Система контроля версий Git

Оглавление

[1. Система контроля версий 5](#_Toc62218789)

[1.1. Что такое «Система контроля версий» и почему это важно? 5](#_Toc62218790)

[1.2. Виды систем контроля версий 5](#_Toc62218791)

[1.2.1. Локальные системы контроля версий 5](#_Toc62218792)

[1.2.2. Централизованные системы контроля версий 6](#_Toc62218793)

[2.1.3. Распределённые системы контроля версий 7](#_Toc62218794)

[2. Git 8](#_Toc62218795)

[2.1. Архитектура Git и ее особенности 8](#_Toc62218796)

[2.1.1. Хранение информации об изменениях в виде снимков, а не различий 8](#_Toc62218797)

[2.1.2. Локальное выполнение почти всех операций 10](#_Toc62218798)

[2.1.3. Вычисление хеш суммы для всего 11](#_Toc62218799)

[2.1.4. Добавление новых данных при выполнений любых действий 11](#_Toc62218800)

[2.2. Состояния файлов в Git 11](#_Toc62218801)

[2.3. Файловая система Git 12](#_Toc62218802)

[2.4. Объекты Git 14](#_Toc62218803)

[2.4.1. Блобы(Blobs) 14](#_Toc62218804)

[2.4.2. Дерево(Trees) 14](#_Toc62218805)

[2.4.3. Коммиты (Commits) 14](#_Toc62218806)

[2.4.4. Ссылки (References) 15](#_Toc62218807)

[3. Установка Git 17](#_Toc62218808)

[3.2. Проверка установки Git 18](#_Toc62218809)

[3.3. Настройка Git 18](#_Toc62218810)

[5. Gitflow 21](#_Toc62218811)

[5.1. Основные ветки (Master) и ветки разработки (Develop) 21](#_Toc62218812)

[5.2. Функциональные ветки (Feature) 22](#_Toc62218813)

[5.3. Ветки выпуска (Release) 22](#_Toc62218814)

[5.4. Ветки исправления (Hotfix) 23](#_Toc62218815)

[5.5. Последовательность действий при работе по модели Gitflow 24](#_Toc62218816)

[4. Работа с Git 27](#_Toc62218817)

[4.1. Жизненный цикл Git 27](#_Toc62218818)

[4.2. Команды Git 28](#_Toc62218819)

[4.3. Конфигурирование – git config 29](#_Toc62218820)

[4.3.1. Установка имени и электронной почты 30](#_Toc62218821)

[4.3.2. Установка редактора 31](#_Toc62218822)

[4.3.3. Установка имени ветки по умолчанию 32](#_Toc62218823)

[4.3.4. Установка параметров окончаний строк 33](#_Toc62218824)

[4.3.5. Установка отображения Unicode 37](#_Toc62218825)

[4.3.6. Установка цветового выделения 38](#_Toc62218826)

[4.3.7. Установка псевдонимов 40](#_Toc62218827)

[4.3.8. Проверка настроек 41](#_Toc62218828)

[4.4. Создание репозитория 42](#_Toc62218829)

[4.4.1. Создание нового репозитория - git init 42](#_Toc62218830)

[4.4.2. Клонирование существующего репозитория - git clone 43](#_Toc62218831)

[4.5. Просмотр информации 44](#_Toc62218832)

[4.5.1. Просмотр состояния файлов в репозитории – git status 44](#_Toc62218833)

[4.5.2. Просмотр конкретных изменений – git diff 46](#_Toc62218834)

[4.5.3. Просмотр истории коммитов – git log 48](#_Toc62218835)

[4.5.4. Просмотр изменений, внесенных отдельным коммитом – git show 56](#_Toc62218836)

[4.5.5. Просмотр информацию о последнем коммите – git blame 57](#_Toc62218837)

[4.5.6. Поиск слов по проекту, состоянию проекта в прошлом – git grep 58](#_Toc62218838)

[4.6. Локальный репозиторий 59](#_Toc62218839)

[4.6.1. Добавление изменений в индекс – git add 59](#_Toc62218840)

[4.6.2. Фиксация изменений – git commit 60](#_Toc62218841)

[4.6.3. Временное сохранение изменений без коммита – git stash 61](#_Toc62218842)

[4.6.4. Отмена коммитов – git revert 62](#_Toc62218843)

[4.6.5. Сброс текущего HEAD – git reset 63](#_Toc62218844)

[4.6.6. Восстановление файлов – git restore 64](#_Toc62218845)

[4.6.7. Удаление файлов из индекса и рабочей директории – git rm 65](#_Toc62218846)

[4.6.8. Удаление неотслеживаемых файлов – git clean 66](#_Toc62218847)

[4.6.9. Перемещение/переименование файлов – git mv 67](#_Toc62218848)

[4.6.10. Тегирование – git tag 68](#_Toc62218849)

[4.7. Удаленный репозиторий 69](#_Toc62218850)

[4.7.1. Работа с удаленным репозиторием – git remote 69](#_Toc62218851)

[4.7.2. Отправка изменений в удаленный репозиторий – git push 70](#_Toc62218852)

[4.7.3. Копирование изменений из удаленного репозитория – git pull 72](#_Toc62218853)

[4.7.4. Получение изменений из удаленного репозитория без слияния – git fetch 73](#_Toc62218854)

[4.8.Ветвление 74](#_Toc62218855)

[4.8.1. Создание ветки – git branch 75](#_Toc62218856)

[4.8.2. Переключение ветки – git checkout 77](#_Toc62218857)

[4.8.3. Слияние веток – git merge 78](#_Toc62218858)

[4.8.4. Перемещение коммитов в ветку – git rebase 80](#_Toc62218859)

[4.8.5. Перемещение коммитов – git cherry-pick 81](#_Toc62218860)

[4.9. Игнорирование файлов – **.gitignore** 82](#_Toc62218861)

# 1. Система контроля версий

## 1.1. Что такое «Система контроля версий» и почему это важно?

**Система контроля версий** —  программное обеспечение, призванное автоматизировать работу с историей файла (или группы файлов), обеспечить мониторинг изменений, синхронизацию данных и организовать защищенное хранилище проекта.

Основная задача систем управления версиями – упростить работу с изменяющейся информацией, то записывать изменения в файл или набор файлов в течение времени и давать возможность вернуться позже к определённой версии.

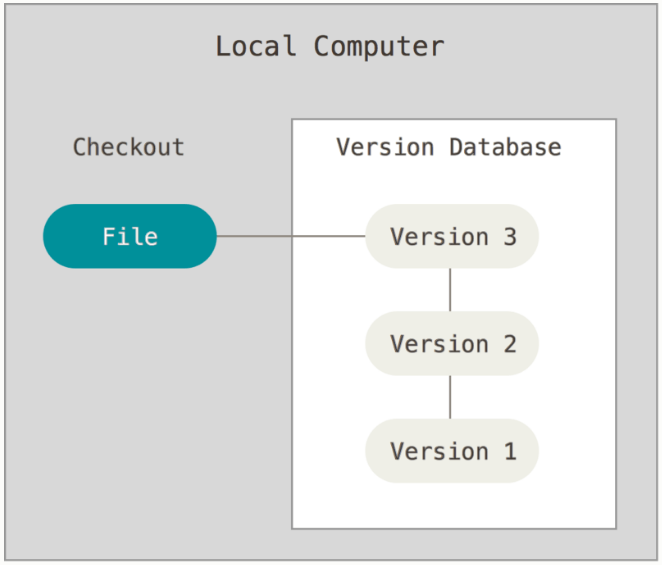
Если вы графический или web-дизайнер и хотите сохранить каждую версию изображения или макета (скорее всего, захотите), система контроля версий (далее СКВ) — как раз то, что нужно. Она позволяет вернуть файлы к состоянию, в котором они были до изменений, вернуть проект к исходному состоянию, увидеть изменения, увидеть, кто последний менял что-то и вызвал проблему, кто поставил задачу и когда и многое другое. Использование СКВ 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.

## 1.2. Виды систем контроля версий

### 1.2.1. Локальные системы контроля версий

Многие люди в качестве метода контроля версий применяют копирование файлов в отдельную директорию (возможно даже, директорию 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 какой директории вы находитесь, и случайно изменить не тот файл или скопировать не те файлы, которые вы хотели.

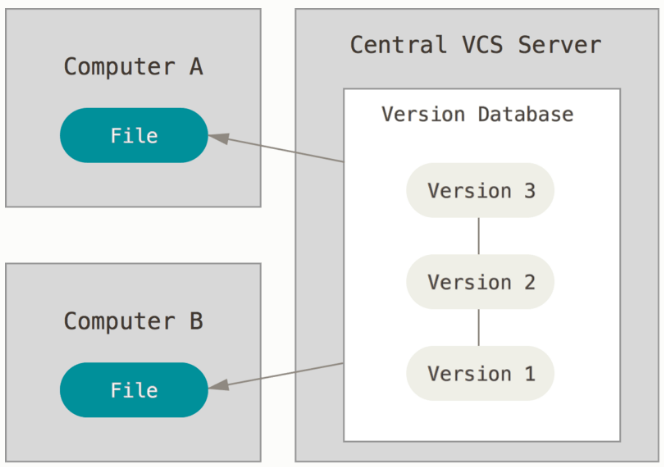
Для того, чтобы решить эту проблему, программисты давным-давно разработали локальные СКВ с простой базой данных, которая хранит записи о всех изменениях в файлах, осуществляя тем самым контроль ревизий.



Одной из популярных СКВ была система RCS, которая и сегодня распространяется со многими компьютерами. RCS хранит на диске наборы патчей (различий между файлами) в специальном формате, применяя которые она может воссоздавать состояние каждого файла в заданный момент времени.

### 1.2.2. 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 (ЦСКВ). 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 ЦСКВ являлось стандартом на протяжении многих лет.

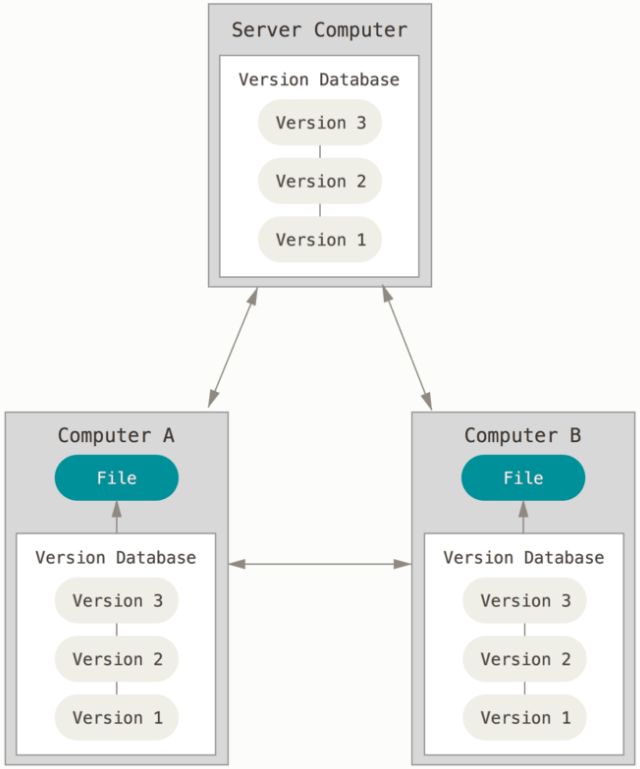


Такой подход имеет множество преимуществ, особенно перед локальными СКВ. Например, все разработчики проекта в определённой степени знают, чем занимается каждый из них. Администраторы имеют полный контроль над тем, кто и что может делать, и гораздо проще администрировать ЦСКВ, чем оперировать локальными 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 СКВ страдают от той же самой проблемы: когда вся история проекта хранится в одном месте, вы рискуете потерять всё.

### 2.1.3. Распределённые системы контроля версий

Здесь в игру вступают распределённые системы контроля версий (РСКВ). В РСКВ (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 РСКВ 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.

# 2. Git

**Git** – распределённая система контроля версий, позволяющая сохранять изменения, внесённые в файлы, которые хранятся в репозитории. Сами изменения сохраняются в виде снимков, называемых коммитами. Они могут размещаться на разных серверах, поэтому вы всегда восстановите код в случае сбоя, а также без проблем откатитесь до любого предыдущего состояния.

Проект был создан Линусом Торвальдсом для управления разработкой ядра Linux, первая версия выпущена 7 апреля 2005 года. На сегодняшний день его поддерживает Джунио Хамано.

## 2.1. Архитектура Git и ее особенности

Ядро Git представляет собой набор утилит командной строки с параметрами. Все настройки хранятся в текстовых файлах конфигурации. Такая реализация делает Git легко портируемым на любую платформу и даёт возможность легко интегрировать Git в другие системы (в частности, создавать графические git-клиенты с любым желаемым интерфейсом).

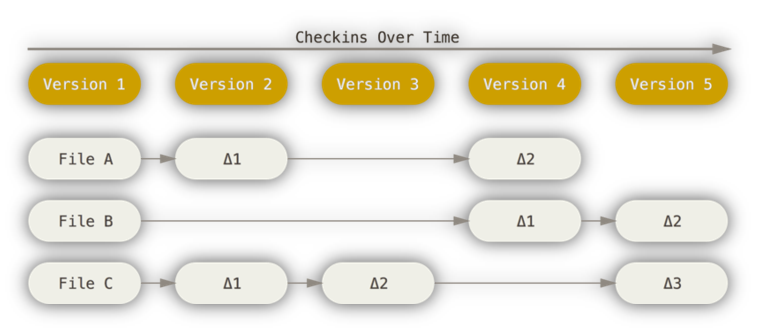
**Репозиторий Git** представляет собой каталог файловой системы, в котором находятся файлы конфигурации репозитория, файлы журналов, хранящие операции, выполняемые над репозиторием, индекс, описывающий расположение файлов, и хранилище, содержащее собственно файлы.

Структура хранилища файлов не отражает реальную структуру хранящегося в репозитории файлового дерева, она ориентирована на повышение скорости выполнения операций с репозиторием. Когда ядро обрабатывает команду изменения (неважно, при локальных изменениях или при получении патча от другого узла), оно создаёт в хранилище новые файлы, соответствующие новым состояниям изменённых файлов. Существенно, что никакие операции не изменяют содержимого уже существующих в хранилище файлов.

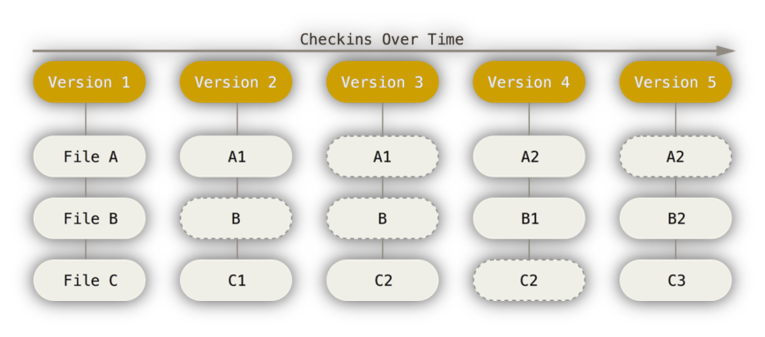
### 2.1.1. Хранение информации об изменениях в виде снимков, а не различий

Основное отличие Git от любой другой СКВ (включая Subversion и её собратьев) — это подход к работе со своими данными. Концептуально, большинство других систем хранят информацию в виде списка изменений в файлах.

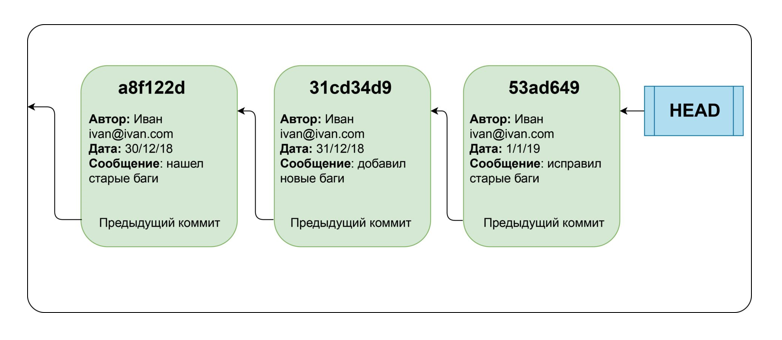
Эти системы (CVS, Subversion, Perforce, Bazaar и т.д.) представляют хранимую информацию в виде набора файлов и изменений, сделанных в каждом файле, по времени (обычно это называют контролем версий, основанным на различиях).



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



Это очень важное отличие между Git и почти любой другой СКВ. Git переосмысливает практически все аспекты контроля версий, которые были скопированы из предыдущего поколения большинством других систем. Это делает Git больше похожим на миниатюрную файловую систему с удивительно мощными утилитами, надстроенными над ней, нежели просто на СКВ.



### 2.1.2. Локальное выполнение почти всех операций

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

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

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

### 2.1.3. Вычисление хеш суммы для всего

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

Механизм, которым пользуется Git при вычислении хеш-сумм, называется SHA-1 хеш. Каждый файл вашего проекта в Git состоит из имени и содержания. Имя – это первые 20 байтов данных, оно наглядно записывается строкой длинной в 40 шестнадцатеричных символов (0-9 и a-f), она вычисляется на основе содержимого файла или структуры каталога. SHA-1 хеш выглядит примерно так:

24b9da6552252987aa493b52f8696cd6d3b00373

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

Кроме того, в репозитории существует каталог refs, который позволяет задать читаемые человеком имена для каких-то объектов Git. В командах Git оба вида ссылок — читаемые человеком из refs, и нижележащие SHA-1 — полностью взаимозаменяемы.

### 2.1.4. Добавление новых данных при выполнений любых действий

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

Всё это превращает использование Git в одно удовольствие, потому что мы знаем, что можем экспериментировать, не боясь серьёзных проблем. Для более глубокого понимания того, как Git хранит свои данные и как вы можете восстановить данные, которые кажутся утерянными

## 2.2. Состояния файлов в Git

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

* Неотслеживаемое (untracked)

**Неотслеживаемый файл** – файл, который создан и не добавлен в репозиторий.

* Изменённое (modified)

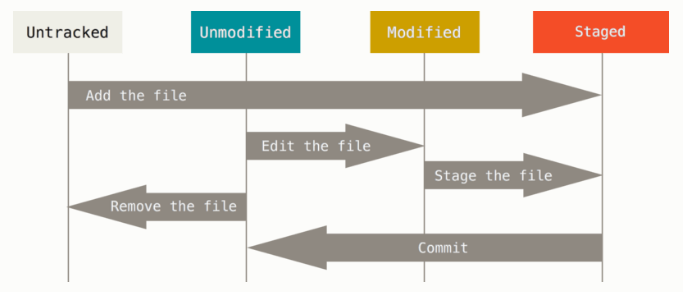
**Измененный файл** – файл, который уже добавлен в гит репозиторий и в котором сделаны изменения.

* Подготовленное (staged)

**Подготовленный файл** — файл, который уже изменен и отмечен для включения в следующий коммит.

* Зафиксированное (committed),

**Зафиксированный файл** – файл, который уже подготовлен для коммита и который попадает в коммит и переходит в гит репозиторий.



Как видно на картинке файлы могут быть Не отслеживаемые (Untracked) и отслеживаемые. Отслеживаемые файлы могут находится в 3 состояниях: Не измененный (Unmodified), Измененный (Modified), Подготовленный (Staged).

Если вы добавляете (с помощью git add) Не отслеживаемый файл, то он переходит в состояние Подготовленный.

Если вы изменяете файл в состояния Не измененный, то он переходит в состояние Измененный.

Если вы сохраняете файл, находящийся в состоянии Измененный, то он переходит в состояние Подготовленный.

Если вы коммитите файла, находящийся в состоянии Подготовленный, то он переходит в состояние Не измененный.

Если версии файла в HEAD и рабочей директории отличаются, то файл будет находится в состоянии Измененный, иначе (если версия в HEAD и в рабочем каталоге одинакова) файл будет находится в состояний Не измененный.

Если версия файла в HEAD отличается от рабочего каталога, но не отличается от версии в индексе, то файл будет в состоянии Подготовленный.

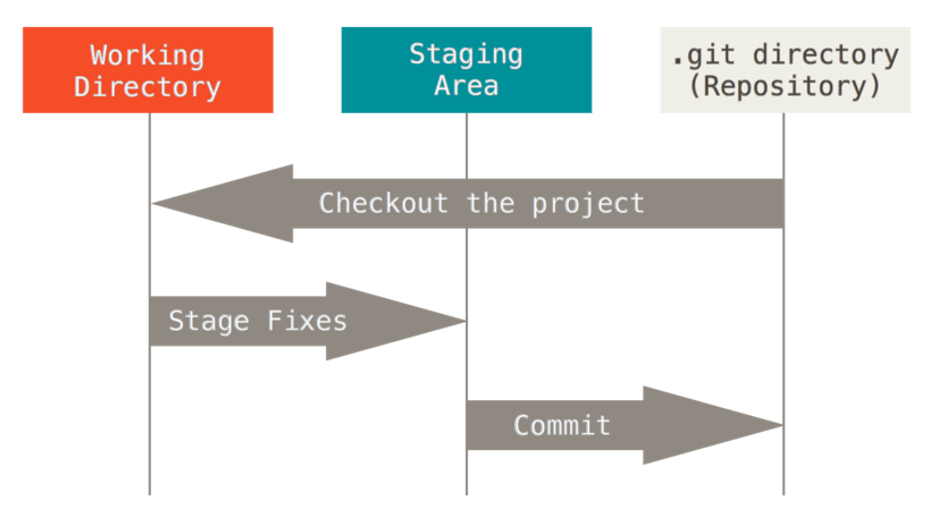
Этот цикл можно представить следующим образом:

Unmodified -> Modified -> Staged -> Unmodified

То есть вы изменяете файл сохраняете его в индексе и делаете коммит и потом все сначала.

## 2.3. Файловая система Git

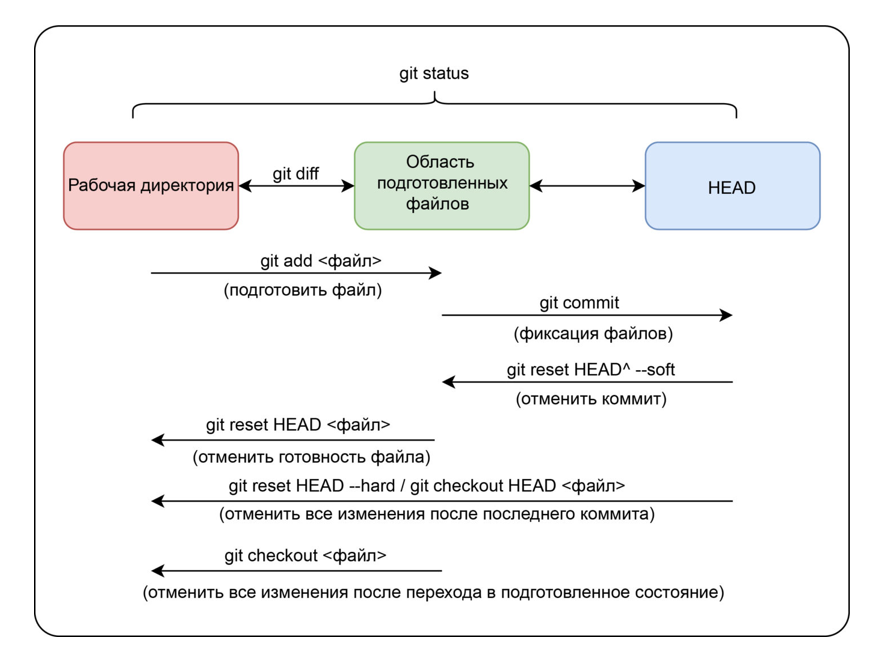
Мы подошли к трём основным секциям проекта Git: Git-директория (Git directory), рабочая директория (working directory) и область подготовленных файлов (staging area).



**Рабочая директория (каталог)** является снимком версии проекта. Файлы распаковываются из сжатой базы данных в Git-директории и располагаются на диске, для того чтобы их можно было изменять и использовать.

**Область подготовленных файлов (индекс)** — это файл, обычно располагающийся в вашей Git-директории, в нём содержится информация о том, какие изменения попадут в следующий коммит. Эту область ещё называют индекс, однако называть её stage-область также общепринято.

**Git-директория (локальный репозиторий)** — это то место, где Git хранит метаданные и базу объектов вашего проекта. Это самая важная часть Git, и это та часть, которая копируется при клонировании репозитория с другого компьютера.



## 2.4. Объекты Git

Работу с версиями файлов в Git можно сравнить с обычными операциями над файловой системой. Структура состоит из четырех типов объектов: Blob, Tree, Commit и References; некоторые из них, в свою очередь, делятся на подобъекты.

### 2.4.1. Блобы(Blobs)

**Блоб (Blob – Binary Large Object)** – тип данных, который вмещает лишь содержимое файла и собственный SHA1-хеш. Blob является основным и единственным носителем данных в структуре Git. Можно провести параллель между данным объектом и инодами (inodes) в файловых системах, поскольку их структура и цели во многом схожи.

Блоб означает большой двоичный объект. Каждая версия файла представлена большим двоичным объектом. Большой двоичный объект содержит данные файла, но не содержит метаданных о файле. Это двоичный файл, и в базе данных Git он называется хеш-кодом SHA1 этого файла. В Git к файлам не обращаются по именам. Все адресовано по содержанию.

### 2.4.2. Дерево(Trees)

**Дерево (Tree)** – тип данных, представляющий каталог, который содержит:

собственный SHA1-хеш;

SHA1-хеш blob’ов и/или деревьев;

права доступа Unix-систем;

символьное имя объекта (название для внутреннего использования в системе).

По своей сути объект является аналогом директории. Он задает иерархию файлов проекта.

### 2.4.3. Коммиты (Commits)

**Коммит (Commit)** – тип данных, хранящий текущее состояние репозитория, который содержит:

собственный SHA1-хеш;

ссылку ровно на одно дерево;

ссылку на предыдущий commit (их может быть и несколько);

имя автора и время создания commit’а;

имя коммитера (commiter – человек, применивший commit к репозиторию, он может отличаться от автора) и время применения commit’а;

произвольный кусок данных (блок можно использовать для электронной подписи или, например, для пояснения изменений commit’а).

Данный объект призван хранить снимок (версию) группы файлов в определенный момент времени, можно сравнить его с контрольной точкой. Commit’ы можно объединять (merge), разветвлять (branch) или, например, установить линейную структуру, тем самым отражая иерархию версий проекта.

### 2.4.4. Ссылки (References)

**Ссылка (Reference)** – тип данных, содержащий ссылку на любой из четырех объектов (Blob, Tree, Commit и References). Основная цель его – прямо или косвенно указывать на объект и являться синонимом файла, на который он ссылается. Тем самым повышается понимание структуры проекта. Очень неудобно оперировать бессмысленным набором символов в названии, ссылку же, в отличие от SHA1-хеша, можно именовать так, как удобнее разработчику.

Из ссылок, в свою очередь, можно выделить ряд подобъектов, имеющих некоторые различия: Ветвь, Тег. Рассмотрим их.

#### 2.4.4.1. Ветви (Head, Branches)

**Ветвь (Branch)** – символьная ссылка, которая указывает на последний в хронологии commit определенной ветви и хранит SHA1-хеш объекта.

Является типом данных журналируемых файловых систем. Данный вид объекта определяется не в самом Git, а наследуется от операционной и файловой систем. Ветвь используется как синоним файла, на который она ссылается, т.е. Git позволяет оперировать ею напрямую. Можно позволить себе не задумываться о том, работаете ли вы с последней версией или нет.

**HEAD** – указатель, который всегда указывает на последнюю фиксацию в ветке. Каждый раз, когда вы делаете коммит, HEAD обновляется последней фиксацией. Головы веток хранятся в каталоге .git/refs/heads/

#### 2.4.4.2. Теги (Tags)

**Тег (tag)** – тип данных, который в отличие от ветвей неизменно ссылается на один и тот же объект типа blob, tree, commit или tag.

Его, в свою очередь, можно разделить на легковесный (light tag) и тяжеловесный или аннотированный (annotated tag).

Легкий тег, кроме неизменности ссылки, ничем не отличается от обычных ветвей, т.е. содержит лишь SHA1-хеш объекта, на который ссылается, внутри себя.

Аннотированный тег состоит из двух частей:

первая часть содержит собственный SHA1-хеш;

вторая часть состоит из:

SHA1 объекта, на который указывает аннотированный тег;

тип указываемого объекта (blob, tree, commit или tag);

символьное имя тега;

дата и время создания тега;

имя и e-mail создателя тега;

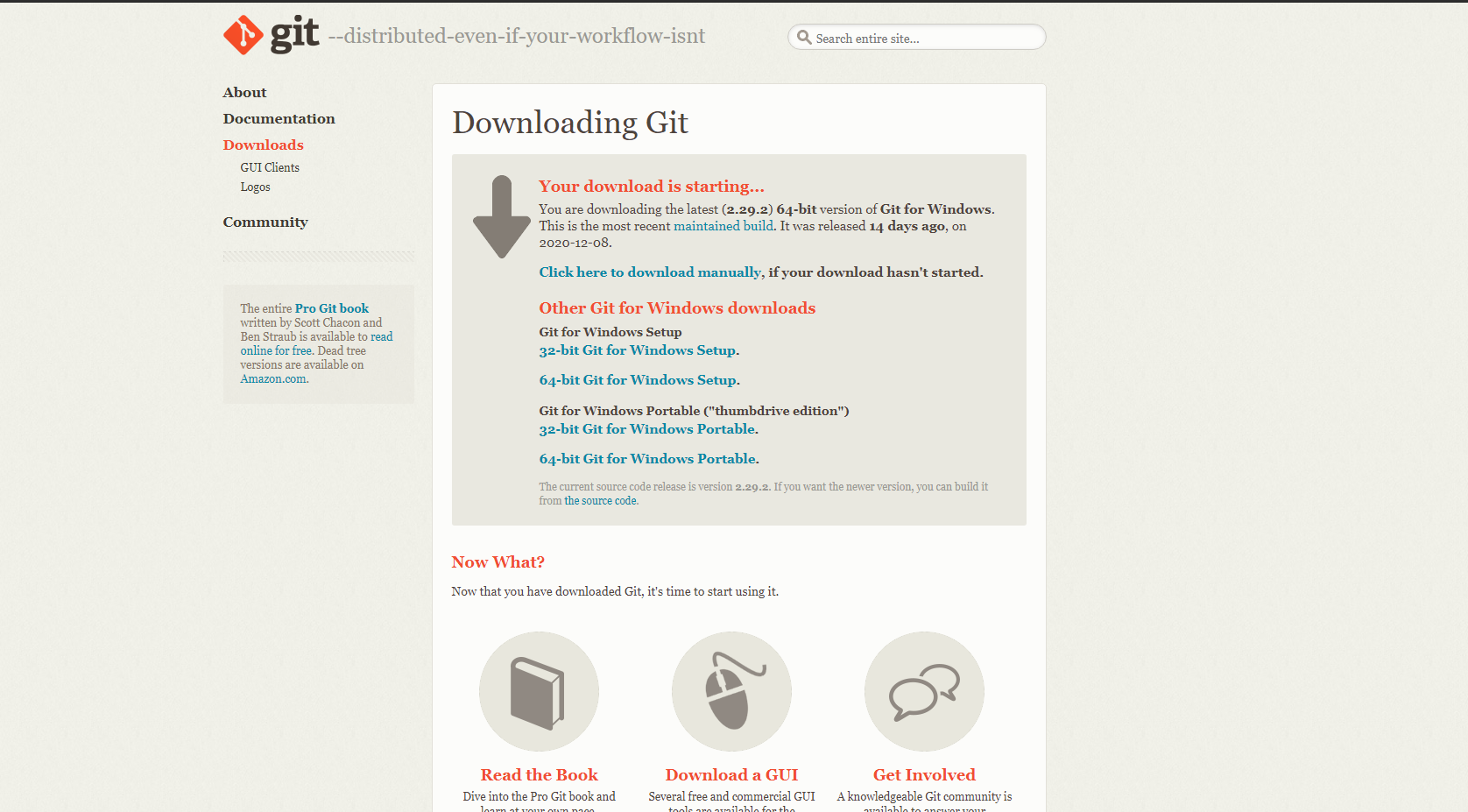
произвольный кусок данных (данный блок можно использовать для электронной подписи или для пояснения тега).

Иными словами, проект в Git представляет собой набор blob’ов, которые связаны сетью деревьев. Полученная иерархическая структура может, в зависимости от времени, быть отражена в виде commit’ов – версий, а для понимания их структуры в Git присутствуют такие объекты, как ссылки. Исключая действия со ссылками, почти вся работа с объектами системы максимально автоматизирована изнутри. Отталкиваясь от механизма ссылок, мы приходим к следующей идее – работать именно над группами файлов.

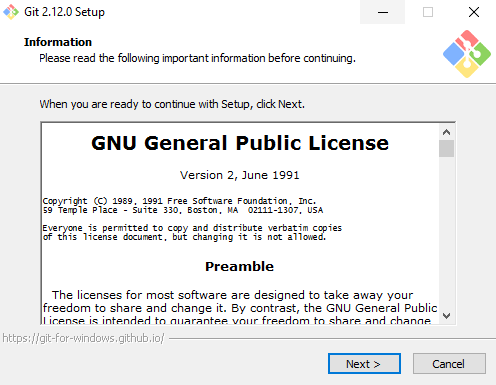
# 3. Установка Git

Шаги:

1. Скачать последнюю версию программы с сайта <https://git-scm.com/download/win>



1. Запустить программу установки



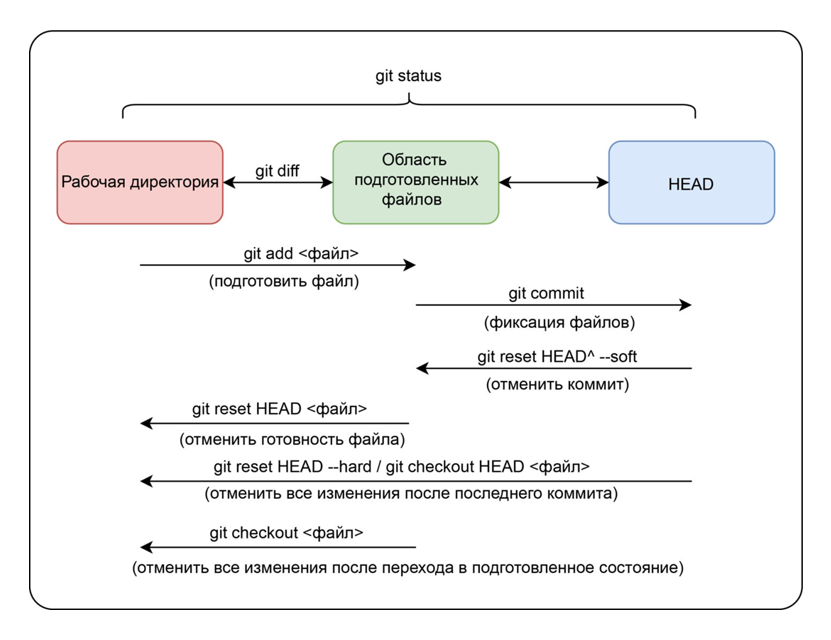
1. После успешного запуска программы установки отобразится экран мастера установки Git. Для завершения установки следуйте инструкциям, нажимая кнопки Далее и Готово. Параметры по умолчанию достаточно хорошо подходят для большинства пользователей.

## 3.2. Проверка установки Git

В командной строке набрать и выполнить команду git –version

## 3.3. Настройка Git

Настройка системы Git предполагает, в первую очередь, указание имени пользователя и e-mail, которые используются для подписи коммитов и отправки изменений в удаленный репозиторий.

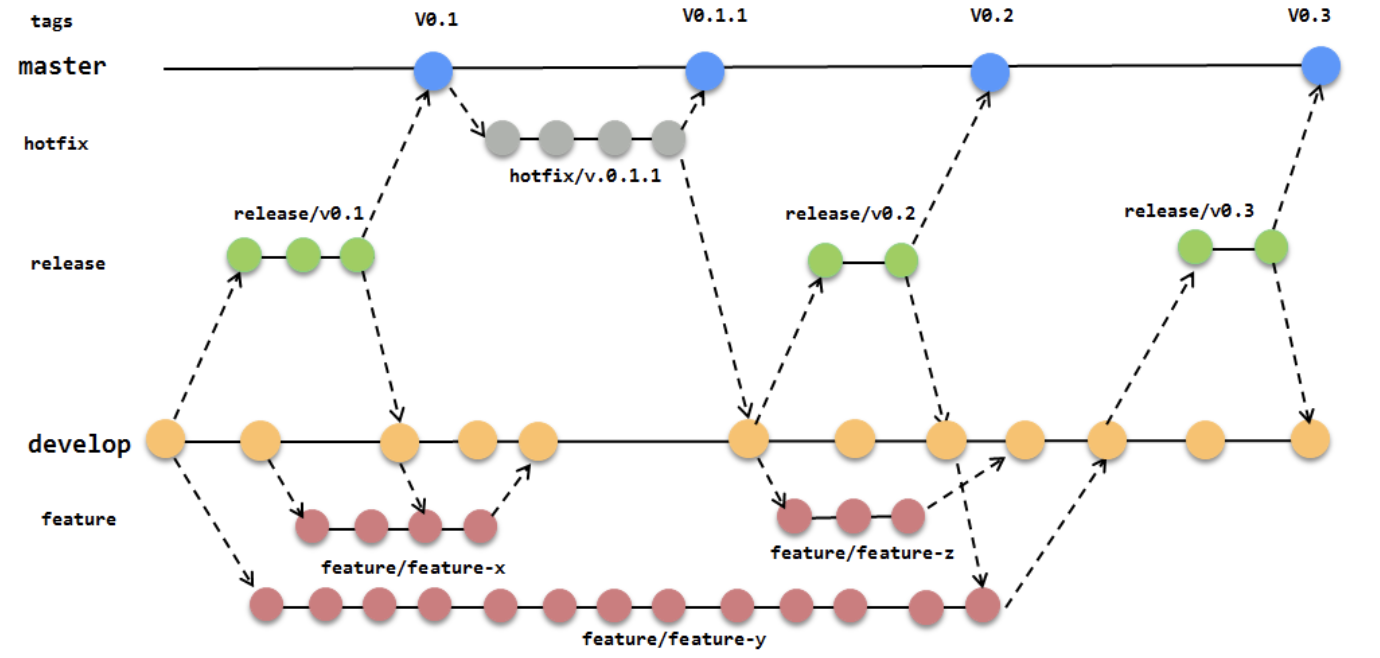


# 5. Gitflow

**Gitflow Workflow** — это модель рабочего процесса Git, которая была впервые опубликована и популяризована Винсентом Дриссеном из компании nvie. Gitflow Workflow предполагает выстраивание строгой модели ветвления с учетом выпуска проекта. Такая модель обеспечивает надежную основу для управления крупными проектами.

Gitflow — это лишь методология работы с Git, то есть в ней определяется, какие виды веток необходимы проекту и как выполнять слияние между ними.

<https://www.atlassian.com/ru/git/tutorials/comparing-workflows/gitflow-workflow>



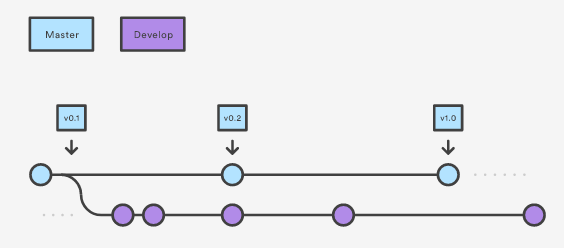
Ветки:

* Master
* Dev
* Feature
* Release
* Hotfix

## 5.1. Основные ветки (Master) и ветки разработки (Develop)

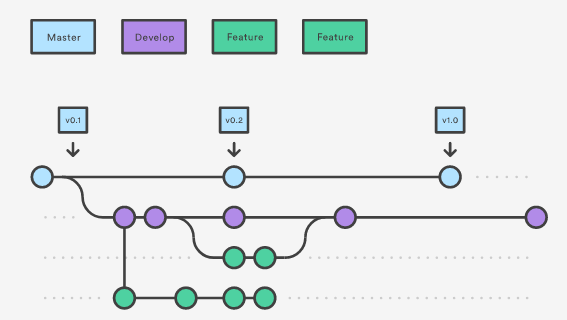
Для фиксации истории проекта в рамках этого процесса вместо одной ветки master используются две ветки. В ветке master хранится официальная история релиза, а ветка develop предназначена для объединения всех функций. Кроме того, для удобства рекомендуется присваивать всем коммитам в ветке master номер версии.

Первый шаг рабочего процесса заключается в создании ветки develop от стандартной ветки master. В этой ветке будет храниться полная история проекта, а в ветке master — сокращенная. Теперь другим разработчикам следует клонировать центральный репозиторий и создать отслеживающую ветку для ветки develop.



## 5.2. Функциональные ветки (Feature)

Под каждую новую функцию должна быть отведена собственная ветка, которую можно отправлять в центральный репозиторий для создания резервной копии или совместной работы команды. Ветки feature создаются не на основе master, а на основе develop. Когда работа над функцией завершается, соответствующая ветка сливается обратно с веткой develop. Функции не следует отправлять напрямую в ветку master.



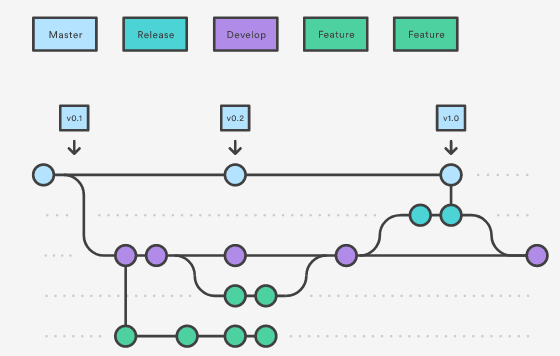
Как правило, ветки feature создаются на основе последней ветки develop.

## 5.3. Ветки выпуска (Release)

Когда в ветке develop оказывается достаточно функций для выпуска (или приближается назначенная дата выпуска), от ветки develop создается ветка release. Создание этой ветки запускает следующий цикл выпуска, и с этого момента новые функции добавить больше нельзя — допускается лишь отладка багов, создание документации и решение других задач, связанных с выпуском. Когда подготовка к поставке завершается, ветка release сливается с master и ей присваивается номер версии. Кроме того, для нее нужно выполнить 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.

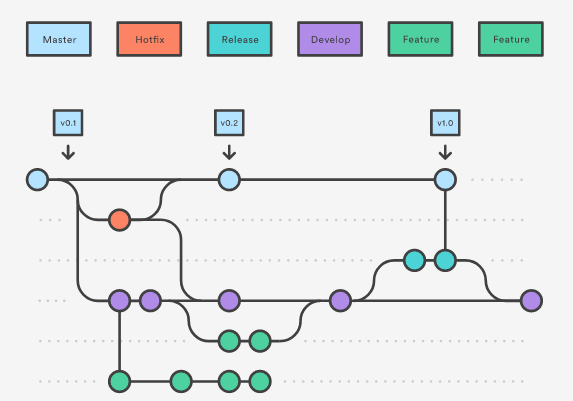
Создание веток release — это еще одна простая операция ветвления. Как и ветки feature, ветки release основаны на ветке develop.



## 5.4. Ветки исправления (Hotfix)

Ветки поддержки или ветки hotfix используются для быстрого внесения исправлений в рабочие релизы. Ветки hotfix очень похожи на ветки release и feature, за исключением того, что они создаются от master, а не от develop. Это единственная ветка, которая должна быть создана непосредственно от master. Как только исправление завершено, ветку следует объединить с master и develop (или текущей веткой release), а ветка master должна быть помечена обновленным номером версии.

Наличие специальной ветки для исправления ошибок позволяет команде решать проблемы, не прерывая остальную часть рабочего процесса и не ожидая следующего цикла релиза. Ветки поддержки можно рассматривать как специальные ветки release, которые работают непосредственно с master.



## 5.5. Последовательность действий при работе по модели Gitflow

1. Из ветки master создается ветка develop.
2. Из ветки develop создается ветка release.
3. Из ветки develop создаются ветки feature.
4. Когда работа над веткой feature завершена, она сливается с веткой develop.
5. Когда работа над веткой релиза release завершена, она сливается в ветки develop и master.
6. Если в master обнаружена проблема, из master создается ветка hotfix.
7. Когда работа над веткой исправления hotfix завершена, она сливается в ветки develop и master.

# 4. Работа с Git

## 4.1. Жизненный цикл Git

Общий рабочий процесс выглядит следующим образом

* клонируется репозиторий Git как рабочая копия;
* вносятся изменения в файлы в рабочей копии;
* выборочно добавляются в индекс только те изменения, которые должны попасть в следующий коммит;
* снимок сохраняется в Git-директорию (используются файлы из индекса как есть);
* при необходимости также обновляется рабочая копия и принимаются изменения других разработчиков;
* просматриваются изменения перед фиксацией;
* фиксируются изменения;
* если все в порядке, то изменения отправляются в репозиторий;
* после фиксации, если что что-то не так, то исправляется последняя фиксация и изменения отправляются в репозиторий.

Ниже показано графическое изображение рабочего процесса.



## 4.2. Команды Git

С Git можно работать и через графический интерфейс (например, GitHub Desktop), и через командную строку. Командную строку изучить необходимо хотя бы потому, что она предоставляет больше возможностей, чем некоторые инструменты с интерфейсом.

Обычно, **команды Git** имеют следующий вид:

git [команда][опции][аргументы]

В качестве аргумента может быть путь к файлу. Также у команд бывают опции, обозначаемые –[опция] либо –[однобуквенная опция]. Они обеспечивают более детальную настройку действия команды.

В нашем материале команды будут представлены в общем виде, а значит всё, что будет в [], вы можете менять на собственные значения.

Кстати, если возникают затруднения с использованием той либо иной команды, рекомендуется открыть руководство посредством

git help [команда]

Если просто нужно напоминание, применяйте

git [команда] *-h*

либо

git [команда] *--help*

В Git -h и --help имеют одинаковое значение.

Пример

git add -h

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

    -n, --dry-run         dry run

    -v, --verbose         be verbose

    -i, --interactive     interactive picking

    -p, --patch           select hunks interactively

    -e, --edit            edit current diff and apply

    -f, --force           allow adding otherwise ignored files

    -u, --update          update tracked files

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

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

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

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

    --refresh             dont add, only refresh the index

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

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

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

## 4.3. Конфигурирование – git config

После того, как Git установлен в 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, можно поменять их в любой момент, выполнив те же команды снова.

Настройки хранятся в конфигурационных файлах и могут быть сохранены в трёх местах:

* уровень системы – файл **/etc/gitconfig**;

Настройки, общие для всех пользователей системы и для всех их репозиториев.

Расположение в Windows – **\Program Files\Git\mingw64\etc\gitconfig**.

* уровень пользователя – файл **~/.gitconfig** или **~/.config/git/config**;

Настройки конкретного пользователя.

Расположение в Windows – **%HOMEPATH%\.gitconfig**.

* уровень репозитория – файл **.git/config** (т.е. config в каталоге Git) репозитория, который вы используете в данный момент;

Настройки конкретного репозитория.

Расположение в Windows – **папка\_с\_проектом\.git\config**.

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

Чтобы просматривать и настраивать параметры, контролирующие все аспекты работы Git, а также его внешний вид нужно выполнить команду **git config**.

git config

При выполнении команды необходимо указать уровень настроек:

* уровень системы:

git config *--system*

* уровень пользователя:

git config *--global*

* уровень приложения:

git config

После этой команды указывается параметр и его значение.

### 4.3.1. Установка имени и электронной почты

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

Чтобы указать имя, воспользуйтесь настройкой **user.name**

git config *--global* user.name [имя]

**Аргументы:**

* **имя** (обязательный);

В качестве параметра в команду user.name передается имя пользователя, которое будет использоваться для подписи коммита.

***Пример:***

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

**Результат выполнения команды:**

После выполнения этой команды будет установлено имя пользователя для подписи коммита.

Чтобы указать адрес электронной почты, воспользуйтесь настройкой **user.** **email.**

git config *--global* user.email [адрес электронной почты]

**Аргументы:**

* **адрес электронной почты** (обязательный);

В качестве параметра в команду user. email передается адрес электронной почты, который будет использоваться для подписи коммита.

***Пример:***

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

**Результат выполнения команды:**

После выполнения этой команды будет установлен адрес электронной почты для подписи коммита.

### 4.3.2. Установка редактора

В Git используется текстовый редактор, который будет применяться, если будет нужно набрать сообщение в Git. По умолчанию Git использует стандартный редактор вашей системы, которым обычно является Vim.

Чтобы изменить редактор по умолчанию, воспользуйтесь настройкой **core.editor**

git config *--global* core.editor [редактор]

**Аргументы:**

* **редактор** (обязательный);

В качестве параметра в команду core.editor передается название редактора, который будет использоваться Git.

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

***Пример:***

git config *--global*

core.editor "C:\Users\DKim\AppData\Local\Programs\Microsoft VS Code\Code.exe"

**Результат выполнения команды:**

После выполнения этой команды будет установлен редактор для Git.

### 4.3.3. Установка имени ветки по умолчанию

Когда инициализируете репозиторий командой git init, Git создаёт ветку с именем master по умолчанию. Начиная с версии 2.28, вы можете задать другое имя для создания ветки по умолчанию.

Чтобы изменить имя ветки по умолчанию при создании репозитория, воспользуйтесь настройкой **init.defaultBranch.**

git config *--global* init.defaultBranch [имя ветки]

**Аргументы:**

* **имя ветки** (обязательный);

В качестве параметра в команду init.defaultBranch передается имя ветки, которое будет использоваться при инициализации нового репозитория.

***Пример:***

git config *--global* init.defaultBranch main

**Результат выполнения команды:**

После выполнения этой команды будет установлено имя ветки по умолчанию для инициализации нового репозитория в Git.

### 4.3.4. Установка параметров окончаний строк

Проблемы форматирования и пробелов являются одними из самых неприятных и незаметных проблем, с которыми сталкиваются разработчики при совместной работе, особенно используя разные платформы. В Git есть несколько настроек, чтобы справиться с этими проблемами.

Если вы программируете в Windows и работаете с людьми, что не использует её (или наоборот), рано или поздно, вы столкнётесь с проблемами переноса строк. Это происходит потому, что Windows при создании файлов использует для обозначения переноса строки два символа «возврат каретки» и «перевод строки», в то время как Mac и Linux используют только один — «перевод строки».

Чтобы Git мог автоматически конвертировать переносы строк CRLF в LF при добавлении файла в индекс и наоборот — при извлечении кода, воспользуйтесь настройкой **core.autocrlf**.

git config *--global* core.autocrlf [значение]

**Аргументы:**

* **значение** (обязательный);

В качестве параметра в команду **core.autocrlf** передается значение true или input.

Для того, чтобы Git автоматически конвертировал **LF** окончания строк в **CRLF**, нужно установить значение **true.**

***Пример:***

git config *--global* core.autocrlf true

Для того, чтобы Git автоматически конвертировал **CRLF** окончания строк в **LF** нужно установить значение **input**.

***Пример:***

git config *--global* core.autocrlf input

**Результат выполнения команды:**

После выполнения этой команды будет установлен режим конвертации строк.

У Гита есть внутренний эвристический метод, который проверяет, двоичный ли файл. Если файл не двоичный, то Git считает его текстовым. Но Git иногда может ошибаться.

На тот случай, если Git ошибётся и изменит окончания строк там, где лучше было бы оставить их в покое, есть настройка **core.safecrlf**

git config *--global* core.safecrlf [значение]

**Аргументы:**

* **значение** (обязательный);

В качестве параметра в команду **core. safecrlf** передается значение true или warn.

Для того, чтобы Git проверил, что можно откатить изменения (из LF в CRLF), а если нет, то отменить операцию нужно установить значение **true.**

***Пример:***

git config *--global* core.safecrlf true

Для того, чтобы Git проверил, что можно откатить изменения (из LF в CRLF), а если нет, то просто предупредить о том, что может случиться что-то нехорошее нужно установить значение **warn.**

***Пример:***

git config *--global* core.safecrlf warn

**Результат выполнения команды:**

После выполнения этой команды будет установлен режим конвертации строк.

Файл .gitattributes.

Можно создать в корне репозитория файл **.gitattributes** и указать в нём настройки для конкретных файлов. Это позволит управлять такими настройками, как autocrlf для каждого типа файлов.

Чтобы Git заменил CRLF на LF во всех текстовых файлах:

\*.txt *crlf*

Чтобы Git никогда не заменял CRLF на LF в текстовых файлах:

\*.txt *-crlf*

Чтобы Гит заменял CRLF на LF в текстовых файлах только при записи в базу данных, но возвращал LF при записи в рабочий каталог:

\*.txt *crlf*=input

C Git версии 1.7.2 все настройки для окончаний строк определяются в файле **.gitattributes** вашего репозитория, инкапсулируя их внутри и делая независимыми от глобальных настроек.

Чтобы указать Гиту, в каких файлах надо заменить CRLF на LF, нужно с помощью атрибута **text** пометить типы файлов в **.gitattributes**

* **text** — устанавливает атрибут text для всех текстовых файлов;

Это значит, что Гит будет запускать процесс замены CRLF на LF каждый раз при записи в БД и делать обратную замену при выводе из базы данных в рабочий репозиторий.

* **-text** — снимет со всех текстовых файлов этот фильтр;

Это значит, что в указанных файлах не будет замены CRLF на LF.

* **text=auto** — установит для всех, подходящих под условие файлов, замену CRLF на LF, если Гит с помощью своего эвристического метода определит эти файлы как текстовые, а не бинарные.

Если файл не определён, Гит вернётся к старой системе и настройке core.autocrlf.

***Пример:***

*# Общие настройки, которые всегда должны быть использованы для настроек вашего языка.*

*# Автоматическое определение текстовых файлов и выполнение нормализации LF*

*# http://davidlaing.com/2012/09/19/customise-your-gitattributes-to-become-a-git-ninja/*

\* text=auto

*#*

*# Строка выше будет обрабатывать все файлы, которых нет в списке ниже.*

*#*

*# Documents*

\*.doc *diff*=astextplain

\*.DOC *diff*=astextplain

\*.docx *diff*=astextplain

\*.DOCX *diff*=astextplain

\*.dot *diff*=astextplain

\*.DOT *diff*=astextplain

\*.pdf *diff*=astextplain

\*.PDF *diff*=astextplain

\*.rtf *diff*=astextplain

\*.RTF *diff*=astextplain

\*.md *text*

\*.tex *text*

\*.adoc *text*

\*.textile *text*

\*.mustache *text*

\*.csv *text*

\*.tab *text*

\*.tsv *text*

\*.sql *text*

*# Graphics*

\*.png *binary*

\*.jpg *binary*

\*.jpeg *binary*

\*.gif *binary*

\*.tif *binary*

\*.tiff *binary*

\*.ico *binary*

*# SVG по дефолту рассматривается как бинарный. Если вы хотите, чтобы он распознавался как текст, закомментируйте  следующую строку и раскомментируйте следующую за ней.*

\*.svg *binary*

*#\*.svg text*

\*.eps *binary*

*#*

*# Исключить файлы из экспорта*

*#*

.gitattributes *export-ignore*

.gitignore *export-ignore*

### 4.3.5. Установка отображения Unicode

По умолчанию, Git будет печатать не-ASCII символы в именах файлов в виде восьмеричных последовательностей \nnn.

Чтобы избежать нечитаемых строк, воспользуйтесь настройкой **core.quotepath**

git config *--global* core.quotepath [значение]

**Аргументы:**

* **значение** (обязательный);

В качестве параметра в команду **core. quotepath** передается значение on или off.

***Пример:***

git config *--global* core.quotepath off

**Результат выполнения команды:**

После выполнения этой команды будет установлен режим печати не-ASCII символы.

### 4.3.6. Установка цветового выделения

По умолчанию вывод не подсвечивается (не раскрашивается разными цветами), что сильно затрудняет чтение результатов (например, при использовании git diff).

Чтобы включить выделение цветом для Git в консоли, воспользуйтесь настройками **color**.\*

git config *--global* color.[ тип] [значение]

**Аргументы:**

* **тип** (обязательный);
  + ui;
  + branch;
  + status;
* **значение** (обязательный);
  + true;
  + false**;**
  + auto;

***Пример:***

git config *--global* color.ui true

git config *--global* color.status auto

git config *--global* color.branch auto

**Результат выполнения команды:**

После выполнения этой команды вывод в консоли будет раскрашен в разные цвета в зависимости от настроек.

Ручная настройка цветов вывода.

Можно вручную задать настройки для текущего пользователя в файле **.gitconfig**.

***Пример:***

[color]

    ui = auto

[color "branch"]

    current = yellow reverse

    local = yellow

    remote = green

[color "diff"]

    meta = yellow bold

    frag = magenta bold

    old = red bold

    new = green bold

[color "status"]

    added = yellow

    changed = green

    untracked = cyan

Это типичная популярная конфигурация.

Первая секция **[color]** использует настройку **ui = auto** это означает использовать автоматическую подсветку (автоматическое раскрашивание). Далее мы для некоторых команд используем свою раскраску. Например, чтобы задать раскраску для команды git status мы используем секцию **[color «status»]**, в которой задаем параметры и цвета. Например, **added = yellow** означает — выводить новые файлы (added) желтым (yellow) цветом.

### 4.3.7. Установка псевдонимов

Для того, чтобы не печатать каждую команду для Git целиком, можно настроить псевдонимы.

Чтобы настроить псевдонимы для Git, воспользуйтесь настройкой **alias**.

git config alias.[сокращенная команда] [полная команда(ы)]

**Аргументы:**

* **сокращенная команда** (необязательный);

В качестве параметра в команду config alias передается сокращенный вариант имени команды.

* **полная команда** (необязательный);

В качестве параметра в команду config alias передается полный вариант имени команды или команд.

***Пример:***

git config *--global* alias.last 'log -1 HEAD'

**Результат выполнения команды:**

После выполнения этой команды можно использовать сокращенный вариант команды. Как в примере выше, можно просматривать информацию о последнем коммите на текущей ветке с помощью **git last**.

Рекомендуется использовать следующие сокращения:

* st = status;
* ch = checkout;
* br = branch;
* mg = merge;
* cm = commit;
* reb = rebase;
* lg = «git log --pretty=format:'%h — %ar: %s'»;

### 4.3.8. Проверка настроек

Чтобы показать все настройки, которые Git найдёт, нужно выполнить команду **git config –list.**

git config *-list*

***Пример:***

git config *-list*

user.name=John Doe

user.email=johndoe@example.com

color.status=auto

color.branch=auto

color.interactive=auto

color.diff=auto

## 4.4. Создание репозитория

### 4.4.1. Создание нового репозитория - git init

Чтобы создать новый репозиторий нужно выполнить команду **git init**.

git init [имя каталога]

**Аргументы:**

* **имя каталога** (необязательный)

Указание в команде git init существующего каталога проекта приведет к исполнению инициализации, но только на уровне этого каталога проекта.

**Результат выполнения команды:**

Создание нового подкаталога .git в рабочем каталоге и новой главной ветки.

**.git** — это папка, которая хранит всю информацию о гит репозитории. Ее удалять не нужно

Команду git init выполняют только один раз для первоначальной настройки нового репозитория.

**Пример:**

mkdir project-dir

cd project-dir

git init

### 4.4.2. Клонирование существующего репозитория - git clone

Чтобы клонировать удаленный репозиторий, то есть создать копию оригинального проекта со всей его историей локально, нужно выполнить команду **git clone**.

git clone [адрес репозитория][имя каталога]

**Аргументы:**

* **адрес репозитория** (обязательный);

В качестве параметра в команду git clone передается URL-адрес репозитория.

Git поддерживает несколько различных сетевых протоколов и соответствующих форматов URL-адресов.

* **имя каталога** (необязательный);

В качестве параметра в команду git clone можно передать имя каталога, в который клонируется репозиторий.

**Опции:**

* **-o, --origin <имя репозитория>** – указание названия репозитория;

По умолчанию клонируется с названием origin.

* **-b, --branch <имя ветки>** – выбор ветки;

По умолчанию клонируется та ветка, на которую указывает HEAD.

* **--single-branch** – выбор только одной ветки;

По умолчанию клонируется та ветка, на которую указывает HEAD.

* **--no-tags** – отмена клонирования меток;

По умолчанию клонируются все метки.

**Результат выполнения команды:**

Создание копии (клонирования) удаленного репозитория.

Клонирование, как и команда git init, обычно выполняется один раз.

**Примеры:**

* клонирование репозитория, используя протокол http

git clone http://user@somehost:port/~user/repository/project.git

* клонирование репозитория с той же машины в директорию myrepo

git clone /home/username/project myrepo

* клонирование репозитория, используя безопасный протокол ssh

git clone ssh://user@somehost:port/~user/repository

* клонирование репозитория, используя собственный протокол git

git clone git://user@somehost:port/~user/repository/project.git/

* клонирование репозитория в указанный каталог;

git clone https://github.com/cyberspacedk/Git-commands.git FolderName

* клонирования репозитория в текущий каталог;

git clone https://github.com/cyberspacedk/Git-commands.git .

## 4.5. Просмотр информации

### 4.5.1. Просмотр состояния файлов в репозитории – git status

Файл в репозитории имеет четыре состояния:

* **untracked** – неотслеживаемый;

В рабочей директории находится файл, которого нет ни в HEAD, ни в области подготовленных файлов. Можно сказать, что Git о нём не знает.

* **modified** – изменён;

В рабочей директории находится более новая версия файла по сравнению с той, которая хранится в HEAD или в области подготовленных файлов (при этом изменения не находятся в следующем коммите).

* **staged** – подготовлен;

В рабочей директории и в области подготовленных файлов есть более новая версия файла, по сравнению с той, которая хранится в HEAD (при этом файл уже готов к коммиту).

* **unmodified** – без изменений;

Во всех разделах содержится одна версия файла, то есть в последнем коммите находится актуальная версия.

Чтобы вывести информацию о состоянии файлов репозитория (отслеживаемые, изменённые, новые файлы и пр.) нужно выполнить команду **git status**.

git status

**Опции:**

* **-s, --short** — краткий вывод изменений;
* **-b, --branch** — информация о ветке;

**Результат выполнения команды:**

Отображение следующей информации:

* состояние файлов в рабочей директории и индексе;
* файлы измененные, но не добавленные в индекс;
* файлы, ожидающие коммита в индексе;
* файлы с неразрешенными конфликтами слияния;
* файлы, игнорируемые git;
* подсказки о том, как изменить состояние файлов;
* имя текущей ветки;

**Примеры:**

* не отслеживаемые файлы;

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)

* не проиндексированные и не закоммиченные изменения;

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

* не проиндексированные и не закоммиченные изменения;

git status

On branch master

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

Changes to be committed:

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

    new file:   README

    modified:   CONTRIBUTING.md

Changes not staged for commit:

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

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

    modified:   CONTRIBUTING.md

### 4.5.2. Просмотр конкретных изменений – git diff

Если результат работы команды git status недостаточно информативен для вас и вам хочется знать, что конкретно поменялось, а не только какие файлы были изменены — вы можете использовать команду git diff.

Чтобы посмотреть не изменённые файлы, а непосредственно изменения, нужно выполнить команду **git diff**.

git diff ][имя файла] [имя ветки-источника][имя ветки-цели]

**Аргументы:**

* **имя файла** (необязательный)

В качестве параметра в команду git diff можно передать имя конкретного файла для сравнения.

* **имя ветки-источника** (необязательный)

В качестве параметра в команду git diff можно передать имя исходной ветки для сравнения.

* **имя ветки-цели** (необязательный)

В качестве параметра в команду git diff можно передать имя целевой ветки для сравнения.

**Опции:**

* **--base <имя файла>** — опция;
* **--cached** – разница c индексом:
* **--staged** – разница между индексом и последним коммитом (для сравнения области подготовленных файлов с HEAD);

Во время конфликта вы можете посмотреть какие различия в каких файлах имеются.

* **--ours** – разница до слияния и после;
* **--theirs** – разница сливаемой ветки до слияния и после;
* **--base** – разница с обеими ветками до слияния и после;
* **--color-words** – отображение отличия в словах;
* **--name-only** – отображение только имен файлов;

**Результат выполнения команды:**

Отображение разницы в файлах между различными файлами в различных областях.

**Примеры:**

* разница между рабочей директорией и индексом;

git diff

* разница между рабочей директорией и индексом в конкретном файле;

git diff index.html

* разница между рабочей директорией и последним коммитом;

git diff HEAD

* разница между рабочей директорией и предпоследним коммитом;

git diff HEAD^

* разница между текущей веткой и заданной;

git diff branchX

* разница между веткой master и веткой feature;

git diff master feature

* разница c индексом;

git diff *--cached*

* разница между индексом и последним коммитом;

git diff *--staged*

* отображение отличия в словах;

git diff *--color-words*

* отображение только имен файлов;

git diff *--name-only*

### 4.5.3. Просмотр истории коммитов – git log

После того, как вы создали несколько коммитов или же склонировали репозиторий с уже существующей историей коммитов, вероятно вам понадобится возможность посмотреть, что было сделано. То есть может потребоваться получить информацию об истории коммитов; коммитах, изменивших отдельный файл; коммитах за определенный отрезок времени и так далее.

Чтобы просмотреть историю предыдущих коммитов в обратном хронологическом порядке нужно выполнить команду **git log**.

git log [имя файла][имя ветки][диапазон редакций]

**Аргументы:**

* **имя файла** (необязательный);

В качестве параметра в команду git log можно передать имя файла.

* **имя ветки** (необязательный);

В качестве параметра в команду git log можно передать имя ветки.

* **диапазон редакций** (необязательный);

В качестве параметра в команду git log можно передать количество коммитов.

**Опции:**

* **-p** – отображение разницы для каждого коммита;
* **-<n>** – отображение истории изменения только последних n коммитов;
* **--stat** – отображение сокращённой статистики измененных файлов для каждого коммита;
* **--shortstat** – отображение только строку с количеством изменений/вставок/удалений для команды –stat;
* **--summary** – отображение информации по созданиям, переименованиям и правам доступа файлов;
* **--name-only** – отображение списка измененных файлов после информации о коммите;
* **--name-status** – отображение списка файлов, которые добавлены/изменены/удалены;
* **--abbrev-commit** - отображение только несколько символов SHA-1 чек-суммы вместо всех 40;
* **--relative-date** – отображение даты в относительном формате (например, "2 weeks ago") вместо стандартного формата даты;
* **--reverse** – отображение истории коммитов в обратном порядке;
* **--graph** – отображение ASCII графа с ветвлениями и историей слияний;
* **--pretty=<формат>** – настройка формата логов;
  + **oneline** – по строчке, состоящей из хэша;
  + **short** - только автор и комментарий;
  + **full/fuller** – полная информация о коммитах, с именем автора, комментарием, датой создания и внесения коммита
  + **format: <шаблон>** – свой шаблон для большей кастомизации;
    - **%H** — хеш коммита;
    - **%h** — сокращенный хеш коммита;
    - **%T** — хеш дерева;
    - **%t** — сокращенный хеш дерева;
    - **%P** — хеш родителей;
    - **%p** — сокращенный хеш родителей;
    - **%an** — имя автора — %ae — Электронная почта автора;
    - **%ad** — дата автора (формат даты можно задать опцией --date=option);
    - **%ar** — относительная дата автора;
    - **%cn** — имя коммитера;
    - **%ce** — электронная почта коммитера;
    - **%cd** — дата коммитера;
    - **%cr** — относительная дата коммитера;
    - **%s** — содержание;
* **-oneline** – сокращение для одновременного использования опций --pretty=oneline --abbrev-commit;
* **--follow <имя файла>** – отображение истории коммитов текущего файла, включая переименования;
* **--since=<врем. промежуток>** – отображение отфильтрованных коммитов по промежутку времени;
  + weeks – недели;
  + days – дни;
  + hours –часы;
* **--until=<врем. промежуток>** – отображение отфильтрованных коммитов по промежутку времени;
* **--after <ГГГГ-мм-дд>** – отображение коммитов, сделанные после указанной даты;
* **--before <ГГГГ-мм-дд>** – отображение коммитов, сделанные до указанной даты;
* **--author <имя автора>** – отображение только тех коммитов, в которых запись author совпадает с указанным именем автора;
* **--committer** **<имя коммитера>** – отображение только тех коммитов, в которых запись committer совпадает с указанной строкой;
* **--grep <строка|шаблон>** – отображение только коммитов, сообщение которых содержит указанную строку или регулярное выражение;
* **-S** – отображение только коммитов, в которых изменение в коде повлекло за собой добавление или удаление указанной строки;
* **--no-merges** – пропуск коммитов со слиянием веток;
* **<ветка1>..<ветка2>** – просмотр, тех коммитов из <ветка2>, которые не находятся в <ветка1> (полезно при слиянии веток);
* **<ветка1>…<ветка2>** – отображение коммитов, которые есть либо в <ветка1>, либо в <ветка2>, но не в обеих; знак < обозначает коммиты из ветка1, а > — из ветка2;
  + знак **<** обозначает коммиты из ветка1;
  + знак **>** обозначает коммиты из ветка2;
* **-L <начало|конец:(файл|функция:файл)>** – отображение истории изменений переданного набора строк или функции в файле;

**Результат выполнения команды:**

Отображение истории коммитов, начиная с самого свежего и уходя к истокам проекта и их хеши SHA1. По умолчанию, она показывает лишь историю текущей ветки, но может быть настроена на вывод истории других, даже нескольких сразу, веток.

**Примеры:**

* отображение истории коммитов в активной ветке;

git log

commit ca82a6dff817ec66f44342007202690a93763949

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

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

changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

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

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

removed unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6

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

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

first commit

[create repository]

* отображение истории коммитов в указанной ветке;

git log master

* отображение истории последних 2 коммитов в активной ветке;

git log *-2*

* отображение истории последних 2 коммитов в активной ветке и внесенной ими разницы;

git log *-2 -p*

commit ca82a6dff817ec66f44342007202690a93763949

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

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

changed the version number

diff --git a/Rakefile b/Rakefile

index a874b73..8f94139 100644

--- a/Rakefile

+++ b/Rakefile

@@ -5,7 +5,7 @@ require 'rake/gempackagetask'

spec = Gem::Specification.new do |s|

s.platform = Gem::Platform::RUBY

s.name = "simplegit"

- s.version = "0.1.0"

+ s.version = "0.1.1"

s.author = "Scott Chacon"

s.email = "schacon@gee-mail.com"

s.summary = "A simple gem for using Git in Ruby code."

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

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

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

removed unnecessary test

diff --git a/lib/simplegit.rb b/lib/simplegit.rb

index a0a60ae..47c6340 100644

--- a/lib/simplegit.rb

+++ b/lib/simplegit.rb

@@ -18,8 +18,3 @@ class SimpleGit

end

end

-

-if $0 == \_\_FILE\_\_

- git = SimpleGit.new

- puts git.show

-end

* отображение истории последних 2 коммитов в активной ветке с статистикой изменений;

git log *-2* *–stat*

commit ca82a6dff817ec66f44342007202690a93763949

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

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

changed the version number

Rakefile | 2 +-

1 file changed, 1 insertion(+), 1 deletion(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

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

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

removed unnecessary test

lib/simplegit.rb | 5 -----

1 file changed, 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6

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

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

first commit

README | 6 ++++++

Rakefile | 23 +++++++++++++++++++++++

lib/simplegit.rb | 25 +++++++++++++++++++++++++

3 files changed, 54 insertions(+)

* отображение истории коммитов в активной ветке с отображением по созданиям, переименованиям и правам доступа файлов

git log *--summary*

* отображение истории коммитов в активной ветке в обратном порядке;

git log *--reverse*

* отображение истории последних 2 коммитов в активной ветке с ASCII-представлением;

git log *-2 --graph*

* отображение истории коммитов в указанном файле в активной ветке;

git log index.html

* отображение истории последних 5 коммитов в указанном файле в активной ветке;

git log *-5* index.html

* отображение истории коммитов в указанном файле в активной ветке (коммиты и изменения);

git log *-p* index.html

* отображение истории коммитов текущего файла, включая переименования;

git log *--follow* index.html

* отображение истории последних 4 коммитов с форматированием выводимых данных;

git log *--pretty*=format:"%h - %an, %ar : %s" *-4*

ca82a6d - Scott Chacon, 6 years ago : changed the version number

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

a11bef0 - Scott Chacon, 6 years ago : first commit

* отображение истории коммитов с форматированием выводимых данных;

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

\* 2d3acf9 ignore errors from SIGCHLD on trap

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

|\

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

\* | 30e367c timeout code and tests

\* | 5a09431 add timeout protection to grit

\* | e1193f8 support for heads with slashes in them

|/

\* d6016bc require time for xmlschema

\* 11d191e Merge branch 'defunkt' into local

* отображение истории коммитов с форматированием выводимых данных – каждый коммит в одну строку;

git log *--pretty*=oneline

ca82a6dff817ec66f44342007202690a93763949 changed the version number

085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test

a11bef06a3f659402fe7563abf99ad00de2209e6 first commit

* отображение истории коммитов с форматированием выводимых данных – каждый коммит в одну строку с сокращением информации;

git log *--oneline*

cf3d9d8 [add] ignore .tmp files

a7b88ee [create]: git ignore file

c185b80 [create]: header for main

2b826bb [create]: main file of program

bc067c8 [add]: caption into README file

a98cce4 [create repository]

* отображение истории коммитов после временной метки (за последние 2 недели);

git log *--since*=2.weeks

* отображение истории коммитов после временной метки (за последние 1 день и 2 часа);

git log *--since*="1 day 2 hours" README

* отображение истории коммитов после временной метки (за последние 2 часа);

git log *--since*="2 hours" README

* отображение истории коммитов после временной метки (после 5-го марта 2018 года 23:21);

git log *--since*="2018-03-05 23:21:00" *--oneline*

cf3d9d8 [add] ignore .tmp files

a7b88ee [create]: git ignore file

* отображение истории коммитов до временной метки (до 5-го марта 2018 года 23:21);

git log *--until*="2018-03-05 23:21:00" *--oneline*

c185b80 [create]: header for main

2b826bb [create]: main file of program

bc067c8 [add]: caption into README file

a98cce4 [create repository]

* отображение истории коммитов, сделанные после указанной даты;

git log *--after* '2018-06-30'

* отображение истории коммитов, сделанные до указанной даты;

git log *--before* '2018-06-30'

* отображение истории коммитов, которые были сделаны указанным автором;

git log *--author*=Gvido Van Rossum *--oneline*

cf3d9d8 [add] ignore .tmp files

a7b88ee [create]: git ignore file

c185b80 [create]: header for main

2b826bb [create]: main file of program

bc067c8 [add]: caption into README file

a98cce4 [create repository]

* отображение истории коммитов, в описании которых есть совпадение с заданным словом (регистрозависимо, только коммиты текущей ветки);

git log *–grep* "create" *--oneline*

a7b88ee [create]: git ignore file

c185b80 [create]: header for main

2b826bb [create]: main file of program

a98cce4 [create repository]

* отображение истории коммитов, в описании которых есть совпадение с заданным словом (регистроНЕзависимо, только коммиты текущей ветки);

git log *–grep* "create" *–oneline* *-i*

* отображение истории коммитов, в описании которых есть совпадения для регулярного выражения (только коммиты текущей ветки);

git log *--grep* 'fix(ing|me)' *-P*

* отображение истории коммитов из ветки branch\_99, которые не влиты в master;

git log master..branch\_99

* отображение истории коммитов из ветки master, которые не влиты в branch\_99;

git log branch\_99..master

* отображение истории коммитов из указанных веток, начиная с их расхождения (коммит расхождения будет показан);

git log master...branch\_99 *--boundary* *--graph*

* отображение истории коммитов от указанного до указанного регулярных выражений в указанном файле;

git log *-L* '/<head>/','/<\/head>/' :index.html

### 4.5.4. Просмотр изменений, внесенных отдельным коммитом – git show

Другой полезной командой является git show<файл>, которая для каждой строки файла показывает автора и контрольную сумму последнего коммита, который изменил эту строку. -L <начало>, <конец> позволяет ограничить эту команду заданными строками. Это можно использовать, например, для выяснения того, какой коммит привёл к определённому багу (чтобы можно было его откатить).

Чтобы … нужно выполнить команду git show

**Аргументы:**

* аргумент (необязательный)

В качестве параметра в команду git show можно передать …

* аргумент (обязательный)

В качестве параметра в команду git show передается …

**Опции:**

* **-A, --all** — опция;

**Примеры:**

* …;

git add README.md

* …;

git add .

**Результат выполнения команды:**

Отображение информации о любом git объекте в простом и человекопонятном виде. Обычно она используется для просмотра информации о метке или коммите.

### 4.5.5. Просмотр информацию о последнем коммите – git blame

Другой полезной командой является git blame <файл>, которая для каждой строки файла показывает автора и контрольную сумму последнего коммита, который изменил эту строку. -L <начало>, <конец> позволяет ограничить эту команду заданными строками. Это можно использовать, например, для выяснения того, какой коммит привёл к определённому багу (чтобы можно было его откатить).

Чтобы … нужно выполнить команду git blame

**Аргументы*:***

* аргумент (необязательный)

В качестве параметра в команду git blame можно передать …

* аргумент (обязательный)

В качестве параметра в команду git blame передается …

**Опции:**

* **-A, --all** — опция;

**Примеры:**

* …;

git add README.md

* …;

git add .

**Результат выполнения команды:**

...

### 4.5.6. Поиск слов по проекту, состоянию проекта в прошлом – git grep

Чтобы … нужно выполнить команду git blame

***Аргументы:***

* аргумент (необязательный)

В качестве параметра в команду git blame можно передать …

* аргумент (обязательный)

В качестве параметра в команду git blame передается …

***Опции:***

* **-A, --all** — опция;

***Примеры:***

* …;

git add README.md

* …;

git add .

***Результат выполнения команды:***

Наконец, есть команда git grep, которая ищет по всем файлам в истории коммитов (а не только в рабочей директории, как grep) по заданному регулярному выражению.

Опция -n отображает соответствующий номер строки в файле для каждого совпадения, а --count показывает количество совпадений для каждого файла.

## 4.6. Локальный репозиторий

### 4.6.1. Добавление изменений в индекс – git add

**Индекс** — промежуточное место между вашим прошлым коммитом и следующим. Вы можете добавлять или удалять файлы из индекса. Когда вы делаете коммит в него попадают данные из индекса, а не из рабочей области.

После того как репозиторий будет создан, вам нужно добавить свои файлы в него. Каждый файл нужно добавлять отдельно или сказать утилите, что необходимо добавить все файлы явно. Пока вы не добавите файл сам он не будет отслеживаться. Новые файлы в будущем тоже нужно добавлять, они не добавляются автоматически.

Чтобы добавить файлы в индекс нужно выполнить команду **git add**

git add [имя файла]

***Аргументы:***

* имя файла (необязательный)

В качестве параметра в команду git add можно передать имя конкретного файла, который добавляется в индекс. Можно пользоваться регулярными выражениями, чтобы добавлять по какому-то шаблону. Например, git add \*.java: это значит, что нужно добавить только файлы с расширением java.

***Примеры:***

* добавление изменений одного файла;

git add README.md

* добавление изменений всех файлов текущей папки и все внутренних;

git add .

* добавление изменений по шаблону;

git add \*.txt

***Опции:***

* **-A, --all** — добавление всех файлов из состояния в staged;
* **-f, --force** — добавление также игнорируемых файлов;
* **-u, --update** — обновление отслеживаемых файлов;
* **-i, --interactive** – запуск интерактивной оболочки для добавления в индекс только выбранных файлов;
* **-p, --patch** – показ новых/изменённых файлов по очереди с указанием их изменений и вопросом об отслеживании/индексировании;

***Результат выполнения команды:***

Добавление содержимого рабочей директории в индекс (staged area) для последующего коммита.

### 4.6.2. Фиксация изменений – git commit

Коммит — это основной объект в управлении контроля версий. Он содержит все изменения за время этого коммита. Коммиты связаны между с собой как односвязный список.

Также у коммита есть еще своя информация, так называемые метаданные:

уникальный идентификатор коммита, по которому можно его найти;

имя автора коммита, который создал его;

дата создания коммита;

комментарий, который описывает, что было сделано во время этого коммита.

Основой истории версий являются коммиты. Для работы с ними используют git commit — эта команда откроет текстовый редактор для ввода сообщения коммита.

Чтобы … нужно выполнить команду git commit

***Аргументы:***

* аргумент (необязательный)

В качестве параметра в команду git commit можно передать …

* аргумент (обязательный)

В качестве параметра в команду git commit передается …

***Опции:***

* **-A, --all** — опция;

***Примеры:***

* …;

git add README.md

* …;

git add .

***Результат выполнения команды:***

...

Полезные опции команды git commit:

-F, --file [file] — Записать сообщение коммита из указанного файла

--author [author] — Подменить автора коммита

--date [date] — Подменить дату коммита

-m, --mesage [message] — Сообщение коммита

-a, --all — Закоммитеть все изменения в файлах

-i, --include [files...] — Добавить в индекс указанные файлы для следующего коммита

-o, --only [files...] — Закоммитеть только указанные файлы

--amend — Перезаписать предыдущий коммит

Ряд советов по коммитам:

— коммитьте часто;

— одно изменение — один коммит, но не коммитьте слишком незначительные изменения (в большом репозитории они могут засорить историю);

— комментируя сообщение о коммите, логически дополняйте фразу this commit will \_\_\_ и не используйте более 50 символов.

### 4.6.3. Временное сохранение изменений без коммита – git stash

Есть одна тонкость — при переключении веток Git требует, чтобы рабочее состояние было чистым, то есть все изменения в отслеживаемых файлах должны быть зафиксированы.

Прим. перев. Это не совсем так. При некоторых обстоятельствах Git может автоматически перенести незафиксированное изменение в другую ветку.

Однако порой у вас есть незавершённые изменения, которые нельзя фиксировать. В такой ситуации их можно сохранить и «спрятать» с помощью команды git stash. Чтобы вернуть изменения, используйте git stash apply.

Чтобы … нужно выполнить команду git stash

***Аргументы:***

* аргумент (необязательный)

В качестве параметра в команду git stash можно передать …

* аргумент (обязательный)

В качестве параметра в команду git stash передается …

***Опции:***

* **-A, --all** — опция;

***Примеры:***

* …;

git add README.md

* …;

git add .

***Результат выполнения команды:***

...

### 4.6.4. Отмена коммитов – git revert

Если хотите откатить коммит:

— git revert <коммит> — создаёт новый коммит, который отменяет изменения, но сохраняет историю;

Чтобы … нужно выполнить команду git revert

***Аргументы:***

* аргумент (необязательный)

В качестве параметра в команду git revert можно передать …

* аргумент (обязательный)

В качестве параметра в команду git revert передается …

***Опции:***

* **-A, --all** — опция;

***Примеры:***

* …;

git add README.md

* …;

git add .

***Результат выполнения команды:***

...

### 4.6.5. Сброс текущего HEAD – git reset

Чтобы удалить файлы из индекса вы можете использовать 2 команды git reset и git restore.

Чтобы … нужно выполнить команду git reset

***Аргументы:***

* аргумент (необязательный)

В качестве параметра в команду git resetможно передать …

* аргумент (обязательный)

В качестве параметра в команду git resetпередается …

***Опции:***

* **-A, --all** — опция;

***Примеры:***

* …;

git add README.md

* …;

git add .

***Результат выполнения команды:***

...

— git reset <коммит> — перемещает указатель HEAD, и создаёт более чистую историю, как будто коммита никогда не было. Но это также значит, что вы не сможете вернуться обратно к изменениям, если решите, что отмена была лишней. В общем, чище — не значит лучше!

Команда git reset, как можно догадаться из названия, используется в основном для отмены изменений. Она изменяет указатель HEAD и, опционально, состояние индекса. Также эта команда может изменить файлы в рабочей директории при использовании параметра --hard, что может привести к потере наработок при неправильном использовании, так что убедитесь в серьёзности своих намерений прежде чем использовать его.

git reset # убрать из индекса все добавленные в него изменения (в рабочей директории все изменения сохранятся), антипод git add

git reset readme.txt # убрать из индекса изменения указанного файла (в рабочей директории изменения сохранятся)

### 4.6.6. Восстановление файлов – git restore

git-restore — восстановит файлы рабочего дерева.

По сути вы можете добиться одного и того же с помощью обеих команд.

рекомендуют для сброса индекса использовать именно git restore -S . Вместо git reset HEAD .

Отмена изменений

git-reset — сбрасывает текущий HEAD до указанного состояния.

Чтобы … нужно выполнить команду git restore

***Аргументы:***

* аргумент (необязательный)

В качестве параметра в команду git restore можно передать …

* аргумент (обязательный)

В качестве параметра в команду git restore передается …

***Опции:***

* **-A, --all** — опция;

***Примеры:***

* …;

git add README.md

* …;

git add .

***Результат выполнения команды:***

...

### 4.6.7. Удаление файлов из индекса и рабочей директории – git rm

Чтобы … нужно выполнить команду git rm

***Аргументы:***

* аргумент (необязательный)

В качестве параметра в команду git rm можно передать …

* аргумент (обязательный)

В качестве параметра в команду git rm передается …

***Опции:***

* **-A, --all** — опция;

***Примеры:***

* …;

git add README.md

* …;

git add .

***Результат выполнения команды:***

...

Команда git rm используется в Git для удаления файлов из индекса и рабочей директории. Она похожа на git add с тем лишь исключением, что она удаляет, а не добавляет файлы для следующего коммита.

git rm text.txt # удалить отслеживаемый неизменённый файл и проиндексировать это изменение

git rm -f text.txt # удалить отслеживаемый изменённый файл и проиндексировать это изменение

git rm -r log/ # удалить всё содержимое отслеживаемой директории log/ и проиндексировать это изменение

git rm ind\* # удалить все отслеживаемые файлы с именем, начинающимся на «ind» в текущей директории и проиндексировать это изменение

git rm --cached readme.txt # удалить из отслеживаемых индексированный файл (ФАЙЛ ОСТАНЕТСЯ НА МЕСТЕ) (часто используется для нечаянно добавленных в отслеживаемые файлов)

### 4.6.8. Удаление неотслеживаемых файлов – git clean

Возможно, вместо этого вы захотите стереть все внесённые изменения. В таком случае используйте команду git clean. Опция -d также удалит неотслеживаемые файлы. Совет: добавьте опцию -n, чтобы увидеть, что произойдёт при запуске git clean без непосредственного использования.

Чтобы … нужно выполнить команду git rm

***Аргументы:***

* аргумент (необязательный)

В качестве параметра в команду git rm можно передать …

* аргумент (обязательный)

В качестве параметра в команду git rm передается …

***Опции:***

* **-A, --all** — опция;

***Примеры:***

* …;

git add README.md

* …;

git add .

***Результат выполнения команды:***

...

### 4.6.9. Перемещение/переименование файлов – git mv

Возможно, вместо этого вы захотите стереть все внесённые изменения. В таком случае используйте команду git mv. Опция -d также удалит неотслеживаемые файлы. Совет: добавьте опцию -n, чтобы увидеть, что произойдёт при запуске git mv без непосредственного использования.

Чтобы … нужно выполнить команду git mv

***Аргументы:***

* аргумент (необязательный)

В качестве параметра в команду git mv можно передать …

* аргумент (обязательный)

В качестве параметра в команду git mv передается …

***Опции:***

* **-A, --all** — опция;

***Примеры:***

* …;

git add README.md

* …;

git add .

***Результат выполнения команды:***

...

### 4.6.10. Тегирование – git tag

Возможно, вместо этого вы захотите стереть все внесённые изменения. В таком случае используйте команду git tag. Опция -d также удалит неотслеживаемые файлы. Совет: добавьте опцию -n, чтобы увидеть, что произойдёт при запуске git clean без непосредственного использования.

Чтобы … нужно выполнить команду git tag

***Аргументы:***

* аргумент (необязательный)

В качестве параметра в команду git tag можно передать …

* аргумент (обязательный)

В качестве параметра в команду git tag передается …

***Опции:***

* **-A, --all** — опция;

***Примеры:***

* …;

git add README.md

* …;

git add .

***Результат выполнения команды:***

...

## 4.7. Удаленный репозиторий

### 4.7.1. Работа с удаленным репозиторием – git remote

Для вывода списка удалённых репозиториев нужна команда git remote –v. С её помощью мы не только загружаем копию репозитория, но и отслеживаем удалённый сервер, находящийся по указанному адресу (ему присваивается имя origin).

Чтобы … нужно выполнить команду git remote

***Аргументы:***

* аргумент (необязательный)

В качестве параметра в команду git remote можно передать …

* аргумент (обязательный)

В качестве параметра в команду git remote передается …

***Опции:***

* **-A, --all** — опция;

***Примеры:***

* …;

git add README.md

* …;

git add .

***Результат выполнения команды:***

...

Другие часто употребляемые команды:

• git remote add <имя> <url> — добавляется удалённый репозиторий с заданным именем;

• git remote remove <имя> — удаляется удалённый репозиторий с заданным именем;

• git remote rename <старое имя> <новое имя> — переименовывается удалённый репозиторий;

• git remote set-url <имя> <url> — репозиторию с именем присваивается новый адрес;

• git remote show <имя> — показывается информация о репозитории.

### 4.7.2. Отправка изменений в удаленный репозиторий – git push

Чтобы … нужно выполнить команду git push

***Аргументы:***

* аргумент (необязательный)

В качестве параметра в команду git push можно передать …

* аргумент (обязательный)

В качестве параметра в команду git push передается …

***Опции:***

* **-A, --all** — опция;

***Примеры:***

* …;

git add README.md

* …;

git add .

***Результат выполнения команды:***

...

Операция push копирует изменения из экземпляра локального репозитория в удаленный. Это используется для постоянного хранения изменений в репозитории Git. Это то же самое, что и операция фиксации в Subversion.

Для отправки данных на сервер используйте

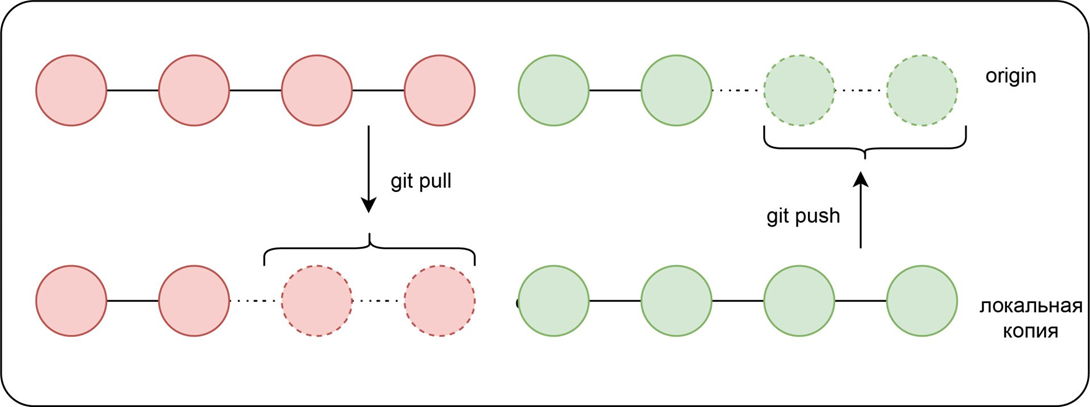
git push <имя> <ветка> — для отправления изменения в ветку заданного репозитория. Когда локальная ветка отслеживает удалённую, достаточно использовать git push или git pull

git push [<rep>] [<br>]

где rep это название внешнего репозитория, а br локальная ветка которую вы хотите отправить. Также вы можете использовать такую запись git push origin master:dev. Таким образом вы выгрузите вашу локальную ветку master на origin (но там она будет называется dev). Вы не сможете отправить данные во внешний репозитории если у вас нет на это прав. Также вы не сможете отправить данные на внешнюю ветку если она опережает вашу (в общем то отправить вы можете используя -f, --forse в этом случае вы перепишите историю на внешнем репозитории). Вы можете не указывать название ветки если ваша ветка отслеживает внешнюю.

Для удаления внешних веток используйте

git push origin --delete branch\_name



### 4.7.3. Копирование изменений из удаленного репозитория – git pull

Операция Pull копирует изменения из экземпляра удаленного репозитория в локальный. Операция Pull используется для синхронизации между двумя экземплярами репозитория. Это то же самое, что и операция обновления в Subversion.

Чтобы … нужно выполнить команду git pull

***Аргументы:***

* аргумент (необязательный)

В качестве параметра в команду git pull можно передать …

* аргумент (обязательный)

В качестве параметра в команду git pull передается …

***Опции:***

* **-A, --all** — опция;

***Примеры:***

* …;

git add README.md

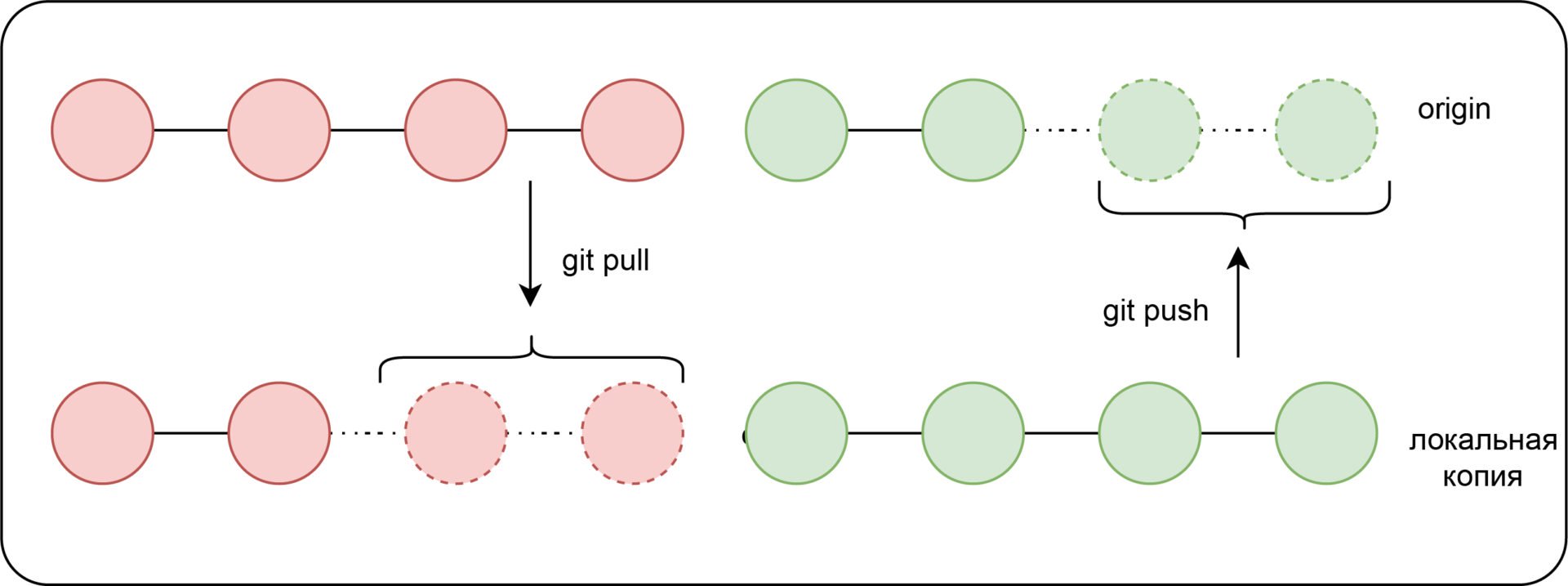
* …;

git add .

***Результат выполнения команды:***

...

git pull <имя> <ветка> — сливает данные из ветки;



### 4.7.4. Получение изменений из удаленного репозитория без слияния – git fetch

git fetch <имя> <ветка> — для получения данных из ветки заданного репозитория;

Мы можем хранить, отслеживать и обновлять историю коммитов не только на локальной машине, но и на удалённых репозиториях. По сути, можно говорить об облачных бэкапах нашей истории коммитов.

Чтобы … нужно выполнить команду git fetch

***Аргументы:***

* аргумент (необязательный)

В качестве параметра в команду git fetch можно передать …

* аргумент (обязательный)

В качестве параметра в команду git fetch передается …

***Опции:***

* **-A, --all** — опция;

***Примеры:***

* …;

git add README.md

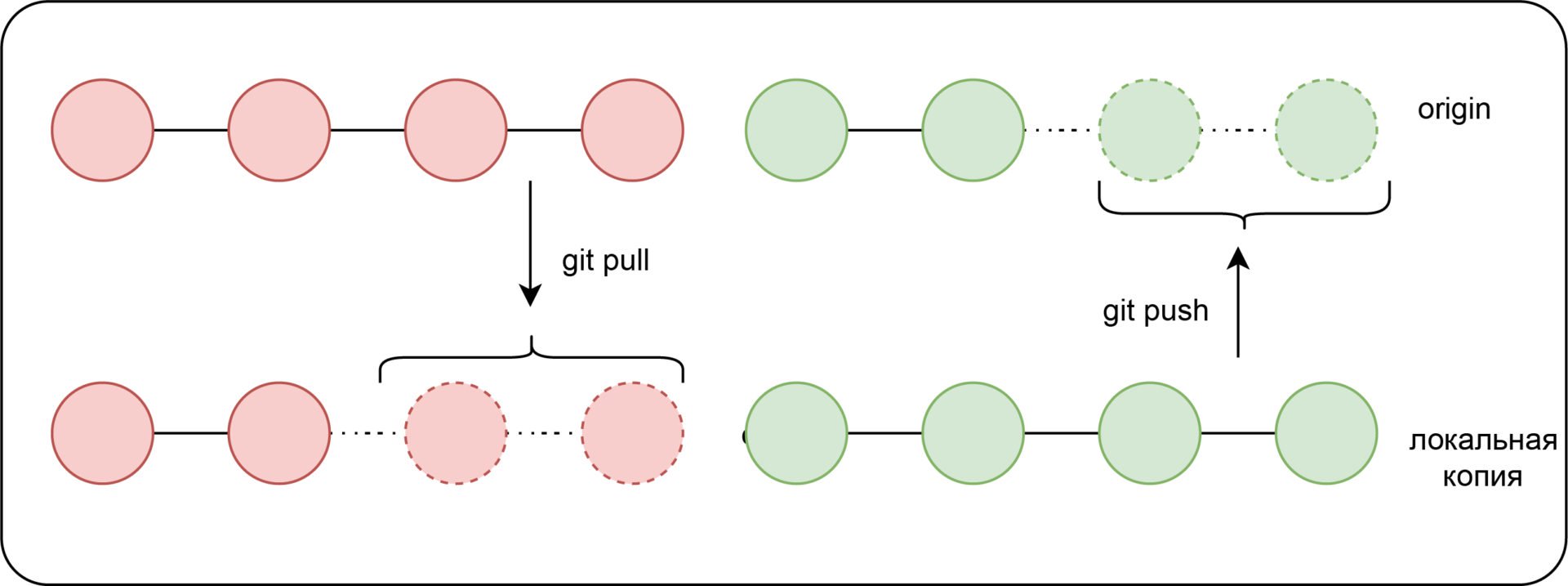
* …;

git add .

***Результат выполнения команды:***

...

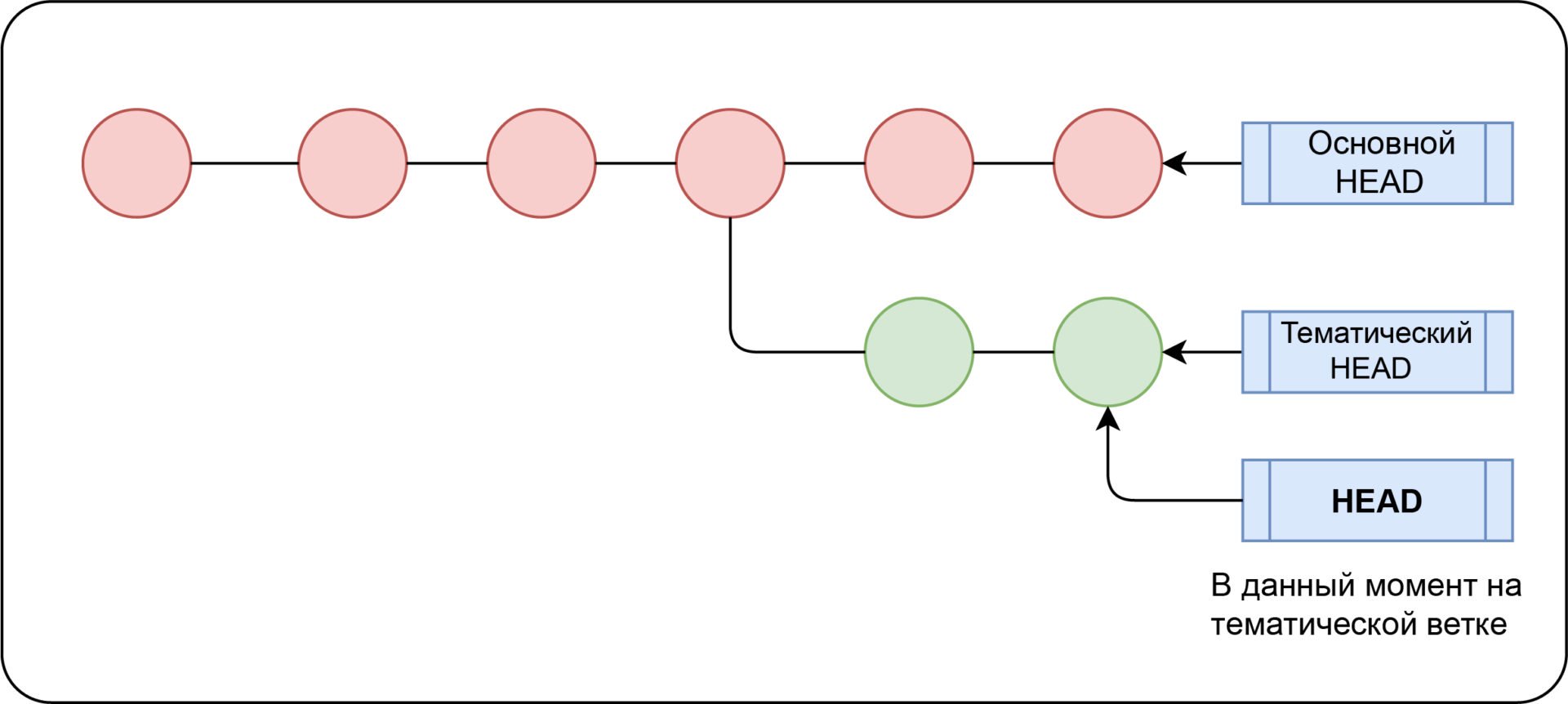
В результате несколько человек могут запрашивать с сервера изменения, выполнять изменения в локальных копиях, а потом отправлять их на удалённый сервер. Всё это позволяет легко взаимодействовать между собой в пределах одного репозитория.



## 4.8.Ветвление

Ветка — это указатель какого-то коммита. Так как коммит знает, какой коммит был до него, когда ветка указывает на какой-то коммит, к ней относятся и все те предыдущие.

Ключевая особенность Git — ветвление, позволяющее работать над разными версиями проекта. Таким образом, вместо одного перечня с упорядоченными коммитами история может расходиться в некоторых точках, поэтому становится похожей на дерево. И каждая ветвь содержит в Git легковесный указатель HEAD, указывающий на последний коммит в данной ветке. В результате можно легко создать много веток. Делая это, называйте ветки согласно разрабатываемой функциональности. Ветку по умолчанию называют master.



### 4.8.1. Создание ветки – git branch

Что ж, у нас есть HEAD для каждой ветки и общий указатель HEAD. Переключение между ветками предполагает лишь перемещение HEAD в HEAD соответствующей ветки.

Стандартные команды при ветвлении в Git:

• git branch <имя ветки> — для создания новой ветки с HEAD, указывающим на HEAD. Если аргумент <имя ветки> передан не будет, команда выведет список всех имеющихся локальных веток;

Чтобы … нужно выполнить команду git rm

***Аргументы:***

* аргумент (необязательный)

В качестве параметра в команду git rm можно передать …

* аргумент (обязательный)

В качестве параметра в команду git rm передается …

***Опции:***

* **-A, --all** — опция;

***Примеры:***

* …;

git add README.md

* …;

git add .

***Результат выполнения команды:***

...

• git branch -d <имя ветки> — для удаления ветки;

git branch -u <имя удалённого репозитория>/<удалённая ветка> —текущая ветка привязывается к указанной удалённой ветке;

• git branch --vv — служит, чтобы показать локальные и отслеживаемые удалённые ветки.

Опции git branch:

-r | -a [--merged | --no-merged] — Список отслеживаемых внешних веток -r. Список и отслеживаемых и локальных веток -a. Список слитых веток --merged. Список не слитых веток --no-merged.

-l, -f <имя-ветки> [<точка-начала>] — Список имён веток -l. Принудительное создание, перемещение или удаление ветки -f. Создание новой ветки <имя ветки>.

-r (-d | -D) — Выполнить действие на отслеживаемой внешней ветке -r. Удалить слитую ветку -d. Принудительное удаление (даже не слитой ветки) -D.

-m | -M [<Старая ветка>] <Новая ветка> — Переместить/переименовать ветки и ее журнал ссылок (-m). Переместить/переименовать ветку, даже если целевое имя уже существует -M.

(-с | -С) [<старая-ветка>] <новая-ветка> — Скопировать ветку и её журнал ссылок -c. Скопировать ветку, даже если целевое имя уже существует -C.

-v, -vv — Список веток с последним коммитом на ветке -v. Список и состояние отслеживаемых веток с последним коммитом на них.

Больше информации смотрите в git branch -h | --help.

### 4.8.2. Переключение ветки – git checkout

Для переключения на ветку используйте git checkout . Также вы можете создать ветку выполнив git checkout -b <ветка>.

git checkout <имя ветки> — для переключения на эту ветку. Если хотим создать новую ветку перед переключением, можем передать опцию –b.

Чтобы … нужно выполнить команду git checkout

***Аргументы:***

* аргумент (необязательный)

В качестве параметра в команду git checkout можно передать …

* аргумент (обязательный)

В качестве параметра в команду git checkout передается …

***Опции:***

* **-A, --all** — опция;

***Примеры:***

* …;

git add README.md

* …;

git add .

***Результат выполнения команды:***

...

И локальный, и удалённый репозиторий могут иметь много веток, поэтому при отслеживании на деле отслеживается удалённая ветка, то есть git clone привязывает ветвь master к ветви origin/master удалённого репозитория.

Парочка команд для привязывания к удалённой ветке:

•

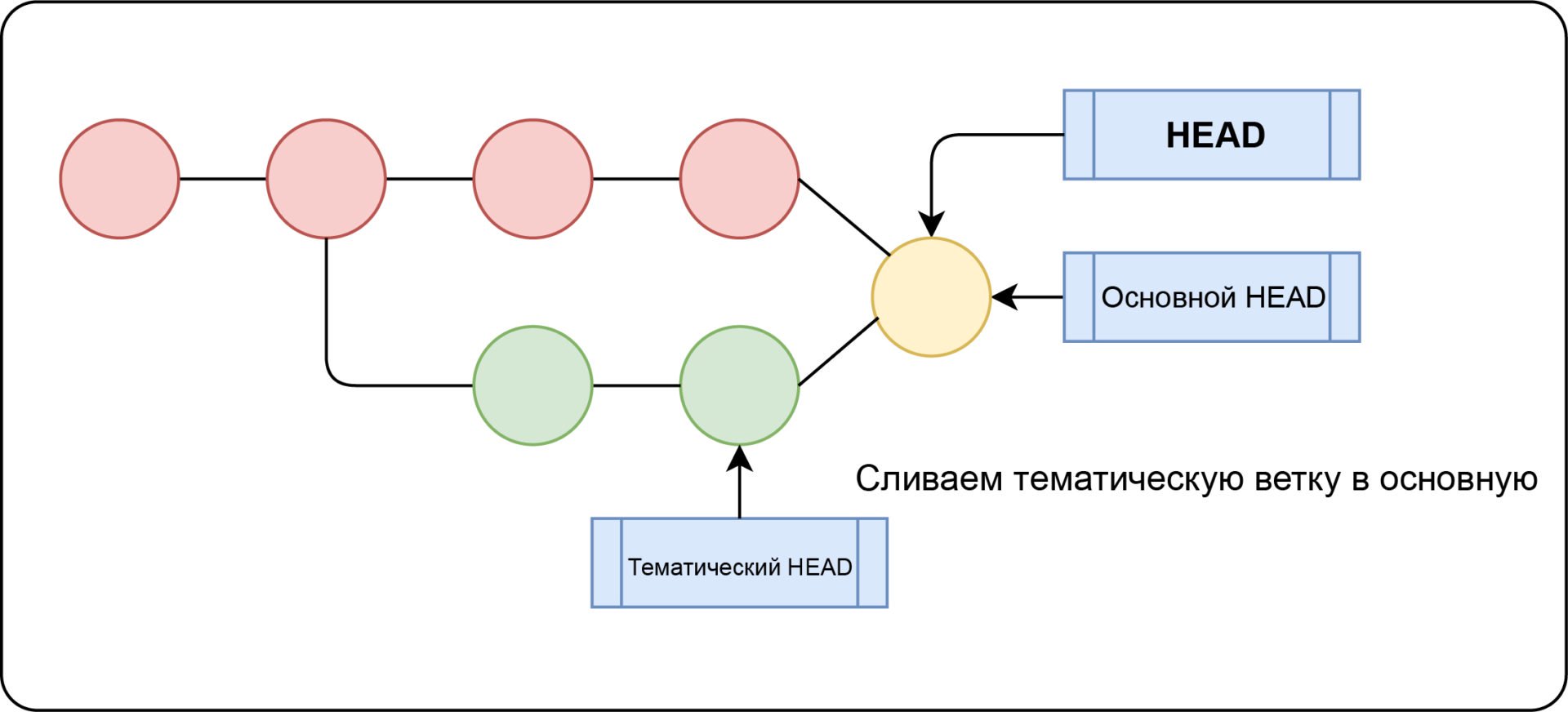
• git checkout --track <имя удалённого репозитория>/<удалённая ветка> — аналогично;

• git checkout -b <ветка> <имя удалённого репозитория>/<удалённая ветка> — создаётся новая локальная ветвь и начинает отслеживать удалённую;

• git checkout <удалённая ветка> — создаётся локальная ветвь с таким же именем, как и у удалённой, плюс начинает её отслеживать;

### 4.8.3. Слияние веток – git merge

Включает в себя создание нового коммита, основанного на общем коммите-предке 2-х ветвей, указывает на оба HEAD. Для осуществления слияния нужно перейти на основную ветки и использовать команду git merge <тематическая ветка>.



Когда обе ветки меняют одну и ту же часть файла, возникает конфликт слияния. В этой ситуации Git не понимает, какую версию файла нужно сохранить. Разрешать конфликт следует вручную. Для просмотра конфликтующих файлов, используйте git status.

Чтобы … нужно выполнить команду git merge

***Аргументы:***

* аргумент (необязательный)

В качестве параметра в команду git merge можно передать …

* аргумент (обязательный)

В качестве параметра в команду git merge передается …

***Опции:***

* **-A, --all** — опция;

***Примеры:***

* …;

git add README.md

* …;

git add .

***Результат выполнения команды:***

...

Маркеры разрешения конфликта:

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

Everything above the ==== is the version in master.

=======

Everything below the ==== is the version in the test branch.

>>>>>>> test:index.html

В этом блоке надо заменить всё на версию, которую хотите оставить, после чего подготовить файл. Разрешив все конфликты, можно завершать слияние, используя git commit.

### 4.8.4. Перемещение коммитов в ветку – git rebase

Осуществляется вместо совмещения 2-ух веток коммитом слияния. Перемещение заново воспроизводит коммиты тематической ветви в виде набора новых коммитов базовой ветви, что обеспечивает более чистую историю коммитов.

Чтобы выполнить перемещение, используют команду git rebase <основная ветка> <тематическая ветка>. Она воспроизводит изменения тематической ветви на основной. При этом HEAD тематической ветви указывает на последний воспроизведённый коммит.

Чтобы … нужно выполнить команду git rebase

***Аргументы:***

* аргумент (необязательный)

В качестве параметра в команду git rebase можно передать …

* аргумент (обязательный)

В качестве параметра в команду git rebase передается …

***Опции:***

* **-A, --all** — опция;

***Примеры:***

* …;

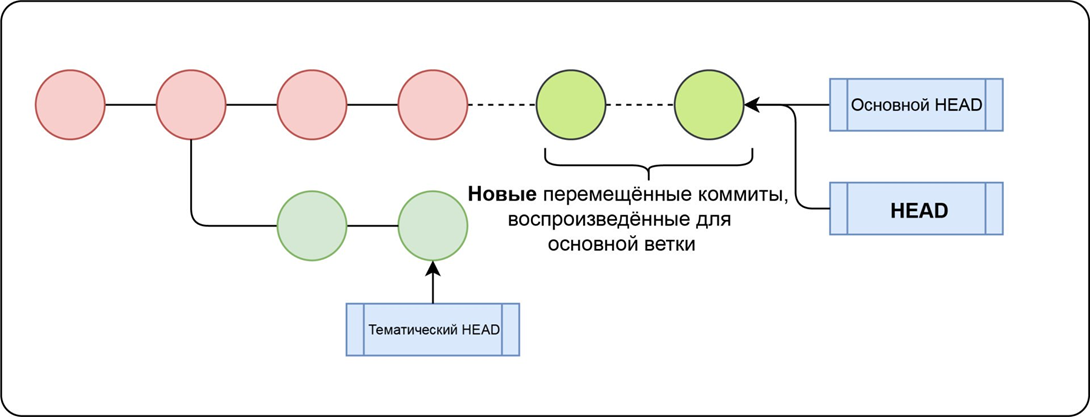
git add README.md

* …;

git add .

***Результат выполнения команды:***

...



Совет: перемещайте изменения лишь на вашей приватной локальной ветке. Не стоит перемещать коммиты, от которых ещё кто-то зависит.

### 4.8.5. Перемещение коммитов – git cherry-pick

Чтобы … нужно выполнить команду git cherry-pick

***Аргументы:***

* аргумент (необязательный)

В качестве параметра в команду git rebase можно передать …

* аргумент (обязательный)

В качестве параметра в команду git rebase передается …

***Опции:***

* **-A, --all** — опция;

***Примеры:***

* …;

git add README.md

* …;

git add .

***Результат выполнения команды:***

...

## 4.9. Игнорирование файлов – **.gitignore**

Иногда не надо, чтобы Git отслеживал все файлы в репозитории, ведь в их число могут входить:

* файлы с конфиденциальной информацией;
* огромные бинарные файлы;
* специфичные файлы;
* файлы сборок, генерируемые после каждой компиляции.

Для игнорирования предусмотрен файл **.gitignore**, где отмечаются файлы для игнорирования.

Чтобы отметить файлы, которые мы хотим игнорировать, можно использовать шаблоны поиска (считайте их упрощёнными регулярными выражениями):

* **/\_\_** — соответствует файлам только в текущей директории (позволяет избежать рекурсивности);
* **\_\_/** — соответствует всем файлам в указанной директории;
* **\*\_\_** — соответствует всем файлам с указанным окончанием;
* **! —** игнорирование файлов, попадающих под указанный шаблон;
* **[\_\_]** — соответствует любому символу из указанных в квадратных скобках;
* **?** — соответствует любому символу;
* **/\*\*/** — соответствует вложенным директориям, например a/\*\*/d соответствует a/d, a/b/d, a/b/c/d и т. д.

***Примеры:***

* **Игнорирование файлов**
* ВСЕ файлы и директории, включая поддиректории и файлы в них.

\*

* Тип файла, будет игнорироваться АБСОЛЮТНО во всех директориях.

***Например*** /files/data.zip, /server.log, /uploads/users/data/info.xls

\*.zip

\*.log

\*.pdf

\*.xls

* Файл во ВСЕХ директориях.

***Например*** /params/db/config.php, /config.php

config.php

* Конкретный файл ТОЛЬКО в корне проекта (корнём считается расположение файла .gitignore)

***Например*** НЕ БУДЕТ проигнорирован файл /db/config.php

/config.php

* Конкретный  файл ТОЛЬКО в указанной директории.

***Например*** НЕ БУДЕТ проигнорирован файл /prod/params/config.php

/params/config.php

* **Игнорирование директорий**
* ВСЕ файлы и папки ТОЛЬКО в конкретной директории ВКЛЮЧАЯ  поддиректории и файлы в них.

***Например*** /images/user.jpg, /images/company/logo.png

НЕ БУДУТ проигнорированы файлы и папки /prod/images/user.jpg

/images/\*

* ВСЕ файлы и папки в ЛЮБЫХ директориях с указанным именем

***Например*** /images/user.jpg, /core/images/user.jpg

images/\*

* ВСЕ html-файлы в ОДНОЙ КОНКРЕТНОЙ директории (НЕ ВКЛЮЧАЯ поддиректории)

***Например*** /private/index.html НЕ БУДУТ проигнорированы файлы в /private/ivan/index.html

/private/\*.html

* ВСЕ html-файлы в КОНКРЕТНОЙ директории ВКЛЮЧАЯ поддиректории

***Например*** /private/info.html, /private/users/ivan/info.html

/private/\*\*/\*.html

* **Разное**
* Игнорирование ВСЕХ файлов и папок внутри директории /secret,

за ИСКЛЮЧЕНИЕМ файла /secret/free.txt, он не будет проигнорирован

/secret/\*

!/secret/free.txt

* Игнорирование файла с именем, содержащим спецсимволы

***Например*** !readme!.txt

\!readme!.txt

* Игнорирование всех JPG и JPEG файлов внутри директорий,

которые начинаются на "h" и МОГУТ содержать ещё один символ после

***Например*** /images/h4/user.jpg, /images/h/company.jpeg

/images/h?/\*.jp?g

Также можно использовать шаблоны поиска при указании файла/папки в других командах. Например, git add src/\*.css добавит все файлы .css в папке src.