## 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!

git commit

## 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 текущих коммитов и всех его родителей.

git branch newBranch – создание новой ветки.

git checkout newBranch – переход на ветку.

## 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.

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 master

git merge bugFix (при выполнение находясь в мастере)

 Git демо

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 master

git merge bugFix

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

**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.

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

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

Применим git rebase

git rebase master ( при этом мы находились в bugFix );

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

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

Единственная проблема - ветка master не обновлена до последних изменений. (мастер на c2 к нему добавился bugFix с3’(HEAD), который был отдельно на с3)Это легко исправить.

 Git демо

Вот мы выбрали ветку master(HEAD). Вперёд - сделаем rebase на bugFix

git rebase bugFix

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

ПРОГУЛКА ПО GIT

## 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.

 Git демо

### Detaching HEAD

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

HEAD -> master -> C1

git checkout c1

**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

## 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.

## Поперемещаем изменения

git cherry-pick

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.

Вот репозиторий, где есть некие изменения в ветке side, которые мы хотим применить и в ветку master. 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

В данном примере мы перенесли коммиты “с2” “с4 “ с ветки side в ветку master.(при этом мы находились в мастере HEAD)

## 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!

git rebase –i HEAD~4

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 master только исправление ошибки из ветки bugFix. Если мы воспользуемся простым fast-forward, то в master попадут также отладочные команды. Должен быть другой способ...

## 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!

## 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, 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

### Git Describe

Теги являются прекрасными ориентирами в истории изменений, поэтому в git есть команда, которая показывает, как далеко текущее состоянии 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.

### 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.

от мерджевый коммит. Если мы перейдём на master^ без номера, то попадём на первого родителя.

(*На нашей визуализации первый родитель находится прямо над коммитом*)

git checkout master^

Теперь попробуем перейти на второго родителя.

git checkout master^2

### Push & Pull - удалённые репозитории в Git!

## Удалённые репозитории в Git

На самом деле удалённые репозитории в Git не так сложны, как кажутся на первый взгляд. Кажется, что в современном мире облачных вычислений под термином «удалённый репозиторий» подразумевается что-то сложное и загадочное. Однако, удалённые репозитории — это всего-навсего копии вашего репозитория, хранящиеся на другом компьютере. Обычно вы можете связываться с этим другим компьютером через Интернет, что позволяет вам передавать коммиты туда и сюда.

Как уже было сказано, удалённые репозитории обладают рядом замечательных свойств:

* В первую очередь, удалённые репозитории - это замечательное средство резервного копирования! Насколько вам известно, локальные репозитории способны восстанавливать файлы, используя предыдущие состояния, но вся эта информация хранится локально. Потеряв все свои локальные данные, вы способны восстановить их при наличии копии своего репозитория на другом компьютере.
* Что ещё более важно, удалённые репозитории позволяют сделать процесс разработки более социальным! Теперь, когда копия вашего проекта размещена в другом месте, ваши друзья запросто могут внести свой вклад в ваш проект или забрать последние и актуальные изменения.

Набирает популярность использование web-сайтов для визуализации активности удалённых репозиториев (например, **[GitHub](https://github.com/)** или **[Phabricator](http://phabricator.org/)**), однако удалённые репозитории всегдавыступают в качестве базы для таких инструментов. Поэтому так важно понимать, как устроены удалённые репозитории!

## Наша команда для создания удалённого репозитория

До настоящего момента мы были сфокусированы на изучении основ работы с локальнымрепозиторием (ветвление, слияние, перемещение и т.д.). Однако теперь, когда мы хотим научиться работать с удалёнными репозиториями, нам нужны новые команды для настройки рабочей среды для этих упражнений. Такой командой нам послужит git clone

Технически, git clone в реальной жизни - это команда, которая создаст локальную копию удалённого репозитория (например, с GitHub).

## Удалённые ветки в Git

Теперь, когда вы уже увидели git clone в действии, давайте углубимся в детали и посмотрим что же на самом деле изменилось.

Во-первых, вы должны были заметить, что у нас в локальном репозитории появилась новая ветка с именем o/master. Такой тип ветки называется удалённой веткой. Поскольку удалённые ветки играют важную и уникальную роль, они обладают рядом специальных свойств.

Удалённые ветки отражают состояние удалённых репозиториев (с того момента, как вы обращались к этим удалённым репозиториям в последний раз). Они позволяют вам отслеживать и видеть разницу между вашими локальными наработками и тем, что было сделано другими участниками - важный шаг, который необходимо делать, прежде чем делиться своими наработками с другими.

Важным свойством удалённых веток является тот факт, что когда вы извлекаете их, вы отделяете (detaching) HEAD. Git делает это потому, что вы не можете работать непосредственно в этих ветках; сперва вам необходимо сделать наработки где-либо, а уж затем делиться ими с удалёнными репозиториями (после чего ваши удалённые ветки будут обновлены).

### Что такое o/ в названии ветки?

Вы, наверное, догадались, что первый символ o/ в названии ветки служит для обозначения именно удалённых веток. Да. Удалённые ветки также имеют (обязательное) правило именования - они отображаются в формате:

* <удалённый репозиторий>/<имя ветки>

Следовательно, если вы взглянете на имя ветки o/master, то здесь master - это имя ветки, а o- это имя удалённого репозитория.

Большинство разработчиков именуют свои главные удалённые репозитории не как o, а как origin. Также общепринятым является именование удалённого репозитория как origin, когда вы клонируете репозиторий командой git clone.

К сожалению, полное имя origin не помещается на элементах дизайна наших уроков, поэтому мы используем краткое o :( Просто помните, когда вы пользуетесь git в реальном проекте, ваш удалённый репозиторий скорее всего будет называться origin!

Давайте посмотрим на всё это в действии.

## Git Fetch

Работа с удалёнными git репозиториями сводится к передаче данных в и из других репозиториев. До тех пор, пока мы можем отправлять коммиты туда-обратно, мы можем делиться любыми изменениями, которые отслеживает git (следовательно, делиться новыми файлами, свежими идеями, любовными письмами и т.д.).

В этом уроке вы научитесь тому, как извлекать данные из удалённого репозитория - и для этого у нас есть соответствующая команда git fetch.

Вы увидите, что как только мы изменим представление нашего удалённого репозитория, наши удалённые ветки обновятся соответствующим образом и отобразят это представление. Это связывает воедино предыдущий урок про удалённые репозитории.

### Что делает fetch

git fetch выполняет две и только две основные операции. А именно:

* связывается с указанным удалённым репозиторием и забирает все те данные проекта, которых у вас ещё нет, при этом...
* у вас должны появиться ссылки на все ветки из этого удалённого репозитория (например, o/master)

Фактически, git fetch синхронизирует локальное представление удалённых репозиториев с тем, что является актуальным на текущий момент времени.

Насколько вы помните, в предыдущем уроке мы сказали, что удалённые ветки отображают состояние удалённых репозиториев на тот момент когда вы 'общались' с ними в последний раз. git fetch является тем механизмом, который даёт вам возможность общаться с удалёнными репозиториями! Надеюсь, что связь между удалёнными ветками и командой git fetch теперь прояснилась.

git fetch обычно 'общается' с удалёнными репозиториями посредством Интернета (через такие протоколы, как http:// или git://).

### Чего fetch не делает

Важно отметить, что команда git fetch забирает данные в ваш локальный репозиторий, но не сливает их с какими-либо вашими наработками и не модифицирует то, над чем вы работаете в данный момент.

Важно это помнить и понимать, потому что многие разработчики думают, что, запустив команду git fetch, они приведут всю свою локальную работу к такому же виду, как и на удалённом репозитории. Команда всего лишь скачивает все необходимые данные, но вам потребуется вручную слить эти данные с вашими, когда вы будете готовы. В следующих уроках мы научимся это делать :D

Одним словом, вы можете относиться к git fetch как к процедуре скачивания.

## Git Pull

Теперь, когда мы познакомились с тем, как извлекать данные из удалённого репозитория с помощью git fetch, давайте обновим нашу работу, чтобы отобразить все эти изменения!

Существует множество вариантов решений - как только у вас имеется локальный коммит, вы можете соединить его с другой веткой. Это значит, вы можете выполнить одну из команд:

* git cherry-pick o/master
* git rebase o/master
* git merge o/master
* и т.д.

Процедура скачивания (fetching) изменений с удалённой ветки и объединения (merging)настолько частая и распространённая, что git предоставляет вместо двух команд - одну! Эта команда - git pull.

## Git Pull

Теперь, когда мы познакомились с тем, как извлекать данные из удалённого репозитория с помощью git fetch, давайте обновим нашу работу, чтобы отобразить все эти изменения!

Существует множество вариантов решений - как только у вас имеется локальный коммит, вы можете соединить его с другой веткой. Это значит, вы можете выполнить одну из команд:

* git cherry-pick o/master
* git rebase o/master
* git merge o/master
* и т.д.

Процедура скачивания (fetching) изменений с удалённой ветки и объединения (merging)настолько частая и распространённая, что git предоставляет вместо двух команд - одну! Эта команда - git pull.

## Симуляция совместной работы

## Git Push

Хорошо, мы скачали изменения с удалённого репозитория и включили их в наши локальные наработки. Всё это замечательно, но как нам поделиться своими наработками и изменениями с другими участниками проекта?

Способ, которым мы воспользуемся, является противоположным тому способу, которым мы пользовались ранее для скачивания наработок (git pull). Этот способ - использование команды git push!

Команда git push отвечает за загрузку ваших изменений в указанный удалённый репозиторий, а также включение ваших коммитов в состав удалённого репозитория. По окончании работы команды git push все ваши друзья смогут скачать себе все сделанные вами наработки.

Вы можете рассматривать команду git push как "публикацию" своей работы. Эта команда скрывает в себе множество тонкостей и нюансов, с которыми мы познакомимся в ближайшее время, а пока что давайте начнём с малого...

замечание - поведение команды *git push* без аргументов варьируется в зависимости от значения *push.default*, указанной в настройках git-а. Значение по умолчанию зависит от версии git, которую вы используете, однако в наших уроках мы будем использовать значение *upstream*. Лучше всегда проверять эту опцию прежде чем push-ить ваши настоящие проекты.

## Git Push

Хорошо, мы скачали изменения с удалённого репозитория и включили их в наши локальные наработки. Всё это замечательно, но как нам поделиться своими наработками и изменениями с другими участниками проекта?

Способ, которым мы воспользуемся, является противоположным тому способу, которым мы пользовались ранее для скачивания наработок (git pull). Этот способ - использование команды git push!

Команда git push отвечает за загрузку ваших изменений в указанный удалённый репозиторий, а также включение ваших коммитов в состав удалённого репозитория. По окончании работы команды git push все ваши друзья смогут скачать себе все сделанные вами наработки.

Вы можете рассматривать команду git push как "публикацию" своей работы. Эта команда скрывает в себе множество тонкостей и нюансов, с которыми мы познакомимся в ближайшее время, а пока что давайте начнём с малого...

замечание - поведение команды *git push* без аргументов варьируется в зависимости от значения *push.default*, указанной в настройках git-а. Значение по умолчанию зависит от версии git, которую вы используете, однако в наших уроках мы будем использовать значение *upstream*. Лучше всегда проверять эту опцию прежде чем push-ить ваши настоящие проекты.

Представьте себе, вы склонировали репозиторий в понедельник и начали разрабатывать какую-то новую и уникальную часть приложения (на сленге разработчиков - фича). В пятницу вечером вы наконец-то готовы опубликовать вашу фичу. Но, о нет! Ваш коллега в течение недели написал кучу кода, который делает все ваши наработки устарелыми. Этот код был также закоммичен и опубликован на общедоступном удалённом репозитории, поэтому теперь *ваш* код базируется на *устаревшей* версии проекта и более не уместен.

В этом случае использование команды git push является сомнительным. Как поведёт себя команда git push, если вы её выполните? Может быть, она изменит удалённый репозиторий и вернёт всё к тому состоянию, которое было в понедельник? А может, команда попробует добавить ваш код, не удаляя при этом новый? Или же она проигнорирует ваши изменения, так как они уже устарели?

По причине того, что в данной ситуации (когда история расходится) слишком много двусмысленностей и неопределённостей, git не даст вам закачать (push) ваши изменения. Он будет принуждать вас включить в состав своей работы все те последние наработки и изменения, которые находятся на удалённом репозитории.

Как же разрешить данную ситуацию? Всё очень просто! Всё, что вам нужно - перебазировать свою работу на самую последнюю версию удалённой ветки.

Существует множество способов сделать это, но наиболее простой способ 'сдвинуть' свои наработки - через перебазировку или rebasing.

git fetch ; git rebase ; git push

А есть ещё какие-либо варианты обновить мои наработки к тому моменту, как удалённый репозиторий был обновлён? Конечно есть! Давайте ознакомимся с парочкой новых штучек, но в этот раз с помощью команды merge.

Несмотря на то, что git merge не передвигает ваши наработки (а всего лишь создаёт новый коммит, в котором Ваши и удалённые изменения объединены), этот способ помогает указать git-у на то, что вы собираетесь включить в состав ваших наработок все изменения с удалённого репозитория. Это значит, что ваш коммит отразится на всех коммитах удалённой ветки, поскольку удалённая ветка является *предком* вашей собственной локальной ветки.

Давайте взглянем на демонстрацию...

Таким образом, если мы объединим (merge) вместо перебазирования (rebase)...

git fetch; git merge o/master; git push

Опа! Мы обновили наше локальное представление удалённого репозитория с помощью git fetch, объединили ваши новые наработки с нашими наработками (чтобы отразить изменения в удалённом репозитории) и затем опубликовали их с помощью git push

Здорово! А можно ли как-то сделать всё то же самое, но с меньшим количеством команд?

Конечно - ведь вы уже знаете команду git pull, которая является аналогом и более кратким аналогом для совместных fetch и merge. А команда git pull --rebase - аналог для совместно вызванных fetch и rebase!

Давайте взглянем на эти оба варианта.

 Git демо

Сперва - с флагом --rebase...

git pull --rebase; git push

Тот же результат, как и ранее, но намного короче вызов команд.

А теперь с обычным pull

git pull; git push

И снова - результат такой же, как и ранее!

Рабочий процесс получения изменений (fetching), перебазирования/объединения (rebase/merging) и публикации изменений (pushing) используется довольно часто. В последующих уроках мы изучим более сложные варианты этих рабочих процессов, но пока что давайте остановимся на том, что есть.

**Слияние фича-бранчей (веток)**

Теперь, когда вы умело владеете командами fetch, pull и push, давайте применим эти навыки в сочетании с новым рабочим процессом (он же workflow).

Среди разработчиков, вовлечённых в большой проект, довольно распространённ приём — выполнять всю свою работу в так называемых фича-бранчах (вне master). А затем, как только работа выполнена, разработчик интегрирует всё, что было им сделано. Всё это, за исключением одного шага, похоже на предыдущий урок (там, где мы закачивали ветки на удалённый репозиторий)

Ряд разработчиков делают push и pull лишь на локальную ветку master - таким образом ветка master всегда синхронизирована с тем, что находится на удалённом репозитории (o/master).

Для этого рабочего процесса мы совместили две вещи:

* интеграцию фича-бранчей в master
* закачку (push) и скачку (pull) с удалённого репозитория

Давайте быстренько вспомним, как нам обновить master и закачать выполненную работу.

git pull --rebase; git push

Здесь мы выполнили две команды, которые:

* перебазировали нашу работу на новенький коммит, пришедший с удалённого репозитория, и
* закачали свои наработки в удалённый репозиторий

**Слияние фича-бранчей (веток)**

Теперь, когда вы умело владеете командами fetch, pull и push, давайте применим эти навыки в сочетании с новым рабочим процессом (он же workflow).

Среди разработчиков, вовлечённых в большой проект, довольно распространённ приём — выполнять всю свою работу в так называемых фича-бранчах (вне master). А затем, как только работа выполнена, разработчик интегрирует всё, что было им сделано. Всё это, за исключением одного шага, похоже на предыдущий урок (там, где мы закачивали ветки на удалённый репозиторий)

Ряд разработчиков делают push и pull лишь на локальную ветку master - таким образом ветка master всегда синхронизирована с тем, что находится на удалённом репозитории (o/master).

Для этого рабочего процесса мы совместили две вещи:

* интеграцию фича-бранчей в master
* закачку (push) и скачку (pull) с удалённого репозитория

Давайте быстренько вспомним, как нам обновить master и закачать выполненную работу.

git pull --rebase; git push

## Merge? Нет, нет, спасибо.

Чтобы закачать (push) новые изменения в удалённый репозиторий, всё, что вам нужно сделать - это смешать последние изменения из удалённого репозитория. Это значит, что вы можете выполнить rebase или merge на удалённом репозитории (например, o/master).

Если мы можем воспользоваться одним из двух методов, то почему же эти упражнения сфокусированы в основном на rebase? К чему такая нелюбовь к merge, когда речь идёт о работе с удалёнными репозиториями?

В среде разработчиков существует огромное количество дебатов около merging и rebasing. Ниже приведены основные за / против метода rebasing:

За:

* Rebasing делает дерево коммитов более чистым и читабельным, потому что всё представляется единой прямой линией.

Против:

* Метод rebasing явно изменяет историю коммитов в дереве.

Например, коммит C1 может быть перебазирован *после* C3. Соответственно, в дереве работа над C1' будет отображаться как идущая после C3, хотя на самом деле она была выполнена раньше.

Некоторые разработчики любят сохранять историю и предпочитают слияние (merging). Другие (такие как я) предпочитают иметь чистое дерево коммитов, и пользуются перебазировкой (rebasing). Всё зависит от ваших предпочтений и вкусов :D

### Удалённые-отслеживаемые ветки

Единственное, что могло бы показаться вам "магией" в нескольких предыдущих уроках - это то, как git знает, что ветка master соответствует o/master. Конечно, эти ветки имеют схожие имена и связь между локальной и удалённой ветками master выглядит вполне логично, однако, эта связь наглядно продемонстрирована в двух сценариях:

* Во время операции pull коммиты скачиваются в ветку o/master и затем соединяются в ветку master. Подразумеваемая цель слияния определяется исходя из этой связи.
* Во время операции push наработки из ветки master закачиваются на удалённую ветку master(которая в локальном представлении выглядит как o/master). Пункт назначения операции push определяется исходя из связи между master и o/master.

### могу ли я сделать это самостоятельно?

Само собой! Вы можете сказать любой из веток, чтобы она отслеживала o/master, и если вы так сделаете, эта ветка будет иметь такой же пункт назначения для push и merge как и локальная ветка master. Это значит, что вы можете выполнить git push, находясь на ветке totallyNotMaster, и все ваши наработки с ветки totallyNotMaster будут закачены на ветку master удалённого репозитория!

Есть два способа сделать это. Первый - это выполнить checkout для новой ветки, указав удалённую ветку в качестве ссылки. Для этого необходимо выполнить команду

git checkout -b totallyNotMaster o/master

, которая создаст новую ветку с именем totallyNotMaster и укажет ей следить за o/master.

### пособ №2

Другой способ указать ветке отслеживать удалённую ветку — это просто использовать команду git branch -u. Выполнив команду

git branch -u o/master foo

вы укажете ветке foo следить за o/master. А если вы ещё при этом находитесь на ветке foo, то её можно не указывать:

git branch -u o/master