Система контроля версий Git

Оглавление

[1. Система контроля версий 4](#_Toc61368421)

[1.1. Что такое «Система контроля версий» и почему это важно? 4](#_Toc61368422)

[1.2. Виды систем контроля версий 4](#_Toc61368423)

[1.2.1. Локальные системы контроля версий 4](#_Toc61368424)

[1.2.2. Централизованные системы контроля версий 5](#_Toc61368425)

[2.1.3. Распределённые системы контроля версий 6](#_Toc61368426)

[2. Git 7](#_Toc61368427)

[2.1. Архитектура Git и ее особенности 7](#_Toc61368428)

[2.1.1. Хранение информации об изменениях в виде снимков, а не различий 7](#_Toc61368429)

[2.1.2. Локальное выполнение почти всех операций 9](#_Toc61368430)

[2.1.3. Вычисление хеш суммы для всего 10](#_Toc61368431)

[2.1.4. Добавление новых данных при выполнений любых действий 10](#_Toc61368432)

[2.2. Состояния файлов в Git 10](#_Toc61368433)

[2.3. Файловая система Git 11](#_Toc61368434)

[2.4. Объекты Git 13](#_Toc61368435)

[2.4.1. Блобы(Blobs) 13](#_Toc61368436)

[2.4.2. Дерево(Trees) 13](#_Toc61368437)

[2.4.3. Коммиты (Commits) 13](#_Toc61368438)

[2.4.4. Ссылки (References) 14](#_Toc61368439)

[3. Установка Git 16](#_Toc61368440)

[3.2. Проверка установки Git 17](#_Toc61368441)

[3.3. Настройка Git 17](#_Toc61368442)

[3.3.1. Установка имени и электронной почты 17](#_Toc61368443)

[3.3.2. Устновка параметров окончаний строк 18](#_Toc61368444)

[3.3.3. Установка отображения Unicode 18](#_Toc61368445)

[3.3.4. Установка редактора 18](#_Toc61368446)

[3.3.5. Установка цветового выделения 18](#_Toc61368447)

[3.3.6. Установка псевдонимов 18](#_Toc61368448)

[3.3.7. Проверка настроек 19](#_Toc61368449)

[4. Работа с Git 20](#_Toc61368450)

[5. Gitflow 36](#_Toc61368451)

[5.1. Основные ветки (Master) и ветки разработки (Develop) 36](#_Toc61368452)

[5.2. Функциональные ветки (Feature) 37](#_Toc61368453)

[5.3. Ветки выпуска (Release) 37](#_Toc61368454)

[5.4. Ветки исправления (Hotfix) 38](#_Toc61368455)

[5.5. Последовательность действий при работе по модели Gitflow 39](#_Toc61368456)

[Создание заготовки проекта 40](#_Toc61368457)

# 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.

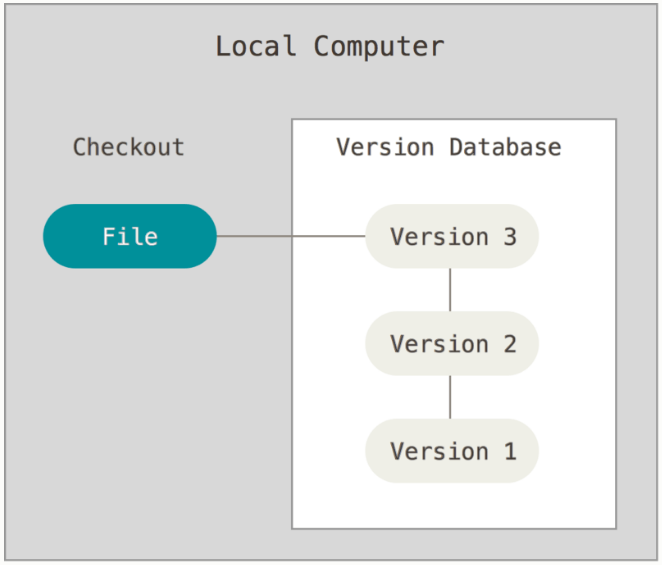
Если вы графический или web-дизайнер и хотите сохранить каждую версию изображения или макета (скорее всего, захотите), система контроля версий (далее СКВ) — как раз то, что нужно. Она позволяет вернуть файлы к состоянию, в котором они были до изменений, вернуть проект к исходному состоянию, увидеть изменения,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

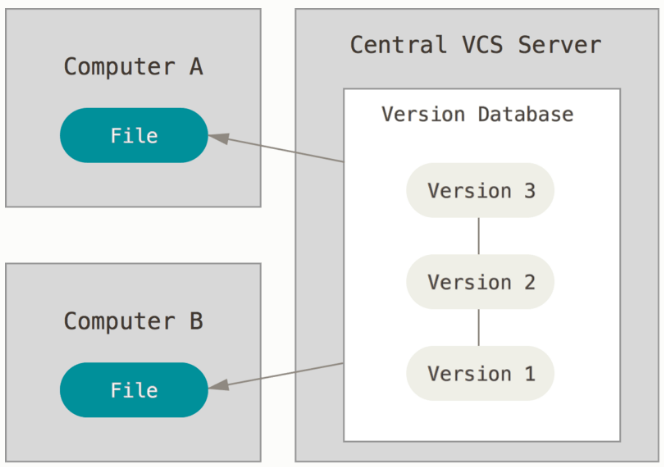
Для того, чтобы решить эту проблему, программисты давным-давно разработали локальные СКВ с простой базой данных, которая хранит записи о всех изменениях в файлах, осуществляя тем самым контроль ревизий.



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.

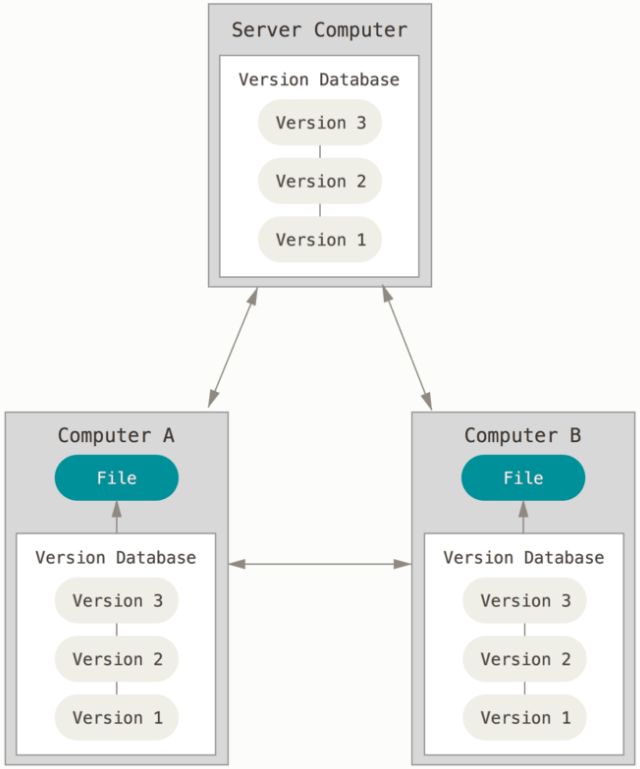


Такой подход имеет множество преимуществ, особенно перед локальными СКВ. Например, все разработчики проекта в определённой степени знают, чем занимается каждый из них. Администраторы имеют полный контроль над тем, кто и что может делать, и гораздо проще администрировать ЦСКВ, чем оперировать локальными 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).

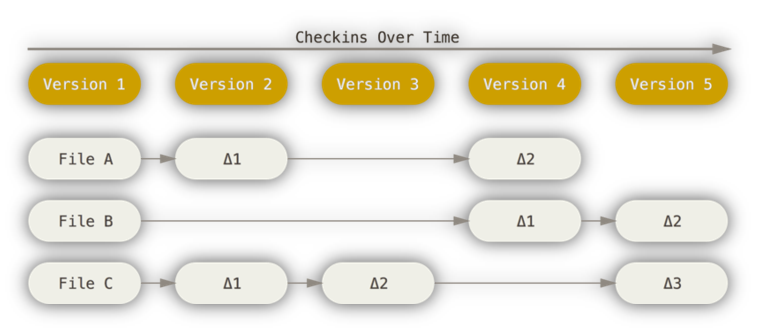
**Репозиторий Git** представляет собой каталог файловой системы, в котором находятся файлы конфигурации репозитория, файлы журналов, хранящие операции, выполняемые над репозиторием, индекс, описывающий расположение файлов, и хранилище, содержащее собственно файлы.

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.

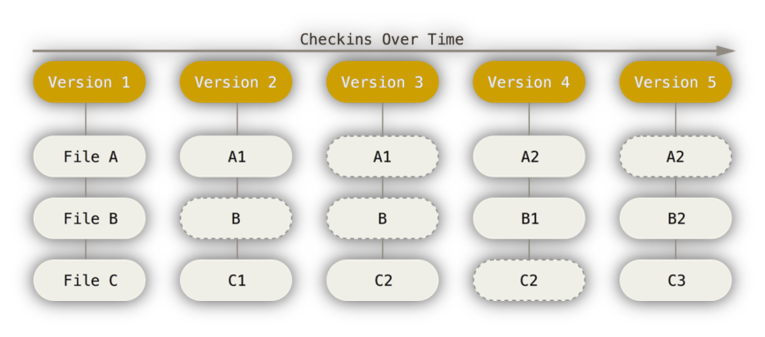
### 2.1.1. Хранение информации об изменениях в виде снимков, а не различий

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.

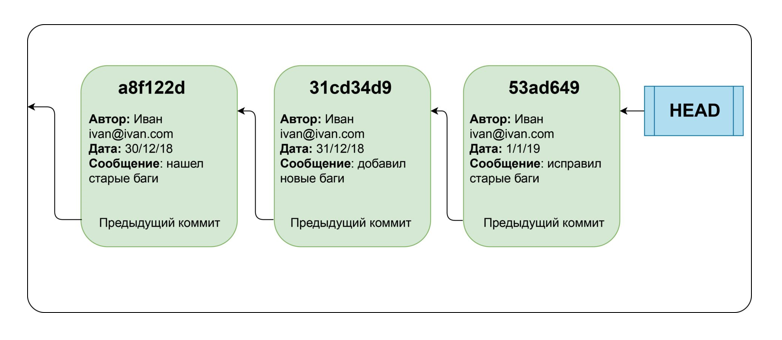
Эти системы (CVS, Subversion, Perforce, Bazaar и т.д.) представляют хранимую информацию в виде набора файлов и изменений, сделанных в каждом файле, по времени (обычно это называют контролем версий, основанным на различиях).



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)

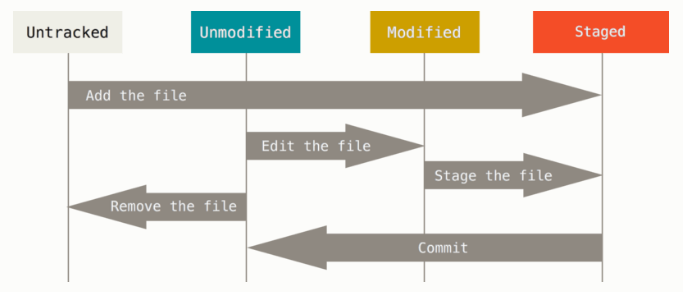
**Измененный файл** – файл, который уже добавлен в гит репозиторий и в котором сделаны изменения.

* 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.

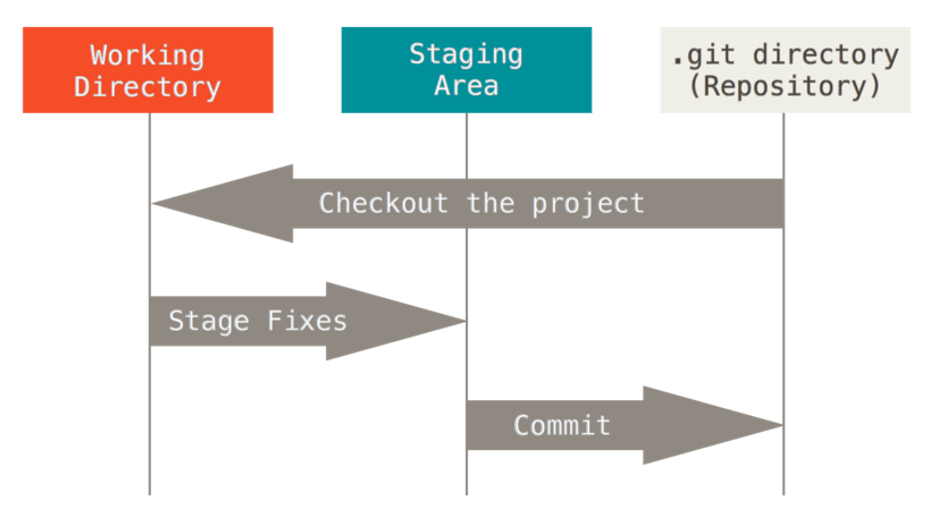
Этот цикл можно представить следующим образом:

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

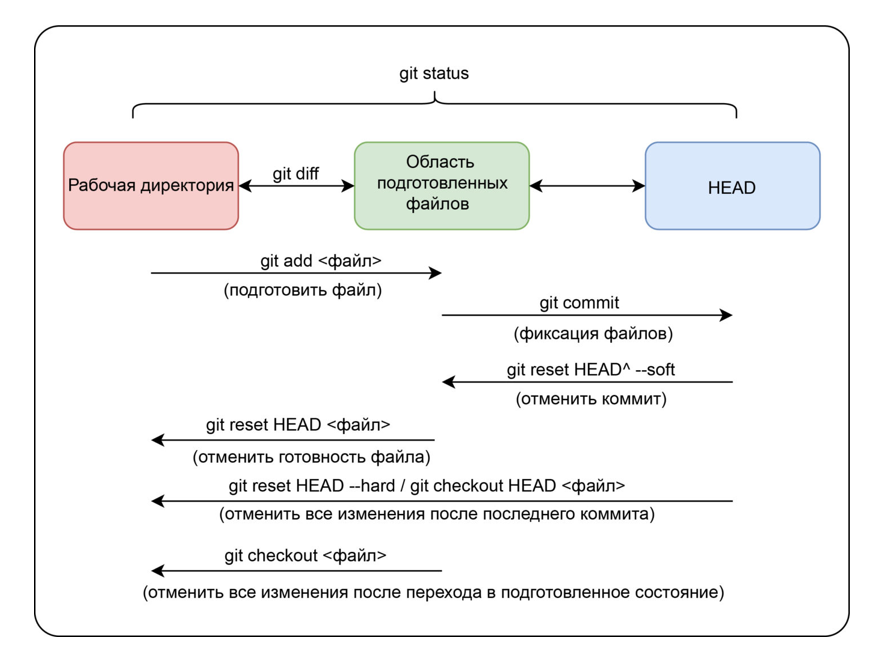
Мы подошли к трём основным секциям проекта Git: Git-директория (Git directory), рабочая директория (working directory) и область подготовленных файлов (staging area).



**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;

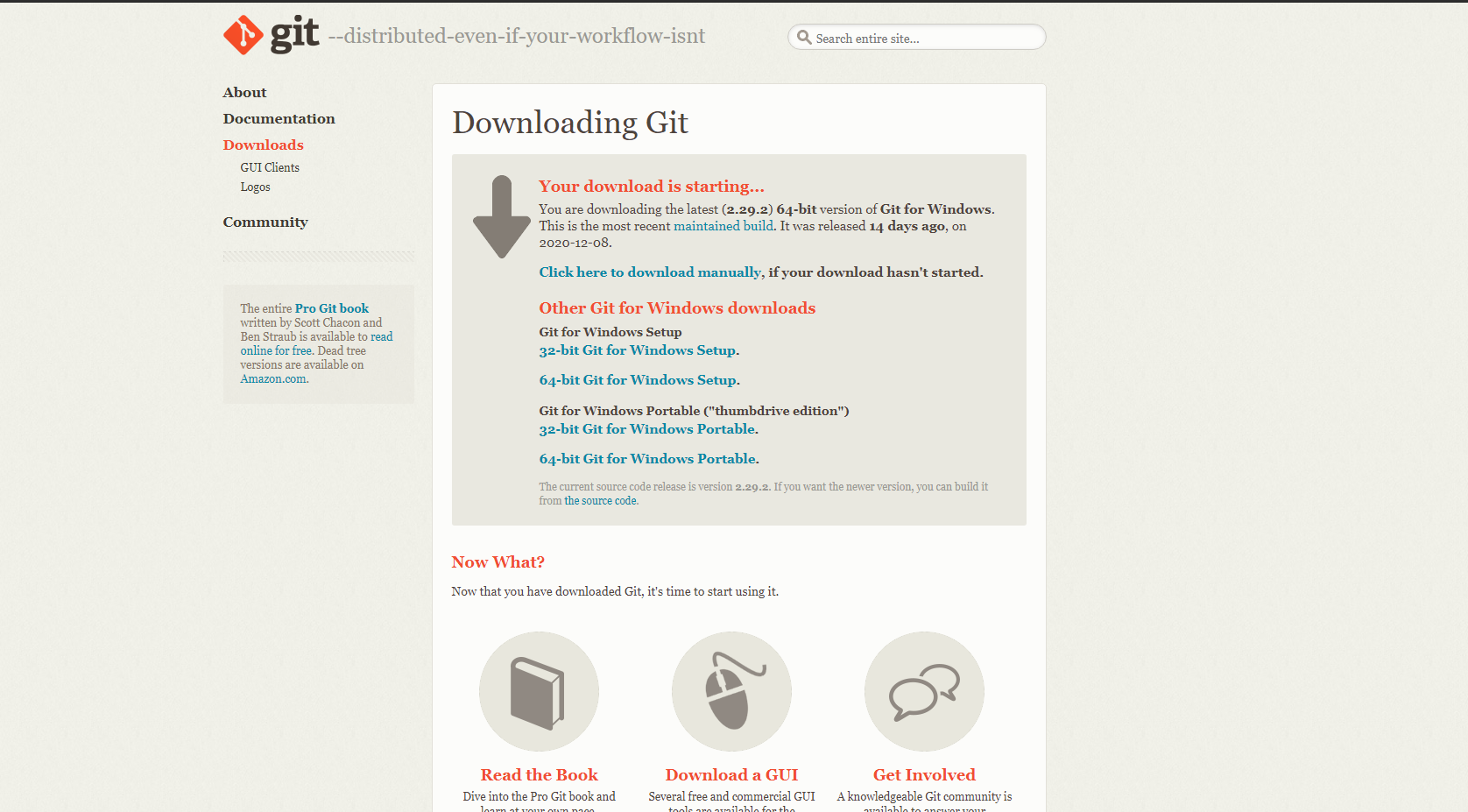
произвольный кусок данных (данный блок можно использовать для электронной подписи или для пояснения тега).

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.

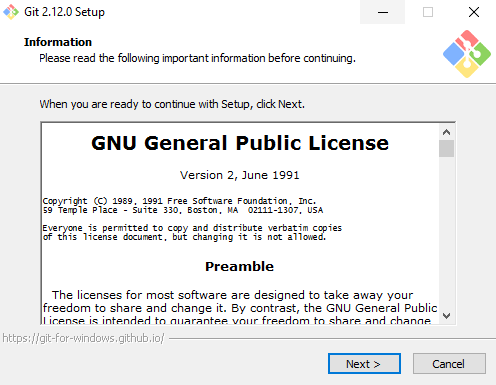
# 3. Установка Git

Шаги:

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



1. Запустить программу установки



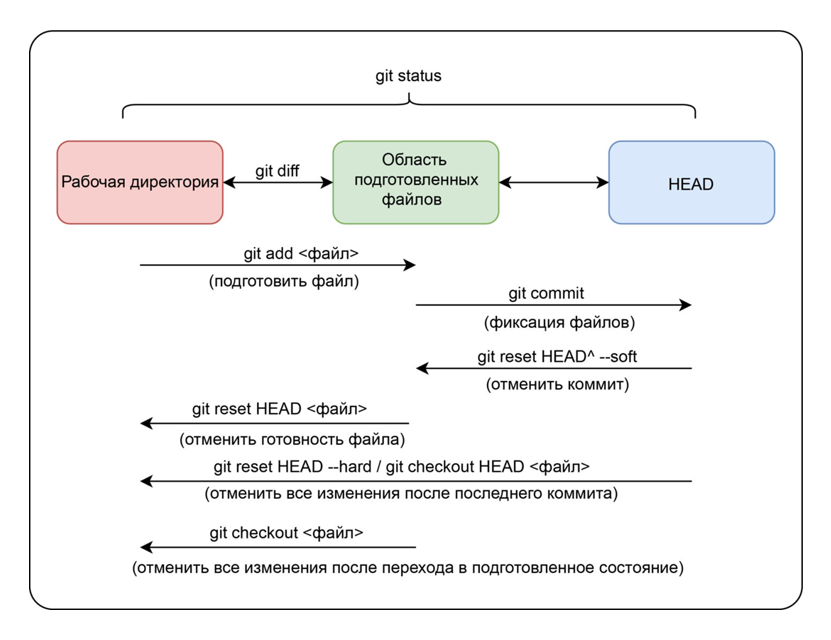
1. После успешного запуска программы установки отобразится экран мастера установки Git. Для завершения установки следуйте инструкциям, нажимая кнопки Далее и Готово. Параметры по умолчанию достаточно хорошо подходят для большинства пользователей.

## 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.

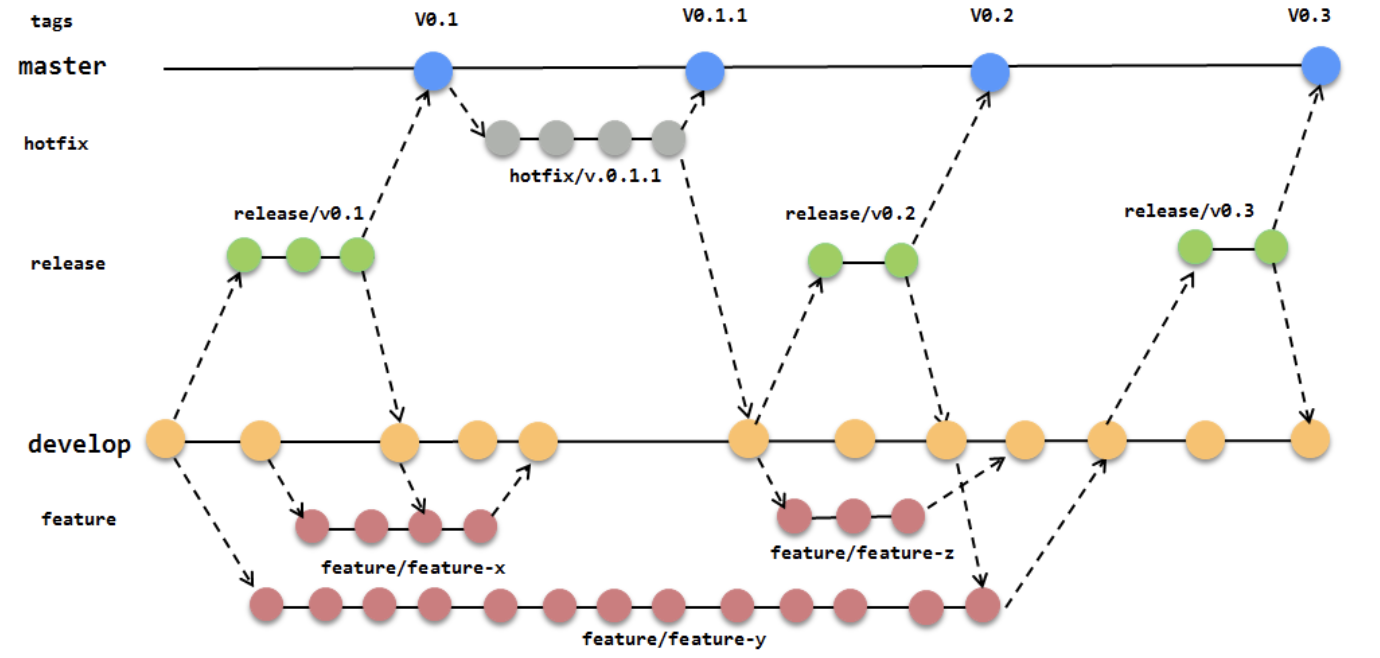


# 5. Gitflow

**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>



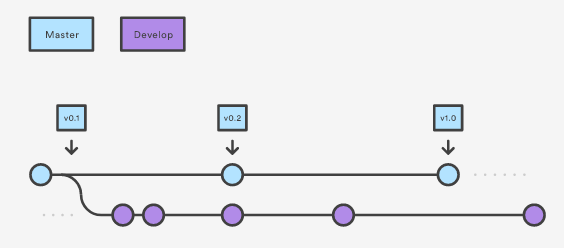
Ветки:

* Master
* Dev
* Feature
* Release
* Hotfix

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

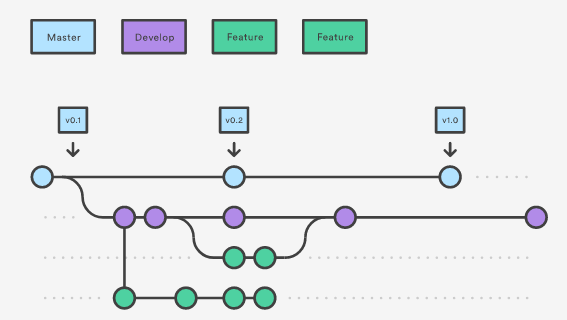
Для фиксации истории проекта в рамках этого процесса вместо одной ветки master используются две ветки. В ветке master хранится официальная история релиза, а ветка develop предназначена для объединения всех функций. Кроме того, для удобства рекомендуется присваивать всем коммитам в ветке master номер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



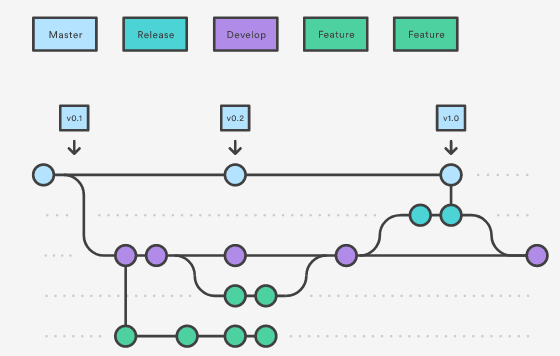
Как правило, ветки feature создаются на основе последней ветки develop.

## 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.

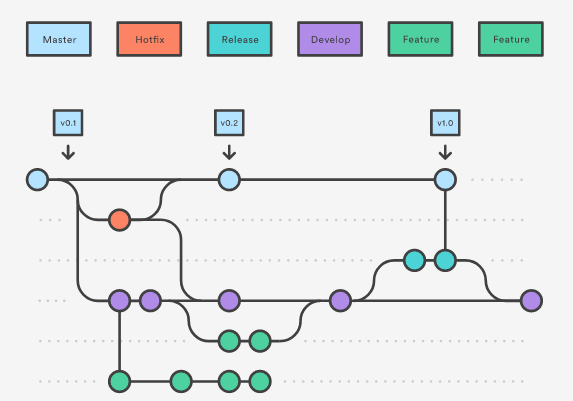
Создание веток release — это еще одна простая операция ветвления. Как и ветки feature, ветки release основаны на ветке develop.



## 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.

# Создание заготовки проекта