**Введение - О системе контроля версий**

**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 этой книге 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 — 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**

Многие люди в качестве метода контроля версий применяют копирование файлов в отдельный каталог (возможно даже, каталог с отметкой по времени, если они достаточно сообразительны). Данный подход очень распространён из-за его простоты, однако он невероятно сильно подвержен появлению ошибок. Можно легко забыть в каком каталоге вы находитесь и случайно изменить не тот файл или скопировать не те файлы, которые вы хотели.

Для того, чтобы решить эту проблему, программисты давным-давно разработали локальные VCS с простой базой данных, которая хранит записи о всех изменениях в файлах, осуществляя тем самым контроль ревизий.

**Централизованные системы контроля версий**

Следующая серьёзная проблема, с которой сталкиваются люди, — это необходимость взаимодействовать с другими разработчиками. Для того, чтобы разобраться с ней, были разработаны централизованные системы контроля версий (Centralized Version Control System, далее CVCS). Такие системы, как CVS, Subversion и Perforce, используют единственный сервер, содержащий все версии файлов, и некоторое количество клиентов, которые получают файлы из этого централизованного хранилища. Применение CVCS являлось стандартом на протяжении многих лет.

Такой подход имеет множество преимуществ, особенно перед локальными VCS. Например, все разработчики проекта в определённой степени знают, чем занимается каждый из них. Администраторы имеют полный контроль над тем, кто и что может делать, и гораздо проще администрировать CVCS, чем оперировать локальными базами данных на каждом клиенте.

Несмотря на это, данный подход тоже имеет серьёзные минусы. Самый очевидный минус — это единая точка отказа, представленная централизованным сервером. Если этот сервер выйдет из строя на час, то в течение этого времени никто не сможет использовать контроль версий для сохранения изменений, над которыми работает, а также никто не сможет обмениваться этими изменениями с другими разработчиками. Если жёсткий диск, на котором хранится центральная БД, повреждён, а своевременные бэкапы отсутствуют, вы потеряете всё — всю историю проекта, не считая единичных снимков репозитория, которые сохранились на локальных машинах разработчиков. Локальные VCS страдают от той же самой проблемы: когда вся история проекта хранится в одном месте, вы рискуете потерять всё.

**Распределённые системы контроля версий**

Здесь в игру вступают распределённые системы контроля версий (Distributed Version Control System, далее DVCS). В DVCS (таких как Git, Mercurial, Bazaar или Darcs) клиенты не просто скачивают снимок всех файлов (состояние файлов на определённый момент времени) — они полностью копируют репозиторий. В этом случае, если один из серверов, через который разработчики обменивались данными, умрёт, любой клиентский репозиторий может быть скопирован на другой сервер для продолжения работы. Каждая копия репозитория является полным бэкапом всех данных.

Более того, многие DVCS могут одновременно взаимодействовать с несколькими удалёнными репозиториями, благодаря этому вы можете работать с различными группами людей, применяя различные подходы единовременно в рамках одного проекта. Это позволяет применять сразу несколько подходов в разработке, например, иерархические модели, что совершенно невозможно в централизованных системах.

**Краткая история Git**

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) 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)
* 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

**Что такое Git?**

Очень важно понять эту часть материала, потому что если вы поймёте, что такое Git и основы того, как он работает, тогда, возможно, вам будет гораздо проще его использовать. Git хранит и использует информацию совсем иначе по сравнению с другими системами, даже несмотря на то, что интерфейс пользователя достаточно похож, и понимание этих различий поможет вам избежать путаницы во время использования.

Основное отличие Git от любой другой системы контроля версий (включая Subversion и её собратьев) — это подход к работе со своими данными. Подход Git к хранению данных больше похож на набор снимков миниатюрной файловой системы. Каждый раз, когда вы делаете коммит, то есть сохраняете состояние своего проекта в Git, система запоминает, как выглядит каждый файл в этот момент, и сохраняет ссылку на этот снимок. Для увеличения эффективности, если файлы не были изменены, Git не запоминает эти файлы вновь, а только создаёт ссылку на предыдущую версию идентичного файла, который уже сохранён. Git представляет свои данные как, скажем, **поток снимков**.

**Почти все операции выполняются локально**

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

**Целостность Git**

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

Механизм, которым пользуется Git при вычислении хеш-сумм, называется SHA-1 хеш. Это строка 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

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

**Git обычно только добавляет данные**

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

**Три состояния**

Теперь слушайте внимательно. Это самая важная вещь, которую нужно запомнить. У Git есть три основных состояния, в которых могут находиться ваши файлы: **изменён** (modified), **индексирован** (staged) и **зафиксирован** (committed):

* К изменённым относятся файлы, которые поменялись, но ещё не были зафиксированы.
* Индексированный — это изменённый файл в его текущей версии, отмеченный для включения в следующий коммит.
* Зафиксированный значит, что файл уже сохранён в вашей локальной базе.

И начнем мы со списка терминов, встречающихся в тексте и необходимых для его понимания.  
**Рабочее дерево (Working tree)** — Любая директория в вашей файловой системе, связанная с *репозиторием* (что можно видеть по наличию в ней поддиректории «.git»). Включает в себя все файлы и поддиректории.

* **Коммит (Commit)**. В роли существительного: «моментальный снимок» *рабочего дерева* в какой-то момент времени. В роли глагола: коммитить (закоммитить) — добавлять *коммит* в *репозиторий*.
* **Репозиторий (Repository)** — это набор *коммитов*, т.е. просто архив прошлых состояний *рабочего дерева* проекта на вашей или чьей-то машине.
* **Ветка (Branch)** — просто имя для *коммита*, также называемое *ссылкой* (reference). Определяет происхождение — «родословную» *коммита*, и таким образом, является типичным представлением «ветки разработки»
* **Checkout** — операция переключения между *ветками* или восстановления файлов *рабочего дерева*
* **Метка (Tag)** — также имя для *коммита*, отличающееся от *ветки* тем, что оно всегда постоянно указывает на один и тот же *коммит*, а также может иметь свое текстовое описание
* **Мастер (Master)**. Условно «главная» или «основная» *ветка репозитория*, но по сути ничем не отличающаяся от прочих *веток*
* **Индекс (Index)**. В отличие от других подобных инструментов, Git не передает изменения из *рабочего дерева* в *репозиторий* напрямую. Вместо этого изменения сначала регистрируются в *индексе*, или «области подготовки» (staging area). Это можно рассматривать как способ «подтверждения» ваших изменений перед совершением *коммита*, который запишет в *репозиторий* все одобренные изменения.
* **HEAD** — заголовок. Используется *репозиторием* для определения того, что выбрано с помощью *checkout*  
  + Если субъект *checkout* — ветка, то *HEAD* будет ссылаться на нее, показывая, что имя ветки должно быть обновлено во время следующего *коммита*
  + Если субъект *checkout* — коммит, то *HEAD* будет ссылаться только на него. В этом случае *HEAD* называется обособленным (detached)

**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.

Любые проекты, включающие в себя больше одного участника, требуют персонализации, и GIT - не исключение. Для корректной работы репозитория, а также для того, чтобы иметь возможность установить автора внесенных изменений, необходимо представиться - ввести имя пользователя и электронную почту.

$ git config --global user.name "Noname Incognito"

$ git config --global user.email innoname@email.com

Отныне все ваши действия в GIT будут фиксироваться под вашим именем.

Чтобы узнать текущие настройки (например, под какими данными мы авторизованы), можно запросить эту  информацию у GIT:

$ git config --list

Данная команда выдаст все конфигурации, установленные для конкретного пользователя. Также есть возможность узнать конкретную настройку, например, ввести такую команду: которая выдаст информацию строго по запросу.

$ git config user.name

Менять имя пользователя и емейл в процессе работы **крайне нежелательно**, поскольку могут возникнуть ошибки, связанные с множественностью авторов, и отследить историю изменений будет проблематично.

Также **git config** позволяет изменить визуальный вывод данных для пользователя - настройки интерфейса могут облегчить восприятие, выделив нужные места разными цветами:

$ git config --global color.ui true

**Как получить помощь?**

Если вам нужна помощь при использовании Git, есть три способа открыть страницу руководства по любой команде Git:

$ git help <команда>

$ git <команда> --help

$ man git-<команда>

Например, так можно открыть руководство по команде git config - $ git help config

Эти команды хороши тем, что ими можно пользоваться всегда, даже без подключения к сети.

Так же, если вам нужно посмотреть только список опций и вы не хотите читать полную документацию по команде, вы можете использовать опцию -h для вывода краткой инструкции по использованию: $ git add -h

**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.

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

**Клонирование существующего репозитория**

Для получения копии существующего Git-репозитория, например, проекта, в который вы хотите внести свой вклад, необходимо использовать команду git clone. Git получает копию практически всех данных, которые есть на сервере. При выполнении git clone с сервера забирается (pulled) каждая версия каждого файла из истории проекта. Фактически, если серверный диск выйдет из строя, вы можете использовать любой из клонов на любом из клиентов, для того, чтобы вернуть сервер в то состояние, в котором он находился в момент клонирования (вы можете потерять часть серверных хуков (server-side hooks) и т. п., но все данные, помещённые под версионный контроль, будут сохранены,

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

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.

## Запись изменений в репозиторий

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

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

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

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

### Определение состояния файлов

Основной инструмент, используемый для определения, какие файлы в каком состоянии находятся — это команда git status.

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

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

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 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,

### 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:

### 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.

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.

|  |  |
| --- | --- |
| **Подсказка** | GitHub поддерживает довольно полный список примеров .gitignore файлов для множества проектов и языков <https://github.com/github/gitignore> это может стать отправной точкой для .gitignore в вашем проекте. |
| **Примечание** | В простейшем случае репозиторий будет иметь один файл .gitignore в корневом каталоге, правила из которого будут рекурсивно применяться ко всем подкаталогам. Так же возможно использовать .gitignore файлы в подкаталогах. Правила из этих файлов будут применяться только к каталогам, в которых они находятся. Например, репозиторий исходного кода ядра Linux содержит 206 файлов .gitignore. |

### Просмотр индексированных и неиндексированных изменений

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

Допустим, вы снова изменили и проиндексировали файл README, а затем изменили файл CONTRIBUTING.md без индексирования.

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.

### 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 – m ‘name\_commit’:

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

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

**Просмотр истории коммитов**

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

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

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

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

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

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

$ git log -p -2

Например, если вы хотите увидеть сокращённую статистику для каждого коммита, вы можете использовать опцию --stat:

$ git log --stat

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

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

$ git log --pretty=oneline

**Операции отмены**

В любой момент вам может потребоваться что-либо отменить. Здесь мы рассмотрим несколько основных способов отмены сделанных изменений. Будьте осторожны, не все операции отмены в свою очередь можно отменить! Это одна из редких областей Git, где неверными действиями можно необратимо удалить результаты своей работы.

Отмена может потребоваться, если вы сделали коммит слишком рано, например, забыв добавить какие-то файлы или комментарий к коммиту. Если вы хотите переделать коммит — внесите необходимые изменения, добавьте их в индекс и сделайте коммит ещё раз, указав параметр --amend:

$ git commit --amend

Эта команда использует область подготовки (индекс) для внесения правок в коммит. Если вы ничего не меняли с момента последнего коммита (например, команда запущена сразу после предыдущего коммита), то снимок состояния останется в точности таким же, а всё что вы сможете изменить — это ваше сообщение к коммиту.

Запустится тот же редактор, только он уже будет содержать сообщение предыдущего коммита. Вы можете редактировать сообщение как обычно, однако, оно заменит сообщение предыдущего коммита.

Например, если вы сделали коммит и поняли, что забыли проиндексировать изменения в файле, который хотели добавить в коммит, то можно сделать следующее:

$ git commit -m 'Initial commit'

$ git add forgotten\_file

$ git commit --amend

В итоге получится единый коммит — второй коммит заменит результаты первого.

**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.

**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:

**Работа с удалёнными репозиториями**

Для того, чтобы внести вклад в какой-либо Git-проект, вам необходимо уметь работать с удалёнными репозиториями. Удалённые репозитории представляют собой версии вашего проекта, сохранённые в интернете или ещё где-то в сети. У вас может быть несколько удалённых репозиториев, каждый из которых может быть доступен для чтения или для чтения-записи. Взаимодействие с другими пользователями предполагает управление удалёнными репозиториями, а также отправку и получение данных из них. Управление репозиториями включает в себя как умение добавлять новые, так и умение удалять устаревшие репозитории, а также умение управлять различными удалёнными ветками, объявлять их отслеживаемыми или нет и так далее. В данном разделе мы рассмотрим некоторые из этих навыков.

**Просмотр удалённых репозиториев**

Для того, чтобы просмотреть список настроенных удалённых репозиториев, вы можете запустить команду git remote. Она выведет названия доступных удалённых репозиториев. Если вы клонировали репозиторий, то увидите как минимум origin — имя по умолчанию, которое Git даёт серверу, с которого производилось клонирование:

Вы можете также указать ключ -v, чтобы просмотреть адреса для чтения и записи, привязанные к репозиторию:

$ git remote -v

**Добавление удалённых репозиториев**

В предыдущих разделах мы уже упоминали и приводили примеры добавления удалённых репозиториев, сейчас рассмотрим эту операцию подробнее. Для того, чтобы добавить удалённый репозиторий и присвоить ему имя (shortname), просто выполните команду git remote add <shortname> <url>:

**Получение изменений из удалённого репозитория — Fetch и Pull**

Как вы только что узнали, для получения данных из удалённых проектов, следует выполнить:

$ git fetch [remote-name]

Данная команда связывается с указанным удалённым проектом и забирает все те данные проекта, которых у вас ещё нет. После того как вы выполнили команду, у вас должны появиться ссылки на все ветки из этого удалённого проекта, которые вы можете просмотреть или слить в любой момент.

Когда вы клонируете репозиторий, команда clone автоматически добавляет этот удалённый репозиторий под именем «origin». Таким образом, git fetch origin извлекает все наработки, отправленные на этот сервер после того, как вы его клонировали (или получили изменения с помощью fetch). Важно отметить, что команда git fetch забирает данные в ваш локальный репозиторий, но не сливает их с какими-либо вашими наработками и не модифицирует то, над чем вы работаете в данный момент. Вам необходимо вручную слить эти данные с вашими, когда вы будете готовы.

Если ветка настроена на отслеживание удалённой ветки (см. следующий раздел и главу [Ветвление в Git](https://git-scm.com/book/ru/v2/ch00/ch03-git-branching) чтобы получить больше информации), то вы можете использовать команду git pull чтобы автоматически получить изменения из удалённой ветки и слить их со своей текущей. Этот способ может для вас оказаться более простым или более удобным. К тому же, по умолчанию команда git clone автоматически настраивает вашу локальную ветку master на отслеживание удалённой ветки master на сервере, с которого вы клонировали репозиторий. Название веток может быть другим и зависит от ветки по умолчанию на сервере. Выполнение git pull, как правило, извлекает (fetch) данные с сервера, с которого вы изначально клонировали, и автоматически пытается слить (merge) их с кодом, над которым вы в данный момент работаете.

**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. Обратитесь к главе [Ветвление в Git](https://git-scm.com/book/ru/v2/ch00/ch03-git-branching) для более подробного описания, как отправлять изменения на удалённый сервер.

**Просмотр удалённого репозитория**

Если хотите получить побольше информации об одном из удалённых репозиториев, вы можете использовать команду git remote show <remote>. Выполнив эту команду с некоторым именем, например, origin, вы получите следующий результат:

$ git remote show origin

\* remote origin

**Удаление и переименование удалённых репозиториев**

Для переименования удалённого репозитория можно выполнить git remote rename. Например, если вы хотите переименовать pb в paul, вы можете это сделать при помощи git remote rename:

$ git remote rename pb paul

$ git remote

origin

paul

Стоит упомянуть, что это также изменит имена удалённых веток в вашем репозитории. То, к чему вы обращались как pb/master, теперь стало paul/master.

Если по какой-то причине вы хотите удалить удалённый репозиторий — вы сменили сервер или больше не используете определённое зеркало, или кто-то перестал вносить изменения — вы можете использовать git remote rm:

$ git remote remove paul

$ git remote

origin

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

# О ветвлении в двух словах

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.

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

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

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.

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

### 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 testing

В результате создаётся новый указатель на текущий коммит.

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

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

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

$ git log --oneline --decorate

Здесь можно увидеть указывающие на коммит f30ab ветки: master и testing.

### Переключение веток

Для переключения на существующую ветку выполните команду git checkout. Давайте переключимся на ветку testing:

$ git checkout testing

В результате указатель HEAD переместится на ветку testing.

Какой в этом смысл? Давайте сделаем ещё один коммит:

$ vim test.rb

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

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

$ git checkout master

|  |  |
| --- | --- |
| **Примечание** | **git log не показывает все ветки по умолчанию**  Если выполнить команду git log прямо сейчас, то в её выводе только что созданная ветка «testing» фигурировать не будет.  Ветка никуда не исчезла; просто Git не знает, что именно она вас интересует, и выводит наиболее полезную по его мнению информацию. Другими словами, по умолчанию git log отобразит историю коммитов только для текущей ветки.  Для просмотра истории коммитов другой ветки необходимо явно указать её имя: git log testing Чтобы посмотреть историю по всем веткам — выполните команду с дополнительным флагом: git log --all. |

Рисунок 16. HEAD перемещается когда вы делаете checkout

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

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

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

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

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

|  |  |
| --- | --- |
| **Примечание** | **Одновременное создание новой ветки и переключение на неё**  Как правило, при создании новой ветки вы хотите сразу на неё переключиться — это можно сделать используя команду git checkout -b <newbranchname>. |
| **Примечание** | Начиная с Git версии 2.23, вы можете использовать git switch вместо git checkout, чтобы:   * Переключиться на существующую ветку: git switch testing-branch. * Создать новую ветку и переключиться на неё: git switch -c new-branch. Флаг -c означает создание, но также можно использовать полный формат:` --create`. * Вернуться к предыдущей извлечённой ветке: git switch -. |

Git-flow — альтернативная модель ветвления Git, в которой используются функциональные ветки и несколько основных веток. Эта модель была впервые опубликована и популяризована [Винсентом Дриссеном на сайте nvie](http://nvie.com/posts/a-successful-git-branching-model/). По сравнению с моделью магистральной разработки, в Git-flow используется больше веток, каждая из которых существует дольше, а коммиты обычно крупнее. В соответствии с этой моделью разработчики создают функциональную ветку и откладывают ее слияние с главной магистральной веткой до завершения работы над функцией. Такие долгосрочные функциональные ветки требуют тесного взаимодействия разработчиков при слиянии и создают повышенный риск отклонения от магистральной ветки. В них также могут присутствовать конфликтующие обновления.

Git-flow можно использовать для проектов, в которых запланирован цикл релизов и реализуется характерная для [DevOps](https://www.atlassian.com/ru/devops/what-is-devops/devops-best-practices) методика [непрерывной поставки](https://www.atlassian.com/ru/continuous-delivery). В этом рабочем процессе используются понятия и команды, которые были предложены в рамках [рабочего процесса с функциональными ветками](https://www.atlassian.com/ru/git/tutorials/comparing-workflows/feature-branch-workflow). Однако Git-flow привносит новые специфические роли для разных веток и определяет характер и частоту взаимодействия между ними. Помимо функциональных веток в рамках этого рабочего процесса используются отдельные ветки для подготовки, поддержки и регистрации релизов. При этом вы по-прежнему можете пользоваться преимуществами рабочего процесса с функциональными ветками, такими как запросы pull, изолированные эксперименты и эффективное командное взаимодействие.

Gitflow — это лишь методика работы с Git; в ней определяется, какие виды веток необходимы проекту и как выполнять слияние между ними. Ниже мы познакомимся с назначением веток. Набор инструментов git-flow представляет собой отдельную утилиту командной строки, которая требует установки. Процесс установки прост. Пакеты команд git-flow доступны для многих операционных систем. В системах OS X можно выполнить команду brew install git-flow. Если вы используете Windows, вам нужно [загрузить и установить git-flow](https://git-scm.com/download/win). После установки решения git-flow необходимо выполнить команду git flow init, чтобы использовать его в проекте. Этот набор инструментов играет роль обертки Git. Команда git flow init является расширением стандартной команды git init и не вносит изменений в репозиторий помимо создания веток.

**Ветка разработки и главная ветка**

В этом рабочем процессе для регистрации истории проекта вместо одной ветки main используются две ветки. В главной ветке main хранится официальная история релиза, а ветка разработки develop предназначена для объединения всех функций. Кроме того, для удобства рекомендуется присваивать всем коммитам в ветке main номер версии.

Первый шаг рабочего процесса заключается в создании ветки develop от стандартной ветки main. Разработчику будет проще создать пустую ветку develop локально и отправить ее на сервер.

git branch develop

git push -u origin develop

В этой ветке будет храниться полная история проекта, а в ветке main — сокращенная. Теперь другим разработчикам следует клонировать центральный репозиторий и создать отслеживающую ветку для ветки develop.

При использовании библиотеки расширений git-flow, для создания ветки develop можно выполнить команду git flow init в существующем репозитории:

$ git flow init

Initialized empty Git repository in ~/project/.git/

No branches exist yet. Base branches must be created now.

Branch name for production releases: [main]

Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?

Feature branches? [feature/]

Release branches? [release/]

Hotfix branches? [hotfix/]

Support branches? [support/]

Version tag prefix? []

$ git branch

\* develop

 main

**Функциональные ветки (feature)**

Под каждую новую функцию нужно выделить собственную ветку, которую можно [отправить в центральный репозиторий](https://www.atlassian.com/ru/git/tutorials/syncing/git-push) для создания резервной копии или совместной работы команды. Ветки feature создаются не на основе main, а на основе develop. Когда работа над функцией завершается, соответствующая ветка [сливается с веткой develop](https://www.atlassian.com/ru/git/tutorials/using-branches/git-merge). Функции не следует отправлять напрямую в ветку main.

Обратите внимание, что комбинация веток feature с веткой develop фактически представляет собой рабочий процесс с функциональными ветками. Но рабочий процесс Gitflow на этом не заканчивается.

Как правило, ветки feature создаются на основе последней ветки develop.

**Создание функциональной ветки**

Без использования расширений git-flow:

git checkout develop

git checkout -b feature\_branch

С использованием расширений git-flow:

git flow feature start feature\_branch

Продолжайте работу с Git как обычно.

**Ветки выпуска (release)**

Когда в ветке develop оказывается достаточно функций для выпуска (или приближается назначенная дата релиза), от ветки develop создается ветка release. Создание этой ветки запускает следующий цикл релиза, и с этого момента новые функции добавить больше нельзя — допускается лишь исправление багов, создание документации и решение других задач, связанных с релизом. Когда подготовка к поставке завершается, ветка release сливается с main и ей присваивается номер версии. Кроме того, нужно выполнить ее слияние с веткой develop, в которой с момента создания ветки релиза могли возникнуть изменения.

Благодаря тому, что для подготовки выпусков используется специальная ветка, одна команда может дорабатывать текущий выпуск, в то время как другая команда продолжает работу над функциями для следующего. Это также позволяет разграничить этапы разработки (например, можно без труда посвятить неделю подготовке к версии 4.0 и действительно увидеть это в структуре репозитория).

Создание веток release — это еще одна простая операция ветвления. Как и ветки feature, ветки release основаны на ветке develop. Создать новую ветку release можно с помощью следующих команд.

Без использования расширений git-flow:

git checkout develop

git checkout -b release/0.1.0

При использовании расширений git-flow:

$ git flow release start 0.1.0

Switched to a new branch 'release/0.1.0'

Когда подготовка к поставке завершается, релиз сливается с ветками main и develop, а ветка release удаляется. Важно слить ее с веткой develop, поскольку в ветку release могли добавить критические обновления, которые должны быть доступны для новых функций. Если в вашей организации уделяют повышенное внимание проверке кода, это идеальное место для выполнения запроса pull.

Для завершения работы в ветке release используйте следующие команды:

Без использования расширений git-flow:

git checkout main

git merge release/0.1.0

Или при использовании расширений git-flow:

git flow release finish '0.1.0'

**Ветки исправления (hotfix)**

Ветки сопровождения или исправления (hotfix) используются для быстрого внесения исправлений в рабочие релизы. Ветки hotfix очень похожи на ветки release и feature. Отличие заключается в том, что они создаются на основе main, а не develop. Это единственная ветка, которую нужно обязательно создавать напрямую от main. Как только исправление завершено, эту ветку следует слить с main и develop (или текущей веткой release), а ветке main присвоить обновленный номер версии.

Благодаря специальной ветке для исправления ошибок команда может устранять проблемы, не прерывая остальную часть рабочего процесса и не дожидаясь следующего цикла релиза. Ветки сопровождения можно рассматривать как специальные ветки release, которые работают непосредственно с main. Ветку hotfix можно создать с помощью следующих команд.

Gitflow — лишь одна из многих методологий [работы с Git](https://www.atlassian.com/ru/git/tutorials/comparing-workflows), доступных вам и вашей команде.

Ключевые идеи, которые нужно запомнить о Gitflow:

* Данная модель отлично подходит для организации рабочего процесса на основе релизов.
* Работа по модели Gitflow предусматривает создание специальной ветки для исправления ошибок в рабочем релизе.

Последовательность действий при работе по модели Gitflow:

1. Из ветки main создается ветка develop.
2. Из ветки develop создается ветка release.
3. Из ветки develop создаются ветки feature.
4. Когда работа над веткой feature завершается, она сливается в ветку develop.
5. Когда работа над веткой release завершается, она сливается с ветками develop и main.
6. Если в ветке main обнаруживается проблема, из main создается ветка hotfix.
7. Когда работа над веткой hotfix завершается, она сливается с ветками develop и main.

**- Чем git fetch отличается от git pull**

Загрузить содержимое из удаленного репозитория можно с помощью двух команд: git pull и git fetch. Из этих двух команд git fetch можно считать «безопасным» вариантом. Она загружает удаленное содержимое, но не обновляет рабочее состояние локального репозитория, оставляя текущую работу нетронутой. Команда git pull действует более агрессивно: она загружает удаленное содержимое для активной локальной ветки и сразу выполняет команду git merge, создавая коммит слияния для нового удаленного содержимого. Если у вас есть ожидающие изменения, то возникнут конфликты, и будет запущен процесс разрешения конфликтов слияния.

Рассмотренная команда git fetch — это основная команда, которая используется для загрузки содержимого из удаленного репозитория. Чтобы обновить локальный репозиторий до состояния удаленного репозитория, команда git fetch используется в сочетании с командами git remote, git branch, git checkout и git reset. Команда git fetch представляет собой важную часть рабочих процессов, связанных с совместной работой в Git. Поведение git fetch сходно с поведением команды git pull, однако git fetch можно считать более безопасным вариантом без разрушения информации.

Команда git fetch связывается с удалённым репозиторием и забирает из него все изменения, которых у вас пока нет и сохраняет их локально.

**- Для чего нужен rebase**

**Золотое правило для команды git rebase — никогда не использовать ее в *публичных* ветках.**

В первую очередь нужно понимать, что команда git rebase помогает решить ту же проблему, что и команда git merge. Обе команды предназначены для включения изменений из одной ветки в другую, но делают это по-разному.

Для включения новых коммитов в свою функциональную ветку feature можно использовать два варианта: слияние или перебазирование.

Способ слияния (merge)

Проще всего слияние ветки main в функциональную ветку выполняется с помощью следующей команды:

git checkout feature

git merge main

При желании этот код можно записать в одну строку:

git merge feature main

Вместо слияния можно выполнить перебазирование функциональной ветки feature на главную ветку main с помощью следующих команд:

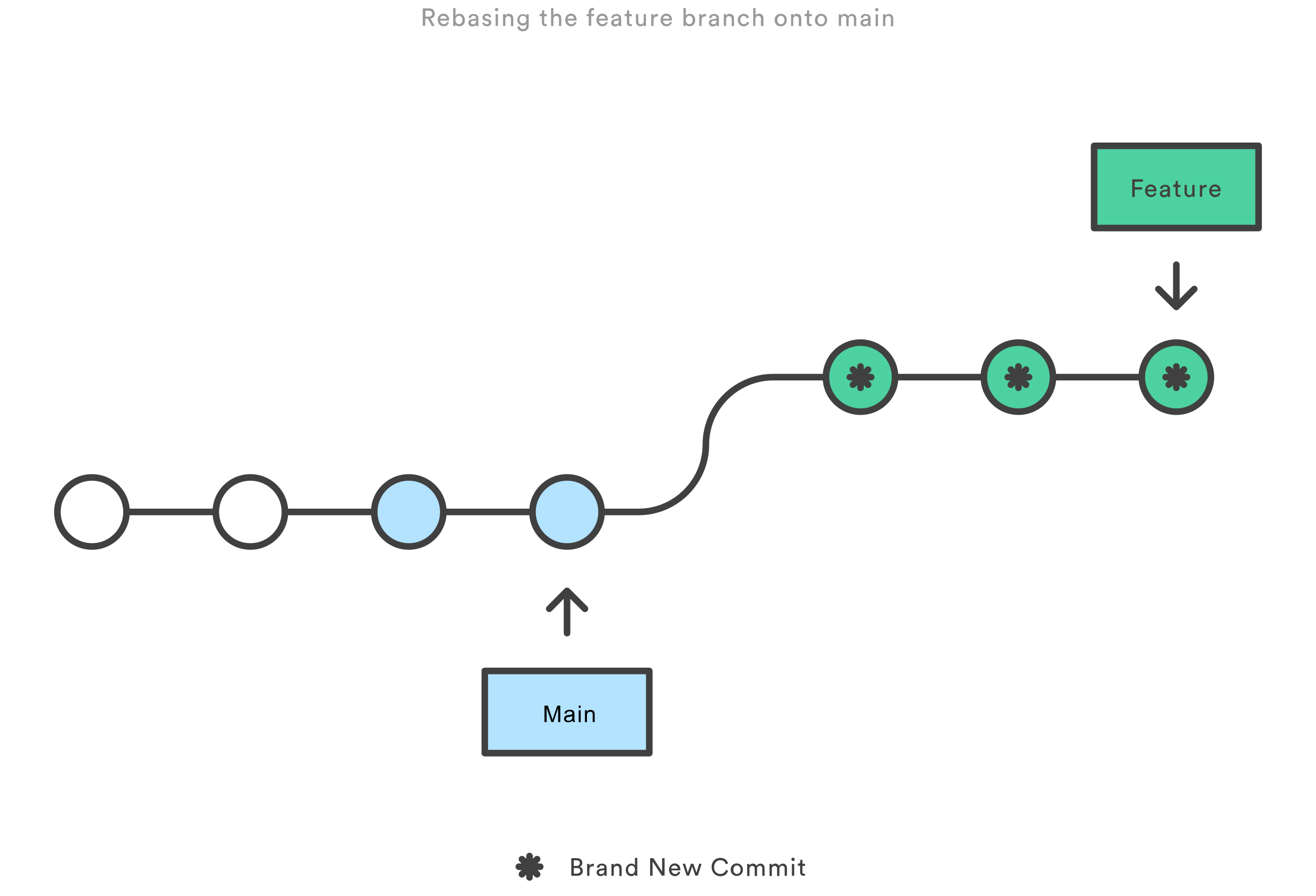
git checkout feature

git rebase main

В результате вся функциональная ветка feature окажется поверх главной ветки main, включая в себя все новые коммиты в ветке main. Если вместо команды merge при коммитах используется rebase, эта команда перезаписывает историю проекта, создавая новые коммиты для каждого коммита в исходной ветке.

Перебазирование функциональной ветки на основную

Главное преимущество rebase — более чистая история проекта. Во-первых, эта команда устраняет ненужные коммиты слияния, необходимые для git merge. Во-вторых, как показано на рисунке выше, команда rebase создает идеальную линейную историю проекта — вы сможете отследить функционал до самого начала проекта без каких-либо форков. Это упрощает навигацию в проекте с помощью таких команд, как git log, git bisect и gitk.



Однако такая безупречная история коммитов требует определенных жертв: жертвовать приходится безопасностью и отслеживаемостью. Если не следовать Золотому правилу Rebase, перезапись истории проекта может обернуться катастрофическими последствиями для совместных рабочих процессов. Кроме того, при выполнении rebase теряется контекст, доступный в коммите со слиянием: вы не сможете увидеть, когда вышестоящие изменения были включены в функционал