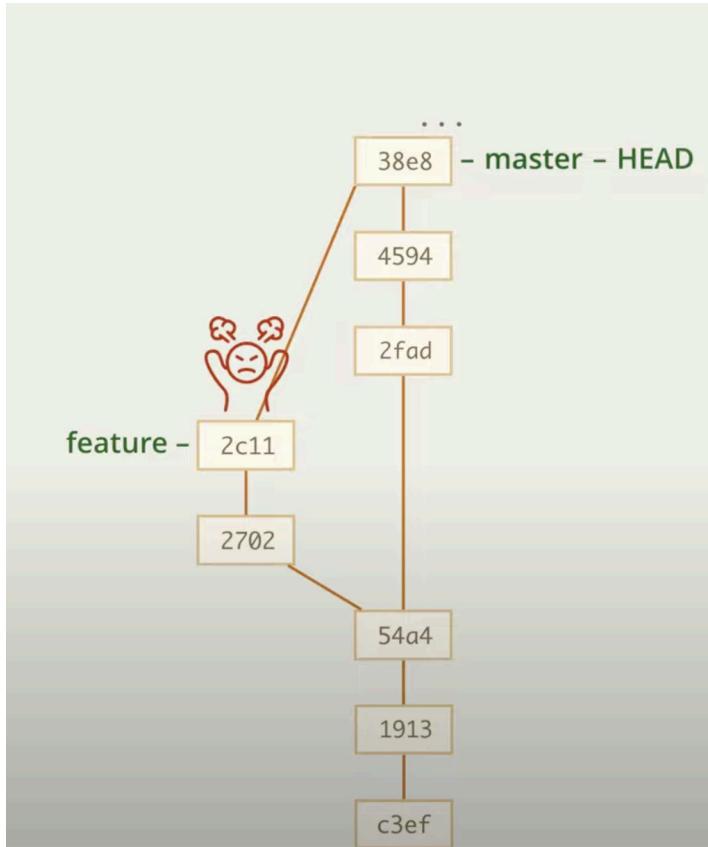
Внутри revert устроена как и cherry-pick, разница в том, что revert создают не копию коммита, а обратный коммит



у команды revert своих опций нет, но она поддерживает почти все флаги cherry-pick

С обращением обычного коммита все просто, куда интереснее ситуация со слиянием

например обнаружили после слияния:



reset отменить нельзя, если уже коммит стал публичным  
(отменит только у нас на компьютере?????)

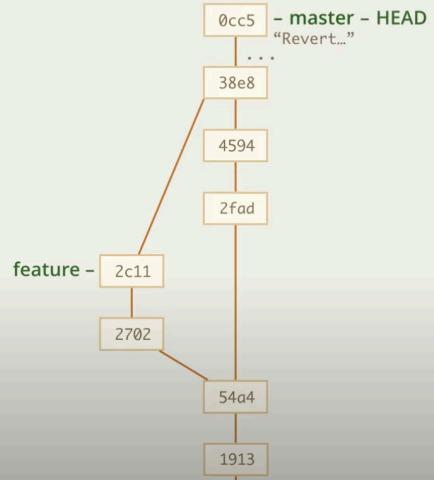
пробуем отменить, ОШИБКА, так как коммит слияния и по сравнению с разными родителями вносятся разные изменения

```
~/project master> git revert 38e8
error: commit 38e84f1b37991a14a1398d5c5c4aa35c899c6648 is a merge
but no -m option was given.
fatal: revert failed
```

поэтому нужно выбрать, какие изменения отменить, в данном случае по сравнения с feature (флаг -m <номер родителя>, основная ветка - это первый родитель, то что вливали - второй)

```
x ~/project master> git revert 38e8 -m 1
[master 0cc5bd8] Revert "Merge branch 'feature'"
 2 files changed, 5 deletions(-)

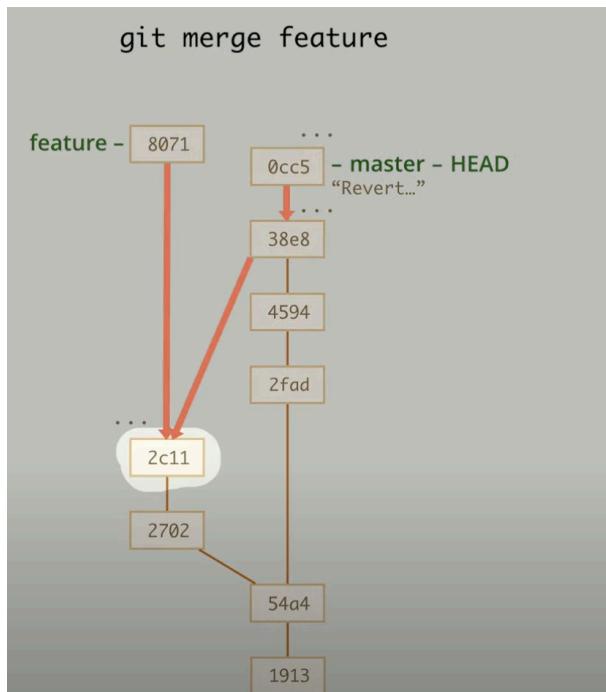
~/project master>
```



здесь должно быть -m 2

далее идет доработка master и feature, и вот feature доделана и нужно опять ее вливать

но если мы сейчас напишем “git merge feature”, то результат будет не ожидаем. Сначала гит поищет общего родителя, то есть 2c11, с точки зрения гит слияние уже было проведено раньше и теперь в master остается только добавить 8071, так как гит считает что коммит 2c11 и 2702 уже добавлены, но на самом деле мы отменили



скорее всего мы это заметим, так как слияние скорее всего не пройдет, будут конфликты, так как 8071 ожидает, что 2c11 и 2702 там есть, а их нет

есть несколько выходов:

скопируем изменения самостоятельно и затем уже мерджить feature

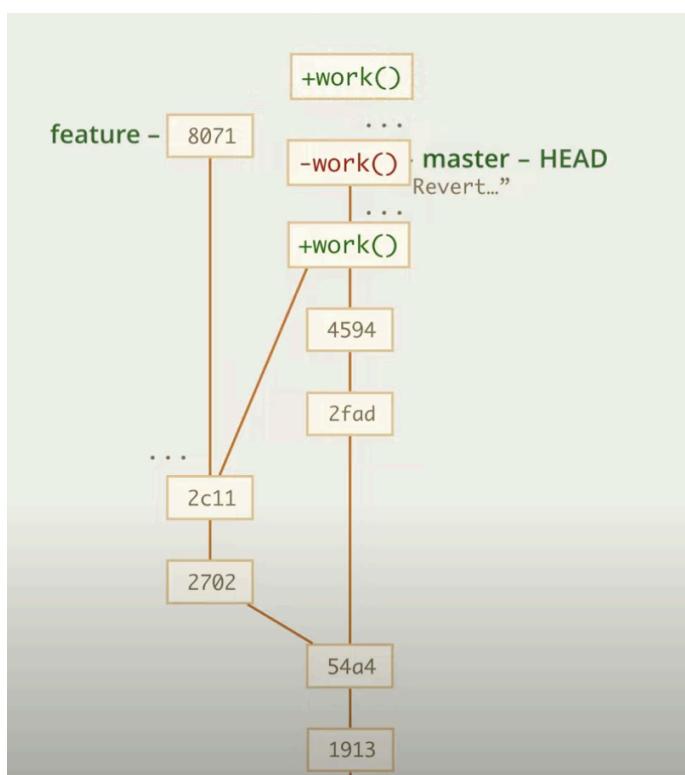
```
~/project master> git cherry-pick 2702 2c11
```

впрочем копировать два коммита нет необходимости, так как 38e8 содержит все эти изменения + если мы решали конфликт, то нам не придется его решать опять

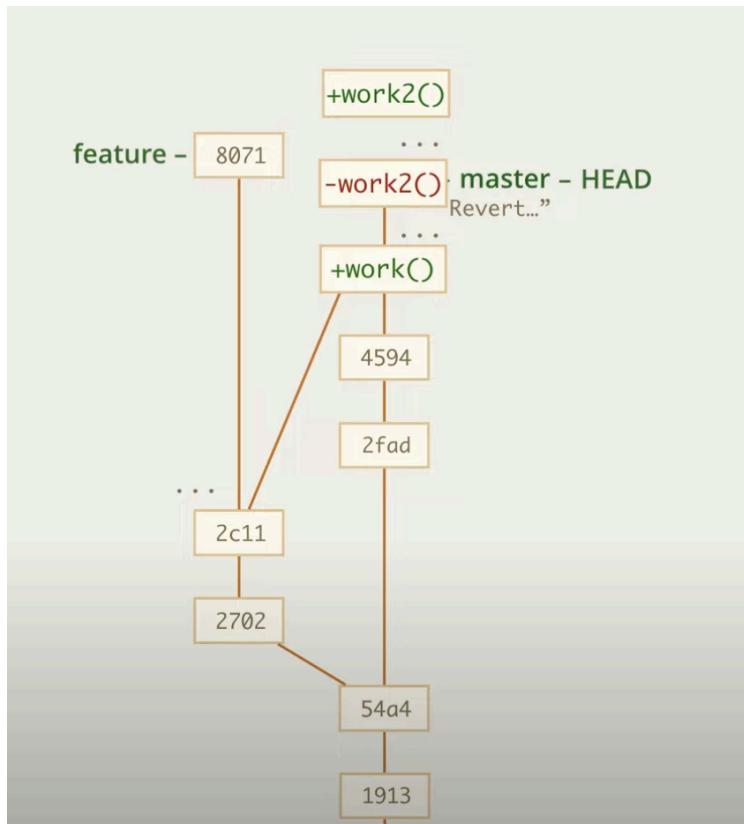
```
~/project master> git cherry-pick 38e8 -m 1
```

либо можно отменить коммит отмены

```
~/project master> git revert 0cc5
```



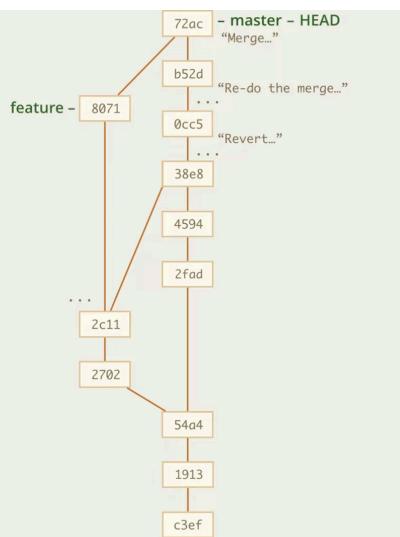
но на практике этот вариант надежнее, так как между коммитом слияния и отмены могли быть другие коммиты и мог возникнуть конфликт при вызове revert, который был разрешен и в итоге коммит отмены не является точной противоположностью слияния. И делая отмену отмены мы как раз это все учтем.



## Завершим слияние:

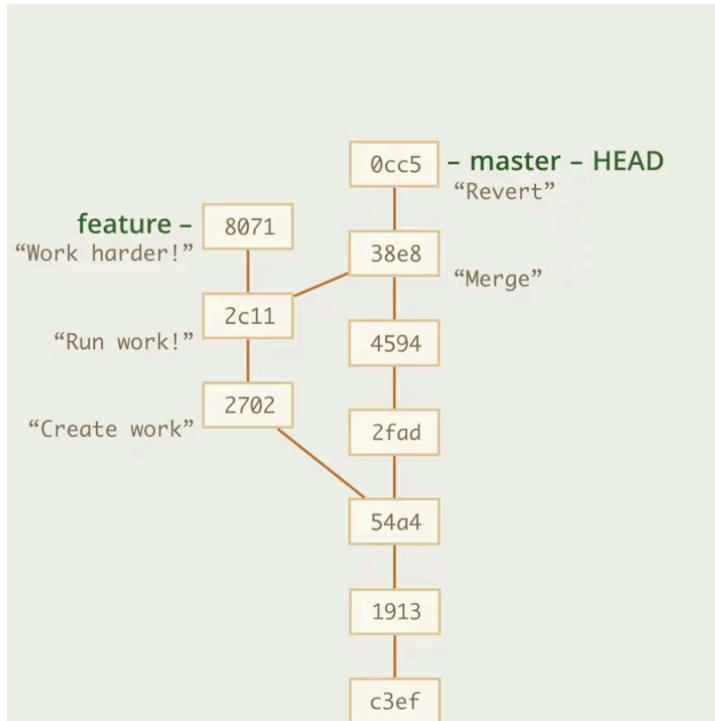
```

~/project master> git merge feature --no-edit
Merge made by the 'recursive' strategy.
script.js | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
~/project master>
  
```



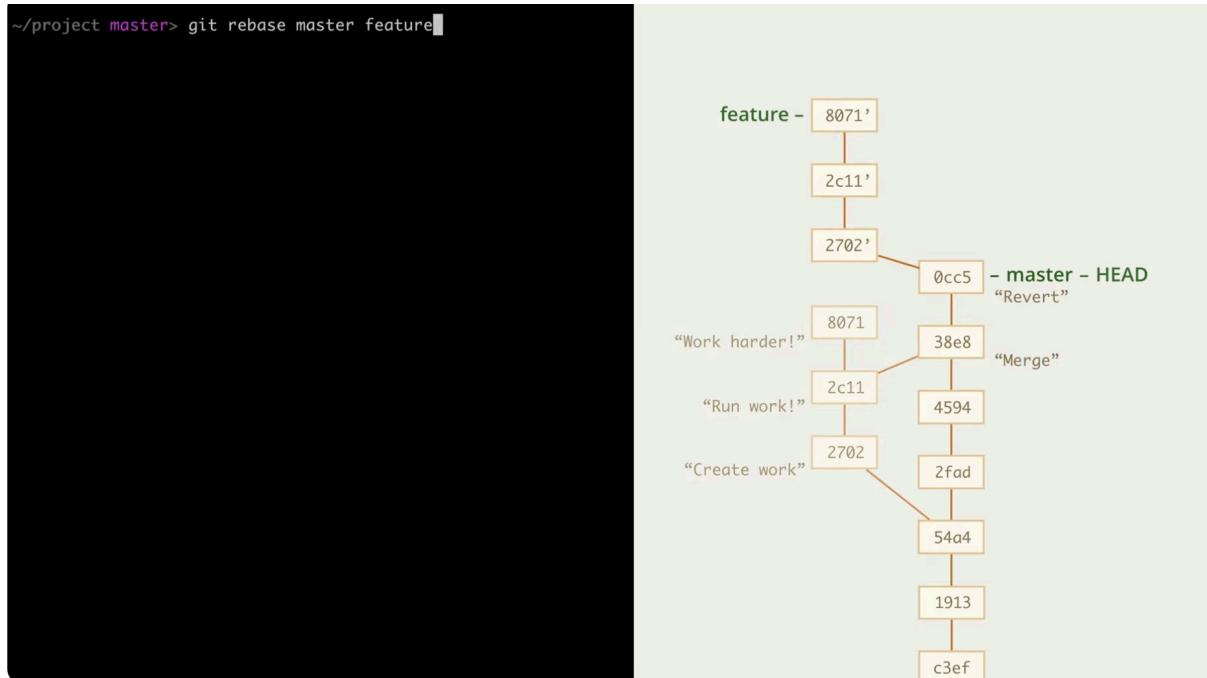
Отмена отмены конечно портит историю разработки

Рассмотрим то же самое другим способом:

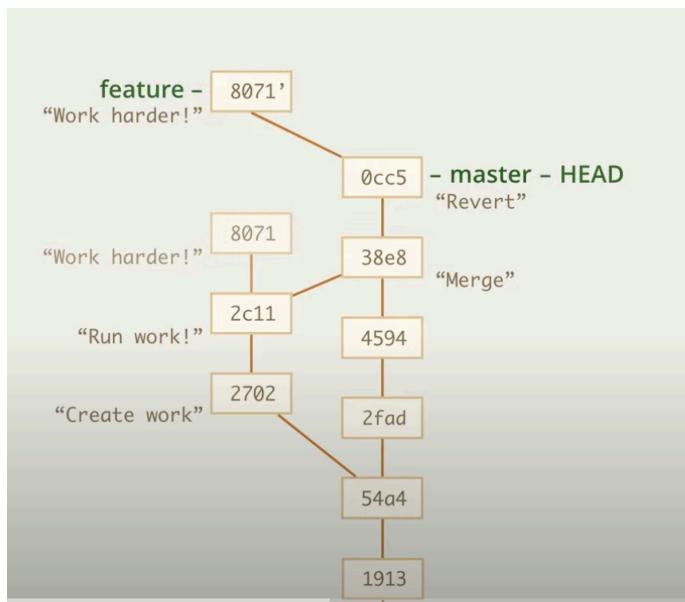


Перебазируем:

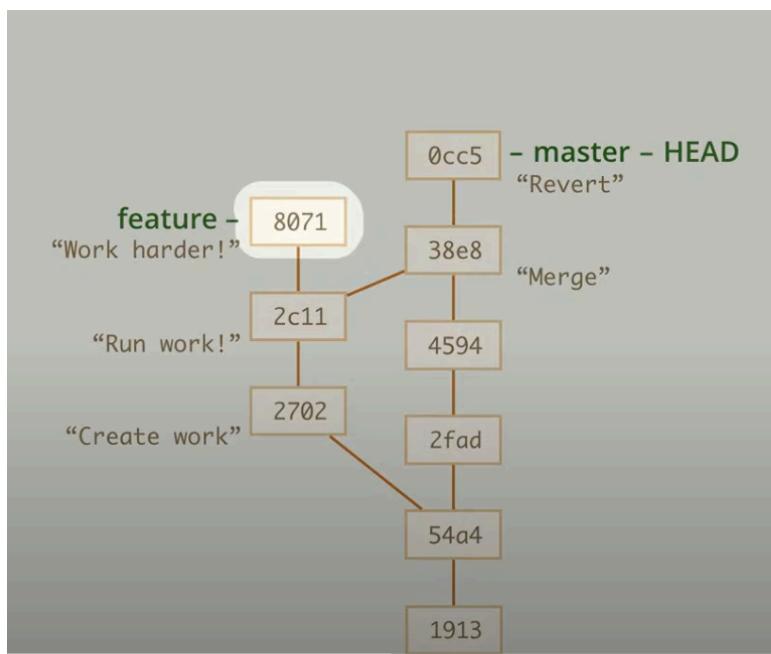
```
~/project master> git rebase master feature
```



а потом повторное слияние:

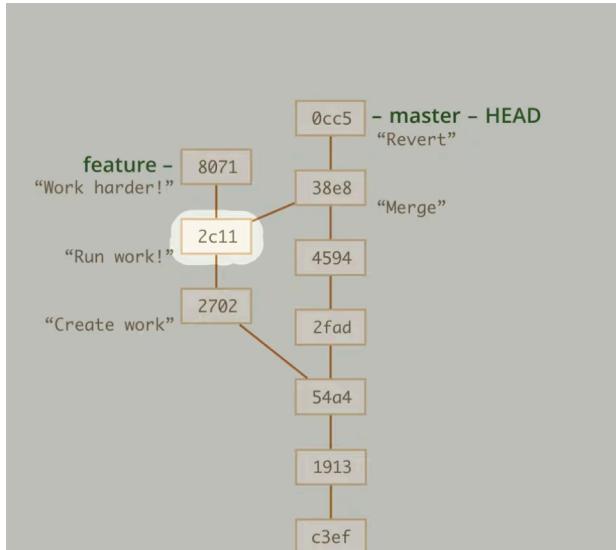


нюансы: перебазирование сработает не так, как нам нужно. При вызове rebase составит список коммитов для переноса и в него войдут все коммиты feature, которых нет в master, то есть появившиеся в feature с момента расхождения веток.



то есть перебазирован будет всего один коммит

Для перебазирования всей ветки использовать флаг --onto, который позволяет указать, с какого момента начинать перебазировать

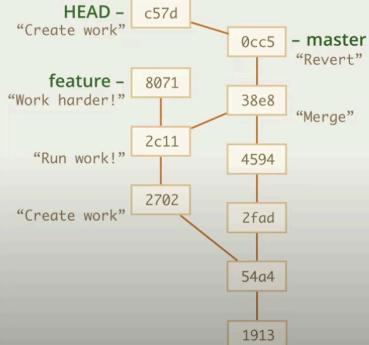


--onto <куда> <с какого момента> <какую ветку (по умолчанию текущая)>

```
~/project master> git rebase --onto master 54a4 feature
First, rewinding head to replay your work on top of it...
Applying: Create work
Applying: Run work
Using index info to reconstruct a base tree...
M      index.html
Falling back to patching base and 3-way merge...
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html.
error: Failed to merge in the changes.
Patch failed at 0002 Run work
Use 'git am --show-current-patch' to see the failed patch

Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase --abort".

x ~/project :c57d315 x1> ■
```



при копировании второго коммита возник конфликт (такой как и в предыдущих сериях, рассматриваем тот же пример). Можно заново разрешить и продолжить, но в реальности конфликты могут быть намного больше и разрешать одно и то же не хочется.

В гит есть специальный механизм **rerere** - специальный механизм для разрешения повторных конфликтов. Он применяет те же изменения которые были в аналогичных ситуациях при таком же конфликте в тех же файлах.

Отменяем перебазирование и включаем rerere

```
x ~/project :c57d315 x1> git rebase --abort  
~/project feature> git config rerere.enabled true
```

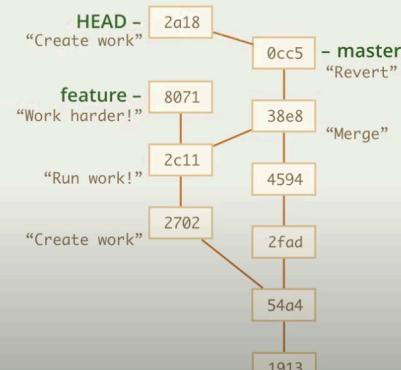
rerere нужно научить конфликтам, чтобы он знал какие были и как мы их разрешили. Обычно он запоминает их сам, если включен. Но у нас он был выключен, поэтому нужно использовать специальный скрипт, который обычно входит в поставку гит, но находится не по системному пути (найти как на макбуке)

флаг --all обучит всем конфликтам из всех веток:

```
~/project feature> /opt/local/share/git/contrib/rerere-train.sh  
--all  
Learning from 38e84f1 Merge branch 'feature'  
Recorded preimage for 'index.html'  
Recorded resolution for 'index.html'.  
Previous HEAD position was 4594f10 Run sayBye  
Switched to branch 'feature'
```

Теперь перебазируем опять:

```
~/project feature> git rebase --onto master 54a4  
First, rewinding head to replay your work on top of it...  
Applying: Create work  
Applying: Run work  
Using index info to reconstruct a base tree...  
M     index.html  
Falling back to patching base and 3-way merge...  
Auto-merging index.html  
CONFLICT (content): Merge conflict in index.html  
Resolved 'index.html' using previous resolution.  
error: Failed to merge in the changes.  
error: Patch failed at 0002 Run work  
Use 'git am --show-current-patch' to see the failed patch  
  
Resolve all conflicts manually, mark them as resolved with  
"git add/rm <conflicted_files>", then run "git rebase --continu e".  
You can instead skip this commit: run "git rebase --skip".  
To abort and get back to the state before "git rebase", run "gi t rebase --abort".  
  
x ~/project :2a18d28 x1>
```



ГИТ ГОВОРЯТ, ЧТО ИСПОЛЬЗОВАЛ ПРЕДЫДУЩЕЕ РЕШЕНИЕ КОНФЛИКТА

проверяем файл, все окей, конфликт решен. Теперь добавляем его в индекс и продолжаем перебазирование.

```
x ~/project :2a18d28 x1> cat index.html
<!DOCTYPE html>
<html>
  <head>
    <title>Git rules!</title>
    <script src="script.js"></script>
  </head>
  <body>
    <script>
      sayHi();
    </script>

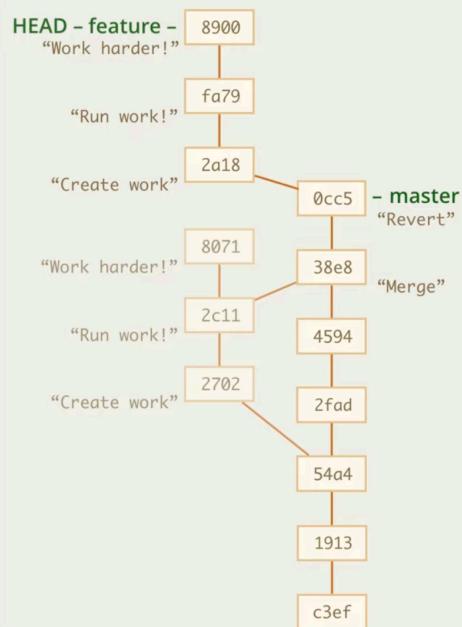
    Git rules!

    <script>
      work();
      sayBye();
    </script>
  </body>
</html>

~/project :2a18d28 x1> git add index.html

~/project :2a18d28 •1> git rebase --continue
Applying: Run work
Applying: Work harder!

~/project feature> █
```

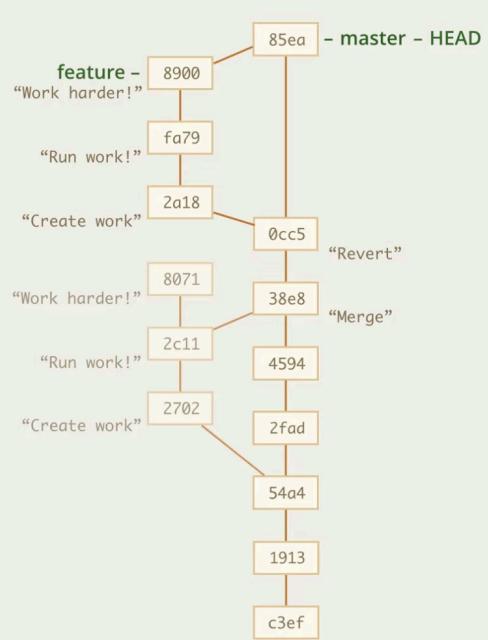


завершим слияние (флаг --no-ff указывает, что слияние должно быть не перемоткой, а полноценным коммитом):

```
~/project feature> git checkout master
Switched to branch 'master'

~/project master> git merge --no-ff --no-edit feature
Merge made by the 'recursive' strategy.
 index.html | 1 +
 script.js  | 4 +++
 2 files changed, 5 insertions(+)

~/project master> █
```



как и ранее мы заново объединили feature и master, но на этот раз не потребовалось отмены отмены

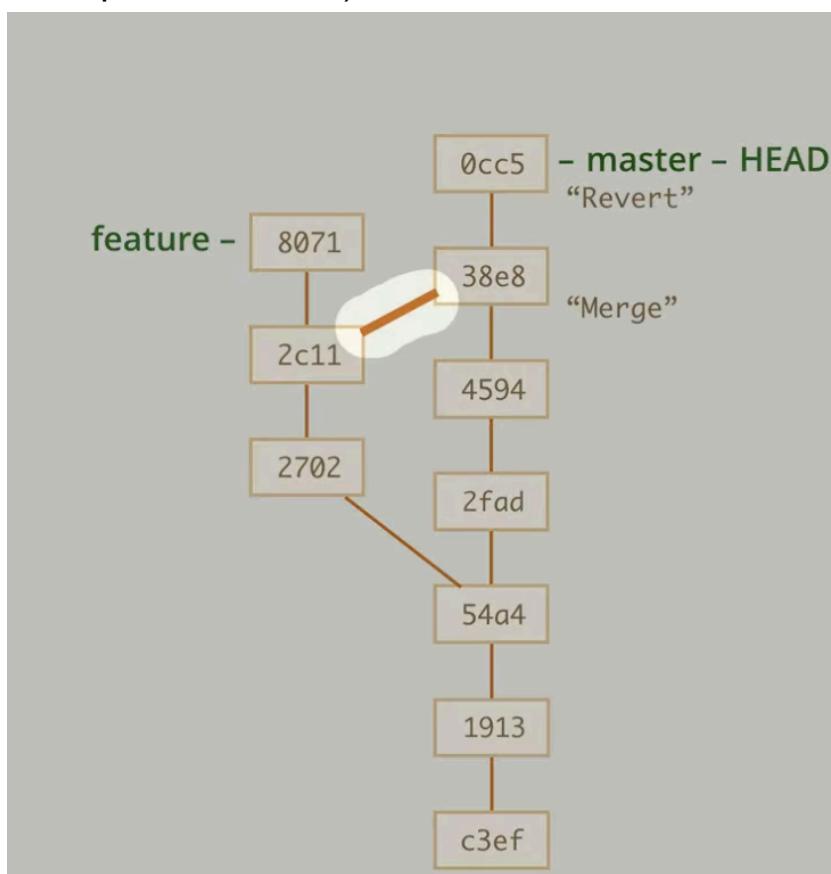
Но нужно отметить, что данное решение подходит, если мы свободно можем перебазировать feature, то есть если эта наша

ветка и не будет проблем с синхронизацией с коллегами. Если это не так, то вариант отмены отмены остается работающим выбором.

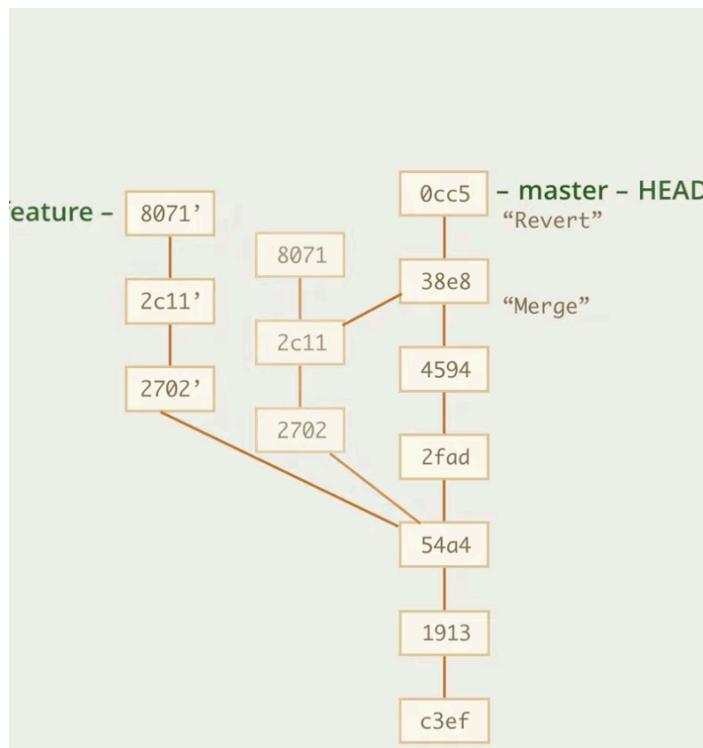
---

Рассмотрим еще один вариант повторного слияния:

как мы помним неудобства возникают из-за этой связи (из-за этой связи гит считает старые коммиты частью master и не берет их при повторном слиянии):



К сожалению просто убрать эту связь мы не можем, но можно перебазировать ветку **feature**, оставив еще на том же месте, где она сейчас, то есть создать копию всех коммитов как на рисунке ниже:



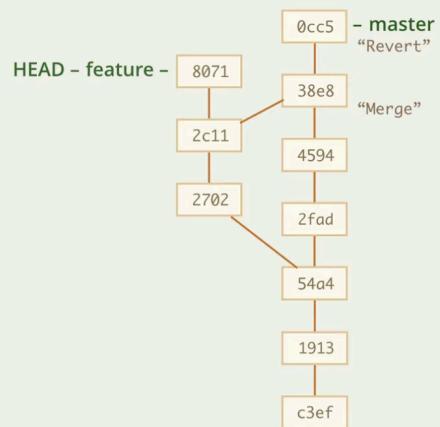
ГИТ не дает перебазировать сюда же, так как видит, что ветка итак растет с этого коммита

```

~/project master> git checkout feature
Switched to branch 'feature'

~/project feature> git rebase 54a4
Current branch feature is up to date.

~/project feature> █
  
```



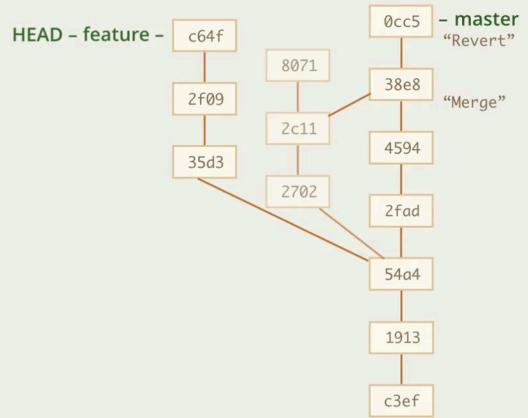
Чтобы заставить гит это сделать - редкий флаг --no-ff: даже если коммиты растут из нужного места, все равно нужно создать их копии

```
~/project master> git checkout feature
Switched to branch 'feature'

~/project feature> git rebase 54a4
Current branch feature is up to date.

~/project feature> git rebase 54a4 --no-ff
Current branch feature is up to date, rebase forced.
First, rewinding head to replay your work on top of it...
Applying: Create work
Applying: Run work
Applying: Work harder!

~/project feature>
```



конфликтов нет, так как ветка перебазировалась на то же место, где она была

теперь сделаем обычное слияние с мастер

это вроде тоже не для публичных веток