Оглавление

[1. Система управления версиями 2](#_Toc528849241)

[2. Git-flow 6](#_Toc528849242)

[3. Шпаргалка по основным командам 10](#_Toc528849243)

# Система управления версиями

Система управления версиями (от англ. Version Control System, VCS или Revision Control System) — программное обеспечение для облегчения работы с изменяющейся информацией. Система управления версиями позволяет хранить несколько версий одного и того же документа, при необходимости возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое.

Такие системы наиболее широко используются при разработке программного обеспечения для хранения исходных кодов разрабатываемой программы. Однако они могут с успехом применяться и в других областях, в которых ведётся работа с большим количеством непрерывно изменяющихся электронных документов. В частности, системы управления версиями применяются в САПР, обычно в составе систем управления данными об изделии (PDM). Управление версиями используется в инструментах конфигурационного управления (Software Configuration Management Tools).

Программное обеспечение Википедии ведёт историю изменений для всех её статей, используя методы, аналогичные тем, которые применяются в системах управления версиями.

Общие сведения

Ситуация, в которой электронный документ за время своего существования претерпевает ряд изменений, достаточно типична. При этом часто бывает важно иметь не только последнюю версию, но и несколько предыдущих. В простейшем случае можно просто хранить несколько вариантов документа, нумеруя их соответствующим образом. Такой способ неэффективен (приходится хранить несколько практически идентичных копий), требует повышенного внимания и дисциплины и часто ведёт к ошибкам, поэтому были разработаны средства для автоматизации этой работы.

Традиционные системы управления версиями используют централизованную модель, когда имеется единое хранилище документов, управляемое специальным сервером, который и выполняет большую часть функций по управлению версиями. Пользователь, работающий с документами, должен сначала получить нужную ему версию документа из хранилища; обычно создаётся локальная копия документа, так называемая «рабочая копия». Может быть получена последняя версия или любая из предыдущих, которая может быть выбрана по номеру версии или дате создания, иногда и по другим признакам. После того, как в документ внесены нужные изменения, новая версия помещается в хранилище. В отличие от простого сохранения файла, предыдущая версия не стирается, а тоже остаётся в хранилище и может быть оттуда получена в любое время. Сервер может использовать т. н. дельта-компрессию — такой способ хранения документов, при котором сохраняются только изменения между последовательными версиями, что позволяет уменьшить объём хранимых данных. Поскольку обычно наиболее востребованной является последняя версия файла, система может при сохранении новой версии сохранять её целиком, заменяя в хранилище последнюю ранее сохранённую версию на разницу между этой и последней версией. Некоторые системы (например, ClearCase) поддерживают сохранение версий обоих видов: большинство версий сохраняется в виде дельт, но периодически (по специальной команде администратора) выполняется сохранение версий всех файлов в полном виде; такой подход обеспечивает максимально полное восстановление истории в случае повреждения репозитория.

Иногда создание новой версии выполняется незаметно для пользователя (прозрачно), либо прикладной программой, имеющей встроенную поддержку такой функции, либо за счёт использования специальной файловой системы. В этом случае пользователь просто работает с файлом, как обычно, и при сохранении файла автоматически создаётся новая версия.

Часто бывает, что над одним проектом одновременно работают несколько человек. Если два человека изменяют один и тот же файл, то один из них может случайно отменить изменения, сделанные другим. Системы управления версиями отслеживают такие конфликты и предлагают средства их решения. Большинство систем может автоматически объединить (слить) изменения, сделанные разными разработчиками. Однако такое автоматическое объединение изменений, обычно, возможно только для текстовых файлов и при условии, что изменялись разные (непересекающиеся) части этого файла. Такое ограничение связано с тем, что большинство систем управления версиями ориентированы на поддержку процесса разработки программного обеспечения, а исходные коды программ хранятся в текстовых файлах. Если автоматическое объединение выполнить не удалось, система может предложить решить проблему вручную.

Часто выполнить слияние невозможно ни в автоматическом, ни в ручном режиме, например, если формат файла неизвестен или слишком сложен. Некоторые системы управления версиями дают возможность заблокировать файл в хранилище. Блокировка не позволяет другим пользователям получить рабочую копию или препятствует изменению рабочей копии файла (например, средствами файловой системы) и обеспечивает, таким образом, исключительный доступ только тому пользователю, который работает с документом.

Многие системы управления версиями предоставляют ряд других возможностей:

* Позволяют создавать разные варианты одного документа, т. н. ветки, с общей историей изменений до точки ветвления и с разными — после неё.
* Дают возможность узнать, кто и когда добавил или изменил конкретный набор строк в файле.
* Ведут журнал изменений, в который пользователи могут записывать пояснения о том, что и почему они изменили в данной версии.
* Контролируют права доступа пользователей, разрешая или запрещая чтение или изменение данных, в зависимости от того, кто запрашивает это действие.

# Git-flow

Git-flow — наиболее популярная методология разработки проектов с использованием git.

Зачем нам вообще нужны какие-то методологии для работы с git?

Дело в том, что сам git не привязывает нас к какому-либо определённому способу разработки, и каждый разработчик, в теории, может работать с контролем версий так, как он хочет. Чтобы в таких условиях не погрузить наш репозиторий в хаос, нам нужно придумать и донести до всех разработчиков некий единый стандарт для работы с контролем версий в проекте.

Git-flow даёт нам готовый стандарт проверенный временем и уже известный многим разработчикам.

В то же время, нужно понимать, что методология git-flow не является единственно верной и на 100% универсальной. В Вашем проекте может существовать собственный подход к работе с git. Однако, если вы работаете или собираетесь работать с git в команде, то стоит знать о том, что такое git-flow и в чём его особенности.

**Суть ветвления в git-flow**

Всего в git-flow существует 5 типов веток, каждый из которых несёт определённую функциональную нагрузку.

**Ветка develop**

Ветка существует на протяжении всего процесса разработки. В данную ветку попадает стабильный код новых фич и багфиксов.

Тривиальные фичи, разработка которых умещается в один полноценный коммит, обычно идут прямо в эту ветку. Но те фичи, для реализации которых нужно уже несколько коммитов, выделяются в отдельные ветки feature-\*.

**Ветки feature-\***

Временные ветки, которые создаются для каждой нетривиальной фичи.

Каждая ветка feature отделяется от **develop** и мержится обратно в него. После завершения работы над фичей и финального мержа в develop, ветка удаляется. При мерже ветки в develop не должен использоваться fast-forward (флаг —no-ff).

Имя ветки должно соответствовать имени разрабатываемой фичи, например «feature-gamecenter-integration». В некоторых проектах префикс «feature-» опускается (тогда любая ветка без префикса — это ветка feature).

**Ветка master**

Ветка, куда поступают самые стабильные изменения, которые идут в релиз. Существует на протяжении всего процесса разработки.

В ветку master мержатся только изменения из веток **release** и **hotfix**. На каждый такой мерж создаётся тег с именем версии. Полный запрет использования fast-forward при мерже в ветку (флаг —no-ff).

По сути, в самой ветке не должно быть никаких коммитов, кроме коммитов мержей, каждый из которых должен быть отмечен тегом версии.

**Ветки release-\***

Временные ветки, создаваемые для подготовки новой версии к релизу. В эти ветки попадают правки багов и настройки перед релизом.

Ветка выходит из **develop** и может мержится в develop по ходу подготовки версии. Как только работа над версией заканчивается, происходит финальный мерж ветки в **master** и **develop**, после чего ветка удаляется, а коммиту мержа в master присваивается тег новой версии.

Имя ветки должно соответствовать выпускаемой версии, например «release-1.4».

**Ветки hotfix-\***

Временные ветки hotfix-\* создаются для правки критических проблем в релизной версии.

Ветка отходит от **master** и по завершению правок багов мержится обратно. Сама ветка после этого удаляется, а коммиту мержа в master присваивается тег новой версии.

Именем ветки обычно является новая версия с учётом версии фикса, например «hotfix-1.4.1» (первый хотфикс версии 1.4).

**Инстументы с поддержкой git-flow**

По сути нам нет необходимости в каких-то особенных инструментах для поддержания процессов разработки, все действия могут быть выполнены из практически любой графической оболочки над git, либо из командной строки.

Однако, есть удобное [расширение для git](https://github.com/nvie/gitflow), которое позволяет немного ускорить ежедневную работу по данной методологиии за счёт уменьшения количества вводимых команд в консоли и работы непосредственно на уровне абстракций git-flow.

Так же, git-flow поддерживается из коробки некоторыми популярными графическими оболочками такими как [SourceTree](https://www.sourcetreeapp.com/) и [SmartGit](http://www.syntevo.com/smartgit/).

Как в случае консольного расширения, так и в случае графических клиентов, мы работаем с более понятными командами вроде «Start Feature» и «Finish Feature», что позволяет проще смотреть на процесс разработки.

**Git-flow и непрерывная интеграция**

При разработке с использованием git-flow, одновременно существует две стабильные ветки, которые можно использовать для сборки билдов или выкладки кода на сервер: это ветки master и develop.

Сборки из ветки **master** можно собирать сразу по поступлении новых изменений. Любые новые правки в этой ветке — это стабильный production-код, который можно выкладывать на сервер (в случае разработки веб-ресурса) либо собирать в релизный бинарник.

Версии собранные из ветки master должны в обязательном порядке подвергаться всем существующим в проекте автоматизированным тестам.

Из ветки **develop** можно собирать ночные сборки (для тестирования) либо, по мере разработки, можно выкладывать код из этой ветки на тестовый сервер (в случае разработки веб-ресурса).

На версиях собранных из ветки develop следует прогонять unit-тесты.

**Когда не стоит применять git-flow**

Во первых, git-flow достаточно сложная методология, расчитанная на большие проекты. Если проект относительно небольшой и над ним работает команда всего из 3-4 человек, то применение git-flow может только усложнить процесс разработки.

Во вторых, git-flow в чистом виде применим только в тех случаях, когда в процессе разработки присутствует такое понятие как «релиз». Если изменения публикуются в продакшн по мере разработки, то смысла применять git-flow тоже нет.

В обоих случаях, можно либо взять git-flow, отказавшись от некоторых типов веток (master, release, hotfix), либо выбрать другую методологию.

# Шпаргалка по основным командам

git add

Команда git add добавляет содержимое рабочей директории в индекс (staging area) для последующего коммита. По умолчанию git commit использует лишь этот индекс, так что вы можете использовать git add для сборки слепка вашего следующего коммита.

git status

Команда git status показывает состояния файлов в рабочей директории и индексе: какие файлы изменены, но не добавлены в индекс; какие ожидают коммита в индексе. Вдобавок к этому выводятся подсказки о том, как изменить состояние файлов.

git dif

Команда git diff используется для вычисления разницы между любыми двумя Git деревьями. Это может быть разница между вашей рабочей директорией и индексом (собственно git diff), разница между индексом и последним коммитом (git diff --staged), или между любыми двумя коммитами (git diff master branchB).

git difftool

Команда git difftool просто запускает внешнюю утилиту сравнения для показа различий в двух деревьях, на случай если вы хотите использовать что-либо отличное от встроенного просмотрщика git diff.

git commit

Команда git commit берёт все данные, добавленные в индекс с помощью git add, и сохраняет их слепок во внутренней базе данных, а затем сдвигает указатель текущей ветки на этот слепок.

git reset

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 rm

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 mv

Команда git mv — это всего лишь удобный способ переместить файл, а затем выполнить git addдля нового файла и git rm для старого.

git clean

Команда git clean используется для удаления мусора из рабочей директории. Это могут быть результаты сборки проекта или файлы конфликтов слияний.

Шпаргалка по ветвлению и слиянию

git branch

Команда git branch — это своего рода “менеджер веток”. Она умеет перечислять ваши ветки, создавать новые, удалять и переименовывать их.

git checkout

Команда git checkout используется для переключения веток и выгрузки их содержимого в рабочую директорию.

git merge

Команда git merge используется для слияния одной или нескольких веток в текущую. Затем она устанавливает указатель текущей ветки на результирующий коммит.

git mergetool

Команда git mergetool просто вызывает внешнюю программу слияний, в случае если у вас возникли проблемы слияния.

git log

Команда git log используется для просмотра истории коммитов, начиная с самого свежего и уходя к истокам проекта. По умолчанию, она показывает лишь историю текущей ветки, но может быть настроена на вывод истории других, даже нескольких сразу, веток. Также её можно использовать для просмотра различий между ветками на уровне коммитов.

git stash

Команда git stash используется для временного сохранения всех незакоммиченных изменений для очистки рабочей директории без необходимости коммитить незавершённую работу в новую ветку.

git tag

Команда git tag используется для задания постоянной метки на какой-либо момент в истории проекта. Обычно она используется для релизов.

Шпаргалка по совместной работе и обновлению проектов

Не так уж много команд в Git требуют сетевого подключения для своей работы, практически все команды оперируют с локальной копией проекта. Когда вы готовы поделиться своими наработками, всего несколько команд помогут вам работать с удалёнными репозиториями.

git fetch

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

git pull

Команда git pull работает как комбинация команд git fetch и git merge, т.е. Git вначале забирает изменения из указанного удалённого репозитория, а затем пытается слить их с текущей веткой.

git push

Команда git push используется для установления связи с удалённым репозиторием, вычисления локальных изменений отсутствующих в нём, и собственно их передачи в вышеупомянутый репозиторий. Этой команде нужно право на запись в репозиторий, поэтому она использует аутентификацию.

git remote

Команда git remote служит для управления списком удалённых репозиториев. Она позволяет сохранять длинные URL репозиториев в виде понятных коротких строк, например «origin», так что вам не придётся забивать голову всякой ерундой и набирать её каждый раз для связи с сервером. Вы можете использовать несколько удалённых репозиториев для работы и git remote поможет добавлять, изменять и удалять их.

git archive

Команда git archive используется для упаковки в архив указанных коммитов или всего репозитория.

git submodule

Команда git submodule используется для управления вложенными репозиториями. Например, это могут быть библиотеки или другие, используемые не только в этом проекте ресурсы. У команды submodule есть несколько под-команд — add, update, sync и др. — для управления такими репозиториями.

Шпаргалка по осмотру и сравнению

git show

Команда git show отображает объект в простом и человекопонятном виде. Обычно она используется для просмотра информации о метке или коммите.

git shortlog

Команда git shortlog служит для подведения итогов команды git log. Она принимает практически те же параметры, что и git log, но вместо простого листинга всех коммитов, они будут сгруппированы по автору.

git describe

Команда git describe принимает на вход что угодно, что можно трактовать как коммит (ветку, тег) и выводит более-менее человекочитаемую строку, которая не изменится в будущем для данного коммита. Это может быть использовано как более удобная, но по-прежнему уникальная, замена SHA-1.

Шпаргалка по отладке

В Git есть несколько команд, используемых для нахождения проблем в коде. Это команды для поиска места в истории, где проблема впервые проявилась и собственно виновника этой проблемы.

git bisect

Команда git bisect — это чрезвычайно полезная утилита для поиска коммита в котором впервые проявился баг или проблема с помощью автоматического бинарного поиска.

git blame

Команда git blame выводит перед каждой строкой файла SHA-1 коммита, последний раз менявшего эту строку и автора этого коммита. Это помогает в поисках человека, которому нужно задавать вопросы о проблемном куске кода.

git grep

Команда git grep используется для поиска любой строки или регулярного выражения в любом из файлов вашего проекта, даже в более ранних его версиях.