**Репозито́рий (от англ. repository — хранилище)** — место, где хранятся и поддерживаются какие-либо данные. Чаще всего данные в *репозитории* хранятся в виде файлов, доступных для дальнейшего распространения по сети.

Существуют *репозитории* для хранения программ, написанных на одном языке (например, CPAN для Perl) или предназначенных для одной платформы. Многие современные операционные системы, такие как OpenSolaris, FreeBSD и большинство дистрибутивов Linux, имеют официальные *репозитории*, но также позволяют устанавливать пакеты из других мест. Большинство *репозиториев* бесплатны, однако некоторые компании предоставляют доступ к собственным *репозиториям* за платную подписку.

*Репозитории* используются в системах управления версиями, в них хранятся все документы вместе с историей их изменения и другой служебной информацией.

Существуют различные автоматизированные системы создания *репозиториев*. Один из типов *репозиториев*: хранилища на CD/DVD — установочные диски для пакетов того или иного ПО.

Среди дистрибутивов Linux популярны *репозитории* с форматом метаданных YUM для дистрибутивов на базе RPM-пакетов, и *репозитории* с метаданными APT для дистрибутивов на основе DEB-пакетов.

**Создание Git-репозитория**

Обычно вы получаете репозиторий Git одним из двух способов:

1. Вы можете взять локальный каталог, который в настоящее время не находится под версионным контролем, и превратить его в репозиторий Git, либо
2. Вы можете **клонировать** существующий репозиторий Git из любого места.

В обоих случаях вы получите готовый к работе Git репозиторий на вашем компьютере.

**Создание репозитория в существующем каталоге**

Если у вас уже есть проект в каталоге, который не находится под версионным контролем Git, то для начала нужно перейти в него. Если вы не делали этого раньше, то для разных операционных систем это выглядит по-разному:

для Linux:

$ cd /home/user/my\_project

для macOS:

$ cd /Users/user/my\_project

для Windows:

$ cd C:/Users/user/my\_project

а затем выполните команду:

$ git init

Эта команда создаёт в текущем каталоге новый подкаталог с именем .git, содержащий все необходимые файлы репозитория — структуру Git репозитория. На этом этапе ваш проект ещё не находится под версионным контролем. Подробное описание файлов, содержащихся в только что созданном вами каталоге .git, приведено в главе Git изнутри

Если вы хотите добавить под версионный контроль существующие файлы (в отличие от пустого каталога), вам стоит добавить их в индекс и осуществить первый коммит изменений. Добиться этого вы сможете запустив команду git add несколько раз, указав индексируемые файлы, а затем выполнив git commit:

$ git add \*.c

$ git add LICENSE

$ git commit -m 'Initial project version'

Мы разберем, что делают эти команды чуть позже. Теперь у вас есть Git-репозиторий с отслеживаемыми файлами и начальным коммитом.

**Клонирование существующего репозитория**

Для получения копии существующего Git-репозитория, например, проекта, в который вы хотите внести свой вклад, необходимо использовать команду git clone. Если вы знакомы с другими системами контроля версий, такими как Subversion, то заметите, что команда называется «clone», а не «checkout». Это важное различие — вместо того, чтобы просто получить рабочую копию, Git получает копию практически всех данных, которые есть на сервере. При выполнении git clone с сервера забирается (pulled) каждая версия каждого файла из истории проекта. Фактически, если серверный диск выйдет из строя, вы можете использовать любой из клонов на любом из клиентов, для того, чтобы вернуть сервер в то состояние, в котором он находился в момент клонирования (вы можете потерять часть серверных хуков (server-side hooks) и т. п., но все данные, помещённые под версионный контроль, будут сохранены, подробнее об этом смотрите в разделе Установка Git на сервер главы 4).

Клонирование репозитория осуществляется командой git clone <url>. Например, если вы хотите клонировать библиотеку libgit2, вы можете сделать это следующим образом:

$ git clone https://github.com/libgit2/libgit2

Эта команда создаёт каталог libgit2, инициализирует в нём подкаталог .git, скачивает все данные для этого репозитория и извлекает рабочую копию последней версии. Если вы перейдёте в только что созданный каталог libgit2, то увидите в нём файлы проекта, готовые для работы или использования. Для того, чтобы клонировать репозиторий в каталог с именем, отличающимся от libgit2, необходимо указать желаемое имя, как параметр командной строки:

$ git clone https://github.com/libgit2/libgit2 mylibgit

Эта команда делает всё то же самое, что и предыдущая, только результирующий каталог будет назван mylibgit.

В Git реализовано несколько транспортных протоколов, которые вы можете использовать. В предыдущем примере использовался протокол https://, вы также можете встретить git:// или user@server:path/to/repo.git, использующий протокол передачи SSH. В разделе [Установка Git на сервер](https://git-scm.com/book/ru/v2/ch00/r_git_on_the_server) главы 4 мы познакомимся со всеми доступными вариантами конфигурации сервера для обеспечения доступа к вашему Git репозиторию, а также рассмотрим их достоинства и недостатки.

## Запись изменений в репозиторий

Итак, у вас имеется настоящий Git-репозиторий и рабочая копия файлов для некоторого проекта. Вам нужно делать некоторые изменения и фиксировать «снимки» состояния (snapshots) этих изменений в вашем репозитории каждый раз, когда проект достигает состояния, которое вам хотелось бы сохранить.

Запомните, каждый файл в вашем рабочем каталоге может находиться в одном из двух состояний: под версионным контролем (отслеживаемые) и нет (неотслеживаемые). Отслеживаемые файлы — это те файлы, которые были в последнем снимке состояния проекта; они могут быть неизменёнными, изменёнными или подготовленными к коммиту. Если кратко, то отслеживаемые файлы — это те файлы, о которых знает Git.

Неотслеживаемые файлы — это всё остальное, любые файлы в вашем рабочем каталоге, которые не входили в ваш последний снимок состояния и не подготовлены к коммиту. Когда вы впервые клонируете репозиторий, все файлы будут отслеживаемыми и неизменёнными, потому что Git только что их извлек и вы ничего пока не редактировали.

Как только вы отредактируете файлы, Git будет рассматривать их как изменённые, так как вы изменили их с момента последнего коммита. Вы индексируете эти изменения, затем фиксируете все проиндексированные изменения, а затем цикл повторяется.

### Определение состояния файлов

Основной инструмент, используемый для определения, какие файлы в каком состоянии находятся — это команда git status. Если вы выполните эту команду сразу после клонирования, вы увидите что-то вроде этого:

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

nothing to commit, working tree clean

Это означает, что у вас чистый рабочий каталог, другими словами — в нем нет отслеживаемых измененных файлов. Git также не обнаружил неотслеживаемых файлов, в противном случае они бы были перечислены здесь. Наконец, команда сообщает вам на какой ветке вы находитесь и сообщает вам, что она не расходится с веткой на сервере. Пока что это всегда ветка master, ветка по умолчанию; в этой главе это не важно

### Отслеживание новых файлов

Для того чтобы начать отслеживать (добавить под версионный контроль) новый файл, используется команда git add. Чтобы начать отслеживание файла README, вы можете выполнить следующее:

$ git add README

Если вы снова выполните команду status, то увидите, что файл README теперь отслеживаемый и добавлен в индекс:

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

new file: README

Вы можете видеть, что файл проиндексирован, так как он находится в секции «Changes to be committed». Если вы выполните коммит в этот момент, то версия файла, существовавшая на момент выполнения вами команды git add, будет добавлена в историю снимков состояния. Как вы помните, когда вы ранее выполнили git init, затем вы выполнили git add (файлы) — это было сделано для того, чтобы добавить файлы в вашем каталоге под версионный контроль. Команда git add принимает параметром путь к файлу или каталогу, если это каталог, команда рекурсивно добавляет все файлы из указанного каталога в индекс.

### Индексация изменённых файлов

Давайте модифицируем файл, уже находящийся под версионным контролем. Если вы измените отслеживаемый файл CONTRIBUTING.md и после этого снова выполните команду git status, то результат будет примерно следующим:

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

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

new file: 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

### Сокращенный вывод статуса

Вывод команды git status довольно всеобъемлющий и многословный. Git также имеет флаг вывода сокращенного статуса, так что вы можете увидеть изменения в более компактном виде. Если вы выполните git status -s или git status --short вы получите гораздо более упрощенный вывод:

$ git status -s

M README

MM Rakefile

A lib/git.rb

M lib/simplegit.rb

?? LICENSE.txt

### Сокращенный вывод статуса

Вывод команды git status довольно всеобъемлющий и многословный. Git также имеет флаг вывода сокращенного статуса, так что вы можете увидеть изменения в более компактном виде. Если вы выполните git status -s или git status --short вы получите гораздо более упрощенный вывод:

$ git status -s

M README

MM Rakefile

A lib/git.rb

M lib/simplegit.rb

?? LICENSE.txt

### Просмотр индексированных и неиндексированных изменений

Если результат работы команды git status недостаточно информативен для вас — вам хочется знать, что конкретно поменялось, а не только какие файлы были изменены — вы можете использовать команду git diff. Позже мы рассмотрим команду git diff подробнее; вы, скорее всего, будете использовать эту команду для получения ответов на два вопроса: что вы изменили, но ещё не проиндексировали, и что вы проиндексировали и собираетесь включить в коммит. Если git status отвечает на эти вопросы в самом общем виде, перечисляя имена файлов, git diff показывает вам непосредственно добавленные и удалённые строки — патч как он есть.

## Просмотр истории коммитов

После того, как вы создали несколько коммитов или же клонировали репозиторий с уже существующей историей коммитов, вероятно вам понадобится возможность посмотреть что было сделано — историю коммитов. Одним из основных и наиболее мощных инструментов для этого является команда git log.

Следующие несколько примеров используют очень простой проект «simplegit». Чтобы клонировать проект, используйте команду:

$ git clone https://github.com/schacon/simplegit-progit

Если вы запустите команду git log в каталоге клонированного проекта, вы увидите следующий вывод:

$ git log

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

Change version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 16:40:33 2008 -0700

Remove unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 10:31:28 2008 -0700

Initial commit

По умолчанию (без аргументов) git log перечисляет коммиты, сделанные в репозитории в обратном к хронологическому порядке — последние коммиты находятся вверху. Из примера можно увидеть, что данная команда перечисляет коммиты с их SHA-1 контрольными суммами, именем и электронной почтой автора, датой создания и сообщением коммита.

Команда git log имеет очень большое количество опций для поиска коммитов по разным критериям. Рассмотрим наиболее популярные из них.

Одним из самых полезных аргументов является -p или --patch, который показывает разницу (выводит **патч**), внесенную в каждый коммит. Так же вы можете ограничить количество записей в выводе команды; используйте параметр -2 для вывода только двух записей:

$ git log -p -2

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

Change version number

diff --git a/Rakefile b/Rakefile

index a874b73..8f94139 100644

--- a/Rakefile

+++ b/Rakefile

@@ -5,7 +5,7 @@ require 'rake/gempackagetask'

spec = Gem::Specification.new do |s|

s.platform = Gem::Platform::RUBY

s.name = "simplegit"

- s.version = "0.1.0"

+ s.version = "0.1.1"

s.author = "Scott Chacon"

s.email = "schacon@gee-mail.com"

s.summary = "A simple gem for using Git in Ruby code."

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 16:40:33 2008 -0700

Remove unnecessary test

diff --git a/lib/simplegit.rb b/lib/simplegit.rb

index a0a60ae..47c6340 100644

--- a/lib/simplegit.rb

+++ b/lib/simplegit.rb

@@ -18,8 +18,3 @@ class SimpleGit

end

end

-

-if $0 == \_\_FILE\_\_

- git = SimpleGit.new

- puts git.show

-end

Эта опция отображает аналогичную информацию но содержит разницу для каждой записи. Очень удобно использовать данную опцию для код ревью или для быстрого просмотра серии внесенных изменений. Так же есть возможность использовать серию опций для обобщения. Например, если вы хотите увидеть сокращенную статистику для каждого коммита, вы можете использовать опцию --stat:

$ git log --stat

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

Change version number

Rakefile | 2 +-

1 file changed, 1 insertion(+), 1 deletion(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 16:40:33 2008 -0700

Remove unnecessary test

lib/simplegit.rb | 5 -----

1 file changed, 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 10:31:28 2008 -0700

Initial commit

README | 6 ++++++

Rakefile | 23 +++++++++++++++++++++++

lib/simplegit.rb | 25 +++++++++++++++++++++++++

3 files changed, 54 insertions(+)

Как вы видите, опция --stat печатает под каждым из коммитов список и количество измененных файлов, а также сколько строк в каждом из файлов было добавлено и удалено. В конце можно увидеть суммарную таблицу изменений.

Следующей действительно полезной опцией является --pretty. Эта опция меняет формат вывода. Существует несколько встроенных вариантов отображения. Опция oneline выводит каждый коммит в одну строку, что может быть очень удобным если вы просматриваете большое количество коммитов. К тому же, опции short, full и fuller делают вывод приблизительно в том же формате, но с меньшим или большим количеством информации соответственно:

$ git log --pretty=oneline

ca82a6dff817ec66f44342007202690a93763949 Change version number

085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 Remove unnecessary test

a11bef06a3f659402fe7563abf99ad00de2209e6 Initial commit

Наиболее интересной опцией является format, которая позволяет указать формат для вывода информации. Особенно это может быть полезным когда вы хотите сгенерировать вывод для автоматического анализа — так как вы указываете формат явно, он не будет изменен даже после обновления Git:

$ git log --pretty=format:"%h - %an, %ar : %s"

ca82a6d - Scott Chacon, 6 years ago : Change version number

085bb3b - Scott Chacon, 6 years ago : Remove unnecessary test

a11bef0 - Scott Chacon, 6 years ago : Initial commit

[Полезные опции для git log --pretty=format](https://git-scm.com/book/ru/v2/ch00/rpretty_format) отображает наиболее полезные опции для изменения формата.

### Ограничение вывода

В дополнение к опциям форматирования вывода, команда git log принимает несколько опций для ограничения вывода — опций, с помощью которых можно увидеть определенное подмножество коммитов. Вы уже видели одну из таких опций — это опция -2, которая показывает только последние два коммита. В действительности вы можете использовать -<n>, где n — это любое натуральное число и представляет собой n последних коммитов. На практике вы не будете часто использовать эту опцию, потому что Git по умолчанию использует постраничный вывод и вы будете видеть только одну страницу за раз.

Однако, опции для ограничения вывода по времени, такие как --since и --until, являются очень удобными. Например, следующая команда покажет список коммитов, сделанных за последние две недели:

$ git log --since=2.weeks

Это команда работает с большим количеством форматов — вы можете указать определенную дату вида 2008-01-15 или же относительную дату, например 2 years 1 day 3 minutes ago.

Также вы можете фильтровать список коммитов по заданным параметрам. Опция --author дает возможность фильтровать по автору коммита, а опция --grep искать по ключевым словам в сообщении коммита.

## 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.

|  |  |
| --- | --- |
| **Примечание** | 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 напомнит 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.

|  |  |
| --- | --- |
| **Примечание** | Команда git reset **может** быть опасной если вызвать её с параметром --hard. В приведенном 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](https://git-scm.com/book/ru/v2/ch00/r_git_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 еще 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** | 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](https://git-scm.com/book/ru/v2/ch00/ch03-git-branching).

Помните, все 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](https://git-scm.com/book/ru/v2/ch00/r_data_recovery)). 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 измененного 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

**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.

|  |  |
| --- | --- |
| **Примечание** | **Удаленный 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 репозиториев 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](https://git-scm.com/book/ru/v2/ch00/r_git_on_the_server) главы 4.

**Добавление удалённых репозиториев**

В предыдущих разделах мы уже упоминали и приводили примеры добавления удалённых репозиториев, 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](https://git-scm.com/book/ru/v2/ch00/ch03-git-branching).

**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](https://git-scm.com/book/ru/v2/ch00/ch03-git-branching) 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.

|  |  |
| --- | --- |
| **Примечание** | 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 удаленный 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](https://git-scm.com/book/ru/v2/ch00/ch03-git-branching) для более подробного описания, как отправлять изменения на удалённый сервер.

**Просмотр удаленного репозитория**

Если хотите получить побольше информации об одном из удалённых репозиториев, 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.

**Работа с тегами**

Как и большинство СКВ, 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

|  |  |
| --- | --- |
| **Примечание** | **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

v1.5

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 помечать 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.

|  |  |
| --- | --- |
| **Примечание** | **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 лёгковесные 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>

## Псевдонимы в Git

Прежде, чем закончить эту главу по основам Git, рассмотрим ещё одну маленькую хитрость, которая поможет сделать использование Git проще, легче, и более привычным: псевдонимы (aliases). Мы не будем ссылаться на них дальше или предполагать, что вы будете пользоваться ими по ходу чтения книги, но вам лучше было бы знать, как их использовать.

Git не будет пытаться сделать вывод о том, какую команду вы хотели ввести, если вы ввели её не полностью. Если вы не хотите печатать каждую команду для Git целиком, вы легко можете настроить псевдонимы (alias) для любой команды с помощью git config. Вот несколько примеров псевдонимов, которые вы, возможно, захотите задать:

$ git config --global alias.co checkout

$ git config --global alias.br branch

$ git config --global alias.ci commit

$ git config --global alias.st status

Это означает, что, например, вместо ввода git commit, вам достаточно набрать только git ci. По мере освоения Git вам, вероятно, придётся часто пользоваться и другими командами. В этом случае без колебаний создавайте новые псевдонимы.

Такой способ может также быть полезен для создания команд, которые, как вы думаете, должны существовать. Например, чтобы исправить неудобство, с которым мы столкнулись при исключении файла из индекса, можно добавить в Git свой собственный псевдоним unstage:

$ git config --global alias.unstage 'reset HEAD --'

Это делает эквивалентными следующие две команды:

$ git unstage fileA

$ git reset HEAD -- fileA

Такой вариант кажется немного более понятным. Также, обычно, добавляют команду last следующим образом:

$ git config --global alias.last 'log -1 HEAD'

Таким образом, можно легко просмотреть последний коммит:

$ git last

commit 66938dae3329c7aebe598c2246a8e6af90d04646

Author: Josh Goebel <dreamer3@example.com>

Date: Tue Aug 26 19:48:51 2008 +0800

Test for current head

Signed-off-by: Scott Chacon <schacon@example.com>

Можно сказать, что Git просто заменяет эти команды на созданные вами псевдонимы (alias). Однако, возможно, вы захотите выполнить внешнюю команду, а не подкоманду Git. В этом случае, следует начать команду с символа !. Это полезно, если вы пишете свои утилиты для работы с Git-репозиторием. Продемонстрируем этот случай на примере создания псевдонима git visual для запуска gitk:

$ git config --global alias.visual '!gitk'