Чтобы быстро начать работу с git нужно открыть интересующую нас папку (с проектом например) , ПКМ - Git Bash Here

Чтобы проверить установлен ли git

git –version

Hастроить имя пользователя и пароль для всех проектов

git config --global user.name ”Ivan Ivanov”

git config --global user.email [ivan.ivanov@gmail.com](mailto:ivan.ivanov@gmail.com)

Если есть необходимость для конкретного проекта поменять автора (для личного проекта, например), можно убрать –global

git config user.name ”Ivan Ivanov”

git config user.email [ivan.ivanov@gmail.com](mailto:ivan.ivanov@gmail.com)

Локальный репозиторий хранит всю информацию только локально в проекте в папке .git.

Удаленный хранит всю информацию где-то на удаленном сервере: локально хранится только копия проекта, изменения которой можно запушить (git push) в удаленный репозиторий.

Создать локальный репозиторий

git init

Посмотреть, какой статус работы на данный момент

git status

**Начало работы:**

1-й способ:

- создать проект на github/gitlab

- склонировать созданный репозиторий

- работа в реппозитории

2-й способ:

- Создать проект в папке

- открыть терминал в папке и проинициализировать проект(папку с ним) **git init**

- создать проект на github/gitlab

- соединить их командой в терминале

**git remote add origin https://github.com/Andrej/33333.**git

Жизненный путь файла:

1. Файл, который создан и не добавлен в репозиторий, будет в состоянии untracked. В этом состоянии файл можно изменять, но git изменения не отслеживает. Git начнет его отслеживать как только будет сделан git add . и с того состояния, в котором он будет при git add .
2. Делаем изменения в файлах, которые уже добавлены в гит репозиторий — переходим в состояние modified(если файл был изменен, но изменения в фале не сохранены, то modified нет).
3. Из тех файлов, которые мы изменили, выбираем только те (или все), которые нужны нам (например, скомпилированные классы нам не нужны), и эти классы после git add с изменениями попадают в состояние staged.

Если файл изменить и сделать git add он попадает в staged зону, но если продолжить его менять и сохранить изменения, то файл окажется одновременно в двух зонах (modified и staged)

1. При git commit из заготовленных файлов(из состояния staged) создается коммит и они переходят уже в гит репозиторий. После этого staged состояние — пустое. А вот modified еще может что-то содержать.

mkdir – добавить директорию

NUL> 33.txt – добавить файл (файл добавляет, хоть и пишет, что такой команды нет)

touch 33.txt – добавить файл

ls – показать файлы в данной папке, кроме скрытых

ls -f – показать файлы в данной папке, включая и скрытые

pwd — Показать текущий каталог (путь к нему(path))

rm test — Удалить пустую папку test

rm -r test — Удалить папку test с файлами внутри неё (удаляет и из staged-зоны)

mv app1/\*.\* app2 — Переместить все файлы из папки app1 в папку app2(перемещает и из staged-зоны)

git mv about.html company.html – переименование файла

git add <имя файла> — добавляет только конкретный файл

git add \*.java – добавляет только файлы с расширением java

git add someDir/ - все файлы в папке someDir

git add --no-all - измененные и новые файлы в текущей директории

После первого git add . файл получает статус **new**.

**git add -** добавляет в **staged** зону **ВСЕ** файлы(**untracked**, **modified**).

**git commit –a –m “сообщение”** - ***НЕ* *добавляет*** в **staged** зону новые (***untracked***) файлы, а добавляет только модифицированные (**modified**) файлы из обеих зон (рабочая и staged).

Закоммитить

git commit -m “what’s happend”

Коммиты связаны между с собой как односвязный список.

Также у коммита есть своя информация, так называемые метаданные:

* уникальный идентификатор коммита, по которому можно его найти;
* имя автора коммита, который создал его;
* дата создания коммита;
* комментарий, который описывает, что было сделано во время этого коммита.

**git commit --amend -m “новое сообщение”** *- (перезаписать последний commit)*дополняет последний commit новыми изменениями и меняет сообщение последнего commit-a(если это необходмо). Новый commit не создается.При этомфайлы, которые мы хотим подложить в последний commit должны быть предварительно добавлены в **staged** зону, соответственно untrucked файлы в commit не при каких обстоятельствах не попадают.

***ПРИ РАБОТЕ С КОММИТАМИ:***

* **коммит - это законченный функционал**
* **всегда проверяйте перед коммитом, что в него попадет. git diff - наш лучший друг**
* **выделяйте мелкие баги и правки в отдельные коммиты**
* **маленький коммит в одну строку - это нормально**
* **видите много изменений - подумайте, можно ли разбить их на отдельные коммиты**
* **мыслите задачей, а не файлами. Выделяйте полезное действие коммита**
* **commit message говорит, ЧТО делает коммит, а не КАК делает**
* **коммит-рефакторинг - это нормально. Не стоит мешать его с другими задачами**
* **git плохо работает с бинарниками (картинками, pdf, видеофайлами) - видит факт изменения, но не сами изменения**
* **подписывайте коммит так, чтобы можно было проследить историю развития проекта**

**Откат коммитов**

Если мы сделали неверный коммит и хотим откатить изменения, сделанные в нем:

git revert hash\_commit

При этом откроется дефолтный текстовый редактор, который предолжит ввести commit message. Если мы хотим оставить commit message по умолчанию, то можно обойтись без открытия редактора:

git revert --no-edit hash\_commit

Посмотреть на историю commit-ов в ветке

git log

Расширенный вывод истории

**git log -p**

Вывод коммитов в одну строку. Показывает только хэш коммита и commit message:

**git log** --**oneline**

**история в виде дерева**

**git log** --**stat** --**graph**

**Коммиты, затронувшие один файл**

**git log index.html**

Поиск по автору

**git log** --**author webdevkin**

Поиск по диапазону дат

**git log** --**after='2020-03-09 15:30'** --**before='2020-03-09 16:00'**

Как переключаться между коммитами в PhpStrom

Вкладка Log, правый клик на нужном коммите и Checkout Revision. Все. История коммитов будет видна по-прежнему вся, но напротив текущего коммита будет стоять значок HEAD с символом "!"

Как вернуться обратно? В правом нижем угле PhpStorm есть пункт git:, кликаем на него, выбираем Local Branches - master - checkout. Значок "!" пропадет - мы вернулись в исходное состояние

**Просмотр отдельного коммита**

git show 43f6afc

Выводится подробная информация о коммите:

* хэш
* автор
* дата
* commit message
* список измененных файлов
* изменения в каждом файле

подтянуть изменения с удаленного репозитория(когда изменения сделаны в том же файле). Делать ВСЕГДА, когда собираемся делать git push.

**git pull origin master**

* **pull** - что сделать, получить данные
* **origin** - откуда, с сервера
* **master** - а точнее, с ветки master

появился редактор

**Ctrl + O** – сохранить изменения

**Ctrl + X** – выйти из редактора кода (**:wq**)

после этого образуется merge-commit в истории коммитов

Что сделать, чтобы этого коммита не было? ->

**git pull --rebase origin master**

**отправка информации на сервер**

**git push origin master**

* **push -** что сделать, отправить
* **origin -** куда, на сервер
* **master -** что, ветку master

если мы отправляем информацию на удаленный сервер, а там уже произведены коммиты, то гит попросит нас сначала сделать git pull. Когда мы сделаем git pull у нас появятся:

1. коммит с нашей работой
2. затем коммиты стянутые с сервера
3. затем мерж комит, о том, что мы сделали merge ветки master(напр.) в свою ветку

(при этом push мы ещё не делали)

После того как мы сделаем git push они в таком же порядке зайдут на сервер.

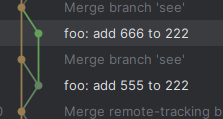
Если же сделать git pull –rebase origin master, то порядок будет таким:

1. коммиты подтянутые с сервера
2. коммит с нашей работой
3. нет мерж коммита при git pull –rebase origin master

Создал ветку, перешел на неё, сделал работу, закоммитал, перешел на master, сделал merge –no-ff, история имеет вид:



Вернулся на ветку, продолжил работу в ветке, сделал работу, закоммитал, перешел на master, сделал merge –no-ff, история имеет вид:



Создал ветку, перешел на неё, сделал работу, закоммитал, перешел на master, сделал git merge имя\_ответвления, история имеет вид:



То же при работе в PHP Shtorm:

Делаем изменения в файле. Ctrl + K – делаем коммит. ПКМ – Git – Repository – Push

В открывшемся окне нажимаем кнопку Push.

Прежде чем работать с файлом(прежде чем пушить изменения) пулим их с сервера:

ПКМ – Git – Repository – Pull. В открывшемся окне нажимаем кнопку Pull.

Если наш коллега уже закоммитал что-либо в мастер и запушил на сервер, то при нашей работе (Делаем изменения в файле. Ctrl + K – делаем коммит. ПКМ – Git – Repository – Push. В открывшемся окне нажимаем кнопку Push) После этого появляется ещё одно окно с опросом merge или rebase. И в зависимости от нашего выбора либо создается merge commit либо нет(как в вариантах выше).

Когда уже есть файлы в состоянии modified, мы можем посмотреть на изменения, которые были произведены над ними

git diff - показывает разницу между текущим **неотслеживаемым** состоянием репозитория и последним commit-ом репозитория (last commit)

git diff --staged - показывает разницу между текущим **отслеживаемым**

(после git add) состоянием репозитория и последним commit-ом репозитория

git diff [commit id] - показывает разницу между текущим состоянием репозитория и указанным commit-ом репозитория

Специальный файл, который нужно создать, чтобы git игнорировал (не контролировал в репозитории, где есть .git) интересуемые нас файлы и папки.

**.gitignore**

Для игнорирования папки, лежащей в корне проекта, файл .gitignore нужно класть в **корень проекта**.

Инструмент для отмены изменений, откату проекта к какому-то снимку.

**git reset**

**git reset** = **git reset HEAD** = **git reset** --**mixed** = **git reset** --**mixed HEAD**

git reset **--soft** [commit id] - возвращает проект к указанному снимку,

git reset **--soft** HEAD при этом переводит все снимки после

git reset **--soft** HEAD^ указанного в **(staged)** зону

git reset **--mixed** [commit id] - возвращает проект к указанному снимку,

git reset **--mixed** HEAD при этом переводит все снимки после

git reset **--mixed** HEAD~2 указанного в **(unstaged)** зону

git reset **--hard** [commit id] - возвращает проект к указанному снимку,

git reset **--hard** HEAD^ , при этом полностью удаляя все снимки

git reset **--hard** HEAD~2 после указанного (**безвозвратно!)**

git **checkout** [commit id] - позволяет «загрузить» любой из сохраненных

git **checkout** HEAD^^ снимков (коммитов) на компьютер разработчика

git **checkout** HEAD~2 (Для просмотра)

Указатель HEAD переходит в состояние **detached** (состояние открепленного указателя). Он отделен от актуальной версии проекта. Перед добавлением коммитов с открепленным указателем HEAD всегда следует создавать новую ветку. **Любые изменения или коммиты, сделанные в этом состоянии удаляются сборщиком мусора при переходе к другому коммиту.**

**Перейти к актуальному коммиту**

git checkout **master**

**Вернуть файл проекта к определенной версии в прошлом**

**git checkout [commit id] -- файл**

**git checkout HEAD^ -- файл**

Команда вернет **файл** к версии, которая была у него в указанном коммите (по умолчанию - HEAD) в staged зону. После чего его нужно закоммитать.

Вернуть все файлы к версии, которая была у них в последнем коммите :

**git checkout -- .**

**git checkout HEAD -- .**

**Работает только для (modified-файлов) в рабочей зоне**.

Удалить все **modified** изменения:

**git reset** или **git reset --mixed HEAD** (перевели все сохранённые в файле изменения в (**unstaged)** зону)

**git checkout -- .** (удалили все modified изменения)

Перед добавлением коммитов с открепленным указателем HEAD всегда следует создавать новую ветку.

Удаление новых неотслеживаемых **untracked** файлов:

**git clean –n -** показывает какие файлы будут удалены (при этом он еще не удален)

**git clean –f -** выполнит команду удаления

С задачей удаления **untracked** файлов не справятся команды:

**git reset --hard HEAD**

**git checkout -- .**

***Работа с ветками***

**Скачивание определённой ветки с удаленного репозитория**

$ git clone [https://dev.intechs.by:3080/lk/brest/back/brest-personal-area.git -b sprint\_06](https://dev.intechs.by:3080/lk/brest/back/brest-personal-area.git%20-b%20sprint_06)

Команда для просмотра на какой ветке мы сейчас находимся:

**git branch**

Создание новой ветки:

**git branch название\_ветки**

Переключение между ветками:

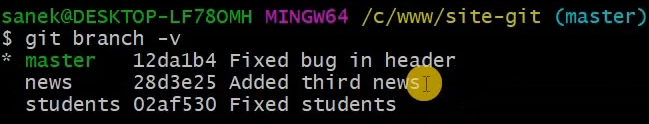
**git checkout название\_ветки**

Создать новую ветку и переключитсья на нее:

**git checkout –b название\_ветки**

Вывод не только всех веток, но последнего коммита на каждой из них

**git branch -v**



переименование ветки (переключиться на нужную верку и выполнить команду)

**git branch -m new\_branch\_name**

если вы сделали изменения и не закомитали их, git не даст нам переключиться на другую ветку. Изменил – закоммитал – работай дальше. (или git stash)

если вручную в изменённых файлах всё вернуть как было – тоже ок.

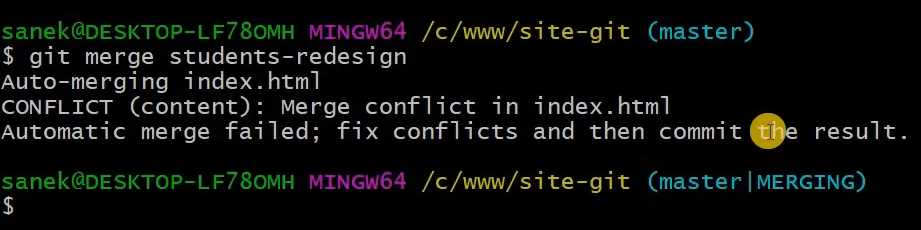
лучше сначала сделать актуальной ветку мастер, а уже потом заливать свои изменения. А еще лучше держать актуальной свою ветку относительно мастера. Это значит, что стоит почаще подтягивать мастер в свою ветку. Таким образом мы в своей ветке будем работать с актуальным кодом и у нас меньше риска что-нибудь поломать. А если поломаем, то лучше чинить это в своей ветке, а не в мастере.

Слияния одной ветки с другой (вливается та, которую указываем в команде, в ту, на которой находимся):

**git merge название\_ветки**

после слияния ветка **название\_ветки** остается

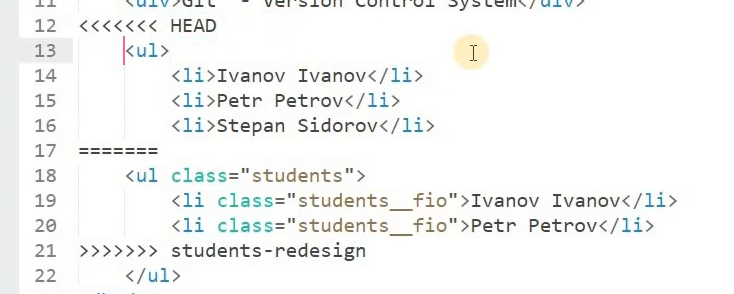
При слиянии веток могут возникать конфликты, то есть если в ветку в которую мы залить свою уже внесены изменения в тот же файл, то возникают ошибки



Открыв файл, где возник конфликт, мы увидим два раздела с изменениями:

HEAD- текущая ветка

другая\_ветка – ту, что вливаем в текущую



Из этого собираем, то что нам нужно(оставляем то, что будет работать в обоих случаях)

После этого git add . и git commit -m ‘то-то’

Удаление ветки:

**git branch –d название\_ветки**

* 1. **Нельзя удалить ветку находясь на ней, нужно переключиться на другую**
  2. **Git не даст сразу удалить ветку если всё что мы делали не запушино в основную ветку**

если есть изменения в ветке и мы хотим все равно её удалить то

**git branch -D name\_branch**

Запушить новую ветку на сервер

**git push origin название\_ветки**

**если мы запушим новую ветку, то на сервере, в github, появится еще одна ветка block-news, и при этом останется старая news. Чтобы не засорять проект, старую ветку нужно удалить как локальную у себя, так и удаленную на сервере.**

**В PHP Shtorm :**

При переключении с ветки на ветку, когда сделали изменения

Force Checkout – изменения не сохраняются, переключение есть

Smart Checkout – сохраняются, переключение есть

Don’t Checkout – не сохраняются, переключения нет

Порядок действий в созданной ветке:

- Создаем приватное ответвление от публичной ветки.

**git checkout –b new\_branch**

- Методично коммитим работу в эту приватную ветку.

**git add .**

**git commit –m ‘message’**

- Как только код достиг совершенства, приводим историю в порядок.

подчистите вашу ветку с пом. reset, rebase, merge --squash и commit --amend.

- Сливаем упорядоченную ветку обратно в публичную.

**git merge (--no-ff) new\_branch** – с мержкоммитом

**git merge new\_branch** – без него

Получение списка удаленных веток:

**git branch –r**

Список **локальных** веток (изменяются только нами).

Список **удаленных** веток (изменяются всеми кто работает с этим репозиторием )

Ветка **new\_test\_2** появится в списке удаленных только когда мы сделаем ее **push** в удаленный репозиторий.

Залить изменения с удаленного репозитория на локальную верку

**git pull origin** **master**

**git pull** = = 

1. git скачает удаленную ветку с удаленного репозитория (обновит нашу удаленную

ветку origin/master )

2) git cольет удаленную ветку с локальной (произведет актуализацию нашей локальной

ветки)

**git fetch** получает изменения с сервера и сохраняет их в каталог refs/remotes/ (“скачивает” удаленный ветки с репозитория, но не производит актуализацию локальных веток, т.е. не делает слияния удаленной ветки с локальной).

Удаление ветки с удаленного репозитория

**git push --delete origin** **название\_ветки**

**Перебазирование** (способ перенести изменения из одной ветки в другую)

**git rebase название\_ветки**

при этом коммиты из ветки на которой мы находились перемещаются после последнего коммита указанной нами ветки (**название\_ветки**)

Перебазировать ветку new\_test\_1 в master(изменения проводятся в разных местах)



git checkout master

git rebase new\_test\_1

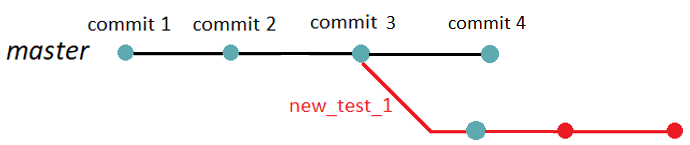


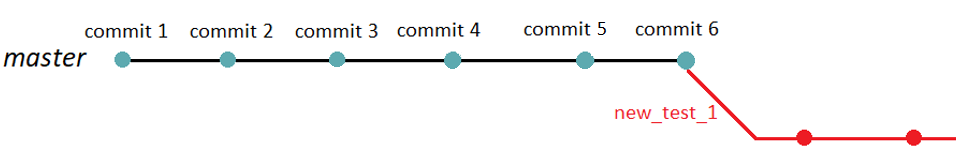
Перебазировать ветку master в new\_test\_1 (изменения проводятся в разных местах)



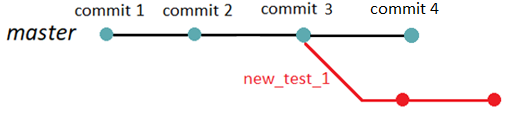
git checkout new\_test\_1

git rebase master





Внесли изменения в одних и тех же местах:



В этом случае конфликты решаются вручную, как и при использовании merge

**git rebase --continue** используем после того как решили конфликты вручную

**git rebase --abort** используем если передумали делать rebase

**ещё о git rebase…**

двое коллег, Вася и Петя, работают над одним проектом. В проекте есть основная ветка master, в нее периодически коммитят и Вася, и Петя. Вася создает новую ветку new-footer, делая свои коммиты в нее и время от времени подтягивая в эту ветку мастер

Начинает проект Вася, он создает первый коммит и пушит его.



Дальше Вася добавляет подвал прямым коммитом в мастер.



Вася хочет поделиться подвалом со своим коллегой Петей и пытается запушить мастер. Но Петя уже успел запушить со своей стороны в мастер коммит, касающийся меню. Как известно, git не даст запушить ветку до тех пор, пока Вася не подтянет свежий коммит от Пети. Что может быть проще? Вася делает

**git pull origin master**

****

Что это за хрень? У нас фактически всего лишь 3 коммита: initial, один от Васи и один от Пети. Но в истории мы видим уже 4. Дополнительный коммит 'merge branch master ...' не несет никакой смысловой нагрузки. Плюс непонятное ответвление, тоже никому не нужное. Мы всего лишь подтянули коммит Пети в той же ветке, мы не сливали одну ветку в другую (с нашей точки зрения). А представьте, какую кучу хлама будет содержать ветка после десятка таких git pull. Как это исправить?

Вернемся к тому моменту, когда мы собрались делать git pull. И сделаем иначе:

**git pull --rebase origin master.**

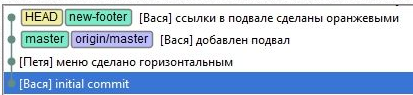
****

Отлично! То, что нужно. Коммит от Пети следует сразу за initial, коммит от Васи самый свежий. Можно пушить ветку на сервер.

**git push origin master**

**Если все в команде договорятся делать git pull --rebase, то основные долгоживущие ветки будут выглядеть гораздо чище!!!**

После этого Вася приступает к работе над оформлением подвала и благоразумно заводит новую ветку, Вася делает в новой ветке первый коммит

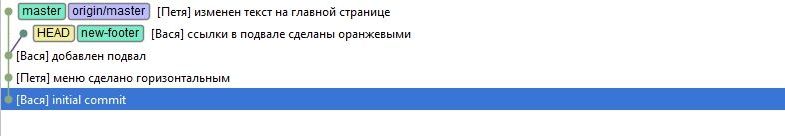


Но за это время шустрый Петя уже успел закоммитить в мастер правки по тексту на главной странице. Вася ответственный разработчик и он помнит, что время от времени нужно подтягивать в свою ветку свежий мастер, дабы убедиться, что его работа не ломает работу Пети (и наоборот). Вася переключается на мастер, стягивает этот свежий коммит (конечно, через git pull --rebase) и переходит обратно в свою ветку, чтобы эти изменения влить в new-footer.

**git checkout master**

**git pull --rebase origin master**

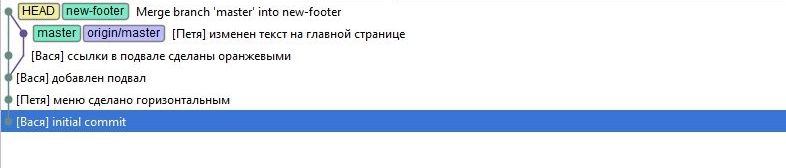
**git checkout new-footer**



Задача Васи - влить ветку мастер в свою ветку new-footer. Вася делает

**git merge master**

и видит это



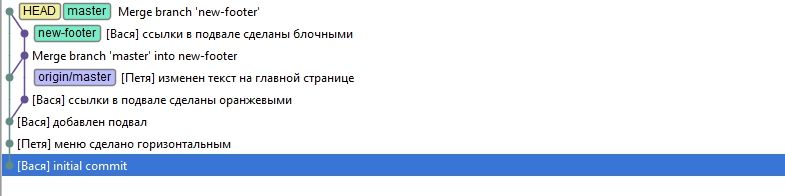
Что-то не очень. Подтянули один коммит и опять получили лишний 'merge branch'. Ладно, некоторые программисты не видят в этом ничего плохого и утверждают, что хотят знать, когда в текущую ветку вливался мастер. - зачем?

Но оставим пока так и работаем дальше. Вася делает еще один коммит.

Затем он тестирует свою ветку, убеждается, что все хорошо, и сливает ее в мастер.

**git checkout master**

**git merge --no-ff new-footer**



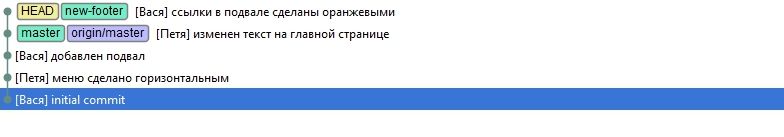
Напомню, в ветке new-footer всего лишь 2 коммита от Васи. Здесь же мы видим лишний merge и лишнюю связь, которая ни о чем не говорит. Легко представить тот бардак, который будет твориться при нескольких мерджах в долгоживущих ветках.

git rebase дает нам возможность сохранить историю коммитов чистой и понятной. Давайте все исправим и вернемся к тому моменту, когда мы собрались мерджить мастер в нашу ветку

Теперь делаем так

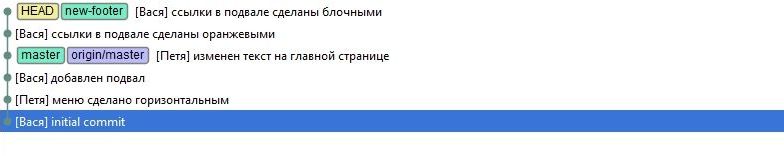
**git rebase master**

и видим следующую картину



Намного лучше! Мы просто передвинули указатель HEAD так, как будто new-footer ответвилась от мастера только что.

Вася делает второй коммит

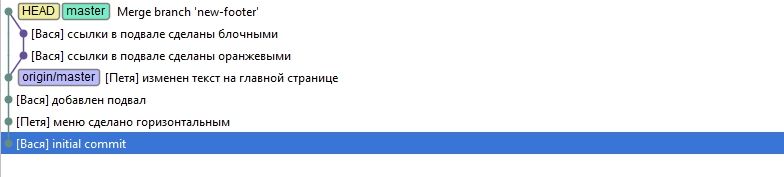


И наконец Вася сливает ветку new-footer в мастер и удаляет ее.

**git checkout master**

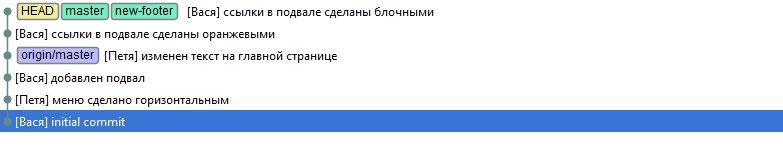
**git merge --no-ff new-footer**

**git branch -d new-footer**

****

Отличная картинка. Мы видим 2 коммита Васи и видим, что они сделаны в отдельной ветке. А если Вася подписывал бы начало своих коммитов названием ветки, а не своим именем, то история выглядела бы еще лучше :-)

Обратим внимание на опцию --no-ff. git merge по умолчанию пытается просто передвинуть указатель, не делая мерджа. Если мы сделаем просто git merge new-footer, то итоговая история будет такой



здесь уже зависит от того, какие договоренности приняты в Вашем проекте

P.S. Есть негласное правило. Без опаски пользуйтесь git rebase смело до тех пор, пока Вы работаете в своей ветке один. Как только к ветке присоединяется ваш коллега, подтягивайте изменения из основной ветки через merge. Это связано с тем, что rebase перезаписывает историю коммитов, и пушить на сервер вам придется с опцией --force. Но это уже другая история.

P.P.S. К сожалению, кроме незнания/неумения, есть еще одна банальная причина, по которой программисты не пользуются rebase. Многие из нас работают с гитом исключительно средствами любимой IDE. Так вот часто случается, что git pull в IDE по умолчанию не ставит опцию --rebase. Нужно для этого тыкать отдельный чекбокс. И git merge находится ближе в меню, чем git rebase. И все.

**Работа с удаленным репозиторием**

Просмотр списка существующих удаленных репозиториев:

**git remote –v**

Добавить удаленный репозиторий, который находится по указанному адресу:

**git remote add название\_репозитория адрес\_репозитория**

Удалить репозиторий с указанным названием:

**git remote remove название\_репозитория**