Адрес учебника <https://git-scm.com/book/ru/v2> (там еще куча всего, т.к. здесь будут приведены только Главы 1-3 ( из 11 ☺ ) )

1.3 Введение - Основы Git

**Основы Git**

Что же такое Git, если говорить коротко? Очень важно понять эту часть материала, потому что если вы поймёте, что такое Git и основы того, как он работает, тогда, возможно, вам будет гораздо проще его использовать. Пока вы изучаете Git, попробуйте забыть всё, что вы знаете о других СКВ, таких как Subversion и Perforce. Это позволит вам избежать определённых проблем при использовании инструмента. Git хранит и использует информацию совсем иначе по сравнению с другими системами, даже несмотря на то, что интерфейс пользователя достаточно похож, и понимание этих различий поможет вам избежать путаницы во время использования.

**Снимки, а не различия**

Основное отличие Git от любой другой СКВ (включая Subversion и её собратьев) — это подход к работе со своими данными. Концептуально, большинство других систем хранят информацию в виде списка изменений в файлах. Эти системы (CVS, Subversion, Perforce, Bazaar и т.д.) представляют хранимую информацию в виде набора файлов и изменений, сделанных в каждом файле, по времени (обычно это называют контролем версий, **основанным на различиях**).



Рисунок 4. Хранение данных как набора изменений относительно первоначальной версии каждого из файлов.

Git не хранит и не обрабатывает данные таким способом. Вместо этого, подход Git к хранению данных больше похож на набор снимков миниатюрной файловой системы. Каждый раз, когда вы делаете коммит, то есть сохраняете состояние своего проекта в Git, система запоминает, как выглядит каждый файл в этот момент, и сохраняет ссылку на этот снимок. Для увеличения эффективности, если файлы не были изменены, Git не запоминает эти файлы вновь, а только создаёт ссылку на предыдущую версию идентичного файла, который уже сохранён. Git представляет свои данные как, скажем, **поток снимков**.



Рисунок 5. Хранение данных как снимков проекта во времени.

Это очень важное отличие между Git и почти любой другой СКВ. Git переосмысливает практически все аспекты контроля версий, которые были скопированы из предыдущего поколения большинством других систем. Это делает Git больше похожим на миниатюрную файловую систему с удивительно мощными утилитами, надстроенными над ней, нежели просто на СКВ. Когда мы будем рассматривать управление ветками в [Ветвление в Git](https://git-scm.com/book/ru/v2/ch00/ch03-git-branching), мы увидим, какие преимущества вносит такой подход к работе с данными в Git.

**Почти все операции выполняются локально**

Для работы большинства операций в Git достаточно локальных файлов и ресурсов — в основном, системе не нужна никакая информация с других компьютеров в вашей сети. Если вы привыкли к ЦСКВ, где большинство операций страдают от задержек из-за работы с сетью, то этот аспект Git заставит вас думать, что боги скорости наделили Git несказанной мощью. Так как вся история проекта хранится прямо на вашем локальном диске, большинство операций кажутся чуть ли не мгновенными.

Для примера, чтобы посмотреть историю проекта, Git не нужно соединяться с сервером для её получения и отображения — система просто считывает данные напрямую из локальной базы данных. Это означает, что вы увидите историю проекта практически моментально. Если вам необходимо посмотреть изменения, сделанные между текущей версией файла и версией, созданной месяц назад, Git может найти файл месячной давности и локально вычислить изменения, вместо того, чтобы запрашивать удалённый сервер выполнить эту операцию, либо вместо получения старой версии файла с сервера и выполнения операции локально.

Это также означает, что есть лишь небольшое количество действий, которые вы не сможете выполнить, если вы находитесь оффлайн или не имеете доступа к VPN в данный момент. Если вы в самолёте или в поезде и хотите немного поработать, вы сможете создавать коммиты без каких-либо проблем (в вашу **локальную** копию, помните?): когда будет возможность подключиться к сети, все изменения можно будет синхронизировать. Если вы ушли домой и не можете подключиться через VPN, вы всё равно сможете работать. Добиться такого же поведения во многих других системах либо очень сложно, либо вовсе невозможно. В Perforce, для примера, если вы не подключены к серверу, вам не удастся сделать многого; в Subversion и CVS вы можете редактировать файлы, но вы не сможете сохранить изменения в базу данных (потому что вы не подключены к БД). Всё это может показаться не таким уж и значимым, но вы удивитесь, какое большое значение это может иметь.

**Целостность Git**

В Git для всего вычисляется хеш-сумма, и только потом происходит сохранение. В дальнейшем обращение к сохранённым объектам происходит по этой хеш-сумме. Это значит, что невозможно изменить содержимое файла или директории так, чтобы Git не узнал об этом. Данная функциональность встроена в Git на низком уровне и является неотъемлемой частью его философии. Вы не потеряете информацию во время её передачи и не получите повреждённый файл без ведома Git.

Механизм, которым пользуется Git при вычислении хеш-сумм, называется SHA-1 хеш. Это строка длинной в 40 шестнадцатеричных символов (0-9 и a-f), она вычисляется на основе содержимого файла или структуры каталога. SHA-1 хеш выглядит примерно так:

24b9da6552252987aa493b52f8696cd6d3b00373

Вы будете постоянно встречать хеши в Git, потому что он использует их повсеместно. На самом деле, Git сохраняет все объекты в свою базу данных не по имени, а по хеш-сумме содержимого объекта.

**Git обычно только добавляет данные**

Когда вы производите какие-либо действия в Git, практически все из них только **добавляют** новые данные в базу Git. Очень сложно заставить систему удалить данные либо сделать что-то, что нельзя впоследствии отменить. Как и в любой другой СКВ, вы можете потерять или испортить свои изменения, пока они не зафиксированы, но после того, как вы зафиксируете снимок в Git, будет очень сложно что-либо потерять, особенно, если вы регулярно синхронизируете свою базу с другим репозиторием.

Всё это превращает использование Git в одно удовольствие, потому что мы знаем, что можем экспериментировать, не боясь серьёзных проблем. Для более глубокого понимания того, как Git хранит свои данные и как вы можете восстановить данные, которые кажутся утерянными, см. [Операции отмены](https://git-scm.com/book/ru/v2/ch00/r_undoing).

**Три состояния**

Теперь слушайте внимательно. Это самая важная вещь, которую нужно запомнить о Git, если вы хотите, чтобы остаток процесса обучения прошёл гладко. У Git есть три основных состояния, в которых могут находиться ваши файлы: **изменённое** (modified), **подготовленное** (staged), **зафиксированное** (committed).

* Измененные файлы - файлы которые поменялись, но ещё не были зафиксированы.
* Подготовленные файлы — это изменённые файлы, отмеченные для включения в следующий коммит(индексированные).
* Зафиксированный значит, что файл уже сохранён в вашей локальной базе.

Мы подошли к трём основным секциям проекта Git: рабочая директория (working directory)(до индексирования), область подготовленных файлов (staging area)(после индексирования до коммита), Git-директория (Git directory)(после коммита),



Рисунок 6. Рабочая директория, область подготовленных файлов и директория Git.

Базовый подход в работе с Git выглядит так:

1. Вы изменяете файлы в вашей рабочей директории.
2. Вы выборочно добавляете в индекс только те изменения, которые должны попасть в следующий коммит, добавляя тем самым снимки **только** этих изменений в область подготовленных файлов.
3. Когда вы делаете коммит, используются файлы из индекса как есть, и этот снимок сохраняется в вашу Git-директорию.

Если определённая версия файла есть в Git-директории, эта версия считается **зафиксированной**. Если версия файла изменена и добавлена в индекс, значит, она **подготовлена**. И если файл был изменён с момента последнего распаковывания из репозитория, но не был добавлен в индекс, он считается **изменённым**. В главе [Основы Git](https://git-scm.com/book/ru/v2/ch00/ch02-git-basics) вы узнаете больше об этих состояниях и какую пользу вы можете извлечь из них или как полностью пропустить часть с индексом.

1.4 Введение - Командная строка

**Командная строка**

Есть много различных способов использования Git. Помимо оригинального клиента, имеющего интерфейс командной строки, существует множество клиентов с графическим пользовательским интерфейсом, в той или иной степени реализующих функциональность Git. В рамках данной книги мы будем использовать Git в командной строке. С одной стороны, командная строка — это единственное место, где вы можете запустить **все** команды Git, так как большинство клиентов с графическим интерфейсом реализуют для простоты только некоторую часть функциональности Git. Если вы знаете, как выполнить какое-либо действие в командной строке, вы, вероятно, сможете выяснить, как то же самое сделать и в GUI-версии, а вот обратное не всегда верно. Кроме того, в то время, как выбор графического клиента — это дело личного вкуса, инструменты командной строки доступны **всем** пользователям сразу после установки Git.

Поэтому мы предполагаем, что вы знаете, как открыть терминал в Mac или командную строку, или Powershell в Windows. Если вам не понятно, о чем мы здесь говорим, то вам, возможно, придётся ненадолго прерваться и изучить эти вопросы, чтобы вы могли понимать примеры и пояснения из этой книги.

1.5 Введение - Установка Git

**Установка Git**

Прежде чем использовать Git, вы должны установить его на своём компьютере. Даже если он уже установлен, наверное, это хороший повод, чтобы обновиться до последней версии. Вы можете установить Git из собранного пакета или другого установщика, либо скачать исходный код и скомпилировать его самостоятельно.

|  |  |
| --- | --- |
| **Note** | В этой книге используется Git версии **2.8.0**. Хотя большинство команд, рассматриваемых в книге, должны корректно работать и в более ранних версиях Git, некоторые из них могут действовать несколько по-иному при использовании старых версий. Поскольку Git довольно хорош в вопросе сохранения обратной совместимости, примеры книги должны корректно работать в любой версии старше 2.0. |

**Установка в Linux**

Если вы хотите установить Git под Linux как бинарный пакет, это можно сделать, используя обычный менеджер пакетов вашего дистрибутива. Если у вас Fedora (или другой похожий дистрибутив, такой как RHEL или CentOS), можно воспользоваться dnf:

$ sudo dnf install git-all

Если же у вас дистрибутив, основанный на Debian, например, Ubuntu, попробуйте apt:

$ sudo apt install git

Чтобы воспользоваться дополнительными возможностями, посмотрите инструкцию по установке для нескольких различных разновидностей Unix на сайте Git <http://git-scm.com/download/linux>.

**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.

$ git --version

Если Git не установлен, вам будет предложено его установить.

Если Вы хотите получить более актуальную версию, то можете воспользоваться бинарным установщиком. Установщик Git для OS X доступен для скачивания с сайта Git <http://git-scm.com/download/mac>.



Рисунок 7. OS X инсталлятор Git.

Вы также можете установить Git при установке GitHub для Mac. Их графический интерфейс Git также имеет возможность установить и утилиты командной строки. Скачать клиент GitHub для Mac вы можете с сайта [https://desktop.github.com](https://desktop.github.com/).

**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 <http://git-scm.com/download/win>, и загрузка запустится автоматически. Обратите внимание, что это отдельный проект, называемый Git для Windows; для получения дополнительной информации о нём перейдите на [https://gitforwindows.org](https://gitforwindows.org/).

Для автоматической установки вы можете использовать [пакет Git Chocolatey](https://chocolatey.org/packages/git). Обратите внимание, что пакет Chocolatey поддерживается сообществом.

Другой простой способ установки Git — установить GitHub для Windows. Его установщик включает в себя утилиты командной строки и GUI Git. Он также корректно работает с Powershell, обеспечивает надёжное сохранение учётных данных и правильные настройки CRLF. Вы познакомитесь с этими вещами подробнее несколько позже, здесь же отметим, что они будут вам необходимы. Вы можете загрузить GitHub для Windows с [вебсайта GitHub Desktop](http://windows.github.com/).

**Установка из исходников**

Многие предпочитают устанавливать Git из исходников, поскольку такой способ позволяет получить самую свежую версию. Обновление бинарных инсталляторов, как правило, немного отстаёт, хотя в последнее время разница не столь существенна.

Если вы действительно хотите установить Git из исходников, у вас должны быть установлены следующие библиотеки, от которых он зависит: autotools, curl, zlib, openssl, expat, and libiconv. Например, если в вашей системе используется dnf (Fedora) или apt-get (системы на базе Debian), вы можете использовать одну из следующих команд для установки всех зависимостей, используемых для сборки и установки бинарных файлов Git:

$ sudo dnf install dh-autoreconf curl-devel expat-devel gettext-devel \

openssl-devel perl-devel zlib-devel

$ sudo apt-get install dh-autoreconf libcurl4-gnutls-dev libexpat1-dev \

gettext libz-dev libssl-dev

Для того, чтобы собрать документацию в различных форматах (doc, html, info), понадобится установить дополнительные зависимости:

$ sudo dnf install asciidoc xmlto docbook2X

$ sudo apt-get install asciidoc xmlto docbook2x

|  |  |
| --- | --- |
| **Note** | Пользоватли RHEL и производных от неё (таких как CentOS или Scientific Linux) должны [подключить репозиторий EPEL](https://fedoraproject.org/wiki/EPEL#How_can_I_use_these_extra_packages.3F) для корректной установки пакета docbook2X |

Если вы используете систему на базе Debian (Debian/Ubuntu/Ubuntu-производные), вам так же понадобится установить пакет install-info:

$ sudo apt-get install install-info

Если вы используете систему на базе RPM (Fedora/RHEL/RHEL-производные), вам так же понадобится установить пакет getopt, который уже установлен в системах на базе Debian:

$ sudo dnf install getopt

К тому же из-за различий имён бинарных файлов вам понадобится сделать следующее:

$ sudo ln -s /usr/bin/db2x\_docbook2texi /usr/bin/docbook2x-texi

Когда все необходимые зависимости установлены, вы можете пойти дальше и скачать самый свежий архив с исходниками из следующих мест: с сайта Kernel.org <https://www.kernel.org/pub/software/scm/git>, или зеркала на сайте GitHub <https://github.com/git/git/releases>. Конечно, немного проще скачать последнюю версию с сайта GitHub, но на странице kernel.org релизы имеют подписи, если вы хотите проверить, что скачиваете.

Затем скомпилируйте и установите:

$ tar -zxf git-2.8.0.tar.gz

$ cd git-2.8.0

$ make configure

$ ./configure --prefix=/usr

$ make all doc info

$ sudo make install install-doc install-html install-info

После этого вы можете получать обновления Git по средством самого Git:

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

1.6 Введение - Первоначальная настройка Git

**Первоначальная настройка Git**

Теперь, когда Git установлен в вашей системе, самое время настроить среду для работы с Git под себя. Это нужно сделать только один раз — при обновлении версии Git настройки сохранятся. Но, при необходимости, вы можете поменять их в любой момент, выполнив те же команды снова.

В состав Git входит утилита git config, которая позволяет просматривать и настраивать параметры, контролирующие все аспекты работы Git, а также его внешний вид. Эти параметры могут быть сохранены в трёх местах:

1. Файл /etc/gitconfig содержит значения, общие для всех пользователей системы и для всех их репозиториев. Если при запуске git config указать параметр --system, то параметры будут читаться и сохраняться именно в этот файл.
2. Файл ~/.gitconfig или ~/.config/git/config хранит настройки конкретного пользователя. Этот файл используется при указании параметра --global.
3. Файл config в каталоге Git (т.е. .git/config) репозитория, который вы используете в данный момент, хранит настройки конкретного репозитория.

Настройки на каждом следующем уровне подменяют настройки из предыдущих уровней, то есть значения в .git/config перекрывают соответствующие значения в /etc/gitconfig.

В системах семейства Windows Git ищет файл .gitconfig в каталоге $HOME (C:\Users\$USER для большинства пользователей). Кроме того, Git ищет файл /etc/gitconfig, но уже относительно корневого каталога MSys, который находится там, куда вы решили установить Git при запуске инсталлятора.

Если вы используете Git для Windows версии 2.х или новее, то так же обрабатывается файл конфигурации уровня системы, который имеет путь C:\Documents and Settings\All Users\Application Data\Git\config в Windows XP или C:\ProgramData\Git\config в Windows Vista и новее. Этот файл может быть изменён только командой git config -f <file>, запущенной с правами администратора.

Чтобы посмотреть все установленные настройки и узнать где именно они заданы, используйте команду:

$ git config --list --show-origin

**Имя пользователя**

Первое, что вам следует сделать после установки Git — указать ваше имя и адрес электронной почты. Это важно, потому что каждый коммит в Git содержит эту информацию, и она включена в коммиты, передаваемые вами, и не может быть далее изменена:

$ git config --global user.name "John Doe"

$ git config --global user.email johndoe@example.com

Опять же, если указана опция --global, то эти настройки достаточно сделать только один раз, поскольку в этом случае Git будет использовать эти данные для всего, что вы делаете в этой системе. Если для каких-то отдельных проектов вы хотите указать другое имя или электронную почту, можно выполнить эту же команду без параметра --global в каталоге с нужным проектом.

Многие GUI-инструменты предлагают сделать это при первом запуске.

**Выбор редактора**

Теперь, когда вы указали своё имя, самое время выбрать текстовый редактор, который будет использоваться, если будет нужно набрать сообщение в Git. По умолчанию Git использует стандартный редактор вашей системы, которым обычно является Vim. Если вы хотите использовать другой текстовый редактор, например, Emacs, можно проделать следующее:

$ git config --global core.editor emacs

В системе Windows следует указывать полный путь к исполняемому файлу при установке другого текстового редактора по умолчанию. Пути могут отличаться в зависимости от того, как работает инсталлятор.

В случае с Notepad++, популярным редактором, скорее всего вы захотите установить 32-битную версию, так как 64-битная версия ещё не поддерживает все плагины. Если у вас 32-битная Windows или 64-битный редактор с 64-битной системой, то выполните следующее:

$ git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin"

|  |  |
| --- | --- |
| **Note** | Vim, Emacs и Notepad++ — популярные текстовые редакторы, которые часто используются разработчиками как в Unix-подобных системах, таких как Linux и Mac, так и в Windows. Если вы используете другой редактор или его 32-битную версию, то обратитесь к разделу [Команды git config core.editor](https://git-scm.com/book/ru/v2/ch00/r_core_editor) за дополнительными инструкциями как использовать его совместно с Git. |
| **Warning** | В случае, если вы не установили свой редактор и не знакомы с Vim или Emacs, вы можете попасть в затруднительное положение, когда какой-либо из них будет запущен. Например, в Windows может произойти преждевременное прерывание команды Git при попытке вызова редактора. |

**Проверка настроек**

Если вы хотите проверить используемую конфигурацию, можете использовать команду git config --list, чтобы показать все настройки, которые Git найдёт:

$ git config --list

user.name=John Doe

user.email=johndoe@example.com

color.status=auto

color.branch=auto

color.interactive=auto

color.diff=auto

...

Некоторые ключи (названия) настроек могут отображаться несколько раз, потому что Git читает настройки из разных файлов (например, из /etc/gitconfig и ~/.gitconfig). В таком случае Git использует последнее значение для каждого ключа.

Также вы можете проверить значение конкретного ключа, выполнив git config <key>:

$ git config user.name

John Doe

|  |  |
| --- | --- |
| **Note** | Так как Git читает значение настроек из нескольких файлов, возможна ситуация когда Git использует не то значение что вы ожидали. В таком случае вы можете спросить Git об **origin** этого значения. Git выведет имя файла, из которого значение для настройки было взято последним:  $ git config --show-origin rerere.autoUpdate  file:/home/johndoe/.gitconfig false |

1.7 Введение - Как получить помощь?

**Как получить помощь?**

Если вам нужна помощь при использовании Git, есть три способа.

$ git help <команда>

$ git <команда> --help

$ man git-<команда>

Например, можно открыть руководство по команде git config

$ git help config

Этим командам нравится даже без подключения к сети. Если вам нужна только эта книга, вам нужна личная помощь, и вы можете обратиться за помощью к IRC- серверу #gitи #githubFreenode, который доступен по адресу [https://freenode.net](https://freenode.net/) . Обычно там сотни людей, отлично знающих Git, которые могут помочь.

Так же, если вам нужно посмотреть только список опций и вы не хотите читать полную документацию по команде , Можете использовать вы опцию -hдля вывода краткой Инструкции по использованию:

$ git add -h

usage: git add [<options>] [--] <pathspec>...

-n, --dry-run dry run

-v, --verbose be verbose

-i, --interactive interactive picking

-p, --patch select hunks interactively

-e, --edit edit current diff and apply

-f, --force allow adding otherwise ignored files

-u, --update update tracked files

--renormalize renormalize EOL of tracked files (implies -u)

-N, --intent-to-add record only the fact that the path will be added later

-A, --all add changes from all tracked and untracked files

--ignore-removal ignore paths removed in the working tree (same as --no-all)

--refresh don't add, only refresh the index

--ignore-errors just skip files which cannot be added because of errors

--ignore-missing check if - even missing - files are ignored in dry run

--chmod (+|-)x override the executable bit of the listed files

1.8 Введение - Заключение

**Заключение**

Git и чем он отличается от централизованных систем контроля версий, которые вы, возможно, использовали. Git в вашей ОС, настроенную и персонализированную. Самое время изучить основы Git.

2.1 Основы Git - Создание Git-репозитория

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

**Создание Git-репозитория**

Для создания Git-репозитория вы можете использовать два основных подхода. Во-первых, импорт в Git уже существующего проекта или директории. Во-вторых, клонирование существующего репозитория с другого сервера.

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

**Создание репозитория в существующей директории**

Если у вас уже есть проект в директории, которая не находится под версионным контролем Git, то для начала нужно перейти в эту директорию. 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 текущей директории новую поддиректорию с именем .git, содержащую 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 созданной вами директории .git приведено в главе [Git изнутри](https://git-scm.com/book/ru/v2/ch00/ch10-git-internals))

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 главе [Установка Git на сервер](https://git-scm.com/book/ru/v2/ch00/r_git_on_the_server)).

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 директорию libgit2, инициализирует в ней поддиректорию .git, скачивает все данные для этого репозитория и извлекает рабочую копию последней версии. Если вы зайдёте в новую директорию libgit2, то увидите в ней файлы проекта, готовые для работы или использования. Для того, чтобы клонировать репозиторий в директорию 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 главе [Установка Git на сервер](https://git-scm.com/book/ru/v2/ch00/r_git_on_the_server) мы познакомимся со всеми доступными вариантами конфигурации сервера для обеспечения доступа к вашему Git репозиторию, а также рассмотрим их достоинства и недостатки.

2.2 Основы Git - Запись изменений в репозиторий

**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.



Рисунок 8. Жизненный цикл состояний файлов.

**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 directory 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. В [Ветвление в Git](https://git-scm.com/book/ru/v2/ch00/ch03-git-branching) будут рассмотрены ветки и ссылки более детально.

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 небыло в предыдущем снимке состояния (коммите); Git не станет добавлять его в ваши коммиты, пока вы его явно об этом не попросите. Это предохранит вас от случайного добавления в репозиторий сгенерированных бинарных файлов или каких-либо других, которые вы и не думали добавлять. Мы хотели добавить README, так давайте сделаем это.

**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

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. Это многофункциональная команда, она используется для добавления под версионный контроль новых файлов, для индексации изменений, а также для других целей, например для указания файлов с исправленным конфликтом слияния. Вам может быть понятнее, если вы будете думать об этом как “добавить этот контент в следующий коммит”, а не как “добавить этот файл в проект”. Выполним git add, чтобы проиндексировать CONTRIBUTING.md, а затем снова выполним git status:

$ 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

Что за чёрт? 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

**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

Новые неотслеживаемые файлы помечены ?? слева от них, файлы добавленные в отслеживаемые помечены A, отредактированные файлы помечены M и так далее. В выводе содержится два столбца — в левом указывается статус файла, а в правом модифицирован ли он после этого. К примеру в нашем выводе, файл README модифицирован в рабочей директории и не проиндексирован, файл lib/simplegit.rb модифицирован и проиндексирован. Файл Rakefile модифицирован, проиндексирован и ещё раз модифицирован, таким образом на данный момент у него есть изменения которые попадут в коммит и те которые не попадут.

**Игнорирование файлов**

Зачастую, у вас имеется группа файлов, которые вы не только не хотите автоматически добавлять в репозиторий, но и видеть в списках неотслеживаемых. К таким файлам обычно относятся автоматически генерируемые файлы (различные логи, результаты сборки программ и т.п.). В таком случае, вы можете создать файл .gitignore. с перечислением шаблонов соответствующих таким файлам. Вот пример файла .gitignore:

$ cat .gitignore

\*.[oa]

\*~

Первая строка предписывает Git игнорировать любые файлы заканчивающиеся на “.o” или “.a” — объектные и архивные файлы, которые могут появиться во время сборки кода. Вторая строка предписывает игнорировать все файлы заканчивающиеся на тильду (~), которая используется во многих текстовых редакторах, например Emacs, для обозначения временных файлов. Вы можете также включить каталоги log, tmp или pid; автоматически создаваемую документацию; и т.д. и т.п. Хорошая практика заключается в настройке файла .gitignore до того, как начать серьёзно работать, это защитит вас от случайного добавления в репозиторий файлов, которых вы там видеть не хотите.

К шаблонам в файле .gitignore применяются следующие правила:

* Пустые строки, а также строки, начинающиеся с #, игнорируются.
* Стандартные шаблоны являются глобальными и применяются рекурсивно для всего дерева каталогов.
* Чтобы избежать рекурсии используйте символ слэш (/) в начале шаблона.
* Чтобы исключить каталог добавьте слэш (/) в конец шаблона.
* Можно инвертировать шаблон, использовав восклицательный знак (!) в качестве первого символа.

Glob-шаблоны представляют собой упрощённые регулярные выражения, используемые командными интерпретаторами. Символ (\*) соответствует 0 или более символам; последовательность [abc] — любому символу из указанных в скобках (в данном примере a, b или c); знак вопроса (?) соответствует одному символу; и квадратные скобки, в которые заключены символы, разделённые дефисом ([0-9]), соответствуют любому символу из интервала (в данном случае от 0 до 9). Вы также можете использовать две звёздочки, чтобы указать на вложенные директории: a/\*\*/z соответствует a/z, a/b/z, a/b/c/z, и так далее.

Вот ещё один пример файла .gitignore:

# Исключить все файлы с расширение .a

\*.a

# Но отслеживать файл lib.a даже если он подпадает под исключение выше

!lib.a

# Исключить файл TODO в корневой директории, но не файл в subdir/TODO

/TODO

# Игнорировать все файлы в директории build/

build/

# Игнорировать файл doc/notes.txt, но не файл doc/server/arch.txt

doc/\*.txt

# Игнорировать все .txt файлы в директории doc/

doc/\*\*/\*.txt

|  |  |
| --- | --- |
| **Tip** | GitHub поддерживает довольно полный список примеров .gitignore файлов для множества проектов и языков <https://github.com/github/gitignore> это может стать отправной точкой для .gitignore в вашем проекте. |
| **Note** | В простейшем случае репозиторий будет иметь один файл .gitignore в корневой директории, правила из которого будут рекурсивно применяться ко всем поддиректориям. Так же возможно использовать .gitignore файлы в поддиректориях. Правила из этих файлов будут применяться только к директории, в которой находятся. Например, репозиторий исходного кода ядра Linux содержит 206 файлов .gitignore.  Детальное рассмотрение использования нескольких .gitignore файлов выходит за пределы этой книги; детали доступны в справке man gitignore. |

**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 посмотра 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

|  |  |
| --- | --- |
| **Note** | **Git Diff во внешних инструментах**  Мы будем продолжать использовать команду git diff различными способами на протяжении всей книги. Существует еще один способ просматривать эти изменения, если вы предпочитаете графический просмотр или внешнюю программу просмотра различий, вместо консоли. Выполнив команду git difftool вместо git diff, вы сможете просмотреть изменения в файле с помощью таких программ как emerge, vimdiff и других (включая комерческие продукты). Выполните git difftool --tool-help чтобы увидеть какие из них уже установлены в вашей системе. |

**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:

$ git commit

Эта команда откроет выбранный вами текстовый редактор.

|  |  |
| --- | --- |
| **Note** | Редактор устанавливается переменной окружения EDITOR — обычно это vim или emacs, хотя вы можете установить любой другой с помощью команды git config --global core.editor, как было показано в главе [Введение](https://git-scm.com/book/ru/v2/ch00/ch01-introduction)). |

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

# 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.

|  |  |
| --- | --- |
| **Note** | Для ещё более подробного напоминания, что же именно вы поменяли, можете передать аргумент -v в команду git commit. Это приведёт к тому, что в комментарий будет также помещена дельта/diff изменений, таким образом вы сможете точно увидеть все изменения которые вы совершили. |

Когда вы выходите из редактора, Git создаёт для вас коммит с этим сообщением, удаляя комментарии и вывод команды diff.

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 "Story 182: Fix benchmarks for speed"

[master 463dc4f] Story 182: Fix benchmarks for speed

2 files changed, 2 insertions(+)

create mode 100644 README

Итак, вы создали свой первый коммит! Вы можете видеть, что коммит вывел вам немного информации о себе: на какую ветку вы выполнили коммит (master), какая контрольная сумма SHA-1 у этого коммита (463dc4f), сколько файлов было изменено, а также статистику по добавленным/удалённым строкам в этом коммите.

Запомните, что коммит сохраняет снимок состояния вашего индекса. Всё, что вы не проиндексировали, так и висит в рабочем каталоге как изменённое; вы можете сделать ещё один коммит, чтобы добавить эти изменения в репозиторий. Каждый раз, когда вы делаете коммит, вы сохраняете снимок состояния вашего проекта, который позже вы можете восстановить или с которым можно сравнить текущее состояние.

**Игнорирование индексации**

Несмотря на то, что индекс может быть удивительно полезным для создания коммитов именно такими, как вам и хотелось, он временами несколько сложнее, чем вам нужно в процессе работы. Если у вас есть желание пропустить этап индексирования, Git предоставляет простой способ. 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 'added new benchmarks'

[master 83e38c7] added new benchmarks

1 file changed, 5 insertions(+), 0 deletions(-)

Обратите внимание, что в данном случае перед коммитом вам не нужно выполнять git add для файла CONTRIBUTING.md, потому что флаг -a включает все файлы. Это удобно, но будте осторожны: флаг -a может включить в коммит нежелательные изменения.

**Удаление файлов**

Для того чтобы удалить файл из Git, вам необходимо удалить его из отслеживаемых файлов (точнее, удалить его из вашего индекса) а затем выполнить коммит. Это позволяет сделать команда git rm, которая также удаляет файл из вашего рабочего каталога, так что в следующий раз вы не увидите его как “неотслеживаемый”.

Если вы просто удалите файл из своего рабочего каталога, он будет показан в секции “Changes not staged for commit” (измененные, но не проиндексированные) вывода команды git status:

$ rm PROJECTS.md

$ git status

On branch master

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

Changes not staged for commit:

(use "git add/rm <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

deleted: PROJECTS.md

no changes added to commit (use "git add" and/or "git commit -a")

Затем, если вы выполните команду git rm, удаление файла попадёт в индекс:

$ git rm PROJECTS.md

rm 'PROJECTS.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)

deleted: PROJECTS.md

После следующего коммита файл исчезнет и больше не будет отслеживаться. Если вы изменили файл и уже проиндексировали его, вы должны использовать принудительное удаление с помощью параметра -f. Это сделано для повышения безопасности, чтобы предотвратить ошибочное удаление данных, которые ещё не были записаны в снимок состояния и которые нельзя восстановить из Git.

Другая полезная штука, которую вы можете захотеть сделать — это удалить файл из индекса, оставив его при этом в рабочем каталоге. Другими словами, вы можете захотеть оставить файл на жёстком диске, но перестать отслеживать изменения в нём. Это особенно полезно, если вы забыли добавить что-то в файл .gitignore и по ошибке проиндексировали, например, большой файл с логами, или кучу промежуточных файлов компиляции. Чтобы сделать это, используйте опцию --cached:

$ git rm --cached README

В команду git rm можно передавать файлы, каталоги или шаблоны. Это означает, что вы можете сделать что-то вроде:

$ git rm log/\\*.log

Обратите внимание на обратный слэш (\) перед \*. Он необходим из-за того, что Git использует свой собственный обработчик имён файлов вдобавок к обработчику вашего командного интерпретатора. Эта команда удаляет все файлы имеющие расширение .log находящиеся в директории log/. Или же вы можете сделать вот так:

$ git rm \\*~

Эта команда удаляет все файлы, имена которых заканчиваются на ~.

**Перемещение файлов**

В отличие от многих других систем версионного контроля, Git не отслеживает перемещение файлов явно. Когда вы переименовываете файл в Git, в нём не сохраняется никаких метаданных, говорящих о том, что файл был переименован. Однако, Git довольно умён в плане обнаружения перемещений постфактум — мы рассмотрим обнаружение перемещения файлов чуть позже.

Таким образом, наличие в Git команды mv выглядит несколько странным. Если вам хочется переименовать файл в Git, вы можете сделать что-то вроде:

$ git mv file\_from file\_to

и это отлично сработает. На самом деле, если вы выполните что-то вроде этого и посмотрите на статус, вы увидите, что Git считает, что произошло переименование файла:

$ git mv README.md README

$ 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)

renamed: README.md -> README

Однако, это эквивалентно выполнению следующих команд:

$ mv README.md README

$ git rm README.md

$ git add README

Git неявно определяет, что произошло переименование, поэтому неважно, переименуете вы файл так или используя команду mv. Единственное отличие состоит лишь в том, что mv — одна команда вместо трёх — это функция для удобства. Важнее другое — вы можете использовать любой удобный способ для переименования файла, а затем воспользоваться командами add/rm перед коммитом.

2.3 Основы Git - Просмотр истории коммитов

**Просмотр истории коммитов**

После того, как вы создали несколько коммитов или же склонировали 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 склонировать проект, используйте команду:

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

Если вы запустите команду git log в папке склонированного 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

changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

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

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

removed unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6

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

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

first 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

changed the 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

removed 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

changed the 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

removed 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

first 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 changed the version number

085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test

a11bef06a3f659402fe7563abf99ad00de2209e6 first 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 : changed the version number

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

a11bef0 - Scott Chacon, 6 years ago : first 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.

| Table 1. Полезные опции для git log --pretty=format | |
| --- | --- |
| **Опция** | **Описания вывода** |
| %H | Хеш коммита |
| %h | Сокращенный хеш коммита |
| %T | Хеш дерева |
| %t | Сокращенный хеш дерева |
| %P | Хеш родителей |
| %p | Сокращенный хеш родителей |
| %an | Имя автора |
| %ae | Электронная почта автора |
| %ad | Дата автора (формат даты можно задать опцией --date=option) |
| %ar | Относительная дата автора |
| %cn | Имя коммитера |
| %ce | Электронная почта коммитера |
| %cd | Дата коммитера |
| %cr | Относительная дата коммитера |
| %s | Содержание |

Вам наверное интересно, какая же разница между **автором** и **коммитером**. Автор – это человек, изначально сделавший работу, а коммитер – это человек, который последним применил эту работу. Другими словами, если вы создадите патч для какого-то проекта, а один из основных членов команды этого проекта применит этот патч, вы оба получите статус участника – вы как автор и основной член команды как коммитер. Более детально мы рассмотрим разицу в разделе [Распределенный Git](https://git-scm.com/book/ru/v2/ch00/ch05-distributed-git).

Опции oneline и format являются особенно полезными с опцией --graph команды log. С этой опцией вы сможете увидеть небольшой граф в формате ASCII, который показывает текущую ветку и историю слияний:

$ git log --pretty=format:"%h %s" --graph

\* 2d3acf9 ignore errors from SIGCHLD on trap

\* 5e3ee11 Merge branch 'master' of git://github.com/dustin/grit

|\

| \* 420eac9 Added a method for getting the current branch.

\* | 30e367c timeout code and tests

\* | 5a09431 add timeout protection to grit

\* | e1193f8 support for heads with slashes in them

|/

\* d6016bc require time for xmlschema

\* 11d191e Merge branch 'defunkt' into local

Данный вывод будет нам очень интересен в следующей главе, где мы рассмотрим ветвление и слияние.

Мы рассмотрели только несколько простых опций для форматирования вывода с помощью команды git log — на самом деле их гораздо больше. [Наиболее распространенные опции для команды git log](https://git-scm.com/book/ru/v2/ch00/rlog_options) содержит описание как уже рассмотренных, так и нескольких новых опций, которые могут быть полезными в зависимости от нужного формата вывода.

| Table 2. Наиболее распространенные опции для команды git log | |
| --- | --- |
| **Опция** | **Описание** |
| -p | Показывает патч для каждого коммита. |
| --stat | Показывает статистику измененных файлов для каждого коммита. |
| --shortstat | Отображает только строку с количеством изменений/вставок/удалений для команды --stat. |
| --name-only | Показывает список измененных 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. |

**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.

|  |  |
| --- | --- |
| **Note** | Допускается указывать несколько параметров --author и --grep для поиска, которые позволят найти коммиты, сооветствующие **любому** указанному --author и **любому** указанному --grep шаблону; однако, применение опции --all-match заставит искать коммиты соответствующие **всем** указанным --grep шаблонам. |

Следующим действительно полезным фильтром является опция -S, которая принимает аргумент в виде строки и показывает только те коммиты, в которых изменение в коде повлекло за собой добавление или удаление этой строки. Например, если вы хотите найти последний коммит, который добавил или удалил вызов определенной функции, вы можете запустить команду:

$ git log -S function\_name

Последней полезной опцией, которую принимает команда git log как фильтр, является путь. Если вы укажете директорию или имя файла, вы ограничите вывод только теми коммитами, в которых были изменения этих файлов. Эта опция всегда указывается последней после двойного тире (--), что отделяет указываемый путь от опций.

В таблице [Опции для ограничения вывода команды git log](https://git-scm.com/book/ru/v2/ch00/rlimit_options) вы можете увидеть эти и другие распространенные опции.

| Table 3. Опции для ограничения вывода команды git log | |
| --- | --- |
| **Опция** | **Описание** |
| -(n) | Показывает только последние n коммитов. |
| --since, --after | Показывает только те коммиты, которые были сделаны после указанной даты. |
| --until, --before | Показывает только те коммиты, которые были сделаны до указанной даты. |
| --author | Показывает только те коммиты, в которых запись author совпадает с указанной строкой. |
| --committer | Показывает только те коммиты, в которых запись committer совпадает с указанной строкой. |
| --grep | Показывает только коммиты, сообщение которых содержит указанную строку. |
| -S | Показывает только коммиты, в которых изменение в коде повлекло за собой добавление или удаление указанной строки. |

Например, если вы хотите увидеть, в каких коммитах произошли изменения в тестовых файлах в исходном коде Git в октябре 2008 года, автором которых был Junio Hamano и которые не были комитами слияния, вы можете запустить следующую команду:

$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \

--before="2008-11-01" --no-merges -- t/

5610e3b - Fix testcase failure when extended attributes are in use

acd3b9e - Enhance hold\_lock\_file\_for\_{update,append}() API

f563754 - demonstrate breakage of detached checkout with symbolic link HEAD

d1a43f2 - reset --hard/read-tree --reset -u: remove unmerged new paths

51a94af - Fix "checkout --track -b newbranch" on detached HEAD

b0ad11e - pull: allow "git pull origin $something:$current\_branch" into an unborn branch

Из почти 40,000 коммитов в истории исходного кода Git, эта команда показывает только 6, которые соответствуют этим критериям.

2.4 Основы Git - 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.

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.

|  |  |
| --- | --- |
| **Note** | 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.

|  |  |
| --- | --- |
| **Note** | 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 главе [Раскрытие тайн reset](https://git-scm.com/book/ru/v2/ch00/r_git_reset).

**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.

|  |  |
| --- | --- |
| **Important** | 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 пребережение — мы рассмотрим их в разделе [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.

2.5 Основы Git - Работа с удалёнными репозиториями

**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.

|  |  |
| --- | --- |
| **Note** | **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 поразумевает 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 котрого 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 главе [Установка Git на сервер](https://git-scm.com/book/ru/v2/ch00/r_git_on_the_server).

**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 [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 склонировали (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 у вас есть ветка, настроенная на отслеживание удалённой ветки (обратитесь к главе 3 [Ветвление в Git](https://git-scm.com/book/ru/v2/ch00/ch03-git-branching) за более подробной информацией), 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 склонировали, и автоматически пытается слить (merge) их с кодом, над которым вы в данный момент работаете.

**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. См. главу 3 [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.

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

$ git remote show origin

\* remote origin

URL: https://github.com/my-org/complex-project

Fetch URL: https://github.com/my-org/complex-project

Push URL: https://github.com/my-org/complex-project

HEAD branch: master

Remote branches:

master tracked

dev-branch tracked

markdown-strip tracked

issue-43 new (next fetch will store in remotes/origin)

issue-45 new (next fetch will store in remotes/origin)

refs/remotes/origin/issue-11 stale (use 'git remote prune' to remove)

Local branches configured for 'git pull':

dev-branch merges with remote dev-branch

master merges with remote master

Local refs configured for 'git push':

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

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

master pushes to master (up to date)

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

**Удаление и переименование удалённых репозиториев**

Для переименования удалённого репозитория можно вылолнить 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 удаленный репозиторий — вы сменили сервер или больше не используете определённое зеркало, или кто-то перестал вносить изменения — вы можете использовать git remote rm:

$ git remote rm paul

$ git remote

origin

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

2.6 Основы Git - Работа с метками

**Работа с метками**

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.). Метки в Git называются тегами. В этом разделе вы узнаете, как посмотреть имеющиеся теги, создать новые или удалить существующие, а так же какие их типы существуют в Git.

**Просмотр списка меток**

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

$ git tag

v0.1

v1.3

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

|  |  |
| --- | --- |
| **Note** | **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 меток**

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 тэги, 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 метки**

Создание аннотированной метки 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

changed the version number

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

**Легковесные метки**

Легковесная метка — это ещё один способ пометить коммит. По сути, это контрольная сумма коммита, сохранённая в файл — больше никакой информации не хранится. Для создания легковесной метки 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

changed the version number

**Отложенная растановка меток**

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

$ git log --pretty=oneline

15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'

a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support

0d52aaab4479697da7686c15f77a3d64d9165190 one more thing

6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'

0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function

4682c3261057305bdd616e23b64b0857d832627b added a todo file

166ae0c4d3f420721acbb115cc33848dfcc2121a started write support

9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile

964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo

8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme

Теперь предположим, что вы забыли отметить версию проекта v1.2, которая была там, где находится коммит “updated rakefile”. Вы можете добавить метку 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

updated rakefile

...

**Обмен метками**

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 склонирует (clone) или выполнит git pull из вашего репозитория, то он получит вдобавок к остальному и ваши метки.

|  |  |
| --- | --- |
| **Note** | **git push отправляет оба типа тегов**  Отправка тегов командой git push <remote> --tags не различает аннотированные и легковесные теги; не существует простой опции, которая позволила бы вам указать тип тегов для отправки. |

**Удаление меток**

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 удаление с сервера. Распространены два способа удаление тега из удалённого 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 удаленного тэга пустым значением, что при водит к его удалению.

Второй способ удалить тег из удалённого репозитория более интуитивный:

$ git push origin --delete <tagname>

**Переход на метку**

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

$ git checkout 2.0.0

Note: checking out '2.0.0'.

You are in 'detached HEAD' state. You can look around, make experimental

changes and commit them, and you can discard any commits you make in this

state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may

do so (now or later) by using -b with the checkout command again. Example:

git checkout -b <new-branch>

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

$ git checkout 2.0-beta-0.1

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

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

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

$ git checkout -b version2 v2.0.0

Switched to a new branch 'version2'

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

2.7 Основы Git - Псевдонимы в Git

**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 неполностью. 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"

2.8 Основы Git - Заключение

**Заключение**

Теперь вы умеете выполнять все базовые локальные операции с Git: создавать или клонировать репозиторий, вносить изменения, индексировать и фиксировать эти изменения, а также просматривать историю всех изменений в репозитории. Дальше мы рассмотрим киллер-фичу Git — его модель ветвления.

3.1 Ветвление в Git - О ветвлении в двух словах

Почти каждая система контроля версий (СКВ) в какой-то форме поддерживает ветвление. Используя ветвление, Вы отклоняетесь от основной линии разработки и продолжаете работу независимо от неё, не вмешиваясь в основную линию. Во многих СКВ создание веток — это очень затратный процесс, часто требующий создания новой копии директории, что может занять много времени для большого проекта.

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

**О ветвлении в двух словах**

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

Как вы можете помнить из [Введение](https://git-scm.com/book/ru/v2/ch00/ch01-introduction), Git не хранит данные в виде последовательности изменений, он использует набор снимков (snapshot).

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

Предположим, в вашей рабочей директории есть три файла и вы добавляете их все в индекс и создаёте коммит. Во время индексации вычисляется контрольная сумма каждого файла (SHA-1 как мы узнали из [Введение](https://git-scm.com/book/ru/v2/ch00/ch01-introduction)), 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 of my project'

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.



Рисунок 9. Коммит и его дерево

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



Рисунок 10. Коммит и его родители

Ветка в Git — это простой перемещаемый указатель на один из таких коммитов. По умолчанию, имя основной ветки в Git — master. Как только вы начнёте создавать коммиты, ветка master будет всегда указывать на последний коммит. Каждый раз при создании коммита указатель ветки master будет передвигаться на следующий коммит автоматически.

|  |  |
| --- | --- |
| **Note** | Ветка “master” в Git — это не какая-то особенная ветка. Она точно такая же, как и все остальные ветки. Она существует почти во всех репозиториях только лишь потому, что её создаёт команда git init, а большинство людей не меняют её название. |



Рисунок 11. Ветка и история коммитов

**Создание новой ветки**

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

$ git branch testing

В результате создаётся новый указатель на текущий коммит.



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

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



Рисунок 13. HEAD указывает на ветку

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

$ git log --oneline --decorate

f30ab (HEAD, master, testing) add feature #32 - ability to add new

34ac2 fixed bug #1328 - stack overflow under certain conditions

98ca9 initial commit of my project

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.



Рисунок 14. HEAD указывает на текущую ветку

Какой в этом смысл? Давайте сделаем ещё один коммит:

$ vim test.rb

$ git commit -a -m 'made a change'



Рисунок 15. Указатель на ветку HEAD переместился вперёд после коммита

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

$ git checkout master



Рисунок 16. HEAD перемещается когда вы делаете checkout

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

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

Давайте сделаем еще несколько изменений и создадим очередной коммит:

$ vim test.rb

$ git commit -a -m 'made other changes'

Теперь история вашего проекта разошлась (см [Разветвлённая история](https://git-scm.com/book/ru/v2/ch00/rdivergent_history)). Вы создали ветку и переключились на нее, 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 все это делается простыми командами: branch, checkout и commit.



Рисунок 17. Разветвлённая история

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

$ git log --oneline --decorate --graph --all

\* c2b9e (HEAD, master) made other changes

| \* 87ab2 (testing) made a change

|/

\* f30ab add feature #32 - ability to add new formats to the

\* 34ac2 fixed bug #1328 - stack overflow under certain conditions

\* 98ca9 initial commit of my project

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

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

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

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

3.2 Ветвление в Git - 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 новой статьи, которую вы пишете.
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 ту ветку, где вы пишете статью и продолжить работать.

**Основы ветвления**

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



Рисунок 18. Простая история коммитов

Вы решаете, что теперь вы будете заниматься проблемой #53 из вашей системы отслеживания ошибок. Чтобы создать ветку и сразу переключиться на нее, можно выполнить команду git checkout с параметром -b:

$ git checkout -b iss53

Switched to a new branch "iss53"

Это тоже самое что и:

$ git branch iss53

$ git checkout iss53



Рисунок 19. Создание нового указателя ветки

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

$ vim index.html

$ git commit -a -m 'added a new footer [issue 53]'



Рисунок 20. Ветка iss53 двигается вперед

Тут вы получаете сообщение об обнаружении уязвимости на вашем сайте, которую нужно немедленно устранить. Благодаря Git, не требуется размещать это исправление вместе с тем, что вы сделали в iss53. Вам даже не придется прилагать усилий, чтобы откатить все эти изменения для начала работы над исправлением. Все, что вам нужно — переключиться на ветку master.

Но перед тем как сделать это — имейте в виду, что если ваш рабочий каталог либо область подготовленных файлов содержат изменения, не попавшие в коммит и конфликтующие с веткой, на которую вы хотите переключиться, то Git не позволит вам переключить ветки. Лучше всего переключаться из чистого рабочего состояния проекта. Есть способы обойти это (спрятать (stash) или исправить (amend) коммиты), но об этом мы поговорим позже в главе [Прибережение и очистка](https://git-scm.com/book/ru/v2/ch00/r_git_stashing). 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 проблемой #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 эту 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 для исправления, в которой будем работать, пока не закончим исправление.

$ git checkout -b hotfix

Switched to a new branch 'hotfix'

$ vim index.html

$ git commit -a -m 'fixed the broken email address'

[hotfix 1fb7853] fixed the broken email address

1 file changed, 2 insertions(+)



Рисунок 21. Ветка hotfix основана на ветке master

Вы можете прогнать тесты, чтобы убедиться, что ваше исправление делает именно то, что нужно. 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 прямо, Git упрощает слияние просто перенося указатель ветки вперед, так как нет расхождений в изменениях. Это называется “fast-forward”.

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



Рисунок 22. master перемотан до hotfix

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 проблемой #53:

$ git checkout iss53

Switched to branch "iss53"

$ vim index.html

$ git commit -a -m 'finished the new footer [issue 53]'

[iss53 ad82d7a] finished the new footer [issue 53]

1 file changed, 1 insertion(+)



Рисунок 23. Продолжение работы над iss53

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, или же вы можете отложить слияние этих изменений до завершения работы, и затем влить ветку iss53 в master.

**Основы слияния**

Предположим, вы решили, что работа по проблеме #53 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.



Рисунок 24. Использование трёх снимков при слиянии

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.



Рисунок 25. Коммит слияния

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, которое 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.

|  |  |
| --- | --- |
| **Note** | Мы рассмотрим более продвинутые инструменты для разрешения сложных конфликтов слияния в разделе [Продвинутое слияние](https://git-scm.com/book/ru/v2/ch00/r_advanced_merging). |

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 разрешенный. 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.

3.3 Ветвление в Git - Управление ветками

**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 readmes

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.

|  |  |
| --- | --- |
| **Tip** | Если в качестве аргумента не указан коммит или ветка, то опции --merged и --no-merged покажут что уже слито или не слито с вашей **текущей** веткой соответственно.  Вы всегда можете указать дополнительный аргумент для вывода той же информации, но относительно указанной ветки предварительно не извлекая и не переходя на неё.  $ git checkout testing  $ git branch --no-merged master  topicA  featureB |

3.4 Ветвление в Git - Работа с ветками

**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 ее 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



Рисунок 26. Линейное представление повышения стабильности веток

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



Рисунок 27. Представление диаграммы стабильности веток в виде многоуровневого накопителя

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 СКВ 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:



Рисунок 28. Набор тематических веток

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:



Рисунок 29. История после слияния веток dumbidea и iss91v2

Более подробно возможные варианты рабочих схем для проектов рассматриваются в [Распределенный Git](https://git-scm.com/book/ru/v2/ch00/ch05-distributed-git), поэтому перед выбором схемы обязательно прочитайте эту главу.

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.

3.5 Ветвление в Git - Удалённые ветки

**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 распространенным 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 удаленным 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 склонируете, 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.

|  |  |
| --- | --- |
| **Note** | **“origin” — это не специальное название**  Подобно названию ветки “master”, “origin” не имеет какого-либо специального значения в Git. В то время как “master” — это название по умолчанию для ветки при выполнении git init только потому что часто используется, “origin” — это название по умолчанию для удалённого сервера когда вы запускаете git clone. Если вы выполните git clone -o booyah, то по умолчанию ветка слежения будет иметь вид booyah/master. |



Рисунок 30. Серверный и локальный репозитории после клонирования

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.



Рисунок 31. Локальная и удалённая работа может расходиться

Для синхронизации ваших изменений с удаленным 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.



Рисунок 32. 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, как было описано в [Основы Git](https://git-scm.com/book/ru/v2/ch00/ch02-git-basics). Назовите этот удалённый сервер teamone — это имя будет сокращением вместо полного URL.



Рисунок 33. 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.



Рисунок 34. Ветка слежения teamone/master

**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 [Git изнутри](https://git-scm.com/book/ru/v2/ch00/ch10-git-internals), 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.

|  |  |
| --- | --- |
| **Note** | **Не вводите каждый раз свой пароль**  Если вы используете HTTPS URL для отправки изменений, Git-сервер будет спрашивать имя пользователя и пароль для аутентификации. По умолчанию вам будет предложено ввести эти данные в терминале, чтобы сервер мог определить разрешена ли вам отправка изменений.  Если вы не хотите вводить свои данные каждый раз при отправке изменений, вы можете настроить “credential cache”. Проще всего держать их в памяти несколько минут, это легко настроить с помощью команды git config --global credential.helper cache.  Для получения более подробной информации о различных вариантах кэша учётных данных обратитесь к разделу [Хранилище учётных данных](https://git-scm.com/book/ru/v2/ch00/r_credential_caching). |

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 ее 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.

|  |  |
| --- | --- |
| **Note** | **Сокращение Upstream**  Если у вас настроена отслеживаемая ветка, вы можете ссылаться на нее при помощью сокращений @{upstream} или @{u}. Итак, если вы находитесь на ветке master и она следит за origin/master, при желании вы можете использовать git merge @{u} вместо git merge origin/master. |

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] forgot the brackets

master 1ae2a45 [origin/master] deploying index fix

\* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this should do it

testing 5ea463a trying 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 удаленную 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 директории. 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.

3.6 Ветвление в Git - Перебазирование

**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](https://git-scm.com/book/ru/v2/ch00/r_basic_merging), вы увидите, что разделили свою работу и сделали коммиты в две разные ветки.



Рисунок 35. История коммитов простого разделения

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).



Рисунок 36. Слияние разделённой истории коммитов

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.



Рисунок 37. Перебазирование изменений из C4 поверх C3

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

$ git checkout master

$ git merge experiment



Рисунок 38. Перемотка ветки master

Теперь снимок, на который указывает C4' абсолютно такой же, как тот, на который указывал C5 в [примере с трёхсторонним слиянием](https://git-scm.com/book/ru/v2/ch00/r_rebasing-merging-example). 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](https://git-scm.com/book/ru/v2/ch00/rrbdiag_e). 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](https://git-scm.com/book/ru/v2/ch00/rrbdiag_g)):

$ 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

Эта команда добавит изменения ветки server в ветку master, как это показано на [Перебазирование ветки server на основании ветки master](https://git-scm.com/book/ru/v2/ch00/rrbdiag_h).



Рисунок 42. Перебазирование ветки server на основании 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](https://git-scm.com/book/ru/v2/ch00/rrbdiag_i):

$ git branch -d client

$ git branch -d server



Рисунок 43. Окончательная история коммитов

**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 склонировали 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 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 бы 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 — решать вам.

В основном, стоит взять лучшее от обоих миров — использовать перебазирование для наведения порядка в истории ваших локальных изменений, ещё не отправленных на удалённый сервер, но никогда не применять перебазирование для уже отправленных куда-нибудь изменений.

3.7 Ветвление в Git - Заключение

**Заключение**

Мы рассмотрели базовые функции ветвления и слияния в Git. Вы должны быть способны свободно создавать и переключаться на новую ветку, переключаться между ветками и сливать локальные ветки вместе. Также вы должны уметь выкладывать ветки на общий сервер, работать с другими людьми над общими ветками и перебазировать ваши ветки до того, как они станут доступны другим разработчикам. Далее мы поговорим о том, что вам необходимо для запуска собственного сервера с хостингом для Git-репозитория.

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | **Git: В чем разница между Fetch и Pull?** | 01.07.2015 16:47 | |
|  |
| **Быстрое объяснение**  git pull — это, по сути, команда git fetch, после которой сразу же следует git merge.  git fetch получает изменения с сервера и сохраняет их в каталог refs/remotes/. Это никак не влияет на локальные ветки и текущие изменения. А git merge уже вливает все эти изменения в локальную копию.  **Подробное объяснение**  **Git fetch**  Для получения данных из удалённых проектов, следует выполнить:  $ git fetch [имя удал. сервера]  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. Теперь эти ветки в любой момент могут быть просмотрены или слиты.  Когда вы клонируете репозиторий, команда clone автоматически добавляет этот удалённый репозиторий под именем origin. Таким образом, git fetch origin извлекает все наработки, отправленные (push) на этот сервер после того, как вы склонировали его (или получили изменения с помощью fetch). Важно отметить, что команда 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 этот код с вашим кодом в рабочей директории, когда вы будете готовы, с помощью git merge.  Возможно, более яснее станет, что делает git fetch, если посмотреть на эту картинку:  https://raw.github.com/xintrea/mytetra_syncro/master/base/14357584259soun9chuj/image1548152635tqvctg3eeo.png  **Git pull**  Если у вас есть ветка, настроенная на отслеживание удалённой ветки, то вы можете использовать команду git pull. Она автоматически извлекает и затем сливает данные из удалённой ветки в вашу текущую ветку. 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 (подразумевается, что на удалённом сервере есть ветка master). Выполнение git pull, как правило, извлекает (fetch) данные с сервера, с которого вы изначально склонировали, **и автоматически пытается слить (merge) их с кодом, над которым вы в данный момент работаете**.  То есть, работа git pull состоит из двух стадий. Первая стадия - это то же самое что и git fetch, а вторая стадия - это мердж с кодом в рабочей директории. Вот так:  https://raw.github.com/xintrea/mytetra_syncro/master/base/14357584259soun9chuj/image1548152747dpzmnyxma8.png |