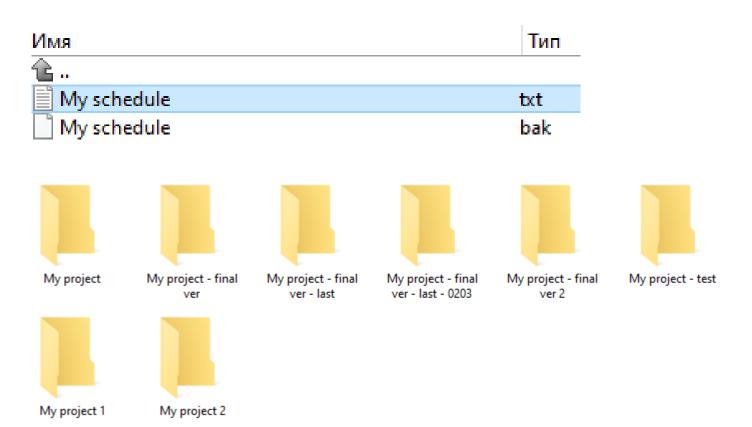


Эволюция человечества: Per aspera ad Git*.

^{* «}Через тернии – к Гиту» (лат) © Луций Анней Сенека

О чем эта лекция?

Что такое система управления версиями (Version Control System, VCS) и зачем она вам нужна?





О чем эта лекция?

Система управления версиями позволяет

- ✓ хранить несколько версий одного и того же документа
- ✓ при необходимости, отменять изменения и возвращаться к более ранним версиям – необязательно к предыдущей
- ✓ определять, кто и когда сделал то или иное изменение
- ✓ упрощает командную разработку и взаимодействие между командами



Классификация VCS

Локальные

Централизованные

Распределенные



Локальные VCS

Локальные VCS используют простую базу данных, которая хранит записи о всех изменениях в файлах. Обычно это набор патчей (различий между файлами) в специальном формате.



Пример:

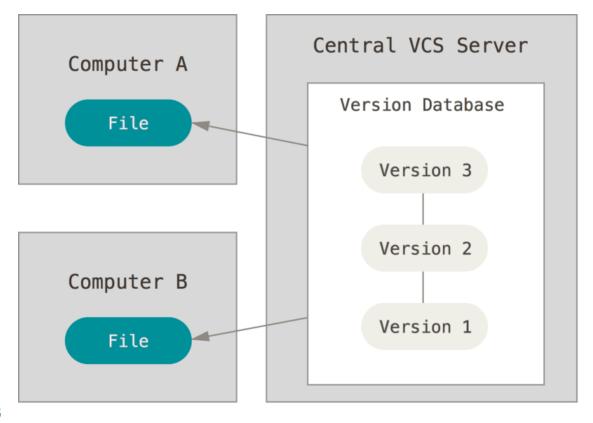
RCS

База данных, где сохраняются версии, называется репозиторием



Централизованные VCS

Централизованные VCS – это единый сервер, содержащий все версии файлов, и некоторое количество клиентов, которые получают файлы из этого централизованного хранилища и отправляют туда изменения.



Примеры:

CVS Subversion Perforce



Централизованные VCS

Самый очевидный минус таких систем — это единая точка отказа: централизованный сервер.

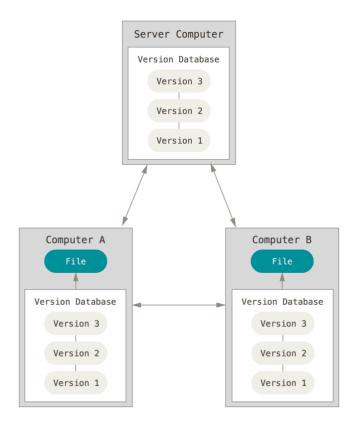
Если этот сервер выйдет из строя, скажем, на час, то в течение этого времени

- □ никто не сможет использовать VCS для сохранения изменений, над которыми работает
- никто не сможет обмениваться этими изменениями с другими разработчиками.

Если жёсткий диск, на котором хранится центральная БД, повреждён, а своевременные бэкапы отсутствуют, вы потеряете всю историю проекта.

Распределенные VCS

В распределенных VCS клиенты не просто скачивают состояние файлов на определённый момент времени — они **полностью** копируют репозиторий.



Примеры:

Git Mercurial Bazaar Darcs



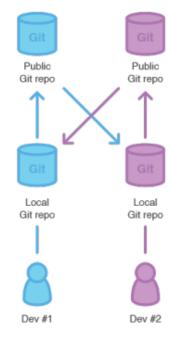
Распределенные VCS

Если один из серверов, через который разработчики обменивались данными, «умрёт», любой клиентский репозиторий может быть скопирован на другой сервер для продолжения работы: каждая копия репозитория является полным бэкапом всех данных.

Основная повседневная работа с VCS, включая поиск по истории, происходит **локально**, без обращения к серверу. Отсюда – высокая скорость!



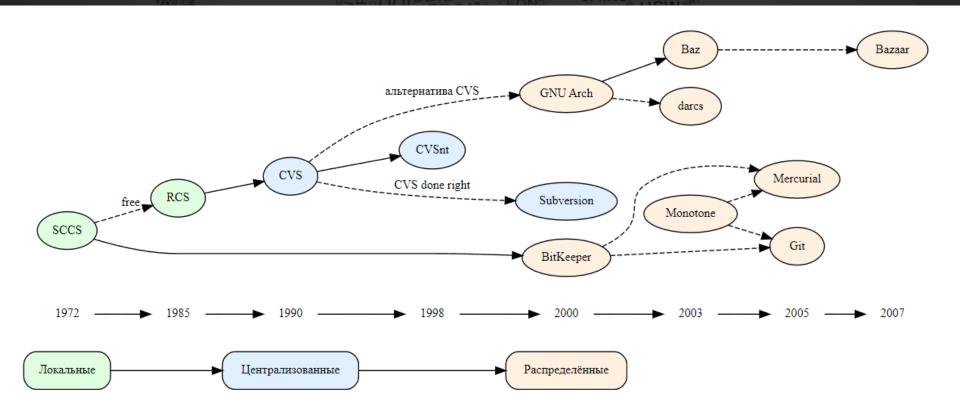
Centralized SVN development



Distributed Git development



Эволюция систем управления версиями



 $\mathsf{Источник}$: Виталий Филиппов, CUSTIS

 $\underline{\text{https://yourcmc.ru/wiki/\$D0\$9F\$D1\$80\$D0\$B5\$D0\$B7\$D0\$B5\$D0\$BD\$D1\$82\$D0\$B0\$D1\$86\$D0\$B8\$D1\$8F\ \$D0\$BF\$D0\$BE\ VCS}$



Причины популярности Git

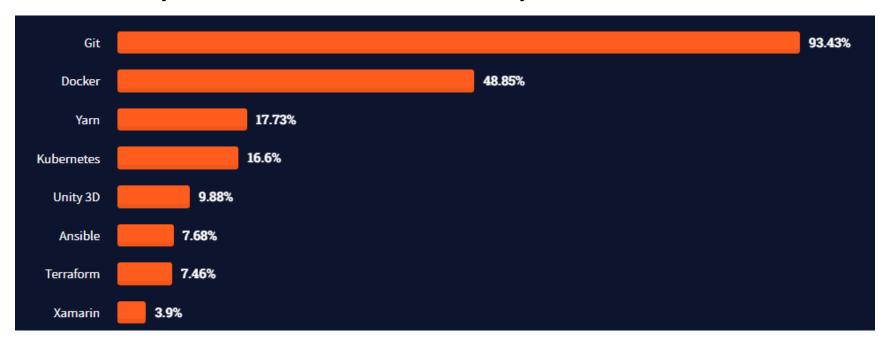
- Производительность"git status" для 14К файлов 0,2 секунды
- Оптимальное хранение истории
 20 лет, 70К коммитов: Git 770 MB, Mercurial 2800 MB
- Простая и изящная архитектура (объектная модель)
 Альтернативные реализации: libgit2, JGit, JS-Git
- Популярность Git тоже причина популярности Git!
 - Интеграция с основными IDE
 - Больший выбор WebUI
 - Хостинг: GitHub, GitLab и др.
 - Много информации в Интернете, вопросов на Stack Overflow



Git и другие

stackoverflow.com, 2021, survey:

«Which tools have you done extensive development work in over the past year, and which do you want to work in over the next year?»

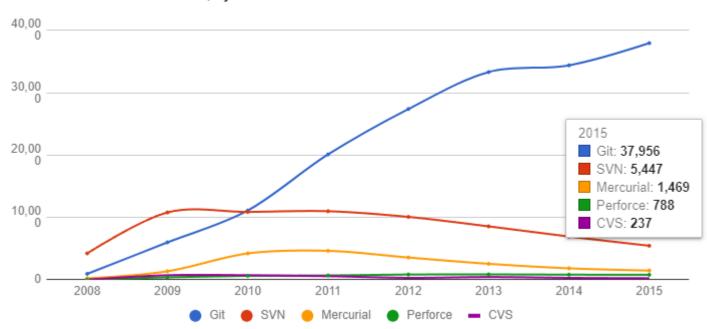






stackoverflow.com:

Questions on Stack Overflow, by Year



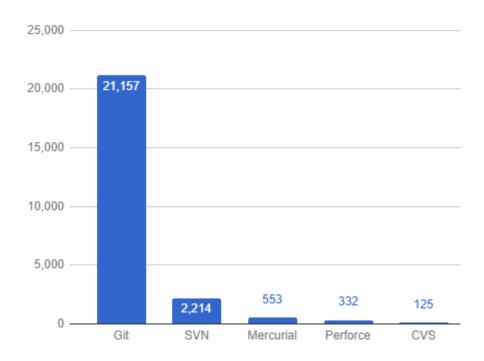
https://rhodecode.com/insights/version-control-systems-2016





stackoverflow.com:

Questions about VCS, by Number, 2016

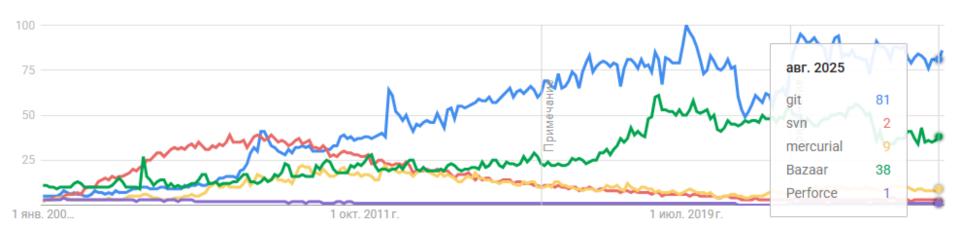


https://rhodecode.com/insights/version-control-systems-2016



Git и другие

Google Trends:



https://trends.google.com/trends/explore?date=all&q=git,svn,mercurial,Bazaar Perforce

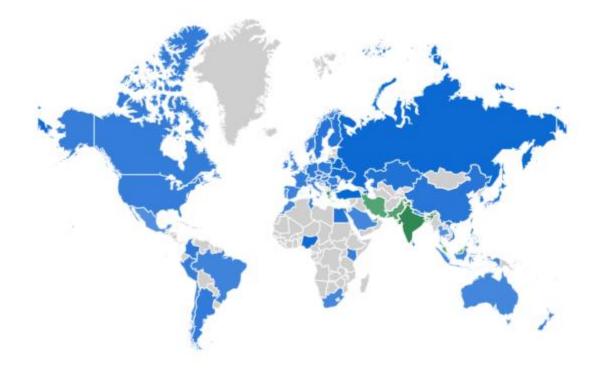
Примечание:

Bazaar – изначально разрабатывался компанией Canonical (создатели Ubuntu). Его философия — простота использования и удобство для пользователя. Bazaar позиционировался как более дружелюбная и интуитивно понятная альтернатива Git и Mercurial, особенно для тех, кто переходил с Subversion (SVN). Paspaбoтка прекращена в 2023 году, Ubuntu переехал на Git.



Git и другие

o git o svn o mercurial o Bazaar o Perforce





Итак, дальше будем говорить про Git!





Git (читается «гит») был создан Линусом Торвальдсом, автором Linux, в 2005 году для использования при разработке ядра Linux.



До этого разработка ядра Linux велась с использованием проприетарной VCS, именуемой «BitKeeper».

Ee автор — Ларри Маквой, тоже разработчик Linux — предоставил BitKeeper для работы над ядром Linux по бесплатной лицензии.



Разработчики Linux написали несколько утилит, и для одной из них произвели реверс-инжиниринг формата передачи данных BitKeeper.



В ответ Маквой обвинил разработчиков в нарушении соглашения и отозвал лицензию.

Торвальдс сам взялся за написание новой VCS, поскольку ни одна из открытых систем не позволяла тысячам программистов кооперировать свои усилия.



Исходные требования к Git:



- ✓ Скорость
- ✓ Простота дизайна
- ✓ Поддержка нелинейной разработки (тысячи параллельных веток)
- ✓ Полная распределённость
- ✓ Возможность эффективной работы с большими проектами (такими, как ядро Linux), как по скорости, так и по размеру данных



Первая версия **Git** была выпущена 7 апреля 2005 года.



Начальная разработка велась *меньше, чем неделю*:

- 3 апреля разработка началась
- 7 апреля сам код Git стал управляться новой, но еще неготовой VCS Git
- 16 июня Linux был переведён на Git
- 25 июля Торвальдс отказался от обязанностей ведущего разработчика.





Git – это НЕ аббревиатура.

Торвальдс так саркастически отозвался о выбранном им названии git, что на английском сленге означает «мерзавец»:

«I'm an egotistical bastard, so I name all my projects after myself. First Linux, now git.»

(«Я самовлюбленный ублюдок, и поэтому называю все свои проекты в честь себя. Сначала Linux, теперь git.»)

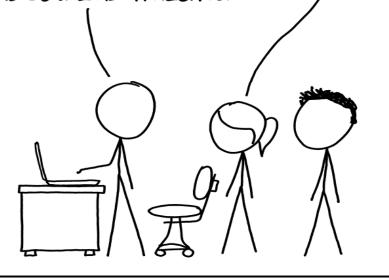


Как устроен Git

THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOU DO WE USE IT?

NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP: IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOUNLOAD A FRESH COPY.



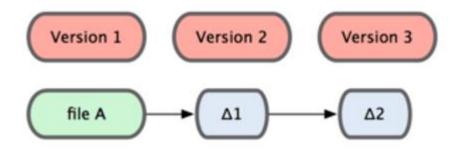




Как устроен Git



Большинство других VCS (CVS, Subversion, Perforce, Bazaar и другие) относятся к хранимым данным как к набору индивидуальных файлов и изменений, сделанных для каждого из этих файлов.



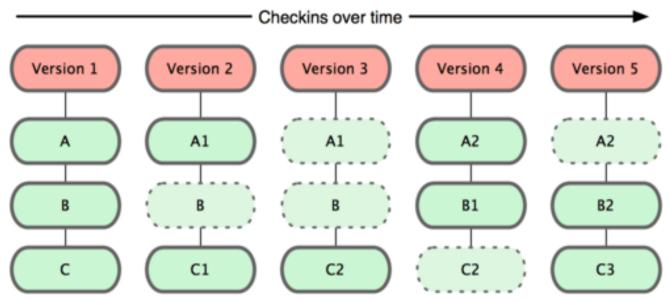
У каждого файла – своя версионность



Как устроен Git



Вместо этого Git считает хранимые данные набором слепков вашей файловой системы. Каждый раз, когда вы фиксируете текущую версию проекта, Git, по сути, сохраняет слепок (snapshot) того, как выглядят все файлы проекта на текущий момент.



(Про пунктир: ради эффективности, если файл не менялся, Git не сохраняет файл снова, а делает ссылку на ранее сохранённый файл.)



Как выглядит история в Git



Пример: история проекта в GitExtensions (одна из GUI оболочек Git):

1	origin/master rebase: allow custom merge_options	Edward Thomson	2/11/2016 9:49:25 PM
+	rebase: introduce inmemory rebasing	Edward Thomson	2/11/2016 9:48:48 PM
	master coverity: use https URL for posting build	Patrick Steinhardt	2/10/2016 1:06:23 PM
	coverity: provide nodef for GITERR_CHECK_ALLOC	Patrick Steinhardt	2/10/2016 12:59:14 PM
	Merge pull request #3599 from libgit2/gpgsign	Vicent Marti	2/9/2016 6:26:58 PM
	Merge pull request #3603 from pks-t/pks/coverity-fixes	Carlos Martín Nieto	2/9/2016 6:10:43 PM
	origin/gpgsign Introduce git_commit_extract_signature	Carlos Martín Nieto	2/9/2016 5:58:16 PM
	Merge pull request #3602 from libgit2/cmn/header-field-2	Carlos Martín Nieto	2/9/2016 4:53:10 PM
	attr_file: fix resource leak	Patrick Steinhardt	2/9/2016 1:11:38 PM
	checkout: fix resource leak	Patrick Steinhardt	2/9/2016 1:09:41 PM

Каждое сохраненный в истории «слепок» файлов проекта с описанием, кто, когда и зачем внес изменения, называется коммитом (commit)



Как устроен Git workflow



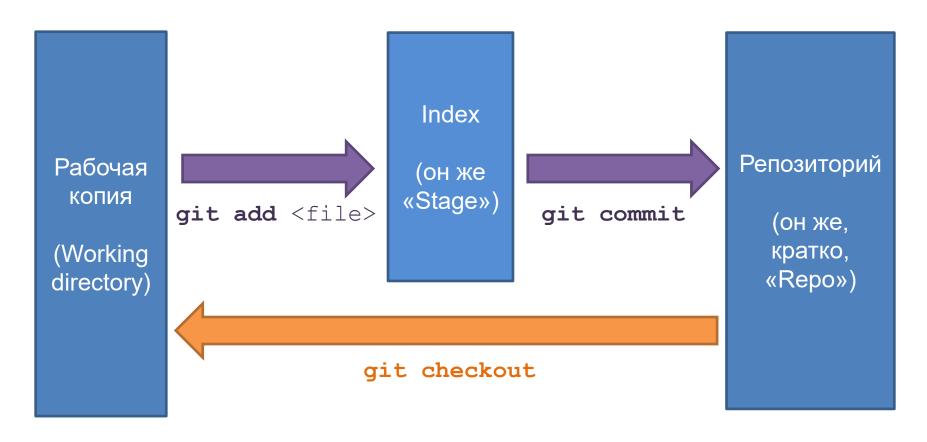
В Git файлы могут находиться в одном из трёх состояний:

- Изменённые файлы, которые поменялись, но ещё не были зафиксированы.
 - Подготовленные изменённые файлы, отмеченные для включения в следующий коммит.
 - Зафиксированные файлы, уже сохранённые в локальной базе.



Процесс фиксации в Git

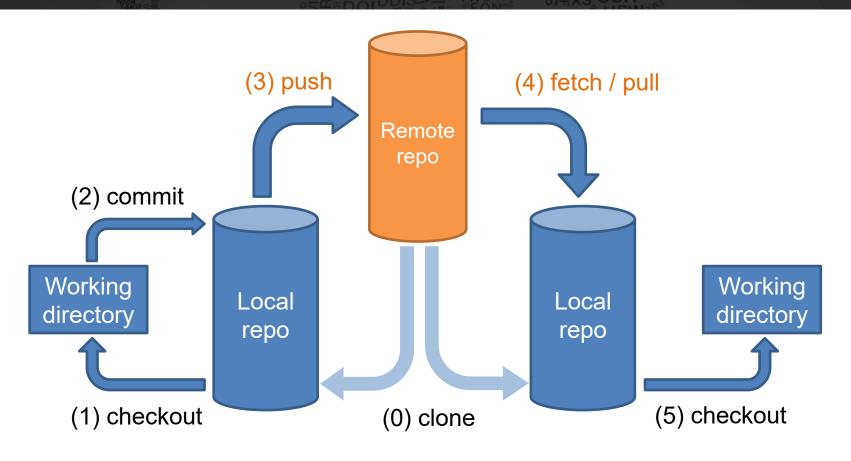






Git workflow







Git commands: Основные команды



- □ git init создание репозитория
- □ git clone клонирование удаленного репозитория себе
- □ git add добавляет файлы в stage (подготовка коммита)
- □ git commit выполняет коммит файлов из stage в репозиторий
- □ git checkout получение файлов, относящихся к указанной версии, из репозитория в рабочую папку
- □ git push отправка изменений, ранее зафиксированных коммитами в локальном репозитории, в удаленный репозиторий
- □ git fetch получение изменений из удаленного репозитория в локальный
- □ git status показывает, какие файлы изменились между рабочей папкой и репозиторием



Git commands: Porcelain & Plumbing



- ~ 110 команд (плюс опции):
- □ Porcelain высокоуровневые команды, которые мы в основном и используем:

add, commit, merge, push, pull, rebase, ...

□ Plumbing – низкоуровневые команды:

hash-object, commit-tree, write-tree, ...





Структура коммита в Git



Что именно мы сохраняем в репозитории, когда делаем очередной коммит?

git commit

Репозиторий

(он же, кратко, «Repo»)

Репозиторий лежит в папке .git

Git почти все хранит внутри себя как *объекты.*

Объекты могут быть разных типов, в частности:

- □ blob содержимое файла
- □ tree структура файлов
- □ commit описание коммита



Структура коммита в Git



Git вычисляет 40-символьный **sha1**-хэш от объекта, например, такой:

a37f3f668f09c61b7c12e857328f587c311e5d1d

и использует этот хэш как **имя файла**, в котором сохраняется сам объект:

.git/objects/a3/7f3f668f09c61b7c12e857328f587c311e5d1d

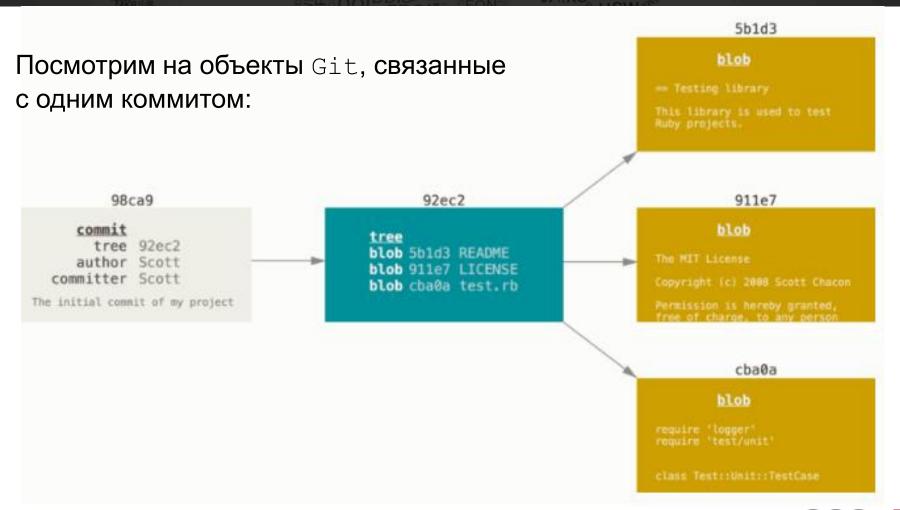
Длина sha1-хэша достаточна, чтобы не приводить к коллизиям.

Более того, практически безопасно в качестве идентификатора объекта можно использовать начальную часть хэша – по умолчанию, первые семь символов: a37f3f6.



Структура коммита в Git



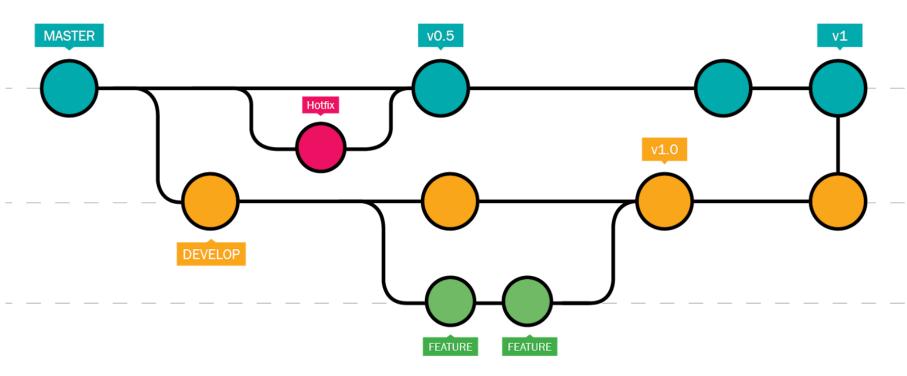




Работа с ветками Git



Давайте посмотрим на работу с ветками. Для чего нужны ветки?



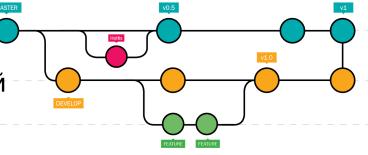


Работа с ветками Git



Давайте посмотрим на работу с ветками. Для чего нужны ветки?

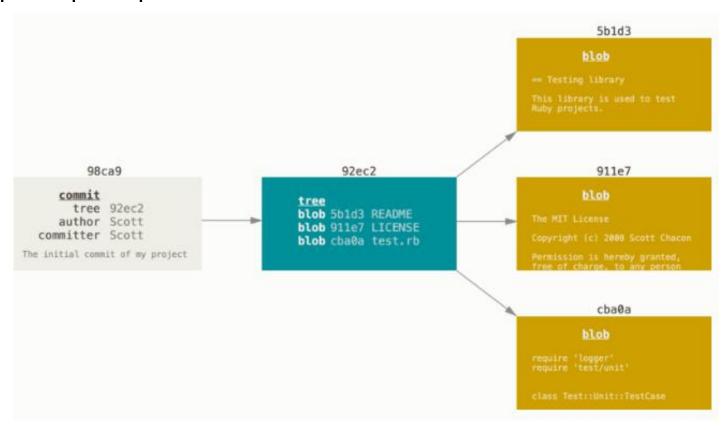
- Поддержание стабильности основной ветки
- Возможность быстро хотфиксить в любой момент времени
- Параллельная разработка
- □ Разные изменения в разных релизных ветках. Например, критические изменения во всех релизных ветках, не очень только в основной, для будущих релизов.







Пусть наш начальный коммит имеет именно такой вид, как мы разобрали ранее:

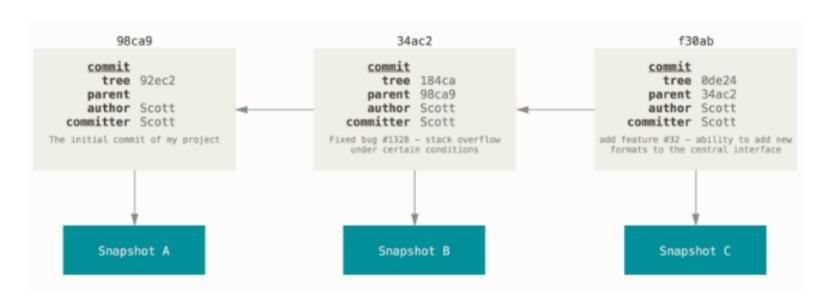






Сделаем некое изменение в рабочей папке, создадим новый коммит в репозитории (с помощью команды commit). А потом еще раз.

В результате мы получим цепочку из трех коммитов, каждый из которых (кроме самого первого) ссылается на предыдущий:





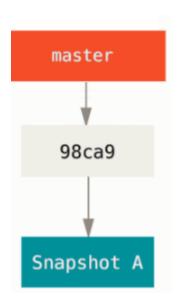


Ветка (branch) – это просто именованная ссылка на какой-то коммит.

Это просто текстовый файл размером 41 байт, лежащий в папке .git/refs/heads, имя которого — это имя ветки, а содержимое — хэш коммита.

Сразу после инициализации репозитория git сразу создает одну такую ссылку, именуемую master.

Она ничем не лучше и не хуже тех, которые будут создаваться позже, просто она создана сразу.







Веток (то есть именованных ссылок на коммиты) может быть много.

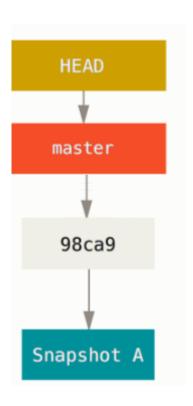
Одна из них всегда является текущей веткой. Это означает, что новые добавляемые коммиты будут добавляться «в нее».

Какая ветка текущая, Git определяет по содержимому специального файлика .git/HEAD.

По умолчанию, текущей веткой является master

То есть, файл неар содержит строку

ref: refs/heads/master

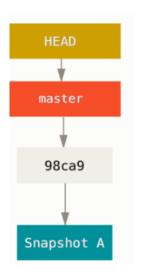




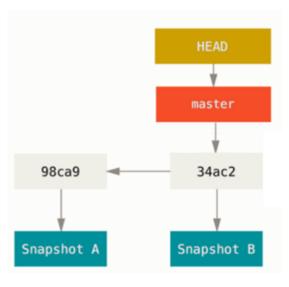


При добавлении новых коммитов в репозиторий с помощь команды commit текущая ссылка — в нашем случае master - обновляется сама и указывает на последний добавленный коммит.

Выполнив commit один раз, получаем





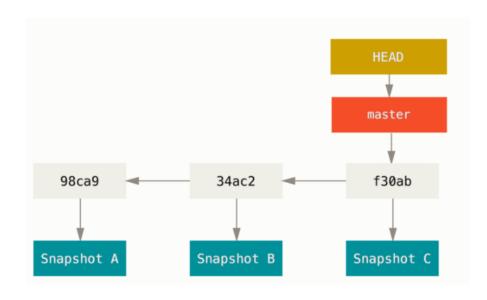






При добавлении новых коммитов в репозиторий с помощь команды commit текущая ссылка — в нашем случае master - обновляется cama и указывает на последний добавленный коммит.

Выполнив commit еще раз, получаем



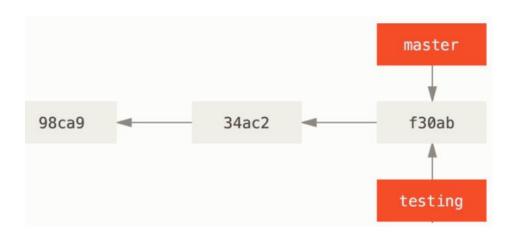




Давайте создадим новую ветку (то есть, новую именованную ссылку). Назовем ее testing. Создадим ссылку с этим именем, которая указывает на последний коммит.

Это сделает команда branch:

git branch testing

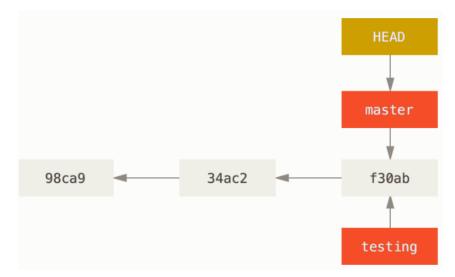






Мы создали новую ветку, но сами еще находимся в ветке master – на нее по-прежнему ссылается файл .git/HEAD.

И если мы продолжим добавлять коммиты командой commit, указатель master будет продолжать двигаться, а указатель testing — нет. Команда git branch создала ветку, но не переключила HEAD на нее.







Давайте переключимся на ветку testing, чтобы добавлять коммиты на нее.

Для этого существует команда git checkout <имя ветки>.

git checkout testing



Теперь файл неар содержит строку

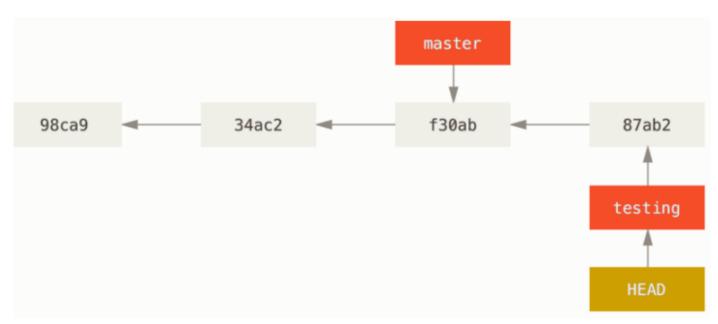
ref: refs/heads/testing





Теперь сделаем еще один commit. Модифицируем один файл в рабочей папке, и *одной командой* добавим измененный файл в индекс и создадим коммит:

git commit -a -m "Made a change"



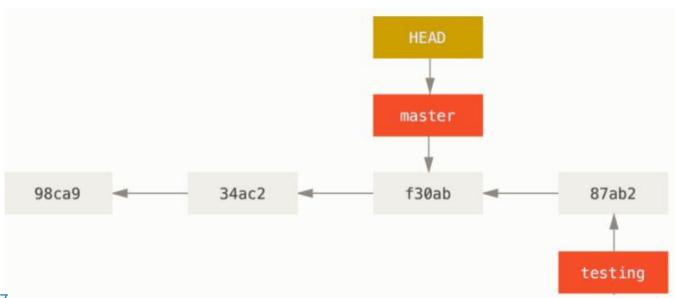




Переключимся снова на ветку master:

git checkout master

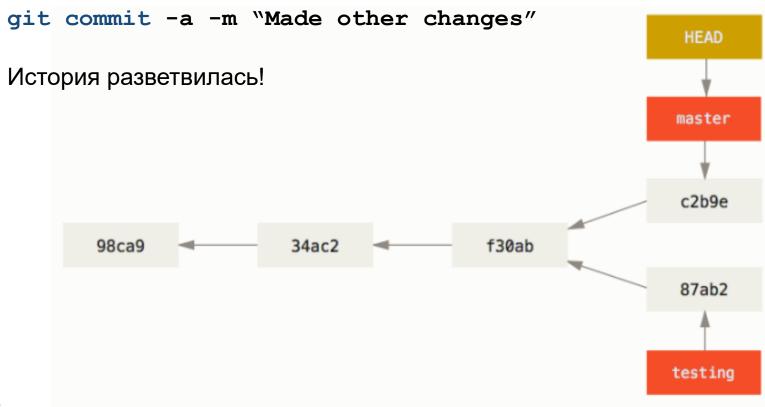
Это (1) поменяет файл HEAD и (2) изменит содержимое рабочей папки, так что ее содержимое станет отвечать «снимку» f30ab:







Сделаем еще один commit. Поменяем что-то еще в рабочей папке, и создадим коммит:







Так что же нам теперь делать с разветвившейся историей?

Обычно нам нужно получить результат, в котором есть:







Так что же нам теперь делать с разветвившейся историей?

Нам нужно научиться смерживать ветки!



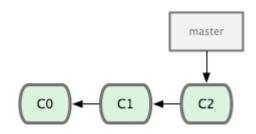






Рассмотрим обычный сценарий.

Пусть у вас есть некий проект, над которым вы работаете и уже сделали несколько коммитов: C0, C1 и C2.



Мы хотим поработать над фиксом некой проблемы #53.

Заведем для этого отдельную ветку iss53.





git checkout -b iss53

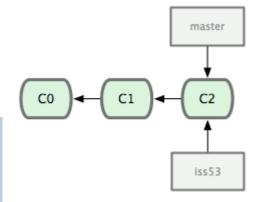
Switched to a new branch "iss53"

Примечание: Такая команда с ключом –b заменяет две последовательные – создания бранча без переключения

git branch iss53

и переключения на него

git checkout iss53

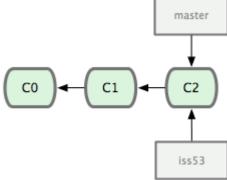






git checkout -b iss53

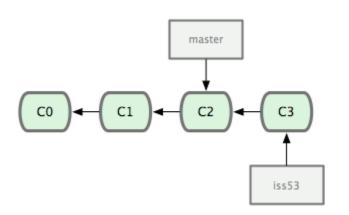
Switched to a new branch "iss53"



Мы поработали над фиксом и сделали коммит С3:

git commit -a -m 'Fixed issue 53'

Указатель ветки iss53 сдвинулся:







Приходит e-mail – выясняется, что срочно нужен фикс в основную ветку master.

Не проблема – мы просто переключаемся на мастер, и код в рабочей папке принимает вид, как будто работы над iss53 и не было:

C0 C1 C2 C3

git checkout master

Switched to branch "master"





Следуя хорошей практике в Git, создадим ветку для работы над этим срочным фиксом:

```
git checkout -b hotfix
Switched to a new branch "hotfix"
```

Мы поработали над срочным фиксом и сделали коммит С4:

```
git commit -a -m 'fixed the broken email address'
[hotfix]: created 3a0874c: "fixed the broken email address"
  1 files changed, 0 insertions(+), 1 deletions(-)
```



Работа с ветками Git. Fast forwarding.



hotfix

iss53

Указатель ветки hotfix тоже сдвинулся:

Вы можете запустить тесты, убедиться, что решение работает, и слить (merge) изменения назад в ветку master, чтобы включить их в продукт. Это делается с помощью команды git merge:

```
git checkout master
git merge hotfix
Updating f42c576..3a0874c
Fast forward
   README | 1 -
   1 files changed, 0 insertions(+), 1 deletions(-)
```



Работа с ветками Git. Fast forwarding.

git checkout master



```
git merge hotfix
Updating f42c576..3a0874c
Fast forward
 README |
 1 files changed, 0 insertions(+), 1 deletions(-)
                      hotfix
               master
                                                                    hotfix
                                                                    iss53
                      iss53
```



Теперь, когда срочная проблема решена, ветку **hotfix** можно удалить:

git branch -d hotfix

Deleted branch hotfix (3a0874c).





А мы вернемся на ветку **iss53** и закончим работу над issue #53:

```
git checkout iss53
Switched to branch "iss53"
```

В рабочей папке чудесным образом появились все наши изменения, сделанные ранее в ветке iss53, но, правда, пропал тот срочный фикс, который мы только что смержили в мастер.

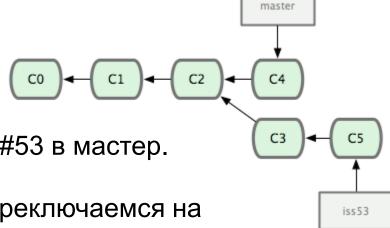
Делаем изменения и коммитим в iss53:

```
git commit -a -m 'Finished the new footer (issue 53)'
[iss53]: created ad82d7a: "Finished the new footer (issue 53)"
1 files changed, 1 insertions(+), 0 deletions(-)
```





Теперь ветка **iss53** продвинулась вперед:



Мы готовы залить решение проблемы #53 в мастер.

Действуем, как и с веткой **hotfix** — переключаемся на ветку **master** и просим Git смержить к нам ветку **iss53**:

```
git checkout master
git merge iss53
Merge made by recursive.
README | 1 +
1 files changed, 1 insertions(+), 0 deletions(-)
```

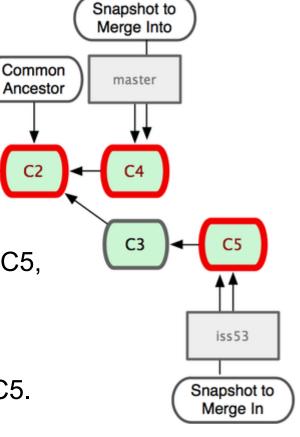




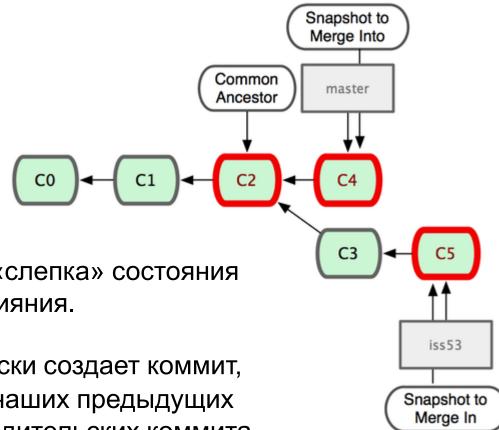
git checkout master git merge iss53 Merge made by recursive.

В этом случае коммит С4 не является предком С5, и Git действует иначе, используя стратегию *recursive*.

Он ищет ближайшего общего предка для C4 и C5. Это коммит C2.





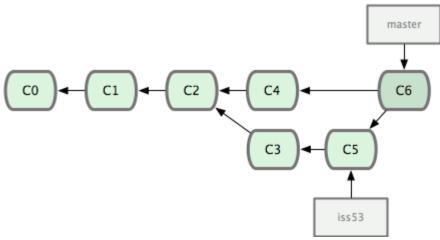


Эти три коммита, то есть три «слепка» состояния проекта, используются для слияния.

В результате **Git** автоматически создает коммит, у которого (в отличие от всех наших предыдущих примеров) будет сразу ДВА родительских коммита.



Вот результат:



Ветку iss53 теперь тоже можно удалить:

git branch -d iss53





Иногда процесс слияния не идёт гладко ⊗.

Если вы изменили одну и ту же часть файла по-разному в двух ветках, **Git** не сможет сделать это «чисто».

Если ваше решение проблемы #53 изменяет ту же часть файла, что и hotfix, вы получите конфликт слияния:

git merge iss53

Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.

Мерж будет приостановлен, пока все конфликты не будут разрешены. При этом Git переходит в специальное состояние – он помнит, что находится в процессе мержа двух веток.





Git добавляет стандартные маркеры к файлам, которые имеют конфликт, так что вы можете открыть их вручную и разрешить эти конфликты.

«Конфликтный» файл содержит секцию, которая выглядит примерно так:

```
<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======

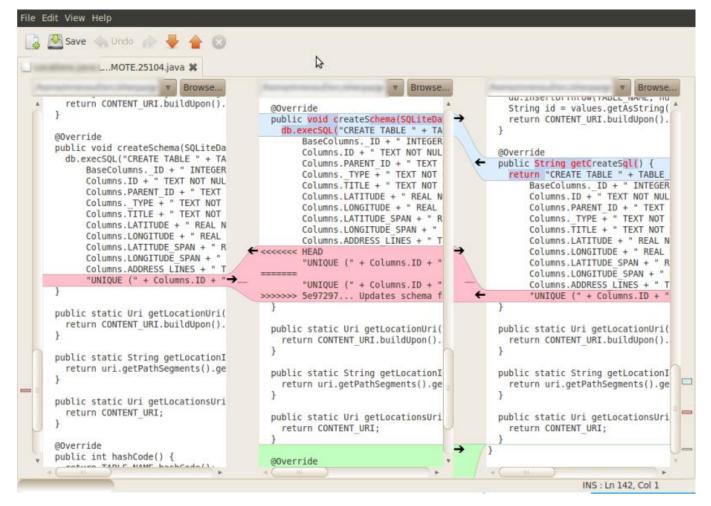
<div id="footer">
    please contact us at support@github.com
</div>
>>>>> iss53:index.html
```





Meld

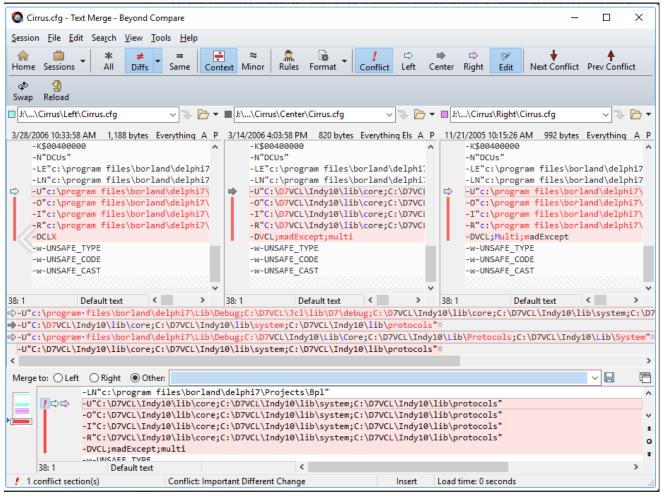
Средства визуализации конфликтов





Beyond Compare

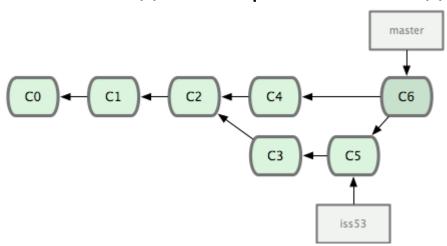
Средства визуализации конфликтов





После разрешения всех конфликтов, такие файлы нужно снова добавить к индексу командой git add, а после этого выполнить git commit.

Git «помнит», что он находит в процессе мержа двух веток и создаст «мерж-коммит» с двумя предками.





Первый коммит Git в Git!



git log e83c5163316f89bfbde7d9ab23ca2e25604af290

commit e83c5163316f89bfbde7d9ab23ca2e25604af290

Author: Linus Torvalds

Date: Thu Apr 7 15:13:13 2005 -0700

Initial revision of 'git', the information manager from hell



Хостинг для репозиториев Git





GitHub



GitHub – это онлайн-хостинг для репозиториев.

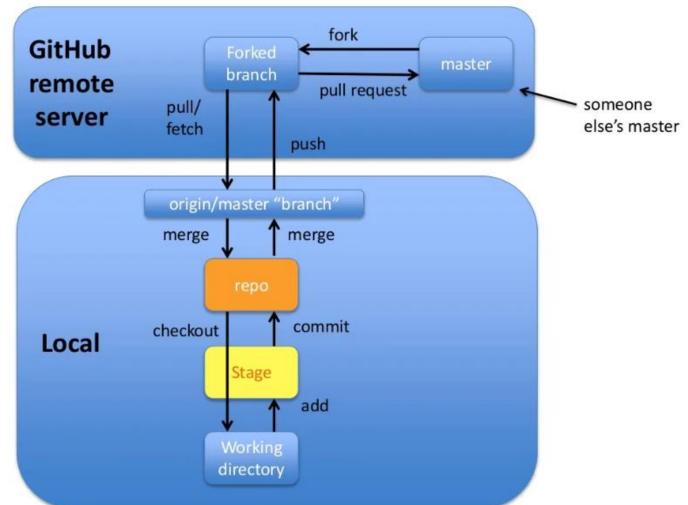
Он представляет собой облачное хранилище файлов на удаленном сервере.

Если Git является инструментом, то GitHub –это сервис, позволяющим использовать этот инструмент.



GitHub





GitHub



GitHub был запущен в 2008 году как платформа для разработки и хостинга репозиториев **git**. Он функционирует как социальная сеть с функциями, позволяющими следить за новостями, лентами и вики.

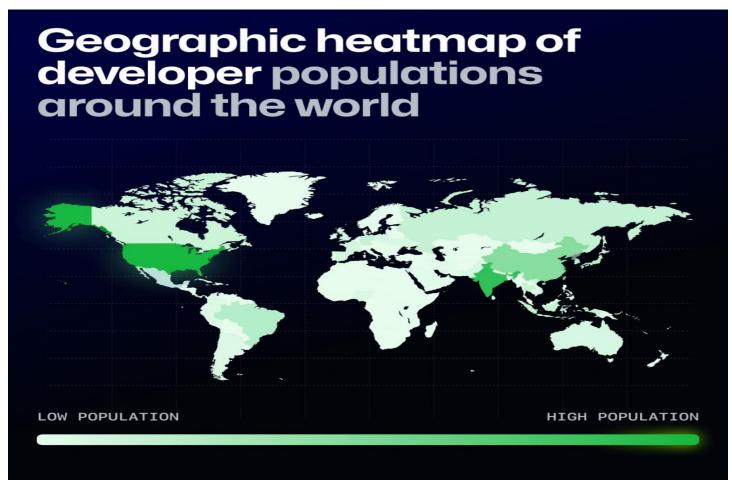
В 2018 году **GitHub** был приобретен Microsoft за удивительные 7,5 миллиардов долларов, и по состоянию на сентябрь 2025 года он мог похвастаться более чем 150 миллионами пользователей и более чем 420 миллионами репозиториев.

Источник: https://github.blog/news-insights/octoverse/octoverse-2024/



GitHub







GitHub









Бесплатные общедоступные репозитории

Вы можете создавать неограниченное количество репозиториев в GitHub для своих проектов, размещать любой код, коммитить и поддерживать все в актуальном состоянии.

Частные репозитории GitHub долгое время были только в платных планах, но теперь GitHub также предоставляет и бесплатные частные репозитории.





Pull Requests

GitHub предоставляет решение для создания пулреквестов для вашей кодовой базы.

Вы можете быть уверены, что ничто не нарушит вашу кодовую базу, и вы сможете легко работать с участниками.





Шаблоны запросов (Issue templates)

Когда вы создаете проект с открытым исходным кодом, важно получать отзывы от сообщества.

На GitHub можно создать специализированные шаблоны для репортов от пользователей и коллег. Например, «отчет об ошибке» или «запрос функционала».

Это очень полезно, так как вы сможете фильтровать проблемы и устанавливать их приоритет.





Доска проектов (Project board)

Они помогают создавать собственные рабочие процессы и расставлять приоритеты в работе над задачами или функциями, а также планировать релизы программного продукта.





Развертывание на своем сервере (Self-hosted solution)

Крупная организация может хранить все на своих собственных серверах, а не в облаке. Это обеспечивает большую гибкость и безопасность.





Страницы GitHub

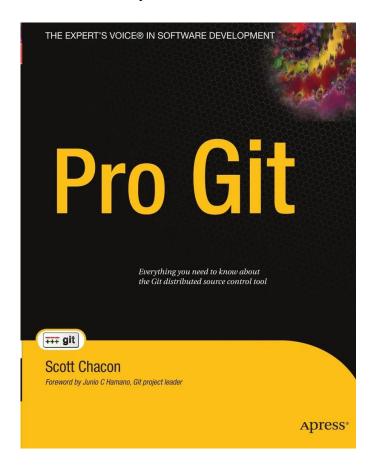
Это бесплатная функциональность GitHub, которая позволяет вам создавать свои страницы, просто создавая репозиторий со своим статическим вебсайтом.

Он будет автоматически развернут и размещен на username.github.io. Так можно разместить, например, свое резюме или веб-сайт портфолио.



Книжки сертина и дентина и дентина

Скотт Чакон | Pro Git

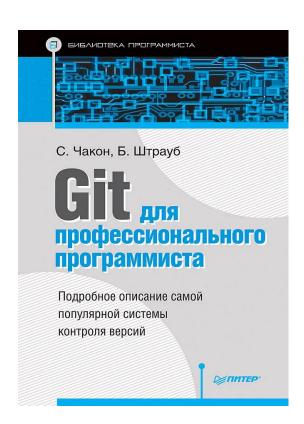


https://git-scm.com/book/en/v2 https://git-scm.com/book/ru/v2



TR MANUE ON THE STATE OF THE ST

Скотт Чакон, Бен Штрауб | Git для профессионального программиста





Ресурсы

Pro Git.

https://git-scm.com/book/ru/v2/



Scott Chacon и Ben Straub 2-е издание

Youtube:

Git. Большой практический выпуск.

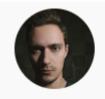
https://www.youtube.com/watch?v=SEvR78OhGtw

Git и GitHub Курс Для Новичков.

https://www.youtube.com/watch?v=zZBiIn 2FhM



Артем Матяшов 51,6 тыс. подписчиков



Workshop:

A Plumber's Guide to Git (English)

https://alexwlchan.net/a-plumbers-guide-to-git/



Alex Chan @alexwlchan



Ресурсы

Подборка online-ресурсов для самостоятельного изучения Git:

https://medium.com/javarevisited/11-best-online-places-to-learn-git-for-beginners-in-2021-6dc2b7c6ef48

Online-упражнения по использованию базовых команд Git:

https://www.w3schools.com/git/exercise.asp

Неплохая презентация по истории систем управления версиями:

https://yourcmc.ru/wiki/%D0%9F%D1%80%D0%B5%D0%B7%D0%B5%D0%BD%D1 %82%D0%B0%D1%86%D0%B8%D1%8F %D0%BF%D0%BE VCS

Git - the simple guide

http://rogerdudler.github.io/git-guide/index.ru.html

Learn Git in 15 minutes

https://try.github.io/levels/1/challenges/1



Задание

- 1. Выполнить задания всех уровней в online-тренинге https://learngitbranching.js.org/
- 2: Строго следуя указаниям в файле HowToDoHomework.md, сделать форк указанного там репозитория, склонировать его и добавить в корневую папку файл Questionnaire.md, в котором кратко рассказать о себе (по-русски или по-английски), своих увлечениях и о своих успехах и планах в области программирования. Напишите, каковы ваши ожидания от курса (в конце курса напишете, почему они не оправдались).

