Hasselhoff вчера в 16:58

Продвинутые функции гита, о которых вы, возможно, не знали

Автор оригинала: Martin Heinz

Блог компании SkillFactory, Программирование, Git, GitHub, Лайфхаки для гиков

Перевод Tutorial

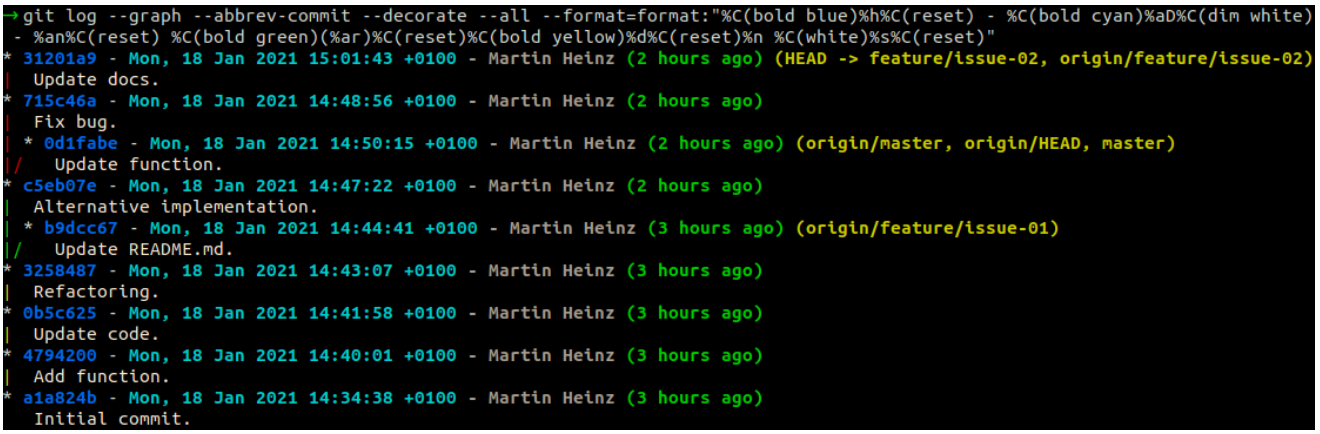
Git – очень мощный инструмент, который практически каждый разработчик должен использовать ежедневно, но для большинства из нас git сводится к нескольким командам: pull commit push. Однако, чтобы быть эффективным, продуктивным и обладать всей мощью git, необходимо знать ещё несколько команд и трюков. Итак, в этой статье мы исследуем функции git, которые просто запомнить, применять и настроить, но которые могут сделать ваше время с git гораздо более приятным.

Прокачиваем базовый рабочий процесс

Прежде чем мы воспользуемся даже самыми базовыми командами – pull, commit и push, необходимо выяснить, что происходит с нашими ветками и изменёнными файлами. Для этого можно воспользоваться git log – довольно известной командой, хотя не все знают, как сделать его вывод *на самом деле* читабельным и красивым:

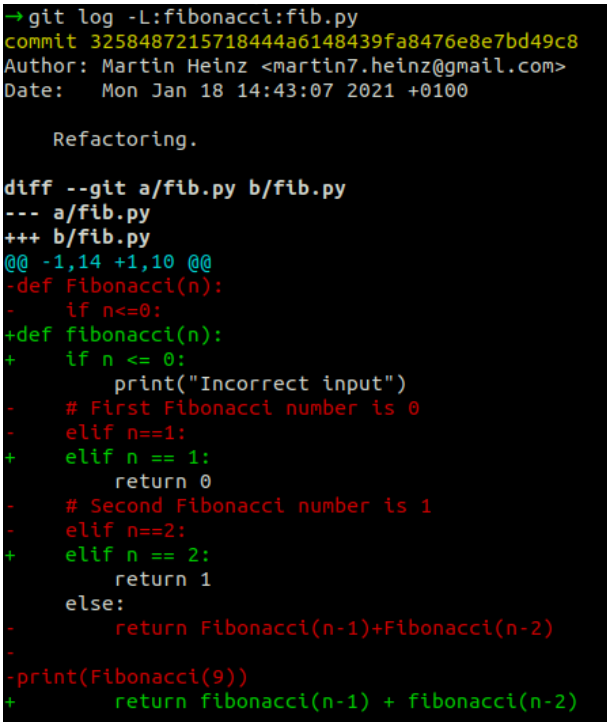
```
git log --graph --abbrev-commit --decorate --all \
    --format=format:"%C(bold blue)%h%C(reset) - %C(bold cyan)%aD%C(dim white) \
    - %an%C(reset) %C(bold green)(%ar)%C(reset)%C(bold yellow)%d%C(reset)%n %C(white)%s%C(reset)"
```

git_log.shell hosted with ❤ by **GitHub** **view raw**



Дерево git log.

Такой граф даст хороший обзор, однако часто нужно копать немного глубже. Например, посмотреть историю (эволюцию) определённых файлов или даже отдельных функций; в этом поможет git log с флагом -L::).



git log для функции.

Теперь, когда мы немного представляем происходящее в репозитории, мы, возможно, захотим проверить различия между обновлёнными файлами и последним коммитом. Здесь можно воспользоваться git diff; опять же ничего нового здесь нет, но у diff есть кое-какие опции и флаги, о которых вы, возможно, не знаете. Например, можно сравнить две ветки: git diff branch -a… branch -b, или даже конкретные файлы в разных ветках: `git diff <commit-a> <commit-b> -- <пути>`.

Иногда чтение git diff становится трудной задачей. Можно попробовать прописать игнорирующий все пробельные символы (white-space) флаг -w, и этим немного заспамить diff, или флаг --word-diff и работать вместо строк с раскрашенными словами.

```
→git diff 9cbae38 291a2df --word-diff --color-words
diff --git a/file.txt b/file.txt
index f45599f..33e5dea 100644
--- a/file.txt
+++ b/file.txt
@@ -1 +1 @@
"Lorem ipsum doloredolor sit amet, consectetur adipiscing elit, sed dogddo eiusmod tempor incididunt ut labore nbvet dolore
 magna aliqua. Ut enim ad minim veniam, quis noexercitationnostrud exercitation ullamco laboris nisi ut aliquip ex ea commo
do consequat. Duis aubeaute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Ex
cepteur abcdsint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim qwidid est laborum."
```

Если простой статичный вывод в оболочке вас не устраивает, можно запустить difftool, вот так: git difftool=vimdiff, команда откроет файлы diff внутри vim в два окна – слева и справа. Очевидно, что Vim – не единственный вариант; можно запустить git difftool --tool-help, чтобы увидеть список всех инструментов, которые можно использовать вместе с diff.

Мы уже видели, как просматривать историю конкретных частей или строк в файла с помощью git log. Было бы удобно делать нечто подобное, например, стейджинг частей файлов, правда? И такое легко делается в в IDE, например, в IntelliJ; то же самое уже сложнее в git CLI, но, конечно же, по-прежнему возможно: в git add пропишите опцию --patch:

```
1  git add fib.py --patch
2  diff --git a/fib.py b/fib.py
3  index e95a02c..8ac1035 100644
4  --- a/fib.py
5  +++ b/fib.py
6  @@ -1,3 +1,5 @@
7  +from math import sqrt
8  +
9   # This function computes Fibonacci sequence
10  def fibonacci(n):
11      if n <= 0:
12  (1/2) Stage this hunk [y,n,q,a,d,j,J,g,/,e,?]? y
```

git_add.shell hosted with ❤ by **GitHub** **view raw**

Команда открывает редактор, в котором отображается один *"hunk"* [кусок], представляющий собой кусок кода с несколькими отличающимися друг от друга строками в нём. Можно много чего сделать с этим куском, но самые важные опции – это y – принять изменения (делает стейджинг), n – не принимать (не делать стейджинг) и e – отредактировать кусок перед стейджингом (полный список опций здесь).

Когда закончите с интерактивным стейджингом, вы можете запустить git status, и увидите, что файл с частичным стейджингом находится в разделах "Changes to be committed:" и "Changes not staged for commit:". Кроме того, можно запустить git add -i (интерактивный стейджинг), а затем воспользоваться командой s (статус), которая покажет вам, какие строки находятся на стейджинге, а какие – нет.

Исправление распространённых ошибок

Закончив со стейджингом, я (слишком) часто осознаю, что добавил то, чего добавлять не хотел. Однако на этот случай у git для файлов нет команды *un-stage*. Чтобы обойти ограничение, можно сбросить репозиторий командой git reset --soft HEAD somefile.txt. Вы также можете включить в git reset флаг -p, который покажет вам тот же UI, что и у git-add -p. Также не забудьте добавить туда флаг --soft, иначе вы сотрёте ваши локальные изменения!

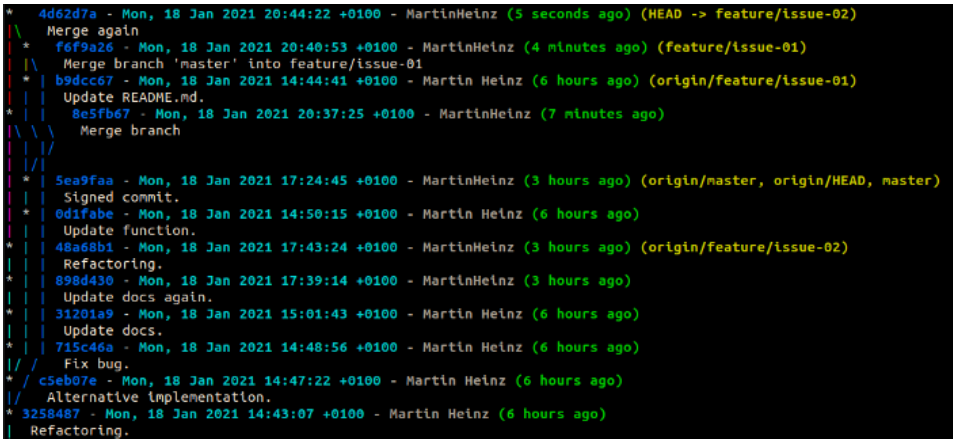
Поменьше грубой силы

Теперь, когда мы закончили стейджинг, всё, что осталось, – commit и push. Но что, если мы забыли что-то добавить или совершили ошибку и хотим исправить уже запушенные коммиты? Есть простое решение, использующее git commit -a и git push --force, но оно может быть довольно опасным, если мы работаем над общей веткой, например, master. Таким образом, чтобы избежать риска перезаписи чужой работы из-за того, что мы решили проблему грубой силой, мы можем воспользоваться флагом --force-with-lease. Этот флаг – в отличие от --force – запушит на изменения только в том случае, если за время работы никто не добавил никаких изменений в ветку. Если ветка была изменялась, код не будет отправлен, и этот факт сам по себе указывает на то, что перед отправкой кода мы должны выполнить git pull.

Правильное слияние веток

Если вы работаете над репозиторием, в котором участвует более одного разработчика, можно с уверенностью предположить, что вы работаете в отдельной ветке, а не в мастере. Это также означает, что рано или поздно вам придётся включить свой код в

кодовую базу (главную ветку). Вполне вероятно, что, пока вы работали над своей веткой, кто-то другой уже добавил свой код в мастер, из-за чего ветка вашей функциональности отстаёт на несколько коммитов. Можно пойти напролом и выполнить слияние вашего кода в мастер с помощью git merge, но команда создаст дополнительный *комммит слияния*, а также, без необходимости на то, затруднит чтение истории и сделает её сложнее:



История с ветвлением.

Подход гораздо лучше (не стесняйтесь спорить со мной по этому поводу, образно говоря, это та высота, на которой я готов умереть) заключается в том, чтобы сделать rebase ветки функции в master, а затем выполнить так называемую *быструю перемотку* (git merge --ff). Подход сохраняет историю линейной, читать такую историю легче, упрощается и последующий поиск коммитов с новым функционалом и коммитов – виновников ошибок.

Но как нам сделать такой *rebase*? Можно выполнить rebase в его базовой форме с помощью git rebase master feature_branch, чего часто бывает достаточно (за этим следует push --force). Однако, чтобы получить от git rebase максимальную отдачу, также следует включить флаг -i, чтобы rebase был *интерактивным*. Интерактивный rebase – удобный инструмент, чтобы, например, переформулировать, сжать или вообще очистить ваши коммиты и всю ветку. В качестве небольшой демонстрации мы можем даже сделать rebase ветки на саму себя:

```
1  git rebase -i @~4  # @ is same as HEAD
```

git_rebase.shell hosted with ❤ by **GitHub** **view raw**

Приём выше позволяет нам повторно применять последние 4 коммита и изменить их, получив полезный результат, например сжать одни коммиты и переформулировать другие:

```
# git log before rebase

* 48a68b1 - Mon, 18 Jan 2021 17:43:24 +0100 - MartinHeinz (64 minutes ago) (HEAD -> feature/issue-02, origin/feature/issue-02)
|  Refactoring.
* 898d430 - Mon, 18 Jan 2021 17:39:14 +0100 - MartinHeinz (68 minutes ago)
|  Update docs again.
* 31201a9 - Mon, 18 Jan 2021 15:01:43 +0100 - Martin Heinz (4 hours ago)
|  Update docs.
* 715c46a - Mon, 18 Jan 2021 14:48:56 +0100 - Martin Heinz (4 hours ago)
|  Fix bug.
| ...

# Start of interactive rebase

pick 715c46a Fix bug.
pick 31201a9 Update docs.
pick 898d430 Update docs again.
pick 48a68b1 Refactoring.
# Rebase c5eb07e..48a68b1 onto 48a68b1 (4 commands)
...

# Rebasing changes

reword 31201a9 Update docs.      # You will be prompted to reword this
fixup 898d430 Update docs again.
pick 48a68b1 Refactoring.        # You will be prompted to reword this
squash 715c46a Fix bug.
# Rebase c5eb07e..48a68b1 onto 48a68b1 (4 commands)
...

# git log after rebase (+ rewording)

* 11495c3 - Mon, 18 Jan 2021 17:43:24 +0100 - Martin Heinz (84 minutes ago) (HEAD -> feature/issue-02)
|  Refactoring and fixes.
* 561c327 - Mon, 18 Jan 2021 15:01:43 +0100 - Martin Heinz (4 hours ago)
|  Update documenatation.
|  ...
```

git_rebase_interactive.shell hosted with ❤ by **GitHub** **view raw**

Выше показан пример сеанса rebase. В верхней части показывается ветка перед перезагрузкой. Вторая часть фрагмента – это список коммитов, представленных после запуска `git rebase …` каждый из них можно выбрать, чтобы включить в работу (pick). Мы можем изменить действие для каждого из них, а также полностью переупорядочить коммиты. Как показано в третьем разделе примера, некоторые допустимые действия – переформулирование (оно говорит git открыть редактор сообщений о коммите), сжатие коммита (объединяет коммиты в предыдущий) и исправление коммита: (исправление работает как сжатие, но при этом сбрасывает сообщение о коммите). После того как мы применим эти изменения и переформулируем изменённые коммиты, мы получим историю, которая показана на скриншоте выше, в его нижней части.

Если во время rebase вы столкнулись с каким-либо конфликтом, чтобы разрешить его, вы можете запустить git mergetool --tool=vimdiff, а затем продолжить rebase с помощью git rebase --continue. git mergetool может быть вам не знаком, на первый взгляд он может показаться пугающим. В действительности же это то же самое, что IDE вроде IntelliJ, просто *в стиле Vim*. Если вы не знаете хотя бы несколько сочетаний клавиш Vim, то, как и в случае с любым другим использующим этот редактор инструментом, вам, может быть, трудно даже понять, на что на самом деле вы смотрите. Если вам нужна помощь, я рекомендую прочитать эту исчерпывающую статью.

Если всё это кажется слишком сложным или вы просто боитесь работать с rebase, в качестве альтернативы создайте пул реквест на GitHub и нажмите кнопку *Rebase and merge*, чтобы сделать, по крайней мере, простые и быстрые rebase и merge с быстрой перемоткой.

Главное – эффективность

Я думаю, что примеры выше показали несколько изящных советов и хитростей, но всё это может быть довольно сложно

запомнить, особенно когда дело касается команд вроде git log. К счастью, чтобы разрешить эти трудности, можно воспользоваться глобальной конфигурацией git. Она находится в ~/.gitconfig и обновляется каждый раз, когда вы запускаете git config --global. Даже если вы не настраивали этот файл, он, вероятно, содержит кое-какие базовые вещи, такие как раздел [user], но можно добавить много других разделов:

```
# .gitconfig
[user]
        name = MartinHeinz
        email = martin7.heinz@gmail.com
        signingkey = 7FBRA885E6354BC3E489CAF3D8B87B8N91F7538Q
[core]
        autocrlf = input
    editor = vim  # git config --global core.editor vim
[alias]
    graph = log --graph --abbrev-commit --decorate --all \
    --format=format:"%C(bold blue)%h%C(reset) - %C(bold cyan)%aD%C(dim white) \
    - %an%C(reset) %C(bold green)(%ar)%C(reset)%C(bold yellow)%d%C(reset)%n %C(white)%s%C(reset)"
[help]
        autocorrect = 10  # git config --global help.autocorrect 10
[commit]
        gpgsign = true
```

gitconfig hosted with ❤ by **GitHub** **view raw**

Выше приведён пример некоторых из доступных опций конфигурации. Примечательно, что длинная команда git log – это только псевдоним git graph. Автокоррекция установлена 10: такое значение включает её и заставляет ждать 1 секунду, прежде чем выполнить правильную команду, в которой была опечатка, и, наконец, последний раздел – подписывание коммита GPG (подробнее об этом читайте ниже).

Настройка .gitconfig с кучей алиасов требует отдельной статьи. Есть довольно много хороших ресурсов и примеров того, что можно прописать в .gitconfig, поэтому вместо полного списка всех опций и псевдонимов я оставлю ссылки:

- https://github.com/jessfraz/dotfiles/blob/master/.gitconfig
- https://gist.github.com/pksunkara/988716
- https://github.com/mgedmin/dotfiles/blob/master/gitconfig

Автозавершение команд – это инструмент не менее продуктивный, чем псевдонимы, и он просто устанавливается:

```
cd ~
curl https://github.com/git/git/blob/master/contrib/completion/git-completion.bash

# Add the following to you .bash_profile or .bashrc
if [ -f ~/.git-completion.bash ]; then
  . ~/.git-completion.bash
fi
```

git_completion.shell hosted with ❤ by **GitHub** **view raw**

Extras

Можно не только писать свои псевдонимы, но и взять на вооружение плагин git-extras, он вводит много полезных команд, которые могут немного упростить вам жизнь. Я не буду вдаваться в подробности обо всех возможностях этого плагина – посмотрите список команд, а я просто покажу один краткий пример из этого списка прямо здесь:

- git delta – список файлов, которые в другой ветке отличаются.
- git show-tree – древовидное представление коммитов всех ветвей, похожее на показанный ранее git log.
- git pull-request – пул-реквест в командной строке.
- git changelog – генерирует журнал изменений (changelog) из тегов и сообщений в коммитах.

Конечно, это не единственный крутой плагин. Например, есть ещё один удобный инструмент, позволяющий открыть репозиторий в браузере прямо из командной строки. Кроме того, в приглашении терминала можно настроить статус репозитория, это делается с помощью zsh или bash-it.

Подписываем коммиты

Даже если вы никогда не вкладывались в какой-либо проект Open Source, вы, вероятно, прокручивали историю коммитов такого проекта. В ней вы, скорее всего, видели значок подтверждённого (sign-off – знак о правах на ПО), проверенного или подписанного коммита. Что это такое и зачем?

Первый значок используется, когда автор подтверждает, что соответствующий код написал именно он, или же значком вы подтверждаете, что, насколько вам известно, он был создан на основе соответствующей лицензии Open Source. Это делается по юридическим причинам, которые связаны со статусом авторских прав на код. Обычно вам не нужно пользоваться этим значком, но, если вы в какой-то момент захотите внести вклад в проект, который требует подтверждения прав, знак подтверждения ставится так:

```
~ $ git commit -m "Update docs." --signoff
[feature/issue-02 a2385f4] Update docs.
 1 file changed, 1 insertion(+)

~ $ git log
commit 31201a9a91983641897ac1e6c2ee0217a4952d7c
Author: Martin Heinz <martin7.heinz@gmail.com>
Date:   Mon Jan 18 15:01:43 2021 +0100

    Update docs.

    Signed-off-by: Martin Heinz <martin7.heinz@gmail.com>
...
```

git_commit_signoff.shell hosted with ❤ by **GitHub** **view raw**

Сверху видно, что в git commit с опцией --sign-off в конце сообщения о коммите автоматически добавляется строка Signed-off-by: …, которая формируется на основе вашего имени пользователя в конфигурации git.

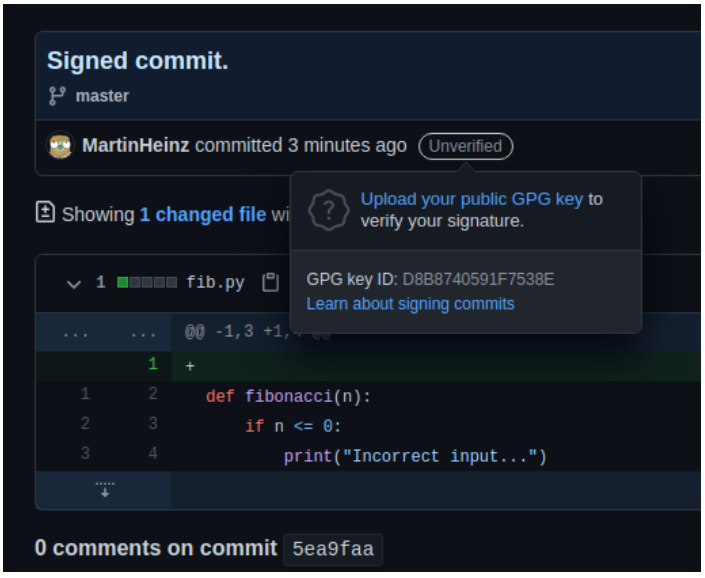
Что касается значка signed/verified, который вы, вероятно, заметили в некоторых репозиториях, он существует, потому что на GitHub довольно легко выдавать себя за других пользователей. Всё, что вам нужно сделать, – изменить имя сделавшего коммит человека и электронную почту в вашей конфигурации и отправить изменения. Чтобы предупредить ситуацию, вы можете подписывать коммиты с помощью ключей GPG, подтверждающих, что автор коммита и отправитель изменений на самом деле является тем, за кого он себя выдаёт. Подпись коммита более распространена, чем подтверждение прав, поскольку важно знать, кто на самом деле внёс код.

Если вы хотите начать пользоваться этой функцией или, возможно, хотите внедрить её в вашей команде, можно сделать следующее:

```
gpg --list-keys  # Check if there are any keys already
gpg --gen-key    # Generate key

...
Real name: Martin Heinz
Email address: martin7.heinz@gmail.com
You selected this USER-ID:
    "Martin Heinz <martin7.heinz@gmail.com>"

pub   rsa3072 2021-01-18 [SC] [expires: 2023-01-18]
      <SOME_VALUE>
uid                      Martin Heinz <martin7.heinz@gmail.com>
sub   rsa3072 2021-01-18 [E] [expires: 2023-01-18]

git config --global user.signingkey <SOME_VALUE>

git commit -m "Signed commit." -S  # You will get asked for password here...
git push
```

git_commit_verified.shell hosted with ❤ by **GitHub** **view raw**

Сначала вы генерируете пару ключей GPG (если у вас её ещё нет), затем устанавливаете ключи при помощи git config … и, наконец, добавляете опцию -S, когда делаете коммит. Затем, посмотрев на информацию о коммите на GitHub, вы увидите значок, как на картинке ниже.



Подписанный непроверенный коммит.

Однако, как видно на изображении, подпись не проверена, потому что GitHub не знает, что ключ GPG принадлежит вам. Чтобы это исправить, открытый ключ из нашей пары ключей нужно отправить на GitHub. Для этого экспортируем ключ командой gpg --export, как здесь:

```
gpg --armor --export <SOME_VALUE>
-----BEGIN PGP PUBLIC KEY BLOCK-----
...
-----END PGP PUBLIC KEY BLOCK-----
```

gpg_export.shell hosted with ❤ by **GitHub** **view raw**

Затем скопируйте этот ключ и вставите его в поле https://github.com/settings/gpg/new. Если вы проверите ранее подписанный коммит после добавления ключа, то увидите, что коммит теперь проверен (verified). Здесь предполагаем, что вы добавили на GitHub именно тот ключ, которым подписывали коммит:

Подписанный проверенный коммит.

Заключение

Git – очень мощный инструмент, у которого слишком много подкоманд и опций, чтобы в одной статье описать их все. Если вы хотите глубже погрузиться в некоторые связанные с Git темы, я бы порекомендовал прочитать Debugging with Git, чтобы узнать больше о blame, bisect или Getting solid at Git rebase vs. merge, чтобы глубже понять rebase и merge. Помимо множества полезных статей в Интернете часто при поиске информации о некоторых тонкостях git лучший выбор – это мануал, который выводится опцией `--help`, или версия в сети.

Теги: skillfactory, git, rebase, gpg, лайфхаки, программирование, версионность

Хабы: Блог компании SkillFactory, Программирование, Git, GitHub, Лайфхаки для гиков

Максим @Hasselhoff
Пользователь

Сайт

ПОХОЖИЕ ПУБЛИКАЦИИ

20 января 2021 в 13:13

САМОЕ ЧИТАЕМОЕ

Сутки | Неделя | Месяц

Судьба предателя, угнавшего новейший МиГ-25 в Японию
+89 660k 119 2397

Суверенный DNS уже здесь, а вы и не заметили
+57 33,8k 58 155

Ящик пива за лучшую сисадминскую байку и наш личный топ историй
+59 23,1k 55 321

Как сделать волоконно-оптическую светодиодную лампу
+55 18,6k 143 17

Куда же плывут облака? Подбиваем итоги опроса и даём аналитику
Мегапост

> git pull-request

Ваш аккаунт | Разделы | Информация | Услуги
Войти | Публикации | Устройство сайта | Реклама
Регистрация | Новости | Для авторов | Тарифы
Хабы | Для компаний | Контент
Компании | Документы | Семинары
Пользователи | Соглашение | Мегапроекты
Песочница | Конфиденциальность | Мерч

Сходил, я уж думал, неужели наконец GH придумал что-то стоящее. Но нет, «за кулисами» там простой curl, использующий API токен :(
Для сравнения, как это сделано в Gitlab — обычными push options, которые гит поддерживает из коробки. `git push ... -o merge_request.create -o merge_request.title="My Wonderful MR"`

Ну а в целом алиасы — это примерно то, что в первую рабочую неделю новому разработчику показывается в рамках вводного тренинга по гиту: если чел уже все знает и у него свои алиасы устоявшиеся, то все ок, введем в курс тех, которые у нас в некотором смысле «общие», но никого не заставляем. Но обычно люди не знают гит на том уровне, на котором мы ведем разработку, поэтому предлагается взять наши командные алиасы как «основу» (естественно, с объяснениями, что магии там нет и что за ними стоит), а дальше отпускаем в вольное плавание.

welovelain вчера в 20:40

Я юзаю pull, merge, commit и push 99.99 процентов времени уже лет пять.
Что за мега процессы такие, что приходится пользоваться всякими эзотерическими командами?

0