## Система контроля версий Git

[**Система контроля версий Git**](#_gjdgxs) **1**

[Основные понятия СКВ](#_30j0zll) 2

[Зачем контролировать версии?](#_1fob9te) 3

[Как происходит контроль версий в ручном режиме?](#_3znysh7) 4

[Какие есть СКВ?](#_2et92p0) 4

[Какие основные понятия СКВ?](#_tyjcwt) 5

[Какой простейший алгоритм использования СКВ?](#_3dy6vkm) 6

[Как СКВ меняет рабочий процесс?](#_4d34og8) 6

[Основы работы с Git](#_2s8eyo1) 7

[Как установить и настроить git?](#_17dp8vu) 7

[Как Git хранит состояния?](#_3rdcrjn) 8

[Каков жизненный цикл файла в git?](#_26in1rg) 8

[Как начать работу с репозиторием?](#_lnxbz9) 9

[Как посмотреть изменения?](#_35nkun2) 10

[Как посмотреть историю коммитов?](#_1ksv4uv) 11

[Как перейти к другому состоянию?](#_44sinio) 12

[Работа с ветвлением](#_2jxsxqh) 13

[Зачем нужны ветки?](#_z337ya) 13

[Как создать новую ветку?](#_3j2qqm3) 13

[Как переключиться на другую ветку?](#_1y810tw) 15

[Как объединять ветки?](#_4i7ojhp) 16

[Как разрешать конфликты слияния?](#_2xcytpi) 17

[Работа с удаленными репозиториями](#_1ci93xb) 18

[Зачем нужны удаленные репозитории?](#_3whwml4) 18

[Что такое DVCS?](#_3whwml4) 19

[Как связать текущий репозиторий с другим?](#_3whwml4) 19

[Как склонировать репозиторий?](#_3whwml4) 20

[Как получить изменения из апстрима?](#_3whwml4) 21

[Как отправить изменения в апстрим?](#_3whwml4) 21

[Работа с GitHub](#_3as4poj) 22

[Что это такое и зачем это нужно?](#_1pxezwc) 22

[Как создать репозиторий?](#_49x2ik5) 22

[Как склонировать репозиторий?](#_49x2ik5) 23

[Что такое форк?](#_49x2ik5) 24

[Что такое пулл реквест?](#_49x2ik5) 25

[Использование GitFlow](#_2p2csry) 25

[Для чего это нужно?](#_147n2zr) 25

[Какие типы веток используются?](#_3o7alnk) 26

[Как происходит рабочий процесс?](#_3o7alnk) 27

[Как работать с веткой develop?](#_3o7alnk) 27

[Как создавать фичи?](#_3o7alnk) 28

[Как происходит работа с релизами?](#_3o7alnk) 28

[Для чего нужны ветки hotfix?](#_2grqrue) 29

[-Использование СКВ в IDE на примере PyCharm](#_4f1mdlm) 29

[Как и зачем?](#_2u6wntf) 29

[Как создать репозиторий в проекте?](#_19c6y18) 30

[Как создать проект из репозитория?](#_19c6y18) 30

[Как смотреть историю и переходить по ней?](#_19c6y18) 30

[Как делать коммит?](#_19c6y18) 30

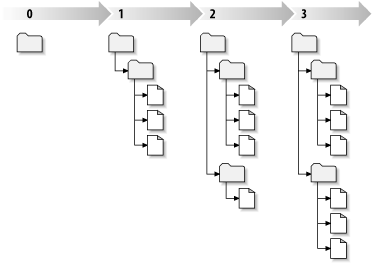
[Как смотреть изменения?](#_19c6y18) 30

[Как работать с удаленным репозиторием?](#_19c6y18) 30

[Литература](#_3tbugp1) 30

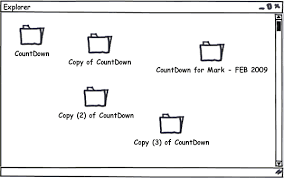
### Основные понятия СКВ

#### Зачем контролировать версии?



1. Вы разрабатываете большой и длительный проект.
2. Большинство работы происходит в текстовых файлах.
3. Вы часто вносите изменения и, бывает, откатываетесь назад.
4. Вам нужно запоминать некоторые состояния (например, гарантированно стабильно работающие, или окончательные).
5. Несколько человек работает одновременно над одним проектом.
6. Вы хотите проводить ревизию изменений другого участника.
7. Вы или другие работаете распределенно и вам нужно облачное хранилище
8. Как Timeshift в Linux.
9. Вы хотите отслеживать, кто вносит какие изменения.
10. Могут происходить конфликты версий

#### Как происходит контроль версий в ручном режиме?



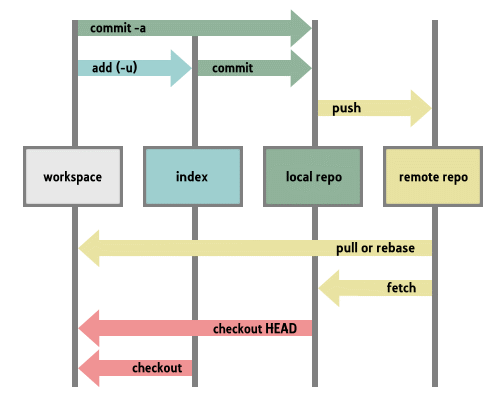
1. История прошлых версий файла/файлов.
2. В простых случаях - рабочий вариант.
3. Быстро становится неуправляемой.
4. Если работает несколько человек - адище.
5. Конфликты версий приходится разрешать в ручном режиме.
6. Занимает много места.
7. Нет индексации, статистики, blame, синхронизации и прочих плюшек.

#### Какие есть СКВ?



1. Subversion, Mercurial, Team Foundation Server.
2. Git - сегодня самая популярная.
3. Была создана Линусом Торвальдсом, когда его достали человеки.
4. Создана в первую очередь для программистов.

#### Какие основные понятия СКВ?



1. Рабочая директория - папка, в которой хранятся все файлы проекта и в которой работает СКВ.
2. **Отслеживание файла** - файлы в рабочей директории могут быть добавлены под СКВ или нет. Это сделано для того, чтобы не отслеживать часто меняющиеся, но неважные для рабочего процесса файлы, например: объектные файлы, настройки сборщика и IDE, временные файлы, кэш и так далее.
3. **Состояние** - снимок рабочей директории в определенный момент времени.
4. **Репозиторий** - служебное хранилище СКВ, в котором хранится информация о всех зафиксированных состояниях рабочей директории. Обычно является скрытой подпапкой в рабочей директории, но не входит в нее.
5. **Коммит** - фиксация изменений в новое состояние.
6. **История** - упорядоченная последовательность коммитов конкретного проекта начиная от исходного.
7. **Удаленный репозиторий** - другой репозиторий, часто расположенный на другой машине, с которым можно наладить взаимодействие (синхронизацию). Используется для организации распределенной работы над проектом.
8. **Клонирование** - создание локальной копии удаленного репозитория. Может происходить по протоколу ssh, HTTPS или просто копированием архива.
9. **Апстрим** - обычное название репозитория, служащего исходным для текущего. Обычно обозначается origin.

#### Какой простейший алгоритм использования СКВ?

cd project/

git init

git add main.py

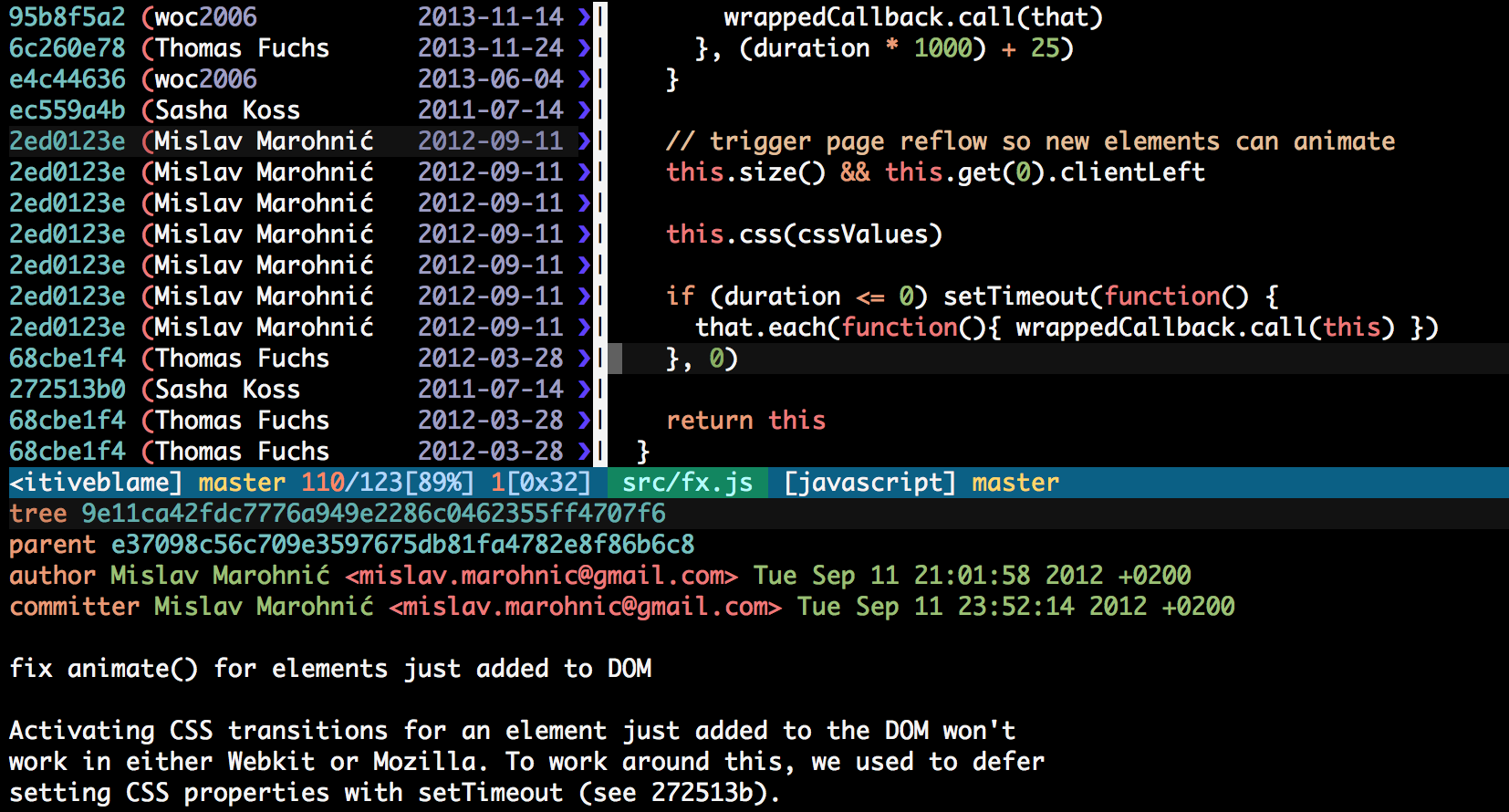
git commit -m “Initial commit”

. . .

git commit -m “New feature added”

1. Установить и настроить клиент git
2. Создать репозиторий в папке с проектом
3. Добавить необходимые файлы под СКВ
4. Сделать первоначальный коммит
5. Сделать логически завершенный участок работы
6. Сделать новый коммит с поясняющим сообщением
7. Повторить сколько нужно
8. Если что-то сломалось, вы всегда сможете вернуться на любое состояние.

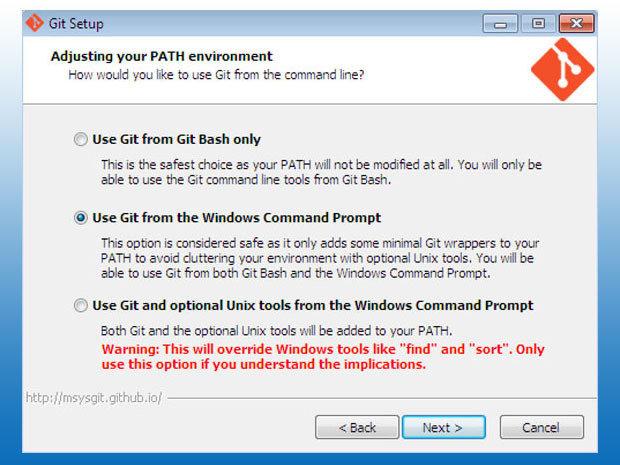
#### Как СКВ меняет рабочий процесс?



1. Ничего не удаляется. Информация не потеряется.
2. Нужно решить, как часто делать коммиты.
3. Позволяют создавать разные варианты одного документа, т. н. ветки, с общей историей изменений до точки ветвления и с разными — после неё.
4. Дают возможность узнать, кто и когда добавил или изменил конкретный набор строк в файле.
5. Ведут журнал изменений, в который пользователи могут записывать пояснения о том, что и почему они изменили в данной версии.
6. Контролируют права доступа пользователей, разрешая или запрещая чтение или изменение данных, в зависимости от того, кто запрашивает это действие.

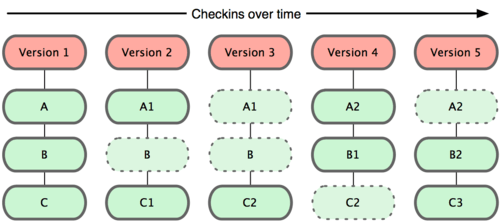
### Основы работы с Git

#### Как установить и настроить git?



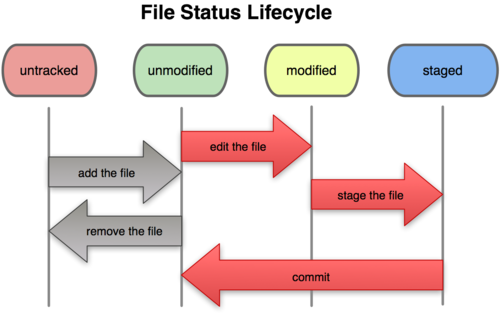
1. Нужно установить клиент git. Обычно это свободные и бесплатные программы.
2. Существуют консольные и графические клиенты.
3. Существуют под все стандартные платформы.
4. В Linux консольный клиент - чаще всего установлен в дистрибутиве по умолчанию.
5. Самая распространенная программа под Windows - git-scm.
6. После установки нужно произвести первоначальную настройку - ввести имя пользователя и e-mail. Это требуется только для идентификации пользователя. Это не регистрация.

#### Как Git хранит состояния?



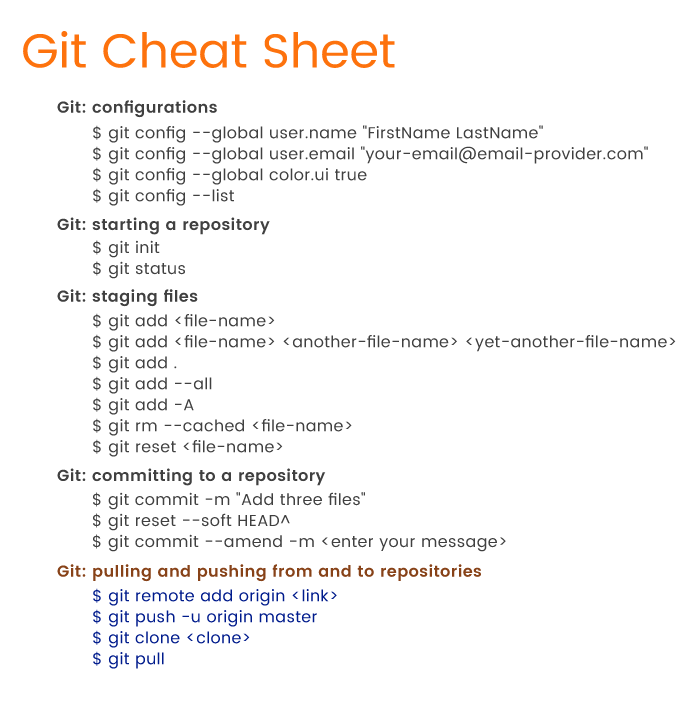
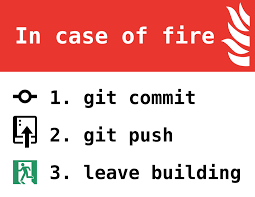
1. Связный список - у состояния есть родитель.
2. Вычисляет патч между состоянием и его родителем.
3. Работает строго построчно.

#### Каков жизненный цикл файла в git?



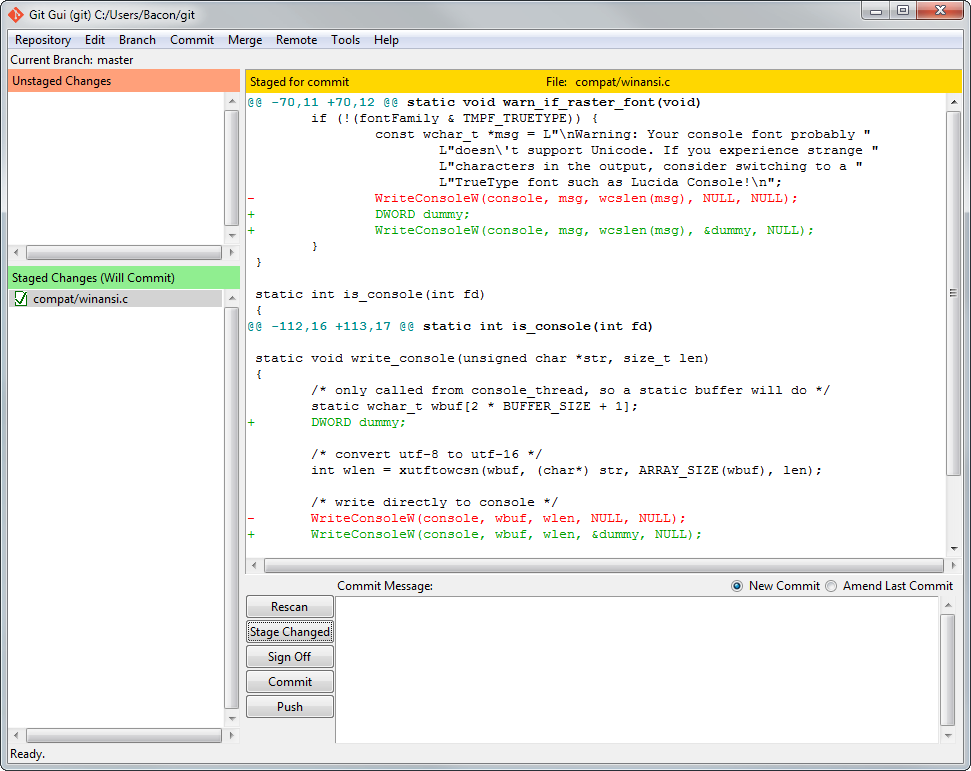
1. В Git различают три области: историю зафиксированных состояний, текущее состояние рабочей директории и индекс.
2. Рабочая директория — это то, что мы видим в окне проводника. Мы работаем с файлами в ней, можем их редактировать любыми программами как обычно.
3. Индекс — это набор файлов рабочей директории, которые отслеживаются СКВ. Когда мы делаем коммит, содержимое индекса записывается в историю состояний.
4. Если мы создадим файл в рабочей директории, он не будет отслеживаться СКВ. Это сделано для того, чтобы не вычищать постоянно мусор из служебный файлов.
5. Файл можно добавить под СКВ командой git add. в дальнейшем будем говорить только о файлах, отслеживаемых СКВ.
6. Если мы изменили файл в рабочей директории, он становится модифицированным по сравнению с последним коммитом.
7. Когда мы делаем коммит, мы записываем новое состояние и все файлы опять становятся неизмененными.
8. Во многих графических средах состояние файла отображается цветом для удобства.

#### Как начать работу с репозиторием?

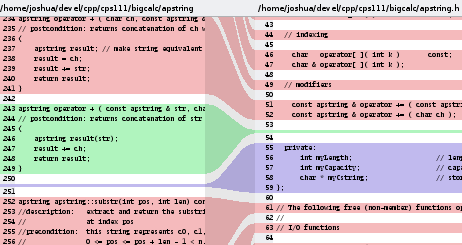


1. Нативно, гит имеет командный интерфейс. Даже если вы планируете использовать только графические клиенты или IDE с интеграцией, полезно знать основные команды и их применение.
2. Команда git init инициализирует пустой репозиторий в текущей папке.
3. Команда git add добавляет файлы в индекс.
4. Команда git commit фиксирует изменения. При этом в командной строке откроется редактор по умолчанию, чтобы вы могли записать сообщение коммита. Рекомендуется использовать форму git commit -m “Описание изменений”.
5. Можно сделать коммит и добавить все новые файлы автоматически с помощью git commit -a
6. Коммиту можно назначить метку для упрощения ориентации по истории.

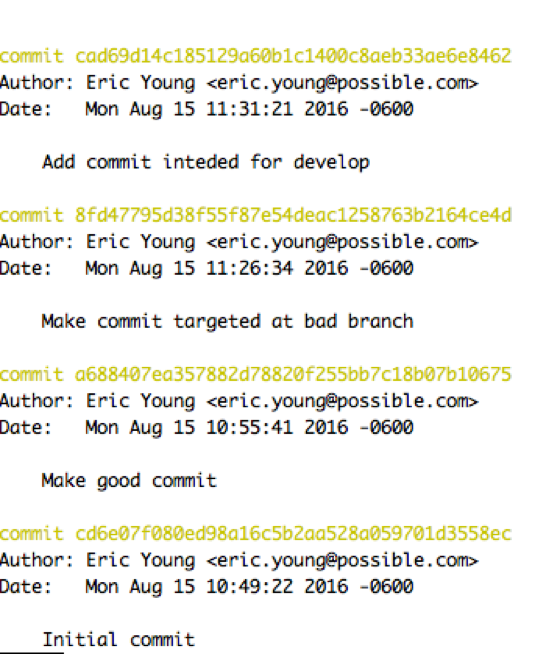
#### Как посмотреть изменения?



1. Команда git diff показывает изменения каждого файла по сравнению с содержанием индекса или, при указании, с любым коммитом.
2. Команда показывает, какой текст был удален, а какой - добавлен.
3. Очень полезная команда для просмотра изменений.
4. Эта команда работает построчно. Если строка изменена, то это будет отображаться как удаление старой версии и вставка новой.
5. Существует множество графических реализаций данного функционала

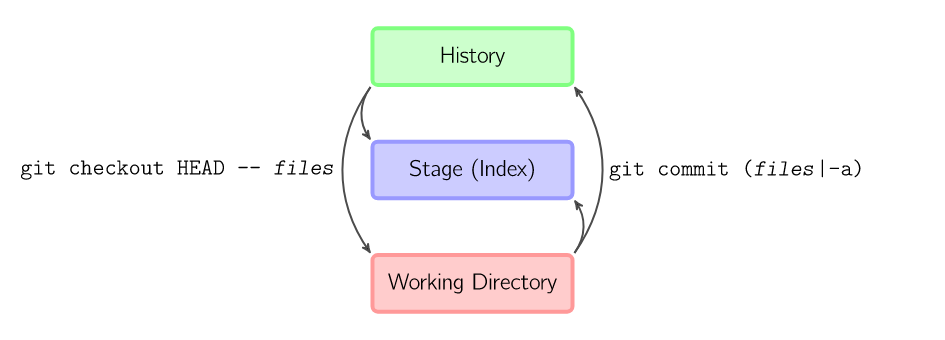


#### Как посмотреть историю коммитов?

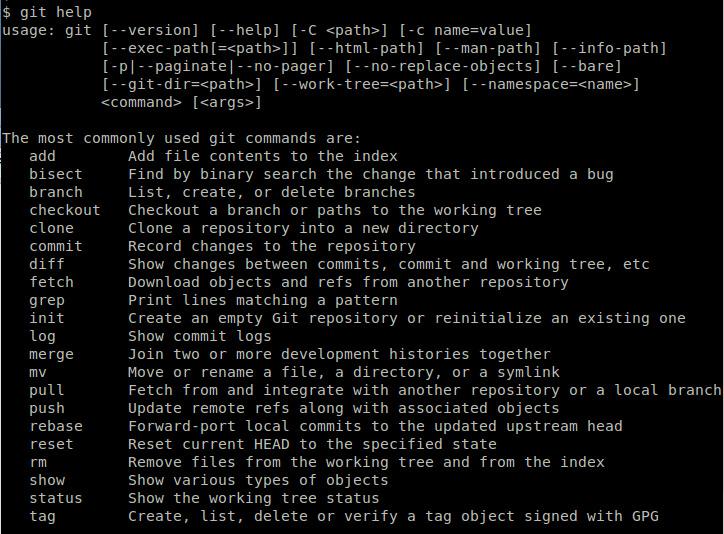


1. Команда git log выводит описание последних коммитов.
2. Обратите внимание, что коммиты идентифицируются хешем. Вы можете использовать этот хэш как ссылку на коммит. Необязательно использовать весь хеш, только уникальную часть.
3. при каждом коммите фиксируется дата, время, пользователь, совершивший коммит, сообщение, ссылка на родителя, хеш коммита.

#### Как перейти к другому состоянию?

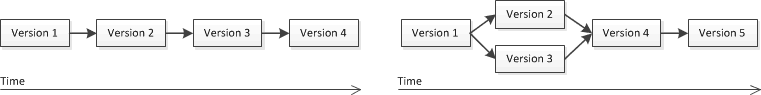


1. Команда git checkout приводит состояние рабочей директории в соответствие с выбранным коммитом.
2. Если в это время какие-то файлы были изменены, изменения потеряются.
3. Можно возвращать не всю рабочую директорию, а только определенные файлы.
4. Всегда можно воспользоваться встроенной справкой по командам.



### Работа с ветвлением

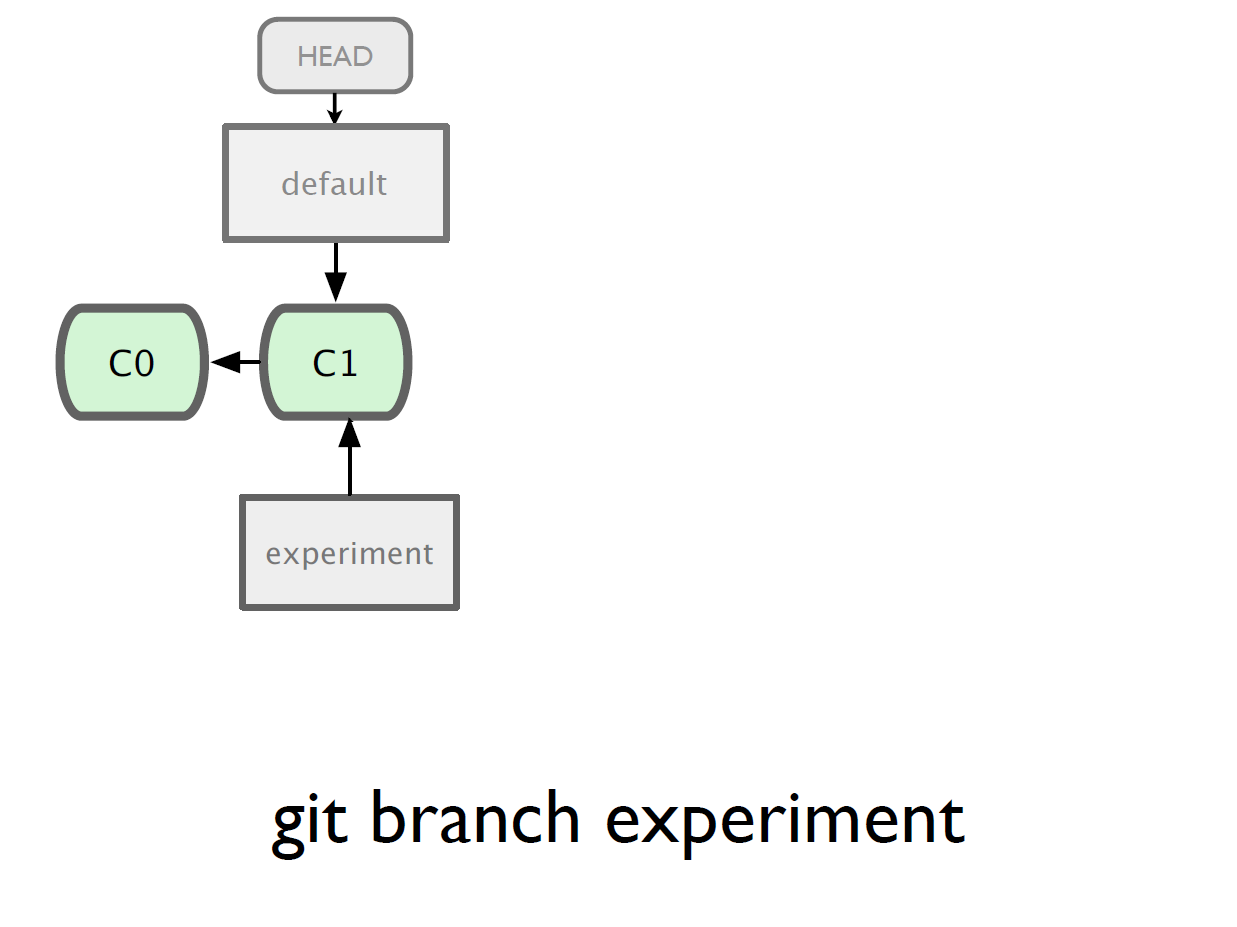
#### Зачем нужны ветки?



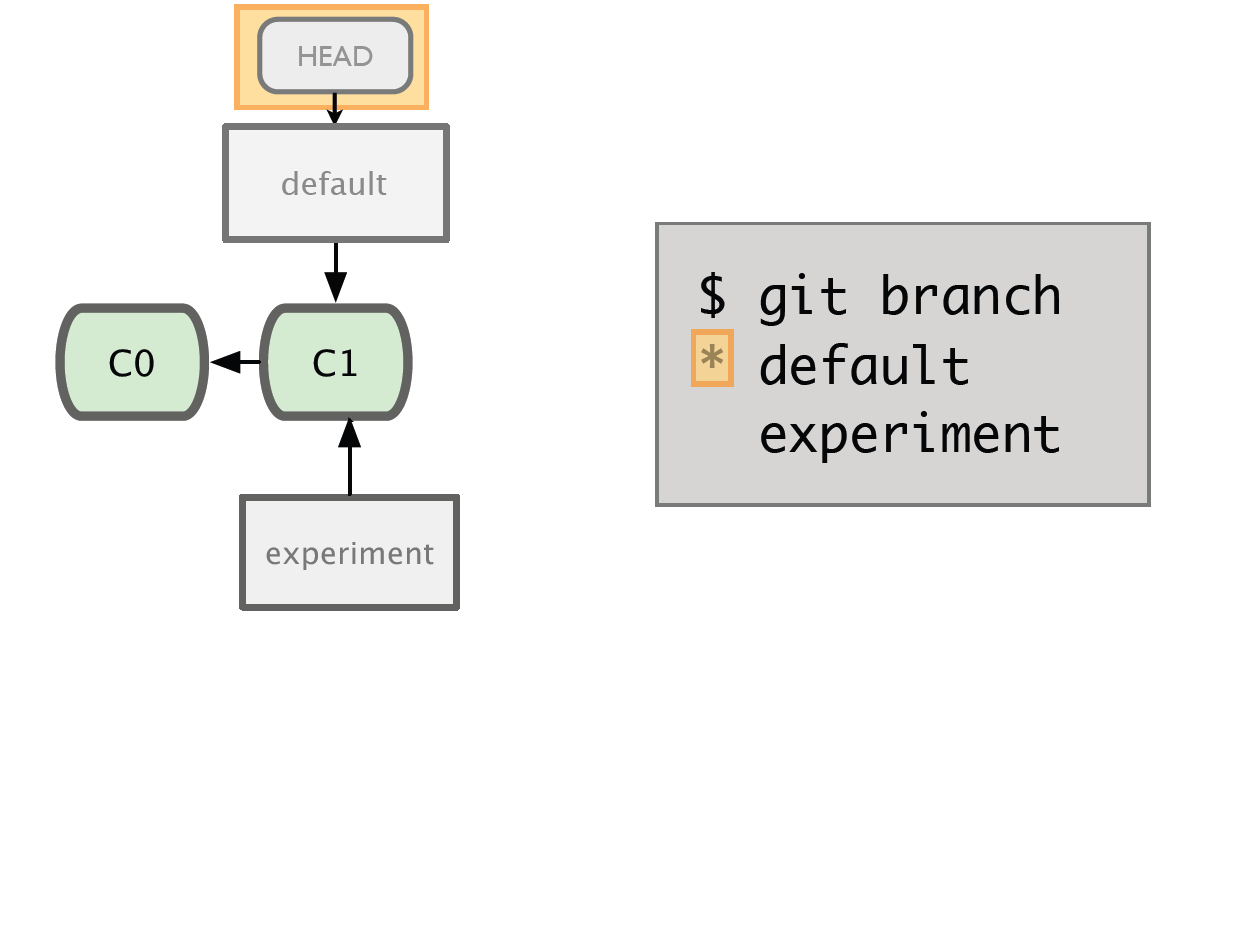
1. Ветвление происходит, когда мы произвели два разных изменения, основываясь на одном и том же состоянии.
2. Часто происходит при командной работе.
3. Кажется, что это что-то плохое.
4. Git позволяет работать с ветками комфортно и, при необходимости, объединять их.
5. Ветвление - это способ изолировать одни изменения от других. Например, разработка одной фичи независимо от хода работ над другой.

#### Как создать новую ветку?

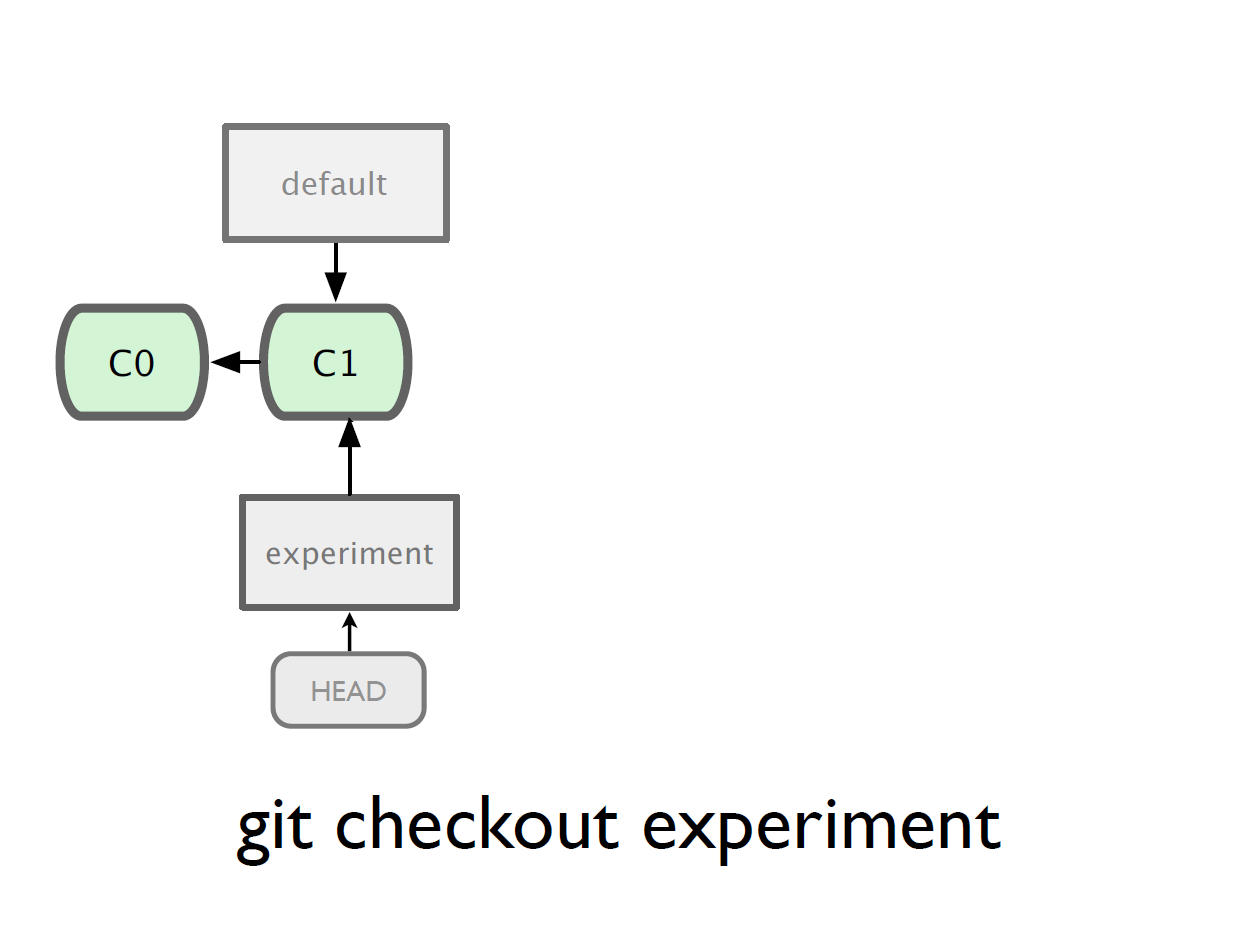
1. Ветки - это специальные указатели на коммиты, которые перемещаются с каждым новым коммитом.



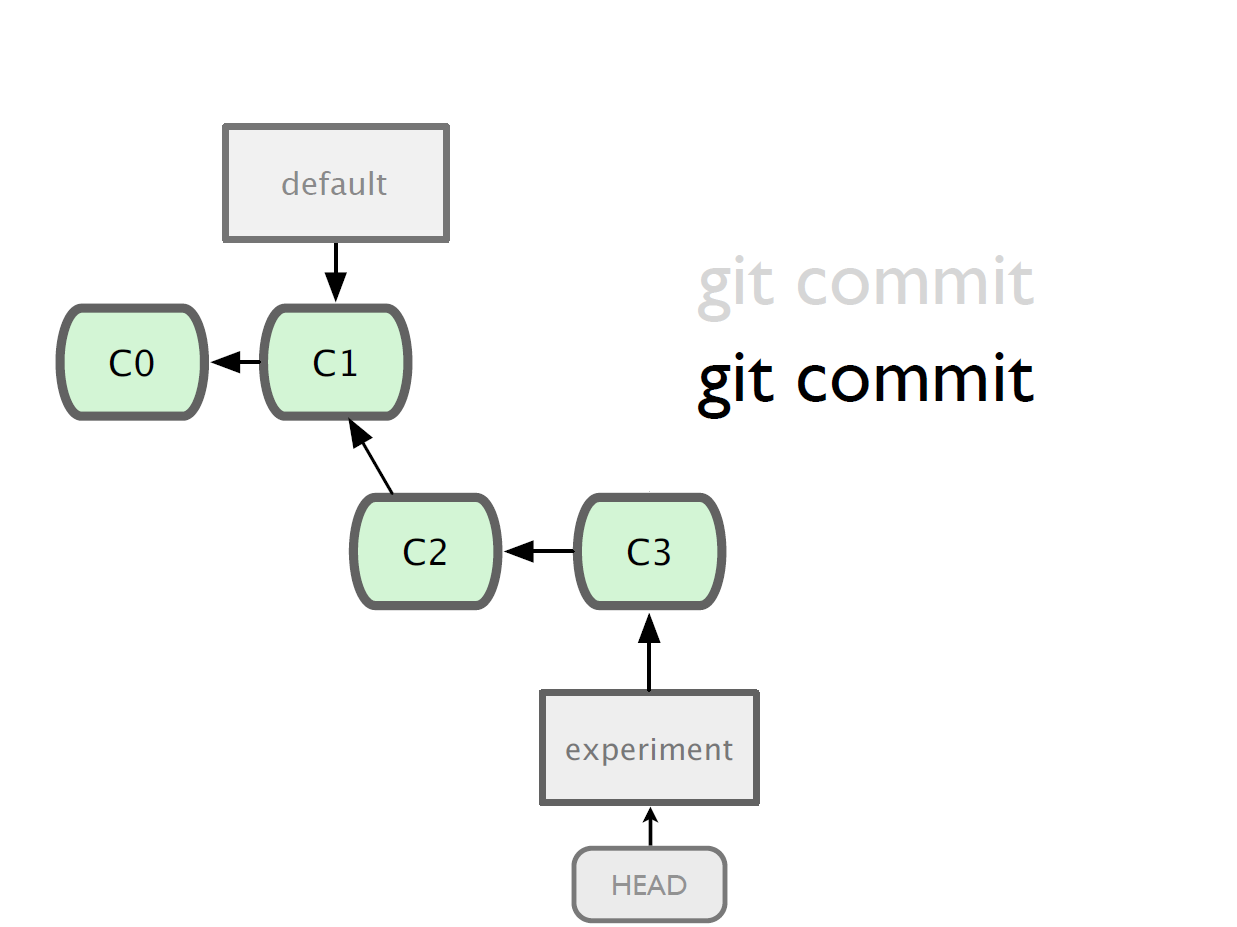
1. Создать новую ветку можно командой git branch <name>



1. Команда git branch выводит все ветки.
2. Ветка по умолчанию по соглашению называется master.
3. Git хранит специальный указатель HEAD - он показывает, на какой ветке вы сейчас находитесь

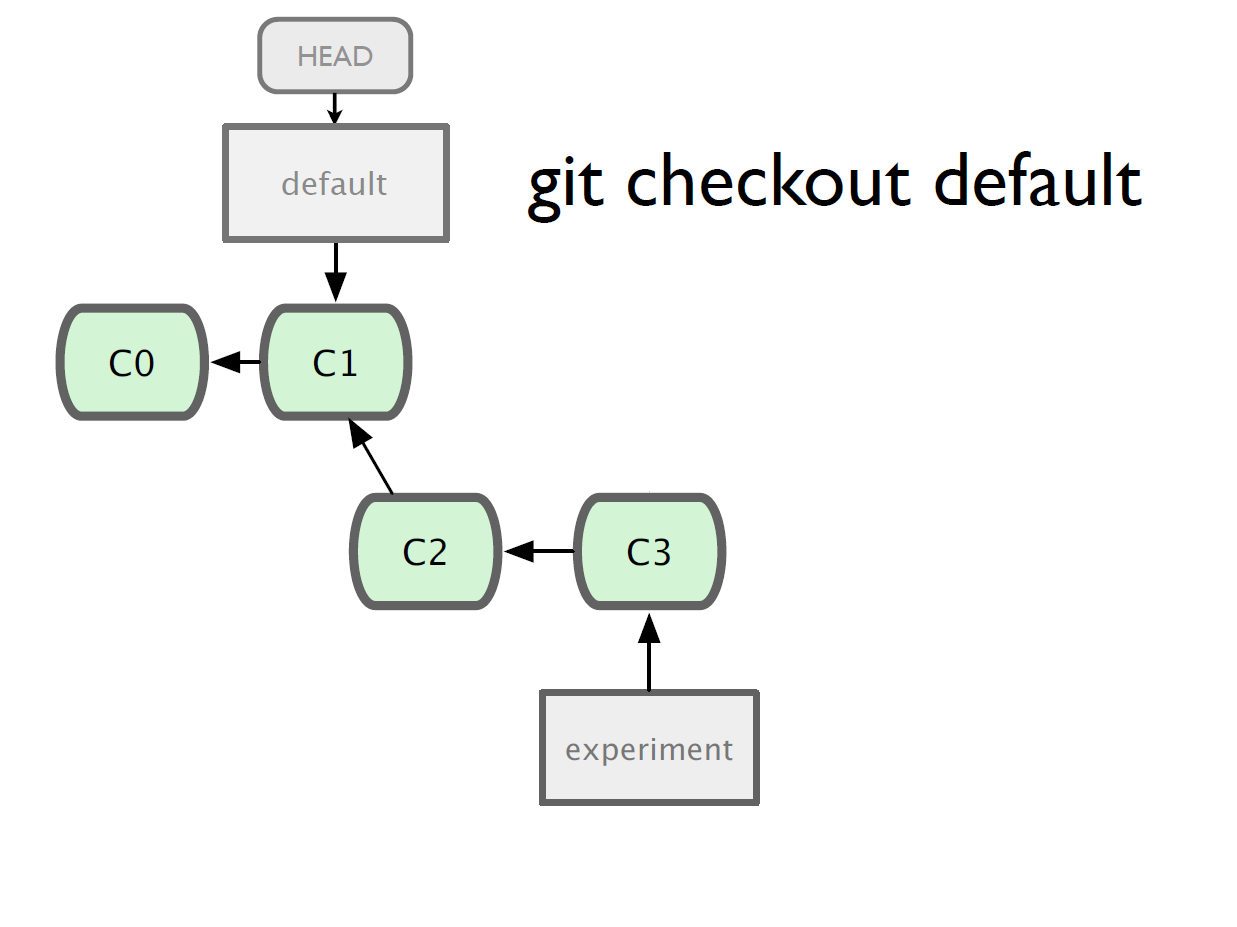


1. Чтобы перейти на существующую ветку, вам надо выполнить команду git checkout
2. Ветка, на которую указывает HEAD, движется вперёд с каждым коммитом.

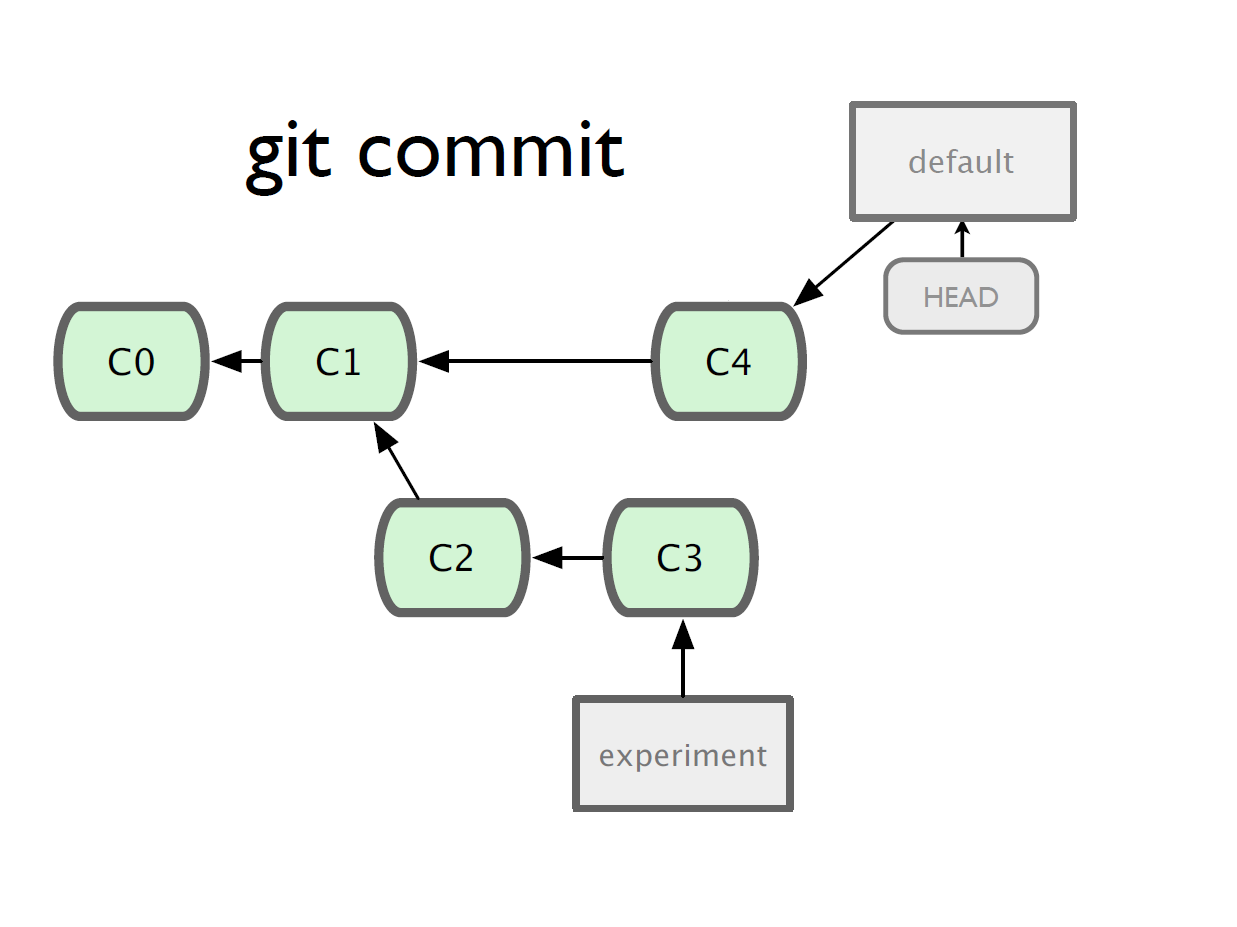


1. Теперь коммиты, которые мы будем делать, будут относиться к новой ветке.

#### Как переключиться на другую ветку?

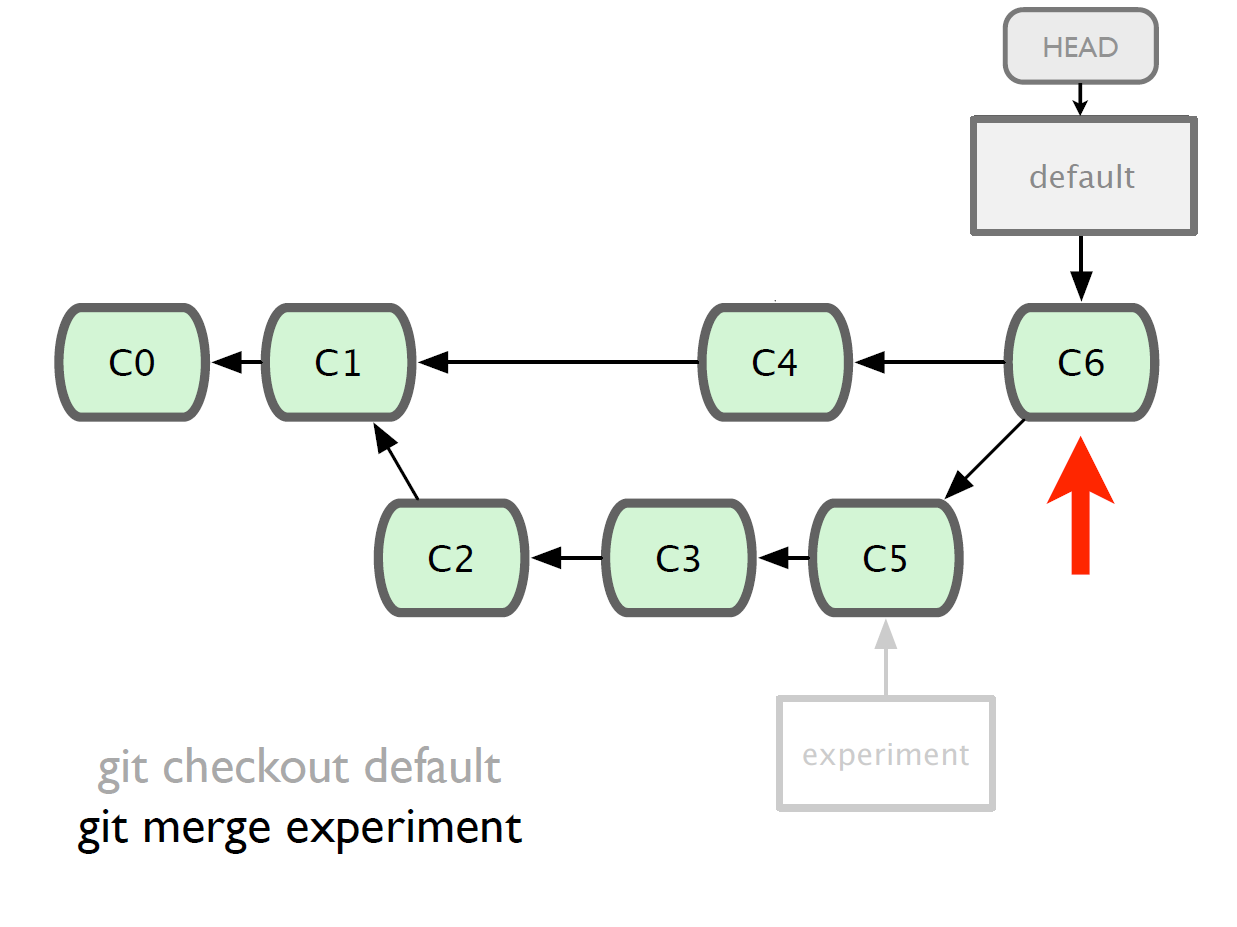


1. В любой момент можно перейти в другую ветку и начать коммитить в нее.
2. При переходе в другую ветку состояние рабочей директории приводится в соответствие с состоянием последнего коммита в этой ветке. Поэтому все изменения будут основываться на том состоянии.
3. Переход между ветками откатывает изменения, которые вы делали в другой ветке, но не удаляет их. Все фиксированные изменения сохраняются в истории веток.

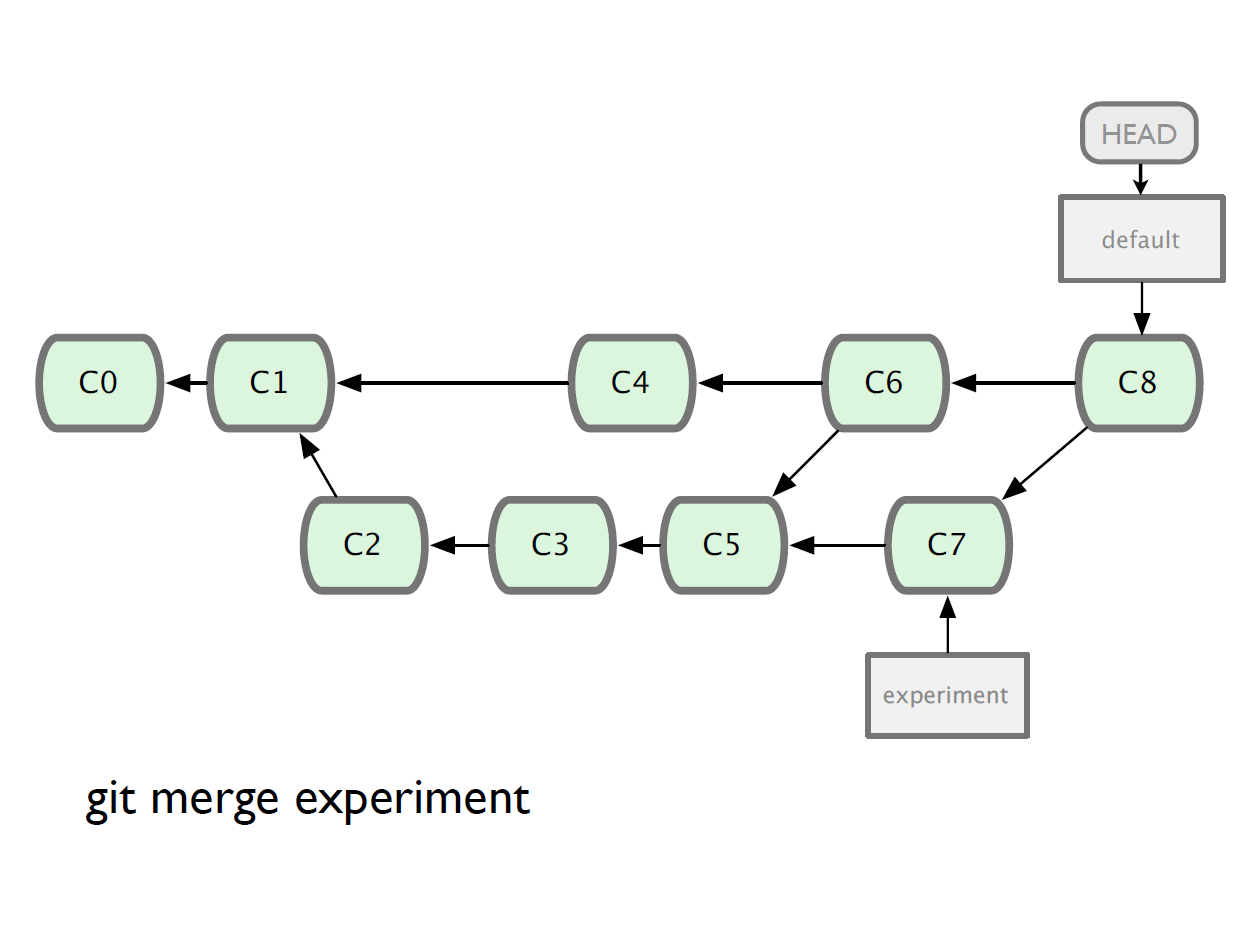


1. При использовании веток история коммитов уже не будет линейной.

#### Как объединять ветки?

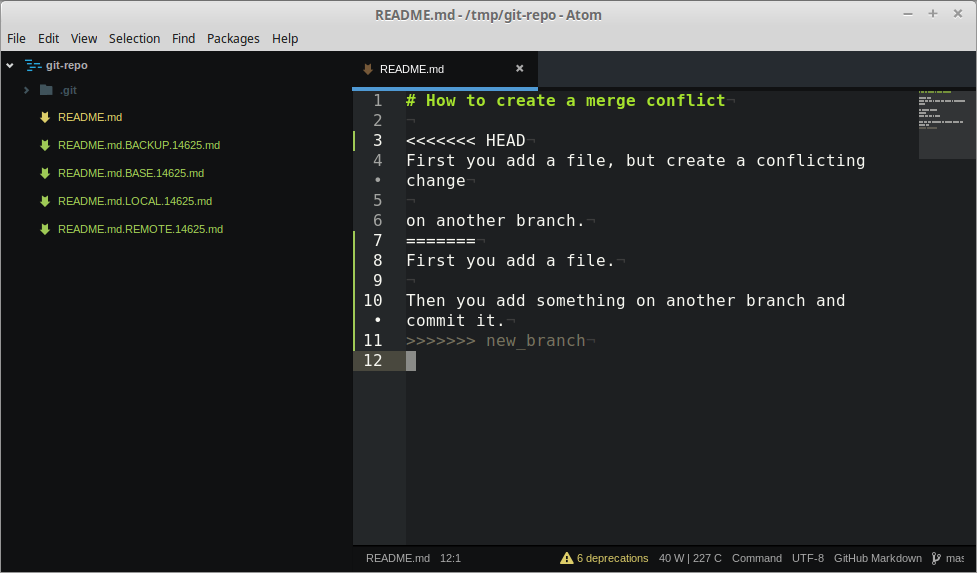


1. Две ветки можно объединить (слить). Это значит, что мы применяем к файлам изменения, которые произошли и в одной ветке и во второй.
2. Команда git merge <branch> сливает указанную ветку в текущую. Это значит, что создастся новый коммит в текущей ветке, учитывающий изменения в указанной.
3. Коммиты слияния имеют двух родителей.
4. Git автоматически определяет наилучшего общего предка для слияния веток.



1. После слияния ветка не уничтожается и над ней можно продолжать работать.

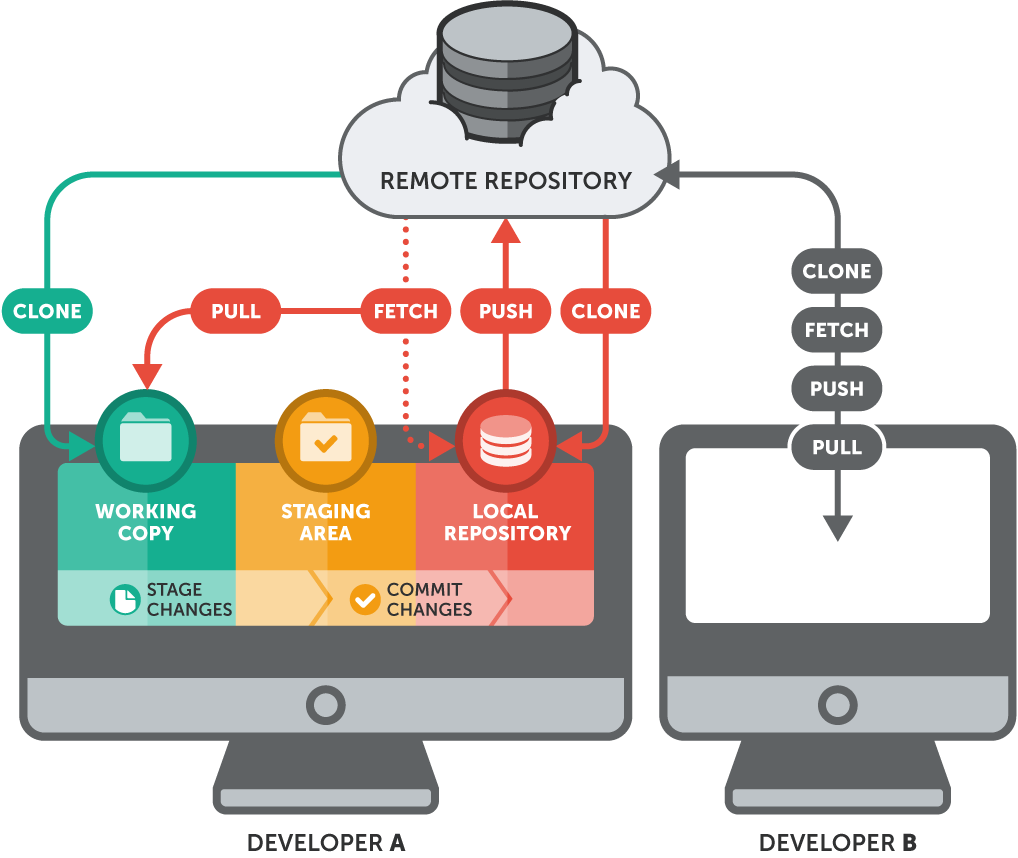
#### Как разрешать конфликты слияния?



1. **Если вы изменили одну и ту же часть файла по-разному в двух ветках**, которые собираетесь слить, Git не сможет сделать это чисто. **Такая ситуация называется конфликтом слияния**.
2. В таком случае, git приостанавливает слияние до тех пор, пока вы не разрешите конфликт.
3. Посмотреть, какие файлы не прошли слияние можно командой git status.
4. Содержимое спорных файлов в рабочей директории будет отражать оба конфликтующих варианта текста.
5. Чтобы разрешить конфликт, вы должны либо выбрать одну из этих частей, либо как-то объединить содержимое по своему усмотрению.
6. После этого необходимо выполнить добавление этих файлов к индексу и коммит для завершения слияния веток.

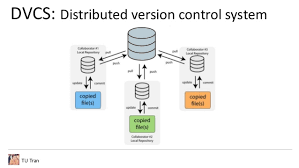
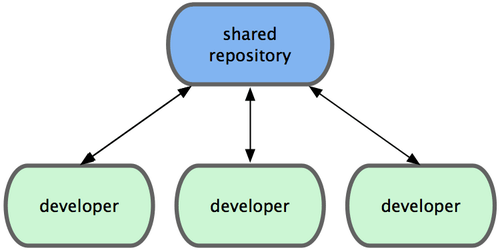
### Работа с удаленными репозиториями

#### Зачем нужны удаленные репозитории?



1. При командной разработке приходится синхронизировать работу нескольких программистов в одном месте.
2. Даже одному разработчику иногда приходится работать и вносить изменения в проект из разных мест.
3. Для этого в системах контроля версий предусмотрена возможность связать локальный репозиторий с другим - удаленным.
4. При связывании репозиториев возникает возможность отправить изменения, сделанные в локальном репозитории в удаленный и наоборот - скачать изменения из удаленного репозитория в локальный.
5. **Чтобы просмотреть, какие удалённые серверы у вас уже настроены, следует выполнить команду git remote**. Она перечисляет список имён-сокращений для всех уже указанных удалённых дескрипторов. Если вы склонировали ваш репозиторий, у вас должен отобразиться, по крайней мере, origin — это имя по умолчанию, которое Git присваивает серверу, с которого вы склонировали.

#### Что такое DVCS?



1. Распределенная модель git позволяет гибко взаимодействовать разработчикам в командном проекте.
2. В централизованных системах все разработчики являются узлами сети, более или менее одинаково работающими на центральном хабе.
3. Это означает, что если два разработчика склонируют репозиторий и каждый внесет изменения, то первый из них сможет отправить свои изменения в репозиторий без проблем. **Второй разработчик должен слить изменения**, сделанные первым разработчиком, чтобы избежать их перезаписи во время отправки на сервер.
4. Однако, в Git'е **каждый разработчик потенциально является и узлом, и хабом**. То есть каждый разработчик может как вносить код в другие репозитории, так и содержать публичный репозиторий, на основе которого работают другие разработчики, и в который они вносят свои изменения.

#### Как связать текущий репозиторий с другим?

$ git remote

origin

$ git remote add pb git://github.com/paulboone/ticgit.git

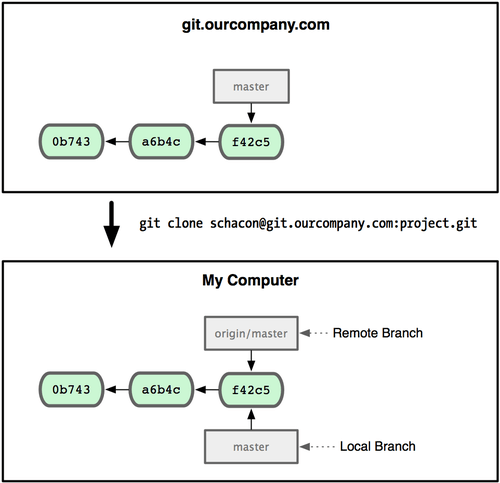
$ git remote -v

origin git://github.com/schacon/ticgit.git

pb git://github.com/paulboone/ticgit.git

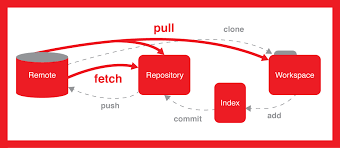
1. Чтобы добавить новый удалённый Git-репозиторий под именем-сокращением, к которому будет проще обращаться, **выполните git remote add [сокращение] [url]**
2. Теперь вы можете использовать в командной строке имя pb вместо полного URL.
3. Если вы хотите извлечь (fetch) всю информацию, которая есть в удаленном репозитории, но нет в вашем, вы можете выполнить git fetch pb.
4. Ветка master удаленного репозитория теперь доступна локально как pb/master. Вы можете слить (merge) её в одну из своих веток или перейти на эту ветку, если хотите её проверить.

#### Как склонировать репозиторий?



1. Для получения копии существующего Git-репозитория, например, проекта, в который вы хотите внести свой вклад, **необходимо использовать команду git clone**.
2. Вместо того, чтобы просто получить рабочую копию, **Git получает копию практически всех данных, которые есть на сервере**.
3. Эта команда создаёт директорию “libgit2”, инициализирует в ней поддиректорию .git, скачивает все данные для этого репозитория и создаёт (checks out) рабочую копию последней версии. **Если вы зайдёте в новую директорию libgit2, то увидите в ней файлы проекта, готовые для работы или использования**.
4. **В Git реализовано несколько транспортных протоколов**, которые вы можете использовать. В предыдущем примере использовался протокол https://, вы также можете встретить git:// или user@server:path/to/repo.git, использующий протокол передачи SSH.

#### Как получить изменения из апстрима?



Важно отметить, что команда git fetch забирает данные в ваш локальный репозиторий, но не модифицирует то, над чем вы работаете в данный момент.

Можно использовать команду git pull чтобы автоматически получить изменения из удалённой ветви и слить их со своей текущей ветвью.

Выполнение git pull, как правило, извлекает (fetch) данные с сервера, с которого вы изначально склонировали, и автоматически пытается слить (merge) их с кодом, над которым вы в данный момент работаете.

#### Как отправить изменения в апстрим?

$ git push origin master

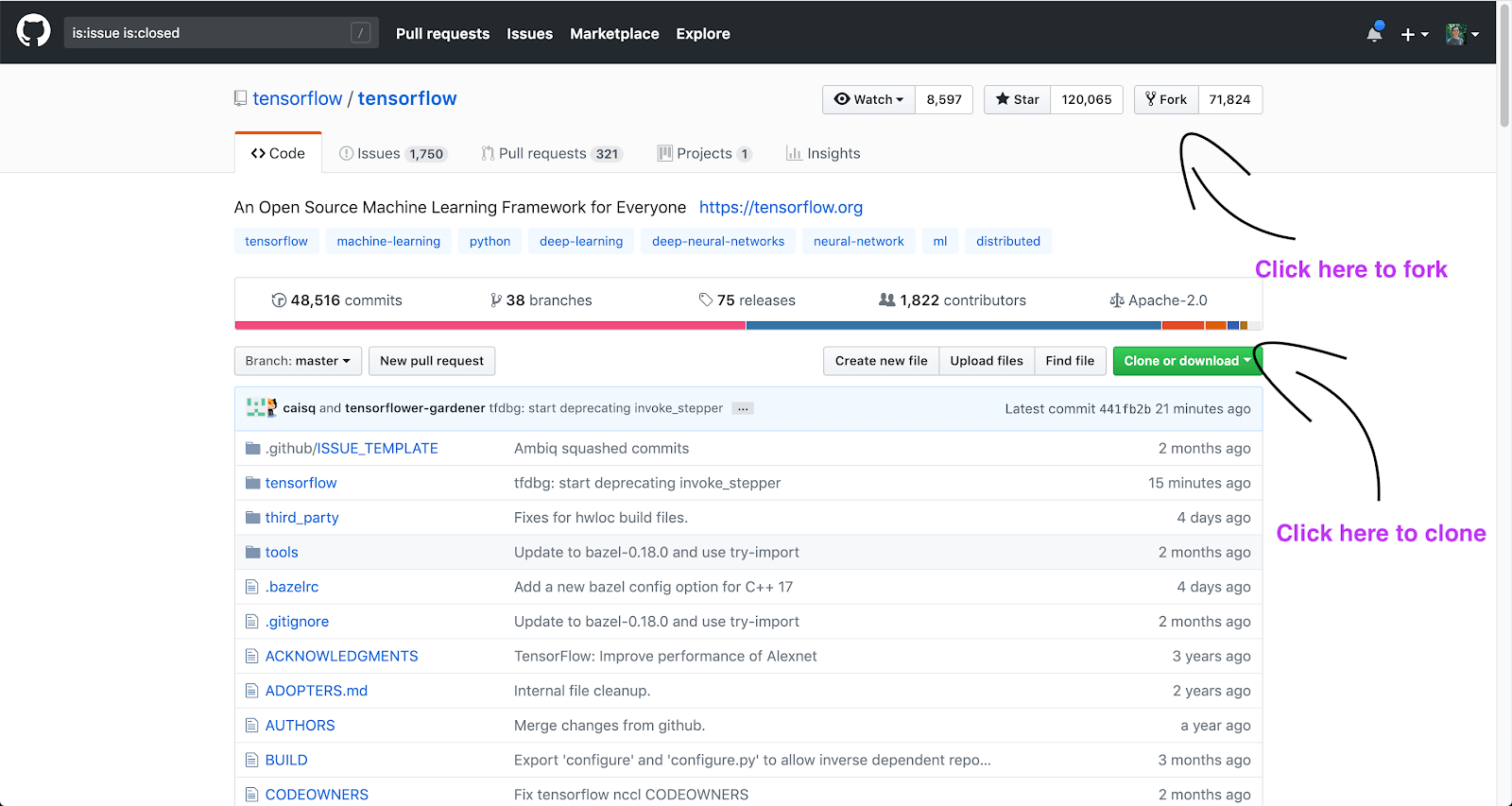
Когда вы хотите поделиться своими наработками, вам необходимо отправить (push) их в главный репозиторий.

Команда для этого действия: git push [удал. сервер] [ветка].

Если вы и кто-то ещё одновременно клонируете, затем он выполняет команду push, а затем команду push выполняете вы, то ваш push точно будет отклонён.

### Работа с GitHub

#### Что это такое и зачем это нужно?



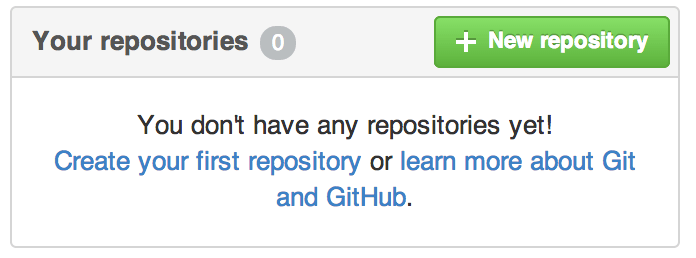
Гитхаб это крупнейшее хранилище Git репозиториев, а также центр сотрудничества для миллионов разработчиков и проектов.

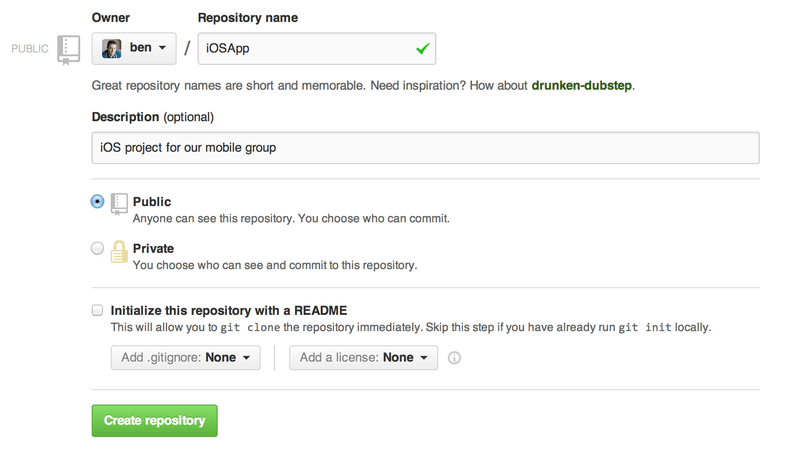
Огромный процент репозиториев хранится на Гитхабе.

Многие проекты с открытым исходным кодом используют его ради Git хостинга, баг-трекера, рецензирования кода и других вещей.

Гитхаб даже используется для отслеживания динамики популярности языков программирования.

#### Как создать репозиторий?





Первым делом нужно создать беcплатную учетную запись.

В панели управления справа нажмите кнопку “New repository”.

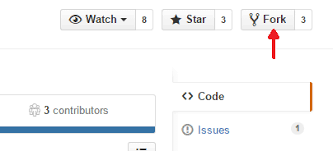
Всё, что в действительности нужно сделать, так это указать название проекта, все остальные поля опциональны.

Теперь ваш проект хостится на GitHub и вы можете предоставить ссылку на него любому желающему.

Все проекты на GitHub доступны как по HTTP https://github.com/<пользователь>/<имя\_проекта>, так по SSH.

Git производит контроль доступа на основании учётных данных пользователя, осуществляющего подключение.

#### Что такое форк?



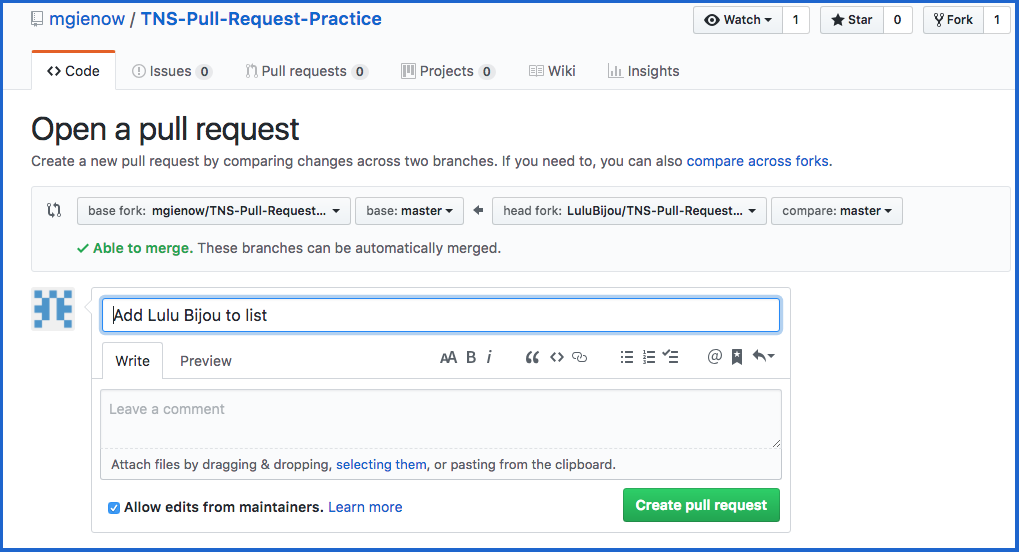
Если вы хотите вносить свой вклад в уже существующие проекты, в которых у нас нет прав на внесения изменений путем отправки (push) изменений, **вы можете создать свое собственное ответвление (“fork”) проекта**.

Это означает, что GitHub создаст вашу собственную копию проекта, данная копия будет находиться в вашем пространстве имен и вы сможете легко делать изменения путем отправки (push) изменений.

Таким образом, проекты не обеспокоены тем, чтобы пользователи, которые хотели бы выступать в роли соавторов, имели право на внесение изменений путем их отправки (push). **Люди просто могут создавать свои собственные ветвления (fork), вносить туда изменения, а затем отправлять свои внесенные изменения в оригинальный репозиторий** проекта путем создания запроса на принятие изменений (Pull Request)

Для того, чтобы создать ответвление проекта (fork), зайдите на страницу проекта и нажмите кнопку “Cоздать ответвление” (“Fork”), которая расположена в правом верхнем углу.

#### Что такое пулл реквест?



При работе над форком чужого репозитория возникает необходимость отправить свои изменения в апстрим - исходный репозиторий, с которого вы сделали форк.

Напрямую писать в репозиторий может только его автор или члены команды разработчика проекта, то есть его владельцы.

Это сделано для разграничения прав доступа в репозитории.

Поэтому, вы не сможете напрямую выполнить push в чужой репозиторий.

Вместо этого вы отправляете pull request - запрос на внесение изменений.

Этот запрос становится виден автору оригинального репозитория. Он может просмотреть вашу работу и затем либо принять его (то есть, с его стороны выполняется pull), либо отклонить.

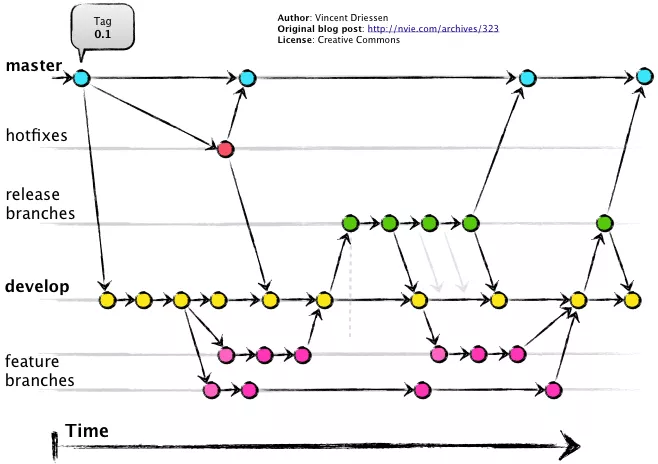
Пулл реквесты на гитхабе оформляются как публичные посты, где все участники могут оставлять свои комментарии. Таким образом организуется публичная командная работа.

### Использование GitFlow

#### Для чего это нужно?

1. Методология работы с git
2. Стандартизировать рабочий процесс с использованием веток
3. Вводит некоторые полезные и распространенные понятия
4. Регламентирует правила слияния веток
5. Сводит к минимуму риск выложить нестабильный код в мастер
6. Хорошо использовать с непрерывной интеграцией
7. Есть инструментальные решения.

#### Какие типы веток используются?



У нас есть две основные ветки: master и develop.

В ветке master содержится ровно тот же код, что и в рабочей (читай, продакт) версии проекта. А вся работа делается в ветке develop.

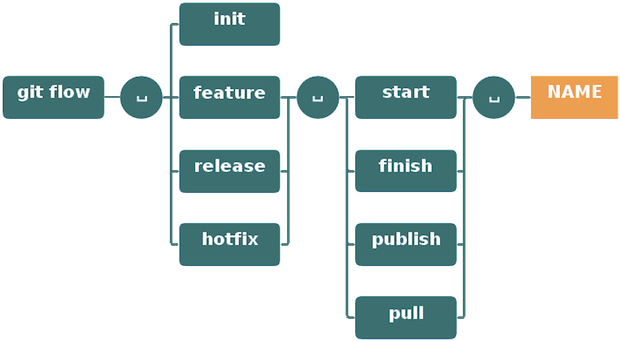
Во время работы на основе develop создаются так называемые feature-ветки. Их может быть неограниченное количество.

Далее, у нас есть ветка release, которая используется для подготовки к новому релизу проекта.

Наконец, есть ветка hotfix, которая служит для срочного исправления багов, найденных, например, на продакте.

Основа работы с гитфлоу - хранить в мастере только релизный код. Там не допускается рабочий процесс, потенциально приводящий к ошибкам.

#### Как происходит рабочий процесс?



После создания репозитория создается ветка develop. Она является основной рабочей.

Начинается работа на ветке develop.

Возникает необходимость опробовать новую штуку – создается feature-ветка и делаются коммиты

Закончив работу на feature-ветке, вы сливаете ее с develop.

Если вы довольны текущей версией, но хотите продолжить работу, создается ветка release, куда перемещается текущая версия. Правка багов будет происходить на этой же ветке.

Когда с веткой release покончено, время слить ее в master и продолжить работу с develop.

#### Как работать с веткой develop?

Ветвь master создаётся при инициализации репозитория, что должно быть знакомо каждому пользователю Git. Параллельно ей также мы создаём ветку для разработки под названием develop.

Мы считаем ветку origin/master главной. То есть, исходный код в ней должен находиться в состоянии production-ready в любой произвольный момент времени.

Ветвь origin/develop мы считаем главной ветвью для разработки.

Эту ветку также можно назвать «интеграционной».

Когда исходный код в ветви develop готов к релизу, все изменения должны быть влиты в ветвь master и помечены тегом с номером релиза.

#### Как создавать фичи?

$ git checkout -b myfeature develop

Switched to a new branch "myfeature"

. . .

$ git checkout develop

Switched to branch 'develop'

$ git merge --no-ff myfeature

Updating ea1b82a..05e9557

(Отчёт об изменениях)

$ git branch -d myfeature

Deleted branch myfeature (was 05e9557).

$ git push origin develop

Ветви функциональностей (feature branches), также называемые иногда тематическими ветвями (topic branches), используются для разработки новых функций, которые должны появиться в текущем или будущем релизах.

Смысл существования ветви функциональности (feature branch) состоит в том, что она живёт так долго, сколько продолжается разработка данной функциональности (фичи).

Когда работа в ветви завершена, последняя вливается обратно в главную ветвь разработки или же удаляется (в случае неудачного эксперимента).

#### Как происходит работа с релизами?

$ git checkout -b release-1.2 develop

Switched to a new branch "release-1.2"

$ ./bump-version.sh 1.2

Files modified successfully, version bumped to 1.2.

$ git commit -a -m "Bumped version number to 1.2"

[release-1.2 74d9424] Bumped version number to 1.2

1 files changed, 1 insertions(+), 1 deletions(-)

$ git checkout master

Switched to branch 'master'

$ git merge --no-ff release-1.2

Merge made by recursive.

(Отчёт об изменениях)

$ git tag -a 1.2

Ветви релизов (release branches) используются для подготовки к выпуску новых версий продукта.

Они позволяют расставить финальные точки над i перед выпуском новой версии.

Кроме того, в них можно добавлять минорные исправления, а также подготавливать метаданные для очередного релиза (номер версии, дата сборки и т.д.).

Очередной релиз получает свой номер версии только в тот момент, когда для него создаётся новая ветвь, но ни в коем случае не раньше.

#### Для чего нужны ветки hotfix?

$ git checkout -b hotfix-1.2.1 master

Switched to a new branch "hotfix-1.2.1"

$ ./bump-version.sh 1.2.1

Files modified successfully, version bumped to 1.2.1.

$ git commit -a -m "Bumped version number to 1.2.1"

[hotfix-1.2.1 41e61bb] Bumped version number to 1.2.1

1 files changed, 1 insertions(+), 1 deletions(-)

$ git commit -m "Fixed severe production problem"

[hotfix-1.2.1 abbe5d6] Fixed severe production problem

5 files changed, 32 insertions(+), 17 deletions(-)

Ветви исправлений (hotfix branches) создаются из главной (master) ветви.

Они нужны для срочного внесения критичных исправлений в текущий релиз, если новый релиз еще не готов.

### Литература

1. [Git: наглядная справка](https://marklodato.github.io/visual-git-guide/index-ru.html#checkout)
2. [Gitflow](https://habr.com/ru/post/106912/)
3. [Pro Git](https://git-scm.com/book/ru/v2)