**Базовый курс**

**Cистема контроля версий (VCS)** — программное обеспечение для облегчения работы с изменяющейся информацией. Основная задача программы – сохранять разные версии проекта и перемещаться между ними.

***Полная история изменений каждого файла за длительный период***. Это касается всех изменений, внесенных огромным количеством людей за долгие годы. Изменением считается создание и удаление файлов, а также редактирование их содержимого. Различные инструменты VCS отличаются тем, насколько хорошо они обрабатывают операции переименования и перемещения файлов. В историю также должны входить сведения об авторе, дата и комментарий с описанием цели каждого изменения. Наличие полной истории позволяет возвращаться к предыдущим версиям, чтобы проводить анализ основных причин возникновения ошибок и устранять проблемы в старых версиях программного обеспечения.

***Ветвление и слияние***. Создание «веток» позволяет иметь несколько независимых друг от друга направлений разработки, а также выполнять их слияние, чтобы разработчики могли проверить, что изменения, внесенные в каждую из веток, не конфликтуют друг с другом. Многие команды разработчиков программного обеспечения создают отдельные ветки для каждой функциональной возможности, для каждого релиза либо и для того, и для другого.

***Отслеживаемость***. Возможность отслеживать каждое изменение, внесенное в программное обеспечение, и связывать его с ПО для управления проектами и отслеживания ошибок. История с комментариями во время чтения кода помогает понять, что этот код делает и почему действие реализовано именно таким образом. Благодаря этому разработчики могут вносить корректные и совместимые изменения в соответствии с долгосрочным планом разработки системы.

Проверить, работает ли git с помощью команды

git --version

Добавить имя и email пользователя с помощью команд

git config --global user.email "email@mail.com"

git config --global user.name "User Name"

Команды:

**git init**

Эта команда создаёт в текущем проекте (папке) новую папку с именем .git, содержащую все необходимые файлы репозитория — структуру Git репозитория. Происходящее в этом проекте будет отслеживаться программой, но на этом этапе ваш проект ещё не находится под версионным контролем. Если вы хотите добавить под версионный контроль существующие файлы (в отличие от пустого каталога), вам стоит добавить их в индекс и осуществить первый коммит изменений. Добиться этого вы сможете запустив команду git add несколько раз, указав индексируемые файлы, а затем выполнив git commit:

**git add (имя файла) / git add --all (все файлы в проекте) или просто .**

Команда git add добавляет содержимое рабочего каталога 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. При изменении файла и новом коммите даже уже отслеживаемый ранее файл нужно снова добавлять в индекс командой **add**.

**git status**

Команда git status показывает состояния файлов в рабочем каталоге 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 commit -m “initial commit”**

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

**git log**

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

**git checkout df7d**

После того как вы нашли ссылку на нужный коммит в истории, для перехода к нему можно использовать команду git checkout. Команда git checkout — это простой способ «загрузить» любой из этих сохраненных снимков на компьютер разработчика.

**git checkout master**

переключение на ветку master в последний коммит (вернуться на актуальное состояние после переключения между версиями).

**git diff**

Показывает различия между текущей версией файлов и последним коммитом

**git diff [первая ветка]...[вторая ветка]**

Показывает разницу между содержанием коммитов двух веток  
**git show [коммит]**

Выводит информацию и показывает изменения в выбранном коммите

**git branch**

Список именованных веток коммитов с указанием выбранной ветки

**git branch [имя ветки]**

Создаёт новую ветку

**git merge [имя ветки]**

Вносит изменения указанной ветки в текущую ветку

**git branch -d [имя ветки]**

Удаляет выбранную ветку

**git log graph**

Графическое представление истории коммитов в разных ветках

**git clone https:….**

Клонируем удаленный репозиторий на свой компьютер (и работаем дальше локально), который автоматически уже связан с удаленным. Папка при этом репозиторием не становится. Нужно перейти в скопированную и работать там.

**git push**

Обновляем данные из локального репозитория в интернет

**git pull**

Обновляем данные из интернета в локальный репозиторий. При этом команда также сразу делает слияние.

**Углубленный курс**

Системы управления Git-репозиториями GitHub и GitLab — не единственные в своём роде, но самые популярные, поэтому знакомиться мы будем именно с ними. Самое главное назначение таких систем — это хранение удалённых репозиториев.

Преимущества систем управления Git-репозиториями:

● Хранение удалённых репозиториев

● Расширенное управление доступами (доступ только нужных пользователей, на определенное время или, например, с правами только на чтение)

● Наглядное представление репозиториев

● Управление задачами

● Связь изменений с задачами

● Обсуждение изменений в коде

|  |  |  |
| --- | --- | --- |
| **Отличие** | **GitHub** | **GitLab** |
| Возможность размещения на своём сервере | Нет | Да |
| Бесплатные возможности для командной работы и DevOps | Нет | Да |
| Возможность доработок | Нет | Да |

Область применния

|  |  |
| --- | --- |
| **GitHub** | **GitLab** |
| Open-source-проекты  Публичный доступ  На серверах GitHub  Нет автоматизации  DevOps-процессов | Коммерческие проекты  Ограниченный доступ  На собственном сервере  Автоматизация  DevOps-процессов |

Подключения к удаленному репозиторию

|  |  |  |
| --- | --- | --- |
| **№** | **Локальный репозиторий** | **Удалённый репозиторий** |
| 1 | Нет | Да |
| 2 | Нет | Нет |
| 3 | Да | Нет |

1) Нет локального, а удалённый есть и он не пустой, в нём уже лежат какие-то файлы. Это происходит всегда, когда вы начинаете работу над уже существующим проектом.

*git clone …*

Если вы будете вносить в такую рабочую копию какие-то изменения, вы сможем сразу их отправлять на удалённый сервер в исходный репозиторий, поскольку они уже автоматически связаны.

2) Вы начинаете работать над новым проектом, и у вас ещё нет ни локального, ни удалённого репозитория. Вы можете сразу добавить в него какие-либо файлы и просто склонировать его к себе на компьютер, как было показано выше, или создать локальный репозиторий и связать его с удаленным.

*git init*

*git add …*

*git commit -m “initial commit”*

*git branch -M master*

*git remote add origin …(origin — это название удалённого репозитори по умолчанию)*

*git push -u origin master(параметр “-u”, который говорит о том, что необходимо установить связь текущей ветки в локальном репозитории с соответствующей веткой в удалённом)*

3) Есть локальный репозиторий с проектом, а удалённого репозитория ещё нет. В этом случае первым действием вам нужно создать пустой удалённый репозиторий и перенести проект как в пункте 2.

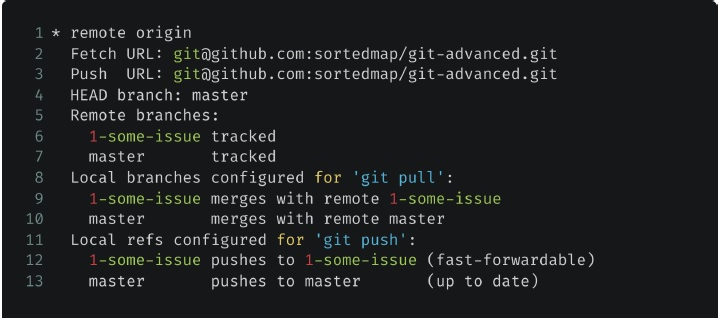
Синхронизация изменений в ветках

**git push -u origin 1-some-issue**

Вывод на удаленный репозитарий ветки 1-some-issue, master не обновляется

**git remote show origin**

Просмотреть, какие локальные ветки связаны с ветками удалённого репозитория и какой у них статус.



В начале этого вывода написано “\* remote origin”, что указывает на то, что эта информация относится к состоянию локального репозитория относительно удалённого репозитория origin. Далее указаны пути к этому репозиторию для операций fetch и push и то, что основная ветка — master.

Затем идёт блок информации о состоянии веток. Вначале перечислены удалённые ветки и информация о том, что они отслеживаются локальными ветками (в примере выше — пометка “tracked”).

После этого указан перечень веток и их конфигурация относительно удалённых веток для команды git pull. В примере выше, в частности, написано, что локальные ветки будут сливаться с удалёнными ветками при выполнении команды git pull.

И в конце блока идёт конфигурация локальных веток для команды git push. Здесь отображается информация о том, в какие удалённые ветки будет отправляться информация из локальных и каков статус локальных веток.

Статус “up to date” говорит о том, что локальная и удалённая ветки синхронизированы.

Статус “fast-forwardable” сообщает нам о том, что в локальной ветке произошли изменения и мы их можем отправить в удалённую ветку.

Здесь также может быть статус “local out of date”, который говорит о том, что локальная ветка отстаёт от удалённой: в удалённом репозитории в этой ветке произошли какие-то изменения, которых ещё нет локально.

Вы можете скачать все изменения из ветки удалённого репозитория в локальный, но без обновления самой ветки обновлять не будем. Это делается при помощи команды:

git fetch

После её выполнения вы можете посмотреть, какие изменения были внесены в удалённую ветку, для этого можно воспользоваться командой git log, в параметрах которой указав имена удалённой и локальной веток:

git log origin/1-issue ^1-issue

Такой вариант команды выведет те коммиты, которые отличают удалённую ветку от локальной. Содержимое каждого такого коммита вы можете просмотреть командой:

git diff hash

В данном случае, hash — это хэш коммита. Если вы захотите влить удалённую ветку в свою локальную, вы можете выполнить уже известную вам команду:

git pull

Эту команду также можно выполнять для отдельных веток, указав имя удалённого репозитория и название нужной ветки:

git pull origin 1-issue

Таким образом, команда git pull затягивает изменения из удалённого репозитория и сливает их с изменениями в локальном репозитории, в то время как команда git fetch их просто затягивает, но слияния не производит.

**Настройка связей с удаленными репозиториями**

**git remote -v**

Вывод информации о путях к вашим удалённым репозиториям

**git remote remove origin**

**git remote add origin path**

Замена репозитория origin (удалить текущий репозиторий и добавить новый)

**git remote set-url origin new-path**

Замена пути к текущему репозиторию

**git remote set-url --push origin new-path**

Замена пути для каждой из операций — fetch и push — отдельно (тут для push)

**git remote add vendor vendor-repo-path**

Добавить к своему локальному репозиторию несколько удалённых

**git push -u origin master**

**git push -u vendor master**

Для отправки изменений в каждый репозиторий отдельно

**git fetch –all**

Затянуть изменения из всех репозиториев в свой локальный. Это можно сделать только без слияния, потому что возможны конфликты не только кода из удалённого репозитория и кодом из локального, но и кодов из двух удалённых репозиториев между собой

**git log origin/master ^master**

**git log vendor/master ^master**

Просмотр затянутых изменений и сравнивнение удалённых веток с ветками локального репозитория

**git merge origin/master**

**git merge vendor/master**

Вливать нужные вам изменения

Способ, который позволяет сделать так, чтобы однократный “git push” отправлял изменения сразу в несколько репозиториев:

Для этого нужно придумать название некого виртуального репозитория, который будет в себе содержать пути ко всем удалённым. Пусть он называется “all”. Давайте добавим его тоже с таким же путём, как путь к нашему первому репозиторию:

git remote add all path-1

Дальше вы можете добавить к этому репозиторию пути:

git remote set-url --add --push all path-1

git remote set-url --add --push all path-2

И теперь вы можете отправлять наши изменения сразу во все репозитории, используя имя all:

git push all master

Разрешение конфликтов

Конфликт - это ситуация, при которой на сервере в удалённом репозитории и у вас в локальном репозитории созданы разные коммиты.

Такие конфликты бывают двух видов: простые, которые могут быть разрешены автоматически, самим Git-ом, и сложные, которые можно разрешить только вручную.

Чтобы создать простой конфликт, вам нужно внести изменения в один файл в локальной ветке и в другой файл в удалённой. В этом случае команда “git status” покажет вам, что ваш локальный репозиторий на один коммит впереди последней версии удалённого репозитория, которую вы к себе загружали:



И, казалось бы, надо просто выполнить “git push”, чтобы отправить этот коммит в удалённый репозиторий и чтобы оба репозитория синхронизировались, но после выполнения этой команды вы видите, что ничего не получилось и появилось “страшное” сообщение об ошибке.

На самом деле, в этом сообщении нет ничего страшного. В нём лишь написано, что удалённый репозиторий содержит изменения, которых у вас ещё нет локально. И также в этом сообщении Git вам предлагает выполнить вначале команду “git pull”, чтобы загрузить эти изменения:

git pull

После выполнения этой команды может появиться текстовый редактор, в котором будет предложено ввести сообщение коммита. Поскольку у нас был конфликт версий, то есть две конфликтующие версии кода, при затягивании изменений Git попытается их слить с вашей рабочей копией (изменения одного файла будут взяты из локального репозитория, изменения второго — из удалённого).

После этого слияние файлов удалённого репозитория с файлами локального завершится, и у вас появится ещё один коммит, но он появится только локально. Чтобы другие разработчики его увидели и увидели ваши изменения в коде, которые вы делали до этого, вам нужно отправить его в удалённый репозиторий:

git push

Теперь давайте рассмотрим сложный конфликт, который Git не сможет разрешить автоматически. Создать его можно, внеся разные изменения в одну и ту же строку одного и того же файла. В этом случае вам также следует сначала выполнить команду:

git pull

Появится сообщение об ошибке: “Automatic merge failed; fix conflicts and then commit the result”. Git, опять же, сольёт файл из удалённого репозитория с файлом, который был в вашем локальном репозитории, но пометит в этом файле обе версии кода, которые он не смог слить, и не сделал коммит. Такой конфликт слияния вам нужно разрешить вручную: отредактировать этот файл и выбрать ту часть, которая будет более правильной, или объединить их как-то по-своему, после чего сохранить и закоммитить сделанные изменения.

**git show**

Содержимое последнего коммита

Сравнение версий и просмотр изменений

**git diff [первый коммит]...[второй коммит] имя файла**

Показывает разницу между содержанием двух коммитов файла

**git diff [первый коммит]...[второй коммит]**

Показывает разницу между содержанием двух коммитов по всему проекту

**git diff [первый файл]...[второй файл]**

Показывает разницу между содержанием двух файлов, даже если один из них не в репозитории

**git blame имя файла**

Показывает кто и когда менял файлы

Игнорирование изменений, gitignore

Не рекомендуется хранить в Git-репозиториях

●Лог-файлы (логи)

●Файлы, загруженные пользователями

●Служебные файлы сред разработки

●Внешние библиотеки

●Файлы локальной конфигурации

●Файлы операционной системы

●Очень большие файлы

\*.log – исключение всех файлов с расширением .log

Log/ - исключение папки с именем Log. Без / будет игнорироваться и папка и файл с таким именем

Часто бывает необходимо задать напрямую правило, указывающее, что конкретный файл или папку игнорировать не следует. К примеру, мы можем начать игнорировать все лог-файлы, лежащие в папке “log”, но какой-то специальный файл исключить из игнорирования, для чего перед его именем нужно поставить восклицательный знак:

logs/\*.log

!logs/special.log

Если бы мы в “.gitignore” указали всю папку целиком, то потом исключить из неё отдельный файл было бы нельзя. То есть вот так работать не будет:

logs/

!logs/special.log

Отмена не сохранённых изменений

При работе с программным кодом часто бывает так, что вы начали вносить какие-то изменения, а потом решили, что этого делать не нужно. Это может касаться как отдельного файла, так и проекта целиком.

**git restore index.html**

Возврат изменений к предыдущему коммиту, если еще не сделали git add

**git restore --staged index.html**

Возврат изменений к предыдущему коммиту, если уже добавили в индекс

**git reset –hard**

Возврат всех изменений в проекте, даже если они добавлены в индекс

Эта команда не удаляет те файлы, которые не отслеживаются и ещё ни разу не добавлялись в индекс — файлы, которые находятся в состоянии “untracked”. Для того, чтобы их удалить, можно воспользоваться командой:

**git clean –f**

Использовать эти команды нужно осторожно, поскольку они удаляют все незакоммиченные изменения безвозвратно.

Способ удаления файла из отслеживания системой контроля версий Git. Представьте, что у вас в проекте есть какой-то файл, который не следует хранить в Git’е, но вы этот файл случайно закоммитили вместе с какими-то другими изменениями в коде. Добавление его в “.gitignore” или комманда “git restore” не помогут, изменения также продолжат в нём отслеживаться.

**git rm --cached file\_name**

Отмена сохранённых изменений

**git checkout b45a983 index.html**

Вернуть файл к конкретной версии на момент определённого коммита. И этот файл будет возвращён к нужной версии и сразу добавлен в индекс. Вы можете закоммитить эти изменения — то есть зафиксировать эту версию файла, а можете их пока отменить: либо командой “git reset --hard”, которая может быть полезна сразу для нескольких файлов, либо обычно предлагаемой в таких случаях Git’ом командой:

**git restore --staged index.html**

Она переведёт изменения из индекса в изменённое состояние. После чего удалить эти изменения совсем можно командой:

**git restore index.html**

Если же вы хотите отменить коммит целиком, вы можете воспользоваться командой “git revert”. С помощью неё вы можете отменить как последний коммит, так и любой предыдущий. Пример выполнения команды:

**git revert b45a983**

При этом указывать нужно коммит к состоянию которого нам нужно вернуть ветку, а не тот, который нужно отменить. Обратите внимание, что “git revert” осуществляет отмену конкретного коммита, то есть все последующие коммиты не будут отменены.

Если указывать при этом параметр “--no-commit”, вы сможете откатиться на несколько версий назад, не делая множество коммитов, а сделав единственный коммит в конце.

Удаление (сброс) коммитов

**git reset --soft [имя коммита]**

Сброс изменения последнего коммита и отправка его в индекс. Удаляется только сам коммит.

**git reset [имя коммита]**

Сброс изменения последнего коммита без отправки его в индекс.

**git reset --hard [имя коммита]**

Bзменения в последующих коммитах просто будут безвозвратно удалены (вместе с измениями)

**git commit --amend -m 'new comment'**

Сообщение коммита изменится на новое “new commet”.

**git commit --amend --no-edit**

Если вы вдруг поняли, что забыли добавить какой-то файл в последний коммит, вам после его изменения нужно его просто добавить в индекс

Отмена слияния веток

**git reset --merge <commit>**

Отмена получившегося слияния веток. Нужно указать коммит, до которого вам нужно откатиться (предшествующий коммиту со слиянием)

**git merge --abort**

Отмена не получившегося слияния из-за конфликта

Откладывание изменений

Система контроля версий Git позволяет откладывать незакоммиченные изменения. Представьте, что вы работаете над какой-то задачей, и тут к вам приходит ваш руководитель и просит срочно сделать другую задачу. Сделать коммит при этом вы не можете, поскольку коммитить принято только законченные задачи. Вы также не сможете переключиться на другую ветку. На этот случай Git предлагает специальный инструмент, позволяющий временно отложить сделанные изменения. Чтобы изменения отложить, вам нужно просто выполнить команду:

**git stash**

Если после выполнения этой команды вы посмотрите статус рабочей директории, то в ней не будет никаких незакоммиченных изменений, а её состоянии будет соответствовать последнему коммиту. После этого вы сможете перейти к выполнению другой задачи и, когда её закоммитите, сможете вернуть более ранние изменения из отложенных командой:

**git stash pop**

При этом отложенные изменения будут слиты с теми, которые вы только что закоммитили. Возможно, что это произойдёт автоматически, и тогда вы увидите просто сообщение со словом “auto-merging”, либо же произойдёт конфликт слияния, который вам придётся разрешать вручную аналогично любому другому конфликту слияния, что вы уже умеете делать.

**git stash list**

Список отложенных изменений

Команда “git stash” позволяет откладывать несколько порций изменений. Вы можете внести изменения, отложить их с помощью “git stash”, потом внести новые изменения, и снова отложить их с помощью “git stash”, и они будут сохранены отдельно. Теперь, если вы будете выполнить команду “git stash pop”, вы будете возвращать изменения в обратном порядке: сначала те, которые были добавлены самой последней командой “git stash”, затем — предпоследней, и так далее.

При необходимости вы также можете отменить последнюю порцию отложенных изменений командой:

**git stash drop**

Вы также можете выполнять эту команду несколько раз, чтобы последовательно отменить все порции изменений, отложенных последовательным выполнением команды “git stash”.

Перемещение изменений

**git merge dev**

Сливаем в master ветку dev. И в случае удаления ветки dev восстановить историю ее коммитов уже не получится.

**git rebase dev**

Перебазируем ветку dev на ветку master. Поскольку мы переносим эти изменения после изменений в ветке master, то это будут уже немного другие коммиты, которые в себе будут содержать

изменения перенесённых коммитов, наложенные на изменения из ветки master. И, разумеется, хэши этих коммитов тоже будут другие. Но каждый из них будет содержать в себе изменения из соответствующих перенесённых коммитов. Преимущество этого способа заключается в том, что история изменений остаётся прямой и удобной для просмотра и анализа, но при этом есть и недостаток: в будущем по такой истории будет достаточно сложно понять, что произошёл именно git rebase.

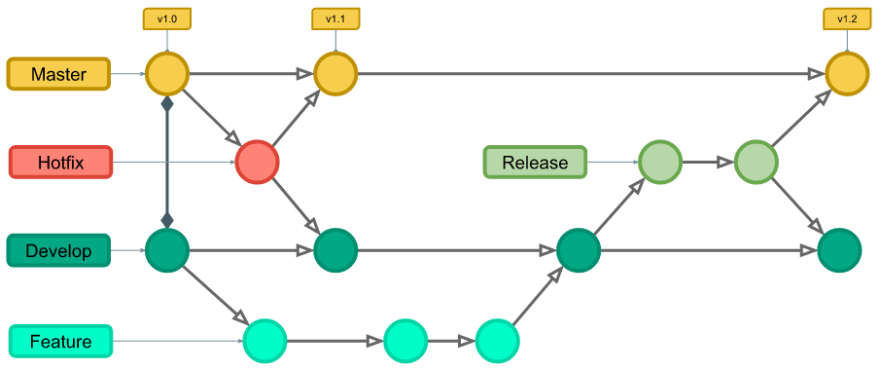
**git cherry-pick D**

Этот механизмпозволяет перенести какой-то один или несколько коммитов из одной ветки в

другую без переноса других изменений. Например, в какой-то ветке в отдельном коммите был исправлен критичный баг, и также сделано много другой работы. Пусть этим коммитом в нашем случае будет коммит, обозначенный буквой D. И вы не хотите вливать эту ветку целиком, а хотите перенести только изменения из коммита D. И перебазировать при помощи rebase тоже не хотите, поскольку хотите сохранить все ветки без изменений.

Модели работы с ветками в Git

Первая система — более классическая, называется “Git flow”. “Flow” — от слова “поток”, имеется в виду поток непрерывных изменений и выкатки — то есть deployment-а — программного обеспечения. Обычно ветки в этой модели выглядят примерно таким образом:

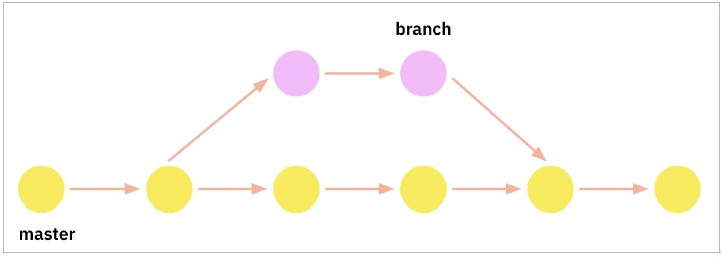


Есть ветка “master”, которую вы тщательно оберегаете от вливания в неё непроверенных изменений. Есть также ветка “develop”, в которую вливаются все свежие изменения и в которой проверяется общая работоспособность кода. Эту ветку в разных проектах могут называть по-разному — “dev”, “devel” или как-то ещё. И вот от неё уже отпочковываются ветки, в которых разрабатываются различные фичи продукта, над которым работают программисты. Как только работа над какой-то задачей завершается, результат вливается в ветку “release”, уже тщательно тестируется и потом заливается в ветку “master”.

Также в такой системе часто присутствует ветка для внедрения быстрых и мелких изменений. Её называют “hotfix” или “hotfixes”. Hot — это горячий, fix —это исправление. Эта ветка позволяет обходить основной процесс выкатки изменений, который может занимать несколько дней, а то и недель.

Основная особенность такой модели состоит в том, что в ней довольно много веток, которые живут долго. Во-первых, каждая фича лежит в своей ветке и может разрабатываться довольно долго. А во-вторых, есть несколько постоянных веток разного назначения. Их, кстати, может быть и больше. В одном из проектов, в которых я работал, была ещё отдельная ветка “test” для отдельной категории тестировщиков.

Вторая модель называется “trunk-based development” или сокращённо TBD. Trunk в переводе с английского означает “ствол”. То есть это разработка, основанная на едином “стволе” — единой ветке. Этой веткой является ветка “master”. Никаких других постоянных веток при этой модели не создаётся, а ветки для отдельных задач создаются ненадолго и удаляются после вливания в ветку “master”:



В идеальной ситуации в этих ветках реализуются не сразу целиком все задачи, а их части, которые можно быстро выливать по-отдельности в ветку “master” и которые можно в ней выключать. Это делается с помощью так называемых “feature flags” — специальных переключателей, с помощью которых можно включать или выключать тот или иной функционал в итоговой ветке “master”.

Управление включением или выключением тех или иных фич происходит посредством, например, конфигурации приложения. А в самом приложении пишутся условия, которые проверяют, установлен ли тот или иной флаг или нет, например, так:



Общепринятые правила работы в Git

**Именование веток**. Первая группа правил относится к именованию веток в репозитории. Имена веток должны:

● Состоять из латинских букв в нижнем регистре, цифр и дефисов. Иные символы в именах веток использовать не рекомендуется.

● В начале содержать номер задачи, решаемой в этой ветке.

● Описывать суть задачи, решаемой в этой ветке.

Также, если в компании используется модель ветвления “git flow”, имена некоторых веток могут быть стандартными:

● master / main — главная ветка;

● dev / devel / develop — ветка, в которую вливаются результаты выполнения всех задач перед вливанием в ветку master;

● test / staging — ветка для выкладывания задач, которые необходимо протестировать перед вливанием в главную ветку или ветку dev;

● prod / production / releases — ветка для релизов;

● fixes / hot-fixes / hotfixes — ветка для “быстрых” и “горячих” изменений.

**Содержимое коммитов.** Коммиты не должны быть хаотичными. Они должны содержать либо завершённую работа, либо логически понятный фрагмент такой работы.

Также допустимо делать промежуточные коммиты, если выполнение задачи занимает больше одного рабочего дня, чтобы не потерять работу за прошедший день. Такие коммиты обычно принято делать в конце рабочего дня. Комментарии к коммитам. Комментарии к коммитам должны отражать суть содержимого этих коммитов и соответствовать стандартам, принятым в компании, если таковые есть. Также рекомендуется их писать по-английски.