Система контроля версий GIT

Введение

О системе контроля версий

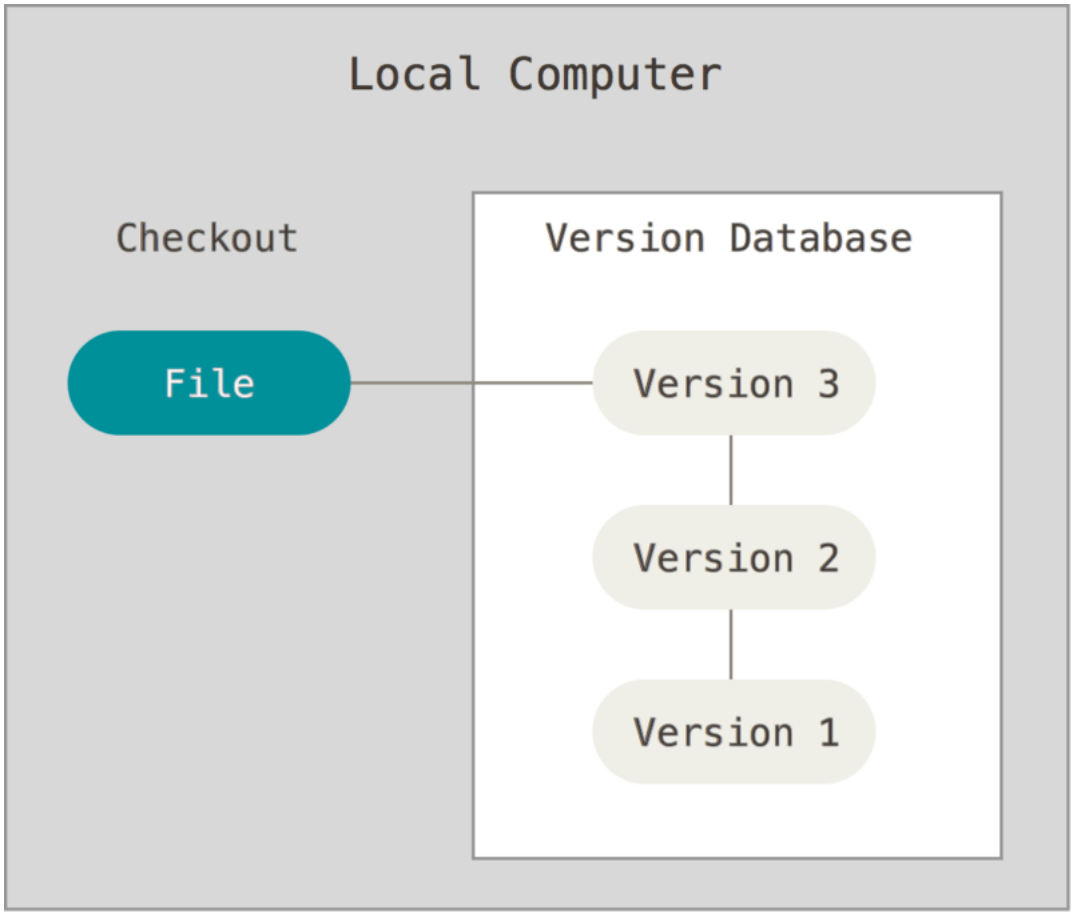
Что такое “система контроля версий” и почему это важно? Система контроля версий - это система, записывающая изменения в файл или набор файлов в течение времени и позволяющая вернуться позже к определённой версии. Для контроля версий файлов в качестве примера будет использоваться исходный код программного обеспечения, хотя на самом деле вы можете использовать контроль версий практически для любых типов файлов.

Если вы графический или web-дизайнер и хотите сохранить каждую версию изображения или макета (скорее всего, захотите), система контроля версий (далее VCS) - как раз то, что нужно. Она позволяет вернуть файлы к состоянию, в котором они были до изменений, вернуть проект к исходному состоянию, увидеть изменения, увидеть, кто последний менял что-то и вызвал проблему, кто поставил задачу и когда и многое другое. Использование VCS также значит в целом, что, если вы сломали что-то или потеряли файлы, вы спокойно можете всё исправить. В дополнение ко всему вы получите всё это без каких-либо дополнительных усилий.

Локальные системы контроля версий

Как уже говорилось ранее - один из примеров локальной СУВ предельно прост: многие предпочитают контролировать версии, просто копируя файлы в другой каталог (как правило добавляя текущую дату к названию каталога). Такой подход очень распространён, потому что прост, но он и чаще даёт сбои. Очень легко забыть, что ты не в том каталоге, и случайно изменить не тот файл, либо скопировать файлы не туда, куда хотел, и затереть нужные файлы. Чтобы решить эту проблему, программисты уже давно разработали локальные VCS с простой базой данных, в которой хранятся все изменения нужных файлов

Источники:
https://git-scm.com
https://ru.hexlet.io



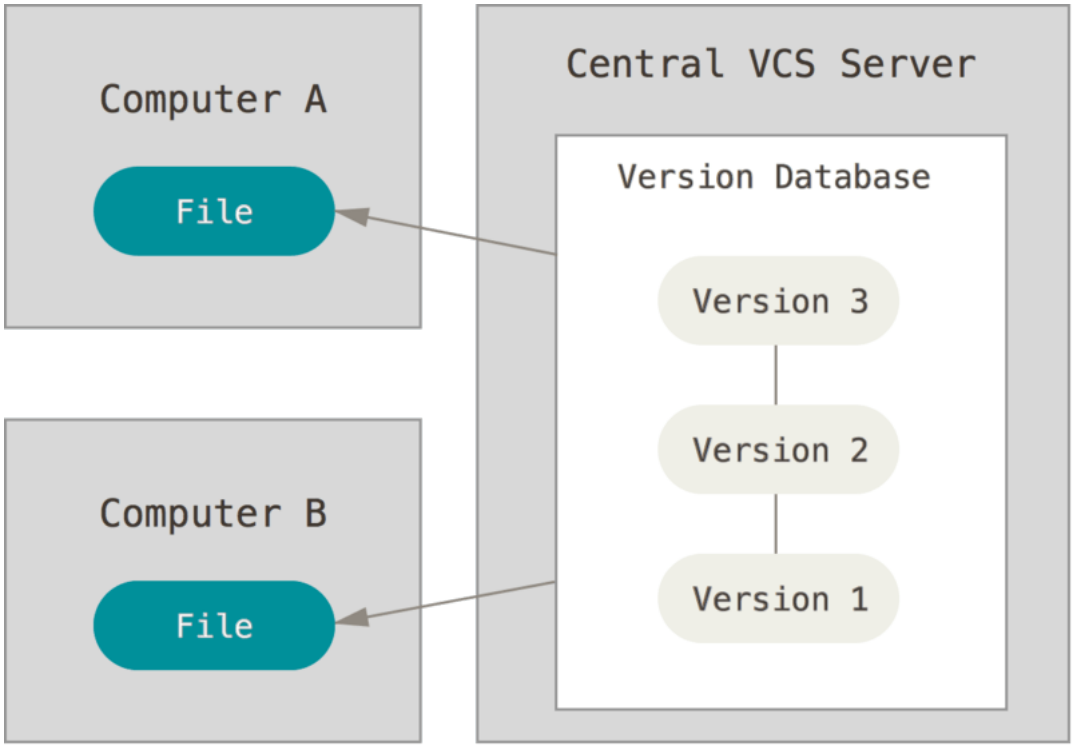
Одной из наиболее популярных VCS такого типа является RCS (Revision Control System, Система контроля ревизий), которая до сих пор устанавливается на многие компьютеры. Даже в современной операционной системе Mac OS X утилита rcs устанавливается вместе с Developer Tools. RCS была разработана в начале 1980-х годов Вальтером Тичи (Walter F. Tichy). Система позволяет хранить версии только одного файла, таким образом управлять несколькими файлами приходится вручную. Для каждого файла находящегося под контролем системы информация о версиях хранится в специальном файле с именем оригинального файла к которому в конце добавлены символы ',v'. Например, для файла file.txt версии будут храниться в файле file.txt,v. Эта утилита основана на работе с наборами патчей между парами версий (патч - файл, описывающий различие между файлами). Это позволяет пересоздать любой файл на любой момент времени, последовательно накладывая патчи. Для хранения версий система использует утилиту diff. Хотя RCS соответствует минимальным требованиям к системе контроля версий она имеет следующие основные недостатки, которые также послужили стимулом для создания следующей рассматриваемой системы:

- Работа только с одним файлом, каждый файл должен контролироваться отдельно;
- Неудобный механизм одновременной работы нескольких пользователей с системой, хранилище просто блокируется пока заблокировавший его пользователь не разблокирует его;
- От бекапов вас никто не освобождает, вы рискуете потерять всё.

Централизованные системы контроля версий

Следующей основной проблемой оказалась необходимость сотрудничать с разработчиками за другими компьютерами. Чтобы решить её, были созданы централизованные системы контроля версий (CVCS). В таких системах, например CVS, Subversion и Perforce, есть центральный сервер, на котором хранятся все файлы под версионным контролем, и ряд клиентов, которые получают копии файлов из него. Много лет это было стандартом для систем контроля версий.

Источники:
https://git-scm.com
https://ru.hexlet.io

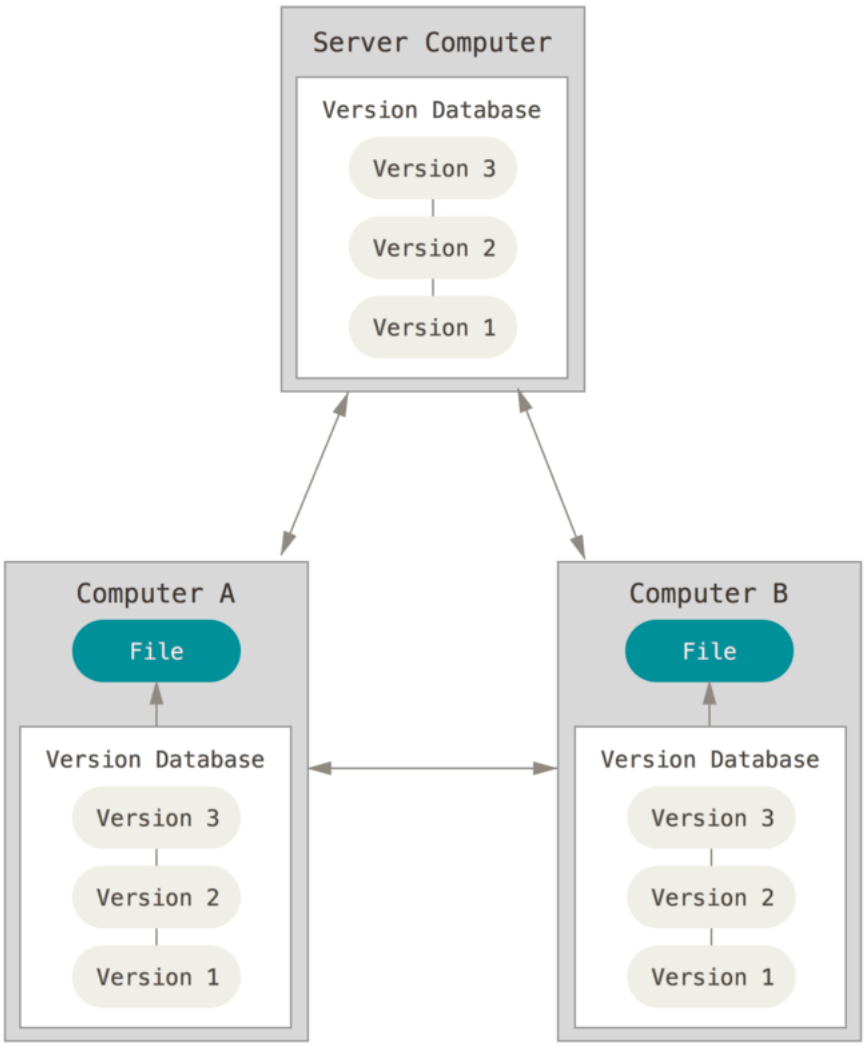


Такой подход имеет множество преимуществ, особенно над локальными VCS. К примеру, все знают, кто и чем занимается в проекте. У администраторов есть чёткий контроль над тем, кто и что может делать, и, конечно, администрировать CVCS намного легче, чем локальные базы на каждом клиенте. Однако при таком подходе есть и несколько серьёзных недостатков. Наиболее очевидный - централизованный сервер является уязвимым местом всей системы. Если сервер выключается на час, то в течение часа разработчики не могут взаимодействовать, и никто не может сохранить новой версии своей работы. Если же повреждается диск с центральной базой данных и нет резервной копии, вы теряете абсолютно всё — всю историю проекта, разве что за исключением нескольких рабочих версий, сохранившихся на рабочих машинах пользователей.

Распределённые системы контроля версий

И в этой ситуации в игру вступают распределённые системы контроля версий (RVCS). В таких системах как Git, Mercurial, Bazaar или Darcs клиенты не просто выгружают последние версии файлов, а полностью копируют весь репозиторий. Поэтому в случае, когда "умирает" сервер, через который шла работа, любой клиентский репозиторий может быть скопирован обратно на сервер, чтобы восстановить базу данных. Каждый раз, когда клиент забирает свежую версию файлов, он создаёт себе полную копию всех данных.

Источники:
https://git-scm.com
https://ru.hexlet.io



Кроме того, в большей части этих систем можно работать с несколькими удалёнными репозиториями, таким образом, можно одновременно работать по-разному с разными группами людей в рамках одного проекта. Так, в одном проекте можно одновременно вести несколько типов рабочих процессов, что невозможно в централизованных системах.

Краткая история Git

Как и многие вещи в жизни, Git начинался с капелькой творческого хаоса и бурных споров.

Ядро Linux - это достаточно большой проект с открытым исходным кодом. Большую часть времени разработки ядра Linux (1991–2002 гг.) изменения передавались между разработчиками в виде патчей и архивов. В 2002 году проект ядра Linux начал использовать проприетарную децентрализованную систему контроля версий BitKeeper.

В 2005 году отношения между сообществом разработчиков ядра Linux и коммерческой компанией, которая разрабатывала BitKeeper, прекратились, и бесплатное использование утилиты стало невозможным. Это сподвигло сообщество разработчиков ядра Linux (а в частности Линуса Торвальдса - создателя Linux ядра) разработать свою собственную утилиту, учитывая уроки, полученные при работе с BitKeeper. Некоторыми целями, которые преследовала новая система, были:

- Скорость
- Простая архитектура
- Хорошая поддержка нелинейной разработки (тысячи параллельных веток)

Источники:
https://git-scm.com
https://ru.hexlet.io

- Полная децентрализация
- Возможность эффективного управления большими проектами, такими как ядро Linux (скорость работы и разумное использование дискового пространства)

С момента своего появления в 2005 году, Git развился в простую в использовании систему, сохранив при этом свои изначальные качества. Он удивительно быстр, эффективен в работе с большими проектами и имеет великолепную систему веток для нелинейной разработки

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 `[path]/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. Так как этот файл является системным, то вам потребуются права суперпользователя для внесения изменений в него.
2. Файл `~/.gitconfig` или `~/.config/git/config` хранит настройки конкретного пользователя. Этот файл используется при указании параметра `--global` и применяется ко **всем** репозиториям, с которыми вы работаете в текущей системе.
3. Файл `config` в каталоге Git (т. е. `.git/config`) репозитория, который вы используете в данный момент, хранит настройки конкретного репозитория. Вы можете заставить Git читать и писать в этот файл с помощью параметра `--local`, но на самом деле это значение по умолчанию. Неудивительно, что вам нужно находиться где-то в репозитории Git, чтобы эта опция работала правильно.

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

В системах семейства Windows Git ищет файл `.gitconfig` в каталоге `$HOME` (`C:\Users\$USER` для большинства пользователей). Кроме того, Git ищет файл `[path]/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.x или новее, то так же обрабатывается файл конфигурации уровня системы, который имеет путь `C:\ProgramData\Git\config` в Windows 7 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:

Источники:
https://git-scm.com
https://ru.hexlet.io

```
$ git config --global user.name "Aleksandr"
$ git config --global user.email test@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' -multiIn
```

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=Aleksandr
user.email=test@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
Aleksandr
```

Источники:
https://git-scm.com
https://ru.hexlet.io

Создание Git-репозитория

Обычно вы получаете репозиторий Git одним из двух способов:

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

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

Создание репозитория в существующем каталоге

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

для Linux:

```
$ cd /home/user/my_project
```

для macOS:

```
$ cd /Users/user/my_project
```

для Windows:

```
$ cd C:/Users/user/my_project
```

а затем выполните команду:

```
$ git init
```

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

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

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'Initial project version'
```

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

Клонирование существующего репозитория

Для получения копии существующего Git-репозитория, например, проекта, в который вы хотите внести свой вклад, необходимо использовать команду `git clone`. Если вы знакомы с другими системами контроля версий, такими как Subversion, то заметите, что команда называется “clone”, а не “checkout”. Это важное различие - вместо того, чтобы просто получить рабочую копию, Git получает копию практически всех данных, которые есть на сервере. При выполнении `git clone` с сервера забирается (pulled) каждая версия каждого файла из истории проекта. Фактически, если серверный диск выйдет из строя, вы можете использовать любой из клонов на любом из клиентов,

Источники:
https://git-scm.com
https://ru.hexlet.io

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

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

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

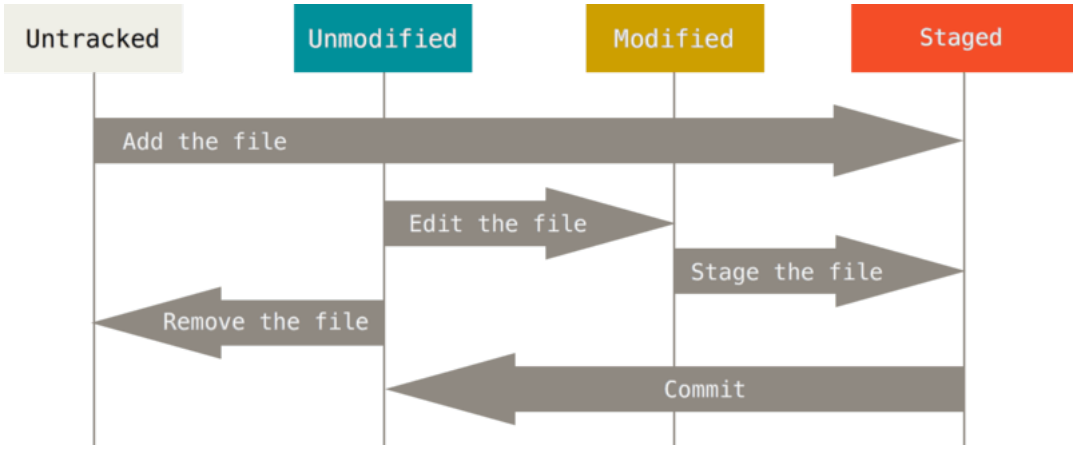
```
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

Эта команда делает всё то же самое, что и предыдущая, только результирующий каталог будет назван mylibgit.
В Git реализовано несколько транспортных протоколов, которые вы можете использовать. В предыдущем примере использовался протокол https://, вы также можете встретить git:// или user@server:path/to/repo.git, использующий протокол передачи SSH. В разделе Установка Git на сервер главы 4 мы познакомимся со всеми доступными вариантами конфигурации сервера для обеспечения доступа к вашему Git репозиторию, а также рассмотрим их достоинства и недостатки.

Запись изменений в репозиторий

Итак, у вас имеется настоящий Git-репозиторий и рабочая копия файлов для некоторого проекта. Вам нужно делать некоторые изменения и фиксировать «снимки» состояния (snapshots) этих изменений в вашем репозитории каждый раз, когда проект достигает состояния, которое вам хотелось бы сохранить.
Запомните, каждый файл в вашем рабочем каталоге может находиться в одном из двух состояний: под версионным контролем (отслеживаемые) и нет (неотслеживаемые). Отслеживаемые файлы - это те файлы, которые были в последнем снимке состояния проекта; они могут быть неизменёнными, изменёнными или подготовленными к коммиту. Если кратко, то отслеживаемые файлы - это те файлы, о которых знает Git.
Неотслеживаемые файлы - это всё остальное, любые файлы в вашем рабочем каталоге, которые не входили в ваш последний снимок состояния и не подготовлены к коммиту. Когда вы впервые клонируете репозиторий, все файлы будут отслеживаемыми и неизменёнными, потому что Git только что их извлек и вы ничего пока не редактировали.
Как только вы отредактируете файлы, Git будет рассматривать их как изменённые, так как вы изменили их с момента последнего коммита. Вы индексируете эти изменения, затем фиксируете все проиндексированные изменения, а затем цикл повторяется.

Источники:
https://git-scm.com
https://ru.hexlet.io



Определение состояния файлов

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

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
```

Это означает, что у вас чистый рабочий каталог, другими словами - в нём нет отслеживаемых изменённых файлов. Git также не обнаружил неотслеживаемых файлов, в противном случае они бы были перечислены здесь. Наконец, команда сообщает вам на какой ветке вы находитесь и сообщает вам, что она не расходится с веткой на сервере.
Предположим, вы добавили в свой проект новый файл, простой файл README. Если этого файла раньше не было, и вы выполните git status, вы увидите свой неотслеживаемый файл вот так:

```
$ echo 'My Project' > README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README

nothing added to commit but untracked files present (use "git add" to track)
```

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

Источники:
https://git-scm.com
https://ru.hexlet.io

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)
```

Источники:
https://git-scm.com
https://ru.hexlet.io

```
  (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)
```

Источники:
https://git-scm.com
https://ru.hexlet.io

```
   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
```

Сокращённый вывод статуса

Вывод команды git status довольно всеобъемлющий и многословный. Git также имеет флаг вывода сокращённого 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 упрощённый 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 рабочем каталоге, но не проиндексирован, а файл lib/simplegit.rb модифицирован и проиндексирован. Файл Rakefile модифицирован, проиндексирован и ещё раз модифицирован, таким образом на данный момент у него есть те изменения, которые попадут в коммит, и те, которые не попадут.

Источники:
https://git-scm.com
https://ru.hexlet.io

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 слеш (/) в начале шаблона.
- Чтобы исключить каталог добавьте слеш (/) 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 каталоги: `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 расширением .a
*.a

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

# Исключить файл TODO в корневом каталоге, но не файл в subdir/TODO
/TODO
```

Источники:
https://git-scm.com
https://ru.hexlet.io

```
# Игнорировать все файлы в каталоге build/
build/

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

# Игнорировать все .txt файлы в каталоге doc/
doc/**/*.txt
```

Просмотр индексированных и неиндексированных изменений

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

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
```

Источники:
https://git-scm.com
https://ru.hexlet.io

```
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
```

Источники:
https://git-scm.com
https://ru.hexlet.io

```
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    modified:   CONTRIBUTING.md
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
    modified:   CONTRIBUTING.md
```

Используйте `git diff` для просмотра непроиндексированных изменений

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

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

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

Источники:
https://git-scm.com
https://ru.hexlet.io

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

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

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

Источники:
https://git-scm.com
https://ru.hexlet.io

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 'Add new benchmarks'
[master 83e38c7] Add 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 будьте 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» (изменённые, 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:
```

Источники:
https://git-scm.com
https://ru.hexlet.io

```
  (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 слеш (`\`) 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` и находящиеся в каталоге `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 контроля версий, 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.

Источники:
https://git-scm.com
https://ru.hexlet.io

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 или rm перед коммитом.

Отправка изменений в удалённый репозиторий (Push)

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

```
$ git push origin master
```

Эта команда срабатывает только в случае, если вы клонировали с сервера, на котором у вас есть права на запись, и если никто другой с тех пор не выполнял команду push. Если вы и кто-то ещё одновременно клонируете, затем он выполняет команду push, а после него выполнить команду push попытаетесь вы, то ваш push точно будет отклонён. Вам придётся сначала получить изменения и объединить их с вашими и только после этого вам будет позволено выполнить push.

Просмотр истории коммитов

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

Источники:
https://git-scm.com
https://ru.hexlet.io

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

```
$ git clone https://github.com/devops321-top/test-clone.git
```

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

```
$ git log
commit 5d1f8b2e (HEAD -> main, origin/main, origin/HEAD)
Author: Aleksandr Vasilyevykh <myvps45@gmail.com>
Date:   Sat May 20 00:27:56 2023 +0600
    Add new file
commit 9ab6b706
Author: Aleksandr Vasilyevykh <myvps45@gmail.com>
Date:   Sat May 20 00:26:28 2023 +0600
    Add new file
commit 92248c10
Author: Aleksandr Vasilyevykh <myvps45@gmail.com>
Date:   Sat May 20 00:07:57 2023 +0600
    Add new file
commit b389df7a
Author: Aleksandr Vasilyevykh <myvps45@gmail.com>
Date:   Fri May 19 23:58:27 2023 +0600
    Initial commit
```

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

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

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

```
commit 5d1f8b2e (HEAD -> main, origin/main, origin/HEAD)
Author: Aleksandr Vasilyevykh <myvps45@gmail.com>
Date:   Sat May 20 00:27:56 2023 +0600
    Add new file
diff --git a/text2.txt b/text2.txt
new file mode 100644
index 00000000..8a6a2d09
--- /dev/null
+++ b/text2.txt
@@ -0,0 +1 @@
+line2
commit 9ab6b706
Author: Aleksandr Vasilyevykh <myvps45@gmail.com>
Date:   Sat May 20 00:26:28 2023 +0600
    Add new file
diff --git a/test.txt b/test.txt
new file mode 100644
```

Источники:

https://git-scm.com

https://ru.hexlet.io

```
index 00000000..47f1908c
--- /dev/null
+++ b/test.txt
@@ -0,0 +1 @@
+teste1
```

Операция 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.

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.

Отмена изменений в файле

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

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

```
Changes not staged for commit:
```

Источники:

https://git-scm.com
https://ru.hexlet.io

```
  (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.

Отмена действий с помощью git restore

Git версии 2.23.0 представил новую команду: `git restore`. По сути, это альтернатива git reset, которую мы только что рассмотрели. Начиная с версии 2.23.0, Git будет использовать `git restore` вместо `git reset` для многих операций отмены.
Давайте проследим наши шаги и отменим действия с помощью `git restore` вместо `git reset`.

Отмена индексации файла с помощью git restore

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

```
$ git add *

$ git status

On branch master

Changes to be committed:

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

        modified:   CONTRIBUTING.md

        renamed:    README.md -> README
```

Источники:
https://git-scm.com
https://ru.hexlet.io

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

```
$ git restore --staged CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

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

        renamed:    README.md -> README


Changes not staged for commit:

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

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

        modified:   CONTRIBUTING.md
```

Файл CONTRIBUTING.md изменен, но снова не индексирован.

Откат изменённого файла с помощью git restore

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

```
Changes not staged for commit:

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

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

        modified:   CONTRIBUTING.md
```

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

```
$ git restore CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

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

Источники:
https://git-scm.com
https://ru.hexlet.io

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

Ветвление в Git

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

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

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

Git не хранит данные в виде последовательности изменений, он использует набор снимков (snapshot).

Snapshot *- файлов и каталогов файловой системы, базы данных или её части на определённый момент времени.*

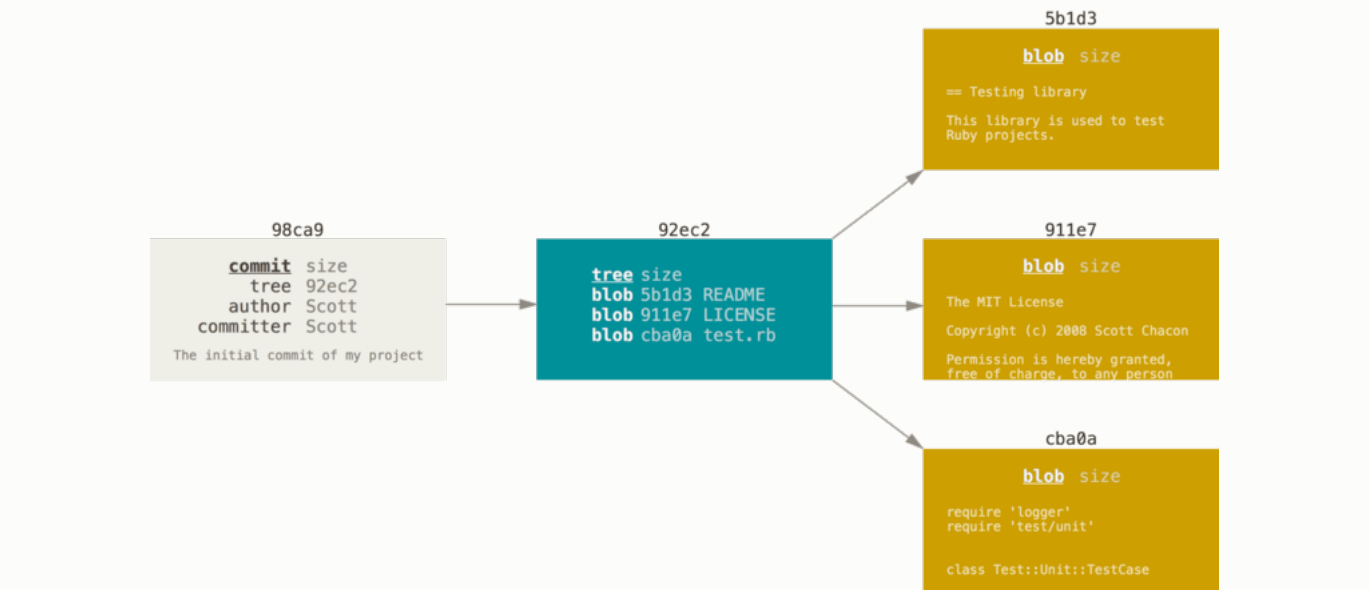
Когда вы делаете коммит, Git сохраняет его в виде объекта, который содержит указатель на снимок (snapshot) подготовленных данных. Этот объект так же содержит имя автора и email, сообщение и указатель на коммит или коммиты непосредственно предшествующие данному (его родителей): отсутствие родителя для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

Предположим, у вас есть каталог с тремя файлами и вы добавляете их все в индекс и создаёте коммит. Во время индексации вычисляется контрольная сумма каждого файла^ затем каждый файл сохраняется в репозиторий (Git называет такой файл **блоб** - большой бинарный объект), а контрольная сумма попадёт в индекс:

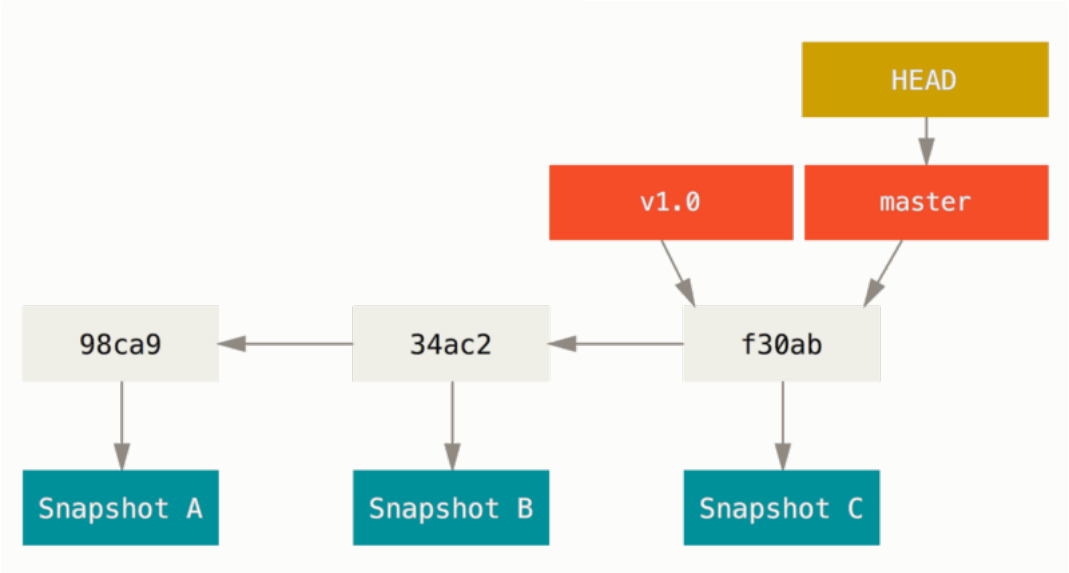
```
$ git add README test.rb LICENSE

$ git commit -m 'Initial commit'
```

Когда вы создаёте коммит командой `git commit`, Git вычисляет контрольные суммы каждого подкаталога (в нашем случае, только основной каталог проекта) и сохраняет его в репозитории как объект дерева каталогов. Затем Git создаёт объект коммита с метаданными и указателем на основное дерево проекта для возможности воссоздать этот снимок в случае необходимости.
Ваш репозиторий Git теперь хранит пять объектов: три блоб объекта (по одному на каждый файл), объект **дерева** каталогов, содержащий список файлов и соответствующих им блобов, а так же объект **коммита**, содержащий метаданные и указатель на объект дерева каталогов.

Источники:
https://git-scm.com
https://ru.hexlet.io



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

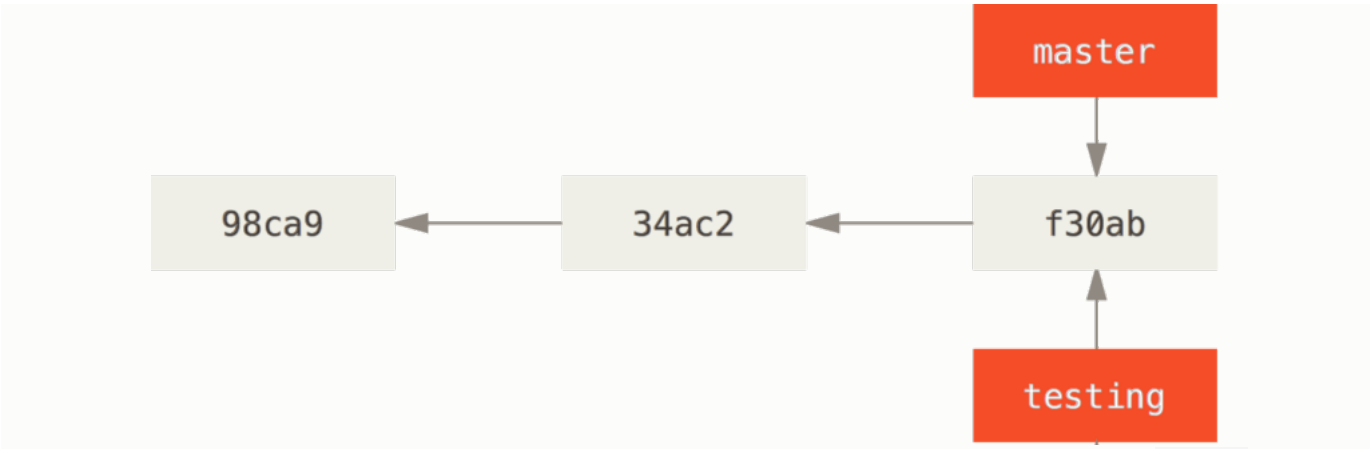


Создание новой ветки

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

```
$ git branch testing
```

Источники:
https://git-scm.com
https://ru.hexlet.io



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

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.

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.

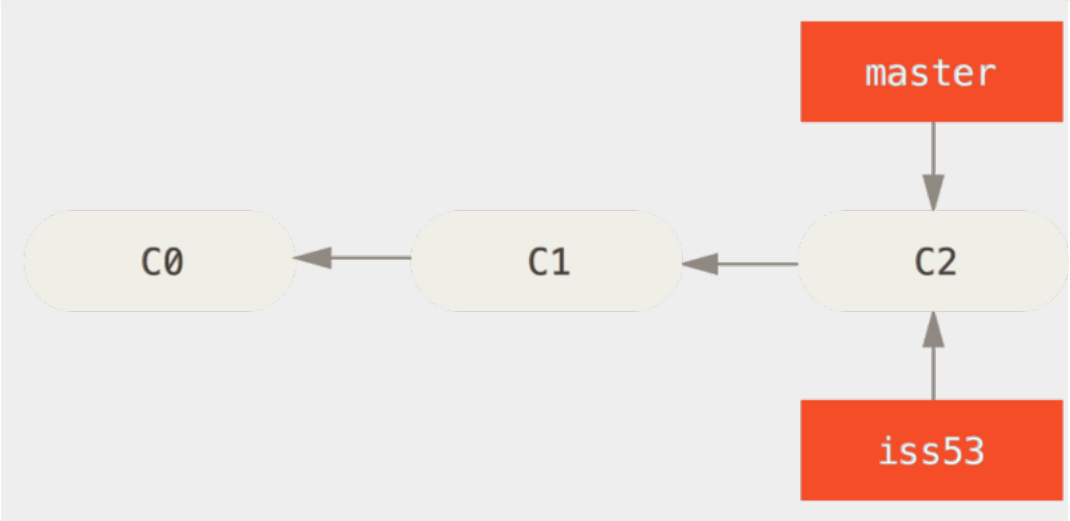
Вы решаете, что теперь вы будете заниматься проблемой #53 из вашей системы отслеживания ошибок. Чтобы создать ветку и сразу переключиться на неё, можно выполнить команду git checkout с параметром -b:

Источники:
https://git-scm.com
https://ru.hexlet.io

```
$ git checkout -b iss53

Switched to a new branch "iss53"
```

Это то же самое что и:

```
$ git branch iss53

$ git checkout iss53
```



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

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`.

```
$ git checkout master

Switched to branch 'master'
```

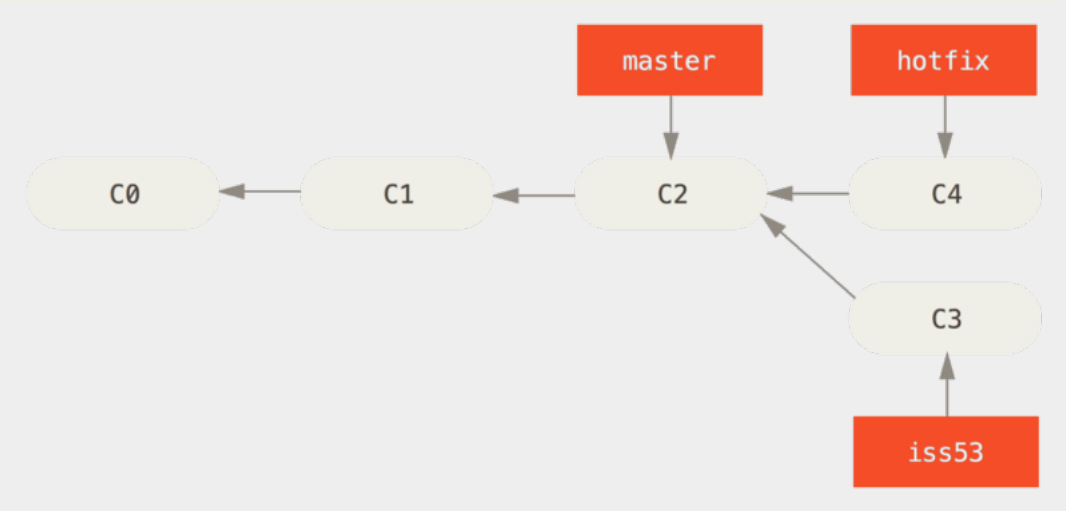
С этого момента ваш рабочий каталог имеет точно такой же вид, какой был перед началом работы над проблемой #53, и вы можете сосредоточиться на работе над исправлением. Важно запомнить: когда вы переключаете ветки, Git возвращает состояние рабочего каталога к тому виду, какой он имел в момент последнего 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 'Fix broken email address'
```

Источники:
https://git-scm.com
https://ru.hexlet.io

```
[hotfix 1fb7853] Fix broken email address

 1 file changed, 2 insertions(+)
```



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.

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:

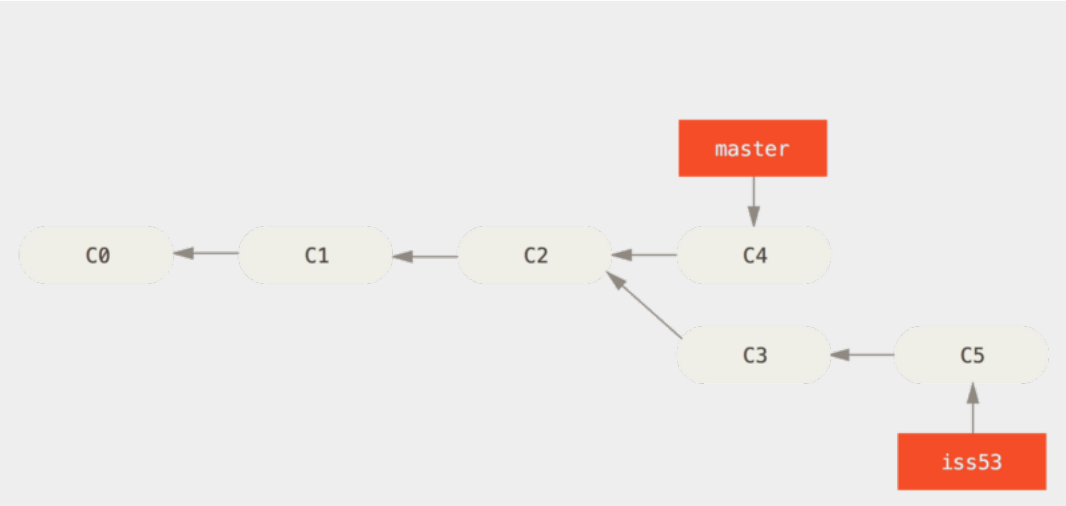
```
$ git checkout iss53
```

Источники:

https://git-scm.com

https://ru.hexlet.io

```
Switched to branch "iss53"

$ vim index.html

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

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

1 file changed, 1 insertion(+)
```



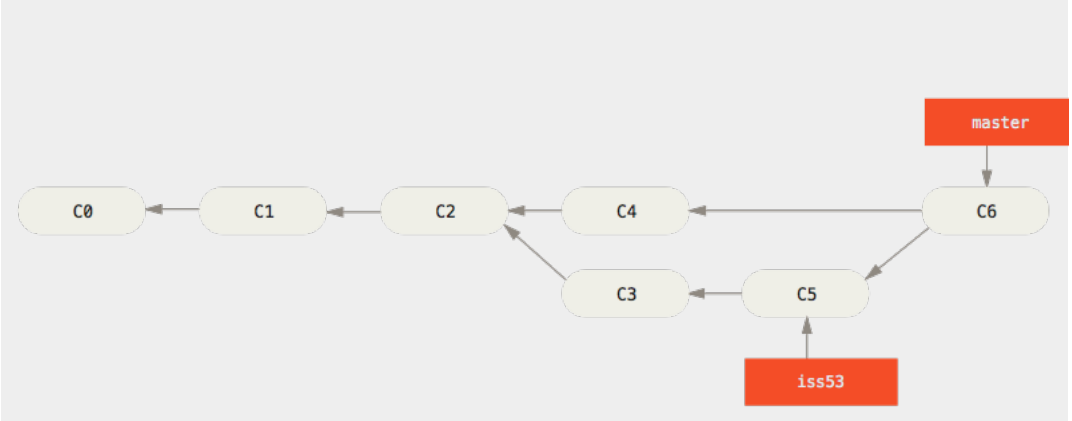
Основы слияния

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.
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.

Источники:
https://git-scm.com
https://ru.hexlet.io



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

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")
```

Источники:
https://git-scm.com
https://ru.hexlet.io

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

```
<<<<<<< HEAD:index.html

<div id="footer">contact : email.support@github.com</div>

=======

<div id="footer">

 please contact us at support@github.com

</div>

>>>>>>> iss53:index.html
```

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

```
<div id="footer">

please contact us at email.support@github.com

</div>
```

В этом разрешении есть немного от каждой части, а строки <<<<<<<, ======= и >>>>>>> полностью удалены. Разрешив каждый конфликт во всех файлах, запустите git add для каждого файла, чтобы отметить конфликт как решённый. Добавление файла в индекс означает для Git, что все конфликты в нём исправлены.
Если вы хотите использовать графический инструмент для разрешения конфликтов, можно запустить git mergetool, который проведет вас по всем конфликтам:

```
$ git mergetool



This message is displayed because 'merge.tool' is not configured.

See 'git mergetool --tool-help' or 'git help config' for more details.

'git mergetool' will now attempt to use one of the following tools:

opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge araxis bc3 codecompare vimdiff emerge

Merging:

index.html
```

Источники:
https://git-scm.com
https://ru.hexlet.io

```
Normal merge conflict for 'index.html':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (opendiff):
```

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

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

```
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

    modified:   index.html
```

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

```
Merge branch 'iss53'

Conflicts:
    index.html
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#       .git/MERGE_HEAD
# and try again.
```

Источники:
https://git-scm.com
https://ru.hexlet.io

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#       modified:   index.html
```

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

Управление ветками

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

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

```
$ git branch
  iss53
* master
  testing
```

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

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

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

Источники:
https://git-scm.com
https://ru.hexlet.io

```
$ git branch --merged

  iss53

* master
```

Ветка iss53 присутствует в этом списке потому что вы ранее слили её в master. Те ветки из этого списка, перед которыми нет символа *, можно смело удалять командой git branch -d; наработки из этих веток уже включены в другую ветку, так что ничего не потеряется.
Чтобы увидеть все ветки, содержащие наработки, которые вы пока ещё не слили в текущую ветку, выполните команду git branch --no-merged:

```
$ git branch --no-merged

  testing
```

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

```
$ git branch -d testing

error: The branch 'testing' is not fully merged.

If you are sure you want to delete it, run 'git branch -D testing'.
```

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

Переименование ветки

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

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

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

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

Теперь проверим, где мы сейчас находимся:

```
$ git branch --all

* corrected-branch-name

  main

  remotes/origin/bad-branch-name

  remotes/origin/corrected-branch-name

  remotes/origin/main
```

Источники:
https://git-scm.com
https://ru.hexlet.io

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

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

Теперь старое имя ветки полностью заменено исправленным.

Изменение имени главной ветки

Переименуйте локальную ветку master в main с помощью следующей команды:

```
$ git branch --move master main
```

После этого, локальной ветки master больше не существует, потому что она была переименована в ветку main.
Чтобы все остальные могли видеть новую ветку main, вам нужно отправить её в общий репозиторий. Это делает переименованную ветку доступной в удалённом репозитории.

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

В итоге, состояние репозитория становится следующим:

```
$ git branch --all

* main

  remotes/origin/HEAD -> origin/master

  remotes/origin/main

  remotes/origin/master
```

Ваша локальная ветка master исчезла, так как она заменена веткой main. Ветка main доступна в удалённом репозитории. Старая ветка master всё ещё присутствует в удалённом репозитории. Остальные участники будут продолжать использовать ветку master в качестве основы для своей работы, пока вы не совершите ряд дополнительных действий.
Теперь, для завершения перехода на новую ветку перед вами стоят следующие задачи:

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

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

Источники:
https://git-scm.com
https://ru.hexlet.io

```
$ git push origin --delete master
```

Получение изменений

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

Удаление веток на удалённом сервере

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

```
$ git push origin --delete serverfix

To https://github.com/schacon/simplegit

 - [deleted]         serverfix
```

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

Перебазирования

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

Как мы выяснили ранее, простейший способ выполнить слияние двух веток - это команда merge. Она осуществляет трёхстороннее слияние между двумя последними снимками сливаемых веток (C3 и C4) и самого недавнего общего для этих веток родительского снимка (C2), создавая новый снимок (и коммит).
Тем не менее есть и другой способ: вы можете взять те изменения, что были представлены в C4, и применить их поверх C3. В Git это называется **перебазированием**. С помощью команды rebase вы можете взять все коммиты из одной ветки и в том же порядке применить их к другой ветке.
В данном примере переключимся на ветку experiment и перебазируем её относительно ветки master следующим образом:

Источники:
https://git-scm.com
https://ru.hexlet.io

```
$ git checkout experiment

$ git rebase master

First, rewinding head to replay your work on top of it...

Applying: added staged command
```

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

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

```
$ git checkout master

$ git merge experiment
```

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

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

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

Дополнительные команды:

git cherry-pick

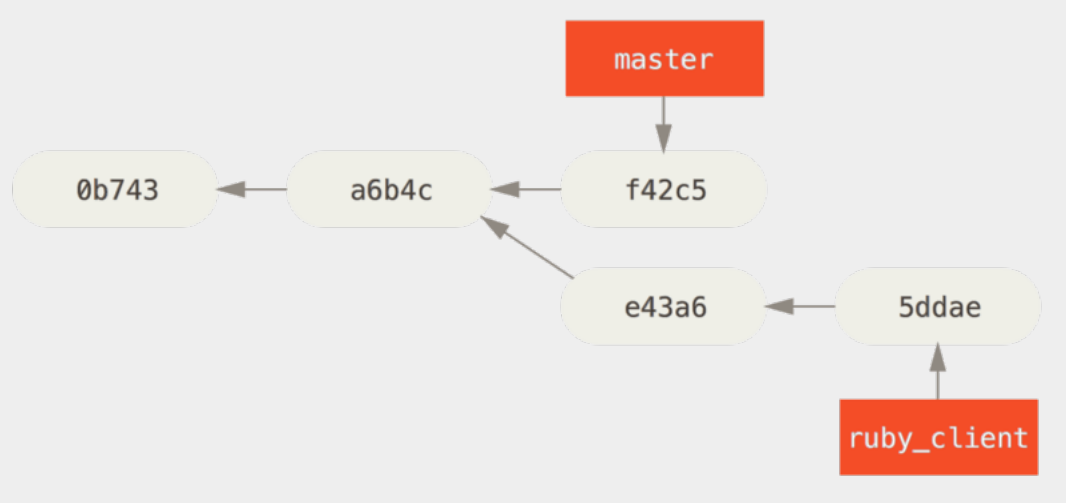
Команда `git cherry-pick` берёт изменения, вносимые одним коммитом, и пытается повторно применить их в виде нового коммита в текущей ветке. Эта возможность полезна в ситуации, когда нужно забрать парочку коммитов из другой ветки, а не сливать ветку целиком со всеми внесёнными в неё изменениями.

Схема с перебазированием и отбором

Некоторые сопровождающие предпочитают перебазировать или выборочно применять (cherry-pick) изменения относительно ветки `master` вместо слияния, что позволяет поддерживать историю проекта в линейном виде. Когда проделанная работа из тематической ветки готова к интеграции, вы

Источники:
https://git-scm.com
https://ru.hexlet.io

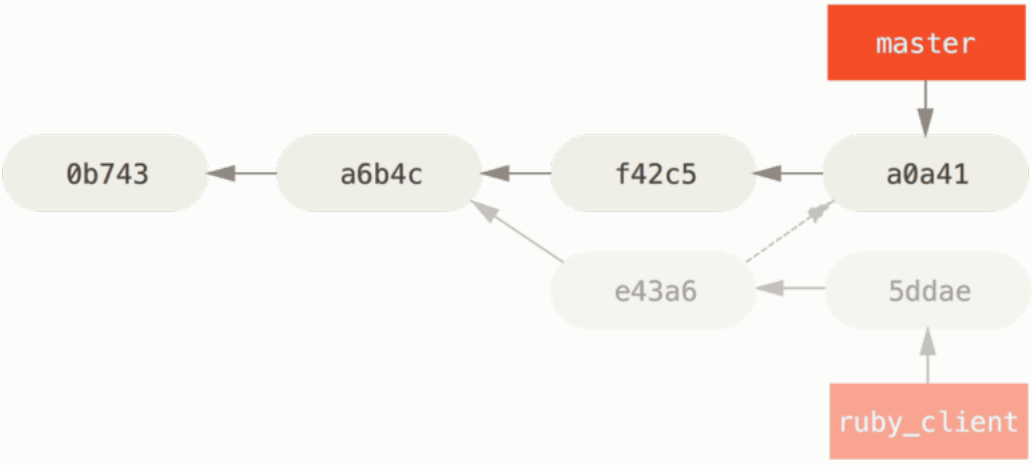
переходите на эту ветку и перебазируете её относительно ветки master (или develop и т. д.). Если конфликты отсутствуют, то вы можете просто сдвинуть состояние ветки master, что обеспечивает линейность истории проекта.
Другим способом переместить предлагаемые изменений из одной ветки в другую является их отбор коммитов (cherry-pick). Отбор в Git похож на перебазирование для одного коммита. В таком случае формируется патч для выбранного коммита и применяется к текущей ветке. Это полезно, когда в тематической ветке присутствует несколько коммитов, а вы хотите взять только один из них, или в тематической ветке только один коммит и вы предпочитаете использовать отбор вместо перебазирования. Предположим, ваш проект выглядит так:



Для применения коммита e43a6 к ветке master выполните команду:

```
$ git cherry-pick e43a6

Finished one cherry-pick.

[master]: created a0a41a9: "More friendly message when locking the index fails."

 3 files changed, 17 insertions(+), 3 deletions(-)
```

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



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

Припрятывание и очистка

Источники:
https://git-scm.com
https://ru.hexlet.io

Часто пока вы работаете над одной частью вашего проекта и всё находится в беспорядке, у вас возникает желание сменить ветку и поработать над чем-то ещё. Сложность при этом заключается в том, что вы не хотите фиксировать наполовину сделанную работу только для того, чтобы иметь возможность вернуться к ней позже. Справиться с ней помогает команда `git stash`.
Операция `stash` берет изменённое состояние вашего рабочего каталога, то есть изменённые отслеживаемые файлы и проиндексированные изменения, и сохраняет их в хранилище незавершённых изменений, которые вы можете в любое время применить обратно.

Припрятывание ваших наработок

Для примера, предположим, что вы перешли в свой проект, начали работать над несколькими файлами и, возможно, добавили в индекс изменения одного из них. Если вы выполните `git status`, то увидите ваше изменённое состояние:

```
$ git status

Changes to be committed:

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

modified:   index.html

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:   lib/simplegit.rb
```

Теперь вы хотите сменить ветку, но пока не хотите фиксировать ваши текущие наработки; поэтому вы припрячете эти изменения. Для того, чтобы припрятать изменение в выделенное для этого специальное хранилище, выполните `git stash` или `git stash push`:

```
$ git stash

Saved working directory and index state \

  "WIP on master: 049d078 Create index file"

HEAD is now at 049d078 Create index file

(To restore them type "git stash apply")
```

Теперь вы можете увидеть, что рабочая копия не содержит изменений:

```
$ git status

# On branch master

nothing to commit, working directory clean
```

В данный момент вы можете легко переключать ветки и работать в любой; ваши изменения сохранены. Чтобы посмотреть список припрятанных изменений, вы можете использовать `git stash list`:

Источники:
https://git-scm.com
https://ru.hexlet.io

```
$ git stash list

stash@{0}: WIP on master: 049d078 Create index file

stash@{1}: WIP on master: c264051 Revert "Add file_size"

stash@{2}: WIP on master: 21d80a5 Add number to log
```

В данном примере, предварительно были припрятаны два изменения, поэтому теперь вам доступны три различных отложенных наработки. Вы можете применить только что припрятанные изменения, используя команду, указанную в выводе исходной команды: `git stash apply`. Если вы хотите применить одно из предыдущих припрятанных изменений, вы можете сделать это, используя его имя, вот так: `git stash apply stash@{2}`. Если вы не укажете имя, то Git попытается восстановить самое последнее припрятанное изменение:

```
$ git stash apply

On branch 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:   index.html

modified:   lib/simplegit.rb

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

Как видите, Git восстановил в файлах изменения, которые вы отменили ранее, когда прятали свои наработки. В данном случае при применении отложенных наработок ваш рабочий каталог был без изменений, а вы пытались применить их в той же ветке, в которой вы их и сохранили; но отсутствие изменений в рабочем каталоге и применение их в той же ветке не являются необходимыми условиями для успешного восстановления припрятанных наработок. Вы можете припрятать изменения, находясь в одной ветке, а затем переключиться на другую и попробовать восстановить эти изменения. Также при восстановлении припрятанных наработок в вашем рабочем каталоге могут присутствовать изменённые и незафиксированные файлы - Git выдаст конфликты слияния, если не сможет восстановить какие-то наработки.

Спрятанные изменения будут применены к вашим файлам, но файлы, которые вы ранее добавляли в индекс, не будут добавлены туда снова. Для того, чтобы это было сделано, вы должны запустить `git stash apply` с опцией `--index`, при которой команда попытается восстановить изменения в индексе. Если вы выполните команду таким образом, то полностью восстановите ваше исходное состояние:

```
$ git stash apply --index

On branch master

Changes to be committed:

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

Источники:
https://git-scm.com
https://ru.hexlet.io

```
modified:   index.html

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:   lib/simplegit.rb
```

Команда `apply` только пытается восстановить припрятанные наработки - при этом они останутся в хранилище. Для того, чтобы удалить их, вы можете выполнить `git stash drop`, указав имя удаляемых изменений:

```
$ git stash list

stash@{0}: WIP on master: 049d078 Create index file

stash@{1}: WIP on master: c264051 Revert "Add file_size"

stash@{2}: WIP on master: 21d80a5 Add number to log

$ git stash drop stash@{0}

Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

Вы также можете выполнить `git stash pop`, чтобы применить припрятанные изменения и тут же удалить их из хранилища.

Необычное припрятывание

У припрятанных изменений есть несколько дополнительных вариантов использования, которые также могут быть полезны. Первый - это использование довольно популярной опции `--keep-index` с командой `git stash`. Она просит Git не только припрятать то, что вы уже добавили в индекс, но одновременно оставить это в индексе.

```
$ git status -s

M  index.html

 M lib/simplegit.rb

$ git stash --keep-index

Saved working directory and index state WIP on master: 1b65b17 added the index file

HEAD is now at 1b65b17 added the index file

$ git status -s

M  index.html
```

Другой распространённый вариант, который вы, возможно, захотите использовать - это припрятать помимо отслеживаемых файлов также и неотслеживаемые. По умолчанию `git stash` будет сохранять только изменённые и проиндексированные **отслеживаемые** файлы. Если вы укажете опцию `--include-untracked` или `-u`, Git также припрячет все неотслеживаемые файлы, которые вы создали.

Источники:
https://git-scm.com
https://ru.hexlet.io

Однако включение этой опции по-прежнему не будет прятать файлы с явным игнорированием; чтобы дополнительно припрятать игнорируемые файлы, используйте --all (или просто -a).

```
$ git status -s
M  index.html
 M lib/simplegit.rb
?? new-file.txt
$ git stash -u
Saved working directory and index state WIP on master: 1b65b17 added the index file
HEAD is now at 1b65b17 added the index file
$ git status -s
```

И наконец, если вы укажете флаг --patch, Git не будет ничего прятать, а вместо этого в интерактивном режиме спросит вас о том, какие из изменений вы хотите припрятать, а какие оставить в вашем рабочем каталоге.

```
$ git stash --patch
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 66d332e..8bb5674 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -16,6 +16,10 @@ class SimpleGit
         return `#{git_cmd} 2>&1`.chomp
       end
     end
+
+    def show(treeish = 'master')
+      command("git show #{treeish}")
+    end
 
 end
 test
```

Источники:
https://git-scm.com
https://ru.hexlet.io

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

Saved working directory and index state WIP on master: 1b65b17 added the index file
```

Создание ветки из припрятанных изменений

Если вы спрятали некоторые изменения, оставили их на время, а сами продолжили работать в той же ветке, у вас могут возникнуть проблемы с восстановлением наработок. Если восстановление будет затрагивать файл, который уже был изменён с момента сохранения наработок, то вы получите конфликт слияния и должны будете попытаться разрешить его. Если вам нужен более простой способ снова протестировать припрятанные изменения, вы можете выполнить команду `git stash branch`, которая создаст для вас новую ветку, перейдёт на коммит, на котором вы были, когда прятали свои наработки, применит на нём эти наработки и затем, если они применились успешно, удалит эти припрятанные изменения:

```
$ git stash branch testchanges

M       index.html

M       lib/simplegit.rb

Switched to a new branch 'testchanges'

On branch testchanges

Changes to be committed:

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

modified:   index.html

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:   lib/simplegit.rb

Dropped refs/stash@{0} (29d385a81d163dfd45a452a2ce816487a6b8b014)
```

Это удобное сокращение для того, чтобы легко восстановить припрятанные изменения и поработать над ними в новой ветке.

Очистка рабочего каталога

Наконец, у вас может возникнуть желание не прятать некоторые из изменений или файлов в вашем рабочем каталоге, а просто избавиться от них. Команда `git clean` сделает это для вас.
Одной из распространённых причин для этого может быть удаление мусора, который был сгенерирован при слиянии или внешними утилитами, или удаление артефактов сборки в процессе её очистки.

Вам нужно быть очень аккуратными с этой командой, так как она предназначена для удаления неотслеживаемых файлов из вашего рабочего каталога. Даже если вы передумаете, очень часто нельзя восстановить содержимое таких файлов. Более безопасным вариантом является
Источники:
https://git-scm.com
https://ru.hexlet.io

использование команды git stash --all для удаления всего, но с сохранением этого в виде припрятанных изменений.
Предположим, вы хотите удалить мусор и очистить ваш рабочий каталог; вы можете сделать это с помощью git clean. Для удаления всех неотслеживаемых файлов в вашем рабочем каталоге, вы можете выполнить команду git clean -f -d, которая удалит все файлы и также все каталоги, которые в результате станут пустыми. Параметр -f (сокращение от слова force - заставить) означает принудительное удаление, подчеркивая, что вы действительно хотите это сделать, и требуется, если переменная конфигурации Git clean.requireForce явным образом не установлена в false.
Если вы хотите только посмотреть, что будет сделано, вы можете запустить команду с опцией -n, которая означает «имитируй работу команды и скажи мне, что ты **будешь** удалять».

```
$ git clean -d -n

Would remove test.o

Would remove tmp/
```

По умолчанию команда git clean будет удалять только неотслеживаемые файлы, которые не добавлены в список игнорируемых. Любой файл, который соответствует шаблону в вашем .gitignore, или другие игнорируемые файлы не будут удалены. Если вы хотите удалить и эти файлы (например, удалить все .o-файлы, генерируемые в процессе сборки, и таким образом полностью очистить сборку), вы можете передать команде очистки опцию -x.

```
$ git status -s

 M lib/simplegit.rb

?? build.TMP

?? tmp/

$ git clean -n -d

Would remove build.TMP

Would remove tmp/

$ git clean -n -d -x

Would remove build.TMP

Would remove test.o

Would remove tmp/
```

Если вы не знаете, что сделает при запуске команда git clean, всегда сначала выполняйте её с опцией -n, чтобы проверить дважды, перед заменой -n на -f и выполнением настоящей очистки. Другой способ, который позволяет вам более тщательно контролировать сам процесс - это выполнение команды с опцией -i (в «интерактивном» режиме).
Ниже выполнена команда очистки в интерактивном режиме.

```
$ git clean -x -i
```

Источники:
https://git-scm.com
https://ru.hexlet.io

```
Would remove the following items:
  build.TMP  test.o
*** Commands ***
    1: clean                2: filter by pattern    3: select by numbers    4: ask each             5: quit
    6: help
What now>
```

Таким образом, вы можете просмотреть каждый файл индивидуально или указать шаблоны для удаления в интерактивном режиме.

Работа с тегами

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

Просмотр списка тегов

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

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

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

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

```
$ git tag -l "v1.8.5*"
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
```

Источники:
https://git-scm.com
https://ru.hexlet.io

```
v1.8.5.3

v1.8.5.4

v1.8.5.5
```

Создание тегов

Git использует два основных типа тегов: легковесные и аннотированные.

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

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

Аннотированные теги

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

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

$ git tag

v0.1

v1.3

v1.4
```

Опция `-m` задаёт сообщение, которое будет храниться вместе с тегом. Если не указать сообщение, то Git запустит редактор, чтобы вы смогли его ввести.
С помощью команды `git show` вы можете посмотреть данные тега вместе с коммитом:

```
$ git show v1.4

tag v1.4

Tagger: Ben Straub <ben@straub.cc>

Date:   Sat May 3 20:19:12 2014 -0700

my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949

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

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

Источники:
https://git-scm.com
https://ru.hexlet.io

```
    Change version number
```

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

Легковесные теги

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

```
$ git tag v1.4-lw

$ git tag

v0.1

v1.3

v1.4

v1.4-lw

v1.5
```

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

```
$ git show v1.4-lw

commit ca82a6dff817ec66f44342007202690a93763949

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

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

    Change version number
```

Отложенная расстановка тегов

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

```
$ git log --pretty=oneline

15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'

a6b4c97498bd301d84096da251c98a07c7723e65 Create write support

0d52aaab4479697da7686c15f77a3d64d9165190 One more thing

6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
```

Источники:
https://git-scm.com
https://ru.hexlet.io

```
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc Add commit function

4682c3261057305bdd616e23b64b0857d832627b Add todo file

166ae0c4d3f420721acbb115cc33848dfcc2121a Create write support

9fceb02d0ae598e95dc970b74767f19372d61af8 Update rakefile

964f16d36dfccde844893cac5b347e7b3d44abbc Commit the todo

8a5cbc430f1a9c3d00faaeffd07798508422908a Update readme
```

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

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

Проверим, что коммит отмечен:

```
$ git tag

v0.1

v1.2

v1.3

v1.4

v1.4-lw

v1.5


$ git show v1.2

tag v1.2

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

Date:   Mon Feb 9 15:32:16 2009 -0800

version 1.2

commit 9fceb02d0ae598e95dc970b74767f19372d61af8

Author: Magnus Chacon <mchacon@gee-mail.com>

Date:   Sun Apr 27 20:43:35 2008 -0700

    Update rakefile

...
```

Обмен тегами

Источники:
https://git-scm.com
https://ru.hexlet.io

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

```
$ git push origin v1.5

Counting objects: 14, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (12/12), done.

Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.

Total 14 (delta 3), reused 0 (delta 0)

To git@github.com:schacon/simplegit.git

 * [new tag]         v1.5 -> v1.5
```

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

```
$ git push origin --tags

Counting objects: 1, done.

Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.

Total 1 (delta 0), reused 0 (delta 0)

To git@github.com:schacon/simplegit.git

 * [new tag]         v1.4 -> v1.4

 * [new tag]         v1.4-lw -> v1.4-lw
```

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

Удаление тегов

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

```
$ git tag -d v1.4-lw

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

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

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

```
$ git push origin :refs/tags/v1.4-lw
```

Источники:
https://git-scm.com
https://ru.hexlet.io

```
To /git@github.com:schacon/simplegit.git

 - [deleted]         v1.4-lw
```

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

Второй способ убрать тег из внешнего репозитория более интуитивный:

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

Переход на тег

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

```
$ git checkout v2.0.0

Note: switching to 'v2.0.0'.

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

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

state without impacting any branches by performing another checkout.

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

do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

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

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

$ git checkout v2.0-beta-0.1

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

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

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

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

Switched to a new branch 'version2'
```

Источники:
https://git-scm.com
https://ru.hexlet.io

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

Источники:
https://git-scm.com
https://ru.hexlet.io