Mail.ru Group
Строим Интернет

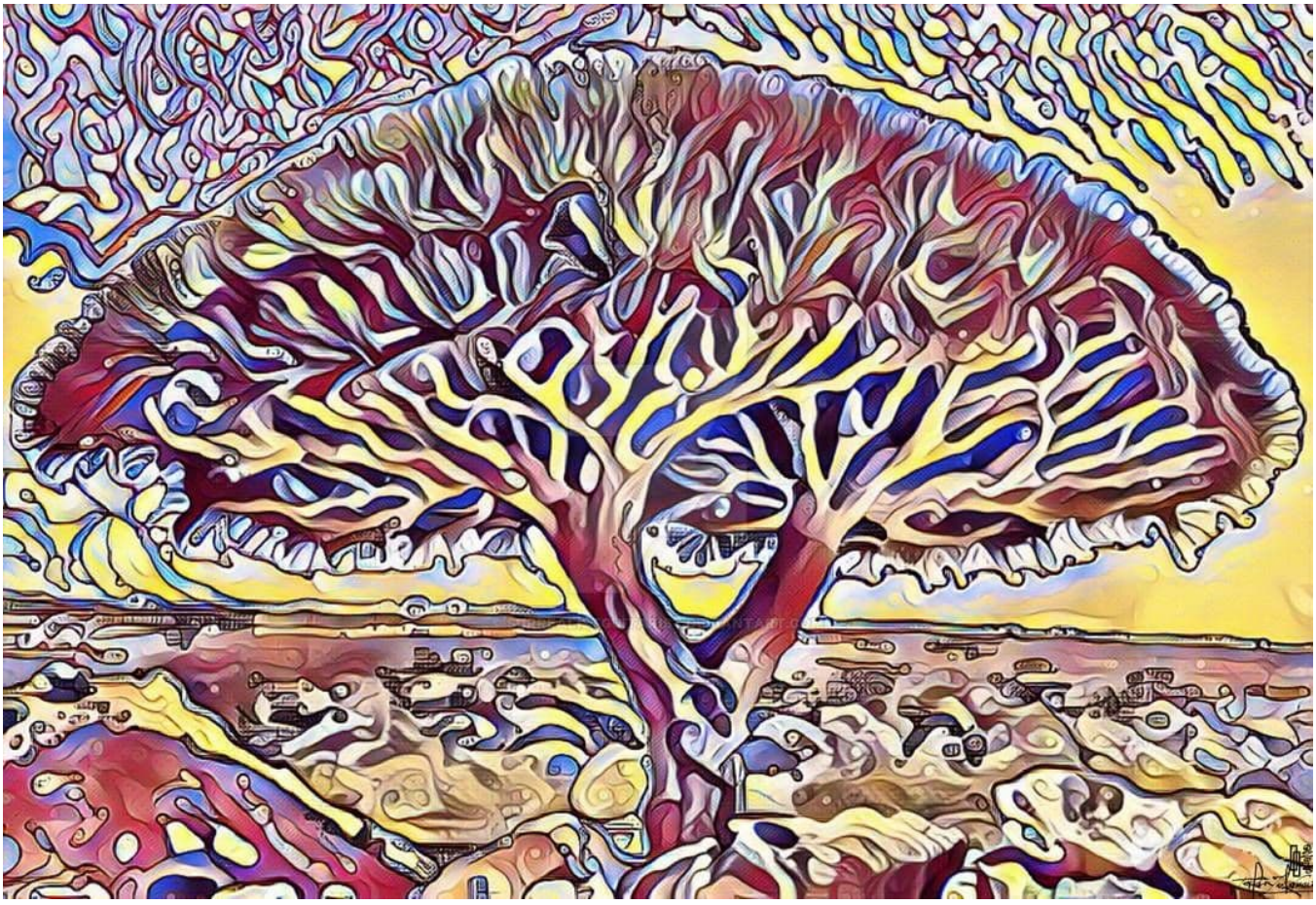
pxeno сегодня в 10:54

Как облегчить себе жизнь при использовании Git (а также подборка материалов для глубокого погружения)

Автор оригинала: Shane Hudson

Блог компании Mail.ru Group, Open source, Git, Системы управления версиями, GitHub

Перевод



Tree of Dragons II by surrealistguitarist

Для тех, кто каждый день использует Git, но чувствует себя неуверенно, команда *Mail.ru Cloud Solutions* перевела статью фронтенд-разработчика Шейна Хадсона. Здесь вы найдете несколько трюков и советов, которые могут немного облегчить работу с Git, а также подборку статей и мануалов более продвинутого уровня.

Git появился почти 15 лет назад. За это время он прошел путь от андердога до непобедимого чемпиона. Сегодня новые

проекты часто начинают с команды `git init`. Несомненно, это важный инструмент, который многие из нас используют ежедневно, но зачастую он напоминает магию — яркую, но опасную.

На Хабре опубликовано много статей, как начать работу с Git, как устроен Git под капотом, описания лучших стратегий ветвления. Здесь автор сфокусировался на том, как упростить работу с Git.

Наводим порядок

Смысл Git в том, чтобы сохранять свою работу, переключать контекст — и делать что-то ещё. Это может быть резервное копирование кода или возможность асинхронно разрабатывать несколько различных функций. Было бы ужасно выбрасывать вторую версию только потому, что в первой обнаружена ошибка. Не менее стыдно сохранять файлы с именами типа v1_final_bug_fixed. Как известно, это ведет к полному бардаку.

Все мы знаем, что жизнь становится гораздо проще, когда наши обновления аккуратно разложены по веткам Git, которыми можно поделиться с коллегами. Но часто возникают ситуации, когда вы меняете контекст, потом возвращаетесь назад — и не можете найти правильную ветвь. Был ли коммит вообще? Может, он скрытый? Может, коммит не прошел, теперь все ушли в неправильную ветку, и всё плохо, и я ужасно справляюсь со своей работой! Да уж, каждый там был и чувствовал такие сомнения. Есть способы справиться с этой ситуацией.

Сортировка веток по дате

Сортировка по дате показывает все ваши локальные ветки, начиная с последней. Довольно банально, но помогало мне много раз:

```
# To sort branches by commit date
git branch --sort=-committerdate
```

Предыдущая ветка

Что делать, если вы не сделали коммит, переключили ветку, а затем захотели вернуться в предыдущую? Вероятно, ее можно найти в списке ветвей, если у вас есть некоторое представление о ее названии. Но что, если это не ветка, а `detached HEAD`, конкретный коммит?

Оказывается, есть простой выход:

```
# Checkout previous branch
git checkout -
```

Оператор `-` представляет собой сокращение для синтаксиса `@{-1}`, который позволяет переключиться на любое количество чекаутов назад. Так что если вы, к примеру, создали ветвь `feature/thing-a`, потом `feature/thing-b`, а потом `bugfix/thing-c`, то параметр `@{-2}` вернет вас к `feature/thing-a`:

```
# Checkout branch N number of checkouts ago
git checkout @{-N}
```

Показать информацию обо всех ветках

Флаг `v` показывает список всех ветвей с последним идентификатором коммита и сообщением. Двойной `vv` также покажет удаленные ветки upstream, за которыми следят локальные ветки:

```
# List branches along with commit ID, commit message and remote
git branch -vv
```

Найти файл

Мы все попадали в такую ситуацию: каким-то образом получилось, что один файл остался не в той ветке. Что делать? Переделать всю работу или скопировать код из одной ветки в другую? Нет, к счастью, есть способ найти конкретный файл.

Способ немного странный, учитывая, что `git checkout -` переносит вас в предыдущую ветвь. В общем, если указать `--` после названия ветки в checkout, то это позволит указать конкретный файл, который вы ищете. О такой функции не догадаешься без подсказки, но она очень удобная, если знать:

```
git checkout feature/my-other-branch -- thefile.txt
```

Понятный статус

Томаш Лакомы твитнул о сокращении выдачи `git status` с помощью флагов `-sb` и добавил: «Много лет уже использую Git, но никто мне об этом не говорил». Речь не только о поиске потерянных файлов. Есть случаи, когда упрощение выдачи облегчает просмотр изменений.

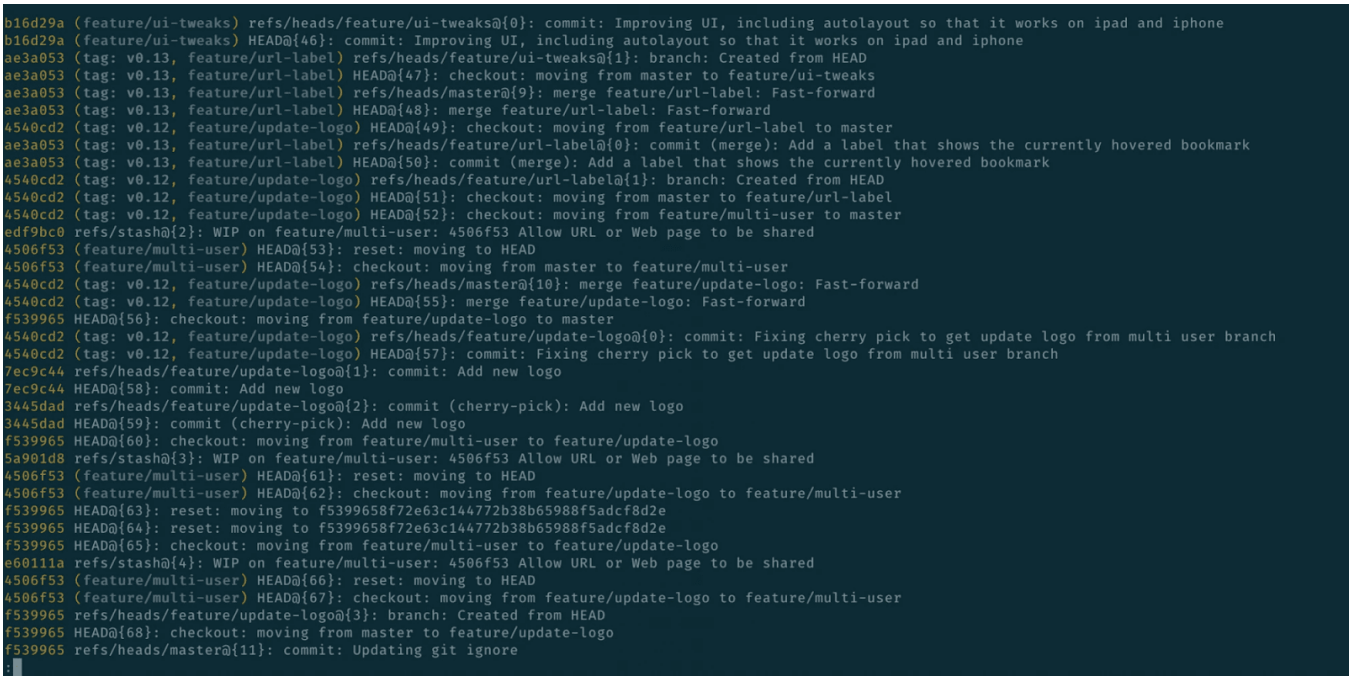
У большинства команд Git есть такие флаги, поэтому стоит изучить, как их использовать для настройки своего рабочего процесса:

```
# Usually we would use git status to check what files have changed
git status

# Outputs:
On branch master
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git checkout -- <file>..." to discard changes in working directory)

modified: README.md

Untracked files:
(use "git add <file>..." to include in what will be committed)

another-file
my-new-file

# Using the flags -sb we can shorten the output
git status -sb

# Outputs:
## master
M README.md
?? another-file
?? my-new-file
```

Вся история

Бывают случаи, когда что-то пошло совершенно неправильно, например, вы случайно отбросили staged (подготовительные) изменения перед их коммитом. Если `git log` не позволяет вернуться в предыдущее состояние и не помогает ни один из вышеперечисленных советов, то есть `git reflog`.

Все ваши действия в Git, которые изменяют содержимое по ссылке `HEAD@{}` (например `push/pull/branch/checkout/commit`), попадают в журнал reflog (reference log). По сути, это история всех ваших действий, независимо от того, в какой ветке вы находитесь. В этом разница с `git log`, который показывает изменения для конкретной ветви.



Вы можете сделать `git show` с ID коммита — и увидеть конкретное изменение. Если это то, что вы искали, то `git checkout` перенесет вас в нужную ветку или даже позволит выбрать конкретный файл, как показано выше:

```
# See the reference log of your activity
git reflog --all

# Look at the HEAD at given point from reflog
git show HEAD@{2}

# Checkout the HEAD, to get back to that point
git checkout HEAD@{2}
```

Подготовительные файлы, которые пропустили коммит

В крайнем случае, если `git reflog` не поможет получить ваши файлы обратно (например, вы запустили жесткий сброс с промежуточными файлами), есть еще один трюк.

Каждое изменение хранится внутри объектов `.git/objects`, которые в активном проекте заполнены файлами, так что там практически невозможно разобраться. Однако существует команда Git под названием `git fsck`, которая используется для проверки целостности (наличия поврежденных файлов) в репозитории. Мы можем использовать ее с флагом `--lost-found` для поиска всех файлов, не связанных с коммитом. Такие файлы называются «висячими блобами» (dangling blob).

Эта команда также позволяет найти «висячие деревья» и «висячие коммиты». Если хотите, можно использовать флаг `--dangling`, но преимущество `--lost-found` в том, что он извлекает все соответствующие файлы в папку `.git/lost-found`. Скорее всего, в активном проекте у вас окажется много таких «висячих» файлов. В Git есть команда очистки от мусора, которая регулярно запускается и удаляет их.

Таким образом, `--lost-found` покажет все файлы и время/дату создания, что значительно облегчает поиск. Обратите внимание, что каждый отдельный файл по-прежнему будет отдельным, то есть вы не можете использовать checkout. Также у всех файлов будут непонятные имена (хэш), поэтому придется скопировать нужные файлы в другое место:

```
# This will find any change that was staged but is not attached to the git tree
git fsck --lost-found

# See the dates of the files
ls -lah .git/lost-found/other/

# Copy the relevant files to where you want them, for example:
cp .git/lost-found/other/73f60804ac20d5e417783a324517eba600976d30 index.html
```

Git в командной работе

Использование Git в одиночку — это одно, но когда вы работаете в команде людей, обычно с совершенно разным опытом, навыками и инструментами, Git может стать как благословением, так и проклятием. Это мощный инструмент для совместного использования одной и той же кодовой базы, проведения код-ревью и наблюдения за прогрессом всей команды. В то же время от всех сотрудников требуется общее понимание, как его использовать в командной работе. Независимо от того, о чем идет речь: о конвенции именования ветвей, форматировании сопроводительного сообщения в коммите или выборе того, какие файлы внести в коммит, — важно обеспечить хорошую коммуникацию и договориться, как использовать этот инструмент.

Всегда важно обеспечить простоту онбординга для новичков и продумать, что произойдет, если они начнут совершать коммиты, не зная принятых в компании принципов и конвенций. Это не конец света, но может вызвать некоторую путаницу и потребует времени, чтобы вернуться к согласованному подходу.

В этом разделе — несколько рекомендаций, как интегрировать принятые соглашения непосредственно в сам репозиторий, автоматизировать и оформить в декларациях максимальное количество задач. В идеальном случае любой новый сотрудник почти сразу начинает работать в том же стиле, что и остальная команда.

Одни и те же окончания строк

По умолчанию Windows использует окончания строк DOS `\r\n` (CRLF), в то время как Mac и Linux используют окончания строк UNIX `\n` (LF), а самые старые версии Mac используют `\r` (CR). Таким образом, по мере роста команды становится более вероятной проблема несовместимых окончаний строк. Это неудобно, они (обычно) не ломают код, но из-за них коммиты и пул-реквесты показывают различные нерелевантные изменения. Часто люди просто игнорируют их, ведь довольно хлопотно ходить и менять все неправильные окончания строк.

Существует решение — вы можете попросить всех членов команды настроить свои локальные конфигурации на автоматическое завершение строк:

```
# This will let you configure line-endings on an individual basis
git config core.eol lf
git config core.autocrlf input
```

Конечно, нужно подписать на эту конвенцию и новичка, о чем легко забыть. Как же сделать это для всей команды? В соответствии с алгоритмом работы Git проверяет наличие конфигурационного файла в репозитории .git/config, затем проверяет общесистемную конфигурацию пользователя в `~/.git/config`, а затем проверяет глобальную конфигурацию в `/etc/gitconfig`.

Всё это хорошо, но оказывается, что ни один из этих файлов конфигурации не может быть установлен через сам репозиторий. Вы можете добавить конфигурации, специфичные для репозитория, но они не будут распространяться на других членов команды.

Однако существует файл, который действительно фиксируется в репозитории. Он называется *.gitattributes*. По умолчанию у вас его нет, поэтому создайте новый файл и сохраните его как `*.gitattributes*`. Он устанавливает атрибуты для каждого файла.

Например, вы можете заставить git diff использовать заголовки exif из файлов изображений вместо того, чтобы пытаться вычислить разницу в бинарных файлах. В этом случае мы можем использовать подстановочный знак, чтобы настройка работала для всех файлов, действуя, по сути, как общий конфигурационный файл для всей команды:

```
# Adding this to your .gitattributes file will make it so all files
# are checked in using UNIX line endings while letting anyone on the team
# edit files using their local operating system’s default line endings.
* text=auto
```

Автоматическое скрытие

Общепринято добавлять в .gitignore скомпилированные файлы (такие как `node_modules/`), чтобы хранить их локально и не заливать в репозиторий. Однако иногда вы все-таки хотите залить файл, но не желаете встречать его потом каждый раз в пул-реквесте.

В такой ситуации (по крайней мере, на GitHub) можно добавить в .gitattributes пути с пометкой `linguist-generated` и убедиться, что .gitattributes лежит в корневой папке репозитория. Это спрячет файлы в пул-реквесте. Они будут в «свернутом» виде: вы по-прежнему можете увидеть факт изменения, но без полного кода.

Всё, что уменьшает стресс и когнитивную нагрузку в процессе код-ревью, улучшает его качество и сокращает время.

Например, вы хотите внести файлы ресурсов (asset) в репозиторий, но не собираетесь их потом изменять и отслеживать, поэтому можно добавить в файл с атрибутами следующую строчку:

```
*.asset linguist-generated
```

Чаще используйте git blame

В статье Гарри Робертса «Маленькие штучки, которые я люблю делать с Git» есть рекомендация для `git blame` (перевод с английского — винить) присвоить алиас `git praise` (перевод с английского — хвалить), чтобы ощущать эту команду как позитивное действие. Конечно, переименование не меняет поведение команды. Но всякий раз, когда на обсуждении заходит речь об использовании функции `git blame`, все напрягаются, и я, конечно, тоже. Вполне естественно воспринимать слово blame (вина) как нечто негативное… но это совершенно неправильно!

Мощная функция `git blame` (или `git praise`, если хотите) показывает, кто последним работал с данным кодом. Мы не собираемся его обвинять или хвалить, а просто хотим прояснить ситуацию. Становится понятнее, какие задавать вопросы и кому, что экономит время.

Следует представлять `git blame` не только как нечто хорошее, но и как средство коммуникации, которое помогает всей команде уменьшить хаос и не тратить время на выяснение, кому что известно. Некоторые IDE, такие как Visual Studio, активируют эту функцию в качестве аннотаций. Для каждой функции вы мгновенно видите, кто последним ее изменил (и, следовательно, с кем об этом говорить).

Аналог git blame для пропавших файлов

Недавно я видел, как разработчик в нашей команде пытался выяснить, кто удалил файл, когда и почему. Кажется, что здесь может помочь `git blame`, но она работает со строками в файле и бесполезна в том случае, если файл отсутствует.

Однако есть и другое решение. Старый верный `git log`. Если вы посмотрите на лог без аргументов, то увидите длинный список всех изменений в текущей ветке. Можете добавить идентификатор коммита, чтобы увидеть журнал этого конкретного коммита, но если указать `--` (который мы использовали ранее для таргетинга на определенный файл), можно получить журнал для файла — даже того, которого больше не существует:

```
# By using -- for a specific file,
# git log can find logs for files that were deleted in past commits
git log -- missing_file.txt
```

Шаблон сообщения коммитов

Сообщения коммитов часто нуждаются в улучшении. Рано или поздно разработчики в команде приходят к такому выводу. Есть много способов улучшения. Например, можно ссылаться на идентификаторы багов из внутреннего инструмента управления проектами или, возможно, поощрить написание хоть какого-то текста вместо пустого сообщения.

Эту команду нужно запускать вручную каждый раз, когда кто-то клонирует репозиторий, поскольку конфигурационные файлы не фиксируются в репозитории. Однако она удобна, потому что можно завести общий файл с любым именем, который действует как шаблон сообщения коммитов:

```
# This sets the commit template to the file given,
# this needs to be run for each contributor to the repository.
git config commit.template ./template-file
```

Git для автоматизации

Git является мощным средством автоматизации. Это не сразу очевидно, но подумайте сами: он видит всю вашу активность в репозитории — плюс активность других участников — и у него много информации, которая может быть очень полезной.

Хуки Git

Довольно часто вы видите, что участники команды во время работы выполняют повторяющиеся задачи. Это может быть проверка прохождения тестов и линтеров перед отправкой ветки на сервер (хук перед отправкой) или принудительная стратегия именования ветвей (хук перед коммитом). На эту тему Константинос Леймонис в журнале Smashing Magazine написал статью под названием «Как упростить рабочий процесс с помощью хуков Git».

Ручная автоматизация

Одной из ключевых функций автоматизации в Git является git bisect. Многие слышали о ней, но мало кто использует. Суть в обработке дерева Git (истории коммитов) и поиске, в каком месте введена ошибка.

Самый простой способ сделать это — вручную. Вы запускаете `git bisect start`, задаете идентификаторы хорошего и плохого коммитов (где нет бага и где есть баг), затем выполняете `git bisect good` или `git bisect bad` для каждого коммита.

Это более мощная фича, чем кажется на первый взгляд, потому что она не прогоняет линейно журнал Git, что можно было сделать вручную как итеративный процесс. Вместо этого она использует двоичный поиск, который эффективно проходит по коммитам с наименьшим количеством шагов:

```
# Begin the bisect
git bisect start

# Tell git which commit does not have the bug
git bisect good c5ba734

# Tell git which commit does have the bug
git bisect bad 6c093f4

# Here, do your test for the bug.
# This could be running a script, doing a journey on a website, unit test etc.

# If the current commit has bug:
git bisect bad

# If the current commit does not have the bug
git bisect good

# This will repeat until it finds the first commit with the bug
# To exit the bisect, either:

# Go back to original branch:
git bisect reset

# Or stick with current HEAD
git bisect reset HEAD

# Or you can exit the bisect at a specific commit
git bisect reset <commit ID>
```

Идем дальше: автоматизация научным методом

В своем докладе «Отладка с помощью научного метода» Стюарт Хэллоуэй объяснил, как использовать команду `git bisect` для автоматизации отладки.
Он фокусируется на Clojure, но нам необязательно знать этот язык, чтобы извлечь пользу из его доклада.

> Git bisect — это отчасти автоматизация научным методом. Вы пишете небольшую программу, которая будет что-то тестировать, а Git прыгает туда-сюда, с каждым прыжком разрезая мир пополам, пока не нащупает ту границу, на которой

ваш тест меняет состояние.
Стюарт Хэллоуэй

Поначалу `git bisect` может показаться интересной и довольно крутой функцией, но в итоге она не очень полезна. Выступление Стюарта в значительной степени показывает, что на самом деле контрпродуктивно выполнять отладку так, как мы привыкли. Если вместо этого сосредоточиться на эмпирических фактах, проходит тест или нет, то вы можете запустить его на всех коммитах, начиная с рабочей версии, и уменьшить ощущение «блуждания в темноте», к которому мы привыкли.

Так как же нам автоматизировать `git bisect`? Можно передать ему скрипт для каждого соответствующего коммита. Ранее я говорил, что можно вручную запускать скрипт на каждом шаге bisect, но если передать команду для запуска, то он будет автоматически запускать скрипт на каждом шаге. Это может быть скрипт специально для отладки этой конкретной проблемы или тест (модульный, функциональный, интеграционный, любой тип теста). Таким образом, вы можете написать тест для проверки, что регрессия не повторяется, и запустить этот тест на предыдущих коммитах:

```
# Begin the bisect
git bisect start

# Tell git which commit does not have the bug
git bisect good c5ba734

# Tell git which commit does have the bug
git bisect bad 6c093f4

# Tell git to run a specific script on each commit
# For example you could run a specific script:
git bisect run ./test-bug

# Or use a test runner
git bisect run jest
```

На каждом прошлом коммите

Одной из сильных сторон `git bisect` является эффективное использование бинарного поиска для обхода всех событий в истории нелинейным способом. Но иногда нужен именно линейный обход. Вы можете написать скрипт, который читает журнал Git и циклически выполняет код на каждом коммите. Но есть знакомая команда, которая сделает это за вас: `git rebase`.

Камран Ахмед в твите написал, как `rebase` находит, какой коммит не проходит тест:

> Находим коммит, который не проходит тест:
>
> ```
> $ git rebase -i --exec "yarn test" d294ae9
> ```
>
> Команда запускает «yarn test» на всех коммитах между d294ae9 и HEAD и останавливается на том коммите, где тест падает.

Мы уже рассматривали `git bisect` для выполнения этой задачи, что может быть более эффективно, но в данном случае мы не ограничены одним вариантом использования.

Здесь есть место для творчества. Возможно, вы хотите сгенерировать отчет о том, как изменялся код с течением времени (или показать историю тестов), и простого парсинга журнала Git недостаточно. Может, это не самый полезный трюк в этой статье, но он интересный и показывает задачу, в реальность выполнения которой мы могли и не поверить раньше:

```
# This will run for every commit between current and the given commit ID
git rebase -i --exec ./my-script
```

Подборка статей и мануалов, которые помогут углубиться в Git

В такой статье невозможно углубиться в тему, иначе получится целая книга. Я выбрал несколько маленьких трюков, о которых могут не знать даже опытные пользователи. Но в Git гораздо больше возможностей: от базовых функций до сложнейших скриптов, точных конфигураций и интеграции в консоль. Поэтому вот некоторые ресурсы, которые могут быть вам интересны:

1. Git Explorer. Интерактивный сайт, помогающий легко понять, как добиться того, что вы хотите.
2. Dang it Git!. Каждый из нас в какой-то момент может потеряться в Git и не знает, как решить какую-либо проблему. Этот сайт подсказывает решения для многих самых распространенных проблем.
3. Pro Git. Эта бесплатная книга — бесценный ресурс для понимания Git.
4. Git Docs. Это уже стало мемом — говорить разработчикам прочитать мануал. Но серьезно, и Git Docs, и маны по Git (например, man git-commit) подробно рассказывают о свойствах Git и могут быть действительно полезны.
5. Thoughtbot. В категории Git на Thoughtbot есть несколько очень полезных советов по использованию этого инструмента.
6. Хуки Git. На сайте собраны ресурсы и идеи для всевозможных хуков Git.
7. Демистификация внутреннего устройства Git. Деревья, блобы… эти термины могут показаться немного странными. Данная статья объясняет некоторые из основных принципов внутреннего устройства Git, которые могут быть полезны для реализации его потенциала по максимум.
8. Git: от новичка к продвинутому уровню. Майк Ритмюллер написал эту полезную статью, которая идеально подходит для начинающих пользователей Git.
9. Маленькие штучки, которые я люблю делать с Git. Именно эта статья Гарри Робертса заставила меня понять, как много возможностей еще скрывается в Git, кроме перемещения кода по веткам.
10. Продвинутые руководства по Git от Atlassian. Эти учебные пособия подробно описывают многие темы, упомянутые в этой статье.
11. Шпаргалка по Git на Github. Всегда удобно иметь хорошую шпаргалку для таких инструментов, как Git.
12. Сокращения Git. Эта статья подробно описывает различные флаги команд Git и рекомендует множество алиасов.

А вот, что еще можно почитать:

1. Руководство по Git. Часть №1: все, что нужно знать про каталог .git
2. Руководство по Git. Часть №2: золотое правило и другие основы rebase.
3. Наш канал в Телеграме о цифровой трансформации.

Теги: mail.ru cloud solutions, Git, Github, git checkout, git status, lost-found, автоматизация, git bisect

Хабы: Блог компании Mail.ru Group, Open source, Git, Системы управления версиями, GitHub


Андрей Пшеничнов @pxeno
Специалист по облакам

Mail.ru Group
Строим Интернет

ПОХОЖИЕ ПУБЛИКАЦИИ

2 августа 2019 в 13:40

Техновыпуск Mail.ru Group 2019

+39 3,9k 15 0

18 июля 2019 в 18:45

Опыт моделеварения от команды Computer Vision Mail.ru

+44 8,2k 57 9

14 июня 2019 в 10:51

@Kubernetes Meetup #3 в Mail.ru Group: 21 июня

+29 2,3k 9 0

Комментарии 1

AnthonyKot сегодня в 15:50 0

Спасибо, полезная статья

Только полноправные пользователи могут оставлять комментарии. Войдите, пожалуйста.

САМОЕ ЧИТАЕМОЕ

Сутки | Неделя | Месяц

Apple Watch за недорого: как меня хотели «развести» на Авито и Юле

+162 45,1k 41 214

На корпусе вашего компьютера напряжение 110 Вольт

+64 18,8k 44 118

Вскрываем чип гальванической развязки с крохотным трансформатором внутри

+57 21,3k 54 14

Как мы переучивали поддержку разговаривать по-человечески, и что получилось

+72 16,5k 61 70

Сатья Наделла: постоянная работа из дома вредит психике и социальным навыкам

+18 9k 5 46

Если нашли опечатку в посте, выделите ее и нажмите Ctrl+Enter, чтобы сообщить автору.