Git Cheatsheet

Owner	Ⓐ Arsenii Dunaev
Tags	

Шпаргалка №1 (Терминал)

https://practicum.yandex.ru/trainer/git-basics/lesson/fe0bcd71-f592-423b-bb81-27c37a6a115b/

Шпаргалка №2 (Основы git)

https://practicum.yandex.ru/trainer/git-basics/lesson/b1ecee27-bb78-46a0-8d13-0364c7803f55/

Хэш коммита

Хэш коммита это уникальный идентификатор коммита, который содержит в себе преобразованную информацию о коммите (автор, дата создания коммита, содержимое коммита, ссылка на предыдущий коммит).

Хэши можно передавать как параметры гит командам чтобы произвести действие над нужным коммитом.

Все хеши и таблицу хеш → информация о коммите Git сохраняет в служебные файлы. Они находятся в скрытой папке .git в репозитории проекта.

При вызове команды ***git log*** выводится полный хэш коммита однако при при использовании ***git log —oneline*** выводится сокращенный хэш, который сокращается таким образом, чтобы все хэши оставались уникальными (соответственно их длина может варьироваться).

Полный лог с полноценным хэшем:

```
arsenydunaev@Arsenijs-Laptop Test-Repository % git log
commit 24424065e55f4691c8df6f1d1338fd22648b94d1 (HEAD -> <main>, origin/<main>)
Author: ArsenyD <arsdunaev@icloud.com>
Date:   Sat Nov 4 18:39:02 2023 +0300

    README.md has been modified

commit 678ef618785e319d1c348dad4e25154ce777d27d
Author: ArsenyD <arsdunaev@icloud.com>
Date:   Sat Nov 4 18:36:21 2023 +0300

    readme.txt has been deleted and README.md has been added

commit 4924511af0afeedd60b7ce114a60c1220f69e338
Author: ArsenyD <arsdunaev@icloud.com>
Date:   Sat Nov 4 18:21:06 2023 +0300

    new swift file has been created

commit 1ed2e279b79d2a60b31acfebe4e300bb2f912e85
Author: ArsenyD <arsdunaev@icloud.com>
Date:   Sat Nov 4 15:27:04 2023 +0300

    updated readme

commit 25b4c634695b87273bc320243a05ce5d239d5eac
Author: ArsenyD <arsdunaev@icloud.com>
Date:   Sat Nov 4 15:25:46 2023 +0300

    added another line of code to hello_world.swift

commit 0adc1e609cdcb6f4054def93ce4c1dfa81859a39
Author: ArsenyD <arsdunaev@icloud.com>
Date:   Sat Nov 4 15:00:00 2023 +0300

    Two files were modified
```

Сокращенный лог с кратким хэшем (более удобен в больших репозиториях с большим количеством коммитов):

```
arsenydunaev@Arsenijs-Laptop Test-Repository % git log --oneline
2442406 (HEAD -> <main>, origin/<main>) README.md has been modified
678ef61 readme.txt has been deleted and README.md has been added
4924511 new swift file has been created
1ed2e27 updated readme
25b4c63 added another line of code to hello_world.swift
0adc1e6 Two files were modified
```

Head

При вызове команды git log выше можно увидеть надпись (HEAD -> <main>)

Файл HEAD - один из файлов в папке .git, который соддержит ссылку на хэш последнего коммита.

Когда создается новый коммит, Git обновляет refs/heads/master(ссылка внутри файла HEAD) — записывает в него хеш последнего коммита. Получается, что HEAD тоже обновляется, так как ссылается на refs/heads/master.

HEAD можно использовать как параметр комманд вместо хэша последнего коммита.

Статусы файлов в Git

1. **Untracked** - Самое первое состояние файла. Как только мы добавляем какой-то новый файл в репозиторий, но еще не “добавили его на сцену”(staged) он

находится в состоянии Untracked. У untracked-файла нет предыдущих версий, зафиксированных в коммитах или через команду git add.

2. **Staged** - Состояние файла, который мы добавили на сцену. В это состояние могут попасть новые файлы, которые мы добавили на сцену с помощью *git add <name>* (эти файлы также переходят из состояния Untracked в состояние Tracked), а так же ранее отслеживаемые файлы, но которые мы изменили и тоже добавли на сцену (**modified**). Когда мы добавляем на сцену новые файлы и/или ранее существующие, но измененные, мы объявляем содержание следующего коммита.
3. **Modified** - Состояние файла, который изменен по сравнению с последней версией этого же файла в коммите (или по сравнению с версией этого же файла на сцене). Эти файлы также находятся в состоянии Tracked.

> 💡 **Про staged и modified.**
>
> Команда *git add* добавляет в staging area только текущее содержимое файла. Если вы, например, сделаете *git add file.txt*, а затем измените *file.txt,* то новое содержимое файла не будет находиться в staging. Git сообщит об этом с помощью статуса modified: файл изменён относительно той версии, которая уже в staging. Чтобы добавить в staging последнюю версию, нужно выполнить *git add file.txt* ещё раз.

4. **Tracked** - Состояние файла, который закоммичен и не имеет изменений после коммита. (Так же подразумевается что файлы на сцене и модифицированные файлы тоже Tracked)

Git status

git status одна из вамных команд при работе с репозиториями. При помощи данной команды можно узнать текущее состояние файлов в репозитории, есть ли новые,

не отслеживаемые файлы; какие файлы изменены и добавлены ли они на сцену; что вообще добавлено на сцену и тд.

В итоге git status показывает только следующие состояния файлов:
staged (Changes to be committed);
modified (Changes not staged for commit);
untracked (Untracked files).

Варианты вывода *git status*

1. Все файлы в состоянии tracked (нет файлов состоянии staged, modified, untracked)

```
$ git status
On branch master
nothing to commit, working tree clean
```

2. В репозитории содержатся untracked файл(ы)

```
$ touch fileA.txt
$ git status
On branch main
Untracked files: # найдены неотслеживаемые файлы
  (use "git add <file>..." to include in what will be committed)
        fileA.txt

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

3. Untracked файл(ы) добавлен на сцену

```
$ git add fileA.txt
$ git status
On branch main
Changes to be committed: # новая секция
```

```
  (use "git restore --staged <file>..." to unstage)
        new file:   fileA.txt
```

4. Есть modified файл(ы)

```
# внесли в fileA.txt правки
# запросили статус
$ git status
On branch master
Changes not staged for commit: # ещё одна секция
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   fileA.txt
```

5. modified файл(ы) добавлен на сцену

```
$ git add fileA.txt
$ git status
On branch master
Changes to be committed: # все изменения готовы к коммиту
  (use "git restore --staged <file>..." to unstage)
        modified:   fileA.txt
```

6. Файл добавлен на сцену, но после этого изменен

```
# изменили fileA.txt
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
          modified:   fileA.txt ##версия файла которая была добавлена на сцену

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:   fileA.txt ##измененная версия файла, которая была изменена
## после добавления на сцену, в данный момент она не входит в коммит
```

Конвенция о коммитах Conventional Commits

Исправление коммита

HEAD коммит можно исправить с помощью опции —amend. Можно добавить новые файлы к последнему коммиту и/или изменить сообщение коммита.

```
$ git commit --amend --no-edit ## добавляет staged файлы в HEAD коммит,
## при этом сообщение к коммиту не изменяется
```

Если необходимо также изменить сообщение коммита то вместо —*no-edit* нужно написать привычное -m ‘<Your commit message>’

Если необходимо только изменить сообщение к последнему коммиту, то необходимо воспользоваться следующей командой:

```
$ git commit --ammend -m '<Your commit message>'
```

Команды для отката состояния репозитория

Команда, которая убирает со сцены какой-либо файл:

```
$ git restore --staged <file>
```

Команда, которая откатывает репозиторий до коммита с хешем <commit hash>:

```
$ git reset --hard <commit hash>
```

Команда, которая откатывает изменения в файле до последней сохраненной версии (в коммите или на сцене)

```
$ git restore <file>
```

Просмотр изменений в файлах

Команда *git diff* позволяет посмотреть конкретные изменения в каком либо существующем коммите или в коммите, который в данный момент подготавливается.

Если мы изменим что-то в файле, который уже отслеживается репозиторием, то мы можем посмотреть изменения командой git diff:

```
$ git diff
diff --git a/firstFile.txt b/firstFile.txt
index c2a33f8..fb48983 100644
--- a/firstFile.txt
+++ b/firstFile.txt
@@ -1,2 +1,2 @@
 # Теремок стоит, и в нём:
-# никого нет
\ No newline at end of file
```

```
+# мышка-норушка
\ No newline at end of file
```

По умолчанию команда git diff работает только с modified файлами, которые еще не были добавлены на сцену, для просмотра modified файлов, которые находятся на сцене надо использовать команду git diff —staged.

Чтобы сравнить состояние двух коммитов надо использовать состояние *git diff <commit hash A> <commit hash B>*. По сути команда выводить список инструкций как превратить состояние A в состояние B.

.gitignore

Файл .gitignore содержит в себе названия файлов, которые находятся в папке репозитория, но не должны быть добавлены в репозиторий.

> 💡 Правила из .gitignore применяются только к новым (untracked) файлам. Если файл уже попал в staging area или в коммит, то правила на него не распространяются.

Для оформления файла .gitignore используются следующие правила:

Комментарий

Если строка начинается с #, то это комментарий, и .gitignore не будет его учитывать.

Просто название файла

Чтобы проигнорировать какой-то конкретный файл, нужно просто ввести его название.

> 💡 В таком случае Git будет игнорировать файлы с введенным именем не только в корне репозитория, но и во всех вложенных папках.

Звёздочка (*)

Символ звёздочки (*) соответствует любой строке, включая пустую. Если такой символ используется в шаблоне в .gitignore, значит, файл будет проигнорирован вне зависимости от того, что будет на месте звёздочки.

```
# игнорировать все файлы, которые заканчиваются на .jpeg
*.jpeg

# игнорировать все файлы "tmp" во всех подпапках папки docs
docs/*/tmp
```

Вопросительный знак (?)

Вопросительный знак ? соответствует одному любому символу.

Квадратные скобки ([…])

Квадратные скобки, как и вопросительный знак, соответствуют одному символу. При этом символ не любой, а только из списка, который указан в скобках.

```
# игнорировать файлы file0.txt, file1.txt и file2.txt
# при этом не игнорировать file3.txt, file4.txt, ...
file[0-2].txt
```

Слеш (/)

Косая черта, или слеш (/), указывает на каталоги. Если шаблон в .gitignore начинается со слеша, то Git проигнорирует файлы или каталоги только в корневой директории.

```
# игнорировать todo.txt в корне репозитория
/todo.txt

# для сравнения: spam.txt будет игнорироваться во всех папках
spam.txt
```

Если шаблон заканчивается слешем, то правило применится только к папке.

```
# игнорировать папку build
build/
```

Парные звёздочки (**)

Функция парных звёздочек (**) работатт с вложенными папками. Двойная звёздочка может соответствовать любому количеству таких папок (в том числе нулю). Одинарная может соответствовать только одной.

```
# игнорировать файлы "docs/current/tmp", "docs/old/tmp",
# а также "docs/old/saved/a/b/c/d/tmp"
# и даже "docs/tmp", потому что ноль вложенных папок тоже подходит
docs/**/tmp

# игнорировать только "docs/current/tmp" и "docs/old/tmp"
# файл "docs/old/saved/a/b/c/d/tmp" не попадает в правило
docs/*/tmp
```

Восклицательный знак (!)

Любое правило в файле .gitignore можно инвертировать с помощью восклицательного знака (!).

Клонирование репозитория и функция fork

С помощью команды *git clone <repository URL>* можно создать локальную копию удаленного репозитория, к тому же между локальным и удаленным репозиториями автоматически создастся связь.

Функция fork относится к GitHub, она позволяет создавать копию репозитория с GitHub и сохранять ее к себе в профиль. При этом fork создает уникальную копию репозитория, т.е вносимые изменения не повлияют на оригинальный репозиторий.

Комбинацию функций fork и клонирования удобно использовать при внесении изменений в публичные репозитории.

Работа с ветками

Ветки в git являются одним из главных инструментов ведения репозитория. Система ветвления позволяет добавлять новые функции в приложение и разбивать их на ветки, при этом сохраняя стабильную версию приложения в основной ветке main.

Звездочка (*) Показывает на текщую ветку.

Команда git branch позволяет посмотреть список веток репозитория и показывает текущую ветку. Если к этой команде добавить параметр в виде названия ветки, то будет создана новая ветка с данным именем.

Команда git checkout <branch-name> позволяет переключатся на ветку с данным названием. Если перед названием новой ветки добавить флаг -b, то репозиторий переключится на новую ветку сразу же после ее создания.

Ветка в Git — это указатель на коммит.
Когда вы делаете новый коммит в ветке, этот указатель передвигается вперёд.
Пока коммиты не будут внесены в новую ветку, она указывает на тот же коммит, что и основная ветка.

Так же ветки можно сравнивать между друг другом, точно так же как и коммиты - с помощью команды git diff. В случае со сравнением веток в качестве аргументов мы передаем ветки, которые надо сравнить. Также можно сравнивать ветки с

конкретными коммитами, если указать их хеш, а если передать HEAD в качестве аргумента, то сравнение ветки будет с последним коммитом.

Суффикс навигации ~

С помощью ~ можно удобно ссылаться на нужный коммит, чтобы постоянно не вводить хэши. Ссылка на коммит происходит относительно HEAD. Например, чтобы сослаться на коммит предыдущий перед HEAD нужно написать HEAD~1, а чтобы сослаться предпредыдущий коммит перед HEAD нужно написать HEAD~2 и тд.

HEAD~1 можно заменить на HEAD~

Объединение и удаление веток

Если версия проекта в отдельной ветке готова к тому, чтобы стать основной версией проекта, нужно произвести слияние веток. Как правило, побочные ветки объединяются с основной, однако можно объединять две любые ветки.

Перед тем как начать процесс слияния, нужно перейти в ветку, куда должны добавиться изменения. После этого надо выполнить команду:

```
$ git merge <branch name>
```

После объединения веток имеет смысл удалить ту, которая стала частью другой, так как они теперь указывают на один и тот же коммит и являются по сути одним целым. Удалением веток можно пользоваться и в других случаях. Для этого надо

убедиться, что в данный момент ты не находишься в той ветке, которую хочешь удалить, иначе произойдет ошибка и удаление не произойдет. Далее надо использовать команду:

```
$ git branch -D <branch name>
```

⚠️ Удаление ветки в локальном репозитории не удаляет ветку в удаленном репозитории.

Конфликт

При выполнении объединения веток могут происходить конфликты. Как правило, они происходят при работе в команде, когда работа ведется одновременно в разных ветках разными людьми и могут изменятся одни и те же файлы. При этом результаты таких модификаций оказались несовместимы и разобраться в том, какой из вариантов правильный, может только человек. В случае возникновения конфликта git выводит в терминал названия файлов, в которых произошел конфликт:

```
$ git commit -m "Исправить опечатки" && git checkout main
$ git merge edit/fix-typoes
Auto-merging pages/table-of-content.txt # тут Git самостоятельно внёс изменения
CONFLICT (content): Merge conflict in github.txt # здесь возник конфликт
Automatic merge failed; fix conflicts and then commit the result # слияния не произошло
```

Git - Продвинутое слияние

This book is available in English.

https://git-scm.com/book/ru/v2/Инструменты-Git-Продвинутое-слияние

Синхронизация локальных веток с удаленными

Если имеется локальный репозиторий с несколькими ветками, который необходимо связать с новым удаленным репозиторием, необходимо сначала создать новый удаленный репозиторий, а потом связать основную ветку локального репозитория с основной удаленного репозитория командой:

```
$ git push -u origin main
```

Чтобы связать другие ветки нужно выполнить эту же команду, только вместо main подставить название нужной ветки.

Pull request

pull request это механизм, который позволяет контролировать слияние веток в репозитории. Как правило, он применяется при командной работе над репозиторием, когда имеется основная ветка со стабильной версией и побочные ветки с экспериментальными функциями (как правило, каждый человек создает собственную ветку) и перед тем как совместить какую то ветку с основной необходимо произвести код-ревью. pull request по сути это запрос на слияние побочной ветки и основной.

> Алгоритм использования pull request такой:
> 1. Вы трудитесь над задачей в своей ветке — например, пишете код новой функциональности.
> 2. Вы заканчиваете работу, а затем создаёте пул-реквест.
> 3. Ваши коллеги проверяют, что код выглядит аккуратно и лаконично, а программа работает корректно; также оставляют комментарии. Этот процесс называют code review (англ. «рассмотрение кода»), или просто ревью.
> 4. После финального согласования вы заливаете свою ветку в основную.

> **У каждого пул-реквеста есть:**
> 1. **Название** — краткое описание предлагаемых изменений.
> 2. **Описание** — развёрнутое описание изменений. Это поле заполнять необязательно, но желательно.
> 3. **Исходная ветка** — та, в которой вы работали.
> 4. **Целевая ветка** — основная ветка проекта, в которую вы хотите внести изменения.

> Также у каждого пул-реквеста может быть два исхода:
> **merge** — предлагаемые изменения приняты; код вливается в целевую ветку; пул-реквест закрывается.
> **close** — пул-реквест закрывается без слияния изменений.

pull request создается через интерфейс GitHub.

Синхронизация локального репозитория с удаленным

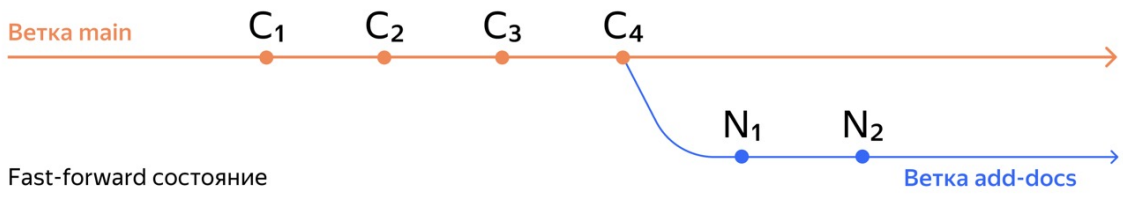
При работе в команде информация в репозитории постоянно меняется и поэтому локальный репозиторий может устаревать. Чтобы локальный репозиторий получил последние обновления из удаленного необходимо использовать команду:

```
$ git pull
```

Шпаргалка №3 (работа с ветками)

https://practicum.yandex.ru/trainer/git-basics/lesson/f22bb418-0a08-4aa7-b937-e21cc77c9298/

Состояние fast-forward

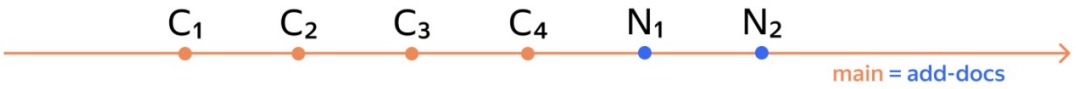


Состояние fast-forward описывает отношение двух веток, когда одна уходит вперед и другую надо будет “перематывать” до состояния первой при объединении. Так

же выполняются условия:

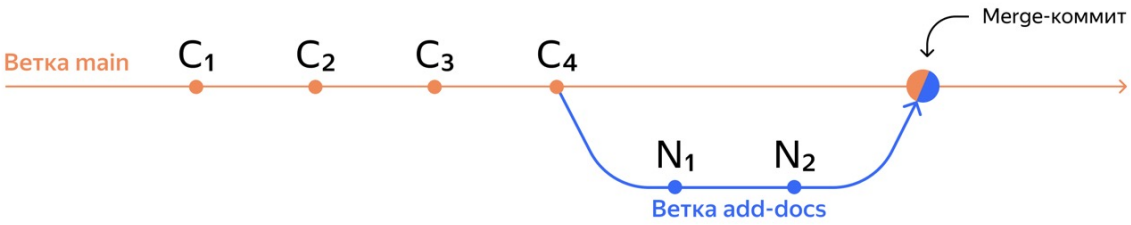
- При слиянии этих двух веток никак не возможен конфликт.
- Истории этих двух веток не «разошлись».
- Одна ветка является продолжением другой.

При объединении веток коммиты одной просто встают вперед ветки, которая отстала (можно сказать, что основная ветка перематывается до состояния побочной):



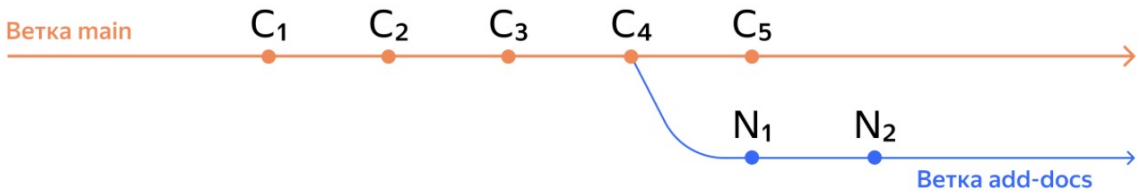
Режим fast-forward также можно и отключить, делается это для того, чтобы при объединении веток создавался отдельный merge commit, который бы указывал, что произошло слияние веток. В режиме fast-forward такого коммита не создается и соответсвенно через время не понятно, происходили ли слияния и какая ветка объединилась с основной.

При объединении режим fast-forward отключается флагом -no-ff. Также его можно отключить «навсегда» (до тех пор, пока вы не вернёте настройку «как было») с помощью настройки merge.ff: git config [--global] merge.ff false.



Состояние non-fast-forward

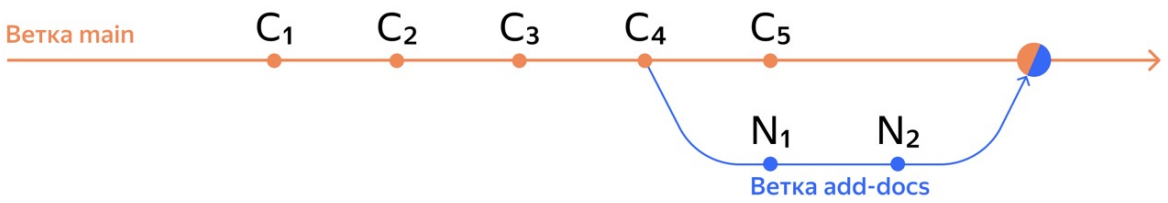
Когда истории двух веток разошлись, например в основной ветке появились коммиты после добавления побочной ветки, их больше нельзя выстроить в одну линию и теперь они в состоянии non-fast-forward.



В таком состоянии возможны конфликты при объединении.

⚠️ Когда Git проверяет ветки на состояние fast-forward, он не «заглядывает» в файлы и не пытается угадать, будет ли конфликт на самом деле. Для Git важно только, что конфликт теоретически возможен (или, наоборот, никак не возможен).

Однако конфликты не обязательно произойдут и сначала гит попытается автоматически объединить ветки, при этом создастся мердж коммит:

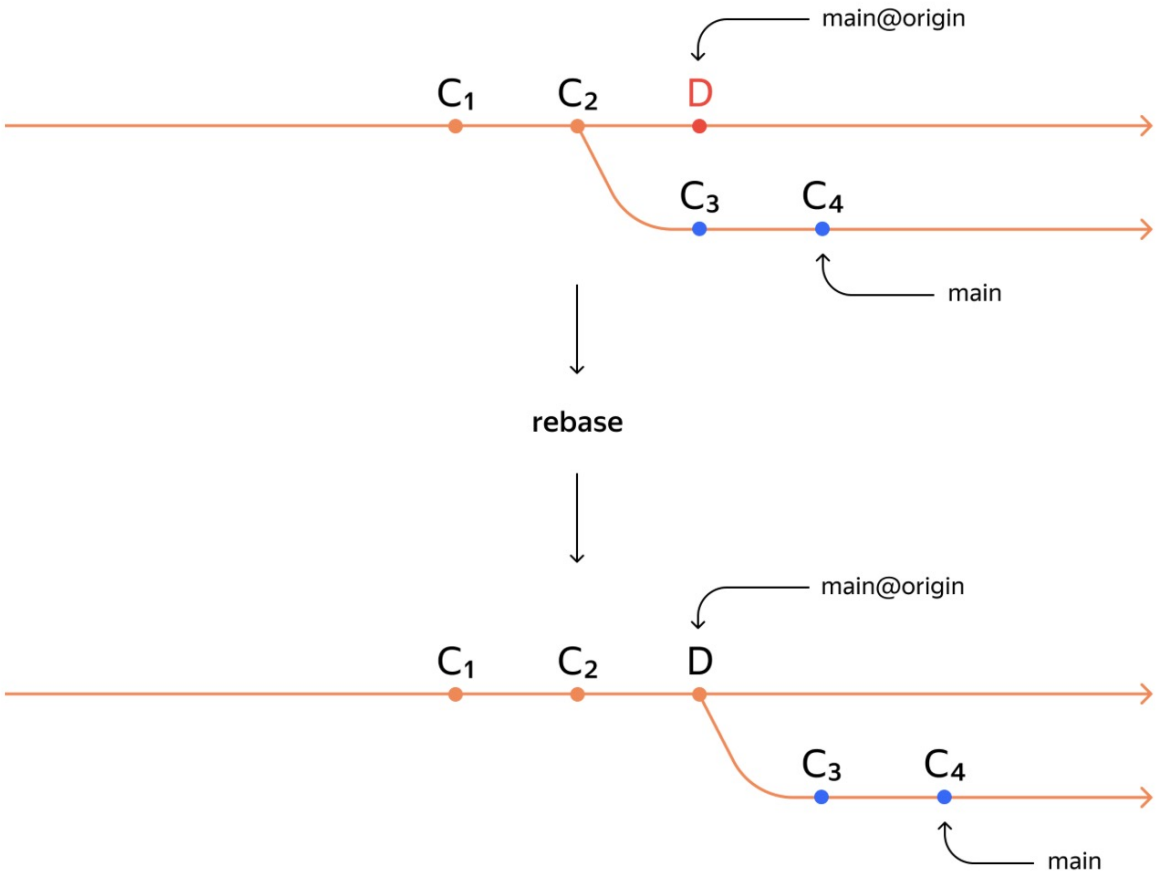


Если конфликты есть и гит не разрешил их автоматически придется решать их вручную.

Синхронизация локальных и удаленных веток

При пуше локальной ветки в удаленную ветки должны быть в состоянии fast-forward, иначе пуш не сможет быть произведен. Если ветки разошлись, то есть несколько способов вернуть состояние fast-forward. Сперва, необходимо перед началом работы “пулить” изменения, чтобы локальный репозиторий всегда был актуален. Если же в удаленной ветке появились изменения после начала работы можно либо: сделать rebase или применить git push —force.

Git rebase позволяет изменить точку отвлетвления:



После rebase ветки будут в состоянии fast-forward и пуши будут произведены успешно. Однако иногда rebase может приводить к проблемам, поэтому надо использовать этот инструмент с осторожностью.

git push —force метод, который приводит ветки в состояние fast-forward путем удаления коммитов в удаленном репозитории, которые вызывают состояние non-fast-forward (например коммит D на картинке выше).

Модели работы с ветками

В зависимости от команды и задач может существовать множество вариантов организации работы в репозитории. Однако, существуют три наиболее распространенных подхода — их также принято называть workflow или сокращённо: flow.

1. Feature branch workflow — простой и самый популярный вариант. Если коротко, в нём для каждого нового изменения создаётся новая ветка, которая позже вливается в main с помощью git merge.
2. Git flow — более сложный вариант. Подход похож на feature branch workflow, но в нём создаётся больше веток, а изменения (коммиты) делят на разные типы: исправление, новая функциональность и так далее. Разные типы коммитов попадают в разные ветки.
3. Trunk-based — популярный в больших компаниях (таких как Яндекс, Google и прочих) подход, который обещает бо́льшую скорость работы в крупных командах. Этот подход тоже похож на feature branch workflow. Главное отличие в том, что участники проекта вливают (merge) свой код в основную ветку максимально часто. Например, каждый день.

feature branch workflow

Основные правила:

Новая функциональность или исправление — новая ветка.
Когда код в feature-ветке готов, он вливается в main.
В main всегда рабочая версия.

Преимущества:
простая модель;
позволяет работать с Git в команде без лишних технических сложностей.

Жизненный цикл пул-реквеста

1. Автор создаёт пул-реквест.
2. Ревьюер просматривает изменения и предлагает правки, если они необходимы.
3. Автор вносит исправления по комментариям ревьюера.
4. Второй и третий пункты могут повториться!

5. Если ревьюера всё устраивает, он одобряет («апрувит») пул-реквест.
6. Вуаля! Теперь автор или ревьюер могут влить изменения в основную ветку.

Решение Конфликтов

Чтобы разрешить конфликт слияния, который возникает, когда главная ветка «уходит» вперёд, можно сделать следующее. Сначала локально получить новые изменения через git pull, а затем выполнить git merge и разрешить конфликт. Далее создать коммит слияния и отправить новые изменения без конфликтов обратно в удалённый репозиторий командой git push.

Шпаргалка №4 (командная работа)

https://practicum.yandex.ru/trainer/git-basics/lesson/cb64571e-3cc2-4259-8561-3e432b4b99c1/