**Контроль версий**, также известный как управление исходным кодом, — это практика отслеживания изменений программного кода и управления ими.

**Система контроля версий** — это программный инструменты, который позволяет управлять изменениями в исходном коде и других файлов проекта. Она позволяет отслеживать изменения, вносить их, восстанавливать предыдущие версии файлов и совместно работать над проектом с другими разработчиками.

Системы контроля не ограничиваются программным кодом. Они могут использоваться для управления версиями любых файлов: документации, графических фалов, настроек проекта.

Наиболее популярные типы VCS:

**Централизованные системы контроля версий (Centralized Version Control Systems, CVCS)** - такие как Subversion (SVN). В CVCS все файлы и история изменений хранятся на центральном сервере, а разработчики получают последние версии файлов из этого сервера и отправляют изменения на сервер. Это позволяет иметь единую историю изменений и контроль над доступом к файлам.

**Минусы:**

* Если сервер выйдет из строя на какое-то время, контроль версий будет недоступен. А если повредится жесткий диск, то вы потеряете всю историю проекта, не считая единичных снимков репозитория, которые сохранили другие разработчики.
* Если кто-то испортит рабочую ветку в SVN, другие разработчики не смогут вносить изменения, пока она не будет исправлена. В Git такой блокировки нет. Все могут спокойно работать в своих локальных репозиториях.
* Сложное ветвление. Чтобы создать новую ветку, нужно скопировать весь рабочий каталог.

**Распределенные системы контроля версий (Distributed Version Control Systems, DVCS) -** такие как Git и Mercurial. В DVCS каждый разработчик получает полную копию репозитория с историей изменений на свой локальный компьютер. Это позволяет им работать независимо друг от друга, без необходимости постоянного подключения к центральному серверу. Изменения могут быть коммитированы и синхронизированы с другими разработчиками при необходимости.

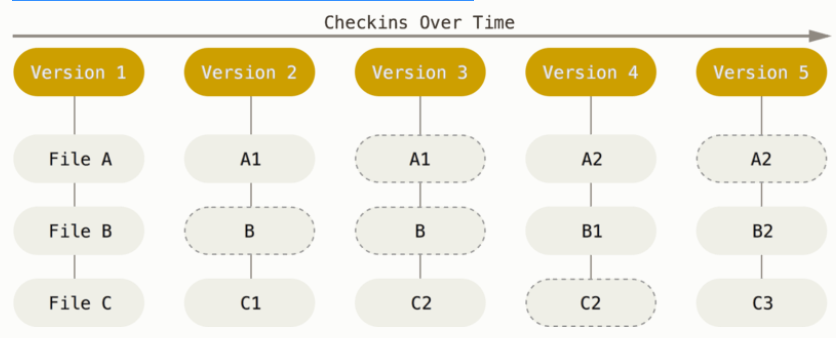
Если сервер, через который разработчики обменивались данными, умрет любой клиентский репозиторий может быть скопирован на другой сервер для продолжения работы. Каждая копия репозитория является полным бэкапом всех данных.

Программное обеспечение контроля версий отслеживает все вносимые в код изменения в специальной базе данных. При обнаружении ошибки разработчики могут вернуться назад и выполнить сравнение с более ранними версиями кода для исправления ошибок.

**GIT**

Большинство систем контроля версий 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.

В Git, каждый раз, когда вы делаете коммит, система запоминает, как выглядит каждый файл в этот момент, и сохраняет ссылку на этот снимок. Если файлы не были изменены, то создается просто ссылка на предыдущую версию идентичного файла, который уже сохранен. Git представляет свои данные как, скажем, **поток снимков**.



Почти все операции в гит выполняются локально, если только вам не нужно получать какие-то данные с сервера. 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.

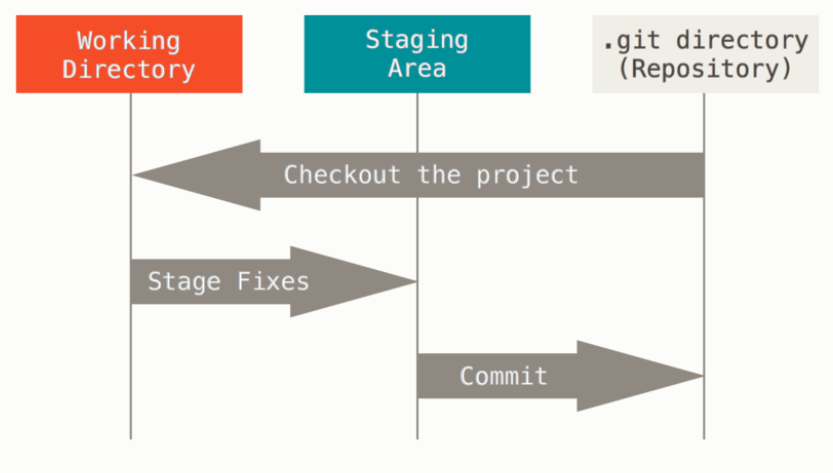
В Git для всего вычисляется хеш-сумма, и только потом происходит сохранение. В дальнейшем обращение к сохранённым объектам происходит по этой хеш-сумме. Это значит, что невозможно изменить содержимое файла или каталога так, чтобы Git не узнал об этом.

Файлы в GIT имеют 3 основных **состояния:**

**изменён (modified)** – файлы, которые поменялись, но еще не были закомичены.

**индексирован (staged) –** измененный файл, отмеченный для включения в следующий коммит.

**зафиксирован (committed):** файл сохранен в локальной базе.



Есть 3 основные секции:

**рабочая копия (working tree)** – снимок одной версии проекта. Эти файлы извлекаются из сжатой бд в каталоге Git и помещаются на диск, для того чтобы их можно было использовать или редактировать.

**Область индексирования (staging area) –** файл, обычно находящийся в каталоге Git. В нем содержится информация о том, что попадет в следующий коммит. Её техническое название на языке Git — «индекс».

**каталог Git (Git directory) -**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.

**Базовый подход в работе с Git выглядит так:**

1. Изменяете файлы вашей рабочей копии.
2. Выборочно добавляете в индекс только те изменения, которые должны попасть в следующий коммит, добавляя тем самым снимки только этих изменений в индекс.
3. Когда вы делаете коммит, используются файлы из индекса как есть, и этот снимок сохраняется в ваш каталог Git.

Если определённая версия файла есть в каталоге Git, эта версия считается **зафиксированной** (**committed**). Если файл был изменён и добавлен в индекс, значит, он **индексирован** (**staged**). И если файл был изменён с момента последнего распаковывания из репозитория, но не был добавлен в индекс, он считается **изменённым** (**modified**).

В состав Git входит утилита **git config**, которая позволяет просматривать и настраивать параметры, контролирующие все аспекты работы Git, а также его внешний вид. Эти параметры могут быть сохранены в трёх местах:

1. **SYSTEM.** Файл **[path]/etc/gitconfig** содержит значения, общие для всех пользователей системы и для всех их репозиториев. Если при запуске git config указать параметр **--system**, то параметры будут читаться и сохраняться именно в этот файл. Является системным файлом.
2. **GLOBAL.** хранит настройки конкретного пользователя. ~/.config/git/config, C:\Users\$USER.
3. **LOCAL.** Файл config  в каталоге Git (т. е. .git/config) репозитория, который вы используете в данный момент, хранит настройки конкретного репозитория.

Настройки на каждом следующем уровне перекрывают настройки предыдущих уровней. Чтобы посмотреть все установленные настройки и узнать, где именно они заданы, используйте команду:

$ git config --list --show-origin

Чтобы посмотреть конкретную настройку:

$ git config user.name

**Основные настройки:**

* Имя пользователя $ git config --global user.name "John Doe"
* Email $ git config --global user.email johndoe@example.com
* Текстовый редактор, чтобы набирать сообщение git. $ git config --global core.editor emacs
* Имя ветки по умолчанию (изначально **master**) $ git config --global init.defaultBranch main

**.gitignore** – файл, с перечислением шаблонов файлов, которые должны игнорироваться гитом. Такие файлы не будут отображаться в списке неотслеживаемых. Таким образом исключается их случайное добавление в индекс и коммит.

В гитигнор добавляют файлы генерируемые EDE, логи, автоматически создаваемую документацию, каталог с собранным проектом и скомпилированными классами, каталоги библиотек.

Может быть несколько фалов .gitignore. Они применяются к тем каталогам, в которых находятся.

К шаблонам в файле .gitignore применяются следующие правила:

* Пустые строки, а также строки, начинающиеся с #, игнорируются.
* Стандартные шаблоны являются глобальными и применяются рекурсивно для всего дерева каталогов.
* Чтобы избежать рекурсии используйте символ слеш (/) в начале шаблона.
* Чтобы исключить каталог добавьте слеш (/) в конец шаблона.
* Можно инвертировать шаблон, использовав восклицательный знак (!) в качестве первого символа.

**Чтобы получить git-репозиторий можно:**

**Клонировать** существующий Git репозиторий.

**Создать** Git репозиторий в существующем каталоге. Выполняется это командой

$ git init

Эта команда создаёт в текущем каталоге новый подкаталог с именем .git, содержащий все необходимые файлы репозитория — структуру Git репозитория.

**git clone <url> [catalogname] –** получить Git репозиторий с сервера в указанный каталог. Cкачивает все данные для этого репозитория и извлекает рабочую копию последней версии. В качестве транспортных протоколов можно использовать http или SSH.

**git clone <url> -o <remote> -** клонировать репозиторий, и дать удаленному репозиторию имя (по умолчанию origin).

**git status –** показывает какие файлы в каком состоянии находятся, текущую ветку и различия с веткой на сервере.

$ git status

On branch master

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

nothing to commit, working tree clean

**git add –** добавить файл(файлы, либо изменения файла) в индекс. Git начнет отслеживать изменения этого файла. Команда git add принимает параметром путь к файлу или каталогу, если это каталог, команда рекурсивно добавляет все файлы из указанного каталога в индекс.

Git не будет сам добавлять созданные файлы в индекс. Это предотвратит попадание в коммит каких то сгенерированных бинарных файлов или каких-либо других, которые вы не думали добавлять.

Если добавить файл в индекс, а потом снова изменить его, то в git status он будет отображаться как проиндексированный и непроиндексированный одновременно. Git индексирует файл в точности в том состоянии, в котором он находился, когда вы выполнили команду git add. Вам придется вновь выполнить команду git add чтобы проиндексировать последнюю версию файла.

**git diff -** сравнивает содержимое вашего рабочего каталога с содержимым индекса. Результат показывает ещё не проиндексированные изменения.

**git diff –staged -** посмотреть, что вы проиндексировали и что войдёт в следующий коммит.

**git commit** – зафиксировать изменения в каталоге Git. Коммит сохраняет состояние вашего индекса. Все, что вы не проиндексировали, остается висеть в working tree.

Откроется редактор по умолчанию, в котором будет приглашение набрать message коммита.

Message можно так же указать после флага **-m.**

**Git commit -a –** позволяет пропустить этап индексирования. С этим флагом гит автоматически индексирует каждый уже отслеживаемый файл перед коммитом.

**Git commit –ammend –** позволяет внести изменения в последний коммит. Новый коммит не будет создаваться, проиндексированные изменения попадут в последний коммит. Так же можно изменить сообщение коммита.

**Git rm –** удалить файл из git. Удаляет файл из индекса, и из рабочего каталога. Если просто удалить файл из рабочего каталога, то он будет показан как непроиндексированные изменения. При git rm удаление файла попадет в индекс.

`git rm` также поддерживает использование шаблонов и масок для удаления нескольких файлов сразу. Например, `git rm \*.txt` удалит все файлы с расширением .txt

Если вы уже проиндексировали файл, вы должны использовать принудительное удаление **-f**. Это сделано для повышения безопасности, чтобы предотвратить ошибочное удаление данных, которые не были закоммичены и которые нельзя восстановить из git.

Опция **--cached** позволяет удалить файл из индекса, оставив его в рабочем каталоге. Может быть полезно, если вы забыли добавить файл в .gitignore и по ошибке проиндексировали.

В git rm можно передать файлы, каталоги или шаблоны.

**Git mv –** переименовать либо переместить файл.

**Git log –** посмотреть историю коммитов. По умолчанию отображает текущую ветку, последние коммиты отображаются вверху. Перечисляет коммиты с их SHA-1 контрольными суммами, именем и электронной почтой автора, датой создания и сообщением коммита.

**-p – patch –** показывает разницу, внесенную в каждый коммит.

**-n –** ограничить количество выводимых коммитов.

--**all –** посмотреть историю коммитов всех веток

**-stat –** сокращенная статистика. Включает в себя количество измененных фалов, добавлений и удалений.

--**pretty –** меняет формат вывода. Возможные опции:

* **oneline –** выводит каждый коммит в одну строку. Опции **short**, **full** и **fuller** делают вывод приблизительно в том же формате, но с меньшим или большим количеством информации соответственно.
* **format –** позволяет указать формат для вывода информации.

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

ca82a6d - Scott Chacon, 6 years ago : Change version number

* **--graph –** граф в формате ASCII, который показывает текущую ветку и историю слияний.
* **-**-**since** и –**until –** вывод коммитов в определенном временном промежутке.
* **-S –** показывает только те коммиты, в которых изменение файла повлекло за собой добавление или удаление этой строки.
* **“--**" – после опций можно указать двойное тире и затем путь. Если указать каталог или имя файла, то будут показаны те коммиты, в которых были изменены эти файлы.
* **--no-merges –** не отображать коммиты слияния.

-**L –** показать историю изменения функции или строки кода в вашей кодовой базе. Эта команда постарается определить границы функции, выполнит поиск по истории и покажет все изменения, которые были сделаны с функцией, в виде набора патчей в обратном порядке до момента создания функции.

**Git reset --soft <commitHash> -** переместить HEAD на указанный коммит. Таким образом мы фактически откатываемся к определенному коммиту. При этом все изменения остаются в индексе, и мы можем их заново закоммитить или удалить из индекса.

**Git reset [--mixed] <commitHash> -** мы откатываемся к определенному коммиту, при этом изменения отмененных коммитов также пропадают из индекса. Теперь они есть только в рабочей директории.

**Git reset --hard <commitHash> -** откатываемся к определенному коммиту, при этом изменения отмененных коммитов нигде не сохраняются (index и working directory полностью соответствуют коммиту, на который мы ресетимся).

Это опасная команда. Остальные вызовы reset легко отменить. но при указании опции --hard команда принудительно перезаписывает файлы в Рабочем Каталоге. Закомиченные данные все еще можно будет вернуть, просматривая наш reflog. Но если вы не комитили данные, они потеряны навсегда.

**Git reset [--mixed] <commit> <file> -** по умолчанию удаляет файл из индекса, если вы добавили его туда по ошибке. Такой эффект создается за счет того, что мы, по сути, восстанавливаем в индексе версию файла, соответствующую определенному коммиту. По умолчанию это последний коммит. Но мы можем указать и более ранний коммит, и в индекс попадут все изменения, сделанные в файле с тех пор. При этом рабочая область не изменится! В рабочей области будет лежать версия файла из текущего коммита.

Для отката последнего коммита мы используем **HEAD~.**

**Git remote –** команда для работы с удаленными репозиториями. Без параметров выводит список настроенных удаленных репозиториев.

**origin –** гит по умолчанию дает это имя репозиторию, с которого производилось клонирование.

**-v –** посмотреть адреса для чтения и записи, привязанные к репозиторию.

**git remote add <shortname> <url> -** добавить удаленный репозиторий.

Доступ к веткам конкретного репозитория осуществляется по имени репозитория и имени ветки: **remote\_name/branch\_name.**

**git remote show <remote> -** посмотреть подробную информацию об удаленном репозитории: url удаленного репозитория, информацию об отслеживаемых ветках. Какая ветка будет отправлена на удаленный сервер при выполнении git push; каких веток с удаленного сервера у нас еще нет; какие ветки еще есть у вас, но уже удалены на сервере; какие ветки будут влиты при выполнении git pull.

**Git remote rename <old> <new> -** переименовать удаленный репозиторий.

**Git remote remove <name> -** удалить удаленный репозиторий. Все отслеживаемые ветки и настройки также будут удалены.

**git fetch [remote-name] –** позволяет забрать данные с удаленного репозитория. У вас должны появиться ссылки на все ветки из этого удалённого проекта, которые вы можете просмотреть. Вам необходимо вручную сливать изменения с сервера с вашими с помощью команды merge.

**Git pull –** получить изменения с удаленной ветки и слить их с текущей веткой (если ветка настроена на отслеживание удаленной ветки).

По умолчанию слияние выполняется с помощью merge. Чтобы использовать rebase нужно изменить настройку:

git config --global pull.rebase "true"

**git push <remote-name> <branch-name> -** отправить локальные изменения в удаленный репозиторий. Если хотите запушить текущую ветку на отслеживаемую удаленную, то можно использовать просто **git push**.

Если с момента вашего последнего обновления, кто-то запушил изменения в ветку, ваш push будет отклонен. Вам придется получить изменения с удаленного репозитория, объединить их с вашими и только после этого выполнить push.

**теги –** специальные пометки коммитов. Это статические ссылки на определенные коммиты в истории. Обычно применяются для обозначения релизов, версий, и других важных моментов в истории разработки.

Это позволяет более эффективно искать нужные коммиты, и проводитть навигацию по истории.

**Git tag [-l] [--list] –** посмотреть список имеющихся тегов.

**git tag -l "v1.8.5\*" –** поиск тега по шаблону.

Git использует 2 типа тегов: легковесные и аннотированные.

**Легковесные –** просто указатель на определенный коммит.

**Аннотированные –** хранятся в базе данных Git как полноценные объекты. Имеют контрольную сумму, содержат имя автора, его e-mail и дату создания, имеют комментарий и могут быть подписаны и проверены с помощью GNU Privacy Guard (GPG).

**git tag -a –** создать аннотированный тег.

$ git tag -a v1.4 -m "my version 1.4"

**Git show <tag> -** посмотреть данные тега вместе с коммитом.

**git tag -a v1.2 <commitHash> -** пометить тегом определенный коммит.

git tag -a v1.2 9fceb02

По умолчанию push не отправляет теги на удаленный сервер. Их нужно отправлять явно. git push origin <tagname>

Чтобы отправить все теги git push origin --tags

**git tag -d <tagname> -** удалить указанный тег с локального репозитория.

Чтобы удалить теги и с удаленного репозитория:

 git push <remote> :refs/tags/<tagname>

$ git push origin --delete <tagname>

**Алиасы –** Git позволяет настроить алиасы для команд или группы команд. Это можно сделать с помощью команды config.

$ git config --global alias.co checkout

$ git config --global alias.br branch

Теперь вместо git checkout достаточно ввести git co.

Также, обычно, добавляют команду **last**, чтобы посмотреть последний коммит следующим образом:

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

**ВЕТВЛЕНИЕ**

**Ветвление –** механизм, который позволяет создавать параллельные линии разработки, чтобы работать над разными задачами или функциональностями независимо друг от друга.

Преимущества ветвления:

* **Параллельная разработка**. Позволяет нескольким разработчикам работать над различными задачами одновременно. Каждый разработчик может создать свою собственную ветку, вносить изменения в код и затем объединить свою ветку с основной линией разработки.
* **Изоляция изменений**: Создание веток позволяет изолировать изменения, чтобы они не влияли на основную линию разработки до тех пор, пока они не будут готовы к объединению. Это помогает предотвратить конфликты и обеспечивает возможность тестирования и проверки изменений до их интеграции в основную ветку.
* **Возврат к предыдущим состояниям**: Ветвление также обеспечивает возможность легкого возврата к предыдущим состояниям кода. Если изменения в ветке не являются успешными или нужно вернуться к предыдущему состоянию, можно просто переключиться на другую ветку.
* **Упрощает код**-**ревью**

Ветвление в Git очень легковесно. Git поощряет процесс работы, при котором ветвление выполняется часто.

**Ветка в гит** – это просто указатель на один из коммитов. Каждый раз, при создании коммита указатель ветки будет передвигаться на следующий коммит автоматически. При создании ветки просто создается новый указатель для перемещения.

Указатель на текущую ветку хранится в **HEAD**.

**git branch –** посмотреть список веток. Звездочкой помечена ветка, на которой вы находитесь в настоящий момент.

**Git branch <branchname> -** создать новую ветку.

**Git baranch -d <branchnmae> -** удалить ветку.

**git branch -v –** посмотреть последний коммит на каждой из веток.

Опции **--merged** и **--no-merged –** ветки слитые (не слитые) в текущую ветку. Можно также указать имя ветки, если хотим посмотреть не текущую.

**git branch –move old new –** переименовать ветку. Чтобы удалить ветку со старым именем с сервера используем команду:

git push origin --delete bad-branch-name

**git branch -vv –** посмотреть настройки веток слежения

**git checkout <branchname> -** переключиться на указанную ветку. HEAD будет указывать на branchname, а все файлы в рабочем каталоге вернутся в состояние, на снимок которого указывает branchname.

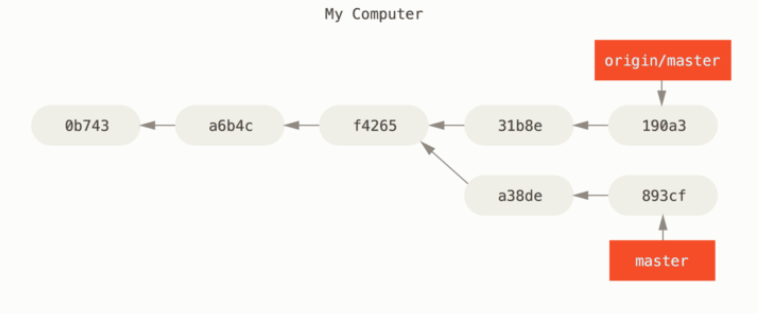
**git checkout -b <newbranchname> -** создать новую ветку и переключиться на нее.

**Git checkout <filename> -** обновляет файл в индексе версией из коммита. Так же обновляет файл и в рабочем каталоге (в отличии от git reset с именем файла).

**Ветки слежения –** это ссылки на определенное состояние удаленных веток. Git перемещает их автоматически при любой коммуникации с удаленным сервером. вид **<remote>/<branch>.**

Для синхронизации ваших изменений с удалённым сервером выполните команду **git fetch <remote>** (в нашем случае **git fetch origin**). Эта команда извлекает с сервера данные, которых у вас еще нет, и обновляет локальную бд, сдвигая указатель origin/master на новую позицию.

Ваша локальная ветка и ветка слежения — это разные ветки, поэтому они могут отличаться, если кто-то запушит туда новые коммиты.



Получение локальной ветки из удалённой ветки автоматически создаёт то, что называется «веткой слежения» (а ветка, за которой следит локальная называется «upstream branch»). Ветки слежения — это локальные ветки, которые напрямую связаны с удалённой веткой. Если, находясь на ветке слежения, выполнить git pull, то Git уже будет знать с какого сервера получать данные и какую ветку использовать для слияния.

Ели есть локальная ветка и вы хотите настроить ее на слежение за удаленной веткой, воспользуйтесь параметрами **-u** или **--set-upstream-to** для команды git branch:

$ git branch -u origin/serverfix

Branch serverfix set up to track remote branch serverfix from origin.

При пуле изменения попадают сразу в отслеживаемую ветку, а потом мержатся с локальной.

**git push origin –delete branch** – удалить ветку на удаленном сервере.

**Git merge –** объединить ветки, слить.

**git merge –squash -** Git создает новый коммит, содержащий все изменения из сливаемой ветки, но без сохранения истории коммитов из этой ветки. Вместо этого, все изменения будут включены в один коммит, который можно настроить перед созданием. Git применит все изменения этих коммитов к вашему рабочему каталогу, а потом вы можете их закомитить.

**Стратегии слияния:**

**Fastforward** – в основной ветке не было коммитов после ответвления, и Git может просто перенести указатель ветки вперед, на последний коммит.

**Recursive -** Если после ответвления в основной ветке были добавлены коммиты, то Fastforward не получится. В этом случае, Git выполняет трехстороннее слияние, используя последние коммит объединяемых веток и общего для них родительского коммита.

Git создает коммит слияния, на основе трех вышеуказанных коммитов.

**Ours –** позволяет слить изменения из любого количества веток. Однако мы просто игнорируем изменения других веток, и в коммит слияния попадают только изменения из нашей ветки.

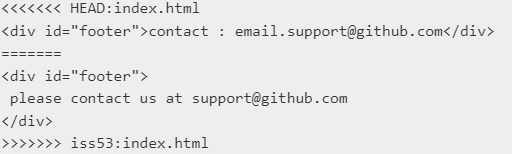
**Octopus –** слияние более двух веток. Однако она не может выполнить слияние в случае конфликтов.

**Конфликт слияния –** ситуация, когда в различных ветках, вы изменили одну и ту же часть файла по-разному. В таком случае, Git не сможет их объединить автоматически. Он остановит процесс до тех пор, пока вы не разрешите конфликт.

Можно воспользоваться **git status,** чтобы посмотреть, какие файлы не объединены.



В конфликтующие файлы Git добавляет специальные маркеры конфликтов, чтобы вы могли исправить их вручную. Чтобы разрешить конфликт, придётся выбрать один из вариантов, либо объединить содержимое по-своему.



Разрешив каждый конфликт во всех файлах, запустите git add для каждого файла, чтобы отметить конфликт как решённый. Добавление файла в индекс означает для Git, что все конфликты в нём исправлены. Выполните команду git commit для создания коммита слияния.

Можно указать гиту, какую версию файла использовать (из нашей ветки или из сливаемой), команда merge имеет два флага: -**Xours** или -**Xtheirs**

**Git cherry-pick –** берет изменения, вносимые одним коммитом, и пытается применить их в виде нового коммита в текущей ветке. Полезна в ситуации, когда нужно забрать парочку коммитов из другой ветки, а не сливать ветку целиком со всеми внесёнными в неё изменениями.

Например, если вы случайно внесли изменения в неправильную ветку, вы можете выбрать соответствующий коммит и применить его на правильную ветку с помощью `git cherry-pick`.

**Git rebase <branchname> -** еще один способ объединить изменения из двух веток. Команда берет коммиты из одной ветки, и в том же порядке применяет их к другой ветке. Это работает следующим образом:

Берется общий родительский коммит двух веток, определяется разница каждого коммита текущей ветки и сохраняется во временный файл, текущая ветка устанавливается на последний коммит ветки, поверх которой выполняется rebase, а затем по очереди применяются изменения из временных файлов.

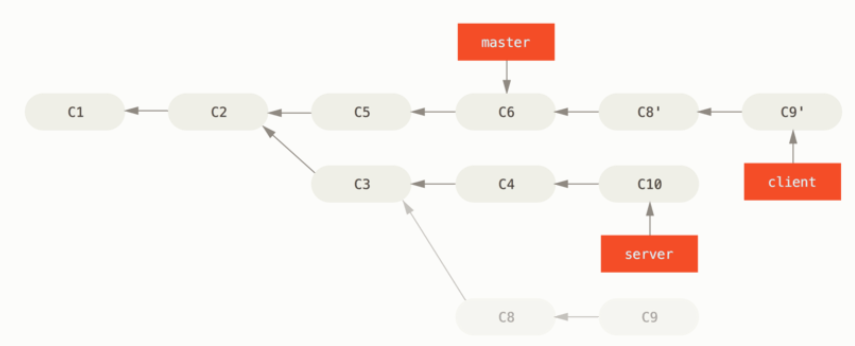
Результат будет такой же, как и в merge. Но rebase делает историю коммитов чище. История выглядит линейно, будто все операции были выполнены последовательно, даже если изначально они совершались параллельно.

При этом ветка относительно который мы делали rebase, может быть смержена с нашей веткой простым fastforward.

Часто вы будете делать rebase для уверенности, что ваши коммиты могут быть бесконфликтно слиты в удалённую ветку.

С помощью rebase мы можем перенести коммиты даже через несколько веток, с помощью опции –**onto:**

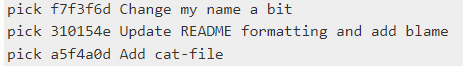
$ git rebase --onto master server client



**git rebase -i –** интерактивный rebase. C помощью него можно останавливаться после каждого коммита и изменять сообщения, добавлять файлы, менять коммиты местами, удалять, объединять несколько коммитов в один или разделять один коммит на несколько.

Выполнение этой команды отобразит в тестовом редакторе скрипт со списком коммитов, и действий, которые будут производиться с ними. Более старые коммиты расположены сверху, т.к. они должны применяться первыми.

Здесь то мы и можем поменять их порядок, или вообще удалить – просто меняем строки местами, и удаляем ненужные.



Возможные действия:

* **pick –** применить коммит без изменений.
* **edit –** остановить скрипт для изменения коммита. В этом случае вы сможете с помощью команды amend обновить сообщение коммита, либо добавить в него файлы. Затем вызовите **git rebase –continue** для продолжения скрипта.  
  С помощью edit можно разбить коммит на несколько. Выполнив **git reset HEAD^** мы отменяем коммит, изменения попадают в рабочий каталог, и можем сделать несколько новых коммитов.
* **squash –** объединить этот коммит с предыдущим. При этом Git запустит редактор, чтобы вы дали новое описание объединенному коммиту.

**git rebase –abort –** отменить rebase и вернуть репозиторий в исходное состояние.

**Не делайте!** Rebase коммитов, уже отправленных на сервер.

Когда вы что-то перемещаете, вы отменяете существующие коммиты и создаёте новые, похожие на старые, но являющиеся другими. Если вы куда-нибудь отправляете свои коммиты и другие люди забирают их себе и в дальнейшем основывают на них свою работу, а затем вы переделываете эти коммиты командой git rebase и выкладываете их снова, то ваши коллеги будут вынуждены заново выполнять слияние для своих наработок.

Rebase может помочь в ситуации, когда кто-то сделал git push --force в рабочую ветку нескольких человек. Если вы скачаете перезаписанную историю и перебазируете её поверх новых коммитов вашего коллеги, в большинстве случаев Git успешно определит, какие именно изменения были внесены вами, и применит их поверх новой ветки.

**Git stash –** перемещает ваши проиндексированные изменения в хранилище незавершенных изменений. Эти изменения можно в любое время вернуть обратно.

Это может понадобиться, когда есть изменения, которые вы не хотите пока коммитить, но вам нужно переключиться на другую ветку (например другая срочная таска), избежав при этом конфликтов и случайного добавления в коммит ненужных изменений.

**git stash list –** посмотреть список припрятанных изменений.

**Git stash apply [name] –** применить к рабочей директории припрятанные изменения. Если не указывать имя, будет применено последнее.

**git stash apply –index –** применить спрятанные изменения и восстановить индекс. По умолчанию все изменения, даже те, которые на момент вызова stash были в индексе, попадут в working tree.

**git stash drop <name> -** удалить спрятанные изменения (apply их просто применяет, но не удаляет).

**git stash pop –** применить припрятанные изменения, и тут же удалить их из хранилища.

С опцией **--include-untracked** или **-u**, Git также припрячет все неотслеживаемые файлы.

**git stash branch –** команда создаст для вас новую ветку, перейдет на коммит, на котором вы были, когда прятали свои наработки, применит на нем эти наработки, и затем, если они применились успешно, удалит эти припрятанные наработки.

**Git clean –** удалить все изменения в рабочем каталоге.

По умолчанию удаляются только неотслеживаемые файлы. Все что в .gitignore не удаляется.

**-x –** удалить также и файлы, которые есть в .gitignore.

**-n** – имитировать работу команды, чтобы посмотреть, что будет удалено.

Одной из распространённых причин для этого может быть удаление мусора, который был сгенерирован при слиянии или внешними утилитами, или удаление артефактов сборки в процессе её очистки.

**Git grep –** команда, позволяющая искать в истории коммитов или в рабочем каталоге по строке или регулярному выражению.

**git reflog -** команда в Git, которая отображает журнал ссылок (reference log) и позволяет просмотреть историю всех изменений указателей (ссылок) в вашем репозитории, включая коммиты, слияния, ветвления и переключения веток.

Когда вы выполняете операции, такие как коммиты, слияния, переключения веток или использование команды `git reset`, Git автоматически обновляет ссылки (например, указатель HEAD, ветки, теги) на новые коммиты. `git reflog` позволяет просмотреть историю всех этих изменений ссылок.

Эта команда может помочь **восстановить удаленные коммиты** (например с помощью git reset или интерактивного rebase).

Эти коммиты **все еще хранятся в бд**, просто мы **потеряли** на них **ссылки**. Тут мы можем найти хеш “Удаленного” коммита, и вернуть его с помощью **git cherry-pick** или **git branch.**

Таким же образом можно восстановить и удаленную ветку, восстановив коммит, на который указывала ветка.

**Git blame –** снабдить файл аннотацией, и увидеть, когда каждая строка кода была изменена последний раз и кем.

**-L –** диапазон строк, которые нужно вывести

$ git blame -L 69,82 Makefile

**git revert** —  создаёт новый коммит, который вносит изменения, противоположные указанному коммиту, по существу отменяя его.

**.geetkeep –** это не фича гита. Просто люди , которые хотят отслеживать пустые каталоги в Git'е, создали традицию помещать в эти каталоги файлы с именем .gitkeep. Файл может называться как угодно, гит не предает этому значения.

**Git bisect** позволяет помечать коммиты как **bad**, указывая что коммит сломан, **git bisect good –** указывая что коммит работает. А git бинарным поиском будет переключаться на коммиты, чтобы вы могли пометить, сломаны они или нет. Таким образом выполняется бинарный поиск поломанного коммита. После того, как дефектный коммит найден выполняем **git bisect reset**, чтобы вернуть HEAD туда, где он был до начала поиска.

**Модель ветвления –** способ организации работы с репозиторием, стратегия организации веток в репозитории. Какие ветки мы будем создавать, и как мы будем их объединять.

**Central Workflow**

Репозиторий содержит только одну главную ветку master. Все изменения комитятся в нее. Репозиторий может быть локальным, без удаленных копий или храниться удаленно, где он может быть клонирован или запушен. Подходит для одиночных проектов, либо небольших комманд которые перешли от svn к git.

**Developer Branch Workflow**

У каждого разработчика есть своя личная ветка или несколько, в которые он пушит. Все изменения, опубликованные в удаленном репозитории будут в этой ветке. Вся работа может быть выполнена на разных ветках, но потом должна будет слита(merged) в одну главную ветвь.

Больше подойдет для небольшого проекта с ограниченным количеством требований и небольшим количеством разработчиков, которым нужно чтобы их изменения в проекте были просмотрены до слияния с веткой master.

**Feature Branch Workflow**

В своей простейшей форме репозиторий мог бы иметь основную ветку со стабильным, доступным кодом и другими ветками для разных фич (или багов, или улучшений), которые можно бы было интегрировать в главную ветку.

Этот подход больше подходит командам, которые используют какой-то метод по управлению проектами(например Agile). Давайте скажем, что ваш проект находится в продолжительной разработке и вам нужно добавить набор фич до следующего релиза. Эти фичи назначены разным разработчикам, которые создают отдельную ветку для каждой опубликованной фичи до того, как она будет слита с dev для тестирования. Когда вы готовы к релизу, dev сливается с master.

**Issue Branch Workflow**

Очень похожа на предыдущую модель с одним лишь различием. Ветки создаются из заданий в проектном трекере. Ветки могут иметь одинаковые названия id заданий. И здесь только одна ветка на задание и одно задание на ветку.

больше подходит тем проектам, в которых фичи готовятся не одним разработчиком. Например, если бы вы работали над самим интерфейсом, а другой разработчик бы работал над его другим аспектом.

**Gitflow Workflow** (Рабочий процесс Gitflow) - это более сложная модель, 1. Основные ветки:

- `**main**` (или `**master**`): это ветка, которая содержит стабильные версии вашего проекта. На этой ветке находятся только релизные версии, которые считаются готовыми для продакшена.

- `**develop**`: это ветка, на которой происходит активная разработка новых функциональностей. Она содержит последние изменения и является основной веткой для интеграции всех фич.

2. Вспомогательные ветки:

- `**feature**/`: ветки, создаваемые для разработки новых функциональностей. Каждая фича имеет свою собственную ветку, которая отходит от ветки `develop`. После завершения работы над фичей, они могут быть объединены обратно в ветку `develop`.

- `**release**/`: ветки, создаваемые для подготовки к выпуску новой версии проекта. Здесь проводится финальное тестирование, исправление ошибок и подготовка релиза. После завершения работы над релизом, он объединяется с ветками `main` и `develop`, и метка с версией создается для стабильного релиза.

- `**hotfix**/`: ветки, создаваемые для исправления критических ошибок в текущей стабильной версии проекта. Они отходят от ветки `main` и после завершения работы над исправлением, изменения объединяются с ветками `main` и `develop`.

3**. Работа с ветками:**

- Новая функциональность разрабатывается в отдельной ветке `feature/`, которая отходит от `develop`.

- После завершения работы над фичей, создается запрос на слияние (pull request) для включения изменений в ветку `develop`. Код рассматривается и проверяется другими разработчиками.

- Когда разработка для новой версии завершена, создается ветка `release/`, где проводится финальное тестирование и подготовка к выпуску.

- После релиза, изменения сливаются в ветки `main` и `develop`, и создается метка с версией для стабильной релизной версии.

- Если в текущей стабильной версии обнаруживается критическая ошибка, создается ветка `hotfix/` для ее исправления. После завершения работы над исправлением, изменения сливаются с ветками `main` и `develop`.