



ИСТОРИЯ РАБОТЫ И ВЕТКИ



ИЛЬНАЗ ГИЛЬЯЗОВ



ИЛЬНАЗ ГИЛЬЯЗОВ

СТО в aims

 coursar@gmail.com

 [ilnaz.gilyazov](https://www.instagram.com/ilnaz.gilyazov)

 [coursar](#)

ПЛАН ЗАНЯТИЯ

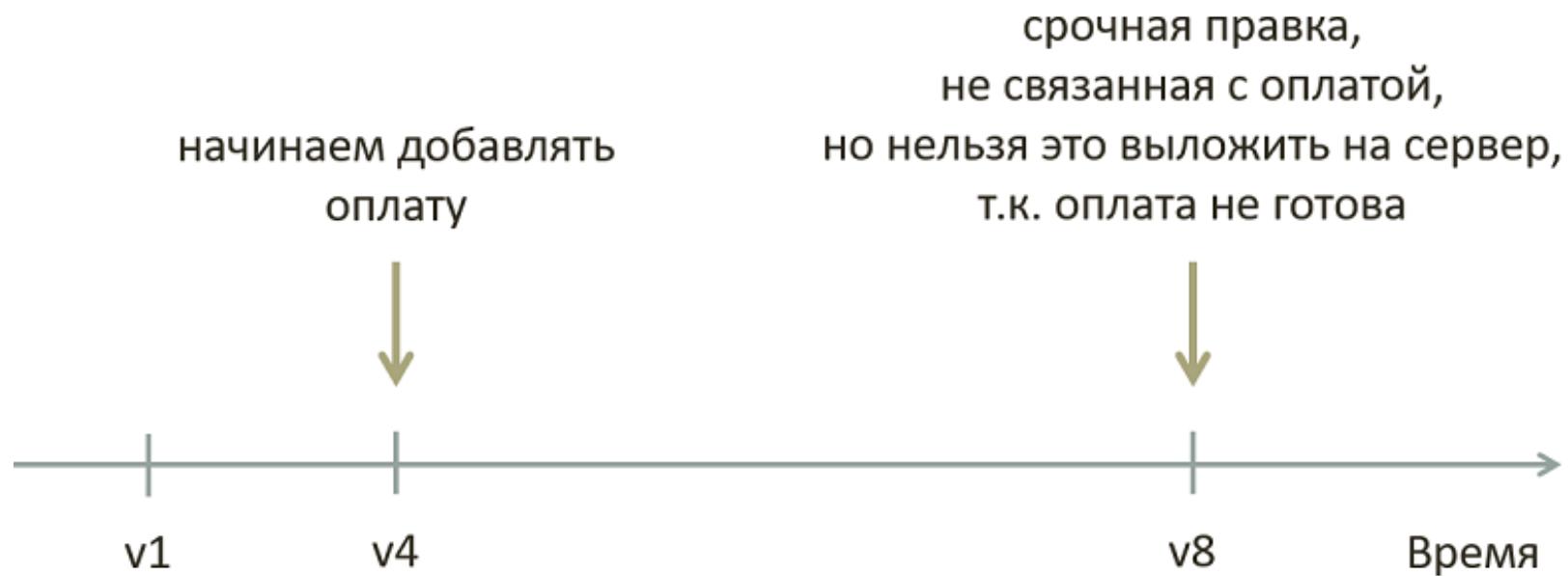
1. Ветки
2. Теги
3. Работа с историей
4. Откат изменений

ПАРАЛЛЕЛЬНАЯ РАЗРАБОТКА

Представим следующую ситуацию: вы используете систему контроля версий и поддерживаете ваш сервис, фиксируя все изменения в этой системе контроля версий.

В один день приходит необходимость добавить к сервису новую функцию (например, покупка с использованием карты онлайн). Это долгая разработка, при этом параллельные мелкие задачи никто не отменял, поэтому мы не можем просто так взять и по шагам добавлять оплату.

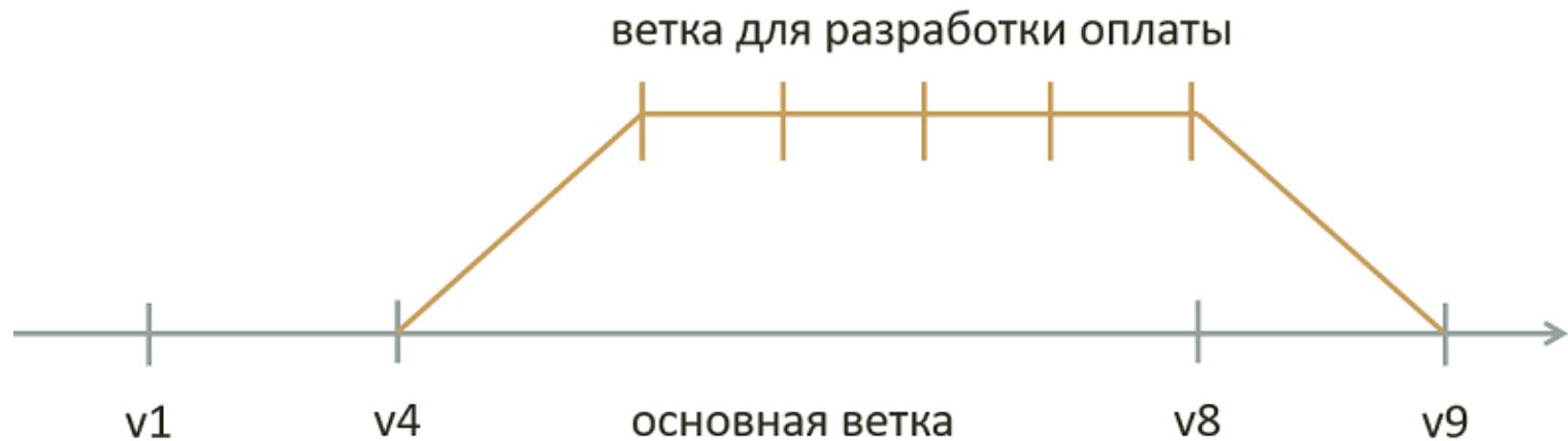
ПАРАЛЛЕЛЬНАЯ РАЗРАБОТКА



ВЕТВЛЕНИЕ

Git (как и другие VCS) предлагает механизм ветвления (`branches`) для решения подобных проблем. То есть в какой-то момент работы создаётся параллельная ветвь разработки, которая не пересекается с основной. И лишь когда все изменения готовы – мы сливаляем готовую протестированную функцию в основную ветку.

ПАРАЛЛЕЛЬНАЯ РАЗРАБОТКА

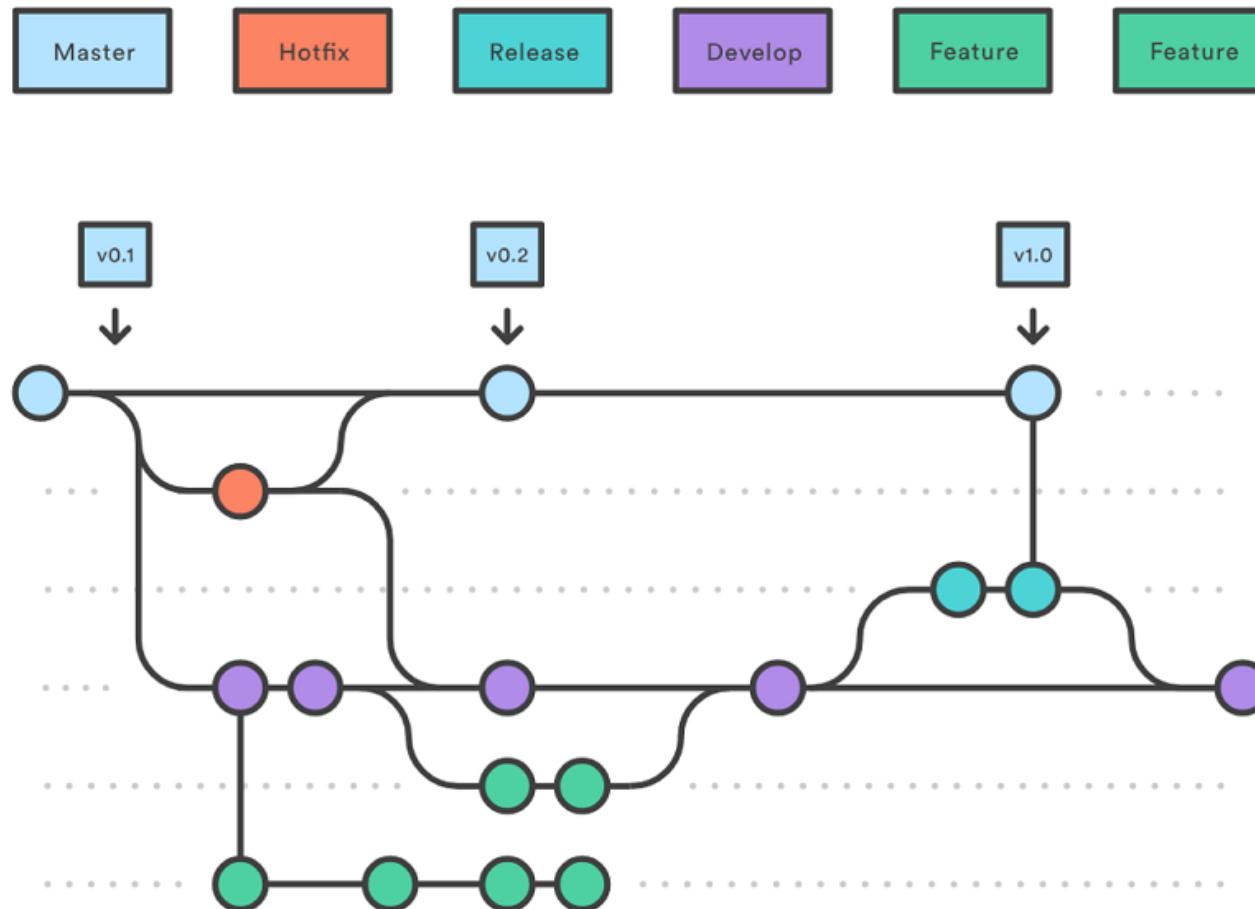


ВЕТВЛЕНИЕ КАК ПОДХОД

Большинством команд ветвление используется в качестве основного подхода при разработке, вплоть до того, что формируются специальные согласованные правила того, как правильно создавать ветви, когда и зачем.

При этом Git не накладывает на вас каких-то идеологических ограничений в плане использования веток, поэтому вы вольны выбрать в своей команде именно тот подход, который близок вам.

GITFLOW



Источник: atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

master branch

По умолчанию Git самостоятельно создаёт ветку и называется она `master`.

Вы можете дополнительно создавать любое количество веток и переключаться между ними (в Git очень эффективно реализован механизм переключения между ветками).

ТЕКУЩАЯ ВЕТКА

В Git есть понятие текущая ветка (`current branch`) – т.е. ветка, в которой мы работаем в настоящий момент. Все коммиты, которые мы будем делать будут относиться к ветке, которая в данный момент является текущей.

Мы можем переключаться на другую ветку и делать коммиты в ней, тогда они будут относиться уже к другой ветке.

master

В Git Bash текущая ветка показывается в строке приглашения:

```
user@desktop MINGW32 /c/project (master)  
$
```

Мы же (для экономии места) будем указывать следующим образом:

```
master$ git branch
```

НАЧНЁМ РАЗРАБАТЫВАТЬ ФУНКЦИЮ ОПЛАТЫ

Давайте начнём разрабатывать функцию оплаты, делать это мы будем с помощью веток Git.

Схема нашей работы будет следующая:

1. Создадим новую ветку `feature/payment`;
2. Переключимся на неё;
3. Сделаем все необходимые изменения;
4. Зальём всё на GitHub;
5. Сольём наши изменения с веткой `master`.

ПРОСМОТР ВЕТОК

Посмотреть существующие ветки можно с помощью команды

```
git branch:
```

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

СОЗДАНИЕ ВЕТКИ

Создать новую ветку можно с помощью команды

```
git branch <имя новой ветки>:
```

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

```
master$ git branch feature/payment
```

```
master$ git branch  
feature/payment  
* master
```

ПЕРЕКЛЮЧЕНИЕ МЕЖДУ ВЕТКАМИ

Для того, чтобы переключиться на другую ветку необходимо использовать команду `git checkout <имя ветки>`:

```
master$ git branch  
feature/payment  
* master
```

```
master$ git checkout feature/payment  
Switched to branch 'feature/payment'
```

```
feature/payment$ git branch  
* feature/payment  
master
```

Важно: чтобы при переключении не было проблем, возьмите за привычку фиксировать все файлы до переключения.

ДОБАВИМ СТРАНИЦУ ОПЛАТЫ

Добавим страницу оплаты и закоммитим её:

```
feature/payment$ git status
On branch feature/payment
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    payment.html

nothing added to commit but untracked files present (use "git add" to track)

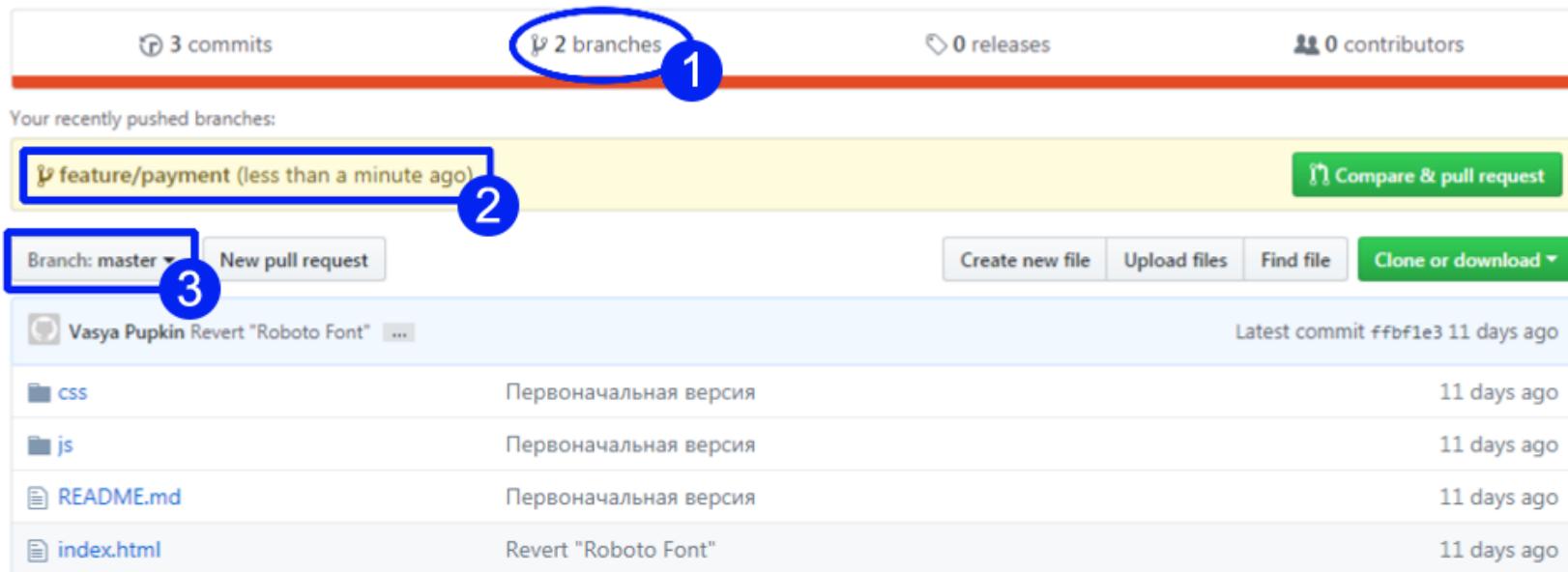
feature/payment$ git add payment.html

feature/payment$ git commit -m "Payment page"
[feature/payment 2502ceb] Payment page
  1 file changed, 12 insertions(+)
  create mode 100644 payment.html
```

REMOTE

По умолчанию наша ветка никак не синхронизирована с тем кодом, который хранится в GitHub.

Чтобы добавить нашу ветку в GitHub, нам необходимо сделать следующее:
`git push -u origin feature/payment`. После этого в интерфейсе GitHub мы увидим информацию:



The screenshot shows a GitHub repository interface. At the top, there are summary statistics: 3 commits, 2 branches (circled with a blue oval and labeled '1'), 0 releases, and 0 contributors. Below this, a section titled 'Your recently pushed branches:' lists a single branch: 'feature/payment (less than a minute ago)'. This branch is highlighted with a yellow background and circled with a blue oval and labeled '2'. To the right of this branch is a green 'Compare & pull request' button. Further down, there's a dropdown menu set to 'Branch: master' (circled with a blue oval and labeled '3') and a 'New pull request' button. On the right side of the page are buttons for 'Create new file', 'Upload files', 'Find file', and 'Clone or download'. The main content area displays the repository's history, showing a commit by 'Vasya Pupkin' titled 'Revert "Roboto Font"' made 11 days ago. Below this commit, four files are listed: 'CSS' (First commit), 'js' (First commit), 'README.md' (First commit), and 'index.html' (Revert "Roboto Font"). All commits are from 11 days ago.

File	Commit Message	Date
CSS	Первоначальная версия	11 days ago
js	Первоначальная версия	11 days ago
README.md	Первоначальная версия	11 days ago
index.html	Revert "Roboto Font"	11 days ago

КЛОНИРОВАНИЕ

При клонировании репозитория с удалённого сервера Git автоматически создаёт удалённую ветку `origin/master` и локальную `master`.

Для всех остальных удалённых веток локальные ветки не создаются их нужно создавать с помощью команды: `git branch <local-branch> <origin/remote-branch>`

СЛИЯНИЕ ИЗМЕНЕНИЙ

После того, как мы завершили все изменения (целиком доработали нашу новую функцию), нам необходимо эти изменения интегрировать (или как говорят «слиять» с веткой `master`).

Общий сценарий выглядит следующим образом:

1. Переключаемся на ветку `master`;
2. Выполняем операцию `merge`;
3. Заливаем всё на GitHub.

СЛИЯНИЕ ИЗМЕНЕНИЙ

```
feature/payment$ git branch
* feature/payment
master

feature/payment$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'

master$ git merge --no-ff feature/payment
Merge made by the 'recursive' strategy.
 payment.html | 12 ++++++
 1 file changed, 12 insertions(+)
 create mode 100644 payment.html

master$ git push
Username for 'https://github.com': netology-git
Enumerating objects: 1, done
Counting objects: 100% (1/1), done.
Compressing objects: 100% (1/1) done.
Writing objects: 100% (1/1), 234 bytes | 234.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To https://github.com/netology-git/demo.git
  ff8bf1e3..fbf634f  master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

СЛИЯНИЕ ИЗМЕНЕНИЙ

В результате слияния создаётся новый коммит, в рамках которого и объединяются изменения из ветки `feature/payment` в ветку `master`.

```
master$ git log --graph --oneline
* fbf634f (HEAD -> master, origin/master, origin/HEAD) Merge branch 'feature/payment'
|\ \
| * df35653 (origin/feature/payment, feature/payment) Payment page
| /
* ffbff1e3 Revert "Roboto Font"
* a1b3412 Roboto Font
* 4ad7e09 Первоначальная версия
```

ВИЗУАЛИЗАЦИЯ ВЕТОК

Для просмотра истории конкретной ветки:

```
master$ git log feature/payment --graph --oneline
* df35653 (origin/feature/payment, feature/payment) Payment page
* ffbf1e3 Revert "Roboto Font"
* a1b3412 Roboto Font
* 4ad7e09 Первоначальная версия
```

СЛИЯНИЕ ИЗМЕНЕНИЙ

Git умеет самостоятельно «сливать» изменения, если они не приводят к конфликтам.

Если же конфликт всё-таки происходит, то вам необходимо вручную его разрешить (указать Git'у как именно слить изменения).

ЗАДАЧА

Перед нами встала задача: интегрировать на наш сайт сервис [Yandex.Metrica](#).

Что мы сделали: по всем правилам создали новую ветку, внесли необходимые изменения, сделали коммит и ждём, чтобы слить все изменения в мастер.

Параллельно пришла вторая задача – добавить [Google Analytics](#). Мы сделали всё то же самое и теперь у нас следующая ситуация (см. скриншот на следующей странице).

ЗАДАЧА

```
master$ git log --graph --oneline --all
* 1235c23 (feature/analytics) Google Analytics
| * 7800a44 (feature/metrica) Yandex Metrica
|/
* fbf634f (HEAD -> master, origin/master, origin/HEAD) Merge branch 'feature/payment'
|\ 
| * df35653 (origin/feature/payment, feature/payment) Payment page
|/
* ffbf1e3 Revert "Roboto Font"
```

Теперь нам нужно изменения обеих веток слить в `master` (это можно сделать и одной командой, но мы с вами будем работать последовательно).

merge

Делаем `merge` для Яндекс.Метрики:

```
master$ git merge feature/metrica --no-ff
Merge made by the 'recursive' strategy.
payment.html | 1 +
 1 file changed, 1 insertion(+)
```

Делаем `merge` для Google Analytics:

```
master$ git merge feature/analytics --no-ff
Auto-merging payment.html
CONFLICT (content): Merge conflict in payment.html
Automatic merge failed; fix conflicts and then commit the result.
```

Получаем конфликт в файле `payment.html`.

ПОЧЕМУ ТАК ВЫШЛО?

Git не смог автоматически слить наши изменения, т.к. в обоих ветках мы вставили код в одной и той же строке. Произошёл конфликт. Конфликт выглядит редакторе вот так:

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <meta http-equiv="X-UA-Compatible" content="ie=edge">
7      <title>Document</title>
Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
8  <<<<< HEAD (Current Change)
9      <!-- yandex.metrika code -->
10 =====
11      <!-- google analytics -->
12 >>>> feature/analytics (Incoming Change)
13 </head>
14 <body>
15     Payment page
16 </body>
17 </html>
```

РАЗРЕШЕНИЕ КОНФЛИКТА

Теперь мы руками должны отредактировать строки 8-12, чтобы не осталось символов <<<<<, =====, >>>>>, приведя файл к нужному нам виду, например такому:

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <meta http-equiv="X-UA-Compatible" content="ie=edge">
7      <title>Document</title>
8      <!-- yandex.metrika code -->
9      <!-- google analytics -->
10     </head>
11     <body>
12         Payment page
13     </body>
14     </html>
```

A red circle containing the number 1, with a red arrow pointing from it to the line of code in the previous block.

После чего можем сделать коммит.

РАЗРЕШЕНИЕ КОНФЛИКТА

Таким образом нам придётся обработать все файлы, в которых возникли конфликты. Посмотреть все файлы с конфликтами можно с помощью `git status`:

```
master$ git status
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:    payment.html

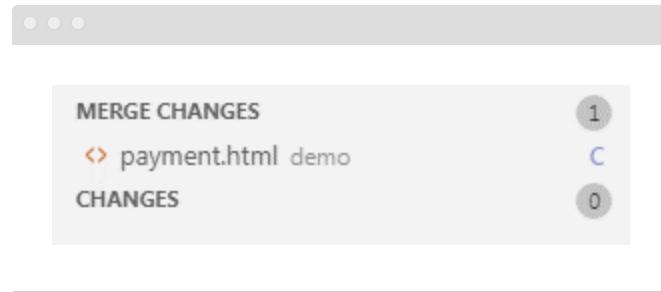
no changes added to commit (use "git add" and/or "git commit -a")
```

После чего использовать `git add` + `git commit`, либо сокращённую версию `git commit -a`.

РАЗРЕШЕНИЕ КОНФЛИКТА

Большинство продвинутых текстовых редакторов предлагают подсветку синтаксиса для разрешения конфликтов (и для отображения файлов с конфликтами) и вам не придётся изменять всё руками.

Например, в редакторе VS Code это будет выглядеть так:



ГРАФ

После разрешения конфликта и коммита, граф будет выглядеть следующим образом:

```
master$ git log --graph --oneline
* c60f753 (HEAD -> master) Merge branch 'feature/analytics'
| \
| * 1235c23 (feature/analytics) Google Analytics
* | 76a2b31 Merge branch 'feature/metrica'
| \ \
| | /
| / |
| * 7800a44 (feature/metrica) Yandex Metrica
| /
* fbf634f (origin/master, origin/HEAD) Merge branch 'feature/payment'
| \
| * df35653 (origin/feature/payment, feature/payment) Payment page
| /
* ffef1e3 Revert "Roboto Font"
* a1b3412 Roboto Font
* 4ad7e09 Первоначальная версия
```

УДАЛЕНИЕ ВЕТКИ

Мы можем удалить ветку, если она стала нам больше не нужна командой

```
git branch -d <имя_ветки>.
```

При этом удаляемая ветка не должна быть текущей.

```
master$ git branch -d feature/payment
Deleted branch feature/payment (was ffbf1e3)
```

GITHUB И УДАЛЕНИЕ ВЕТКИ

Удалить ветку с GitHub и локально можно с помощью следующего набора команд:

```
master$ git push --delete origin feature/payment
```

```
master$ git branch -d feature/payment
```

ТЕГИ

ЗАДАЧА

Выпустили мы первый релиз нашего сайта и хотим отметить этот момент в истории, чтобы не запоминать хэш-коды коммитов.

Для этого и существуют теги – мы можем отметить определённый момент в истории удобным нам обозначением.

РАБОТА С ТЕГАМИ

Давайте добавим тег в нашу историю:

```
master$ git tag -a v1.0 -m "Версия 1.0"
```

Просмотреть теги можно с помощью команды `git tag`:

```
master$ git tag  
v1.0
```

РАБОТА С ТЕГАМИ

Кроме того, команда `git show` позволяет получить информацию о метке:

```
master$ git show v1.0
tag v1.0
Trigger: Vasya Pupkin <vasya@localhost>
Date:   Tue Nov 6 23:05:57 2018 +0400

Version 1.0

commit fbf634f51ee7be25992564cd1789c5c158bca784 (HEAD -> master, tag: v1.0, origin/master, origin/HEAD)
Merge:  ffbf1e3 df35653
Author: Vasya Pupkin <vasya@localhost>
Date:   Tue Nov 6 19:20:48 2018 +0400

Merge branch 'feature/payment'
```

РАБОТА С ТЕГАМИ

Теги можно добавить и позже, указав id-коммита:

```
git tag -a v1.0 -m "Версия 1.0" <commit-id>
```

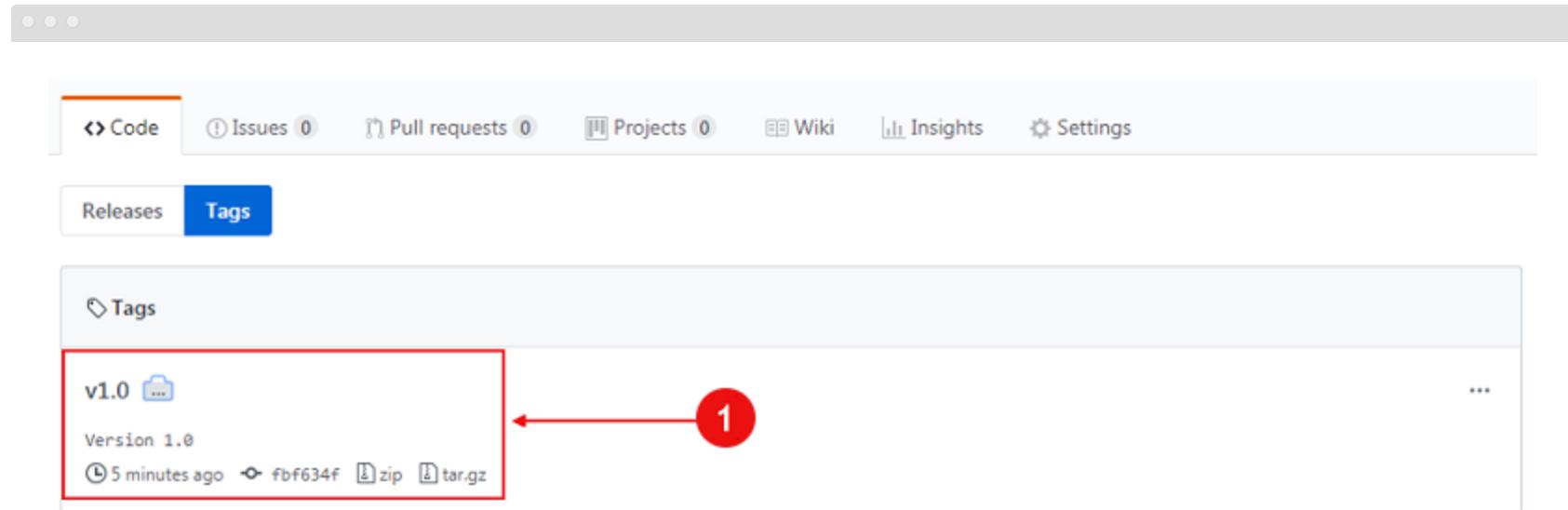
GITHUB И ТЕГИ

По умолчанию теги не отправляются при выполнении `git push`

Это необходимо указать явно:

```
master$ git push --tags
Username for 'https://github.com': netology-git
Enumerating objects: 1, done
Counting objects: 100% (1/1), done.
Compressing objects: 100% (1/1) done.
Writing objects: 100% (1/1), 162 bytes | 40.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To https://github.com/netology-git/demo.git
 * [new tag]          v1.0 -> v1.0
```

GITHUB И ТЕГИ



УДАЛЕНИЕ ТЕГА

Если вы ещё не отправили тег на GitHub, то удалить его можно командой:

```
master$ git tag -d v1.0
```

Если же уже отправили, то:

```
master$ git push --delete origin v1.0
```

```
master$ git tag -d v1.0
```

РАБОТА С ИСТОРИЕЙ

ПРОСМОТР ИСТОРИИ

Представим себе ситуацию: мы хотим посмотреть, кто менял файл `index.html` (после кого конкретно поехала вёрстка).

Смотрим историю через `git log`:

```
master$ git log
commit fbf634f51ee7be25992564cd1789c5c158bca784 (HEAD -> master, origin/master, origin/HEAD)
Merge: ffbff1e3 df35653
Author: Vasya Pupkin <vasya@localhost>
Date:   Tue Nov 6 19:20:48 2018 +0400

    Merge branch 'feature/payment'

commit df356533839c94ad3ed5b3ca8a34b9a00fbcc92b (origin/feature/payment, feature/payment)
Author: Vasya Pupkin <vasya@localhost>
Date:   Tue Nov 6 19:04:35 2018 +0400

    Payment page
```

ПРОСМОТР ИСТОРИИ

Не очень-то помогло, правда?

Попробуем указать конкретный путь:

```
master$ git log -- index.html
commit ffbff1e32bb8fcfcdd35c14276ec375770214c2ba
Author: Vasya Pupkin <vasya@localhost>
Date:   Fri Oct 26 20:13:32 2018 +0400
```

Revert "Roboto Font"

This reverts commit a1b34121a1147e68f9fd2545af1004aa23ab3cd8.

```
commit a1b34121a1147e68f9fd2545af1004aa23ab3cd8
Author: Vasya Pupkin <vasya@localhost>
Date:   Fri Oct 26 19:42:48 2018 +0400
```

Roboto Font

```
commit 4ad7e09ee8d76ce6c6f25e233cf186689b3b55b2
Author: Vasya Pupkin <vasya@localhost>
Date:   Fri Oct 26 19:26:05 2018 +0400
```

Первоначальная версия

ПРОСМОТР ИСТОРИИ

Уже лучше, теперь мы можем для каждого коммита посмотреть изменения:

```
master$ git show ffbf1e -- index.html
commit ffbf1e32bb8fcfcdd35c14276ec375770214c2ba
Author: Vasya Pupkin <vasya@localhost>
Date:   Fri Oct 26 20:13:32 2018 +0400

        Revert "Roboto Font"

This reverts commit a1b34121a1147e68f9fd2545af1004aa23ab3cd8.

diff --git a/index.html b/index.html
...
<title>Document</title>
<link rel="stylesheet" href="css/style.css">
- <link href="https://fonts.googleapis.com/css?family=Roboto&subset=cyrillic" rel="stylesheet">
...
```

ПРОСМОТР ИСТОРИИ

Так явно лучше, можем ли мы совместить это в одной команде? Конечно:

```
master$ git log -p -- index.html
commit ffbff1e32bb8fcfcdd35c14276ec375770214c2ba
Author: Vasya Pupkin <vasya@localhost>
Date:   Fri Oct 26 20:13:32 2018 +0400

    Revert "Roboto Font"

    This reverts commit a1b34121a1147e68f9fd2545af1004aa23ab3cd8.

...
<title>Document</title>
<link rel="stylesheet" href="css/style.css">
- <link href="https://fonts.googleapis.com/css?family=Roboto&subset=cyrillic" rel="stylesheet">
...
commit a1b34121a1147e68f9fd2545af1004aa23ab3cd8
Author: Vasya Pupkin <vasya@localhost>
Date:   Fri Oct 26 19:42:48 2018 +0400

    "Roboto Font"

...
<title>Document</title>
<link rel="stylesheet" href="css/style.css">
+ <link href="https://fonts.googleapis.com/css?family=Roboto&subset=cyrillic" rel="stylesheet">
...
```

ПОИСК ПО ИСТОРИИ

Если же мы хорошо и правильно пишем комментарии к коммитам, то можем выполнить поиск и по ним:

```
master$ git log --grep 'Перво'
commit 4ad7e09ee8d76ce6c6f25e233cf186689b3b55b2
Author: Vasya Pupkin <vasya@localhost>
Date:   Fri Oct 26 19:26:05 2018 +0400
```

Первоначальная версия

ПОИСК ПО ИСТОРИИ

Если же мы знаем конкретную строку для поиска, то мы можем поискать все коммиты, связанные с добавлением/удалением этой строки: `git log -S'строка' -p`.

```
master$ git log -S'page' -p
commit df356533839c94ad3ed5b3ca8a34b9a00fbcc92b (origin/feature/payment, feature/payment)
Author: Vasya Pupkin <vasya@localhost>
Date:   Tue Nov 6 19:04:35 2018 +0400

    Payment page

diff --git a/payment.html b/payment.html
...
+<!DOCTYPE html>
+<html lang="en">
+<head>
...
+</head>
+<body>
+  Payment page
+</body>
+</html>
\ No newline at end of file
```

```
git log --all
```

По умолчанию `git` ищет по той ветке, на которой вы в данный момент находитесь. Флаг `--all` позволяет искать по всем веткам.

АВТОРСТВО ИЗМЕНЕНИЙ

Если мы не хотим путешествовать по истории, а посмотреть по конкретной строке кода, кто этот код в последний раз менял и в каком коммите, то нам поможет команда:

```
git blame -- <path>
```

```
master$ git blame -- index.html
^4ad7e09 (Vasya Pupkin 2018-10-26 19:26:05 +0400 1) <!DOCTYPE html>
^4ad7e09 (Vasya Pupkin 2018-10-26 19:26:05 +0400 2) <html lang="en">
^4ad7e09 (Vasya Pupkin 2018-10-26 19:26:05 +0400 3) <head>
^4ad7e09 (Vasya Pupkin 2018-10-26 19:26:05 +0400 4)     <meta charset="UTF-8">
...
^4ad7e09 (Vasya Pupkin 2018-10-26 19:26:05 +0400 7)     <title>Document</title>
^4ad7e09 (Vasya Pupkin 2018-10-26 19:26:05 +0400 8)     <link rel="stylesheet" href="css/style.css">
a1b34121 (Vasya Pupkin 2018-10-26 19:26:05 +0400 9)     <link href="https://fonts.googleapis.com/css?
family=Roboto&subset=cyrillic" rel="stylesheet">
...
```

ОТКАТ ИЗМЕНЕНИЙ

ОСОБЕННОСТИ РЕАЛИЗАЦИИ

Данная информация относится ко внутреннему устройству Git, но критически важна с точки зрения понимания всего дальнейшего процесса работы.

Фактически, ветка представляет из себя указатель на определённый коммит в истории, который Git автоматически передвигает при создании нового коммита.

ОСОБЕННОСТИ РЕАЛИЗАЦИИ



Делаем коммит:



ИСТОРИЯ

Откуда же мы видим визуализацию истории? В каждом коммите есть специальное поле, которое содержит «родительский» (или «родительские», если их несколько) коммиты. Таким образом, понятие принадлежности коммита ветки определяется лишь тем, содержится ли он в цепочке родителя, родителя родителя и т.д. коммита, на который сейчас указывает ветка.

HEAD

В рамках Git помимо веток, которые являются указателями на коммиты, есть ещё один указатель – это `HEAD`. Он как раз и определяет то, в какой ветке мы сейчас находимся, т.к. содержит в себе ссылку на ветку (напоминаю, что ветка – это указатель на коммит).

А что будет, если мы попробуем передвинуть указатель `HEAD` не на ветку, а на определённый коммит (на который не указывает ни одна ветка)?

ВОЗВРАТ К ОПРЕДЕЛЁННОМУ КОММИТУ

Новая задача: периодически случается, что мы вносим в своё приложение ошибку и хотели бы посмотреть, когда этой ошибки не было. Мы можем вернуться к любому моменту в истории с помощью команды `git checkout` :

```
git checkout <commit-id>
```

DETACHED HEAD

Состояние, когда HEAD указывает не на ветку, а на коммит (на который не указывает ни одна ветка) называется Detached HEAD. В этом состоянии можно делать коммиты, но не рекомендуется:

```
master$ git checkout a1b3412
Note: checking out 'a1b3412'.
```

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 a1b3412 Roboto Font

DETACHED HEAD

Для такой ситуации есть два ключевых сценария (с учётом того уровня глубины, на котором мы изучаем Git):

- создать на базе этого коммита новую ветку:

```
git branch <name>
```

```
git checkout <name>
```

или

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

- вернуться на другую ветку (если мы хотели просто посмотреть состояние этого коммита).

```
git checkout master
```

--no-ff

В Git реализован механизм, который называется `fast-forward`. Что он делает: если он видит при слиянии изменений, что самым простым сценарием является просто переместить указатель на ветку на определённый коммит и не делать `merge commit`, то он так и поступает.

В большинстве случаев это не желательное поведение, поэтому мы отключаем этот механизм с помощью флага `--no-ff`.

git reset

Позволяет нам передвигать указать, тем самым эмулируя «отмену» коммитов (сами коммиты по факту не отменяются, мы просто перемещаем указатель).

На самом деле отменить коммиты нельзя, можно лишь сделать новые.

ЗАДАЧА

Задача у нас следующая – отменить ряд неудачных коммитов, вернув проект к одному из предыдущих состояний.

git reset

Режимы:

- `hard` – передвигаем указатель на определённый коммит, не сохраняя никаких изменений;
- `soft` – передвигаем указатель на определённый коммит, при этом предыдущие изменения сохраняются в рабочем каталоге и `index`'е;
- `mixed` – передвигаем указатель на определённый коммит, при этом предыдущие изменения сохраняются в рабочем каталоге, но не в `index`'е.

СИНТАКСИС

```
git reset <mode> <commit-id>
```

где <mode>:

- --hard;
- --soft;
- --mixed (по умолчанию, можно не указывать).

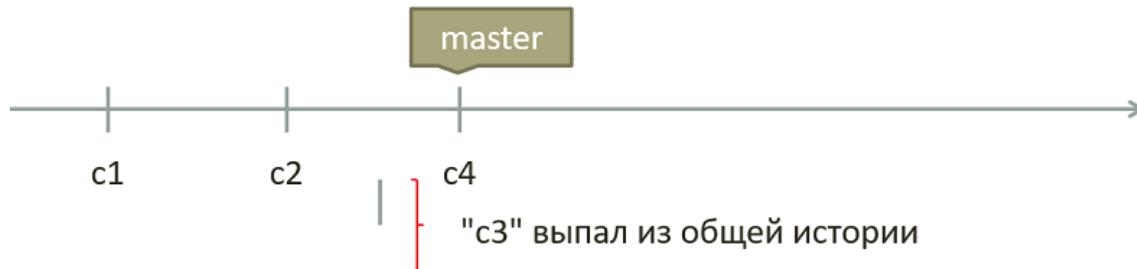
reset



```
git reset --hard HEAD~
```



```
git commit -a -m "New commit"
```



СИНТАКСИС

Вместо того, чтобы каждый раз указывать id-коммита, мы можем воспользоваться специальным синтаксисом:

```
git reset <mode> HEAD~<n>
```

Это будет значить, что мы хотим отойти от `HEAD` на n-шагов.

ПРИМЕР

Рассмотрим следующую ситуацию:

```
master$ git log --graph --oneline --all
* 9b599b3 (HEAD -> master) Third
* d6406cf Second
* c4e43c9 First

master$ git status
On branch master
Your branch is ahead of 'origin/master' by 5 commits.
(use "git push" to publish your local commits)

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:   README.md

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

hard

Сделаем hard reset:

```
master$ git reset --hard HEAD~2
HEAD is now at c4e43c9 First

master$ git log --graph --oneline --all
* c4e43c9 (HEAD -> master) First

master$ git status
On branch master
Your branch is ahead of 'origin/master' by 3 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

«Следующие» коммиты по истории потерялись и все изменения в индексе и рабочем каталоге тоже.

soft

Сделаем soft reset:

```
master$ git reset --soft HEAD~2
master$ git status
On branch master
Your branch is ahead of 'origin/master' by 3 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   README.md

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:   README.md
```

«Следующие» коммиты по истории хоть и потерялись, но все изменения лежат в `stage area`, а также сохранились наши изменения в рабочем каталоге.

mixed

Сделаем mixed reset:

```
master$ git reset HEAD~2
Unstaged changes after reset:
M       README.md
```

```
master$ git status
On branch master
Your branch is ahead of 'origin/master' by 3 commits.
(use "git push" to publish your local commits)
```

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: README.md

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

«Следующие» коммиты по истории хоть и потерялись, но все изменения из коммитов и из рабочего каталога лежат в рабочем каталоге (но не в stage area).

ИТОГИ

Сегодня мы с вами разобрали достаточно много вопросов:

1. Работа с ветками (создание, переключение, слияние, удаление);
2. Некоторые особенности внутренней работы Git;
3. Хранение веток на GitHub;
4. Работа с историей и откат изменений.



Задавайте вопросы и напишите отзыв о лекции!

ИЛЬНАЗ ГИЛЬЯЗОВ



coursar@gmail.com



[ilnaz.gilyazov](https://www.instagram.com/ilnaz.gilyazov)



[coursar](#)