learngitbranching.js.org

<https://gist.github.com/CraftingGamerTom/d334740303b647339b00875d27d38d34>

# Добро пожаловать в LearnGitBranching!

Это приложение создано, чтобы помочь новичкам постичь мощные возможности ветвления и работы с git. Мы надеемся, что вам понравится эта игра и может вы что-то усвоите!

## Демо!

Если ты не видел демонстрацию – посмотри её тут:

<https://pcottle.github.io/learnGitBranching/?locale=ru_RU&demo>

Достало это сообщение? Добавь NODEMO к адресу и навсегда забудь о нём, ниже ссылка для удобства:

[https://pcottle.github.io/learnGitBranching/?locale=ru\_RU&NODEMO](https://learngitbranching.js.org/?locale=ru_RU&NODEMO)

## Команды Git

В нашей песочнице можно использовать множество команд:

* commit
* branch
* checkout
* cherry-pick
* reset
* revert
* rebase
* merge

## Бог велел – делись!

Ты можешь делиться результатами с друзьями при помощи export tree и import tree.

Хочешь создать классный уровень? Сделай это при помощи build level или добавь уровень друга при помощи import level.

Команда show commands покажет все доступные инструкции. Там есть очень полезные, например undo и reset.

А пока просто начни игру при помощи levels...

# Основы

## 1. Введение

Хорошо подобранное введение в основные команды git

### 1.1: Знакомство с Git Commit

#### Коммиты в GIT

Коммит в git репозитории хранит снимок всех файлов в директории. Почти как огромная копия, только лучше.

Git пытается быть лёгким и быстрым насколько это только возможно, так что он не просто слепо копирует всю директорию каждый раз, а ужимает (когда это возможно) коммит в набор изменений или «дельту» между текущей версией и предыдущей.

Также Git хранит всю историю о том, когда какой коммит был сделан. Вот почему большинство коммитов имеют предков - мы указываем на предков стрелками при визуализации. Поддержка истории коммитов более чем важна для всех, кто работает над проектом!

Можно ещё долго рассказывать о коммитах, но для простоты будем считать их полными снимками проекта. Коммиты очень легки, так что переключение между ними происходит предельно быстро!

Посмотрим, как это выглядит на практике. Справа расположена визуализация небольшого git репозитория. Сейчас в нём два коммита: первый, исходный коммит С0 и один коммит С1 после него, содержащий изменения.

Нажми на кнопку, чтобы совершить коммит

|  |  |
| --- | --- |
| git commit | |
|  |  |

Отлично. Мы только что внесли изменения в репозиторий и сохранили их как коммит. У коммита, который мы только что сделали, есть родитель, С1, который указывает на предыдущий коммит.

Пора попробовать! После того, как это окно закроется, сделай два коммита, чтобы пройти этот уровень.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| $ git commit; git commit |

#### Супер!

Ты прошёл уровень. Количество использованных команд - 2 ; а наше решение состоит из 2.

Отлично! Ваше решение соответствует или превосходит наше.

Хотите перейти на следующий уровень: "Ветвление в Git"?

### 1.2: Ветвление в Git

#### Ветвление в Git

Ветки в Git, как и коммиты, невероятно легковесны. Это просто ссылки на определённый коммит — ничего более. Вот почему многие фанаты Git повторяют мантру

делай ветки сразу, делай ветки часто

Так как создание множества веток никак не отражается на памяти или жестком диске, удобнее и проще разбивать свою работу на много маленьких веток, чем хранить все изменения в одной огромной ветке.

Чуть позже мы попробуем использовать ветки и коммиты, и вы увидите, как две эти возможности сочетаются. Можно сказать, что созданная ветка хранит изменения текущего коммита и всех его родителей.

Посмотрим, что такое ветки на практике

Создадим здесь новую ветку с именем newImage.

|  |  |
| --- | --- |
| git branch newImage | |
|  |  |

Вот и всё, ребята! Ветка newImage теперь указывает на коммит C1.

Теперь попробуем сделать некоторые изменения в этой ветке. Для этого нажми кнопку ниже.

|  |  |
| --- | --- |
| git branch newImage | |
|  |  |

О-оу! Ветка main сдвинулась, тогда как ветка newImage - нет! Всё из-за того, что мы не переключились на новую ветку, а остались в старой, о чём говорит звёздочка около ветки main.

Сообщим Git, что хотим выбрать ветку

git checkout [name]

Эта команда перенесёт нас на новую ветку в момент, когда мы ещё не коммитили изменения

|  |  |
| --- | --- |
| git checkout newImage; git commit | |
|  |  |

Вот так! Наши изменения записаны уже в новую ветку

Ну что ж, теперь ты готов к работе с ветками. Как только это окно закроется, создай ветку с именем bugFix и переключись на неё.

Кстати вот тебе совет, ты можешь создать новую ветку и переключиться на неё с помощью одной команды: git checkout -b [yourbranchname].

|  |  |  |
| --- | --- | --- |
|  |  |  |
| $ git branch bugfix;  $ git checkout bugFix |

#### Супер!

Ты прошёл уровень. Количество использованных команд - 2 ; а наше решение состоит из 2.

Отлично! Ваше решение соответствует или превосходит наше.

Хотите перейти на следующий уровень: "Слияния веток в Git"?

### 1.3: Слияние веток в Git

#### Ветки и слияния

Ок! Мы уже знаем, как создавать ветки и коммитить наши изменения. Теперь надо понять, как объединять изменения из двух разных веток. Очень удобно создать ветку, сделать свою часть работы в ней и потом объединить изменения из своей ветки с общими.

Первый способ объединения изменений, который мы рассмотрим – это git merge – слияние или просто мердж. Слияния в Git создают особый вид коммита, который имеет сразу двух родителей. Коммит с двумя родителями обычно означает, что мы хотим объединить изменения из одного коммита с другим коммитом и всеми их родительскими коммитами.

Слишком запутанно =) На схеме всё проще и понятнее.

Вот у нас две ветки, каждая содержит по одному уникальному коммиту. Это означает, что ни одна из веток не содержит полный набор "работ", выполненных в этом репозитории. Можно исправить эту ситуацию, выполнив слияние.

Мы сделаем merge ветки bugFix в ветку main.

|  |  |
| --- | --- |
| git merge bugFix | |
|  |  |

Что мы видим? Во-первых, ветка main теперь указывает на коммит, у которого два родителя. Если проследовать по стрелкам от этого коммита, вы пройдёте через каждый коммит в дереве прямиком к началу. Это означает, что теперь в ветке main содержатся все изменения репозитория.

Во-вторых, обрати внимание, как изменились цвета коммитов. Мы ввели цветовую дифференциацию, чтобы помочь пониманию. У каждой ветки – свой цвет. Каждый коммит становится того цвета, какого его ветка. Если в нём изменения сразу двух веток – он становится цветом, смешанным из цветов родительских веток.

И вот мы видим, что цвет ветки main подмешан к каждому коммиту, а ветки bugFix – нет. Это можно поправить.

Смерджим ветку main в ветку bugFix.

|  |  |
| --- | --- |
| git checkout bugfix; git merge main | |
|  |  |

Так как ветка bugFix была предшественницей main, Git не делал ничего, только сдвинул bugFix на тот же коммит, где находится main.

Теперь все коммиты одного цвета, что означает, что каждая ветка содержит все изменения репозитория! Поздравляем!

Чтобы пройти этот уровень, сделай следующее:

* Создай новую ветку под названием bugFix
* Переключись на новую ветку bugFix командой git checkout bugFix
* Сделай один коммит
* Вернись на ветку main при помощи git checkout
* Сделай ещё один коммит
* Слей ветку bugFix с веткой main при помощи git merge
* Если что-то пошло не так - можешь подглядеть в эту шпаргалку командой "objective"!\*

|  |  |  |
| --- | --- | --- |
|  |  |  |
| git checkout -b bugFix; git commit  git checkout main; git commit  git merge bugFix |

#### Супер!

Ты прошёл уровень. Количество использованных команд - 5 ; а наше решение состоит из 5.

Отлично! Ваше решение соответствует или превосходит наше.

Хотите перейти на следующий уровень: "Введение в rebase"?

### 1.4: Введение в rebase

#### Git Rebase

Второй способ объединения изменений в ветках – это rebasing. При ребейзе Git по сути копирует набор коммитов и переносит их в другое место.

Несмотря на то, что это звучит достаточно непонятно, преимущество rebase в том, что c его помощью можно делать чистые и красивые линейные последовательности коммитов. История коммитов будет чище, если вы применяете rebase.

Посмотрим, как это работает...

У нас здесь снова две ветки. Обрати внимание, что выбрана ветка bugFix (отмечена звёздочкой)

Хочется сдвинуть наши изменения из bugFix прямо на вершину ветки main. Благодаря этому всё будет выглядеть, как будто эти изменения делались последовательно, хотя на самом деле - параллельно.

Применим git rebase.

|  |  |
| --- | --- |
| Git rebase main | |
|  |  |

Супер! Теперь изменения из bugFix находятся в конце ветки main и являют собой линейную последовательность коммитов.

Обрати внимание, что коммит С3 до сих пор существует где-то, а С3' - это его "копия" в ветке main

Единственная проблема - ветка main не обновлена до последних изменений. Это легко исправить.

Вот мы выбрали ветку main. Вперёд - сделаем rebase на bugFix.

|  |  |
| --- | --- |
| Git rebase main | |
|  |  |

Вуаля! Так как main был предком bugFix, git просто сдвинул ссылку на main вперёд.

Чтобы пройти этот уровень, сделай следующее:

* Переключись на ветку bugFix
* Сделай коммит
* Вернись на main и сделай коммит ещё раз
* Переключись на bugFix и сделай rebase на main

Удачи!

|  |  |  |
| --- | --- | --- |
|  |  |  |
| git checkout -b bugFix; git commit  git checkout main; git commit  git checkout bugFix; git rebase main |

#### Супер!

Ты прошёл уровень. Количество использованных команд - 6 ; а наше решение состоит из 6.

Отлично! Ваше решение соответствует или превосходит наше.

Хотите перейти на следующий уровень: "Теряем голову, или detached HEAD"?

## 2. Едем дальше

Следующая порция абсолютной git-крутотенюшки. Проголодались?

### 2.1: Теряем голову, или detached HEAD

#### Прогулка по Git

Прежде чем перейти к более продвинутым фичам Git, важно понять различные способы перемещения по дереву коммитов вашего проекта.

Как только вы научитесь свободно передвигаться по дереву коммитов, ваши возможности в Git приумножатся.

#### HEAD

В первую очередь, поговорим о "HEAD". HEAD - это символическое имя текущего выбранного коммита — это, по сути, тот коммит, над которым мы в данным момент работаем.

HEAD всегда указывает на последний коммит из вашего локального дерева. Большинство команд Git, изменяющих рабочее дерево, начнут с изменения HEAD.

Обычно HEAD указывает на имя ветки (например, bugFix). Когда вы делаете коммит, статус ветки bugFix меняется и это изменение видно через HEAD.

Посмотрим, как это работает. Обратите внимание на то, где находится HEAD до и после коммита.

|  |  |
| --- | --- |
| git checkout C1; git checkout main; git commit; git checkout C2 | |
|  |  |

Вот! HEAD всё это время скрывался за веткой main.

#### Detaching HEAD

Отделение (detaching) HEAD означает лишь присвоение его не ветке, а конкретному коммиту. Посмотрим, что было до отделения:

HEAD -> main -> C1

|  |  |
| --- | --- |
| git checkout C1 | |
|  |  |

А вот что получилось теперь

HEAD -> C1

Чтобы пройти уровень, давай отделим HEAD от ветки bugFix и присвоим его последнему коммиту в этой же ветке.

Укажи коммит при помощи его идентификатора (hash). Hash для каждого коммита указан в кружке на схеме.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| git checkout C4 |

#### Супер!

Ты прошёл уровень. Количество использованных команд - 1 ; а наше решение состоит из 1.

Отлично! Ваше решение соответствует или превосходит наше.

Хотите перейти на следующий уровень: "Относительные ссылки (^)"?

### 2.2: Относительные ссылки (^)

#### Относительные ссылки

Передвигаться по дереву Git при помощи указания хешей коммитов немного неудобно. В реальной ситуации у вас вряд ли будет красивая визуализация дерева в терминале, так что придётся каждый раз использовать git log, чтобы найти хеш нужного коммита

Более того, хеши в реальном репозитории Git намного более длинные. Например, хеш для коммита, который приведён в предыдущем уровне - fed2da64c0efc5293610bdd892f82a58e8cbc5d8. Не очень просто для произношения =)

Хорошая новость в том, что Git достаточно умён в работе с хешами. Ему нужны лишь первые несколько символов для того, чтобы идентифицировать конкретный коммит. Так что можно написать просто fed2 вместо колбасы выше.

Как мы уже говорили, указание на коммит при помощи его хеша - не самый удобный способ, поэтому Git поддерживает относительные ссылки и они прекрасны!

С относительными ссылками можно начать с какого-либо удобного места (например, с ветки bugFix или от HEAD) и двигаться от него

Относительные ссылки - мощный инструмент, но мы покажем два простых способа использования:

* Перемещение на один коммит назад ^
* Перемещение на несколько коммитов назад ~<num>

Для начала рассмотрим оператор каретки (^). Когда мы добавляем его к имени ссылки, Git воспринимает это как указание найти родителя указанного коммита.

Так что main^ означает "первый родитель ветки main".

main^^ означает прародитель (родитель родителя) main

Давайте переключимся на коммит Выше main

|  |  |
| --- | --- |
| git checkout main^ | |
|  |  |

Опачки! Готово. Сильно проще, чем поиск и указание хеша.

Можно также использовать HEAD как относительную ссылку. Попробуем пройти несколько раз назад по дереву коммитов

|  |  |
| --- | --- |
| git checkout C3; git checkout HEAD^; git checkout HEAD^; git checkout HEAD^ | |
|  |  |

Изи! Мы можем путешествовать во времени при помощи HEAD^

Чтобы пройти этот уровень, переместись на первого родителя ветки bugFix. Это отделит HEAD от ветки.

Конечно, можно указать хеш, но надо попробовать использовать относительные ссылки!

|  |  |  |
| --- | --- | --- |
|  |  |  |
| git checkout C3 |

#### Супер!

Ты прошёл уровень. Количество использованных команд - 1 ; а наше решение состоит из 1.

Отлично! Ваше решение соответствует или превосходит наше.

Хотите перейти на следующий уровень: "Относительные ссылки №2"?

### 2.3: Относительные ссылки №2

#### Оператор "~"

Предположим, нужно переместиться на много шагов назад по дереву. Было бы неудобно печатать ^ несколько раз (или несколько десятков раз), так что Git поддерживает также оператор тильда (~).

К тильде (опционально) можно добавить количество родительских коммитов, через которые нужно пройти. Посмотрим, как это работает.

Укажем после ~ число коммитов, через которые надо пройти.

|  |  |
| --- | --- |
| git checkout HEAD~4 | |
|  |  |

Оп! Очевидно, относительные ссылки прекрасны.

#### Перемещение ветки (branch forcing)

Теперь мы разбираемся в относительных ссылках, так что можно реально использовать их для дела.

Одна из наиболее распространённых целей, для которых используются относительные ссылки - это перемещение веток. Можно напрямую прикрепить ветку к коммиту при помощи опции -f. Например, команда:

git branch -f main HEAD~3

Переместит (принудительно) ветку main на три родителя назад от HEAD.

Посмотрим, как работает эта команда

|  |  |
| --- | --- |
| git branch -f main HEAD~3 | |
|  |  |

Вуаля! Относительная ссылка дала нам возможность просто сослаться на C1, а branch forcing (-f) позволил быстро переместить указатель ветки на этот коммит.

Мы рассмотрели относительные ссылки и branch forcing вкупе, так что теперь пришло время пройти следующий уровень.

Чтобы пройти этот уровень, передвинь HEAD, main и bugFix так, как показано на визуализации.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| git checkout HEAD^;  git branch -f main C6;  git branch -f bugFix HEAD^ |

#### Супер!

Ты прошёл уровень. Количество использованных команд - 3 ; а наше решение состоит из 3.

Отлично! Ваше решение соответствует или превосходит наше.

Хотите перейти на следующий уровень: "Отмена изменений в Git"?

### 2.4: Отмена изменений в Git

#### Отмена изменений в Git

Есть много путей для отмены изменений в Git. Так же как и коммит, отмена изменений в Git возможна и на низком уровне (добавление в коммит отдельных файлов и наборов строк), и на высоком (как изменения реально отменяются). Сейчас сфокусируемся на высокоуровневой части.

Есть два основных способа отмены изменений в Git: первый - это git reset, а второй - git revert. Попробуем оба на следующем шаге.

#### Git Reset

git reset отменяет изменения, перенося ссылку на ветку назад, на более старый коммит. Это своего рода "переписывание истории"; git reset перенесёт ветку назад, как будто некоторых коммитов вовсе и не было.

Посмотрим, как это работает:

|  |  |
| --- | --- |
| git reset HEAD~1 | |
|  |  |

Неплохо! Git просто перенёс ссылку на main обратно на коммит C1. Теперь наш локальный репозиторий в состоянии, как будто C2 никогда не существовал.

#### Git Revert

Reset отлично работает на локальных ветках, в локальных репозиториях. Но этот метод переписывания истории не сработает на удалённых ветках, которые используют другие пользователи.

Чтобы отменить изменения и поделиться отменёнными изменениями с остальными, надо использовать git revert. Посмотрим, как это работает

|  |  |
| --- | --- |
| git revert HEAD | |
|  |  |

Забавно, появился новый коммит. Дело в том, что новый коммит C2' просто содержит изменения, полностью противоположные тем, что сделаны в коммите C2.

После revert можно сделать push и поделиться изменениями с остальными.

Чтобы пройти этот уровень, отмени два последних коммита и в local, и в pushed

Помни, что pushed - это remote ветка, а local - это локальная ветка. Это поможет выбрать способ отмены изменений.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| git reset HEAD^;  git checkout pushed;  git revert pushed |

#### Супер!

Ты прошёл уровень. Количество использованных команд - 3 ; а наше решение состоит из 3.

Отлично! Ваше решение соответствует или превосходит наше.

Хотите перейти на следующий уровень: "Введение в Cherry-pick"?

## 3. Перемещаем труды туда-сюда

Не стесняйтесь менять историю

### 3.1: Введение в Cherry-pick

#### Поперемещаем изменения

Итак, мы уже освоили основы Git: коммиты, ветки, перемещение по дереву изменений. Уже этих знаний достаточно, чтобы овладеть 90% мощью Git-репозиториев и покрыть нужды разработчиков.

А оставшиеся 10% будут очень полезны при сложных workflow (или если ты попал в сложную ситуацию). Теперь речь пойдёт о перемещении изменений — возможности, позволяющей разработчику сказать "Хочу, чтобы эти изменения были вот тут, а вот эти — вон там" и получить точные, правильные результаты, не теряя при этом гибкости разработки.

На первый взгляд запутанно, но на самом деле всё просто.

#### Git Cherry-pick

Первая из таких команд - это git cherry-pick. Она выглядит вот так:

* git cherry-pick <Commit1> <Commit2> <...>

Это очень простой и прямолинейный способ сказать, что ты хочешь копировать несколько коммитов на место, где сейчас находишься (HEAD). Мы обожаем cherry-pick за то, что в нём очень мало магии и его очень просто понять и применять.

Посмотрим на демонстрацию.

Вот репозиторий, где есть некие изменения в ветке side, которые мы хотим применить и в ветку main. Мы можем сделать это при помощи команды rebase, которую мы уже прошли, но давай посмотрим, как cherry-pick справится с этой задачей.

|  |  |
| --- | --- |
| git cherry-pick C2 C4 | |
|  |  |

Вуаля! Мы хотели перенести коммиты C2 и C4, Git дал нам их там, где они нужны. Всё просто!

Чтобы пройти этот уровень, просто скопируй изменения из этих трёх веток в мастер. Чтобы понять, какие коммиты копировать, посмотри на визуализацию уровня.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| git cherry-pick C3 C4 C7 |

#### Супер!

Ты прошёл уровень. Количество использованных команд - 1 ; а наше решение состоит из 1.

Отлично! Ваше решение соответствует или превосходит наше.

Хотите перейти на следующий уровень: "Введение в интерактивный Rebase"?

### 3.2: Введение в интерактивный Rebase

#### Git Interactive Rebase

Git cherry-pick прекрасен, когда точно известно, какие коммиты нужны (и известны их точные хеши)

Но как быть в случае, когда точно не известно какие коммиты нужны? К счастью, Git позаботился о таких ситуациях! Можно использовать интерактивный rebase для этого - лучший способ отобрать набор коммитов для rebase.

Углубимся в детали.

Всё, что нужно для интерактивного rebase - это опция -i

Если добавить эту опцию, Git откроет интерфейс просмотра того, какие коммиты готовы к копированию на цель rebase (target). Также показываются хеши коммитов и комментарии к ним, так что можно легко понять что к чему.

Для "реального" Git, этот интерфейс означает просто открытие файла в редакторе типа vim. Для этой обучалки, я сделал небольшое диалоговое окно, которое по сути делает то же, что и редактор.

После открытия окна интерактивного rebase есть три варианта для каждого коммита:

* Можно сменить положение коммита по порядку, переставив строчку с ним в редакторе (у нас в окошке строку с коммитом можно перенести просто мышкой).
* Можно "выкинуть" коммит из ребейза. Для этого есть pick - переключение его означает, что нужно выкинуть коммит.
* Наконец, можно соединить коммиты. В этом уровне игры у нас не реализована эта возможность, но, вкратце, при помощи этой функции можно объединять изменения двух коммитов.

Ну что ж, посмотрим на примеры!

После нажатия на эту кнопку появится окно интерактивного rebase. Переставь несколько коммитов (или удали кое-какие) и посмотри, что получится в итоге!

|  |  |
| --- | --- |
| git rebase -I HEAD~4 | |
|  |  |
| Rebasing 4 Commits  (Drag and drop to re-order. Toggle the “Omit/Pick” button to omit or re-add a commit) | |

Бах! Git скопировал коммиты в точности так, как было указано через интерфейс!

Чтобы пройти этот уровень, переставь коммиты при помощи интерактивного rebase в таком порядке, как указано на визуализации. На всякий случай, помни, что всегда можно исправить ошибку, вызвав команду undo или reset.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| git rebase -i main~4 --aboveAll; |

#### Супер!

Ты прошёл уровень. Количество использованных команд - 1 ; а наше решение состоит из 1.

Отлично! Ваше решение соответствует или превосходит наше.

Хотите перейти на следующий уровень: "Выберем один коммит."?

## 4. Сборная солянка

Ассорти из приёмов работы с Git, хитростей и советов

### 4.1: Выберем один коммит

Вот ситуация, которая часто случается при разработке: мы пытаемся отследить ошибку, но она не очень очевидна. Для того, чтобы достичь успеха на этом поприще, мы используем несколько команд для отладки и вывода

Каждая отладочная команда (команды) вывода находится в своём коммите. В итоге мы нашли ошибку, исправили её и порадовались!

Но проблема в том, что мы хотим добавить в main только исправление ошибки из ветки bugFix. Если мы воспользуемся простым fast-forward, то в main попадут также отладочные команды. Должен быть другой способ...

Надо заставить git копировать только один из коммитов. Это почти как в предыдущем уровне – мы можем использовать уже известные нам команды:

* git rebase -i
* git cherry-pick

Чтобы достичь желаемого результата.

В этом уровне тебе решать, какую команду использовать, но чтобы закончить уровень, убедись, что в мастер попал коммит, на который ссылается bugFix.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| git checkout main; git cherry-pick c4 |

#### Супер!

Ты прошёл уровень. Количество использованных команд - 2 ; а наше решение состоит из 2.

Отлично! Ваше решение соответствует или превосходит наше.

Хотите перейти на следующий уровень: "Жонглируем коммитами"?

### 4.2: Жонглируем коммитами

#### Жонглируем коммитами

Вот ещё одна ситуация, которая часто случается. Есть некоторые изменения (newImage) и другие изменения (caption), которые связаны так, что находятся друг поверх друга в репозитории.

Штука в том, что иногда нужно внести небольшие изменения в более ранний коммит. В таком случае надо немного поменять newImage, несмотря на то, что коммит уже в прошлом!

Преодолеть эти трудности можно следующим образом:

* Переставить коммит так, чтобы нужный находился наверху при помощи git rebase -i
* Внести изменения при помощи git commit --amend
* Переставить всё обратно при помощи git rebase -i
* И наконец, переместить main на изменённую часть дерева, чтобы закончить уровень.

Это задание можно выполнить несколькими способами (и, гляжу, ты посматриваешь на cherry-picking), но сейчас сосредоточься на вышеописанном методе.

Обрати внимание на итоговое состояние в этом уровне – так как мы дважды перемещаем коммиты, оба они получат по апострофу. Ещё один апостроф добавляется, когда мы делаем git commit --amend.

Важно, чтобы совпадало не только дерево коммитов, но и количество апострофов.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| git rebase -i caption~2 --aboveAll;  git commit --amend;  git rebase -i caption~2 --aboveAll;  git branch -f main caption |
|  | | |

#### Супер!

Ты прошёл уровень. Количество использованных команд - 4 ; а наше решение состоит из 4.

Отлично! Ваше решение соответствует или превосходит наше.

Хотите перейти на следующий уровень: "Жонглируем коммитами №2"?

### 4.3: Жонглируем коммитами №2

#### Жонглируем коммитами №2

Перед прохождением этого уровня обязательно надо пройти предыдущий уровень – 'Жонглируем коммитами №1'

В прошлом уровне мы использовали rebase -i, чтобы переставлять коммиты. Как только нужный нам коммит оказывался в конце, мы могли спокойно изменить его при помощи --amend и переставить обратно.

Единственная проблема тут - это множество перестановок, которые могут спровоцировать конфликты. Посмотрим, как с этой же задачей справится cherry-pick.

Важно помнить, что cherry-pick поместит любой коммит сразу после HEAD (только если этот коммит не является предком HEAD)

Вот небольшое демо для напоминания:

|  |  |
| --- | --- |
| git cherry-pick C2 | |
|  |  |

Ок! Едем дальше!

Итак, в этом уровне нужно достичь того же эффекта, но без использования rebase -i. Остальное – по усмотрению.

Важно, чтобы совпадало не только дерево коммитов, но и количество апострофов.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| git checkout main;  git cherry-pick c2;  git cherry-pick c3 |

#### Супер!

Ты прошёл уровень. Количество использованных команд - 3 ; а наше решение состоит из 4.

Отлично! Ваше решение соответствует или превосходит наше.

Хотите перейти на следующий уровень: "git tag"?

### 4.4: git tag

#### Теги

В прошлых уроках мы усвоили, что ветки просто двигать туда-сюда и они часто ссылаются на разные коммиты как на изменения данных в ветке. Ветки просто изменить, они часто временны и постоянно меняют своё состояние.

В таком случае, где взять постоянную ссылку на момент в истории изменений? Для таких вещей, как релиз и большие слияния, нужно нечто более постоянное, чем ветка.

Такое средство имеется. Git предоставляет нам теги, чья основная задача – ссылаться постоянно на конкретный коммит.

Важно, что после создания они никогда не сменят своего положения, так что можно с лёгкостью сделать checkout конкретного момента в истории изменений

Посмотрим на это на практике.

Создадим тег на C1, который будет нашей версией 1

|  |  |
| --- | --- |
| git tag v1 C1 | |
|  |  |

Готово! Всё просто. Мы назвали тег v1 и заставили его ссылаться на C1 явным образом. Если конкретный коммит не указан, гит пометит тегом HEAD.

Чтобы пройти этот уровень, просто создай теги так, как показано на визуализации, и потом перейди на тег v1. Обрати внимание, что ты перейдёшь в состояние detached HEAD, так как нельзя сделать коммит прямо в тег v1.

В следующем уровне мы попробуем более интересные способы применения тегов.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| git tag v0 c1;  git tag v1 c2;  git checkout c2 |

#### Супер!

Ты прошёл уровень. Количество использованных команд - 3 ; а наше решение состоит из 3.

Отлично! Ваше решение соответствует или превосходит наше.

Хотите перейти на следующий уровень: "Git describe"?

### 4.5: Git describe

#### Git Describe

Теги являются прекрасными ориентирами в истории изменений, поэтому в git есть команда, которая показывает, как далеко текущее состояние от ближайшего тега. И эта команда называется git describe

Git describe помогает сориентироваться после отката на много коммитов по истории изменений. Такое может случиться, когда вы сделали git bisect или если вы недавно вернулись из отпуска =)

Git describe выглядит примерно так:

git describe <ref>

Где ref — это что-либо, что указывает на конкретный коммит. Если не указать ref, то git будет считать, что указано текущее положение (HEAD).

Вывод команды выглядит примерно так:

<tag>\_<numCommits>\_g<hash>

Где tag – это ближайший тег в истории изменений, numCommits – это на сколько далеко мы от этого тега, а hash – это хеш коммита, который описывается.

Посмотрим на простой пример. Для дерева, показанного ниже:

|  |  |
| --- | --- |
| git tag v2 C3 | |
|  |  |

Команда git describe main выведет:

v1\_2\_gC2

Тогда как git describe side выведет:

v2\_1\_gC4

Это, в общем-то, всё, что можно сказать про git describe. Попробуй выполнить команду на нескольких коммитах.

Как только наиграешься, просто сделай один коммит, и уровень будет пройден.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| git commit |

#### Супер!

Ты прошёл уровень. Количество использованных команд - 1 ; а наше решение состоит из 1.

Отлично! Ваше решение соответствует или превосходит наше.

Хотите перейти на следующий уровень: "Rebase over 9000 раз"?

## 5. Продвинутый уровень

Если ты смелый, ловкий, умелый – потренируйся тут

### 5.1: Rebase over 9000 раз

#### Rebase на нескольких ветках

У нас тут куча веток! Было бы круто перенести все изменения из них в мастер.

Но начальство усложняет нашу задачу тем, что желает видеть все коммиты по порядку. Так что коммит С7' должен идти после коммита С6' и так далее.

Если что-то пойдёт не так – не надо стесняться использовать reset, чтобы начать всё с чистого листа. Постарайся сделать как можно меньше манипуляций!

|  |  |  |
| --- | --- | --- |
|  |  |  |
| git rebase main bugFix;  git rebase bugFix side;  git rebase side another;  git rebase another main |

#### Супер!

Ты прошёл уровень. Количество использованных команд - 4 ; а наше решение состоит из 4.

Отлично! Ваше решение соответствует или превосходит наше.

Хотите перейти на следующий уровень: "Здоровая семья, или несколько родителей"?

### 5.2: Здоровая семья, или несколько родителей

#### Определение родителей

Так же как тильда (~), каретка (^) принимает номер после себя.

Но в отличие от количества коммитов, на которые нужно откатиться назад (как делает ~), номер после ^ определяет, на какого из родителей мерджа надо перейти. Учитывая, что мерджевый коммит имеет двух родителей, просто указать ^ нельзя.

Git по умолчанию перейдёт на "первого" родителя коммита, но указание номера после ^ изменяет это поведение.

Посмотрим, как это работает.

Вот мерджевый коммит. Если мы перейдём на main^ без номера, то попадём на первого родителя.

(На нашей визуализации первый родитель находится прямо над коммитом)

|  |  |
| --- | --- |
| git checkout main^ | |
|  |  |

Просто - прямо как мы любим.

Теперь попробуем перейти на второго родителя.

|  |  |
| --- | --- |
| git checkout main^2 | |
|  |  |

Вот. Мы на втором родительском коммите.

Модификаторы ^ и ~ сильно помогают перемещаться по дереву коммитов:

|  |  |
| --- | --- |
| git checkout HEAD~; git checkout HEAD^2; git checkout HEAD~2 | |
|  |  |

Быстро как Флэш!

Более того, эти модификаторы можно применять вместе. Например, так:

|  |  |
| --- | --- |
| git checkout HEAD~^2~2 | |
|  |  |

Сделаем то же самое, что перед этим, только в одну команду.

#### На практике

Чтобы пройти этот уровень, создай ветку в указанном месте.

Очевидно, что (в данном случае) будет проще указать коммит напрямую, но для того, чтобы закрепить пройденное, используй модификаторы, о которых мы говорили выше.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| git branch bugWork main~^2~ |

#### Супер!

Ты прошёл уровень. Количество использованных команд - 1 ; а наше решение состоит из 1.

Отлично! Ваше решение соответствует или превосходит наше.

Хотите перейти на следующий уровень: "Спутанные ветки"?

### 5.3: Спутанные ветки

#### Спутанные ветки

УОУ! В этом уровне придётся попотеть!

У нас тут по несколько коммитов в ветках one, two и three. Не важно почему, но нам надо видоизменить эти три ветки при помощи более поздних коммитов из ветки main.

Ветка one нуждается в изменении порядка и удалении C5. two требует полного перемешивания, а three хочет получить только один коммит

Как пройти этот уровень – решать тебе, а как найдёшь решение – сравни его с нашим при помощи show solution.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| git checkout one;  git cherry-pick c4 c3 c2;  git checkout two;  git cherry-pick c5 c4 c3 c2;  git branch -f three c2 |

#### Супер!

Ты прошёл уровень. Количество использованных команд - 5 ; а наше решение состоит из 5.

Отлично! Ваше решение соответствует или превосходит наше.

Хотите перейти на следующий уровень: "Введение в клонирование"?

# Удалённые репозитории

## Puch & Pull – удалённые репозитории в Git!

Настало время поделиться своими единичками и нулями. Время коллективного программирования

### 1: Введение в клонирование

git clone

### 2: Удалённые ветки

git commit;

git checkout o/main;

git commit

### 3: Git fetch

git fetch

### 4: Git pull

git pull

### 5: Коллективная работа

git clone;

git fakeTeamwork main 2;

git commit;

git pull

### 6: Git push

git clone;

git commit;

git commit;

git push

### 7: Расхождение в истории

git clone;

git fakeTeamwork;

git commit;

git pull --rebase;

git push

### 8: Заблокированная ветвь main

git branch feature;

git checkout feature;

git branch -f main c1;

git push

## 2. Через origin – к звёздам. Продвинутое использование Git Remotes

### 1: Push Мастер!

### 2: Слияние с удалённым репозиторием

### 3: Слежка за удалённым репозиторием

### 4: Аргументы git push

### 5: Аргументы для push -- расширенная версия!

### 6: Аргументы для fetch

### 7: Пустой источник

### 8: Аргументы для pull

# Результаты

