

GIT

Распределенная система управления версиями, созданная Линусом Торвальдсом для управления разработкой ядра Linux. Первая версия выпущена 7 апреля 2005 года.

- [Работа с изменениями](#)
- [Команды git](#)
 - [Внесение исправлений](#)
- [Основы git](#)
 - [Работа с тегами](#)
 - [Операции отмены](#)
 - [Работа с удалёнными репозиториями](#)
- [Распределённый Git](#)
 - [Сопровождение проекта](#)
- [Ветвление в Git](#)
 - [Работа с ветками](#)
 - [О ветвлении в двух словах](#)
- [Список используемых источников](#)
- [Инструменты работы с git](#)

Работа с изменениями

Просмотр изменений

Сводная информация

```
git status
```

Изменения относительно индекса

```
git diff
```

Изменения подготовленные для последующей их записи

```
git diff --cached
```

Смена ветки при наличии изменений

Смена ветви при наличии изменения происходит таким же способом как и обычная смена ветви.

```
git checkout -b <имя_ветки>
```

Отмена изменений

Полный сброс

```
git reset --hard <hash_commit or HEAD>
```

Сброс с сохранением изменений

```
git reset <hash_commit or HEAD>
```

Отмена изменений заданного файла

```
git checkout <имя_файла>
```

или

```
git checkout <hash_commit or HEAD> -- <имя_файла>
```

Отправка изменений

```
git push origin <имя_ветки>
```

В случае, когда требуется принудительно отправить изменения в ветку, независимо от её состояния, в конце добавляется параметр `--force`

```
git push origin <имя_ветки> --force
```

Скрытие изменений

Вы выполнили какие-нибудь изменения в файлах и хотите переключиться на другую ветку, но чтобы там не было ваших текущих изменений. С помощью команды `git stash` можно скрыть эти изменения. Ваши изменения помещаются в отдельное хранилище — в стек, а вы можете спокойно переключиться на другую ветку с дальнейшим извлечением скрытых изменений.

Скрытие изменений с добавлением комментария

```
git stash save "Комментарий"
```

Вывести список скрытых изменений

Самые старые скрытые изменения отображаются внизу списка, самые свежие сверху. Каждое скрытое изменение имеет идентификатор с номером, например, `stash@{0}`

```
git stash list
```

Применение скрытых изменений

Команда `git stash apply` берет самое свежее скрытое изменение (`stash@{0}`) и применяет его к текущему репозиторию. Это похоже на то, как вы применяете патч, только в качестве патча выступает ваше скрытое изменение.

```
git stash apply
```

Можно указать идентификатор для его конкретного применения.

```
git stash apply stash@{<stash_number>}
```

Применение скрытых изменений с их удалением из скрытого

Команда `git stash pop` сделать всё тоже самое, что и команда `git stash apply`, при этом удалив скрытые изменения из списка скрытых изменений.

```
git stash pop
```

```
git stash pop stash@{<stash_number>}
```

Обзор скрытых изменений

```
git stash show
```

```
git stash show stash@{<номер_скрытого_изменения>}
```

Полный список скрытых изменений

```
git stash show -p
```

Создание отдельной ветки из скрытых изменений

```
git stash branch <новое_имя_ветки>
```

```
git stash branch <новое_имя_ветки> stash@{<номер_скрытого_изменения>}
```

При этом скрытое изменение удаляется из списка скрытых изменений.

Удаление скрытых изменений

```
git stash drop
```

```
git stash drop stash@{<номер_скрытого_изменения>}
```

Удаление всех скрытых изменений

```
git stash clear
```

Список используемых источников

- [Полезные команды Git: безопасная отмена коммитов, добавление файла из другой ветки и другие](#)

Команды git

Внесение исправлений

Некоторые команды в Git основываются на подходе к рассмотрению коммитов в терминах внесённых ими изменений, т. е. рассматривают историю коммитов как цепочку патчей. Ниже перечислены эти команды.

git cherry-pick

Команда `git cherry-pick` берёт изменения, вносимые одним коммитом, и пытается повторно применить их в виде нового коммита в текущей ветке. Эта возможность полезна в ситуации, когда нужно забрать парочку коммитов из другой ветки, а не сливать ветку целиком со всеми внесёнными в неё изменениями.

Этот процесс описан и показан в разделе [Схема с перебазируванием и отбором](#) главы 5.

git rebase

`git rebase` — это «автоматизированный» `cherry-pick`. Он выполняет ту же работу, но для цепочки коммитов, тем самым как бы перенося ветку на новое место.

Мы в деталях разобрались с механизмом переноса веток в разделе [Перебазирование](#) главы 3, включая рассмотрение потенциальных проблем переноса опубликованных веток при совместной работе.

Мы использовали эту команду на практике для разбиения истории на два репозитория в разделе [Замена](#) главы 7, наряду с использованием флага `--onto`.

В разделе [Rerere](#) главы 7 мы рассмотрели случай возникновения конфликта во время переноса коммитов.

Также мы познакомились с интерактивным вариантом `git rebase`, включающемся с помощью опции `-i`, в разделе [Изменение сообщений нескольких коммитов](#) главы 7.

git revert

Команда `git revert` — полная противоположность `git cherry-pick`. Она создаёт новый коммит, который вносит изменения, противоположные указанному коммиту, по существу отменяя его.

Мы использовали её в разделе [Отмена коммита](#) главы 7 чтобы отменить коммит слияния (merge commit).

Основы git

Если вы хотите начать работать с Git, прочитав всего одну главу, то эта глава — то, что вам нужно. Здесь рассмотрены все базовые команды, необходимые вам для решения подавляющего большинства задач, возникающих при работе с Git. После прочтения этой главы вы научитесь настраивать и инициализировать репозиторий, начинать и прекращать контроль версий файлов, а также подготавливать и фиксировать изменения. Мы также продемонстрируем вам, как настроить в Git игнорирование отдельных файлов или их групп, как быстро и просто отменить ошибочные изменения, как просмотреть историю вашего проекта и изменения между отдельными коммитами (commit), а также как отправлять (push) и получать (pull) изменения в/из удалённого (remote) репозитория.

Работа с тегами

Как и большинство других систем контроля версий, Git имеет возможность помечать определённые моменты в истории как важные. Как правило, эта функциональность используется для отметки моментов выпуска версий (v1.0, и т. п.). Такие пометки в Git называются тегами. В этом разделе вы узнаете, как посмотреть имеющиеся теги, как создать новые или удалить существующие, а также какие типы тегов существуют в Git.

Просмотр списка тегов

Просмотреть список имеющихся тегов в Git можно очень просто. Достаточно набрать команду `git tag` (параметры `-l` и `--list` опциональны):

```
$ git tag
v1.0
v2.0
```

Данная команда перечисляет теги в алфавитном порядке; порядок их отображения не имеет существенного значения.

Так же можно выполнить поиск тега по шаблону. Например, репозиторий Git содержит более 500 тегов. Если вы хотите посмотреть теги выпусков 1.8.5, то выполните следующую команду:

```
$ git tag -l "v1.8.5*"
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

Для отображение тегов согласно шаблону требуются параметры `-l` или `--list`. Если вы хотите посмотреть весь список тегов, запуск команды `git tag` неявно подразумевает это и выводит полный список; использование параметров `-l` или `--list` в этом случае опционально. Если вы хотите отфильтровать список тегов согласно шаблону, использование параметров `-l` или `--list` становится обязательным.

Создание тегов

Git использует два основных типа тегов: легковесные и аннотированные.

Легковесный тег — это что-то очень похожее на ветку, которая не изменяется — просто указатель на определённый коммит.

А вот аннотированные теги хранятся в базе данных Git как полноценные объекты. Они имеют контрольную сумму, содержат имя автора, его e-mail и дату создания, имеют комментарий и могут быть подписаны и проверены с помощью GNU Privacy Guard (GPG). Обычно рекомендуется создавать аннотированные теги, чтобы иметь всю перечисленную информацию; но если вы хотите сделать временную метку или по какой-то причине не хотите сохранять остальную информацию, то для этого годятся и легковесные.

Аннотированные теги

Создание аннотированного тега в Git выполняется легко. Самый простой способ — это указать `-a` при выполнении команды `tag`:

```
$ git tag -a v1.4 -m "my version 1.4"
$ git tag
v0.1
```

v1.3

v1.4

Опция `-m` задаёт сообщение, которое будет храниться вместе с тегом. Если не указать сообщение, то Git запустит редактор, чтобы вы смогли его ввести.

С помощью команды `git show` вы можете посмотреть данные тега вместе с КОММИТОМ:

```
$ git show v1.4
tag v1.4
Tagger: Ben Straub <ben@straub.cc>
Date: Sat May 3 20:19:12 2014 -0700

my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date: Mon Mar 17 21:52:11 2008 -0700

Change version number
```

Здесь приведена информация об авторе тега, дате его создания и аннотирующее сообщение перед информацией о коммите.

Легковесные теги

Легковесный тег — это ещё один способ пометить коммит. По сути, это контрольная сумма коммита, сохранённая в файл — больше никакой информации не хранится. Для создания легковесного тега не передавайте опций `-a`, `-s` и `-m`, укажите только название:

```
$ git tag v1.4-lw
$ git tag
v0.1
v1.3
v1.4
v1.4-lw
```

На этот раз при выполнении `git show` для этого тега вы не увидите дополнительной информации. Команда просто покажет коммит:

```
$ git show v1.4-lw
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date: Mon Mar 17 21:52:11 2008 -0700

    Change version number
```

Отложенная расстановка тегов

Также возможно пометить уже пройденные коммиты. Предположим, история коммитов выглядит следующим образом:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 Create write support
0d52aaab4479697da7686c15f77a3d64d9165190 One more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc Add commit function
4682c3261057305bdd616e23b64b0857d832627b Add todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a Create write support
9fceb02d0ae598e95dc970b74767f19372d61af8 Update rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc Commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a Update readme
```

Теперь предположим, что вы забыли отметить версию проекта v1.2, которая была там, где находится коммит «Update rakefile». Вы можете добавить тег и позже. Для отметки коммита укажите его контрольную сумму (или её часть) как параметр команды:

```
$ git tag -a v1.2 9fceb02
```

Проверим, что коммит отмечен:

```
$ git tag
v0.1
v1.2
v1.3
v1.4
v1.4-lw
v1.5

$ git show v1.2
tag v1.2
Tagger: Scott Chacon <schacon@gee-mail.com>
Date: Mon Feb 9 15:32:16 2009 -0800

version 1.2
commit 9fceb02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date: Sun Apr 27 20:43:35 2008 -0700

    Update rakefile
...
```

Обмен тегами

По умолчанию, команда `git push` не отправляет теги на удалённые сервера. После создания теги нужно отправлять явно на удалённый сервер. Процесс аналогичен отправке веток — достаточно выполнить команду `git push origin <tagname>`.

```
$ git push origin v1.5
Counting objects: 14, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.
Total 14 (delta 3), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
* [new tag]      v1.5 -> v1.5
```


Если у вас много тегов, и вам хотелось бы отправить все за один раз, то можно использовать опцию `--tags` для команды `git push`. В таком случае все ваши теги отправятся на удалённый сервер (если только их уже там нет).

```
$ git push origin --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
* [new tag]      v1.4 -> v1.4
* [new tag]      v1.4-lw -> v1.4-lw
```

Теперь, если кто-то клонирует (`clone`) или выполнит `git pull` из вашего репозитория, то он получит вдобавок к остальному и ваши метки.

`git push` отправляет оба типа тегов. Отправка тегов командой `git push <remote> --tags` не различает аннотированные и легковесные теги. В настоящее время не существует опции чтобы отправить только легковесные теги, но если использовать команду `git push <remote> --follow-tags`, то отправятся только аннотированные теги.

Удаление тегов

Для удаления тега в локальном репозитории достаточно выполнить команду `git tag -d <tagname>`. Например, удалить созданный ранее легковесный тег можно следующим образом:

```
$ git tag -d v1.4-lw
Deleted tag 'v1.4-lw' (was e7d5add)
```

Обратите внимание, что при удалении тега не происходит его удаления с внешних серверов. Существует два способа изъятия тега из внешнего репозитория.

Первый способ — это выполнить команду `git push <remote> :refs/tags/<tagname>`:

```
$ git push origin :refs/tags/v1.4-lw
To /git@github.com:schacon/simplegit.git
- [deleted]      v1.4-lw
```

Это следует понимать как обновление внешнего тега пустым значением, что приводит к его удалению.

Второй способ убрать тег из внешнего репозитория более интуитивный:

```
$ git push origin --delete <tagname>
```

Переход на тег

Если вы хотите получить версии файлов, на которые указывает тег, то вы можете сделать `git checkout` для тега. Однако, это переведёт репозиторий в состояние «detached HEAD», которое имеет ряд неприятных побочных эффектов.

```
$ git checkout v2.0.0
```

```
Note: switching to 'v2.0.0'.
```

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-c` with the `switch` command. Example:

```
git switch -c <new-branch-name>
```

Or undo this operation with:

```
git switch -
```

Turn off this advice by setting config variable `advice.detachedHead` to false

```
HEAD is now at 99ada87... Merge pull request #89 from schacon/appendix-final
```

```
$ git checkout v2.0-beta-0.1
```

Previous HEAD position was 99ada87... Merge pull request #89 from schacon/appendix-final

HEAD is now at df3f601... Add atlas.json and cover image

Если в состоянии «detached HEAD» внести изменения и сделать коммит, то тег не изменится, при этом новый коммит не будет относиться ни к какой из веток, а доступ к нему можно будет получить только по его хешу. Поэтому, если вам нужно внести изменения — исправить ошибку в одной из старых версий — скорее всего вам следует создать ветку:

```
$ git checkout -b version2 v2.0.0
```

Switched to a new branch 'version2'

Если сделать коммит в ветке `version2`, то она сдвинется вперед и будет отличаться от тега `v2.0.0`, так что будьте с этим осторожны.

Операции отмены

В любой момент вам может потребоваться что-либо отменить. Здесь мы рассмотрим несколько основных способов отмены сделанных изменений. Будьте осторожны, не все операции отмены в свою очередь можно отменить! Это одна из редких областей Git, где неверными действиями можно необратимо удалить результаты своей работы.

Отмена может потребоваться, если вы сделали коммит слишком рано, например, забыв добавить какие-то файлы или комментарий к коммиту. Если вы хотите переделать коммит — внесите необходимые изменения, добавьте их в индекс и сделайте коммит ещё раз, указав параметр `--amend`:

```
$ git commit --amend
```

Эта команда использует область подготовки (индекс) для внесения правок в коммит. Если вы ничего не меняли с момента последнего коммита (например, команда запущена сразу после предыдущего коммита), то снимок состояния останется в точности таким же, а всё что вы сможете изменить — это ваше сообщение к коммиту.

Запустится тот же редактор, только он уже будет содержать сообщение предыдущего коммита. Вы можете редактировать сообщение как обычно, однако, оно заменит сообщение предыдущего коммита.

Например, если вы сделали коммит и поняли, что забыли проиндексировать изменения в файле, который хотели добавить в коммит, то можно сделать следующее:

```
$ git commit -m 'Initial commit'
$ git add forgotten_file
$ git commit --amend
```

В итоге получится единый коммит — второй коммит заменит результаты первого.

Очень важно понимать, что когда вы вносите правки в последний коммит, вы не столько исправляете его, сколько *заменяете* новым, который полностью его перезаписывает. В результате всё выглядит так, будто первоначальный коммит никогда не существовал, а так же он больше не появится в истории вашего репозитория. Очевидно, смысл изменения коммитов в добавлении незначительных правок в последние коммиты и, при этом, в избежании засорения истории сообщениями вида «Ой, забыл добавить файл» или «Исправление грамматической ошибки».

Отмена индексации файла

Следующие два раздела демонстрируют как работать с индексом и изменениями в рабочем каталоге. Радует, что команда, которой вы определяете состояние этих областей, также подсказывает вам как отменять изменения в них. Например, вы изменили два файла и хотите добавить их в разные коммиты, но случайно выполнили команду `git add *` и добавили в индекс оба. Как исключить из индекса один из них? Команда `git status` напомним вам:

```
$ git add *
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:   README.md -> README
    modified:  CONTRIBUTING.md
```

Прямо под текстом «Changes to be committed» говорится: используйте `git reset HEAD <file>...` для исключения из индекса. Давайте последуем этому совету и отменим индексирование файла `CONTRIBUTING.md`:

```
$ git reset HEAD CONTRIBUTING.md
Unstaged changes after reset:
M CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README

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:   CONTRIBUTING.md
```

Команда выглядит несколько странно, но — работает! Файл `CONTRIBUTING.md` изменен, но больше не добавлен в индекс.

Команда `git reset` *может* быть опасной если вызвать её с параметром `--hard`. В приведённом примере файл не был затронут, следовательно команда относительно безопасна.

На текущий момент этот магический вызов — всё, что вам нужно знать о команде `git reset`. Мы рассмотрим в деталях что именно делает `reset` и как с её помощью делать действительно интересные вещи в разделе [Раскрытие тайн reset](#) главы 7.

Отмена изменений в файле

Что делать, если вы поняли, что не хотите сохранять свои изменения файла `CONTRIBUTING.md`? Как можно просто отменить изменения в нём — вернуть к тому состоянию, которое было в последнем коммите (или к начальному после клонирования, или ещё как-то полученному)? Нам повезло, что `git status` подсказывает и это тоже.

В выводе команды из последнего примера список изменений выглядит примерно так:

```
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: CONTRIBUTING.md
```

Здесь явно сказано как отменить существующие изменения. Давайте так и сделаем:

```
$ git checkout -- CONTRIBUTING.md
```

```
$ git status
```

```
On branch master
```

```
Changes to be committed:
```

```
(use "git reset HEAD <file>..." to unstage)
```

```
renamed: README.md -> README
```

Как видите, откат изменений выполнен.

Важно понимать, что `git checkout -- <file>` — опасная команда. Все локальные изменения в файле пропадут — Git просто заменит его версией из последнего коммита. Ни в коем случае не используйте эту команду, если вы не уверены, что изменения в файле вам не нужны.

Если вы хотите сохранить изменения в файле, но прямо сейчас их нужно отменить, то есть способы получше, такие как ветвление и припрятывание — мы рассмотрим их в главе [Ветвление в Git](#).

Помните, всё что попало в *коммит* почти всегда Git может восстановить. Можно восстановить даже коммиты из веток, которые были удалены, или коммиты, перезаписанные параметром `--amend` (см. [Восстановление данных](#)). Но всё, что не было включено в коммит и потеряно — скорее всего, потеряно навсегда.

Отмена действий с помощью git restore

Git версии 2.23.0 представил новую команду: `git restore`. По сути, это альтернатива `git reset`, которую мы только что рассмотрели. Начиная с версии 2.23.0, Git будет использовать `git restore` вместо `git reset` для многих операций отмены.

Давайте проследим наши шаги и отменим действия с помощью `git restore` вместо `git reset`.

Отмена индексации файла с помощью git restore

В следующих двух разделах показано, как работать с индексом и изменениями рабочей копии с помощью `git restore`. Приятно то, что команда, которую вы используете для определения состояния этих двух областей, также напоминает вам, как отменить изменения в них. Например, предположим, что вы изменили два файла и хотите зафиксировать их как два отдельных изменения, но случайно набираете `git add *` и индексируете их оба. Как вы можете убрать из индекса один из двух? Команда `git status` напоминает вам:

```
$ git add *
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
   modified:   CONTRIBUTING.md
   renamed:   README.md -> README
```

Прямо под текстом «Changes to be committed», написано использовать `git restore --staged <file> ...` для отмены индексации файла. Итак, давайте воспользуемся этим советом, чтобы убрать из индекса файл `CONTRIBUTING.md`:


```
$ git restore --staged CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    renamed:   README.md -> README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   CONTRIBUTING.md
```

Файл `CONTRIBUTING.md` изменен, но снова не индексирован.

Откат изменённого файла с помощью `git restore`

Что, если вы поймете, что не хотите сохранять изменения в файле `CONTRIBUTING.md`? Как легко его откатить — вернуть обратно к тому, как он выглядел при последнем коммите (или изначально клонирован, или каким-либо образом помещён в рабочий каталог)? К счастью, `git status` тоже говорит, как это сделать. В выводе последнего примера, неиндексированная область выглядит следующим образом:

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   CONTRIBUTING.md
```

Он довольно недвусмысленно говорит, как отменить сделанные вами изменения. Давайте сделаем то, что написано:

```
$ git restore CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    renamed:   README.md -> README
```

Важно понимать, что `git restore <file>` — опасная команда. Любые локальные изменения, внесённые в этот файл, исчезнут — Git просто заменит файл последней зафиксированной версией. Никогда не используйте эту команду, если точно не знаете, нужны ли вам эти несохранённые локальные изменения.

Работа с удалёнными репозиториями

Для того, чтобы внести вклад в какой-либо Git-проект, вам необходимо уметь работать с удалёнными репозиториями. Удалённые репозитории представляют собой версии вашего проекта, сохранённые в интернете или ещё где-то в сети. У вас может быть несколько удалённых репозиториев, каждый из которых может быть доступен для чтения или для чтения-записи. Взаимодействие с другими пользователями предполагает управление удалёнными репозиториями, а также отправку и получение данных из них. Управление репозиториями включает в себя как умение добавлять новые, так и умение удалять устаревшие репозитории, а также умение управлять различными удалёнными ветками, объявлять их отслеживаемыми или нет и так далее. В данном разделе мы рассмотрим некоторые из этих навыков.

Удалённый репозиторий может находиться на вашем локальном компьютере. Вполне возможно, что удалённый репозиторий будет находиться на том же компьютере, на котором работаете вы. Слово «удалённый» не означает, что репозиторий обязательно должен быть где-то в сети или Интернет, а значит только — где-то ещё. Работа с таким удалённым репозиторием подразумевает выполнение стандартных операций отправки и получения, как и с любым другим удалённым репозиторием.

Просмотр удалённых репозиториев

Для того, чтобы просмотреть список настроенных удалённых репозиториев, вы можете запустить команду `git remote`. Она выведет названия доступных удалённых репозиториев. Если вы клонировали репозиторий, то увидите как минимум `origin` — имя по умолчанию, которое Git даёт серверу, с которого производилось клонирование:

```
$ git clone https://github.com/schacon/ticgit
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.
$ cd ticgit
$ git remote
origin
```

Вы можете также указать ключ `-v`, чтобы просмотреть адреса для чтения и записи, привязанные к репозиторию:

```
$ git remote -v
origin https://github.com/schacon/ticgit (fetch)
origin https://github.com/schacon/ticgit (push)
```

Если у вас больше одного удалённого репозитория, команда выведет их все. Например, для репозитория с несколькими настроенными удалёнными репозиториями в случае совместной работы нескольких пользователей, вывод команды может выглядеть примерно так:

```
$ cd grit
$ git remote -v
bakkdoor https://github.com/bakkdoor/grit (fetch)
bakkdoor https://github.com/bakkdoor/grit (push)
cho45 https://github.com/cho45/grit (fetch)
cho45 https://github.com/cho45/grit (push)
defunkt https://github.com/defunkt/grit (fetch)
defunkt https://github.com/defunkt/grit (push)
koke git://github.com/koke/grit.git (fetch)
koke git://github.com/koke/grit.git (push)
```

```
origin  git@github.com:mojombo/grit.git (fetch)
origin  git@github.com:mojombo/grit.git (push)
```

Это означает, что мы можем легко получить изменения от любого из этих пользователей. Возможно, что некоторые из репозиториях доступны для записи и в них можно отправлять свои изменения, хотя вывод команды не даёт никакой информации о правах доступа.

Обратите внимание на разнообразие протоколов, используемых при указании адреса удалённого репозитория; подробнее мы рассмотрим протоколы в разделе [Установка Git на сервер](#) главы 4.

Добавление удалённых репозиториях

В предыдущих разделах мы уже упоминали и приводили примеры добавления удалённых репозиториях, сейчас рассмотрим эту операцию подробнее. Для того, чтобы добавить удалённый репозиторий и присвоить ему имя (shortname), просто выполните команду `git remote add <shortname> <url>`:

```
$ git remote
origin
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin=https://github.com/schacon/ticgit (fetch)
origin=https://github.com/schacon/ticgit (push)
pb=https://github.com/paulboone/ticgit (fetch)
pb=https://github.com/paulboone/ticgit (push)
```

Теперь вместо указания полного пути вы можете использовать `pb`. Например, если вы хотите получить изменения, которые есть у Пола, но нету у вас, вы можете выполнить команду `git fetch pb`:

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
Unpacking objects: 100% (43/43), done.
From https://github.com/paulboone/ticgit
* [new branch]    master    -> pb/master
* [new branch]    ticgit    -> pb/ticgit
```

Ветка `master` из репозитория Пола сейчас доступна вам под именем `pb/master`. Вы можете слить её с одной из ваших веток или переключить на неё локальную ветку, чтобы просмотреть содержимое ветки Пола. Более подробно работа с ветками рассмотрена в главе [Ветвление в Git](#).

Отправка в несколько репозиториев одновременно

После добавления репозитория устанавливается два способа взаимодействия с ним - получение изменений и их отправка. Для отправки изменений в несколько репозиториев, необходимо выполнить команду.

```
$ git remote set-url --add --push origin <origin_repository_url>
$ git remote set-url --add --push origin <additional_repository_url>
```

где `origin` - имя репозитория, в который необходимо добавить возможность отправки изменений, `origin_repository_url` - исходный репозиторий, `additional_repository_url` - дополнительный репозиторий.

Получение изменений из удалённого репозитория — Fetch и Pull

Как вы только что узнали, для получения данных из удалённых проектов, следует выполнить:

```
$ git fetch [remote-name]
```

Данная команда связывается с указанным удалённым проектом и забирает все те данные проекта, которых у вас ещё нет. После того как вы выполнили команду, у вас должны появиться ссылки на все ветки из этого удалённого проекта, которые вы можете просмотреть или слить в любой момент.

Когда вы клонируете репозиторий, команда `clone` автоматически добавляет этот удалённый репозиторий под именем «origin». Таким образом, `git fetch origin` извлекает все наработки, отправленные на этот сервер после того, как вы его клонировали (или получили изменения с помощью `fetch`). Важно отметить, что команда `git fetch` забирает данные в ваш локальный репозиторий, но не сливает их с какими-либо вашими наработками и не модифицирует то, над чем вы работаете в данный момент. Вам необходимо вручную слить эти данные с вашими, когда вы будете готовы.

Если ветка настроена на отслеживание удалённой ветки (см. следующий раздел и главу [Ветвление в Git](#) чтобы получить больше информации), то вы можете использовать команду `git pull` чтобы автоматически получить изменения из удалённой ветки и слить их со своей текущей. Этот способ может для вас оказаться более простым или более удобным. К тому же, по умолчанию команда `git clone` автоматически настраивает вашу локальную ветку `master` на отслеживание удалённой ветки `master` на сервере, с которого вы клонировали репозиторий. Название веток может быть другим и зависит от ветки по умолчанию на сервере. Выполнение `git pull`, как правило, извлекает (`fetch`) данные с сервера, с которого вы изначально клонировали, и автоматически пытается слить (`merge`) их с кодом, над которым вы в данный момент работаете.

Начиная с версии 2.27, команда `git pull` выдаёт предупреждение, если настройка `pull.rebase` не установлена. Git будет выводить это предупреждение каждый раз пока настройка не будет установлена. Если хотите использовать поведение Git по умолчанию (простое смещение вперёд если возможно — иначе создание коммита

слияния): `git config --global pull.rebase "false"`. Если хотите использовать перебазирование при получении изменений: `git config --global pull.rebase "true"`.

Отправка изменений в удалённый репозиторий (Push)

Когда вы хотите поделиться своими наработками, вам необходимо отправить их в удалённый репозиторий. Команда для этого действия простая: `git push <remote-name> <branch-name>`. Чтобы отправить вашу ветку `master` на сервер `origin` (повторимся, что клонирование обычно настраивает оба этих имени автоматически), вы можете выполнить следующую команду для отправки ВАШИХ КОММИТОВ:

```
$ git push origin master
```

Эта команда срабатывает только в случае, если вы клонировали с сервера, на котором у вас есть права на запись, и если никто другой с тех пор не выполнял команду `push`. Если вы и кто-то ещё одновременно клонируете, затем он выполняет команду `push`, а после него выполнить команду `push` попытаетесь вы, то ваш `push` точно будет отклонён. Вам придётся сначала получить изменения и объединить их с вашими и только после этого вам будет позволено выполнить `push`. Обратитесь к главе [Ветвление в Git](#) для более подробного описания, как отправлять изменения на удалённый сервер.

Просмотр удалённого репозитория

Если хотите получить побольше информации об одном из удалённых репозиториях, вы можете использовать команду `git remote show <remote>`.

Выполнив эту команду с некоторым именем, например, `origin`, вы получите следующий результат:

```
$ git remote show origin
* remote origin

Fetch URL: https://github.com/schacon/ticgit
Push URL: https://github.com/schacon/ticgit
HEAD branch: master

Remote branches:
  master                tracked
  dev-branch            tracked

Local branch configured for 'git pull':
  master merges with remote master

Local ref configured for 'git push':
  master pushes to master (up to date)
```

Она выдаёт URL удалённого репозитория, а также информацию об отслеживаемых ветках. Эта команда любезно сообщает вам, что если вы, находясь на ветке `master`, выполните `git pull`, ветка `master` с удалённого сервера будет автоматически влита в вашу сразу после получения всех необходимых данных. Она также выдаёт список всех полученных ею ссылок.

Это был пример для простой ситуации и вы наверняка встречались с чем-то подобным. Однако, если вы используете Git более интенсивно, вы можете увидеть гораздо большее количество информации от `git remote show`:

```
$ git remote show origin
* remote origin

URL: https://github.com/my-org/complex-project
Fetch URL: https://github.com/my-org/complex-project
Push URL: https://github.com/my-org/complex-project
HEAD branch: master

Remote branches:
  master                tracked
  dev-branch            tracked
  markdown-strip        tracked
  issue-43              new (next fetch will store in remotes/origin)
  issue-45              new (next fetch will store in remotes/origin)
  refs/remotes/origin/issue-11  stale (use 'git remote prune' to remove)
```

Local branches configured for 'git pull':

dev-branch merges with remote dev-branch

master merges with remote master

Local refs configured for 'git push':

dev-branch pushes to dev-branch (up to date)

markdown-strip pushes to markdown-strip (up to date)

master pushes to master (up to date)

Данная команда показывает какая именно локальная ветка будет отправлена на удалённый сервер по умолчанию при выполнении `git push`. Она также показывает, каких веток с удалённого сервера у вас ещё нет, какие ветки всё ещё есть у вас, но уже удалены на сервере, и для нескольких веток показано, какие удалённые ветки будут в них влиты при выполнении `git pull`.

Удаление и переименование удалённых репозиториев

Для переименования удалённого репозитория можно выполнить `git remote rename`. Например, если вы хотите переименовать `pb` в `paul`, вы можете это сделать при помощи `git remote rename`:

```
$ git remote rename pb paul
$ git remote
origin
paul
```

Стоит упомянуть, что это также изменит имена удалённых веток в вашем репозитории. То, к чему вы обращались как `pb/master`, теперь стало `paul/master`.

Если по какой-то причине вы хотите удалить удалённый репозиторий — вы сменили сервер или больше не используете определённое зеркало, или кто-то перестал вносить изменения — вы можете использовать `git remote rm`:

```
$ git remote remove paul
```

```
$ git remote
```

```
origin
```

При удалении ссылки на удалённый репозиторий все отслеживаемые ветки и настройки, связанные с этим репозиторием, так же будут удалены.

Распределённый Git

Теперь, когда вы обзавелись настроенным удалённым Git-репозиторием, т. е. местом, где разработчики могут обмениваться своим кодом, а также познакомились с основными командами Git для локальной работы, мы рассмотрим, как задействовать некоторые распределённые рабочие процессы, предлагаемые Git.

В этой главе мы рассмотрим работу с Git в распределённой среде как в роли рядового разработчика, так и в роли системного интегратора. То есть вы научитесь успешно вносить свой код в проект, делая это как можно более просто и для вас, и для владельца проекта, а также научитесь тому, как сопровождать проекты, в которых участвует множество разработчиков.

Сопровождение проекта

В дополнение к эффективному участию в проекте, было бы неплохо знать как его сопровождать. Сопровождение может включать в себя принятие и применение патчей, сгенерированных с помощью `format-patch` и отправленных вам по почте, или интеграцию изменений в ветках удалённых репозиторий. Независимо от того, поддерживаете ли вы канонический репозиторий или просто хотите помочь в проверке или применении патчей, вам необходимо знать каким образом следует принимать работу, чтобы это было наиболее понятно для других участников и было бы приемлемым для вас в долгосрочной перспективе.

Работа с тематическими ветками

Перед интеграцией новых изменений желательно проверить их в тематической ветке — временной ветке, специально созданной для проверки работоспособности новых изменений. Таким образом, можно применять патчи по одному и пропускать неработающие, пока не найдётся время к ним вернуться. Если вы создадите ветку с коротким и понятным названием, основанным на тематике изменений, например, `ruby_client` или что-то похожее, то без труда можно будет вернуться к ней, если пришлось на какое-то время отказаться от работы с ней. Сопровождающему Git проекта свойственно использовать пространство имен для веток, например, `sc/ruby_client`, где `sc` — это сокращение от имени того, кто проделал работу. Как известно, ветки можно создавать на основании базовой ветки, например:

```
$ git branch sc/ruby_client master
```

Если вы хотите сразу переключиться на создаваемую ветку, то используйте опцию `checkout -b`:

```
$ git checkout -b sc/ruby_client master
```

Теперь вы можете добавить новые изменения в созданную тематическую ветку и определить хотите ли слить эти изменения в ваши долгосрочные ветки.

Применение патчей, полученных по почте

Если вы получили патч по почте и его нужно интегрировать в проект, то следует проанализировать его, применив сначала в тематической ветке. Существует два варианта применения полученного по почте патча: `git apply` или `git am`.

Применение патча командой `apply`

Если полученный по почте патч был создан командой `git diff` или Unix командой `diff` (что не рекомендуется делать), то применить его можно командой `git apply`. Предположим, патч сохранен здесь `/tmp/patch-ruby-client.patch`, тогда применить его можно вот так:

```
$ git apply /tmp/patch-ruby-client.patch
```

Это действие модифицирует файлы в вашем рабочем каталоге. Выполнение команды практически эквивалентно выполнению команды `patch -p1`, однако, является более параноидальным и принимает меньше неточных совпадений, чем `patch`. При этом обрабатываются добавления, удаления и переименования файлов, указанные в формате `git diff`, тогда как `patch` этого не делает. Наконец, `git apply` использует модель «применить всё или отменить всё», где изменения либо применяются полностью, либо не применяются вообще, тогда как `patch` может частично применить патч файлы, приведя ваш рабочий каталог в непонятное состояние. В целом, `git apply`

более консервативен, чем `patch`. После выполнения команды новый коммит не создаётся и его нужно делать вручную.

Командой `git apply` можно проверить корректность применения патча до его фактического применения, используя `git apply --check`:

```
$ git apply --check 0001-see-if-this-helps-the-gem.patch
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

Если ошибок не выведено, то патч может быть применён без проблем. Так же, в случае ошибки эта команда возвращает отличное от 0 значение, что позволяет использовать её в скриптах.

Применение патча командой `am`

Если участник проекта пользователь Git и умеет пользоваться командой `format-patch` для генерации патчей, то вам будет легче, так как в патч включается информация об авторе и сообщение коммита. Если возможно, требуйте от ваших участников использовать команду `format-patch` вместо `diff` для генерации патчей. Вам останется использовать `git apply` только для устаревших патчей и подобного им.

Для применения патча, созданного с помощью `format-patch`, используйте `git am` (команда названа `am` потому что применяет «apply» набор патчей в формате «mailbox»). С технической точки зрения она просто читает mbox-файл, в котором в виде обычного текста хранится одно или несколько электронных писем. Этот файл имеет следующий вид:

```
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] Add limit to log function

Limit log functionality to the first 20
```

Это начало вывода команды `format-patch`, которая рассматривалась в предыдущем разделе; это так же представляет собой валидный формат

mbox. Если кто-то отправил патч, корректно сформированный командой `git send-email`, и вы сохранили его в формате mbox, то можно указать передать этот файл в качестве аргумента команде `git am`, которая начнёт применять все найденные в файле патчи. Если вы используете почтовый клиент, который умеет сохранять несколько писем в формате mbox, то можно сохранить сразу серию патчей в один файл, а затем применить их за раз, используя `git am`.

Так или иначе, если кто-нибудь загрузит созданный с помощью `format-patch` патч файл в систему управления задачами, то вы сможете сохранить его себе и применить локально с помощью `git am`:

```
$ git am 0001-limit-log-function.patch
Applying: Add limit to log function
```

Как вы могли заметить, патч применился без конфликтов, а так же был создан новый коммит. Информация об авторе была извлечена из заголовков письма `From` и `Date`, а сообщение коммита — из заголовка `Subject` и тела письма (до патча). Например, для применённого патча из примера выше коммит будет выглядеть следующим образом:

```
$ git log --pretty=fuller -1
commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Author:   Jessica Smith <jessica@example.com>
AuthorDate: Sun Apr 6 10:17:23 2008 -0700
Commit:   Scott Chacon <schacon@gmail.com>
CommitDate: Thu Apr 9 09:19:06 2009 -0700

    Add limit to log function

    Limit log functionality to the first 20
```

`Commit` информация указывает на того, кто применил патч и когда это было сделано. `Author` информация указывает на того, кто изначально создал патч и когда это было сделано.

Однако, возможна ситуация, когда патч не может быть бесконфликтно применён. Возможно, ваша основная ветка сильно расходится с той веткой, на основании которой был создан патч, или он зависит от другого, ещё не

применённого патча. В таком случае работа `git am` будет прервана, а так же выведена подсказка со списком возможных действий:

```
$ git am 0001-see-if-this-helps-the-gem.patch
Applying: See if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Patch failed at 0001.

When you have resolved this problem run "git am --resolved".
If you would prefer to skip this patch, instead run "git am --skip".
To restore the original branch and stop patching run "git am --abort".
```

Эта команда добавит соответствующие маркеры во все файлы где есть конфликты, аналогично конфликтам слияния или перебазирования. Для решения такой проблемы используется аналогичный подход — отредактируйте файлы исправив конфликты, добавьте их в индекс и выполните команду `git am --resolved` для перехода к следующему патчу:

```
$ (fix the file)
$ git add ticgit.gemspec
$ git am --resolved
Applying: See if this helps the gem
```

При желании, вы можете указать опцию `-3`, чтобы Git попробовал провести трёхстороннее слияние. Эта опция не включена по умолчанию, так как она не будет работать, если коммит, на который ссылается патч, отсутствует в вашем репозитории. Если у вас есть тот коммит, на который ссылается конфликтующий патч, то использование опции `-3` приводит к более умному применению конфликтующего патча:

```
$ git am -3 0001-see-if-this-helps-the-gem.patch
Applying: See if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
No changes -- Patch already applied.
```

В данном случае, без использования опции `-3` патч будет расценён как конфликтующий. При использовании опции `-3` патч будет применён без конфликтов.

Если вы применяете несколько патчей из файла mbox, то можно запустить `git am` в интерактивном режиме, в котором перед обработкой каждого патча будет задаваться вопрос о дальнейших действиях:

```
$ git am -3 -i mbox
Commit Body is:
-----
See if this helps the gem
-----
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

Это отличная возможность посмотреть содержимое патча перед его применением или пропустить его, если он уже был применён.

Когда все патчи применены и созданы коммиты в текущей ветке, вы уже можете решить стоит ли и как интегрировать их в более долгоживущую ветку.

Извлечение удалённых веток

Если участник проекта создал свой Git репозиторий, отправил в него свои изменения, а затем прислал вам ссылку и название ветки, куда были отправлены изменения, то вы можете добавить этот репозиторий как удалённый и провести слияние локально.

К примеру, Джессика отправила вам письмо, в котором сказано, у неё есть новый функционал в ветке `ruby-client` её репозитория. Добавив удалённый репозиторий и получив изменения из этой ветки, вы можете протестировать изменения извлекая их локально:

```
$ git remote add jessica git://github.com/jessica/myproject.git
$ git fetch jessica
$ git checkout -b rubyclient jessica/ruby-client
```

Если она снова пришлёт вам письмо с указанием на новый функционал уже в другой ветке, то для его получения достаточно `fetch` и `checkout`, так как удалённый репозиторий уже подключён.

Это очень полезно, когда вы постоянно работаете с этим человеком. Однако, от тех, кто редко отправляет небольшие патчи, будет проще принимать их по почте, чем требовать от всех поддержания собственных серверов с репозиториями, постоянно добавлять их как удалённые, а затем удалять и всё это ради нескольких патчей. Так же вы вряд ли захотите иметь сотни удалённых репозиторий, каждый из которых нужен только для одного или нескольких патчей. К счастью, скрипты и различные сервисы облегчают задачу, но во многом зависят от того как работаете вы и участники вашего проекта.

Отличительным преимуществом данного подхода является получение истории коммитов. Хотя возникновение конфликтов слияния и закономерно, но вы знаете с какого момента это произошло; корректное трёхстороннее слияние более предпочтительно, чем указать опцию `-3` и надеяться, что патч основан на коммите, к которому у вас есть доступ.

Если вы с кем-то не работаете постоянно, но всё равно хотите использовать удалённый репозиторий, то можно указать ссылку на него в команде `git pull`. Это приведёт к однократному выполнению, а ссылка на репозиторий сохранена не будет.

```
$ git pull https://github.com/onetimeguy/project
From https://github.com/onetimeguy/project
* branch      HEAD    -> FETCH_HEAD
Merge made by the 'recursive' strategy.
```

Определение применяемых изменений

На текущий момент у вас есть тематическая ветка, содержащая предоставленные изменения. Сейчас вы можете определиться что с ними

делать. В этом разделе рассматривается набор команд, которые помогут вам увидеть что именно будет интегрировано, если вы решите слить изменения в основную ветку.

Обычно, полезно просмотреть все коммиты текущей ветки, которые ещё не включены в основную. Вы можете исключить коммиты, которые уже есть в вашей основной ветке добавив опцию `--not` перед её названием. Это аналогично указанию использовавшегося ранее формата `master..contrib`. Например, если участник проекта отправил вам два патча, а вы создали ветку с названием `contrib` и применили их, то можно выполнить следующую команду:

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
Author: Scott Chacon <schacon@gmail.com>
Date: Fri Oct 24 09:53:59 2008 -0700

    See if this helps the gem

commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
Date: Mon Oct 22 19:38:36 2008 -0700

    Update gemspec to hopefully work better
```

Для просмотра изменений, представленных в каждом коммите, можно использовать опцию `-p` команды `git log`, которая выведет разницу по каждому коммиту.

Для просмотра полной разницы того, что произойдёт если вы сольёте изменения в другую ветку, вам понадобится использовать возможно странный способ для получения корректных результатов:

```
$ git diff master
```

Эта команда может вводить в заблуждение, но точно покажет разницу. Если ваша `master` ветка продвинулась вперед с тех пор как вы создали тематическую ветку, то вы получите на первый взгляд странные результаты. Это происходит потому, что Git непосредственно сравнивает снимки

последних коммитов текущей и `master` веток. Например, если вы добавили строку в файл в ветке `master`, то прямое сравнение снимков будет выглядеть как будто тематическая ветка собирается удалить эту строку.

Это не проблема, если ветка `master` является непосредственным родителем вашей тематической ветки, но если история обеих веток изменилась, то разница будет выглядеть как добавление всех изменений из тематической ветки и удаление всего нового из `master` ветки.

Что действительно нужно видеть, так это изменения тематической ветки, которые предстоит слить в `master` ветку. Это можно сделать, сказав Git сравнивать последний коммит тематической ветки с первым общим родителем для обеих веток.

Технически это делается за счёт явного указания общего коммита и применения разницы к нему:

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

или более кратко:

```
$ git diff $(git merge-base contrib master)
```

Однако это не удобно, поэтому Git предоставляет более короткий способ: синтаксис троеточия. При выполнении команды `diff`, следует поставить три точки после имени ветки для получения разницы между ней и текущей веткой, относительно общего родителя с другой веткой:

```
$ git diff master...contrib
```

Данная команда отобразит проделанную работу только из тематической ветки, относительно общего родителя с веткой `master`. Полезно запомнить указанный синтаксис.

Интеграция совместной работы

Когда все изменения в текущей тематической ветке готовы к интеграции с основной веткой, возникает вопрос как это сделать. Кроме этого, какой рабочий процесс вы хотите использовать при сопровождении вашего проекта? У вас несколько вариантов, давайте рассмотрим некоторые из них.

Схемы слияния

В простом рабочем процессе проделанная работа просто сливается в ветку `master`. При таком сценарии у вас есть ветка `master`, которая содержит стабильный код. Когда работа в тематической ветке завершена или вы проверили чью-то работу, вы сливаете её в ветку `master` и удаляете, затем процесс повторяется.

Если в репозитории присутствуют две ветки `ruby_client` и `php_client` с проделанной работой, как показано на рисунке [История с несколькими тематическими ветками](#), и вы сначала сливаете ветку `ruby_client`, а затем `php_client`, то состояние вашего репозитория будет выглядеть как показано на рисунке [Слияние тематической ветки](#).

 История с несколькими тематическими ветками

Рисунок 72. История с несколькими тематическими ветками

 Слияние тематической ветки

Рисунок 73. Слияние тематической ветки

Это, пожалуй, простейший рабочий процесс и его использование проблематично в больших или более стабильных проектах, где вы должны быть более осторожны с предоставленными изменениями.

Если у вас очень важный проект, то возможно вам стоит использовать двухступенчатый цикл слияния. При таком сценарии у вас имеются две долгоживущие ветки `master` и `develop`, где в `master` сливаются только очень стабильные изменения, а все новые доработки интегрируются в ветку `develop`. Обе ветки регулярно отправляются в публичный репозиторий.

Каждый раз, когда новая тематическая ветка готова к слиянию ([Перед слиянием тематической ветки](#)), вы сливаете её в `develop` ([После слияния](#)

[тематической ветки](#)); затем, когда вы выпускаете релиз, ветка `master` смещается на стабильное состояние ветки `develop` ([После релиза проекта](#)).

Перед слиянием тематической ветки

Рисунок 74. Перед слиянием тематической ветки

После слияния тематической ветки

Рисунок 75. После слияния тематической ветки

После релиза тематической ветки

Рисунок 76. После релиза проекта

Таким образом, люди могут клонировать репозиторий вашего проекта и использовать ветку `master` для сборки последнего стабильного состояния и получения актуальных изменений или использовать ветку `develop`, которая содержит самые последние изменения. Вы также можете продолжить эту концепцию, имея интеграционную ветку `integrate`, в которой объединяется вся работа. После того, как кодовая база указанной ветки стабильна и пройдены все тесты, она сливается в ветку `develop`, а после того, как стабильность слитых изменений доказана, вы перемещаете состояние ветки `master` на стабильное.

Схема с большим количеством слияний

В проекте Git присутствуют четыре долгоживущие ветки: `master`, `next`, `seen` (ранее `pu` — предложенные обновления) для новой работы и `maint` для поддержки обратной совместимости. Предложенные участниками проекта наработки накапливаются в тематических ветках основного репозитория по ранее описанному принципу (рис. [Управление сложным набором параллельно разрабатываемых тематических веток](#)). На этом этапе производится оценка содержимого тематических веток, чтобы определить, работают ли предложенные фрагменты так, как положено, или им требуется доработка. Если всё в порядке, тематические ветки сливаются в ветку `next`, которая отправляется на сервер, чтобы у каждого была возможность опробовать результат интеграции.

Управление сложным набором параллельно разрабатываемых тематических веток

Рисунок 77. Управление сложным набором параллельно разрабатываемых тематических веток

Если содержимое тематических веток требует доработки, оно сливается в ветку `seen`. Когда выясняется, что предложенный код полностью стабилен, он сливается в ветку `master`. Затем ветки `next` и `seen` перестраиваются на основании `master`. Это означает, что `master` практически всегда двигается только вперед, `next` время от времени перебазируется, а `seen` перебазируется ещё чаще:

Слияние тематических веток в долгоживущие ветки интеграции

Рисунок 78. Слияние тематических веток в долгоживущие ветки интеграции

После того, как тематическая ветка окончательно слита в `master`, она удаляется из репозитория. Репозиторий также содержит ветку `maint`, которая ответвляется от последнего релиза для предоставления патчей, если требуется поддержка обратной совместимости. Таким образом, после клонирования проекта у вас будет четыре ветки, дающие возможность перейти на разные стадии его разработки, в зависимости от того, на сколько передовым вы хотите быть или как вы собираетесь участвовать в проекте; вместе с этим, рабочий процесс структурирован, что помогает сопровождающему проекта проверять поступающий код. Рабочий процесс проекта Git специфицирован. Для полного понимания процесса обратитесь к [Git Maintainer's guide](#).

Схема с перебазированием и отбором

Некоторые сопровождающие предпочитают перебазировать или выборочно применять (`cherry-pick`) изменения относительно ветки `master` вместо слияния, что позволяет поддерживать историю проекта в линейном виде. Когда проделанная работа из тематической ветки готова к интеграции, вы переходите на эту ветку и перебазируете её относительно ветки `master` (или `develop` и т. д.). Если конфликты отсутствуют, то вы можете просто сдвинуть состояние ветки `master`, что обеспечивает линейность истории проекта.

Другим способом переместить предлагаемые изменения из одной ветки в другую является их отбор коммитов (`cherry-pick`). Отбор в Git похож на перебазирование для одного коммита. В таком случае формируется патч для

выбранного коммита и применяется к текущей ветке. Это полезно, когда в тематической ветке присутствует несколько коммитов, а вы хотите взять только один из них, или в тематической ветке только один коммит и вы предпочитаете использовать отбор вместо перебазирования. Предположим, ваш проект выглядит так:

Пример истории, из которой нужно отобрать отдельные коммиты

Рисунок 79. Пример истории, из которой нужно отобрать отдельные коммиты

Для применения коммита `e43a6` к ветке `master` выполните команду:

```
$ git cherry-pick e43a6
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
3 files changed, 17 insertions(+), 3 deletions(-)
```

Это действие применит изменения, содержащиеся в коммите `e43a6`, но будет сформирован новый коммит с другим значением SHA-1. После этого история будет выглядеть так:

История после отбора коммита из тематической ветки

Рисунок 80. История после отбора коммита из тематической ветки

Теперь тематическую ветку можно удалить, отбросив коммиты, которые вы не собираетесь включать в проект.

Возможность «Rerere»

Если вы часто производите перебазирование и слияние или поддерживаете долгоживущие тематические ветки, то в Git есть специальная возможность под названием «rerere», призванная вам помочь.

Rerere означает «reuse recorded resolution» (повторно использовать сохранённое решение) — это способ сокращения количества операций ручного разрешения конфликтов. Когда эта опция включена, Git будет сохранять набор образов до и после успешного слияния, а также разрешать конфликты самостоятельно, если аналогичные конфликты уже были разрешены ранее.

Эта возможность реализована как команда и как параметр конфигурации. Параметр конфигурации называется `rerere.enabled`, который можно включить глобально следующим образом:

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

После этого любое разрешение конфликта слияния будет записано на случай повторного использования.

Если нужно, вы можете обращаться к кэшу «rerere» напрямую, используя команду `git rerere`. Когда команда вызвана без параметров, Git проверяет базу данных и пытается найти решение для разрешения текущего конфликта слияния (точно так же как и при установленной настройке `rerere.enabled` в значение `true`). Существует множество дополнительных команд для просмотра, что именно будет записано, удаления отдельных записей из кэша, а так же его полной очистки. Более детально «rerere» будет рассмотрено в разделе [Rerere](#) главы 7.

Помечайте свои релизы

После выпуска релиза, возможно, вы захотите пометить текущее состояние так, чтобы можно было вернуться к нему в любой момент. Для этого можно добавить тег, как было описано в главе [Основы Git](#). Кроме этого, вы можете добавить цифровую подпись для тега, выглядеть это будет вот так:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
You need a passphrase to unlock the secret key for
user: "Scott Chacon <schacon@gmail.com>"
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

Если вы используете цифровую подпись при расстановке тегов, то возникает проблема распространения публичной части PGP ключа, использованного при создании подписи. Сопровождающий Git проекта может решить эту проблему добавив в репозиторий свой публичный ключ как бинарный объект и установив ссылающийся на него тег. Чтобы это сделать, выберите нужный

ключ из списка доступных, который можно получить с помощью команды `gpg --list-keys`:

```
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg
-----
pub 1024D/F721C45A 2009-02-09 [expires: 2010-02-09]
uid          Scott Chacon <schacon@gmail.com>
sub 2048g/45D02282 2009-02-09 [expires: 2010-02-09]
```

Затем экспортируйте выбранный ключ и поместите его непосредственно в базу данных Git при помощи команды `git hash-object`, которая создаст новый объект с содержимым ключа и вернёт SHA-1 этого объекта:

```
$ gpg -a --export F721C45A | git hash-object -w --stdin
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Теперь, когда ваш публичный ключ находится в репозитории, можно поставить указывающий на него тег, используя полученное ранее значение SHA-1:

```
$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Выполнив команду `git push --tags`, `maintainer-pgp-pub` тег станет общедоступным. Теперь все, кто захочет проверить вашу подпись, могут импортировать ваш публичный ключ, предварительно получив его из репозитория:

```
$ git show maintainer-pgp-pub | gpg --import
```

После этого можно проверять цифровую подпись ваших тегов. Кроме этого, вы можете включить дополнительные инструкции по проверке вашей подписи в сообщение тега, которое будет отображаться каждый раз при вызове команды `git show <tag>`.

Генерация номера сборки

Git не использует монотонно возрастающие идентификаторы для коммитов, поэтому если вы хотите получить читаемые имена коммитов, то воспользуйтесь командой `git describe` для нужного коммита. Git вернёт имя ближайшего тега, количество коммитов после него и частичное значение SHA-1 для указанного коммита (с префиксом в виде буквы «g» — означает Git):

```
$ git describe master  
v1.6.2-rc1-20-g8c5b85c
```

Таким образом, вы можете сделать снимок или собрать сборку и дать ей понятное для человека название. К слову, если вы клонируете репозиторий Git и соберете его из исходного кода, то вывод команды `git --version` будет примерно таким же. Если попытаться получить имя коммита, которому назначен тег, то результатом будет название самого тега.

По умолчанию, команда `git describe` поддерживает только аннотированные теги (созданные с использованием опций `-a` или `-s`); если вы хотите использовать легковесные (не аннотированные) теги, то укажите команде параметр `--tags`. Также это название можно использовать при выполнении команд `git checkout` и `git show`, но в будущем они могут перестать работать из-за сокращённого значения SHA-1. К примеру, ядро Linux недавно перешло к использованию 10 символов в SHA-1 вместо 8 чтобы обеспечить уникальность каждого объекта, таким образом предыдущие результаты `git describe` стали недействительными.

Подготовка релиза

Время делать релиз сборки. Возможно, вы захотите сделать архив последнего состояния вашего кода для тех, кто не использует Git. Для создания архива выполните команду `git archive`:

```
$ git archive master --prefix='project/' | gzip > `git describe master`.tar.gz  
$ ls *.tar.gz  
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

Открывший этот tarball-архив пользователь получит последнее состояние кода проекта в каталоге `project`. Точно таким же способом можно создать zip-архив, просто добавив опцию `--format=zip` для команды `git archive`:

```
$ git archive master --prefix='project/' --format=zip > `git describe master`.zip
```

В итоге получим tarball- и zip-архивы с релизом проекта, которые можно загрузить на сайт или отправить по почте.

Краткая история (Shortlog)

Сейчас самое время оповестить людей из списка рассылки, которые хотят знать что происходит с вашим проектом. С помощью команды `git shortlog` можно быстро получить список изменений, внесённых в проект с момента последнего релиза или предыдущей рассылки. Она собирает все коммиты в заданном интервале; например, следующая команда выведет список коммитов с момента последнего релиза с названием `v1.0.1`:

```
$ git shortlog --no-merges master --not v1.0.1
```

Chris Wanstrath (6):

- Add support for annotated tags to Grit::Tag
- Add packed-refs annotated tag support.
- Add Grit::Commit#to_patch
- Update version and History.txt
- Remove stray `puts`
- Make ls_tree ignore nils

Tom Preston-Werner (4):

- fix dates in history
- dynamic version method
- Version bump to 1.0.2
- Regenerated gems spec for version 1.0.2

И так, у вас есть готовая к отправке сводка коммитов начиная с версии `v1.0.1`, сгруппированных по авторам.

Ветвление в Git

Почти каждая система контроля версий в той или иной форме поддерживает ветвление. Используя ветвление, Вы отклоняетесь от основной линии разработки и продолжаете работу независимо от неё, не вмешиваясь в основную линию. Во многих системах контроля версий создание веток — это очень затратный процесс, часто требующий создания новой копии каталога с исходным кодом, что может занять много времени для большого проекта. Некоторые люди, говоря о модели ветвления Git, называют её «киллер-фича», что выгодно выделяет Git на фоне остальных систем контроля версий. Что в ней такого особенного? Ветвление Git очень легковесно: операция создания ветки выполняется почти мгновенно, переключение между ветками туда-сюда, обычно, также быстро. В отличие от многих других систем контроля версий, Git поощряет процесс работы, при котором ветвление и слияние выполняется часто, даже по несколько раз в день. Понимание и владение этой функциональностью дает вам уникальный и мощный инструмент, который может полностью изменить привычный процесс разработки.

Работа с ветками

Выведение списка веток из удаленного репозитория

```
$ git ls-remote origin
```

Добавление веток удаленного репозитория origin

```
$ git fetch origin
```

Переключение на ветку

```
$ git checkout branch_name
```

Удаление ветки

Локальное удаление ветки

```
$ git branch -d <branchname>
```

Удаление ветки в удаленном репозитории

```
$ git push -d <remote_name> <branchname>
```


О ветвлении в двух словах

Почти каждая система контроля версий в той или иной форме поддерживает ветвление. Используя ветвление, Вы отклоняетесь от основной линии разработки и продолжаете работу независимо от неё, не вмешиваясь в основную линию. Во многих системах контроля версий создание веток — это очень затратный процесс, часто требующий создания новой копии каталога с исходным кодом, что может занять много времени для большого проекта.

Некоторые люди, говоря о модели ветвления Git, называют её «киллер-фича», что выгодно выделяет Git на фоне остальных систем контроля версий. Что в ней такого особенного? Ветвление Git очень легковесно: операция создания ветки выполняется почти мгновенно, переключение между ветками туда-сюда, обычно, также быстро. В отличие от многих других систем контроля версий, Git поощряет процесс работы, при котором ветвление и слияние выполняется часто, даже по несколько раз в день. Понимание и владение этой функциональностью дает вам уникальный и мощный инструмент, который может полностью изменить привычный процесс разработки.

О ветвлении в двух словах

Для точного понимания механизма ветвлений, необходимо вернуться назад и изучить то, как Git хранит данные.

Как вы можете помнить из [Что такое Git?](#), Git не хранит данные в виде последовательности изменений, он использует набор снимков (snapshot).

Когда вы делаете коммит, Git сохраняет его в виде объекта, который содержит указатель на снимок (snapshot) подготовленных данных. Этот объект так же содержит имя автора и email, сообщение и указатель на коммит или коммиты непосредственно предшествующие данному (его родителей): отсутствие родителя для первоначального коммита, один родитель для обычного коммита, и несколько родителей для результатов слияния двух и более веток.

Предположим, у вас есть каталог с тремя файлами и вы добавляете их все в индекс и создаёте коммит. Во время индексации вычисляется контрольная сумма каждого файла (SHA-1 как мы узнали из [Что такое Git?](#)), затем каждый файл сохраняется в репозиторий (Git называет такой файл *блоб* — большой бинарный объект), а контрольная сумма попадёт в индекс:

```
$ git add README test.rb LICENSE
$ git commit -m 'Initial commit'
```

Когда вы создаёте коммит командой `git commit`, Git вычисляет контрольные суммы каждого подкаталога (в нашем случае, только основной каталог проекта) и сохраняет его в репозитории как объект дерева каталогов. Затем Git создаёт объект коммита с метаданными и указателем на основное дерево проекта для возможности воссоздать этот снимок в случае необходимости.

Ваш репозиторий Git теперь хранит пять объектов: три *блоба* объекта (по одному на каждый файл), объект *дерева* каталогов, содержащий список файлов и соответствующих им *блобов*, а так же объект *коммита*, содержащий метаданные и указатель на объект дерева каталогов.

Коммит и его дерево

Рисунок 9. Коммит и его дерево

Если вы сделаете изменения и создадите ещё один коммит, то он будет содержать указатель на предыдущий коммит.

Коммит и его родители

Рисунок 10. Коммит и его родители

Ветка в Git — это простой перемещаемый указатель на один из таких коммитов. По умолчанию, имя основной ветки в Git — `master`. Как только вы начнёте создавать коммиты, ветка `master` будет всегда указывать на

последний коммит. Каждый раз при создании коммита указатель ветки `master` будет передвигаться на следующий коммит автоматически.

Примечание	Ветка «master» в Git — это не какая-то особенная ветка. Она точно такая же, как и все остальные ветки. Она существует почти во всех репозиториях только лишь потому, что её создаёт команда <code>git init</code> , а большинство людей не меняют её название.
------------	--

Ветка и история коммитов
Рисунок 11. Ветка и история коммитов

Создание новой ветки

Что же на самом деле происходит при создании ветки? Всего лишь создаётся новый указатель для дальнейшего перемещения. Допустим вы хотите создать новую ветку с именем `testing`. Вы можете это сделать командой `git branch` :

```
$ git branch testing
```

В результате создаётся новый указатель на текущий коммит.

Две ветки указывают на одну и ту же последовательность коммитов
Рисунок 12. Две ветки указывают на одну и ту же последовательность коммитов

Как Git определяет, в какой ветке вы находитесь? Он хранит специальный указатель `HEAD`. Имейте ввиду, что в Git концепция `HEAD` значительно отличается от других систем контроля версий, которые вы могли использовать раньше (Subversion или CVS). В Git — это указатель на текущую локальную ветку. В нашем случае мы всё ещё находимся в ветке `master`. Команда `git branch` только *создаёт* новую ветку, но не переключает на неё.

HEAD указывает на ветку

Рисунок 13. HEAD указывает на ветку

Вы можете легко это увидеть при помощи простой команды `git log`, которая покажет вам куда указывают указатели веток. Эта опция называется `--decorate`.

```
$ git log --oneline --decorate
f30ab (HEAD -> master, testing) Add feature #32 - ability to add new formats to the central interface
34ac2 Fix bug #1328 - stack overflow under certain conditions
98ca9 Initial commit
```

Здесь можно увидеть указывающие на коммит `f30ab` ветки: `master` и `testing`.

Переключение веток

Для переключения на существующую ветку выполните команду `git checkout`. Давайте переключимся на ветку `testing`:

```
$ git checkout testing
```

В результате указатель `HEAD` переместится на ветку `testing`.

HEAD указывает на текущую ветку

Рисунок 14. HEAD указывает на текущую ветку

Какой в этом смысл? Давайте сделаем ещё один коммит:

```
$ vim test.rb  
$ git commit -a -m 'made a change'
```

Указатель на ветку HEAD переместился вперёд после коммита

Рисунок 15. Указатель на ветку HEAD переместился вперёд после коммита

Интересная ситуация: указатель на ветку `testing` переместился вперёд, а `master` указывает на тот же коммит, где вы были до переключения веток командой `git checkout`. Давайте переключимся назад на ветку `master`:

```
$ git checkout master
```

Примечание

`git log` не показывает все ветки по умолчанию. Если выполнить команду `git log` прямо сейчас, то в её выводе только что созданная ветка «testing» фигурировать не будет. Ветка никуда не исчезла; просто Git не знает, что именно она вас интересует, и выводит наиболее полезную по его мнению информацию. Другими словами, по умолчанию `git log` отобразит историю коммитов только для текущей ветки. Для просмотра истории коммитов другой ветки необходимо явно указать её имя: `git log testing`. Чтобы посмотреть историю по всем веткам — выполните команду с дополнительным флагом: `git log --all`.

HEAD перемещается когда вы делаете checkout

Рисунок 16. HEAD перемещается когда вы делаете checkout

Эта команда сделала две вещи: переместила указатель `HEAD` назад на ветку `master` и вернула файлы в рабочем каталоге в то состояние, на снимок которого указывает `master`. Это также означает, что все вносимые с этого момента изменения будут относиться к старой версии проекта. Другими словами, вы откатали все изменения ветки `testing` и можете продолжать в другом направлении.

Примечание	Переключение веток меняет файлы в рабочем каталоге Важно запомнить, что при переключении веток в Git происходит изменение файлов в рабочем каталоге. Если вы переключаетесь на старую ветку, то рабочий каталог будет выглядеть так же, как выглядел на момент последнего коммита в ту ветку. Если Git по каким-то причинам не может этого сделать — он не позволит вам переключиться вообще.
------------	--

Давайте сделаем ещё несколько изменений и создадим очередной коммит:

```
$ vim test.rb  
$ git commit -a -m 'made other changes'
```

Теперь история вашего проекта разошлась (см [Разветвлённая история](#)). Вы создали ветку и переключились на неё, поработали, а затем вернулись в основную ветку и поработали в ней. Эти изменения изолированы друг от друга: вы можете свободно переключаться туда и обратно, а когда понадобится — объединить их. И всё это делается простыми командами: `branch`, `checkout` и `commit`.

Разветвлённая история

Рисунок 17. Разветвлённая история

Все описанные действия можно визуализировать с помощью команды `git log`. Для отображения истории коммитов, текущего положения указателей веток и истории ветвления выполните команду `git log --oneline --decorate --graph --all`.

```
$ git log --oneline --decorate --graph --all  
* c2b9e (HEAD, master) Made other changes  
| * 87ab2 (testing) Made a change  
|/
```

- * f30ab Add feature #32 - ability to add new formats to the central interface
- * 34ac2 Fix bug #1328 - stack overflow under certain conditions
- * 98ca9 initial commit of my project

Ветка в Git — это простой файл, содержащий 40 символов контрольной суммы SHA-1 коммита, на который она указывает; поэтому операции с ветками являются дешёвыми с точки зрения потребления ресурсов или времени. Создание новой ветки в Git происходит так же быстро и просто как запись 41 байта в файл (40 знаков и перевод строки).

Это принципиально отличает процесс ветвления в Git от более старых систем контроля версий, где все файлы проекта копируются в другой подкаталог. В зависимости от размера проекта, операции ветвления в таких системах могут занимать секунды или даже минуты, когда в Git эти операции мгновенны. Поскольку при коммите мы сохраняем указатель на родительский коммит, то поиск подходящей базы для слияния веток делается автоматически и, в большинстве случаев, очень прост. Эти возможности побуждают разработчиков чаще создавать и использовать ветки.

Давайте посмотрим, почему и вам имеет смысл делать так же.

Примечание	Одновременное создание новой ветки и переключение на неё Как правило, при создании новой ветки вы хотите сразу на неё переключиться — это можно сделать используя команду <code>git checkout -b <newbranchname></code> .
------------	---

Примечание

Начиная с Git версии 2.23, вы можете использовать `git switch` вместо `git checkout`, чтобы:

- Переключиться на существующую ветку: `git switch testing-branch`.
- Создать новую ветку и переключиться на неё: `git switch -c new-branch`. Флаг `-c` означает создание, но также можно использовать полный формат: `--create`.
- Вернуться к предыдущей извлечённой ветке: `git switch -`.

Список используемых ИСТОЧНИКОВ

- [ProGit](#)

Инструменты работы с git

- [lazygit](#)