## Установка

### Установка в Linux

Linux — необходимо открыть терминал и установить приложение при помощи пакетного менеджера вашего дистрибутива. Для Ubuntu команда будет выглядеть следующим образом:

sudo apt-get install git

### Установка в Windows

Для установки Git в Windows также имеется несколько способов. Официальная сборка доступна для скачивания на официальном сайте Git. Просто перейдите на страницу <https://git-scm.com/download/win>, и загрузка запустится автоматически.

### Установка на Mac

Существует несколько способов установки Git на Mac. Самый простой — установить Xcode Command Line Tools. В версии Mavericks (10.9) и выше вы можете добиться этого просто первый раз выполнив 'git' в терминале. Установщик Git для OS X доступен для скачивания с сайта Git <https://git-scm.com/download/mac>.

## Настройка репозитория

### Установка имени и электронной почты (установки окончаний строк)

git config --global user.name "Your Name"

git config --global user.email "your\_email@whatever.com"

### Для пользователей Unix/Mac:

git config --global core.autocrlf input

git config --global core.safecrlf warn

### Для пользователей Windows:

git config --global core.autocrlf true

git config --global core.safecrlf warn

### Установка отображения unicode

По умолчанию, git будет печатать не-[ASCII символы](https://unicode-table.com/ru/#basic-latin) в именах файлов в виде восьмеричных последовательностей \nnn. Чтобы избежать нечитаемых строк, установите соответствующий флаг

git config --global core.quotepath off

### 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 /home/user/my\_project |

Для Windows:

|  |
| --- |
| $ cd /Users/user/my\_project |

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.

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' |

### Клонирование существующего репозитория

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/EvgeniyUryvaev/company.git

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/EvgeniyUryvaev/company.git>

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 git@github.com:EvgeniyUryvaev/company.git , использующий протокол передачи SSH.

### Проверка состояния репозитория и фиксация изменений

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.

## Запись изменений в репозиторий

### 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 нём нет отслеживаемых изменённых 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

Предположим, вы добавили в свой проект новый файл, простой файл README. Если этого файла раньше не было, и вы выполните git status, вы увидите свой неотслеживаемый файл вот так:

|  |
| --- |
| $ echo 'My Project' > README  $ git status  On branch master  Your branch is up-to-date with 'origin/master'.  Untracked files:  (use "git add <file>..." to include in what will be committed)  README  nothing added to commit but untracked files present (use "git add" to track) |

Понять, что новый файл README неотслеживаемый можно по тому, что он находится в секции «Untracked files» в выводе команды status. Статус Untracked означает, что Git видит файл, которого не было в предыдущем снимке состояния (коммите); Git не станет добавлять его в ваши коммиты, пока вы его явно об этом не попросите

### 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

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 |

Файл CONTRIBUTING.md находится в секции «Changes not staged for commit» — это означает, что отслеживаемый файл был изменён в рабочем каталоге, но пока не проиндексирован. Чтобы проиндексировать его, необходимо выполнить команду git add

|  |
| --- |
| $ git add CONTRIBUTING.md  $ 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  modified: CONTRIBUTING.md |

Теперь оба файла проиндексированы и войдут в следующий коммит. В этот момент вы, предположим, вспомнили одно небольшое изменение, которое вы хотите сделать в CONTRIBUTING.md до коммита. Вы открываете файл, вносите и сохраняете необходимые изменения и вроде бы готовы к коммиту. Но давайте-ка ещё раз выполним git status:

|  |
| --- |
| $ vim CONTRIBUTING.md  $ 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  modified: CONTRIBUTING.md  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 |

Теперь CONTRIBUTING.md отображается как проиндексированный и непроиндексированный одновременно. Как такое возможно? Такая ситуация наглядно демонстрирует, что Git индексирует файл в точности в том состоянии, в котором он находился, когда вы выполнили команду git add. Если вы выполните коммит сейчас, то файл CONTRIBUTING.md попадёт в коммит в том состоянии, в котором он находился, когда вы последний раз выполняли команду git add , а не в том, в котором он находится в вашем рабочем каталоге в момент выполнения git commit. Если вы изменили файл после выполнения git add, вам придётся снова выполнить git add, чтобы проиндексировать последнюю версию файла:

|  |
| --- |
| $ git add CONTRIBUTING.md  $ 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  modified: CONTRIBUTING.md |

### 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.

Допустим, вы снова изменили и проиндексировали файл README, а затем изменили файл CONTRIBUTING.md без индексирования. Если вы выполните команду git status, вы опять увидите что-то вроде:

|  |
| --- |
| $ 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)  modified: 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 |

Чтобы увидеть, что же вы изменили, но пока не проиндексировали, наберите git diff без аргументов:

|  |
| --- |
| $ git diff  diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md  index 8ebb991..643e24f 100644  --- a/CONTRIBUTING.md  +++ b/CONTRIBUTING.md  @@ -65,7 +65,8 @@ branch directly, things can get messy.  Please include a nice description of your changes when you submit your PR;  if we have to read the whole diff to figure out why you're contributing  in the first place, you're less likely to get feedback and have your change  -merged in.  +merged in. Also, split your changes into comprehensive chunks if you patch is  +longer than a dozen lines.  If you are starting to work on a particular area, feel free to submit a PR  that highlights your work in progress (and note in the PR title that it's |

Эта команда сравнивает содержимое вашего рабочего каталога с содержимым индекса. Результат показывает ещё не проиндексированные изменения.

Если вы хотите посмотреть, что вы проиндексировали и что войдёт в следующий коммит, вы можете выполнить git diff --staged. Эта команда сравнивает ваши проиндексированные изменения с последним коммитом:

|  |
| --- |
| $ git diff --staged  diff --git a/README b/README  new file mode 100644  index 0000000..03902a1  --- /dev/null  +++ b/README  @@ -0,0 +1 @@  +My Project |

Важно отметить, что git diff сама по себе не показывает все изменения сделанные с последнего коммита — только те, что ещё не проиндексированы. Такое поведение может сбивать с толку, так как если вы проиндексируете все свои изменения, то git diff ничего не вернёт.

Другой пример: вы проиндексировали файл CONTRIBUTING.md и затем изменили его, вы можете использовать git diff для просмотра как проиндексированных изменений в этом файле, так и тех, что пока не проиндексированы. Если наше окружение выглядит вот так:

|  |
| --- |
| $ git add CONTRIBUTING.md  $ echo '# test line' >> CONTRIBUTING.md  $ 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)  modified: CONTRIBUTING.md  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 |

Используйте git diff для просмотра непроиндексированных изменений

|  |
| --- |
| $ git diff  diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md  index 643e24f..87f08c8 100644  --- a/CONTRIBUTING.md  +++ b/CONTRIBUTING.md  @@ -119,3 +119,4 @@ at the  ## Starter Projects  See our [projects list](https://github.com/libgit2/libgit2/blob/development/PROJECTS.md).  +# test line |

а так же git diff --cached для просмотра проиндексированных изменений (--staged и --cached синонимы):

|  |
| --- |
| $ git diff --cached  diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md  index 8ebb991..643e24f 100644  --- a/CONTRIBUTING.md  +++ b/CONTRIBUTING.md  @@ -65,7 +65,8 @@ branch directly, things can get messy.  Please include a nice description of your changes when you submit your PR;  if we have to read the whole diff to figure out why you're contributing  in the first place, you're less likely to get feedback and have your change  -merged in.  +merged in. Also, split your changes into comprehensive chunks if you patch is  +longer than a dozen lines.  If you are starting to work on a particular area, feel free to submit a PR  that highlights your work in progress (and note in the PR title that it's |

### Коммит изменений

Теперь, когда ваш индекс находится в таком состоянии, как вам и хотелось, вы можете зафиксировать свои изменения. Запомните, всё, что до сих пор не проиндексировано — любые файлы, созданные или изменённые вами, и для которых вы не выполнили git add после редактирования — не войдут в этот коммит. Они останутся изменёнными файлами на вашем диске. В нашем случае, когда вы в последний раз выполняли git status, вы видели что всё проиндексировано, и вот, вы готовы к коммиту. Простейший способ зафиксировать изменения — это набрать git commit:

Простейший способ зафиксировать изменения — это набрать git commit:

|  |
| --- |
| $ git commit |

Эта команда откроет выбранный вами текстовый редактор.

В редакторе будет отображён следующий текст (это пример окна Vim):

|  |
| --- |
| # Please enter the commit message for your changes. Lines starting  # with '#' will be ignored, and an empty message aborts the commit.  # On branch master  # Your branch is up-to-date with 'origin/master'.  #  # Changes to be committed:  # new file: README  # modified: CONTRIBUTING.md  #  ~  ~  ~  ".git/COMMIT\_EDITMSG" 9L, 283C |

Вы можете видеть, что комментарий по умолчанию для коммита содержит закомментированный результат работы команды git status и ещё одну пустую строку сверху. Вы можете удалить эти комментарии и набрать своё сообщение или же оставить их для напоминания о том, что вы фиксируете.

Есть и другой способ — вы можете набрать свой комментарий к коммиту в командной строке вместе с командой commit указав его после параметра -m, как в следующем примере:

|  |
| --- |
| $ git commit - m “my comments ” |

|  |
| --- |
| $ git commit -m "Story 182: fix benchmarks for speed"  [master 463dc4f] Story 182: fix benchmarks for speed  2 files changed, 2 insertions(+)  create mode 100644 README |

### Игнорирование индексации

Добавление параметра -a в команду git commit заставляет Git автоматически индексировать каждый уже отслеживаемый на момент коммита файл, позволяя вам обойтись без git add:

|  |
| --- |
| $ git status  On branch master  Your branch is up-to-date with 'origin/master'.  Changes not staged for commit:  (use "git add <file>..." to update what will be committed)  (use "git checkout -- <file>..." to discard changes in working directory)  modified: CONTRIBUTING.md  no changes added to commit (use "git add" and/or "git commit -a")  $ git commit -a -m 'Add new benchmarks'  [master 83e38c7] Add new benchmarks  1 file changed, 5 insertions(+), 0 deletions(-) |

В данном случае перед коммитом вам не нужно выполнять git add для файла CONTRIBUTING.md, потому что флаг -a включает все файлы

|  |  |
| --- | --- |
| git status | Просмотр статуса нужного репозитория, его действие распространяется на подготовленные, неподготовленные и неотслеживаемые файлы. |
| git add <file> | Проиндексировать все изменения в файле <file> для следующего коммита. |
| git add <directory> | Проиндексировать все изменения в каталоге <directory> для следующего коммита. |
| git add --all | Добавляет все измененные и не отслеживаемые файлы в репозиторий и обновляет дерево изменений репозитория. |
| git reset | Команда используется для отмены коммита или проиндексированного снимка состояния |
| git commit | Команда делает для проекта снимок текущего состояния изменений, добавленных в раздел проиндексированных файлов |
| git commit -m | Быстрая команда, которая создает коммит с указанным комментарием |
| git commit -am | Команда с параметрами -a и -m. Эта комбинация параметров создает коммит всех проиндексированных изменений и добавляет к коммиту подставленный комментарий. |
| git commit --amend | Этот параметр добавляет команде commit новый уровень функциональности. При передаче этого параметра будет изменен последний коммит. Вместо создания нового коммита проиндексированные изменения будут добавлены в предыдущий коммит. Кроме того, эта команда откроет настроенный текстовый редактор системы с предложением изменить указанный ранее комментарий к коммиту. |

## Просмотр истории коммитов

### Просмотр 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/EvgeniyUryvaev/company.git |

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), внесённую 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 |

|  |
| --- |
| $ 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 внесённых изменений. Так же есть возможность использовать серию опций для обобщения. Например, если вы хотите увидеть сокращённую статистику для каждого коммита, вы можете использовать опцию --stat:

|  |
| --- |
| git log --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 изменённых 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 |

|  |
| --- |
| $ git log --pretty=oneline  ca82a6dff817ec66f44342007202690a93763949 Change version number  085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 Remove unnecessary test  a11bef06a3f659402fe7563abf99ad00de2209e6 Initial commit |

### Таблица опций для команды git log

|  |  |
| --- | --- |
| -p | Показывает патч для каждого коммита. |
| --stat | Показывает статистику изменённых файлов для каждого коммита. |
| --shortstat | Отображает только строку с количеством изменений/вставок/удалений для команды --stat. |
| --name-only | Показывает список изменённых файлов после информации о коммите. |
| --name-status | Показывает список файлов, которые добавлены/изменены/удалены. |
| --abbrev-commit | Показывает только несколько символов SHA-1 чек-суммы вместо всех 40. |
| --relative-date | Отображает дату в относительном формате (например, «2 weeks ago») вместо стандартного формата даты. |
| --graph | Отображает ASCII граф с ветвлениями и историей слияний. |
| --pretty | Показывает коммиты в альтернативном формате. Возможные варианты опций: oneline, short, full, fuller и format (с помощью последней можно указать свой формат |
| --oneline | Сокращение для одновременного использования опций --pretty=oneline --abbrev-commit. |

## Операции отмены

### 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.

Если вы сделали коммит и поняли, что забыли проиндексировать изменения в файле, который хотели добавить в коммит, то можно сделать следую

|  |
| --- |
| $ git commit -m 'Initial commit'  $ git add forgotten\_file  $ git commit --amend |

### Отмена индексации файла

Команда для отмены индексации в файле git reset HEAD <file>…​

|  |
| --- |
| $ 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 |

|  |
| --- |
| $ 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 |

### Отмена изменений в файле

Команда для отмены изменений в файле git checkout -- <file>...

|  |
| --- |
| $ git checkout -- CONTRIBUTING.md  $ git status  On branch master  Changes to be committed:  (use "git reset HEAD <file>..." to unstage)  renamed: README.md -> README |

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 |

Откат изменений выполнен.

## 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.

### 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 |

Чтобы просмотреть адреса для чтения и записи, привязанные к репозиторию используте ключ -v:

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

### Добавление удалённых репозиториев

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.

### 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.

### 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.

### Просмотр удалённого 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.

### 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.

|  |  |
| --- | --- |
| git show | Команда git show используется для отображения полной информации о любом объекте в Git, будь то коммит или ветка. По умолчанию git show отображает информацию коммита, на который в данный момент времени указывает HEAD. |
| git reflog | Эта команда выводит упорядоченный список коммитов, на которые указывал HEAD. Грубо говоря, она отображает историю всех ваших перемещений по проекту. |

# Алиасы

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](mailto: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'

# Ветвление

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, 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 блоб 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.



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



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.



### 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



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.



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.



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

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 |



 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 откатили все изменения ветки testing и можете продолжать в другом направлении.

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.



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 |

### Основы ветвления и слияния

### Основы ветвления и слияния

Давайте рассмотрим простой пример рабочего процесса, который может быть полезен в вашем проекте. Ваша работа построена так:

1. Вы работаете над сайтом.
2. Вы создаете ветку для реализации новой функциональности в соответствии с пользовательской историей.
3. Вы работаете в этой ветке.

В этот момент вы получаете сообщение, что обнаружена критическая ошибка, требующая скорейшего исправления. Ваши действия:

1. Переключиться на основную ветку.
2. Создать ветку для добавления исправления.
3. После тестирования слить ветку, содержащую исправление, с основной веткой.
4. Переключиться назад в ветку для реализации пользовательской истории и продолжить работать.

### Основы ветвления

Предположим, вы работаете над проектом и уже имеете несколько коммитов.



Чтобы создать ветку и сразу переключиться на неё, можно выполнить команду git checkout с параметром –b

|  |
| --- |
| $ git checkout -b iss53  Switched to a new branch "iss53" |

Это то же самое, что и:

|  |
| --- |
| $ git branch iss53  $ git checkout iss53 |

Создаем новый указатель ветки



Вы работаете над сайтом и делаете коммиты. Это приводит к тому, что ветка iss53 движется вперед, так как вы переключились на неё ранее (HEAD указывает на неё).

|  |
| --- |
| $ vim index.html  $ git commit -a -m 'Create new footer [issue 53]' |



И тут вы получаете сообщение об обнаружении на сайте уязвимости, и эту уязвимость устранить нужно немедленно. Благодаря Git вам не придётся ни пытаться реализовать исправление вместе с изменениями, которые вы сделали в ходе разработки iss53, ни прилагать усилия для отката этих изменений и возвращения к исходному состоянию перед началом разработки исправления. Все, что вам нужно — переключиться на ветку master.

Имейте в виду, что если рабочий каталог или индекс содержат незафиксированные изменения, конфликтующие с веткой, на которую вы хотите переключиться, то Git не позволит переключить ветки. Лучше всего переключаться из чистого рабочего состояния проекта: все изменённые файлы добавить в индекс и сделать коммит. Есть способы обойти это (припрятать изменения (stash) или добавить их в последний коммит (amend)). Теперь предположим, что вы зафиксировали все свои изменения и можете переключиться на ветку master:

|  |
| --- |
| $ git checkout master  Switched to branch 'master' |

С этого момента ваш рабочий каталог имеет точно такой же вид, какой был перед началом работы над задачей #53, и вы можете сосредоточиться на работе над исправлением. Важно запомнить: когда вы переключаете ветки, Git возвращает состояние рабочего каталога к тому виду, какой он имел в момент последнего коммита в переключаемую ветку. Он добавляет, удаляет и изменяет файлы автоматически, чтобы состояние рабочего каталога соответствовало тому, когда был сделан последний коммит.

Теперь вы можете перейти к написанию исправления. Давайте создадим новую ветку, в которой реализуем исправление.

|  |
| --- |
| $ git checkout -b hotfix  Switched to a new branch 'hotfix'  $ vim index.html  $ git commit -a -m 'Fix broken email address'  [hotfix 1fb7853] Fix broken email address  1 file changed, 2 insertions(+) |



Ветка hotfix основана на ветке master

Вы можете прогнать тесты, чтобы убедиться, что ваше уязвимость в самом деле исправлена. И если это так — выполнить слияние ветки hotfix с веткой master для включения изменений в продукт. Это делается командой git merge:

|  |
| --- |
| $ git checkout master  $ git merge hotfix  Updating f42c576..3a0874c  Fast-forward  index.html | 2 ++  1 file changed, 2 insertions(+) |

Заметили фразу «fast-forward» в этом слиянии? Git просто переместил указатель ветки вперед, потому что коммит C4, на который указывает слитая ветка hotfix, был прямым потомком коммита C2, на котором вы находились до этого. Другими словами, если коммит сливается с тем, до которого можно добраться, двигаясь по истории вперёд, Git упрощает слияние, просто перенося указатель ветки вперед, потому что в этом случае нет никаких разнонаправленных изменений, которые нужно было бы свести воедино. Это называется «fast-forward».

Теперь ваши изменения включены в коммит, на который указывает ветка master, и исправление можно внедрять.



После внедрения вашего архиважного исправления вы готовы вернуться к работе над тем, что были вынуждены отложить. Но сначала нужно удалить ветку hotfix, потому что она больше не нужна — ветка master указывает на то же самое место. Для удаления ветки выполните команду git branch с параметром -d:

|  |
| --- |
| $ git branch -d hotfix  Deleted branch hotfix (3a0874c). |

Теперь вы можете переключиться обратно на ветку iss53 и продолжить работу над задачей

|  |
| --- |
| $ git checkout iss53  Switched to branch "iss53"  $ vim index.html  $ git commit -a -m 'Finish the new footer [issue 53]'  [iss53 ad82d7a] Finish the new footer [issue 53]  1 file changed, 1 insertion(+) |



Стоит обратить внимание на то, что все изменения из ветки hotfix не включены в вашу ветку iss53. Если их нужно включить, вы можете влить ветку master в вашу ветку iss53 командой git merge master, а можете отложить слияние этих изменений до завершения работы, и затем влить ветку iss53 в master.

### Основы слияния

Предположим, вы решили, что работа по проблеме закончена и её можно влить в ветку master. Для этого нужно выполнить слияние ветки iss53 точно так же, как вы делали это с веткой hotfix ранее. Все, что нужно сделать — переключиться на ветку, в которую вы хотите включить изменения, и выполнить команду git merge:

|  |
| --- |
| $ git checkout master  Switched to branch 'master'  $ git merge iss53  Merge made by the 'recursive' strategy.  index.html | 1 +  1 file changed, 1 insertion(+) |

Результат этой операции отличается от результата слияния ветки hotfix. В данном случае процесс разработки ответвился в более ранней точке. Так как коммит, на котором мы находимся, не является прямым родителем ветки, с которой мы выполняем слияние, Git придётся немного потрудиться. В этом случае Git выполняет простое трёхстороннее слияние, используя последние коммиты объединяемых веток и общего для них родительского коммита.



Вместо того, чтобы просто передвинуть указатель ветки вперёд, Git создаёт новый результирующий снимок трёхстороннего слияния, а затем автоматически делает коммит. Этот особый коммит называют коммитом слияния, так как у него более одного предка.



Теперь, когда изменения слиты, ветка iss53 больше не нужна. Вы можете закрыть задачу в системе отслеживания ошибок и удалить ветку:

|  |
| --- |
| $ git branch -d iss53 |

### Основные конфликты слияния

Иногда процесс не проходит гладко. Если вы изменили одну и ту же часть одного и того же файла по-разному в двух объединяемых ветках, Git не сможет их чисто объединить. Если ваше исправление ошибки #53 потребовало изменить ту же часть файла что и hotfix, вы получите примерно такое сообщение о конфликте слияния:

|  |
| --- |
| $ git merge iss53  Auto-merging index.html  CONFLICT (content): Merge conflict in index.html  Automatic merge failed; fix conflicts and then commit the result. |

Git не создал коммит слияния автоматически. Он остановил процесс до тех пор, пока вы не разрешите конфликт. Чтобы в любой момент после появления конфликта увидеть, какие файлы не объединены, вы можете запустить git status:

|  |
| --- |
| $ git status  On branch master  You have unmerged paths.  (fix conflicts and run "git commit")  Unmerged paths:  (use "git add <file>..." to mark resolution)  both modified: index.html  no changes added to commit (use "git add" and/or "git commit -a") |

Всё, где есть неразрешённые конфликты слияния, перечисляется как неслитое. В конфликтующие файлы Git добавляет специальные маркеры конфликтов, чтобы вы могли исправить их вручную. В вашем файле появился раздел, выглядящий примерно так:

|  |
| --- |
| <<<<<<< HEAD:index.html  <div id="footer">contact : email.support@github.com</div>  =======  <div id="footer">  please contact us at support@github.com  </div>  >>>>>>> iss53:index.html |

Это означает, что версия из HEAD (вашей ветки master, поскольку именно её вы извлекли перед запуском команды слияния) — это верхняя часть блока (всё, что над =======), а версия из вашей ветки iss53 представлена в нижней части. Чтобы разрешить конфликт, придётся выбрать один из вариантов, либо объединить содержимое по-своему. Например, вы можете разрешить конфликт, заменив весь блок следующим:

|  |
| --- |
| <div id="footer">  please contact us at email.support@github.com  </div> |

В этом разрешении есть немного от каждой части, а строки <<<<<<<, ======= и >>>>>>> полностью удалены. Разрешив каждый конфликт во всех файлах, запустите git add для каждого файла, чтобы отметить конфликт как решённый. Добавление файла в индекс означает для Git, что все конфликты в нём исправлены.

Если вы хотите использовать графический инструмент для разрешения конфликтов, можно запустить git mergetool, который проведет вас по всем конфликтам:

|  |
| --- |
| $ git mergetool  This message is displayed because 'merge.tool' is not configured.  See 'git mergetool --tool-help' or 'git help config' for more details.  'git mergetool' will now attempt to use one of the following tools:  opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge araxis bc3 codecompare vimdiff emerge  Merging:  index.html  Normal merge conflict for 'index.html':  {local}: modified file  {remote}: modified file  Hit return to start merge resolution tool (opendiff): |

Если вы хотите использовать инструмент слияния не по умолчанию (в данном случае Git выбрал opendiff, поскольку команда запускалась на Mac), список всех поддерживаемых инструментов представлен вверху после фразы «one of the following tools». Просто введите название инструмента, который хотите использовать.

После выхода из инструмента слияния Git спросит об успешности процесса. Если вы ответите скрипту утвердительно, то он добавит файл в индекс, чтобы отметить его как разрешённый. Теперь можно снова запустить git status, чтобы убедиться в отсутствии конфликтов:

|  |
| --- |
| $ git status  On branch master  All conflicts fixed but you are still merging.  (use "git commit" to conclude merge)  Changes to be committed:  modified: index.html |

Если это вас устраивает и вы убедились, что все файлы, где были конфликты, добавлены в индекс — выполните команду git commit для создания коммита слияния. Комментарий к коммиту слияния по умолчанию выглядит примерно так:

|  |
| --- |
| Merge branch 'iss53'  Conflicts:  index.html  #  # It looks like you may be committing a merge.  # If this is not correct, please remove the file  # .git/MERGE\_HEAD  # and try again.  # Please enter the commit message for your changes. Lines starting  # with '#' will be ignored, and an empty message aborts the commit.  # On branch master  # All conflicts fixed but you are still merging.  #  # Changes to be committed:  # modified: index.html  # |

Если вы считаете, что коммит слияния требует дополнительных пояснений — опишите как были разрешены конфликты и почему были применены именно такие изменения, если это не очевидно

### Управление ветками

Теперь, когда вы уже попробовали создавать, объединять и удалять ветки, пора познакомиться с некоторыми инструментами для управления ветками, которые вам пригодятся, когда вы начнёте использовать ветки постоянно.

Команда git branch делает несколько больше, чем просто создаёт и удаляет ветки. При запуске без параметров, вы получите простой список имеющихся у вас веток:

|  |
| --- |
| $ git branch  iss53  \* master  testing |

Обратите внимание на символ \*, стоящий перед веткой master: он указывает на ветку, на которой вы находитесь в настоящий момент (т. е. ветку, на которую указывает HEAD). Это означает, что если вы сейчас сделаете коммит, ветка master переместится вперёд в соответствии с вашими последними изменениями. Чтобы посмотреть последний коммит на каждой из веток, выполните команду git branch -v:

|  |
| --- |
| $ git branch -v  iss53 93b412c Fix javascript issue  \* master 7a98805 Merge branch 'iss53'  testing 782fd34 Add scott to the author list in the readme |

Опции --merged и --no-merged могут отфильтровать этот список для вывода только тех веток, которые слиты или ещё не слиты в текущую ветку. Чтобы посмотреть те ветки, которые вы уже слили с текущей, можете выполнить команду git branch --merged:

|  |
| --- |
| $ git branch --merged  iss53  \* master |

Ветка iss53 присутствует в этом списке потому что вы ранее слили её в master. Те ветки из этого списка, перед которыми нет символа \*, можно смело удалять командой git branch -d; наработки из этих веток уже включены в другую ветку, так что ничего не потеряется.

Чтобы увидеть все ветки, содержащие наработки, которые вы пока ещё не слили в текущую ветку, выполните команду git branch --no-merged:

|  |
| --- |
| $ git branch --no-merged  testing |

Вы увидите оставшуюся ветку. Так как она содержит ещё не слитые наработки, попытка удалить её командой git branch -d приведёт к ошибке:

|  |
| --- |
| $ git branch -d testing  error: The branch 'testing' is not fully merged.  If you are sure you want to delete it, run 'git branch -D testing'. |

Если вы действительно хотите удалить ветку вместе со всеми наработками, используйте опцию -D, как указано в подсказке.

### Переименование ветки

Предположим, у вас есть ветка с именем bad-branch-name, и вы хотите изменить её на corrected-branch-name, сохранив при этом всю историю. Вместе с этим, вы также хотите изменить имя ветки на удалённом сервере (GitHub, GitLab или другой сервер). Как это сделать?

Переименуйте ветку локально с помощью команды git branch --move:

|  |
| --- |
| $ git branch --move bad-branch-name corrected-branch-name |

Ветка bad-branch-name будет переименована в corrected-branch-name, но это изменение пока только локальное. Чтобы все остальные увидели исправленную ветку в удалённом репозитории, отправьте её туда:

|  |
| --- |
| $ git push --set-upstream origin corrected-branch-name |

Теперь проверим, где мы сейчас находимся:

|  |
| --- |
| $ git branch --all  \* corrected-branch-name  main  remotes/origin/bad-branch-name  remotes/origin/corrected-branch-name  remotes/origin/main |

Обратите внимание, что текущая ветка corrected-branch-name, которая также присутствует и на удалённом сервере. Однако, старая ветка всё ещё по-прежнему там, но её можно удалить с помощью команды:

|  |
| --- |
| $ git push origin --delete bad-branch-name |

Теперь старое имя ветки полностью заменено исправленным.

Переименуйте локальную ветку master в main с помощью следующей команды:

|  |
| --- |
| $ git branch --move master main |

После этого, локальной ветки master больше не существует, потому что она была переименована в ветку main.

Чтобы все остальные могли видеть новую ветку main, вам нужно отправить её в общий репозиторий. Это делает переименованную ветку доступной в удалённом репозитории.

|  |
| --- |
| $ git push --set-upstream origin main |

В итоге, состояние репозитория становится следующим:

|  |
| --- |
| $ git branch --all  \* main  remotes/origin/HEAD -> origin/master  remotes/origin/main  remotes/origin/master |

Ваша локальная ветка master исчезла, так как она заменена веткой main. Ветка main доступна в удалённом репозитории. Старая ветка master всё ещё присутствует в удалённом репозитории. Остальные участники будут продолжать использовать ветку master в качестве основы для своей работы, пока вы не совершите ряд дополнительных действий.

Теперь, для завершения перехода на новую ветку перед вами стоят следующие задачи:

* Все проекты, которые зависят от текущего, должны будут обновить свой код и/или конфигурацию.
* Обновите конфигурацию всех запускаемых тестов.
* Исправьте скрипты сборки и публикации артефактов.
* Поправьте настройки репозитория на сервере: задайте новую ветку по умолчанию, обновите правила слияния, а также прочие настройки, которые зависят от имени веток.
* Обновите документацию, исправив ссылки, указывающие на старую ветку.
* Слейте или отмените запросы на слияние изменений, нацеленные на старую ветку.

После того, как вы выполнили все эти задачи и уверены, что ветка main работает так же, как ветка master, вы можете удалить ветку master:

|  |
| --- |
| $ git push origin --delete master  prev | next  About this site |

### Работа с ветками

Теперь, когда вы познакомились с основами ветвления и слияния, возникает вопрос: что можно или нужно делать с этим? В этом разделе мы разберём некоторые основные рабочие процессы, ставшие возможными благодаря облегчённой процедуре ветвления, которые вы возможно захотите применить в собственном цикле разработки.

**Долгоживущие ветки**

Так как в Git применяется простое трёхстороннее слияние, ничто не мешает многократно объединять ветки в течение длительного времени. Это значит, что у вас может быть несколько постоянно открытых веток, которые вы используете для разных этапов вашего цикла разработки; вы можете регулярно сливать изменения из одной ветки в другую.

Многие разработчики, использующие Git, придерживаются именно такого подхода, оставляя полностью стабильный код только в ветке master — возможно, только тот код, который был или будет выпущен. При этом существует и параллельная ветка с именем develop или next, предназначенная для работы и тестирования стабильности; она не обязательно должна быть всегда стабильной, но при достижении стабильного состояния её содержимое можно слить в ветку master. Она используется для слияния завершённых задач из тематических веток (временных веток наподобие iss53), чтобы гарантировать, что эти задачи проходят тестирование и не вносят ошибок.

По сути, мы говорим про указатели, перемещающиеся по линии создаваемых вами коммитов. Стабильные ветки находятся в нижнем конце истории коммитов, а самые свежие наработки — ближе к её верхней части



В общем случае это можно представить в виде накопителей, в которых наборы коммитов перемещаются на более стабильный уровень только после полного тестирования.



Число уровней стабильности можно увеличить. В крупных проектах зачастую появляется ветка proposed или pu (предлагаемые обновления), объединяющая ветки с содержимым, которое ещё не готово к включению в ветки next или master. Идея состоит в том, что каждая ветка представляет собой определённый уровень стабильности; как только он повышается, содержимое сливается в ветку уровнем выше. Разумеется, можно вообще обойтись без долгоживущих веток, но зачастую они имеют смысл, особенно при работе над большими и сложными проектами.

### Тематические ветки

А вот такая вещь, как тематические ветки, полезна вне зависимости от величины проекта. Тематической веткой называется временная ветка, создаваемая и используемая для работы над конкретной функциональной возможностью или решения сопутствующих задач. Скорее всего, при работе с другими системами контроля версий вы никогда ничего подобного не делали, так как там создание и слияние веток — затратные операции. Но в Git это обычное дело — много раз в день создавать ветки, работать с ними, сливать их и удалять.

Пример тематических веток вы видели в предыдущем разделе, когда мы создавали ветки iss53 и hotfix. В каждой из них было создано несколько коммитов, после чего, сразу же после слияния с основной веткой, они были удалены. Такая техника позволяет быстро и полностью осуществлять переключения контекста — так как работа разделена по уровням и все изменения в конкретной ветке относятся к определённой теме, что позволяет легко увидеть что именно было сделано во время процедуры просмотра кода или аналогичной. Ветки с внесёнными в них изменениями можно хранить минуты, дни или даже месяцы, а слияние выполнить только когда это действительно потребуется, вне зависимости от порядка их создания.

Предположим, мы работаем в ветке master, ответвляемся для решения попутной проблемы (iss91), некоторое время занимаемся ею, затем создаём ветку, чтобы попробовать решить эту задачу другим способом (iss91v2), возвращаемся в ветку master и выполняем там некие действия, затем создаём новую ветку для изменений, в результате которых не уверены (ветка dumbidea). Результирующая история коммитов будет выглядеть примерно так:



Предположим, вам больше нравится второй вариант решения задачи (iss91v2), а ветку dumbidea вы показали коллегам, и оказалось, что там содержится гениальная идея. Фактически вы можете удалить ветку iss91 (потеряв коммиты C5 и C6) и слить две другие ветки. После этого история будет выглядеть так:



Важно помнить, что во время всех этих манипуляций ветки полностью локальны. Ветвления и слияния выполняются только в вашем Git репозитории — связь с сервером не требуется.

### Удалённые ветки

Удалённые ссылки — это ссылки (указатели) в ваших удалённых репозиториях, включая ветки, теги и так далее. Полный список удалённых ссылок можно получить с помощью команды git ls-remote <remote> или команды git remote show <remote> для получения удалённых веток и дополнительной информации. Тем не менее, более распространённым способом является использование веток слежения.

Ветки слежения — это ссылки на определённое состояние удалённых веток. Это локальные ветки, которые нельзя перемещать; Git перемещает их автоматически при любой коммуникации с удалённым репозиторием, чтобы гарантировать точное соответствие с ним. Представляйте их как закладки для напоминания о том, где ветки в удалённых репозиториях находились во время последнего подключения к ним.

Имена веток слежения имеют вид <remote>/<branch>. Например, если вы хотите посмотреть, как выглядела ветка master на сервере origin во время последнего соединения с ним, используйте ветку origin/master. Если вы с коллегой работали над одной задачей и он отправил на сервер ветку iss53, при том что у вас может быть своя локальная ветка iss53, удалённая ветка будет представлена веткой слежения с именем origin/iss53.

Возможно, всё это сбивает с толку, поэтому давайте рассмотрим на примере. Скажем, у вас в сети есть свой Git-сервер с адресом git.ourcompany.com. Если вы с него что-то клонируете, команда clone автоматически назовёт его origin, заберёт оттуда все данные, создаст указатель на то, на что там указывает ветка master, и назовёт его локально origin/master. Git также создаст вам локальную ветку master, которая будет начинаться там же, где и ветка master в origin, так что вам будет с чего начать.



Если вы сделаете что-то в своей локальной ветке master, а тем временем кто-то отправит изменения на сервер git.ourcompany.com и обновит там ветку master, то ваши истории продолжатся по-разному. Пока вы не свяжетесь с сервером origin ваш указатель origin/master останется на месте.

Локальная и удалённая работа может расходиться



Для синхронизации ваших изменений с удалённым сервером выполните команду git fetch <remote> (в нашем случае git fetch origin). Эта команда определяет какому серверу соответствует «origin» (в нашем случае это git.ourcompany.com), извлекает оттуда данные, которых у вас ещё нет, и обновляет локальную базу данных, сдвигая указатель origin/master на новую позицию.

git fetch обновляет ветки слежения



Чтобы продемонстрировать, как будут выглядеть удалённые ветки в ситуации с несколькими удалёнными серверами, предположим, что у вас есть ещё один внутренний Git-сервер, который используется для разработки только одной из ваших команд разработчиков. Этот сервер находится на git.team1.ourcompany.com. Вы можете добавить его в качестве новой удалённой ссылки для текущего проекта с помощью команды git remote add. Назовите этот удалённый сервер teamone — это имя будет сокращением вместо полного URL

Добавление ещё одного сервера в качестве удалённой ветки



Теперь вы можете выполнить команду git fetch teamone для получения всех изменений с сервера teamone, которых у вас нет локально. Так как в данный момент на этом сервере есть только те данные, что содержит сервер origin, Git ничего не получит, но создаст ветку слежения с именем teamone/master, которая будет указывать на тот же коммит, что и ветка master на сервере teamone.



### Отправка изменений

Когда вы хотите поделиться веткой, вам необходимо отправить её на удалённый сервер, где у вас есть права на запись. Ваши локальные ветки автоматически не синхронизируются с удалёнными при отправке — вам нужно явно указать те ветки, которые вы хотите отправить. Таким образом, вы можете использовать свои личные ветки для работы, которую не хотите показывать, а отправлять только те тематические ветки, над которыми вы хотите работать с кем-то совместно.

Если у вас есть ветка serverfix, над которой вы хотите работать с кем-то ещё, вы можете отправить её точно так же, как вы отправляли вашу первую ветку. Выполните команду git push <remote> <branch>:

|  |
| --- |
| $ git push origin serverfix  Counting objects: 24, done.  Delta compression using up to 8 threads.  Compressing objects: 100% (15/15), done.  Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.  Total 24 (delta 2), reused 0 (delta 0)  To https://github.com/schacon/simplegit  \* [new branch] serverfix -> serverfix |

Это в некотором роде сокращение. Git автоматически разворачивает имя ветки serverfix до refs/heads/serverfix:refs/heads/serverfix, что означает «возьми мою локальную ветку serverfix и обнови ей удалённую ветку serverfix». Мы подробно рассмотрим часть с refs/heads/ в главе Git изнутри, но обычно её можно пропустить. Вы также можете выполнить git push origin serverfix:serverfix — произойдёт то же самое — здесь говорится «возьми мою ветку serverfix и сделай её удалённой веткой serverfix». Можно использовать этот формат для отправки локальной ветки в удалённую ветку с другим именем. Если вы не хотите, чтобы на удалённом сервере ветка называлась serverfix, то вместо предыдущей команды выполните git push origin serverfix:awesomebranch, которая отправит локальную ветку serverfix в ветку awesomebranch удалённого репозитория.

В следующий раз, когда один из ваших соавторов будет получать обновления с сервера, он получит ссылку на то, на что указывает serverfix на сервере, как удалённую ветку origin/serverfix:

|  |
| --- |
| $ git fetch origin  remote: Counting objects: 7, done.  remote: Compressing objects: 100% (2/2), done.  remote: Total 3 (delta 0), reused 3 (delta 0)  Unpacking objects: 100% (3/3), done.  From https://github.com/schacon/simplegit  \* [new branch] serverfix -> origin/serverfix |

Необходимо отметить, что при получении данных создаются ветки слежения, вы не получаете автоматически для них локальных редактируемых копий. Другими словами, в нашем случае вы не получите новую ветку serverfix — только указатель origin/serverfix, который вы не можете изменять.

Чтобы слить эти наработки в свою текущую рабочую ветку, выполните git merge origin/serverfix. Если вам нужна локальная ветка serverfix, в которой вы сможете работать, то вы можете создать её на основе ветки слежения:

|  |
| --- |
| $ git checkout -b serverfix origin/serverfix  Branch serverfix set up to track remote branch serverfix from origin.  Switched to a new branch 'serverfix' |

Это даст вам локальную ветку, в которой можно работать и которая будет начинаться там же, где и origin/serverfix.

### Отслеживание веток

Получение локальной ветки из удалённой ветки автоматически создаёт то, что называется «веткой слежения» (а ветка, за которой следит локальная называется «upstream branch»). Ветки слежения — это локальные ветки, которые напрямую связаны с удалённой веткой. Если, находясь на ветке слежения, выполнить git pull, то Git уже будет знать с какого сервера получать данные и какую ветку использовать для слияния.

При клонировании репозитория, как правило, автоматически создаётся ветка master, которая следит за origin/master. Однако, при желании вы можете настроить отслеживание и других веток — следить за ветками на других серверах или отключить слежение за веткой master. Вы только что видели простейший пример, что сделать это можно с помощью команды git checkout -b <branch> <remote>/<branch>. Это часто используемая команда, поэтому Git предоставляет сокращённую форму записи в виде флага --track

|  |
| --- |
| $ git checkout --track origin/serverfix  Branch serverfix set up to track remote branch serverfix from origin.  Switched to a new branch 'serverfix' |

В действительности, это настолько распространённая команда, что существует сокращение для этого сокращения. Если вы пытаетесь извлечь ветку, которая не существует, но существует только одна удалённая ветка с точно таким же именем, то Git автоматически создаст ветку слежения:

|  |
| --- |
| $ git checkout serverfix  Branch serverfix set up to track remote branch serverfix from origin.  Switched to a new branch 'serverfix' |

Чтобы создать локальную ветку с именем, отличным от имени удалённой ветки, просто укажите другое имя:

|  |
| --- |
| $ git checkout -b sf origin/serverfix  Branch sf set up to track remote branch serverfix from origin.  Switched to a new branch 'sf' |

Теперь ваша локальная ветка sf будет автоматически получать изменения из origin/serverfix.

Если у вас уже есть локальная ветка и вы хотите настроить её на слежение за удалённой веткой, которую вы только что получили, или хотите изменить используемую upstream-ветку, то воспользуйтесь параметрами -u или --set-upstream-to для команды git branch, чтобы явно установить новое значение.

|  |
| --- |
| $ git branch -u origin/serverfix  Branch serverfix set up to track remote branch serverfix from origin. |

Если вы хотите посмотреть как у вас настроены ветки слежения, воспользуйтесь опцией -vv для команды git branch. Это выведет список локальных веток и дополнительную информацию о том, какая из веток отслеживается, отстаёт, опережает или всё сразу относительно отслеживаемой.

|  |
| --- |
| $ git branch -vv  iss53 7e424c3 [origin/iss53: ahead 2] Add forgotten brackets  master 1ae2a45 [origin/master] Deploy index fix  \* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] This should do it  testing 5ea463a Try something new |

Итак, здесь мы видим, что наша ветка iss53 следит за origin/iss53 и «опережает» её на два изменения — это значит, что у нас есть два локальных коммита, которые не отправлены на сервер. Мы также видим, что наша ветка master отслеживает ветку origin/master и находится в актуальном состоянии. Далее мы можем видеть, что локальная ветка serverfix следит за веткой server-fix-good на сервере teamone, опережает её на три коммита и отстает на один — это значит, что на сервере есть один коммит, который мы ещё не слили, и три локальных коммита, которые ещё не отправлены на сервер. В конце мы видим, что наша ветка testing не отслеживает удалённую ветку.

Важно отметить, что эти цифры описывают состояние на момент последнего получения данных с каждого из серверов. Эта команда не обращается к серверам, а лишь говорит вам о том, какая информация с этих серверов сохранена в локальном кэше. Если вы хотите иметь актуальную информацию об этих числах, вам необходимо получить данные со всех ваших удалённых серверов перед запуском команды. Сделать это можно вот так:

|  |
| --- |
| $ git fetch --all; git branch -vv |

### Получение изменений

Команда git fetch получает с сервера все изменения, которых у вас ещё нет, но не будет изменять состояние вашей рабочей копии. Эта команда просто получает данные и позволяет вам самостоятельно сделать слияние. Тем не менее, существует команда git pull, которая в большинстве случаев является командой git fetch, за которой непосредственно следует команда git merge. Если у вас настроена ветка слежения как показано в предыдущем разделе, или она явно установлена, или она была создана автоматически командами clone или checkout, git pull определит сервер и ветку, за которыми следит ваша текущая ветка, получит данные с этого сервера и затем попытается слить удалённую ветку.

Обычно, лучше явно использовать команды fetch и merge, поскольку магия git pull может часто сбивать с толку.

### Удаление веток на удалённом сервере

Скажем, вы и ваши соавторы закончили с нововведением и слили его в ветку master на удалённом сервере (или в какую-то другую ветку, где хранится стабильный код). Вы можете удалить ветку на удалённом сервере используя параметр --delete для команды git push. Для удаления ветки serverfix на сервере, выполните следующую команду:

|  |
| --- |
| $ git push origin --delete serverfix  To https://github.com/schacon/simplegit  - [deleted] serverfix |

Всё, что делает эта строка — удаляет указатель на сервере. Как правило, Git сервер хранит данные пока не запустится сборщик мусора, поэтому если ветка была удалена случайно, чаще всего её легко восстановить.

### Перебазирование

В Git есть два способа внести изменения из одной ветки в другую: слияние и перебазирование. В этом разделе вы узнаете, что такое перебазирование, как его осуществлять и в каких случаях этот удивительный инструмент использовать не следует.

Простейшее перебазирование

Если вы вернётесь к более раннему примеру из Основы слияния, вы увидите, что разделили свою работу и сделали коммиты в две разные ветки.



Как мы выяснили ранее, простейший способ выполнить слияние двух веток — это команда merge. Она осуществляет трёхстороннее слияние между двумя последними снимками сливаемых веток (C3 и C4) и самого недавнего общего для этих веток родительского снимка (C2), создавая новый снимок (и коммит).



Тем не менее есть и другой способ: вы можете взять те изменения, что были представлены в C4, и применить их поверх C3. В Git это называется перебазированием. С помощью команды rebase вы можете взять все коммиты из одной ветки и в том же порядке применить их к другой ветке.

В данном примере переключимся на ветку experiment и перебазируем её относительно ветки master следующим образом:

|  |
| --- |
| $ git checkout experiment  $ git rebase master  First, rewinding head to replay your work on top of it...  Applying: added staged command |

Это работает следующим образом: берётся общий родительский снимок двух веток (текущей, и той, поверх которой вы выполняете перебазирование), определяется дельта каждого коммита текущей ветки и сохраняется во временный файл, текущая ветка устанавливается на последний коммит ветки, поверх которой вы выполняете перебазирование, а затем по очереди применяются дельты из временных файлов.



После этого вы можете переключиться обратно на ветку master и выполнить слияние перемоткой.

|  |
| --- |
| $ git checkout master  $ git merge experiment |



Теперь снимок, на который указывает C4' абсолютно такой же, как тот, на который указывал C5 в примере с трёхсторонним слиянием. Нет абсолютно никакой разницы в конечном результате между двумя показанными примерами, но перебазирование делает историю коммитов чище. Если вы взглянете на историю перебазированной ветки, то увидите, что она выглядит абсолютно линейной: будто все операции были выполнены последовательно, даже если изначально они совершались параллельно.

Часто вы будете делать так для уверенности, что ваши коммиты могут быть бесконфликтно слиты в удалённую ветку — возможно, в проекте, куда вы пытаетесь внести вклад, но владельцем которого вы не являетесь. В этом случае вам следует работать в своей ветке и затем перебазировать вашу работу поверх origin/master, когда вы будете готовы отправить свои изменения в основной проект. Тогда владельцу проекта не придётся делать никакой лишней работы — всё решится простой перемоткой или бесконфликтным слиянием.

Учтите, что снимок, на который ссылается ваш последний коммит — является ли он последним коммитом после перебазирования или коммитом слияния после слияния — в обоих случаях это один и тот же снимок, отличаются только истории коммитов. Перебазирование повторяет изменения из одной ветки поверх другой в том порядке, в котором эти изменения были сделаны, в то время как слияние берет две конечные точки и сливает их вместе.

### Более интересные перемещения

Также возможно сделать так, чтобы при перебазировании воспроизведение коммитов применялось к совершенно другой ветке. Для примера возьмём История разработки с тематической веткой, ответвлённой от другой тематической ветки. Вы создаёте тематическую ветку server, чтобы добавить в проект некоторую функциональность для серверной части, и делаете коммит. Затем вы выполнили ответвление, чтобы сделать изменения для клиентской части, и создали несколько коммитов. Наконец, вы вернулись на ветку server и сделали ещё несколько коммитов.



Рисунок 39. История разработки с тематической веткой, ответвлённой от другой тематической ветки

Предположим, вы решили, что хотите внести изменения клиентской части в основную линию разработки для релиза, но при этом не хотите добавлять изменения серверной части до полного тестирования. Вы можете взять изменения из ветки client, которых нет в server (C8 и C9), и применить их на ветке master при помощи опции --onto команды git rebase

|  |
| --- |
| $ git rebase --onto master server client |

В этой команде говорится: «Переключись на ветку client, найди изменения относительно ветки server и примени их для ветки master». Несмотря на некоторую сложность этого способа, результат впечатляет.



Рисунок 40. Перемещение тематической ветки, ответвлённой от другой тематической ветки

Теперь вы можете выполнить перемотку (fast-forward) для ветки master (см Перемотка ветки master для добавления изменений из ветки client):

|  |
| --- |
| $ git checkout master  $ git merge client |



Рисунок 41. Перемотка ветки master для добавления изменений из ветки client

Представим, что вы решили добавить наработки и из ветки server. Вы можете выполнить перебазирование ветки server относительно ветки master без предварительного переключения на неё при помощи команды git rebase <basebranch> <topicbranch>, которая извлечёт тематическую ветку (в данном случае server) и применит изменения в ней к базовой ветке (master):

|  |
| --- |
| $ git rebase master server |

Это повторит работу, сделанную в ветке server поверх ветки master, как показано на рисунке:



Рисунок 42. Перебазирование ветки server на вершину ветки master

После чего вы сможете выполнить перемотку основной ветки (master):

|  |
| --- |
| $ git checkout master  $ git merge server |

Теперь вы можете удалить ветки client и server, поскольку весь ваш прогресс уже интегрирован и тематические ветки больше не нужны, а полную историю вашего рабочего процесса отражает рисунок Окончательная история коммитов:

|  |
| --- |
| $ git branch -d client  $ git branch -d server |



### Опасности перемещения

Но даже перебазирование, при всех своих достоинствах, не лишено недостатков, которые можно выразить одной строчкой:

**Не перемещайте коммиты, уже отправленные в публичный репозиторий**

Если вы будете придерживаться этого правила, всё будет хорошо. Если не будете, люди возненавидят вас, а ваши друзья и семья будут вас презирать.

Когда вы что-то перемещаете, вы отменяете существующие коммиты и создаёте новые, **похожие** на старые, но являющиеся другими. Если вы куда-нибудь отправляете свои коммиты и другие люди забирают их себе и в дальнейшем основывают на них свою работу, а затем вы переделываете эти коммиты командой git rebase и выкладываете их снова, то ваши коллеги будут вынуждены заново выполнять слияние для своих наработок. В итоге, когда вы в очередной раз попытаетесь включить их работу в свою, вы получите путаницу.

Давайте рассмотрим пример того, как перемещение публично доступных наработок может вызвать проблемы. Предположим, вы клонировали репозиторий с сервера и сделали какую-то работу. И ваша история коммитов выглядит так:



Рисунок 44. Клонирование репозитория и выполнение в нём какой-то работы

Теперь кто-то другой внёс свои изменения, слил их и отправил на сервер. Вы стягиваете их к себе, включая новую удалённую ветку, что изменяет вашу историю следующим образом:



Рисунок 45. Извлекаем ещё коммиты и сливаем их со своей работой

Затем автор коммита слияния решает вернуться назад и перебазировать свою ветку; выполнив git push --force, он перезаписывает историю на сервере. При получении изменений с сервера вы получите и новые коммиты.



Рисунок 46. Кто-то выложил перебазированные коммиты, отменяя коммиты, на которых основывалась ваша работа

Теперь вы оба в неловком положении. Если вы выполните git pull, вы создадите коммит слияния, включающий обе линии истории, и ваш репозиторий будет выглядеть следующим образом:



Рисунок 47. Вы снова выполняете слияние для той же самой работы в новый коммит слияния

Если вы посмотрите git log в этот момент, вы увидите два коммита с одинаковыми авторами, датой и сообщением, что может сбить с толку. Помимо этого, если вы отправите свою историю на удалённый сервер в таком состоянии, вы вернёте все эти перебазированные коммиты на сервер, что ещё больше всех запутает. Логично предположить, что разработчик не хочет, чтобы C4 и C6 были в истории, и именно поэтому она перебазируется в первую очередь.

### Меняя базу, меняй основание

Если вы попали в такую ситуацию, у Git есть особая магия чтобы вам помочь. Если кто-то в вашей команде форсирует отправку изменений на сервер, переписывающих работу, на которых базировалась ваша работа, то ваша задача будет состоять в определении того, что именно было ваше, а что было переписано **ими**.

Оказывается, что помимо контрольной суммы коммита SHA-1, Git также вычисляет контрольную сумму отдельно для патча, входящего в этот коммит. Это контрольная сумма называется «patch-id».

Если вы скачаете перезаписанную историю и перебазируете её поверх новых коммитов вашего коллеги, в большинстве случаев Git успешно определит, какие именно изменения были внесены вами, и применит их поверх новой ветки.

К примеру, если в предыдущем сценарии вместо слияния в [Кто-то выложил перебазированные коммиты, отменяя коммиты, на которых основывалась ваша работа](https://git-scm.com/book/ru/v2/ch00/r_pre_merge_rebase_work) мы выполним git rebase teamone/master, Git будет:

* Определять, какая работа уникальна для вашей ветки (C2, C3, C4, C6, C7)
* Определять, какие коммиты не были коммитами слияния (C2, C3, C4)
* Определять, что не было перезаписано в основной ветке (только C2 и C3, поскольку C4 — это тот же патч, что и C4')
* Применять эти коммиты к ветке teamone/master

Таким образом, вместо результата, который мы можем наблюдать на [Вы снова выполняете слияние для той же самой работы в новый коммит слияния](https://git-scm.com/book/ru/v2/ch00/r_merge_rebase_work), у нас получилось бы что-то вроде [Перемещение в начало force-pushed перемещённой работы](https://git-scm.com/book/ru/v2/ch00/r_rebase_rebase_work).



Рисунок 48. Перемещение в начало force-pushed перемещённой работы

Это возможно, если C4 и C4' фактически являются одним и тем же патчем, который был сделан вашим коллегой. В противном случае rebase не сможет определить дубликат и создаст ещё один патч, подобный C4 (который с большой вероятностью не удастся применить чисто, поскольку в нём уже присутствуют некоторые изменения).

Вы можете это упростить, применив git pull --rebase вместо обычного git pull. Или сделать это вручную с помощью git fetch, а затем git rebase teamone/master.

Если вы используете git pull и хотите использовать --rebase по умолчанию, вы можете установить соответствующее значение конфигурации pull.rebase с помощью команды git config --global pull.rebase true.

Если вы рассматриваете перебазирование как способ наведения порядка и работаете с коммитами локально до их отправки или ваши коммиты никогда не будут доступны публично — у вас всё будет хорошо. Однако, если вы перемещаете коммиты, отправленные в публичный репозиторий, и есть вероятность, что работа некоторых людей основывается на этих коммитах, то ваши действия могут вызвать существенные проблемы, а вы — вызвать презрение вашей команды.

Если в какой-то момент вы или ваш коллега находите необходимость в этом, убедитесь, что все знают, как применять команду git pull --rebase для минимизации последствий от подобных действий.

### Перемещение vs. Слияние

Теперь, когда вы увидели перемещение и слияние в действии, вы можете задаться вопросом, что из них лучше. Прежде чем ответить на этот вопрос, давайте вернёмся немного назад и поговорим о том, что означает история.

Одна из точек зрения заключается в том, что история коммитов в вашем репозитории — это запись того, что на самом деле произошло. Это исторический документ, ценный сам по себе, и его нельзя подделывать. С этой точки зрения изменение истории коммитов практически кощунственно; вы лжёте о том, что на самом деле произошло. Но что, если произошла путаница в коммитах слияния? Если это случается, репозиторий должен сохранить это для потомков.

Противоположная точка зрения заключается в том, что история коммитов — это история того, как был сделан ваш проект. Вы не публикуете первый черновик книги или инструкции по поддержке вашего программного обеспечения, так как это нуждается в тщательном редактировании. Сторонники этого лагеря считают использование инструментов rebase и filter-branch способом рассказать историю проекта наилучшим образом для будущих читателей.

Теперь к вопросу о том, что лучше — слияние или перебазирование: надеюсь, вы видите, что это не так просто. Git — мощный инструмент, позволяющий вам делать многое с вашей историей, однако каждая команда и каждый проект индивидуален. Теперь, когда вы знаете, как работают оба эти приёма, выбор — какой из них будет лучше в вашей ситуации — зависит от вас.

При этом, вы можете взять лучшее от обоих миров: использовать перебазирование для наведения порядка в истории ваших локальных изменений, но никогда не применять его для уже отправленных куда-нибудь изменений.