Основы DevOps

КОДЕБАЙ
АКАДЕМИЯ

ОСНОВЫ СИСТЕМ КОНТРОЛЯ ВЕРСИЙ. ЗНАКОМСТВО С GIT

ОГЛАВЛЕНИЕ

НЕКОТОРЫЕ ТЕРМИНЫ

.git - директория, где Git хранит метаданные и базу объектов репозитория. Это самая важная часть Git, и это та часть, которая копируется при клонировании.

Branch — это ветвь развития. Самый последний коммит в ветке называется вершиной этой ветки. На вершину ветки ссылается головка (*HEAD*) ветки, которая продвигается вперед по мере того, как в ветке появляются дополнительные коммиты. Один репозиторий Git может отслеживать произвольное количество ветвей.

Commit - специальное название сохранения версии. Так называются все изменения в файлах, зафиксированные в репозитории, одним именем.

Fetch - получение ветки из удаленного репозитория, чтобы выяснить, каких объектов не хватает в локальной базе данных, а также получить их.

Index - набор файлов со статистической информацией, содержимое которых хранится в виде объектов (*objects*). Индекс — это сохраненная версия вашего рабочего дерева. Представляет из себя файл, располагающийся в *.git* директории.

Merge – объединение содержимого другой ветки (возможно, из внешнего репозитория) с текущей веткой. В случае, когда объединяемая ветвь находится в другом репозитории, это делается путем сначала извлечения удаленной ветки, а затем слияния результата с текущей ветвью. Эта комбинация операций получения и слияния называется извлечением (*pull*). Слияние выполняется с помощью автоматического процесса, который идентифицирует изменения, внесенные с момента расхождения ветвей, а затем применяет все эти изменения вместе. В случаях конфликта изменений может потребоваться ручное вмешательство для завершения слияния.

Objects - единица хранения в Git. Он однозначно идентифицируется SHA-1 своего содержимого. Объект не может быть изменен.

Pull – получение ветки (*fetch*) и ее объединение (*merge*).

Push - отправка изменений на удаленный репозиторий (*remote repository*).

Remote repository - удаленный репозиторий, который используется для отслеживания того же проекта, но находится в другом месте.

Working tree – рабочая директория, дерево фактически извлеченных файлов. Рабочее дерево обычно содержит содержимое репозитория на которое указывает *HEAD*, а также любые локальные изменения, которые вы сделали, но еще не зафиксировали.

ОСНОВЫ СИСТЕМ КОНТРОЛЯ ВЕРСИЙ

О СИСТЕМЕ КОНТРОЛЯ ВЕРСИЙ

Проблема работы с данными, представленными файлами, без применения систем контроля версий заключается в том, что:

- контроль версий осуществляется копированием файлов в другой каталог, как правило добавляя текущую дату к названию каталога;
- случайное изменение не тех фалов или изменение файлов не в том каталоге;
- копирование файлов не туда, куда надо было, и в результате затирание нужных файлов;
- нет информации о том, что изменяется от версии к версии.

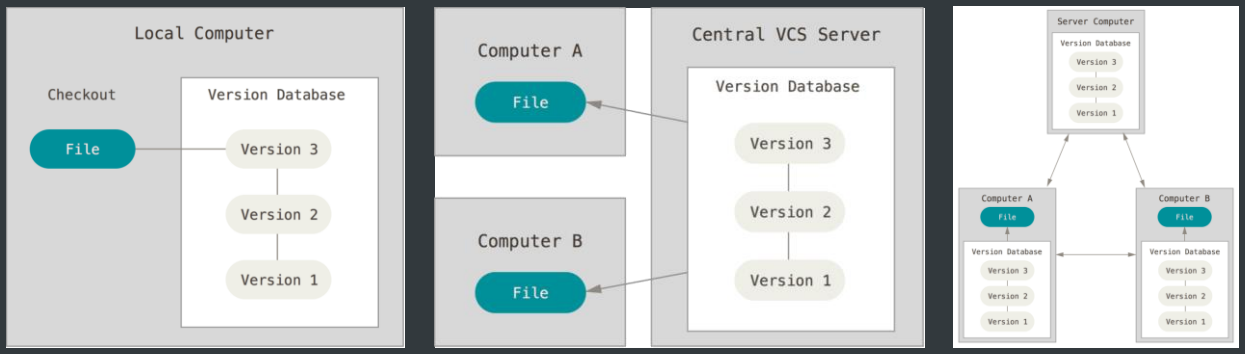
Системы контроля версий (Version Control System, VCS) – система, записывающая изменения файла или набора файлов в течение большого периода времени, так чтобы была возможность позже вернуться к определенной версии.

Системы контроля версий – это:

- единый “источник правды”;
- основа для коллективной работы;
- основа процесса разработки ПО и всего процесса непрерывной поставки;
- основа для DevOps-практик.

ФУНКЦИИ СИСТЕМЫ КОНТРОЛЯ ВЕРСИЙ

ТИПЫ СИСТЕМ КОНТРОЛЯ ВЕРСИЙ



ЛОКАЛЬНЫЕ

Основываются на простой базе данных, в которой хранятся изменения нужных файлов

Пример:

RCS (англ. Revision Control System) — одна из самых первых систем управления версиями, разработанная в 1982 году. Для каждого файла, зарегистрированного в системе, она хранит полную историю изменений.

Преимущества:

- проста использования, хорошо подходит для ознакомления с принципами работы систем контроля версий;
- хорошо подходит для резервного копирования отдельных файлов;

Недостатки:

- отслеживает изменения только отдельных файлов, что не позволяет использовать ее для управления версиями больших проектов;
- не позволяет одновременно вносить изменения в один и тот же файл несколькими пользователями;
- низкая функциональность, по сравнению с современными системами контроля версий.

ЦЕНТРАЛИЗОВАННЫЕ

Позволяют сотрудничать разработчикам. Есть центральный сервер, на котором хранятся все файлы под версионным контролем, и ряд клиентов, которые взаимодействуют с этим сервером.

Пример:

Subversion (также известная как «SVN») — свободная централизованная система управления версиями, официально выпущенная в 2004 году компанией CollabNet. С 2010 года Subversion является одним из проектов Apache Software Foundation и официально называется Apache Subversion.

Преимущества:

- все знают, кто и чем занимается в проекте;
- у администраторов есть четкий контроль над тем, кто и что может делать

Недостатки:

- централизованный сервер является уязвимым местом всей системы. Если сервер не работает, то разработчики не могут взаимодействовать, и никто не может сохранить новой версии своей работы;
- все знают, кто и чем занимается в проекте;
- если отсутствует подключение сети у разработчика, в это время он также не может взаимодействовать с сервером;
- если повреждается диск с центральной базой данных и нет резервной копии, теряется абсолютно вся история проекта, за исключением нескольких рабочих версий, сохранившихся на рабочих машинах пользователей.

РАСПРЕДЕЛЕННЫЕ

В отличие от централизованных систем клиенты не просто выгружают последние версии файлов, а полностью копируют весь репозиторий. Каждая копия репозитория является полным бэкапом всех данных.

Пример:

Mercurial — кроссплатформенная распределённая система управления версиями, разработанная для эффективной работы с очень большими репозиториями кода.

Git — распределённая система управления версиями. Проект был создан Линусом Торвальдсом для управления разработкой ядра Linux, первая версия выпущена 7 апреля 2005 года.

Преимущества:

- полноценная локальная копия проекта;
- локальный репозиторий равно резервная копия;

- можно работать оффлайн;
- скорость работы над проектом.

Недостатки:

- высокий порог вхождения (в случае с Git)
- особенности работы в различных ОС (в случае с Git)

СЕРВЕРЫ GIT

Для совместной работы в Git, необходим удалённый репозиторий. Несмотря на то, что технически можно отправлять и забирать изменения непосредственно из личных репозиториев, делать это не рекомендуется. Предпочтительный метод взаимодействия с кем-либо — это создание промежуточного репозитория, к которому все участники будут иметь доступ, и отправка и получение изменений через него.

Подобный Git-сервер можно развернуть самостоятельно, либо воспользоваться уже готовыми решениями, реализующими работы, как с приватными, так и с публичными репозиториями. Существуют множество решений для развертывания on-premises (например, Gitea, GitLab, BitBucket), либо использования в режиме SaaS (например, GitLab, BitBucket, Github).

В данном курсе для работы с Git будет использоваться GitHub. Для этого вам необходимо создать аккаунт в системе, создать публичный репозиторий, а также дополнительно можно настроить доступ к репозиторию по протоколу SSH.

АРХИТЕКТУРА И ОБЪЕКТЫ GIT

КАК РАБОТАЕТ GIT

Git — простое хранилище типа ключ-значение:

- В Git для всего вычисляется хеш-сумма, и только потом происходит сохранение.
- Обращение к сохранённым объектам происходит по хеш-сумме.
- Невозможно изменить содержимое файла или директории так, чтобы Git не узнал об этом.

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

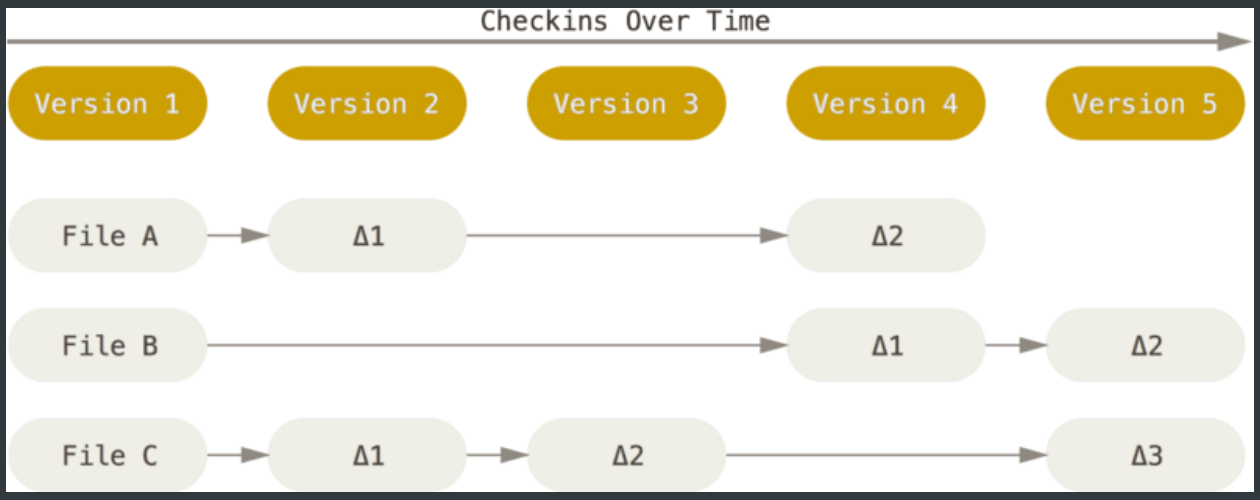
```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Данная функциональность встроена в Git на низком уровне и является неотъемлемой частью его философии. Вы не потеряете информацию во время её передачи и не получите повреждённый файл без ведома Git.

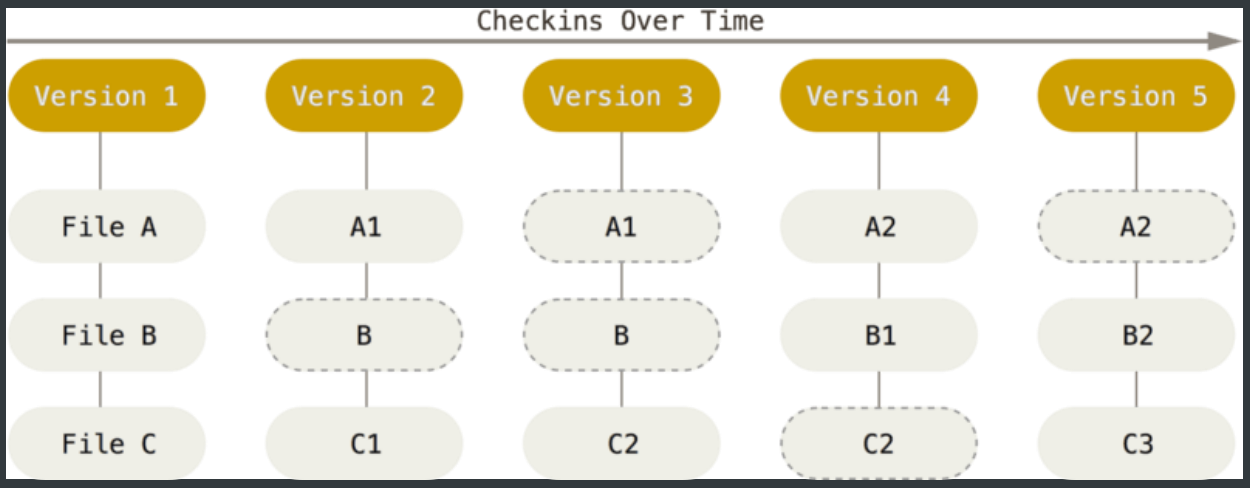
ХРАНЕНИЕ ИЗМЕНЕНИЙ В GIT

Основное отличие Git от других VCS — это подход к работе со своими данными. Концептуально, большинство других систем хранят информацию в виде списка изменений в файлах. Такие системы представляют хранимую информацию в виде набора файлов

и изменений, сделанных в каждом файле, по времени (обычно это называют контролем версий, *основанным на различиях*).



Подход Git к хранению данных больше похож на набор снимков файловой системы. Когда вы делаете коммит, система запоминает, как выглядит каждый файл в этот момент, и сохраняет ссылку на этот снимок. Для увеличения эффективности, если файлы не были изменены, Git только создаёт ссылку на предыдущую версию идентичного файла, который уже сохранён. Git представляет свои данные, как *поток снимков*.



Git-репозиторий хранится в директории `.git` в корне проекта.

```
.git
├── COMMIT_EDITMSG
├── FETCH_HEAD
├── HEAD
├── config
├── description
├── hooks
├── index
├── info
├── logs
│   ├── HEAD
│   └── refs
│       └── heads
├── objects
│   ├── 01
│   │   └── 2dc51bccc1
│   ├── 5a
│   │   └── 2669f47b80
│   ├── 7b
│   │   ├── bcb3154035
│   │   └── cf9b7d210c
│   ├── 9d
│   │   └── 4fc2f6782e
│   ├── f7
│   │   └── a9d2e60188
│   ├── fd
│   │   └── 96eb24c77b
│   ├── info
│   └── pack
└── refs
    ├── heads
    │   └── master
    └── tags
```

В репозитории содержатся:

- объекты коммитов
- ссылки на коммиты и ветки
- конфигурация
- скрипты-хуки

Ключевые элементы Git:

- каталог objects - база данных объектов Git
- каталог refs - ссылки на объекты коммитов в базе
- файл HEAD - указывает на текущую ветку
- файл index - хранит содержимое индекса

СТРУКТУРА ХРАНИМЫХ ДАННЫХ

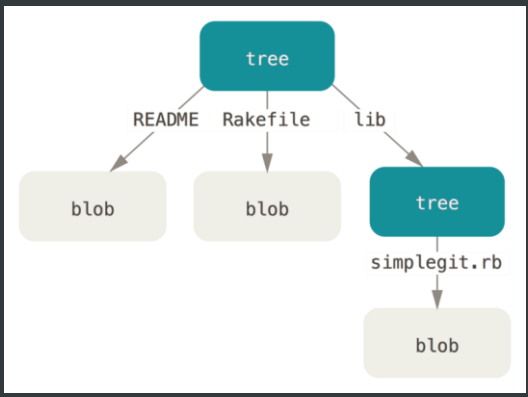
Концептуально, данные хранятся в Git примерно в виде дерева. Git создаёт дерево путём создания набора объектов из состояния области подготовленных файлов или индекса.

Посмотреть тип объекта можно командой:

```
$ git cat-file -t ${ХЭШ_ОБЪЕКТА}
```

Посмотреть содержимое объекта – командой:

```
$ git cat-file -p ${ХЭШ_ОБЪЕКТА}
```



BLOB

Blob - базовая единица хранения данных в Git. Хранит snapshot (снимок) содержимого файла. В качестве имени объекта берется SHA1 хеш, содержимого файла и заголовка.

```
$ git cat-file -t 8c7a1cf
blob
$ git cat-file -p 8c7a1cf
```

```
this is file.txt
second commit did this
$ cat file.txt
this is file.txt
second commit did this
```

TREE

Деревья содержат информацию о блобах, а также других поддеревьях. Деревья решают проблему хранения имён файлов, а также позволяют хранить группы файлов вместе.

Git хранит данные сходным с файловыми системами UNIX способом, но в немного упрощённом виде. Содержимое хранится в деревьях и блобах, где дерево соответствует каталогу на файловой системе, а блоб более или менее соответствует inode или содержимому файла.

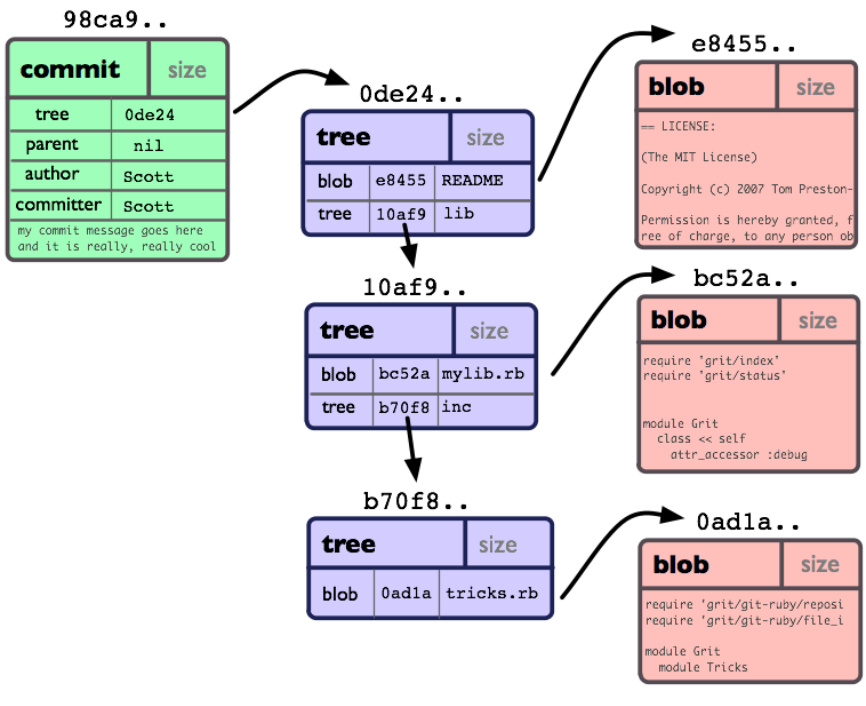
Дерево может содержать одну или более записей, содержащих SHA-1 хеш, соответствующий блобу или поддереву, права доступа к файлу, тип и имя файла.

COMMIT

В объекте коммита указывается дерево верхнего уровня, соответствующее состоянию проекта на некоторый момент; родительские коммиты, если существуют; имена автора и коммиттера (берутся из полей конфигурации user.name и user.email) с указанием временной метки; пустая строка и сообщение коммита.

```
$>tree
.
|-- README
`-- lib
    |-- inc
```

```
    |   `-- tricks.rb
    `-- mylib.rb
```



TAGS

Различают два типа:

- Легковесные теги - указатели на определенный коммит
- Аннотированные - содержат имя, email поставившего тег, дату, комментарий и могут иметь цифровую подпись. Хранятся как полноценные объекты.

Объект тега очень похож на объект коммита: он содержит имя своего автора, дату, сообщение и указатель. Разница же в том, что объект тега указывает на коммит, а не на дерево. Он похож на ветку, которая никогда не перемещается: он всегда указывает на один и тот же коммит, просто давая ему понятное имя.

```
$ git cat-file tag v1.5.0
object 437b1b20df4b356c9342dac8d38849f24ef44f27
type commit
tag v1.5.0
tagger Junio C Hamano <junkio@cox.net> 1171411200 +0000

GIT 1.5.0
-----BEGIN PGP SIGNATURE-----
```

```
Version: GnuPG v1.4.6 (GNU/Linux)

iD8DBQBF0lGqwMbZpPMRm5oRAuRiAJ9ohBLd7s2kqjkKlq1qqC57SbnmzQCdG4ui
nLE/L9aUXdWeTFPron96DLA=
=2E+0
-----END PGP SIGNATURE-----
```

РАБОТА С ПРОЕКТАМИ В GIT

УСТАНОВКА И КОНФИГУРАЦИЯ GIT

Установку Git в Linux просто можно выполнить из пакетов используемого в ОС пакетного менеджера.

```
$ sudo dnf install git-all
```

или

```
$ sudo apt install git
```

Больше информации об установке в разных ОС можно получить в документации.

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 места.

Файл **/etc/gitconfig** содержит значения, общие для всех пользователей системы и для всех их репозиториев. Если при запуске git config указать параметр --system, то параметры будут читаться и сохраняться именно в этот файл.

Файл **~/.gitconfig** или **~/.config/git/config** хранит настройки конкретного пользователя. Этот файл используется при указании параметра --global

Файл **config** в каталоге Git (т.е. **.git/config**) репозитория, который вы используете в данный момент, хранит настройки конкретного репозитория. Этот файл используется при указании параметра --local.

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

Первое, что следует сделать после установки Git — указать имя и адрес электронной почты пользователя. Каждый коммит в Git содержит эту информацию, она включена в коммиты и не может быть далее изменена:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Также желательно выбрать удобный текстовый редактор, который будет использоваться, если будет нужно набрать сообщение в Git. По умолчанию Git использует стандартный редактор системы, которым обычно является Vim.

```
$ git config --global core.editor “code –wait”
```

Чтобы проверить используемую конфигурацию, можно использовать команду `git config --list`, чтобы показать все настройки:

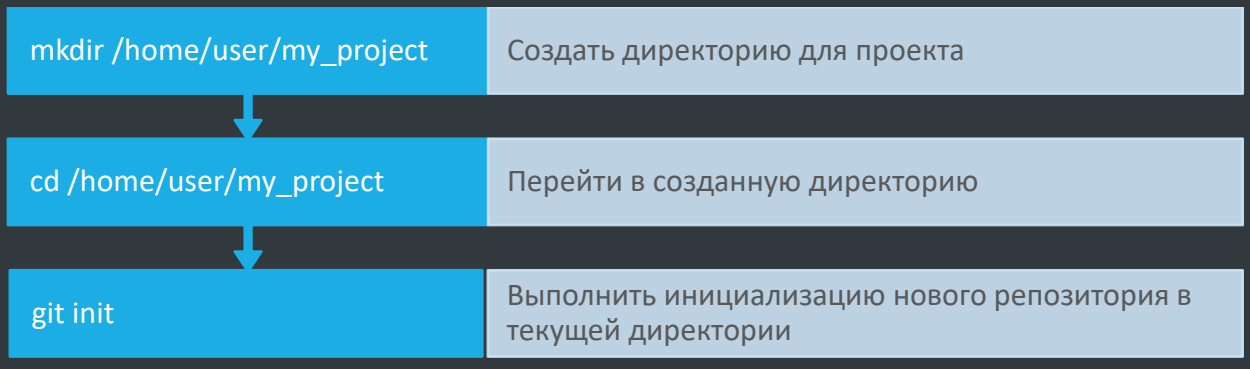
```
$ git config --list
user.name=John Doe

user.email=johndoe@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
```

Чтобы посмотреть для какого уровня конфигурации установлены настройки, необходимо выполнить команду `git config --list --show-origin`.

СОЗДАНИЕ РЕПОЗИТОРИЯ

Создать репозиторий можно одним из двух способов: инициализировать либо из существующих файлов, которые еще не находятся в системе управления версиями, либо из пустого каталога или клонировать уже существующий репозиторий.

ИНИЦИАЛИЗАЦИЯ НОВОГО РЕПОЗИТОРИЯ



Эта команда создаёт в текущем каталоге новый подкаталог с именем `.git`, содержащий все необходимые файлы репозитория — структуру Git репозитория. На этом этапе проект ещё не находится под версионным контролем.

Далее необходимо добавить под версионный контроль файлы. Для этого следует выполнить команду `git add`, указав индексируемые файлы, а затем выполнить `git commit`:

```
$ git add <file>
$ git add LICENSE
$ git commit -m 'Initial project version'
```

Теперь существует Git-репозиторий с отслеживаемыми файлами и начальным коммитом.

КЛОНИРОВАНИЕ СУЩЕСТВУЮЩЕГО РЕПОЗИТОРИЯ

Клонирование репозитория осуществляется командой *git clone <url>*.

Чтобы клонировать, созданный вами ранее репозиторий на Github выполните:

```
$ git clone https://github.com/<account_name>/<repo_name>
```

или, чтобы выполнить клонирование по ssh протоколу:

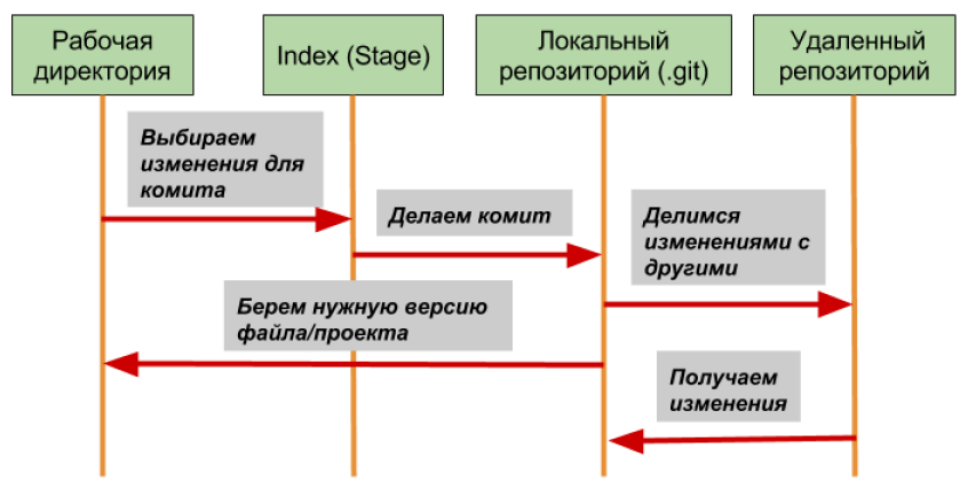
```
$ git clone git@github.com:<account_name>/<repo_name>.git
```

Эта команда создаёт в текущей директории директорию с именем вашего репозитория, 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 репозитория, то увидите в нём файлы проекта, готовые для работы или использования.

ПРОЦЕСС РАБОТЫ С GIT

В общем случае процесс работы с репозиторием выглядит следующим образом:

- изменение содержимого рабочей директории репозитория;
- частичное или полное добавление всего измененного объема, которое требуется зафиксировать в репозитории, в Index;
- выполнение фиксации изменений в репозитории (commit);
- отправка изменений из локального репозитория в удаленный репозиторий;
- получение изменений из удаленного репозитория;
- переключение рабочей директории на нужную версию (branch/commit).



О ВЕТВЛЕНИИ В GIT

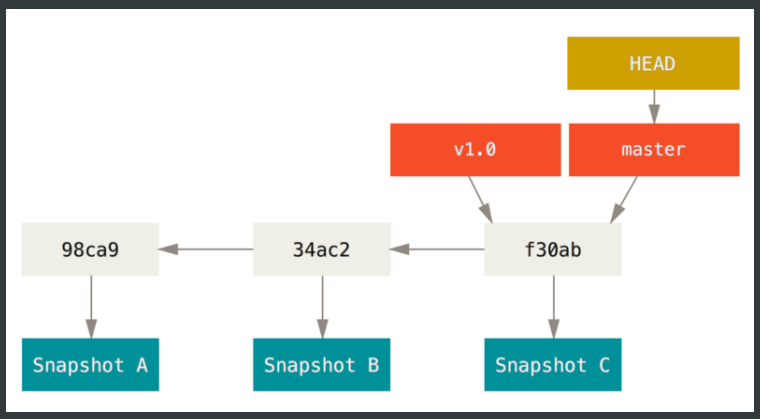
Используя ветвление, вы отклоняетесь от основной линии разработки и продолжаете работу независимо от неё, не вмешиваясь в основную линию. Во многих СКВ создание веток — это очень затратный процесс, часто требующий создания новой копии каталога с исходным кодом, что может занять много времени для большого проекта.

Ветвление Git очень легковесно: операция создания ветки и переключение между ветками выполняются очень быстро. Git рекомендует процесс работы, при котором ветвление и слияние выполняется часто.

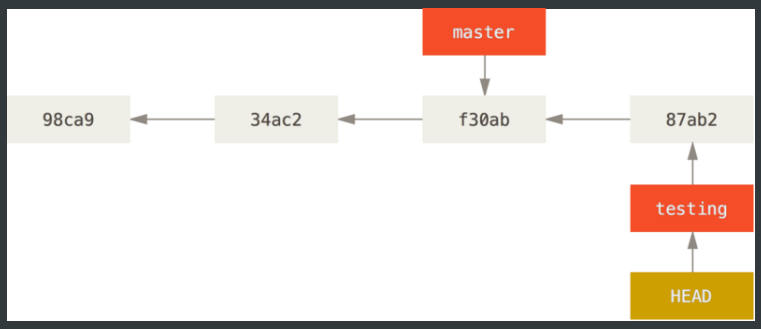
Когда вы делаете коммит, Git сохраняет его в виде объекта, который содержит указатель на снимок (snapshot) подготовленных данных. Ветка в Git — это простой перемещаемый указатель на один из таких коммитов. По умолчанию, имя основной ветки в Git — `master`. Как только вы начнёте создавать коммиты, ветка `master` будет всегда указывать на последний коммит. Каждый раз при создании коммита указатель ветки `master` будет передвигаться на следующий коммит автоматически.

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

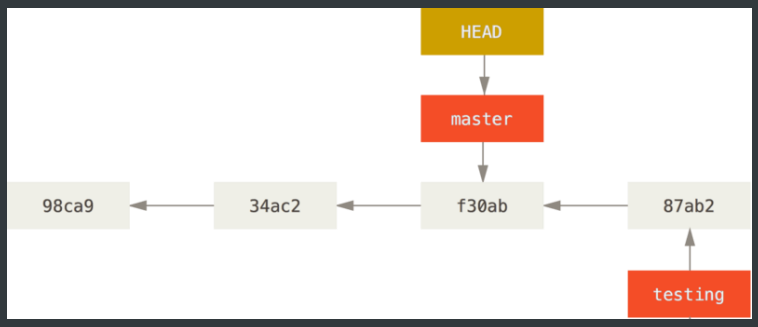
Для определения текущей ветки Git хранит специальный указатель `HEAD` — это указатель на текущую локальную ветку.



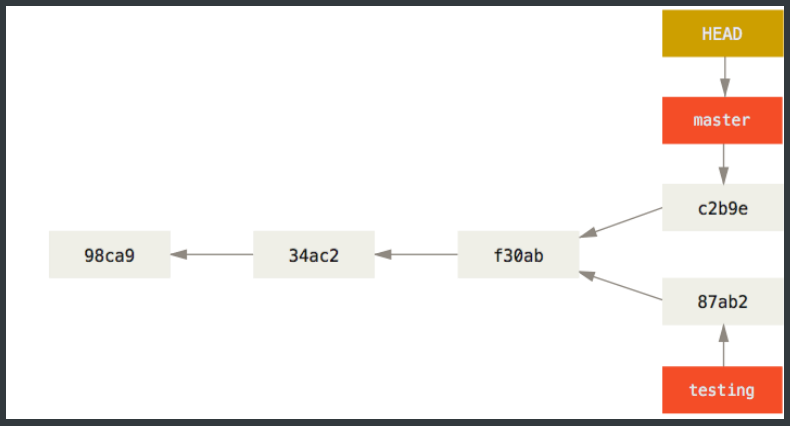
При переключении в другую ветку (например `testing`) и добавлении в нее новых коммитов указатель на ветку `testing` перемещается вперед, а `master` указывает на тот же коммит, где вы были до переключения веток.



После обратного переключения в ветку master указатель `HEAD` перемещается назад на ветку `master` и файлы в рабочем каталоге возвращаются в то состояние, на снимок которого указывает `master`.



Добавление очередного коммита в ветку `master` приведет к разветвлению истории. Изменения в различных ветках изолированы друг от друга: вы можете свободно переключаться туда и обратно, а когда понадобится — объединить их.

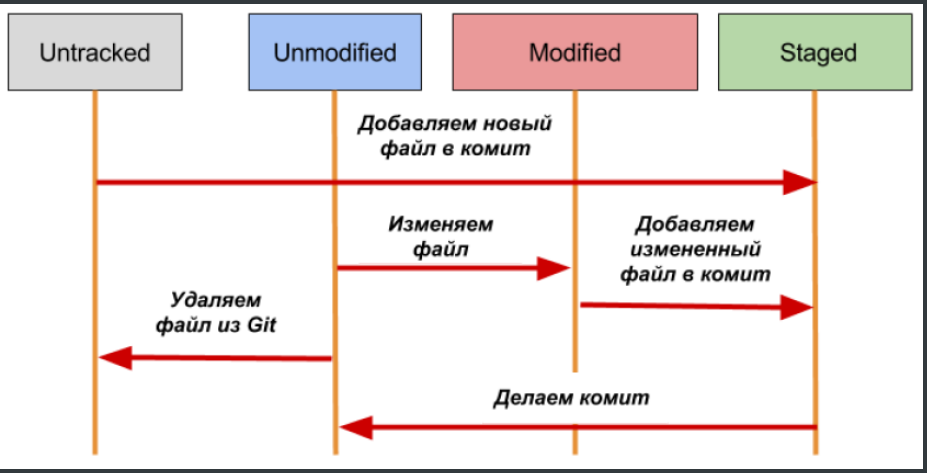


ОСНОВНЫЕ КОМАНДЫ GIT

ЗАПИСЬ ИЗМЕНЕНИЙ В РЕПОЗИТОРИЙ

У Git есть три основных состояния, в которых могут находиться файлы:

- зафиксированное (unmodified, commited) - означает, что файл уже сохранён в локальной базе репозитория и не изменялся;
- изменённое (modified) - файл, который изменялся, но ещё не был зафиксированы;
- подготовленное (staged) - файл, отмеченный для включения в следующий коммит для фиксации в репозитории.



ОПРЕДЕЛЕНИЕ СОСТОЯНИЯ ФАЙЛОВ

Основной инструмент, используемый для определения, какие файлы в каком состоянии находятся — это команда `git status`. Если выполнить команду сразу после клонирования, можно увидеть 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 в рабочем каталоге нет отслеживаемых измененных файлов. Git также не обнаружил неотслеживаемых файлов, в противном случае они бы были перечислены здесь.

Добавим в проект новый простой файл `README`. При выполнении `git status` увидим неотслеживаемый файл:

```
$ 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)
```

Статус `Untracked` означает, что Git видит файл, которого не было в предыдущем коммите; Git не станет добавлять его в новые коммиты, пока явно об этом не попросить. Это предохранит вас от случайного добавления в репозиторий сгенерированных бинарных файлов или каких-либо других, которые вы и не думали добавлять.

ОТСЛЕЖИВАНИЕ НОВЫХ ФАЙЛОВ

Для того чтобы начать отслеживать новый файл, используется команда `git add`. Чтобы начать отслеживание файла `README`, необходимо выполнить:

```
$ git add README
```

Если теперь выполнить команду `status`, можно увидеть, 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
```

Команда `git add` принимает параметром путь к файлу или каталогу, если это каталог, команда рекурсивно добавляет все файлы из указанного каталога в индекс.

ИНДЕКСАЦИЯ ИЗМЕНЁННЫХ ФАЙЛОВ

Если изменить файл, который уже находящийся под версионным контролем, например, отслеживаемый файл `CONTRIBUTING.md`, и после этого снова выполнить команду `git status`, то результат будет примерно следующим:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
```

```
    new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)

    modified:   CONTRIBUTING.md
```

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

ИГНОРИРОВАНИЕ ФАЙЛОВ

Иногда может иметься группа файлов, которую не следует автоматически добавлять в репозиторий и отображать в списках неотслеживаемых. К таким файлам могут относиться автоматически генерируемые файлы (логи, результаты сборки программ и т. п.) или какие-то секретные данные (пароли, ключи и т.п.). Для этого необходимо создать файл `.gitignore`. с перечислением шаблонов соответствующих таким файлам.

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.

Пример файла `.gitignore`:

```
# Исключить все файлы с расширением .a
```

```
*.a

# Не 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

# 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
```

КОММИТ ИЗМЕНЕНИЙ

Теперь, когда индекс находится в требуемом состоянии, можно зафиксировать изменения. Если есть любые файлы, созданные или изменённые, и для которых не выполнено `git add` после редактирования — не войдут в коммит. Они останутся изменёнными файлами на диске. Простейший способ зафиксировать изменения —выполнить `git commit`:

```
$ git commit
```

Эта команда откроет выбранный в конфигурации текстовый редактор для того, чтобы указать комментарий для коммита. Комментарий по умолчанию для коммита содержит закомментированный результат работы команды `git status` и ещё одну пустую строку сверху. Можно удалить эти комментарии и набрать другое сообщение или оставить их для напоминания о том, что зафиксировано. После выхода из редактора, Git создаст коммит с этим сообщением, удаляя комментарии и вывод команды `diff`.

Другой способ - набрать комментарий к коммиту в командной строке вместе с командой `commit` указав его после параметра `-m`:

```
$ 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
```

Коммит сохраняет снимок состояния индекса. Всё, что не проиндексировано, так и останется в рабочем каталоге как изменённое; можно сделать следующий коммит, чтобы добавить эти изменения в репозиторий. Каждый раз, когда выполняется коммит, сохраняется снимок состояния проекта, который позже можно восстановить или с которым можно сравнить текущее состояние.

ПРОСМОТР ИСТОРИИ КОММИТОВ

Одним из основных и наиболее мощных инструментов для просмотра истории коммитов является команда `git log`. Если выполнить команду, можно увидеть примерно следующий вывод:

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

    Change version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    Remove unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    Initial commit
```

По умолчанию `git log` перечисляет коммиты, сделанные в репозитории в обратном к хронологическому порядке — последние коммиты находятся вверху. Коммиты перечисляются с указанием SHA-1 контрольной суммы, 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, внесенную в каждый коммит. Так же вы можете ограничить количество записей в выводе команды; используйте параметр `-1` для вывода только одной записи:

```
$ git log -p -1
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    Change version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
```

```
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s|
     s.platform  =   Gem::Platform::RUBY
     s.name      =   "simplegit"
-    s.version   =   "0.1.0"
+    s.version   =   "0.1.1"
     s.author    =   "Scott Chacon"
     s.email     =   "schacon@gee-mail.com"
     s.summary   =   "A simple gem for using Git in Ruby code."
```

СТАТИСТИКА LOG

Чтобы увидеть сокращенную статистику для каждого коммита, можно использовать опцию `--stat`. Это печатает под каждым из коммитов список и количество измененных файлов, а также сколько строк в каждом из файлов было добавлено и удалено.

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

    Change version number

 Rakefile | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

ФОРМАТИРОВАНИЕ LOG

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

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

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

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 6 years ago : Change version number
085bb3b - Scott Chacon, 6 years ago : Remove unnecessary test
a11bef0 - Scott Chacon, 6 years ago : Initial commit
```

Обо всех опциях для `format` можно прочитать в документации.

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

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 Ignore errors from SIGCHLD on trap
*  5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
|\
| * 420eac9 Add method for getting the current branch
* | 30e367c Timeout code and tests
* | 5a09431 Add timeout protection to grit
* | e1193f8 Support for heads with slashes in them
|/
* d6016bc Require time for xmlschema
*  11d191e Merge branch 'defunkt' into local
```

РАБОТА С УДАЛЁННЫМИ РЕПОЗИТОРИЯМИ

ПРОСМОТР УДАЛЁННЫХ РЕПОЗИТОРИЕВ

Для того, чтобы просмотреть список настроенных удалённых репозиториев, можно запустить команду `git remote`. Она выведет названия доступных удалённых репозиториев. Если ранее репозиторий был клонирован, то должен быть отображен, как минимум `origin` — имя по умолчанию, которое Git даёт серверу, с которого производилось клонирование:

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

Команда с ключом `-v` выведет адреса для чтения и записи, привязанные к репозиторию:

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

ДОБАВЛЕНИЕ УДАЛЁННЫХ РЕПОЗИТОРИЕВ

Чтобы добавить удалённый репозиторий и присвоить ему имя (shortname), необходимо выполнить команду `git remote add <shortname> <url>`:

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

Теперь вместо указания полного пути можно использовать `pb`. Например, если необходимо получить изменения, которые есть у Пола, можно выполнить команду `git fetch pb`:

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

Ветка `master` из репозитория Пола сейчас доступна вам под именем `pb/master`. Вы можете слить её с одной из ваших веток или переключить на неё локальную ветку, чтобы просмотреть содержимое ветки Пола.

ПОЛУЧЕНИЕ ИЗМЕНЕНИЙ ИЗ УДАЛЁННОГО РЕПОЗИТОРИЯ

Для получения данных из удалённых проектов, следует выполнить:

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

Данная команда связывается с указанным удалённым проектом и забирает все те данные проекта, которых нет локально. После ее выполнения должны появиться ссылки на все ветки из удалённого проекта, которые можно просмотреть или слить в любой момент.

Команда `clone` автоматически добавляет удалённый репозиторий под именем «origin». Таким образом, `git fetch origin` извлекает все наработки, отправленные на этот сервер после того, как вы его клонировали (или получили изменения с помощью fetch). Команда `git fetch` забирает данные в локальный репозиторий, но не сливает их с какими-либо локальными наработками и не модифицирует их. Необходимо вручную выполнить слияние.

Если ветка настроена на отслеживание удалённой ветки, то вы можно использовать команду git pull чтобы автоматически получить изменения из удалённой ветки и слить их с локальной текущей. Этот способ может оказаться более простым или более удобным. К тому же, по умолчанию команда `git clone` автоматически настраивает локальную

ветку `master` на отслеживание удалённой ветки `master` на сервере, с которого клонировали репозиторий. Название веток может быть другим и зависит от ветки по умолчанию на сервере. Выполнение `git pull`, как правило, извлекает (fetch) данные с сервера, с которого изначально клонировали, и автоматически пытается слить (merge) их с кодом, над которым в данный момент идет работа.

ОТПРАВКА ИЗМЕНЕНИЙ В УДАЛЕННЫЙ РЕПОЗИТОРИЙ

Чтобы поделиться своими наработками, необходимо отправить их в удалённый репозиторий. Команда для этого действия: `git push <remote-name> <branch-name>`. Чтобы отправить локальную ветку master на сервер `origin`, можно выполнить следующую команду для отправки локальных коммитов:

```
$ git push origin master
```

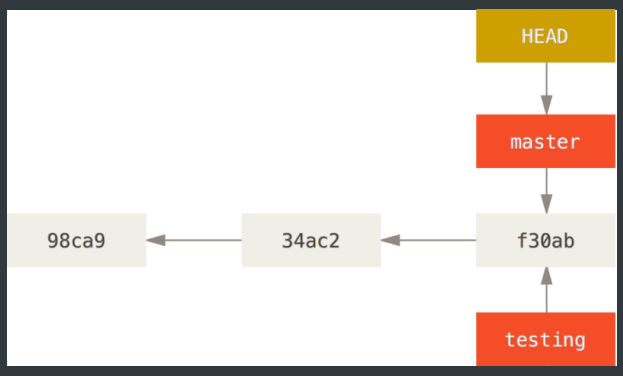
Эта команда срабатывает только в случае, если у пользователя есть права на запись в удаленном репозитории, и, если никто другой с момента клонирования репозитория не выполнял команду `push`. В противном случае необходимо сначала получить изменения из удаленного репозитория и объединить их с локальными и после этого выполнить `push`.

СОЗДАНИЕ НОВОЙ ВЕТКИ

Для создания новой ветки, например, с именем `testing`, можно выполнить команду:

```
$ git branch testing
```

В результате создаётся новый указатель на текущий коммит. Но команда `git branch` только создаёт новую ветку, но не переключает на неё.



Можно это увидеть при помощи команды `git log`, которая покажет, куда указывают указатели веток. Эта опция называется `--decorate`.

```
$ git log --oneline --decorate
f30ab (HEAD -> master, testing) Add feature #32 - ability to add new
formats to the central interface
```

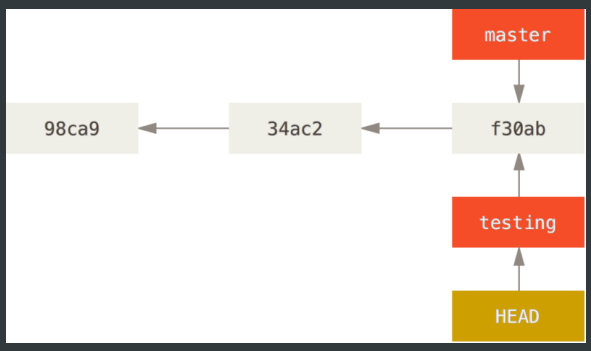
```
34ac2 Fix bug #1328 - stack overflow under certain conditions
98ca9 Initial commit
```

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`.



Для создания новой ветки и одновременного переключения на нее можно выполнить команду `checkout` с ключом `-b`:

```
$ git checkout -b testing
```

РАБОТА С ВЕТКАМИ И СЛИЯНИЕ В GIT

ОСНОВЫ ВЕТВЛЕНИЯ И СЛИЯНИЯ

Рассмотрим простой пример рабочего процесса, который может быть полезен в проекте:

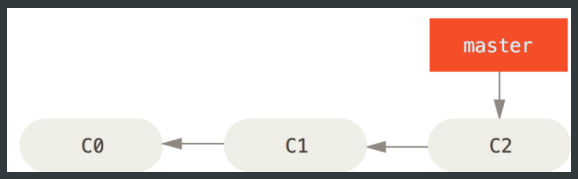
- Вы работаете над сайтом.
- Вы создаете ветку для новой статьи, которую вы пишете.
- Вы работаете в этой ветке.

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

- Переключиться на основную ветку.
- Создать ветку для добавления исправления.
- После тестирования слить ветку, содержащую исправление, с основной веткой.
- Переключиться назад в ту ветку, где вы пишете статью и продолжить работать.

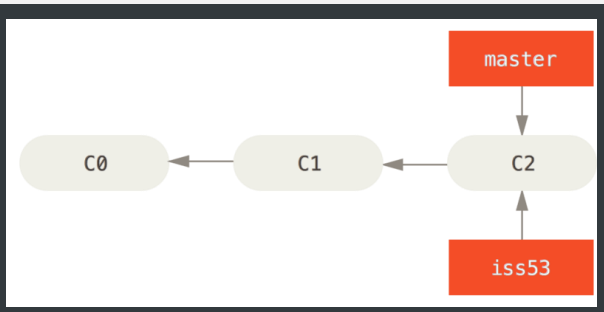
ОСНОВЫ ВЕТВЛЕНИЯ

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

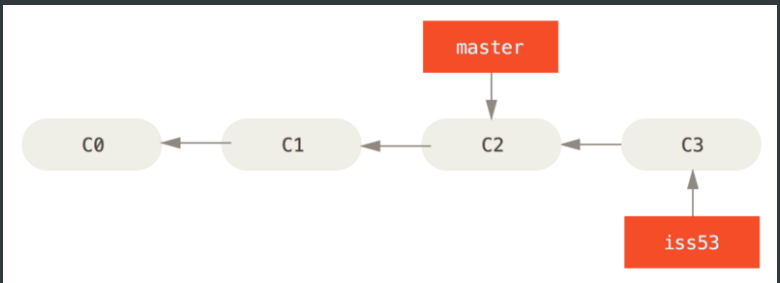


Вы решаете, что теперь вы будете заниматься проблемой #53 из вашей системы отслеживания ошибок. Чтобы создать ветку и сразу переключиться на нее, можно выполнить команду `git checkout` с параметром `-b`:

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



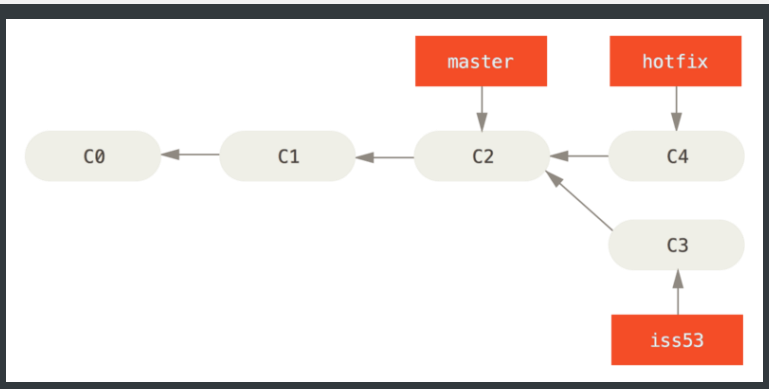
Вы работаете над своим сайтом и делаете коммиты. Это приводит к тому, что ветка `iss53` движется вперед, так как вы переключились на нее ранее (`HEAD` указывает на нее).



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`. Все, что вам нужно — зафиксировать все свои изменения в ветке `iss53` переключиться на ветку `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. 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'
[hotfix 1fb7853] Fix broken email address
 1 file changed, 2 insertions(+)
```



После выполнения работы над исправлением можно выполнить слияние ветки `hotfix` с веткой `master` для включения изменений в продукт. Это делается командой `git merge`:

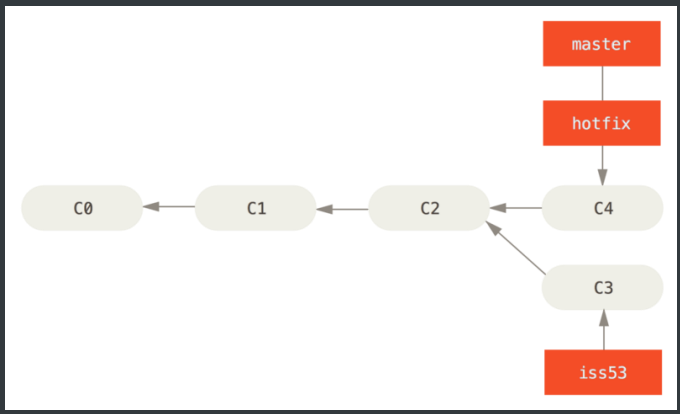
```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
```

```
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

Фраза «fast-forward» в выводе команды означает, что 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.

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*».

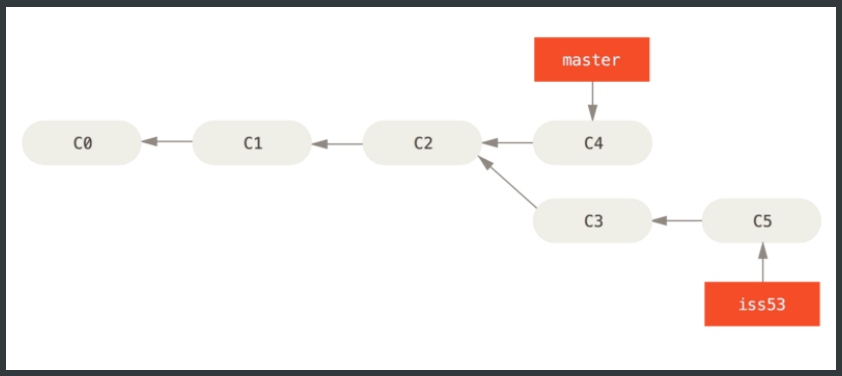
Теперь ваши изменения включены в коммит, на который указывает ветка `master`, и исправление можно внедрять.



После внедрения 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 можно переключиться обратно на ветку `iss53` и продолжить работу.



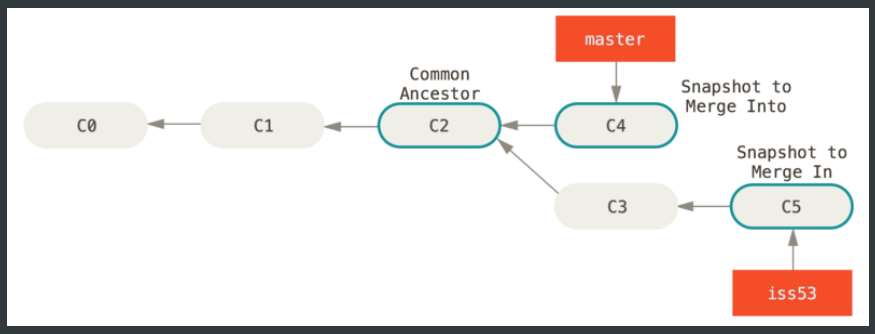
Стоит обратить внимание на то, что все изменения из ветки `hotfix` не включены в вашу ветку `iss53`. Если их нужно включить, вы можете влить ветку `master` в вашу ветку `iss53` командой `git merge master`, или же вы можете отложить слияние этих изменений до завершения работы, и затем влить ветку `iss53`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

ОСНОВЫ СЛИЯНИЯ

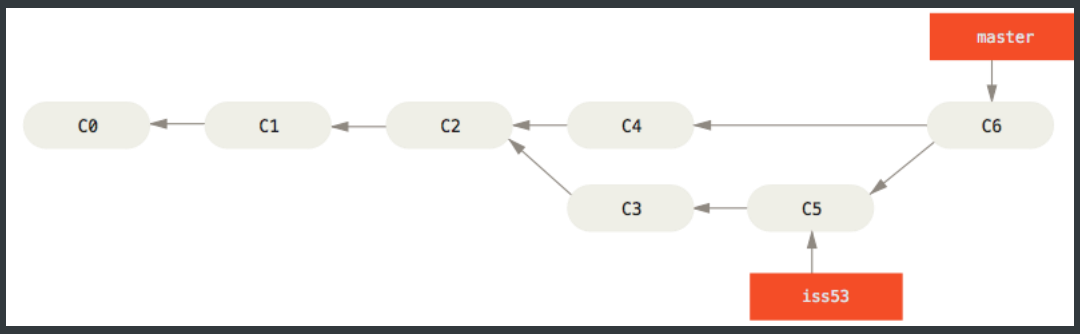
Когда работа по проблеме #53 закончена и её можно влить в ветку `master`. Для этого нужно выполнить слияние ветки `iss53` точно так же, как с веткой `hotfix` ранее:

```
$ 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 выполняет простое трёхстороннее слияние, используя последние коммиты объединяемых веток и общего для них родительского коммита.



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.



ОСНОВНЫЕ КОНФЛИКТЫ СЛИЯНИЯ

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

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

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

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

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

Всё, где есть неразрешённые конфликты слияния, перечисляется как `unmerged`. 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.

Разрешив каждый конфликт во всех файлах и удалив строки `<<<<<<<`, `=======` и `>>>>>>>`, запустите `git add` для каждого файла, чтобы отметить

конфликт, как решённый. Добавление файла в индекс означает для Git, что все конфликты в нём исправлены.

Убедившись, что все файлы, где были конфликты, добавлены в индекс — выполните команду `git commit` для создания коммита слияния. Комментарий к коммиту слияния по умолчанию выглядит примерно так:

```
Merge branch 'iss53'

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


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

УДАЛЁННЫЕ ВЕТКИ

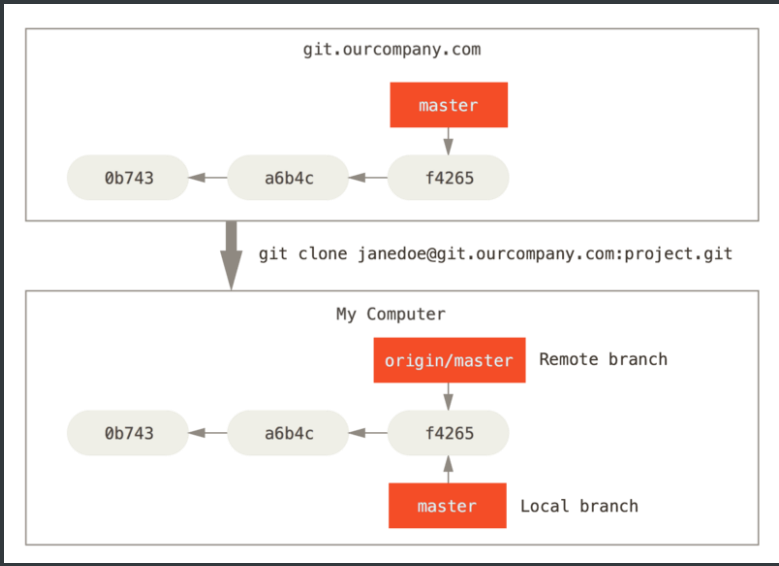
Удалённые ссылки — это ссылки (указатели) в ваших удалённых репозиториях, включая ветки, теги и так далее. Полный список удалённых ссылок можно получить с помощью команды `git ls-remote <remote>` или команды `git remote show <remote>` для получения удалённых веток и дополнительной информации. Тем не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

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

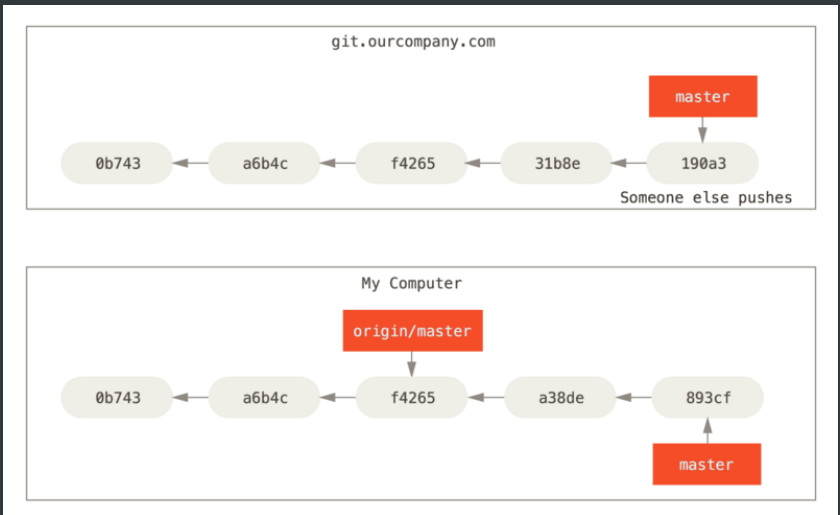
Имена веток слежения имеют вид `<remote>/<branch>`. Например, если вы хотите посмотреть, как выглядела ветка `master` на сервере `origin` во время последнего соединения с ним, используйте ветку `origin/master`. Если вы с коллегой работали над

одной задачей и он отправил на сервер ветку `iss53`, при том, что у вас может быть своя локальная ветка `iss53`, удалённая ветка будет представлена веткой слежения с именем `origin/iss53`.

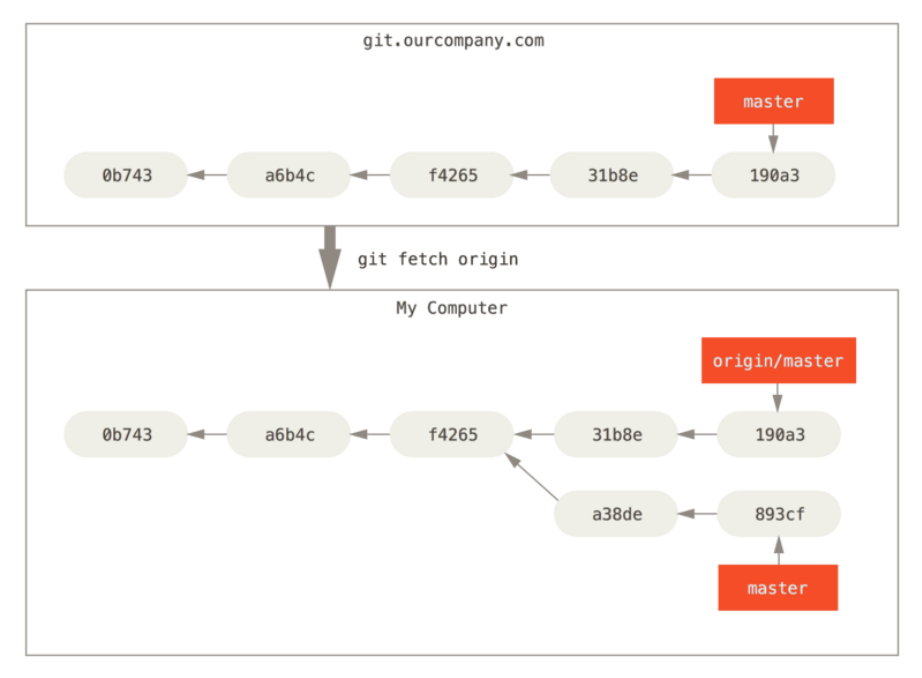
Предположим, у вас в сети есть Git-сервер с адресом `git.ourcompany.com`. Если вы с него что-то клонируете, команда `clone` автоматически назовёт его `origin`, заберёт оттуда все данные, создаст указатель на то, на что там указывает ветка `master`, и назовёт его локально `origin/master`. Git также создаст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



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



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



ОТПРАВКА ИЗМЕНЕНИЙ

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

Если у вас есть ветка `serverfix`, над которой вы хотите работать с кем-то ещё выполните команду `git push <remote> <branch>`:

```
$ git push origin serverfix
```

Git автоматически разворачивает имя ветки `serverfix` до `refs/heads/serverfix:refs/heads/serverfix`, что означает «возьми мою локальную ветку `serverfix` и обнови ей удалённую ветку `serverfix`». Вы также можете выполнить `git push origin serverfix:serverfix` — произойдёт то же самое.

Можно использовать этот формат для отправки локальной ветки в удалённую ветку с другим именем: `git push origin serverfix:awesomebranch`.

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

```
$ git fetch origin
...
From https://github.com/schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

При получении данных создаются ветки слежения, вы не получаете автоматически для них локальных редактируемых копий. В нашем случае вы не получите новую ветку `serverfix` — только указатель `origin/serverfix`, который вы не можете изменять.

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

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

ОТСЛЕЖИВАНИЕ ВЕТОК

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

При клонировании репозитория, как правило, автоматически создаётся ветка `master`, которая следит за `origin/master`. Вы можете настроить отслеживание и других веток — следить за ветками на других серверах или отключить слежение за веткой `master`.

Если вы пытаетесь извлечь ветку, которая не существует локально, но существует только одна удалённая ветка с точно таким же именем, то Git автоматически создаст ветку слежения:

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

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

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

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

Чтобы посмотреть, как у вас настроены ветки слежения, воспользуйтесь опцией `-vv` для команды `git branch`:

```
$ git branch -vv
  iss53     7e424c3 [origin/iss53: ahead 2] Add forgotten brackets
  master    1ae2a45 [origin/master] Deploy index fix
  testing   5ea463a Try something new
```

Здесь видно, что:

- ветка `iss53` следит за `origin/iss53` и «опережает» её на два изменения — это значит, что есть два локальных коммита, которые не отправлены на сервер;
- ветка `master` отслеживает ветку `origin/master` и ее состояние актуально;
- ветка `testing` не отслеживает удаленную ветку.

Важно отметить, что эти цифры описывают состояние на момент последнего получения данных.

ПОЛУЧЕНИЕ ИЗМЕНЕНИЙ

Команда `git fetch` получает с сервера все изменения, которых у вас ещё нет, но не изменяет состояние рабочей копии. Эта команда просто получает данные и позволяет вам самостоятельно сделать слияние.

Команда `git pull` выполняет команду `git fetch`, а после `git merge`. Если 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.

УДАЛЕНИЕ ВЕТОК НА УДАЛЁННОМ СЕРВЕРЕ

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
```

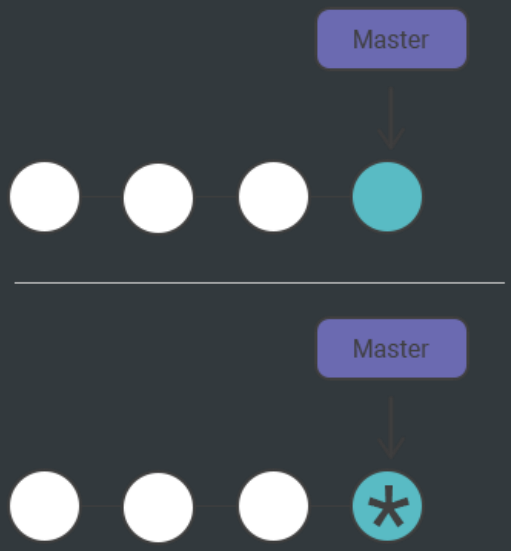
Команда удаляет указатель на сервере. Как правило, Git сервер хранит данные пока не запустится сборщик мусора, поэтому если ветка была удалена случайно, её можно восстановить.

РАБОТА С ИСТОРИЕЙ В GIT

ПЕРЕПИСЫВАНИЕ ИСТОРИИ В GIT

ИЗМЕНЕНИЕ ПОСЛЕДНЕГО КОММИТА

Команда `git commit --amend` — это удобный способ изменить последний коммит. Она позволяет объединить проиндексированные изменения с предыдущим коммитом без создания нового коммита.



Изменение сообщения коммита:

```
$ git commit --amend -m "an updated commit message"
```

Изменение файлов в коммите без изменения сообщения:

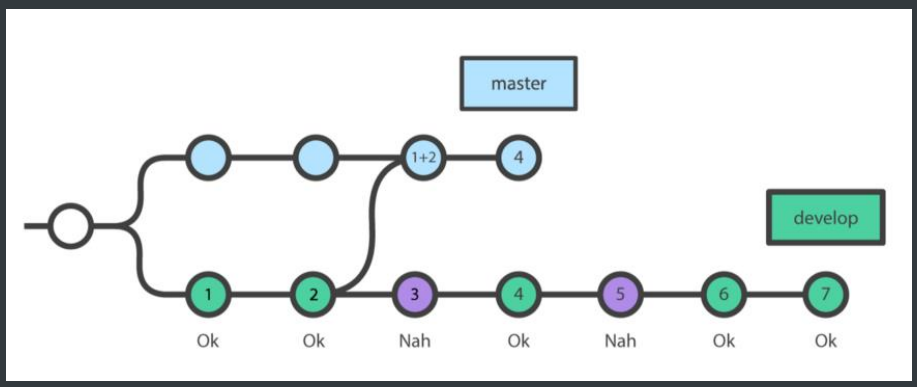
```
$ git commit --amend --no-edit
```

Не используйте `amend` для публичных коммитов. Измененные коммиты по сути являются новыми коммитами. При этом предыдущий коммит не останется в текущей ветке. Не изменяйте коммит, после которого уже начали работу другие разработчики.

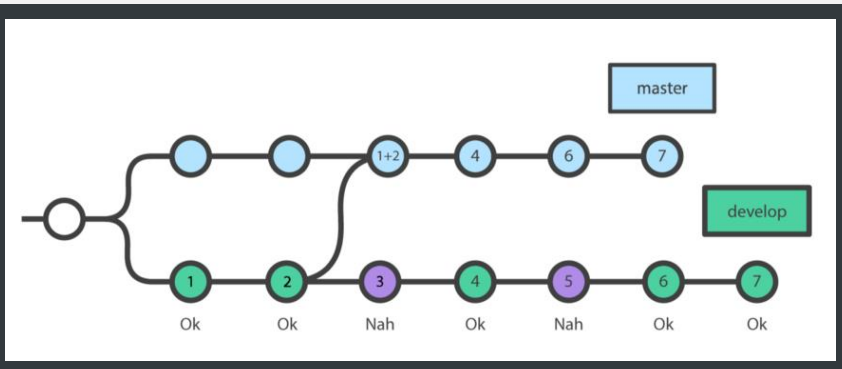
ПРИМЕНЕНИЕ ИЗМЕНЕНИЙ ИЗ СУЩЕСТВУЮЩИХ КОММИТОВ

Для перемещения изменений из одной ветки в другую применяется отбор коммитов (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 внесенными в нее изменениями.

```
$ git cherry-pick commit4
```




```
git cherry-pick commit5..develop
```



Обратите внимание, что в качестве начала указан commit5, а не commit6. Это потому, что диапазоны в Git начинаются с предыдущего коммита.

ИЗМЕНЕНИЕ НЕСКОЛЬКИХ КОММИТОВ

В Git отсутствуют инструменты для переписывания истории, но вы можете использовать команду `rebase`, чтобы перебазировать группу коммитов туда же на HEAD, где они были изначально, вместо перемещения их в другое место. С помощью интерактивного режима `-i` команды `rebase`, вы можно останавливаться после каждого коммита и изменять сообщения, добавлять файлы или делать что-то другое, что вам нужно.

Необходимо указать, какие коммиты вы хотите изменить, указав коммит, на который нужно выполнить перебазирование. Если вы хотите изменить сообщения последних трёх коммитов:

```
$ git rebase -i HEAD~3
```

Каждый коммит, входящий в диапазон `HEAD~3..HEAD`, будет изменён вне зависимости от того, изменили вы сообщение или нет. **Не включайте в такой диапазон коммит, который уже был отправлен на центральный сервер: сделав это, вы можете запутать других разработчиков, предоставив вторую версию одних и тех же изменений.**

Выполнение этой команды отобразит в вашем текстовом редакторе список коммитов, в нашем случае, например, следующее:

```
pick f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log
message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --
continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to
bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Обратите внимание на обратный порядок комитов. Команда `rebase` в интерактивном режиме предоставит вам скрипт, который она будет выполнять. Она начнет с коммита, который вы указали в командной строке (`HEAD~3`) и повторит изменения, внесённые каждым из коммитов, сверху вниз. Наверху отображается самый старый коммит, а не самый новый, потому что он будет повторен первым.

Вам необходимо изменить скрипт так, чтобы он остановился на коммите, который вы хотите изменить. Для этого измените слово `pick` на слово `edit` напротив каждого из коммитов, после которых скрипт должен остановиться. Например, для изменения сообщения только третьего коммита, измените файл следующим образом:

```
edit f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
```

```
pick a5f4a0d Add cat-file
...
```

Когда вы сохраните сообщение и выйдете из редактора, Git переместит вас к самому раннему коммиту из списка и вернёт вас в командную строку со следующим сообщением:

```
$ git rebase -i HEAD~3
Stopped at f7f3f6d... Change my name a bit
You can amend the commit now, with

       git commit --amend

Once you're satisfied with your changes, run

       git rebase --continue
```

Эти инструкции говорят, что нужно сделать. Выполните:

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

Измените сообщение коммита и выйдите из редактора. Затем выполните:

```
$ git rebase --continue
```

Эта команда автоматически применит два оставшиеся коммита и завершится. Если вы измените `pick` на `edit` в других строках, то можете повторить эти шаги для соответствующих коммитов. Каждый раз Git будет останавливаться, позволяя вам исправить коммит, и продолжит, когда вы закончите.

УПОРЯДОЧИВАНИЕ И УДАЛЕНИЕ КОММИТОВ

Вы также можете использовать интерактивное перебазирование для изменения порядка или полного удаления коммитов. Если вы хотите удалить коммит «Add cat-file» и изменить порядок, в котором были внесены два оставшихся, то вы можете изменить скрипт перебазирования с такого:

```
pick f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file
```

на такой:

```
pick 310154e Update README formatting and add blame
pick f7f3f6d Change my name a bit
```

Когда вы сохраните скрипт и выйдете из редактора, Git переместит вашу ветку на родителя этих коммитов, применит `310154e`, затем `f7f3f6d` и после этого остановится. Вы, фактически, изменили порядок этих коммитов и полностью удалили коммит «Add cat-file».

ОБЪЕДИНЕНИЕ КОММИТОВ

С помощью интерактивного режима команды `rebase` также можно объединить несколько коммитов в один. Git добавляет полезные инструкции в сообщение скрипта перебазирования:

```
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log
message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --
continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to
bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Если вместо «pick» или «edit» вы укажете «squash», Git применит изменения из текущего и предыдущего коммитов и предложит вам объединить их сообщения. Таким образом, если вы хотите из этих трёх коммитов сделать один, вы должны изменить скрипт следующим образом:

```
pick f7f3f6d Change my name a bit
squash 310154e Update README formatting and add blame
squash a5f4a0d Add cat-file
```

Когда вы сохраните скрипт и выйдете из редактора, Git применит изменения всех трёх коммитов и затем вернёт вас обратно в редактор, чтобы вы могли объединить сообщения коммитов:

```
# This is a combination of 3 commits.
# The first commit's message is:
Change my name a bit
```

```
# This is the 2nd commit message:

Update README formatting and add blame

# This is the 3rd commit message:

Add cat-file
```

После сохранения сообщения, вы получите один коммит, содержащий изменения всех трёх коммитов, существовавших ранее.

ПРОДВИНУТЫЙ ИНСТРУМЕНТ: FILTER-BRANCH

Существует ещё один способ переписывания истории, который вы можете использовать при необходимости изменить большое количество коммитов каким-то программируемым способом — например, удалить файл из всех коммитов. Для этого используется команда `filter-branch`.

УДАЛЕНИЕ ФАЙЛА ИЗ КАЖДОГО КОММИТА

Такое случается довольно часто. Кто-нибудь случайно зафиксировал огромный бинарный файл, неосмотрительно выполнив `git add .`, и вы хотите отовсюду его удалить. Возможно, вы случайно зафиксировали файл, содержащий пароль, а теперь хотите сделать ваш проект общедоступным. В общем, утилиту `filter-branch` вы, вероятно, захотите использовать, чтобы привести к нужному виду всю вашу историю.

Для удаления файла `passwords.txt` из всей вашей истории вы можете использовать опцию `--tree-filter` команды `filter-branch`:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

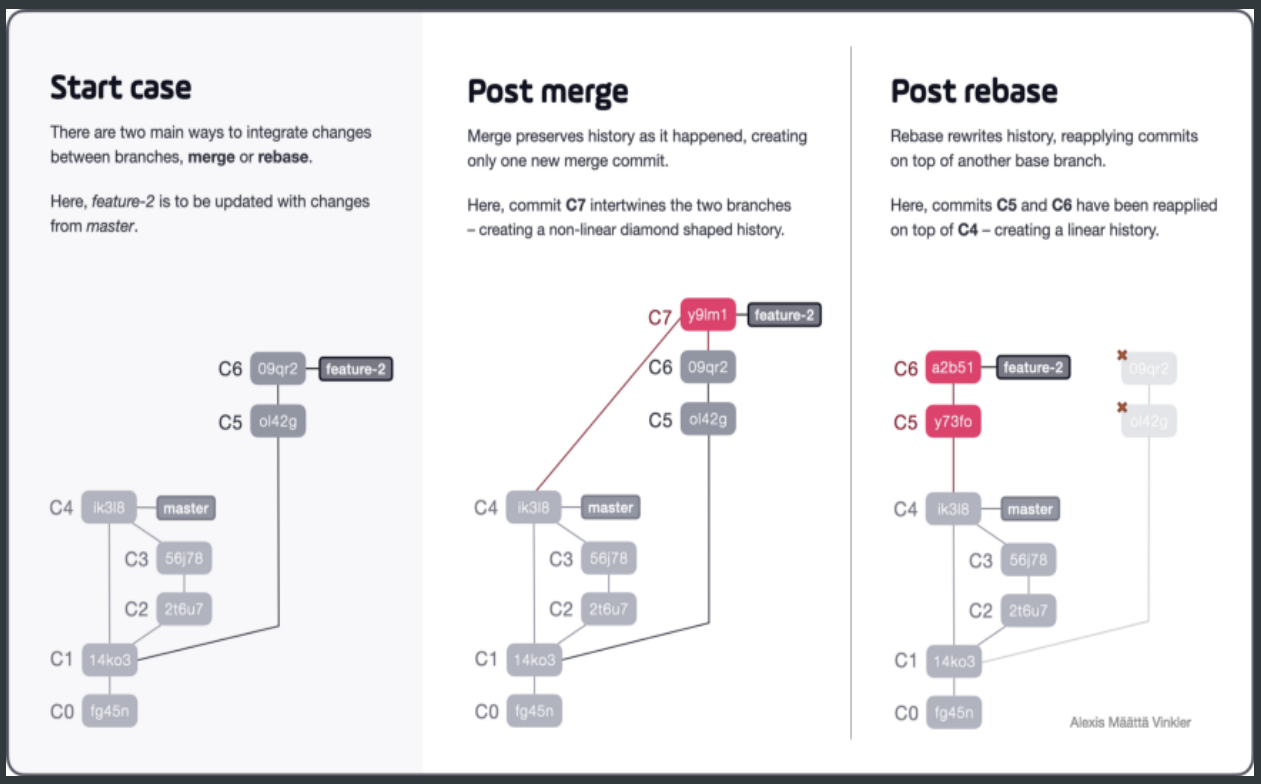
Опция `--tree-filter` выполняет указанную команду после переключения на каждый коммит и затем повторно фиксирует результаты. В данном примере, вы удаляете файл `passwords.txt` из каждого снимка вне зависимости от того, существует он или нет. Если вы хотите удалить все случайно зафиксированные резервные копии файлов, созданные текстовым редактором, то вы можете выполнить нечто подобное `git filter-branch --tree-filter 'rm -f *~' HEAD`.

Вы можете посмотреть, как Git изменит деревья и коммиты, а затем уже переместить указатель ветки. Как правило, хорошим подходом будет выполнение всех этих действий в тестовой ветке и, после проверки полученных результатов, установка на неё указателя

основной ветки. Для выполнения `filter-branch` на всех ваших ветках, вы можете передать команде опцию `--all`.

ОТЛИЧИЯ МЕЖДУ GIT MERGE И GIT REBASE

Когда над проектом работает несколько человек, в какой-то момент нужно объединить код. Команды `rebase` и `merge` имеют разные подходы для консолидации изменений из одной ветви в другую.



GIT MERGE

В `git merge` используется неразрушающая операция для объединения историй двух веток без их изменения. Команда создает новый merge commit. Это отличный способ консолидировать изменения, однако ваша история коммитов может получить несколько merge-коммитов в зависимости от того, насколько активна мастер-ветка.

Чтобы объединить последние изменения из `master` в `feature-2` с помощью merge выполним:

```
git checkout feature-2
git merge master
```

После слияния `master` в `feature-2` можно продолжать развивать ветку, заканчивая ее и объединяя обратно в `master`. Merge переплел ветви вместе, создав новый коммит слияния (`C7`), что вызвало нелинейную историю в форме ромба — по сути, сохраняя историю.

GIT REBASE

Команда `git rebase` перемещает историю всей ветки поверх другой, переписывая историю проекта новыми коммитами. Команда полезна, если вы предпочитаете чистую и линейную историю проекта. Однако перестроить изменения, перенесенные в главную ветку удаленного репозитория небезопасно, т. к. вы будете изменять историю главной ветви, в то время как члены вашей команды будут продолжать работать над ней.

Кроме того, Git не позволит вам легко запушить пересобранный бранч на удаленный репозиторий. Вам придется заставить его это сделать при помощи `git push --force`, который перезаписывает удаленную ветку и может вызвать проблемы у других участников

Чтобы объединить последние изменения из `master` в `feature-2` с помощью rebase выполним:

```
git checkout feature-2
git rebase master feature-2
```

После перебазирования можно продолжить работу над feature-2 веткой. В Rebase коммит слияния не был создан, вместо этого два коммита `C5` и `C6` были просто перемотаны и повторно применены прямо поверх `C4`, сохраняя линейность истории. Хэши изменились, что указывает на то, что rebase действительно переписывает историю.

Поскольку в `feature-2` ветке есть те же коммиты из `master` слияние ее с `master` будет выполнен в режиме `Fast-forward`.

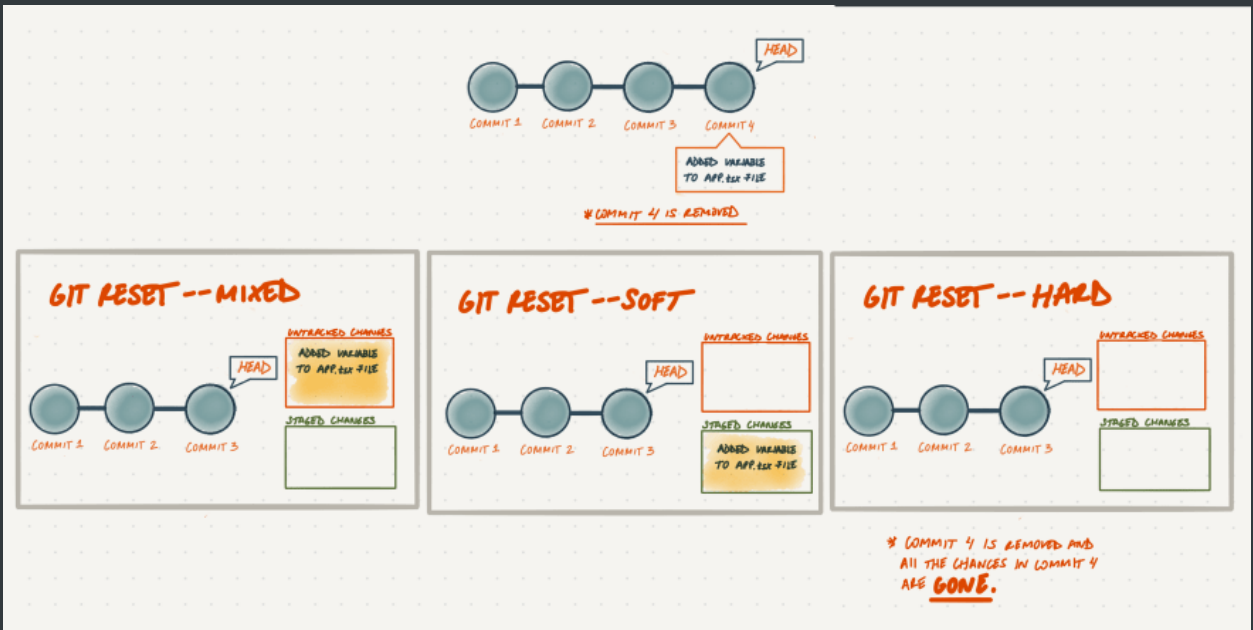
ОПЕРАЦИИ ОТМЕНЫ В GIT

GIT RESET

Git reset позволяет отменять такие операции, как внесение изменений в рабочей директории, добавление их в индекс и сохранение в репозиторий. Имеет три основные режима:

- git reset --soft Обновление HEAD
- git reset --mixed Обновление индекса
- git reset --hard Обновление рабочей директории

Предположим, имеется структура коммитов `– C1 – C2 – C3 – C4 (master)` `HEAD` указывает на `C4` и индекс совпадает с `C4`. Рассмотрим, как повлияет выполнение команды `git reset` в различных режимах.




```
git reset --mixed C3
```

или просто

```
git reset C3
```

`HEAD` будет указывать на `C3`, изменения файлов из `C4` не будут находится в индексе и если вы запустите здесь `git commit` ничего не произойдет т.к. ничего нет в индексе. У нас есть все изменения из `C4`, но если запустить `git status` то вы увидите, что все изменения `not staged`. Чтобы их закоммитить нужно сначала добавить их в индекс командой `git add` и только после этого `git commit`.

```
git reset –soft C3
```

`HEAD` будет указывать на `C3`, изменения из коммита `C4` будут в индексе, как будто вы их добавили командой `git add`. Если выполнить `git commit`, получим коммит полностью идентичный `C4`.

```
git reset --hard C3
```

`HEAD` будет указывать на `C3`, изменения из `C4`, равно как и незакоммиченные изменения, будут удалены и файлы в репозитории будут совпадать с `C3`. Учитывая то, что этот режим подразумевает потерю изменений, вы всегда должны проверять `git status` перед тем, как выполнить жесткий reset чтобы убедиться, что нет незакоммиченных изменений (или они не нужны).

	меняет индекс	меняет файлы в рабочей директории	нужно быть внимательным
reset --soft	-	-	-
reset [--mixed]	+	-	-
reset --hard	+	+	+

GIT REVERT

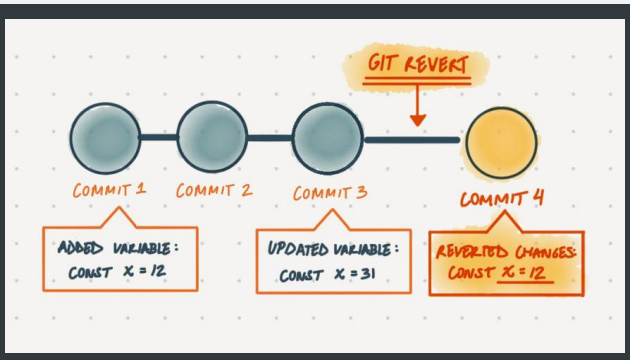
Команда `git revert` – безопасный способ отменить операцию без потери истории коммитов. Команда отменяет действия прошлых коммитов, создавая новый, содержащий все отменённые изменения.

Откатиться с помощью commit-хэшей:

```
git revert [SHA]
```

Можно и с помощью диапазонов:

```
git revert HEAD~[num-of-commits-back]
```



Эта команда полезна, когда вы уже запушили изменения в удаленный репозиторий, так как она сохраняет нетронутым исходный коммит.

GUI КЛИЕНТЫ ДЛЯ GIT

Несмотря на то, что практически все действия в Git можно выполнить при помощи командной строки, для этого инструмента существует достаточное количество инструментов, реализующих графических интерфейсов (GUI). Эти GUI-клиенты способны существенно ускорить вашу работу с системой контроля версий, особенно, если вы еще не слишком хорошо с ней знакомы.

Рассмотрим некоторые GUI для Git.

GITHUB DESKTOP

GitHub Desktop это бесплатное приложение с открытым исходным кодом, разработанное GitHub. С его помощью можно взаимодействовать с GitHub и другими платформами (например Bitbucket и GitLab).

Функционал приложения позволяет легко замечать пул-реквесты в ветках, а также просматривать различия в версиях изображений и блоков кода. При этом элементы для дальнейшего управления можно добавлять в приложение даже путем перетаскивания.

Приложение доступно для macOS и Windows.

FORK

Fork это весьма продвинутый GUI-клиент для macOS и Windows (с бесплатным пробным периодом). В фокусе этого инструмента скорость, дружественность к пользователю и эффективность. К особенностям Fork можно отнести красивый вид, кнопки быстрого доступа, встроенную систему разрешения конфликтов слияния, менеджер репозитория, уведомления GitHub.

С помощью этого инструмента вам будут доступны интуитивный rebase в красивом UI, cherry-pick, подмодули и многое другое.

SOURCETREE

Sourcetree это бесплатный GUI Git для macOS и Windows. Его применение упрощает работу с контролем версий и позволяет сфокусироваться на действительно важных задачах.

Красивый пользовательский интерфейс дает возможность прямого доступа к потокам Git. К вашим услугам локальный поиск по коммитам, интерактивный rebase, менеджер удаленных репозиториев, поддержка больших файлов. Все происходящее вы можете видеть наглядно, а это очень облегчает понимание процессов.

Sourcetree был разработан Atlassian для Bitbucket, но вполне может использоваться в сочетании с другими Git-платформами. Имеет встроенную поддержку Mercurial-репозиториев.

SMARTGIT

SmartGit это Git-клиент для Mac, Linux и Windows. Имеет богатый функционал. Поддерживает пул-реквесты для SVN, GitHub и Bitbucket. В арсенале SmartGit вы найдете CLI для Git, графическое отображение слияний и истории коммитов, SSH-клиент, Git-Flow, программу для разрешения конфликтов слияния.

SmartGit может использоваться бесплатно в некоммерческих проектах.

GITKRAKEN

GitKraken это кроссплатформенный GUI Git для использования с различными платформами контроля версий (включая GitHub, Bitbucket, GitLab). Его цель — повысить вашу продуктивность в использовании Git. Для этого вам предоставляется интуитивный UI, возможность отслеживать задачи, встроенный редактор кода, редактор конфликтов слияния и поддержка интеграции с другими платформами.

Коммерческое использование платное. Также придется купить Pro-версию, если хотите получить расширенный функционал.

Доступен для Mac, Windows и Linux.

GIT В VISUAL STUDIO CODE

Visual Studio Code имеет встроенную поддержку Git. Вам потребуется установить Git версии не ниже, чем 2.0.0.

Основные особенности:

- Просмотр изменений редактируемого файла

- Панель состояния Git (слева внизу), на которой отображается текущая ветка, индикатор ошибок, входящие и исходящие коммиты.
- В редакторе можно делать основные Git операции:
    - Инициализация репозитория.
    - Клонирование репозитория.
    - Создание веток и тегов.
    - Индексация изменений и создание коммитов.
    - Push/pull/sync с удалённой веткой.
    - Разрешение конфликтов слияния.
    - Просмотр изменений.
- С помощью плагина можно работать с запросами слияния на GitHub: https://marketplace.visualstudio.com/items?itemName=GitHub.vscode-pull-request-github.

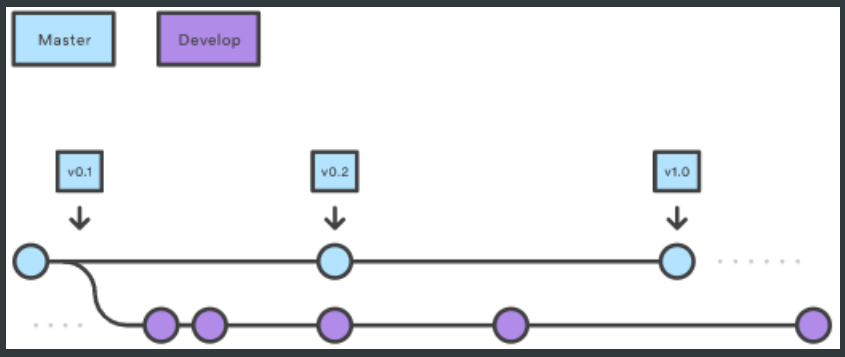
Также VS Code позволяет установить дополнительные плагины, которые не только облегчают работу, но и расширяют доступный функционал Git. Например:

- Git Graph – позволяет видеть древовидную структуру коммитов, а это помогает в осуществлении сложных операций.
- Git Tree Compare — это удобное расширение для сравнения вашего рабочего дерева с веткой, тегом или коммитом.
- GitLens — показывает аннотации к файлам внутри редактора, включая blame (видно коммит и автора каждой строчки), изменения (подсвечивает локальные изменения) и тепловую карту (видно, насколько давно менялись строки).

GIT WORKFLOWS

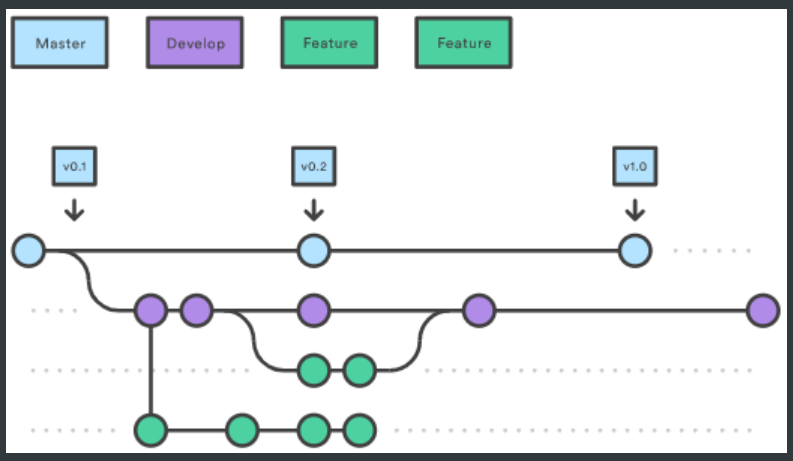
GITFLOW

ВЕТКИ MASTER И DEVELOP



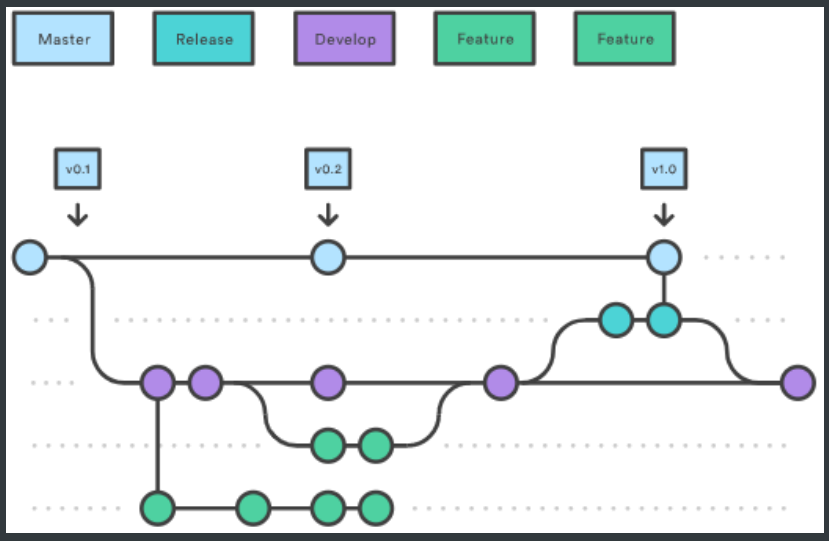
Вместо использования одной ветки `master`, в этой модели используется две ветки для записи истории проекта. В ветке `master` хранится официальная история релиза, а ветка `develop` служит в качестве интеграционной ветки для новых функций. Также, удобно тегировать все коммиты в ветке `master` номером версии.

ВЕТКИ ДЛЯ ФУНКЦИЙ (FEATURE BRANCHES)



Каждая новая функциональность должна разрабатываться в отдельной ветке, которую можно отправлять в центральный репозиторий для создания резервной копии или для совместной работы команды. Ветки функций создаются не на основе `master`, а на основе `develop`. Когда работа над новой функциональностью завершена, она вливается назад в `develop`. Новый код не должен отправляться напрямую в `master`.

ВЕТКИ РЕЛИЗА



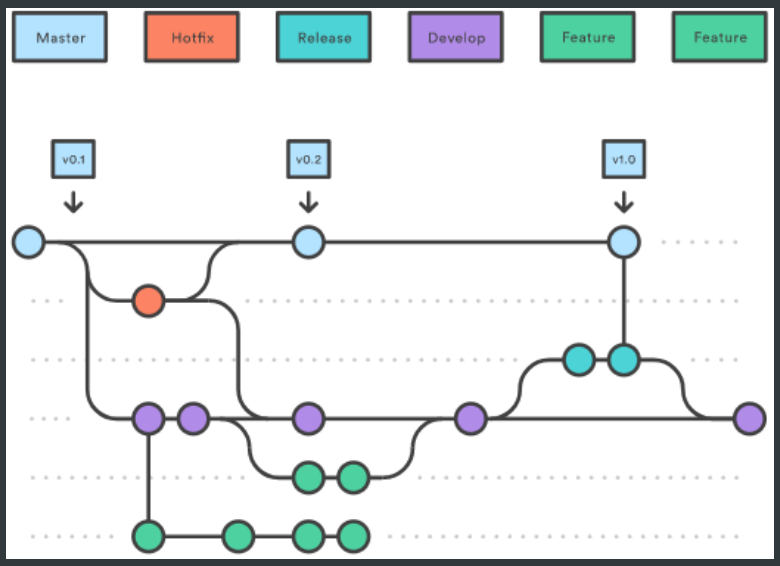
Когда в ветку `develop` уже слито достаточно нового кода для релиза (или подходит установленная дата предрелиза), от ветки develop создается ветка `release`. Создание данной ветки означает начало следующего цикла релиза, в ходе которой новая функциональность уже не добавляется, а производится только отладка багов, создание документации и решение других задач, связанных с релизом. Когда все готово, ветка `release` сливается в `master`, и ей присваивается тег с версией. Кроме этого, она должна быть также слита обратно в ветку `develop`, в которой с момента создания ветки релиза могли добавляться изменения с момента создания ветки релиза.

Использование отдельной ветки для подготовки релиза позволяет одной команде дорабатывать текущий релиз пока другая команда уже работает над функциональностью для следующего релиза.

ВЕТКИ HOTFIX

Ветки `hotfix` используются для быстрого внесения исправлений в рабочую версию кода. Ветки `hotfix` очень похожи на ветки `release` и `feature`, за исключением того, что они созданы от `master`, а не от `develop`. Это единственная ветка, которая должна быть создана непосредственно от `master`. Как только исправление завершено, ветка `hotfix` должна быть объединена как с `master`, так и с `develop` (или с веткой текущего релиза), а `master` должен быть помечен обновленным номером версии.

Наличие специальной ветки для исправления ошибок позволяет команде решать проблемы, не прерывая остальную часть рабочего процесса и не ожидая следующего цикла подготовки к релизу. Можно говорить о ветках `hotfix`, как об особых ветках `relese`, которые работают напрямую с `master`.



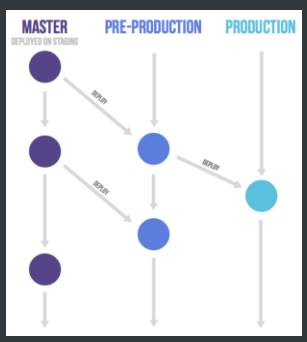
GITLAB FLOW

PRODUCTION ВЕТКА

Для управления выпуском кода в продакшен GitLab flow предлагает использовать специальную ветку `production`. Настройте автоматическое развёртывание кода из этой ветки при каждом изменении в ней. Теперь для релиза достаточно сделать мерж из ветки `master` в `production`. Состояние ветки даст вам точную информацию о том, какая версия кода сейчас выпущена.

ВЕТКИ ДЛЯ НЕСКОЛЬКИХ СРЕД

Может быть полезно иметь отдельную среду (environment), в которую происходит развёртывание из ветки `master`. В этом единственном случае название среды может отличаться от названия ветки.

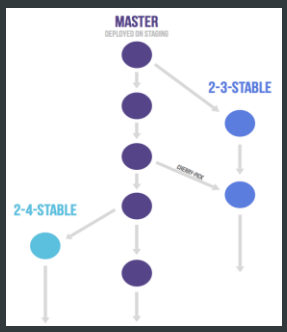


Предположим, что у вас есть несколько сред: стейджинг (staging), пре-продакшен (pre-production) и продакшен (production). Код из `master` автоматически развёртывается на стейджинг. Такой процесс, когда все коммиты проходят через ветки в строго определенном порядке, гарантирует, что изменения прошли тестирование во всех средах.

Если вам нужно быстро «протащить» хотфикс на продакшен, то можно реализовать его в обычной `feature`-ветке, а потом открыть мерж-реквест в `master`, не удаляя ветку. Теперь, если код в `master` проходит тесты и жизнеспособен (правильно настроенная непрерывная доставка должна гарантировать это), вы можете замержить ветку хотфикса последовательно в `pre-production` и `production`. Если же изменения требуют дополнительного тестирования, то вместо немедленного мержа нужно открыть мерж-реквесты в те же ветки. В «экстремальном» случае отдельная среда может создаваться для каждой ветки.

РЕЛИЗНЫЕ ВЕТКИ

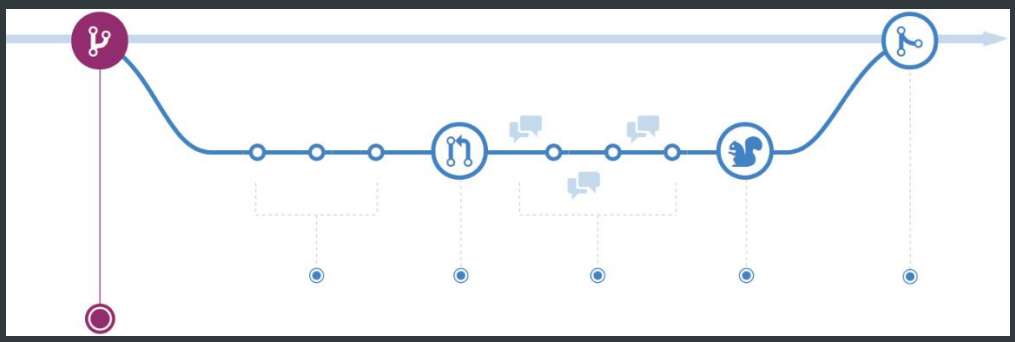
Ветки релизов понадобятся вам только если вы выпускаете ПО для внешних клиентов. В таком случае каждая минорная версия будет храниться в отдельной ветке (2.3-stable, 2.4-stable и т.п.).



Стабильные (stable) ветки должны создаваться от ветки `master`. Их нужно создавать как можно позже, чтобы минимизировать добавление хотфиксов в несколько веток. После того, как релизная ветка создана, в неё можно включать только исправления серьёзных багов. Следуйте правилу "upstream first": всегда, когда это возможно, сначала делайте мерж исправлений в `master`, и только оттуда — cherry-pick в релизную ветку. Благодаря этому правилу вы не забудете сделать cherry-pick исправлений в master и не встретите тот же самый баг в следующем релизе. Каждый раз, когда в релизную ветку добавляется исправление бага, нужно повысить третье число в номере версии (по правилам семантического версионирования). Обозначьте эту версию новым тегом в git. В некоторых проектах используется ветка `stable`, которая всегда указывает на тот же коммит, что и последний релиз. Ветка `production` (или master в правилах git flow) в таком случае не нужна.

GITHUB FLOW

ВЕТКИ ДЛЯ ФУНКЦИЙ

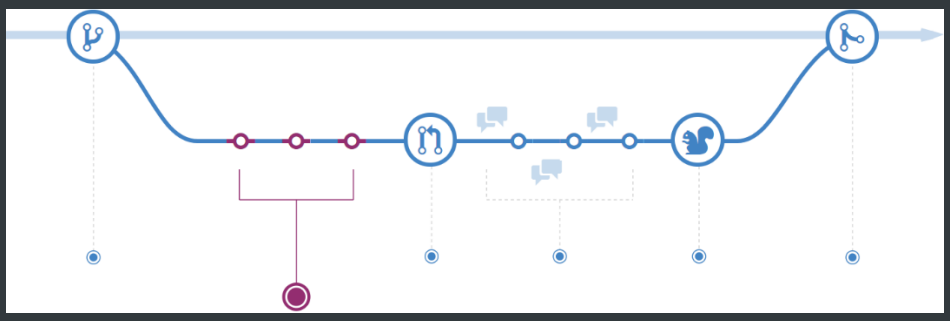


Пока вы работаете над одним проектом, может существовать множество различных реализуемых параллельно улучшений. Некоторые из них готовы к работе, а другие — нет. Ветвление позволяет вам управлять этим рабочим процессом.

При создании ветви в проекте создается среда, в которой можно опробовать новые идеи. Изменения, вносимые в отдельной ветке, не влияют на основной ствол (`master branch`). Это позволяет вам экспериментировать и фиксировать изменения безопасно, зная, что ваша ветвь никак не повлияет на другие, пока не будет действительно готова.

GitHub Flow основан на ветвлении и согласно ему есть только одно правило: всё, что находится в стволе — гарантированно стабильно и готово к деплою в любой момент. Поэтому чрезвычайно важно, чтобы любая ваша новая ветвь создавалась именно от ствола. А имя ветви было быть описательным, чтобы другим было понятно, над чем в ней идёт работа. Например: `refactor-authentication`, `user-content-cache-key`, `make-retina-avatars`.

ФИКСАЦИЯ ИЗМЕНЕНИЙ



Создав ветвь, начинайте вносить в неё изменения. Добавляя, редактируя или удаляя файлы, не забывайте делать новые коммиты в ветви. Последовательность коммитов

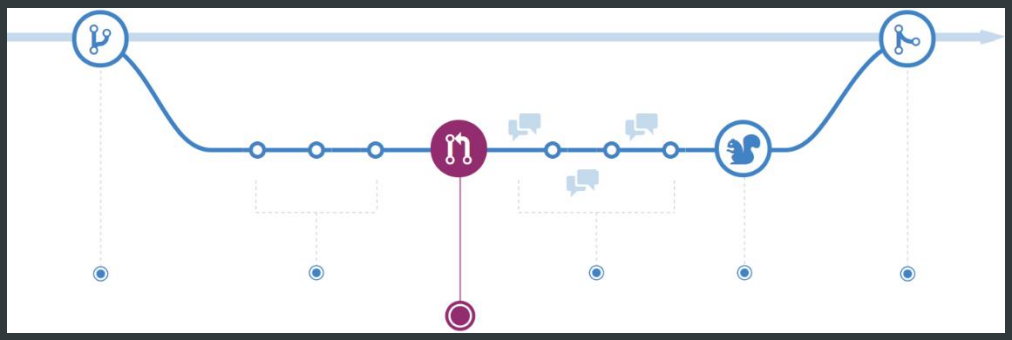
образует в конечном счёте прозрачную историю работы над задачей, по которой остальные смогут понять, что делали и почему.

У каждого коммита есть связанное сообщение, являющееся объяснением, почему было сделано то или иное изменение. Также каждый коммит считается отдельной единицей изменения. Это позволяет откатить изменения, если обнаружилась ошибка, или если вы решите пойти в другом направлении.

Внятное описание коммита очень важно, так как позволяет остальным разработчикам сразу понять смысл и оценить насколько внесённые изменения им соответствуют. А значит обратная связь от них придёт быстрее и окажется полезней.

Вливайте изменения из ствола в функциональные ветви как можно чаще, чтобы она всегда оставалась актуальной и готовой к обратному слиянию. Разрешение возможных конфликтов слияния — право и обязанность разработчика ветви, так как именно он лучше всего знает зафиксированные в ней изменения.

ЗАПРОСЫ НА СЛИЯНИЕ



Запросы на слияние (pull request) инициируют обсуждение коммитов. Поскольку они тесно интегрированы с базовым git-репозиторием, любой может однозначно понять, какие изменения будут внесены в ствол, если они примут запрос.

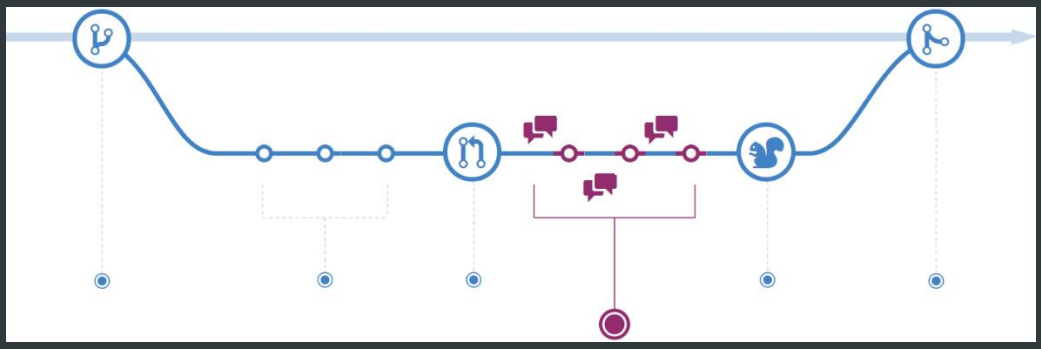
Запрос на слияние может быть открыт в любой момент процесса разработки:

- когда у вас мало или вообще нет кода, но вы хотите поделиться некоторыми скриншотами или общими идеями
- когда вы застряли и нуждаетесь в помощи или совете
- когда вы готовы к тому, чтобы кто-либо проверил вашу работу

Используя систему @упоминаний GitHub в сообщении запроса на слияние, вы можете запросить обратную связь от конкретных людей или целых команд, будь то сосед по офису или кто-то в десяти часовых поясах от вас.

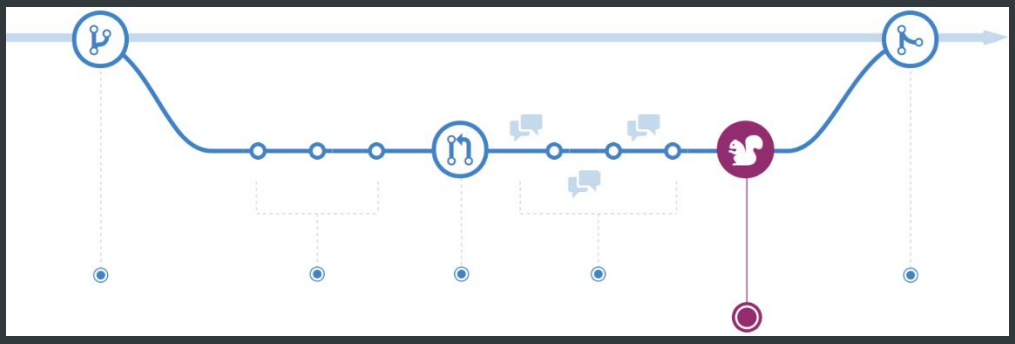
Запросы на слияние полезны не только в рамках одного репозитория, но и как инструмент переноса кода между форками. Просто создайте запрос на слияние ветви из одного репозитория в ветвь из другого и действуйте дальше.

ПРОВЕРКА И ОБСУЖДЕНИЕ КОДА



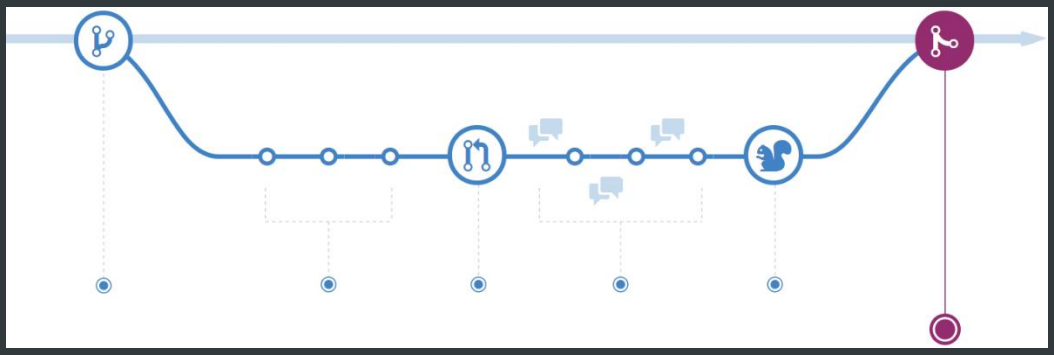
После открытия запроса на слияние команда рассматривает изменения, задавая вопросы и оставляя комментарии. Если вам указывают, что вы забыли что-то сделать или в коде есть ошибка, вы можете исправить это в своей ветке и обновить удаленную ветку. GitHub покажет ваши новые коммиты и любую дополнительную информацию в том же унифицированном представлении запроса на слияние.

ПРОВЕРКА В БОЮ



После проверки запроса на слияние и прохождения тестов ветвь можно развернуть в боевом окружении, чтобы окончательно убедиться в её работоспособности. Если ветвь вызывает какие-либо проблемы, то ее можно быстро откатить, развернув вместо неё гарантированно работоспособный основной ствол. Так как ветвь ещё не была никуда влита, то всякое её влияние на другие ветви по прежнему исключено и проблемный код не сможет поломать другие ветви, так что можно продолжить работу над задачей, пока она не будет действительно готова.

СЛИЯНИЕ



Теперь, когда изменения были проверены в бою, пришло время влить код в основной ствол.

При слиянии в стволе создаётся коммит со всеми изменениями из ветки. Как и любые другие коммиты, он доступен для поиска и "перемещения во времени".

А ЕСЛИ ЧАСТЫЕ РЕЛИЗЫ НЕВОЗМОЖНЫ?

Если вы не практикуете непрерывную поставку (Continous Delivery), то вам может быть сложно доводить до ствола каждую ветвь по отдельности. В этом случае просто создавайте интеграционный ветви, куда вливайте лишь те ветви, что считаете готовыми. Если изменения одной из ветвей вызовут проблемы, то интеграционную ветвь всегда можно будет пересобрать заново, но уже не включая проблемную. Это позволит вам не срывать график релизов, даже если какие-либо из планируемых задач оказались не до конца готовы к запланированной дате.

ОТЛИЧИЕ ОТ GITFLOW

В GitFlow есть дополнительная ветвь develop куда сливаются все разрабатываемые в текущий момент ветви. develop необходимо "стабилизировать" перед релизом, что часто приводит либо к переносу релиза, либо "релизу с замечаниями".

ОТЛИЧИЕ ОТ GITLAB FLOW?

В GitLab Flow вы сначала вливаете ветвь в основной ствол и лишь потом разворачиваете в тестовом, боевом и других окружениях. Если релиз окажется проблемным и его потребуется откатить, то потребуется порой весьма проблемный "reverse merge" либо "стабилизация" ствола, как в случае GitFlow.