Бесплатная электронная книга

УЧУСЬ Git

Free unaffiliated eBook created from
Stack Overflow contributors.

#git

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: git

It is an unofficial and free Git ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Git.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с Git

замечания

Git - это бесплатная, распределенная система управления версиями, которая позволяет программистам отслеживать изменения кода через «моментальные снимки» (коммиты) в текущем состоянии. Использование коммитов позволяет программистам тестировать, отлаживать и создавать новые функции совместно. Все коммиты хранятся в так называемом «Репозитории Git», который может размещаться на вашем компьютере, на частных серверах или на сайтах с открытым исходным кодом, например, в Github.

Git также позволяет пользователям создавать новые «ветви» кода, которые позволяют различным версиям кода жить рядом друг с другом. Это позволяет сценарии, в которых одна ветвь содержит самую последнюю стабильную версию, другая ветвь содержит набор новых функций, которые разрабатываются, а еще одна ветвь содержит другой набор функций. Гит делает процесс создания этих ветвей, а затем объединяя их вместе, почти безболезненно.

Git имеет 3 разных «области» для вашего кода:

- **Рабочий каталог** : область, в которой вы будете выполнять всю свою работу (создание, редактирование, удаление и организация файлов)
- **Область постановки** : область, в которой вы перечислите изменения, внесенные в рабочий каталог
- **Репозиторий** : где Git постоянно сохраняет изменения, внесенные вами в разные версии проекта

Первоначально Git был создан для управления исходным кодом ядра Linux. Благодаря тому, что они легче, они поощряют мелкие коммиты, разрастание проектов и слияние между вилами и наличие много недолговечных филиалов.

Самое большое изменение для людей, которые привыкли к CVS или Subversion, состоит в том, что каждая проверка содержит не только исходное дерево, но и всю историю проекта. Обычные операции, такие как изменение версий, проверка старых версий, фиксация (для вашей локальной истории), создание ветки, проверка другой ветви, объединение ветвей или файлов патчей, могут выполняться локально без необходимости связываться с центральным сервером. Таким образом, самый большой источник латентности и ненадежности удаляется. Общение с репозиторией «вверх по течению» необходимо только для получения последних изменений и публикации локальных изменений для других разработчиков. Это превращает то, что ранее было техническим ограничением (кто имеет репозиторий владеет проектом) в организационный выбор (ваш «вверх по течению» - это тот, с кем вы хотите синхронизировать).

Версии

Версия	Дата выхода
2,13	2017-05-10
2,12	2017-02-24
2.11.1	2017-02-02
2,11	2016-11-29
2.10.2	2016-10-28
2,10	2016-09-02
2,9	2016-06-13
2,8	2016-03-28
2,7	2015-10-04
2,6	2015-09-28
2.5	2015-07-27
2,4	2015-04-30
2,3	2015-02-05
2,2	2014-11-26
2,1	2014-08-16
2,0	2014-05-28
1,9	2014-02-14
1.8.3	2013-05-24
1,8	2012-10-21
1.7.10	2012-04-06
1,7	2010-02-13
1.6.5	2009-10-10
1.6.3	2009-05-07

Версия	Дата выхода
1,6	2008-08-17
1.5.3	2007-09-02
1,5	2007-02-14
1.4	2006-06-10
1,3	2006-04-18
1.2	2006-02-12
1,1	2006-01-08
1,0	2005-12-21
0,99	2005-07-11

Examples

Создайте свой первый репозиторий, затем добавьте и скопируйте файлы

В командной строке сначала убедитесь, что у вас установлен Git:

Во всех операционных системах:

```
git --version
```

В UNIX-подобных операционных системах:

```
which git
```

Если ничего не будет возвращено или команда не будет распознана, вам может потребоваться установить Git в вашу систему, загрузив и запустив программу установки. См. Домашнюю страницу Git для исключительно четких и простых инструкций по установке.

После установки Git настройте свое имя пользователя и адрес электронной почты . Сделайте это, *прежде* чем совершать фиксацию.

После установки Git перейдите в каталог, который вы хотите разместить под управлением версии, и создайте пустой репозиторий Git:

```
git init
```

Это создает скрытую папку `.git` , которая содержит сантехнику, необходимую для работы Git.

Затем проверьте, какие файлы Git добавит в ваш новый репозиторий; этот шаг стоит особого внимания:

```
git status
```

Просмотрите полученный список файлов; вы можете указать Git, какой из файлов будет размещен в контроле версий (не добавляйте файлы с конфиденциальной информацией, такой как пароли или файлы, которые просто загромождают репо):

```
git add <file/directory name #1> <file/directory name #2> < ... >
```

Если все файлы в списке должны быть доступны всем, у кого есть доступ к репозиторию, одна команда добавит все в ваш текущий каталог и его подкаталоги:

```
git add .
```

Это «сценирует» все файлы, которые будут добавлены в элемент управления версиями, готовясь к их совершению в первом коммите.

Для файлов, которые вы хотите никогда не контролировать с помощью версии, создайте и `.gitignore` файл с именем `.gitignore` перед запуском команды `add` .

Зафиксируйте все файлы, которые были добавлены, вместе с сообщением фиксации:

```
git commit -m "Initial commit"
```

Это создает новую фиксацию с данным сообщением. Конец - это как сохранение или моментальный снимок всего проекта. Теперь вы можете нажать или загрузить его в удаленный репозиторий, а затем вы можете вернуться к нему, если это необходимо. Если вы опустите параметр `-m` , откроется ваш редактор по умолчанию, и вы сможете редактировать и сохранять сообщение о фиксации там.

Добавление удаленного

Чтобы добавить новый удаленный доступ, используйте команду `git remote add` на терминале, в каталоге, в котором хранится ваш репозиторий.

Команда `git remote add` принимает два аргумента:

1. Удаленное имя, например, `origin`
2. Удаленный URL-адрес, например `https://<your-git-service-address>/user/repo.git`

```
    git remote add origin https://<your-git-service-address>/owner/repository.git
```

ПРИМЕЧАНИЕ. Перед добавлением удаленного вам необходимо создать необходимый репозиторий в своей службе git, после добавления пульта вы сможете нажать / вытащить фиксации.

Клонировать хранилище

Команда `git clone` используется для копирования существующего репозитория Git с сервера на локальный компьютер.

Например, чтобы клонировать проект GitHub:

```
cd <path where you'd like the clone to create a directory>
git clone https://github.com/username/projectname.git
```

Чтобы клонировать проект BitBucket:

```
cd <path where you'd like the clone to create a directory>
git clone https://yourusername@bitbucket.org/username/projectname.git
```

Это создает каталог под названием `projectname` на локальном компьютере, содержащий все файлы в удаленном репозитории Git. Сюда входят исходные файлы для проекта, а также подкаталог `.git` который содержит всю историю и конфигурацию для проекта.

Чтобы указать другое имя каталога, например `MyFolder` :

```
git clone https://github.com/username/projectname.git MyFolder
```

Или клонировать в текущем каталоге:

```
git clone https://github.com/username/projectname.git .
```

Замечания:

1. При клонировании в указанный каталог каталог должен быть пустым или не существующим.
2. Вы также можете использовать команду `ssh` для команды:

   ```
   git clone git@github.com:username/projectname.git
   ```

Версия `https` версия `ssh` эквивалентны. Однако некоторые сервисы хостинга, такие как GitHub, рекомендуют использовать `https` вместо `ssh` .

Настройка удаленного пула

Если вы клонировали вилку (например, проект с открытым исходным кодом в Github), у вас

может не быть push-доступ к восходящему репозиторию, поэтому вам нужна ваша вилка, но вы сможете получить репозиторий вверх по течению.

Сначала проверьте удаленные имена:

```
$ git remote -v
origin    https://github.com/myusername/repo.git (fetch)
origin    https://github.com/myusername/repo.git (push)
upstream  # this line may or may not be here
```

Если `upstream` уже есть (в *некоторых* версиях Git), вам нужно установить URL-адрес (в настоящее время он пуст):

```
$ git remote set-url upstream https://github.com/projectusername/repo.git
```

Если выше **не** существует, или если вы хотите добавить друг / вилку коллег (в настоящее время они не существуют):

```
$ git remote add upstream https://github.com/projectusername/repo.git
$ git remote add dave https://github.com/dave/repo.git
```

Общий код

Чтобы поделиться своим кодом, вы создаете репозиторий на удаленном сервере, на который вы скопируете локальный репозиторий.

Чтобы свести к минимуму использование пространства на удаленном сервере, вы создаете пустой репозиторий: тот, который имеет только объекты `.git` и не создает рабочую копию в файловой системе. В качестве бонуса вы устанавливаете этот пульт в качестве восходящего сервера, чтобы легко обмениваться обновлениями с другими программистами.

На удаленном сервере:

```
git init --bare /path/to/repo.git
```

На локальной машине:

```
git remote add origin ssh://username@server:/path/to/repo.git
```

(Обратите внимание, что `ssh:` это всего лишь один из возможных способов доступа к удаленному репозиторию.)

Теперь скопируйте локальный репозиторий на удаленный компьютер:

```
git push --set-upstream origin master
```

Добавление `--set-upstream` (или `-u`) создало `--set-upstream` (отслеживание), которая используется без аргументов Git-команд, например `git pull` .

Настройка имени пользователя и электронной почты

```
You need to set who you are *before* creating any commit.  That will allow commits to have the
right author name and email associated to them.
```

Это не имеет ничего общего с аутентификацией при нажатии в удаленный репозиторий (например, при нажатии в удаленный репозиторий с использованием вашей учетной записи GitHub, BitBucket или GitLab)

Чтобы объявить этот идентификатор для *всех* репозиториев, используйте `git config --global`
Это сохранит настройку в файле `.gitconfig` вашего пользователя: например `$HOME/.gitconfig` или для Windows, `%USERPROFILE%\.gitconfig` .

```
git config --global user.name "Your Name"
git config --global user.email mail@example.com
```

Чтобы объявить идентификатор для одного репозитория, используйте `git config` внутри репо.
Это сохранит параметр внутри отдельного репозитория, в файле `$GIT_DIR/config` . например `/path/to/your/repo/.git/config` .

```
cd /path/to/my/repo
git config user.name "Your Login At Work"
git config user.email mail_at_work@example.com
```

Настройки, хранящиеся в файле конфигурации репозитория, будут иметь приоритет над глобальной конфигурацией при использовании этого репозитория.

Советы: если у вас разные идентификаторы (один для проекта с открытым исходным кодом, один для работы, один для частных репозиториев и т. Д.), И вы не хотите забывать устанавливать правильный для каждого разного репозитория, над которым вы работаете :

- **Удалить глобальную идентификацию**

  ```
  git config --global --remove-section user.name
  git config --global --remove-section user.email
  ```

2,8

- Чтобы заставить git искать вашу личность только в настройках репозитория, а не в глобальной конфигурации:

```
git config --global user.useConfigOnly true
```

Таким образом, если вы забудете указать свой `user.name` и `user.email` для данного репозитория и попытайтесь сделать фиксацию, вы увидите:

```
no name was given and auto-detection is disabled
no email was given and auto-detection is disabled
```

Изучение команды

Чтобы получить дополнительную информацию о любой команде git, то есть о том, что делает команда, доступных опциях и другой документации, используйте параметр `--help` или команду `help` .

Например, чтобы получить всю доступную информацию о команде `git diff` , используйте:

```
git diff --help
git help diff
```

Аналогично, чтобы получить всю доступную информацию о команде `status` , используйте:

```
git status --help
git help status
```

Если вам нужна только краткая помощь, показывающая значение наиболее используемых флагов командной строки, используйте `-h` :

```
git checkout -h
```

Настройка SSH для Git

Если вы используете **Windows,** открываете Git Bash . Если вы используете **Mac** или **Linux,** откройте свой терминал.

Прежде чем генерировать SSH-ключ, вы можете проверить, есть ли у вас какие-либо существующие ключи SSH.

Перечислите содержимое вашего каталога `~/.ssh` :

```
$ ls -al ~/.ssh
# Lists all the files in your ~/.ssh directory
```

Проверьте список каталогов, чтобы узнать, есть ли у вас общедоступный ключ SSH. По умолчанию имена открытых ключей являются следующими:

```
id_dsa.pub
```

```
id_ecdsa.pub
id_ed25519.pub
id_rsa.pub
```

Если вы видите существующую пару открытых и закрытых ключей, которые вы хотели бы использовать на своей учетной записи Bitbucket, GitHub (или аналогичной), вы можете скопировать содержимое файла `id_*.pub` .

Если нет, вы можете создать новую пару открытого и закрытого ключей со следующей командой:

```
$ ssh-keygen
```

Нажмите клавишу Enter или Return, чтобы принять местоположение по умолчанию. Введите и повторно введите парольную фразу при появлении запроса или оставьте его пустым.

Убедитесь, что ваш SSH-ключ добавлен в ssh-agent. Запустите ssh-agent в фоновом режиме, если он еще не запущен:

```
$ eval "$(ssh-agent -s)"
```

Добавьте ключ SSH к агенту ssh. Обратите внимание, что вам нужно, чтобы te заменил `id_rsa` в команде именем вашего **файла закрытого ключа** :

```
$ ssh-add ~/.ssh/id_rsa
```

Если вы хотите изменить выше существующий репозиторий с HTTPS на SSH, вы можете запустить следующую команду:

```
$ git remote set-url origin ssh://git@bitbucket.server.com:7999/projects/your_project.git
```

Чтобы клонировать новый репозиторий через SSH, вы можете запустить следующую команду:

```
$ git clone ssh://git@bitbucket.server.com:7999/projects/your_project.git
```

Установка Git

Давайте перейдем к использованию Git. Прежде всего, вы должны установить его. Вы можете получить его несколькими способами; два основных из них - установить его из источника или установить существующий пакет для вашей платформы.

Установка из источника

Если это возможно, обычно полезно установить Git из источника, потому что вы получите

самую последнюю версию. Каждая версия Git имеет тенденцию включать полезные усовершенствования пользовательского интерфейса, поэтому получение последней версии часто является лучшим маршрутом, если вам удобнее компилировать программное обеспечение из источника. Кроме того, многие дистрибутивы Linux содержат очень старые пакеты; поэтому, если вы не находитесь на самом современном дистрибутиве или не используете backports, лучше всего установить установку из источника.

Чтобы установить Git, вам нужно иметь следующие библиотеки, которые Git зависит от: curl, zlib, openssl, expat и libiconv. Например, если вы используете систему с yum (например, Fedora) или apt-get (например, на основе Debian), вы можете использовать одну из этих команд для установки всех зависимостей:

```
$ yum install curl-devel expat-devel gettext-devel \
  openssl-devel zlib-devel

$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
  libz-dev libssl-dev
```

Когда у вас есть все необходимые зависимости, вы можете перейти к последнему снимку с веб-сайта Git:

http://git-scm.com/download Затем скомпилируйте и установите:

```
$ tar -zxf git-1.7.2.2.tar.gz
$ cd git-1.7.2.2
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

После этого вы также можете получить Git через Git для обновлений:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

Установка на Linux

Если вы хотите установить Git на Linux через двоичный установщик, вы можете сделать это с помощью основного инструмента управления пакетами, который поставляется вместе с вашим дистрибутивом. Если вы используете Fedora, вы можете использовать yum:

```
$ yum install git
```

Или, если вы используете дистрибутив на базе Debian, например Ubuntu, попробуйте apt-get:

```
$ apt-get install git
```

Установка на Mac

Существует три простых способа установки Git на Mac. Самый простой способ - использовать графический установщик Git, который вы можете загрузить со страницы SourceForge.

http://sourceforge.net/projects/git-osx-installer/

Рисунок 1-7. Установка Git OS X. Другой важный способ - установить Git через MacPorts (http://www.macports.org) . Если у вас установлен MacPorts, установите Git через

```
$ sudo port install git +svn +doc +bash_completion +gitweb
```

Вам не нужно добавлять все дополнительные функции, но вы, вероятно, захотите включить + svn, если вам когда-либо понадобится использовать Git с репозиториями Subversion (см. Главу 8).

Homebrew (http://brew.sh/) - еще одна альтернатива установке Git. Если у вас установлен Homebrew, установите Git через

```
$ brew install git
```

Установка в Windows

Установка Git на Windows очень проста. Проект msysGit имеет одну из более простых процедур установки. Просто загрузите exe-файл установщика с страницы GitHub и запустите его:

```
http://msysgit.github.io
```

После его установки у вас есть версия командной строки (включая клиент SSH, который будет полезен позже) и стандартный графический интерфейс.

Примечание по использованию Windows: вы должны использовать Git с установленной оболочкой msysGit (стиль Unix), она позволяет использовать сложные строки команды, приведенные в этой книге. Если вам по какой-то причине необходимо использовать родную консоль командной строки Windows / командной строки, вам нужно использовать двойные кавычки вместо одиночных кавычек (для параметров с пробелами в них), и вы должны указать параметры, заканчивающиеся на circumflex accent (^), если они являются последними в строке, так как это символ продолжения в Windows.

Прочитайте Начало работы с Git онлайн: https://riptutorial.com/ru/git/topic/218/начало-работы-с-git

глава 2: .mailmap file: Связанные авторы и псевдонимы электронной почты

Синтаксис

- # Только заменить адреса электронной почты <primary@example.org> <alias@example.org>
- # Заменить имя по адресу электронной почты Участник <primary@example.org>
- # Объединить несколько псевдонимов под одним именем и электронной почтой. # Примечание. Это не будет связывать «Other <alias2@example.org>». Участник <primary@example.org> <alias1@example.org> Участник < alias2@example.org>

замечания

Файл `.mailmap` может быть создан в любом текстовом редакторе и представляет собой просто текстовый файл, содержащий дополнительные имена участников, основные адреса электронной почты и их псевдонимы. он должен быть помещен в корень проекта, рядом с каталогом `.git` .

Имейте в виду, что это просто изменяет визуальный вывод команд, таких как `git shortlog` или `git log --use-mailmap` . Это **не** будет переписывать историю фиксации или предотвращать коммит с разными именами и / или адресами электронной почты.

Чтобы предотвратить коммит на основе информации, такой как адреса электронной почты, вместо этого вы должны использовать git hooks .

Examples

Объедините вкладчиков с помощью псевдонимов, чтобы показывать количество фиксаций в коротком сообщении.

Когда вкладчики добавляют к проекту с разных компьютеров или операционных систем, может случиться так, что они используют для этого разные адреса или имена электронной почты, которые будут фрагментировать списки участников и статистику.

Запуск `git shortlog -sn` для получения списка участников и количества коммитов от них может привести к следующему результату:

```
Patrick Rothfuss 871
```

```
Elizabeth Moon 762
E. Moon 184
Rothfuss, Patrick 90
```

Эта фрагментация / диссоциация может быть скорректирована с помощью простого текстового файла `.mailmap` , содержащего сопоставления электронной почты.

Все имена и адреса электронной почты, перечисленные в одной строке, будут связаны с первым именованным объектом соответственно.

В приведенном выше примере отображение может выглядеть так:

```
Patrick Rothfuss <fussy@kingkiller.com> Rothfuss, Patrick <fussy@kingkiller.com>
Elizabeth Moon <emoon@marines.mil> E. Moon <emoon@scifi.org>
```

Как только этот файл существует в корне проекта, запуск `git shortlog -sn` снова приведет к сокращенному списку:

```
Patrick Rothfuss 961
Elizabeth Moon 946
```

Прочитайте .mailmap file: Связанные авторы и псевдонимы электронной почты онлайн: https://riptutorial.com/ru/git/topic/1270/-mailmap-file--связанные-авторы-и-псевдонимы-электронной-почты

глава 3: Diff-дерево

Вступление

Сравнивает содержимое и режим капли, найденные с помощью двух древовидных объектов.

Examples

См. Файлы, измененные в конкретной фиксации

```
git diff-tree --no-commit-id --name-only -r COMMIT_ID
```

использование

```
git diff-tree [--stdin] [-m] [-c] [--cc] [-s] [-v] [--pretty] [-t] [-r] [--root] [<common-
diff-options>] <tree-ish> [<tree-ish>] [<path>...]
```

вариант	объяснение
-p	diff рекурсивно
--root	включить начальную фиксацию как diff против / dev / null

Общие параметры diff

вариант	объяснение
-z	выход diff-raw с линиями, заканчивающимися NUL.
-п	выходной формат патча.
-u	синоним для -p.
--patch-c-сырец	выводит как патч, так и формат diff-raw.
--stat	show diffstat вместо патча.
--numstat	show numeric diffstat вместо патча.
--patch-c-стат	выведите патч и добавьте его diffstat.
--name только	показать только имена измененных файлов.

вариант	объяснение
--name-статус	показать имена и статус измененных файлов.
--full индекс	показать полное имя объекта в индексных строках.
--abbrev = <n>	сокращенные имена объектов в заголовке diff-tree и diff-raw.
-P	swap входные пары файлов.
-B	обнаруживают полные перезаписи.
-M	обнаруживать переименования.
-C	обнаруживать копии.
--find-копии-тверже	попробуйте неизменные файлы в качестве кандидата для обнаружения копии.
-l <n>	limit переименовывает попытки до путей.
-O	переупорядочить diffs в соответствии с.
-S	найдите filepair, у которого только одна сторона содержит строку.
--pickaxe-все	показать все файлы diff, когда используется -S, и найден результат.
-текст	обрабатывать все файлы как текст.

Прочитайте Diff-дерево онлайн: https://riptutorial.com/ru/git/topic/10937/diff-дерево

глава 4: Git Clean

Синтаксис

- `git clean [-d] [-f] [-i] [-n] [-q] [-e <pattern>] [-x | -X] [--] <path>`

параметры

параметр	подробности
-d	Удалите ненужные каталоги в дополнение к необработанным файлам. Если неподписанный каталог управляется другим репозиторием Git, он не удаляется по умолчанию. Используйте вариант -f дважды, если вы действительно хотите удалить такой каталог.
-f, --force	Если переменная конфигурации Git очистится. `requireForce` не установлен в false, git clean откажется удалить файлы или каталоги, если не указано -f, -n или -i. Git откажется удалить каталоги с подкаталог или файл .git, если не указана секунда -f.
-i, --interactive	Интерактивно запрашивает удаление каждого файла.
-n, --dry-run	Отображает только список файлов, которые нужно удалить, без их удаления.
-q, - тихо	Отображаются только ошибки, а не список удаленных файлов.

Examples

Очистить проигнорированные файлы

```
git clean -fX
```

Удалит все проигнорированные файлы из текущего каталога и всех подкаталогов.

```
git clean -Xn
```

Просмотрите все файлы, которые будут очищены.

Очистить все неисследованные каталоги

```
git clean -fd
```

Удалит все необработанные каталоги и файлы внутри них. Он будет запускаться в текущем рабочем каталоге и будет проходить через все подкаталоги.

```
git clean -dn
```

Просмотрите все каталоги, которые будут очищены.

Сильно удалить необработанные файлы

```
git clean -f
```

Удалит все необработанные файлы.

Чистый интерактивный

```
git clean -i
```

Распечатайте элементы, которые нужно удалить, и попросите подтверждение с помощью команд, таких как:

```
Would remove the following items:
  folder/file1.py
  folder/file2.py
*** Commands ***
    1: clean        2: filter by pattern        3: select by numbers        4: ask each
    5: quit        6: help
What now>
```

Интерактивный параметр `i` можно добавить вместе с другими параметрами, такими как `x` , `d` и т. Д.

Прочитайте Git Clean онлайн: https://riptutorial.com/ru/git/topic/1254/git-clean

глава 5: Git Diff

Синтаксис

- `git diff [options] [<commit>] [--] [<path>…]`
- `git diff [options] --cached [<commit>] [--] [<path>…]`
- `git diff [options] <commit> <commit> [--] [<path>…]`
- `git diff [options] <blob> <blob>`
- `git diff [options] [--no-index] [--] <path> <path>`

параметры

параметр	подробности
-p, -u, --patch	Создание патча
-s, -no-patch	Подавление разностного выхода. Полезно для таких команд, как `git show` которые показывают патч по умолчанию, или для отмены эффекта `--patch`
--raw	Создать diff в необработанном формате
--diff-алгоритм =	Выберите алгоритм дифференциала. `myers` следующие варианты: `myers` , `minimal` , `patience` , `histogram`
--резюме	Выведите сжатое резюме расширенной информации заголовка, например, создания, переименования и изменения режима
--name только	Показывать только имена измененных файлов
--name-статус	Показать имена и статусы измененных файлов Наиболее распространенными статусами являются M (Модифицировано), A (Добавлено) и D (Удалено)
-- проверять	Предупреждать, если изменения вводят маркеры конфликтов или ошибки пробелов. То, что считается ошибочными ошибками, управляется конфигурацией `core.whitespace` . По умолчанию заглавные пробелы (включая строки, состоящие исключительно из пробелов) и пробельный символ, который сразу же следует за символом табуляции внутри начального отступа строки, считаются пробельными ошибками. Выходы с ненулевым статусом при обнаружении проблем. Не совместим с --exit-кодом

параметр	подробности
--full индекс	Вместо первых нескольких символов, покажите полные имена объектов до и после изображения в строке «индекс» при генерации выходного файла патча
--binary	В дополнение к `--full-index` вывести двоичную разницу, которая может применяться с применением `git apply`
-текст	Относитесь ко всем файлам в виде текста.
--цвет	Установите цветной режим; т.е. используйте `--color=always` если вы хотите направить diff на меньшее и сохранить окраску git

Examples

Показать отличия в рабочей ветви

```
git diff
```

Это покажет *неустановленные* изменения в текущей ветке от фиксации до нее. Он будет показывать только изменения относительно индекса, то есть показывает, что вы *можете* добавить к следующему фиксации, но не имеет. Чтобы добавить (этап) эти изменения, вы можете использовать `git add .`

Если файл поставлен, но был изменен после его постановки, `git diff` покажет различия между текущим файлом и поэтапной версией.

Показать различия для поэтапных файлов

```
git diff --staged
```

Это покажет изменения между предыдущим фиксацией и текущими поэтапными файлами.

ПРИМЕЧАНИЕ. Вы также можете использовать следующие команды, чтобы выполнить одно и то же:

```
git diff --cached
```

Это всего лишь синоним для `--staged` или

```
git status -v
```

Это вызовет подробные настройки команды `status .`

Показать как поэтапные, так и неустановленные изменения

Чтобы показать все поэтапные *и* неустановленные изменения, используйте:

```
git diff HEAD
```

ПРИМЕЧАНИЕ. Вы также можете использовать следующую команду:

```
git status -vv
```

Разница заключается в том, что вывод последнего фактически скажет вам, какие изменения были поставлены для фиксации, а какие нет.

Показать изменения между двумя коммитами

```
git diff 1234abc..6789def    # old   new
```

Например: Покажите изменения, внесенные в последние 3 фиксации:

```
git diff @~3..@    # HEAD -3   HEAD
```

Примечание: две точки (..) являются необязательными, но добавляются четкость.

Это покажет текстовую разницу между коммитами, независимо от того, где они находятся в дереве.

Использование meld для просмотра всех изменений в рабочем каталоге

```
git difftool -t meld --dir-diff
```

будет отображаться изменения рабочего каталога. С другой стороны,

```
git difftool -t meld --dir-diff  [COMMIT_A] [COMMIT_B]
```

покажет различия между двумя конкретными коммитами.

Показать отличия для определенного файла или каталога

```
git diff myfile.txt
```

Показывает изменения между предыдущим фиксацией указанного файла (`myfile.txt`) и локально модифицированной версией, которая еще не была поставлена.

Это также работает для каталогов:

```
git diff documentation
```

Вышеприведенное показывает изменения между предыдущим фиксацией всех файлов в указанном каталоге (`documentation/`) и локально модифицированными версиями этих файлов, которые еще не были поставлены.

Чтобы показать разницу между некоторой версией файла в данной фиксации и локальной версией `HEAD` вы можете указать фиксацию, которую хотите сравнить:

```
git diff 27fa75e myfile.txt
```

Или если вы хотите увидеть версию между двумя отдельными коммитами:

```
git diff 27fa75e ada9b57 myfile.txt
```

Чтобы показать разницу между версией, указанной хешей `ada9b57` и последней фиксацией на ветке `my_branchname` только для относительного каталога с именем `my_changed_directory/` вы можете сделать это:

```
git diff ada9b57 my_branchname my_changed_directory/
```

Просмотр слова-diff для длинных строк

```
git diff [HEAD|--staged...] --word-diff
```

Вместо того, чтобы отображать строки, это будет отображать различия в строках. Например, вместо:

```
-Hello world
+Hello world!
```

Если вся строка помечена как измененная, `word-diff` изменяет вывод на:

```
Hello [-world-]{+world!+}
```

Вы можете опустить маркеры `[-` , `-]` , `{+` , `+}` , указав `--word-diff=color` или `--color-words` . Для обозначения различия используется только цветовое кодирование:

Просмотр трехстороннего слияния, включая общего предка

```
git config --global merge.conflictstyle diff3
```

Устанавливает стиль `diff3` по умолчанию: вместо обычного формата в конфликтующих разделах, показывающий два файла:

```
<<<<<<< HEAD
left
=======
right
>>>>>>> master
```

он будет включать дополнительный раздел, содержащий исходный текст (исходящий из общего предка):

```
<<<<<<< HEAD
first
second
|||||||
first
=======
last
>>>>>>> master
```

Этот формат упрощает понимание конфликта слиянием, т. Е. в этом случае добавляется локально `second` , а удаленное изменение `first` до `last` , разрешая:

```
last
second
```

Такое же разрешение было бы намного сложнее использовать по умолчанию:

```
<<<<<<< HEAD
first
second
=======
last
>>>>>>> master
```

Покажите разницу между текущей версией и последней версией

```
git diff HEAD^ HEAD
```

Это покажет изменения между предыдущей фиксацией и текущей фиксацией.

Diff UTF-16 закодированные текстовые и двоичные файлы plist

Вы можете различать кодированные файлы UTF-16 (файлы строк локализации os iOS и macOS являются примерами), указав, как git должен различать эти файлы.

Добавьте в файл `~/.gitconfig` .

```
[diff "utf16"]
textconv = "iconv -f utf-16 -t utf-8"
```

`iconv` - программа для преобразования различных кодировок .

Затем отредактируйте или создайте файл `.gitattributes` в корне репозитория, где вы хотите его использовать. Или просто редактируйте `~/.gitattributes` .

```
*.strings diff=utf16
```

Это преобразует все файлы, заканчивающиеся на `.strings` перед git diff.

Вы можете делать подобные вещи для других файлов, которые могут быть преобразованы в текст.

Для двоичных файлов `.gitconfig` вы редактируете `.gitconfig`

```
[diff "plist"]
textconv = plutil -convert xml1 -o -
```

и `.gitattributes`

```
*.plist diff=plist
```

Сравнение ветвей

Покажите изменения между кончиками **`new`** и концом **`original`** :

```
git diff original new     # equivalent to original..new
```

Показать все изменения на **`new`** так как он разветвлен от **`original`** :

```
git diff original...new     # equivalent to $(git merge-base original new)..new
```

Использование только одного параметра, такого как

git diff original

эквивалентно

git diff original..HEAD

Показать изменения между двумя ветвями

```
git diff branch1..branch2
```

Создайте совместимую с патчем diff

Иногда вам просто нужен diff для применения с использованием патча. Обычный `git --diff` не работает. Попробуйте это вместо этого:

```
git diff --no-prefix > some_file.patch
```

Затем в другом месте вы можете изменить его:

```
patch -p0 < some_file.patch
```

разница между двумя фиксацией или ветвью

Чтобы просмотреть разницу между двумя ветвями

```
git diff <branch1>..<branch2>
```

Чтобы просмотреть разницу между двумя ветвями

```
git diff <commitId1>..<commitId2>
```

Чтобы просмотреть diff с текущей ветвью

```
git diff <branch/commitId>
```

Чтобы просмотреть сводку изменений

```
git diff --stat <branch/commitId>
```

Просмотр файлов, которые были изменены после определенного фиксации

```
git diff --name-only <commitId>
```

Просмотр файлов, отличных от ветви

```
git diff --name-only <branchName>
```

Просмотр файлов, которые были изменены в папке после определенного фиксации

```
git diff --name-only <commitId> <folder_path>
```

Прочитайте Git Diff онлайн: https://riptutorial.com/ru/git/topic/273/git-diff

глава 6: Git Patch

Синтаксис

- git am [--signoff] [--keep] [- [no-] keep-cr] [- [no-] utf8] [--3way] [--interactive] [--committer-date-is -автор-дата] [--ignore-date] [--ignore-space-change | --ignore-whitespace] [--whitespace = <option>] [-C <n>] [-p <n>] [--directory = <dir>] [--exclude = <путь>] [- include = <путь>] [--reject] [-q | --quiet] [- [no-] scissors] [-S [<keyid>]] [--patch-format = <format>] [(<mbox> | <Maildir>) ...]
- git am (--continue | --skip | --abort)

параметры

параметр	подробности
(<Mbox> | <Maildir>) ...	Список файлов почтовых ящиков для чтения исправлений. Если вы не укажете этот аргумент, команда будет считываться со стандартного ввода. Если вы предоставите каталоги, они будут рассматриваться как Maildirs.
-s, --signoff	Добавьте строку «Отключено:» в сообщение фиксации, используя идентификатор коммиттера.
-q, --quiet	Будь спокоен. Только печатать сообщения об ошибках.
-u, --utf8	Pass `-u` флаг `git mailinfo` . Предложенное сообщение журнала фиксации, взятое из электронной почты, перекодировано в кодировку UTF-8 (для переменной предпочтительного кодирования проекта можно использовать конфигурационную переменную `i18n.commitencoding` если это не UTF-8). Вы можете использовать `--no-utf8` чтобы переопределить это.
--no-utf8	Pass -n флаг для git mailinfo.
-3, --3way	Когда патч не применяется чисто, откиньтесь на трехстороннее слияние, если патч записывает личность blob, к которому он должен применяться, и мы имеем эти blob-файлы, доступные локально.
--ignore-date, --ignore-space-change, --ignore-whitespace, -	Эти флаги передаются в прикладную программу git, которая применяет патч.

параметр	подробности
-whitespace = <option>, -C <n>, -p <n>, --directory = <dir>, - exclude = <путь>, --include = <путь>, --reject	
--patch-формат	По умолчанию команда попытается автоматически определить формат патча. Эта опция позволяет пользователю обойти автоматическое обнаружение и указать формат патча, который должен интерпретировать патч (ы). Допустимыми форматами являются `mbox` , `stgit` , `stgit-series` и `hg` .
-i, --interactive	Запуск в интерактивном режиме.
--committer-дата-это-автор-дата	По умолчанию команда записывает дату из сообщения электронной почты в качестве даты фиксации фиксации и использует время создания фиксации как дату коммиттера. Это позволяет пользователю лгать о дате коммиттера, используя то же значение, что и дата автора.
--ignore-дата	По умолчанию команда записывает дату из сообщения электронной почты в качестве даты фиксации фиксации и использует время создания фиксации как дату коммиттера. Это позволяет пользователю лгать о дате автора, используя то же значение, что и дата коммиттера.
--пропускать	Пропустить текущий патч. Это имеет смысл только при перезапуске прерванного патча.
-S [<keyid>], -gpg-sign [= <keyid>]	Знак GPG фиксируется.
- продолжить, -r, - разрешено	После отказа патча (например, попытки применить конфликтующий патч) пользователь применил его вручную, а индексный файл сохранил результат приложения. Сделайте фиксацию, используя авторство и зафиксируйте журнал, извлеченный из сообщения электронной почты и текущего файла индекса, и продолжите.
--resolvemsg = <сбщ>	Когда произойдет сбой патча, перед выходом на экран будет `<msg>` . Это отменяет стандартное сообщение,

параметр	подробности
	информирующее вас об использовании - `--continue` или `--skip` для `--skip` сбоя. Это исключительно для внутреннего использования между `git rebase` и `git am`.
--abort	Восстановите исходную ветвь и прервите операцию исправления.

Examples

Создание патча

Чтобы создать патч, есть два шага.

1. Внесите свои изменения и зафиксируйте их.
2. Запустите `git format-patch <commit-reference>` чтобы преобразовать все коммиты с момента фиксации <commit-reference> (не включая) в файлы исправлений.

Например, если исправления должны быть сгенерированы из последних двух коммитов:

```
git format-patch HEAD~~
```

Это создаст 2 файла, по одному для каждой фиксации с `HEAD~~` , например:

```
0001-hello_world.patch
0002-beginning.patch
```

Применение патчей

Мы можем использовать `git apply some.patch` чтобы иметь изменения из файла `.patch` примененного к вашему текущему рабочему каталогу. Они будут неустановленными и должны быть совершены.

Чтобы применить патч как фиксацию (с сообщением фиксации), используйте

```
git am some.patch
```

Чтобы применить все файлы патча к дереву:

```
git am *.patch
```

Прочитайте Git Patch онлайн: https://riptutorial.com/ru/git/topic/4603/git-patch

глава 7: Git Remote

Синтаксис

- `git remote [-v | --verbose]`
- `git remote add [-t <branch>] [-m <master>] [-f] [--[no-]tags] [--mirror=<fetch|push>]<name> <url>`
- `git remote rename <old> <new>`
- `git remote remove <name>`
- `git remote set-head <name> (-a | --auto | -d | --delete | <branch>)`
- `git remote set-branches [--add] <name> <branch>...`
- `git remote set-url [--push] <name> <newurl> [<oldurl>]`
- `git remote set-url --add [--push] <name> <newurl>`
- `git remote set-url --delete [--push] <name> <url>`
- `git remote [-v | --verbose] show [-n] <name>...`
- `git remote prune [-n | --dry-run] <name>...`
- `git remote [-v | --verbose] update [-p | --prune] [(<group> | <remote>)...]`
- `git remote show <name>`

параметры

параметр	подробности
-v, --verbose	Выполните многословие.
-m <мастер>	Настраивает ветвь <master>
--mirror = выборки	Refs не будет храниться в пространстве имен refs / remotes, но вместо этого будет отражено в локальном репо
--mirror = толчок	`git push` будет вести себя так, как если бы --mirror прошло
--no-теги	`git fetch <name>` не импортирует теги из удаленного репо
-t <branch>	Указывает пульт дистанционного управления *только* для отслеживания <branch>
-f	`git fetch <name>` запускается сразу после настройки удаленного
--tags	`git fetch <name>` импортирует каждый тег из удаленного репо
-a, --auto	HEAD символа-ref установлен в ту же ветку, что и HEAD пульта
-d, --delete	Все перечисленные ссылки удалены из удаленного репозитория
--добавлять	Добавляет <имя> в список отслеживаемых в данный момент ветвей

параметр	подробности
	(ветви набора)
--добавлять	Вместо изменения URL-адреса добавляется новый URL-адрес (set-url)
--все	Нажмите все ветви.
--удалять	Все URL-адреса, соответствующие <url>, удаляются. (Установленный URL)
--От себя	Push URLS обрабатываются вместо URL-адресов извлечения
-n	Удаленные головки не запрашиваются сначала с `git ls-remote <name>` , вместо этого используется кешированная информация
--пробный прогон	сообщите, какие ветки будут обрезаны, но на самом деле их не обрезают
--чернослив	Удалите удаленные ветви, у которых нет локального экземпляра

Examples

Добавить удаленный репозиторий

Чтобы добавить удаленный `git remote add` , используйте `git remote add` в корневом каталоге вашего локального репозитория.

Для добавления удаленного репозитория Git <url> в качестве простого короткого имени <имя> используйте

```
git remote add <name> <url>
```

Затем команда `git fetch <name>` может использоваться для создания и обновления ветвей удаленного отслеживания `<name>/<branch>` .

Переименовать удаленный репозиторий

Переименуйте удаленный файл с именем `<old>` в `<new>` . Обновлены все ветви удаленного отслеживания и настройки конфигурации для удаленного.

Чтобы переименовать имя удаленной ветви `dev` на `dev1` :

```
git remote rename dev dev1
```

Удалить удаленный репозиторий

Удалите удаленный файл с именем `<name>` . Удалены все ветви удаленного отслеживания и настройки конфигурации для удаленного устройства.

Чтобы удалить удаленный репозиторий `dev` :

```
git remote rm dev
```

Отображать удаленные репозитории

Чтобы просмотреть все настроенные удаленные репозитории, используйте `git remote` .

Он показывает краткое имя (псевдонимы) каждого удаленного дескриптора, который вы настроили.

```
$ git remote
premium
premiumPro
origin
```

Чтобы показать более подробную информацию, можно использовать флаг `--verbose` или `-v` . Результат будет включать URL-адрес и тип пульта (`push` или `pull`):

```
$ git remote -v
premiumPro    https://github.com/user/CatClickerPro.git (fetch)
premiumPro    https://github.com/user/CatClickerPro.git (push)
premium    https://github.com/user/CatClicker.git (fetch)
premium    https://github.com/user/CatClicker.git (push)
origin    https://github.com/ud/starter.git (fetch)
origin    https://github.com/ud/starter.git (push)
```

Изменить удаленный URL-адрес вашего репозитория Git

Вы можете сделать это, если удаленный репозиторий будет перенесен. Команда для изменения удаленного URL-адреса:

```
git remote set-url
```

Он принимает 2 аргумента: существующее удаленное имя (происхождение, вверх по течению) и URL.

Проверьте текущий удаленный URL:

```
git remote -v
origin    https://bitbucket.com/develop/myrepo.git (fetch)
origin    https://bitbucket.com/develop/myrepo.git (push)
```

Измените свой удаленный URL:

```
git remote set-url origin https://localserver/develop/myrepo.git
```

Еще раз проверьте свой удаленный URL:

```
git remote -v
origin    https://localserver/develop/myrepo.git (fetch)
origin    https://localserver/develop/myrepo.git (push)
```

Показать дополнительную информацию о удаленном репозитории

Вы можете просмотреть дополнительную информацию о удаленном репозитории с помощью `git remote show <remote repository alias>`

```
git remote show origin
```

результат:

```
remote origin
Fetch URL:  https://localserver/develop/myrepo.git
Push  URL:  https://localserver/develop/myrepo.git
HEAD branch: master
Remote branches:
  master      tracked
Local branches configured for 'git pull':
  master      merges with remote master
Local refs configured for 'git push':
  master      pushes to master      (up to date)
```

Прочитайте Git Remote онлайн: https://riptutorial.com/ru/git/topic/4071/git-remote

глава 8: Git rerere

Вступление

`rerere` (повторное использование записанного разрешения) позволяет вам сообщать git о том, как вы разрешили конфликт с hunk. Это позволяет автоматически решать этот вопрос в следующий раз, когда git сталкивается с одним и тем же конфликтом.

Examples

Включение ререре

Чтобы включить `rerere` выполните следующую команду:

```
$ git config --global rerere.enabled true
```

Это можно сделать как в конкретном репозитории, так и во всем мире.

Прочитайте Git rerere онлайн: https://riptutorial.com/ru/git/topic/9156/git-rerere

глава 9: git send-email

Синтаксис

- git send-email [options] <файл | каталог | параметры rev-list> ...
- git send-email --dump-aliases

замечания

https://git-scm.com/docs/git-send-email

Examples

Используйте git send-email с Gmail

Предпосылки: если вы работаете над проектом, например с ядром Linux, вместо того, чтобы делать запрос на вытягивание, вам нужно будет отправить свои коммиты в список рассылки для просмотра. В этой записи подробно описывается использование git-send электронной почты с Gmail.

Добавьте в свой файл .gitconfig следующее:

```
[sendemail]
    smtpserver = smtp.googlemail.com
    smtpencryption = tls
    smtpserverport = 587
    smtpuser = name@gmail.com
```

Затем в Интернете: перейдите в Google -> Моя учетная запись -> Подключенные приложения и сайты -> Разрешить менее безопасные приложения -> Включить

Чтобы создать набор патчей:

```
git format-patch HEAD~~~~ --subject-prefix="PATCH <project-name>"
```

Затем отправьте патчи в список рассылки:

```
git send-email --annotate --to project-developers-list@listserve.example.com 00*.patch
```

Чтобы создать и отправить обновленную версию (версия 2 в этом примере) патча:

```
git format-patch -v 2 HEAD~~~~  ......
git send-email --to project-developers-list@listserve.example.com v2-00*.patch
```

составление

- from * Email От: - [no-] to * Email Кому: - [no-] cc * Email Cc: - [no-] bcc * Email Bcc: --subject * Email "Subject:" - -in-reply-to * Электронная почта «In-Reply-To:» - [no-] xmailer * Добавить заголовок «X-Mailer:» (по умолчанию). - [no-] annotate * Просмотрите каждый патч, который будет отправлен в редакторе. --compose * Откройте редактор для введения. - комком-кодирование * Кодирование, которое предполагается ввести. --8bit-encoding * Кодирование для принятия 8-битных писем, если необъявленное - преобразование-кодирование * Передача кодировки для использования (кавычки, 8 бит, base64)

Отправка патчей по почте

Предположим, у вас есть много обязательств против проекта (здесь ulogd2, официальная ветка - git-svn) и что вы хотите отправить свой набор патчей в список Mailling devel@netfilter.org. Для этого просто откройте оболочку в корне каталога git и используйте:

```
git format-patch --stat -p --raw --signoff  --subject-prefix="ULOGD PATCH" -o /tmp/ulogd2/ -n
git-svn
git send-email --compose --no-chain-reply-to --to devel@netfilter.org /tmp/ulogd2/
```

Первая команда создаст серию писем из патчей в / tmp / ulogd2 / со статистическим отчетом, а вторая запустит ваш редактор, чтобы составить вводное письмо для набора патчей. Чтобы избежать ужасной серии почтовых сообщений, можно использовать:

```
git config sendemail.chainreplyto false
```

источник

Прочитайте git send-email онлайн: https://riptutorial.com/ru/git/topic/4821/git-send-email

глава 10: Git Tagging

Вступление

Как и большинство систем управления версиями (VCS), `Git` имеет возможность `tag` определенные моменты в истории как важные. Обычно эти функции используются для обозначения точек выпуска (`v1.0` и т. Д.).

Синтаксис

- git tag [-a | -s | -u <keyid>] [-f] [-m <msg> | -F <файл>] <tagname> [<commit> | <объект>]
- git tag -d <тэг>
- git tag [-n [<num>]] -l [--contains <commit>] [--contains <commit>] [--points-at <object>] [--column [= <options>] | --no-column] [--create-reflog] [--sort = <key>] [--format = <format>] [-[no-] merged [<commit>]] [<pattern> ...]
- git tag -v [--format = <format>] <tagname> ...

Examples

Список всех доступных тегов

С помощью команды `git tag` перечислены все доступные теги:

```
$ git tag
<output follows>
v0.1
v1.3
```

> **Примечание** : `tags` выводятся в **алфавитном** порядке.

Можно также `search` доступные `tags` :

```
$ git tag -l "v1.8.5*"
<output follows>
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

Создать и нажать тег (ы) в GIT

Создать тег:

- Чтобы создать тег в текущей ветке:

```
git tag < tagname >
```

  Это создаст локальный `tag` с текущим состоянием ветки, в которой вы находитесь.

- Чтобы создать тег с некоторой фиксацией:

```
git tag tag-name commit-identifier
```

  Это создаст локальный `tag` с идентификатором фиксации ветви, в которой вы находитесь.

Нажмите фиксацию в GIT:

- Нажмите отдельный тег:

```
git push origin tag-name
```

- Нажимайте сразу все теги

```
git push origin --tags
```

Прочитайте Git Tagging онлайн: https://riptutorial.com/ru/git/topic/10098/git-tagging

глава 11: Reflog - восстановление коммитов, не показанных в git log

замечания

Рефлог Git записывает позицию HEAD (ref для текущего состояния репозитория) каждый раз, когда он изменяется. Как правило, каждая операция, которая может быть разрушительной, включает в себя перемещение указателя HEAD (поскольку, если что-либо изменится, в том числе и в прошлом, хэш хэша будет изменяться), поэтому всегда можно вернуться к более старому состоянию перед опасной операцией , найдя правильную линию в рефлоге.

Объекты, на которые не ссылаются никакие ссылки, обычно собирают мусор в течение ~ 30 дней, поэтому reflog не всегда может помочь.

Examples

Восстановление от плохой перестановки

Предположим, что вы начали интерактивную перезагрузку:

```
git rebase --interactive HEAD~20
```

и по ошибке вы раздавили или сбросили некоторые коммиты, которые вы не хотели потерять, но затем завершили перезагрузку. Чтобы восстановить, выполните `git reflog` , и вы можете увидеть некоторые результаты следующим образом:

```
aaaaaaa HEAD@{0} rebase -i (finish): returning to refs/head/master
bbbbbbb HEAD@{1} rebase -i (squash): Fix parse error
...
ccccccc HEAD@{n} rebase -i (start): checkout HEAD~20
ddddddd HEAD@{n+1} ...
...
```

В этом случае последнее commit, `ddddddd` (или `HEAD@{n+1}`) является верхушкой ветви *pre-rebase* . Таким образом, чтобы восстановить эту фиксацию (и все родительские коммиты, включая те, которые были случайно раздавлены или сброшены), выполните:

```
$ git checkout HEAD@{n+1}
```

Затем вы можете создать новую ветку при этом фиксации с помощью `git checkout -b [branch]` . Дополнительную информацию см. В разделе « Ветвление» .

Прочитайте Reflog - восстановление коммитов, не показанных в git log онлайн: https://riptutorial.com/ru/git/topic/5149/reflog---восстановление-коммитов--не-показанных-в-git-log

глава 12: Rev-List

Синтаксис

- git rev-list [опции] <commit> ...

параметры

параметр	подробности
--одна линия	Дисплей фиксируется как одна строка с названием.

Examples

Список Задает мастер, но не в оригинале / мастер

```
git rev-list --oneline master ^origin/master
```

Git rev-list будет перечислять коммиты в одной ветви, которые не находятся в другой ветке. Это отличный инструмент, когда вы пытаетесь выяснить, был ли код объединен в ветку или нет.

- С `--oneline` опции `--oneline` будет отображаться заголовок каждого фиксации.
- Оператор `^` исключает коммиты в указанной ветви из списка.
- Вы можете передать более двух филиалов, если хотите. Например, `git rev-list foo bar ^baz` перечисляет коммиты в foo и bar, но не baz.

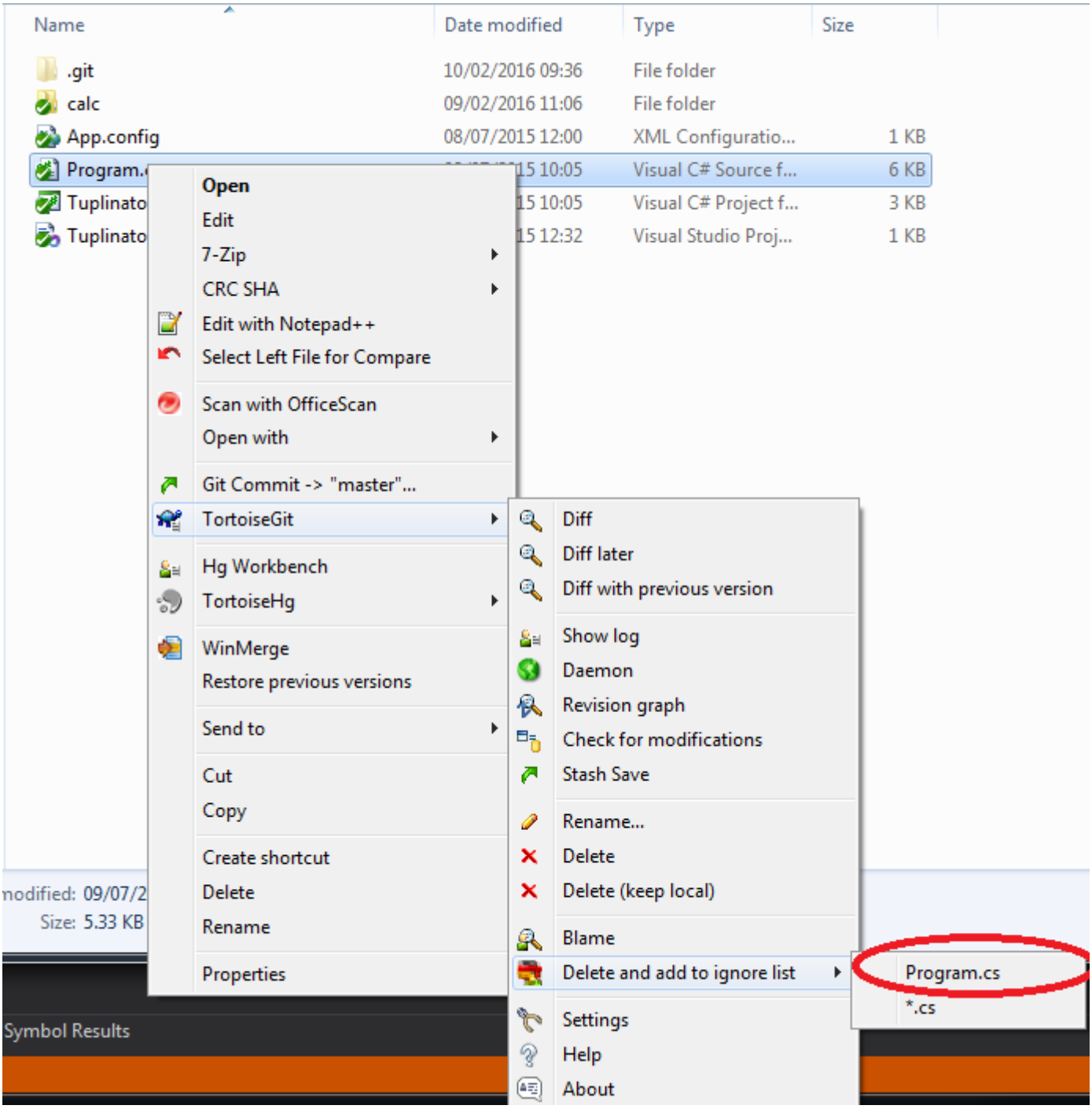
Прочитайте Rev-List онлайн: https://riptutorial.com/ru/git/topic/431/rev-list

глава 13: TortoiseGit

Examples

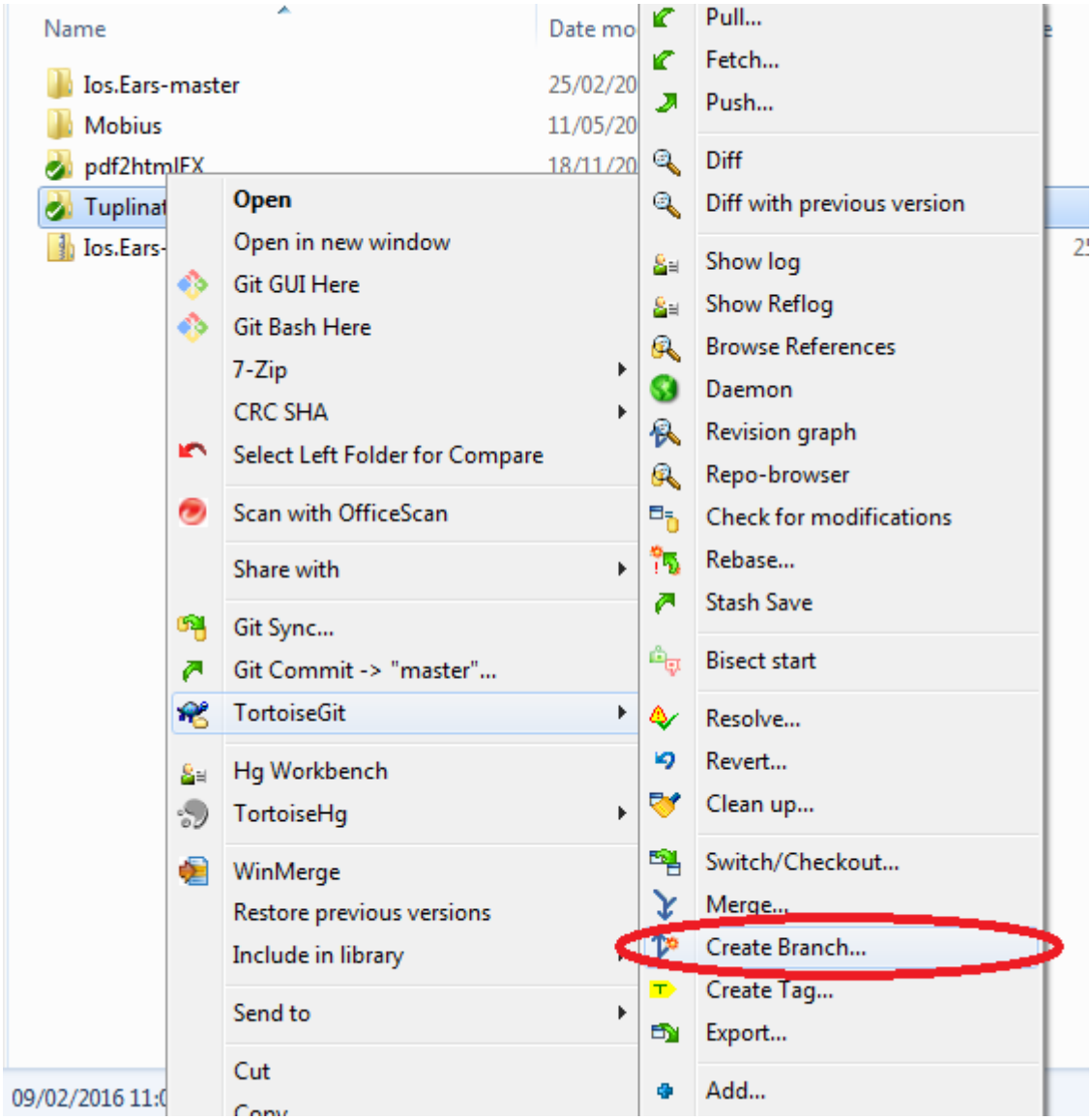
Игнорирование файлов и папок

Те, которые используют TortioseGit UI, нажимают « `Правая мышь`» на файл (или папку), который вы хотите игнорировать, -> `TortoiseGit` -> `Delete and add to ignore list` , здесь вы можете игнорировать все файлы этого типа или этот конкретный диалог -> появится « `Ok` и вы должны сделать это.

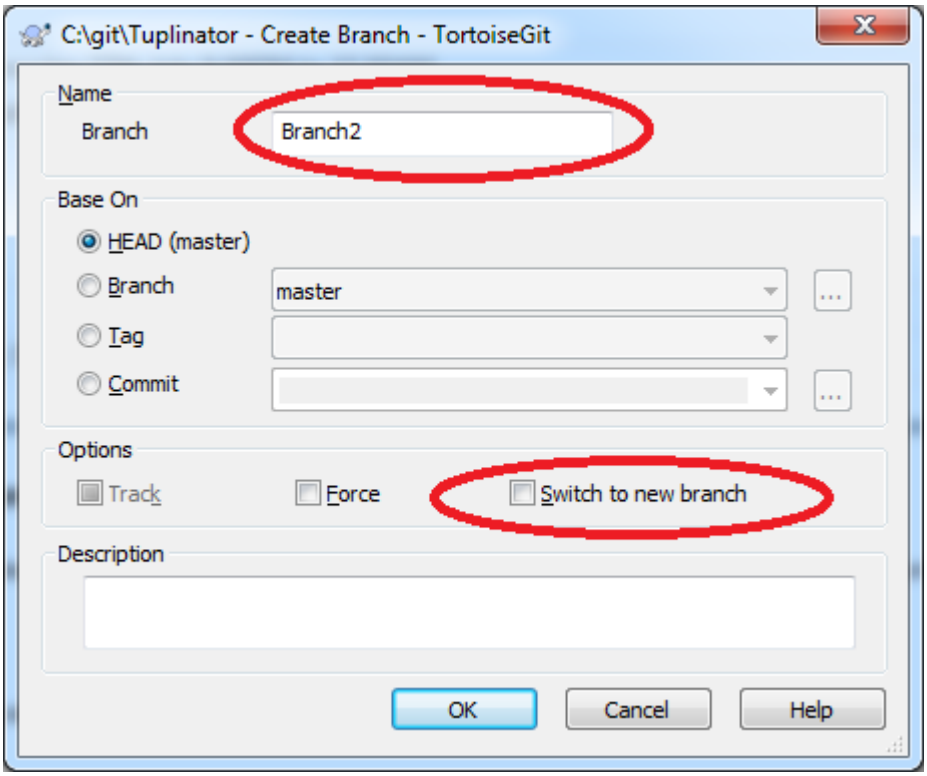


разветвление

Для тех, кто использует UI для перехода, щелкните `правой` кнопкой `мыши` в репозитории, затем `Tortoise Git` -> `Create Branch...`

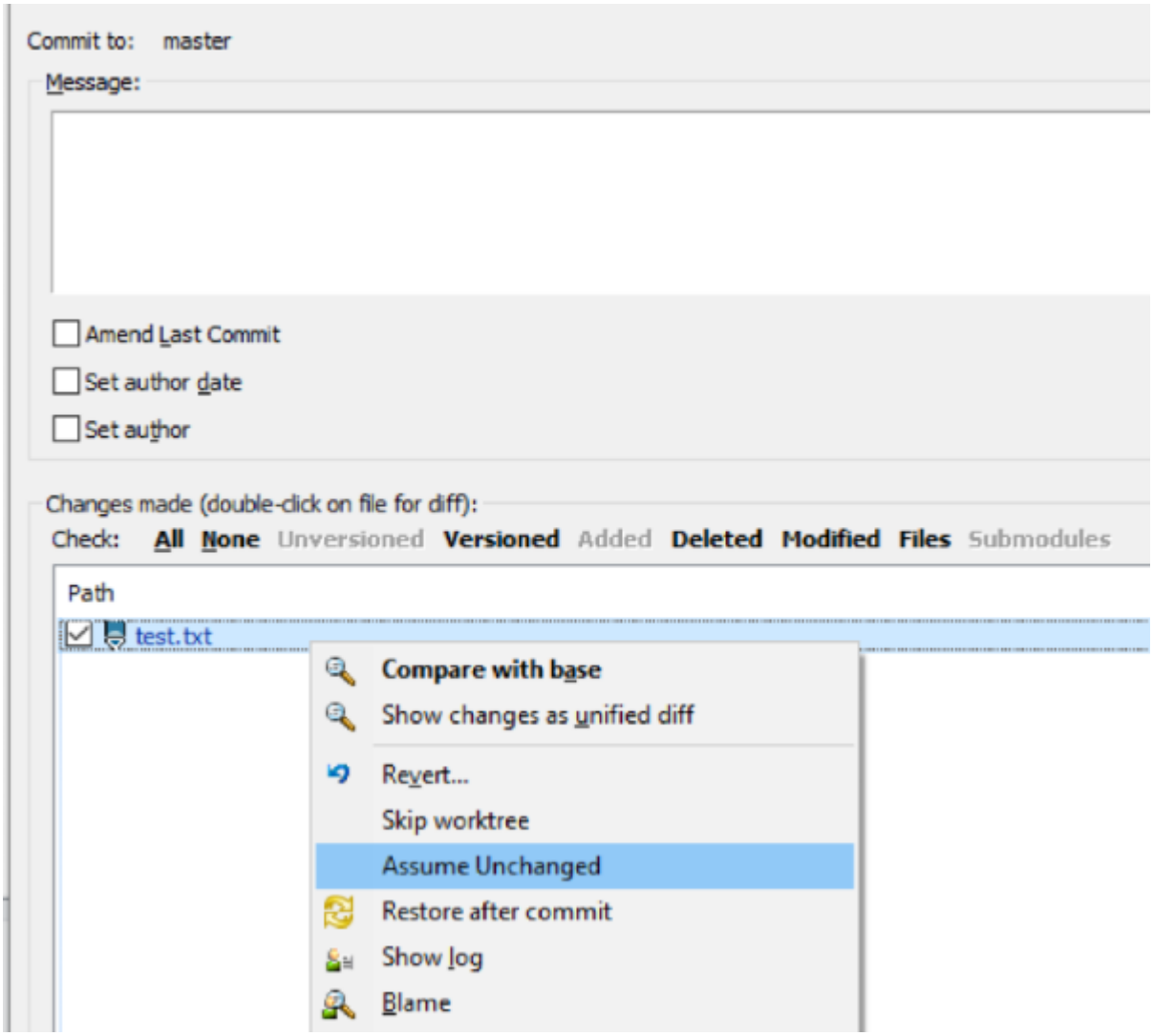


Откроется новое окно -> `Give branch a name` -> Отметьте поле « `Switch to new branch` (возможно, вы хотите начать работать с ней после разветвления). -> Нажмите « `OK` и вы должны сделать это.



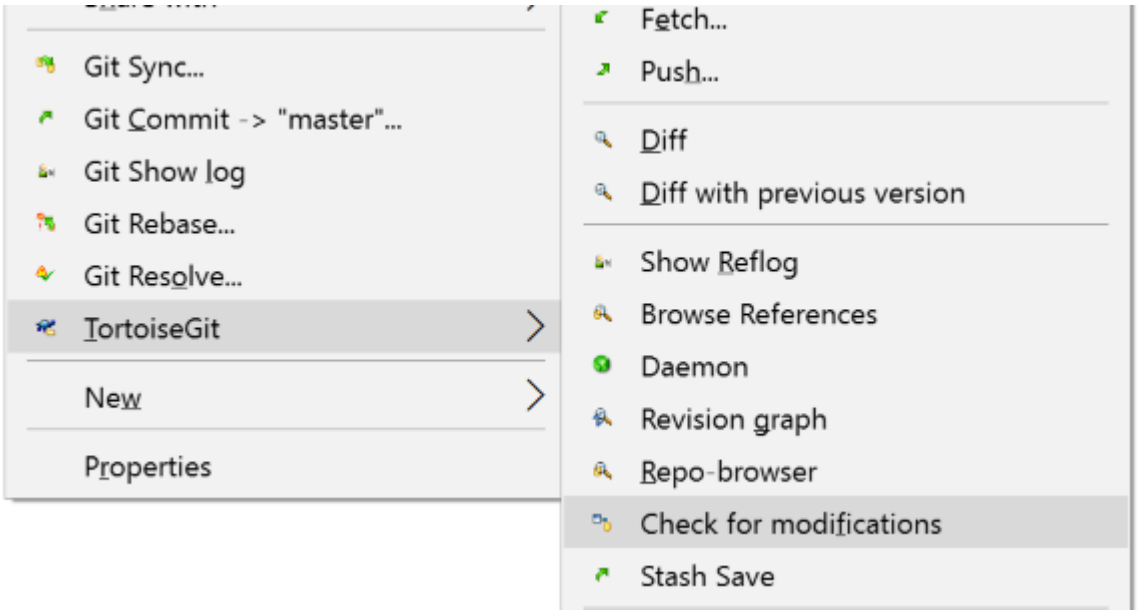
Предположим, что

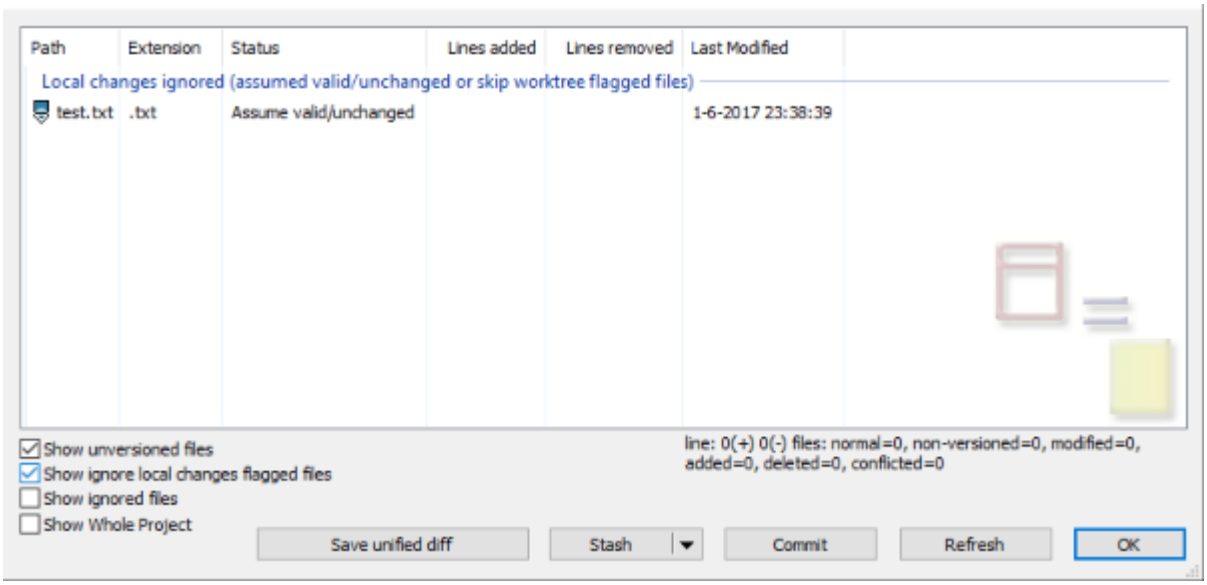
Если файл изменен, но вы его не любите, установите его как «Предположите без изменений»,

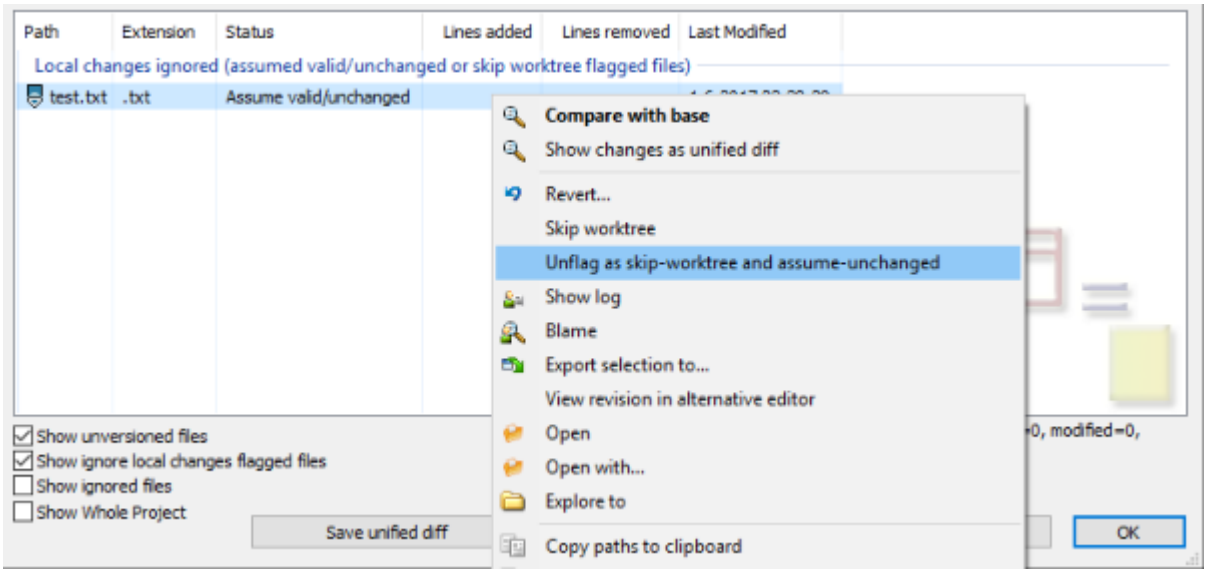


Вернуть «Предположим, что не изменилось»

Нужны некоторые шаги:



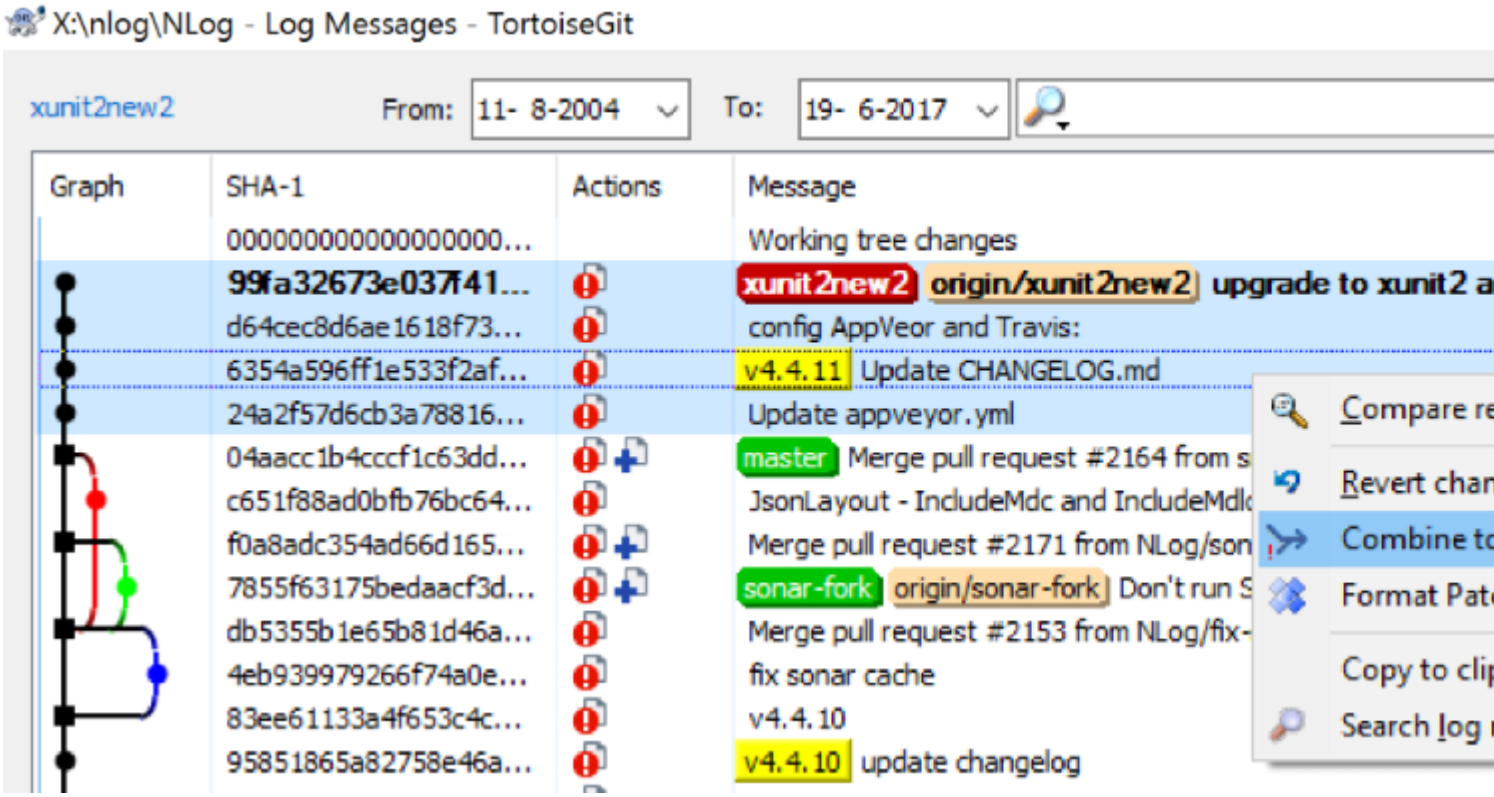




Сквош фиксирует

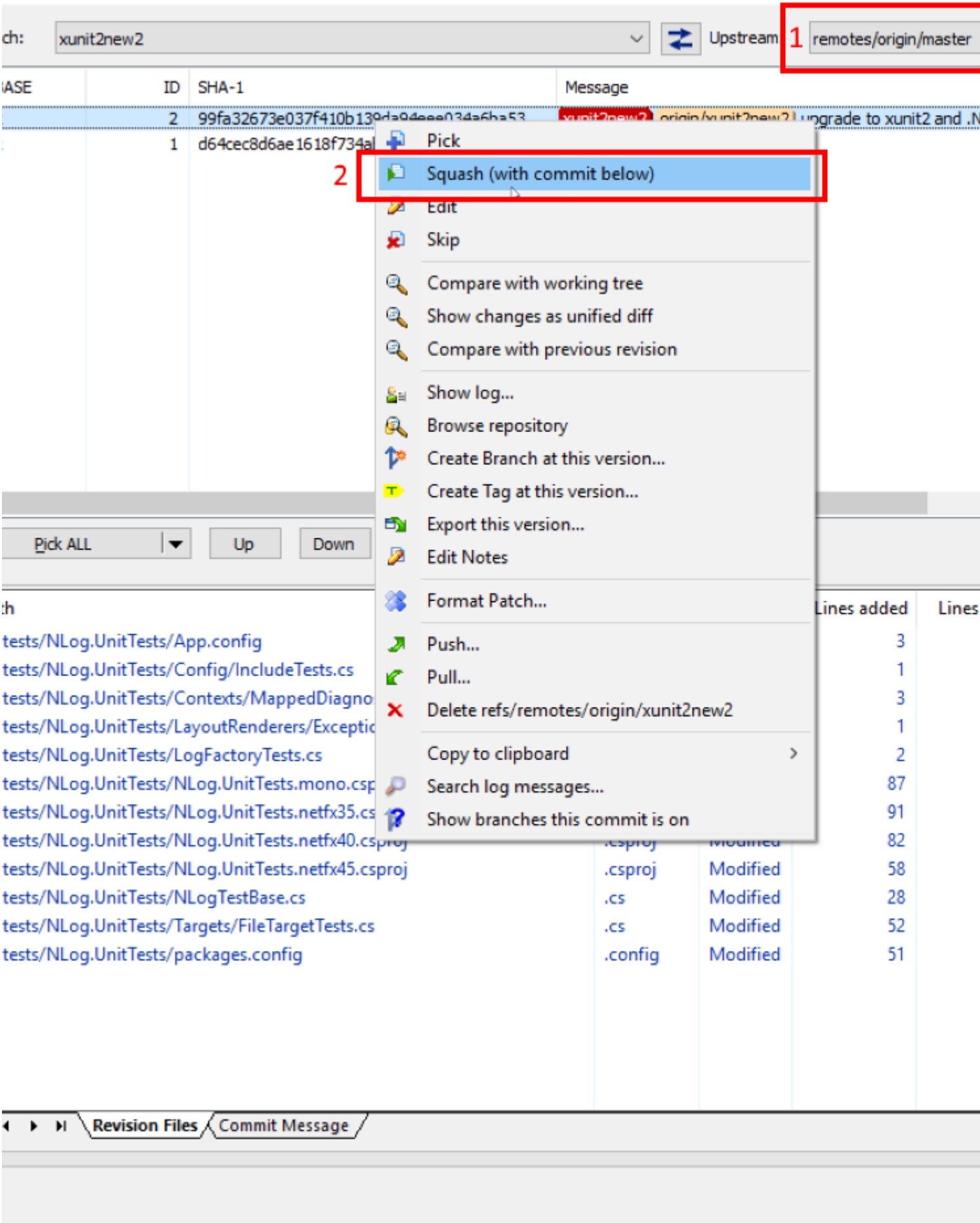
Простой способ

Это не будет работать, если в вашем выборе есть коммиты слияния



Расширенный способ

Запустите диалог восстановления:



Прочитайте TortoiseGit онлайн: https://riptutorial.com/ru/git/topic/5150/tortoisegit

глава 14: Worktrees

Синтаксис

- git worktree add [-f] [--detach] [--checkout] [-b <new-branch>] <path> [<branch>]
- git worktree prune [-n] [-v] [--expire <expire>]
- git worktree list [--porcelain]

параметры

параметр	подробности
-f --force	По умолчанию добавить отказ для создания нового рабочего дерева, когда `<branch>` уже проверено другим рабочим деревом. Этот параметр переопределяет эту гарантию.
-b `<new-branch>` -B `<new-branch>`	С помощью add создайте новую ветку с именем `<new-branch>` начиная с `<branch>` , и запустите `<new-branch>` в новом рабочем дереве. Если `<branch>` опущена, по умолчанию используется `HEAD` . По умолчанию `-b` отказывается создавать новую ветку, если она уже существует. `-B` переопределяет эту защиту, перезагружая `<new-branch>` в `<branch>` .
--detach	С добавлением отделите `HEAD` в новом рабочем дереве.
- [no-] checkout	По умолчанию добавьте проверки `<branch>` , однако, для `--no-checkout` настроек можно использоватьno-checkout, чтобы выполнить настройки, такие как настройка разреженной проверки.
-n -dry-run	С черносливом ничего не удаляйте; просто сообщите, что он удалит.
--фарфор	Со списком выведите в формате простого для разбора скриптов. Этот формат будет оставаться стабильным в версиях Git и независимо от пользовательской конфигурации.
-v --verbose	С черновиком сообщите обо всех удалениях.
--expire `<time>`	С черносливом, только срок действия неиспользуемых рабочих деревьев старше `<time>` .

замечания

Дополнительную информацию см. В официальной документации: https://git-

scm.com/docs/git-worktree .

Examples

Использование рабочей группы

Вы правы в середине работы над новой функцией, и ваш босс требует, чтобы вы что-то исправили. Обычно вам может понадобиться использовать `git stash` для временного хранения ваших изменений. Однако на данный момент ваше рабочее дерево находится в состоянии беспорядка (с новыми, перемещенными и удаленными файлами и другими разбросанными кусками), и вы не хотите нарушать ваш прогресс.

Добавляя рабочую строку, вы создаете временное связанное рабочее дерево, чтобы выполнить аварийное исправление, удалите его, когда закончите, а затем возобновите предыдущий сеанс кодирования:

```
$ git worktree add -b emergency-fix ../temp master
$ pushd ../temp
# ... work work work ...
$ git commit -a -m 'emergency fix for boss'
$ popd
$ rm -rf ../temp
$ git worktree prune
```

ПРИМЕЧАНИЕ. В этом примере исправление все еще находится в ветке аварийного исправления. На этом этапе вы, вероятно, захотите `git merge` или `git format-patch` а затем удалите ветвь аварийного исправления.

Перемещение рабочей группы

В настоящее время (с версии 2.11.0) нет встроенных функций для перемещения уже существующей рабочей станции. Это указано как официальная ошибка (см. Https://git-scm.com/docs/git-worktree#_bugs) .

Чтобы обойти это ограничение, можно выполнять ручные операции непосредственно в файлах ссылок `.git` .

В этом примере главная копия репо находится в `/home/user/project-main` а вторичная рабочая группа находится в `/home/user/project-1` и мы хотим переместить ее в `/home/user/project-2` .

Не выполняйте команду git между этими шагами, иначе сборщик мусора может быть запущен, а ссылки на вторичное дерево могут быть потеряны. Выполните эти шаги с самого начала до конца без перерыва:

1. Измените файл `.git` рабочей группы, чтобы указать на новое местоположение внутри главного дерева. Файл `/home/user/project-1/.git` теперь должен содержать следующее:

```
gitdir: /home/user/project-main/.git/worktrees/project-2
```

2. Переименуйте рабочую папку внутри каталога `.git` основного проекта, перемещая каталог рабочей директории, который существует там:

```
$ mv /home/user/project-main/.git/worktrees/project-1 /home/user/project-
main/.git/worktrees/project-2
```

3. Измените ссылку внутри `/home/user/project-main/.git/worktrees/project-2/gitdir` чтобы указать на новое местоположение. В этом примере файл будет иметь следующее содержимое:

```
/home/user/project-2/.git
```

4. Наконец, переместите свою рабочую папку в новое место:

```
$ mv /home/user/project-1 /home/user/project-2
```

Если вы все сделали правильно, то перечисление существующих рабочих деревьев должно относиться к новому местоположению:

```
$ git worktree list
/home/user/project-main  23f78ad [master]
/home/user/project-2     78ac3f3 [branch-name]
```

Теперь также должно быть безопасно запускать `git worktree prune` .

Прочитайте Worktrees онлайн: https://riptutorial.com/ru/git/topic/3801/worktrees

глава 15: Анализ типов рабочих процессов

замечания

Использование программного обеспечения для управления версиями, такого как Git, может сначала немного страшно, но его интуитивно понятный дизайн, специализирующийся на ветвлении, позволяет сделать несколько различных типов рабочих процессов. Выберите тот, который подходит для вашей собственной команды разработчиков.

Examples

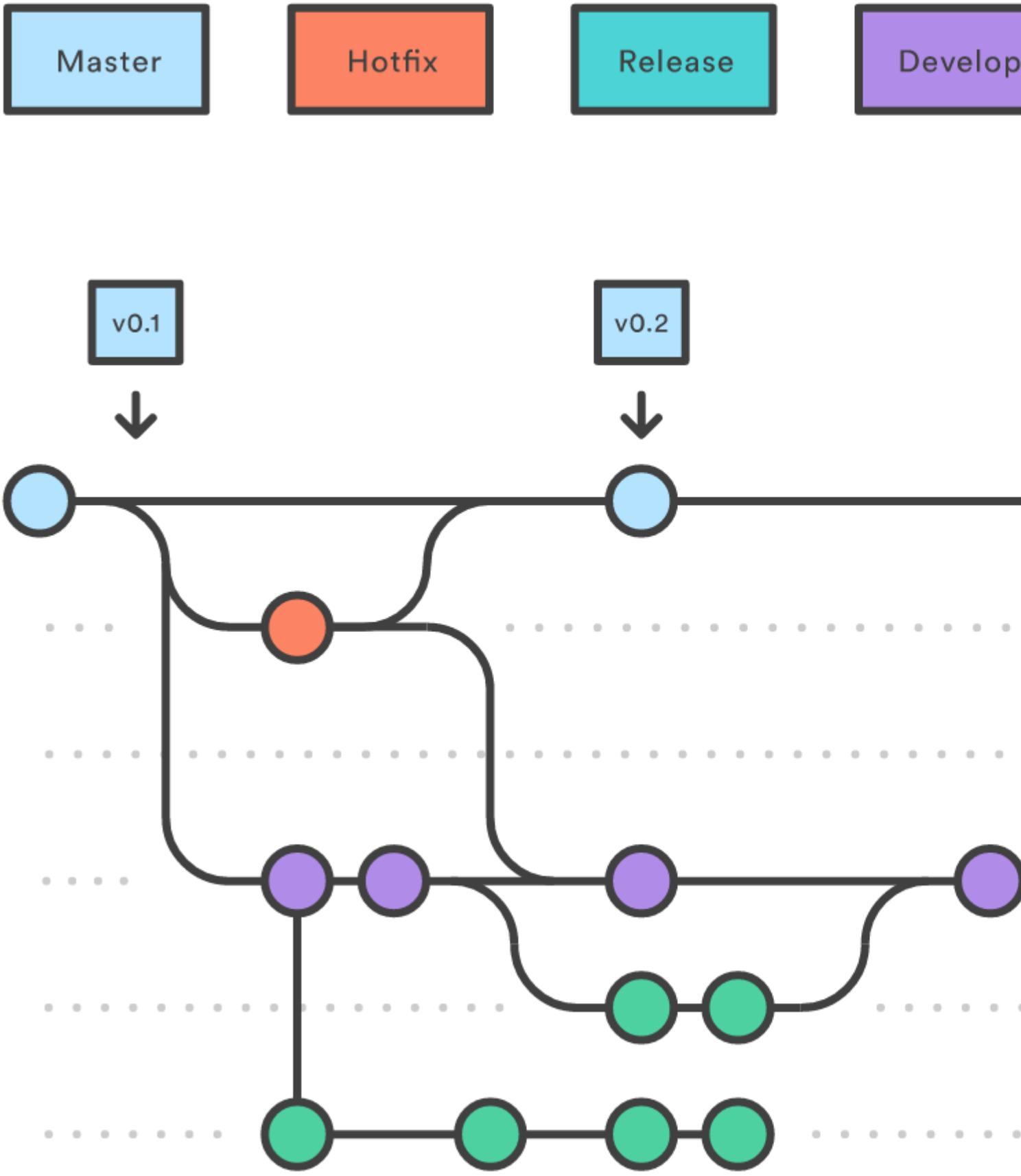
Рабочий процесс Gitflow

Первоначально предложенный Vincent Driessen , Gitflow - это рабочий процесс разработки с использованием git и нескольких заранее определенных ветвей. Это можно рассматривать как частный случай рабочего процесса Feature Feature .

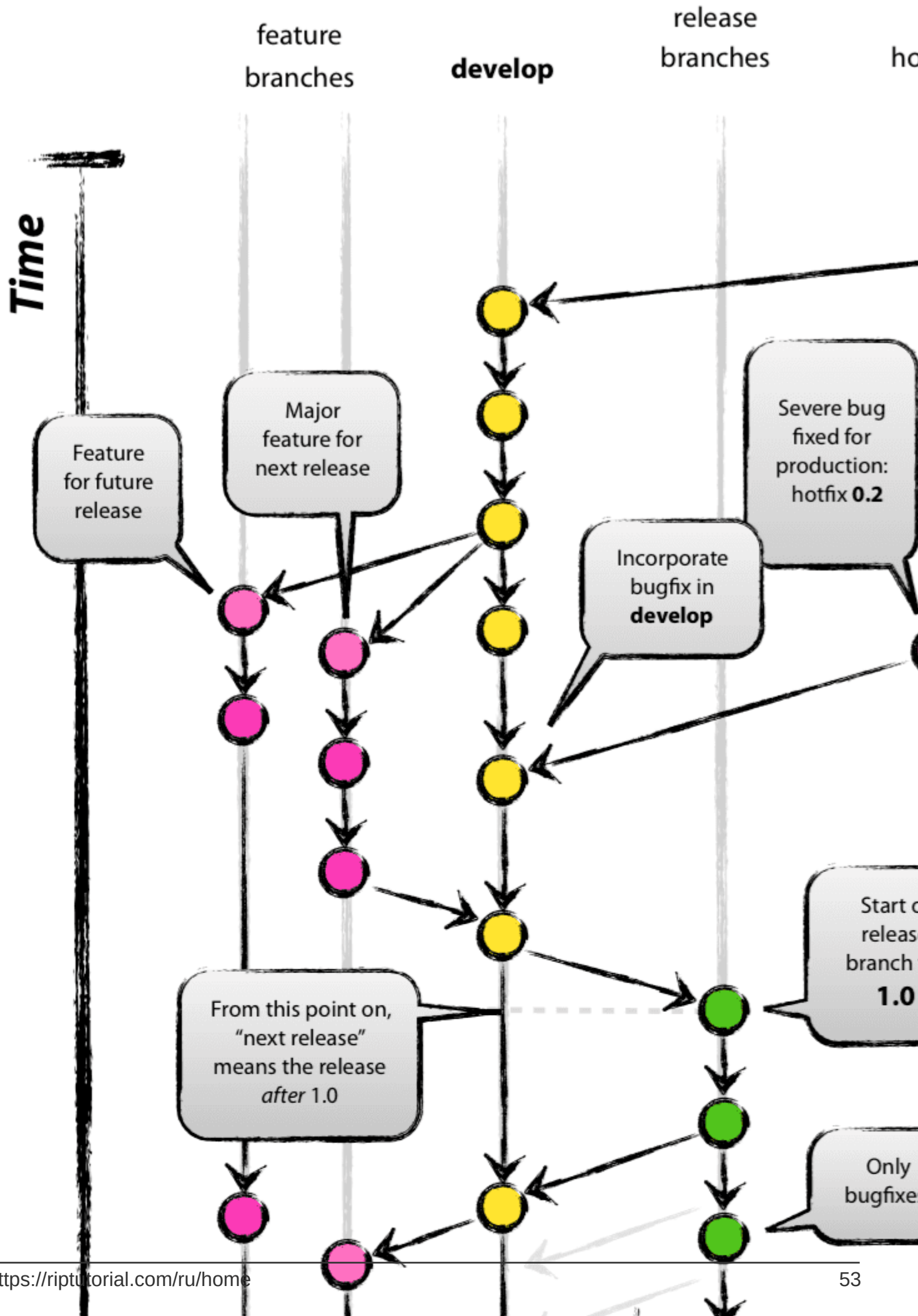
Идея этого состоит в том, чтобы отдельные ветви были зарезервированы для определенных частей в разработке:

- `master` ветвь всегда является самым последним *производственным* кодом. Экспериментальный код здесь не принадлежит.
- `develop` отрасли содержит все последние *разработки* . Эти изменения в области развития могут быть практически любыми, но более крупные функции зарезервированы для их собственных филиалов. Код здесь всегда обрабатывается и объединяется в `release` до выпуска / развертывания.
- ветви `hotfix` для незначительных исправлений ошибок, которые не могут дождаться следующей версии. ветви `hotfix` отрываются от `master` и объединяются обратно в оба `master` и `develop` .
- `release` ветви используются для выпуска новой разработки от `develop` до `master` . Любые изменения в последнюю минуту, такие как набивные номера версий, выполняются в ветви релиза, а затем объединяются обратно в `master` и `develop` . При развертывании новой версии `master` должен быть помечен текущим номером версии (например, с использованием семантического управления версиями) для дальнейшего использования и легкого откат.
- ветви `feature` зарезервированы для больших возможностей. Они специально разработаны в назначенных отраслях и интегрированы с `develop` когда закончены. Выделенные ветви `feature` помогают разделить разработку и возможность развернуть *выполненные* функции независимо друг от друга.

Визуальное представление этой модели:

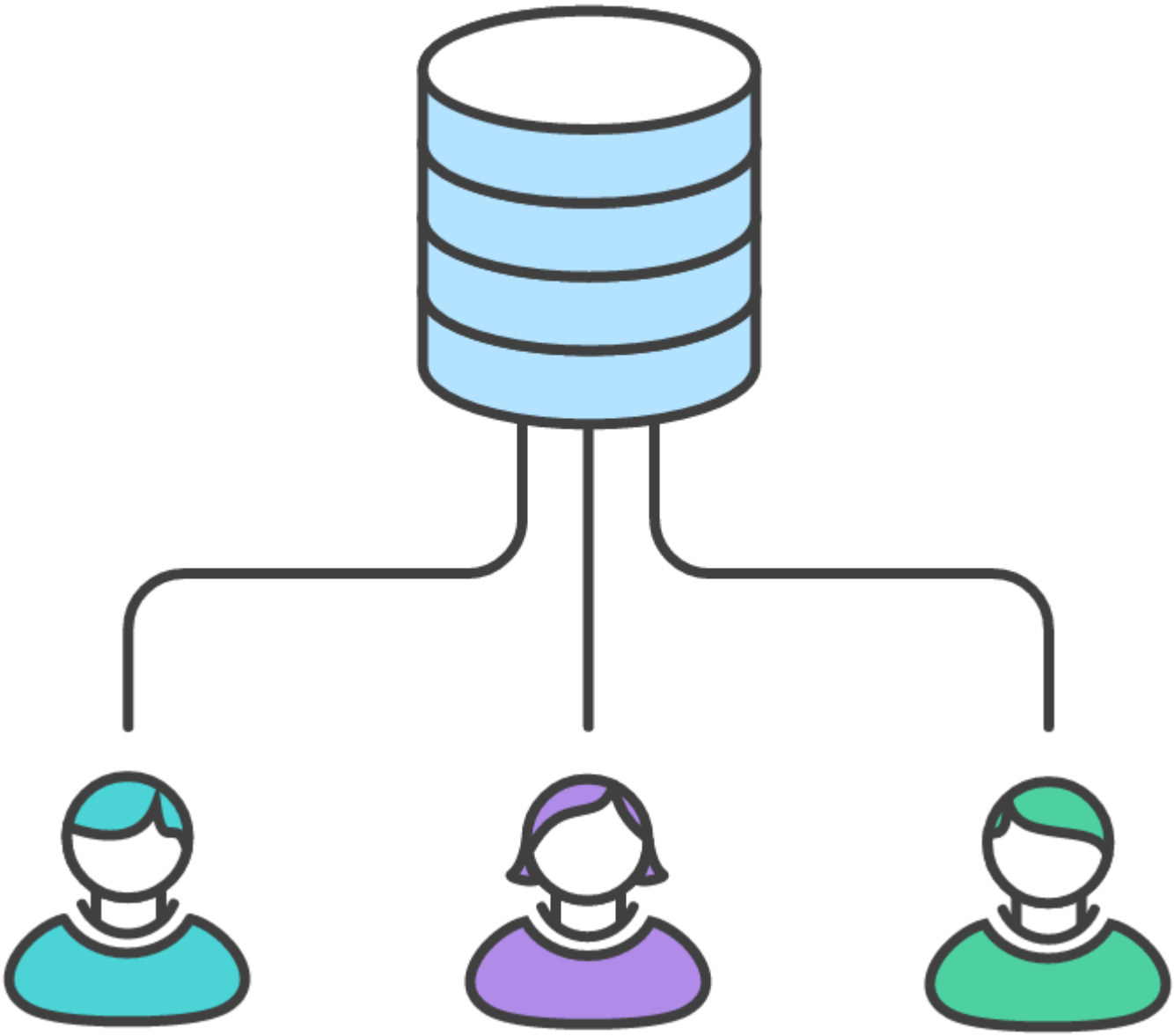


Исходное представление этой модели:


feature branches
develop
release branches
Time
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
From this point on, "next release" means the release after 1.0

ветвь содержит все активные разработки. Авторы должны быть особенно уверены, что они принесут последние изменения до продолжения развития, поскольку эта отрасль будет быстро меняться. Каждый имеет доступ к этому репо и может вносить изменения в главную ветку.

Визуальное представление этой модели:



Это классическая парадигма управления версиями, на которой были созданы более старые системы, такие как Subversion и CVS. Программные средства, которые работают таким образом, называются централизованными системами управления версиями или CVCS. В то время как Git способен работать таким образом, существуют значительные недостатки, такие как необходимость предшествовать каждому притяжению слиянием. Это очень возможно для команды, чтобы работать таким образом, но постоянное слияние разрешения конфликтов может в конечном итоге есть много ценного времени.

Вот почему Линус Торвальдс создал Git не как CVCS, а скорее как *DVCS* или *систему управления распределенной версией* , похожую на Mercurial. Преимущество этого нового

способа работы - гибкость, продемонстрированная в других примерах на этой странице.

Функциональный блок

Основная идея рабочего процесса Feature Branch заключается в том, что все функции разработки должны иметь место в отдельной ветви вместо `master` ветви. Эта инкапсуляция позволяет нескольким разработчикам работать над определенной функцией, не нарушая основную кодовую базу. Это также означает, что `master` ветвь никогда не будет содержать сломанный код, что является огромным преимуществом для непрерывных интеграционных сред.

Инкапсулирующее функционирование также позволяет задействовать запросы на загрузку, которые позволяют инициировать дискуссии вокруг филиала. Они дают другим разработчикам возможность подписаться на функцию до того, как она интегрируется в официальный проект. Или, если вы застряли в середине функции, вы можете открыть запрос на тяну с просьбами о предложениях от ваших коллег. Дело в том, что запросы на pull делают невероятно легким для вашей команды прокомментировать работу друг друга.

основанный на учебниках по Atlassian .

GitHub Flow

Популярно во многих проектах с открытым исходным кодом, но не только.

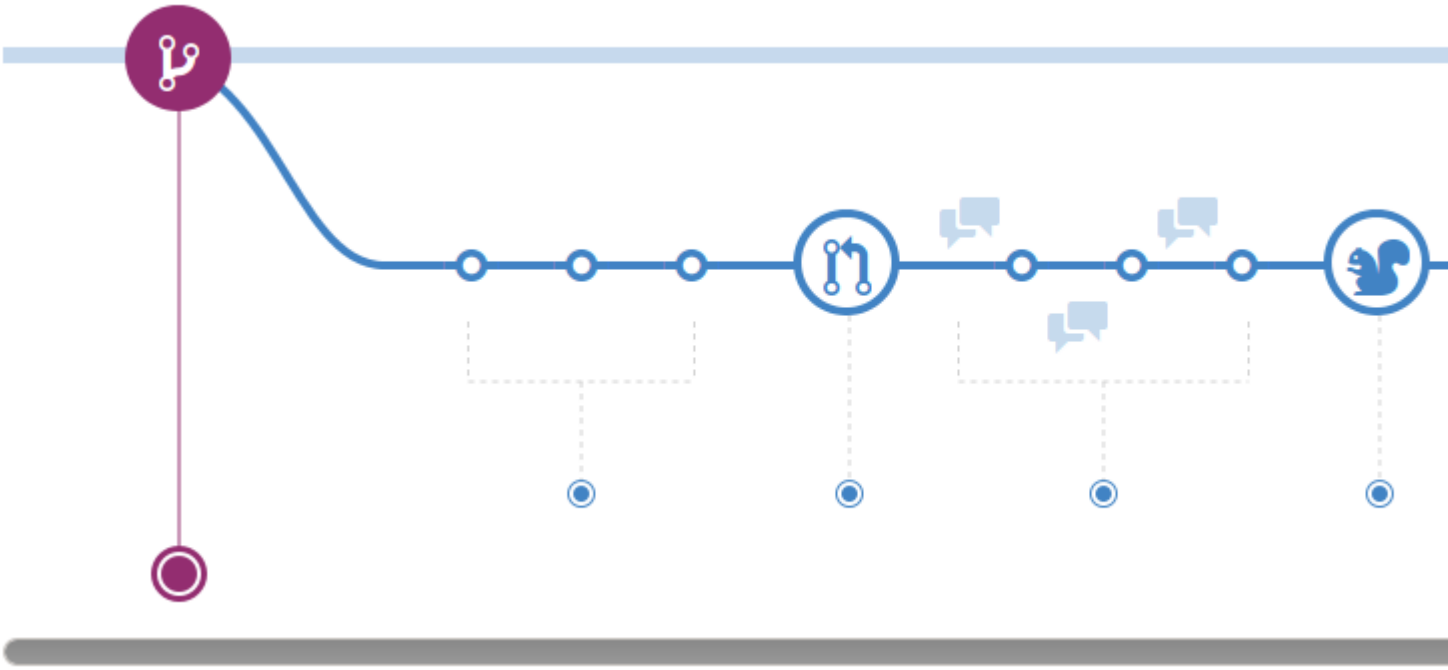
Главный филиал определенного места (Github, Gitlab, Bitbucket, локальный сервер) содержит самую последнюю версию для судоходства. Для каждого нового исправления / исправления ошибок / архитектурного изменения каждый разработчик создает ветку.

Изменения происходят в этой ветви и могут обсуждаться в запросе на вытягивание, просмотре кода и т. Д. После принятия они объединяются в главную ветку.

Полный поток Скотта Чакона:

- Все, что находится в главной ветке, развертывается
- Чтобы работать над чем-то новым, создайте описательно названную ветку мастера (то есть: new-oauth2-scopes)
- Фиксируйте эту ветвь локально и регулярно подталкивайте свою работу к той же именованной ветке на сервере
- Когда вам нужна обратная связь или помощь или вы считаете, что ветка готова к слиянию, откройте запрос на перенос
- После того, как кто-то еще просмотрел и подписал эту функцию, вы можете объединить ее в мастер
- После того, как он будет объединен и нажат на «master», вы можете и должны немедленно развернуть

Первоначально представлен на личном веб-сайте Скотта Чакона .



Изображение предоставлено ссылкой GitHub Flow

Прочитайте Анализ типов рабочих процессов онлайн: https://riptutorial.com/ru/git/topic/1276/анализ-типов-рабочих-процессов

глава 16: Архив

Синтаксис

- git archive [--format = <fmt>] [--list] [--prefix = <префикс> /] [<extra>] [-o <файл> | --output = <файл>] [--worktree-attributes] [--remote = <repo> [--exec = <git-upload-archive>]] <tree-ish> [<путь> ...]

параметры

параметр	подробности
--format = <FMT>	Формат полученного архива: `tar` или `zip` . Если эти параметры не заданы и выходной файл указан, формат выводится из имени файла, если это возможно. В противном случае по умолчанию используется `tar` .
-l, --list	Показать все доступные форматы.
-v, --verbose	Сообщите о прогрессе в stderr.
prefix = <префикс> /	Prepend <prefix> / для каждого имени файла в архиве.
-o <файл>, --output = <файл>	Запишите архив в <файл> вместо stdout.
--worktree-атрибуты	Ищите атрибуты в файлах `.gitattributes` в рабочем дереве.
<Дополнительное>	Это могут быть любые параметры, которые понимает сервер архиватора. Для `zip` бэкенда использование `-0` будет хранить файлы без их дефляции, в то время как от `-1` до `-9` можно использовать для настройки скорости и коэффициента сжатия.
--remote = <репо>	Извлеките архив tar из удаленного репозитория `<repo>` а не из локального репозитория.
--exec = <ГИТ-загрузка-архив>	Используется с `--remote` для указания пути к файлу `<git-upload-archive>` на пульте дистанционного управления.
<Дерево-иш>	Дерево или фиксация для создания архива.
<Путь>	Без необязательного параметра все файлы и каталоги в текущем рабочем каталоге включены в архив. Если указан один или несколько путей, включены только эти.

Examples

Создайте архив репозитория git с префиксом каталога

Считается хорошей практикой использовать префикс при создании git-архивов, так что извлечение помещает все файлы в каталог. Чтобы создать архив `HEAD` с префиксом каталога:

```
git archive --output=archive-HEAD.zip --prefix=src-directory-name HEAD
```

При извлечении все файлы будут извлечены внутри каталога с именем `src-directory-name` в текущем каталоге.

Создайте архив репозитория git на основе конкретной ветки, ревизии, тега или каталога

Также возможно создавать архивы других предметов, кроме `HEAD` , таких как ветки, коммиты, теги и каталоги.

Чтобы создать архив локальной ветви `dev` :

```
git archive --output=archive-dev.zip --prefix=src-directory-name dev
```

Для создания архива источника удаленной ветви `origin/dev` :

```
git archive --output=archive-dev.zip --prefix=src-directory-name origin/dev
```

Чтобы создать архив тега `v.01` :

```
git archive --output=archive-v.01.zip --prefix=src-directory-name v.01
```

Создайте архив файлов внутри определенной подкаталога (`sub-dir`) версии `HEAD` :

```
git archive zip --output=archive-sub-dir.zip --prefix=src-directory-name HEAD:sub-dir/
```

Создание архива репозитория git

С `git archive` можно создавать сжатые архивы репозитория, например, для распространения релизов.

Создайте tar-архив текущей версии `HEAD` :

```
git archive --format tar HEAD | cat > archive-HEAD.tar
```

Создайте tar-архив текущей версии `HEAD` с сжатием gzip:

```
git archive --format tar HEAD | gzip > archive-HEAD.tar.gz
```

Это также можно сделать с помощью (который будет использовать встроенную обработку tar.gz):

```
git archive --format tar.gz HEAD > archive-HEAD.tar.gz
```

Создайте zip-архив текущей версии `HEAD` :

```
git archive --format zip HEAD > archive-HEAD.zip
```

В качестве альтернативы можно просто указать выходной файл с допустимым расширением, и из него будет выведен формат и тип сжатия:

```
git archive --output=archive-HEAD.tar.gz HEAD
```

Прочитайте Архив онлайн: https://riptutorial.com/ru/git/topic/2815/архив

глава 17: Биссекция / поиск ошибочных обязательств

Синтаксис

- `git bisect <subcommand> <options>`
- `git bisect start <bad> [<good>...]`
- `git bisect reset`
- `git bisect good`
- `git bisect bad`

Examples

Бинарный поиск (git bisect)

`git bisect` позволяет вам найти, какая фиксация ввела ошибку, используя двоичный поиск.

Начните с дебазации сеанса путем предоставления двух ссылок на фиксацию: хорошая фиксация перед ошибкой и неудачная фиксация после ошибки. Как правило, плохой фиксацией является `HEAD` .

```
# start the git bisect session
$ git bisect start

# give a commit where the bug doesn't exist
$ git bisect good 49c747d

# give a commit where the bug exist
$ git bisect bad HEAD
```

`git` запускает двоичный поиск: он разделяет ревизию пополам и переключает репозиторий на промежуточную ревизию. Проверьте код, чтобы определить, хороша или плохая ревизия:

```
# tell git the revision is good,
# which means it doesn't contain the bug
$ git bisect good

# if the revision contains the bug,
# then tell git it's bad
$ git bisect bad
```

`git` продолжит выполнение двоичного поиска по каждому оставшемуся подмножеству

плохих версий в зависимости от ваших инструкций. `git` представит единственную ревизию, которая, если бы ваши флаги не были неправильными, будет представлять собой именно ту версию, где была введена ошибка.

Впоследствии помните, чтобы запустить `git bisect reset` чтобы завершить сеанс bisect и вернуться в HEAD.

```
$ git bisect reset
```

Если у вас есть скрипт, который может проверить наличие ошибки, вы можете автоматизировать процесс с помощью:

```
$ git bisect run [script] [arguments]
```

Где `[script]` - это путь к вашему скрипту, а `[arguments]` - это любые аргументы, которые должны быть переданы вашему скрипту.

Выполнение этой команды будет автоматически выполняться через двоичный поиск, выполняя `git bisect good` или `git bisect bad` на каждом шаге в зависимости от кода выхода вашего скрипта. Выход с 0 указывает на `good` , а выход с 1-124, 126 или 127 указывает на плохое. 125 указывает, что сценарий не может протестировать эту ревизию (которая приведет к `git bisect skip`).

Полуавтоматически найдите ошибочную фиксацию

Представьте, что вы находитесь на `master` ветке, и что-то не работает должным образом (регрессия была введена), но вы не знаете, где. Все, что вы знаете, это то, что работало в последнем выпуске (например, с тегом или вы знаете хеш-код фиксации, давайте возьмем `old-rel` здесь).

Git помогает вам, обнаружив ошибочную фиксацию, которая вводила регрессию с очень небольшим количеством шагов (двоичный поиск).

Прежде всего начните деление пополам:

```
git bisect start master old-rel
```

Это скажет git, что `master` - это сломанная ревизия (или первая сломанная версия), а `old-rel` - последняя известная версия.

Теперь Git проверит отдельную голову в середине обоих коммитов. Теперь вы можете провести тестирование. В зависимости от того, работает ли это или нет

```
git bisect good
```

или же

```
git bisect bad
```

, В случае, если эта фиксация не может быть протестирована, вы можете легко `git reset` и протестировать ее, git willl позаботится об этом.

После нескольких шагов git выведет ошибочный хеш фиксации.

Чтобы прервать процесс bisect, просто выполните

```
git bisect reset
```

и git восстановит предыдущее состояние.

Прочитайте Биссекция / поиск ошибочных обязательств онлайн:
https://riptutorial.com/ru/git/topic/3645/биссекция---поиск-ошибочных-обязательств

глава 18: Внешние слияния и диффицирования

Examples

Настройка Beyond Compare

Вы можете установить путь к `bcomp.exe`

```
git config --global difftool.bc3.path 'c:\Program Files (x86)\Beyond Compare 3\bcomp.exe'
```

и настройте `bc3` по умолчанию

```
git config --global diff.tool bc3
```

Настройка KDiff3 в качестве инструмента слияния

В ваш глобальный файл `.gitconfig` следует добавить `.gitconfig`

```
[merge]
    tool = kdiff3
[mergetool "kdiff3"]
    path = D:/Program Files (x86)/KDiff3/kdiff3.exe
    keepBackup = false
    keepbackup = false
    trustExitCode = false
```

Не забудьте установить свойство `path` для указания на каталог, в котором вы установили KDiff3

Настройка KDiff3 как инструмента сравнения

```
[diff]
    tool = kdiff3
    guitool = kdiff3
[difftool "kdiff3"]
    path = D:/Program Files (x86)/KDiff3/kdiff3.exe
    cmd = \"D:/Program Files (x86)/KDiff3/kdiff3.exe\" \"$LOCAL\" \"$REMOTE\"
```

Настройка IntelliJ IDE в качестве инструмента слияния (Windows)

```
[merge]
    tool = intellij
[mergetool "intellij"]
    cmd = cmd \"/C D:\\workspace\\tools\\symlink\\idea\\bin\\idea.bat merge $(cd $(dirname
```

```
"$LOCAL") && pwd)/$(basename "$LOCAL") $(cd $(dirname "$REMOTE") && pwd)/$(basename "$REMOTE")
$(cd $(dirname "$BASE") && pwd)/$(basename "$BASE") $(cd $(dirname "$MERGED") &&
pwd)/$(basename "$MERGED")\"
    keepBackup = false
    keepbackup = false
    trustExitCode = true
```

Тот, который получил здесь, заключается в том, что это свойство `cmd` не принимает никаких странных символов в пути. Если в вашем месте установки IDE есть странные символы (например, он установлен в `Program Files (x86)` , вам нужно создать символическую ссылку

Настройка IntelliJ IDE в качестве инструмента сравнения (Windows)

```
[diff]
    tool = intellij
    guitool = intellij
[difftool "intellij"]
    path = D:/Program Files (x86)/JetBrains/IntelliJ IDEA 2016.2/bin/idea.bat
    cmd = cmd \"/C D:\\workspace\\tools\\symlink\\idea\\bin\\idea.bat diff $(cd $(dirname
"$LOCAL") && pwd)/$(basename "$LOCAL") $(cd $(dirname "$REMOTE") && pwd)/$(basename
"$REMOTE")\"
```

Тот, который получил здесь, заключается в том, что это свойство `cmd` не принимает никаких странных символов в пути. Если в вашем месте установки IDE есть странные символы (например, он установлен в `Program Files (x86)` , вам нужно создать символическую ссылку

Прочитайте Внешние слияния и диффицирования онлайн:
https://riptutorial.com/ru/git/topic/5972/внешние-слияния-и-диффицирования

глава 19: Внутренности

Examples

Repo

`git repository` представляет собой структуру данных на диске, в которой хранятся метаданные для набора файлов и каталогов.

Он живет в папке `.git/` project вашего проекта. Каждый раз, когда вы передаете данные git, он хранится здесь. И наоборот, `.git/` содержит все отдельные фиксации.

Это базовая структура:

```
.git/
    objects/
    refs/
```

Объекты

`git` в основном является хранилищем ключей. Когда вы добавляете данные в `git` , он создает `object` и использует хэш SHA-1 содержимого `object` в качестве ключа.

Поэтому любой контент в `git` может быть просмотрен с помощью хэша:

```
git cat-file -p 4bb6f98
```

Существует 4 типа `Object` :

- `blob`
- `tree`
- `commit`
- `tag`

HEAD ref

`HEAD` - специальная `ref` . Он всегда указывает на текущий объект.

Вы можете видеть, где он сейчас указывает, проверяя файл `.git/HEAD` .

Обычно `HEAD` указывает на другой `ref` :

```
$cat .git/HEAD
ref: refs/heads/mainline
```

Но он также может указывать прямо на `object` :

```
$ cat .git/HEAD
4bb6f98a223abc9345a0cef9200562333
```

Это то, что известно как «отсоединенная голова» - потому что `HEAD` не привязан к (указывает на) какой-либо `ref` , а скорее указывает непосредственно на `object` .

Refs

`ref` является по существу указателем. Это имя указывает на `object` . Например,

```
"master" --> 1a410e...
```

Они хранятся в файлах .git / refs / heads / в текстовых файлах.

```
$ cat .git/refs/heads/mainline
4bb6f98a223abc9345a0cef9200562333
```

Обычно это называется `branches` . Тем не менее, вы заметите, что в `git` нет такой вещи, как `branch` - только `ref` .

Теперь можно ориентироваться в `git` исключительно, перепрыгивая на разные `objects` непосредственно своими хэшами. Но это было бы ужасно неудобно. `ref` дает вам удобное имя для ссылки на `objects` . Гораздо проще попросить `git` перейти к определенному месту по имени, а не к хэшу.

Объект фиксации

`commit` , вероятно, `object` типа наиболее знакомый `git` пользователей, так как это то , что они используются для создания с `git commit` команд.

Однако `commit` не содержит непосредственно никаких измененных файлов или данных. Скорее, он содержит в основном метаданные и указатели на другие `objects` которые содержат фактическое содержимое `commit` .

`commit` содержит несколько вещей:

- хэш `tree`
- хэш родительской `commit`
- имя автора / адрес электронной почты, commiter name / email
- сообщение фиксации

Вы можете увидеть содержимое любого фиксации следующим образом:

```
$ git cat-file commit 5bac93
tree 04d1daef...
parent b7850ef5...
author Geddy Lee <glee@rush.com>
```

```
commiter Neil Peart <npeart@rush.com>

First commit!
```

дерево

Очень важно отметить, что `tree` объекты хранят в вашем проекте КАЖДЫЙ файл, и он хранит целые файлы, которые не отличаются друг от друга. Это означает, что каждая `commit` содержит моментальный снимок всего проекта *.

* *Технически сохраняются только измененные файлы. Но это более подробная информация об эффективности. С точки зрения дизайна, `commit` должна рассматриваться как содержащая полную копию проекта .*

родитель

`parent` строка содержит хэш другого объекта `commit` и может считаться «родительским указателем», который указывает на «предыдущий фиксатор». Это неявно образует граф **коммитов,** известный как **граф фиксации** . В частности, это ориентированный ациклический граф (или DAG).

Объект дерева

`tree` основном представляет собой папку в традиционной файловой системе: вложенные контейнеры для файлов или других папок.

`tree` содержит:

- 0 или более объектов `blob`
- 0 или более объектов `tree`

Так же, как вы можете использовать `ls` или `dir` для отображения содержимого папки, вы можете перечислить содержимое `tree` объекта.

```
$ git cat-file -p 07b1a631
100644 blob b91bba1b   .gitignore
100644 blob cc0956f1   Makefile
040000 tree 92e1ca7e   src
...
```

Вы можете искать файлы в `commit` , сначала обнаруживая хэш `tree` в `commit` , а затем глядя на это `tree` :

```
$ git cat-file commit 4bb6f93a
tree 07b1a631
```

```
parent ...
author ...
commiter ...

$ git cat-file -p 07b1a631
100644 blob b91bba1b   .gitignore
100644 blob cc0956f1   Makefile
040000 tree 92e1ca7e   src
...
```

Объект Blob

`blob` содержит произвольное содержимое двоичного файла. Как правило, это будет сырой текст, такой как исходный код или статья в блоге. Но так же легко могли быть байты PNG-файла или чего-то еще.

Если у вас есть хеш `blob` , вы можете посмотреть его содержимое.

```
$ git cat-file -p d429810
package com.example.project

class Foo {
 ...
}
...
```

Например, вы можете просматривать `tree` как указано выше, а затем посмотреть на один из `blobs` в нем.

```
$ git cat-file -p 07b1a631
100644 blob b91bba1b   .gitignore
100644 blob cc0956f1   Makefile
040000 tree 92e1ca7e   src
100644 blob cae391ff   Readme.txt

$ git cat-file -p cae391ff
Welcome to my project! This is the readmefile
...
```

Создание новых коммитов

Команда `git commit` выполняет несколько действий:

1. Создавайте `blobs` и `trees` чтобы представлять каталог проекта - хранится в `.git/objects`
2. Создает новый объект `commit` с информацией об авторе, сообщением фиксации и корневым `tree` с шага 1 - также сохраняется в `.git/objects`
3. Обновляет `HEAD` ref в `.git/HEAD` для хеша вновь созданного `commit`

Это приведет к добавлению нового снимка вашего проекта в `git` который связан с предыдущим состоянием.

Перемещение HEAD

Когда вы запускаете `git checkout` при фиксации (заданной хешем или ref), вы говорите `git` чтобы сделать ваш рабочий каталог похожим на то, как это было сделано при создании моментального снимка.

1. Обновите файлы в рабочем каталоге, чтобы они соответствовали `tree` внутри `commit`
2. Обновить `HEAD` чтобы указать на указанный хэш или ref

Перемещение ссылок

Запуск `git reset --hard` перемещает refs в указанный хеш / ref.

Перемещение `MyBranch` на `b8dc53` :

```
$ git checkout MyBranch      # moves HEAD to MyBranch
$ git reset --hard b8dc53    # makes MyBranch point to b8dc53
```

Создание новых Refs

Запуск `git checkout -b <refname>` создаст новый ref, который указывает на текущую `commit` .

```
$ cat .git/head
1f324a

$ git checkout -b TestBranch

$ cat .git/refs/heads/TestBranch
1f324a
```

Прочитайте Внутренности онлайн: https://riptutorial.com/ru/git/topic/2637/внутренности

глава 20: Восстановление

Examples

Восстановление из потерянной фиксации

В случае, если вы вернулись назад к прошлому коммиту и потеряли новое коммитирование, вы можете восстановить потерянный коммит, выполнив

```
git reflog
```

Затем найдите свой потерянный коммит и верните его обратно, сделав

```
git reset HEAD --hard <sha1-of-commit>
```

Восстановить удаленный файл после фиксации

Если вы случайно совершили удаление файла, а затем поняли, что вам это нужно.

Сначала найдите идентификатор фиксации, который удалил ваш файл.

```
git log --diff-filter=D --summary
```

Дает вам отсортированное резюме коммитов, которые удалили файлы.

Затем перейдите к восстановлению файла на

```
git checkout 81eeccf~1 <your-lost-file-name>
```

(Замените 81eeccf на свой собственный код фиксации)

Восстановить файл до предыдущей версии

Чтобы восстановить файл в предыдущей версии, вы можете использовать `reset` .

```
git reset <sha1-of-commit> <file-name>
```

Если вы уже внесли локальные изменения в файл (который вам не нужен!), Вы также можете использовать опцию `--hard`

Восстановление удаленной ветви

Чтобы восстановить удаленную ветку, вам нужно найти фиксацию, которая была главой

удаленной ветви, запустив

```
git reflog
```

Затем вы можете воссоздать ветвь, запустив

```
git checkout -b <branch-name> <sha1-of-commit>
```

Вы не сможете восстановить удаленные ветки, если git's garbage collector удаляет оборванные коммиты - те, у кого нет ссылок. Всегда сохраняйте резервную копию своего репозитория, особенно когда вы работаете в небольшом командном / проприетарном проекте

Восстановление после сброса

С Git вы можете (почти) всегда поворачивать часы назад

Не бойтесь экспериментировать с командами, которые переписывают историю *. Git не удаляет ваши фиксации в течение 90 дней по умолчанию, и за это время вы можете легко восстановить их из reflog:

```
$ git reset @~3   # go back 3 commits
$ git reflog
c4f708b HEAD@{0}: reset: moving to @~3
2c52489 HEAD@{1}: commit: more changes
4a5246d HEAD@{2}: commit: make important changes
e8571e4 HEAD@{3}: commit: make some changes
... earlier commits ...
$ git reset 2c52489
... and you're back where you started
```

* *Следите за варианты* , *как* `--hard` *и* `--force` , *хотя - они могут отказаться* от *данных.*
* *Также избегайте переписывать историю в любых филиалах, с которыми вы работаете.*

Восстановление от git stash

Чтобы получить последний тайник после запуска git stash, используйте

```
git stash apply
```

Чтобы просмотреть список ваших кошельков, используйте

```
git stash list
```

Вы получите список, который выглядит примерно так:

```
stash@{0}: WIP on master: 67a4e01 Merge tests into develop
stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Выберите другой git stash для восстановления с номером, который отображается для кошелька, который вы хотите

```
git stash apply stash@{2}
```

Вы также можете выбрать «git stash pop», он работает так же, как «git stash apply», как ..

```
 git stash pop
```

или же

```
 git stash pop stash@{2}
```

Разница в git stash применяется и git stash pop ...

git stash pop : - данные stash будут удалены из стека списка закладок.

Пример: -

```
git stash list
```

Вы получите список, который выглядит примерно так:

```
stash@{0}: WIP on master: 67a4e01 Merge tests into develop
stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Теперь данные поп-штифта с использованием команды

```
git stash pop
```

Снова проверьте список закладок

```
git stash list
```

Вы получите список, который выглядит примерно так:

```
 stash@{0}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Вы можете увидеть, что один файл stash удаляется (выталкивается) из списка закладок, а stash @ {1} становится stash @ {0}.

Прочитайте Восстановление онлайн: https://riptutorial.com/ru/git/topic/725/восстановление

глава 21: Выбор вишни

Вступление

Вишневый подборщик берет патч, который был введен в фиксацию, и пытается повторно применить его в филиале, в котором вы сейчас находитесь.

Источник: Git SCM Book

Синтаксис

- git cherry-pick [--edit] [-n] [-m parent-number] [-s] [-x] [--ff] [-S [key-id]] commit ...
- git cherry-pick - продолжение
- git cherry-pick --quit
- git cherry-pick --bort

параметры

параметры	подробности
-e, --edit	С помощью этой опции `git cherry-pick` позволит вам отредактировать сообщение фиксации до фиксации.
-Икс	При записи фиксации добавьте строку, в которой говорится: «(вишня выбрана из фиксации ...)» в исходное сообщение фиксации, чтобы указать, какое из этих изменений выбрано вишневым. Это делается только для выбора вишни без конфликтов.
--ff	Если текущий HEAD совпадает с родительским элементом фиксации вишни, тогда будет выполнена быстрая перемотка вперед к этой фиксации.
--Продолжить	Продолжайте работу, используя информацию в .git / sequencer. Может использоваться для продолжения после разрешения конфликтов при неудачном наборе вишни или возврате.
--уволиться	Забудьте о текущей текущей операции. Может использоваться для очистки состояния секвенсора после неудачного набора вишни или возврата.
--abort	Отмените операцию и верните ее в состояние предварительной последовательности.

Examples

Копирование фиксации из одной ветки в другую

`git cherry-pick <commit-hash>` будет применять изменения, сделанные в существующей фиксации к другой ветке, при записи нового фиксации. По существу, вы можете скопировать фиксации из ветки в ветвь.

Учитывая следующее дерево (Источник)

```
dd2e86 - 946992 - 9143a9 - a6fd86 - 5a6057 [master]
           \
            76cada - 62ecb3 - b886a0 [feature]
```

Предположим, мы хотим скопировать `b886a0` на мастер (сверху `5a6057`).

Мы можем запускать

```
git checkout master
git cherry-pick b886a0
```

Теперь наше дерево будет выглядеть примерно так:

```
dd2e86 - 946992 - 9143a9 - a6fd86 - 5a6057 - a66b23 [master]
           \
            76cada - 62ecb3 - b886a0 [feature]
```

Если новый commit `a66b23` имеет одинаковый контент (источник diff, сообщение фиксации) как `b886a0` (но другой родительский элемент). Обратите внимание, что выбор вишни будет только забирать изменения в этой фиксации (`b886a0` в этом случае) не все изменения в ветке функций (для этого вам придется либо использовать перезагрузку, либо слияние).

Копирование диапазона транзакций из одной ветви в другую

`git cherry-pick <commit-A>..<commit-B>` поместит каждую фиксацию *после* A и до `git cherry-pick <commit-A>..<commit-B>` включит B поверх текущей отмеченной ветви.

`git cherry-pick <commit-A>^..<commit-B>` поместит фиксацию A и каждую фиксацию до и включительно B поверх текущей отмеченной ветви.

Проверка наличия вишни

Прежде чем вы начнете процесс вишневого выбора, вы можете проверить, существует ли фиксация, которую вы хотите использовать для зависания, в целевой ветке, и в этом случае вам не нужно ничего делать.

`git branch --contains <commit>` отображает локальные ветви, содержащие указанную фиксацию.

`git branch -r --contains <commit>` также включает в себя удаленные ветви отслеживания в списке.

Найти коммиты, которые еще не применяются для восходящего потока

Команда `git cherry` показывает изменения, которые еще не выбраны вишней.

Пример:

```
git checkout master
git cherry development
```

... и см. вывод немного так:

```
+ 492508acab7b454eee8b805f8ba906056eede0ff
- 5ceb5a9077ddb9e78b1e8f24bfc70e674c627949
+ b4459544c000f4d51d1ec23f279d9cdb19c1d32b
+ b6ce3b78e938644a293b2dd2a15b2fecb1b54cd9
```

Конец, что с `+` будет теми, кто еще не вишневый в `development` .

Синтаксис:

```
git cherry [-v] [<upstream> [<head> [<limit>]]]
```

Опции:

-v Показывать объекты фиксации рядом с SHA1.

<вверх по течению> Отправление вверх для поиска эквивалентных коммитов. По умолчанию используется дочерняя ветвь HEAD.

<head> Рабочая ветвь; по умолчанию - HEAD.

<limit> Не сообщать о фиксации до (и включая) лимита.

Ознакомьтесь с документацией на git-cherry для получения дополнительной информации.

Прочитайте Выбор вишни онлайн: https://riptutorial.com/ru/git/topic/672/выбор-вишни

глава 22: ГИТ-SVN

замечания

Клонирование действительно больших хранилищ SVN

Если история SVN-репо действительно действительно большая, эта операция может занять несколько часов, так как git-svn нуждается в восстановлении полной истории репо SVN. К счастью, вам нужно только клонировать SVN repo один раз; как и в любом другом репозитории git, вы можете просто скопировать папку репо другим сотрудникам. Копирование папки на несколько компьютеров будет быстрее, чем просто клонирование больших SVN-репозитов с нуля.

О коммиттах и SHA1

Ваши местные git-коммиты будут *переписаны* при использовании команды `git svn dcommit` . Эта команда добавит текст в сообщение git commit, ссылающееся на версию SVN, созданную на сервере SVN, что очень полезно. Однако добавление нового текста требует изменения существующего сообщения фиксации, которое фактически не может быть выполнено: git commits не поддаются обработке. Решением является создание нового коммита с тем же содержимым и новым сообщением, но в любом случае это технически новая фиксация (т. Е. SHA1 команды git commit)

Поскольку git-фиксации, созданные для git-svn, являются локальными, идентификаторы SHA1 для git-коммитов отличаются между каждым репозиторием git! Это означает, что вы не можете использовать SHA1 для ссылки на фиксацию от другого лица, потому что тот же фиксат будет иметь различный SHA1 в каждом локальном репозитории git. Вы должны полагаться на номер версии svn, добавленный к сообщению о фиксации, когда вы нажимаете на сервер SVN, если вы хотите ссылаться на фиксацию между различными копиями репозитория.

Вы можете использовать SHA1 для локальных операций, хотя (показать / отличить конкретную фиксацию, выбор вишней и сброс и т. Д.),

Поиск проблемы

Команда git svn rebase выдает ошибку несоответствия контрольной суммы

Команда git svn rebase выдает ошибку, подобную этой:

```
Checksum mismatch: <path_to_file> <some_kind_of_sha1>
expected: <checksum_number_1>
  got: <checksum_number_2>
```

Решение этой проблемы сбрасывается svn на ревизию, когда поврежденный файл был изменен в последний раз, и выполните git svn fetch, чтобы восстановить историю SVN. Команды для выполнения сброса SVN:

- git log -1 - `<path_to_file>` (скопируйте номер версии SVN, который появляется в сообщении фиксации)
- git svn reset `<revision_number>`
- git svn fetch

Вы сможете снова и снова извлекать данные из SVN

Файл не найден в commit Когда вы пытаетесь извлечь или вытащить из SVN, вы получите ошибку, подобную этой

```
<file_path> was not found in commit <hash>
```

Это означает, что ревизия в SVN пытается изменить файл, который по какой-то причине не существует в вашей локальной копии. Лучший способ избавиться от этой ошибки - заставить fetch игнорировать путь к этому файлу, и он будет обновлен до статуса в последней версии SVN:

- `git svn fetch --ignore-paths <file_path>`

Examples

Клонирование хранилища SVN

Вам необходимо создать новую локальную копию репозитория с помощью команды

```
git svn clone SVN_REPO_ROOT_URL [DEST_FOLDER_PATH] -T TRUNK_REPO_PATH -t TAGS_REPO_PATH -b
BRANCHES_REPO_PATH
```

Если ваш SVN-репозиторий соответствует стандартным макетам (соединительные линии, ветви, теги), вы можете сохранить некоторые типизации:

```
git svn clone -s SVN_REPO_ROOT_URL [DEST_FOLDER_PATH]
```

`git svn clone` проверяет каждую ревизию SVN один за другим и делает git-фиксацию в вашем локальном репозитории, чтобы воссоздать историю. Если репозиторий SVN имеет много коммитов, это займет некоторое время.

Когда команда будет завершена, у вас будет полноценный git-репозиторий с локальной ветвью, называемой master, которая отслеживает ветвь соединительной линии в репозитории SVN.

Получение последних изменений из SVN

Эквивалентом `git pull` является команда

```
git svn rebase
```

Это извлекает все изменения из репозитория SVN и применяет их *поверх* ваших локальных коммитов в вашей текущей ветке.

Вы также можете использовать команду

```
git svn fetch
```

для извлечения изменений из репозитория SVN и переноса их на локальную машину, но без применения их к локальной ветке.

Нажатие локальных изменений в SVN

Команда

```
git svn dcommit
```

создаст ревизию SVN для каждого из ваших локальных git-коммитов. Как и в случае с SVN, ваша локальная история git должна быть синхронизирована с последними изменениями в репозитории SVN, поэтому, если команда не работает, попробуйте выполнить сначала `git svn rebase` .

Работа на местном уровне

Просто используйте локальный репозиторий git как обычное git-репо с обычными командами git:

- `git add FILE` и `git checkout -- FILE` Чтобы создать / отключить файл
- `git commit` Чтобы сохранить изменения. Эти коммиты будут локальными и не будут «толкаться» к репо SVN, как в обычном репозитории git
- `git stash` и `git stash pop` Позволяет использовать приступы
- `git reset HEAD --hard` все ваши локальные изменения
- `git log` Доступ ко всей истории в репозитории
- `git rebase -i` чтобы вы могли свободно переписывать свою местную историю
- `git branch` и `git checkout` для создания локальных ветвей

Как указано в документации git-svn, «Subversion - это система, которая намного менее сложна, чем Git», поэтому вы не можете использовать всю полноту git, не испортив историю на сервере Subversion. К счастью, правила очень просты: **держите историю линейной**

Это означает, что вы можете сделать почти любую операцию git: создание ветвей,

удаление / переупорядочение / раздачу коммитов, перемещение истории вокруг, удаление коммитов и т. Д. Все, *кроме слияния* . Если вам нужно реинтегрировать историю локальных ветвей, вместо этого используйте `git rebase` .

Когда вы выполняете слияние, создается слияние. Особая вещь о слиянии заключается в том, что у них есть два родителя, и это делает историю нелинейной. Нелинейная история будет путать SVN в том случае, если вы «нажмете» фиксацию слияния в репозиторий.

Однако не беспокойтесь: **вы ничего не сломаете, если вы «нажмете» фиксацию слияния git на SVN** . Если вы это сделаете, когда коммит git merge будет отправлен на сервер svn, он будет содержать все изменения всех коммитов для этого слияния, поэтому вы потеряете историю этих коммитов, но не изменения в коде.

Обработка пустых папок

git не распознает концепцию папок, он просто работает с файлами и файлами. Это означает, что git не отслеживает пустые папки. SVN, однако, делает. Использование git-svn означает, что по умолчанию *любые изменения, которые вы делаете с пустыми папками с git, не будут распространяться на SVN* .

Использование флага `--rmdir` при выпуске комментария исправляет эту проблему и удаляет пустую папку в SVN, если вы локально удаляете последний файл внутри него:

```
git svn dcommit --rmdir
```

К сожалению, **он не удаляет существующие пустые папки** : вам нужно сделать это вручную.

Чтобы не добавлять флаг каждый раз, когда вы выполняете dcommit, или играть в него безопасно, если вы используете инструмент git GUI (например, SourceTree), вы можете установить это поведение по умолчанию с помощью команды:

```
git config --global svn.rmdir true
```

Это изменяет ваш файл .gitconfig и добавляет следующие строки:

```
[svn]
rmdir = true
```

Чтобы удалить все необработанные файлы и папки, которые должны быть пустыми для SVN, используйте команду git:

```
git clean -fd
```

Обратите внимание: предыдущая команда удалит все необработанные файлы и пустые папки, даже те, которые должны отслеживаться SVN! Если вам нужно создать агацию пустых папок, отслеживаемых SVN, используйте команду

```
git svn mkdirs
```

В практике это означает, что если вы хотите очистить свое рабочее пространство от неиспользуемых файлов и папок, вы всегда должны использовать обе команды для воссоздания пустых папок, отслеживаемых SVN:

```
git clean -fd && git svn mkdirs
```

Прочитайте ГИТ-SVN онлайн: https://riptutorial.com/ru/git/topic/2766/гит-svn

глава 23: ГИТ-ТФС

замечания

Git-tfs является сторонним инструментом для подключения репозитория Git к репозиторию Team Foundation Server («TFS»).

Большинство удаленных экземпляров TFVS будут запрашивать ваши учетные данные при каждом взаимодействии, а установка Git-Credential-Manager для Windows может не помочь. Его можно преодолеть, добавив свое *имя* и *пароль* в ваш `.git/config`

```
[tfs-remote "default"]
  url = http://tfs.mycompany.co.uk:8080/tfs/DefaultCollection/
  repository = $/My.Project.Name/
  username = me.name
  password = My733TPwd
```

Examples

gone-tfs clone

Это создаст папку с тем же именем, что и проект, т. Е. /My.Project.Name

```
$ git tfs clone http://tfs:8080/tfs/DefaultCollection/ $/My.Project.Name
```

Клон git-tfs из голого репозитория git

Клонирование из хранилища git в десять раз быстрее, чем клонирование непосредственно из TFVS и хорошо работает в командной среде. По крайней мере, один член команды должен будет создать голый репозиторий git, выполнив сначала обычный git-tfs clone. Тогда новый репозиторий может быть загружен для работы с TFVS.

```
$ git clone x:/fileshare/git/My.Project.Name.git
$ cd My.Project.Name
$ git tfs bootstrap
$ git tfs pull
```

git-tfs установить через Chocolatey

Следующее предполагает, что вы будете использовать kdiff3 для файла, различающегося, и хотя это не существенно, это хорошая идея.

```
C:\> choco install kdiff3
```

Сначала можно установить Git, чтобы вы могли указать любые параметры. Здесь также устанавливаются все инструменты Unix и «NoAutoCrlf» означает checkout as is, commit as is.

```
C:\> choco install git -params '"/GitAndUnixToolsOnPath /NoAutoCrlf"'
```

Это все, что вам действительно нужно, чтобы установить git-tfs через шоколадный.

```
C:\> choco install git-tfs
```

git-tfs Check In

Запустите диалоговое окно «Проверка» для TFVS.

```
$ git tfs checkintool
```

Это займет все ваши местные коммиты и создаст единую регистрацию.

git-tfs push

Нажмите все локальные коммиты на пульт TFVS.

```
$ git tfs rcheckin
```

Примечание: это произойдет, если требуются заметки регистрации. Их можно обойти, добавив `git-tfs-force: rcheckin` в сообщение фиксации.

Прочитайте ГИТ-ТФС онлайн: https://riptutorial.com/ru/git/topic/2660/гит-тфс

глава 24: давя

замечания

Что подавляет?

Скриптинг - это процесс принятия нескольких коммитов и объединения их в единую транзакцию, включающую все изменения от начальных коммитов.

Скручивающие и удаленные ветви

Обратите особое внимание, когда раздавливание фиксируется на ветке, которая отслеживает удаленную ветку; если вы раздавите коммит, который уже был перенесен в удаленную ветвь, две ветви будут разнесены, и вам придется использовать `git push -f` чтобы принудительно изменить эти изменения на удаленной ветке. **Имейте в виду, что это может вызвать проблемы для других, отслеживающих эту удаленную ветвь** , поэтому следует проявлять осторожность, когда раздача принудительного нажатия коммитируется в общедоступные или общие репозитории.

Если проект размещен на GitHub, вы можете включить «принудительную защиту» на некоторых ветвях, например, `master` , добавив его в `Settings` - `Branches` - `Protected Branches` .

Examples

Сквош Последние фиксации без ресайзинга

Если вы хотите, чтобы предыдущие `x` записывались в один, вы можете использовать следующие команды:

```
git reset --soft HEAD~x
git commit
```

Замените `x` числом предыдущих коммитов, которые вы хотите включить в сжатый коммит.

Имейте в виду, что это создаст *новый* коммит, по сути забывая информацию о предыдущих `x` коммитах, включая их автора, сообщение и дату. Вероятно, вы захотите *сначала* скопировать-вставить существующее сообщение фиксации.

Скрининг завершается во время перезагрузки

Записи могут быть раздавлены во время `git rebase` . Рекомендуется, чтобы вы поняли, что нужно переустановить, прежде чем пытаться выполнить сквош.

1. Определите, какую фиксацию вы хотите переустановить, и обратите внимание на хеш-код фиксации.

2. Запустить `git rebase -i [commit hash]` .

   Кроме того, вы можете набрать `HEAD~4` вместо хэша commit, чтобы просмотреть последнюю фиксацию и еще 4 фиксации до последней.

3. В редакторе, который открывается при запуске этой команды, определите, какие коммиты вы хотите сквозировать. Замените `pick` в начале этих строк `squash` чтобы сквоить их в предыдущую фиксацию.

4. Выбрав, какие коммиты вы хотите раздавить, вам будет предложено написать сообщение фиксации.

Logging Commits, чтобы определить, где переустанавливать

```
> git log --oneline
612f2f7 This commit should not be squashed
d84b05d This commit should be squashed
ac60234 Yet another commit
36d15de Rebase from here
17692d1 Did some more stuff
e647334 Another Commit
2e30df6 Initial commit

> git rebase -i 36d15de
```

На данный момент появляется ваш редактор по выбору, где вы можете описать, что вы хотите делать с фиксациями. Git предоставляет помощь в комментариях. Если вы оставите это как есть, ничего не произойдет, потому что каждая фиксация будет сохранена, и их порядок будет таким же, как и до переустановки. В этом примере мы применяем следующие команды:

```
pick ac60234 Yet another commit
squash d84b05d This commit should be squashed
pick 612f2f7 This commit should not be squashed

# Rebase 36d15de..612f2f7 onto 36d15de (3 command(s))
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
```

```
#
# Note that empty commits are commented out
```

Git log после записи сообщения о фиксации

```
> git log --oneline
77393eb This commit should not be squashed
e090a8c Yet another commit
36d15de Rebase from here
17692d1 Did some more stuff
e647334 Another Commit
2e30df6 Initial commit
```

Autosquash: код, который вы хотите раздавить во время переустановки

Учитывая следующую историю, представьте, что вы вносили изменения, которые вы хотите раздавить в commit `bbb2222 A second commit` :

```
$ git log --oneline --decorate
ccc3333 (HEAD -> master) A third commit
bbb2222 A second commit
aaa1111 A first commit
9999999 Initial commit
```

После того, как вы внесли свои изменения, вы можете добавить их в индекс как обычно, а затем зафиксировать их с `--fixup` аргумента `--fixup` со ссылкой на фиксацию, которую вы хотите выполнить:

```
$ git add .
$ git commit --fixup bbb2222
[my-feature-branch ddd4444] fixup! A second commit
```

Это создаст новую фиксацию с сообщением фиксации, которое Git может распознать во время интерактивной переадресации:

```
$ git log --oneline --decorate
ddd4444 (HEAD -> master) fixup! A second commit
ccc3333 A third commit
bbb2222 A second commit
aaa1111 A first commit
9999999 Initial commit
```

Затем выполните интерактивную `--autosquash` аргументом `--autosquash` :

```
$ git rebase --autosquash --interactive HEAD~4
```

Git предложит вам раздавить фиксацию, которую вы сделали, с `commit --fixup` в правильную позицию:

```
pick aaa1111 A first commit
pick bbb2222 A second commit
fixup ddd4444 fixup! A second commit
pick ccc3333 A third commit
```

Чтобы избежать необходимости вводить `--autosquash` при каждой перестановке, вы можете включить эту опцию по умолчанию:

```
$ git config --global rebase.autosquash true
```

Сбой во время слияния

Вы можете использовать `git merge --squash` чтобы вырезать изменения, введенные веткой, в единую фиксацию. Никакая фактическая фиксация не будет создана.

```
git merge --squash <branch>
git commit
```

Это более или менее эквивалентно использованию `git reset` , но более удобно, когда внесенные изменения имеют символическое имя. Для сравнения:

```
git checkout <branch>
git reset --soft $(git merge-base master <branch>)
git commit
```

Автосохранение и исправления

При совершении изменений можно указать, что в будущем коммит будет передан другому фиксатору, и это можно сделать так,

```
git commit --squash=[commit hash of commit to which this commit will be squashed to]
```

Можно также использовать, `--fixup=[commit hash]` альтернативно для исправления.

Также возможно использовать слова из сообщения фиксации вместо хеширования фиксации, например,

```
git commit --squash :/things
```

где будет использоваться последняя фиксация со словом «вещи».

Это сообщение `'fixup!'` начнется с `'fixup!'` или `'squash!'` за которым следует остальная часть сообщения фиксации, к которому будут переданы эти коммиты.

Когда перебазирования `--autosquash` флаг должен использоваться для использования функции autosquash / FixUp.

Прочитайте давя онлайн: https://riptutorial.com/ru/git/topic/598/давя

глава 25: Игнорирование файлов и папок

Вступление

В этом разделе показано, как избежать добавления нежелательных файлов (или изменений файлов) в репозитории Git. Существует несколько способов (глобальный или локальный `.gitignore` , `.git/exclude` , `git update-index --assume-unchanged` и `git update-index --skip-tree`), но имейте в виду, что Git управляет *контентом* , что означает: игнорирование фактически игнорирует *содержимое* папки (то есть файлы). По умолчанию пустая папка будет проигнорирована, так как она не может быть добавлена в любом случае.

Examples

Игнорирование файлов и каталогов с помощью файла .gitignore

Вы можете заставить Git игнорировать определенные файлы и каталоги, то есть исключить их от отслеживания Git - путем создания одного или нескольких файлов `.gitignore` в вашем репозитории.

В проектах программного обеспечения `.gitignore` обычно содержит список файлов и / или каталогов, которые генерируются во время процесса сборки или во время выполнения. Записи в файле `.gitignore` могут включать имена или пути, указывающие на:

1. временные ресурсы, например, кэши, файлы журналов, скомпилированный код и т. д.
2. файлы локальной конфигурации, которые не должны использоваться совместно с другими разработчиками
3. файлы, содержащие секретную информацию, такие как пароли входа, ключи и учетные данные

При создании в каталоге верхнего уровня правила будут применяться рекурсивно ко всем файлам и подкаталогам во всем репозитории. При создании в подкаталоге правила будут применяться к этому конкретному каталогу и его подкаталогам.

Когда файл или каталог игнорируются, это не будет:

1. отслеживается Git
2. сообщается командами, такими как `git status` или `git diff`
3. с такими командами, как `git add -A`

В необычном случае, когда вам нужно игнорировать отслеживаемые файлы, следует соблюдать особую осторожность. См .: Игнорировать файлы, которые уже были переданы в репозиторий Git .

Примеры

Вот некоторые общие примеры правил в файле `.gitignore` , основанные на шаблонах файлов glob :

```
# Lines starting with `#` are comments.

# Ignore files called 'file.ext'
file.ext

# Comments can't be on the same line as rules!
# The following line ignores files called 'file.ext # not a comment'
file.ext # not a comment

# Ignoring files with full path.
# This matches files in the root directory and subdirectories too.
# i.e. otherfile.ext will be ignored anywhere on the tree.
dir/otherdir/file.ext
otherfile.ext

# Ignoring directories
# Both the directory itself and its contents will be ignored.
bin/
gen/

# Glob pattern can also be used here to ignore paths with certain characters.
# For example, the below rule will match both build/ and Build/
[bB]uild/

# Without the trailing slash, the rule will match a file and/or
# a directory, so the following would ignore both a file named `gen`
# and a directory named `gen`, as well as any contents of that directory
bin
gen

# Ignoring files by extension
# All files with these extensions will be ignored in
# this directory and all its sub-directories.
*.apk
*.class

# It's possible to combine both forms to ignore files with certain
# extensions in certain directories. The following rules would be
# redundant with generic rules defined above.
java/*.apk
gen/*.class

# To ignore files only at the top level directory, but not in its
# subdirectories, prefix the rule with a `/`
/*.apk
/*.class

# To ignore any directories named DirectoryA
# in any depth use ** before DirectoryA
# Do not forget the last /,
# Otherwise it will ignore all files named DirectoryA, rather than directories
**/DirectoryA/
# This would ignore
```

```
# DirectoryA/
# DirectoryB/DirectoryA/
# DirectoryC/DirectoryB/DirectoryA/
# It would not ignore a file named DirectoryA, at any level

# To ignore any directory named DirectoryB within a
# directory named DirectoryA with any number of
# directories in between, use ** between the directories
DirectoryA/**/DirectoryB/
# This would ignore
# DirectoryA/DirectoryB/
# DirectoryA/DirectoryQ/DirectoryB/
# DirectoryA/DirectoryQ/DirectoryW/DirectoryB/

# To ignore a set of files, wildcards can be used, as can be seen above.
# A sole '*' will ignore everything in your folder, including your .gitignore file.
# To exclude specific files when using wildcards, negate them.
# So they are excluded from the ignore list:
!.gitignore

# Use the backslash as escape character to ignore files with a hash (#)
# (supported since 1.6.2.1)
\#*#
```

Большинство файлов `.gitignore` являются стандартными для разных языков, поэтому для начала работы здесь приведены образцы файлов `.gitignore` перечисленных на языке, из которого можно клонировать или копировать / вносить изменения в ваш проект. Кроме того, для нового проекта вы можете автоматически генерировать стартовый файл с помощью онлайн-инструмента .

Другие формы .gitignore

Файлы `.gitignore` предназначены для передачи как часть репозитория. Если вы хотите игнорировать определенные файлы без соблюдения правил игнорирования, вот несколько вариантов:

- Отредактируйте файл `.git/info/exclude` (используя тот же синтаксис, что и `.gitignore`). Правила будут глобальными в объеме хранилища;
- Настройте глобальный файл gitignore, который будет применять правила игнорирования ко всем вашим локальным репозиториям:

Кроме того, вы можете игнорировать локальные изменения в отслеживаемых файлах без изменения глобальной конфигурации git с помощью:

- `git update-index --skip-worktree [<file>...]` : для небольших локальных изменений
- `git update-index --assume-unchanged [<file>...]` : для производства готовые, не изменяющиеся файлы вверх по течению

Подробнее о различиях между последними флагами и документации по `git update-index` см.

Подробнее .

Очистка игнорируемых файлов

Вы можете использовать `git clean -X` для очистки игнорируемых файлов:

```
git clean -Xn #display a list of ignored files
git clean -Xf #remove the previously displayed files
```

Примечание: `-X` (caps) очищает *только* игнорируемые файлы. Используйте `-x` (без ограничений), чтобы удалить ненужные файлы.

Подробнее см. Документацию `git clean` .

Дополнительную информацию см. В руководстве Git .

Исключения в файле .gitignore

Если вы игнорируете файлы с помощью шаблона, но имеете исключения, префикс восклицательного знака (!) К исключению. Например:

```
*.txt
!important.txt
```

В приведенном выше примере Git игнорирует все файлы с расширением `.txt` за исключением файлов с именем `important.txt` .

Если файл находится в папке проигнорировано, вы **не** можете повторно включить его так легко:

```
folder/
!folder/*.txt
```

В этом примере все .txt-файлы в папке будут игнорироваться.

Правильный способ заключается в повторном включении самой папки в отдельной строке, а затем игнорировать все файлы в `folder` на `*` , наконец, повторно включить `*.txt` в `folder` , как показано ниже:

```
!folder/
folder/*
!folder/*.txt
```

Примечание . Для имен файлов, начинающихся с восклицательного знака, добавьте два

восклицательных знака или выйдите с символом `\` :

```
!!includethis
\!excludethis
```

Глобальный файл .gitignore

Чтобы Git игнорировал определенные файлы во всех репозиториях, вы можете создать глобальный .gitignore со следующей командой в своем терминале или командной строке:

```
$ git config --global core.excludesfile <Path_To_Global_gitignore_file>
```

Теперь Git будет использовать это в дополнение к собственному файлу .gitignore каждого репозитория. Правила для этого:

- Если локальный файл `.gitignore` явно содержит файл, а глобальный `.gitignore` игнорирует его, локальный `.gitignore` имеет приоритет (файл будет включен)
- Если репозиторий клонирован на нескольких компьютерах, глобальный `.gigignore` должен быть загружен на всех машинах или, по крайней мере, включать его, поскольку проигнорированные файлы будут `.gitignore` на репо, тогда как ПК с глобальным `.gitignore` не будет обновлять его , Вот почему специфический `.gitignore` - лучшая идея, чем глобальная, если проект обрабатывается командой

Этот файл является хорошим местом для игнорирования игнорирования `.DS_Store` платформы, компьютера или пользователя, например OSX `.DS_Store` , Windows `Thumbs.db` или Vim `*.ext~` и `*.ext.swp` игнорирует, если вы не хотите сохранять их в репозитории , Поэтому один член команды, работающий над OS X, может добавить все `.DS_STORE` и `_MACOSX` (что фактически бесполезно), в то время как другой член команды в Windows может игнорировать все `thumbs.bd`

Игнорировать файлы, которые уже были переданы в репозиторий Git

Если вы уже добавили файл в свой репозиторий Git и теперь хотите **прекратить его отслеживать** (чтобы он не присутствовал в будущих коммитах), вы можете удалить его из индекса:

```
git rm --cached <file>
```

Это приведет к удалению файла из репозитория и предотвращению отслеживания дальнейших изменений Git. Параметр `--cached` гарантирует, что файл не будет физически удален.

Обратите внимание, что ранее добавленное содержимое файла по-прежнему будет отображаться через историю Git.

Имейте в виду, что если кто-то еще вытащит из репозитория после удаления файла из индекса, **их копия будет физически удалена** .

Вы можете заставить Git притвориться, что версия рабочего каталога файла обновлена и вместо этого прочитала индексную версию (таким образом, игнорируя изменения в ней) с битом « skip worktree »:

```
git update-index --skip-worktree <file>
```

На этот бит не влияет запись, безопасность контента по-прежнему является первоочередной задачей. Вы никогда не потеряете свои драгоценные игнорируемые изменения; с другой стороны, этот бит конфликтует с тиснением: чтобы удалить этот бит, используйте

```
git update-index --no-skip-worktree <file>
```

Иногда ***ошибочно*** рекомендуется лгать Гиту и предположить, что файл остается неизменным, не изучая его. Он выглядит на первый взгляд как игнорирование любых дальнейших изменений в файле, не удаляя его из его индекса:

```
git update-index --assume-unchanged <file>
```

Это заставит git игнорировать любые изменения, внесенные в файл (имейте в виду, что если вы поместите какие-либо изменения в этот файл или вы его запишете, **ваши проигнорированные изменения будут потеряны**)

Если вы хотите, чтобы git снова «заботился» об этом файле, выполните следующую команду:

```
git update-index --no-assume-unchanged <file>
```

Проверка игнорирования файла

Команда `git check-ignore` сообщает о файлах, игнорируемых Git.

Вы можете передавать имена файлов в командной строке, а `git check-ignore` будет отображать имена файлов, которые игнорируются. Например:

```
$ cat .gitignore
*.o
$ git check-ignore example.o Readme.md
example.o
```

Здесь только * .o файлы определены в .gitignore, поэтому Readme.md не указан в выводе

`git check-ignore .`

Если вы хотите увидеть строку, в которой .gitignore отвечает за игнорирование файла, добавьте -v в команду git check-ignore:

```
$ git check-ignore -v example.o Readme.md
.gitignore:1:*.o        example.o
```

Начиная с Git 1.7.6, вы также можете использовать `git status --ignored` , чтобы увидеть проигнорированные файлы. Дополнительную информацию об этом можно найти в официальной документации или в разделе «Поиск файлов, игнорируемых с помощью .gitignore» .

Игнорирование файлов в подпапках (несколько файлов gitignore)

Предположим, что у вас есть структура репозитория:

```
examples/
    output.log
src/
    <files not shown>
    output.log
README.md
```

`output.log` в каталоге примеров действителен и требуется, чтобы проект собирал понимание, в то время как один под `src/` создается во время отладки и не должен находиться в истории или части хранилища.

Существует два способа игнорировать этот файл. Вы можете поместить абсолютный путь в файл `.gitignore` в корень рабочего каталога:

```
# /.gitignore
src/output.log
```

Кроме того, вы можете создать файл `.gitignore` каталоге `src/` и проигнорировать файл, относящийся к этому `.gitignore` :

```
# /src/.gitignore
output.log
```

Игнорирование файла в любом каталоге

Чтобы игнорировать файл `foo.txt` в **любом** каталоге, вы должны просто написать его имя:

```
foo.txt # matches all files 'foo.txt' in any directory
```

Если вы хотите игнорировать файл только в части дерева, вы можете указать подкаталоги

определенного каталога с `**` pattern:

```
bar/**/foo.txt # matches all files 'foo.txt' in 'bar' and all subdirectories
```

Или вы можете создать файл `.gitignore` каталоге `bar/` . Эквивалентным предыдущему примеру будет создание файла `bar/.gitignore` с этим содержимым:

```
foo.txt # matches all files 'foo.txt' in any directory under bar/
```

Игнорировать файлы локально без правил игнорирования

`.gitignore` игнорирует файлы локально, но предназначен для `.gitignore` к репозиторию и совместно с другими участниками и пользователями. Вы можете установить глобальный `.gitignore` , но тогда все ваши репозитории будут делиться этими настройками.

Если вы хотите игнорировать определенные файлы в репозитории локально и не создавать файловую часть какого-либо репозитория, отредактируйте файл `.git/info/exclude` внутри своего репозитория.

Например:

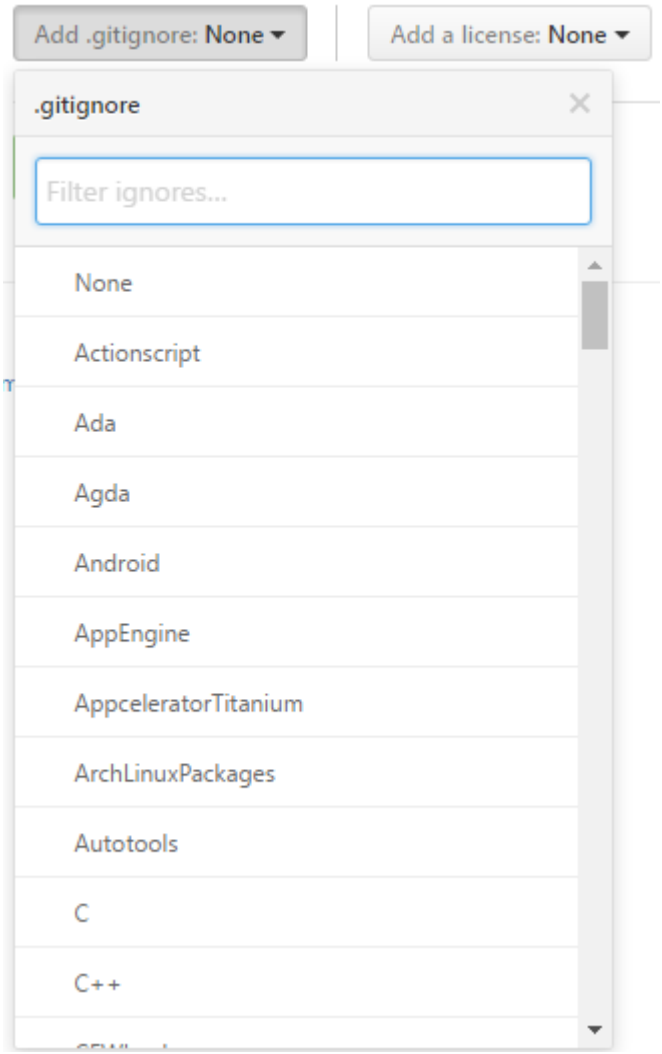
```
# these files are only ignored on this repo
# these rules are not shared with anyone
# as they are personal
gtk_tests.py
gui/gtk/tests/*
localhost
pushReports.py
server/
```

Заполненные шаблоны .gitignore

Если вы не знаете, какие правила перечислять в вашем файле `.gitignore` или просто хотите добавить общепринятые исключения в свой проект, вы можете выбрать или сгенерировать файл `.gitignore` :

- https://www.gitignore.io/
- https://github.com/github/gitignore

Многие хостинговые сервисы, такие как GitHub и BitBucket, предлагают возможность генерации файлов `.gitignore` на основе языков программирования и IDE, которые вы можете использовать:



Игнорирование последующих изменений в файле (без его удаления)

Иногда вы хотите иметь файл, хранящийся в Git, но игнорировать последующие изменения.

Скажите Git игнорировать изменения в файле или каталоге с помощью `update-index` :

```
git update-index --assume-unchanged my-file.txt
```

Вышеупомянутая команда дает указание Git предположить, `my-file.txt` не был изменен, а не проверять или сообщать об изменениях. Файл все еще присутствует в репозитории.

Это может быть полезно для предоставления значений по умолчанию и разрешения переопределения локальной среды, например:

```
# create a file with some values in
cat <<EOF
MYSQL_USER=app
MYSQL_PASSWORD=FIXME_SECRET_PASSWORD
EOF > .env

# commit to Git
git add .env
```

```
git commit -m "Adding .env template"

# ignore future changes to .env
git update-index --assume-unchanged .env

# update your password
vi .env

# no changes!
git status
```

Игнорирование только части файла [заглушки]

Иногда вам может потребоваться локальные изменения в файле, который вы не хотите комментировать или публиковать. В идеале локальные настройки должны быть сконцентрированы в отдельном файле, который может быть помещен в `.gitignore` , но иногда в качестве краткосрочного решения может быть полезно иметь что-то локальное в зарегистрированном файле.

Вы можете сделать Git «unsee» эти строки, используя чистый фильтр. Они даже не появятся в разностях.

Предположим, что это фрагмент файла `file1.c` :

```
struct settings s;
s.host = "localhost";
s.port = 5653;
s.auth = 1;
s.port = 15653; // NOCOMMIT
s.debug = 1; // NOCOMMIT
s.auth = 0; // NOCOMMIT
```

Вы не хотите публиковать `NOCOMMIT` .

Создайте фильтр «nocommit», добавив его в конфигурационный файл Git, например `.git/config` :

```
[filter "nocommit"]
    clean=grep -v NOCOMMIT
```

Добавьте (или создайте) это в `.git/info/attributes` или `.gitmodules` :

```
file1.c filter=nocommit
```

И ваши линии NOCOMMIT скрыты от Git.

Предостережения:

- Использование чистого фильтра замедляет обработку файлов, особенно в Windows.
- Пропущенная строка может исчезнуть из файла, когда Git обновляет ее. Его можно

противодействовать фильтром размытия, но это сложнее.
- Не тестировалось в Windows

Игнорирование изменений в отслеживаемых файлах. [Заглушка]

.gitignore и `.git/info/exclude` работают только для файлов без следа.

Чтобы установить флаг игнорирования в отслеживаемом файле, используйте команду update-index :

```
git update-index --skip-worktree myfile.c
```

Чтобы восстановить это, используйте:

```
git update-index --no-skip-worktree myfile.c
```

Вы можете добавить этот фрагмент в свою глобальную конфигурацию git, чтобы иметь более удобную `git hidden` `git hide` , `git unhide` и `git hidden` команды:

```
[alias]
    hide   = update-index --skip-worktree
    unhide = update-index --no-skip-worktree
    hidden  = "!git ls-files -v | grep ^[hsS] | cut -c 3-"
```

Вы также можете использовать опцию --assume-неизменной с функцией update-index

```
git update-index --assume-unchanged <file>
```

Если вы хотите снова просмотреть этот файл для изменений, используйте

```
git update-index --no-assume-unchanged <file>
```

Когда задан флаг -измеренный неизмененный, пользователь обещает не изменять файл и позволяет Git предположить, что рабочий файл дерева соответствует тому, что записано в index.Git не удастся, если ему необходимо изменить этот файл в индексе например, при слиянии в фиксации; таким образом, в случае, если файл с необработанной версией изменен вверх по потоку, вам придется обрабатывать ситуацию вручную. В этом случае основное внимание уделяется производительности.

Хотя флаг -skip-worktree полезен, когда вы даете указание git не касаться определенного файла из-за того, что файл будет изменен локально, и вы не захотите случайно зафиксировать изменения (т. Е. Файл конфигурации / свойств, сконфигурированный для определенного среда). Skip-worktree имеет приоритет над принятием-неизменным, когда оба установлены.

Очистить уже зафиксированные файлы, но включенные в .gitignore

Иногда случается, что файл отслеживается git, но в более поздний момент времени был добавлен в .gitignore, чтобы остановить его отслеживание. Очень распространенный сценарий забыть очистить такие файлы до его добавления в .gitignore. В этом случае старый файл все равно будет висящим в репозитории.

Чтобы устранить эту проблему, можно было выполнить «сухое» удаление всего в репозитории, а затем повторное добавление всех файлов обратно. Пока у вас нет ожидающих изменений и `--cached` параметр `--cached` , эта команда достаточно безопасна для запуска:

```
# Remove everything from the index (the files will stay in the file system)
$ git rm -r --cached .

# Re-add everything (they'll be added in the current state, changes included)
$ git add .

# Commit, if anything changed. You should see only deletions
$ git commit -m 'Remove all files that are in the .gitignore'

# Update the remote
$ git push origin master
```

Создать пустую папку

Невозможно добавить и зафиксировать пустую папку в Git из-за того, что Git управляет *файлами* и прикрепляет к ним свой каталог, который сглаживает и фиксирует скорость. Чтобы обойти это, существует два метода:

Метод первый: `.gitkeep`

Один хак, чтобы обойти это, - использовать файл `.gitkeep` для регистрации папки для Git. Для этого просто создайте требуемый каталог и добавьте файл `.gitkeep` в папку. Этот файл пуст и не служит никакой другой цели, кроме как просто зарегистрировать эту папку. Для этого в Windows (который имеет неудобные соглашения об именах файлов) просто запустите git bash в каталоге и запустите команду:

    $ touch .gitkeep

Эта команда просто делает пустой файл `.gitkeep` в текущем каталоге

`dummy.txt` способ: `dummy.txt`

Другой взлом для этого очень похож на выше, и те же шаги могут быть выполнены, но вместо `.gitkeep` просто используйте вместо него `dummy.txt` . Это дает дополнительный бонус, позволяющий легко создавать его в Windows с помощью контекстного меню. И вы также можете оставить в них смешные сообщения. Вы также можете использовать файл `.gitkeep` для отслеживания пустого каталога. `.gitkeep` обычно представляет собой пустой файл, который добавляется для отслеживания пустой строки.

Поиск файлов, игнорируемых .gitignore

Вы можете перечислить все файлы, игнорируемые git в текущем каталоге командой:

```
git status --ignored
```

Итак, если у нас есть структура репозитория, вот так:

```
.git
.gitignore
./example_1
./dir/example_2
./example_2
```

... и .gitignore файл, содержащий:

```
example_2
```

... чем результат команды будет:

```
$ git status --ignored

On branch master

Initial commit

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

.gitignore
.example_1

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

dir/
example_2
```

Если вы хотите перечислить рекурсивно проигнорированные файлы в каталогах, вам нужно использовать дополнительный параметр - `--untracked-files=all`

Результат будет выглядеть так:

```
$ git status --ignored --untracked-files=all
On branch master

Initial commit

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

.gitignore
example_1
```

```
Ignored files:
  (use "git add -f <file>..." to include in what will be committed)

dir/example_2
example_2
```

Прочитайте Игнорирование файлов и папок онлайн: https://riptutorial.com/ru/git/topic/245/игнорирование-файлов-и-папок

глава 26: Изменить имя репозитория git

Вступление

Если вы измените имя репозитория на удаленной стороне, например, ваш github или битбакет, когда вы нажмете свой существующий код, вы увидите сообщение об ошибке: Fatal error, repository not found **.

Examples

Изменение локальной настройки

Перейдите к терминалу,

```
cd projectFolder
git remote -v (it will show previous git url)
git remote set-url origin https://username@bitbucket.org/username/newName.git
git remote -v (double check, it will show new git url)
git push (do whatever you want.)
```

Прочитайте Изменить имя репозитория git онлайн: https://riptutorial.com/ru/git/topic/9291/изменить-имя-репозитория-git

глава 27: инсценировка

замечания

Стоит отметить, что эта постановка имеет мало общего с «файлами» и все, что связано с изменениями в каждом заданном файле. Мы создаем файлы, содержащие изменения, и git отслеживает изменения как фиксации (даже когда изменения в фиксации совершаются через несколько файлов).

Различие между файлами и коммитами может показаться незначительным, но понимание этой разницы является основополагающим для понимания основных функций, таких как вишневый выбор и diff. (См. Разочарование в комментариях относительно сложности принятого ответа, который предлагает вишневый выбор в качестве инструмента управления файлами .)

Что хорошего для объяснения понятий? Это в комментариях?

Ключевые идеи:

Файлы являются более распространенной метафорой этих двух в области информационных технологий. Лучшая практика диктует, что имя файла не изменяется по мере изменения его содержимого (с несколькими признанными исключениями).

Конец - это метафора, которая уникальна для управления исходным кодом. Commits - это изменения, связанные с конкретными усилиями, такими как исправление ошибок. Записи часто включают несколько файлов. Единственное незначительное исправление ошибки может включать в себя настройки шаблонов и CSS в уникальных файлах. Поскольку изменения описаны, разработаны, задокументированы, рассмотрены и развернуты, изменения в отдельных файлах можно аннотировать и обрабатывать как единое целое. Единственным элементом в этом случае является фиксация. Не менее важно, сосредоточив внимание только на фиксации во время обзора, позволяет без изменений игнорировать неизмененные строки кода в различных затронутых файлах.

Examples

Создание отдельного файла

Чтобы создать файл для совершения, выполните

```
git add <filename>
```

Постановка всех изменений в файлы

```
git add -A
```

2,0

```
git add .
```

В версии 2.x `git add .` будет сгенерировать все изменения в файлах в текущем каталоге и во всех его подкаталогах. Однако в 1.x он будет создавать только новые и измененные файлы, а не удаленные файлы .

Используйте `git add -A` или его эквивалентную команду `git add --all` , чтобы `git add --all` все изменения файлов в любой версии git.

Снятые с этапа файлы

```
git rm filename
```

Чтобы удалить файл из git без его удаления с диска, используйте флаг `--cached`

```
git rm --cached filename
```

Нестандартный файл, содержащий изменения

```
git reset <filePath>
```

Интерактивное добавление

`git add -i` (или `--interactive`) даст вам интерактивный интерфейс, в котором вы можете отредактировать индекс, чтобы подготовить то, что вы хотите иметь в следующем коммите. Вы можете добавлять и удалять изменения во всех файлах, добавлять необработанные файлы и удалять файлы из отслеживаемых объектов, а также выбирать подразделы для внесения изменений в индекс, выбирая куски добавляемых изменений, разделяя эти фрагменты или даже редактируя diff , Многие графические инструменты фиксации для Git (например, `git gui`) включают такую функцию; это может быть проще в использовании, чем версия командной строки.

Это очень полезно (1), если у вас есть запутанные изменения в рабочем каталоге, который вы хотите поместить в отдельные коммиты, а не все в одном единственном коммите (2), если вы находитесь в середине интерактивной перестановки и хотите разбить большой фиксация.

```
$ git add -i
           staged     unstaged path
  1:    unchanged        +4/-4 index.js
  2:        +1/-0      nothing package.json
```

```
*** Commands ***
  1: status       2: update       3: revert       4: add untracked
  5: patch        6: diff         7: quit         8: help
What now>
```

Верхняя половина этого вывода показывает текущее состояние индекса, разбитого на поставленные и неустановленные столбцы:

1. `index.js` было добавлено 4 строки и удалено 4 строки. В настоящее время он не поставлен, поскольку текущие отчеты о состоянии «неизменны». Когда этот файл будет поставлен, бит `+4/-4` будет передан в поэтапный столбец, а столбец с неустановленностью будет читать «ничего».
2. `package.json` добавил одну строку и был поставлен. Дальнейших изменений нет, так как он был поставлен, как указано «ничейной» строкой под неустановленным столбцом.

Нижняя половина показывает, что вы можете сделать. Введите либо число (1-8), либо букву (`s` , `u` , `r` , `a` , `p` , `d` , `q` , `h`).

`status` показывает выход, идентичный верхней части вышеприведенного вывода.

`update` позволяет вам вносить дополнительные изменения в поставленные фиксации с дополнительным синтаксисом.

`revert` вернет информацию о поставленной фиксации обратно в HEAD.

`add untracked` позволяет добавлять пути к файлам, ранее не отслеживаемые с помощью контроля версий.

`patch` позволяет выбрать один путь из выхода, аналогичного `status` для дальнейшего анализа.

`diff` показывает, что будет сделано.

`quit` завершает команду.

`help` дает дополнительную помощь в использовании этой команды.

Добавить изменения в hunk

Вы можете увидеть, какие «чанки» работы будут поставлены для фиксации с использованием флага патча:

```
git add -p
```

или же

```
git add --patch
```

Это открывает интерактивное приглашение, которое позволяет вам взглянуть на разницу и позволить вам решить, хотите ли вы включить их или нет.

```
Stage this hunk [y,n,q,a,d,/,s,e,?]?
```

- `y` ставьте этот кусок для следующего фиксации
- `n` не ставьте этот кусок для следующего фиксации
- `q` quit; не ставьте этот кусок или какой-либо из оставшихся кусков
- `на` сцене этот кусок и все последующие ханки в файле
- `d` не ставьте этот кусок или какой-либо из последующих хаков в файл
- `g` выберите кусок, чтобы перейти к
- `/` поиск совпадающего с заданным регулярным выражением
- `j` оставить этот кусок нерешенным, см. следующий неустановленный кусок
- `Я` оставляю этот кусок нерешенным, см. Следующий кусок
- `k` оставить этот кусок нерешенным, см. предыдущий неопределенный кусок
- `K` оставить этот кусок неопределенным, см. Предыдущий кусок
- `ы` разделить текущий ломоть на более мелкие куски
- `e` вручную изменить текущий ломоть
- `?` распечатать hunk help

Это позволяет легко поймать изменения, которые вы не хотите совершать.

Вы также можете открыть это через `git add --interactive` и выбрать `p` .

Показать поэтапные изменения

Чтобы отобразить сегменты, которые поставлены для фиксации:

```
git diff --cached
```

Прочитайте инсценировка онлайн: https://riptutorial.com/ru/git/topic/244/инсценировка

глава 28: Использование файла .gitattributes

Examples

Отключить нормализацию окончания строки

Создайте файл `.gitattributes` в корне проекта, содержащий:

```
* -text
```

Это эквивалентно установке `core.autocrlf = false` .

Нормальная нормализация линейного окончания

Создайте файл `.gitattributes` в корне проекта, содержащий:

```
* text=auto
```

Это приведет к тому, что все текстовые файлы (идентифицированные Git) будут переданы с LF, но будут проверены в соответствии с дефолтом операционной системы хоста.

Это эквивалентно рекомендуемым значениям `core.autocrlf` умолчанию:

- `input` в Linux / macOS
- `true` в Windows

Идентификация двоичных файлов

Git очень хорошо разбирается в двоичных файлах, но вы можете явно указать, какие файлы являются двоичными. Создайте файл `.gitattributes` в корне проекта, содержащий:

```
*.png binary
```

`binary` - это встроенный атрибут макроса, эквивалентный `-diff -merge -text` .

Заполненные шаблоны .gitattribute

Если вы не знаете, какие правила перечислять в файле `.gitattributes` или просто хотите добавить общепринятые атрибуты в свой проект, вы можете создать или сгенерировать файл `.gitattributes` адресу:

- https://gitattributes.io/

- https://github.com/alexkaratarakis/gitattributes

Прочитайте Использование файла .gitattributes онлайн: https://riptutorial.com/ru/git/topic/1269/использование-файла--gitattributes

глава 29: История перезаписи с фильтром

Examples

Изменение автора коммитов

Вы можете использовать фильтр среды для изменения автора коммитов. Просто измените и экспортируйте `$GIT_AUTHOR_NAME` в скрипт, чтобы изменить, кто создал коммит.

Создайте файл `filter.sh` с таким содержимым:

```
if [ "$GIT_AUTHOR_NAME" = "Author to Change From" ]
then
    export GIT_AUTHOR_NAME="Author to Change To"
    export GIT_AUTHOR_EMAIL="email.to.change.to@example.com"
fi
```

Затем запустите `filter-branch` из командной строки:

```
chmod +x ./filter.sh
git filter-branch --env-filter ./filter.sh
```

Установка git-коммиттера, равная фиксации автора

Эта команда, заданная диапазоном фиксации `commit1..commit2` , перезаписывает историю, так что git commit author становится также git- `commit1..commit2` :

```
git filter-branch -f --commit-filter \
   'export GIT_COMMITTER_NAME=\"$GIT_AUTHOR_NAME\";
    export GIT_COMMITTER_EMAIL=\"$GIT_AUTHOR_EMAIL\";
    export GIT_COMMITTER_DATE=\"$GIT_AUTHOR_DATE\";
    git commit-tree $@' \
    -- commit1..commit2
```

Прочитайте История перезаписи с фильтром онлайн: https://riptutorial.com/ru/git/topic/2825/история-перезаписи-с-фильтром

глава 30: Клиенты Git GUI

Examples

Рабочий стол GitHub

Веб-сайт: https://desktop.github.com
Цена: бесплатно
Платформы: OS X и Windows
Разработано: GitHub

Git Kraken

Веб-сайт: https://www.gitkraken.com
Цена: $ 60 / лет (бесплатно для некоммерческих, образовательных, некоммерческих, стартапов или личного пользования)
Платформы: Linux, OS X, Windows
Разработано: Axosoft

SourceTree

Веб-сайт: https://www.sourcetreeapp.com
Цена: бесплатно (учет необходим)
Платформы: OS X и Windows
Разработчик: Atlassian

gitk и git-gui

> Когда вы устанавливаете Git, вы также получаете свои визуальные инструменты, gitk и git-gui.
>
> `gitk` - графический просмотрщик. Подумайте об этом как о мощной оболочке графического интерфейса для git log и git grep. Это инструмент для использования, когда вы пытаетесь найти что-то, что происходило в прошлом, или визуализировать историю вашего проекта.
>
> Gitk проще всего вызывать из командной строки. Просто cd в репозиторий Git и введите:
>
> ```
> $ gitk [git log options]
> ```
>
> Gitk принимает множество параметров командной строки, большинство из которых передаются в основное действие git log. Вероятно, одним из самых

полезных является флаг `--all` , который сообщает gitk, чтобы показать, что коммиты достижимы из любого ref, а не только HEAD. Интерфейс Gitk выглядит так:

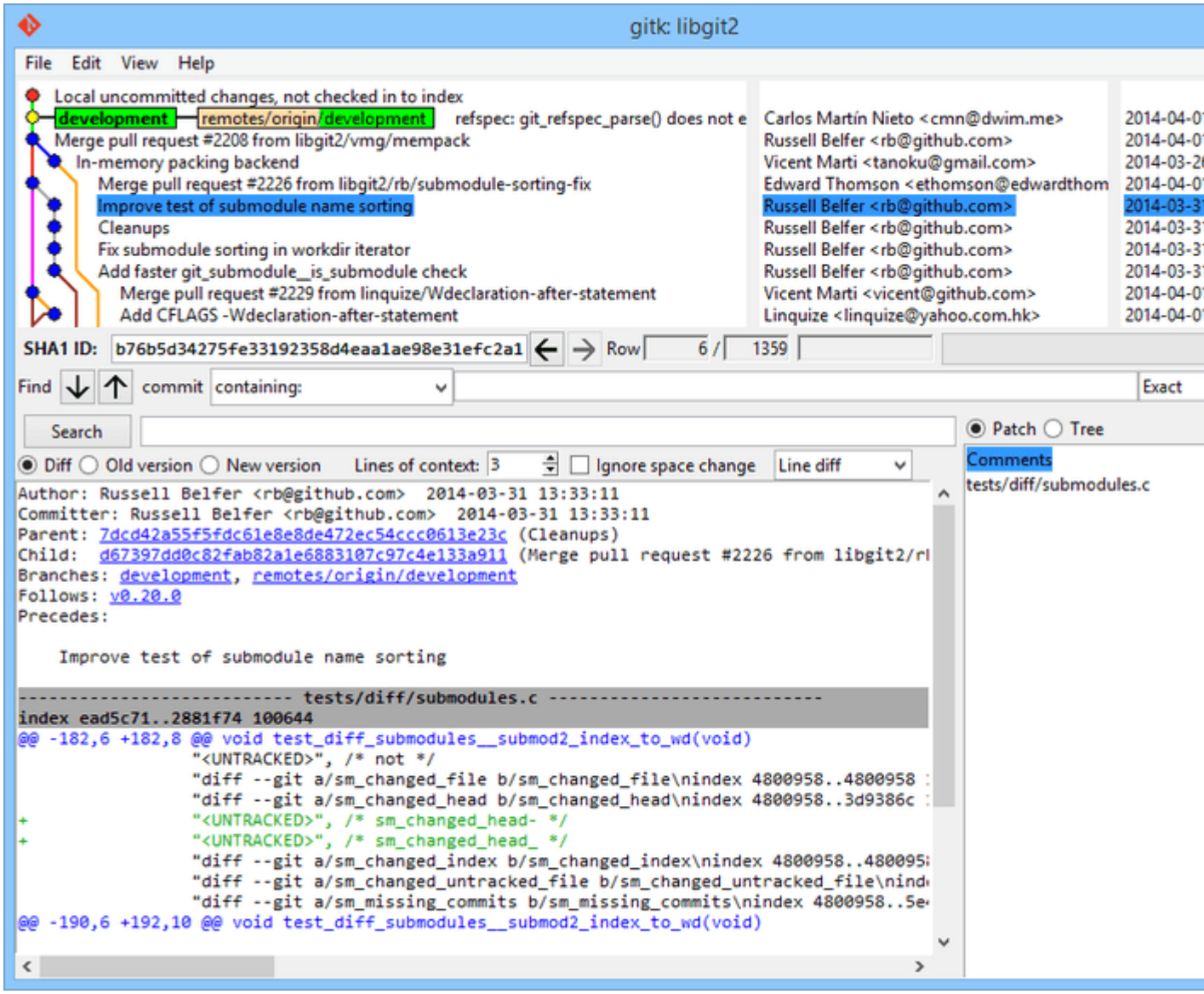


Рисунок 1-1. Смотритель истории гитков.

Наверху что-то похожее на вывод git log -graph; каждая точка представляет собой фиксацию, линии представляют отношения родителя, а ссылки отображаются как цветные прямоугольники. Желтая точка представляет HEAD, а красная точка представляет собой изменения, которые еще не стали фиксацией. В нижней части находится вид выбранной фиксации; комментарии и патч слева, а также сводный обзор справа. В промежутке находится набор элементов управления, используемых для поиска истории.

Вы можете получить доступ ко многим связанным с git функциям, щелкнув правой кнопкой мыши имя филиала или сообщение фиксации. Например, при проверке другой ветви или вишни выберите фиксацию, которую легко сделать одним щелчком мыши.

`git-gui` , с другой стороны, в первую очередь является инструментом для совершения коммитов. Это также проще всего вызвать из командной строки:

```
$ git gui
```

И это выглядит примерно так:

Инструмент фиксации `git-gui` .

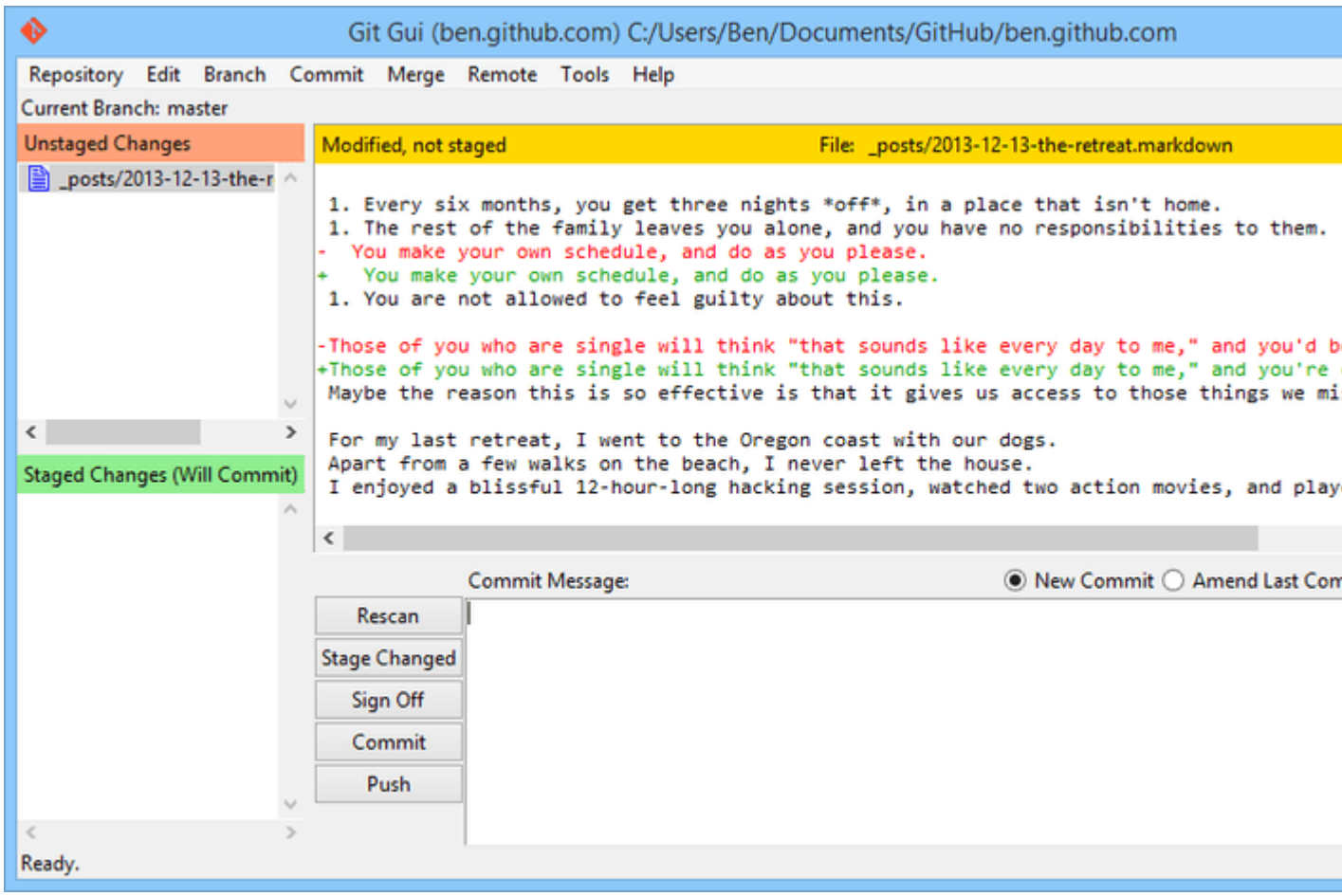


Рисунок 1-2. Инструмент фиксации git-gui.

Слева - индекс; неустановленные изменения находятся сверху, поэтапные изменения внизу. Вы можете перемещать целые файлы между двумя состояниями, нажимая на их значки или вы можете выбрать файл для просмотра, щелкнув его имя.

В правом верхнем углу находится представление diff, в котором отображаются изменения для выбранного в данный момент файла. Вы можете создавать отдельные куски (или отдельные линии), щелкнув правой кнопкой мыши в этой области.

В правом нижнем углу находится сообщение и область действия. Введите свое сообщение в текстовое поле и нажмите «Зафиксировать», чтобы сделать что-то похожее на git commit. Вы также можете изменить последнее фиксацию, выбрав

переключатель «Изменить», который обновит область «Поэтапные изменения» с содержимым последнего фиксации. Затем вы можете просто выполнить или изменить некоторые изменения, изменить сообщение фиксации и снова нажать «Зафиксировать», чтобы заменить старый фиксатор на новый.

gitk и git-gui являются примерами целевых инструментов. Каждый из них предназначен для определенной цели (просмотр истории и создание коммитов, соответственно) и опускание функций, не необходимых для этой задачи.

Источник: https://git-scm.com/book/en/v2/Git-in-Other-Environments-Graphical-Interfaces

SmartGit

Веб-сайт: http://www.syntevo.com/smartgit/
Цена: Бесплатно только для некоммерческого использования. Бессрочная лицензия стоит 99 долларов США
Платформы: Linux, OS X, Windows
Разработано: syntevo

Расширения Git

Веб-сайт: https://gitextensions.github.io
Цена: бесплатно
Платформа: Windows

Прочитайте Клиенты Git GUI онлайн: https://riptutorial.com/ru/git/topic/5148/клиенты-git-gui

глава 31: Клонирование репозиториев

Синтаксис

- git clone [<options>] [-] <repo> [<dir>]
- git clone [-template = <template_directory>] [-l] [-s] [-no-hardlinks] [-q] [-n] [--bare] [--mirror] [-o <name>] [-b <имя>] [-u <upload-pack>] [--reference <репозиторий>] [--dissociate] [--separate-git-dir <git dir>] [--depth <depth>] [- [no-] single-branch] [--рекурсивный | --recurse-subodules] [- [no-] мелкие субмодули] [--jobs <n>] [-] <репозиторий> [<каталог>]

Examples

Мелкий клон

Клонирование огромного хранилища (например, проект с многолетней историей) может занять много времени или сбой из-за количества передаваемых данных. В случаях, когда вам не нужна полная история, вы можете сделать мелкий клон:

```
git clone [repo_url] --depth 1
```

Вышеприведенная команда будет извлекать только последнюю фиксацию из удаленного репозитория.

Имейте в виду, что вы не сможете разрешать слияния в неглубоком хранилище. Часто бывает неплохо взять хотя бы столько коммитов, что вам нужно будет отступить, чтобы разрешить слияния. Например, чтобы вместо этого получить последние 50 коммитов:

```
git clone [repo_url] --depth 50
```

Позже, если потребуется, вы можете получить остальную часть репозитория:

1.8.3

```
git fetch --unshallow     # equivalent of git fetch --depth=2147483647
                          # fetches the rest of the repository
```

1.8.3

```
git fetch --depth=1000    # fetch the last 1000 commits
```

Обычный клон

Чтобы загрузить весь репозиторий, включая всю историю и все ветви, введите:

```
git clone <url>
```

Приведенный выше пример поместит его в каталог с тем же именем, что и имя репозитория.

Чтобы загрузить репозиторий и сохранить его в определенном каталоге, введите:

```
git clone <url> [directory]
```

Для получения дополнительной информации посетите Clone a repository .

Клонировать конкретную ветвь

Чтобы клонировать конкретную ветвь репозитория, введите `--branch <branch name>` до URL-`--branch <branch name>` репозитория:

```
git clone --branch <branch name> <url> [directory]
```

Чтобы использовать сокращенный вариант для `--branch` , введите `-b` . Эта команда загружает весь репозиторий и проверяет `<branch name>` .

Чтобы сэкономить дисковое пространство, вы можете клонировать историю, ведущую только к одной ветви с:

```
git clone --branch <branch_name> --single-branch <url> [directory]
```

Если `--single-branch` не добавляется в команду, история всех ветвей будет клонирована в `[directory]` . Это может быть проблема с большими репозиториями.

Чтобы позже отменить флаг `--single-branch` и выбрать команду остальной части репозитория:

```
git config remote.origin.fetch "+refs/heads/*:refs/remotes/origin/*"
git fetch origin
```

Клонировать рекурсивно

1.6.5

```
git clone <url> --recursive
```

Клонирует репозиторий, а также клонирует все подмодули. Если сами подмодули содержат дополнительные подмодули, Git также будет клонировать их.

Клонирование с использованием прокси-сервера

Если вам нужно загружать файлы с помощью git под прокси-сервером, установить прокси-сервер в системной системе недостаточно. Вы также можете попробовать следующее:

```
git config --global http.proxy http://<proxy-server>:<port>/
```

Прочитайте Клонирование репозиториев онлайн: https://riptutorial.com/ru/git/topic/1405/клонирование-репозиториев

глава 32: конфигурация

Синтаксис

- git config [<file-option>] name [значение] # один из наиболее распространенных вариантов использования git config

параметры

параметр	подробности
`--system`	Редактирует общесистемный файл конфигурации, который используется для каждого пользователя (в Linux этот файл находится в `$(prefix)/etc/gitconfig`)
`--global`	Редактирует глобальный файл конфигурации, который используется для каждого репозитория, над которым вы работаете (в Linux этот файл находится в `~/.gitconfig`
`--local`	Редактирует конфигурационный файл для репозитория, который находится в `.git/config` в вашем репозитории; Это значение по умолчанию

Examples

Имя пользователя и адрес электронной почты

Сразу после установки Git первое, что вам нужно сделать, это установить свое имя пользователя и адрес электронной почты. Из оболочки введите:

```
git config --global user.name "Mr. Bean"
git config --global user.email mrbean@example.com
```

- `git config` - команда для получения или установки параметров
- `--global` означает, что файл конфигурации, специфичный для вашей учетной записи пользователя, будет отредактирован
- `user.name` и `user.email` являются ключами для переменных конфигурации; `user` - это раздел файла конфигурации. `name` и `email` являются именами переменных.
- `"Mr. Bean"` и `mrbean@example.com` - это значения, которые вы сохраняете в двух переменных. Обратите внимание на цитаты вокруг `"Mr. Bean"` , которые требуются, потому что ценность, которую вы храните, содержит пробел.

Несколько конфигураций git

У вас есть до 5 источников для конфигурации git:

- 6 файлов:
  - **`%ALLUSERSPROFILE%\Git\Config`** (только для Windows)
  - (система) **`<git>/etc/gitconfig`** , причем `<git>` является способом установки git. (в Windows это **`<git>\mingw64\etc\gitconfig`**)
  - (система) **`$XDG_CONFIG_HOME/git/config`** (только для Linux / Mac)
  - (глобальный) `~/.gitconfig` (Windows: `%USERPROFILE%\.gitconfig`)
  - (локальный) `.git/config` (в пределах git repo `$GIT_DIR`)
  - **выделенный файл** (с `git config -f`), используемый, например, для изменения конфигурации подмодулей: `git config -f .gitmodules ...`
- **командная строка с `git -c`** : `git -c core.autocrlf=false fetch` переопределит *любой* другой `core.autocrlf` на `false` , *только* для этой команды `fetch` .

Порядок важен: любая конфигурация, установленная в одном источнике, может быть переопределена источником, указанным ниже.

`git config --system/global/local` - команда для перечисления 3 из этих источников, но только git config -l перечисляет *все разрешенные* конфиги.
«разрешено» означает, что он содержит только окончательное переопределенное значение конфигурации.

Поскольку git 2.8, если вы хотите узнать, какая конфигурация исходит из какого файла, вы вводите:

```
git config --list --show-origin
```

Настройка того, какой редактор использовать

Существует несколько способов установить, какой редактор использовать для фиксации, перезагрузки и т. Д.

- Измените настройку конфигурации `core.editor` .

  ```
  $ git config --global core.editor nano
  ```

- Установите переменную среды `GIT_EDITOR` .

  Для одной команды:

  ```
  $ GIT_EDITOR=nano git commit
  ```

  Или для всех команд, выполняемых в терминале. **Примечание.** Это применимо только

до закрытия терминала.

```
$ export GIT_EDITOR=nano
```

- Чтобы изменить редактор для *всех* терминальных программ, а не только Git, установите `EDITOR` среды `VISUAL` или `EDITOR` . (См. `VISUAL` vs `EDITOR` .)

```
$ export EDITOR=nano
```

Примечание. Как и выше, это относится только к текущему терминалу; у вашей оболочки обычно будет файл конфигурации, который позволит вам установить его навсегда. (В `bash` , например, добавьте вышеприведенную строку в ваш `~/.bashrc` или `~/.bash_profile` .)

Некоторые текстовые редакторы (в основном графические интерфейсы) будут запускать только один экземпляр за раз и обычно уходят, если у вас уже есть экземпляр из них. Если это относится к вашему текстовому редактору, Git напечатает сообщение `Aborting commit due to empty commit message.` не позволяя сначала редактировать сообщение о фиксации. Если это произойдет с вами, обратитесь к документации вашего текстового редактора, чтобы узнать, есть `--wait` флаг `--wait` (или аналогичный), который заставит его приостановить до закрытия документа.

Настройка окончаний строки

Описание

При работе с командой, которая использует различные операционные системы (ОС) по всему проекту, иногда вы можете столкнуться с проблемами при работе с окончанием строки.

Майкрософт Виндоус

При работе с операционной системой Microsoft Windows (OS) окончание строк обычно имеет форму - возврат каретки + линия (CR + LF). Открытие файла, который был отредактирован с использованием Unix-машины, такой как Linux или OSX, может вызвать проблемы, поэтому кажется, что текст вообще не имеет окончаний строки. Это связано с тем, что системы Unix применяют только линейные конвейеры фидов линейных каналов (LF).

Чтобы исправить это, вы можете запустить следующую инструкцию

```
git config --global core.autocrlf=true
```

В процессе **проверки** эта команда гарантирует, что контуры строк будут настроены в

соответствии с ОС Microsoft Windows (LF -> CR + LF)

Unix на основе (Linux / OSX)

Точно так же могут возникнуть проблемы, когда пользователь на ОС на основе Unix пытается прочитать файлы, которые были отредактированы в ОС Microsoft Windows. Чтобы предотвратить непредвиденные проблемы

```
git config --global core.autocrlf=input
```

При **фиксации** это изменит окончание строки из CR + LF -> + LF

настройка только для одной команды

вы можете использовать `-c <name>=<value>` чтобы добавить конфигурацию только для одной команды.

Для фиксации в качестве другого пользователя без изменения настроек в .gitconfig:

```
git -c user.email = mail@example commit -m "some message"
```

Примечание: для этого примера вам не нужно точно `user.name` как `user.name` и `user.email` , git завершит отсутствующую информацию из предыдущих коммитов.

Настройка прокси

Если вы находитесь за прокси-сервером, вы должны сказать git об этом:

```
git config --global http.proxy http://my.proxy.com:portnumber
```

Если вы больше не находитесь за прокси-сервером:

```
git config --global --unset http.proxy
```

Автоматическое исправление опечаток

```
git config --global help.autocorrect 17
```

Это позволяет автокорректировать в git и простит вам за ваши незначительные ошибки (например, `git stats` вместо `git status`). Параметр, который вы `help.autocorrect` для `help.autocorrect` определяет, как долго система должна ждать в десятые доли секунды, прежде чем автоматически применить команду автокоррекции. В приведенной выше команде 17 означает, что git должен ждать 1,7 секунды, прежде чем применять команду автокоррекции.

Тем не менее, большие ошибки будут рассматриваться как недостающие команды, поэтому `git testingit` чего-то типа `git testingit` приведет к `git testingit` что `testingit is not a git command.`

Перечислите и отредактируйте текущую конфигурацию

Git config позволяет вам настроить, как работает git. Он обычно используется для установки вашего имени, электронной почты или любимого редактора или того, как должны выполняться слияния.

Чтобы просмотреть текущую конфигурацию.

```
$ git config --list
...
core.editor=vim
credential.helper=osxkeychain
...
```

Чтобы отредактировать конфигурацию:

```
$ git config <key> <value>
$ git config core.ignorecase true
```

Если вы намерены изменить это для всех своих хранилищ, используйте `--global`

```
$ git config --global user.name "Your Name"
$ git config --global user.email "Your Email"
$ git config --global core.editor vi
```

Вы можете снова перечислить свои изменения.

Несколько имен пользователей и адрес электронной почты

Поскольку Git 2.13, несколько имен пользователей и адреса электронной почты можно настроить с помощью фильтра папок.

Пример для Windows:

.gitconfig

Изменить: `git config --global -e`

Добавлять:

```
[includeIf "gitdir:D:/work"]
  path = .gitconfig-work.config

[includeIf "gitdir:D:/opensource/"]
```

```
  path = .gitconfig-opensource.config
```

Заметки

- Заказ зависит от последнего, который соответствует «победам».
- `/` в конце необходимо - например, `"gitdir:D:/work"` не будет работать.
- `gitdir:` префикс `gitdir:`

.gitconfig-work.config

Файл в том же каталоге, что и *.gitconfig*

```
[user]
  name = Money
  email = work@somewhere.com
```

.gitconfig-opensource.config

Файл в том же каталоге, что и *.gitconfig*

```
[user]
  name = Nice
  email = cool@opensource.stuff
```

Пример для Linux

```
[includeIf "gitdir:~/work/"]
  path = .gitconfig-work
[includeIf "gitdir:~/opensource/"]
  path = .gitconfig-opensource
```

Содержимое файла и примечания в разделе Windows.

Прочитайте конфигурация онлайн: https://riptutorial.com/ru/git/topic/397/конфигурация

глава 33: Крючки

Синтаксис

- .git / крючки / applypatch-тзд
- .git / крючки / фиксации-тзд
- .git / Крючки / после обновления
- .git / крючки / предварительно applypatch
- .git / крючки / предварительной фиксации
- .git / Крючки / подготовить фиксации-сообщ
- .git / Крючки / предварительно толчок
- .git / крючки / предварительно Rebase
- .git / Крючки / обновление

замечания

`--no-verify` или `-n` чтобы пропустить все локальные перехватчики в заданной команде git. Например: `git commit -n`

Информация на этой странице была собрана из официальных документов Git и Atlassian .

Examples

Commit-сообщ

Этот крючок похож на hook `prepare-commit-msg` , но он вызывается после того, как пользователь вводит сообщение фиксации, а не раньше. Обычно это используется для предупреждения разработчиков, если их сообщение о фиксации находится в неправильном формате.

Единственным аргументом, переданным этому крюку, является имя файла, содержащего это сообщение. Если вам не нравится сообщение, введенное пользователем, вы можете либо изменить этот файл на месте (то же самое, что и `prepare-commit-msg`), либо полностью прервать фиксацию, выйдя с ненулевым статусом.

Следующий пример используется для проверки того, присутствует ли в сообщении о фиксации текстовый билет, сопровождаемый номером

```
word="ticket [0-9]"
isPresent=$(grep -Eoh "$word" $1)

if [[ -z $isPresent ]]
  then echo "Commit message KO, $word is missing"; exit 1;
```

```
  else echo "Commit message OK"; exit 0;
fi
```

Локальные крючки

Локальные крючки влияют только на локальные репозитории, в которых они находятся. Каждый разработчик может изменять свои собственные локальные перехватчики, поэтому их нельзя использовать надежно, чтобы обеспечить соблюдение политики фиксации. Они предназначены для того, чтобы разработчикам было легче придерживаться определенных рекомендаций и избегать потенциальных проблем в будущем.

Существует шесть типов локальных перехватов: pre-commit, prepare-commit-msg, commit-msg, post-commit, post-checkout и pre-rebase.

Первые четыре крючка относятся к фиксации и позволяют вам контролировать некоторый элемент в жизненном цикле фиксации. Последние два позволяют выполнять некоторые дополнительные действия или проверки безопасности для команд git checkout и git rebase.

Все «предварительные» крючки позволяют вам изменить действие, которое должно произойти, в то время как «post-» hooks используются в основном для уведомлений.

Пост-контроль

Этот хук работает так же, как и `post-commit` , но он вызывается всякий раз, когда вы успешно проверяете ссылку с помощью `git checkout` . Это может быть полезным инструментом для очистки вашего рабочего каталога автоматически сгенерированных файлов, которые в противном случае могли бы вызвать путаницу.

Этот крючок принимает три параметра:

1. ссылка предыдущей главы,
2. рефери нового HEAD и
3. флаг, указывающий, была ли это проверка филиала или проверка файла (`1` или `0` , соответственно).

Его статус выхода не влияет на команду `git checkout` .

После совершения

Этот крючок вызывается сразу после крючка `commit-msg` . Он не может изменить результат операции `git commit` , поэтому он используется в основном для целей уведомления.

Сценарий не принимает никаких параметров, и его статус выхода никак не влияет на фиксацию.

После приема

Этот крючок вызывается после успешной операции нажатия. Он обычно используется для целей уведомления.

Сценарий не принимает никаких параметров, но отправляется такая же информация, как и `pre-receive` через стандартный ввод:

```
<old-value> <new-value> <ref-name>
```

Предварительно совершить

Этот крючок выполняется каждый раз, когда вы запускаете `git commit` , чтобы проверить, что должно произойти. Вы можете использовать этот крючок для проверки моментального снимка, который должен быть зафиксирован.

Этот тип крючка полезен для запуска автоматических тестов, чтобы убедиться, что входящая фиксация не нарушает существующие функции вашего проекта. Этот тип крюка может также проверять ошибки пробела или EOL.

Никакие аргументы не передаются в сценарий предварительной фиксации, а выход с ненулевым статусом прерывает всю фиксацию.

Приготовьте фиксации-сообщ

Этот крючок вызывается после крючка `pre-commit` для заполнения текстового редактора сообщением фиксации. Обычно это используется для изменения автоматически генерируемых сообщений фиксации для раздавленных или объединенных коммитов.

От одного до трех аргументов передается этот крючок:

- Имя временного файла, содержащего сообщение.
- Тип фиксации, либо
    - сообщение (`-m` или `-F`),
    - шаблон (опция `-t`),
    - merge (если это слияние), или
    - сквош (если он сжимает другие коммиты).
- SHA1 хеш соответствующего коммита. Это дается только в том случае, если `--amend` опция `-c` , `-C` или `--amend` .

Подобно `pre-commit` , выход с ненулевым статусом прерывает фиксацию.

Pre-перебазироваться

Этот hook вызывается до того, как `git rebase` начинает изменять структуру кода. Этот

крючок обычно используется для обеспечения надлежащей операции перезаписи.

Этот крючок принимает 2 параметра:

1. ветвь вверх по течению, из которой серия была раздвоена, и
2. ветвь переустанавливается (пустая при перезагрузке текущей ветви).

Вы можете прервать операцию переадресации, выйдя с ненулевым статусом.

Предварительно получить

Этот крючок выполняется каждый раз, когда кто-то использует `git push` для толкания коммитов в репозиторий. Он всегда находится в удаленном репозитории, который является местом назначения push, а не в исходном (локальном) репозитории.

Захват выполняется до того, как будут обновлены ссылки. Он обычно используется для обеспечения любого вида политики развития.

Скрипт не принимает никаких параметров, но каждый возвращаемый ref передается сценарию в отдельной строке на стандартном входе в следующем формате:

```
<old-value> <new-value> <ref-name>
```

Обновить

Этот крюк вызывается после `pre-receive` , и он работает одинаково. Он вызывается до того, как что-то действительно обновляется, но вызывается отдельно для каждого ref, который был нажат, а не только для всех ссылок.

Этот крючок принимает следующие 3 аргумента:

- имя обновляемого ref,
- старое имя объекта, сохраненное в ref, и
- новое имя объекта, сохраненное в ref.

Это та же самая информация, которая была передана для `pre-receive` , но поскольку `update` вызывается отдельно для каждого ref, вы можете отклонить некоторые ссылки, разрешая другим.

Предварительное давление

Доступно в Git 1.8.2 и выше.

1,8

Предварительно нажимные крюки могут использоваться для предотвращения толчка.

Причины, по которым это полезно, включают в себя: блокирование случайного ручного нажатия на определенные ветви или блокирование нажатий, если установленная проверка завершается с ошибкой (модульные тесты, синтаксис).

Предзапутный крюк создается простым созданием файла с именем `pre-push` под `.git/hooks/` и (**getcha alert**), убедившись, что файл является исполняемым: `chmod +x ./git/hooks/pre-push` .

Вот пример из Ханны Вулф, которая блокирует толчок хозяину:

```
#!/bin/bash

protected_branch='master'
current_branch=$(git symbolic-ref HEAD | sed -e 's,.*/\(.*\),\1,')

if [ $protected_branch = $current_branch ]
then
    read -p "You're about to push master, is that what you intended? [y|n] " -n 1 -r <
/dev/tty
    echo
    if echo $REPLY | grep -E '^[Yy]$' > /dev/null
    then
        exit 0 # push will execute
    fi
    exit 1 # push will not execute
else
    exit 0 # push will execute
fi
```

Вот пример из Volkan Unsal, который гарантирует, что тесты RSpec пройдут, прежде чем разрешить push:

```
#!/usr/bin/env ruby
require 'pty'
html_path = "rspec_results.html"
begin
  PTY.spawn( "rspec spec --format h > rspec_results.html" ) do |stdin, stdout, pid|
  begin
    stdin.each { |line| print line }
  rescue Errno::EIO
  end
end
rescue PTY::ChildExited
  puts "Child process exit!"
end

# find out if there were any errors
html = open(html_path).read
examples = html.match(/(\d+) examples/)[0].to_i rescue 0
errors = html.match(/(\d+) errors/)[0].to_i rescue 0
if errors == 0 then
  errors = html.match(/(\d+) failure/)[0].to_i rescue 0
end
pending = html.match(/(\d+) pending/)[0].to_i rescue 0

if errors.zero?
```

```
  puts "0 failed! #{examples} run, #{pending} pending"
  # HTML Output when tests ran successfully:
  # puts "View spec results at #{File.expand_path(html_path)}"
  sleep 1
  exit 0
else
  puts "\aCOMMIT FAILED!!"
  puts "View your rspec results at #{File.expand_path(html_path)}"
  puts
  puts "#{errors} failed! #{examples} run, #{pending} pending"
  # Open HTML Ooutput when tests failed
  # `open #{html_path}`
  exit 1
end
```

Как вы можете видеть, есть много возможностей, но основная часть - `exit 0` если что-то случилось, и `exit 1` если что-то случилось. Каждый раз, когда вы `exit 1` нажатие будет предотвращено, и ваш код будет находиться в состоянии, которое было до запуска `git push...`

При использовании крючков на стороне клиента имейте в виду, что пользователи могут пропустить все клики на стороне клиента, используя опцию «--no-verify» при нажатии. Если вы полагаетесь на крючок для принудительного выполнения процесса, вы можете получить ожог.

Документация: https://git-scm.com/docs/githooks#_pre_push
Официальный образец: https://github.com/git/git/blob/87c86dd14abe8db7d00b0df5661ef8cf147a72a3/templates/hooks--pre-push.sample

Проверьте сборку Maven (или другую систему сборки) перед фиксацией

`.git/hooks/pre-commit`

```
#!/bin/sh
if [ -s pom.xml ]; then
    echo "Running mvn verify"
    mvn clean verify
    if [ $? -ne 0 ]; then
        echo "Maven build failed"
        exit 1
    fi
fi
```

Автоматически пересылать определенные нажатия на другие репозитории

`post-receive` hooks могут использоваться для автоматической пересылки входящих нажатий в другой репозиторий.

```
$ cat .git/hooks/post-receive
```

```
#!/bin/bash

IFS=' '
while read local_ref local_sha remote_ref remote_sha
do

  echo "$remote_ref" | egrep '^refs\/heads\/[A-Z]+-[0-9]+$' >/dev/null && {
    ref=`echo $remote_ref | sed -e 's/^refs\/heads\///'`
    echo Forwarding feature branch to other repository: $ref
    git push -q --force other_repos $ref
  }

done
```

В этом примере `egrep` regexp ищет определенный формат ветвления (здесь: JIRA-12345, используемый для обозначения проблем Jira). Вы можете оставить эту часть, если хотите переправить все ветки, конечно.

Прочитайте Крючки онлайн: https://riptutorial.com/ru/git/topic/1330/крючки

глава 34: Крючки с клиентской стороны Git

Вступление

Как и многие другие системы управления версиями, Git имеет возможность отключить пользовательские скрипты, когда происходят определенные важные действия. Есть две группы этих крючков: клиентская и серверная. Крюки на стороне клиента запускаются с помощью таких операций, как фиксация и слияние, в то время как перехватчики на стороне сервера работают в сетевых операциях, таких как прием нажатых коммитов. Вы можете использовать эти крючки по разным причинам.

Examples

Установка крючка

Перехватчики все хранятся в подкаталоге `hooks` каталога Git. В большинстве проектов это `.git/hooks` .

Чтобы включить скрипт hook, поместите файл в подкаталог `hooks` вашего каталога `.git` который назван соответствующим образом (без какого-либо расширения) и является исполняемым.

Git для предварительного натяга

pre-push script вызывается `git push` после того, как он проверил удаленный статус, но прежде чем что-либо было нажато. Если этот скрипт выходит с ненулевым статусом, ничего не будет нажато.

Этот крючок вызывается со следующими параметрами:

```
 $1 -- Name of the remote to which the push is being done (Ex: origin)
 $2 -- URL to which the push is being done (Ex:
https://<host>:<port>/<username>/<project_name>.git)
```

Информация о выполненных коммитах предоставляется в виде строк стандартного ввода в форме:

```
<local_ref> <local_sha1> <remote_ref> <remote_sha1>
```

Примеры значений:

```
local_ref = refs/heads/master
local_sha1 = 68a07ee4f6af8271dc40caae6cc23f283122ed11
remote_ref = refs/heads/master
remote_sha1 = efd4d512f34b11e3cf5c12433bbedd4b1532716f
```

Ниже пример сценария предварительного толкования был взят из стандартного pre-push.sample, который был автоматически создан, когда новый репозиторий инициализирован с помощью `git init`

```
# This sample shows how to prevent push of commits where the log message starts
# with "WIP" (work in progress).

remote="$1"
url="$2"

z40=0000000000000000000000000000000000000000

while read local_ref local_sha remote_ref remote_sha
do
    if [ "$local_sha" = $z40 ]
    then
        # Handle delete
        :
    else
        if [ "$remote_sha" = $z40 ]
        then
            # New branch, examine all commits
            range="$local_sha"
        else
            # Update to existing branch, examine new commits
            range="$remote_sha..$local_sha"
        fi

        # Check for WIP commit
        commit=`git rev-list -n 1 --grep '^WIP' "$range"`
        if [ -n "$commit" ]
        then
            echo >&2 "Found WIP commit in $local_ref, not pushing"
            exit 1
        fi
    fi
done

exit 0
```

Прочитайте Крючки с клиентской стороны Git онлайн: https://riptutorial.com/ru/git/topic/8654/крючки-с-клиентской-стороны-git

глава 35: Миграция в Git

Examples

Миграция из SVN в Git с использованием утилиты преобразования Atlassian

Загрузите утилиту конверсии Atlassian здесь . Для этой утилиты требуется Java, поэтому убедитесь, что на компьютере, на котором планируется преобразование, установлена JRE Java Runtime Environment JRE .

Используйте команду `java -jar svn-migration-scripts.jar verify` чтобы проверить, не хватает ли вашей машины какой-либо из программ, необходимых для завершения преобразования. В частности, эта команда проверяет утилиты Git, subversion и `git-svn` . Он также проверяет, что вы выполняете миграцию в зависящей от регистратора файловой системе. Миграция в Git должна выполняться в зависящей от регистратора файловой системе, чтобы не повредить репозиторий.

Затем вам нужно сгенерировать файл авторов. Subversion отслеживает изменения только по имени пользователя коммиттера. Однако Git использует две части информации, чтобы отличить пользователя: настоящее имя и адрес электронной почты. Следующая команда генерирует текстовый файл, сопоставляющий имена пользователей subversion с их эквивалентами Git:

```
java -jar svn-migration-scripts.jar authors <svn-repo> authors.txt
```

где `<svn-repo>` - это URL-адрес репозитория subversion, который вы хотите преобразовать. После выполнения этой команды идентификационная информация участников будет отображаться в `authors.txt` . Адреса электронной почты будут иметь форму `<username>@mycompany.com` . В файле авторов вам необходимо вручную изменить имя пользователя по умолчанию (которое по умолчанию стало их именем пользователя) для их фактических имен. Перед продолжением убедитесь, что все правильные адреса электронной почты проверяются.

Следующая команда будет клонировать репозиторий svn как Git:

```
git svn clone --stdlayout --authors-file=authors.txt <svn-repo> <git-repo-name>
```

где `<svn-repo>` - это тот же самый URL-адрес репозитория, который использовался выше, и `<git-repo-name>` - это имя папки в текущем каталоге для клонирования репозитория. Перед использованием этой команды есть несколько соображений:

- Флаг `--stdlayout` сверху говорит Git, что вы используете стандартную раскладку с

папками для `trunk` , `branches` и `tags` . Репозитории Subversion с нестандартными макетами требуют указания местоположений папки `trunk` линии, любых / всех папок `branch` папки `tags` . Это можно сделать, следуя этому примеру: `git svn clone --trunk=/trunk --branches=/branches --branches=/bugfixes --tags=/tags --authors-file=authors.txt <svn-repo> <git-repo-name>` .

- Эта команда может занять много часов, в зависимости от размера вашего репо.
- Чтобы сократить время преобразования для больших репозиториев, преобразование можно запустить непосредственно на сервере, на котором размещен репозиторий subversion, чтобы устранить сетевые издержки.

`git svn clone` импортирует ветви subversion (и соединительные линии) в качестве удаленных ветвей, включая теги subversion (удаленные ветви с префиксом `tags/`). Чтобы преобразовать их в фактические ветви и теги, выполните следующие команды на машине Linux в том порядке, в котором они предоставляются. После запуска `git branch -a` должен показать правильные имена ветвей, а `git tag -l` должен показать теги репозитория.

```
git for-each-ref refs/remotes/origin/tags | cut -d / -f 5- | grep -v @ | while read tagname;
do git tag $tagname origin/tags/$tagname; git branch -r -d origin/tags/$tagname; done
git for-each-ref refs/remotes | cut -d / -f 4- | grep -v @ | while read branchname; do git
branch "$branchname" "refs/remotes/origin/$branchname"; git branch -r -d "origin/$branchname";
done
```

Преобразование из svn в Git завершено! Просто `push` локальное репо на сервер, и вы можете продолжать вносить свой вклад с помощью Git, а также иметь полностью сохраненную историю версий из svn.

SubGit

SubGit может использоваться для однократного импорта репозитория SVN в git.

```
$ subgit import --non-interactive --svn-url http://svn.my.co/repos/myproject myproject.git
```

Миграция из SVN в Git с использованием svn2git

svn2git - это оболочка Ruby, поддерживающая поддержку SVN git c помощью git-svn , помогая вам переводить проекты из Subversion в Git, сохраняя историю (включая историю стволов, тегов и ветвей).

Примеры

Перенос репозитория svn со стандартным макетом (т. Е. Ветви, теги и соединительные линии на корневом уровне репозитория):

```
$ svn2git http://svn.example.com/path/to/repo
```

Чтобы перенести репозиторий svn, который не находится в стандартном макете:

```
$ svn2git http://svn.example.com/path/to/repo --trunk trunk-dir --tags tags-dir --branches
branches-dir
```

Если вы не хотите переносить (или не иметь) ветви, теги или туловище, вы можете использовать опции `--notrunk` , `--nobranches` и `--notags` .

Например, `$ svn2git http://svn.example.com/path/to/repo --trunk trunk-dir --notags --nobranches` будет переносить только историю стволов.

Чтобы сократить пространство, необходимое вашему новому репозиторию, вы можете исключить любые каталоги или файлы, которые вы когда-то добавляли, пока вы не должны (например, каталог или архивы сборки):

```
$ svn2git http://svn.example.com/path/to/repo --exclude build --exclude '.*\.zip$'
```

Оптимизация после миграции

Если у вас уже есть несколько тысяч коммитов (или больше) в вашем вновь созданном репозитории git, вам может понадобиться сократить пространство, используемое до нажатия вашего репозитория на удаленном компьютере. Это можно сделать, используя следующую команду:

```
$ git gc --aggressive
```

Примечание: предыдущая команда может занимать до нескольких часов на больших репозиториях (десятки тысяч коммитов и / или сотни мегабайт истории).

Перенести из Team Version Version Control (TFVC) в Git

Вы можете перенести из командной строки управления базой в git с помощью инструмента с открытым исходным кодом Git-TF. Миграция также перенесет вашу существующую историю, переведя tfs checkins в git commits.

Чтобы разместить решение в Git с помощью Git-TF, выполните следующие действия:

Скачать Git-TF

Вы можете скачать (и установить) Git-TF из Codeplex: Git-TF @ Codeplex

Клонирование решения TFVC

Запустите powershell (win) и введите команду

```
git-tf clone http://my.tfs.server.address:port/tfs/mycollection
'$/myproject/mybranch/mysolution' --deep
```

Переключатель --deep - это ключевое слово, которое следует отметить, поскольку это

говорит Git-Tf, чтобы скопировать вашу историю проверок. Теперь у вас есть локальный репозиторий git в папке, из которой вы вызвали команду cloe.

уборка

- Добавьте файл .gitignore. Если вы используете Visual Studio, редактор может сделать это для вас, иначе вы можете сделать это вручную, загрузив полный файл из github / gitignore .
- Утилиты управления удалением RemoveTFS из решения (удалите все файлы * . vsssсc). Вы также можете изменить файл решения, удалив GlobalSection (TeamFoundationVersionControl) ... EndClobalSection

Commit & Push

Завершите преобразование, совершив и нажав локальный репозиторий на свой пульт.

```
git add .
git commit -a -m "Coverted solution source control from TFVC to Git"

git remote add origin https://my.remote/project/repo.git

git push origin master
```

Миграция Mercurial в Git

Для импорта `Mercurial` Repo в `Git` можно использовать следующие методы:

1. Использование быстрого экспорта :

```
cd
git clone git://repo.or.cz/fast-export.git
git init git_repo
cd git_repo
~/fast-export/hg-fast-export.sh -r /path/to/old/mercurial_repo
git checkout HEAD
```

2. Использование Hg-Git : очень подробный ответ здесь: https://stackoverflow.com/a/31827990/5283213

3. Использование импортера GitHub : следуйте инструкциям (подробным) в GitHub .

Прочитайте Миграция в Git онлайн: https://riptutorial.com/ru/git/topic/3026/миграция-в-git

глава 36: Нажимать

Вступление

После изменения, постановки и передачи кода с помощью Git, нажатие необходимо сделать ваши изменения доступными для других и переносит локальные изменения на сервер репозитория. В этом разделе рассказывается, как правильно нажать код с помощью Git.

Синтаксис

- git push [-f | --force] [-v | --verbose] [<remote> [<refspec> ...]]

параметры

параметр	подробности
--force	Перезаписывает удаленный ref, чтобы он соответствовал вашему местному справочнику. *Может привести к тому, что удаленный репозиторий потеряет фиксации, поэтому используйте его с осторожностью* .
--подробный	Выполните многословие.
<Пульт>	Удаленный репозиторий, предназначенный для операции push.
<Refspec> ...	Укажите, какой удаленный ref обновить с помощью локального ref или объекта.

замечания

Вверх-вниз

С точки зрения контроля источника, вы находитесь **«вниз по течению»** при копировании (клонирование, проверка и т. Д.) Из репозитория. Информация передавалась вам «вниз по течению».

Когда вы вносите изменения, вы обычно хотите отправить их **«вверх по течению»,** чтобы они попали в этот репозиторий, чтобы все, вытаскивая из одного источника, работали со всеми теми же изменениями. Это в основном

социальная проблема того, как каждый может координировать свою работу, а не технические требования контроля источника. Вы хотите внести свои изменения в основной проект, чтобы не отслеживать разные направления развития.

Иногда вы будете читать о менеджерах пакетов или релизов (люди, а не инструмент), говорящие о внесении изменений в «вверх по течению». Обычно это означает, что они должны были корректировать исходные источники, чтобы они могли создать пакет для своей системы. Они не хотят продолжать делать эти изменения, поэтому, если они отправят «вверх по течению» в исходный источник, им не придется иметь дело с той же проблемой в следующей версии.

(Источник)

Examples

От себя

```
git push
```

будет подталкивать ваш код к существующему выше по течению. В зависимости от конфигурации push он либо выдает код из текущей ветки (по умолчанию в Git 2.x), либо из всех ветвей (по умолчанию в Git 1.x).

Укажите удаленный репозиторий

При работе с git может быть удобно иметь несколько удаленных репозиториев. Чтобы указать удаленный репозиторий для нажатия, просто добавьте его имя в команду.

```
git push origin
```

Укажите ветвь

Чтобы нажать на конкретную ветвь, скажем, `feature_x` :

```
git push origin feature_x
```

Установите ветвь удаленного

отслеживания

Если ветвь, из которой вы работаете, изначально поступает из удаленного репозитория, просто использование `git push` не будет работать в первый раз. Вы должны выполнить следующую команду, чтобы сообщить git, чтобы нажать текущую ветвь на определенную комбинацию удаленного / ветви

```
git push --set-upstream origin master
```

Здесь `master` - это имя ветки в удаленном `origin` . Вы можете использовать `-u` как стенографию для `--set-upstream` .

Нажатие на новый репозиторий

Чтобы нажать на репозиторий, который вы еще не создали, или пусто:

1. Создайте репозиторий на GitHub (если применимо)
2. Скопируйте указанный вами URL-адрес в форме `https://github.com/USERNAME/REPO_NAME.git`
3. Перейдите в локальный репозиторий и выполните `git remote add origin URL`
    - Чтобы проверить, что это было добавлено, запустите `git remote -v`
4. Запустить `git push origin master`

Теперь ваш код должен быть на GitHub

Для получения дополнительной информации Добавление удаленного репозитория

объяснение

Push-код означает, что git проанализирует различия ваших локальных коммитов и удаленных файлов и отправит их для записи по восходящему потоку. Когда нажмите «Успешно», ваш локальный репозиторий и удаленный репозиторий синхронизируются, а другие пользователи могут видеть ваши коммиты.

Более подробную информацию о концепциях «вверх по течению» и «вниз по течению» см. В примечаниях .

Форсирование

Иногда, когда у вас есть локальные изменения, несовместимые с удаленными изменениями

(т. Е. Когда вы не можете переадресовать удаленную ветку, или удаленная ветвь не является прямым предком вашей локальной ветви), единственный способ подтолкнуть ваши изменения - это принудительное нажатие ,

```
git push -f
```

или же

```
git push --force
```

Важные заметки

Это **перезапишет** любые удаленные изменения, и ваш пульт будет соответствовать вашему локальному.

Внимание: использование этой команды может привести к **потере коммитов** удаленного хранилища. Более того, настоятельно рекомендуется отказаться от принудительного нажатия, если вы делитесь этим удаленным репозиторием с другими, поскольку их история сохранит каждый перезаписанный фиксатор, тем самым нарушив их работу от синхронизации с удаленным репозиторием.

Как правило, только принудительное нажатие, когда:

- Никто кроме вас не вытащил изменения, которые вы пытаетесь переписать
- Вы можете заставить всех клонировать новую копию после принудительного толчка и заставить всех применить к ней свои изменения (люди могут вас ненавидеть).

Нажмите конкретный объект на удаленную ветвь

Общий синтаксис

```
git push <remotename> <object>:<remotebranchname>
```

пример

```
git push origin master:wip-yourname
```

`wip-yourname` вашу основную ветвь к ветке происхождения `wip-yourname` (большую часть времени, клонированный вами репозиторий).

Удалить удаленную ветку

Удаление удаленной ветви - это эквивалент нажатия на пустой объект.

```
git push <remotename> :<remotebranchname>
```

пример

```
git push origin :wip-yourname
```

`wip-yourname` удаленную ветку `wip-yourname`

Вместо использования двоеточия вы также можете использовать флаг -delete, который лучше читается в некоторых случаях.

пример

```
git push origin --delete wip-yourname
```

Нажимать одиночную фиксацию

Если в вашей ветке есть одна фиксация, которую вы хотите нажать на пульт, не нажимая ничего, вы можете использовать следующее

```
git push <remotename> <commit SHA>:<remotebranchname>
```

пример

Предполагая, что история git подобна этой

```
eeb32bc Commit 1 - already pushed
347d700 Commit 2 - want to push
e539af8 Commit 3 - only local
5d339db Commit 4 - only local
```

для того чтобы нажать только commit *347d700* на удаленный *мастер,* используйте следующую команду

```
git push origin 347d700:master
```

Изменение поведения push по умолчанию

Current обновляет ветку в удаленном репозитории, который имеет имя с текущей рабочей ветвью.

```
git config push.default current
```

Простое нажатие на ветвь вверх по течению, но не будет работать, если ветвь вверх по течению называется чем-то другим.

```
git config push.default simple
```

Восходящий поток подталкивает к ветке вверх по течению, независимо от того, что она называется.

```
git config push.default upstream
```

Соответствие подталкивает все ветви, которые соответствуют локальному и удаленному git config push.default upstream

После того, как вы установили предпочтительный стиль, используйте

```
git push
```

для обновления удаленного репозитория.

Push-теги

```
git push --tags
```

Выталкивает все `git tags` в локальном репозитории, которые не находятся в удаленном.

Прочитайте Нажимать онлайн: https://riptutorial.com/ru/git/topic/2600/нажимать

глава 37: Название Git Branch на Bash Ubuntu

Вступление

В этой документации рассматривается **название ветки** git на терминале **bash** . Мы, разработчики, должны очень часто находить имя ветки git. Мы можем добавить имя ветки вместе с путём к текущему каталогу.

Examples

Название филиала в терминале

Что такое PS1

PS1 обозначает строку с запросом 1. Это одно из приглашений, доступных в оболочке Linux / UNIX. Когда вы откроете терминал, он отобразит содержимое, определенное в переменной PS1, в вашем приглашении bash. Чтобы добавить название ветки в приглашение bash, нам нужно отредактировать переменную PS1 (установить значение PS1 в ~ / .bash_profile).

Отображать название ветки git

Добавьте следующие строки в файл ~ / .bash_profile.

```
git_branch() {
 git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \(.*\)/ (\1)/'
}
export PS1="\u@\h \[\033[32m\]\w\[\033[33m\]\$(git_branch)\[\033[00m\] $ "
```

Эта функция git_branch найдет имя ветки, в которой мы находимся. Как только мы закончим с этими изменениями, мы сможем переходить на git-репо на терминал и увидим название ветки.

Прочитайте Название Git Branch на Bash Ubuntu онлайн:
https://riptutorial.com/ru/git/topic/8320/название-git-branch-на-bash-ubuntu

глава 38: Обвинять

Синтаксис

- git wame [имя файла]
- git wame [-f] [- e] [- w] [имя файла]
- git винить [-L диапазон] [имя_файла]

параметры

параметр	подробности
имя файла	Имя файла, для которого необходимо проверить данные
-f	Показывать имя файла в исходной фиксации
-e	Показывать письмо автора вместо имени автора.
-w	Игнорировать пробелы при сравнении между дочерней и родительской версиями
-L начать, завершить	Показывать только данный диапазон строк. Пример: `git blame -L 1,2 [filename]`
--show-статистика	Показывает дополнительную статистику в конце вины
-l	Показать длинный оборот (по умолчанию: выключено)
-t	Показывать сырую временную метку (по умолчанию: выключено)
-задний ход	Прогулка вперед вперед, а не назад
-p, --porcelain	Выход для потребления машины
-M	Обнаружение перемещенных или скопированных строк внутри файла
-C	В дополнение к -M, обнаруживают строки, перемещенные или скопированные из других файлов, которые были изменены в одном и том же коммите
-час	Показать справочное сообщение
-c	Используйте тот же режим вывода, что и git-annotate (по умолчанию: off)

параметр	подробности
-n	Показывать номер строки в исходной фиксации (по умолчанию: выключено)

замечания

Команда git blame очень полезна, когда выясняется, кто внес изменения в файл в каждой строке.

Examples

Показать фиксацию, которая в последний раз модифицировала строку

```
git blame <file>
```

покажет файл с каждой строкой, аннотированной коммитом, который в последний раз его модифицировал.

Игнорировать изменения только для пробелов

Иногда у репозиториев будут фиксации, которые регулируют только пробелы, например, фиксируют отступы или переключают между вкладками и пробелами. Это затрудняет поиск фиксации, где был написан код.

```
git blame -w
```

будут игнорировать переменные, содержащие только пробелы, чтобы найти, откуда эта линия действительно появилась.

Показывать только определенные строки

Выход может быть ограничен путем указания диапазонов строк как

```
git blame -L <start>,<end>
```

Где `<start>` и `<end>` могут быть:

- номер строки

  ```
  git blame -L 10,30
  ```

- / Регулярное выражение /

  ```
  git blame -L /void main/ , git blame -L 46,/void foo/
  ```

- + offset, -offset (только для `<end>`)

`git blame -L 108,+30 , git blame -L 215,-15`

Можно указать несколько диапазонов линий, и допустимы перекрывающиеся диапазоны.

```
git blame -L 10,30 -L 12,80 -L 120,+10 -L ^/void main/,+40
```

Чтобы узнать, кто изменил файл

```
// Shows the author and commit per line of specified file
git blame test.c

// Shows the author email and commit per line of specified
git blame -e test.c file

// Limits the selection of lines by specified range
git blame -L 1,10 test.c
```

Прочитайте Обвинять онлайн: https://riptutorial.com/ru/git/topic/3663/обвинять

глава 39: Обновить имя объекта в ссылке

Examples

Обновить имя объекта в ссылке

использование

Обновить имя объекта, которое хранится в ссылке

СИНТАКСИС

```
git update-ref [-m <reason>] (-d <ref> [<oldvalue>] | [--no-deref] [--create-reflog] <ref>
<newvalue> [<oldvalue>] | --stdin [-z])
```

Общий синтаксис

1. Разыменовывая символические ссылки, обновите текущую ветвь ветки до нового объекта.

   ```
   git update-ref HEAD <newvalue>
   ```

2. Сохраняет `newvalue` в `ref` , после проверки того, что текущее значение `ref` соответствует `oldvalue` .

   ```
   git update-ref refs/head/master <newvalue> <oldvalue>
   ```

   выше синтаксиса обновляет головную ветвь филиала до `newvalue` только в том случае, если ее текущее значение является `oldvalue` значением.

Используйте флаг `-d` для удаления именованного `<ref>` после проверки, он все еще содержит `<oldvalue>` .

Используйте `--create-reflog` , update-ref создаст reflog для каждого ref, даже если его обычно не создавали.

Используйте флаг `-z` для указания в формате NUL-terminated, который имеет такие значения, как обновление, создание, удаление, проверка.

Обновить

Установите `<ref>` на `<newvalue>` после проверки `<oldvalue>` , если задано. Задайте нулевое

значение `<newvalue>` чтобы убедиться, что ref не существует после обновления и / или нуля `<oldvalue>` чтобы убедиться, что ref не существует до обновления.

Создайте

Создайте `<ref>` с помощью `<newvalue>` после проверки, что он не существует. Данное значение `<newvalue>` может быть не равным нулю.

удалять

Удалить `<ref>` после проверки, что существует с `<oldvalue>` , если задано. Если задано, `<oldvalue>` может быть не равным нулю.

проверить

Проверьте `<ref>` на `<oldvalue>` но не меняйте его. Если `<oldvalue>` ноль или отсутствует, ref не должен существовать.

Прочитайте Обновить имя объекта в ссылке онлайн: https://riptutorial.com/ru/git/topic/7579/обновить-имя-объекта-в-ссылке

глава 40: Основное хранилище файлов Git (LFS)

замечания

Git Large File Storage (LFS) стремится избежать ограничения системы управления версиями Git, что он плохо работает при версировании больших файлов, особенно в двоичных файлах. LFS решает эту проблему, сохраняя содержимое таких файлов на внешнем сервере, а затем вместо текстового указателя на путь этих активов в базе данных объектов git.

Общие типы файлов, которые хранятся через LFS, как правило, скомпилированы источником; графические активы, такие как PSD и JPEG; или 3D-активы. Таким образом, ресурсы, используемые проектами, могут управляться в одном хранилище, а не поддерживать отдельную систему управления извне.

LFS был первоначально разработан GitHub (https://github.com/blog/1986-announcing-git-large-file-storage-lfs) ; однако, Atlasssian работал над аналогичным проектом почти в то же время, называемый git-lob . Вскоре эти усилия были объединены во избежание фрагментации в отрасли.

Examples

Установка LFS

Загрузите и установите либо через Homebrew, либо с веб-сайта .

Для Brew,

```
brew install git-lfs
git lfs install
```

Часто вам также потребуется выполнить некоторую настройку в службе, на которой размещен ваш пульт, чтобы он работал с lfs. Это будет отличаться для каждого хоста, но, скорее всего, будет проверяться поле, в котором вы хотите использовать git lfs.

Объявлять определенные типы файлов для хранения извне

Обычный рабочий процесс для использования Git LFS заключается в том, чтобы объявить, какие файлы перехватываются через систему на основе правил, так же как `.gitignore` файлы `.gitignore` .

Значительная часть времени, подстановочные знаки используются для выбора

определенных типов файлов в фонотеку.

например, `git lfs track "*.psd"`

Когда файл, соответствующий указанному выше шаблону, будет добавлен, они будут переданы, когда он будет перенесен на удаленный компьютер, он будет загружен отдельно, указатель заменит файл в удаленном репозитории.

После того, как файл будет отслежен с помощью lfs, ваш файл `.gitattributes` будет соответствующим образом обновлен. Github рекомендует `.gitattributes` локальный файл `.gitattributes` , а не работать с глобальным файлом `.gitattributes` , чтобы гарантировать, что у вас нет проблем при работе с разными проектами.

Установите конфигурацию LFS для всех клонов

Чтобы установить параметры LFS, применимые ко всем клонам, создайте и зафиксируйте файл с именем `.lfsconfig` в корне репозитория. Этот файл может указывать параметры LFS так же, как разрешено в `.git/config` .

Например, чтобы исключить определенный файл из `.lfsconfig` LFS по умолчанию, создайте и зафиксируйте `.lfsconfig` со следующим содержимым:

```
[lfs]
    fetchexclude = ReallyBigFile.wav
```

Прочитайте Основное хранилище файлов Git (LFS) онлайн:
https://riptutorial.com/ru/git/topic/4136/основное-хранилище-файлов-git--lfs-

глава 41: Отобразить историю фиксации графически с помощью Gitk

Examples

Отобразить историю фиксации для одного файла

```
gitk path/to/myfile
```

Отображать все фиксации между двумя коммитами

Предположим, у вас есть две коммиты `d9e1db9` и `5651067` и вы хотите посмотреть, что произошло между ними. `d9e1db9` является самым старым предком, а `5651067` - последним потомком в цепочке `5651067` .

```
gitk --ancestry-path d9e1db9 5651067
```

Отображение завершается с тега версии

Если у вас есть тег версии `v2.3` вы можете отображать все фиксации с этого тега.

```
gitk v2.3..
```

Прочитайте Отобразить историю фиксации графически с помощью Gitk онлайн: https://riptutorial.com/ru/git/topic/3637/отобразить-историю-фиксации-графически-с-помощью-gitk

глава 42: перебазировка

Синтаксис

- `git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>] [<upstream>] [<branch>]`
- `git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>] --root [<branch>]`
- `git rebase --continue | --skip | --abort | --edit-todo`

параметры

параметр	подробности
--Продолжить	Перезапустите процесс перезагрузки после разрешения конфликта слияния.
--abort	Отмените операцию перезагрузки и сбросьте HEAD в исходную ветвь. Если ветвь была предоставлена при запуске операции переадресации, то HEAD будет сброшен в ветвь. В противном случае HEAD будет сбрасываться до того места, где была начата операция перезагрузки.
--keep пусто	Держите коммиты, которые ничего не меняют от родителей в результате.
--пропускать	Перезапустите процесс перезагрузки, пропустив текущий патч.
-m, --merge	Используйте стратегии объединения для перерасчета. Когда используется рекурсивная (по умолчанию) стратегия слияния, это позволяет переупаковать, чтобы знать о переименованиях на стороне выше по течению. Обратите внимание, что слияние рефаймов работает, переигрывая каждую фиксацию из рабочей ветви поверх верхней ветви. Из-за этого, когда происходит конфликт слиянием, сторона, о которой сообщается, как наша, является так называемой переустановленной серией, начиная с восходящего потока, а их рабочая ветвь. Другими словами, стороны меняются местами.
--stat	Покажите diffstat того, что изменилось вверх по течению с момента последней перезагрузки. Диффестат также управляется параметром конфигурации rebase.stat.
-x, --exec `command`	Выполнять интерактивную перезагрузку, останавливаясь между каждой `command` фиксации и выполнением `command`

замечания

Имейте в виду, что rebase эффективно перезаписывает историю хранилища.

Повторные фиксации, существующие в удаленном репозитории, могут переписывать узлы репозитория, используемые другими разработчиками в качестве базового узла для их разработки. Если вы действительно не знаете, что делаете, лучше переустановить, прежде чем нажимать ваши изменения.

Examples

Локальная реинжиниринг

Rebasing повторяет серию коммитов поверх другой фиксации.

Чтобы `rebase` ветвь, проверьте ветку, а затем `rebase` ее поверх другой ветви.

```
git checkout topic
git rebase master  # rebase current branch onto master branch
```

Это приведет к:

```
      A---B---C topic
     /
D---E---F---G master
```

Превратиться в:

```
              A'--B'--C' topic
             /
D---E---F---G master
```

Эти операции можно объединить в одну команду, которая проверяет ветвь и немедленно ее переустанавливает:

```
git rebase master topic   # rebase topic branch onto master branch
```

Важно: после переустановки применяемые коммиты будут иметь другой хеш. Вы не должны переустанавливать фиксацию, которую вы уже нажали на удаленный хост. Следствием может быть неспособность `git push` вашу локальную ветвь с переустановкой на удаленный хост, оставив ваш единственный вариант `git push --force` .

Rebase: наши и их, местные и удаленные

Базеба переключает значение «наш» и «их»:

```
git checkout topic
git rebase   master    # rebase topic branch on top of master branch
```

Независимо от того, что говорит HEAD, это «наш»,

Первое, что делает rebase, - это сброс `HEAD` на `master` ; до того, как вишневый сбор перейдет из старой ветвиной `topic` в новую (каждая фиксация в ветке прежней `topic` будет переписана и будет идентифицирована другим хэшем).

Что касается терминологии, используемой средствами слияния (не путать с локальным ref или удаленным ref)

```
=> local is master ("ours"),
=> remote is topic ("theirs")
```

Это означает, что инструмент merge / diff будет представлять ветвь восходящего потока как `local` (`master` : ветвь, поверх которой вы перегружаете), а рабочая ветвь - как `remote` (`topic` : ветвь переустанавливается)

```
+---------------------------------------+
| LOCAL:master |    BASE    | REMOTE:topic |
+---------------------------------------+
|             MERGED                    |
+---------------------------------------+
```

Показана инверсия

При слиянии:

```
c--c--x--x--x(*) <- current branch topic ('*'=HEAD)
    \
     \
      \--y--y--y <- other branch to merge
```

Мы не меняем текущую `topic` ветви, поэтому у нас есть то, над чем мы работали (и мы сливаемся с другой веткой)

```
c--c--x--x--x---------o(*)  MERGE, still on branch topic
    \       ^        /
     \     ours     /
      \             /
       --y--y--y--/
              ^
             theirs
```

На rebase:

Но **при переустановке** мы переключаем стороны, потому что первое, что нужно сделать, - это проверить выходную ветвь, чтобы воспроизвести текущую фиксацию поверх нее!

```
c--c--x--x--x(*) <- current branch topic ('*'=HEAD)
    \
     \
      \--y--y--y <- upstream branch
```

`git rebase upstream` сначала устанавливает `HEAD` в верхнюю ветвь, следовательно, переключатель «ours» и «theirs» по сравнению с предыдущей «текущей» рабочей ветвью.

```
c--c--x--x--x <- former "current" branch, new "theirs"
    \
     \
      \--y--y--y(*) <- set HEAD to this commit, to replay x's on it
                ^       this will be the new "ours"
                |
            upstream
```

После этого rebase повторит «их» фиксацию в новой ветке «нашей» `topic` :

```
c--c..x..x..x <- old "theirs" commits, now "ghosts", available through "reflogs"
    \
     \
      \--y--y--y--x'--x'--x'(*) <- topic  once all x's are replayed,
                ^                       point branch topic to this commit
                |
        upstream branch
```

Интерактивная ребаза

Этот пример предназначен для описания того, как можно использовать `git rebase` в интерактивном режиме. Ожидается, что у вас есть базовое понимание того, что такое `git rebase` и что она делает.

Интерактивная перезагрузка начинается с следующей команды:

```
git rebase -i
```

Опция `-i` относится к *интерактивному режиму* . Используя интерактивную переустановку, пользователь может изменять сообщения фиксации, а также выполнять переупорядочивание, разделение и / или сквош (в сочетании с одним).

Скажите, что вы хотите изменить свои последние три фиксации. Для этого вы можете запустить:

```
git rebase -i HEAD~3
```

После выполнения вышеуказанной инструкции в текстовом редакторе откроется файл, в

котором вы сможете выбрать способ переустановки ваших коммитов. Для целей этого примера просто измените порядок своих коммитов, сохраните файл и закройте редактор. Это приведет к переустановке с заказом, который вы применили. Если вы проверите `git log` вы увидите свои коммиты в новом указанном вами порядке.

Сообщения о фиксации перезаписи

Теперь вы решили, что одно из сообщений фиксации является неопределенным, и вы хотите, чтобы оно было более наглядным. Давайте рассмотрим последние три коммиты, используя ту же команду.

```
git rebase -i HEAD~3
```

Вместо того, чтобы переупорядочить заказ, коммиты будут переустановлены, на этот раз мы изменим `pick` , по умолчанию, чтобы `reword` на коммит, где вы хотите изменить сообщение.

Когда вы закрываете редактор, rebase будет инициироваться, и он остановится на конкретном сообщении о коммитстве, которое вы хотите переписать. Это позволит вам изменить сообщение фиксации в зависимости от того, что вы хотите. После того как вы изменили сообщение, просто закройте редактор, чтобы продолжить.

Изменение содержимого фиксации

Помимо изменения сообщения о фиксации, вы также можете адаптировать изменения, сделанные фиксацией. Для этого просто измените `pick` для `edit` для одного коммита. Git остановится, когда он достигнет этого коммита и предоставит исходные изменения фиксации в промежуточной области. Теперь вы можете адаптировать эти изменения, отказавшись от них или добавив новые изменения.

Как только промежуточная область содержит все изменения, которые вы хотите в этом коммите, зафиксируйте изменения. Старое сообщение фиксации будет показано и может быть адаптировано для отражения нового фиксации.

Разделение одного фиксации на несколько

Предположим, что вы совершили коммит, но решили, что позднее этот фиксатор может быть разделен на две или более коммитов. Используя ту же команду , как и прежде, заменить `pick` с `edit` вместо и нажмите клавишу ВВОД.

Теперь git остановится на фиксации, которую вы отметили для редактирования, и поместите все его содержимое в промежуточную область. С этого момента вы можете запустить `git reset HEAD^` чтобы поместить фиксацию в ваш рабочий каталог. Затем вы можете добавлять и фиксировать свои файлы в другой последовательности - в конечном счете, разделяя одну фиксацию на *n* коммитов вместо этого.

Сжатие нескольких коммитов в один

Скажите, что вы проделали определенную работу и имеете несколько коммитов, которые, по вашему мнению, могут быть единственной фиксацией. Для этого вы можете выполнить `git rebase -i HEAD~3` , заменив `3` соответствующим количеством коммитов.

На этот раз заменить `pick` с `squash` вместо этого. Во время перезагрузки коммит, который вы поручили раздавить, будет раздавлен поверх предыдущего фиксации; вместо этого они превращают их в одну фиксацию.

Прерывание интерактивной ребазы

Вы начали интерактивную перезагрузку. В редакторе, где вы выбираете свои коммиты, вы решаете, что что-то идет не так (например, коммит отсутствует или вы выбрали неправильное назначение переадресации), и вы хотите прервать переустановку.

Для этого просто удалите все коммиты и действия (т.е. все строки, не начинающиеся с знака `#`), и rebase будет прерван!

Текст справки в редакторе действительно дает этот намек:

```
# Rebase 36d15de..612f2f7 onto 36d15de (3 command(s))
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# Note that empty commits are commented out
```

Нажатие после перестановки

Иногда вам нужно переписать историю с rebase, но `git push` жалуется на это, потому что

вы переписали историю.

Это можно решить с помощью `git push --force` , но рассмотрите `git push --force-with-lease` , что указывает на то, что вы хотите, чтобы нажатие было неудачным, если локальная ветвь удаленного отслеживания отличается от ветки на удаленном компьютере, например, кто-то иначе нажмите на удаленный компьютер после последней выборки. Это позволяет избежать непреднамеренного перезаписи чужих недавних нажатий.

Примечание : `git push --force` - и даже `--force-with-lease` в этом отношении - может быть опасной командой, поскольку она перезаписывает историю ветки. Если другой человек вытащил ветку до принудительного толчка, у его / ее приведения `git pull` или `git fetch` будут ошибки, потому что местная история и удаленная история расходятся. Это может привести к непредвиденным ошибкам. При достаточном рассмотрении рефлогов другая работа пользователя может быть восстановлена, но это может привести к большому количеству потраченного впустую времени. Если вы должны принудительно нажать на ветку с другими участниками, попробуйте согласовать с ними, чтобы им не приходилось иметь дело с ошибками.

Перебазируйте до первоначальной фиксации

Так как Git 1.7.12 можно переустановить до корневой фиксации. Корневая фиксация - это первая фиксация, когда-либо сделанная в репозитории, и обычно ее нельзя редактировать. Используйте следующую команду:

```
git rebase -i --root
```

Повторное восстановление перед просмотром кода

Резюме

Эта цель состоит в том, чтобы реорганизовать все ваши разрозненные коммиты в более значимые коммиты для более простых обзоров кода. Если слишком много уровней изменений для слишком большого количества файлов одновременно, сложнее выполнить проверку кода. Если вы можете реорганизовать свои хронологически созданные коммиты в актуальные коммиты, то процесс обзора кода будет проще (и, возможно, меньше ошибок проскальзывает процесс проверки кода).

Этот чрезмерно упрощенный пример - не единственная стратегия использования git для улучшения обзора кода. Это то, как я это делаю, и это то, что вдохновляет других на то, чтобы подумать о том, как сделать обзоры кода и историю гитов легче / лучше.

Это также педагогически демонстрирует силу всеобщей ребалансировки.

В этом примере предполагается, что вы знаете об интерактивной перезагрузке.

Предполагая, что:

- вы работаете над ветвью функции мастера
- ваша функция имеет три основных уровня: front-end, back-end, DB
- вы сделали много коммитов во время работы над ветвью функций. Каждая фиксация затрагивает сразу несколько слоев
- вы хотите (в конце концов) только три фиксации в вашей ветке
    - один, содержащий все изменения переднего конца
    - один, содержащий все задние изменения
    - один, содержащий все изменения БД

Стратегия:

- мы собираемся изменить наши хронологические фиксации на «актуальные» коммиты.
- во-первых, расщепить все коммиты на несколько меньших коммитов - каждый из которых содержит только одну тему за раз (в нашем примере темы - это интерфейс, конец, изменения БД)
- Затем переупорядочивайте наши локальные коммиты вместе и «сквоите» их в отдельные актуальные сообщения

Пример:

```
$ git log --oneline master..
975430b db adding works: db.sql logic.rb
3702650 trying to allow adding todo items: page.html logic.rb
43b075a first draft: page.html and db.sql
$ git rebase -i master
```

Это будет показано в текстовом редакторе:

```
pick 43b075a first draft: page.html and db.sql
pick 3702650 trying to allow adding todo items: page.html logic.rb
pick 975430b db adding works: db.sql logic.rb
```

Измените его так:

```
e 43b075a first draft: page.html and db.sql
e 3702650 trying to allow adding todo items: page.html logic.rb
e 975430b db adding works: db.sql logic.rb
```

Затем git будет применять одно сообщение за раз. После каждой фиксации появится приглашение, а затем вы можете сделать следующее:

```
Stopped at 43b075a92a952faf999e76c4e4d7fa0f44576579... first draft: page.html and db.sql
You can amend the commit now, with

        git commit --amend

Once you are satisfied with your changes, run

        git rebase --continue

$ git status
rebase in progress; onto 4975ae9
You are currently editing a commit while rebasing branch 'feature' on '4975ae9'.
  (use "git commit --amend" to amend the current commit)
  (use "git rebase --continue" once you are satisfied with your changes)

nothing to commit, working directory clean
$ git reset HEAD^ #This 'uncommits' all the changes in this commit.
$ git status -s
 M db.sql
 M page.html
$ git add db.sql  #now we will create the smaller topical commits
$ git commit -m "first draft: db.sql"
$ git add page.html
$ git commit -m "first draft: page.html"
$ git rebase --continue
```

Затем вы повторите эти шаги для каждой фиксации. В итоге у вас есть следующее:

```
$ git log --oneline
0309336 db adding works: logic.rb
06f81c9 db adding works: db.sql
3264de2 adding todo items: page.html
675a02b adding todo items: logic.rb
272c674 first draft: page.html
08c275d first draft: db.sql
```

Теперь мы запускаем rebase еще раз, чтобы изменить порядок и сквош:

```
$ git rebase -i master
```

Это будет показано в текстовом редакторе:

```
pick 08c275d first draft: db.sql
pick 272c674 first draft: page.html
pick 675a02b adding todo items: logic.rb
pick 3264de2 adding todo items: page.html
pick 06f81c9 db adding works: db.sql
pick 0309336 db adding works: logic.rb
```

Измените его так:

```
pick 08c275d first draft: db.sql
s 06f81c9 db adding works: db.sql
pick 675a02b adding todo items: logic.rb
s 0309336 db adding works: logic.rb
pick 272c674 first draft: page.html
s  3264de2 adding todo items: page.html
```

ВНИМАНИЕ: убедитесь, что вы указали git rebase на применение / сквош меньших локальных коммитов *в том порядке, в котором они были хронологически зафиксированы* . В противном случае у вас могут быть ложные, ненужные конфликты слияния.

Когда эта интерактивная перебаза все сказано и сделано, вы получаете следующее:

```
$ git log --oneline master..
74bdd5f adding todos: GUI layer
e8d8f7e adding todos: business logic layer
121c578 adding todos: DB layer
```

резюмировать

Теперь вы уложили свои хронологические фиксации в актуальные дела. В реальной жизни вам может не понадобиться делать это каждый раз, но когда вы этого хотите или хотите сделать, теперь можете. Плюс, надеюсь, вы узнали больше о git rebase.

Настройка git-pull для автоматического выполнения rebase вместо слияния

Если ваша команда выполняет рабочий процесс на основе базы данных, может быть полезно настроить git, чтобы каждая вновь созданная ветвь выполняла операцию переадресации вместо операции слияния во время `git pull` .

Чтобы настроить каждую *новую* ветвь для автоматической переадресации, добавьте следующее в ваш `.gitconfig` или `.git/config` :

```
[branch]
autosetuprebase = always
```

Командная строка: `git config [--global] branch.autosetuprebase always`

В качестве альтернативы вы можете настроить команду `git pull` чтобы всегда вести себя так, как будто была передана опция `--rebase` :

```
[pull]
rebase = true
```

Командная строка: `git config [--global] pull.rebase true`

Тестирование всех коммитов во время rebase

Прежде чем делать запрос на извлечение, полезно убедиться, что компиляция прошла успешно, и тесты проходят для каждой фиксации в ветке. Мы можем сделать это автоматически, используя параметр `-x` .

Например:

```
git rebase -i -x make
```

будет выполнять интерактивную перезагрузку и останавливаться после каждой фиксации для выполнения `make` . В случае сбоя `make` , git остановится, чтобы дать вам возможность исправить проблемы и исправить фиксацию, прежде чем приступать к выбору следующего.

Настройка автозапуска

Autostash - очень полезный параметр конфигурации при использовании rebase для локальных изменений. Зачастую вам может потребоваться внести фиксации из ветки вверх по течению, но еще не готовы к фиксации.

Однако Git не разрешает перезагрузку, если рабочий каталог не является чистым. Autostash на помощь:

```
git config --global rebase.autostash    # one time configuration
git rebase @{u}                         # example rebase on upstream branch
```

Автозагрузка будет применяться всякий раз, когда будет завершена перезагрузка. Не имеет значения, успешно ли завершена перебаза или если она прервана. В любом случае будет применен автозапуск. Если перебаза была успешной, и поэтому базовая фиксация была изменена, тогда может возникнуть конфликт между автосохранением и новыми коммитами. В этом случае вам придется разрешать конфликты перед совершением. Это ничем не отличается от того, если бы вы вручную спрятали, а затем применили, поэтому нет недостатка в том, чтобы делать это автоматически.

Прочитайте перебазировка онлайн: https://riptutorial.com/ru/git/topic/355/перебазировка

глава 43: Переименование

Синтаксис

- `git mv <source> <destination>`
- `git mv -f <source> <destination>`

параметры

параметр	подробности
`-f` или `--force`	Принудительное переименование или перемещение файла, даже если цель существует

Examples

Переименование папок

Чтобы переименовать папку из `oldName` в `newName`

```
git mv directoryToFolder/oldName directoryToFolder/newName
```

Далее следует `git commit` и / или `git push`

Если эта ошибка возникает:

> fatal: переименовано 'directoryToFolder / oldName' failed: Недействительный аргумент

Используйте следующую команду:

```
git mv directoryToFolder/oldName temp && git mv temp directoryToFolder/newName
```

Переименование локальной ветви

Вы можете переименовать ветвь в локальном репозитории, используя эту команду:

```
git branch -m old_name new_name
```

переименовать локальную и удаленную ветку

самый простой способ - проверить локальную ветку:

```
git checkout old_branch
```

затем переименуйте локальную ветвь, удалите старый пульт и установите новую переименованную ветвь как вверху:

```
git branch -m new_branch
git push origin :old_branch
git push --set-upstream origin new_branch
```

Прочитайте Переименование онлайн: https://riptutorial.com/ru/git/topic/1814/переименование

глава 44: Поддеревья

Синтаксис

- `git subtree add -P <prefix> <commit>`
- `git subtree add -P <prefix> <repository> <ref>`
- `git subtree pull -P <prefix> <repository> <ref>`
- `git subtree push -P <prefix> <repository> <ref>`
- `git subtree merge -P <prefix> <commit>`
- `git subtree split -P <prefix> [OPTIONS] [<commit>]`

замечания

Это альтернатива использованию `submodule`

Examples

Создание, вытягивание и подтип Backport

Создать субтитры

Добавьте новый удаленный `plugin` указывающий на репозиторий плагина:

```
git remote add plugin https://path.to/remotes/plugin.git
```

Затем создайте поддерево, определяющее новые `plugins/demo` префикса папки. `plugin` - это удаленное имя, а `master` ссылается на главную ветку в репозитории поддерева:

```
git subtree add --prefix=plugins/demo plugin master
```

Обновления субтитров

Вытяните обычные фиксации, сделанные в плагине:

```
git subtree pull --prefix=plugins/demo plugin master
```

Обновления подкаталогов Backport

1. Укажите фиксации, сделанные в суперпроекте для резервного копирования:

```
git commit -am "new changes to be backported"
```

2. Оформить новую ветку для слияния, установить для отслеживания репозитория поддерева:

```
git checkout -b backport plugin/master
```

3. Вишневый выбор:

```
git cherry-pick -x --strategy=subtree master
```

4. Направьте изменения обратно на источник плагина:

```
git push plugin backport:master
```

Прочитайте Поддеревья онлайн: https://riptutorial.com/ru/git/topic/1634/поддеревья

глава 45: подмодули

Examples

Добавление подмодуля

Вы можете включить другой репозиторий Git в качестве папки в вашем проекте, отслеживаемой Git:

```
$ git submodule add https://github.com/jquery/jquery.git
```

Вы должны добавить и зафиксировать новый файл `.gitmodules` ; это говорит Git, какие подмодули должны быть клонированы при запуске `git submodule update` .

Клонирование хранилища Git с подмодулями

Когда вы клонируете репозиторий, который использует подмодули, вам нужно будет инициализировать и обновить их.

```
$ git clone --recursive https://github.com/username/repo.git
```

Это будет клонировать упомянутые подмодули и помещать их в соответствующие папки (включая подмодули внутри подмодулей). Это эквивалентно запуску `git submodule update --init --recursive` сразу же после завершения клонирования.

Обновление подмодуля

Подмодуль ссылается на конкретную фиксацию в другом репозитории. Чтобы проверить точное состояние, на которое ссылаются все подмодули, запустите

```
git submodule update --recursive
```

Иногда вместо использования состояния, на которое вы ссылаетесь, вы хотите обновить локальную проверку до последнего состояния этого субмодуля на удаленном компьютере. Чтобы проверить все подмодули в последнем состоянии на пульте дистанционного управления с помощью одной команды, вы можете использовать

```
git submodule foreach git pull <remote> <branch>
```

или используйте аргументы `git pull` по умолчанию

```
git submodule foreach git pull
```

Обратите внимание, что это просто обновит вашу локальную рабочую копию. Запуск `git status` будет отображать каталог подмодулей как грязный, если он изменится из-за этой команды. Чтобы обновить репозиторий вместо ссылки на новое состояние, вы должны зафиксировать изменения:

```
git add <submodule_directory>
git commit
```

Могут быть некоторые изменения, которые у вас есть, которые могут иметь конфликт слияния, если вы используете `git pull` чтобы вы могли использовать `git pull --rebase` для перемотки ваших изменений вверх, большую часть времени это уменьшает вероятность конфликта. Также он тянет все ветви на локальные.

```
git submodule foreach git pull --rebase
```

Чтобы проверить последнее состояние конкретного подмодуля, вы можете использовать:

```
git submodule update --remote <submodule_directory>
```

Установка подмодуля для следования ветви

Подмодуль всегда извлекается при определенном коммите SHA1 («gitlink», специальная запись в индексе родительского репо)

Но можно запросить обновление этого подмодуля до последней фиксации ветви субмодуля удаленного репо.

Вместо того, чтобы идти в каждом подмодуле, выполняя `git checkout abranch --track origin/abranch, git pull` , вы можете просто сделать (из родительского репо) а:

```
git submodule update --remote --recursive
```

Поскольку SHA1 подмодуля изменится, вам все равно нужно следовать этому:

```
git add .
git commit -m "update submodules"
```

Это предполагает, что подмодули были:

- либо добавлено с веткой, чтобы следовать:

  ```
  git submodule -b abranch -- /url/of/submodule/repo
  ```

- или настроить (для существующего подмодуля), чтобы следовать за веткой:

```
cd /path/to/parent/repo
git config -f .gitmodules submodule.asubmodule.branch abranch
```

Удаление подмодуля

1,8

Вы можете удалить подмодуль (например, `the_submodule`), вызвав:

```
$ git submodule deinit the_submodule
$ git rm the_submodule
```

- `git submodule deinit the_submodule` удаляет `the_submodule` s 'из .git / config. Это исключает the_submodule из `git submodule update` , `git submodule sync` и `git submodule foreach` вызовов и удаляет его локальный контент (источник) . Кроме того, это не будет отображаться как изменение в родительском репозитории. `git submodule init` и `git submodule update` восстановит подмодуль, опять же без изменений в родительском репозитории.

- `git rm the_submodule` удалит подмодуль из дерева работ. Файлы исчезнут, а также запись подмодулей в файле `.gitmodules` (источник) . Если только `git rm the_submodule` (без предварительного `git submodule deinit the_submodule` выполняется, однако, запись подмодулей в вашем файле .git / config останется.

1,8

Взято отсюда :

1. Удалите соответствующий раздел из файла `.gitmodules` .
2. Этап изменения `.gitmodules` `git add .gitmodules`
3. Удалите соответствующий раздел из `.git/config` .
4. Запустить `git rm --cached path_to_submodule` (без косой черты).
5. Запустить `rm -rf .git/modules/path_to_submodule`
6. Commit `git commit -m "Removed submodule <name>"`
7. Удалите теперь необработанные файлы подмодулей
8. `rm -rf path_to_submodule`

Перемещение подмодуля

1,8

Бежать:

```
$ git mv old/path/to/module new/path/to/module
```

1,8

1. Измените .gitmodules и соответствующим образом измените путь подмодуля и поместите его в

индекс с помощью git add .gitmodules .

2. При необходимости создайте родительский каталог нового местоположения подмодуля (mkdir -p *new/path/to*).
3. Переместите весь контент из старого в новый каталог (mv -vi *old/path/to/module* *new/path/to/submodule*).
4. Убедитесь, что Git отслеживает этот каталог (git add *new/path* /to).
5. Удалите старый каталог с git rm --cached *old/path/to/module* .
6. Переместите каталог .git/modules/ *old/path/to/module* со всем содержимым в .git/modules/ *new/path/to/module* .
7. Отредактируйте файл .git/modules/ *new/path/to* /config , убедитесь, что элемент worktree указывает на новые местоположения, поэтому в этом примере это должно быть worktree = ../../../../../ *old/path/to/module* . Как правило, должно быть еще два .. затем каталоги на прямом пути в этом месте. , Отредактируйте файл *new/path/to/module* /.git , убедитесь, что путь в нем указывает на правильное новое местоположение внутри основной папки проекта .git , поэтому в этом примере gitdir: ../../../.git/modules/ *new/path/to/module* .

   Выход git status выглядит следующим образом:

   ```
   # On branch master
   # Changes to be committed:
   #   (use "git reset HEAD <file>..." to unstage)
   #
   #       modified:   .gitmodules
   #       renamed:    old/path/to/submodule -> new/path/to/submodule
   #
   ```

8. Наконец, зафиксируйте изменения.

Этот пример из Stack Overflow , Axel Beckert

Прочитайте подмодули онлайн: https://riptutorial.com/ru/git/topic/306/подмодули

Синтаксис

- `git stash list [<options>]`
  - `git stash show [<stash>]`
  - `git stash drop [-q|--quiet] [<stash>]`
  - `git stash ( pop | apply ) [--index] [-q|--quiet] [<stash>]`
  - `git stash branch <branchname> [<stash>]`
  - `git stash [save [-p|--patch] [-k|--[no-]keep-index] [-q|--quiet] [-u|--include-untracked] [-a|--all] [<message>]]`
  - `git stash clear`
  - `git stash create [<message>]`
  - `git stash store [-m|--message <message>] [-q|--quiet] <commit>`

параметры

параметр	подробности
шоу	Покажите изменения, записанные в кошельке, как разницу между сохраненным состоянием и исходным родителем. Когда нет <stash>, отображается последний.
список	Перечислите сохраненные в данный момент штампы. Каждый тайник указан с его именем (например, stash @ {0} - это последний тайник, stash @ {1} - тот, который был ранее и т. Д.), Имя ветки, которая была текущей, когда был сделан тайник, и короткий описание фиксации основы.
поп	Удалите из списка закладок одно спрятанное состояние и примените его поверх текущего рабочего дерева.
применять	Подобно `pop` , но не удаляйте состояние из списка.
Чисто	Удалите все сжатые состояния. Обратите внимание, что эти состояния будут подвергнуты обрезке и могут быть невозможны для восстановления.
падение	Удалите из списка закладок одно спрятанное состояние. Когда нет < stash>, он удаляет последнюю. т.е. stash @ {0}, в противном случае < stash> должна быть действительной ссылкой журнала пробелов формы stash @ {<revision>}.
Создайте	Создайте stash (который является обычным объектом фиксации) и верните его имя объекта, не сохраняя его нигде в пространстве имен ref. Это предназначено для использования в скриптах. Это, вероятно, не та команда, которую вы хотите использовать; см. «сохранить» выше.
хранить	Храните заданный кошелек, созданный с помощью git stash create (который является обманывающим слиянием) в stash ref, обновляя stash

параметр	подробности
	reflog. Это предназначено для использования в скриптах. Это, вероятно, не та команда, которую вы хотите использовать; см. «сохранить» выше.

замечания

Stashing позволяет нам иметь чистый рабочий каталог без потери информации. Затем можно начать работать над чем-то другим и / или переключать ветви.

Examples

Что такое Stashing?

Когда вы работаете над проектом, вы можете быть на полпути через изменение ветки функции, когда ошибка связана с мастером. Вы не готовы совершать свой код, но также не хотите терять свои изменения. Именно здесь пригодится git stash .

Запустите git status на ветке, чтобы показать свои незафиксированные изменения:

```
(master) $ git status
On branch master
Your branch is up-to-date with 'origin/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:   business/com/test/core/actions/Photo.c

no changes added to commit (use "git add" and/or "git commit -a")
```

Затем запустите git stash чтобы сохранить эти изменения в стек:

```
(master) $ git stash
Saved working directory and index state WIP on master:
2f2a6e1 Merge pull request #1 from test/test-branch
HEAD is now at 2f2a6e1 Merge pull request #1 from test/test-branch
```

Если вы добавили файлы в рабочий каталог, они также могут быть спрятаны. Вам просто нужно сначала сдать их.

```
(master) $ git stash
Saved working directory and index state WIP on master:
(master) $ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        NewPhoto.c

nothing added to commit but untracked files present (use "git add" to track)
(master) $ git stage NewPhoto.c
(master) $ git stash
Saved working directory and index state WIP on master:
(master) $ git status
On branch master
nothing to commit, working tree clean
(master) $
```

В вашем рабочем каталоге теперь нет никаких изменений. Вы можете увидеть это, повторно запустив git status :

```
(master) $ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Чтобы применить последний тайник, запустите git stash apply (кроме того, вы можете применить *и* удалить последнее спрятанное измененное с помощью git stash pop):

```
(master) $ git stash apply
On branch master
Your branch is up-to-date with 'origin/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:   business/com/test/core/actions/Photo.c

no changes added to commit (use "git add" and/or "git commit -a")
```

Обратите внимание, однако, что удержание не помнит ветку, над которой вы работали. В приведенных выше примерах пользователь нажимал на **master** . Если они переключаются на ветвь **dev** , **dev** и запускают git stash apply последний тайник помещается в ветвь **dev** .

```
(master) $ git checkout -b dev
Switched to a new branch 'dev'
(dev) $ git stash apply
On branch dev
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:   business/com/test/core/actions/Photo.c

no changes added to commit (use "git add" and/or "git commit -a")
```

Создать кошелек

Сохраните текущее состояние рабочего каталога и индекс (также известный как промежуточная область) в стеке stashes.

```
git stash
```

Чтобы включить все незатребованные файлы в приложении, используйте флаги --include-untracked или -u .

```
git stash --include-untracked
```

Чтобы включить сообщение с вашим кошельком, чтобы сделать его более легко идентифицируемым позже

```
git stash save "<whatever message>"
```

Чтобы оставить промежуточную область в текущем состоянии после сохранения, используйте флаги --keep-index или -k .

```
git stash --keep-index
```

Список сохраненных штампов

```
git stash list
```

В этом списке будут перечислены все стопки в стеке в обратном хронологическом порядке.
Вы получите список, который выглядит примерно так:

```
stash@{0}: WIP on master: 67a4e01 Merge tests into develop
stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Вы можете обратиться к определенному типу по его имени, например stash@{1} .

Показать кошелек

Показывает изменения, сохраненные в последнем тайме

```
git stash show
```

Или конкретный тираж

```
git stash show stash@{n}
```

Чтобы показать содержимое изменений, сохраненных для конкретного заклада

```
git stash show -p stash@{n}
```

Снять сальник

Удалите все тайники

```
git stash clear
```

Удаляет последний тайник

```
git stash drop
```

Или конкретный тираж

```
git stash drop stash@{n}
```

Применить и удалить прикрытие

Чтобы применить последний тайник и удалить его из типа стека:

```
git stash pop
```

Чтобы применить конкретный штамп и удалить его из типа стека:

```
git stash pop stash@{n}
```

Применить штамп, не удаляя его

Применяет последний тайник, не удаляя его из стека

```
git stash apply
```

Или конкретный тираж

```
git stash apply stash@{n}
```

Восстановление ранее внесенных изменений

Чтобы получить последний тайник после запуска git stash, используйте

```
 git stash apply
```

Чтобы просмотреть список ваших кошельков, используйте

```
 git stash list
```

Вы получите список, который выглядит примерно так:

```
stash@{0}: WIP on master: 67a4e01 Merge tests into develop
stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Выберите другой git stash для восстановления с номером, который отображается для кошелька, который вы хотите

```
git stash apply stash@{2}
```

Частичный задел

Если вы хотите сохранить только *некоторые* отличия в вашем рабочем наборе, вы можете использовать частичный тираж.

```
git stash -p
```

А затем в интерактивном режиме выберите, какие кучки застревать.

Начиная с версии 2.13.0 вы также можете избежать интерактивного режима и создать частичный stash с помощью pathspec с использованием нового ключевого слова **push** .

```
git stash push -m "My partial stash" -- app.config
```

Применить часть кошелька с выпиской

Вы сделали кошелек и хотите проверить только некоторые файлы в этом тире.

```
git checkout stash@{0} -- myfile.txt
```

Интерактивное скрепление

Stashing берет грязное состояние вашего рабочего каталога - то есть ваши измененные отслеживаемые файлы и поэтапные изменения - и сохраняет его в стеке незавершенных изменений, которые вы можете повторно применить в любое время.

Удаление только измененных файлов:

Предположим, вы не хотите зашивать поэтапные файлы и только хранить измененные файлы, чтобы вы могли использовать:

```
git stash --keep-index
```

Это будет содержать только измененные файлы.

Сохранение невоспроизводимых файлов:

Stash никогда не сохраняет неиспользуемые файлы, а только сохраняет измененные и поэтапные файлы. Предположим, что если вам нужно также спрятать файлы без следа, вы можете использовать это:

```
git stash -u
```

это будет отслеживать не проверенные, поставленные и измененные файлы.

Только некоторые изменения:

Предположим, вам нужно занести только часть кода из файла или только некоторые файлы только из всех измененных и спрятанных файлов, тогда вы можете сделать это следующим образом:

```
git stash --patch
```

Git не будет хранить все, что было изменено, но вместо этого предложит вам интерактивно, какие из изменений вы хотели бы сохранить и которые вы хотели бы сохранить в своем рабочем каталоге.

Переместите свою работу в другую ветку

Если во время работы вы осознаете, что находитесь в неправильной ветке, и вы еще не создали никаких коммитов, вы можете легко переместить свою работу, чтобы исправить ветку, используя stashing:

```
git stash
git checkout correct-branch
git stash pop
```

Помните, что git stash pop применит последний тайник и удалит его из списка. Чтобы сохранить список в списке и примениться только к какой-либо отрасли, вы можете использовать:

```
git stash apply
```

Восстановить упавший штемпель

Если вы только что вытащили его и терминал все еще открыт, вы все равно будете иметь значение хэша, напечатанное git stash pop на экране:

```
$ git stash pop
[...]
Dropped refs/stash@{0} (2ca03e22256be97f9e40f08e6d6773c7d41dbfd1)
```

(Обратите внимание, что git stash drop также создает одну и ту же строку.)

В противном случае вы можете найти это, используя это:

```
git fsck --no-reflog | awk '/dangling commit/ {print $3}'
```

Это покажет вам все коммиты на кончиках вашего графика фиксации, которые больше не ссылаются ни

на одну ветку, ни на тег – каждая потерянная фиксация, включая все сделанные вами котировки, будет где-то на этом графике.

Самый простой способ найти нужное вам сообщение – вероятно, передать этот список в gitk :

```
gitk --all $( git fsck --no-reflog | awk '/dangling commit/ {print $3}' )
```

Это запустит браузер репозитория, который покажет вам *каждую фиксацию в репозитории* , независимо от того, достигнут он или нет.

Вы можете заменить gitk там чем-то вроде git log --graph --oneline --decorate если вы предпочитаете хороший граф на консоли в отдельном графическом приложении.

Чтобы выявить фиксации закладок, найдите сообщения фиксации этой формы:

WIP на *somebranch* : *commithash Некоторое старое сообщение фиксации*

Как только вы узнаете хэш коммита, который вы хотите, вы можете применить его как кэш:

```
git stash apply $stash_hash
```

Или вы можете использовать контекстное меню в gitk для создания ветвей для любых недостижимых коммитов, которые вас интересуют. После этого вы можете делать с ними все, что хотите, со всеми обычными инструментами. Когда вы закончите, просто удалите эти ветви снова.

Прочитайте припрятать онлайн: https://riptutorial.com/ru/git/topic/1440/припрятать

глава 47: Просмотр истории

Синтаксис

- `git log [параметры] [диапазон версий] [[-] путь ...]`

параметры

параметр	объяснение
-q, --quiet	Тихий, подавляет diff-выход
--источник	Показывает источник фиксации
--use-mailmap	Использовать файл карты почты (изменяет информацию пользователя для пользователя)
--decorate [= ...]	Украсить варианты
--L <n, m: файл>	Показать журнал для определенного диапазона строк в файле, считая с 1. Начинает с строки n, переходит в строку m. Также показывает diff.
--show подпись	Отображать подписи подписанных коммитов
-i, --regexp-ignore-case	Сопоставьте шаблоны ограничения регулярного выражения без учета буквы

замечания

Ссылки и современная **документация** : официальная документация git-log

Examples

«Обычный» Git Log

```
git log
```

будет отображать все ваши коммиты с автором и хешем. Это будет показано в нескольких строках за фиксацию. (Если вы хотите показать одну строку за фиксацию, посмотрите на onelineing). Используйте клавишу q для выхода из журнала.

> По умолчанию, без аргументов, git-журнал перечисляет коммиты, сделанные в этом репозитории в обратном хронологическом порядке, то есть сначала появляются первые коммиты. Как вы можете видеть, эта команда перечисляет каждую фиксацию с ее контрольной суммой SHA-1, именем и адресом автора, датой и сообщением фиксации. - **источник**

Пример (из репозитория **Free Code Camp**):

```
commit 87ef97f59e2a2f4dc425982f76f14a57d0900bcf
Merge: e50ff0d eb8b729
Author: Brian <sludge256@users.noreply.github.com>
Date:   Thu Mar 24 15:52:07 2016 -0700

    Merge pull request #7724 from BKinahan/fix/where-art-thou

    Fix 'its' typo in Where Art Thou description

commit eb8b7298d516ea20a4aadb9797c7b6fd5af27ea5
Author: BKinahan <b.kinahan@gmail.com>
Date:   Thu Mar 24 21:11:36 2016 +0000

    Fix 'its' typo in Where Art Thou description

commit e50ff0d249705f41f55cd435f317dcfd02590ee7
Merge: 6b01875 2652d04
Author: Mrugesh Mohapatra <raisedadead@users.noreply.github.com>
Date:   Thu Mar 24 14:26:04 2016 +0530

    Merge pull request #7718 from deathsythe47/fix/unnecessary-comma

    Remove unnecessary comma from CONTRIBUTING.md
```

Если вы хотите ограничить свою команду, чтобы последний журнал совершил n вы можете просто передать параметр. Например, если вы хотите перечислить последние 2 фиксированных лога

```
git log -2
```

Журнал Oneline

```
git log --oneline
```

покажет все ваши коммиты только с первой частью хэша и сообщением фиксации. Каждая фиксация будет находиться в одной строке, как oneline флаг oneline .

> Опция oneline печатает каждую фиксацию в одной строке, что полезно, если вы смотрите на множество коммитов. - **источник**

Пример (из репозитория **Free Code Camp** , с тем же разделом кода из другого примера):

```
87ef97f Merge pull request #7724 from BKinahan/fix/where-art-thou
eb8b729 Fix 'its' typo in Where Art Thou description
e50ff0d Merge pull request #7718 from deathsythe47/fix/unnecessary-comma
2652d04 Remove unnecessary comma from CONTRIBUTING.md
6b01875 Merge pull request #7667 from zerkms/patch-1
766f088 Fixed assignment operator terminology
d1e2468 Merge pull request #7690 from BKinahan/fix/unsubscribe-crash
bed9de2 Merge pull request #7657 from Rafase282/fix/
```

Если вы хотите ограничить прикажете до последнего n совершает журнал вы можете просто передать параметр. Например, если вы хотите перечислить последние 2 фиксированных лога

```
git log -2 --oneline
```

Более подробный журнал

Чтобы увидеть журнал в более красивой графитовой структуре, используйте:

```
git log --decorate --oneline --graph
```

выход образца:

```
* e0c1cea (HEAD -> maint, tag: v2.9.3, origin/maint) Git 2.9.3
*   9b601ea Merge branch 'jk/difftool-in-subdir' into maint
|\
| * 32b8c58 difftool: use Git::* functions instead of passing around state
| * 98f917e difftool: avoid $GIT_DIR and $GIT_WORK_TREE
| * 9ec26e7 difftool: fix argument handling in subdirs
* |   f4fd627 Merge branch 'jk/reset-ident-time-per-commit' into maint
...
```

Поскольку это довольно большая команда, вы можете назначить псевдоним:

```
git config --global alias.lol "log --decorate --oneline --graph"
```

Чтобы использовать версию псевдонима:

```
# history of current branch :
git lol

# combined history of active branch (HEAD), develop and origin/master branches :
git lol HEAD develop origin/master

# combined history of everything in your repo :
git lol --all
```

Вход с изменениями inline

Чтобы просмотреть журнал с изменениями в --patch , используйте параметры -p или --patch .

```
git log --patch
```

Пример (из репозитория Trello Scientist)

```
ommit 8ea1452aca481a837d9504f1b2c77ad013367d25
Author: Raymond Chou <info@raychou.io>
Date:   Wed Mar 2 10:35:25 2016 -0800

    fix readme error link

diff --git a/README.md b/README.md
index 1120a00..9bef0ce 100644
--- a/README.md
+++ b/README.md
@@ -134,7 +134,7 @@ the control function threw, but *after* testing the other functions and
readying
 the logging. The criteria for matching errors is based on the constructor and
 message.

-You can find this full example at [examples/errors.js](examples/error.js).
+You can find this full example at [examples/errors.js](examples/errors.js).

 ## Asynchronous behaviors


commit d3178a22716cc35b6a2bdd679a7ec24bc8c63ffa
```

```
:
```

Поиск в журнале

```
git log -S"#define SAMPLES"
```

Поиски **добавления** или **удаления** конкретной строки или **соответствия** строки предоставленной RegExp. В этом случае мы ищем добавление / удаление строки #define SAMPLES . Например:

```
+#define SAMPLES  100000
```

или же

```
-#define SAMPLES  100000
```

```
git log -G"#define SAMPLES"
```

Ищет **изменения** в **строках, содержащих** определенную строку или **сопоставление** строк, предоставляемое REGEXP. Например:

```
-#define SAMPLES  100000
+#define SAMPLES  100000000
```

Список всех вкладов, сгруппированных по имени автора

git shortlog суммирует git log и группы по автору

Если параметры не заданы, список всех коммитов, сделанных на коммиттер, будет отображаться в хронологическом порядке.

```
$ git shortlog
Committer 1 (<number_of_commits>):
    Commit Message 1
    Commit Message 2
    ...
Committer 2 (<number_of_commits>):
    Commit Message 1
    Commit Message 2
    ...
```

Чтобы просто увидеть количество коммитов и подавить описание фиксации, перейдите к сводной опции:

-s

--summary

```
$ git shortlog -s
<number_of_commits> Committer 1
<number_of_commits> Committer 2
```

Чтобы отсортировать вывод по количеству коммитов вместо алфавитного имени коммиттера, перейдите в нумерованную опцию:

-n

--numbered

Чтобы добавить письмо коммиттера, добавьте его по электронной почте:

-e

--email

Опция настраиваемого формата также может быть предоставлена, если вы хотите отображать информацию, отличную от объекта фиксации:

--format

Это может быть любая строка, принятая с помощью опции --format git log .

Для **получения** дополнительной информации см. **Раздел «Раскрашивание журналов»** выше.

Журналы фильтров

```
git log --after '3 days ago'
```

Конкретные даты тоже работают:

```
git log --after 2016-05-01
```

Как и в случае с другими командами и флагами, которые принимают параметр даты, формат разрешенных дат поддерживается датой GNU (очень гибкий).

Псевдоним - --after - --since .

Флаги существуют и для обратного: - --before и - --until .

Вы также можете фильтровать журналы по author . например

```
git log --author=author
```

Войдите в диапазон строк внутри файла

```
$ git log -L 1,20:index.html
commit 6a57fde739de66293231f6204cbd8b2feca3a869
Author: John Doe <john@doe.com>
Date:   Tue Mar 22 16:33:42 2016 -0500

    commit message

diff --git a/index.html b/index.html
--- a/index.html
+++ b/index.html
@@ -1,17 +1,20 @@
 <!DOCTYPE HTML>
 <html>
-       <head>
-        <meta charset="utf-8">
+
+<head>
+    <meta charset="utf-8">
     <meta http-equiv="X-UA-Compatible" content="IE=edge">
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1">
```

Расписывать журналы

```
git log --graph --pretty=format:'%C(red)%h%Creset -%C(yellow)%d%Creset %s %C(green)(%cr)
%C(yellow)<%an>%Creset'
```

Опция format позволяет указать собственный формат вывода журнала:

параметр	подробности
%C(color_name)	опция цветов выводит результат
%h или% H	abbreviates commit hash (используйте% H для полного хэша)
%Creset	сбрасывает цвет до цвета терминала по умолчанию
%d	имена ссылок
%s	subject [commit message]
%cr	дата коммиттера, относительно текущей даты
%an	имя автора

Одна строка, показывающая имя и время коммиттера с момента фиксации

```
tree = log --oneline --decorate --source --pretty=format:'"%Cblue %h %Cgreen %ar %Cblue %an
%C(yellow) %d %Creset %s"'  --all --graph
```

пример

```
*    40554ac  3 months ago  Alexander Zolotov    Merge pull request #95 from
gmandnepr/external_plugins
|\
| *  e509f61  3 months ago  Ievgen Degtiarenko    Documenting new property
| *  46d4cb6  3 months ago  Ievgen Degtiarenko    Running idea with external plugins
| *  6253da4  3 months ago  Ievgen Degtiarenko    Resolve external plugin classes
| *  9fdb4e7  3 months ago  Ievgen Degtiarenko    Keep original artifact name as this may be
important for intellij
| *  22e82e4  3 months ago  Ievgen Degtiarenko    Declaring external plugin in intellij
section
|/
*  bc3d2cb  3 months ago  Alexander Zolotov    Ignore DTD in plugin.xml
```

Git Log между двумя ветвями

git log master..foo покажет коммиты, которые находятся на foo а не на master . Полезно узнать, что коммиты вы добавили после разветвления!

Журнал, показывающий зарегистрированные файлы

```
git log --stat
```

Пример:

```
commit 4ded994d7fc501451fa6e233361887a2365b91d1
Author: Manassés Souza <manasses.inatel@gmail.com>
Date:   Mon Jun 6 21:32:30 2016 -0300

    MercadoLibre java-sdk dependency

 mltracking-poc/.gitignore |  1 +
 mltracking-poc/pom.xml    | 14 ++++++++++++--
 2 files changed, 13 insertions(+), 2 deletions(-)

commit 506fff56190f75bc051248770fb0bcd976e3f9a5
Author: Manassés Souza <manasses.inatel@gmail.com>
Date:   Sat Jun 4 12:35:16 2016 -0300

    [manasses] generated by SpringBoot initializr

 .gitignore                                                                 |  42
++++++++++++
 mltracking-poc/mvnw                                                        | 233
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 mltracking-poc/mvnw.cmd                                                    | 145
++++++++++++++++++++++++++++++++++++++
 mltracking-poc/pom.xml                                                     |  74
++++++++++++++++++++
 mltracking-poc/src/main/java/br/com/mls/mltracking/MltrackingPocApplication.java      |  12
++++
 mltracking-poc/src/main/resources/application.properties                   |   0
 mltracking-poc/src/test/java/br/com/mls/mltracking/MltrackingPocApplicationTests.java |  18
+++++
 7 files changed, 524 insertions(+)
```

Показывать содержимое одной фиксации

Используя git show мы можем просмотреть одно сообщение

```
git show 48c83b3
git show 48c83b3690dfc7b0e622fd220f8f37c26a77c934
```

пример

```
commit 48c83b3690dfc7b0e622fd220f8f37c26a77c934
Author: Matt Clark <mrclark32493@gmail.com>
Date:   Wed May 4 18:26:40 2016 -0400

    The commit message will be shown here.

diff --git a/src/main/java/org/jdm/api/jenkins/BuildStatus.java
b/src/main/java/org/jdm/api/jenkins/BuildStatus.java
index 0b57e4a..fa8e6a5 100755
--- a/src/main/java/org/jdm/api/jenkins/BuildStatus.java
+++ b/src/main/java/org/jdm/api/jenkins/BuildStatus.java
@@ -50,7 +50,7 @@ public enum BuildStatus {

                        colorMap.put(BuildStatus.UNSTABLE, Color.decode( "#FFFF55" ));
-                       colorMap.put(BuildStatus.SUCCESS,  Color.decode( "#55FF55" ));
+                       colorMap.put(BuildStatus.SUCCESS,  Color.decode( "#33CC33" ));
                        colorMap.put(BuildStatus.BUILDING, Color.decode( "#5555FF" ));
```

Поиск строки фиксации в git-журнале

Поиск git-журнала с использованием некоторой строки в журнале:

```
git log [options] --grep "search_string"
```

Пример:

```
git log --all --grep "removed file"
```

Будет искать removed file строку removed file во **всех журналах** во **всех филиалах** .

Начиная с git 2.4+, поиск можно инвертировать с помощью опции --invert-grep .

Пример:

```
git log --grep="add file" --invert-grep
```

Покажет все коммиты, которые не содержат add file .

Прочитайте Просмотр истории онлайн: https://riptutorial.com/ru/git/topic/240/просмотр-истории

глава 48: Псевдонимы

Examples

Простые псевдонимы

Существует два способа создания псевдонимов в Git:

- с файлом ~/.gitconfig :

```
[alias]
    ci = commit
    st = status
    co = checkout
```

- с командной строкой:

```
git config --global alias.ci "commit"
git config --global alias.st "status"
git config --global alias.co "checkout"
```

После создания псевдонима введите:

- git ci вместо git commit ,
- git st вместо git status ,
- git co вместо git checkout .

Как и в обычных командах git, псевдонимы могут использоваться рядом с аргументами. Например:

```
git ci -m "Commit message..."
git co -b feature-42
```

Список / поиск существующих псевдонимов

Вы можете перечислить существующие псевдонимы git с помощью --get-regexp :

```
$ git config --get-regexp '^alias\.'
```

Поиск псевдонимов

Чтобы искать псевдонимы , добавьте следующее в свой .gitconfig под [alias] :

```
aliases = !git config --list | grep ^alias\\. | cut -c 7- | grep -Ei --color \"$1\" "#"
```

Тогда ты можешь:

- git aliases - показать ВСЕ псевдонимы
- git aliases commit - только псевдонимы, содержащие "commit"

Расширенные псевдонимы

Git позволяет использовать команды non-git и полный синтаксис оболочки sh в ваших псевдонимах, если вы их префикс ! ,

В файле ~/.gitconfig :

```
[alias]
    temp = !git add -A && git commit -m "Temp"
```

Тот факт, что полный синтаксис оболочки доступен в этих префиксированных псевдонимах, также означает, что вы можете использовать функции оболочки для создания более сложных псевдонимов, таких как те, которые используют аргументы командной строки:

```
[alias]
    ignore = "!f() { echo $1 >> .gitignore; }; f"
```

Вышеупомянутый псевдоним определяет функцию f , затем запускает ее с любыми аргументами, которые вы передаете псевдониму. Таким образом, запуск git ignore .tmp/ будет добавлять .tmp/ в ваш .gitignore файл.

Фактически, этот шаблон настолько полезен, что Git определяет для вас переменные $1 , $2 и т. Д., Поэтому вам даже не нужно определять для него специальную функцию. (Но имейте в виду, что Git также добавит аргументы в любом случае, даже если вы получите доступ к ним через эти переменные, поэтому вам может понадобиться добавить фиктивную команду в конце.)

Обратите внимание, что псевдонимы с префиксом ! таким образом, запускаются из корневого каталога вашего git checkout, даже если ваш текущий каталог глубже в дереве. Это может быть полезным способом запускать команду из корня без необходимости явно указывать cd .

```
[alias]
    ignore = "! echo $1 >> .gitignore"
```

Временно игнорировать отслеживаемые файлы

Временно пометить файл как проигнорированный (передать файл как параметр в псевдоним) – введите:

```
unwatch = update-index --assume-unchanged
```

Чтобы снова запустить файл отслеживания – введите:

```
watch = update-index --no-assume-unchanged
```

Чтобы просмотреть все файлы, которые были временно проигнорированы, введите:

```
unwatched = "!git ls-files -v | grep '^[[:lower:]]'"
```

Чтобы очистить список неподписанных списков – введите:

```
watchall = "!git unwatched | xargs -L 1 -I % sh -c 'git watch `echo % | cut -c 2-`'"
```

Пример использования псевдонимов:

```
git unwatch my_file.txt
git watch my_file.txt
git unwatched
git watchall
```

Показать довольно журнал с графом ветвей

```
[alias]
  logp=log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short
```

```
  lg =  log --graph --date-order --first-parent \
     --pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold
cyan)<%an>%Creset'
  lgb = log --graph --date-order --branches --first-parent \
     --pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold
cyan)<%an>%Creset'
 lga = log --graph --date-order --all \
   --pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold
cyan)<%an>%Creset'
```

Здесь объяснение опций и заполнителей, используемых в формате --pretty (исчерпывающий список доступен с git help log)

--graph - нарисовать дерево фиксации

--date-order - использовать фиксацию метки времени когда возможно

- first-parent - следует только за первым родителем на узле объединения.

--branches - показать все локальные ветви (по умолчанию отображается только текущая ветка)

--all - показать все локальные и удаленные ветви

% h - значение хеша для фиксации (сокращенно)

% ad - штамп даты (автор)

% an - Имя пользователя автора

% an - Записать имя пользователя

% C (авто) - использование цветов, определенных в разделе [цвет]

% Creset - сбросить цвет

% d - --decorate (имена ветвей и тегов)

% s - сообщение фиксации

% ad - дата автора (будет следовать директиве -date) (а не дата commiter)

% an - имя автора (может быть% cn для имени commiter)

Обновление кода при сохранении линейной истории

Иногда вам необходимо сохранить линейную (не ветвящуюся) историю вашего кода. Если вы какое-то время работаете над веткой, это может быть сложно, если вам нужно сделать обычный git pull так как это приведет к слиянию с восходящим потоком.

```
[alias]
  up = pull --rebase
```

Это будет обновляться с вашим исходным исходным кодом, а затем повторно использовать любую работу, которую вы не нажимали поверх того, что вы сняли.

Использовать:

```
git up
```

Посмотрите, какие файлы игнорируются вашей конфигурацией .gitignore.

```
[ alias ]
```

```
    ignored = ! git ls-files --others --ignored --exclude-standard --directory \
            && git ls-files --others -i --exclude-standard
```

Показывает одну строку на файл, поэтому вы можете grep (только каталоги):

```
$ git ignored | grep '/$'
.yardoc/
doc/
```

Или считать:

```
~$ git ignored | wc -l
199811                # oops, my home directory is getting crowded
```

Нестандартные поэтапные файлы

Как правило, для удаления файлов, которые поставлены для фиксации с помощью обязательства reset git reset , reset имеет множество функций в зависимости от аргументов, предоставленных ему. Чтобы полностью отключить все поставленные файлы, мы можем использовать псевдонимы git для создания нового псевдонима, который использует reset но теперь нам не нужно помнить о том, чтобы предоставить правильные аргументы для reset .

```
git config --global alias.unstage "reset --"
```

Теперь, в любое время, когда вы хотите **разбить** файлы с этапами, введите git unstage и вам хорошо идти.

Прочитайте Псевдонимы онлайн: https://riptutorial.com/ru/git/topic/337/псевдонимы

глава 49: Пустые каталоги в Git

Examples

Git не отслеживает каталоги

Предположим, вы инициализировали проект со следующей структурой каталогов:

```
/build
app.js
```

Затем вы добавляете все, что вы создали до сих пор, и совершаете:

```
git init
git add .
git commit -m "Initial commit"
```

Git будет отслеживать только файл app.js.

Предположим, вы добавили шаг сборки в свое приложение и полагаетесь на каталог «build», который будет там в качестве выходного каталога (и вы не хотите делать его инструкцией по установке, за которой должен следовать каждый разработчик), *соглашение* должно включать «.gitkeep» внутри каталога и пусть Git отслеживает этот файл.

```
/build
  .gitkeep
app.js
```

Затем добавьте новый файл:

```
git add build/.gitkeep
git commit -m "Keep the build directory around"
```

Теперь Git будет отслеживать файл build / .gitkeep, и поэтому папка сборки будет доступна для проверки.

Опять же, это просто соглашение, а не функция Git.

Прочитайте Пустые каталоги в Git онлайн: https://riptutorial.com/ru/git/topic/2680/пустые-каталоги-в-git

глава 50: Работа с пультом дистанционного управления

Синтаксис

- git remote [-v | --verbose]
  - git remote add [-t <branch>] [-m <master>] [-f] [--[no-]tags] [--mirror=<fetch|push>] <name> <url>
  - git remote rename <old> <new>
  - git remote remove <name>
  - git remote set-head <name> (-a | --auto | -d | --delete | <branch>)
  - git remote set-branches [--add] <name> <branch>…
  - git remote get-url [--push] [--all] <name>
  - git remote set-url [--push] <name> <newurl> [<oldurl>]
  - git remote set-url --add [--push] <name> <newurl>
  - git remote set-url --delete [--push] <name> <url>
  - git remote [-v | --verbose] show [-n] <name>…
  - git remote prune [-n | --dry-run] <name>…
  - git remote [-v | --verbose] update [-p | --prune] [(<group> | <remote>)…]

Examples

Добавление нового удаленного репозитория

```
git remote add upstream git-repository-url
```

Добавляет удаленный репозиторий , представленный git-repository-url , как новый удаленный по имени upstream по upstream в репозиторий

Обновление из репозитория Upstream

Предполагая, что вы устанавливаете восходящий поток (как в «настройке восходящего репозитория»)

```
git fetch remote-name
git merge remote-name/branch-name
```

Команда pull объединяет fetch и merge .

```
git pull
```

pull с --rebase команды флаг комбинирует fetch и rebase вместо merge .

```
git pull --rebase remote-name branch-name
```

LS-дистанционный

git ls-remote - это уникальная команда, позволяющая запрашивать удаленное репо *без необходимости сначала клонировать / извлекать его* .

Он будет перечислять ссылки refs / head и refs / tags указанного удаленного репо.

Вы увидите иногда refs/tags/v0.1.6 *и* refs/tags/v0.1.6^{} : the ^{} чтобы перечислить отмеченный аннотированный тег (т. refs/tags/v0.1.6^{} Тег, на который указывает этот тег)

Начиная с git 2.8 (март 2016), вы можете избежать двойной записи для тега и перечислить непосредственно те теги с разнесением:

```
git ls-remote --ref
```

Он также может помочь разрешить фактический URL-адрес, используемый удаленным репо, когда у вас установлена настройка « url.<base>.insteadOf ».
Если git remote --get-url <aremotename> возвращает https://server.com/user/repo , и вы установили git config url.ssh://git@server.com:.insteadOf https://server.com/ :

```
git ls-remote --get-url <aremotename>
ssh://git@server.com:user/repo
```

Удаление удаленного отделения

Чтобы удалить удаленную ветку в Git:

```
git push [remote-name] --delete [branch-name]
```

или же

```
git push [remote-name] :[branch-name]
```

Удаление локальных копий удаленных удаленных филиалов

Если удаленная ветвь удалена, вашему локальному репозиторию должно быть предложено обрезать ссылку на него.

Чтобы обрезать удаленные ветки с определенного пульта:

```
git fetch [remote-name] --prune
```

Сократить удаленные ветки из *всех* пультов:

```
git fetch --all --prune
```

Показать информацию о конкретном удаленном

Вывести некоторую информацию о известном удалении: origin

```
git remote show origin
```

Распечатайте только URL-адрес удаленного устройства:

```
git config --get remote.origin.url
```

С 2.7+ также можно сделать, что, возможно, лучше, чем предыдущий, который использует команду config .

```
git remote get-url origin
```

Список существующих пультов

Список всех существующих пультов, связанных с этим репозиторием:

```
git remote
```

Перечислите все существующие пульты, связанные с этим репозиторием, в том числе URL-адреса fetch и push :

```
git remote --verbose
```

или просто

```
git remote -v
```

Начиная

Синтаксис для нажатия на удаленную ветвь

git push <remote_name> <branch_name>

пример

git push origin master

Настройка восходящего потока в новом филиале

Вы можете создать новую ветку и переключиться на нее, используя

```
git checkout -b AP-57
```

После того, как вы используете git checkout для создания новой ветки, вам нужно будет установить начало восходящего потока, чтобы нажать на использование

```
git push --set-upstream origin AP-57
```

После этого вы можете использовать git push, пока вы находитесь на этой ветке.

Изменение удаленного хранилища

Чтобы изменить URL-адрес репозитория, на который вы хотите указать ваш удаленный объект, вы можете использовать параметр set-url , например:

```
git remote set-url <remote_name> <remote_repository_url>
```

Пример:

```
git remote set-url heroku https://git.heroku.com/fictional-remote-repository.git
```

Изменение удаленного URL-адреса Git

Проверить существующий пульт

```
git remote -v
# origin https://github.com/username/repo.git (fetch)
# origin https://github.com/usernam/repo.git (push)
```

Изменение URL-адреса репозитория

```
git remote set-url origin https://github.com/username/repo2.git
# Change the 'origin' remote's URL
```

Проверить новый удаленный URL-адрес

```
git remote -v
# origin  https://github.com/username/repo2.git (fetch)
# origin  https://github.com/username/repo2.git (push)
```

Переименование удаленного

Чтобы переименовать удаленный, используйте команду git remote rename

Команда git remote rename принимает два аргумента:

- Существующее удаленное имя, например: **origin**
- Новое имя для удаленного, например: **destination**

Получить существующее удаленное имя

```
git remote
# origin
```

Проверить существующий пульт с URL-адресом

```
git remote -v
# origin https://github.com/username/repo.git (fetch)
# origin https://github.com/usernam/repo.git (push)
```

Переименовать удаленный

```
 git remote rename origin destination
 # Change remote name from 'origin' to 'destination'
```

Подтвердить новое имя

```
git remote -v
# destination https://github.com/username/repo.git (fetch)
# destination https://github.com/usernam/repo.git (push)
```

=== Положительные ошибки ===

1. Не удалось переименовать раздел конфигурации «remote». [Old name] 'to' remote. [Новое имя] '

   Эта ошибка означает, что удаленный пользователь, которому вы пробовали старое удаленное имя (**источник**), не существует.

2. Удаленный [новое имя] уже существует.

   Сообщение об ошибке самоочевидно.

Задайте URL-адрес для конкретного удаленного

Вы можете изменить URL существующего пульта по команде

```
git remote set-url remote-name url
```

Получить URL-адрес для конкретного удаленного

Вы можете получить URL-адрес существующего пульта, используя команду

```
git remote get-url <name>
```

По умолчанию это будет

```
git remote get-url origin
```

Прочитайте Работа с пультом дистанционного управления онлайн:
https://riptutorial.com/ru/git/topic/243/работа-с-пультом-дистанционного-управления

глава 51: разветвление

Синтаксис

- `git branch [--set-upstream | --track | --no-track] [-l] [-f] <branchname> [<start-point>]`
  - `git branch (--set-upstream-to=<upstream> | -u <upstream>) [<branchname>]`
  - `git branch --unset-upstream [<branchname>]`
  - `git branch (-m | -M) [<oldbranch>] <newbranch>`
  - `git branch (-d | -D) [-r] <branchname>…`
  - `git branch --edit-description [<branchname>]`
  - `git branch [--color[=<when>] | --no-color] [-r | -a] [--list] [-v [--abbrev=<length> | --no-abbrev]] [--column[=<options>] | --no-column] [(--merged | --no-merged | --contains) [<commit>]] [--sort=<key>] [--points-at <object>] [<pattern>…]`

параметры

параметр	подробности
-d, --delete	Удалите ветвь. Филиал должен быть полностью объединен в своей восходящей ветке или в `HEAD` если не было установлено `--track` с помощью `--track` или `--set-upstream`
-D	Ярлык для `--delete --force`
-m, --move	Переместить / переименовать ветку и соответствующий рефлог
-M	Ярлык для `--move --force`
-r, --remotes	Список или удалить (если используется с -d) ветви удаленного отслеживания
-a, --все	Список ветвей удаленного отслеживания и локальных ветвей
--список	Активируйте режим списка. `git branch <pattern>` попытается создать ветвь, используйте `git branch --list <pattern>` чтобы `git branch --list <pattern>` соответствующие ветви
--set-вверх по течению	Если указанная ветвь еще не существует или если `--force` был указан, действует точно так же, как `--track` . В противном случае настройка конфигурации, например, --track, будет возникать при создании ветки, за исключением того, что точки ветвления не изменяются

замечания

Каждый репозиторий git имеет одну или несколько *ветвей* . Ветвь является именованной ссылкой на HEAD последовательности коммитов.

Git repo имеет *текущую* ветвь (обозначенную * в списке названий ветвей, напечатанной командой git branch). Всякий раз, когда вы создаете новую фиксацию с помощью команды git commit , ваша новая фиксация становится HEAD текущей ветви и предыдущий HEAD становится родителем нового коммита.

Новая ветвь будет иметь тот же HEAD что и ветвь, из которой она была создана, пока что-то не будет привязано к новой ветке.

Examples

Листинговые ветки

Git предоставляет несколько команд для перечисления филиалов. Все команды используют функцию git branch , которая предоставит список определенных ветвей, в зависимости от того, какие параметры помещаются в командной строке. Git, если возможно, укажет текущую выбранную ветку со звездой рядом с ней.

Цель	команда
Список местных филиалов	`git branch`
Список локальных филиалов verbose	`git branch -v`
Список удаленных и локальных филиалов	`git branch -a` ИЛИ `git branch --all`
Список удаленных и локальных ветвей (verbose)	`git branch -av`
Список удаленных филиалов	`git branch -r`
Список удаленных филиалов с последней фиксацией	`git branch -rv`
Список объединенных филиалов	`git branch --merged`
Перечислить несвязанные ветви	`git branch --no-merged`
Список ветвей, содержащих фиксацию	`git branch --contains [<commit>]`

Примечания :

- Добавление дополнительного v в -v например, $ git branch -avv или $ git branch -vv также напечатает имя ветви вверх по течению.
- Филиалы, показанные красным цветом, являются удаленными ветвями

Создание и проверка новых ветвей

Чтобы создать новую ветку, оставаясь в текущей ветке, используйте:

```
git branch <name>
```

Как правило, имя филиала не должно содержать пробелов и подлежит другим спецификациям, перечисленным здесь . Чтобы перейти к существующей ветке:

```
git checkout <name>
```

Чтобы создать новую ветку и переключиться на нее:

```
git checkout -b <name>
```

Чтобы создать ветвь в точке, отличной от последней фиксации текущей ветви (также известной как HEAD), используйте одну из следующих команд:

```
git branch <name> [<start-point>]
git checkout -b <name> [<start-point>]
```

<start-point> может быть любой версией, известной git (например, другое имя ветки, commit SHA или символическая ссылка, такая как HEAD или имя тега):

```
git checkout -b <name> some_other_branch
git checkout -b <name> af295
git checkout -b <name> HEAD~5
git checkout -b <name> v1.0.5
```

Чтобы создать ветку из удаленной ветви (по умолчанию <remote_name> есть начало):

```
git branch <name> <remote_name>/<branch_name>
git checkout -b <name> <remote_name>/<branch_name>
```

Если заданное имя филиала найдено только на одном пульте, вы можете просто использовать

```
git checkout -b <branch_name>
```

что эквивалентно

```
git checkout -b <branch_name> <remote_name>/<branch_name>
```

Иногда вам может потребоваться переместить несколько ваших недавних коммитов в новую ветку. Это может быть достигнуто путем ветвления и «откидывания назад», например:

```
git branch <new_name>
git reset --hard HEAD~2 # Go back 2 commits, you will lose uncommitted work.
git checkout <new_name>
```

Вот иллюстративное объяснение этого метода:

```
 Initial state       After git branch <new_name>    After git reset --hard HEAD~2
                             newBranch                      newBranch
                                ↓                              ↓
A-B-C-D-E (HEAD)          A-B-C-D-E (HEAD)               A-B-C-D-E (HEAD)
        ↑                         ↑                              ↑
      master                    master                         master
```

Удалять ветвь локально

```
$ git branch -d dev
```

Удаляет ветвь с именем dev *если* ее изменения объединены с другой ветвью и не будут потеряны. Если ветвь dev содержит изменения, которые еще не были объединены, которые будут потеряны, git branch -d завершится с ошибкой:

```
$ git branch -d dev
error: The branch 'dev' is not fully merged.
If you are sure you want to delete it, run 'git branch -D dev'.
```

За предупреждающее сообщение вы можете принудительно удалить ветвь (и потерять любые несвязанные изменения в этой ветке) с помощью флага -D :

```
$ git branch -D dev
```

Проверьте новую ветку, отслеживающую удаленную ветку

Существует три способа создания новой feature ветвления, которая отслеживает origin/feature удаленного филиала:

- git checkout --track -b feature origin/feature ,
- git checkout -t origin/feature ,
- git checkout feature - при условии, что нет локальной ветки feature и есть только один пульт с ветвью feature .

Чтобы настроить восходящий поток для отслеживания удаленного типа ветви:

- git branch --set-upstream-to=<remote>/<branch> <branch>
  - git branch -u <remote>/<branch> <branch>

где:

- <remote> может быть: origin , develop или созданный пользователем,
- <branch> - <branch> пользователя для отслеживания на удаленном компьютере.

Чтобы проверить, какие удаленные отделения отслеживают ваши локальные филиалы:

- git branch -vv

Переименовать ветвь

Переименуйте выделенную ветку:

```
git branch -m new_branch_name
```

Переименовать другую ветку:

```
git branch -m branch_you_want_to_rename new_branch_name
```

Перезаписать одиночный файл в текущем рабочем каталоге с тем же именем из другой ветви

Выбранный файл **перезапишет** еще не внесенные изменения, которые вы сделали в этом файле.

Эта команда проверит файл file.example (который находится в path/to/ каталога path/to/) и **перезапишет любые изменения, которые** вы могли бы внести в этот файл.

```
git checkout some-branch path/to/file
```

some-branch может быть любым tree-ish известным git (см. Редактирование выбора и gitrevisions для получения дополнительной информации)

Вы должны добавить -- до пути, если ваш файл может быть ошибочно принят за файл (необязательно в противном случае). После -- .

```
git checkout some-branch -- some-file
```

Второй some-file - это файл в этом примере.

Удаление удаленной ветви

Чтобы удалить ветку на origin удаленном хранилище, вы можете использовать для Git версии 1.5.0 и новее

```
git push origin :<branchName>
```

и с версии Git версии 1.7.0 вы можете удалить удаленную ветку, используя

```
git push origin --delete <branchName>
```

Чтобы удалить локальную ветвь удаленного отслеживания:

```
git branch --delete --remotes <remote>/<branch>
git branch -dr <remote>/<branch> # Shorter

git fetch <remote> --prune # Delete multiple obsolete tracking branches
git fetch <remote> -p      # Shorter
```

Чтобы удалить ветвь локально. Обратите внимание, что это не приведет к удалению ветки, если она имеет любые несанкционированные изменения:

```
git branch -d <branchName>
```

Чтобы удалить ветвь, даже если она имеет несвязанные изменения:

```
git branch -D <branchName>
```

Создайте сиротскую ветвь (т.е. ветвь без фиксации родителя)

```
git checkout --orphan new-orphan-branch
```

> Первая фиксация, сделанная в этой новой ветке, не будет иметь родителей, и она станет корнем новой истории, полностью отключенной от всех других ветвей и совершит.

источник

Вставить ветку в удаленный

Используйте, чтобы нажимать фиксации, сделанные на вашей локальной ветке, в удаленный репозиторий.

Команда git push принимает два аргумента:

- Удаленное имя, например, origin
- Название ветки, например, master

Например:

```
git push  <REMOTENAME> <BRANCHNAME>
```

Например, вы обычно запускаете git push origin master чтобы перенаправить локальные изменения в свой онлайн-репозиторий.

Использование -u (short для --set-upstream) будет настраивать информацию отслеживания во время нажатия.

```
git push -u <REMOTENAME> <BRANCHNAME>
```

По умолчанию git помещает локальную ветвь в удаленную ветку с тем же именем. Например, если у вас есть локальная new-feature , если вы нажмете локальную ветвь, она также создаст new-feature ветвь. Если вы хотите использовать другое имя для удаленного филиала, добавьте имя удаленного после имени локального филиала, отделенных : :

```
git push <REMOTENAME> <LOCALBRANCHNAME>:<REMOTEBRANCHNAME>
```

Переместить текущую ветвь HEAD в произвольное

Ветвь - это всего лишь указатель на фиксацию, поэтому вы можете свободно перемещать ее. Чтобы сделать это так, чтобы ветвь aabbcc на commit aabbcc , выполните команду

```
git reset --hard aabbcc
```

Обратите внимание, что это приведет к перезаписыванию текущей фиксации вашего филиала и, как следствие, всей его истории. Вы можете потерять некоторую работу, выпустив эту команду. Если это так, вы можете использовать reflog для восстановления потерянных коммитов . Рекомендуется выполнить эту команду на новой ветке вместо текущей.

Однако эта команда может быть особенно полезна при перезагрузке или выполнении таких больших изменений истории.

Быстрый переход к предыдущей ветке

Вы можете быстро переключиться на предыдущую ветку, используя

```
git checkout -
```

Поиск в филиалах

Чтобы просмотреть локальные ветви, содержащие конкретную фиксацию или тег

```
git branch --contains <commit>
```

Список локальных и удаленных филиалов, которые содержат конкретную фиксацию или тег

```
git branch -a --contains <commit>
```

Прочитайте разветвление онлайн: https://riptutorial.com/ru/git/topic/415/разветвление

глава 52: развязывание

Examples

Отмена слияния

Отмена слияния еще не нажата на удаленный

Если вы еще не нажали свое слияние в удаленный репозиторий, вы можете выполнить ту же процедуру, что и в отмене фиксации, хотя есть некоторые тонкие отличия.

Сброс - это самый простой вариант, поскольку он отменяет как фиксацию слияния, так и любые фиксации, добавленные из ветки. Тем не менее, вам нужно будет знать, что SHA для возврата обратно, это может быть сложно, так как ваш git log теперь будет показывать фиксации из обеих ветвей. Если вы переустановите неверную фиксацию (например, одну на другой ветке), **она может уничтожить совершенную работу.**

```
> git reset --hard <last commit from the branch you are on>
```

Или, предполагая, что слияние было вашим последним фиксатором.

```
> git reset HEAD~
```

Возврат более безопасен, поскольку он не уничтожит совершенную работу, но требует больше работы, так как вам нужно вернуть реверс, прежде чем вы сможете снова объединить ветку (см. Следующий раздел).

Отмена слияния, нажатого на удаленный

Предположим, вы слились в новую функцию (add-gremlins)

```
> git merge feature/add-gremlins
...
   #Resolve any merge conflicts
> git commit #commit the merge
...
> git push
...
   501b75d..17a51fd  master -> master
```

После этого вы обнаружите, что функция, которую вы только что объединили, нарушила систему для других разработчиков, ее нужно немедленно отменить, и исправление самой функции займет слишком много времени, поэтому вы просто хотите отменить слияние.

```
> git revert -m 1 17a51fd
...
> git push
...
   17a51fd..e443799  master -> master
```

На данный момент гремлины вышли из системы, и ваши коллеги-разработчики перестали кричать на вас. Тем не менее, мы еще не закончили. Как только вы устраните проблему с помощью функции add-gremlins, вам нужно будет отменить это возвращение, прежде чем вы сможете снова объединиться.

```
> git checkout feature/add-gremlins
...
   #Various commits to fix the bug.
> git checkout master
```

```
...
> git revert e443799
...
> git merge feature/add-gremlins
...
   #Fix any merge conflicts introduced by the bug fix
> git commit #commit the merge
...
> git push
```

На данный момент ваша функция теперь успешно добавлена. Однако, учитывая, что ошибки этого типа часто вводятся конфликтами слияния, иногда бывает более полезным несколько иной рабочий процесс, поскольку он позволяет исправить конфликт слияния в вашей ветке.

```
> git checkout feature/add-gremlins
...
   #Merge in master and revert the revert right away.  This puts your branch in
   #the same broken state that master was in before.
> git merge master
...
> git revert e443799
...
   #Now go ahead and fix the bug (various commits go here)
> git checkout master
...
   #Don't need to revert the revert at this point since it was done earlier
> git merge feature/add-gremlins
...
   #Fix any merge conflicts introduced by the bug fix
> git commit #commit the merge
...
> git push
```

Использование reflog

Если вы испортите rebase, один из вариантов начать снова – вернуться к фиксации (pre rebase). Вы можете сделать это с помощью reflog (который имеет историю всего, что вы делали за последние 90 дней – это можно настроить):

```
$ git reflog
4a5cbb3 HEAD@{0}: rebase finished: returning to refs/heads/foo
4a5cbb3 HEAD@{1}: rebase: fixed such and such
904f7f0 HEAD@{2}: rebase: checkout upstream/master
3cbe20a HEAD@{3}: commit: fixed such and such
...
```

Вы можете увидеть фиксацию до того, как rebase будет HEAD@{3} (вы также можете проверить хэш):

```
git checkout HEAD@{3}
```

Теперь вы создаете новую ветку / удаляете старую / снова пытаетесь переустановить.

Вы также можете вернуться обратно к точке вашего reflog , но только сделайте это, если вы на 100% уверены, что это то, что вы хотите сделать:

git reset --hard HEAD@{3}

Это приведет к тому, что ваше текущее дерево git будет соответствовать тому, как оно было в этот момент (см. «Отмена изменений»).

Это можно использовать, если вы временно видите, насколько хорошо работает ветка, когда она переустанавливается на другую ветку, но вы не хотите сохранять результаты.

Возврат к предыдущему фиксации

Чтобы вернуться к предыдущей фиксации, сначала найдите хеш фиксации, используя git log .

Чтобы временно вернуться к этой фиксации, отделите голову:

```
git checkout 789abcd
```

Это помещает вас в фиксацию 789abcd . Теперь вы можете совершать новые коммиты поверх этой старой фиксации, не затрагивая ветку, на которой находится ваша голова. Любые изменения могут быть внесены в правильную ветку с использованием branch или checkout -b .

Чтобы вернуться к предыдущему фиксации при сохранении изменений:

```
git reset --soft 789abcd
```

Откат ***последнего*** фиксации:

```
git reset --soft HEAD~
```

Чтобы навсегда отказаться от любых изменений, сделанных после определенного фиксации, используйте:

```
git reset --hard 789abcd
```

Чтобы окончательно отказаться от любых изменений, сделанных после ***последнего*** фиксации:

```
git reset --hard HEAD~
```

Остерегайтесь: хотя вы можете восстановить отброшенные коммиты с использованием reflog и reset , незафиксированные изменения не могут быть восстановлены. Используйте git stash; git reset вместо git reset --hard - безопасно.

Отмена изменений

Отменить изменения в файле или каталоге **рабочей копии** .

```
git checkout -- file.txt
```

Используется во всех путях файлов, рекурсивно из текущего каталога, он отменяет все изменения в рабочей копии.

```
git checkout -- .
```

Для отмены только частей изменений используйте --patch . Вас спросят, для каждого изменения, если оно должно быть отменено или нет.

```
git checkout --patch -- dir
```

Чтобы отменить изменения, добавленные в **индекс** .

```
git reset --hard
```

Без --hard это сделает мягкий сброс.

С локальными фиксациями, которые вы еще должны нажать на удаленный компьютер, вы также можете выполнить мягкий сброс. Таким образом, вы можете переделать файлы, а затем совершить.

```
git reset HEAD~2
```

В приведенном выше примере будут разматываться ваши последние две коммиты и возвращать файлы в вашу рабочую копию. Затем вы можете внести дальнейшие изменения и новые коммиты.

Остерегайтесь: все эти операции, помимо мягких сбросов, навсегда удалит ваши изменения. Для более безопасного варианта используйте git stash -p или git stash , соответственно. Позднее вы можете отменить stash pop или удалить навсегда с stash drop .

Отменить некоторые существующие коммиты

Используйте git revert, чтобы вернуть существующие коммиты, особенно когда эти коммиты были перенесены в удаленный репозиторий. Он записывает некоторые новые коммиты, чтобы отменить эффект некоторых более ранних коммитов, которые вы можете безопасно продвигать без перезаписи истории.

Не используйте git push --force если вы не хотите сбить оппортунизм всех других пользователей этого репозитория. Никогда не переписывайте публичную историю.

Если, например, вы только что подтолкнули коммит, содержащий ошибку, и вам нужно сделать это, выполните следующие действия:

```
git revert HEAD~1
git push
```

Теперь вы можете повторно отменить фиксацию фиксации локально, исправить свой код и нажать хороший код:

```
git revert HEAD~1
work .. work .. work ..
git add -A .
git commit -m "Update error code"
git push
```

Если фиксация, которую вы хотите вернуть, уже возвращается в историю, вы можете просто передать хеш фиксации. Git создаст контр-фиксацию, отменив вашу первоначальную фиксацию, которую вы можете безопасно нажать на свой пульт.

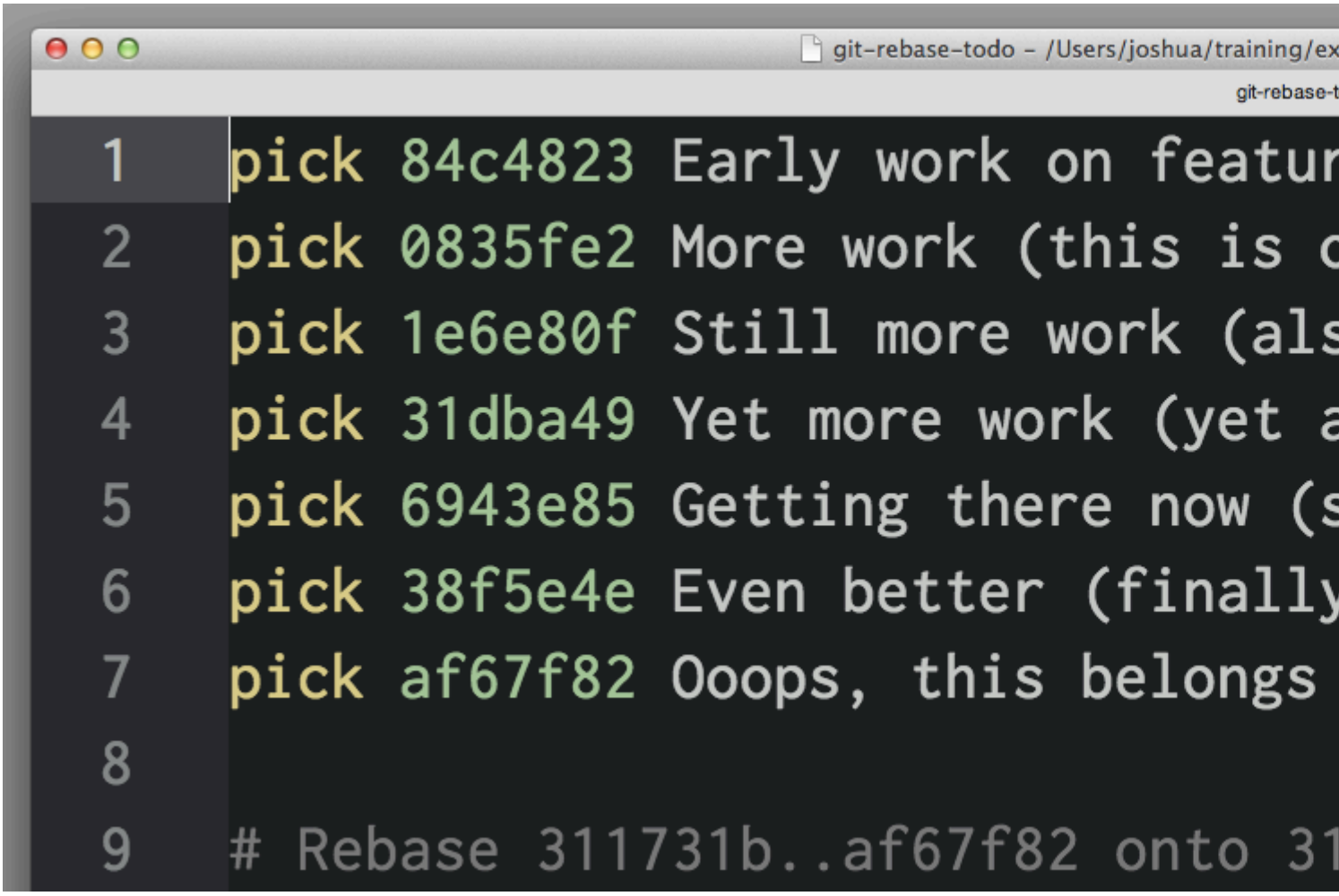
```
git revert 912aaf0228338d0c8fb8cca0a064b0161a451fdc
git push
```

Отменить / Повторить серию коммитов

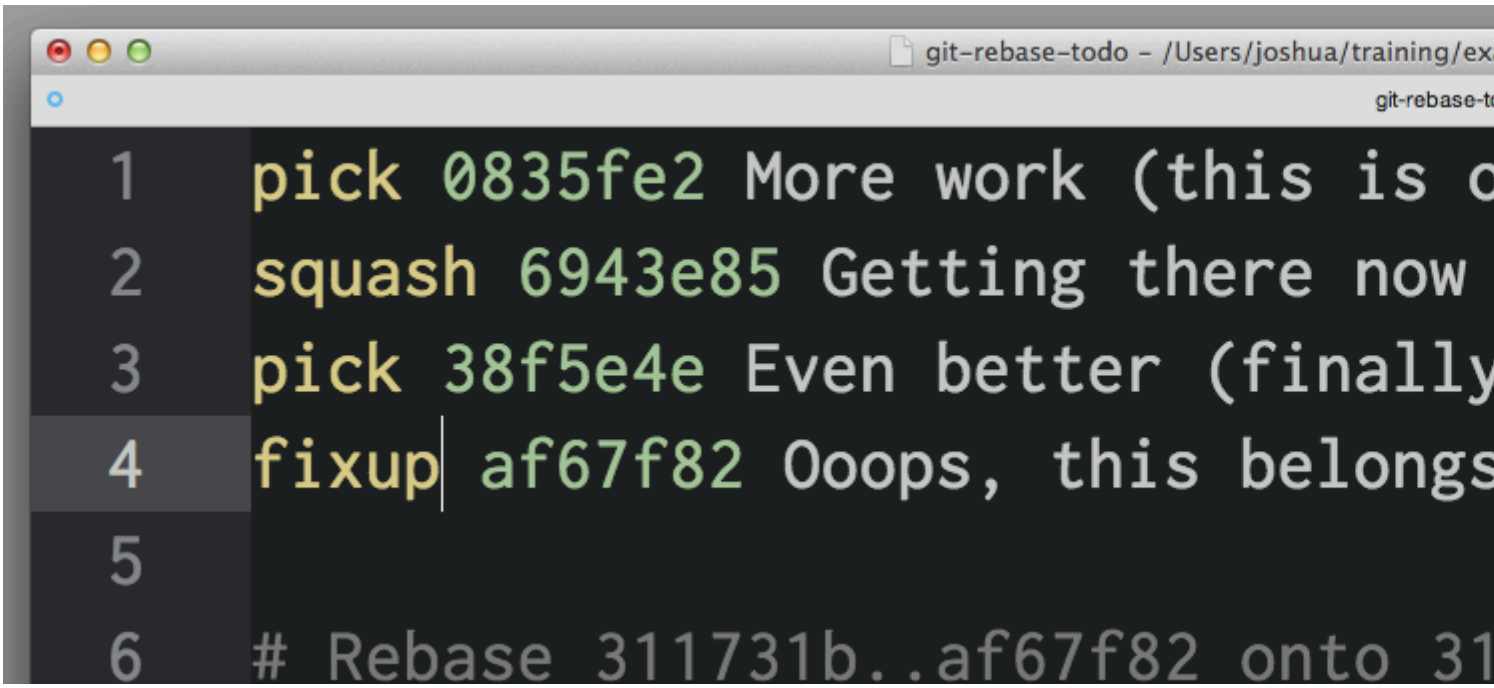
Предположим, вы хотите отменить десяток коммитов, и вы хотите только некоторые из них.

```
git rebase -i <earlier SHA>
```

-i помещает rebase в «интерактивный режим». Он начинается, как обсуждаемый выше, но, прежде чем повторять какие-либо коммиты, он приостанавливается и позволяет вам мягко модифицировать каждую фиксацию по мере ее повторного воспроизведения. *rebase -i* откроется в текстовом редакторе по умолчанию со списком применяемых коммитов:



Чтобы удалить фиксацию, просто удалите эту строку в своем редакторе. Если вам больше не нужны плохие коммиты в вашем проекте, вы можете удалить строки 1 и 3-4 выше. Если вы хотите объединить две коммиты вместе, вы можете использовать команды squash или fixup



Прочитайте развязывание онлайн: https://riptutorial.com/ru/git/topic/285/развязывание

глава 53: Разрешение конфликтов слияния

Examples

Ручное разрешение

При выполнении git merge вы можете обнаружить, что git сообщает об ошибке «слияние конфликта». Он сообщит вам, какие файлы имеют конфликты, и вам нужно будет разрешить конфликты.

git status в любой момент поможет вам понять, что еще нужно редактировать с помощью полезного сообщения, например

```
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

Git оставляет маркеры в файлах, чтобы сообщить вам, где возник конфликт:

```
<<<<<<<<< HEAD: index.html #indicates the state of your current branch
<div id="footer">contact : email@somedomain.com</div>
========= #indicates break between conflicts
<div id="footer">
please contact us at email@somedomain.com
</div>
>>>>>>>>> iss2: index.html #indicates the state of the other branch (iss2)
```

Чтобы разрешить конфликты, вы должны соответствующим образом отредактировать область между <<<<<< и >>>>>>> маркерами, удалить строки состояния (<<<<<<<, >>>>> >>, и ======== строки) полностью. Затем git add index.html чтобы пометить его, и git commit чтобы завершить слияние.

Прочитайте Разрешение конфликтов слияния онлайн: https://riptutorial.com/ru/git/topic/3233/разрешение-конфликтов-слияния

глава 54: Связки

замечания

Ключом к этой работе является начало клонирования пакета, который начинается с начала истории репо:

```
git bundle create initial.bundle master
git tag -f some_previous_tag master  # so the whole repo does not have to go each time
```

получение этого исходного пакета на удаленную машину; а также

```
git clone -b master initial.bundle remote_repo_name
```

Examples

Создание пакета git на локальной машине и использование его на другом

Иногда вам может потребоваться поддерживать версии git-репозитория на компьютерах, которые не имеют сетевого подключения. Связки позволяют вам упаковывать объекты и ссылки git в репозиторий на одном компьютере и импортировать их в репозиторий на другом.

```
git tag 2016_07_24
git bundle create changes_between_tags.bundle [some_previous_tag]..2016_07_24
```

Как-то **передайте** файл **changes_between_tags.bundle** на удаленный компьютер; например, с помощью флэш-накопителя. Когда у вас есть это:

```
git bundle verify changes_between_tags.bundle  # make sure bundle arrived intact
git checkout [some branch]       # in the repo on the remote machine
git bundle list-heads changes_between_tags.bundle # list the references in the bundle
git pull changes_between_tags.bundle [reference from the bundle, e.g. last field from the
previous output]
```

Также возможно обратное. После того, как вы внесли изменения в удаленный репозиторий, вы можете объединить дельта; поместите изменения на, например, флэш-накопитель, и объедините их обратно в локальный репозиторий, чтобы они могли оставаться в синхронизации, не требуя прямого доступа к протоколам git , ssh , rsync или http между машинами.

Прочитайте Связки онлайн: https://riptutorial.com/ru/git/topic/3612/связки

глава 55: Синтаксис ревизий Git

замечания

Многие команды Git принимают параметры ревизии в качестве аргументов. В зависимости от команды они обозначают конкретную фиксацию или, для команд, которые ходят по графику ревизии (например, **git-log (1)**), все коммиты, которые могут быть достигнуты из этой фиксации. Они обычно обозначаются как <commit> , или <rev> или <revision> в описании синтаксиса.

Справочной документацией для синтаксиса редакций Git является **справочная** страница **gitrevisions (7)** .

Все еще отсутствует на этой странице:

- [_] Выход из git describe , например, v1.7.4.2-679-g3bee7fb
- [_] @ как ярлык для HEAD
- [_] @{-<n>} , например @{-1} , и - значение @{-1}
- [_] <branchname>@{push}
- [_] <rev>^@ , для всех родителей <rev>

Требуется отдельная документация:

- [_] Ссылаясь на blobs и деревья в репозитории и в индексе: <rev>:<path> и :<n>:<path> синтаксис
- Диапазоны пересмотра, такие как A..B , A...B , B ^A , A^1 , и ограничение пересмотра, такое как -<n> , - --since

Examples

Указание ревизии по имени объекта

```
$ git show dae86e1950b1277e545cee180551750029cfe735
$ git show dae86e19
```

Вы можете указать ревизию (или, по правде говоря, любой объект: тег, дерево, т.е. содержимое каталога, blob, т.е. содержимое файла), используя имя объекта SHA-1, либо полную 40-байтную шестнадцатеричную строку, либо подстроку, которая уникальна для репозитория.

Символьные имена ссылок: ветви, теги, ветви удаленного отслеживания

```
$ git log master    # specify branch
$ git show v1.0     # specify tag
$ git show HEAD     # specify current branch
$ git show origin   # specify default remote-tracking branch for remote 'origin'
```

Вы можете указать ревизию с использованием символьного имени ref, которое включает в себя ветви (например, «master», «next», «maint»), теги (например, «v1.0», «v0.6.3-rc2») (например, « origin», «origin / master») и специальные ссылки, такие как «HEAD» для текущей ветви.

Если символическое имя ссылки неоднозначно, например, если у вас есть как ветка, так и тег с именем «fix» (с веткой и тегом с тем же именем не рекомендуется), вам нужно указать тип ссылки, которую вы хотите использовать:

```
$ git show heads/fix      # or 'refs/heads/fix', to specify branch
$ git show tags/fix       # or 'refs/tags/fix', to specify tag
```

Версия по умолчанию: HEAD

```
$ git show         # equivalent to 'git show HEAD'
```

«HEAD» называет фиксацию, на которой вы основываете изменения в рабочем дереве, и обычно является символическим именем для текущей ветви. Многие (но не все) команды, для которых параметр ревизии по умолчанию имеет значение «HEAD», если он отсутствует.

Ссылки Reflog: @ { }

```
$ git show @{1}            # uses reflog for current branch
$ git show master@{1}      # uses reflog for branch 'master'
$ git show HEAD@{1}        # uses 'HEAD' reflog
```

Ссылка, обычно ветка или HEAD, за которой следует суффикс @ с порядковой спецификацией, заключенной в пару скобок (например, {1} , {15}), указывает n-ое предшествующее значение этого ref *в вашем* ***локальном*** *репозитории* . Вы можете проверить последние записи reflog с git reflog команды git reflog или --walk-reflogs / -g для git log .

```
$ git reflog
08bb350 HEAD@{0}: reset: moving to HEAD^
4ebf58d HEAD@{1}: commit: gitweb(1): Document query parameters
08bb350 HEAD@{2}: pull: Fast-forward
f34be46 HEAD@{3}: checkout: moving from af40944bda352190f05d22b7cb8fe88beb17f3a7 to master
af40944 HEAD@{4}: checkout: moving from master to v2.6.3

$ git reflog gitweb-docs
4ebf58d gitweb-docs@{0}: branch: Created from master
```

Примечание : использование журналов практически заменило старый механизм использования ORIG_HEAD ref (примерно эквивалентный HEAD@{1}).

Ссылки Reflog: @ { }

```
$ git show master@{yesterday}
$ git show HEAD@{5 minutes ago}   # or HEAD@{5.minutes.ago}
```

Ссылка, за которой следует суффикс @ с спецификацией даты, заключенной в пару скобок (например, {yesterday} , {1 month 2 weeks 3 days 1 hour 1 second ago} или {1979-02-26 18:30:00}) указывает значение ref в предшествующий момент времени (или ближайшая точка к нему). Обратите внимание, что это просматривает состояние вашего **локального** ref в данный момент времени; например, то, что было на вашем местном *«хозяине» на* прошлой неделе.

Вы можете использовать git reflog с спецификатором даты, чтобы найти точное время, когда вы сделали что-то, чтобы дать ref в локальном репозитории.

```
$ git reflog HEAD@{now}
08bb350 HEAD@{Sat Jul 23 19:48:13 2016 +0200}: reset: moving to HEAD^
4ebf58d HEAD@{Sat Jul 23 19:39:20 2016 +0200}: commit: gitweb(1): Document query parameters
08bb350 HEAD@{Sat Jul 23 19:26:43 2016 +0200}: pull: Fast-forward
```

Отслеживаемый / восходящий филиал: @ {Вверх}

```
$ git log @{upstream}..       # what was done locally and not yet published, current branch
$ git show master@{upstream}  # show upstream of branch 'master'
```

Суффикс @{upstream} добавленный к branchname (короткая форма <branchname>@{u}), относится к ветке, которую ветвь, указанная branchname, устанавливается на вершину (настроена с помощью branch.<name>.remote и branch.<name>.merge , или с git branch --set-upstream-to=<branch>).

Отсутствует branchname по умолчанию для текущего.

Вместе с синтаксисом для диапазонов ревизий очень полезно видеть, что ваша ветка впереди вверх (коммиты в вашем локальном репозитории еще не представлены вверх по течению), и что заставляет вас отстать (совершает восходящий поток, не объединенный в локальную ветвь), или и то и другое:

```
$ git log --oneline @{u}..
$ git log --oneline ..@{u}
$ git log --oneline --left-right @{u}...  # same as ...@{u}
```

Целевая цепочка: ^, ~ , так далее.

```
$ git reset --hard HEAD^             # discard last commit
$ git rebase --interactive HEAD~5    # rebase last 4 commits
```

Суффикс ^ к параметру ревизии означает первый родитель этого объекта фиксации. ^<n> означает <n> -й родительский элемент (т. е. <rev>^ эквивалентен <rev>^1).

Суффикс ~<n> к параметру ревизии означает объект commit, который является предком n-го поколения именованного объекта commit, следуя только первым родителям. Это означает, что, например, <rev>~3 эквивалентно <rev>^^^ . В качестве ярлыка <rev>~ означает <rev>~1 и эквивалентно <rev>^1 или <rev>^ вкратце.

Этот синтаксис является составным.

Чтобы найти такие символические имена, вы можете использовать команду git name-rev :

```
$ git name-rev 33db5f4d9027a10e477ccf054b2c1ab94f74c85a
33db5f4d9027a10e477ccf054b2c1ab94f74c85a tags/v0.99~940
```

Обратите внимание, что в следующем примере необходимо использовать --pretty=oneline и not --oneline

```
$ git log --pretty=oneline | git name-rev --stdin --name-only
master Sixth batch of topics for 2.10
master~1 Merge branch 'ls/p4-tmp-refs'
master~2 Merge branch 'js/am-call-theirs-theirs-in-fallback-3way'
[...]
master~14^2 sideband.c: small optimization of strbuf usage
master~16^2 connect: read $GIT_SSH_COMMAND from config file
[...]
master~22^2~1 t7810-grep.sh: fix a whitespace inconsistency
master~22^2~2 t7810-grep.sh: fix duplicated test name
```

Развертывание разделов и тегов: ^ 0, ^ { }

В некоторых случаях поведение команды зависит от того, задано ли ей имя ветки, имя тега или произвольная ревизия. Вы можете использовать синтаксис «de-referencing», если вам нужен последний.

Суффикс ^ за которым следует имя типа объекта (tag , commit , tree , blob), заключенное в пару скобок (например, v0.99.8^{commit}), означает разыменование объекта в <rev> рекурсивно, пока объект типа <type> или объект не может быть разыменован. <rev>^0 является короткой рукой для <rev>^{commit} .

```
$ git checkout HEAD^0    # equivalent to 'git checkout --detach' in modern Git
```

Суффикс ^ за которым следует пустая фигурная скобка (например, v0.99.8^{}), означает разыменование тега рекурсивно до тех пор, пока не будет найден объект без тегов.

сравнить

```
$ git show v1.0
$ git cat-file -p v1.0
$ git replace --edit v1.0
```

с

```
$ git show v1.0^{}
$ git cat-file -p v1.0^{}
$ git replace --edit v1.0^{}
```

Наименьшая согласованная фиксация: ^ {/ },: /

```
$ git show HEAD^{/fix nasty bug}   # find starting from HEAD
$ git show ':/fix nasty bug'       # find starting from any branch
```

Двоеточие (' : '), за которым следует слэш (' / '), за которым следует текст, называет коммит, чье сообщение фиксации соответствует указанному регулярному выражению. Это имя возвращает самую младшую совпадающую фиксацию, доступную из *любого* ref. Регулярное выражение может соответствовать любой части сообщения фиксации. Чтобы сопоставить сообщения, начинающиеся со строки, можно использовать, например :/^foo . Специальная последовательность :/! зарезервирован для модификаторов к тому, что соответствует. :/!-foo выполняет отрицательное совпадение, а :/!!foo соответствует литералу! символ, за которым следует foo .

Суффикс ^ к параметру ревизии, за которым следует пара скобок, содержащая текст, приводимый косой чертой, совпадает с синтаксисом :/<text> ниже, который возвращает младшую согласованную фиксацию, доступную от <rev> до ^ .

Прочитайте Синтаксис ревизий Git онлайн: https://riptutorial.com/ru/git/topic/3735/синтаксис-ревизий-git

глава 56: совершение

Вступление

Обязательства с Git обеспечивают подотчетность, приписывая авторов изменениям кода. Git предлагает несколько функций для специфики и безопасности транзакций. В этом разделе объясняются и демонстрируются правильные методы и процедуры, связанные с Git.

Синтаксис

- `git commit [flags]`

параметры

параметр	подробности
--message, -m	Сообщение для включения в коммит. Указание этого параметра обходит обычное поведение Git при открытии редактора.
--amend	Укажите, что изменения, которые в настоящее время выполняются, должны быть добавлены (изменены) к *предыдущему* фиксации. Будьте осторожны, это может переписать историю!
--no-редактировать	Используйте выбранное сообщение фиксации без запуска редактора. Например, `git commit --amend --no-edit` вносит изменения в коммит без изменения сообщения о фиксации.
-all, -a	Зафиксируйте все изменения, включая изменения, которые еще не поставлены.
--Дата	Вручную установите дату, которая будет связана с фиксацией.
--только	Зафиксируйте только указанные пути. Это не приведет к тому, что вы в настоящее время поставили, если не сказали об этом.
--patch, -p	Используйте интерактивный интерфейс выбора патчей для выбора изменений для фиксации.
--Помогите	Отображает страницу man для `git commit`
-S [keyid], -S --gpg-sign [= keyid], -S --no-gpg-sign	Sign commit, фиксация GPG-знака, переменная конфигурации countermand `commit.gpgSign`

параметр	подробности
-n, --no-verify	Этот параметр обходит крючки pre-commit и commit-msg. См. Также Крючки

Examples

Выполнение без открытия редактора

Обычно Git обычно открывает редактор (например, vim или emacs), когда вы запускаете git commit . Передайте параметр -m для указания сообщения из командной строки:

```
git commit -m "Commit message here"
```

Ваше сообщение фиксации может проходить через несколько строк:

```
git commit -m "Commit 'subject line' message here

More detailed description follows here (after a blank line)."
```

Кроме того, вы можете передать несколько аргументов -m :

```
git commit -m "Commit summary" -m "More detailed description follows here"
```

См. Как написать сообщение Git Commit .

Руководство по стилю сообщения об уродстве Git

Изменение коммита

Если ваша **последняя фиксация еще не опубликована** (не помещена в репозиторий восходящего потока), вы можете изменить свое сообщение.

```
git commit --amend
```

Это поместит текущие поэтапные изменения в предыдущую фиксацию.

Примечание. Это также можно использовать для редактирования некорректного сообщения фиксации. Он откроет редактор по умолчанию (обычно vi / vim / emacs) и позволит вам изменить предыдущее сообщение.

Чтобы указать встроенное сообщение фиксации:

```
git commit --amend -m "New commit message"
```

Или использовать предыдущее сообщение фиксации без его изменения:

```
git commit --amend --no-edit
```

Изменение изменений даты фиксации, но оставляет дату автора нетронутой. Вы можете сообщить git, чтобы обновить информацию.

```
git commit --amend --reset-author
```

Вы также можете изменить автора фиксации с помощью:

```
git commit --amend --author "New Author <email@address.com>"
```

Примечание. Имейте в виду, что изменение последней фиксации полностью заменяет ее, а предыдущая фиксация удаляется из истории ветви. Это следует учитывать при работе с публичными хранилищами и в филиалах с другими сотрудниками.

Это означает, что если предыдущая фиксация уже была нажата, после внесения в нее изменений вам придется push --force .

Перенос изменений непосредственно

Как правило, вам нужно использовать git add или git rm для добавления изменений в индекс, прежде чем вы сможете git commit их. Передайте параметр -a или --all для автоматического добавления каждого изменения (для отслеживаемых файлов) в индекс, включая удаление:

```
git commit -a
```

Если вы также хотите добавить сообщение фиксации, которое вы бы сделали:

```
git commit -a -m "your commit message goes here"
```

Кроме того, вы можете присоединиться к двум флагам:

```
git commit -am "your commit message goes here"
```

Вам не обязательно обязательно фиксировать все файлы одновременно. Опустите флаг -a или --all и укажите, какой файл вы хотите зафиксировать напрямую:

```
git commit path/to/a/file -m "your commit message goes here"
```

Для прямого ввода нескольких файлов можно указать один или несколько файлов, каталогов и шаблонов:

```
git commit path/to/a/file path/to/a/folder/* path/to/b/file -m "your commit message goes here"
```

Создание пустой фиксации

Вообще говоря, пустые коммиты (или коммиты с состоянием, которые идентичны родительскому) являются ошибкой.

Однако при тестировании сборок сборки, CI-систем и других систем, которые вызывают фиксацию, удобно иметь возможность легко создавать коммиты без необходимости редактировать / прикоснуться к фиктивному файлу.

--allow-empty будет обходить проверку.

```
git commit -m "This is a blank commit" --allow-empty
```

Изменения этапа и фиксации

Основы

После внесения изменений в исходный код вы должны **сгенерировать** эти изменения с помощью Git, прежде чем сможете их совершить.

Например, если вы измените README.md и program.py :

```
git add README.md program.py
```

Это говорит git, что вы хотите добавить файлы к следующему фиксации.

Затем выполните свои изменения с помощью

```
git commit
```

Обратите внимание, что это откроет текстовый редактор, который часто является vim . Если вы не знакомы с vim, вам может понадобиться знать, что вы можете нажать i чтобы перейти в режим *вставки* , написать сообщение о фиксации, а затем нажать Esc и :wq для сохранения и выхода. Чтобы не открывать текстовый редактор, просто -m флаг -m с вашим сообщением

```
git commit -m "Commit message here"
```

Фиксированные сообщения часто следуют определенным правилам форматирования, см. « Сообщения о хорошей фиксации» для получения дополнительной информации.

Ярлыки

Если вы изменили большое количество файлов в каталоге, а не перечисляете каждый из них, вы можете использовать:

```
git add --all        # equivalent to "git add -a"
```

Или добавить все изменения, *не считая удаленных файлов* , из каталога верхнего уровня и подкаталогов:

```
git add .
```

Или добавлять только файлы, которые в настоящее время отслеживаются («обновление»):

```
git add -u
```

При желании просмотрите поэтапные изменения:

```
git status           # display a list of changed files
git diff --cached    # shows staged changes inside staged files
```

Наконец, зафиксируйте изменения:

```
git commit -m "Commit message here"
```

В качестве альтернативы, если вы только модифицировали существующие файлы или удаленные файлы и не создали никаких новых, вы можете комбинировать действия git add и git commit в одной команде:

```
git commit -am "Commit message here"
```

Обратите внимание, что это будет сгенерировать **все** измененные файлы так же, как git add --all .

Чувствительные данные

Вы никогда не должны передавать конфиденциальные данные, такие как пароли или даже закрытые ключи. Если это произойдет и изменения уже перенесены на центральный сервер, рассмотрите любые конфиденциальные данные как скомпрометированные. В противном случае после этого можно удалить

такие данные. Быстрое и простое решение - это использование «BFG Repo-Cleaner»: https://rtyley.github.io/bfg-repo-cleaner/ .

Команда bfg --replace-text passwords.txt my-repo.git считывает пароли из файла passwords.txt и заменяет их ***REMOVED*** . Эта операция рассматривает все предыдущие коммиты всего репозитория.

Задание от имени кого-то другого

Если кто-то другой написал код, который вы совершаете, вы можете дать им кредит с опцией --author :

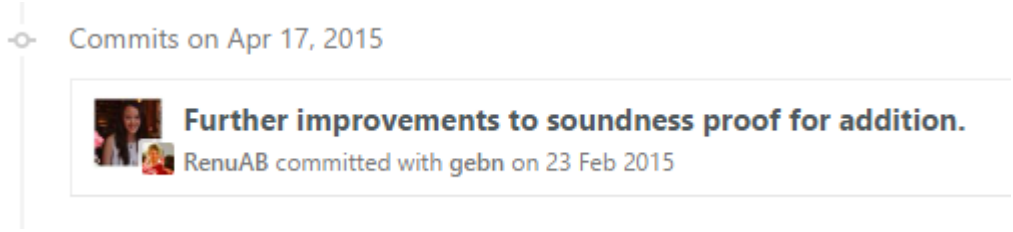
```
git commit -m "msg" --author "John Smith <johnsmith@example.com>"
```

Вы также можете указать шаблон, который Git будет использовать для поиска предыдущих авторов:

```
git commit -m "msg" --author "John"
```

В этом случае будет использоваться информация автора из последней фиксации с автором, содержащим «Джон».

На GitHub коммиты, сделанные одним из вышеперечисленных способов, покажут большой авторский миниатюру, а коммиттер будет меньше и впереди:



Задание изменений в определенных файлах

Вы можете зафиксировать изменения, внесенные в определенные файлы, и пропустить их, используя git add :

```
git commit file1.c file2.h
```

Или вы можете сначала сфабриковать файлы:

```
git add file1.c file2.h
```

и передать их позже:

```
git commit
```

Хорошие сообщения о фиксации

Важно, чтобы кто-то проходил через git log чтобы легко понять, в чем заключается каждая фиксация. Хорошие сообщения о совершении сообщения обычно включают в себя ряд задач или проблем в трекер и краткое описание того, что было сделано и почему, а иногда и как это было сделано.

Более эффективные сообщения могут выглядеть так:

```
TASK-123: Implement login through OAuth
TASK-124: Add auto minification of JS/CSS files
TASK-125: Fix minifier error when name > 200 chars
```

В то время как следующие сообщения не были бы столь полезными:

```
fix                          // What has been fixed?
just a bit of a change       // What has changed?
TASK-371                     // No description at all, reader will need to look at the tracker
themselves for an explanation
Implemented IFoo in IBar     // Why it was needed?
```

Способ проверить, записано ли сообщение фиксации в правильном настроении, - заменить пустой сообщение и посмотреть, имеет ли смысл:

Если я добавлю это коммит, я буду ___ в мой репозиторий.

Семь правил большого сообщения git commit

1. Отделите строку темы от тела пустой строкой
2. Ограничьте строку темы до 50 символов
3. Заглавие строки темы
4. Не завершайте строку темы с периодом
5. Используйте императивное настроение в строке темы
6. Вручную оберните каждую линию тела 72 символами
7. Используйте тело, чтобы объяснить, *что* и *почему*, а не *как*

7 правил из блога Криса Бима .

Фиксирование на определенную дату

```
git commit -m 'Fix UI bug' --date 2016-07-01
```

Параметр --date задает *дату автора* . Эта дата появится, например, в стандартном выпуске журнала git log .

Чтобы принудительно установить *дату фиксации* :

```
GIT_COMMITTER_DATE=2016-07-01 git commit -m 'Fix UI bug' --date 2016-07-01
```

Параметр date принимает гибкие форматы, поддерживаемые датой GNU, например:

```
git commit -m 'Fix UI bug' --date yesterday
git commit -m 'Fix UI bug' --date '3 days ago'
git commit -m 'Fix UI bug' --date '3 hours ago'
```

Когда дата не укажет время, будет использовано текущее время, и только дата будет переопределена.

Выбор, какие строки должны быть поставлены для совершения

Предположим, что у вас много изменений в одном или нескольких файлах, но из каждого файла вы хотите только зафиксировать некоторые изменения, вы можете выбрать нужные изменения, используя:

```
git add -p
```

или же

```
git add -p [file]
```

Каждое из ваших изменений будет отображаться индивидуально, и для каждого изменения вам будет предложено выбрать один из следующих вариантов:

```
y - Yes, add this hunk

n - No, don't add this hunk

d - No, don't add this hunk, or any other remaining hunks for this file.
    Useful if you've already added what you want to, and want to skip over the rest.

s - Split the hunk into smaller hunks, if possible

e - Manually edit the hunk.  This is probably the most powerful option.
    It will open the hunk in a text editor and you can edit it as needed.
```

Это будет составлять части файлов, которые вы выберете. Затем вы можете выполнить все поэтапные изменения следующим образом:

```
git commit -m 'Commit Message'
```

Изменения, которые не были поставлены или зафиксированы, по-прежнему будут отображаться в ваших рабочих файлах и могут быть совершены позже, если потребуется. Или, если остальные изменения нежелательны, их можно отбросить с помощью:

```
git reset --hard
```

Помимо разложения больших изменений на мелкие коммиты, этот подход также полезен для *рассмотрения* того, что вы собираетесь совершить. Индивидуально подтверждая каждое изменение, у вас есть возможность проверить, что вы написали, и можете избежать случайного размещения нежелательного кода, такого как инструкции println / logging.

Изменение времени фиксации

Вы изменяете время фиксации, используя

```
git commit --amend --date="Thu Jul 28 11:30 2016 -0400"
```

или даже

```
git commit --amend --date="now"
```

Изменение автора фиксации

Если вы сделаете фиксацию как неправильный автор, вы можете изменить ее, а затем изменить

```
git config user.name "Full Name"
git config user.email "email@example.com"

git commit --amend --reset-author
```

Подписание GPG

1. Определите свой идентификатор ключа

   ```
   gpg --list-secret-keys --keyid-format LONG

   /Users/davidcondrey/.gnupg/secring.gpg
   ------------------------------------
   sec   2048R/YOUR-16-DIGIT-KEY-ID YYYY-MM-DD [expires: YYYY-MM-DD]
   ```

Ваш идентификатор представляет собой буквенно-цифровой 16-значный код, следующий за первой косой чертой.

2. Определите свой идентификатор ключа в настройке git

```
git config --global user.signingkey YOUR-16-DIGIT-KEY-ID
```

3. Начиная с версии 1.7.9, git commit принимает параметр -S для прикрепления подписи к вашим записям. С помощью этой опции будет предложено ввести ключевую фразу GPG и добавить свою подпись в журнал фиксации.

```
git commit -S -m "Your commit message"
```

Прочитайте совершение онлайн: https://riptutorial.com/ru/git/topic/323/совершение

глава 57: сращивание

Синтаксис

- `git merge` **another_branch** `[опции]`
- `git merge` **--bort**

параметры

параметр	подробности
`-m`	Сообщение, которое должно быть включено в фиксацию слияния
`-v`	Показать подробный вывод
`--abort`	Попытайтесь вернуть все файлы обратно в их состояние
`--ff-only`	Прерывается мгновенно, когда требуется слияние.
`--no-ff`	Принуждает к созданию слияния, даже если это не является обязательным
`--no-commit`	Притворяется, что слияние не позволило проверить и настроить результат
`--stat`	Показывать diffstat после завершения слияния
`-n` / `--no-stat`	Не показывать diffstat
`--squash`	Позволяет выполнить одно фиксацию в текущей ветке с объединенными изменениями

Examples

Объединить одну ветку в другую

```
git merge incomingBranch
```

Это объединяет ветвь incomingBranch в ветку, в которой вы сейчас находитесь. Например, если вы в настоящий момент находитесь в master , то incomingBranch будет объединен в master .

Слияние может создавать конфликты в некоторых случаях. Если это произойдет, вы увидите сообщение « Automatic merge failed; fix conflicts and then commit the result. Вам нужно будет вручную отредактировать конфликтующие файлы или отменить попытку слияния, запустите:

```
git merge --abort
```

Автоматическое слияние

Когда коммиты на двух ветвях не конфликтуют, Git может автоматически объединить их:

```
~/Stack Overflow(branch:master) » git merge another_branch
Auto-merging file_a
Merge made by the 'recursive' strategy.
 file_a | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Прерывание слияния

После запуска слияния вы можете остановить слияние и вернуть все в свое состояние предварительного слияния. Использовать --abort :

```
git merge --abort
```

Сохранять изменения только с одной стороны слияния

Во время слияния вы можете передать --ours или - --theirs git checkout чтобы принять все изменения для файла с одной стороны или другого слияния.

```
$ git checkout --ours   -- file1.txt # Use our version of file1, delete all their changes
$ git checkout --theirs -- file2.txt # Use their version of file2, delete all our changes
```

Объединить с фиксацией

Поведение по умолчанию - это когда слияние разрешается как перемотка вперед, только обновляет указатель ветвления, не создавая фиксацию слияния. Используйте --no-ff для разрешения.

git merge <branch_name> --no-ff -m "<commit message>"

Поиск всех ветвей без слияния изменений

Иногда у вас могут быть ветви, лежащие вокруг, которые уже сменили их изменения на мастера. Это находит все ветви, которые не являются master , которые не имеют уникальных коммитов по сравнению с master . Это очень полезно для поиска ветвей, которые не были удалены после того, как PR был объединен с мастером.

```
 for branch in $(git branch -r) ; do
   [ "${branch}" != "origin/master" ] && [ $(git diff master...${branch} | wc -l) -eq 0 ] &&
echo -e `git show --pretty=format:"%ci %cr" $branch | head -n 1`\\t$branch
 done | sort -r
```

Прочитайте сращивание онлайн: https://riptutorial.com/ru/git/topic/291/сращивание

глава 58: Статистика Git

Синтаксис

- `git log [<options>] [<диапазон модификаций>] [[-] <путь>]`
- `git log --pretty = короткий | git shortlog [<options>]`
- `git shortlog [<options>] [<диапазон версий>] [[-] <путь>]`

параметры

параметр	подробности
`-n` , `--numbered`	Сортировка результата в соответствии с количеством коммитов на автора вместо алфавитного порядка
`-s` , `--summary`	Предоставлять только подсчет количества коммандов
`-e` , `--email`	Показать адрес электронной почты каждого автора
`--format` [= <формат>]	Вместо темы фиксации используйте некоторую другую информацию для описания каждого фиксации. <format> может быть любой строкой, принятой с помощью опции `--format git log` .
`-w` [<ширина> [, <indent1> [, <indent2>]]]	Линейный вывод вывода путем обертывания каждой строки по `width` . Первая строка каждой записи имеет отступы от `indent1` число пробелов, а последующие строки `indent2` промежуткам `indent2` .
<диапазон изменений>	Показывать только коммиты в указанном диапазоне версий. По умолчанию вся история до текущей фиксации.
[`--`] <путь>	Показывайте только коммиты, которые объясняют, как совпадают файлы, соответствующие `path` . Возможно, для путей можно использовать префикс «-», чтобы отделить их от параметров или диапазона ревизий.

Examples

Фиксирует за разработчика

Git shortlog используется для суммирования выходов журнала git и группировки коммитов по автору.

По умолчанию отображаются все сообщения фиксации, но аргумент --summary или -s пропускает сообщения и дает список авторов с их общим количеством коммитов.

--numbered или -n изменяет порядок от алфавита (по возрастанию автора) до числа спускаемых --numbered .

```
git shortlog -sn        #Names and Number of commits
```

```
git shortlog -sne       #Names along with their email ids and the Number of commits
```

или же

```
git log --pretty=format:%ae \
| gawk -- '{ ++c[$0]; } END { for(cc in c) printf "%5d %s\n",c[cc],cc; }'
```

Примечание. Записи одного и того же лица не могут быть сгруппированы вместе, если их имя и / или адрес электронной почты были написаны по-разному. Например, John Doe и Johnny Doe будут отображаться отдельно в списке. Чтобы решить эту проблему, обратитесь к функции .mailmap .

Фиксирует дату

```
git log --pretty=format:"%ai" | awk '{print " : "$1}' | sort -r | uniq -c
```

Общее количество фиксаций в филиале

```
git log  --pretty=oneline |wc -l
```

Список каждой ветви и дата последней ревизии

```
for k in `git branch -a | sed s/^..//`; do echo -e `git log -1 --pretty=format:"%Cgreen%ci
%Cblue%cr%Creset" $k --`\\t"$k";done | sort
```

Строки кода для каждого разработчика

```
git ls-tree -r HEAD | sed -Ee 's/^.{53}//' | \
while read filename; do file "$filename"; done | \
grep -E ': .*text' | sed -E -e 's/: .*//' | \
while read filename; do git blame --line-porcelain "$filename"; done | \
sed -n 's/^author //p' | \
sort | uniq -c | sort -rn
```

Список всех коммитов в хорошем формате

```
git log --pretty=format:"%Cgreen%ci %Cblue%cn  %Cgreen%cr%Creset %s"
```

Это даст хороший обзор всех коммитов (1 на строку) с сообщением о дате, пользователе и фиксации.

Параметр --pretty имеет много заполнителей, каждый из которых начинается с % . Все варианты можно найти здесь

Найти все локальные хранилища Git на компьютере

Чтобы перечислить все местоположения репозитория git, вы можете запустить следующие

```
find $HOME -type d -name ".git"
```

Если у вас есть locate , что это должно быть намного быстрее:

```
locate .git |grep git$
```

Если у вас есть gnu locate или mlocate , это будет выбирать только git dirs:

```
locate -ber \\.git$
```

Показывать общее количество фиксаций на автора

Чтобы получить общее количество git shortlog сделанных каждым разработчиком или вкладчиком в репозитории, вы можете просто использовать git shortlog :

```
git shortlog -s
```

который дает имена авторов и количество фиксаций каждым из них.

Кроме того, если вы хотите, чтобы результаты были рассчитаны для всех ветвей, добавьте в команду флаг --all :

```
git shortlog -s --all
```

Прочитайте Статистика Git онлайн: https://riptutorial.com/ru/git/topic/4609/статистика-git

глава 59: тянущий

Вступление

В отличие от нажатия Git, где ваши локальные изменения отправляются на сервер центрального репозитория, потянув с помощью Git, текущий код на сервере и «вытаскивает» его из сервера хранилища на ваш локальный компьютер. В этом разделе описывается процесс вытягивания кода из репозитория с использованием Git, а также ситуаций, которые могут возникнуть при переносе кода в локальную копию.

Синтаксис

- `git pull [опции [<репозиторий> [<refspec> ...]]`

параметры

параметры	подробности
`--quiet`	Нет текстового вывода
`-q`	сокращение для `--quiet`
`--verbose`	подробный вывод текста. Передано для извлечения и объединения / переупорядочения команд соответственно.
`-v`	стенография для `--verbose`
`--[no-]recurse-submodules[=yes | on-demand | no]`	Получить новые коммиты для подмодулей? (Не то, чтобы это не вытягивание / проверка)

замечания

`git pull` запускает `git fetch` с заданными параметрами и вызывает `git merge` чтобы объединить полученные ветви ветви в текущую ветку.

Examples

Обновление с локальными изменениями

Когда локальные изменения присутствуют, команда `git pull` отменяет отчетность:

```
Ошибка: ваши локальные изменения в следующих файлах будут перезаписаны слиянием
```

Чтобы обновить (например, обновление svn с помощью subversion), вы можете запустить:

```
git stash
git pull --rebase
git stash pop
```

Удобным способом может быть определение псевдонима с использованием:

2,9

```
git config --global alias.up '!git stash && git pull --rebase && git stash pop'
```

2,9

```
git config --global alias.up 'pull --rebase --autostash'
```

Затем вы можете просто использовать:

```
git up
```

Вытянуть код из удаленного

```
git pull
```

Вытянуть, перезаписать локальный

```
git fetch
git reset --hard origin/master
```

Остерегайтесь: в то время как reset --hard сбрасывается с помощью reset --hard может быть восстановлена с использованием reflog и reset , незафиксированные изменения удаляются навсегда.

Измените origin и master на удаленный и ветвь, которые вы хотите принудительно вывести, соответственно, если они названы по-разному.

Сохранение линейной истории при вытягивании

Сбрасывание при вытягивании

Если вы извлекаете новые коммиты из удаленного репозитория, и у вас есть локальные изменения в текущей ветви, тогда git автоматически объединит удалённую версию и вашу версию. Если вы хотите уменьшить количество слияний в своем филиале, вы можете сказать git, чтобы переустановить ваши коммиты на удаленной версии ветки.

```
git pull --rebase
```

Поведение по умолчанию

Чтобы сделать это по умолчанию для вновь созданных ветвей, введите следующую команду:

```
git config branch.autosetuprebase always
```

Чтобы изменить поведение существующей ветки, используйте следующую команду:

```
git config branch.BRANCH_NAME.rebase true
```

А также

```
git pull --no-rebase
```

Выполнять нормальное слияние.

Проверьте, есть ли возможность быстрой пересылки

Чтобы разрешить быструю переадресацию локальной ветви, вы можете использовать:

```
git pull --ff-only
```

При этом будет отображаться ошибка, когда локальная ветка не является быстрой переадресацией, и ее необходимо либо переустановить, либо объединить с восходящим потоком.

Потяните, «разрешение отклонено»

Некоторые проблемы могут возникнуть, если папка .git имеет неправильное разрешение. Устранить эту проблему, установив владельца полной папки .git . Иногда бывает, что другой пользователь тянет и меняет права на .git папку или файлы.

Решить проблему:

```
chown -R youruser:yourgroup .git/
```

Вытягивание изменений в локальный репозиторий

Простое натяжение

Когда вы работаете с удаленным репозиторием (например, GitHub) с кем-то другим, вы в какой-то момент захотите поделиться своими изменениями с ними. Как только они переместили свои изменения в удаленный репозиторий, вы можете получить эти изменения, *потянув* из этого репозитория.

```
git pull
```

Сделаю это, в большинстве случаев.

Потяните с другого пульта или ветви

Вы можете тянуть изменения с другого удаленного или ветви, указав их имена

```
git pull origin feature-A
```

Вытащить feature-A ветки feature-A origin формы в вашу локальную ветку. Обратите внимание, что вы можете напрямую указывать URL вместо удаленного имени и имя объекта, например, SHA вместо имени филиала.

Ручная тяга

Чтобы имитировать поведение git pull, вы можете использовать git fetch затем git merge

```
git fetch origin # retrieve objects and update refs from origin
git merge origin/feature-A # actually perform the merge
```

Это может дать вам больше контроля и позволяет вам проверять удаленную ветвь перед ее объединением. Действительно, после извлечения вы можете видеть удаленные ветви с git branch -a и проверить их с помощью

```
git checkout -b local-branch-name origin/feature-A # checkout the remote branch
# inspect the branch, make commits, squash, ammend or whatever
```

```
git checkout merging-branches # moving to the destination branch
git merge local-branch-name # performing the merge
```

Это может быть очень удобно при обработке запросов на загрузку.

Прочитайте тянущий онлайн: https://riptutorial.com/ru/git/topic/1308/тянущий

глава 60: Утилизация локального и удаленного хранилища

Examples

Удаление локальных ветвей, удаленных на удаленном компьютере

Для удаленного отслеживания между локальными и удаленными удаленными ветвями используйте

```
git fetch -p
```

вы можете использовать

```
git branch -vv
```

чтобы увидеть, какие ветви больше не отслеживаются.

Филиалы, которые больше не отслеживаются, будут представлены ниже:

```
 branch               12345e6 [origin/branch: gone] Fixed bug
```

вы можете использовать комбинацию из приведенных выше команд, ища, где «git branch -vv» возвращает «нет», а затем используя «-d» для удаления ветвей

```
git fetch -p && git branch -vv | awk '/: gone]/{print $1}' | xargs git branch -d
```

Прочитайте Утилизация локального и удаленного хранилища онлайн:
https://riptutorial.com/ru/git/topic/10934/утилизация-локального-и-удаленного-хранилища

глава 61: Шоу

Синтаксис

- git show [опции] <объект> ...

замечания

Показывает различные объекты Git.

- Для коммитов отображает сообщение фиксации и разницу
- Для тегов отображается сообщение тега и ссылочный объект

Examples

обзор

git show показывает различные объекты Git.

Для коммиттов:

Показывает сообщение о фиксации и разницу введенных изменений.

команда	Описание
git show	показывает предыдущую фиксацию
git show @~3	показывает 3-е-последнее сообщение

Для деревьев и капель:

Показывает дерево или blob.

команда	Описание
git show @~3:	показывает корневой каталог проекта, поскольку это было 3 месяца назад (дерево)
git show @~3:src/program.js	показывает src/program.js поскольку это было 3 месяца назад (blob)
git show @:a.txt @:b.txt	показывает a.txt с b.txt из текущей фиксации

Для тегов:

Показывает сообщение тега и связанный объект.

Прочитайте Шоу онлайн: https://riptutorial.com/ru/git/topic/3030/шоу

кредиты

S. No	Главы	Contributors
1	Начало работы с Git	Ajedi32, Ala Eddine JEBALI, Allan Burleson, Amitay Stern, Andy Hayden, AnimiVulpis, ArtOfWarfare, bahrep, Boggin, Brian, Community, Craig Brett, Dan Hulme, ericdwang, eykanal, Fernando Hoces De La Guardia, Fred Barclay, Henrique Barcelos, intboolstring, Irfan, Jackson Blankenship, janos, Jav_Rock, jeffdill2, JonasCz, JonyD, Joseph Dasenbrock, Kageetai, Karthik , KartikKannapur, Kayvan N, Knu, Lambda Ninja, maccard, Marek Skiba, Mateusz Piotrowski, Mingle Li, mouche, Nathan Arthur, Neui, NRKirby, ob1, ownsourcing dev training, Pod, Prince J, RamenChef, Rick, Roald Nefs, ronnyfm, Sazzad Hissain Khan, Scott Weldon, Sibi Raj, TheDarkKnight, theheadofabroom, ʇolɐəz əɥʇ qoq, Tot Zam, Tyler Zika, tymspy, Undo, VonC
2	.mailmap file: Связанные авторы и псевдонимы электронной почты	Mario, Michael Plotke
3	Diff-дерево	fybw id
4	Git Clean	gnis, MayeulC, n0shadow, pktangyue, Priyanshu Shekhar, Ralf Rafael Frix
5	Git Diff	Aaron Critchley, Abhijeet Kasurde, Adi Lester, anderas, apidae, Brett, Charlie Egan, eush77, □□□□□, intboolstring, Jack Ryan, JakeD, Jakub Narębski, jeffdill2, Joseph K. Strauss, khanmizan, Luke Taylor, Majid, mnoronha, Nathaniel Ford, Ogre Psalm33, orkoden, Ortomala Lokni, penguincoder, pylang, SurDin, Will, ydaetskcoR, Zaz
6	Git Patch	Dartmouth, Liju Thomas
7	Git Remote	AER, ambes, Dániel Kis, Dartmouth, Elizabeth, Jav_Rock, Kalpit, RamenChef, sonali, sunkuet02
8	Git rerere	Isak Combrinck
9	git send-email	Aaron Skomra, Dong Thang, fybw id, Jav_Rock, kofemann
10	Git Tagging	Atul Khanduri, demonplus, TheDarkKnight
11	Reflog - восстановление коммитов, не показанных в git log	Braiam, Peter Amidon, Scott Weldon
12	Rev-List	mkasberg
13	TortoiseGit	Julian, Matas Vaitkevicius
14	Worktrees	andipla, Configure, Victor Schröder
15	Анализ типов рабочих процессов	Boggin, Configure, Daniel Käfer, Dimitrios Mistriotis, forresthopkinsa, hardmooth, Horen, Kissaki, Majid, Sardathrion, Scott Weldon

16	Архив	Dartmouth, forevergenin, Neto Buenrostro, RamenChef
17	Биссекция / поиск ошибочных обязательств	4444, Hannoun Yassir, jornh, Kissaki, MrTux, Scott Weldon, Simone Carletti, zebediah49
18	Внешние слияния и диффицирования	AesSedai101, Micha Wiedenmann
19	Внутренности	nighthawk454
20	Восстановление	Creative John, Hardik Kanjariya ツ, Julie David, kisanme, ANAYI TIS, Scott Weldon, strangeqargo, Zaz
21	Выбор вишни	Atul Khanduri, Braiam, bud-e, dubek, Florian Hämmerle, intboolstring, Julian, kisanme, Lochlan, mpromonet, RedGreenCode
22	ГИТ-SVN	Bryan, Randy, Ricardo Amores, RobPethi
23	ГИТ-ТФС	Boggin, Kissaki
24	давя	adarsh, ams, AndiDog, bandi, Braiam, Caleb Brinkman, eush77, georgebrock, jpkrohling, Julian, Mateusz Piotrowski, Ortomala Lokni, RamenChef, Tall Sam, WMios
25	Игнорирование файлов и папок	AER, AesSedai101, agilob, Alex, Amitay Stern, AnimiVulpis, Ates Goral, Aukhan, Avamander, Ben, bpoiss, Braiam, bwegs, Cache Staheli, Collin M, Community, Dartmouth, David Grayson, Devesh Saini, Dheeraj vats, eckes, Ed Cottrell, enrico.bacis, Everettss, Fabio, fracz, Franck Dernoncourt, Fred Barclay, Functino, geek1011, Guillaume Pascal, HerrSerker, intboolstring, Irfan, Jakub Narębski, Jeff Puckett, Jens, joaquinlpereyra, John Slegers, JonasCz, Jörn Hees, joshng, Kačer, Kapep, Kissaki, knut, LeftRight92, Mackattack, Marvin, Matt, MayeulC, Mitch Talmadge, Narayan Acharya, Nathan Arthur, Neui, noʇʎpʎzɐɔ, Nuri Tasdemir, Ortomala Lokni, PaladiN, Panda, peci1, pktangyue, poke, pylang, RhysO, Rick, rokonoid, Sascha, Scott Weldon, Sebastianb, SeeuD1, sjas, Slayther, SnoringFrog, spikeheap, theJollySin, Toby, ʇolɐəz əuʇ qoq, Tom Gijselinck, Tomasz Bąk, Vi., Victor Schröder, VonC, Wilfred Hughes, Wolfgang, ydaetskcoR, Yosvel Quintero, Yury Fedorov, Zaz, Zeeker
26	Изменить имя репозитория git	xiaoyaoworm
27	инсценировка	AesSedai101, Andy Hayden, Asaph, Confiqure, intboolstring, Jakub Narębski, jkdev, Muhammad Abdullah, Nathan Arthur, ownsourcing dev training, Richard Dally, Wolfgang
28	Использование файла .gitattributes	Chin Huang, dahlbyk, Toby
29	История перезаписи с фильтром	gavinbeatty, gavv, Glenn Smith
30	Клиенты Git GUI	Alu, Daniel Käfer, Greg Bray, Nemanja Trifunovic, Pedro Pinheiro
31	Клонирование репозиториев	AER, Andrea Romagnoli, Andy Hayden, Blundering Philosopher, Dartmouth, Ezra Free, ganesshkumar, □□□□□, kartik, KartikKannapur, mnoronha, Peter Mitrano, pkowalczyk, Rick, Undo

		, Wojciech Kazior
32	конфигурация	APerson, Asenar, Cache Staheli, Chris Rasys, e.doroskevic, Julian, Liyan Chang, Majid, Micah Smith, Ortomala Lokni, Peter Mitrano, Priyanshu Shekhar, Scott Weldon, VonC, Wolfgang
33	Крючки	AesSedai101, AnoE, Christiaan Maks, Confiqure, Eidolon, Flows, fracz, kaartic, lostphilosopher, mwarsco
34	Крючки с клиентской стороны Git	Kelum Senanayake, kiamlaluno
35	Миграция в Git	AesSedai101, Boggin, Confiqure, Guillaume Pascal, Indregaard, Rick, TheDarkKnight
36	Нажимать	AER, Cody Guldner, cringe, frlan, Guillaume, intboolstring, Mário Meyrelles, Marvin, Matt S, MayeulC, pcm, pogosama, Thomas Gerot, Tomás Cañibano
37	Название Git Branch на Bash Ubuntu	Manishh
38	Обвинять	fracz, Matthew Hallatt, nighthawk454, Priyanshu Shekhar, WPrecht
39	Обновить имя объекта в ссылке	Keyur Ramoliya, RamenChef
40	Основное хранилище файлов Git (LFS)	Alex Stuckey, Matthew Hallatt, shoelzer
41	Отобразить историю фиксации графически с помощью Gitk	orkoden
42	перебазировка	AER, Alexander Bird, anderas, Ashwin Ramaswami, Braiam, BusyAnt, Confiqure, Daniel Käfer, Derek Liu, Dunno, e.doroskevic, Enrico Campidoglio, eskwayrd, □□□□□, Hugo Ferreira, intboolstring, Jeffrey Lin, Joel Cornett, Joseph K. Strauss, jtbandes, Julian, Kissaki, LeGEC, Libin Varghese, Luca Putzu, lucash, madhukar93, Majid, Matt, Matthew Hallatt, Menasheh, Michael Mrozek, Nemanja Boric, Ortomala Lokni, Peter Mitrano, pylang, Richard, takteek, Travis, Victor Schröder, VonC, Wasabi Fan, yarons, Zaz
43	Переименование	bud-e, Karan Desai, P.J.Meisch, PhotometricStereo
44	Поддеревья	4444, Jeff Puckett
45	подмодули	321hendrik, Chin Huang, ComicSansMS, foraidt, intboolstring, JF, kowsky, mpromonet, PaladiN, tinlyx, Undo, VonC
46	припрятать	aavrug, AesSedai101, Asaph, Brian Hinchey, bud-e, Cache Staheli, Deep, e.doroskevic, fracz, GingerPlusPlus, Guillaume, inkista, Jakub Narębski, Jarede, jeffdill2, joeytwiddle, Julie David, Kara, Koraktor, Majid, manasouza, Ortomala Lokni, Patrick, Peter Mitrano, Ralf Rafael Frix, Sebastianb, Tomás Cañibano, Wojciech Kazior
47	Просмотр истории	Ahmed Metwally, Andy Hayden, Aratz, Atif Hussain, Boggin, Brett, Confiqure, davidcondrey, Fabio, Flows, fracz, Fred Barclay, guleria, intboolstring, janos, jaredr, Kamiccolo, KraigH, LeGEC

		, manasouza, Matt Clark, Matthew Hallatt, MByD, mpromonet, Muhammad Abdullah, Noah, Oleander, Pedro Pinheiro, RedGreenCode, Toby Allen, Vogel612, ydaetskcoR
48	Псевдонимы	AesSedai101, Ajedi32, Andy, Anthony Staunton, Asenar, bstpierre, erewok, eush77, fracz, Gaelan, jrf, jtbandes, madhead, Michael Deardeuff, mickeyandkaka, nus, penguincoder, riyadhalnur, thanksd, Tom Hale, Wojciech Kazior, zinking
49	Пустые каталоги в Git	Ates Goral
50	Работа с пультом дистанционного управления	Boggin, Caleb Brinkman, forevergenin, heitortsergent, intboolstring, jeffdill2, Julie David, Kalpit, Matt Clark, MByD, mnoronha, mpromonet, mystarrocks, Pascalz, Raghav, Ralf Rafael Frix, Salah Eddine Lahniche, Sam, Scott Weldon, Stony, Thamilan, Vivin George, VonC, Zaz
51	разветвление	Amitay Stern, Andrew Kay, AnimiVulpis, Bad, BobTuckerman, Community, dan, Daniel Käfer, Daniel Stradowski, Deepak Bansal, djb, Don Kirkby, Duncan X Simpson, Eric Bouchut, forevergenin, fracz, Franck Dernoncourt, Fred Barclay, Frodon, gavv, Irfan, james large, janos, Jason, Joel Cornett, Jon Schneider, Jonathan, Joseph Dasenbrock, jrf, kartik, KartikKannapur, khanmizan, kirrmann, kisanme, Majid, Martin, MayeulC, Michael Richardson, Mihai, Mitch Talmadge, mkasberg, nepda, Noah, Noushad PP, Nowhere man, olegtaranenko, Ortomala Lokni, Ozair Kafray, PaladiN, ANAYITIS, Priyanshu Shekhar, Ralf Rafael Frix, Richard Hamilton, Robin, RudolphEst, Siavas, Simone Carletti, the12, Uwe, Vlad, wintersolider, Wojciech Kazior, Wolfgang, Yerko Palma, Yury Fedorov, zygimantus
52	развязывание	Adi Lester, AesSedai101, Alexander Bird, Andy Hayden, Boggin, brentonstrine, Brian, Colin D Bennett, ericdwang, Karan Desai, Matthew Hallatt, Nathan Arthur, Nathaniel Ford, Nithin K Anil, Pace, Rick, textshell, Undo, Zaz
53	Разрешение конфликтов слияния	Braiam, Dartmouth, David Ben Knoble, Fabio, nus, Vivin George, Yury Fedorov
54	Связки	jwd630
55	Синтаксис ревизий Git	Dartmouth, Jakub Narębski
56	совершение	Aaron Critchley, AER, Alan, Allan Burleson, Amitay Stern, Andrew Sklyarevsky, Andy Hayden, Anonymous Entity, APerson, bandi, Cache Staheli, Chris Forrence, Cody Guldner, cormacrelf, davidcondrey, Deep, depperm, ericdwang, Ethunxxx, Fred Barclay, George Brighton, Igor Ivancha, intboolstring, JacobLeach, James Taylor, janos, joeytwiddle, Jordan Knott, KartikKannapur, kisanme, Majid, Matt Clark, Matthew Hallatt, MayeulC, Micah Smith, Pod, Rick, Scott Weldon, SommerEngineering, Sonny Kim, Thomas Gerot, Undo, user1990366, vguzmanp, Vladimir F, Zaz
57	сращивание	brentonstrine, Liam Ferris, Noah, penguincoder, Undo, Vogel612, Wolfgang
58	Статистика Git	Dartmouth, Farhad Faghihi, Hugo Buff, KartikKannapur, lxer, penguincoder, RamenChef, SashaZd, Tyler Hyndman, vkluge
59	тянущий	Kissaki, MayeulC, mpromonet, rene, Ryan, Scott Weldon, Shog9, Stony, Thamilan, Thomas Gerot, Zaz

60	Утилизация локального и удаленного хранилища	Thomas Crowley
61	Шоу	Zaz